UNIT-I

1. DATA STRUCTURES

1.1 DEFINITION:

A ***data structure*** is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

When selecting a data structure to solve a problem, the following steps must be performed.

1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centred view of the design process. In the approach, the first concern is the data and the operations that are to be performed on them. The second concern is the representation of the data, and the final concern is the implementation of that representation.

1.2 CLASSIFICATION OF DATA STRUCTURES:

Data structures are generally categorized into two classes: ***primitive*** and ***non-primitive*** data structures.

1. Primitive Data Structures:

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character and Boolean. The terms ‘data type’, ‘basic data type’ and ‘primitive data type’ are often used interchangeably.

2. Non-Primitive Data Structures:

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: ***linear*** and ***non-linear*** data structures.

Linear and Non-linear Data Structures:

a) Linear Data Structure:

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links. C supports a variety of data structures.

ARRAYS:

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an ***index*** (also known as the *subscript*).

In C, arrays are declared using the following syntax: type name [size];

For example,

int marks [10];

The above statement declares an array marks that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in marks [0], second element in marks [1], so on and so forth. Therefore, the last element, that is the 10th element, will be stored in marks [9]. In the memory, the array will be stored as shown in Fig. 1.1.

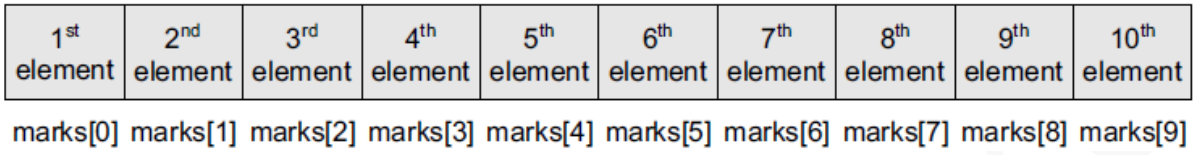


Figure 1.1 Memory representation of an array of 10 elements

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

LINKED LISTS:

A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list. In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance.

In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list. Since the memory for a node is dynamically allocated when it is added to the list, the

total number of nodes that may be added to a list is limited only by the amount of memory available. Figure 1.2 shows a linked list of seven nodes.

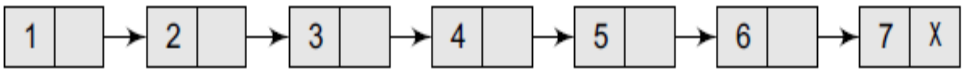


Figure 1.2 Simple linked list

Note:

Advantage: Easier to insert or delete data elements

Disadvantage: Slow search operation and requires more memory space

STACKS:

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

In the computer's memory, stacks can be implemented using arrays or linked lists. Figure 1.3 shows the array implementation of a stack. Every stack has a variable top associated with it. Top is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted. There is another variable MAX, which is used to store the maximum number of elements that the stack can store.

If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.

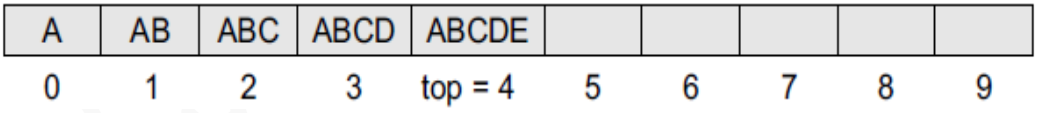


Figure 1.3 Array representation of a stack

In Fig. 1.3, top = 4, so insertions and deletions will be done at this position. Here, the stack can store a maximum of 10 elements where the indices range from 0–9. In the above stack, five more elements can still be stored.

A stack supports three basic operations: push, pop, and peep. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the peep operation returns the value of the topmost element of the stack (without deleting it).

However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full.

Similarly, before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

QUEUES:

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front. Like stacks, queues can be implemented by using either arrays or linked lists.

Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in Fig. 1.4.

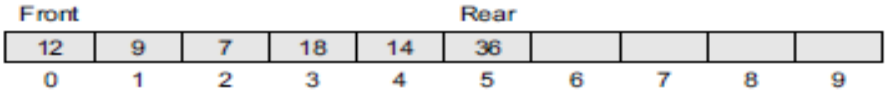


Figure 1.4 Array representation of a queue

Here, front = 0 and rear = 5. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the rear would be incremented by 1 and the value would be stored at the position pointed by the rear. The queue, after the addition, would be as shown in Fig. 1.5.

Here, front = 0 and rear = 6. Every time a new element is to be added, we will repeat the same procedure.

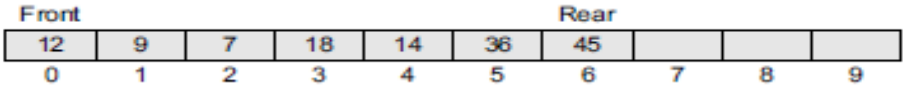


Figure 1.5 Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 1.6.

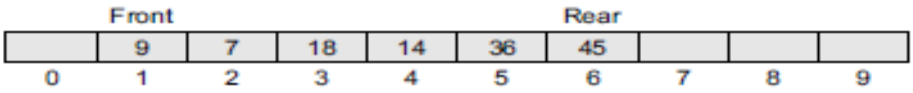


Figure 1.6 Queue after deletion of an element

However, before inserting an element in the queue, we must check for overflow conditions. An overflow occurs when we try to insert an element into a queue that is already full. A queue is full when rear = MAX – 1, where MAX is the size of the queue, that is MAX specifies the maximum number of elements in the queue. Note that we have written MAX – 1 because the index starts from 0.

Similarly, before deleting an element from the queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If front = NULL and rear = NULL, then there is no element in the queue.

b) Non-linear Data Structures:

If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

TREES:

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub tree. The root element is the topmost node which is pointed by a ‘root’ pointer. If root = NULL then the tree is empty.

Figure 1.7 shows a binary tree, where R is the root node and T1 and T2 are the left and right subtrees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty, then it is called the right successor of R.

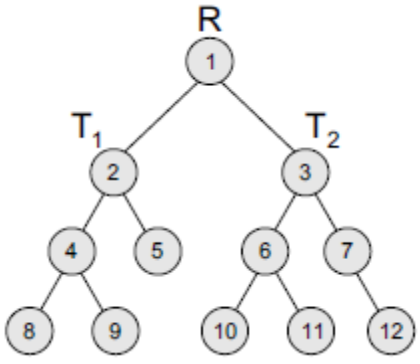


Figure 1.7 Binary tree

In Fig. 1.7, node 2 is the left child and node 3 is the right child of the root node 1. Note that the left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

Note:

Advantage: Provides quick search, insert, and delete operations.

Disadvantage: Complicated deletion algorithm.

GRAPHS:

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure 1.8 shows a graph with five nodes.

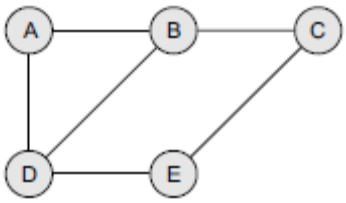


Figure 1.8 Graph

A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

Note that unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as *neighbours*. For example, in Fig. 1.8, node A has two neighbours: B and D.

Note:

Advantage: Best models real-world situations

Disadvantage: Some algorithms are slow and very complex.

1.3 OPERATIONS ON DATA STRUCTURES:

The different operations that can be performed on the various data structures previously mentioned.

Traversing:

It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching:

It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting:

It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course?

Deleting*:*

It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course?

Sorting:

Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging:

Lists of two sorted data items can be combined to form a single list of sorted data items.

Many a time, two or more operations are applied simultaneously in a given situation. For example, if we want to delete the details of a student whose name is X, then we first have to search the list of students to find whether the record of X exists or not and if it exists then at which location, so that the details can be deleted from that particular location.

1.4 ABSTRACT DATA TYPE:

An ***abstract data type*** (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples

of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.

To further understand the meaning of an abstract data type, we will break the term into 'data type' and 'abstract', and then discuss their meanings.

Data type:

Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include int, char, float, and double. When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data. For example, an int variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators +, –, *, and /. In other words, the operations that can be performed on a data type are an inseparable part of its identity. Therefore, when we declare a variable of an abstract data type (e.g., stack or a queue), we also need to specify the operations that can be performed on it.

Abstract:

The word 'abstract' in the context of data structures means considered apart from the detailed specifications or implementation.

In C, an abstract data type can be a structure considered without regard to its implementation.

It can be thought of as a 'description' of the data in the structure with a list of operations that can be performed on the data within that structure.

The end-user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

For example, when we use a stack or a queue, the user is concerned only with the type of data and the operations that can be performed on it. Therefore, the fundamentals of how the data is stored should be invisible to the user. They should not be concerned with how the methods work or what structures are being used to store the data. They should just know that

to work with stacks, they have push() and pop() functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

Advantage of using ADTs:

In the real world, programs evolve as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to separate the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

1.5 PRELIMINARIES OF ALGORITHMS:

The typical definition of algorithm is 'a formally defined procedure for performing some calculation'. An algorithm is basically a set of instructions that solve a problem. In general terms, an algorithm provides a blueprint to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Characteristics of an Algorithm:

1. **Input:** Externally we have to supply '0' or '1' input.
2. **Output:** After supplying of input we have to produce at least one output.
3. **Finiteness:** The algorithm must be terminate (or) end at some point.
4. **Definiteness:** We have to define clear and unambiguous statements.
5. **Effectiveness:** It should allow only necessary statements, need not to allow unnecessary statements.

Different approaches to designing an algorithm:

There are two main approaches to design an algorithm. They are: top-down approach and bottom-up approach, as shown in Fig. 1.9.

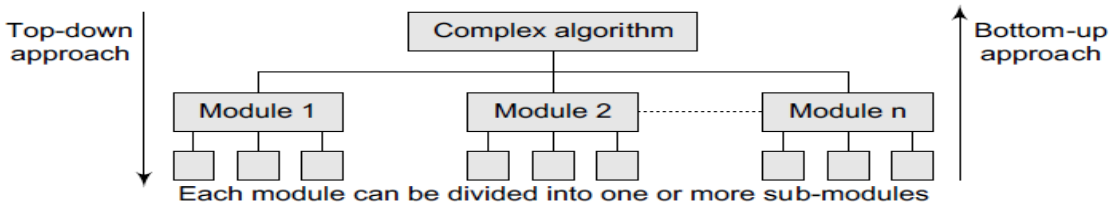


Figure 1.9 Different approaches of designing an algorithm

Top-down approach: A top-down design approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved. Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls. Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Bottom-up approach: A bottom-up approach is just the reverse of top-down approach. In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules. The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.

Control structures used in algorithms:

An algorithm may employ one of the following control structures:

(a) sequence: means that each step of an algorithm is executed in a specified order.

(b) decision: when the execution of a process depends on the outcome of some condition.

(c) repetition: which involves executing one or more steps for a number of times,

1.6 TIME AND SPACE COMPLEXITY:

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

The ***time complexity*** of an algorithm is basically the running time of a program as a function of the input size. Similarly, the ***space complexity*** of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Generally, the space needed by a program depends on the following two parts:

- **Fixed part:** It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- **Variable part:** It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

However, running time requirements are more critical than memory requirements. Therefore, in this section, we will concentrate on the running time efficiency of algorithms.

1.6.1 Worst-case, Average-case, Best-case, and Amortized Time Complexity:

Worst-case running time*:*

This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time:

The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time*:*

The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Amortized running time*:*

Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

1.6.2 Time–Space Trade-off:

The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution. But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other.

Hence, there exists a time–space trade-off among algorithms. So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time. On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

1.6.3 Expressing Time and Space Complexity:

The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved. Expressing the complexity is required when

- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

1.6.4 Algorithm Efficiency:

If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Let us consider different cases in which loops determine the efficiency of an algorithm.

Linear Loops:

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for (i=0; i<100;i++)
statement block;
```

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as **f(n) = n**

However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

```
for(i=0;i<100;i+=2)
statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as **f(n) = n/2**

Logarithmic Loops:

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)        for(i=1000;i>=1;i/=2)
statement block;            statement block;
```

Consider the first for loop in which the loop-controlling variable i is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles. Now, consider the second loop in which the loop-controlling variable i is divided by 2.

In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when n = 1000, the number of iterations can be given by log 1000 which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as **f(n) = log n**

Nested Loops:

Loops that contain loops are known as *nested loops*. In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

Linear logarithmic loop:

Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is log 10. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as 10 log 10.

```
for(i=0;i<10;i++)
for(j=1; j<10;j*=2)
statement block;
```

In more general terms, the efficiency of such loops can be given as **f(n) = n log n.**

Quadratic loop:

In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
for(j=0; j<10;j++)
statement block;
```

The generalized formula for quadratic loop can be given as **f(n) = n2.**

Dependent quadratic loop:

In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
for(j=0; j<=i;j++)
statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

1 + 2 + 3 + ... + 9 + 10 = 55

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates (n+ 1)/2 times. Therefore, the efficiency of such a code can be given as **f(n) = n (n + 1)/2.**

ASYMPTOTIC NOTATIONS:

BIG O NOTATION:

The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of n. For example, if a sorting algorithm performs n2 operations to sort just n elements, then that algorithm would be described as an O(n2) algorithm.

When expressing complexity using the Big O notation, constant multipliers are ignored. So, an O(4n) algorithm is equivalent to O(n), which is how it should be written. If f(n) and g(n) are the functions defined on a positive integer number n, then f(n) = O(g(n)).

That is, f of n is Big–O of g of n if and only if positive constants c and n exist, such that f(n) ≤ cg(n). It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). Hence, g provides an upper bound. Note that here c is a constant which depends on the following factors:

- the programming language used,
- the quality of the compiler or interpreter,
- the CPU speed,
- the size of the main memory and the access time to it,
- the knowledge of the programmer, and
- the algorithm itself, which may require simple but also time-consuming machine instructions.

We have seen that the Big O notation provides a strict upper bound for f(n). This means that the function f(n) can do better but not worse than the specified value. Big O notation is simply written as f(n) ∈ O(g(n)) or as f(n) = O(g(n)). Here, n is the problem size and O(g(n)) = {h(n): ∃ positive constants c, n0 such that $0 \leq h(n) \leq cg(n)$, ∀ $n \geq n0$}. Hence, we can say that O(g(n)) comprises a set of all the functions h(n) that are less than or equal to cg(n) for all values of $n \geq n0$.

If $f(n) \leq cg(n)$, $c > 0$, ∀ $n \geq n0$, then f(n) = O(g(n)) and g(n) is an asymptotically tight upper bound for f(n).

Examples of functions in O(n3) include: n2.9, n3, n3 + n, 540n3 + 10.

Examples of functions not in O(n3) include: n3.2, n2, n2 + n, 540n + 10, 2n

To summarize,

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as O(1)
- Linear time algorithm: running time complexity given as O(n)
- Logarithmic time algorithm: running time complexity given as O(log n)
- Polynomial time algorithm: running time complexity given as O(nk) where k > 1
- Exponential time algorithm: running time complexity given as O(2n)

Example 2.1 Show that 4n2 = O(n3).

Solution By definition, we have

$0 \leq h(n) \leq cg(n)$

Substituting 4n2 as h(n) and n3 as g(n), we get

$0 \leq 4n2 \leq cn3$

Dividing by n3

$0/n3 \leq 4n2/n3 \leq cn3/n3$

$0 \leq 4/n \leq c$

Now to determine the value of c, we see that 4/n is maximum when n=1. Therefore, c=4.

To determine the value of n0,

$0 \leq 4/n0 \leq 4$

$0 \leq 4/4 \leq n0$

$0 \leq 1 \leq n0$

This means n0=1. Therefore, $0 \leq 4n2 \leq 4n3$, $\forall\ n \geq n0=1$.

OMEGA NOTATION (Ω)

The Omega notation provides a tight lower bound for f(n). This means that the function can never do better than the specified value but it may do worse. Ω notation is simply written as, f(n) ∈ Ω(g(n)), where n is the problem size and Ω(g(n)) = {h(n): ∃ positive constants c > 0, n0 such that 0 ≤ cg(n) ≤ h(n), ∀ n ≥ n0}.

Hence, we can say that Ω(g(n)) comprises a set of all the functions h(n) that are greater than or equal to cg(n) for all values of n ≥ n0. If cg(n) ≤ f(n), c > O, ∀ n ≥ nO, then f(n) ∈ Ω(g(n)) and g(n) is an asymptotically tight lower bound for f(n).

Examples of functions in Ω(n2) include: n2, n2.9, n3 + n2, n3

Examples of functions not in Ω(n3) include: n, n2.9, n2

To summarize,

- Best case Ω describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.
- If we simply write Ω, it means same as best case Ω.

Example: Show that 5n2 + 10n = Ω(n2).

Solution By the definition, we can write

$0 \le cg(n) \le h(n)$

$0 \le cn2 \le 5n2 + 10n$

Dividing by n2

$0/n2 \le cn2/n2 \le 5n2/n2 + 10n/n2$

$0 \le c \le 5 + 10/n$

Now, $\lim_{n->infinite} 5 + 10/n = 5$. Therefore, $0 \le c \le 5$.

Hence, c = 5

Now to determine the value of n0

$0 \le 5 \le 5 + 10/n0$

$-5 \le 5 - 5 \le 5 + 10/n0 - 5$

$-5 \le 0 \le 10/n0$

So n0 = 1 as $\lim_{n->infinite} \cdot 1/n = 0$

Hence, 5n2 + 10n = Ω(n2) for c=5 and ∀ n ≥ n_O=1.

THETA NOTATION (Θ):

Theta notation provides an asymptotically tight bound for f(n). Θ notation is simply written as, f(n) ∈ Θ(g(n)), where n is the problem size and Θ(g(n)) = {h(n): ∃ positive constants c1, c2, and n0 such that $0 \leq c1g(n) \leq h(n) \leq c2g(n)$, ∀ $n \geq n0$}.

Hence, we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between c1g(n) and c2g(n) for all values of $n \geq n0$. If f(n) is between c1g(n) and c2g(n), ∀ $n \geq n0$, then f(n) ∈ Θ(g(n)) and g(n) is an asymptotically tight bound for f(n) and f(n) is amongst h(n) in the set.

To summarize,

- The best case in Θ notation is not used.
- Worst case Θ describes asymptotic bounds for worst case combination of input values.
- If we simply write Θ, it means same as worst case Θ.

Example 2.7 Show that $n2/2 - 2n = \Theta(n2)$.

Solution By the definition, we can write

$c1g(n) \leq h(n) \leq c2g(n)$

$c1n2 \leq n2/2 - 2n \leq c2n2$

Dividing by n2, we get

$c1n2/n2 \leq n2/2n2 - 2n/n2 \leq c2n2/n2$

$c1$

$\leq 1/2 - 2/n \leq c2$

This means $c2 = 1/2$ because $\lim_{n->infinite} 1/2 - 2/n = 1/2$ (Big O notation)

To determine c1 using Ω notation, we can write

$0 < c1 \leq 1/2 - 2/n$

We see that $0 < c1$ is minimum when $n = 5$. Therefore,

$0 < c1 \leq 1/2 - 2/5$

Hence, $c1 = 1/10$

Now let us determine the value of n0

$1/10 \leq 1/2 - 2/n0 \leq 1/2$

$2/n0 \leq 1/2 - 1/10 \leq 1/2$

$2/n0 \leq 2/5 \leq ½$

$n0 \geq 5$

You may verify this by substituting the values as shown below.

$c1n2 \leq n2/2 - 2n \leq c2n2$

$c1 = 1/10$, $c2 = 1/2$ and $n0 = 5$

$1/10(25) \leq 25/2 - 20/2 \leq 25/2$

$5/2 \leq 5/2 \leq 25/2$

Thus, in general, we can write, $1/10n2 \leq n2/2 - 2n \leq 1/2n2$ for $n \geq 5$.

OTHER USEFUL NOTATIONS:

There are other notations like little o notation and little ω notation which have been discussed below.

Little o Notation:

This notation provides a non-asymptotically tight upper bound for f(n). To express a function using this notation, we write $f(n) \in o(g(n))$ where $o(g(n)) = \{h(n) : \exists$ positive constants c, n0 such that for any $c > 0$, $n0 > 0$, and $0 \leq h(n) \leq cg(n), \forall\ n \geq n0\}$.

This is unlike the Big O notation where we say for some $c > 0$ (not any). For example, $5n3 = O(n3)$ is asymptotically tight upper bound but $5n2 = o(n3)$ is non-asymptotically tight bound for f(n).

Examples of functions in o(n3) include: n2.9, n3 / log n, 2n2

Examples of functions not in o(n3) include: 3n3, n3, n3 / 1000

Example: Show that $n3 / 1000 \neq o(n3)$.

Solution By definition, we have

$0 \leq h(n) < cg(n)$, for any constant $c > 0$

$0 \leq n3 / 1000 \leq cn3$

This is in contradiction with selecting any $c < 1/1000$.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

$f(n) = O(g(n)) \approx f(n) \leq g(n)$ $f(n) = o(g(n)) \approx f(n) < g(n)$ $f(n) = \Theta(g(n)) \approx f(n) = g(n)$

Little Omega Notation *(w)*:

This notation provides a non-asymptotically tight lower bound for f(n). It can be simply written as, $f(n) \in \omega(g(n))$, where $\omega(g(n)) = \{h(n) : \exists$ positive constants c, n0 such that for any $c > 0$, $n0 > 0$, and $0 \leq cg(n) < h(n), \forall\ n \geq n0\}$.

This is unlike the Ω notation where we say for some $c > 0$ (not any). For example, $5n3 = \Omega(n3)$ is asymptotically tight upper bound but $5n2 = \omega(n3)$ is non-asymptotically tight bound for f(n).

Example of functions in $\omega(g(n))$ include: $n3 = \omega(n2)$, $n3.001 = \omega(n3)$, $n2logn = \omega(n2)$

Example of a function not in ω(g(n)) is 5n2 ≠ ω(n2) (just as 5≠5)

Example: Show that 50n3/100 ≠ ω(n3).

Solution By definition, we have

$0 \leq cg(n) < h(n)$, for any constant $c > 0$

$0 \leq cn3 < 50n3/100$

Dividing by n3, we get

$0 \leq c < 50/100$

This is a contradictory value as for any value of c as it cannot be assured to be less than 50/100 or 1/2.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

$f(n) = \Omega(g(n)) \approx f(n) \geq g(n)$ $f(n) = \omega(g(n)) \approx f(n) > g(n)$

2. SEARCHING

INTRODUCTION TO SEARCHING:

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements: ***linear search*** and ***binary search*****.** The algorithm that should be used depends entirely on how the values are organized in the array.

2.1 LINEAR SEARCH:

Linear search, also called as ***sequential search*****,** is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[] is declared and initialized as,

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 2.1 Algorithm for linear search

Figure 2.1 shows the algorithm for linear search.

(a) In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.

(b) In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).

(c) In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm:

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In

both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

Programming Example:

Write a program to search an element in an array using the linear search technique.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
int arr[size], num, i, n, found = 0, pos = -1;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
        scanf("%d", &arr[i]);
}
printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
for(i=0;i<n;i++)
{
        if(arr[i] == num)
        {
                found =1;
                pos=i;
                printf("\n %d is found in the array at position= %d", num,i+1);
/* +1 added in line 23 so that it would display the number in the first place in
the array as in position 1 instead of 0 */
                break;
        }
}
if (found == 0)
{
```

```
        printf("\n %d does not exist in the array", num);
        return 0;
    }
```

2.2 BINARY SEARCH:

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array A[] that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.

In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially, BEG = lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.
(b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Figure 2.2 shows the algorithm for binary search.

(a) In Step 1, we initialize the value of variables, BEG, END, and POS.
(b) In Step 2, a while loop is executed until BEG is less than or equal to END.
(c) In Step 3, the value of MID is calculated.
(d) In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered.
(e) In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 2.2 Algorithm for binary search

Complexity of Binary Search Algorithm:

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given a**s 2f(n) > n or f(n) = log2n.**

Programming Example:

Write a program to search an element in an array using binary search.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10 // Added to make changing size of array easier
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[])
{
```

```
    int arr[size], num, i, n, beg, end, mid, found=0;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    selection_sort(arr, n); // Added to sort the array
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
    printf(" %d\t", arr[i]);
    printf("\n\n Enter the number that has to be searched: ");
    scanf("%d", &num);
    beg = 0, end = n-1;
    while(beg<=end)
    {
        mid = (beg + end)/2;
        if (arr[mid] == num)
        {
            printf("\n %d is present in the array at position %d",
        num, mid+1);
            found =1;
            break;
        }
        else if (arr[mid]>num)
            end = mid-1;
        else
            beg = mid+1;
    }
    if (beg > end && found == 0)
        printf("\n %d does not exist in the array", num);
    return 0;
}
```

```
int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}
void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];
                arr[k] = arr[pos];
                arr[pos] = temp;
        }
}
```

2.3 FIBONACCI SEARCH:

We are all well aware of the Fibonacci series in which the first two terms are 0 and 1 and then each successive term is the sum of previous two terms. In the Fibonacci series given below, each number is called a Fibonacci number.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The same series and concept can be used to search for a given value in a list of numbers. Sucha search algorithm which is based on Fibonacci numbers is called Fibonacci

search and was developed by Kiefer in 1953. The search follows a divide-and-conquer technique and narrows down possible locations with the help of Fibonacci numbers.

Fibonacci search is similar to binary search. It also works on a sorted list and has a run time complexity of **O(log n).** However, unlike the binary search algorithm, Fibonacci search does not divide the list into two equal halves rather it subtracts a Fibonacci number from the index to reduce the size of the list to be searched. So, the key advantage of Fibonacci search over binary search is that comparison dispersion is low.

3. SORTING

INTRODUCTION TO SORTING:

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N].For example, if we have an array that is declared and initialized as

int A[] = {21, 34, 11, 9, 1, 0, 22};

Then the sorted array (ascending order) can be given as:

A[] = {0, 1, 9, 11, 21, 22, 34;

3.1 INSERTION SORT:

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

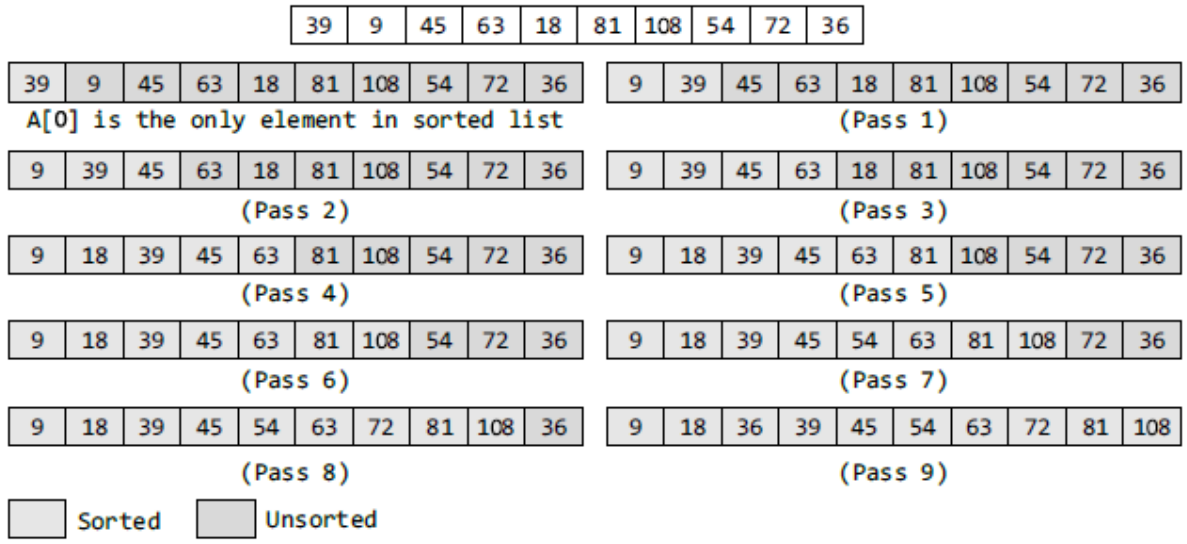
Technique:

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example: Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution:



Initially, A[0] is the only element in the sorted set.

- In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted.
- In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1].
- In Pass 3, A[3] will be placed in its proper place. In Pass N–1, A[N–1] will be placed in its proper place to keep the array sorted.

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location. The algorithm for insertion sort is given in Fig. 3.1.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:     SET TEMP = ARR[K]
Step 3:     SET J = K - 1
Step 4:     Repeat while TEMP <= ARR[J]
                  SET ARR[J + 1] = ARR[J]
                  SET J = J - 1
            [END OF INNER LOOP]
Step 5:     SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Figure 3.1 Algorithm for insertion sort

In the algorithm,

(a) In Step 1 executes a for loop which will be repeated for each element in the array.

(b) In Step 2, we store the value of the Kth element in TEMP.

(c) In Step 3, we set the Jth index in the array.

(d) In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements.

(e) Finally, In Step 5, the element is stored at the (J+1)th location.

Complexity of Insertion Sort:

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element.

Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n2)). Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

Advantages of Insertion Sort:

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Programming Example:

Write a program to sort an array using insertion sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        insertion_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
```

```
}
void insertion_sort(int arr[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
                temp = arr[i];
                j = i-1;
                while((temp < arr[j]) && (j>=0))
                {
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = temp;
        }
}
```

Output:

Enter the number of elements in the array : 5

Enter the elements of the array : 500 1 50 23 76

The sorted array is :

1 23 50 76 500

3.2 SELECTION SORT:

Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2), thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique:

Consider an array ARR with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted.

Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.

Example: Sort the array given below using selection sort.

	39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

The algorithm for selection sort is shown in Fig. 3.2 In the algorithm, during the Kth pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], ..., ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Figure 3.2 Algorithm for selection sort

Complexity of Selection Sort:

Selection sort is a sorting algorithm that is independent of the original order of elements in the array.

(a) In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position.

(b) In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on.

Therefore, $(n – 1) + (n – 2) + ... + 2 + 1 = n(n – 1) / 2 = O(n2)$ comparisons

Advantages of Selection Sort:

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Programming Example:

Write a program to sort an array using selection sort algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int smallest(int arr[], int k, int n);
```

```
void selection_sort(int arr[], int n);
void main(int argc, char *argv[])
{
	int arr[10], i, n;
	printf("\n Enter the number of elements in the array: ");
	scanf("%d", &n);
	printf("\n Enter the elements of the array: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &arr[i]);
	}
	selection_sort(arr, n);
	printf("\n The sorted array is: \n");
	for(i=0;i<n;i++)
	printf(" %d\t", arr[i]);
}
int smallest(int arr[], int k, int n)
{
	int pos = k, small=arr[k], i;
	for(i=k+1;i<n;i++)
	{
		if(arr[i]< small)
		{
			small = arr[i];
			pos = i;
		}
	}
	return pos;
}
void selection_sort(int arr[],int n)
{
	int k, pos, temp;
	for(k=0;k<n;k++)
	{
```

```
            pos = smallest(arr, k, n);
            temp = arr[k];
            arr[k] = arr[pos];
            arr[pos] = temp;
        }
    }
```

3.3 EXCHANGE (BUBBLE SORT, QUICK SORT):

3.3.1 BUBBLE SORT:

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note:

If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Technique:

The basic methodology of the working of bubble sort is given as follows:

a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.
b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2

involves n–2 comparisons and places the second biggest element at the second highest index of the array.

c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

Example: To discuss bubble sort in detail, let us consider an array A[] that has the following elements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.

(b) Compare 52 and 29. Since 52 > 29, swapping is done.

30, **29, 52**, 87, 63, 27, 19, 54

(c) Compare 52 and 87. Since 52 < 87, no swapping is done.

(d) Compare 87 and 63. Since 87 > 63, swapping is done.

30, 29, 52, **63, 87**, 27, 19, 54

(e) Compare 87 and 27. Since 87 > 27, swapping is done.

30, 29, 52, 63, **27, 87**, 19, 54

(f) Compare 87 and 19. Since 87 > 19, swapping is done.

30, 29, 52, 63, 27, **19**, **87**, 54

(g) Compare 87 and 54. Since 87 > 54, swapping is done.

30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.

29, 30, 52, 63, 27, 19, 54, 87

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 63. Since 52 < 63, no swapping is done.

(d) Compare 63 and 27. Since 63 > 27, swapping is done.

29, 30, 52, **27, 63**, 19, 54, 87

(e) Compare 63 and 19. Since 63 > 19, swapping is done.

29, 30, 52, 27, **19, 63**, 54, 87

(f) Compare 63 and 54. Since 63 > 54, swapping is done.

29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 27. Since 52 > 27, swapping is done.

29, 30, **27, 52**, 19, 54, 63, 87

(d) Compare 52 and 19. Since 52 > 19, swapping is done.

29, 30, 27**, 19, 52**, 54, 63, 87

(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 27. Since 30 > 27, swapping is done.

29, **27, 30**, 19, 52, 54, 63, 87

(c) Compare 30 and 19. Since 30 > 19, swapping is done.

29, 27, **19, 30**, 52, 54, 63, 87

(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping is done.

27, 29, 19, 30, 52, 54, 63, 87

(b) Compare 29 and 19. Since 29 > 19, swapping is done.

27, **19, 29**, 30, 52, 54, 63, 87

(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, 27, 29, 30, 52, 54, 63, 87

(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Observe that the entire list is sorted now.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Figure 3.3 Algorithm for bubble sort

Figure 3.3 shows the algorithm for bubble sort.

In this algorithm, the outer loop is for the total number of passes which is N–1. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position.

Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of the pass.

Complexity of Bubble Sort:

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1$

$f(n) = n (n – 1)/2$

$f(n) = n2/2 + O(n) = O(n2)$

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

Programming Example:

Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, temp, j, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr [i]);
        }
        for(i=0;i<n;i++)
```

```
        {
            for(j=0;j<n–i–1;j++)
            {
                if(arr[j] > arr[j+1])
                {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
            printf("%d\t", arr[i]);
        getch();
        return 0;
    }
```

Output:

Enter the number of elements in the array : 10

Enter the elements : 8 9 6 7 5 4 2 3 1 10

The array sorted in ascending order is :

1 2 3 4 5 6 7 8 9 10

Bubble Sort Optimization:

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n–1 passes. We may even have an array that will be sorted in 2 or 3 passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to TRUE before each pass and is made FALSE when a swapping is performed. The code for the optimized bubble sort can be given as:

```
void bubble_sort(int *arr, int n)
{
        int i, j, temp, flag = 0;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n–i–1; j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                flag = 1;
                                temp = arr[j+1];
                                arr[j+1] = arr[j];
                                arr[j] = temp;
                        }
                }
                if(flag == 0) // array is sorted
                return;
        }
}
```

Complexity of Optimized Bubble Sort Algorithm:

In the best case, when the array is already sorted, the optimized bubble sort will take O(n) time.

In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes O(n2) in all the cases.

3.3.2 QUICK SORT:

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O(n log n) algorithms, because its efficient implementation can minimize

the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays. The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique:

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].
    (a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

(c) Set loc = right

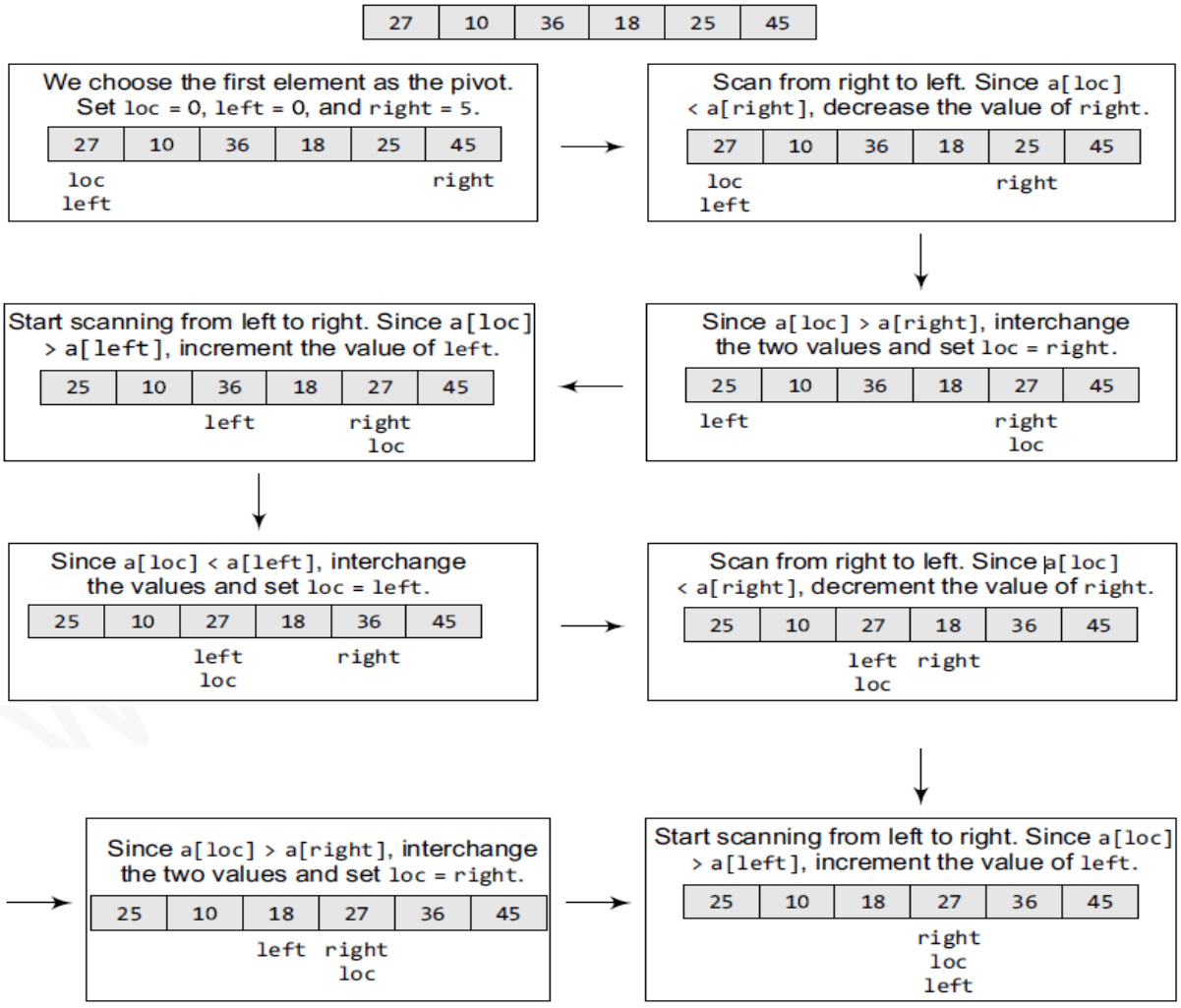
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

(c) Set loc = left.

Example: Sort the elements given in the following array using quick sort algorithm



Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

The quick sort algorithm (Fig. 3.4) makes use of a function Partition to divide the array into two sub-arrays.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Figure 3.4 Algorithm for quick sort

Complexity of Quick Sort:

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of the array which simply loops over the elements of the array once uses O(n) time.

In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(n log n) time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as O(n2). The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.

However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

Pros and Cons of Quick Sort:

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Programming Example:

Write a program to implement quick sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
```

```
        {
                scanf("%d", &arr[i]);
        }
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                right--;
                if(loc==right)
                flag =1;
                else if(a[loc]>a[right])
                {
                        temp = a[loc];
                        a[loc] = a[right];
                        a[right] = temp;
                        loc = right;
                }
                if(flag!=1)
                {
                        while((a[loc] >= a[left]) && (loc!=left))
                        left++;
                        if(loc==left)
                                flag =1;
```

```
                else if(a[loc] <a[left])
                {
                        temp = a[loc];
                        a[loc] = a[left];
                        a[left] = temp;
                        loc = left;
                }
            }
        }
        return loc;
}
void quick_sort(int a[], int beg, int end)
{
        int loc;
        if(beg<end)
        {
            loc = partition(a, beg, end);
            quick_sort(a, beg, loc-1);
            quick_sort(a, loc+1, end);
        }
}
```

3.4 DISTRIBUTION (RADIX SORT):

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digits.

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:           SET I = 0 and INITIALIZE buckets
Step 6:           Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT  = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                  [END OF LOOP]
Step 10:          Collect the numbers in the bucket
       [END OF LOOP]
Step 11: END
```

Figure 3.5 Algorithm for radix sort

Example: Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result.

After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort:

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O (kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Pros and Cons of Radix Sort:

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.

Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Programming Example:

Write a program to implement radix sort algorithm.

```
##include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
```

```
        {
                scanf("%d", &arr[i]);
        }
        radix_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int largest(int arr[], int n)
{
        int large=arr[0], i;
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                large = arr[i];
        }
        return large;
}
void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                        bucket_count[i]=0;
                for(i=0;i<n;i++)
```

```
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}
```

3.5 MERGING (MERGE SORT) ALGORITHMS:

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

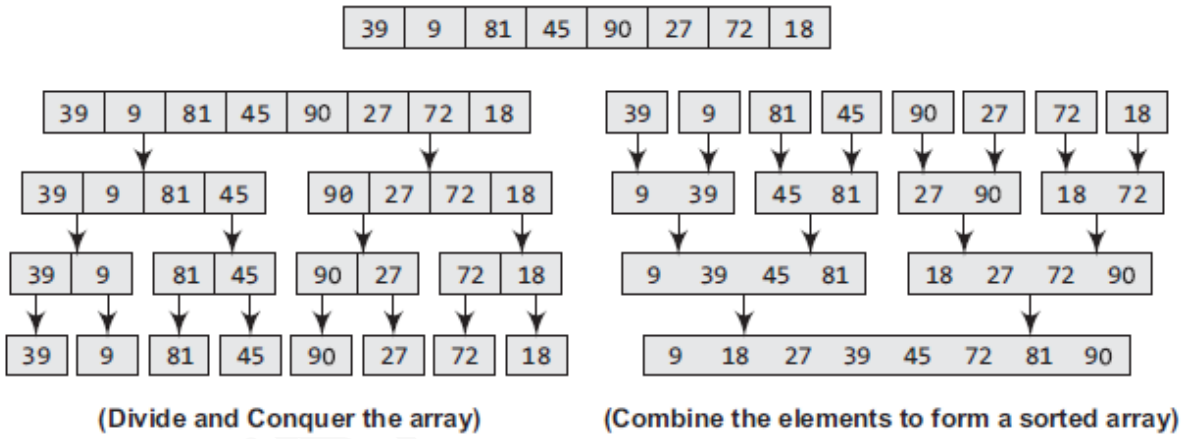
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

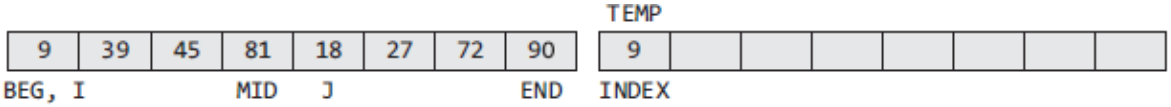
Example: Sort the array given below using merge sort.

Solution

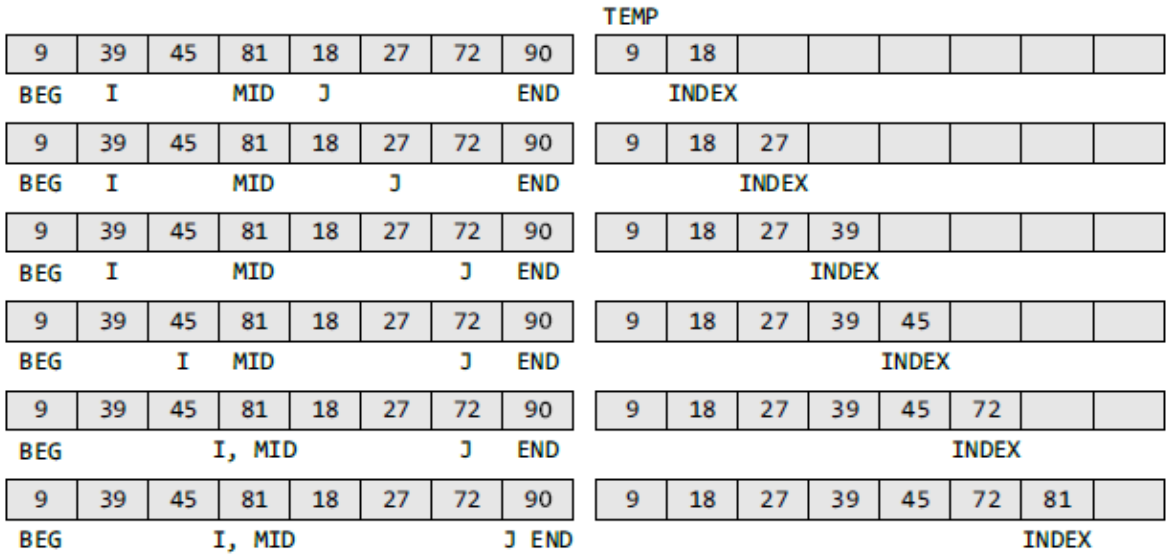


The merge sort algorithm (Fig. 3.6) uses a function merge which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.



Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.



When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.


```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = INDEX + 1, SET I = I + 1
               [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

Figure 3.6 Algorithm for merge sort

The running time of merge sort in the average case and the worst case can be given as O(n log n). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Programming Example:

Write a program to implement merge sort.

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        merge_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
```

```
	getch();
}
void merge(int arr[], int beg, int mid, int end)
{
	int i=beg, j=mid+1, index=beg, temp[size], k;
	while((i<=mid) && (j<=end))
	{
		if(arr[i] < arr[j])
		{
			temp[index] = arr[i];
			i++;
		}
		else
		{
			temp[index] = arr[j];
			j++;
		}
		index++;
	}
	if(i>mid)
	{
		while(j<=end)
		{
			temp[index] = arr[j];
			j++;
			index++;
		}
	}
	else
	{
		while(i<=mid)
		{
			temp[index] = arr[i];
			i++;
```

```
                index++;
            }
        }
        for(k=beg;k<index;k++)
        arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
            mid = (beg+end)/2;
            merge_sort(arr, beg, mid);
            merge_sort(arr, mid+1, end);
            merge(arr, beg, mid, end);
        }
}
```

UNIT-II

1. LINKED LIST

1.1 Introduction:

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

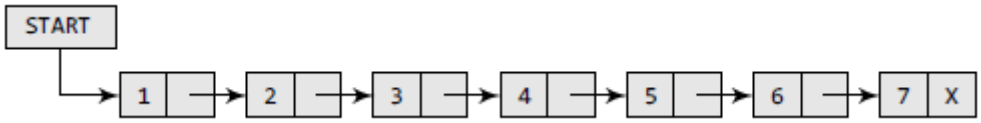


Figure 1.1 Simple linked list

In Fig. 1.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The left part of the node which contains data may include a simple data type, an array, or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL.

In Fig. 1.1, the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code:

```
struct node
{
    int data;
    struct node *next;
};
```

Note:

Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.

1.2 SINGLE LINKED LISTS:

A single linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A single linked list allows traversal of data only in one way. Figure 1.2 shows a single linked list.

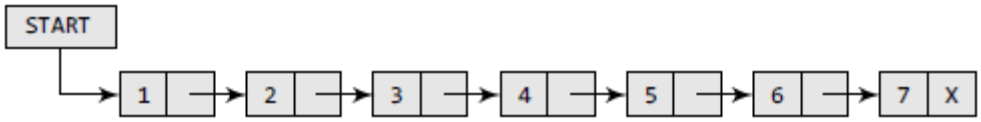


Figure 1.2 Single linked list

1.3 REPRESENTATION OF LINKED LIST IN MEMORY:

Let us see how a linked list is maintained (represented) in the memory. In order to form a linked list, we need a structure called *node* which has two fields, DATA and NEXT. DATA will store the information part and NEXT will store the address of the next node in sequence.

Consider Fig. 1.3.

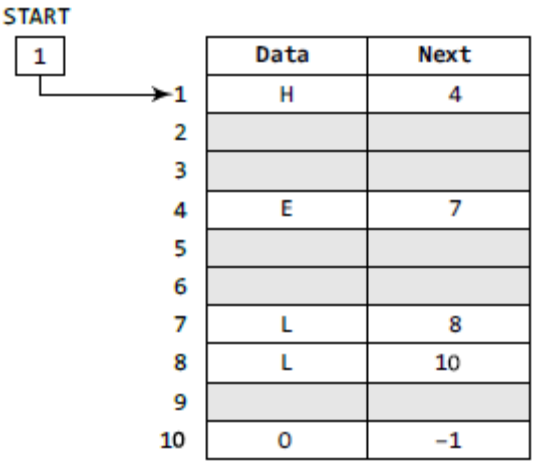


Figure 1.3 Memory Representation of a linked list

In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START = 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.

The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list. When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Note that Fig. 1.3 shows a chunk of memory locations which range from 1 to 10. The shaded portion contains data for other applications. Remember that the nodes of a linked list need not be in consecutive memory locations. In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.

1.4 OPERATIONS ON SINGLE LINKED LIST:

1.4.1 Inserting a New Node in a Linked List:

In this section, we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Inserting a Node at the Beginning of a Linked List:

Consider the linked list shown in Fig. 1.4. Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.

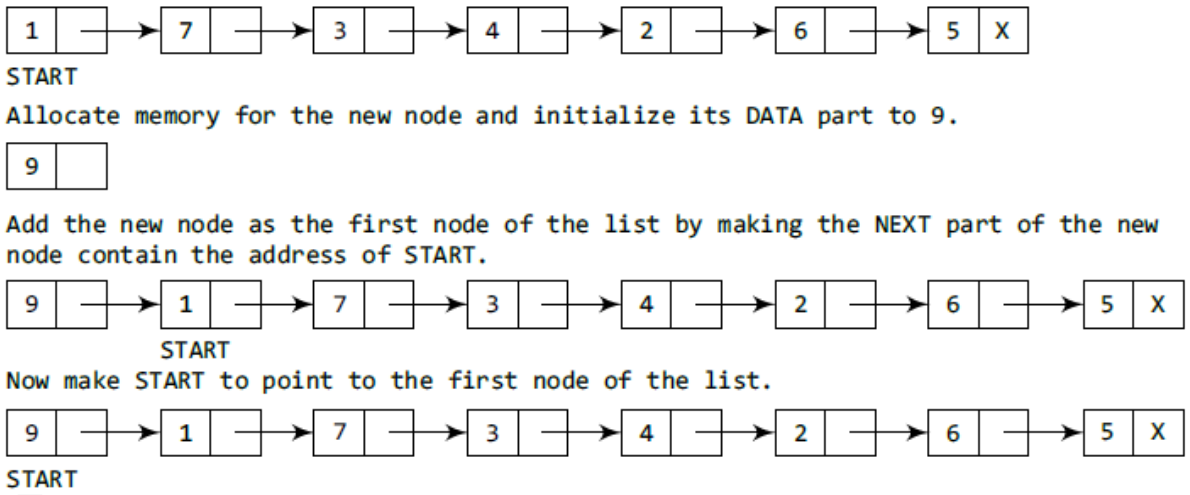


Figure 1.4 Inserting an element at the beginning of a linked list

Figure 1.5 shows the algorithm to insert a new node at the beginning of a linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if a free memory cell is

available, then we allocate space for the new node. Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE. Note the following two steps:

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc(), and calloc() which automatically do the memory allocation on behalf of the user.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 1.5 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Linked List:

Consider the linked list shown in Fig. 1.6. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

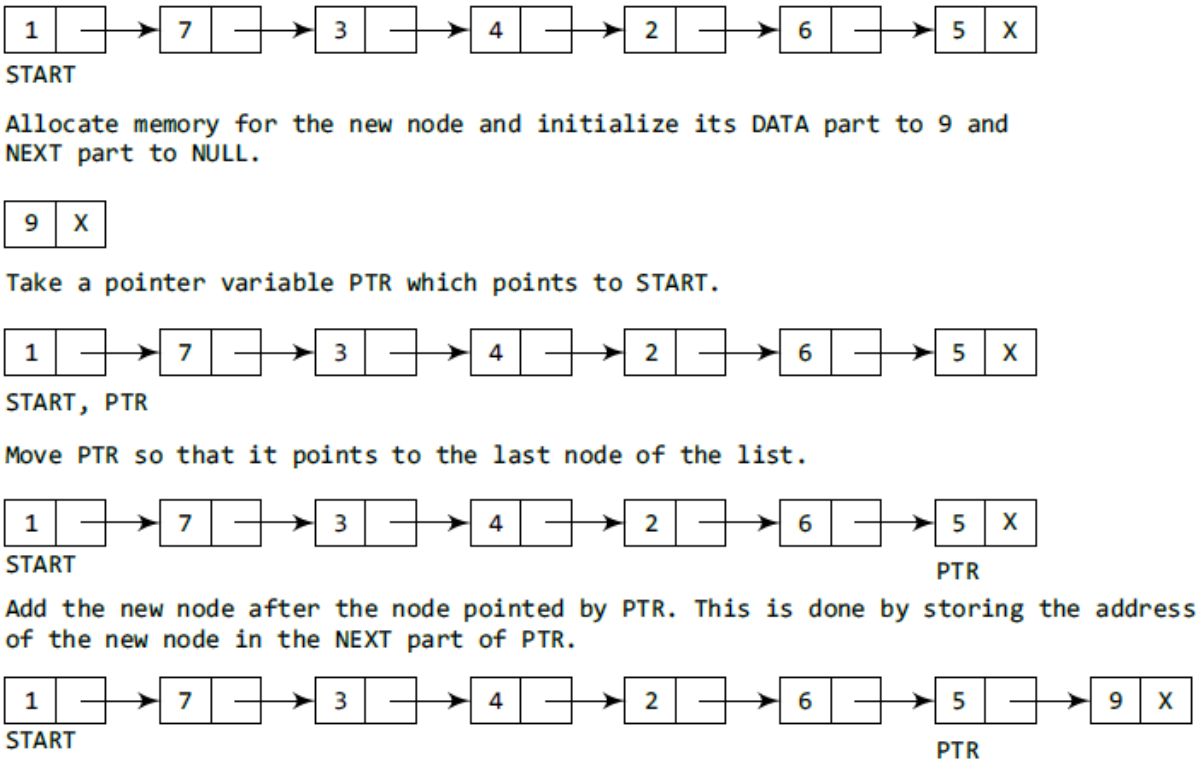


Figure 1.6 Inserting an element at the end of a linked list

Figure 1.7 shows the algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 9: SET PTR->NEXT = NEW_NODE
Step 10: EXIT
```

Figure 1.7 Algorithm to insert a new node at the end

Inserting a Node After a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.8. Suppose we want to add a new node with value 9 after the node containing data 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.9.

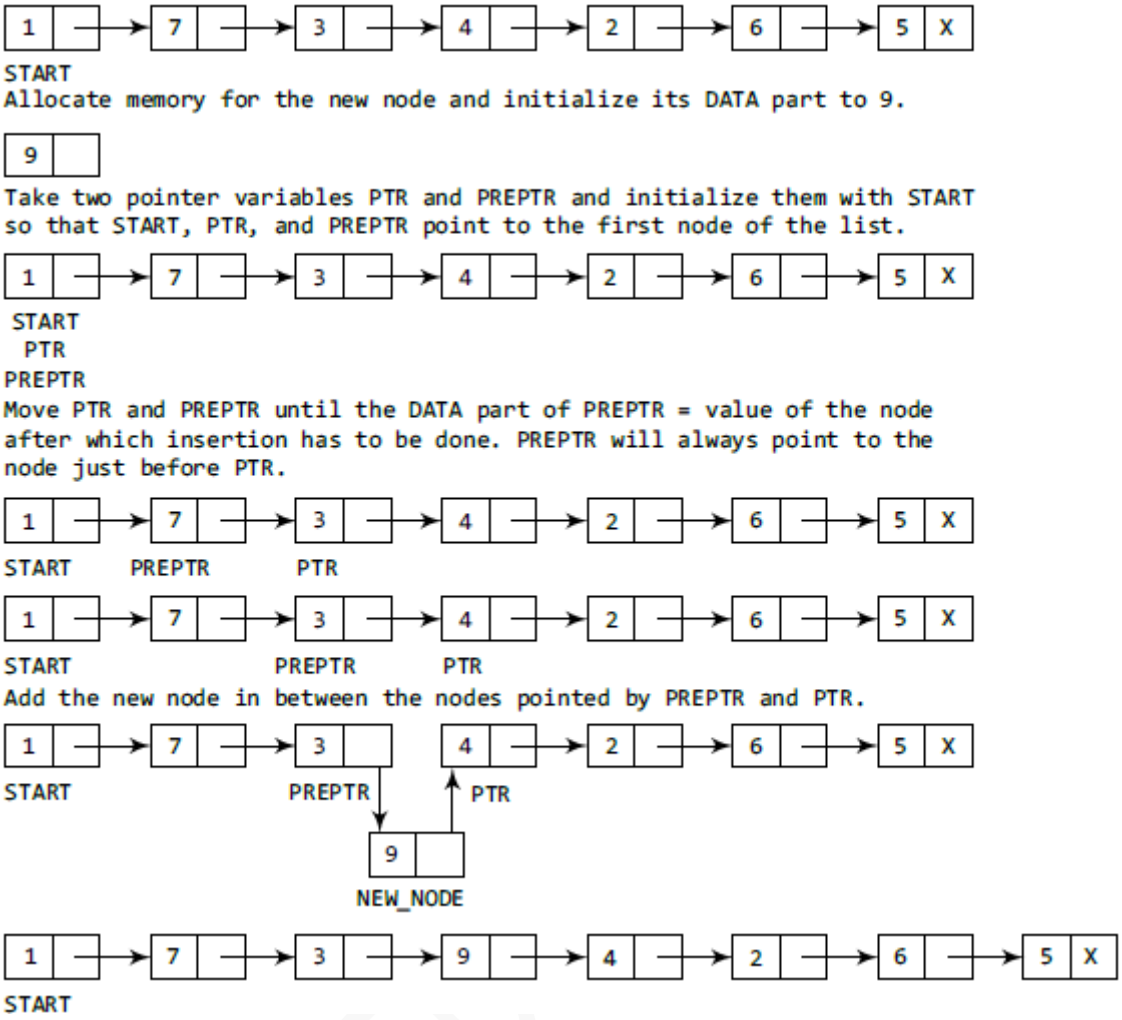


Figure 1.8 Inserting an element after a given node in a linked list

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 1.9 Algorithm to insert a new node after a node that has value NUM

Inserting a Node Before a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.10. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.11.

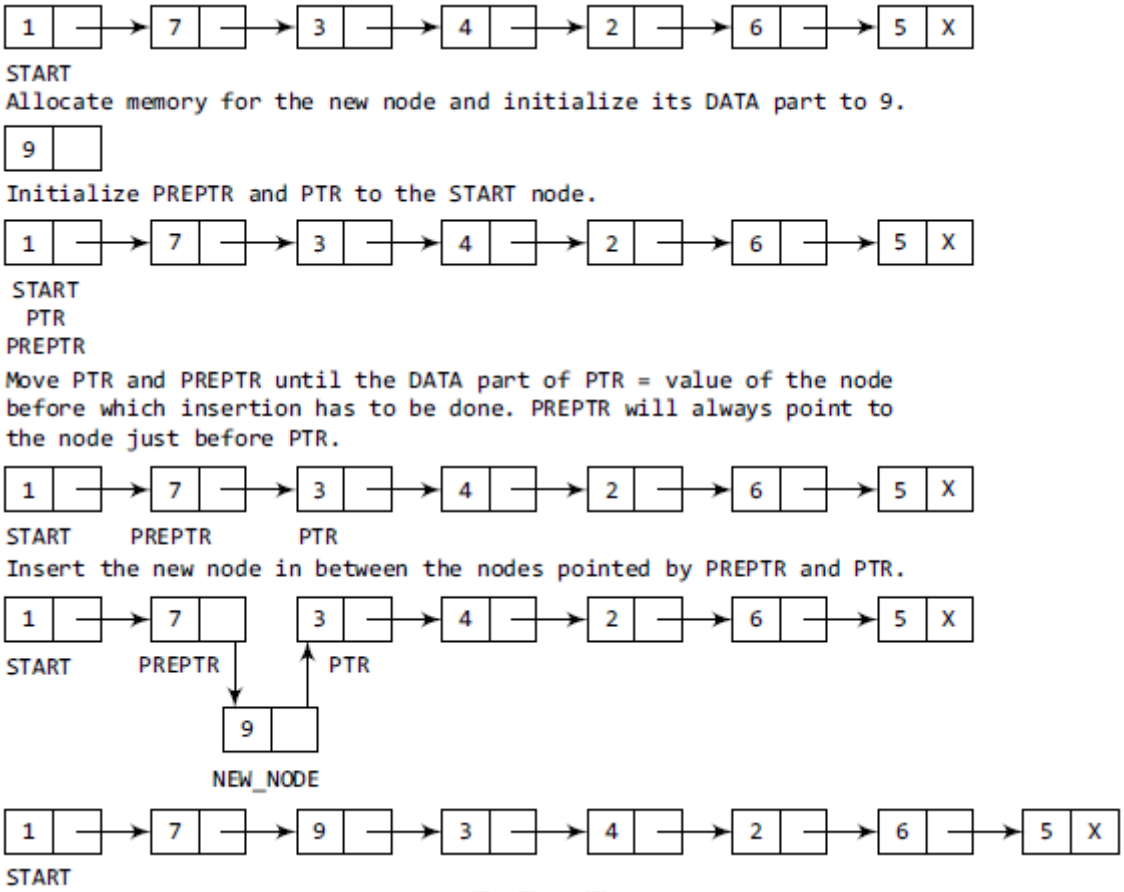


Figure 1.10 Inserting an element before a given node in a linked list

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

Figure 1.11 Algorithm to insert a new node before a node that has value NUM

1.4.2 Deleting a Node from a Linked List:

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Deleting the First Node from a Linked List:

Consider the linked list in Fig. 1.12. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

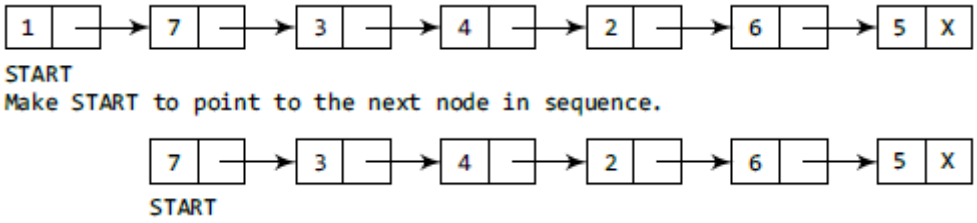


Figure 1.12 Deleting the first node of a linked list

Figure 1.13 shows the algorithm to delete the first node from a linked list. In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 1.13 Algorithm to delete the first node

Deleting the Last Node from a Linked List:

Consider the linked list shown in Fig. 1.14. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

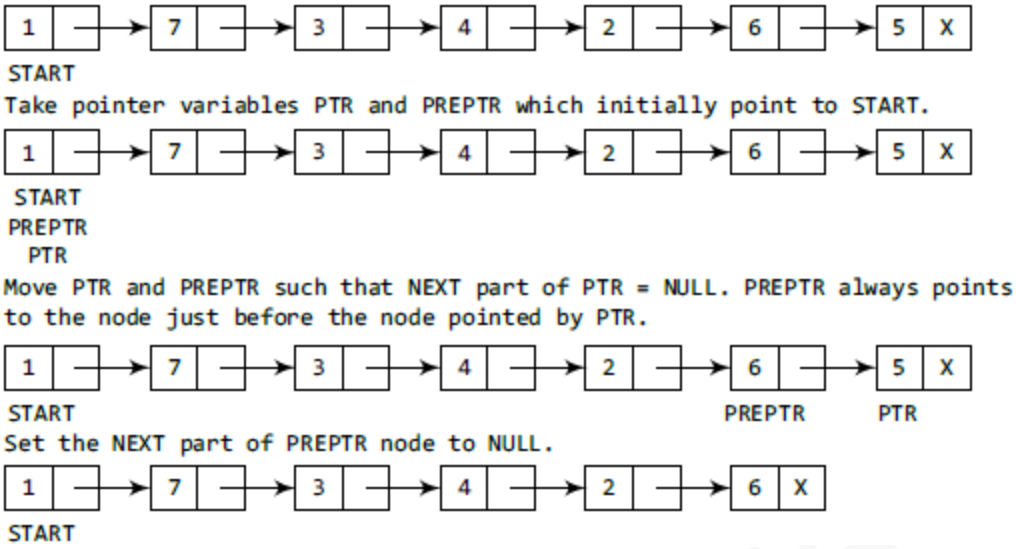


Figure 1.14 Deleting the last node of a linked list

Figure 1.15 shows the algorithm to delete the last node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned back to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR->NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 6: SET PREPTR->NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 1.15 Algorithm to delete the last node

Deleting the Node After a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.16. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

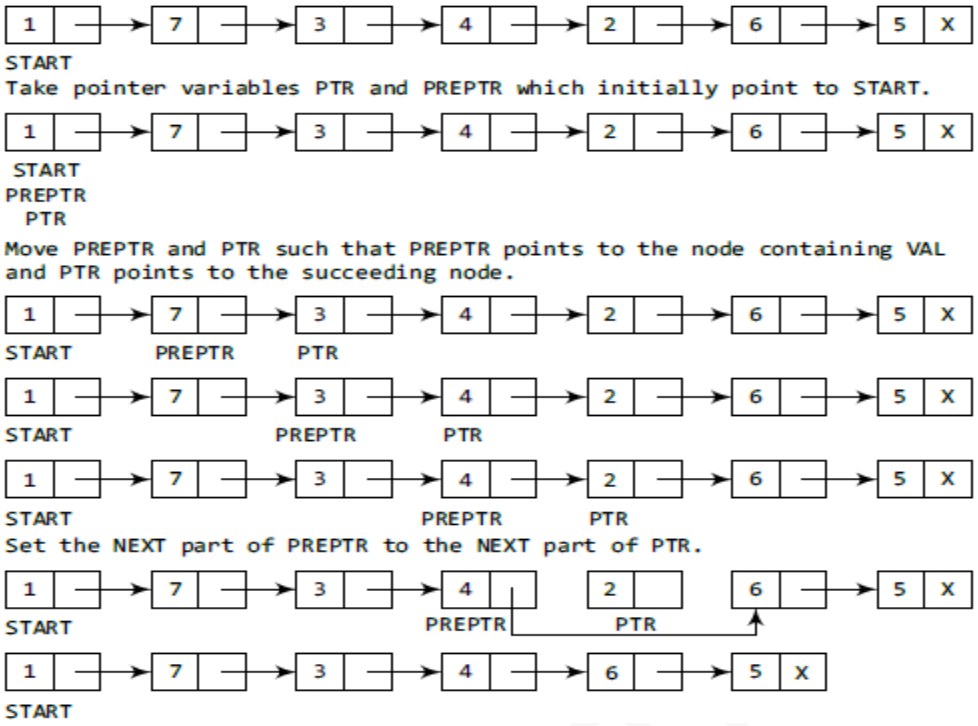


Figure 1.16 Deleting the node after a given node in a linked list

Figure 1.17 shows the algorithm to delete the node after a given node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Figure 1.17 Algorithm to delete the node after a given node

1.4.3 Searching for a Value in a Linked List:

Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                   SET POS = PTR
                   Go To Step 5
             ELSE
                   SET PTR = PTR -> NEXT
             [END OF IF]
       [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 1.18 Algorithm to search a linked list

Figure 1.18 shows the algorithm to search a linked list. In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made. If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Consider the linked list shown in Fig.1.19. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

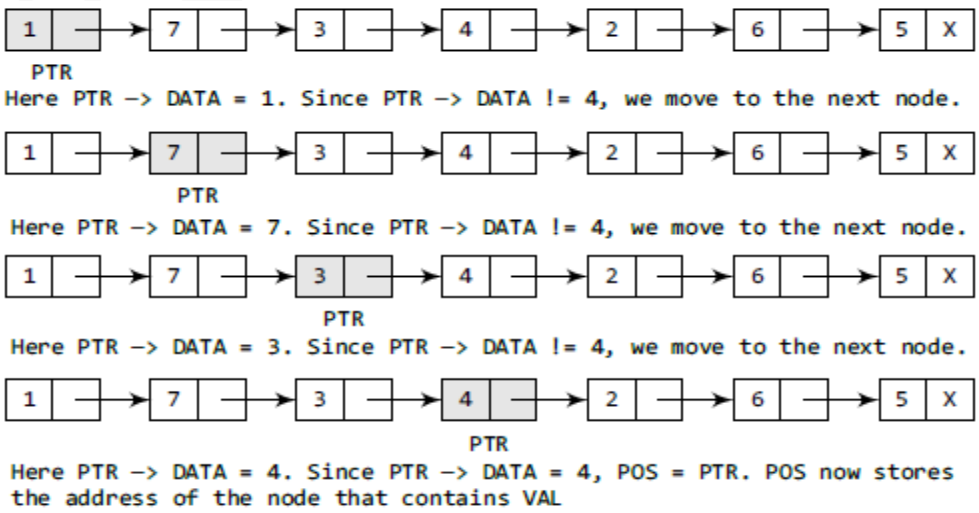


Figure 1.19 Searching a linked list

6.2.1 Traversing a Linked List:

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. The algorithm to traverse a linked list is shown in Fig. 1.20.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:           Apply Process to PTR->DATA
Step 4:           SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: EXIT
```

Figure 1.20 Algorithm for traversing a linked list

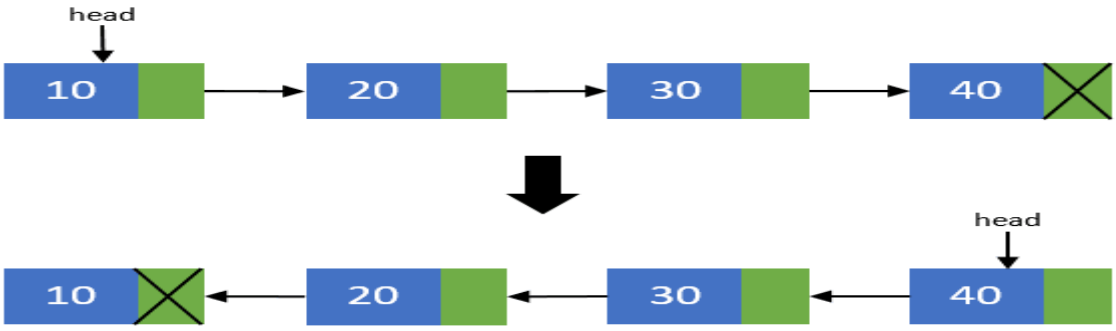
In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list. Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL. In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR. In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

Let us now write an algorithm to count the number of nodes in a linked list. To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Figure 1.21 shows the algorithm to print the number of nodes in a linked list.

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:           SET COUNT = COUNT + 1
Step 5:           SET PTR = PTR->NEXT
       [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Figure 1.21 Algorithm to print the number of nodes in a linked list

1.5 REVERSING SINGLE LINKED LIST:



Algorithm to reverse a Single Linked List:

Input: **head** node of the linked list

Begin:

If (head != **NULL**) then

prevNode ← head

head ← head.next

curNode ← head

prevNode.next ← **NULL**

While (head != **NULL**) do

head ← head.next

curNode.next ← prevNode

prevNode ← curNode

curNode ← head

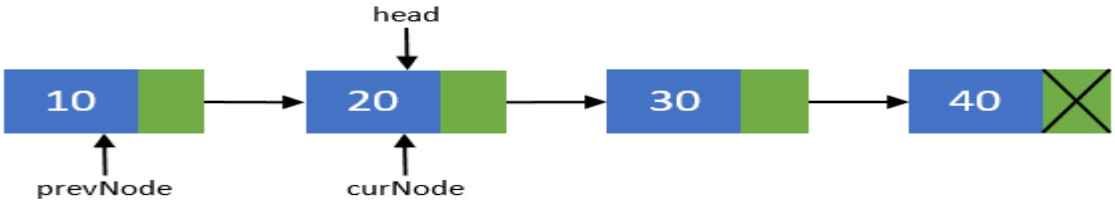
End while

head ← prevNode

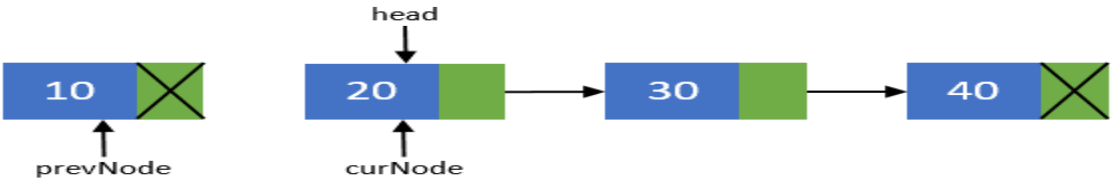
End if

End

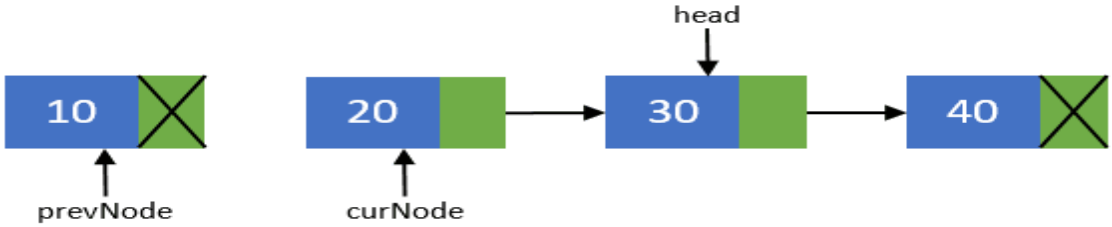
1) Create two more pointers other than head namely prevNode and curNode that will hold the reference of previous node and current node respectively. Make sure that prevNode points to first node i.e. prevNode = head. head should now point to its next node i.e. the second node head = head->next. curNode should also points to the second node i.e. curNode = head.



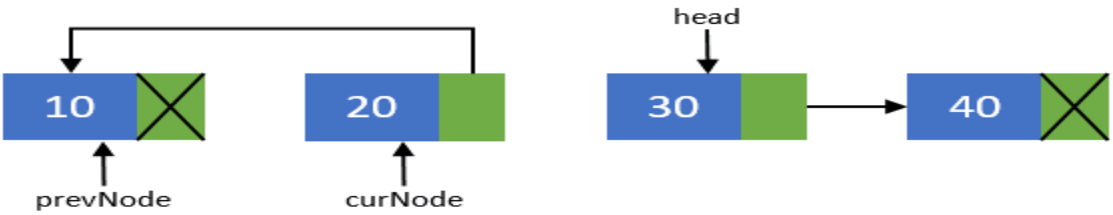
2) Now, disconnect the previous node i.e. the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation prevNode->next = NULL.



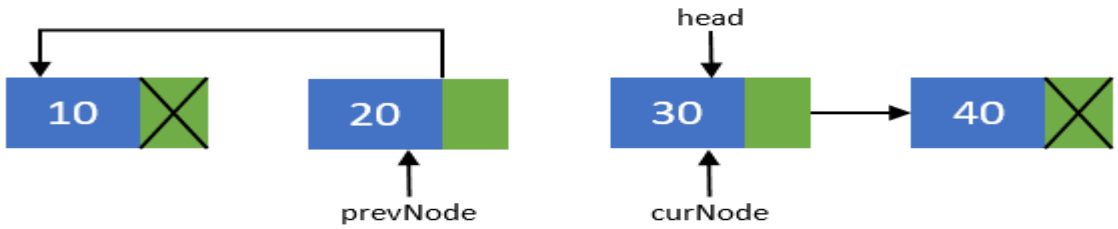
3) Move head node to its next node i.e. head = head->next.



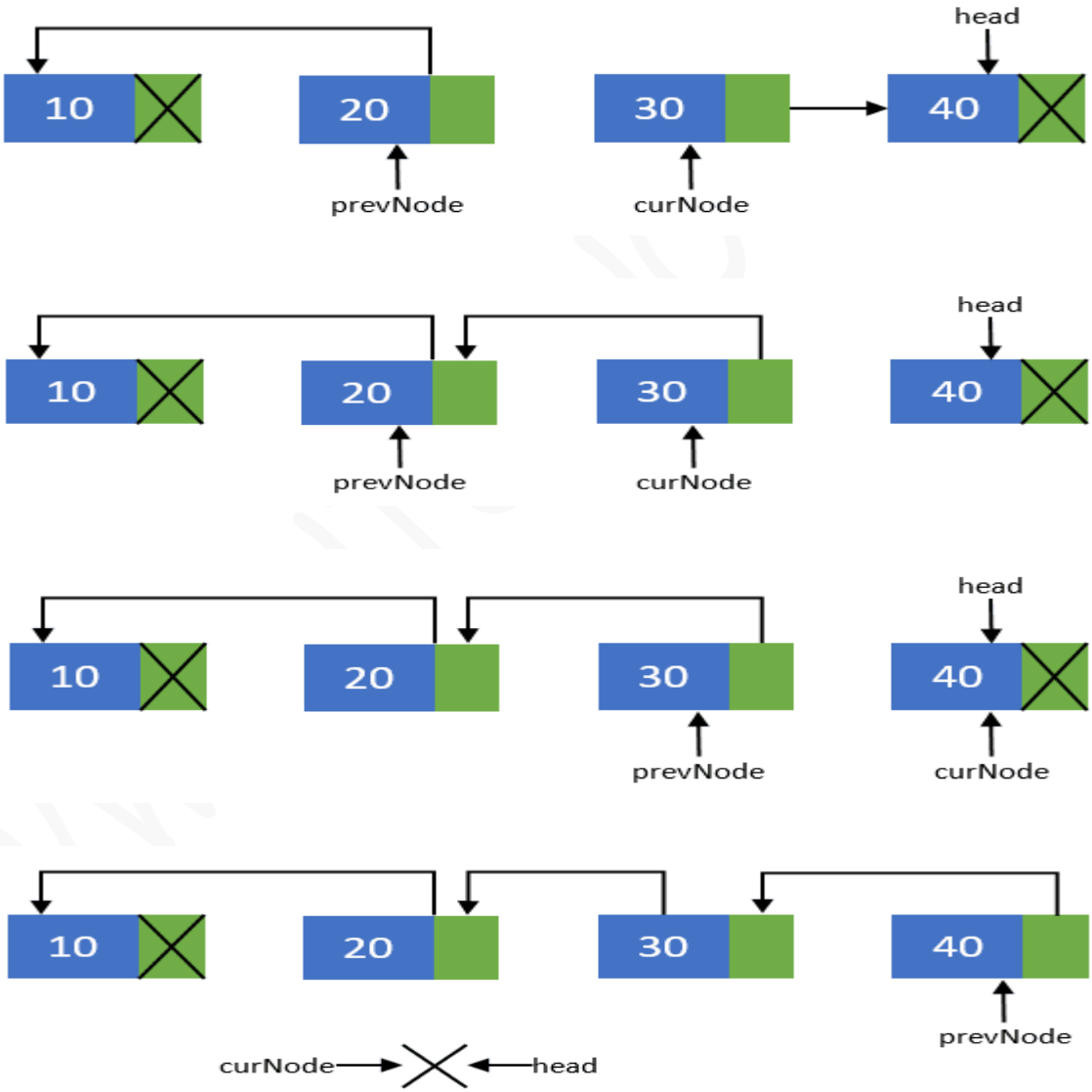
4) Now, re-connect the current node to its previous node i.e. curNode->next = prevNode;.



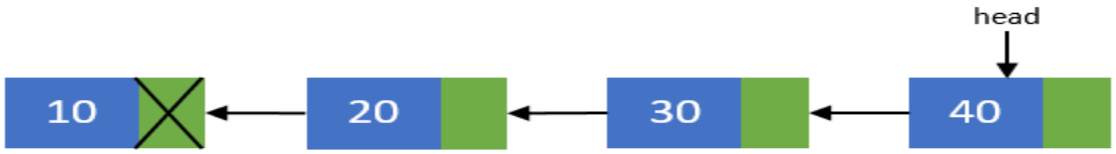
5) Point the previous node to current node and current node to head node. Means they should now point to prevNode = curNode; and curNode = head.



6) Repeat steps 3-5 till head pointer becomes NULL.



7) Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the **head** pointer should point to prevNode pointer. Perform head = prevNode;. Finally you end up with a reversed linked list of its original.



1.6 APPLICATIONS ON SINGLE LINKED LIST:

Linked lists can be used to represent polynomials and the different operations that can be performed on them. In this section, we will see how polynomials are represented in the memory using linked lists.

1.6.1 Polynomial Expression Representation:

Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

Every term of a polynomial can be represented as a node of the linked list. Figure 1.22 shows the linked representation of the terms of the above polynomial.

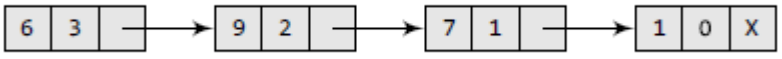


Figure 1.22 Linked representation of a polynomial

1.6.2 Addition and Multiplication:

Addition of Polynomials:

In this approach we will add the coefficients of elements having the same power in both the polynomial equations.

Example:

Input:

1st polynomial Expression = 5x^2 + 4x^1 + 2x^0

2nd polynomial Expression = 5x^1 + 5x^0

Output:

5x^2 + 9x^1 + 7x^0

Consider the below Fig 1.23, which adds the two polynomials. The two polynomials are represented in linked list. The elements which having the same power those corresponding coefficients are added.

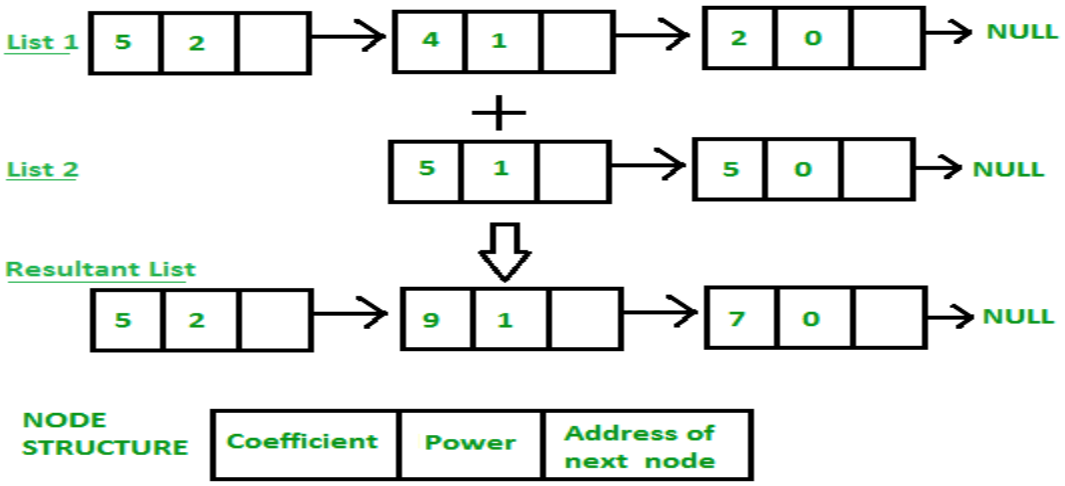


Figure 1.23 Addition of polynomials

Multiplication of Polynomials:

1. In this approach we will multiply the 2nd polynomial with each term of 1st polynomial.
2. Store the multiplied value in a new linked list.
3. Then we will add the coefficients of elements having the same power in resultant polynomial.

Example:

Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8

Output: 18x^3 + 54x^2 + 76x^1 + 48

On multiplying each element of 1st polynomial with elements of 2nd polynomial, we get

18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48

On adding values with same power of x,

18x^3 + 54x^2 + 76x^1 + 48

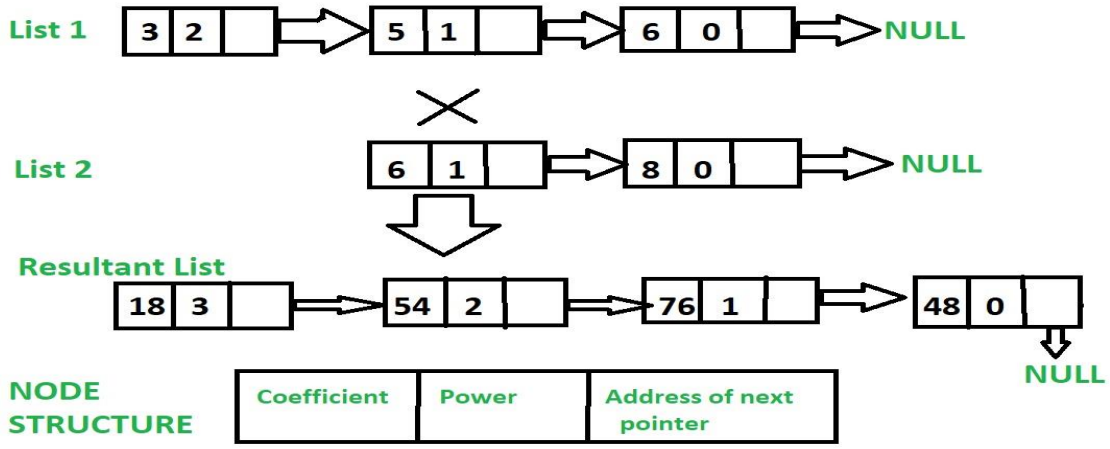


Figure 1.24 Multiplication of polynomials

1.6.3 Sparse Matrix Representation using Linked Lists:

Sparse Matrix:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

The sparse matrix shown in Fig. 1.25 can be represented using a linked list for every row and column. Since a value is in exactly one row and one column, it will appear in both lists exactly once. A node in the multi-linked will have four parts. First stores the data, second stores a pointer to the next node in the row, third stores a pointer to the next node in the column, and the fourth stores the coordinates or the row and column number in which the data appears in the matrix. However, as in case of doubly linked lists, we can also have a corresponding inverse pointer for every pointer in the multi-linked list representation of a sparse matrix.

y \ x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0

Figure 1.25 Sparse matrix

Note:

When a non-zero value in the sparse matrix is set to zero, the corresponding node in the multi-linked list must be deleted.

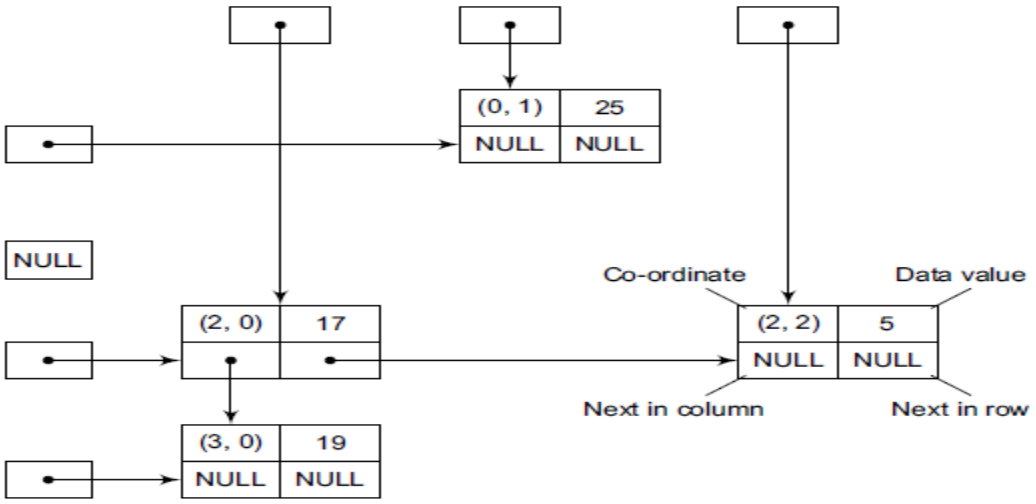


Figure 1.26 Multi-linked representation of sparse matrix shown in Fig. 1.25

1.7 ADVANTAGES AND DISADVANTAGES OF SINGLE LINKED LIST:

ADVANTAGES:

- Insertions and Deletions can be done easily.
- It does not need movement of elements for insertion and deletion.
- It space is not wasted as we can get space according to our requirements.
- Its size is not fixed.
- It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- It is less expensive.

DISADVANTAGES:

- It requires more space as pointers are also stored with information.
- Different amount of time is required to access each element.
- If we have to go to a particular element then we have to go through all those elements that come before that element.
- We cannot traverse it from last & only from the beginning.
- It is not easy to sort the elements stored in the linear linked list.

1.8 DOUBLE LINKED LIST:

A double linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Fig. 1.27

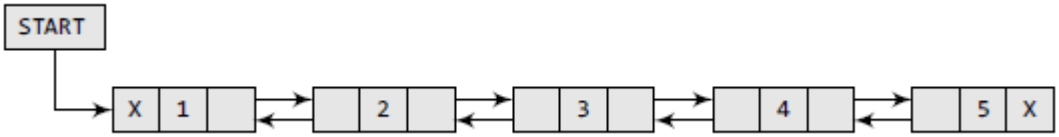


Figure 1.27 Double linked list

In C, the structure of a double linked list can be given as,

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

Thus, we see that a double linked list calls for more space per node and more expensive basic operations. However, a double linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a double linked list is that it makes searching twice as efficient. Let us view how a double linked list is maintained in the memory.

Consider Fig. 1.28

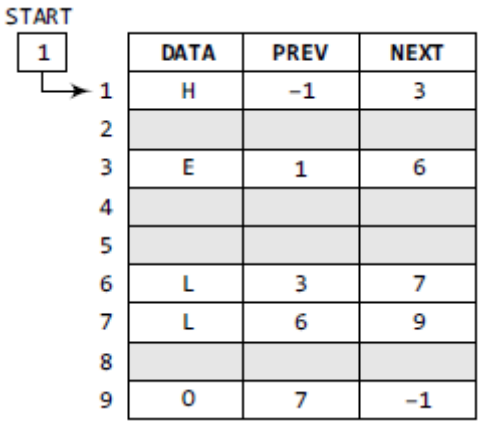


Figure 1.28 Memory representation of a doubly linked list

In the figure, we see that a variable START is used to store the address of the first node. In this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field. We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

1.8.1 Inserting a New Node in a Double Linked List:

In this section, we will discuss how a new node is added into an already existing double linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

Inserting a Node at the Beginning of a Double Linked List:

Consider the double linked list shown in Fig. 1.29 Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

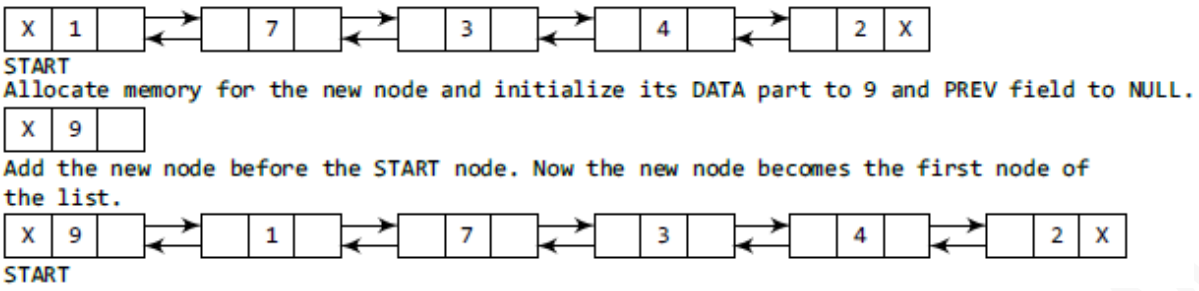


Figure 1.29 Inserting a new node at the beginning of a double linked list

Figure 1.30 shows the algorithm to insert a new node at the beginning of a double linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Figure 1.30 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Double Linked List:

Consider the double linked list shown in Fig. 1.31. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

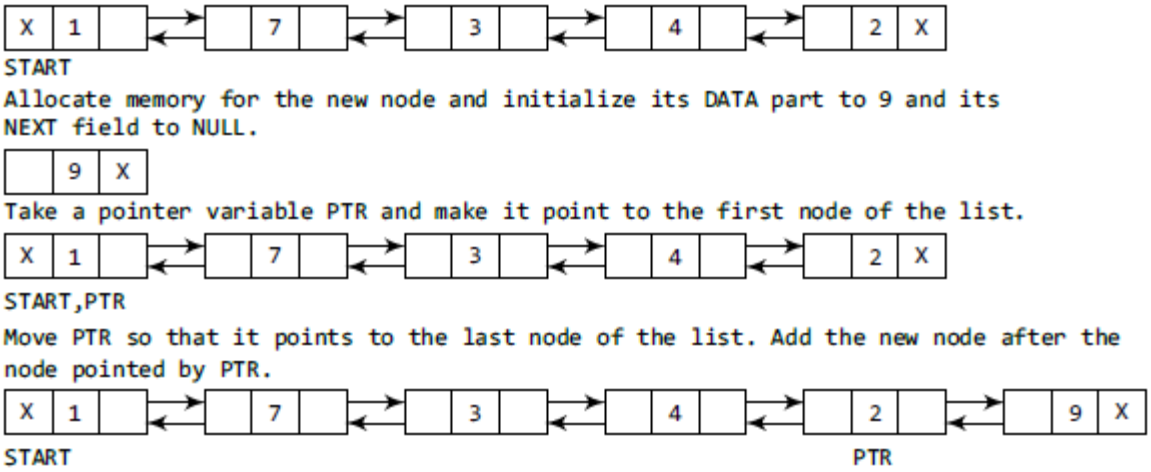


Figure 1.31 Inserting a new node at the end of a double linked list

Figure 1.32 shows the algorithm to insert a new node at the end of a double linked list. In Step 6, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Figure 1.32 Algorithm to insert a new node at the end

Inserting a Node After a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.33. Suppose we want to add a new node with value 9 after the node containing 3.

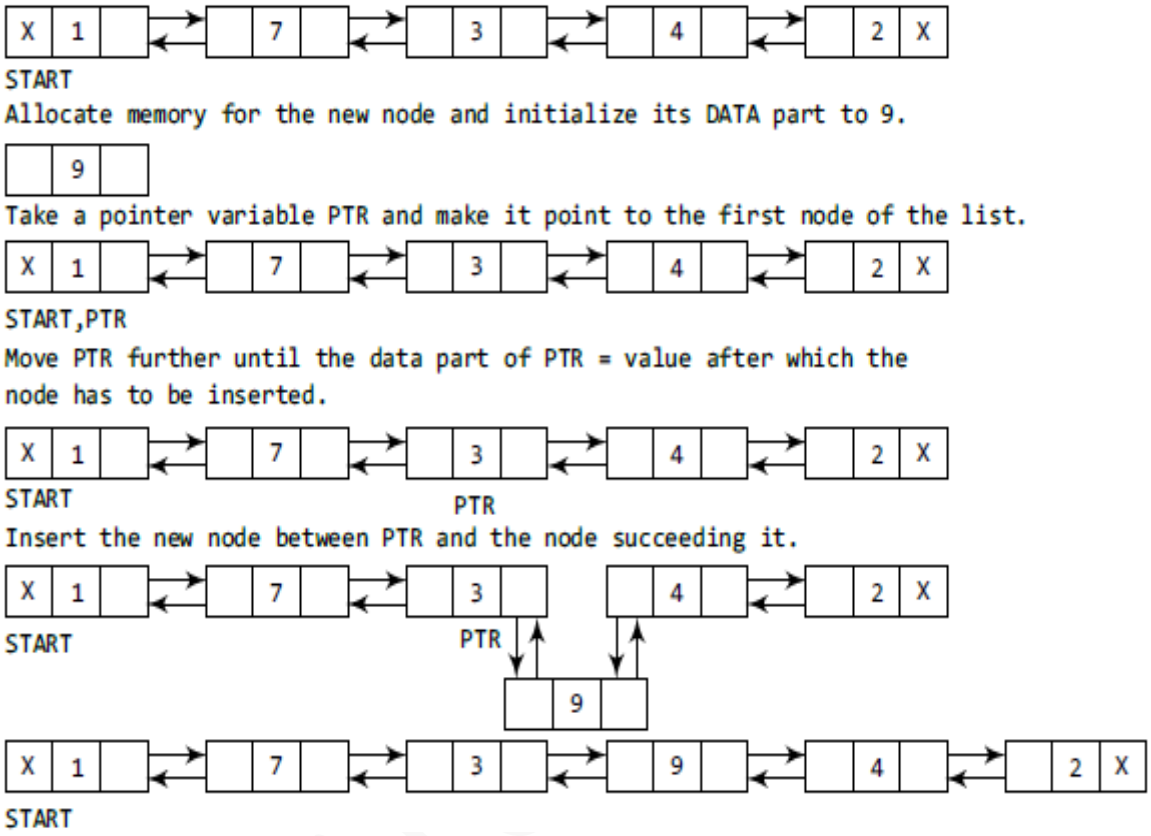


Figure 1.33 Inserting a new node after a given node in a double linked list

Figure 1.34 shows the algorithm to insert a new node after a given node in a double linked list. In Step 5, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 1.34 Algorithm to insert a new node after a given node

Inserting a Node Before a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.35. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.36 In Step 1, we first check whether memory is available for the new node. In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

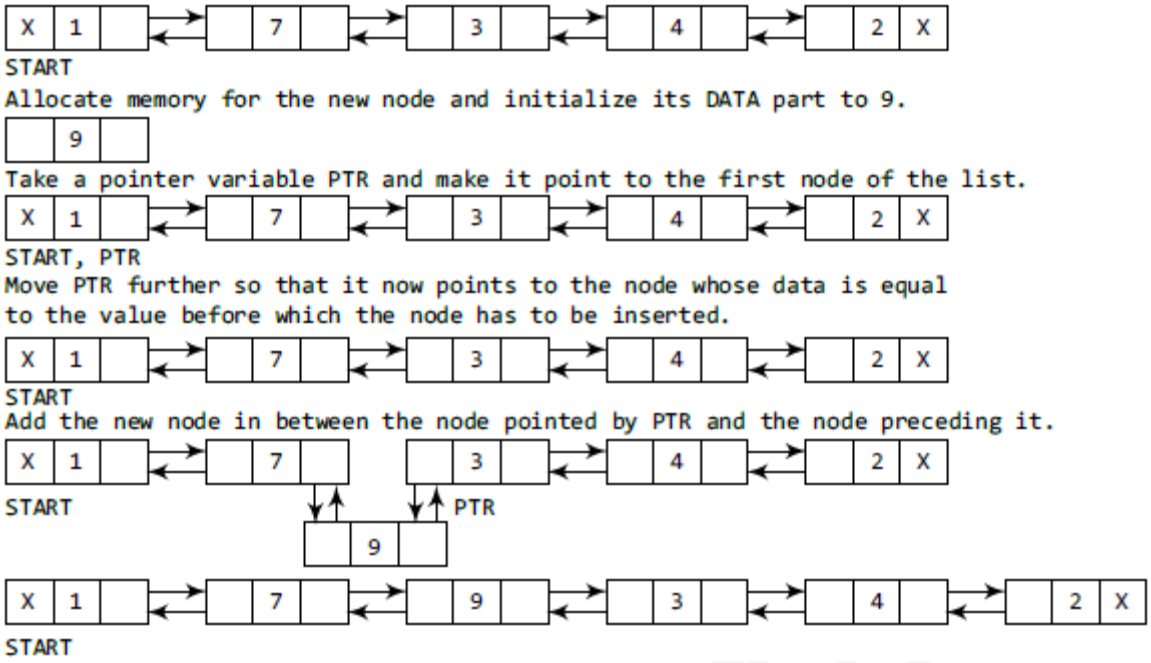


Figure 1.35 Inserting a new node before a given node in a double linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR  -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

Figure 1.36 Algorithm to insert a new node before a given node

1.8.2 Deleting a Node from a Double Linked List:

In this section, we will see how a node is deleted from an already existing double linked list. We will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 4: The node before a given node is deleted.

Deleting the First Node from a Double Linked List:

Consider the double linked list shown in Fig. 1.37. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

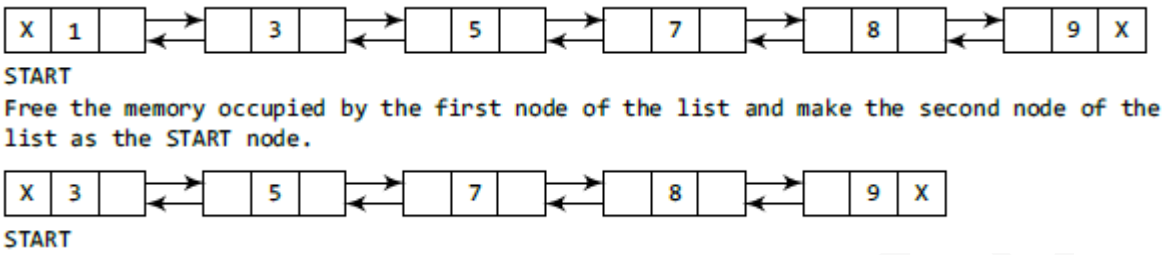


Figure 1.37 Deleting the first node from a double linked list

Figure 1.38 shows the algorithm to delete the first node of a double linked list. In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Figure 1.38 Algorithm to delete the first node

Deleting the Last Node from a Double Linked List:

Consider the double linked list shown in Fig. 1.39. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

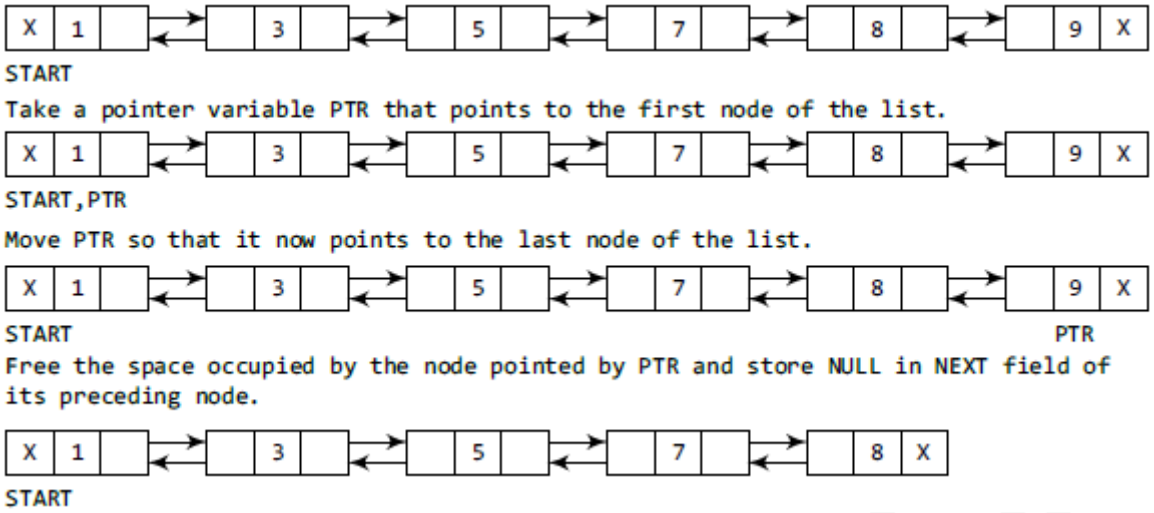


Figure 1.39 Deleting the last node from a double linked list

Figure 1.40 shows the algorithm to delete the last node of a double linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Figure 1.40 Algorithm to delete the last node

Deleting the Node After a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.41. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

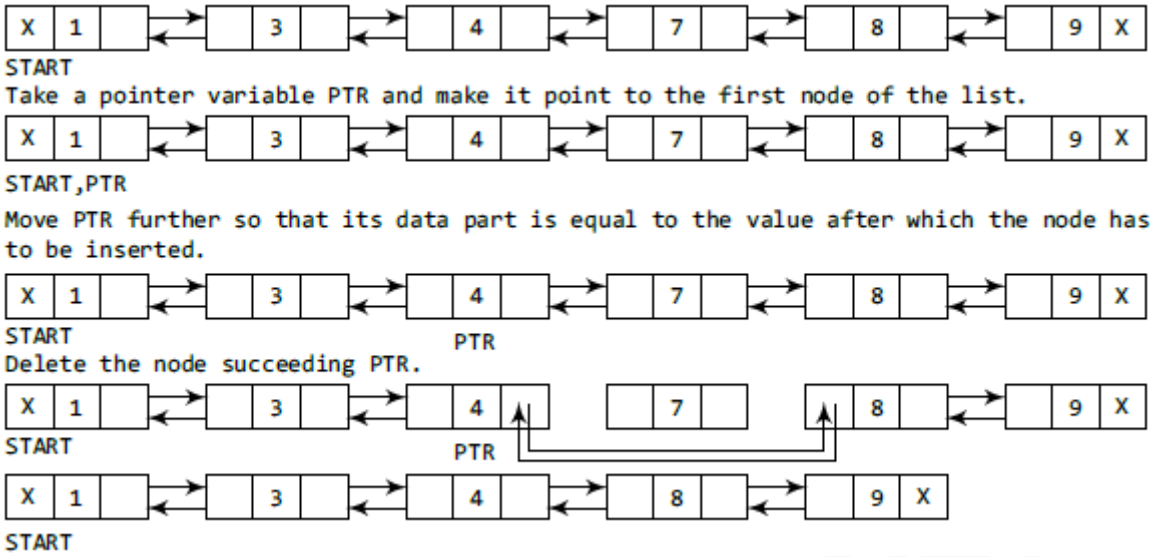


Figure 1.41 Deleting the node after a given node in a double linked list

Figure 1.42 shows the algorithm to delete a node after a given node of a double linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->DATA != NUM
Step 4:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR->NEXT
Step 6: SET PTR->NEXT = TEMP->NEXT
Step 7: SET TEMP->NEXT->PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Figure 1.42 Algorithm to delete a node after a given node

Deleting the Node Before a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.43. Suppose we want to delete the node preceding the node with value 4. Before discussing the changes that will be done in the linked list, let us first look at the algorithm.

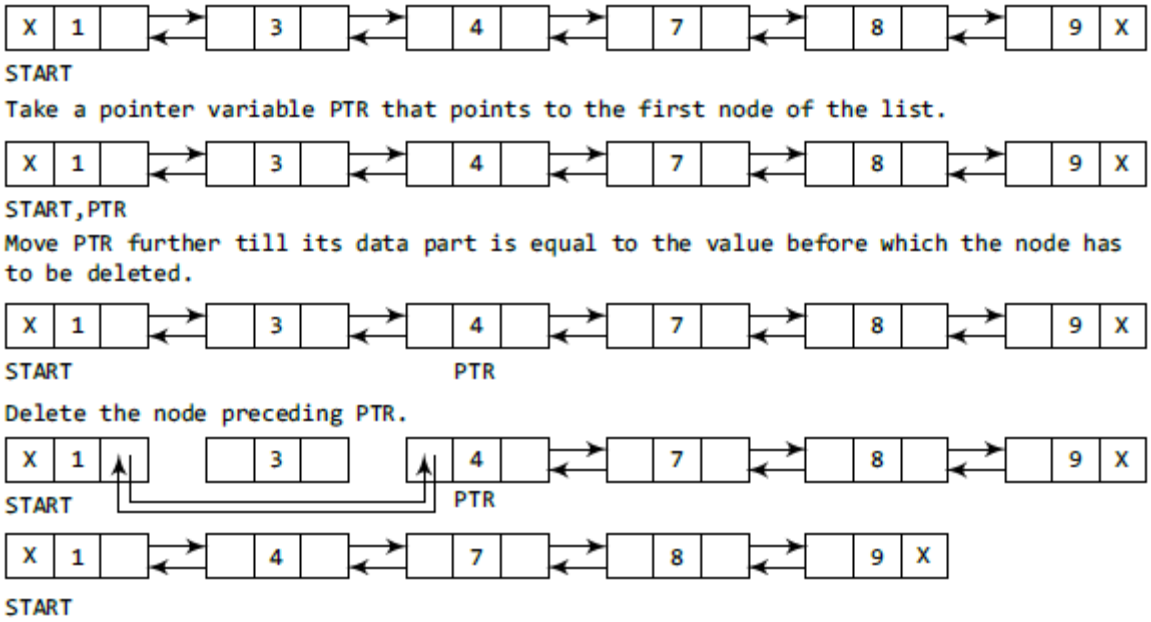


Figure 1.43 Deleting a node before a given node in a double linked list

Figure 1.44 shows the algorithm to delete a node before a given node of a double linked list. In Step2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the linked list to reach the desired node. Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR. The memory of the node preceding PTR is freed and returned to the free pool. Hence, we see that we can insert or delete a node in a constant number of operations given only that node's address. Note that this is not possible in the case of a singly linked list which requires the previous node's address also to perform the same operation.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

Figure 1.44 Algorithm to delete a node before a given node

1.9 CIRCULAR LINKED LIST:

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list.

1) Circular Single Linked List:

While traversing a circular Single linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending. Figure 1.45 shows a circular Single linked list.

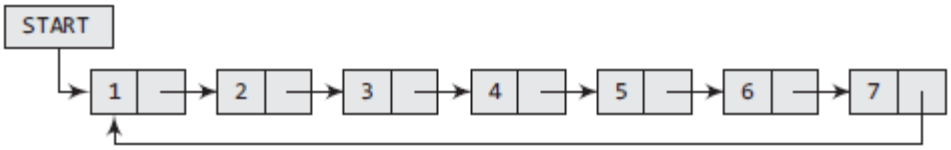


Figure 1.45 Circular Single linked list

The only downside of a circular linked list is the complexity of iteration. Note that there are no NULL values in the NEXT part of any of the nodes of list.

Consider Fig. 1.46.

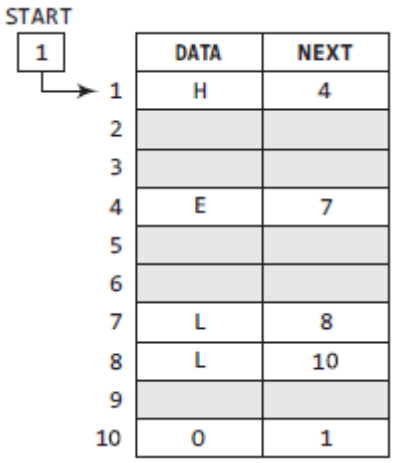


Figure 1.46 Memory representation of a circular single linked list

We can traverse the list until we find the NEXT entry that contains the address of the first node of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in Fig. 1.46 stores characters that when put together form the word HELLO.

1.9.1 Inserting a New Node in a Circular Linked List:

In this section, we will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

Inserting a Node at the Beginning of a Circular Linked List:

Consider the linked list shown in Fig. 1.47. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

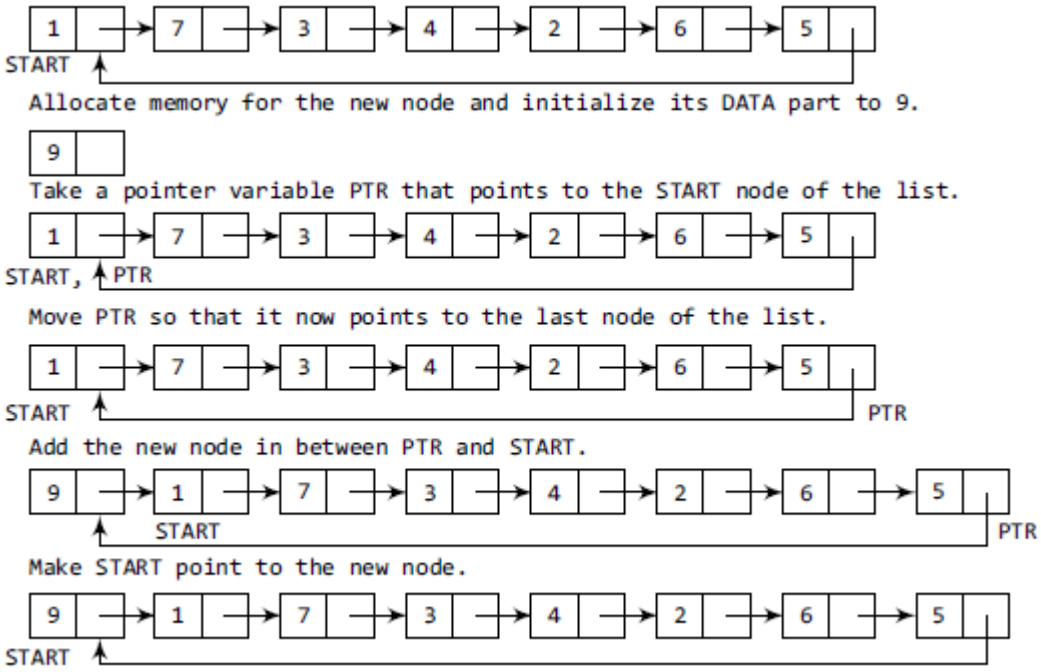


Figure 1.47 Inserting a new node at the beginning of a circular linked list

Figure 1.48 shows the algorithm to insert a new node at the beginning of a linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.

While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Figure 1.48 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Linked List:

Consider the linked list shown in Fig. 1.49. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

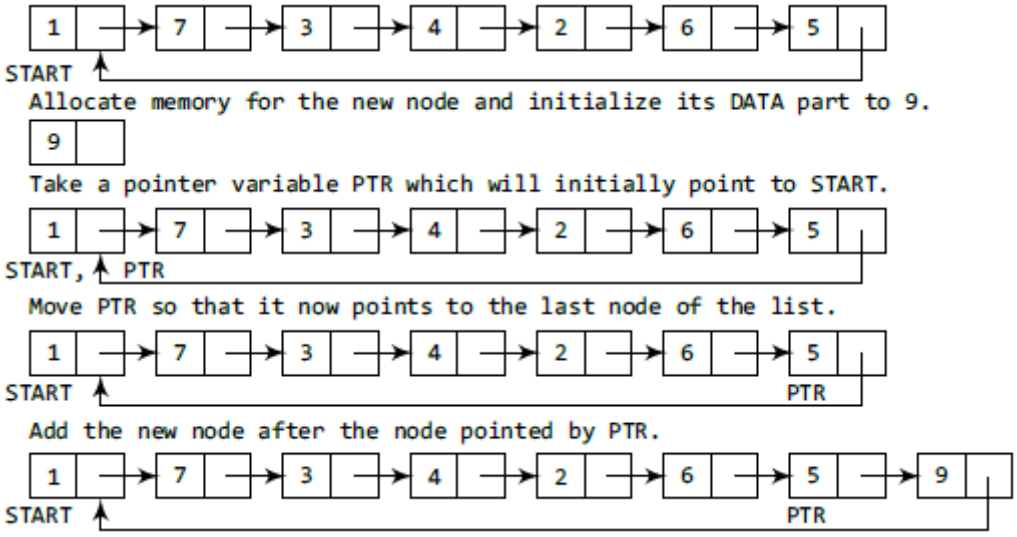


Figure 1.49 Inserting a new node at the end of a circular linked list

Figure 1.50. shows the algorithm to insert a new node at the end of a circular linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 1.50 Algorithm to insert a new node at the end

1.9.2 Deleting a Node from a Circular Linked List

In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Deleting the First Node from a Circular Linked List:

Consider the circular linked list shown in Fig. 1.51. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

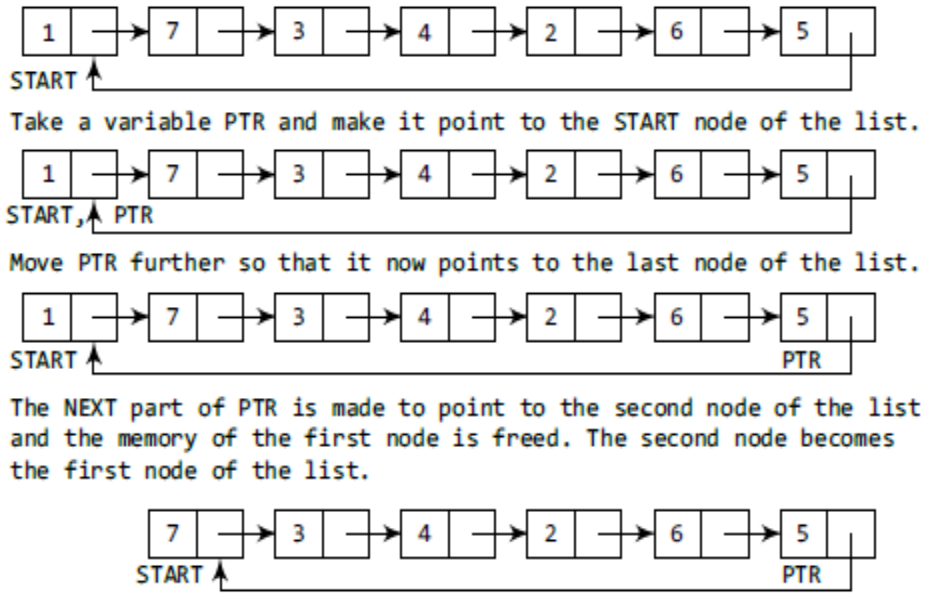


Figure 1.51 Deleting the first node from a circular linked list

Figure 1.52 shows the algorithm to delete the first node from a circular linked list. In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node. In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list. In Step 6, the memory occupied by the first node is freed. Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->NEXT != START
Step 4:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: SET PTR->NEXT = START->NEXT
Step 6: FREE START
Step 7: SET START = PTR->NEXT
Step 8: EXIT
```

Figure 1.52 Algorithm to delete the first node

Deleting the Last Node from a Circular Linked List:

Consider the circular linked list shown in Fig. 1.53. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

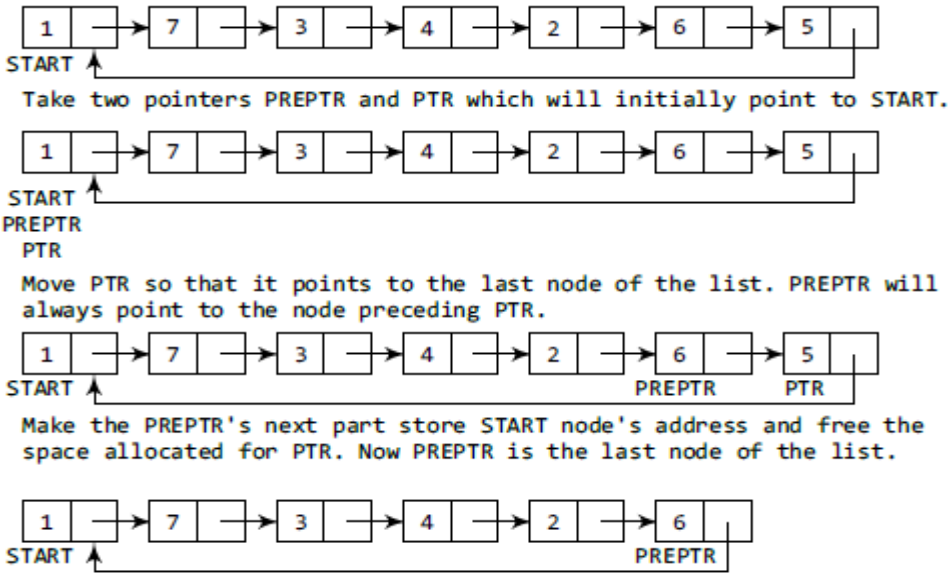


Figure 1.53 Deleting the last node from a circular linked list

Figure 1.54 shows the algorithm to delete the last node from a circular linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR. Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

Figure 1.54 Algorithm to delete the last node

2) Circular Doubly Linked List:

A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list. The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., START. Similarly, the previous field of the first field stores the address of the last node. A circular doubly linked list is shown in Fig. 1.55.

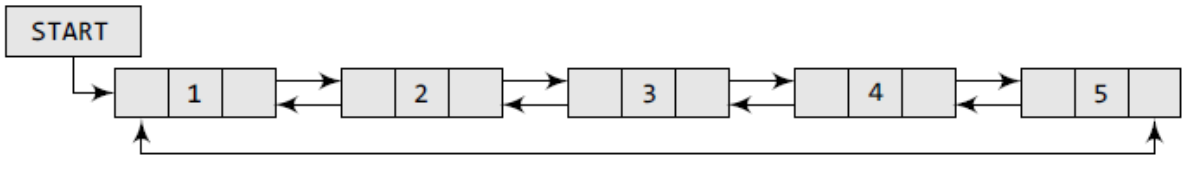


Figure 1.55 Circular doubly linked list

Since a circular doubly linked list contains three parts in its structure, it calls for more space per node and more expensive basic operations. However, a circular doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in

both the directions (forward and backward). The main advantage of using a circular doubly linked list is that it makes search operation twice as efficient.

Let us view how a circular doubly linked list is maintained in the memory. Consider Fig. 1.56. In the figure, we see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it stores the address of the last node of the list in its previous field. The corresponding NEXT stores the address of the next node, which is 3. So, we will look at address 3 to fetch the next data item. The previous field will contain the address of the first node. The second data element obtained from address 3 is E. We repeat this procedure until we reach a position where the NEXT entry stores the address of the first element of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

START = 1

	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	1

Figure 1.56 Memory representation of a circular doubly linked list

1.9.3 Inserting a New Node in a Circular Doubly Linked List:

In this section, we will see how a new node is added into an already existing circular doubly linked list. We will take two cases and then see how insertion is done in each case. Rest of the cases are similar to that given for doubly linked lists.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Inserting a Node at the Beginning of a Circular Doubly Linked List:

Consider the circular doubly linked list shown in Fig. 1.57. Suppose we want to add a new node with data 9 as the first node of the list. Then, the following changes will be done in the linked list.

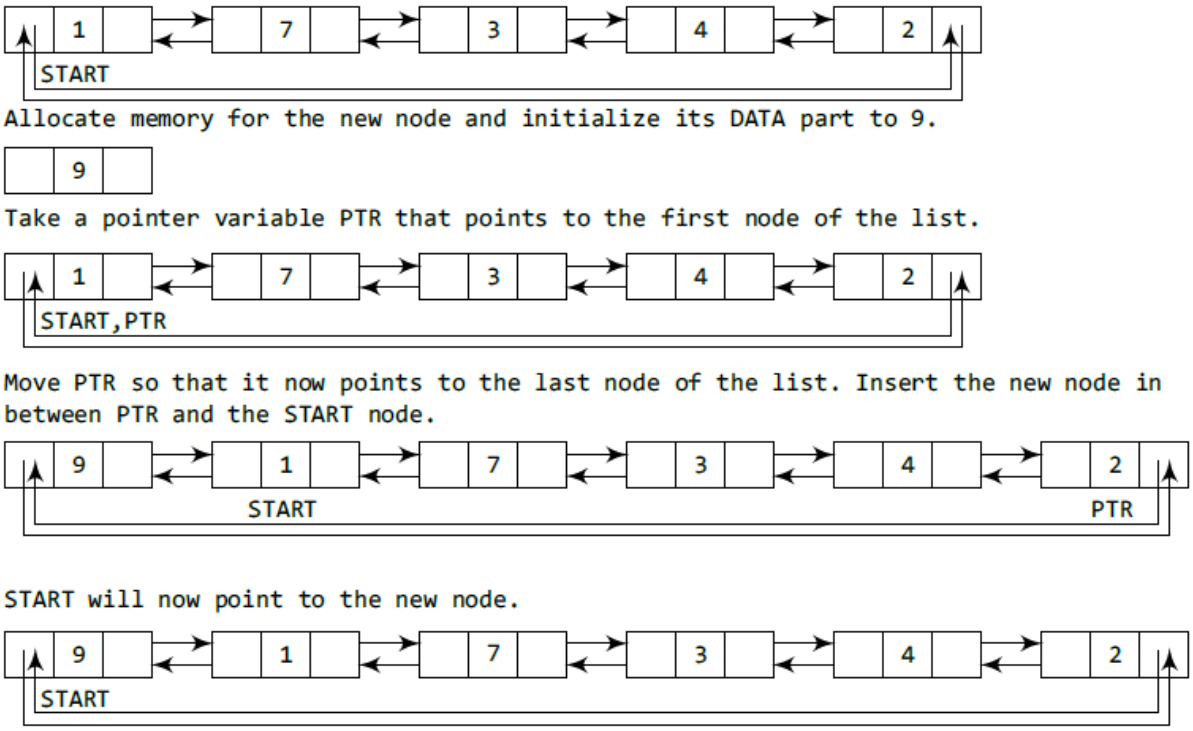


Figure 1.57 Inserting a new node at the beginning of a circular doubly linked list

Figure 1.58 shows the algorithm to insert a new node at the beginning of a circular doubly linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, we allocate space for the new node. Set its data part with the given VAL and its next part is initialized with the address of the first node of the list, which is stored in START. Now since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE. Since it is a circular doubly linked list, the PREV field of the NEW_NODE is set to contain the address of the last node.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 13
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```

Figure 1.58 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Doubly Linked List:

Consider the circular doubly linked list shown in Fig. 1.59. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

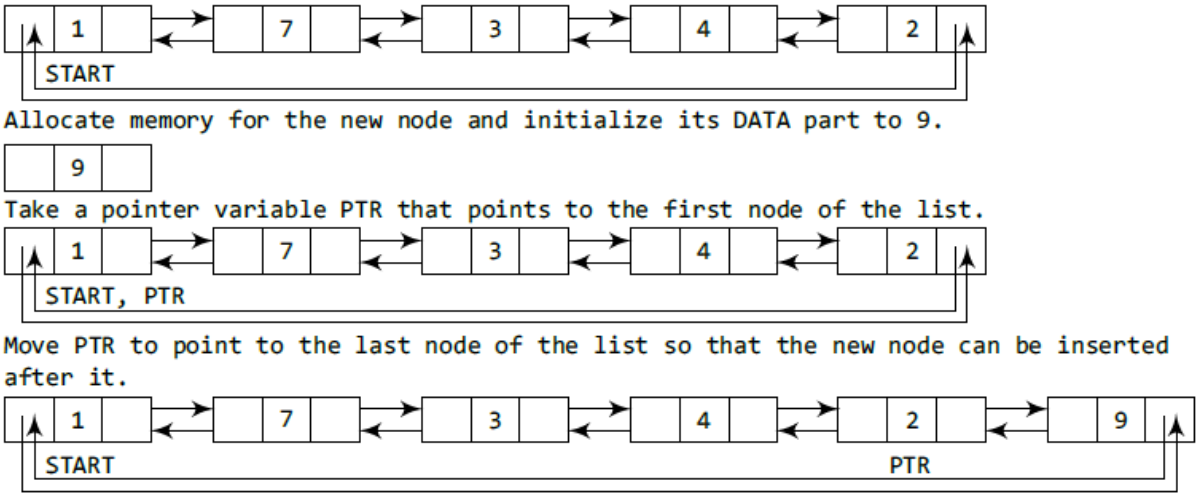


Figure 1.59 Inserting a new node at the end of a circular doubly linked list

Figure 1.60 shows the algorithm to insert a new node at the end of a circular doubly linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the

NEXT pointer of the last node to store the address of the new node. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 1.60 Algorithm to insert a new node at the end

1.9.4 Deleting a Node from a Circular Doubly Linked List:

In this section, we will see how a node is deleted from an already existing circular doubly linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases are same as that given for doubly linked lists.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Deleting the First Node from a Circular Doubly Linked List:

Consider the circular doubly linked list shown in Fig. 1.61. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

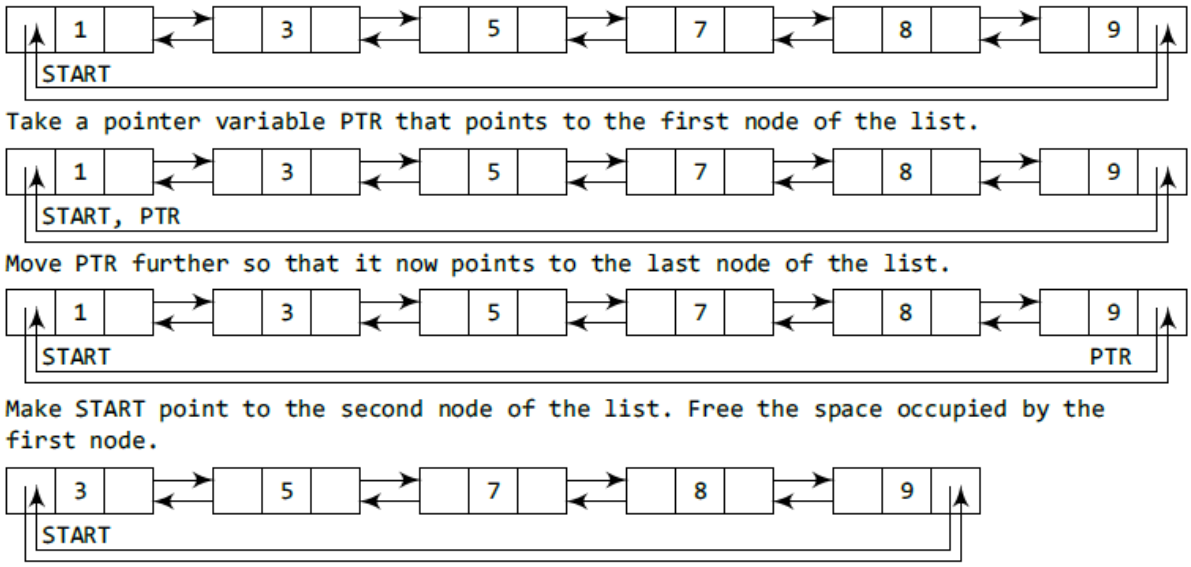


Figure 1.61 Deleting the first node from a circular doubly linked list

Figure 1.62 shows the algorithm to delete the first node from a circular doubly linked list. In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.

However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. The while loop traverses through the list to reach the last node. Once we reach the last node, the NEXT pointer of PTR is set to contain the address of the node that succeeds START. Finally, START is made to point to the next node in the sequence and the memory occupied by the first node of the list is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: SET START -> NEXT -> PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR -> NEXT
```

Figure 1.62 Algorithm to delete the first node

Deleting the Last Node from a Circular Doubly Linked List:

Consider the circular doubly linked list shown in Fig. 1.63. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

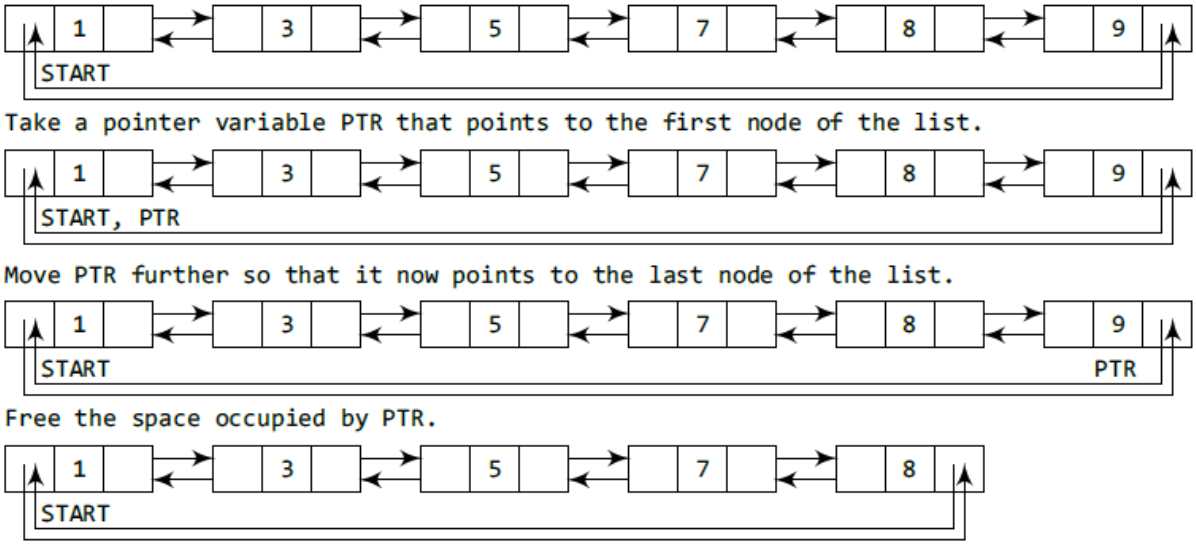


Figure 1.63 Deleting the last node from a circular doubly linked list

Figure 1.64 shows the algorithm to delete the last node from a circular doubly linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of the second last node to contain the address of START, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4  while PTR -> NEXT != START
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```

Figure 1.64 Algorithm to delete the last node

UNIT-III

QUEUES

1. Introduction of Queues

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure. A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists. In this section, we will see how queues are implemented using each of these data structures.

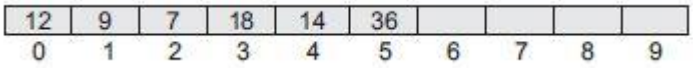
2. Representation Of Queues

In this we have two types of Representation Of Queues. They are:

1. Array Representation
2. Linked list Representation

1. Array Representation:

- Queues can be easily represented using linear arrays
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.



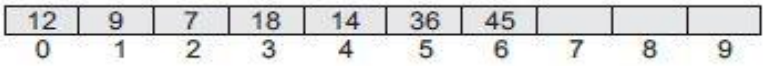
ARRAY REPRESENTATION OF QUEUE

OPERATION ON QUEUES

We have two types of operations on Queues. They are:

1. INSERTION
2. DELETION

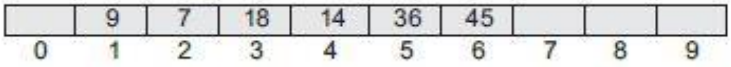
1.INSERTION:



QUEUE AFTER INSERTION OF A NEW ELEMENT

- FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.
- Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.
- However, before inserting an element in a queue, we must check for overflow conditions
- An overflow will occur when we try to insert an element into a queue that is already full.
- When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0.

2. DELETION:



QUEUE AFTER DELETION OF AN ELEMENT

- If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue.
- Similarly, before deleting an element from a queue, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a queue that is already empty.
- If FRONT = –1 and REAR = –1, it means there is no element in the queue.

ALGORITHM TO INSERT AN ELEMENT IN A QUEUE

Step 1: IF REAR = MAX-1
Write OVERFLOW
Goto step 4
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR =
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT

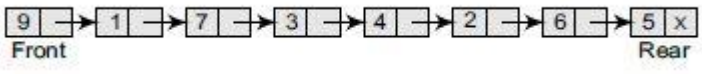
ALGORITHM TO DELETE AN ELEMENT FROM A QUEUE

Step 1: IF FRONT = -1 OR FRONT > REAR
Write UNDERFLOW
ELSE
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT

2. LINKED LIST REPRESENTATION:

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.



LINKED QUEUE

OPERATION ON LINKED QUEUES

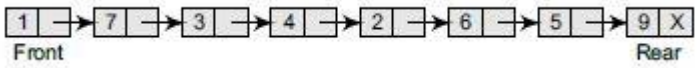
- A queue has two basic operations: insert and delete.
- The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue. Apart from

this, there is another operation peek which returns the value of the first element of the queue.

1. INSERT OPERATION:
    - The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue.

LINKED QUEUE

- To insert an element with value 9, we first check if FRONT=NULL.
- If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part.
- The new node will then be called both FRONT and REAR.
- However, if FRONT!= NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear.
- Thus the updated queue becomes as:



LINKED AFTER INSERTING A NEW NODE

ALGORITHM TO INSERT AN ELEMENT IN A LINKED QUEUE:

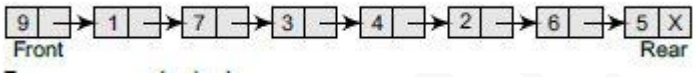
Step 1: Allocate memory for the new node and name
it as PTR

Step 2: SET PTR DATA = VAL

Step 3: IF FRONT = NULL
SET FRONT = REAR = PT
SET FRONT NEXT = REAR NEXT = NULL
ELSE
SET REAR NEXT = PTR
SET REAR = PTR
SET REAR NEXT = NULL
[END OF IF]

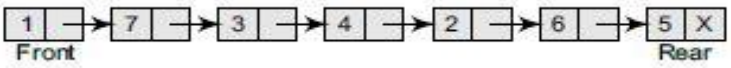
Step 4: END

2. DELETE OPERATION:
   - The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT.
   - However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done.
   - If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.



LINKED QUEUE

- To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue.
- Thus the updated queue becomes as:



LINKED QUEUE AFTER DELETION OF AN ELEMENT

ALGORITHM TO DELETE AN ELEMENT FROM AN LINKED QUEUE:

Step 1: IF FRONT = NULL
Write Underflow
Go to Step 5
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END

3. IMPLEMENTATION OF QUEUES

PROGRAMMING EXAMPLE 1:

1. Write a program to implement a linear queue.

```
##include <stdio.h>
#include <conio.h>
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
int option, val;
do
{
printf("\n\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
case 1:
insert();
break;
```

```
case 2:
val = delete_element();
if (val != -1)
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if (val != -1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
}while(option != 5);
getch();
return 0;
}
void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(rear == MAX-1)
printf("\n OVERFLOW");
else if(front == -1 && rear == -1)
front = rear = 0;
else
rear++;
queue[rear] = num;
}
```

```
int delete_element()
{
int val;
if(front == -1 || front>rear)
{
printf("\n UNDERFLOW");
return -1;
}
else
{
val = queue[front];
front++;
if(front > rear)
front = rear = -1;
return val;
}
}
int peek()
{
if(front==-1 || front>rear)
{
printf("\n QUEUE IS EMPTY");
return -1;
}
else
{
return queue[front];
}
}
void display()
{
```

```
int i;
printf("\n");
if(front == -1 || front > rear)
printf("\n QUEUE IS EMPTY");
else
{
for(i = front;i <= rear;i++)
printf("\t %d", queue[i]);
}
}
```

Output

```
***** MAIN MENU *****"
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 50
```

PROGRAMMING EXAMPLE 2:

2. Write a program to implement a linked queue.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
int data;
struct node *next;
};
struct queue
{
```

```
struct node *front;
struct node *rear;
};
struct queue *q;
void create_queue(struct queue *);
struct queue *insert(struct queue *,int);
struct queue *delete_element(struct queue *);
struct queue *display(struct queue *);
int peek(struct queue *);
int main()
{
int val, option;
create_queue(q);
clrscr();
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. INSERT");
printf("\n 2. DELETE");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to insert in the queue:");
scanf("%d", &val);
q = insert(q,val);
break;
```

```
case 2:
q = delete_element(q);
break;
case 3:
val = peek(q);
if(val! = –1)
printf("\n The value at front of queue is : %d", val);
break;
case 4:
q = display(q);
break;
}
}while(option != 5);
getch();
return 0;
}
void create_queue(struct queue *q)
{
q -> rear = NULL;
q -> front = NULL;
}
struct queue *insert(struct queue *q,int val)
{
struct node *ptr;
ptr = (struct node*)malloc(sizeof(struct node));
ptr -> data = val;
if(q -> front == NULL)
{
q -> front = ptr;
q -> rear = ptr;
q -> front -> next = q -> rear -> next = NULL;
```

```
}
else
{
q -> rear -> next = ptr;
q -> rear = ptr;
q -> rear -> next = NULL;
}
return q;
}
struct queue *display(struct queue *q)
{
struct node *ptr;
ptr = q -> front;
if(ptr == NULL)
printf("\n QUEUE IS EMPTY");
else
{
printf("\n");
while(ptr!=q -> rear)
{
printf("%d\t", ptr -> data);
ptr = ptr -> next;
}
printf("%d\t", ptr -> data);
}
return q;
}
struct queue *delete_element(struct queue *q){
struct node *ptr;
ptr = q -> front;
if(q -> front == NULL)
```

```
printf("\n UNDERFLOW");
else
{
q -> front = q -> front -> next;
printf("\n The value being deleted is : %d", ptr -> data);
free(ptr);
}
return q;
}
int peek(struct queue *q)
{
if(q->front==NULL)
{
printf("\n QUEUE IS EMPTY");
return –1;
}
else
return q->front->data;
```

Output

```
*****MAIN MENU*****
1. INSERT
2. DELETE
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

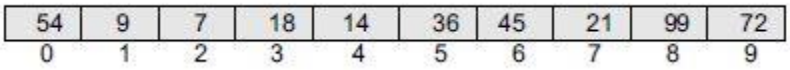
4.APPLICATIONS OF QUEUES

A queue data structure can be classified into the following types:

1. Circular Queue 2. Deque 3. Priority Queue 4. Multiple Queue

1. CIRCULAR QUEUE:

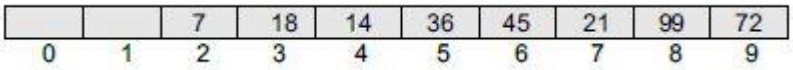
In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.



LINEAR QUEUE

Here, FRONT = 0 and REAR = 9.

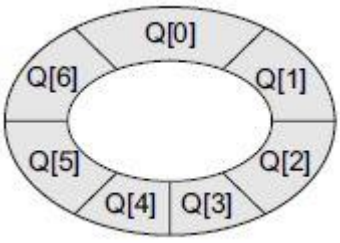
Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made.



Queue After Two Successive Deletions

Here, front = 2 and REAR = 9.

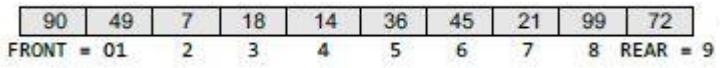
- Suppose we want to insert a new element in the queue shown in Fig. 8.14. Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue.
- To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.
- The second option is to use a circular queue. In the circular queue, the first index comes right after the last index.



CIRCULAR QUEUE

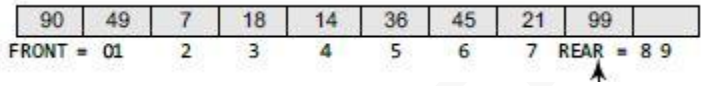
- The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations.
- For insertion, we now have to check for the following three conditions:

1. If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given .



FULL QUEUE

2. If rear != MAX – 1, then rear will be incremented and the value will be inserted



QUEUE WITH VACCANT LOCATIONS

3. If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there,

Inserting An Element In A Circular Queue

ALGORITHM TO INSERT AN ELEMENT IN CIRCULAR QUEUE

Step 1: IF FRONT = and Rear = MAX - 1

Write OVERFLOW

Goto step 4

[End OF IF]

STEP 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR =0

ELSE IF REAR = MAX - 1 and FRONT != 0

SET REAR =0

ELSE

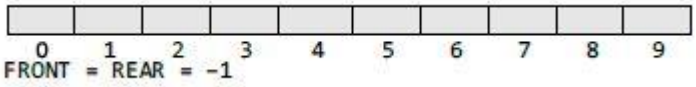
SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE [REAR] = VAL

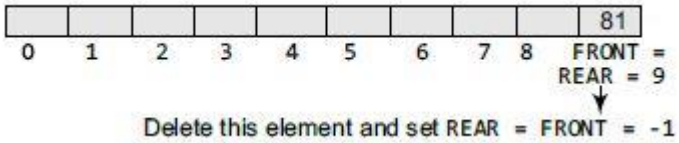
Step 4: EXIT

- After seeing how a new element is added in a circular queue, let us now discuss how deletions are performed in this case.
- To delete an element, again we check for three conditions.
  1. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.



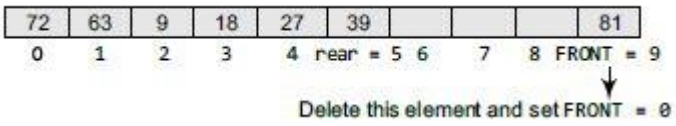
EMPTY QUEUE

  2. If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1.



QUEUE WITH A SINGLE ELEMENT

  3. If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0.



Queue Where Front=Max-1 Before Deletion

ALGORITHM TO DELETE AN ELEMENT IN CIRCULAR QUEUE

Step 1: IF FRONT = -1

Write UNDERFLOW

Goto Step 4

[END of IF]

Step 2: SET VAL = QUEUE [FRONT]

Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

IF FRONT = MAX -1

SET FRONT =0

```
        ELSE
            SET FRONT = FRONT + 1
        [END of IF]
        [END OF IF]
```

Step 4: EXIT

PROGRAMMING EXAMPLE:

3. Write a program to implement a circular queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front=–1, rear=–1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
int option, val;
clrscr();
do
{
printf("\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
```

```
{
case 1:
insert();
break;
case 2:
val = delete_element();
if(val!=–1)
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if(val!=–1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
}while(option!=5);
getch();
return 0;
}
void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(front==0 && rear==MAX–1)
printf("\n OVERFLOW");
else if(front==–1 && rear==–1)
{
```

```
front=rear=0;
queue[rear]=num;
}
else if(rear==MAX–1 && front!=0)
{
rear=0;
queue[rear]=num;
}
else
{
rear++;
queue[rear]=num;
}
}
int delete_element()
{
int val;
if(front==–1 && rear==–1)
{
printf("\n UNDERFLOW");
return –1;
}
val = queue[front];
if(front==rear)
front=rear=–1;
else
{
if(front==MAX–1)
front=0;
else
front++;
```

```
}
return val;
}
int peek()
{
if(front==–1 && rear==–1)
{
printf("\n QUEUE IS EMPTY");
return –1;
else
{
return queue[front];
}
}
void display()
{
int i;
printf("\n");
if (front ==–1 && rear= =–1)
printf ("\n QUEUE IS EMPTY");
else
{
if(front<rear)
{
for(i=front;i<=rear;i++)
printf("\t %d", queue[i]);
}
else
{
for(i=front;i<MAX;i++)
printf("\t %d", queue[i]);
```

```
for(i=0;i<=rear;i++)
printf("\t %d", queue[i]);
}
}
}
```

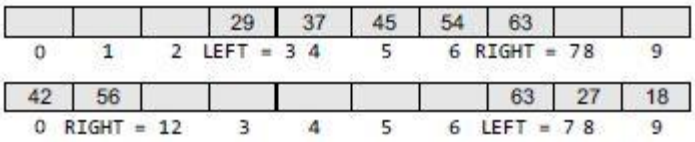
Output

```
***** MAIN MENU *****
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 25
Enter your option : 2
The number deleted is : 25
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

2.<u>DEQUES:</u>

A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end. It is also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].



DOUBLE END QUEUES

There are two variants of a double-ended queue. They include:

- **Input restricted deque:** In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- **Output restricted deque:** In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

PROGRAMMING EXAMPLE:

4. Write a program to implement input and output restricted deques.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int deque[MAX];
int left = –1, right = –1;
void input_deque(void);
void output_deque(void);
void insert_left(void);
void insert_right(void);
void delete_left(void);
void delete_right(void);
void display(void);
int main()
{
int option;
clrscr();
printf("\n *****MAIN MENU*****");
printf("\n 1.Input restricted deque");
printf("\n 2.Output restricted deque");
printf("Enter your option : ");
```

```
scanf("%d",&option);
switch(option)
{
case 1:
input_deque();
break;
case 2:
output_deque();
break;
}
return 0;
}
void input_deque()
{
int option;
do
{
printf("\n INPUT RESTRICTED DEQUE");
printf("\n 1.Insert at right");
printf("\n 2.Delete from left");
printf("\n 3.Delete from right");
printf("\n 4.Display");
printf("\n 5.Quit");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1:
insert_right();
break;
case 2:
```

```
delete_left();
break;
case 3:
delete_right();
break;
case 4:
display();
break;
}
}while(option!=5);
}
void output_deque()
{
int option;
do
{
printf("OUTPUT RESTRICTED DEQUE");
printf("\n 1.Insert at right");
printf("\n 2.Insert at left");
printf("\n 3.Delete from left");
printf("\n 4.Display");
printf("\n 5.Quit");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1:
insert_right();
break;
case 2:
insert_left();
```

```
break;
case 3:
delete_left();
break;
case 4:
display();
break;
}
}while(option!=5);
}
void insert_right()
{
int val;
printf("\n Enter the value to be added:");
scanf("%d", &val);
if((left == 0 && right == MAX–1) || (left == right+1))
{
printf("\n OVERFLOW");
return;
}
if (left == –1) /* if queue is initially empty */
{
left = 0;
right = 0;
}
else
{
if(right == MAX–1) /*right is at last position of queue */
right = 0;
else
right = right+1;
```

```
}
deque[right] = val ;
}
void insert_left()
{
int val;
printf("\n Enter the value to be added:");
scanf("%d", &val);
if((left == 0 && right == MAX–1) || (left == right+1))
{
printf("\n Overflow");
return;
}
if (left == –1)/*If queue is initially empty*/
{
left = 0;
right = 0;
}
else
{
if(left == 0)
left=MAX–1;
else
left=left–1;
}
deque[left] = val;
}
void delete_left()
{
if (left == –1)
{
```

```
printf("\n UNDERFLOW");
return ;
}
printf("\n The deleted element is : %d", deque[left]);
if(left == right) /*Queue has only one element */
{
left = –1;
right = –1;
}
else
{
if(left == MAX–1)
left = 0;
else
left = left+1;
}
}
void delete_right()
{
if (left == –1)
{
printf("\n UNDERFLOW");
return ;
}
printf("\n The element deleted is : %d", deque[right]);
if(left == right) /*queue has only one element*/
{
left = –1;
right = –1;
}
else
```

```
{
if(right == 0)
right=MAX–1;
else
right=right–1;
}
}
void display()
{
int front = left, rear = right;
if(front == –1)
{
printf("\n QUEUE IS EMPTY");
return;
}
printf("\n The elements of the queue are : ");
if(front <= rear )
{
while(front <= rear)
{
printf("%d",deque[front]);
front++;
}
}
else
{
while(front <= MAX–1)
{
printf("%d", deque[front]);
front++;
}
```

```
front = 0;
while(front <= rear)
{
printf("%d",deque[front]);
front++;
}
}
printf("\n");
}
```

Output

```
***** MAIN MENU *****
1.Input restricted deque
2.Output restricted deque
Enter your option : 1
INPUT RESTRICTED DEQUEUE
1.Insert at right
2.Delete from left
3.Delete from right
4.Display
5.Quit
Enter your option : 1
Enter the value to be added : 5
Enter the value to be added : 10
Enter your option : 2
The deleted element is : 5
Enter your option : 5
```

3.PRIORITY QUEUES:

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors.

Linked Representation of a Priority Queue

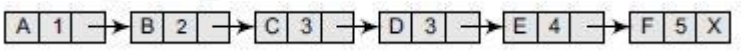
In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts:

(a) the information or data part,

(b) the priority number of the element, and

(c) the address of the next element.

If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.



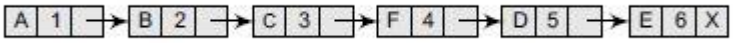
PRIORITY QUEUE

- Lower priority number means higher priority
- The priority queue in is a sorted priority queue having six elements.
- From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before because the list is not sorted based on FCFS.
- Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

Insertion: When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element.

PRIOIRTY QUEUE

If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element.



Priority Queue After Insertion A New Node

However, if we have a new element with data = F and priority number = 2, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS.

Priority Queue After Insertion A New Node

Deletion: Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

Array Representation of a Priority Queue

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed as.

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

PRIORITY QUEUE MATRIX

- FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number.Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

Insertion To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if w have to insert an element R with priority number 3,

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Priority Queue Matrix After Insertion Of A New Element

Deletion To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

PROGRAMMING EXAMPLE:

5. Write a program to implement a priority queue.

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
struct node
{
int data;
int priority;
```

```
struct node *next;
}
struct node *start=NULL;
struct node *insert(struct node *);
struct node *delete(struct node *);
void display(struct node *);
int main()
{
int option;
clrscr();
do
{
printf("\n *****MAIN MENU*****);
printf("\n 1. INSERT");
printf("\n 2. DELETE");
printf("\n 3. DISPLAY");
printf("\n 4. EXIT");
printf("\n Enter your option : ");
scanf( "%d", &option);
switch(option)
{
case 1:
start=insert(start);
break;
case 2:
start = delete(start);
break;
case 3:
display(start);
break;
}
```

```
}while(option!=4);
}
struct node *insert(struct node *start)
{
int val, pri;
struct node *ptr, *p;
ptr = (struct node *)malloc(sizeof(struct node));
printf("\n Enter the value and its priority : " );
scanf( "%d %d", &val, &pri);
ptr–>data = val;
ptr–>priority = pri;
if(start==NULL || pri < start–>priority )
{
ptr–>next = start;
start = ptr;
}
else
{
p = start;
while(p–>next != NULL && p–>next–>priority <= pri)
p = p–>next;
ptr–>next = p–>next;
p–>next = ptr;
}
return start;
}
struct node *delete(struct node *start)
{
struct node *ptr;
if(start == NULL)
{
```

```
printf("\n UNDERFLOW" );
return;
}
Else
{
ptr = start;
printf("\n Deleted item is: %d", ptr–>data);
start = start–>next;
free(ptr);
}
return start;
}
void display(struct node *start)
{
struct node *ptr;
ptr = start;
if(start == NULL)
printf("\nQUEUE IS EMPTY" );
else
{
printf("\n PRIORITY QUEUE IS : " );
while(ptr != NULL)
{
printf( "\t%d[priority=%d]", ptr–>data, ptr–>priority );
ptr=ptr–>next;
}
}
}
```

Output

*****MAIN MENU*****

1. INSERT

2. DELETE

3. DISPLAY

4. EXIT

Enter your option : 1

Enter the value and its priority : 5 2

Enter the value and its priority : 10 1

Enter your option : 3

PRIORITY QUEUE IS :

10[priority = 1] 5[priority = 2]

Enter your option : 4

4.MULTIPLE QUEUES:

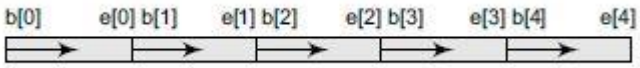
When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

- In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.

MULTIPLE QUEUES

In the figure, an array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing—queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i].



MULTIPLE QUEUES

PROGRAMMING EXAMPLE:

6. Write a program to implement multiple queues.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int QUEUE[MAX], rearA=–1,frontA=–1, rearB=MAX, frontB = MAX;
void insertA(int val)
{
if(rearA==rearB –1)
printf("\n OVERFLOW");
else
{
if(rearA ==–1 && frontA == –1)
{ rearA = frontA = 0;
QUEUE[rearA] = val;
}
else
QUEUE[++rearA] = val;
}
}
int deleteA()
{
int val;
if(frontA==–1)
{
```

```
printf("\n UNDERFLOW");
return –1;
}
else
{
val = QUEUE[frontA];
frontA++;
if (frontA>rearA)
frontA=rearA=–1
return val;
}
}
void display_queueA()
{
int i;
if(frontA==–1)
printf("\n QUEUE A IS EMPTY");
else
{
for(i=frontA;i<=rearA;i++)
printf("\t %d",QUEUE[i]);
}
}
void insertB(int val)
{
if(rearA==rearB–1)
printf("\n OVERFLOW");
else
{
if(rearB == MAX && frontB == MAX)
{ rearB = frontB = MAX–1;
```

```
QUEUE[rearB] = val;
}
else
QUEUE[—rearB] = val;
}
}
int deleteB()
{
int val;
if(frontB==MAX)
{
printf("\n UNDERFLOW");
return –1;
}
else
{
val = QUEUE[frontB];
frontB—;
if (frontB<rearB)
frontB=rearB=MAX;
return val;
}
}
void display_queueB()
{
int i;
if(frontB==MAX)
printf("\n QUEUE B IS EMPTY");
else
{
for(i=frontB;i>=rearB;i—)
```

```
printf("\t %d",QUEUE[i]);
}
}
int main()
{
int option, val;
clrscr();
do
{
printf("\n *******MENU******");
printf("\n 1. INSERT IN QUEUE A");
printf("\n 2. INSERT IN QUEUE B");
printf("\n 3. DELETE FROM QUEUE A");
printf("\n 4. DELETE FROM QUEUE B");
printf("\n 5. DISPLAY QUEUE A");
printf("\n 6. DISPLAY QUEUE B");
printf("\n 7. EXIT");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1: printf("\n Enter the value to be inserted in Queue A : ");
scanf("%d",&val);
insertA(val);
break;
case 2: printf("\n Enter the value to be inserted in Queue B : ");
scanf("%d",&val);
insertB(val);
break;
case 3: val=deleteA();
if(val!=–1)
```

```
printf("\n The value deleted from Queue A = %d",val);
break;
case 4 : val=deleteB();
if(val!=–1)
printf("\n The value deleted from Queue B = %d",val);
break;
case 5: printf("\n The contents of Queue A are : \n");
display_queueA();
break;
case 6: printf("\n The contents of Queue B are : \n");
display_queueB();
break;
}
}while(option!=7);
getch();
}
```

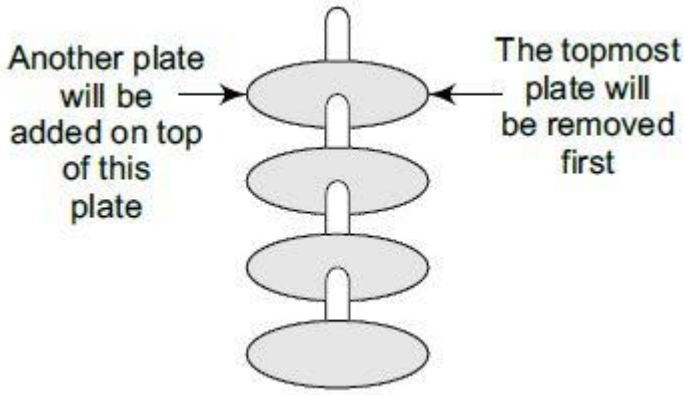
Output

```
*******MENU******"
1. INSERT IN QUEUE A
2. INSERT IN QUEUE B
3. DELETE FROM QUEUE A
4. DELETE FROM QUEUE B
5. DISPLAY QUEUE A
6. DISPLAY QUEUE B
7. EXIT
Enter your option : 2
Enter the value to be inserted in Queue B : 10
Enter the value to be inserted in Queue B : 5
Enter your option: 6
The contents of Queue B are : 10 5
Enter your option : 7
```

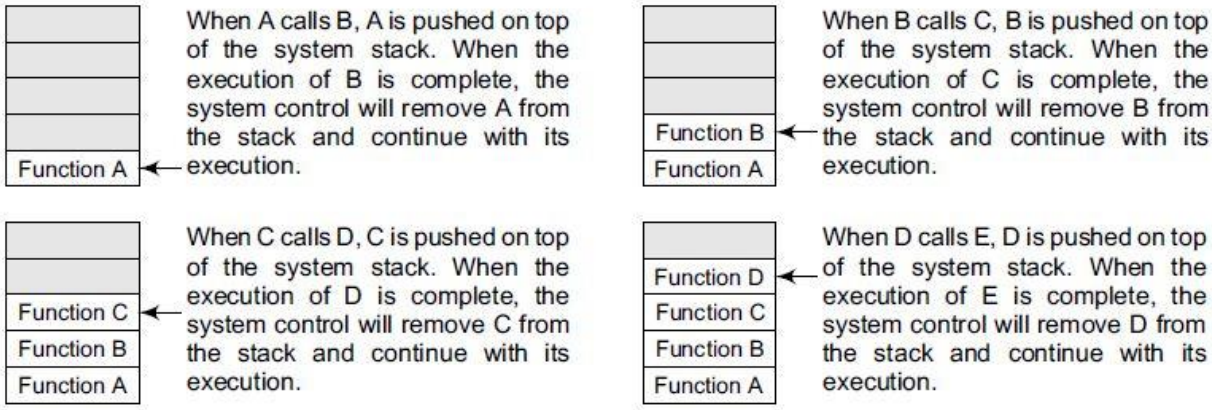
STACKS

1.INTRODUCTION TO STACKS

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.



STACK OF PLATES



SYSTEM STACK IN CASE OF FUNCTION CALL

2.ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.

(You must be wondering why we have written MAX–1. It is because array indices start from 0.)

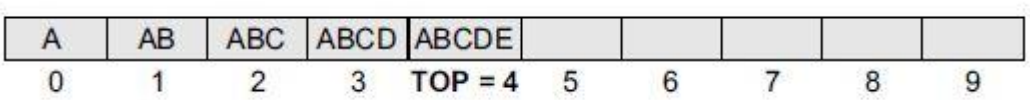


FIG. STACK

The stack in Fig. shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

OPERATIONS ON STACK:

A stack supports three basic operations: push, pop, and peek. The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack. The peek operation returns the value of the topmost element of the stack.

1. **PUSH OPERATION:**
   - The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
   - However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

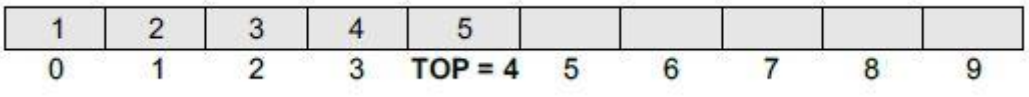


FIG. STACK

- To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as shown in Fig.

FIG. STACK AFTER INSERTION

ALGORITHM TO INSERT AN ELEMENT IN STACK:

Step 1: IF TOP = MAX-1

PRINT OVERFLOW

Goto Step 4

[END OF IF]

Step 2: SET TOP = TOP + 1

Step 3: SET STACK [TOP] = VALUE

Step 4: END

2. POP OPERATION:

- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

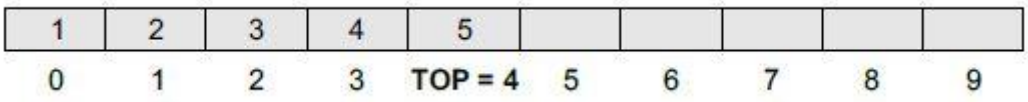


FIG. STACK

- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig.

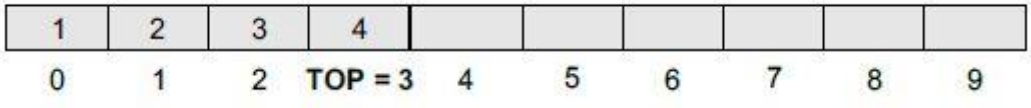


FIG. STACK AFTER FELETION

ALGORITHM TO DELETE AN ELEMENT FROM A STACK:

Step 1: IF TOP = NULL

PRINT UNDERFLOW

GOTO STEP 4

[END OF IF]

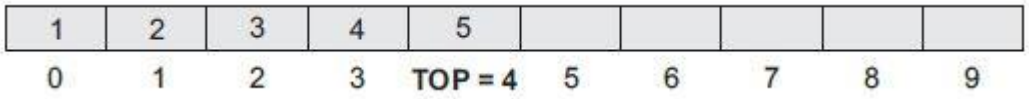
Step 2: SET VAL = STACK [TOP]

Step 3: SET TOP = TOP - 1

Step 4: END

3. PEEK OPERATION:

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.



STACK

- Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

ALGORITHM FOR PEEK OPERATION:

Step 1: IF TOP = NULL

PRINT STACK IS EMPTY

Goto Step 3

Step 2: RETURN STACK [TOP]

Step 3: END

PROGRAMMING EXAMPLE:

1. Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created
int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);
int main(int argc, char *argv[]) {
int val, option;
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option: ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to be pushed on stack: ");
scanf("%d", &val);
push(st, val);
break;
case 2:
```

```
val = pop(st);
if(val != -1)
printf("\n The value deleted from stack is: %d", val);
break;
case 3:
val = peek(st);
if(val != -1)
printf("\n The value stored at top of stack is: %d", val);
break;
case 4:
display(st);
break;
}
}while(option != 5);
return 0;
}
void push(int st[], int val)
{
if(top == MAX-1)
{
printf("\n STACK OVERFLOW");
}
else
{
top++;
st[top] = val;
}
}
int pop(int st[])
{
int val;
```

```
if(top == -1)
{
printf("\n STACK UNDERFLOW");
return -1;
}
else
{
val = st[top];
top--;
return val;
}
}
void display(int st[])
{
int i;
if(top == -1)
printf("\n STACK IS EMPTY");
else
{
for(i=top;i>=0;i--)
printf("\n %d",st[i]);
printf("\n"); // Added for formatting purposes
}
}
int peek(int st[])
{
if(top == -1)
{
printf("\n STACK IS EMPTY");
return -1;
}
```

```
else
return (st[top]);
}
```

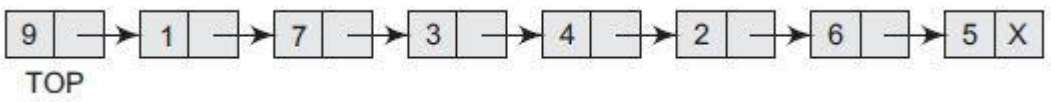
Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 500
```

3.LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.

- But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

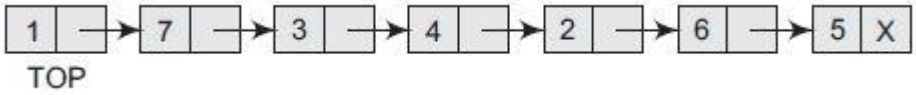


LINKED STACK

OPERATIONS ON A LINKED STACK:

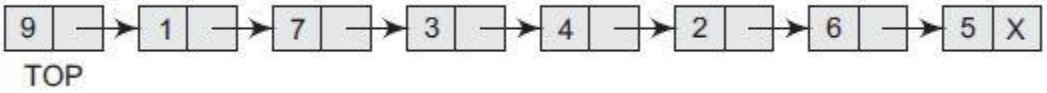
A linked stack supports all the three stack operations, that is, push, pop, and peek.

1. **PUSH OPERATION:**
   - The push operation is used to insert an element into the stack. The new element is added a topmost position of the stack.



LINKED STACK

   - To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



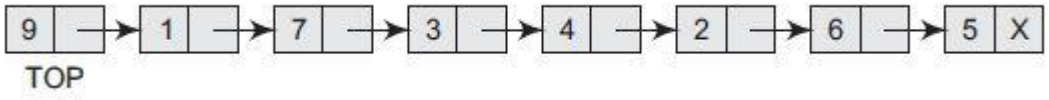
LINKED STACK AFTER INSERTING A NEW NODE

ALGORITHM TO INSERT AN ELEMENT IN A LINKED STACK:

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

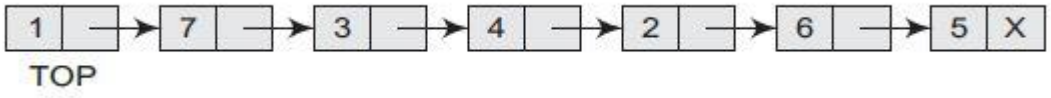
2. **POP OPERATION:**

- The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.



LINKED STACK

- In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack.



LINKED STACK AFTER DELETION OF THE TOPMOST ELEMENT

ALGORITHM TO DELETE AN ELEMENT FROM A LINKED LIST:

Step 1: IF TOP = NULL
PRINT UNDERFLOW
Goto Step 5
[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP -> NEXT

Step 4: FREE PTR

Step 5: END

PROGRAMMING EXAMPLE:

2. Write a program to implement a linked stack.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
#include <malloc.h>
struct stack
{
int data;
struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);
int main(int argc, char *argv[]) {
int val, option;
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option: ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to be pushed on stack: ");
scanf("%d", &val);
top = push(top, val);
break;
case 2:
```

```
top = pop(top);
break;
case 3:
val = peek(top);
if (val != -1)
printf("\n The value at the top of stack is: %d", val);
else
printf("\n STACK IS EMPTY");
break;
case 4:
top = display(top);
break;
}
}while(option != 5);
return 0;
}
struct stack *push(struct stack *top, int val)
{
struct stack *ptr;
ptr = (struct stack*)malloc(sizeof(struct stack));
ptr -> data = val;
if(top == NULL)
{
ptr -> next = NULL;
top = ptr;
}
else
{
ptr -> next = top;
top = ptr;
}
```

```
return top;
}
struct stack *display(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK IS EMPTY");
else
{
while(ptr != NULL)
{
printf("\n %d", ptr -> data);
ptr = ptr -> next;
}
}
return top;
}
struct stack *pop(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK UNDERFLOW");
else
{
top = top -> next;
printf("\n The value being deleted is: %d", ptr -> data);
free(ptr);
}
return top;
```

```
}
int peek(struct stack *top)
{
if(top==NULL)
return -1;
else
return top ->data;
}
```

Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. Peek
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 100
```

4.APPLICATIONS OF STACKS

1. REVERSING A LIST:

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

PROGRAMMING EXAMPLE:

3. Write a program to reverse a list of given numbers.

```
#include <stdio.h>
#include <conio.h>
int stk[10];
int top=–1;
int pop();
```

```
void push(int);
int main()
{
int val, n, i,
arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements of the array : ");
for(i=0;i<n;i++)
scanf("%d", &arr[i]);
for(i=0;i<n;i++)
push(arr[i]);
for(i=0;i<n;i++)
{
val = pop();
arr[i] = val;
}
printf("\n The reversed array is : ");
for(i=0;i<n;i++)
printf("\n %d", arr[i]);
getche"();
return 0;
}
void push(int val)
{
stk[++top] = val;
}
int pop()
{
return(stk[top—]);
```

}

Output

Enter the number of elements in the array : 5

Enter the elements of the array : 1 2 3 4 5

The reversed array is : 5 4 3 2 1

2.CONVERTING INFIX TO POSTFIX EXPRESSION:

- Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators. The precedence of these operators can be given as follows:
    a. Higher priority *, /, %
    b. Lower priority +, –
- No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

EXAMPLE: Convert the following infix expressions into prefix expressions.

Solution

(a) (A + B) * C

(+AB)*C

*+ABC

(b) (A–B) * (C+D)

[–AB] * [+CD]

*–AB+CD

(c) (A + B) / (C + D) – (D * E)

[+AB] / [+CD] – [*DE]

[/+AB+CD] – [*DE]

–/+AB+CD*DE

ALGORITHM TO CONVERT INFIX EXPRESSION TO POSTFIX EXPRESSION

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

EXAMPLE: Convert the following infix expression into postfix expression using the algorithm given.

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

SOLUTION:

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

PROGRAMMING EXAMPLE:

4.Write a program to convert an infix expression into its equivalent postfix notation.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=–1;
void push(char st[], char);
char pop(char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
int main()
{
char infix[100], postfix[100];
clrscr();
printf("\n Enter any infix expression : ");
gets(infix);
strcpy(postfix, "");
InfixtoPostfix(infix, postfix);
printf("\n The corresponding postfix expression is : ");
puts(postfix);
getch();
return 0;
}
void InfixtoPostfix(char source[], char target[])
{
int i=0, j=0;
char temp;
strcpy(target, "");
```

```
while(source[i]!='\0')
{
if(source[i]=='(')
{
push(st, source[i]);
i++;
}
else if(source[i] == ')')
{
while((top!=–1) && (st[top]!='('))
{
target[j] = pop(st);
j++;
}
if(top==–1)
{
printf("\n INCORRECT EXPRESSION");
exit(1);
}
temp = pop(st);//remove left parenthesis
i++;
}
else if(isdigit(source[i]) || isalpha(source[i]))
{
target[j] = source[i];
j++;
i++;
}
else if (source[i] == '+' || source[i] == '–' || source[i] == '*' ||
source[i] == '/' || source[i] == '%')
{
```

```
while( (top!=–1) && (st[top]!= '(') && (getPriority(st[top])
> getPriority(source[i])))
{
target[j] = pop(st);
j++;
}
push(st, source[i]);
i++;
}
Else
{
printf("\n INCORRECT ELEMENT IN EXPRESSION");
exit(1);
}
}
while((top!=–1) && (st[top]!='('))
{
target[j] = pop(st);
j++;
}
target[j]='\0';
}
int getPriority(char op)
{
if(op=='/' || op == '*' || op=='%')
return 1;
else if(op=='+' || op=='–')
return 0;
}
void push(char st[], char val)
{
```

```
if(top==MAX–1)
printf("\n STACK OVERFLOW");
else
{
top++;
st[top]=val;
}
}
char pop(char st[])
{
char val=' ';
if(top==–1)
printf("\n STACK UNDERFLOW");
else
{
val=st[top];
top—;
}
return val;
}
```

Output

Enter any infix expression : A+B–C*D

The corresponding postfix expression is : AB+CD*–

3. EVALUATION OF POSTFIX EXPRESSION:

- Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.
- Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack.

- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

ALGORITHM TO EVALUATE POSTFIX EXPRESSION:

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

EXAMPLE:

Let us now take an example that makes use of this algorithm. Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.

The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation. Look at Table which shows the procedure.

Evaluation of a postfix expression.

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

PROGRAMMING EXAMPLE:

5.Write a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=–1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
float val;
char exp[100];
clrscr();
printf("\n Enter any postfix expression : ");
```

```
gets(exp);
val = evaluatePostfixExp(exp);
printf("\n Value of the postfix expression = %.2f", val);
getch();
return 0;
}
float evaluatePostfixExp(char exp[])
{
int i=0;
float op1, op2, value;
while(exp[i] != '\0')
{
if(isdigit(exp[i]))
push(st, (float)(exp[i]–'0'));
else
{
op2 = pop(st);
op1 = pop(st);
switch(exp[i])
{
case '+':
value = op1 + op2;
break;
case '–':
value = op1 – op2;
break;
case '/':
value = op1 / op2;
break;
case '*':
value = op1 * op2;
```

```
break;
case '%':
value = (int)op1 % (int)op2;
break;
}
push(st, value);
}
i++;
}
return(pop(st));
}
void push(float st[], float val)
{
if(top==MAX–1)
printf("\n STACK OVERFLOW");
else
{
top++;
st[top]=val;
}
}
float pop(float st[])
{
float val=–1;
if(top==–1)
printf("\n STACK UNDERFLOW");
else
{
val=st[top];
top—;
}
```

return val;

}

Output

Enter any postfix expression : 9 3 4 * 8 + 4 / –

Value of the postfix expression = 4.00

4. FACTORIAL CALCULATION:

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are

Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.

Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number.

To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

In other words, n! = n X (n–1)!

Let us say we need to find the value of 5!

5! = 5 X 4 X 3 X 2 X 1

= 120

This can be written as

5! = 5 X 4!, where 4!= 4 X 3!

Therefore,

5! = 5 X 4 X 3!

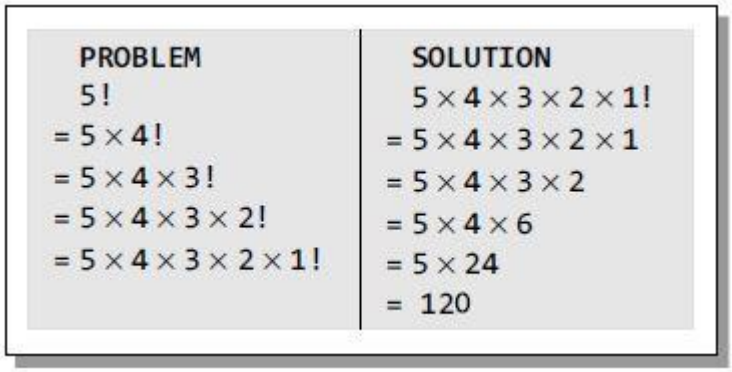
Similarly, we can also write,

5! = 5 X 4 X 3 X 2!

Expanding further

5! = 5 X 4 X 3 X 2 X 1!

We know, 1! = 1



RECURSIVE FACTORIAL FUNCTION

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. every recursive function must have a base case and a recursive case. For the factorial function,

Base case is when n = 1, because if n = 1, the result will be 1 as 1! = 1.

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)

PROGRAMMING EXAMPLE:

6. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int); // FUNCTION DECLARATION
int main()
{
int num, val;
printf("\n Enter the number: ");
scanf("%d", &num);
```

```
val = Fact(num);
printf("\n Factorial of %d = %d", num, val);
return 0;
}
int Fact(int n)
{
if(n==1)
return 1;
else
return (n * Fact(n–1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```

UNIT-IV

1. TREES

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. Figure 1.1 shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

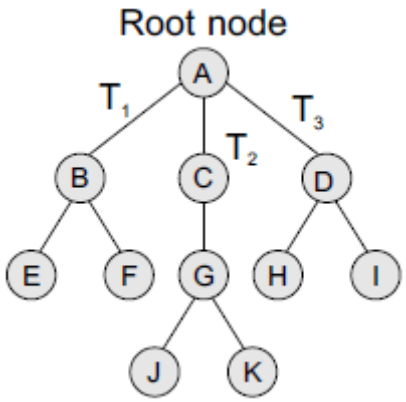


Figure 1.1 Tree

1.1 BASIC TERMINOLOGY IN TREES:

Root node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Sub-trees: If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.

Leaf node: A node that has no children is called the leaf node or the terminal node.

Path: A sequence of consecutive edges is called a *path.* For example, in Fig. 1.1, the path from the root node A to node I is given as: A, D, and I.

Ancestor node: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 1.1, nodes A, C, and G are the ancestors of node K.

Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 1.1, nodes C, G, J, and K are the descendants of node A.

Level number: Every node in the tree is assigned a *level number* in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

Degree: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

In-degree: In-degree of a node is the number of edges arriving at that node.

Out-degree: Out-degree of a node is the number of edges leaving that node.

1.2 BINARY TREES:

Definition:

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

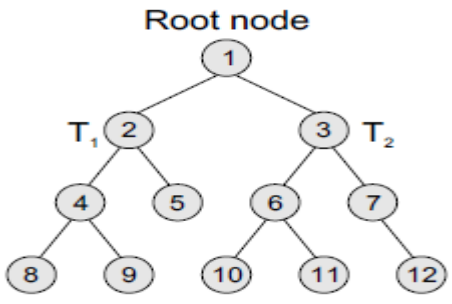


Figure 1.2 Binary tree

Figure 1.2 shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

Note that the left sub-tree of the root node consists of the nodes: 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of nodes: 3, 6, 7, 10, 11, and 12.

In the tree, root node 1 has two successors: 2 and 3. Node 2 has two successor nodes: 4 and 5. Node 4 has two successors: 8 and 9. Node 5 has no successor. Node 3 has two successor nodes: 6 and 7. Node 6 has two successors: 10 and 11. Finally, node 7 has only one successor: 12.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree. Look at Fig. 1.2, nodes 5, 8, 9, 10, 11, and 12 have no successors and thus said to have empty sub-trees.

Complete Binary Trees:

A *complete binary tree* is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.

In a complete binary tree T_n, there are exactly n nodes and level r of T can have at most 2^r nodes. Figure 1.3 shows a complete binary tree.

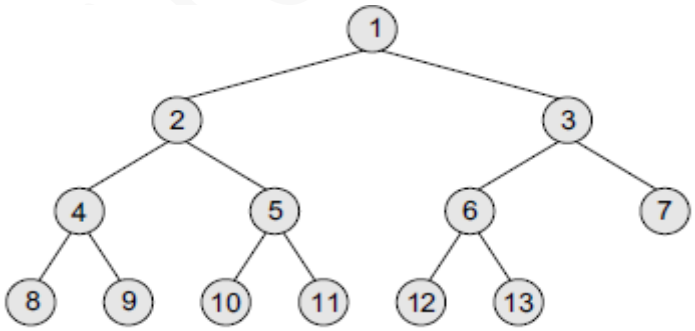


Figure 1.3 Complete binary tree

Note that in Fig. 1.3, level 0 has $2^0 = 1$ node, level 1 has $2^1 = 2$ nodes, level 2 has $2^2 = 4$ nodes, level 3 has 6 nodes which is less than the maximum of $2^3 = 8$ nodes.

In Fig. 1.3, tree T_{13} has exactly 13 nodes. They have been purposely labelled from 1 to 13, so that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node. The formula can be given as—if K is a parent node, then its left

child can be calculated as 2 × K and its right child can be calculated as 2 × K + 1. For example, the children of the node 4 are 8 (2 × 4) and 9 (2 × 4 + 1). Similarly, the parent of the node K can be calculated as | K/2 |. Given the node 4, its parent can be calculated as | 4/2 | = 2. The height of a tree T_n having exactly n nodes is given as: $H_n = | \log_2 (n + 1) |$. This means, if a tree T has 10,00,000 nodes, then its height is 21.

Extended Binary Trees:

A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children. Figure 1.4 shows how an ordinary binary tree is converted into an extended binary tree.

In an extended binary tree, nodes having two children are called *internal nodes* and nodes having no children are called *external nodes*. In Fig. 1.4, the internal nodes are represented using circles and the external nodes are represented using squares.

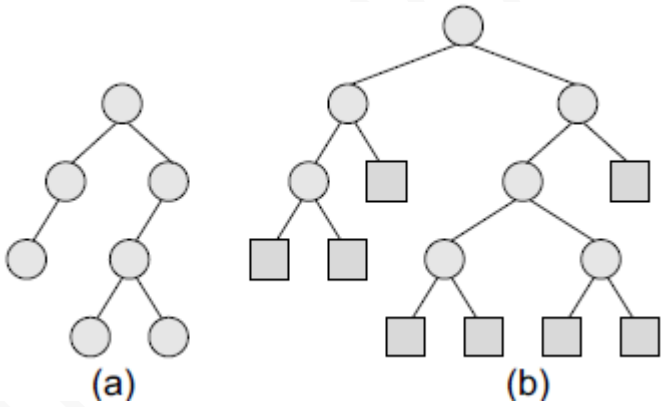


Figure 1.4 (a) Binary tree and (b) extended binary tree

To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

1.2.1 Properties of Binary Trees:

- A tree with 'n' nodes has exactly (n-1) edges or branches.
- The height or depth of a binary tree is the number of levels in it.
- In a tree every node except the root has exactly one parent (and the root node does not have a parent node).
- The maximum number of nodes in a binary tree of height 'h' is (2^{h+1})-1 where h>=0.

1.2.2 Representation of Binary Trees:

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential (Array) representation.

1) Sequential Representation of Binary Trees:

Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. A sequential binary tree follows the following rules:

- A one-dimensional array, called TREE, is used to store the elements of tree.
- The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
- The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

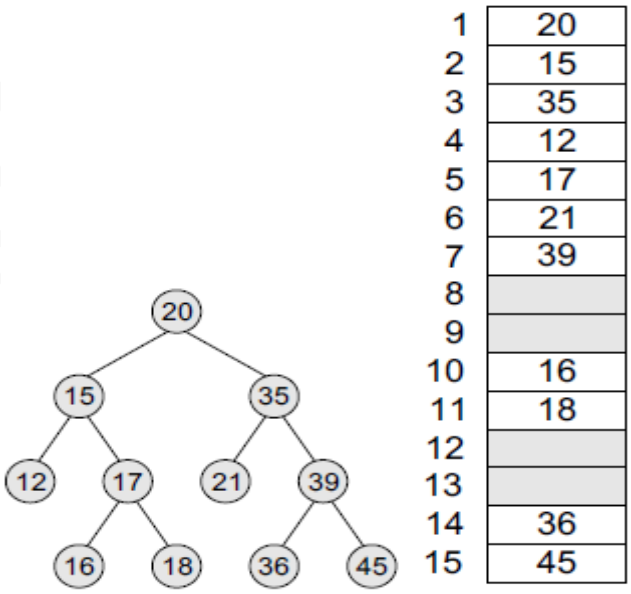


Figure 1.5 Binary tree and its sequential representation

Figure 1.5 shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.

2) Linked Representation of Binary Trees:

In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.

So in C, the binary tree is built with a node type given below.

```
struct node {
        struct node *left;
        int data;
        struct node *right;
};
```

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty. Consider the binary tree given in Fig. 1.6. The schematic diagram of the linked representation of the binary tree is shown in Fig. 1.7.

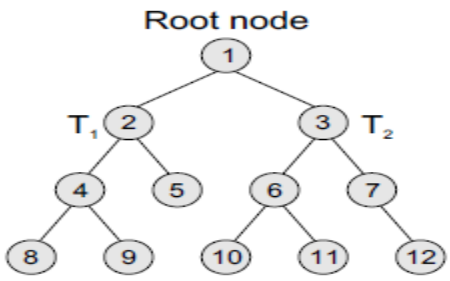


Figure 1.6 Binary tree

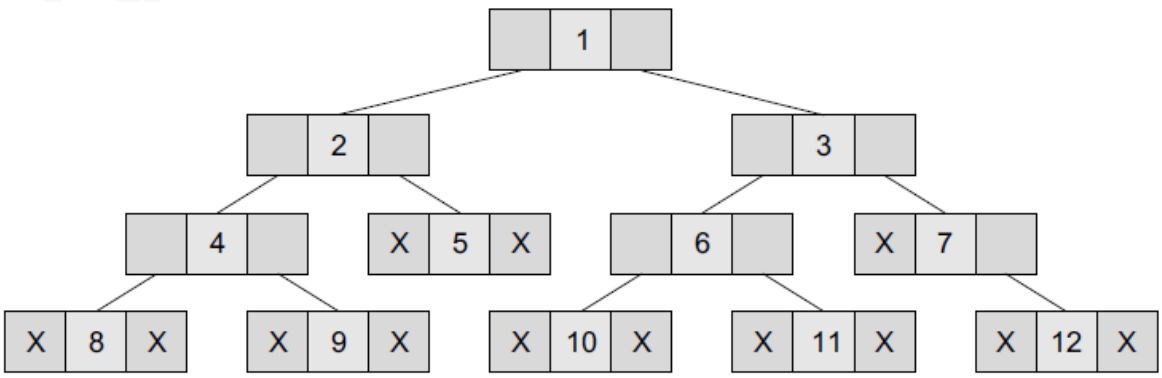


Figure 1.7 Linked representation of a binary tree

In Fig. 1.7, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

1.3 BINARY SEARCH TREES:

1.3.1 Basic Concepts:

A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.

In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

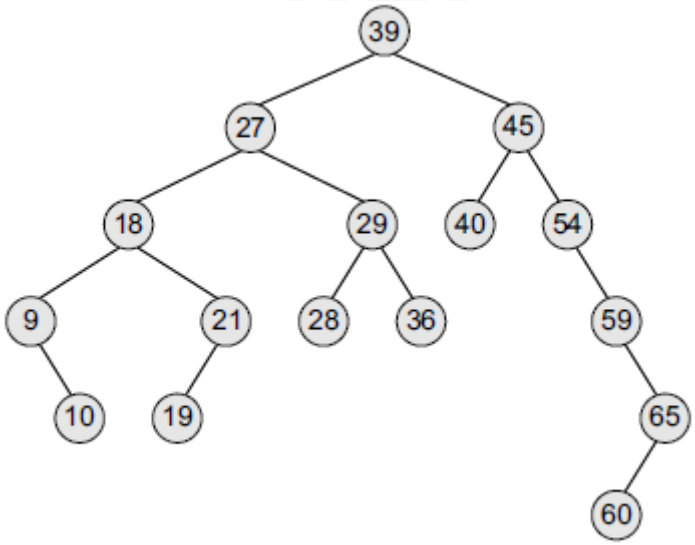


Figure 1.8 Binary search tree

Look at Fig. 1.8. The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10,

18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in. For example, in the given tree, if we have to search for 29, then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). The left sub-tree has a root node with the value 27. Since 29 is greater than 27, we will move to the right sub-tree, where we will find the element. Thus, the average running time of a search operation is O(log2n), as at every step, we eliminate half of the sub-tree from the search process. Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly.

To summarize, a binary search tree is a binary tree with the following properties:

- The left sub-tree of a node N contains values that are less than N's value.
- The right sub-tree of a node N contains values that are greater than N's value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

1.3.2 BST Operations:

1) Inserting a New Node in a Binary Search Tree:

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. Figure 1.9 shows the algorithm to insert a given value in a binary search tree.

The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Figure 1.9 Algorithm to insert a given value in a binary search tree

In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node, else it looks at the current node's value and then recurs down the left or right sub-tree. If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node. The new node is added by following the rules of the binary search trees. That is, if the new node's value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree. The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

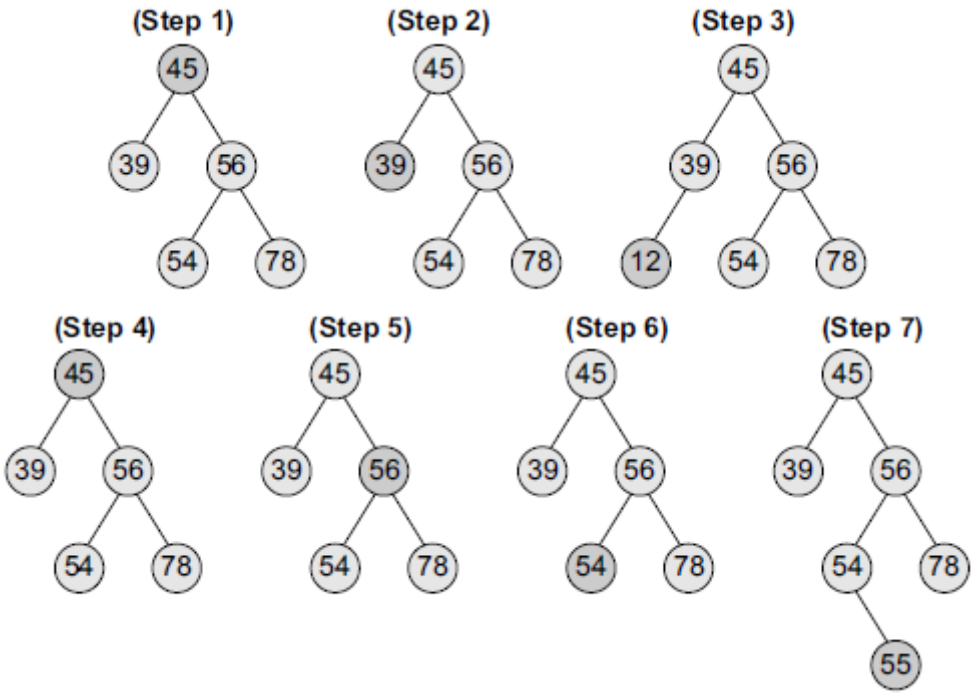


Figure 1.10 Inserting nodes with values 12 and 55 in the given binary search tree

Look at Fig. 1.10 which shows insertion of values in a given tree. We will take up the case of inserting 12 and 55.

- In the step1, the value 12 is compared with root node 45, it is less than 45. So, it goes to left sub tree of 45.
- In the step2, again 12 is compared with 39, and 12 is less than 39. So, again it goes to left sub tree of 39.
- In the step3, it checks for the values but there are no elements at the left sub tree of 39. So, the new value 12 is inserted at left side of the node 39.
- Similarly, this process repeats for the element 55.

2) Deleting a Node from a Binary Search Tree:

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. We will take up three cases in this section and discuss how a node is deleted from a binary search tree.

Case 1: Deleting a Node that has No Children:

Look at the binary search tree given in Fig. 1.11. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.

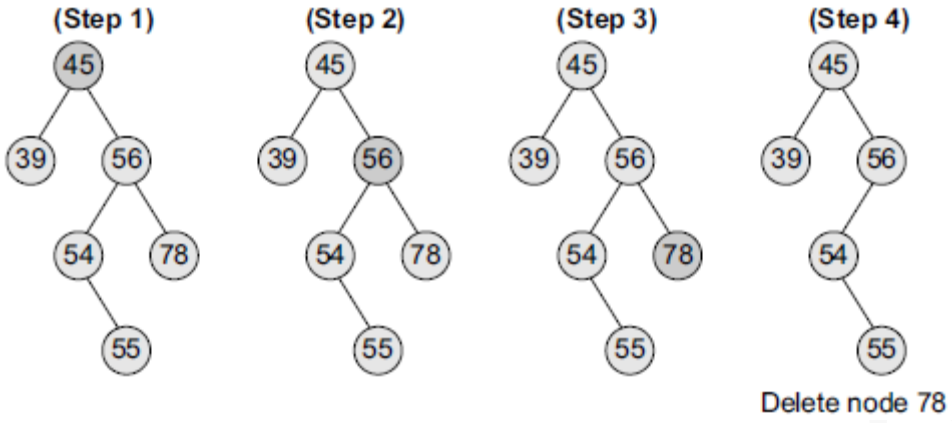


Figure 1.11 Deleting node 78 from the given binary search tree

Case 2: Deleting a Node with One Child:

To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent. Look at the binary search tree shown in Fig. 1.12 and see how deletion of node 54 is handled.

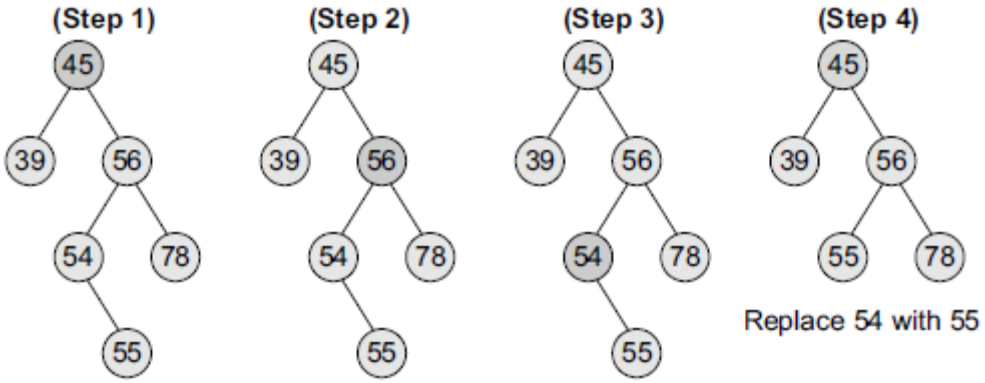


Figure 1.12 Deleting node 54 from the given binary search tree

Case 3: Deleting a Node with Two Children:

To handle this case, replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Look at the binary search tree given in Fig. 1.13 and see how deletion of node with value 56 is handled.

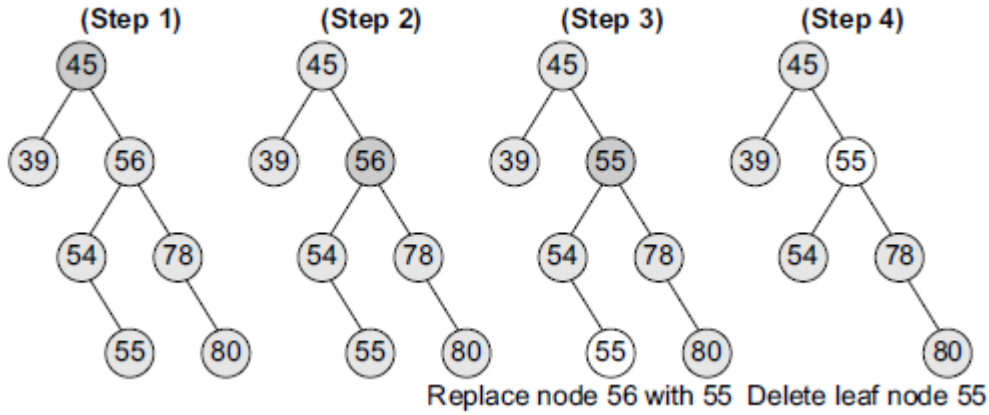


Figure 1.13 Deleting node 56 from the given binary search tree

This deletion could also be handled by replacing node 56 with its in-order successor, as shown in Fig. 1.14.

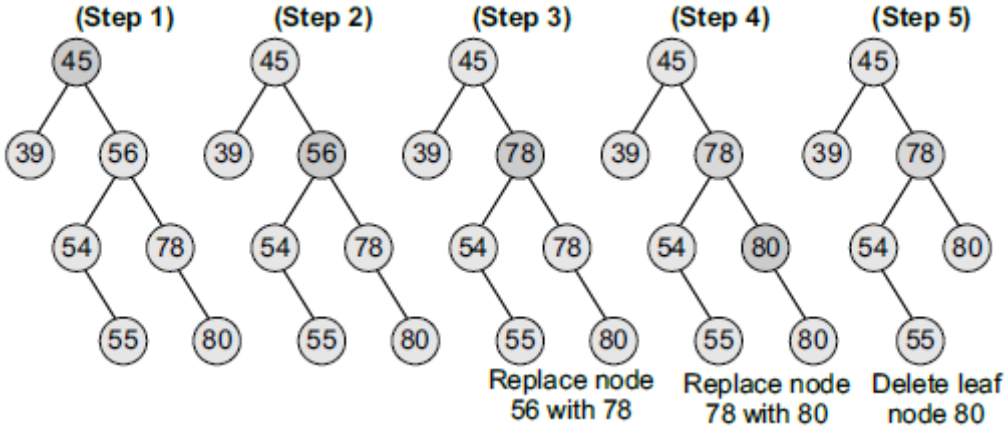


Figure 1.14 Deleting node 56 from the given binary search tree

Now, let us look at Fig. 1.15 which shows the algorithm to delete a node from a binary search tree. In Step 1 of the algorithm, we first check if TREE=NULL, because if it is true, then the node to be deleted is not present in the tree. However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm is called recursively on the node's right sub-tree.

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 1.15 Algorithm to delete a node from a binary search tree

Note that if we have found the node whose value is equal to VAL, then we check which case of deletion it is. If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling findLargestNode(TREE -> LEFT) and replace the current node's value with that of its in-order predecessor. Then, we call Delete(TREE -> LEFT, TEMP -> DATA) to delete the initial node of the in-order predecessor. Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

If the node to be deleted does not have any child, then we simply set the node to NULL. Last but not the least, if the node to be deleted has either a left or a right child but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.

The delete function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and W(n) time in the worst case.

3) Tree Traversals:

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a nonlinear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited. In this section, we will discuss these algorithms.

a) Pre-order Traversal:

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

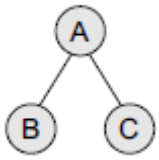


Figure 1.16 Binary tree

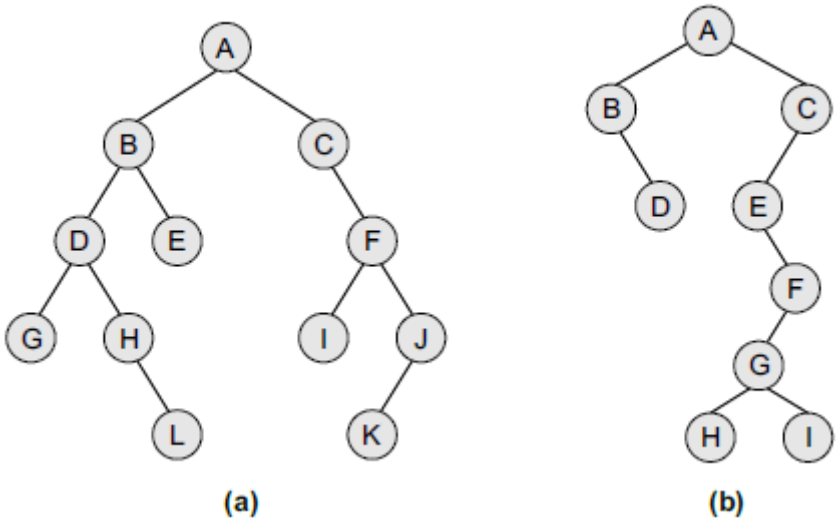
Consider the tree given in Fig. 1.16. The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as *depth-first traversal*. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right). The algorithm for pre-order traversal is shown in Fig. 1.17.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           Write TREE -> DATA
Step 3:           PREORDER(TREE -> LEFT)
Step 4:           PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

Figure 1.17 Algorithm for pre-order traversal

Example: In Figs (a) and (b), find the sequence of nodes that will be visited using pre-order traversal algorithm.

Solution:



TRAVERSAL ORDER for a: A, B, D, G, H, L, E, C, F, I, J, and K

TRAVERSAL ORDER for b: A, B, D, C, D, E, F, G, H, and I

b) In-order Traversal:

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

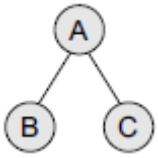


Figure 1.18 Binary tree

Consider the tree given in Fig. 1.18. The in-order traversal of the tree is given as B, A, and C. Left sub-tree first, the root node next, and then the right sub-tree. In-order traversal is also called as *symmetric traversal*. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree. The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right). The algorithm for in-order traversal is shown in Fig. 1.19.

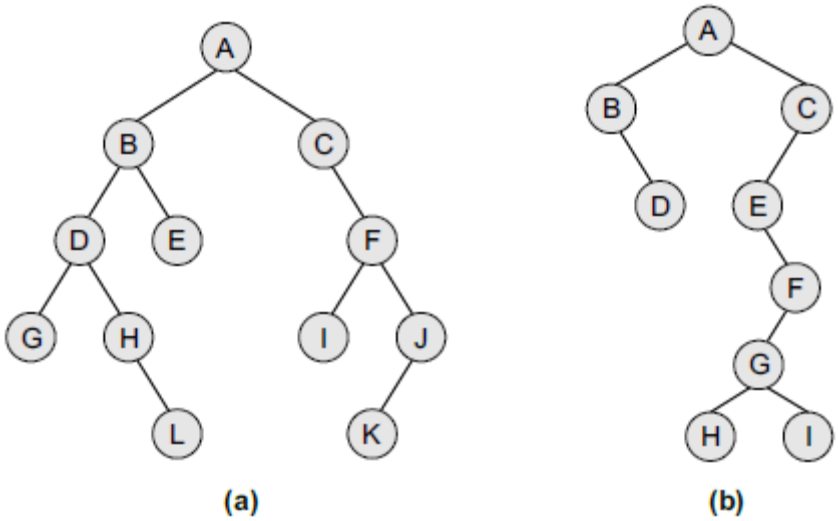
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

Figure 1.19 Algorithm for in-order traversal

In-order traversal algorithm is usually used to display the elements of a binary search tree. Here, all the elements with a value lower than a given value are accessed before the elements with a higher value.

Example: In Figs (a) and (b), find the sequence of nodes that will be visited using In-order traversal algorithm.

Solution:



TRAVERSAL ORDER for a: G, D, H, L, B, E, A, C, I, F, K, and J

TRAVERSAL ORDER for b: B, D, A, E, H, G, I, F, and C

c) Post-order Traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

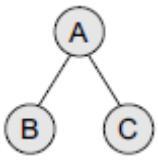


Figure 1.20 Binary tree

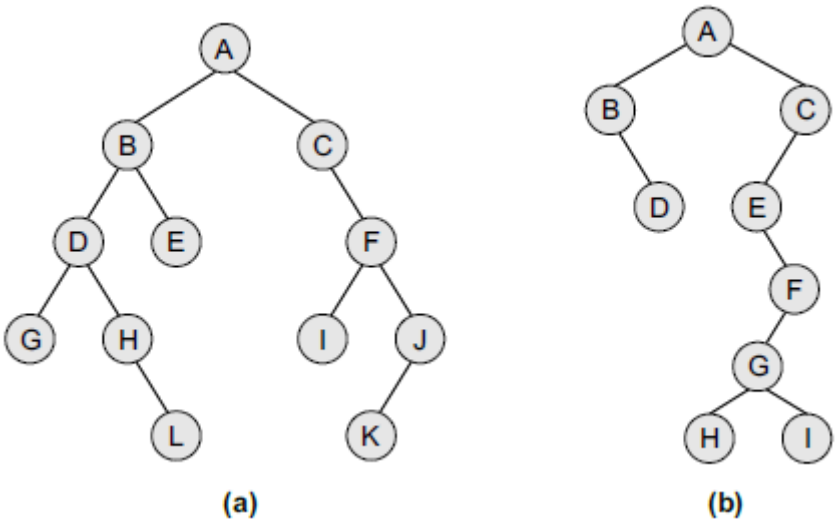
Consider the tree given in Fig. 1.20. The post-order traversal of the tree is given as B, C, and A. Left sub-tree first, the right sub-tree next, and finally the root node. In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word 'post' in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post-order algorithm is also known as the LRN traversal algorithm (Left Right-Node). The algorithm for post-order traversal is shown in Fig. 1.21. Post-order traversals are used to extract postfix notation from an expression tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          POSTORDER(TREE -> LEFT)
Step 3:          POSTORDER(TREE -> RIGHT)
Step 4:          Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

Figure 1.21 Algorithm for post-order traversal

Example: In Figs (a) and (b), find the sequence of nodes that will be visited using Post-order traversal algorithm.

Solution:



TRAVERSAL ORDER for a: G, L, H, D, E, B, I, K, J, F, C, and A

TRAVERSAL ORDER for b: D, B, H, I, G, F, E, C, and A

d) Level-order Traversal

In level-order traversal, all the nodes at a level are accessed before going to the next level. This algorithm is also called as the *breadth-first traversal algorithm*. Consider the trees given in Fig. 1.22 and note the level order of these trees.

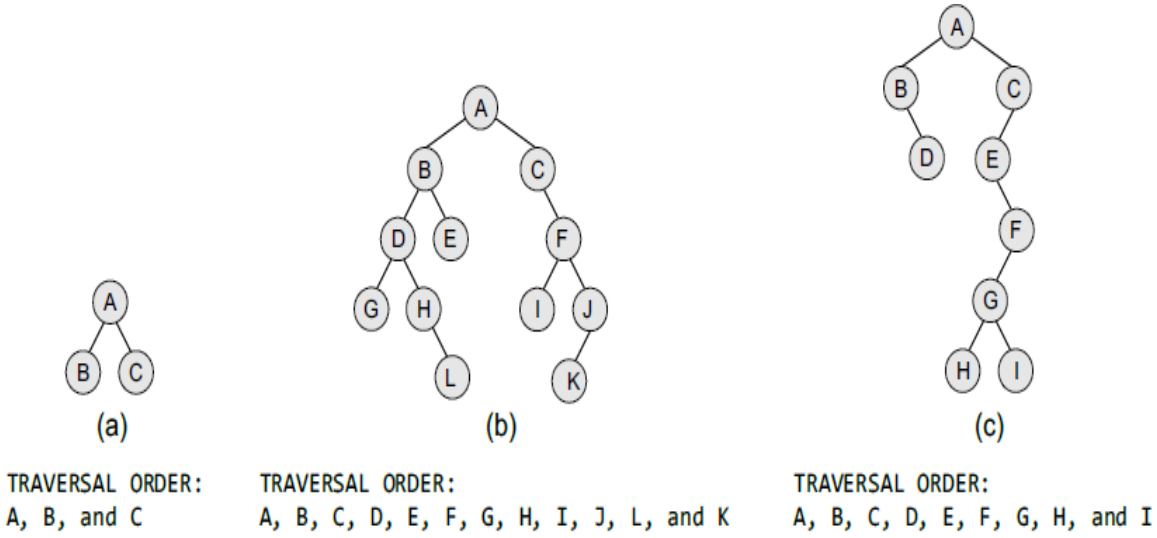


Figure 1.22 Binary trees

1.4 APPLICATIONS:

1.4.1 Expression Trees:

A binary expression tree is a specific kind of a binary tree used to represent expressions. Two common types of expressions that a binary expression tree can represent are algebraic and boolean. These trees can represent expressions that contain both unary and binary operators.

Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression given as: Exp = (a – b) + (c * d)

This expression can be represented using a binary tree as shown in Fig. 1.23

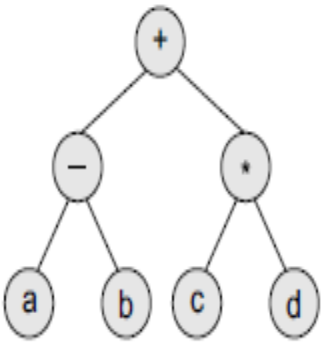
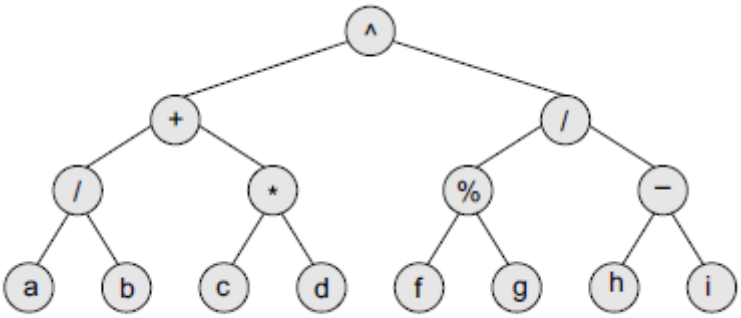


Figure 1.23 Expression tree.

Example: Given the binary tree, write down the expression that it represents.

Solution:



Expression for the above binary tree is [{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

1.4.2 Heap Sort (Binary Heap):

A binary heap is a complete binary tree in which every node satisfies the heap property which states that: If B is a child of A, then key(A) ≥ key(B)

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a *max-heap*.

Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a *min-heap*.

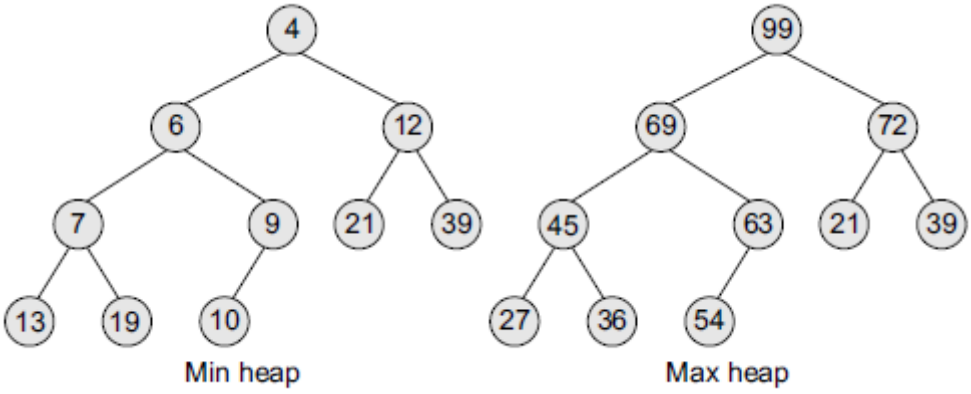


Figure 1.24 Binary heaps

Figure 1.24 shows a binary min heap and a binary max heap. The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position

2i and its right child at position 2i+1. Conversely, an element at position i has its parent stored at position i/2.

- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as log2n, where n is the number of elements.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap. A binary tree is an efficient data structure, but a binary heap is more space efficient and simpler.

1) Inserting a New Element in a Binary Heap:

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

Example: Consider the max heap given in Fig. 1.25 and insert 99 in it.

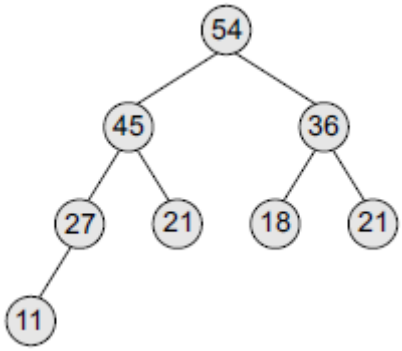


Figure 1.25 Binary heap

Solution:

The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig. 1.26.

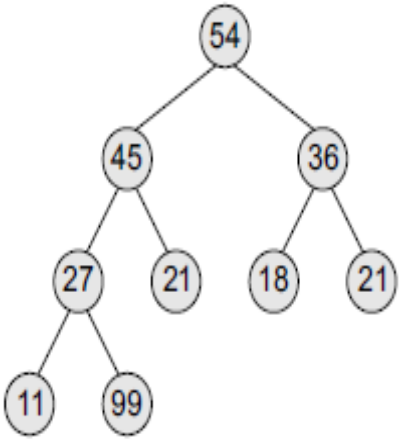


Figure 1.26 Binary heap after insertion of 99

Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well. Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Fig. 1.27.

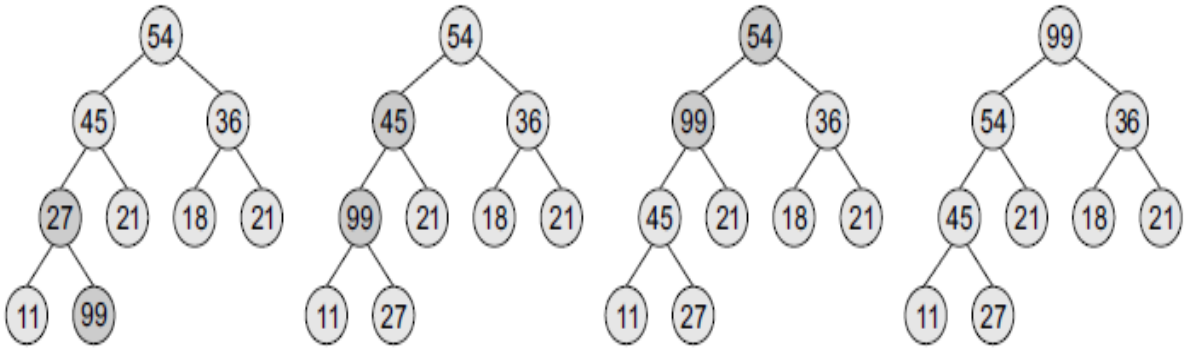


Figure 1.27 Heapify the binary heap

After discussing the concept behind inserting a new value in the heap, let us now look at the algorithm to do so as shown in Fig. 1.28.

```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Figure 1.28 Algorithm to insert an element in a max heap

We assume that H with n elements is stored in array HEAP. VAL has to be inserted in HEAP. The location of VAL as it rises in the heap is given by POS, and PAR denotes the location of the parent of VAL.

Note that this algorithm inserts a single value in the heap. In order to build a heap, use this algorithm in a loop. For example, to build a heap with 9 elements, use a for loop that executes 9 times and in each pass, a single value is inserted.

The complexity of this algorithm in the average case is O(1). This is because a binary heap has O(log n) height. Since approximately 50% of the elements are leaves and 75% are in the bottom two levels, the new element to be inserted will only move a few levels upwards to maintain the heap.

In the worst case, insertion of a single value may take O(log n) time and, similarly, to build a heap of n elements, the algorithm will execute in O(n log n) time.

2) Deleting an Element from a Binary Heap:

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.

Example: Consider the max heap H shown in Fig. 1.29 and delete the root node's value.

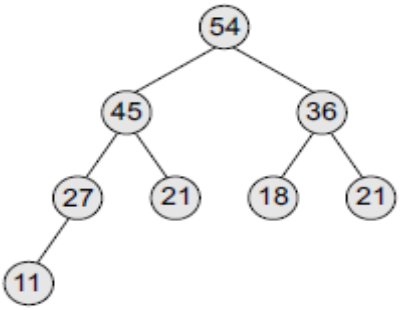


Figure 1.29 Binary heap

Solution:

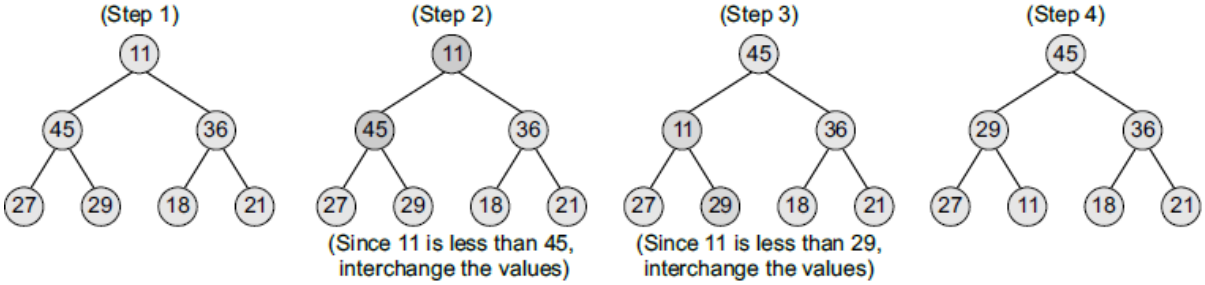


Figure 1.30 Binary heap

After discussing the concept behind deleting the root element from the heap, let us look at the algorithm given in Fig. 1.31.

```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
         HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Figure 1.31 Algorithm to delete the root element from a max heap

We assume that heap H with n elements is stored using a sequential array called HEAP. LAST is the last element in the heap and PTR, LEFT, and RIGHT denote the position of LAST and its left and right children respectively as it moves down the heap.

1.5 BALANCED BINARY TREES:

AVL TREES:

AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is named AVL in honour of its inventors. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).

The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the BalanceFactor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a *left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a *right-heavy tree*.

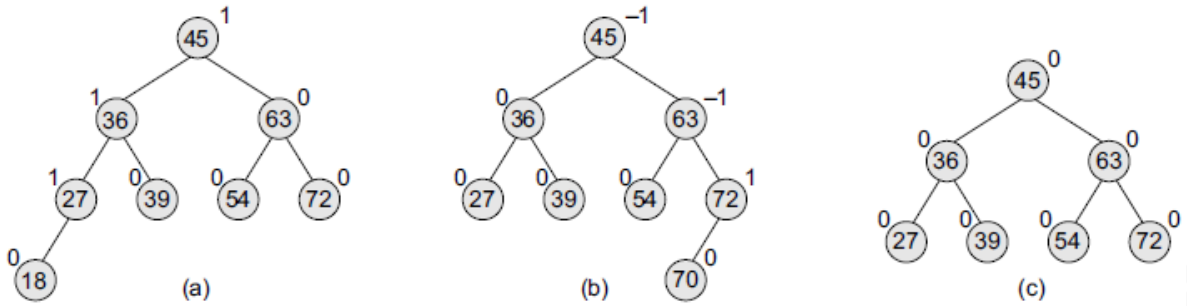


Figure 1.32 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Look at Fig. 1.32. Note that the nodes 18, 39, 54, and 72 have no children, so their balance factor = 0. Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height of right sub-tree = 0. Thus, its balance factor = 1. Look at node 36, it has a left sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = 2 – 1 = 1. Similarly, the balance factor of node 45 = 3 – 2 =1; and node 63 has a balance factor of 0 (1 – 1).

Now, look at Figs 1.32 (a) and (b) which show a right-heavy AVL tree and a balanced AVL tree.

The trees given in Fig. 1.32 are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done. The tree is rebalanced by performing rotation at the critical node. There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation. The type of rotation that has to be done will vary depending on the particular situation.

1) Inserting a New Node in an AVL Tree:

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree.

However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.

During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node. The possible changes which may take place in any node on the path are as follows:

- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a *critical node*.

Consider the AVL tree given in Fig. 1.33.

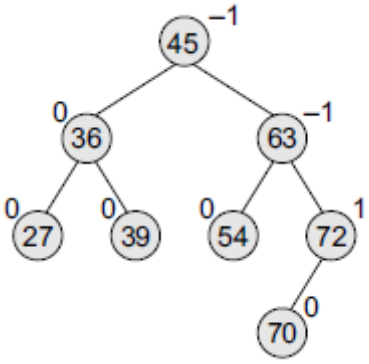


Figure 1.33 AVL tree

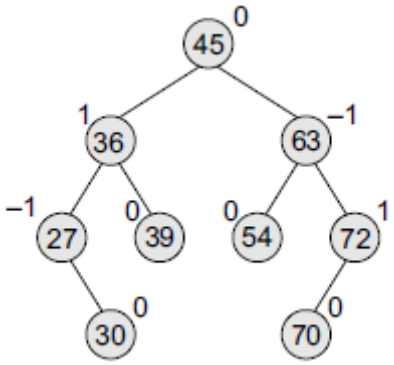


Figure 1.34 AVL tree after inserting a node with the value 30

If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case. Look at the tree given in Fig. 1.34 which shows the tree after inserting node 30.

Let us take another example to see how insertion can disturb the balance factors of the nodes and how rotations are done to restore the AVL property of a tree. Look at the tree given in Fig. 1.35.

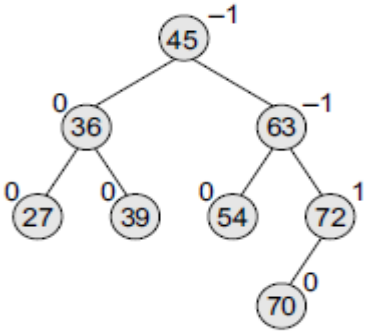


Figure 1.35 AVL tree

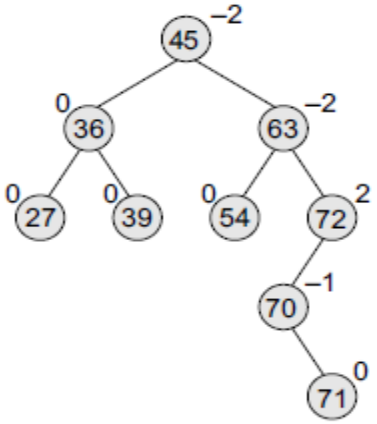


Figure 1.36 AVL tree after inserting a node with the value 71

After inserting a new node with the value 71, the new tree will be as shown in Fig. 1.36. Note that there are three nodes in the tree that have their balance factors 2, –2, and –2, thereby disturbing the *AVLness* of the tree. So, here comes the need to perform rotation. To perform rotation, our first task is to find the critical node. Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1. In the tree given above, the critical node is 72. The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node. The four categories of rotations are:

1. **LL rotation** The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
2. **RR rotation** The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
3. **LR rotation** The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
4. **RL rotation** The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

LL Rotation*:*

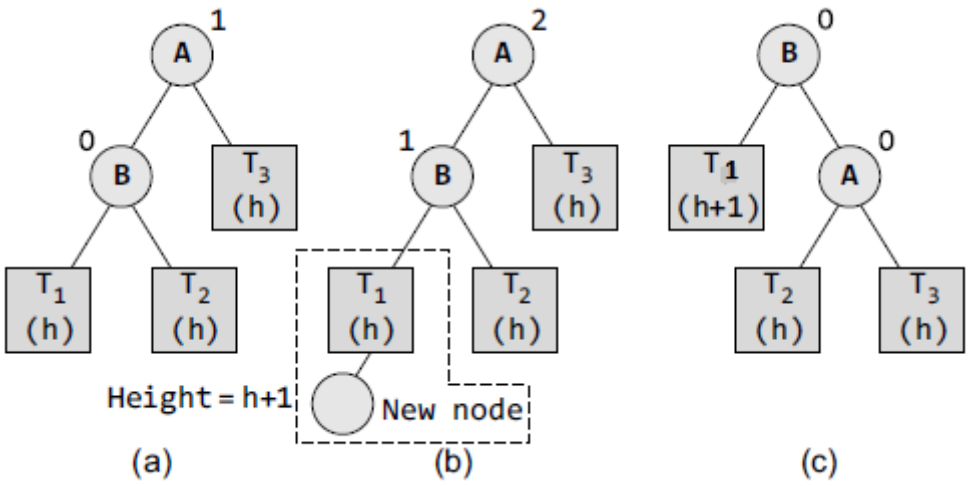


Figure 1.37 LL rotation in an AVL tree

Consider the tree given in Fig. 1.37 which shows an AVL tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply LL rotation as shown in tree (c).

While rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A. Note that the new node has now become a part of tree T1.

Example: Consider the AVL tree given in Fig. 1.38 and insert 18 into it.

Solution:

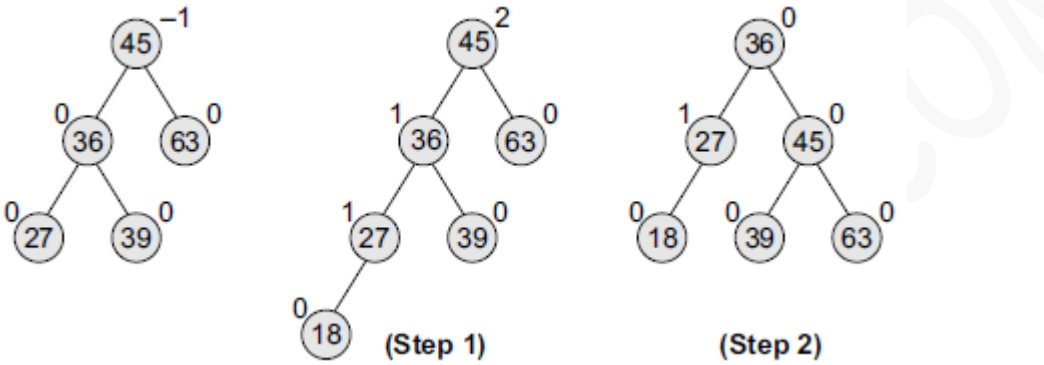


Figure 1.38 AVL tree

RR Rotation:

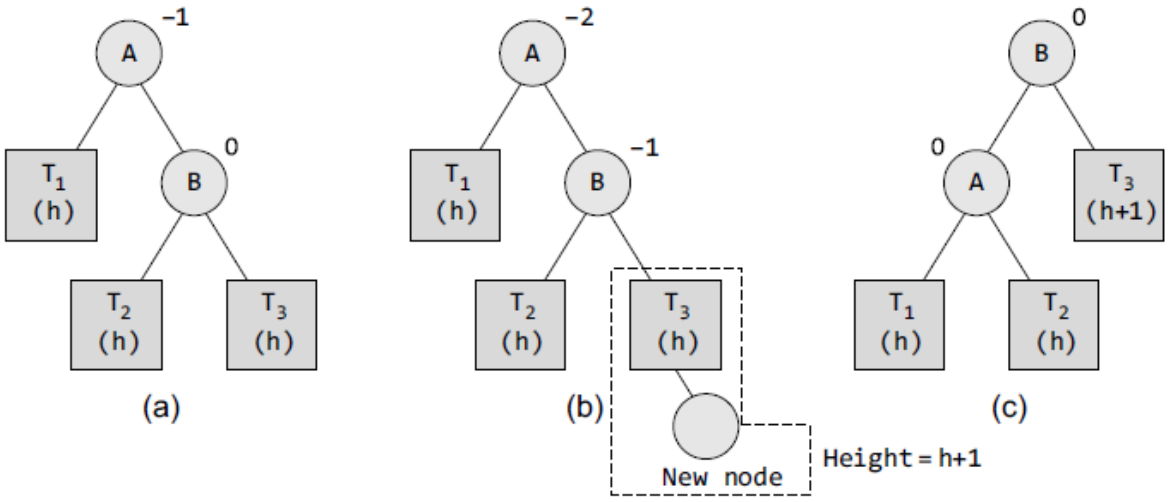


Figure 1.39 RR rotation in an AVL tree

Let us now discuss where and how RR rotation is applied. Consider the tree given in Fig. 1.39 which shows an AVL tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RR rotation as shown in tree (c). Note that the new node has now become a part of tree T3.

While rotation, node B becomes the root, with A and T3 as its left and right child. T1 and T2 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.40 and insert 89 into it.

Solution:

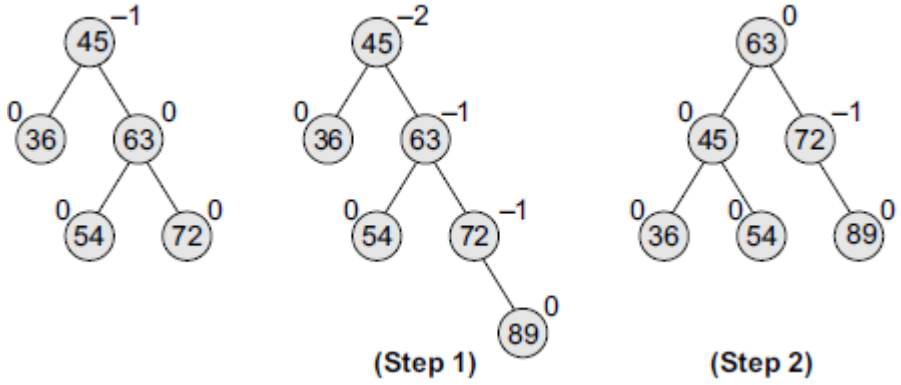


Figure 1.40 AVL tree

LR Rotation:

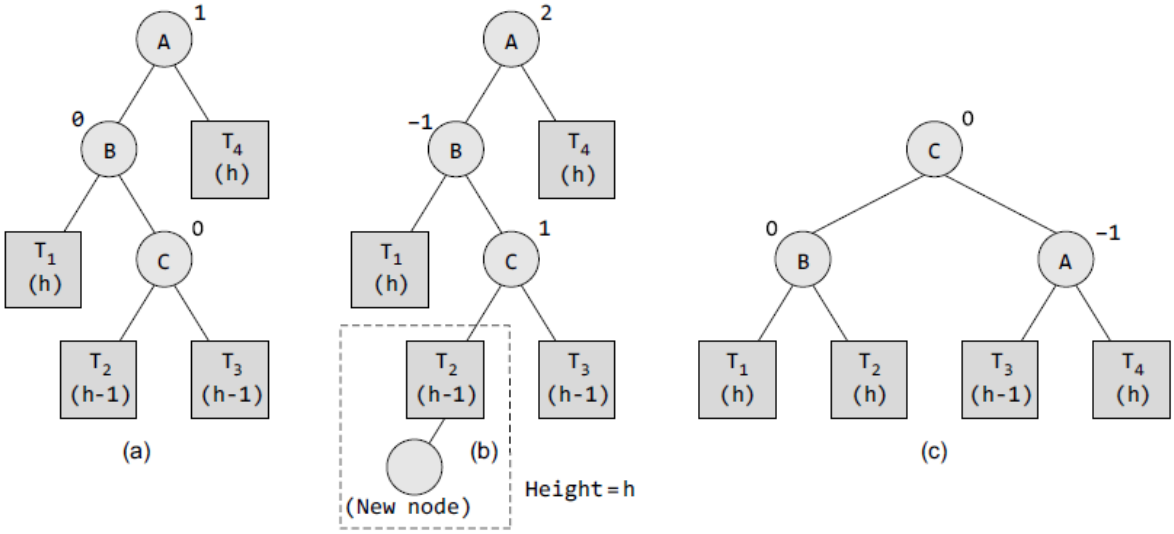


Figure 1.41 LR rotation in an AVL tree

Consider the AVL tree given in Fig. 1.41 and see how LR rotation is done to rebalance the tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1), so we apply LR rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with B and A as its left and right children. Node B has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node A.

Example: Consider the AVL tree given in Fig. 1.42 and insert 37 into it.

Solution:

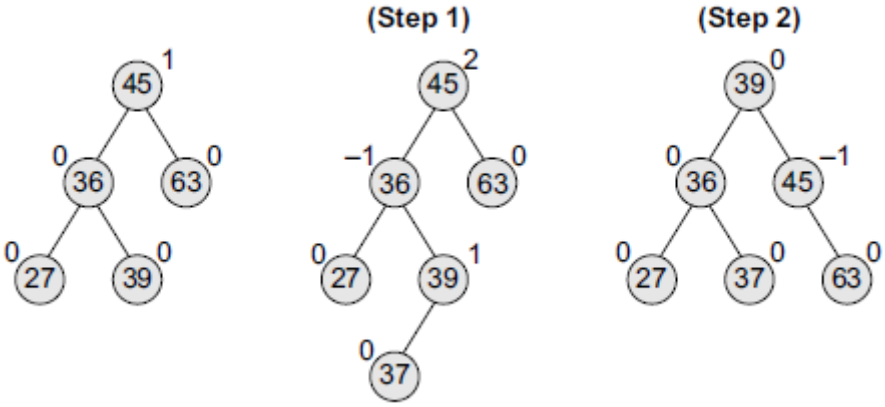


Figure 1.42 AVL tree

RL Rotation:

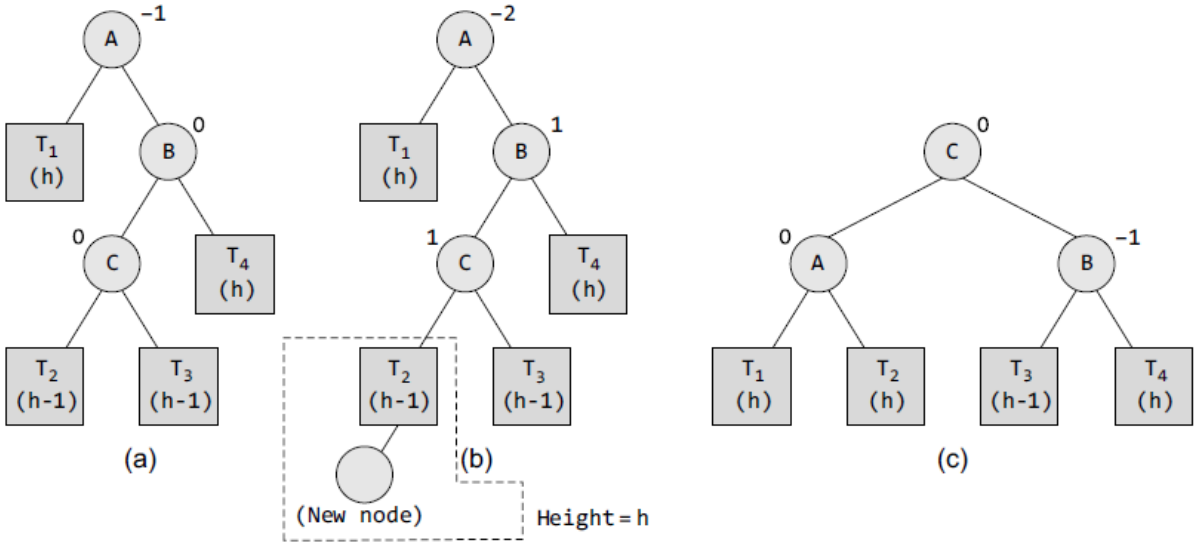
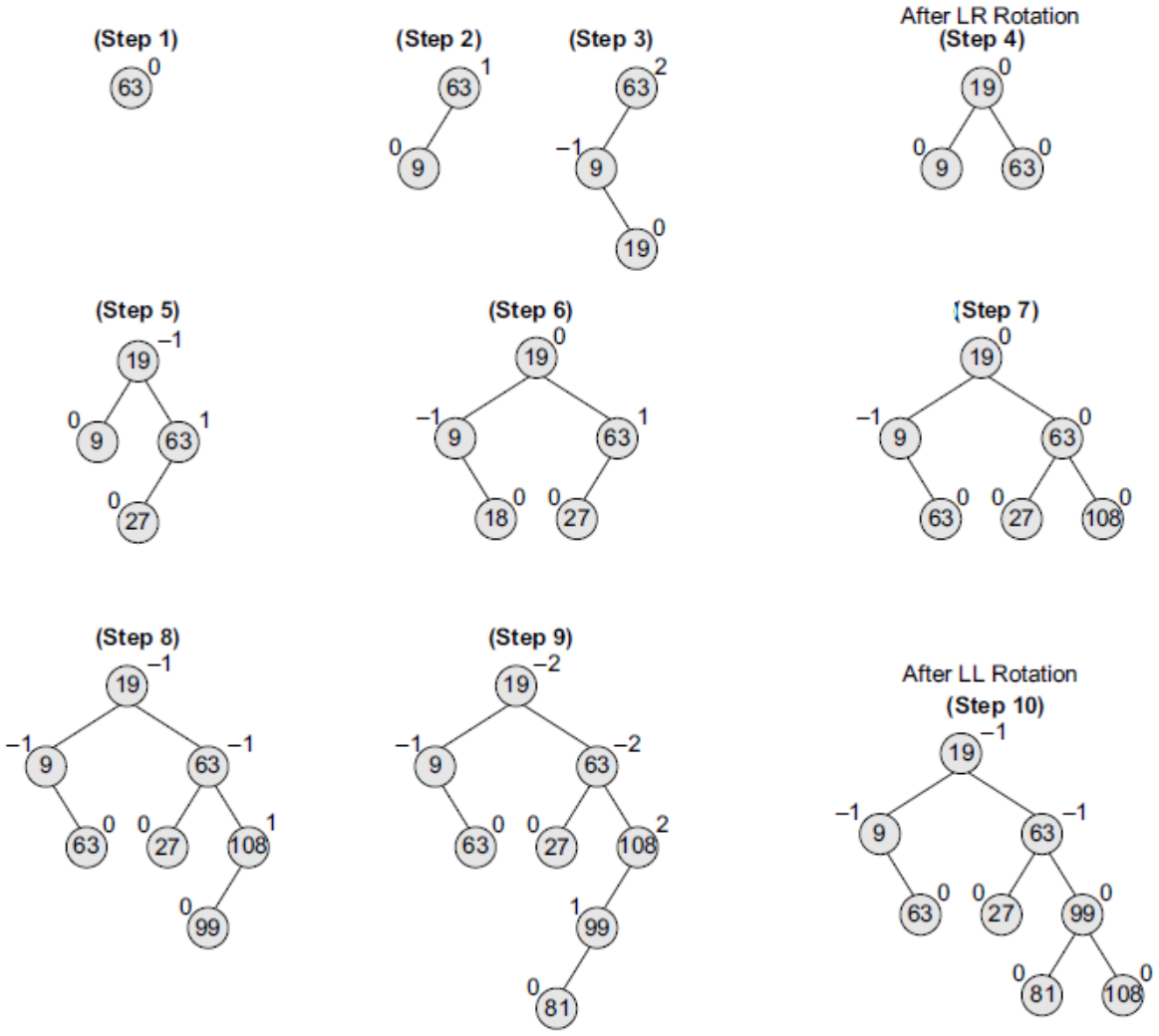


Figure 1.43 RL rotation in an AVL tree

Consider the AVL tree given in Fig. 1.43 and see how RL rotation is done to rebalance the tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RL rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with A and B as its left and right children. Node A has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node B.

Example: Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

Solution:



2) Deleting a Node from an AVL Tree:

Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation. On deletion of node X from the AVL tree, if node A becomes the critical node (closest ancestor node on the path from X to the root node that does not have its balance factor as 1, 0, or –1), then the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree. If the node to be deleted is present in the left sub-tree of A, then L rotation is applied,

else if X is in the right sub-tree, R rotation is performed. Further, there are three categories of L and R rotations. The variations of L rotation are L–1, L0, and L1 rotation. Correspondingly for R rotation, there are R0, R–1, and R1 rotations. In this section, we will discuss only R rotation. L rotations are the mirror images of R rotations.

R0 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0. This is illustrated in Fig. 1.44.

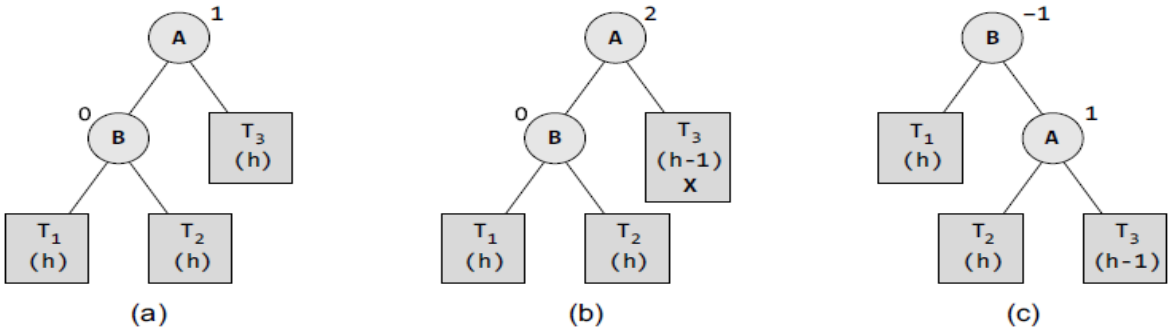


Figure 1.44 R0 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 0, we apply R0 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.45 and delete 72 from it.

Solution:

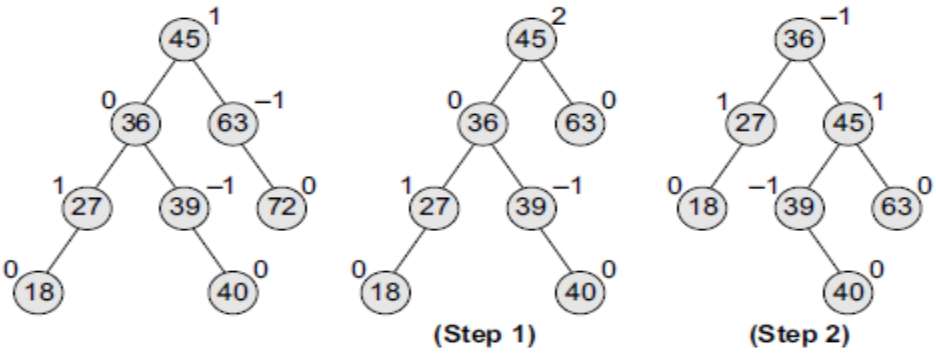


Figure 1.45 AVL tree

R1 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1. Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors. This is illustrated in Fig. 1.46.

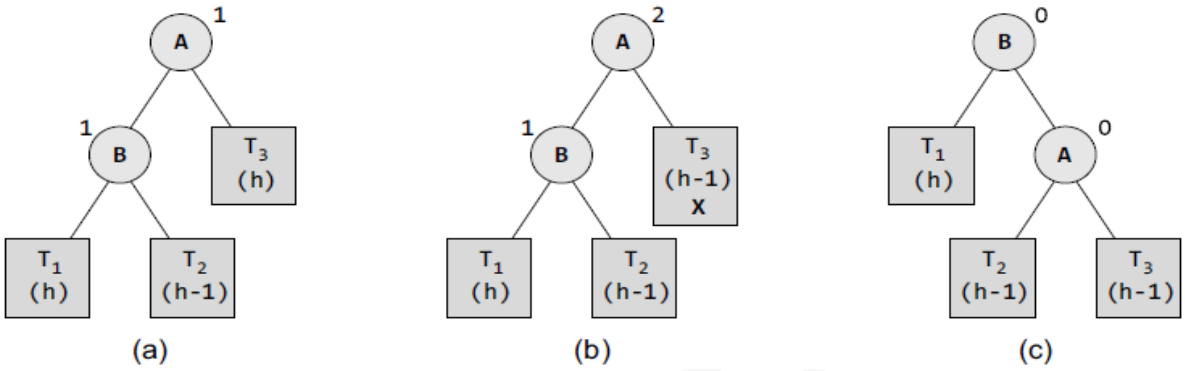


Figure 1.46 R1 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c).

During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.47 and delete 72 from it.

Solution:

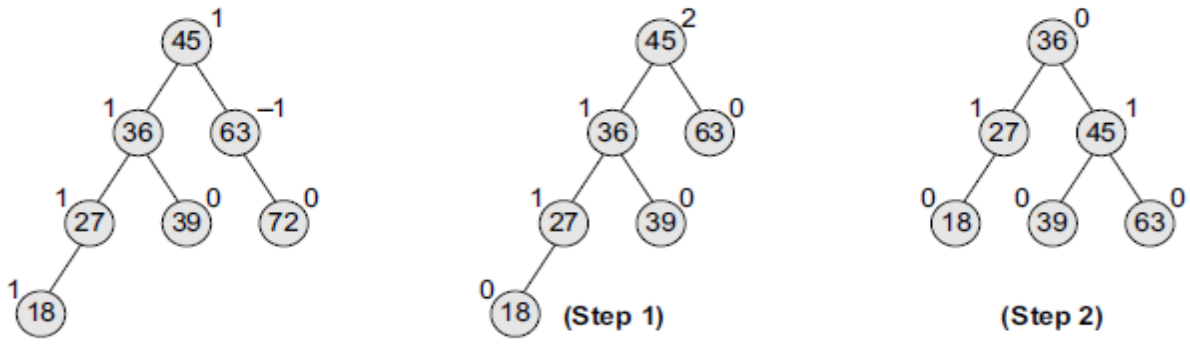


Figure 1.47 AVL tree

R–1 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R–1 rotation is applied if the balance factor of B is –1. Observe that R–1 rotation is similar to LR rotation. This is illustrated in Fig. 1.48

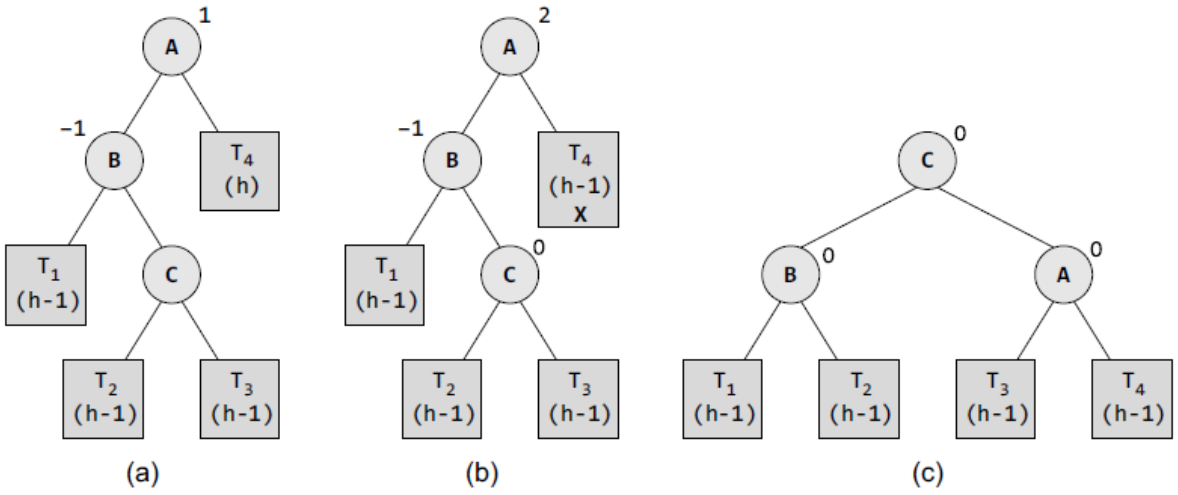


Figure 1.48 R-1 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1). Since the balance factor of node B is –1, we apply R–1 rotation as shown in tree (c). While rotation, node C becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.49 and delete 72 from it.

Solution:

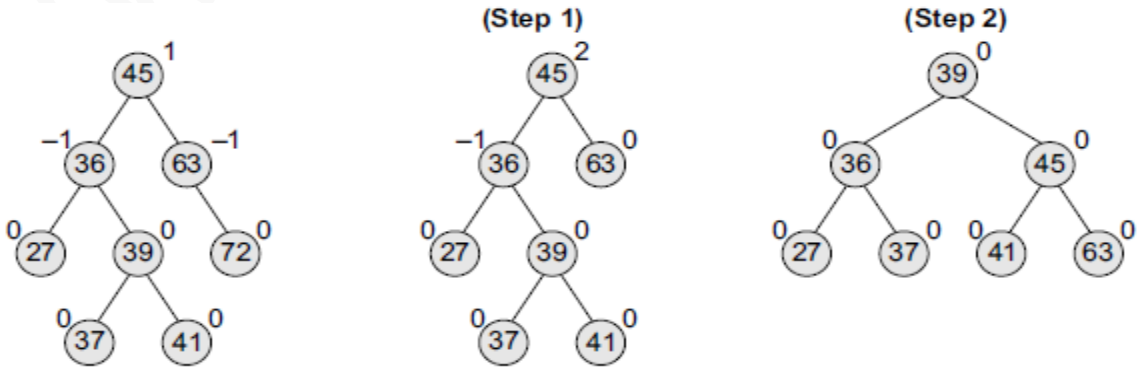
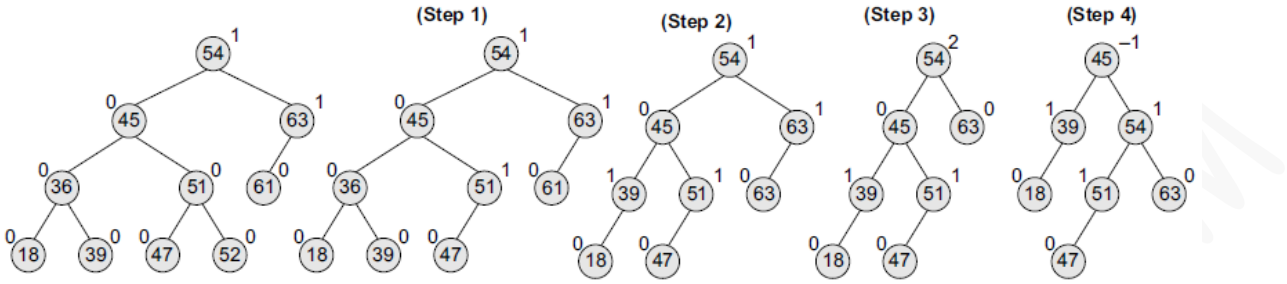


Figure 1.49 AVL tree

Example: Delete nodes 52, 36, and 61 from the given AVL tree.

Solution:



UNIT-V

GRAPHS

1. BASIC CONCEPTS

INTRODUCTION

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

WHY GRAPHS ARE USEFUL

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- *Family trees:* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks :* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

DEFINATION:

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

We have two types of Graphs. Basically:

1. UNDIRECTED GRAPH
2. DIRECTED GRAPH

UNDIRECTED GRAPH**:**

Shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D,E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

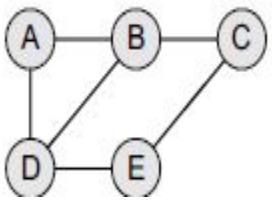


FIGURE 5.1

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 5.1 shows an undirected graph because it does not give any information about the direction of the edges.

DIRECTED GRAPH:

A directed graph G, also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

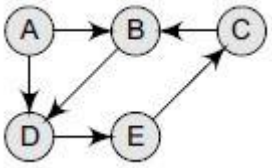


FIGURE 5.2

Which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

2. REPRESENTATION OF GRAPHS

There are two common ways of storing graphs in the computer's memory. They are:

- ***Sequential representation*** by using an adjacency matrix.
- ***Linked representation*** by using an adjacency list that stores the neighbours of a node using a linked list.

2.1 ADJACENCY MATRIX REPRESENTATION

An adjacency matrix is used to represent which nodes are adjacent to one another.

By definition: Two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n X n.

In an adjacency matrix, the rows and columns are labelled by graph vertices.

- An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other.
- However, if the nodes are not adjacent, a_{ij} will be set to zero.

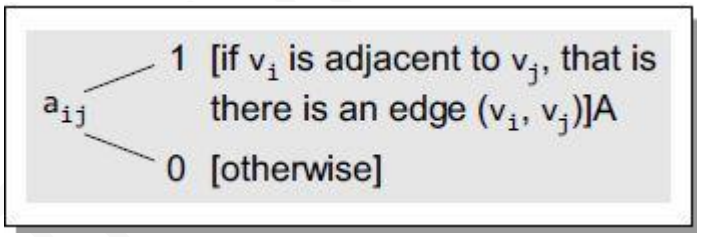


FIGURE 5.3 Adjacency Matrix Entry

Since an adjacency matrix contains only 0s and 1s, it is called a *bit matrix* or a *Boolean matrix.* The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

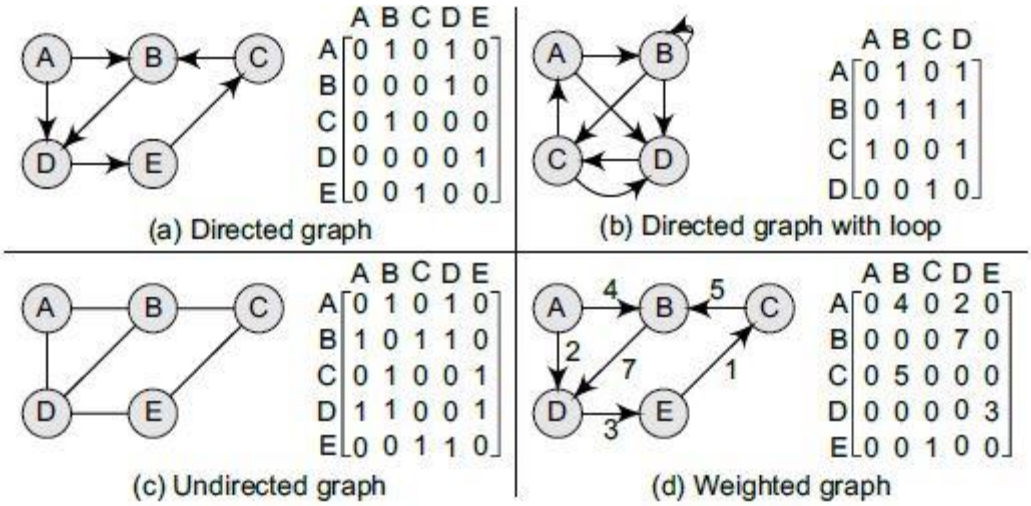


Figure 5.4 shows some graphs and their corresponding adjacency matrices.

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix:

From adjacency matrix A1, we can conclude that an entry 1 in the ith row and jth column means that there exists a path of length 1 from v_i to v_j. Now consider, A2, A3, and A4.

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (v_i, v_k) and (v_k, v_j), then there is a path from vi to vj of length 2.

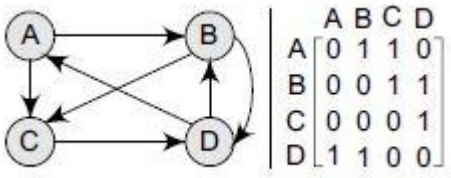


FIGURE 5.5 Directed graph with its adjacency matrix

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$B_r = A_1 + A_2 + A_3 + \ldots + A_r$

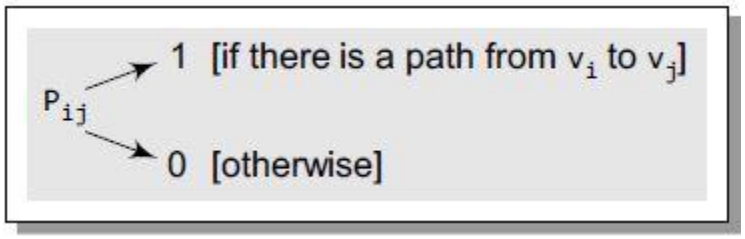


FIGURE 5.6 Path Matrix Entry

The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry Pij = 1, if Bij is non-zero and Pij = 0, if otherwise. The path matrix is used to show whether there exists a simple path from node vi to vj or not.

Let us now calculate matrix B and matrix P using the above discussion.

$$\begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\1&1&0&0\end{bmatrix}+\begin{bmatrix}0&0&1&2\\1&1&0&1\\1&1&0&0\\1&1&2&1\end{bmatrix}+\begin{bmatrix}2&2&0&1\\1&2&2&1\\0&1&2&1\\1&2&2&3\end{bmatrix}+\begin{bmatrix}1&3&4&2\\1&2&3&4\\1&1&1&3\\3&4&3&4\end{bmatrix}=\begin{bmatrix}3&6&6&5\\3&5&6&7\\2&3&3&5\\6&8&7&8\end{bmatrix}$$

B

Now the path matrix P can be given as:

$$P=\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}$$

2.2 ADJACENCY LINKED LIST REPRESEENTATION

- An adjacency list is another way in which graphs can be represented in the computer's memory.
- This structure consists of a list of all nodes in G.
- Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

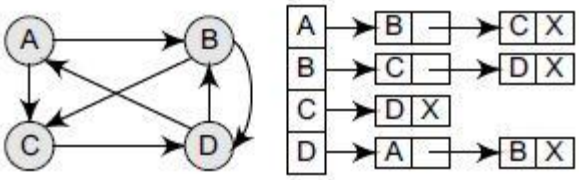


FIGURE 5.7 Graph G and its adjacency list

- For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.
- However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u.
- Adjacency lists can also be modified to store weighted graphs.

Let us now see an adjacency list for an undirected graph as well as a weighted graph.

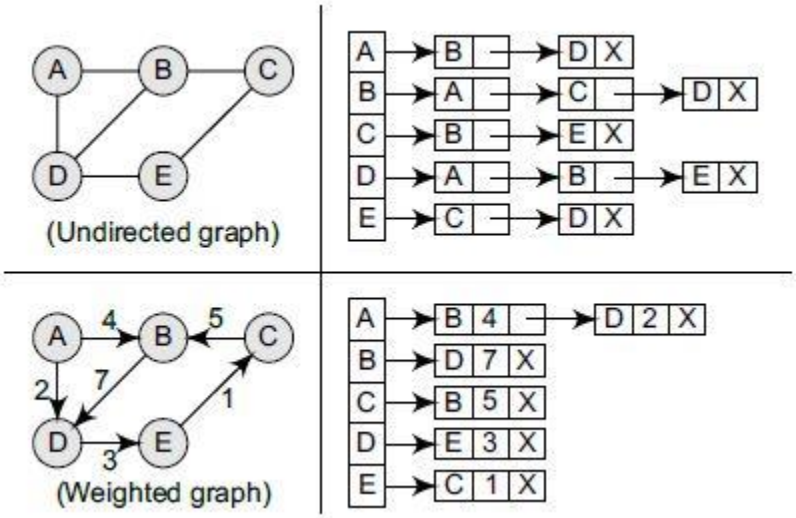


FIGURE 5.8 Adjacency list for an undirected graph and a weighted graph

PROGRAMMING EXAMPLE

1. Write a program to create a graph of *n* vertices using an adjacency list. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
char vertex;
struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);
int main()
{
struct node *Adj[10];
int i, no_of_nodes;
clrscr();
printf("\n Enter the number of nodes in G: ");
scanf("%d", &no_of_nodes);
for(i = 0; i < no_of_nodes; i++)
Adj[i] = NULL;
createGraph(Adj, no_of_nodes);
printf("\n The graph is: ");
displayGraph(Adj, no_of_nodes);
deleteGraph(Adj, no_of_nodes);
getch();
return 0;
}
```

```
void createGraph(struct node *Adj[], int no_of_nodes)
{
struct node *new_node, *last;
int i, j, n, val;
for(i = 0; i < no_of_nodes; i++)
{
last = NULL;
printf("\n Enter the number of neighbours of %d: ", i);
scanf("%d", &n);
for(j = 1; j <= n; j++)
{
printf("\n Enter the neighbour %d of %d: ", j, i);
scanf("%d", &val);
new_node = (struct node *) malloc(sizeof(struct node));
new_node –> vertex = val;
new_node –> next = NULL;
if (Adj[i] == NULL)
Adj[i] = new_node;
else
last –> next = new_node;
last = new_node
}
}
}
void displayGraph (struct node *Adj[], int no_of_nodes)
```



```
{
struct node *ptr;
int i;
for(i = 0; i < no_of_nodes; i++)
{
```

```
ptr = Adj[i];
printf("\n The neighbours of node %d are:", i);
while(ptr != NULL)
{
printf("\t%d", ptr –> vertex);
ptr = ptr –> next;
}
}
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{
int i;
struct node *temp, *ptr;
for(i = 0; i <= no_of_nodes; i++)
{
ptr = Adj[i];
while(ptr ! = NULL)
{
temp = ptr;
ptr = ptr –> next;
free(temp);
}
Adj[i] = NULL;
}
}
```

Output

```
Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 2
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0
```

Enter the neighbour 2 of 1: 2

Enter the number of neighbours of 2: 1

Enter the neighbour 1 of 2: 1

The neighbours of node 0 are: 1

The neighbours of node 1 are: 0 2

The neighbours of node 2 are: 0

Note If the graph in the above program had been a weighted graph, then the structure of the node would have been:

```
typedef struct node
{
int vertex;
int weight;
struct node *next;
};
```

3.GRAPH TRAVERSALS

There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search
2. Depth-first search

1.Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

ALGORITHM

Step 1: SET STATUS = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node A

and set its STATUS = 2

(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).
Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT

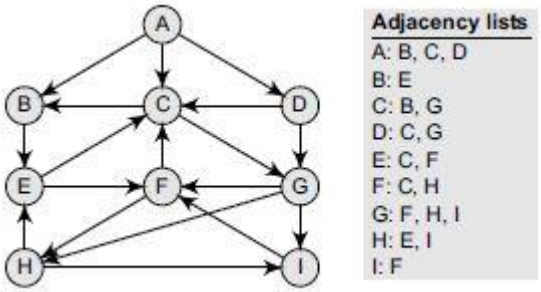


FIGURE 5.9 Graph G And Its Adjacnecy List

EXAMPLE

Consider the graph G given in Fig. 5.9.The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1**.**

SOLUTION:

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:

1. QUEUE
2. ORIG

- While QUEUE is used to hold the nodes that have to be processed,
- ORIG is used to keep track of the origin of each edge.
- Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0 QUEUE = A

REAR = 0 ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1 QUEUE = A B C D

REAR = 3 ORIG = \0 A A A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2 QUEUE = A B C D E

REAR = 4 ORIG = \0 A A A B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5 QUEUE = A B C D E G F

REAR = 6 ORIG = \0 A A A B C E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6 QUEUE = A B C D E G F H I

REAR = 9 ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have
P as A -> C -> G -> I.

Features of Breadth-First Search Algorithm

Space complexity:

The space complexity is therefore proportional to the number of nodes at the deepest level of the graph.

Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(^{bd})$.

The space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time Complexity:

In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(^{bd})$.

However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness:

Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality:

Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node.

we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Programming Example

2. Write a program to implement the breadth-first search algorithm.

```
#include <stdio.h>

#define MAX 10

void breadth_first_search(int adj[][MAX],int visited[],int start)

{

int queue[MAX],rear = –1,front =– 1, i;

queue[++rear] = start;

visited[start] = 1;

while(rear != front)

{

start = queue[++front];

if(start == 4)

printf("5\t");

else

printf("%c \t",start + 65);

for(i = 0; i < MAX; i++)

{
```

```
if(adj[start][i] == 1 && visited[i] == 0)
{
queue[++rear] = i;
visited[i] = 1;
}
}
}
}
int main()
{
int visited[MAX] = {0};
int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E
```

2. Depth First Algorithm

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.

When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set
its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its
STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that
are in the ready state (whose STATUS = 1) and
set their STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT

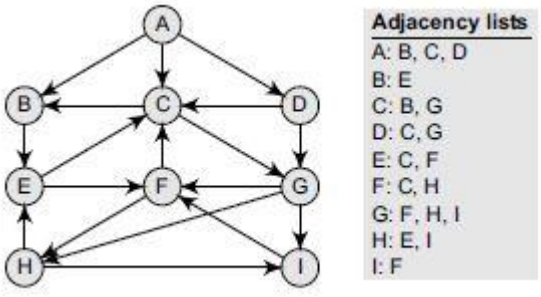


FIGURE 5.10 Graph G And Its Adjacency List

Example:

Consider the graph G given in. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

Solution:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity:

The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity:

The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V|+|E|)).

Completeness:

Depth-first search is said to be a complete algorithm. If there is a solution, depthfirst search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Programming Example:

3. Write a program to implement the depth-first search algorithm.

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
int stack[MAX];
int top = –1, i;
printf("%c–",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top ! = –1)
{
start = stack[top];
for(i = 0; i < MAX; i++)
{
if(adj[start][i] && visited[i] == 0)
{
stack[++top] = i;
printf("%c–", i + 65);
visited[i] = 1;
break;
}
}
if(i == MAX)
top—;
}
}
int main()
{
int adj[MAX][MAX];
```

```
int visited[MAX] = {0}, i, j;
```



```
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
printf("DFS Traversal: ");
depth_first_search(adj,visited,0);
printf("\n");
return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A –> C –> E –>
```

APPLICATIONS

MINIMUM SPANNING TREES:

- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together
- A graph G can have many different spanning trees.
- We can assign *weights* to each edge (which is a number that represents how unfavourable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.
- A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning

tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Example: Consider an unweighted graph G given below (Fig. 5.11). From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.

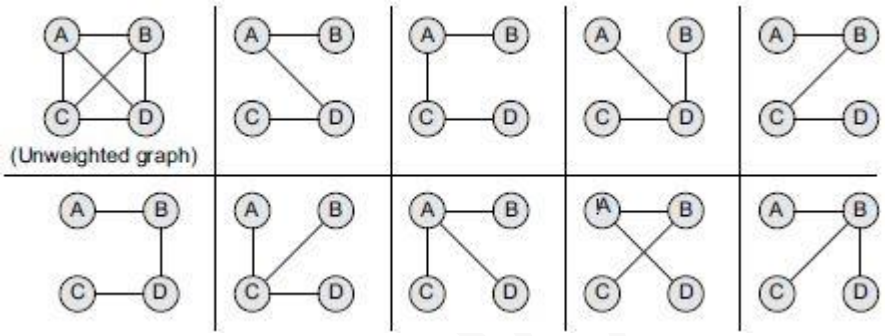


FIGURE 5.11 Unweighted Graph And Its Spanning Trees

EXAMPLE: Consider a weighted graph G shown in Fig. 5.12. From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. Of all the spanning trees given in Fig. 5.12, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.

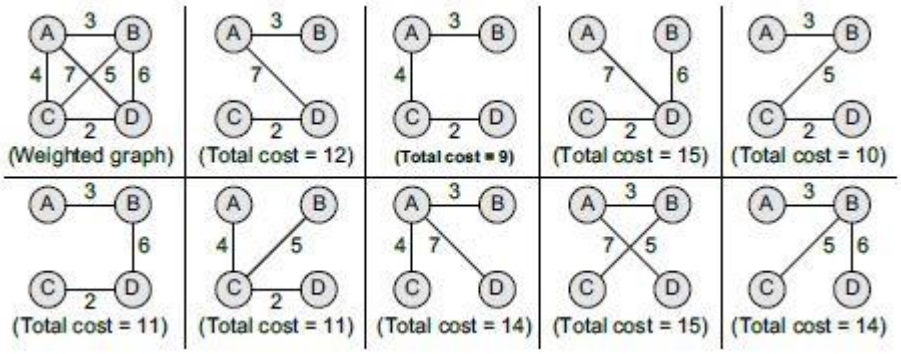


FIGURE 5.12 Weighted Graph And Its Spanning Trees.

APPLICATIONS FOR MINIMUM SPANNING TREES:

- **MST'S** is widely used for designing networks.
- **MST'S** are used to find airlane routes.
- **MST'S** are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- **MST'S** are applied in routing algorithms for finding the most efficient path.

We have two types of ALGORITHMS in Minimum Spanning Trees. They are:

1. PRIM'S ALGORITHM
2. KRUSKAL'S ALGORITHM

1.PRIM'S ALGORITHM

- Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.
- In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized.

For this, the algorithm maintains three sets of vertices which can be given as below:

- **Tree vertices** Vertices that are a part of the minimum spanning tree T.
- **Fringe vertices** Vertices that are currently not a part of T, but are adjacent to some tree vertex.
- **Unseen vertices** Vertices that are neither tree vertices nor fringe vertices fall under this category.

ALGORITHM

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and
fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the
minimum spanning tree T
[END OF LOOP]

Step 5: EXIT

EXAMPLE: Construct a minimum spanning tree of the graph given in Fig. 5.13

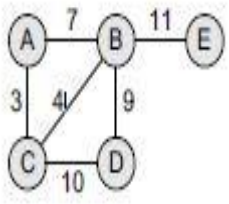


FIGURE 5.13

Step 1: Choose a starting vertex A.

Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.

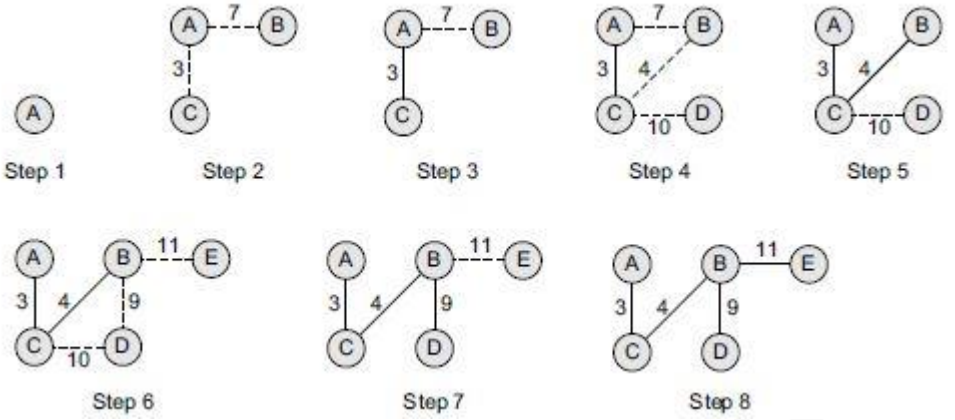
Step 4: Add the fringe vertices (that are adjacent to C).

Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.

Step 6: Add the fringe vertices (that are adjacent to B).

Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.

Step 8: Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n–1 edges that have minimum weight. So, the minimum spanning tree can now be given as,



2.KRUSKAL'S ALGORITHM

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized.
- However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

ALGORITHM

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees,
then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge

Step 6: END

EXAMPLE: Apply Kruskal's algorithm on the graph given in Fig. 5.14.

Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

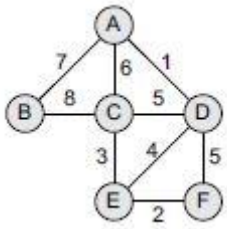
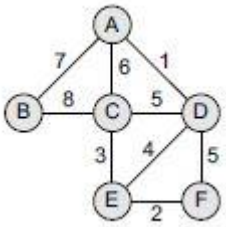


FIGURE 5.14

Step 1: Remove the edge (A, D) from Q and make the following changes:

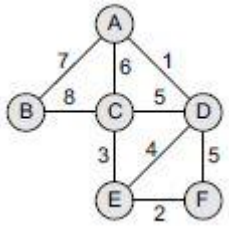


F = {{A, D}, {B}, {C}, {E}, {F}}

MST = {A, D}

Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 2: Remove the edge (E, F) from Q and make the following changes:

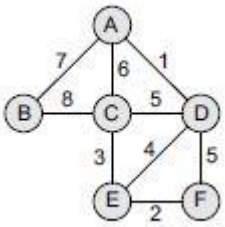


F = {{A, D}, {B}, {C}, {E, F}}

MST = {(A, D), (E, F)}

Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3*:* Remove the edge (C, E) from Q and make the following changes:

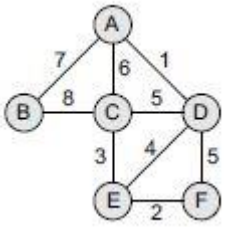


F = {{A, D}, {B}, {C, E, F}}

MST = {(A, D), (C, E), (E, F)}

Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge (E, D) from Q and make the following changes:



F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q= {(C, D), (D, F), (A, C), (A, B), (B, C)}

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(D, F), (A, C), (A, B), (B, C)}

Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(A, C), (A, B), (B, C)}

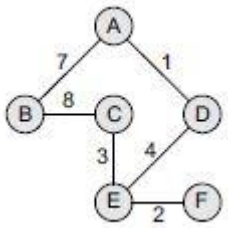
Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(A, B), (B, C)}

Step 8: Remove the edge (A, B) from Q and make the following changes:

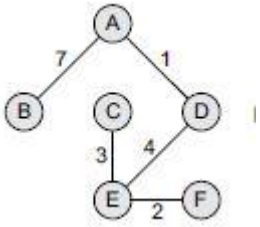


F = {A, B, C, D, E, F}

MST = {(A, D), (C, E), (E, F), (E,D), (A, B)}

Q = {(B, C)}

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below



F = {A, B, C, D, E, F}

MST = {(A, D), (C, E), (E, F), (E,D), (A, B)}

Q = {}

PROGRAMMING EXAMPLE:

5. Write a program which finds the cost of a minimum spanning tree.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int adj[MAX][MAX], tree[MAX][MAX], n;
void readmatrix()
{
int i, j;
printf("\n Enter the number of nodes in the Graph : ");
scanf("%d", &n);
printf("\n Enter the adjacency matrix of the Graph");
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
scanf("%d", &adj[i][j]);
}
int spanningtree(int src)
{
```

```
int visited[MAX], d[MAX], parent[MAX];
int i, j, k, min, u, v, cost;
for (i = 1; i <= n; i++)
{
d[i] = adj[src][i];
visited[i] = 0;
parent[i] = src;
}
visited[src] = 1;
cost = 0;
k = 1;
for (i = 1; i < n; i++)
{
min = 9999;
for (j = 1; j <= n; j++)
{
if (visited[j]==0 && d[j] < min)
{
min = d[j];
u = j;
cost += d[u];
}
}
visited[u] = 1;
//cost = cost + d[u];
tree[k][1] = parent[u];
tree[k++][2] = u;
for (v = 1; v <= n; v++)
if (visited[v]==0 && (adj[u][v] < d[v]))
{
d[v] = adj[u][v];
```

```
parent[v] = u;
}
}
return cost;
}
void display(int cost)
{
int i;
printf("\n The Edges of the Mininum Spanning Tree are");
for (i = 1; i < n; i++)
printf(" %d %d \n", tree[i][1], tree[i][2]);
printf("\n The Total cost of the Minimum Spanning Tree is : %d", cost);
}
main()
{
int source, treecost;
readmatrix();
printf("\n Enter the Source : ");
scanf("%d", &source);
treecost = spanningtree(source);
display(treecost);
return 0;
}
```

Output

Enter the number of nodes in the Graph : 4

Enter the adjacency matrix : 0 1 1 0

0 0 0 1

0 1 0 0

1 0 1 0

Enter the source : 1

The edges of the Minimum Spanning Tree are 1 4

4 2

2 3

The total cost of the Minimum Spanning Tree is : 1

Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

ALGORITHM

- Dijkstra's algorithm is used to find the length of an *optimal* path between two nodes in a graph.
- The term *optimal* can mean anything, shortest, cheapest, or fastest.
- If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node.

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with , and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.
(b) Else if the node that is not in N was already labelled, then SET its new
label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in N that has the smallest label assigned to it and add it
to N.

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node.
There are two kinds of labels: **temporary** and ***permanent*.**
Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.
The execution of this algorithm will produce either of the following two results:
1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labelled, then there is no path from the source to the destination node.

EXAMPLE:
Consider the graph G given in Fig. 5.14. Taking D as the initial node, execute the Dijkstra's algorithm on it.

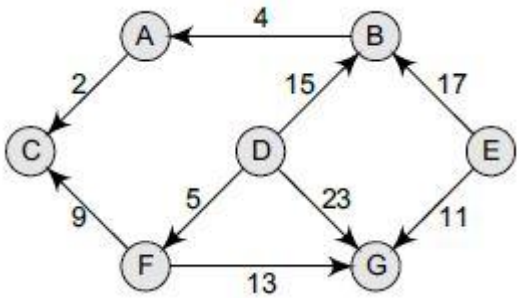


FIGURE 5.14

Step 1: Set the label of D = 0 and N = {D}.

Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.

Step 3: Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.

Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.

Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.

Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

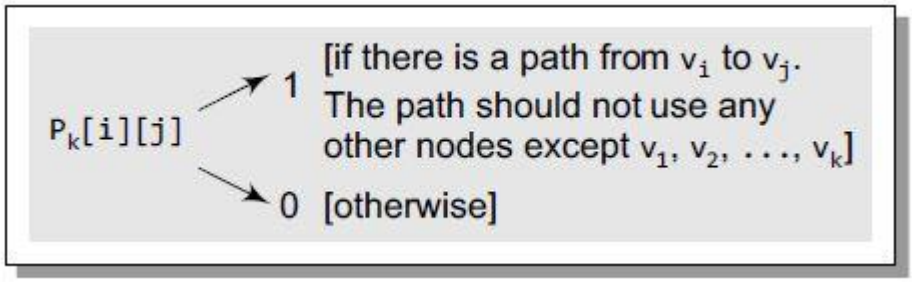
Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as O(|V|2+|E|)=O(|V|2) where V is the set of vertices and E in the graph.

Warshall's Algorithm

If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, P = A + A2 + A3 + ... + An.

This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices P0, P1, P2, º, Pn.



Path Matrix Entry

- This means that if P0[i][j] = 1, then there exists an edge from node vi to vj.
- If P1[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1.

  Hence, the path matrix Pn can be calculated with the formula given as:

  Pk[i][j] = Pk–1[i][j] V (Pk–1[i][k] ^ Pk–1[k][j])

  where **V** indicates logical OR operation and ^ indicates logical AND operation.

ALGORITHM

Step 1: [the Path Matrix] Repeat Step 2 for I = to n-1,

where n is the number of nodes in the graph

Step 2: Repeat Step 3 for J = to n-1

Step 3: IF A[I][J] = , then SET P[I][J] =

ELSE P[I][J] = 1

[END OF LOOP]

[END OF LOOP]

Step 4: [Calculate the path matrix P] Repeat Step 5 for K = to n-1

Step 5: Repeat Step 6 for I = to n-1

Step 6: Repeat Step 7 for J= to n-1

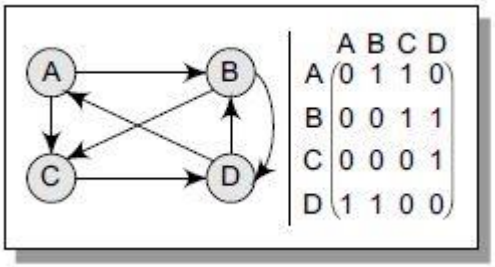
Step 7: SET P [I][J] = P [I][J] V (P [I][K]

P [K][J])

Step 8: EXIT

EXAMPLE:

Consider the graph in Fig. 13.39 and its adjacency matrix A. We can straightaway calculate the path matrix P using the Warshall's algorithm. The path matrix P can be given in a single step as:



$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

GRAPH G AND ITS PATH MATRIX

PROGRAMMING EXAMPLE

6. Write a program to implement Warshall's algorithm to find the path matrix.

```
#include <stdio.h>
#include <conio.h>
void read (int mat[5][5], int n);
void display (int mat[5][5], int n);
void mul(int mat[5][5], int n);
int main()
{
int adj[5][5], P[5][5], n, i, j, k;
clrscr();
printf("\n Enter the number of nodes in the graph : ");
scanf("%d", &n);
printf("\n Enter the adjacency matrix : ");
read(adj, n);
```

```
clrscr();
printf("\n The adjacency matrix is : ");
display(adj, n);
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(adj[i][j] == 0)
P[i][j] = 0;
else
P[i][j] = 1;
}
}
for(k=0; k<n;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
P[i][j] = P[i][j] | ( P[i][k] & P[k][j]);
}
}
printf("\n The Path Matrix is :");
display (P, n);
getch();
return 0;
}
void read(int mat[5][5], int n)
{
int i, j;
for(i=0;i<n;i++)
{
```

```
for(j=0;j<n;j++)
{
printf("\n mat[%d][%d] = ", i, j);
scanf("%d", &mat[i][j]);
}
}
}
void display(int mat[5][5], int n)
{
int i, j;
for(i=0;i<n;i++)
printf("\n");
for(j=0;j<n;j++)
printf("%d\t", mat[i][j]);
}
}
```

Output

The adjacency matrix is

0 1 1 0

0 0 1 1

0 0 0 1

1 1 0 0

The Path Matrix is

1 1 1 1

1 1 1 1

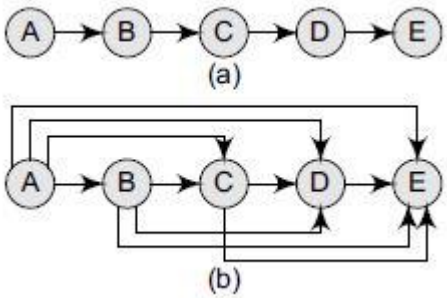
1 1 1 1

1 1 1 1

Transitive Closure of a Directed Graph

Definition

For a directed graph G = (V,E), where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph G* = (V,E*). In G*, for every vertex pair v, w in V there is an edge (v, w) in E* if and only if there is a valid path from v to w in G.



(a) A graph G and its
(b) transitive closure G*

Where and Why is it Needed?

Finding the transitive closure of a directed graph is an important problem in the following computational tasks:

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.I
- It is used in the construction of parsing automata in compiler construction.
- Recently, transitive closure computation is being used to evaluate recursive database queries (because almost all practical recursive queries are transitive in nature).

ALGORITHM

Transitive_Closure(A, t, n)

Step 1: SET i=1, j=1, k=1

Step 2: Repeat Steps 3 and 4 while i<=n

Step 3: Repeat Step 4 while j<=n

Step 4: IF (A[i][j] = 1)

SET t[i][j] = 1

ELSE

SET t[i][j] =

INCREMENT j

[END OF LOOP]

INCREMENT i

[END OF LOOP]

Step 5: Repeat Steps 6 to 11 while k<=n

Step 6: Repeat Steps 7 to 1 while i<=n

Step 7: Repeat Steps 8 and 9 while j<=n

Step 8: SET t[i,j] = t[i][j] V (t[i][k] t[k][j])

Step 9: INCREMENT j

[END OF LOOP]

Step 10 : INCREMENT i

[END OF LOOP]

Step 11: INCREMENT k

[END OF LOOP]

Step 12: END