RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

CSE-Artificial Intelligence and Machine Learning/ Artificial Intelligence and Machine Learning, III-Semester

AL303 Data Structures

1.Introduction to Data Structure: Concepts of Data and Information, Classification of Data structures, Abstract Data Types, Implementation aspects: Memory representation. Data structures operations and its cost estimation. Introduction to linear data structures- Arrays, Linked List: Representation of linked list in memory, different implementation of linked list. Circular linked list, doubly linked list, etc. Application of linked list: polynomial manipulation using linked list, etc.

2. Stacks and Queue: Stacks as ADT, Different implementation of stack, multiple stacks. Application of Stack: Conversion of infix to postfix notation using stack, evaluation of postfix expression, Recursion. Queues: Queues as ADT, Different implementation of queue, Circular queue, Concept of Dqueue and Priority Queue, Queue simulation, Application of queues.

3. Tree: Definitions - Height, depth, order, degree etc. Binary Search Tree - Operations, Traversal, Search. AVL Tree, Heap, Applications and comparison of various types of tree; Introduction to forest, multi-way Tree, B tree, B+ tree, B* tree and red-black tree.

4. Graphs: Introduction, Classification of graph: Directed and Undirected graphs, etc, Representation, Graph Traversal: Depth First Search (DFS), Breadth First Search (BFS), Graph algorithm: Minimum Spanning Tree (MST)-Kruskal, Prim's algorithms. Dijkstra's shortest path algorithm; Comparison between different graph algorithms. Application of graphs.

5. Sorting: Introduction, Sort methods like: Bubble Sort, Quick sort. Selection sort, Heap sort, Insertion sort, Shell sort, Merge sort and Radix sort; comparison of various sorting techniques. Searching: Basic Search Techniques: Sequential search, Binary search, Comparison of search methods. Hashing & Indexing. Case Study: Application of various data structures in operating system, DBMS etc.

Text Books

1. AM Tanenbaum, Y Langsam& MJ Augustein, "Data structure using C and C++", Prentice Hall India.

2. Robert Kruse, Bruse Leung, "Data structures & Program Design in C", Pearson Education.

Reference Books

1. Aho, Hopcroft, Ullman, "Data Structures and Algorithms", Pearson Education.

2. N. Wirth, "Algorithms + Data Structure = Programs", Prentice Hall.

3. Jean – Paul Trembly , Paul Sorenson, "An Introduction to Data Structure with application", TMH.

4. Richard, GilbergBehrouz, Forouzan ,"Data structure – A Pseudocode Approach with C", Thomson press.

Subject Notes
AL303 – Data Structure
Unit -1

REVIEW OF C PROGRAMMING LANGUAGE:

C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. By design, C provides constructs that map efficiently to typical machine instructions, and therefore it has found lasting use in applications that had formerly been coded in assembly language, including operating systems, as well as various application software for computers ranging from supercomputers to embedded systems.
C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs.C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support.

INTRODUCTION:

~~Computer Scienc~~e is the study of data, its representation and transformation by Computer. For every data object, we consider the class of operations to be performed and then the way to represent the object so that these operations may be efficiently carried out. We require two techniques for this:

- Devise alternative forms of data representation
- Analyse the algorithm which operates on the structure.

These are several terms involved above which we need to know carefully before we proceed. These include data structure, data type and data representation.

A data type is a term which refers to the kinds of data that variables may hold. With every programming language there is a set of built-in data types. This means that the language allows variables to name data of that type and provides a set of operations which meaningfully manipulates these variables. Some data types are easy to provide because they are built-in into the computer's machine language instruction set, such as integer, character etc. Other data types require considerably more efficient to implement. In some languages, these are features which allow one to construct combinations of the built-in types(like structures in 'C'). However, it is necessary to have such mechanism to create the new complex data types which are not provided by the programming language. The new type also must be meaningful for manipulations. Such meaningful data types are referred as abstract data type.

DATA STRUCTURE:

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every ~~enterprise applicat~~ion uses various types of data structures in one or the other way.
Thus, a data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion.**:**Data Structure is a way of collecting and organising data in such a way that
we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type. **Programming:**
Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer. Although many programming languages and many different types of computers exist, the important first step is the need to have the solution. Without an

algorithm there can be no program.Computer science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

CONCEPTS OF DATA AND INFORMATION:

Data: Data are simply values or set of values. A data item refers to a single unit of item.
Information: Information is any entity or form that provides the answer to a question of some kind or resolves uncertainty. It is thus related to data and knowledge, as data represents values attributed to parameters, and knowledge signifies understanding of real things or abstract concepts.

CLASSIFICATION OF DATA STRUCTURES: -

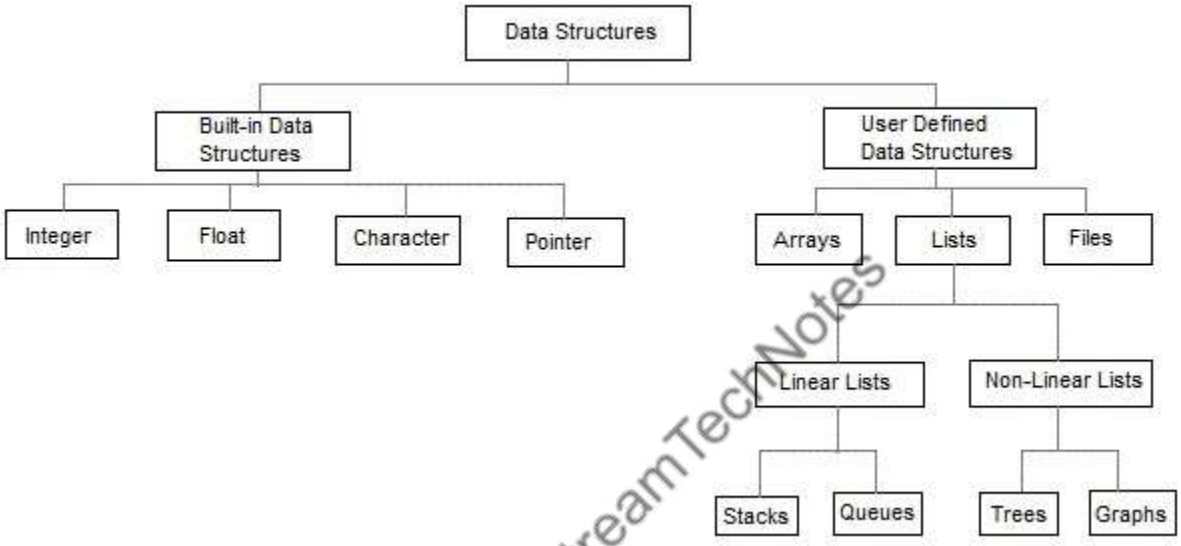


Figure 1.1: Classification of Data Structure

A data structure can be broadly classified into
(i) Built-in/Primitive data structure
(ii) User Defined/Non-primitive data structure

(i) Primitive data structure

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double incase of 'c' are known as primitive data structures.

(ii) Non-primitive data structure

The data structures, which are not primitive, are called non-primitive data structures.
There are two types of non-primitive data structures.

Linear Data Structures:-

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

Non-linear data structure:-

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Linear Data Structure:

A list is an ordered list, which consists of different data items connected by means of a link or pointer. This type of list is also called a linked list. A linked list may be a single list or double linked list.

- **Single linked list: -** A single linked list is used to traverse among the nodes in one direction.
- **Double linked list: -** A double linked list is used to traverse among the nodes in both the directions.

A linked list is normally used to represent any data used in word-processing applications, also applied in different DBMS packages.

A list has two subsets. They are: -

- **Stack: -** It is also called as last-in-first-out (LIFO) system. It is a linear list in which insertion and deletion take place only at one end. It is used to evaluate different expressions.
- **Queue:** - It is also called as first-in-first-out (FIFO) system. It is a linear list in which insertion takes place at once end and deletion takes place at other end. It is generally used to schedule a job in operating systems and networks.

Non-linear data structure:-

The frequently used non-linear data structures are

- **Trees: -** It maintains hierarchical relationship between various elements
- **Graphs:** - It maintains random relationship or point-to-point relationship between various elements.

The data structures can also be classified on the basis of the following characteristics:

ABSTRACT DATA TYPES:

Abstract Data type (ADT) is a type or class for objects whose behavior is defined by a set of value and a set of operations.ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Example: Stack, Queue etc.

IMPLEMENTATION ASPECTS:

MEMORY REPRESENTATION:

There are 2 types of memory allocations possible in C:

Compile time or static allocation

Runtime or Dynamic allocation

1. Compile time or static allocation: Compile time memory allocation means reserving memory for variables, constants during the compilation process. So you must exactly know how many bytes you require? This type of allocation is done with the help of Array.The biggest disadvantage of compile time memory allocation, we do not have control on allocated memory. You cannot increase, decrease or free memory, the compiler takes care of memory management. We can also refer compile time memory allocation as static or stack memory allocation.

2. Runtime or Dynamic allocation: Memory allocated at runtime either through malloc(),calloc() or realloc().You can also refer runtime memory allocation as dynamic or heap memory allocation. Professional

programmers prefer dynamic memory allocation more over static memory allocation. Since, we have full control over the allocated memory. Which means we can allocate, de-allocate and can also reallocate memory when needed.

a. malloc():allocates requested size of bytes and return a void pointer pointing to the first byte the allocated space.
Syntax: malloc (no. of elements* size of each element);
For example:
int *ptr;
ptr=(int*)malloc(10*sizeof(int));

b. calloc(): allocate space for an array of element and initialize them to zero and then return a void pointer to memory.
Syntax: malloc (no. of elements, size of data type);
For example:
int *ptr;
ptr=(int*)calloc(10,2);

c. realloc(): modify the size of previously allocated space.
Syntax: ptr=realloc(ptr,newsize);

d. free(): releases previously allocated memory.

Memory Allocation Process:

• Global variable, static variable and program instruction get their memory in permanent storage and where local variable are stored in memory area called stack.

• The memory space between 2 regions is known as Heap area. This region is used for dynamic memory allocation during execution of program. The sizes of heap keep changing.

OPERATION ON DATA STRUCTURESAND ITS COST ESTIMATION:

The four major operations performed on data structures are:

• **Insertion:** - Insertion means adding new details or new node into the data structure.

• **Deletion:** - Deletion means removing a node from the data structure.

• **Traversal: -** Traversing means accessing each node exactly once so that the nodes of a data structure can be processed. Traversing is also called as visiting.

• **Searching: -** Searching means finding the location of node for a given key value.

Apart from the four operations mentioned above, there are two more operations occasionally performed on data structures. They are:

• **Sorting: -**Sorting means arranging the data in a particular order.

• **Merging: -**Merging means joining two lists.

INTRODUCTION TO LINEAR DATA STRUCTURES:

ARRAYS:

Array is set of homogenous data items represented in contiguous memory locations using a common name and sequence of indices starting from 0. Array is a simplest data structure that makes use of computed address to locate its elements. An array size is fixed and therefore requires a fixed number of memory locations.

Suppose A is an array of n elements and the starting address is given then the location and element I will be

LOC(Ai) = Base address of A + (i – 1) * W

Where W is the width of each element.

1 Dimensional Arrays:A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.
As an example consider the C declaration intanArrayName[10];
Syntax :datatypeArrayname[sizeofArray];

2 Dimensional Arrays:A multidimensional array can be represented by an equivalent one-dimensional array. A two dimensional array consisting of number of rows and columns is a combination of more than 1 one-dimensional array. A 2 dimensional array is referred in two different ways. Considering row as major order or column as major order any array may be used to refer the elements.
Declaration: inta[MAX_ROWS][MAX_COLS];
We can initialize all elements of an array to 0 like:

```
for(i = 0; i < MAX_ROWS; i++)
for(j = 0; j < MAX_COLS; j++)
a[i][j] = 0;
```

LINKED LIST:

A linked list is a linked representation of theordered list.Itisalinear collection of data elements termed as nodes whose linear order is given by means oflinkor pointer.Everynode consists of two parts. The first part is called INFO, contains information of the dataandsecondpartiscalled LINK, contains the address of the next node in the list. A variable called START, alwayspoints tothefirstnode of the list and the link part of the last node always contains null value. A null valueinthe START variabledenotes that the list is empty.

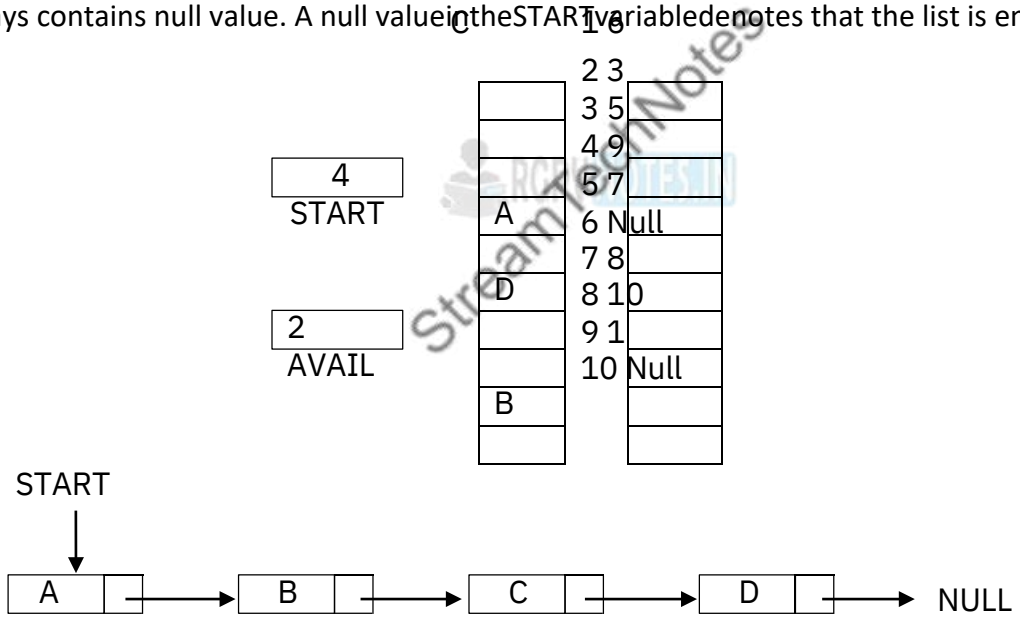


Figure 1.2: Linked List Representation
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

REPRESENTATION OF LINKED LIST IN MEMORY:
A linked list can be implemented using structure and pointers.

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

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.
sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

DIFFERENT IMPLEMENTATION OF LINKED LIST:

We can implement linked list in four ways:
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Circular doubly Linked List

Singly Linked List:

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.
The operations we can perform on singly linked lists are insertion, deletion and traversal.

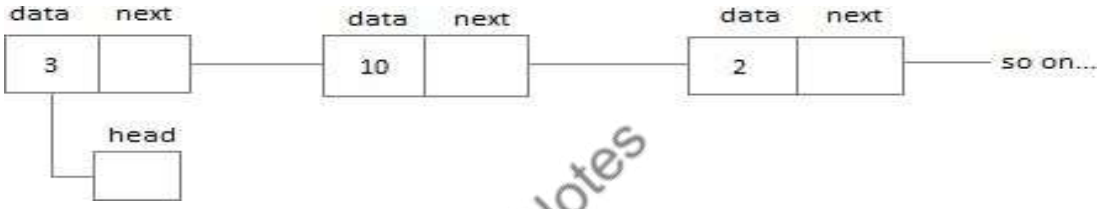


Figure 1.3: Singly Linked List Representation

Doubly Linked List:

In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.

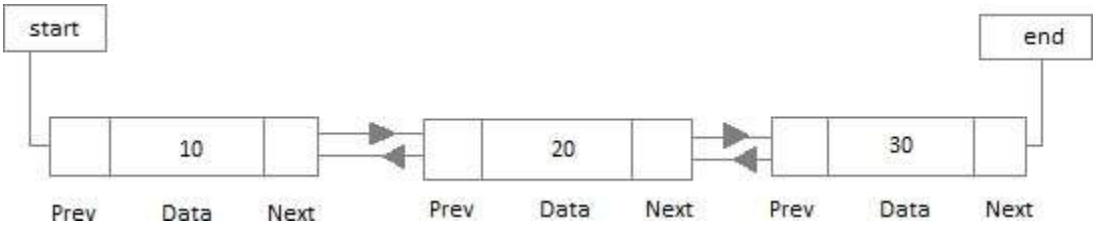


Figure 1.4: Doubly Linked List Representation

CIRCULARLINKEDLIST:

In Circular linked list the last node of the list holds the address of the first node hence forming a circular chain.
Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list.

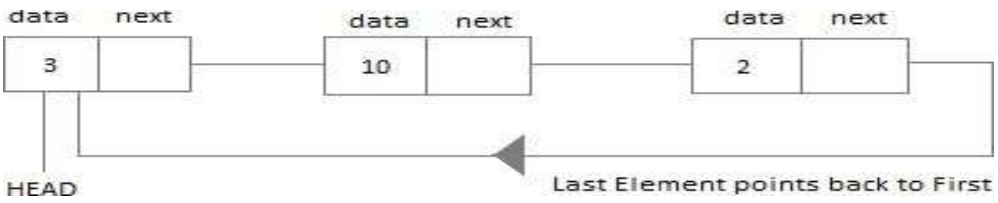


Figure 1.5: Circular Linked List Representation

Circular doubly Linked List:

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.

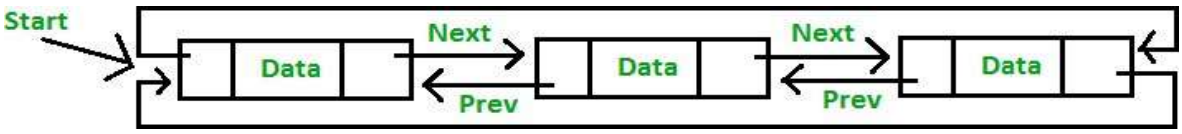


Figure 1.6: Circular Doubly Linked List Representation

APPLICATION OF LINKED LIST:

Polynomial Manipulation:

A polynomial has multiple terms with same information such as coefficient and powers. Each term of a polynomial is treated as a node of a list and normally a linked list used to represent a polynomial. The implementation of polynomial addition is the only operation that is discussed many place. Multiplication of polynomials can be obtained by performing repeated additions.

Each polynomial is stored in decreasing order of by term according to the criteria of that polynomial. i.e. The term whose powers are more are stored at first node and the least power term is stored at last. This ordering of polynomials makes the addition of polynomials easy. In fact two polynomials can be added or multiplied by scanning each of their terms only once.

Linked Dictionary:

An important part of any compiler is the construction and maintenance of a dictionary containing names and their associated values. Such dictionary is also called Symbol Table. There may be several symbols corresponding to variable names, labels, literals, etc.

The constraints, which must be considered in the design of the symbol tables, are processing time and memory space. There are many phases associated with the construction of symbol tables. The main phases are building and referencing.

It is very easy to construct a very fast symbol table system, provided that a large section of memory is available. In such case a unique address is assigned to each name. The most straightforward method of accessing a symbol table is linear search technique. This method involves arranging the symbols sequentially in memory via an array or by using a simple linked list. An insertion can be easily handled by adding new element to the end of the list. When it is desired to access a particular symbol, the table is searched sequentially from its beginning until it is found. It will take n/2 comparisons to find a particular symbol.

Data Structure Lecture Notes

Unit -2

STACKS:

STACKS AS ADT:

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

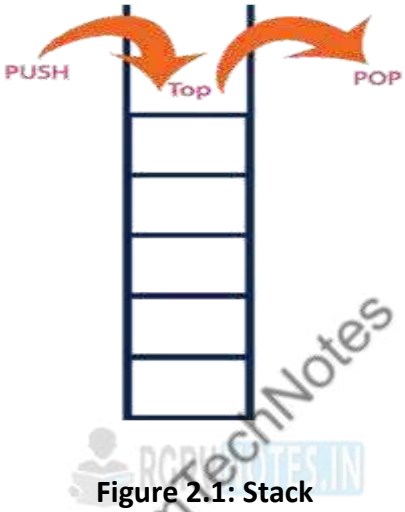


Figure 2.1: Stack

In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function POP.
In the figure 2.1, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

Representation of Stacks

A stack may be represented by means of a one way list or a linear array. Unless, otherwise stated, each of the stacks will be maintained by a linear array STACK; a variable TOP contains the location of the top element of the stack. A variable N gives the maximum number elements that can be held by the stack. The condition where TOP is NULL, indicate that the stack is empty. The condition where TOP is N, will indicate that the stack is full.

DIFFERENT IMPLEMENTATION OF STACK:

Stack data structure can be implementing in two ways. They are as follows...
1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements.

When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Using Array:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array:

1. push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

• Step 1: Check whether stack is FULL. (top == SIZE-1)

• Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

• Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

2. pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

• Step 1: Check whether stack is EMPTY. (top == -1)

• Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.

• Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

3. display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

• Step 1: Check whether stack is EMPTY. (top == -1)

• Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.

• Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

• Step 4: Repeat above step until i value becomes '0'.

MULTIPLE STACKS:

When a stack is created using single array, we cannot able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

In order to maintain 2 stacks, there should be 2 top variables to indicate the top of both the stacks. Both the stacks can grow up to any extent from 1st position to maximum, hence there should be one

more variable to keep track of the total number of values stored. The overflow condition will appear.If count becomes equal to or greater than array size and the underflow condition from empty stack will appear if count=0.
The structure for such implementation can be given as:

```
struct multistack
{
Int top0,top1;
Int count;
Int num;
        };
```

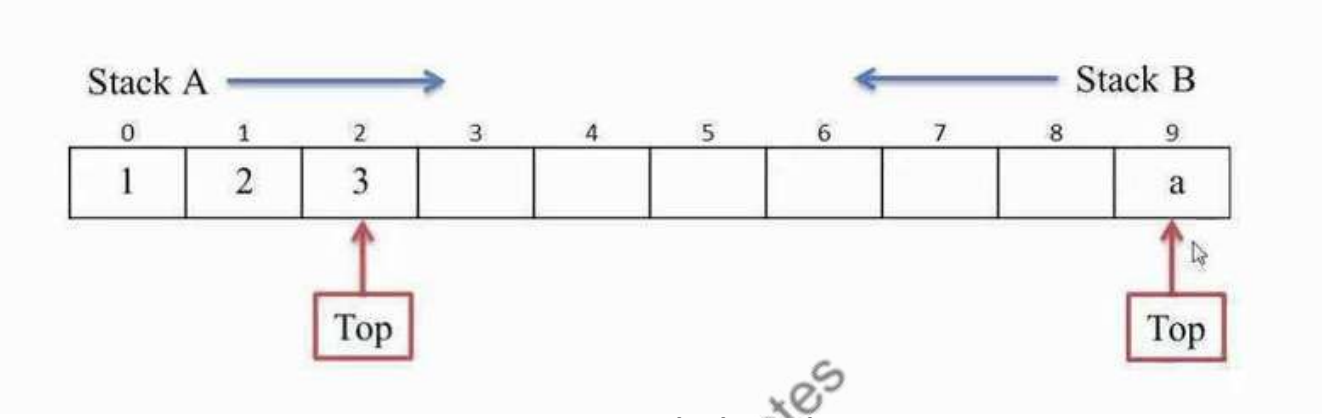


Figure 2.2: Multiple Stacks

APPLICATION OF STACK:

There are two important applications of stacks.

- Recursion
- Reversing a String
- Calculation of Arithmetic Expression

Recursion:

Recursion means function calling itself.
A function is called recursive if the function definition refers to itself or does refer to another function which in turn refers back to the same function. In-order for the definition not to be circular, it must have the following properties:

(i) There must be certain arguments called base values, for which the function does not refer to itself.

(ii) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with those two properties is said to be well defined.

Let us consider the factorial of a number and its algorithm described recursively:

We know that N! = N * (N-1)!

(N-1)! = N-1 * (N-2)! and so on up to 1.
FACT(N)

1. if N=1
return 1
2. else
return N * FACT(N-1)
3. end
Let N be 5.

Then according to the definition FACT(5) will call FACT(4), FACT(4) will call FACT(3), FACT(3) will call FACT(2), FACT(2) will call FACT(1). Then the execution will return back by finishing the execution of FACT(1), then FACT(2) and so on up to FACT(5) as described below.

1) 5! = 5 * 4!
2) 4! = 4 * 3!
3) 3! = 3 * 2!
4) 2! = 2 * 1!
5) 1! = 1
6) 2! = 2 * 1 = 2
7) 3! = 3 * 2 = 6
8) 4! = 4 * 6 = 24
9) 5! = 5 * 24 = 120

Calculation of Arithmetic Expression:

An expression is a collection of operators and operands that represents a specific value. This section deals with the mechanical evaluation or compilation of infix expression. The stack is find to be more efficient to evaluate an infix arithmetical expression by first converting to a suffix or postfix expression and then evaluating the suffix expression. This approach will eliminate the repeated scanning of infix expressions in order to obtain its value.

A normal arithmetic expression is normally called as infix expression. E.g A+B

A Polish mathematician found a way to represent the same expression called polish notation or prefix expression by keeping operators as prefix. E.g +AB

We use the reverse way of the above expression for our evaluation. The representation is called Reverse Polish Notation (RPN) or postfix expression. E.g. AB+

CONVERSION OF INFIX TO POSTFIX NOTATION USING STACK:

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.
- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example:
Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

• Step 1: The Operators in the given Infix Expression : = , + , *
• Step 2: The Order of Operators according to their preference : * , + , =
• Step 3: Now, convert the first operator * ------ D = A + B C *
• Step 4: Convert the next operator + ----- D = A BC* +
• Step 5: Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

EVALUATION OF POSTFIX EXPRESSION:

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

• Read all the symbols one by one from left to right in the given Postfix Expression
• If the reading symbol is operand, then push it on to the Stack.
• If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
• Finally! perform a pop operation and display the popped value as final result.

RECURSION:

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".

A simple example of recursion would be:

```
void recurse()
{
recurse(); /* Function calls itself */
}

int main()
{
recurse(); /* Sets off the recursion */
return 0;
}
```

QUEUE:

A queue is a sequential storage structure that permits access only at the two ends of the sequence. We refer to the ends of the sequence as the front and rear. A queue inserts new elements at the rear and removes elements from the front of the sequence. You will note that a queue removes elements in the same order in which they were stored, and hence a queue provides FIFO (first-in / first-out), or FCFS (first-come / first-served), ordering.

Operations on Queue:

Mainly the following four basic operations are performed on queue:

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

• Step 1: Check whether queue is FULL. (rear == SIZE-1)

• Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

• Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

• Step 1: Check whether queue is EMPTY. (front == rear)

• Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

• Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

• Step 1: Check whether queue is EMPTY. (front == rear)

• Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

• Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

• Step 4: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

Front: Get the front item from queue.

Rear: Get the last item from queue.

Graphical Presentation:

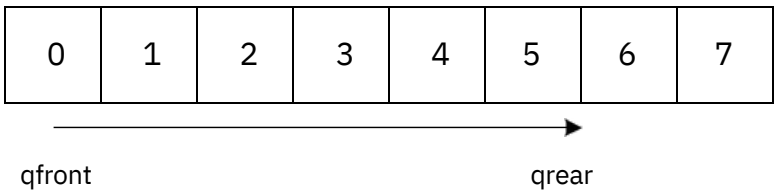


Figure 2.3: Queue

Time Complexity:

Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

QUEUES AS ADT:

In case of Queue we know that what to implement but how to implement is not known, hence queue is called ADT.

DIFFERENT IMPLEMENTATION OF QUEUE:

- Circular Queue.
- Priority Queue.
- Dqueue

CIRCULAR QUEUE:

~~Circular Queue is a~~lso a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Basic features of Circular Queue

In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.

Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.

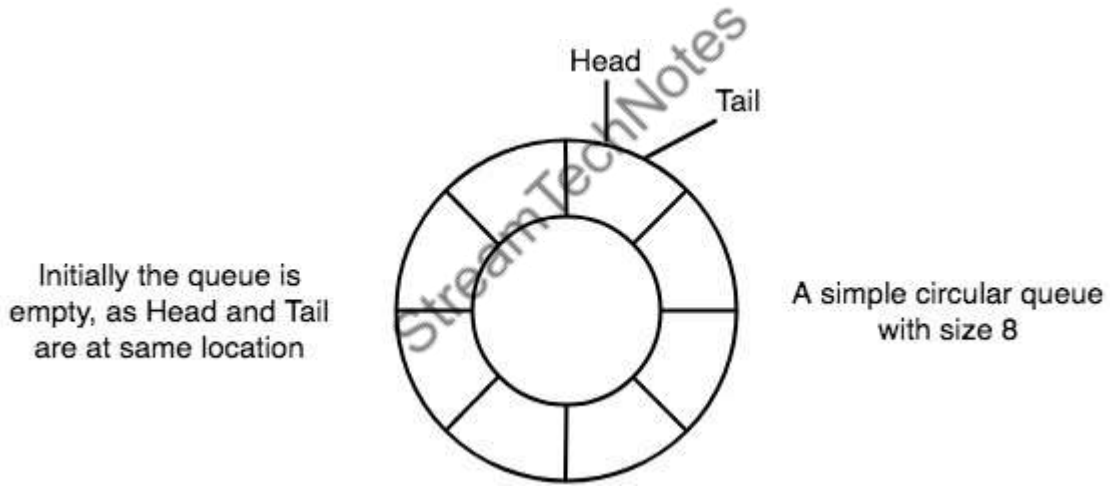


Figure 2.4: Circular Queue

New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.

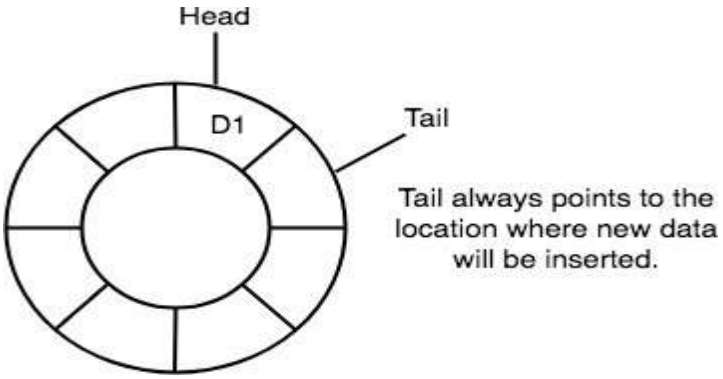


Figure 2.5: Circular Queue Insertion

In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.

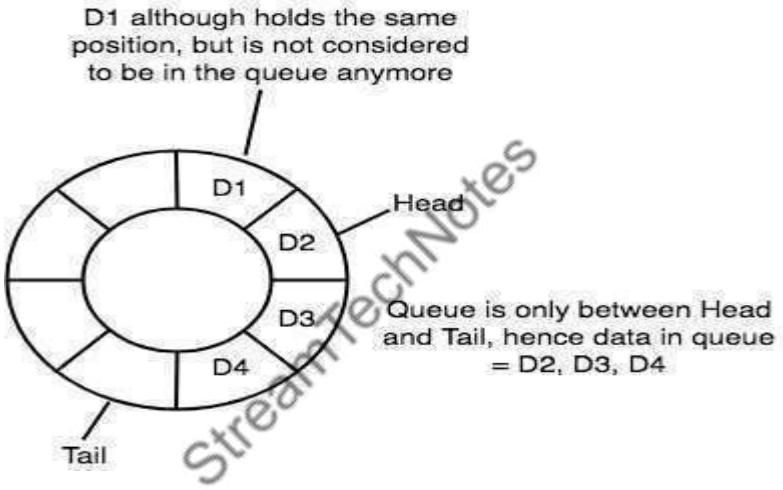


Figure 2.6: Circular Queue Insertion

The head and the tail pointer will get reinitialized to **0** every time they reach the end of the queue.

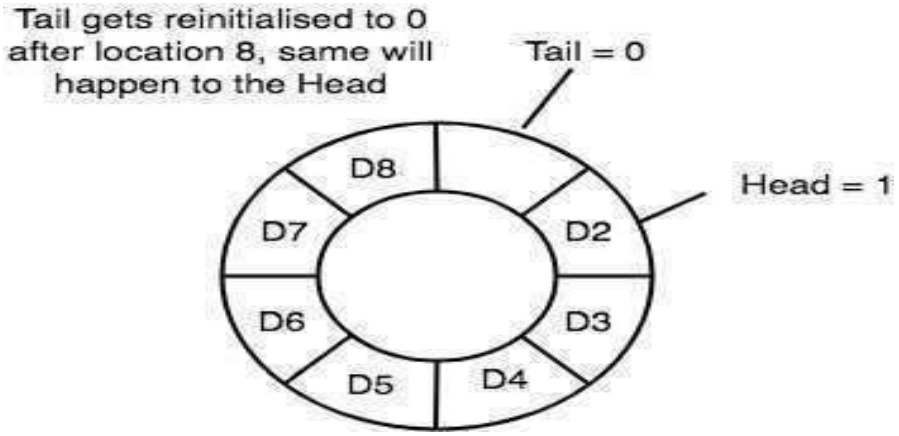


Figure 2.7: Circular Queue Insertion

Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialized upon reaching the end of the queue.

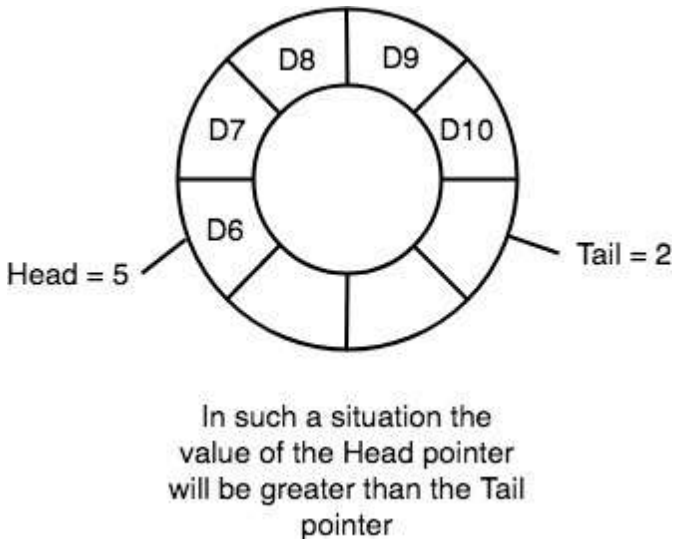


Figure 2.8: Circular Queue deletion

Application of Circular Queue

- Computer controlled Traffic Signal System uses circular queue.
- CPU scheduling and Memory management.

CONCEPT OF DQUEUE:

A **dqueue**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

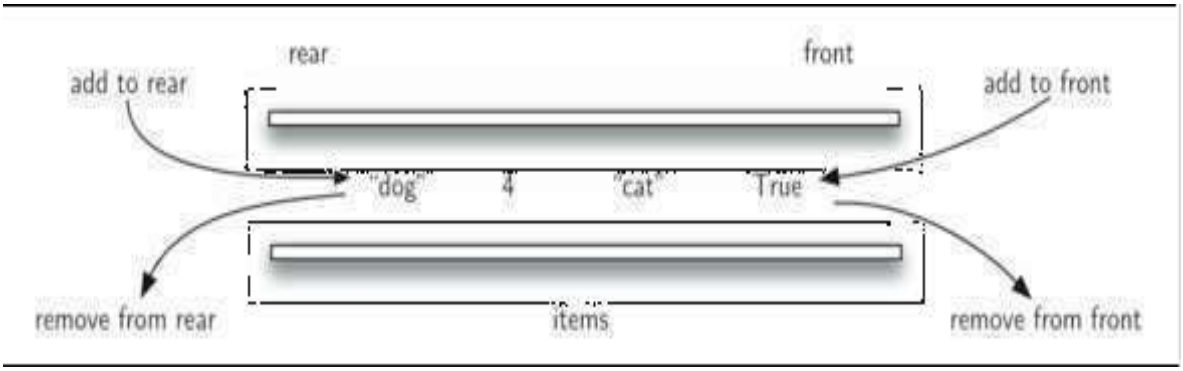


Figure 2.9: Dqueue

Operation on Deque:

- **insertFront():** Adds an item at the front of Deque.
- **insertLast():** Adds an item at the rear of Deque.
- **deleteFront():** Deletes an item from front of Deque.
- **deleteLast():** Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

- **getFront():** Gets the front item from queue.
- **getRear():** Gets the last item from queue.
- **isEmpty():** Checks whether Deque is empty or not.
- **isFull():** Checks whether Deque is full or not.

PRIORITY QUEUE:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- insert / enqueue − add an item to the rear of the queue.
- remove / dequeue − remove an item from the front of the queue.

Priority Queue Representation

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

QUEUE SIMULATION:

Queuing simulation as a method is used to analyze how systems with limited resources distribute those resources to elements waiting to be served, while waiting elements may exhibit discrete variability in demand, i.e. arrival times and require discrete processing time.

Queuing theory based analysis is regularly used for e.g. telecommunications, computer networks, predicting computer performance, traffic, call centres, etc.

APPLICATIONS OF QUEUES:

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n First **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

CLASS NOTES SUBJECT DATA STRUCTURE
UNIT-3 TREE

Tree: Definitions - Height, depth, order, degree etc. Binary Search Tree - Operations, Traversal, Search. AVL Tree, Heap, Applications and comparison of various types of tree; Introduction to forest, multi-way Tree, B tree, B+ tree, B* tree and red-black tree

Tree: Definitions

In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows:-
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.
In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual
data of that particular element and link to next element in hierarchical structure.
In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example

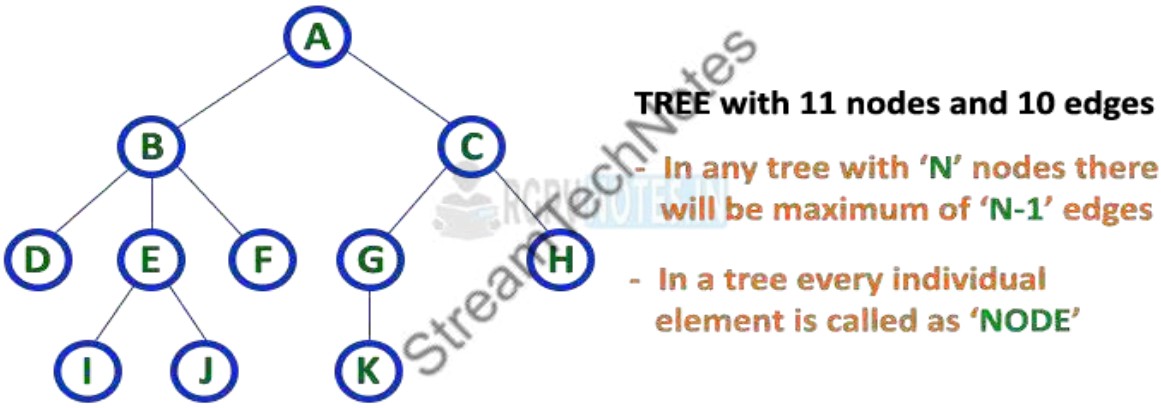


Figure 3.1 Tree

In a tree data structure, we use the following terminology:-

1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

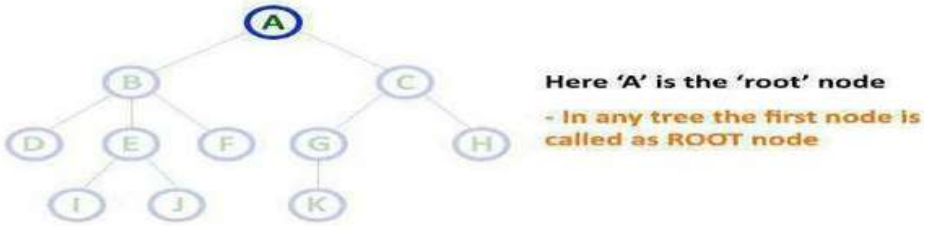


Figure 3.2 Root Node

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

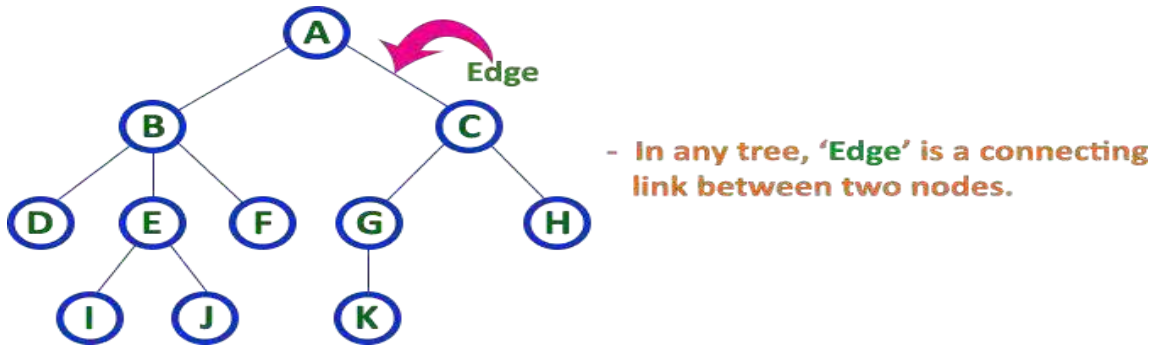


Figure 3.3 Edge

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".

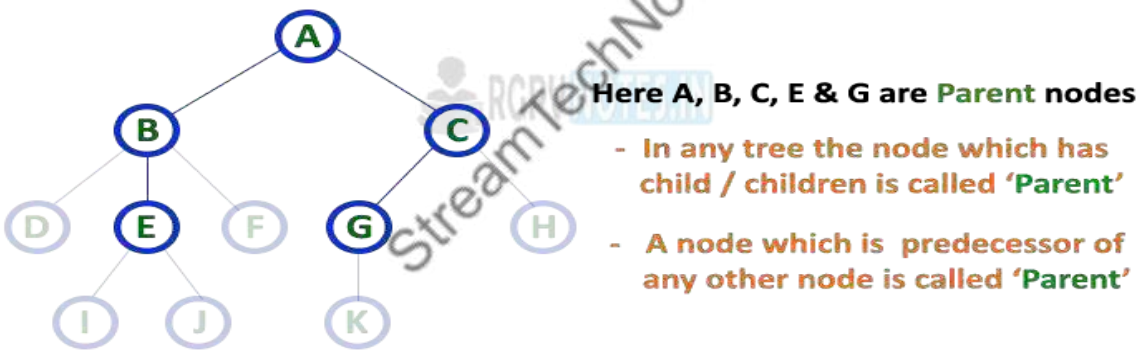


Figure 3.4 Parent Node

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

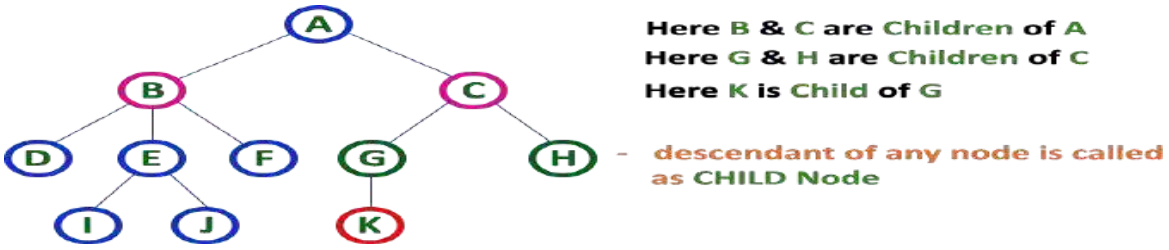


Figure 3.5 Child Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.

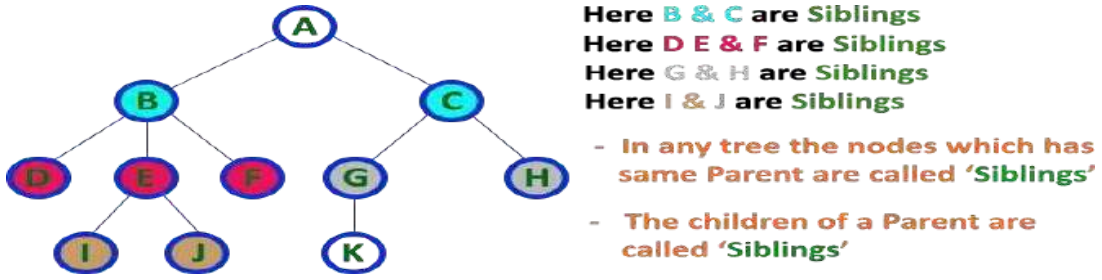


Figure 3.6 Siblings

6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.

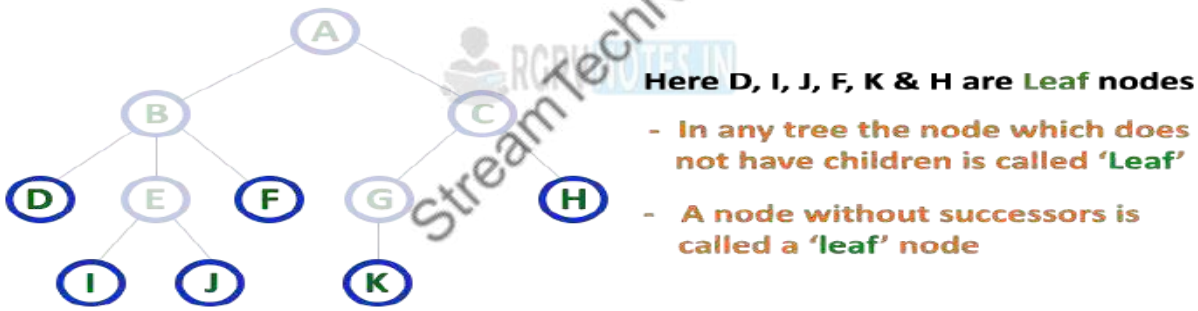


Figure 3.7 Leaf Node

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.In a tree data structure, nodes other than leaf nodes are called
as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

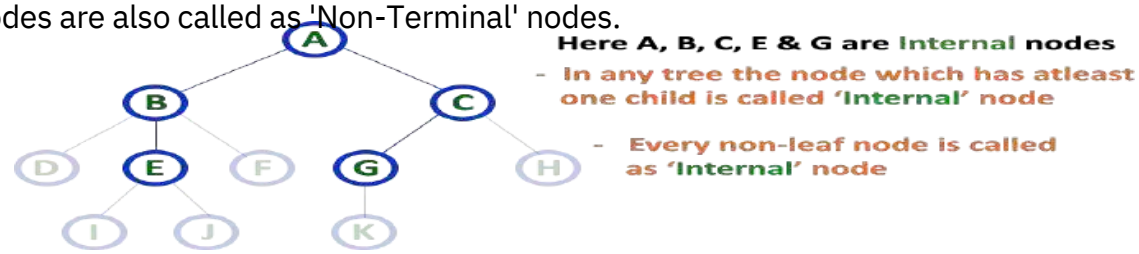


Figure 3.8 Internal Nodes

8. Degree

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nod in a tree is called as 'Degree of Tree'

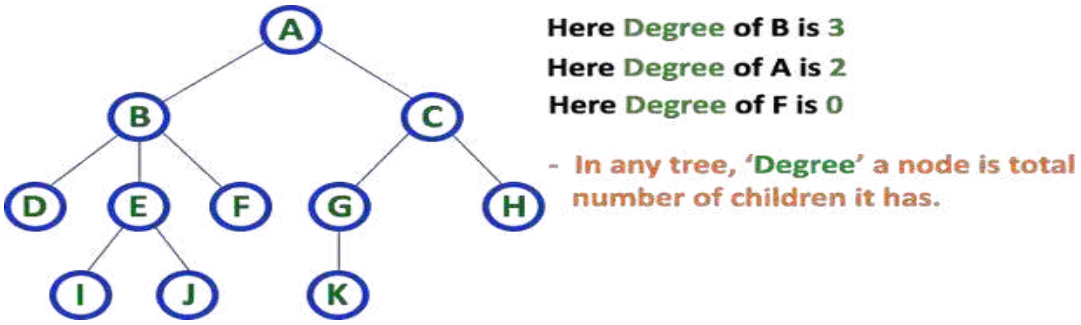


Figure 3.9 Degree

9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and t children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each leve (Step).

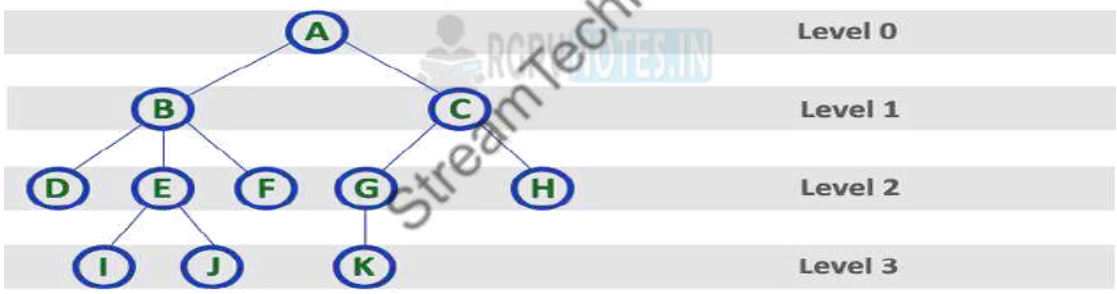


Figure 3.10 Level of Tree

10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

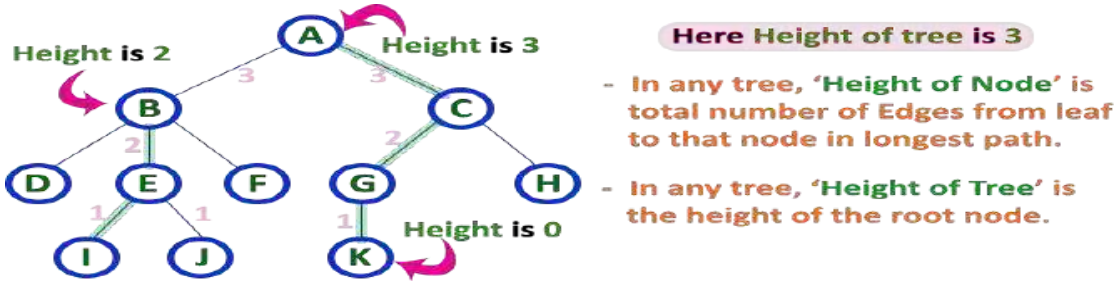


Figure 3.11 Height Of Tree

11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a
tree, depth of the root node is '0'.

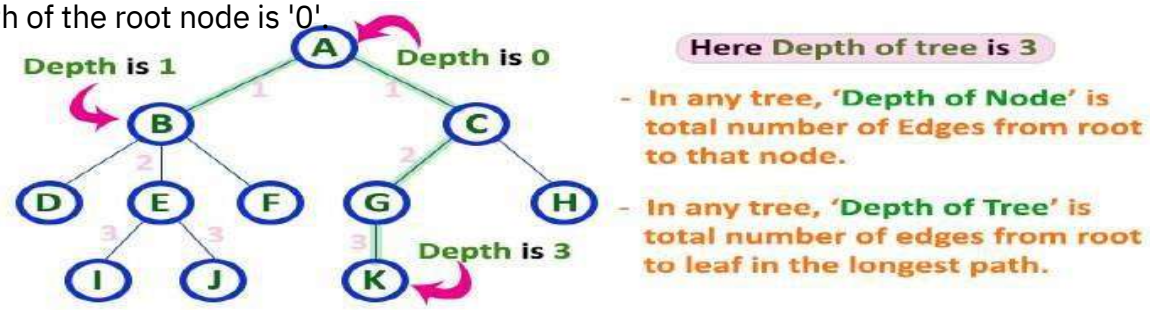


Figure 3.12 Depth Of Tree

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.

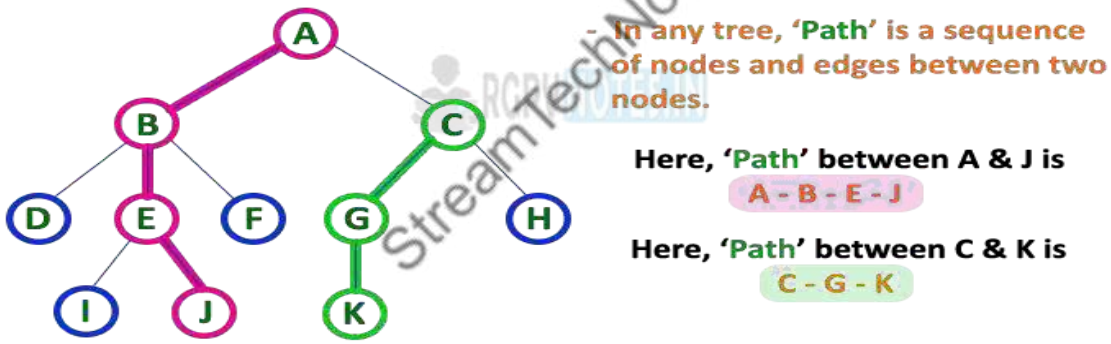


Figure 3.13 Path

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

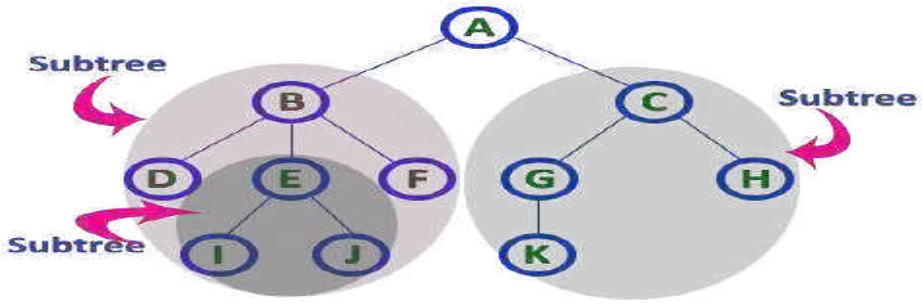


Figure 3.14 Sub Tree

Binary Search Tree - Operations, Search

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right. To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows:-
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with

smaller
values and right subtree of every node contains larger values.

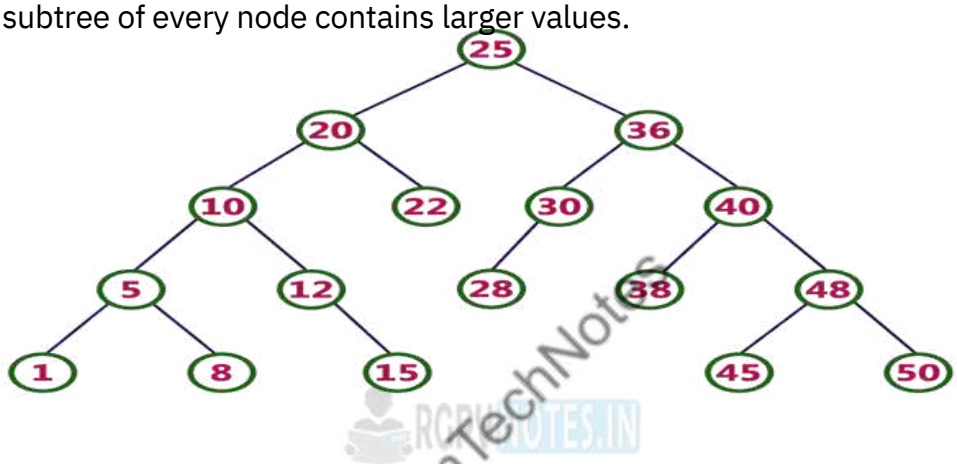


Figure 3.15 Binary Search Tree

The following operations are performed on a binary search tree:-1) Search 2) Insertion 3) Deletion

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows:-

➢
Step 1: Read the search element from the user
➢
Step 2: Compare, the search element with the value of root node in the tree.
➢
Step 3: If both are matching, then display "Given node found!!!" and terminate the function
➢
Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.
➢
Step 5: If search element is smaller, then continue the search process in left subtree.
➢
Step 6: If search element is larger, then continue the search process in right subtree.
➢
Step 7: Repeat the same until we found exact element or we completed with a leaf node

➢
Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
➢
Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows:-
Step 1: Create a newNode with given value and set its left and right to NULL.
Step 2: Check whether tree is Empty.
Step 3: If the tree is Empty, then set set root to newNode.
Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases:-

Case 1: Deleting a leaf node
We use the following steps to delete a leaf node from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child
We use the following steps to delete a node with one child from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: If it has only one child, then create a link between its parent and child nodes. Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children
We use the following steps to delete a node with two children from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3: Swap both deleting node and node which found in above step.
Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2
Step 5: If it comes to case 1, then delete using case 1 logic.
Step 6: If it comes to case 2, then delete using case 2 logic.
Step 7: Repeat the same process until node is deleted from the tree.

Binary Search Tree – Traversal

There are three types of binary search tree traversals.
1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order

traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtre of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows:-An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.

There are four rotations and they are classified into two types.

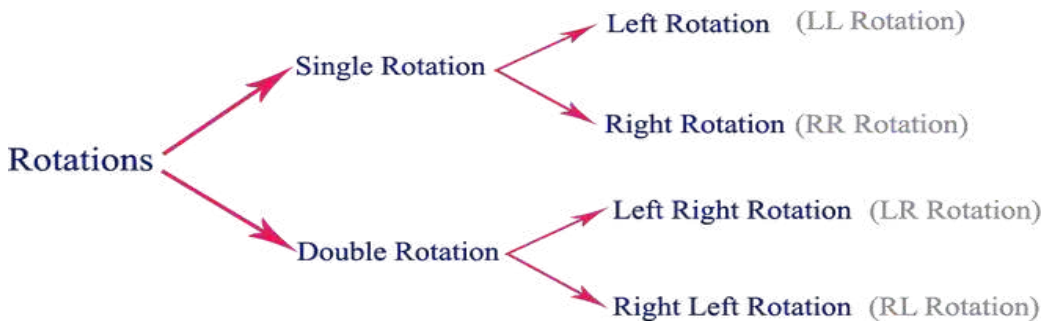


Figure 3.16 Rotation AVL Tree

The following operations are performed on an AVL tree:-

1. Search
2. Insertion
3. Deletion

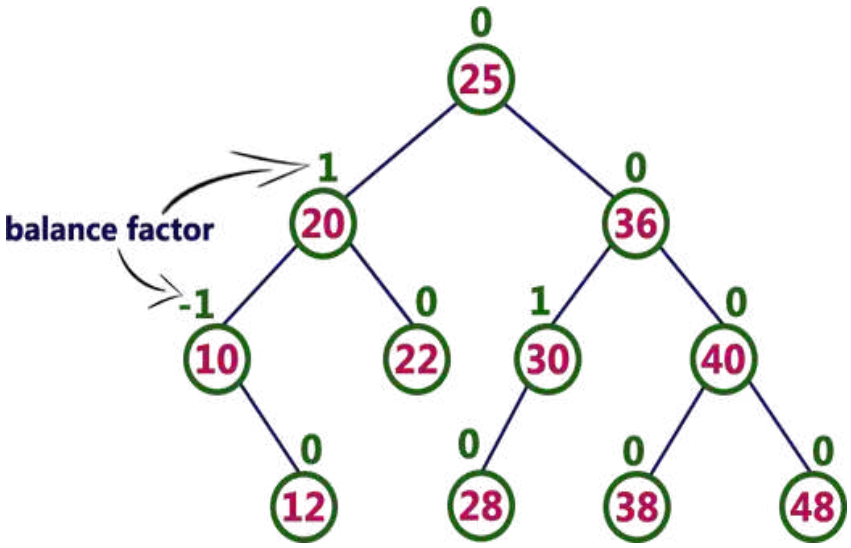


Figure 3.17 AVL Tree

The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Heap

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on thier value. A heap data structure, som time called as Binary Heap.

There are two types of heap data structures and they are as follows...

1. Max Heap
2. Min Heap

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min hea

Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

Max Heap

Max heap data structure is a specialized full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value.

Max heap is defined as follows:- Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes. And last leaf node can be alone.

Min-Heap – Where the value of the root node is less than or equal to either of its children.

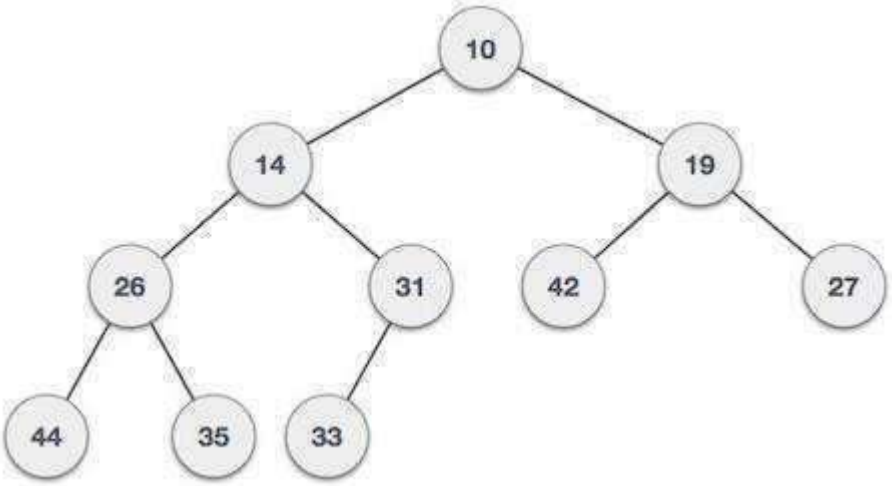


Figure 3.18 Min Heap

Max-Heap – Where the value of the root node is greater than or equal to either of its children.

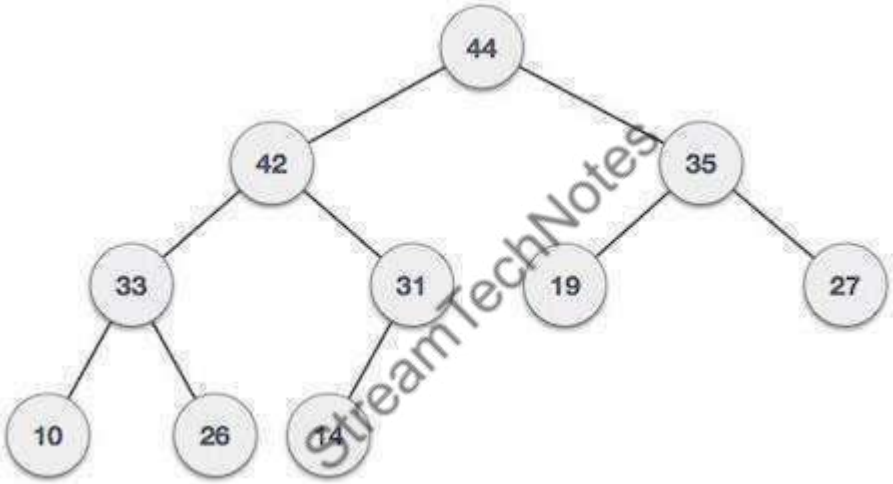


Figure 3.19 Max Heap

Heap application

Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in O(logn) time which is a O(n) operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.

Comparison of various types of tree

There are different types of binary trees and they are

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly

Binary Tree can be defined as follows:- A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

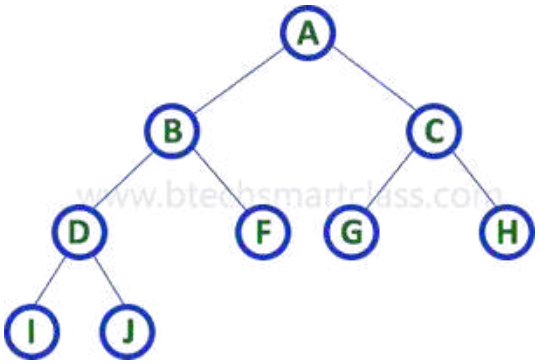


Figure 3.20 Strictly Binary Tree

2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level 2 there must be 22 = 4 nodes and at level 3 there must be 23 = 8 nodes. A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.
Complete binary tree is also called as Perfect Binary Tree

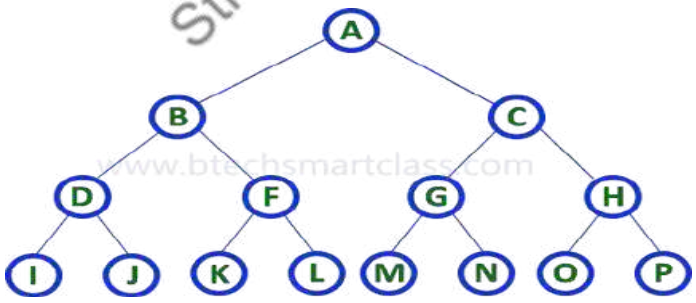


Figure 3.21 Complete Binary Tree

3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required. The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

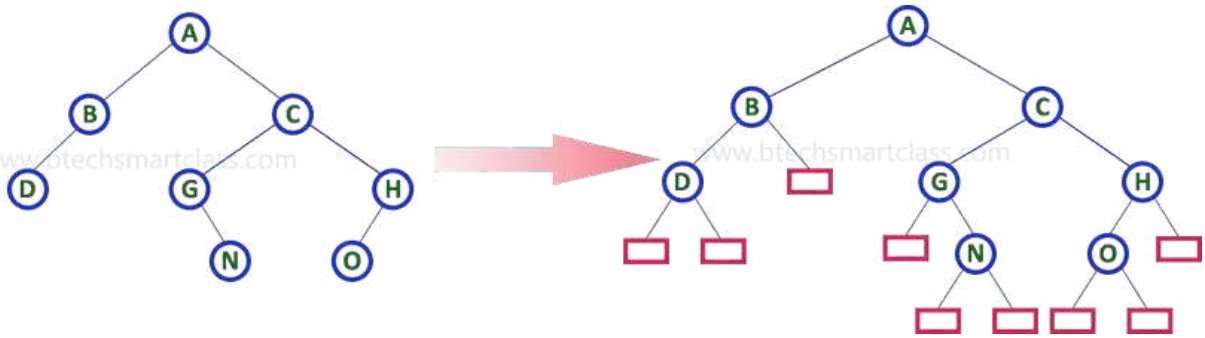


Figure 3.22 Extended Binary Tree

In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Introduction to Forest

Forest is a collection of disjoint trees. In other words, we can also say that forest is a collection of an acyclic graph which is not connected. Here is a pictorial representation of a forest.

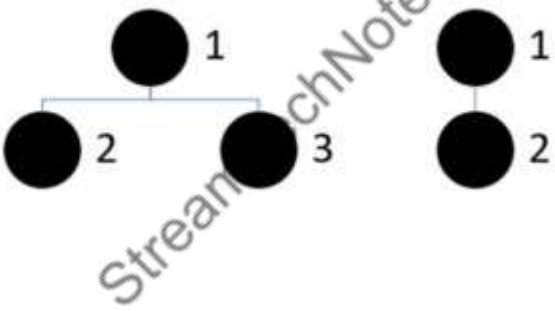


Figure 3.23 Forest

Here you can see that there is no connected tree in the example. A single tree and empty graph is also an example of forest data structure.

Uses of Forest Data Structure

Social Networking Websites

Social networking sites (Such as Facebook, LinkedIn, Twitter etc) are using tree and graphdata structure for their data representation. When you work on adding two people as a friend, you are creating a forest of two people.

Big Data and Web Scrapers

Websites are organized in the form of a tree data structure where the main page forms the root node and the subsequent hyperlinks from that page represents the rest of the tree. When Web scrapers scrap multiple such websites, they represent it in the form of a forest of several trees.

Operating System Storage

If you are working on a Windows-based operating system, you would be able to see various disks in the system
such as C drive (C:\), D drive (D:\), etc. You can think of each drive as different trees and the collection of all
storage as the forest.

Introduction to B & B * tree

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows... B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

B-Tree of Order m has the following properties...

- Property #1 - All the leaf nodes must be at same level.
- Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
- Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
- Property #4 - If the root node is a non leaf node, then it must have at least 2 children.
- Property #5 - A non leaf node with n-1 keys must have n number of children.
- Property #6 - All the key values within a node must be in Ascending Order.

Construct a B-Tree of Order 5 by inserting following numbers

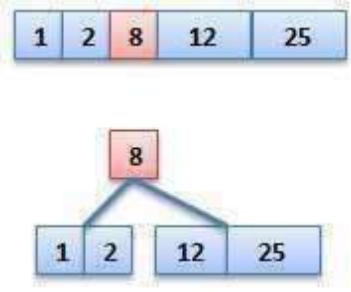
1,12,8,2,25,6,14,28,17,7,52,16,48,68,3,26,29,53,55,45,6

7.

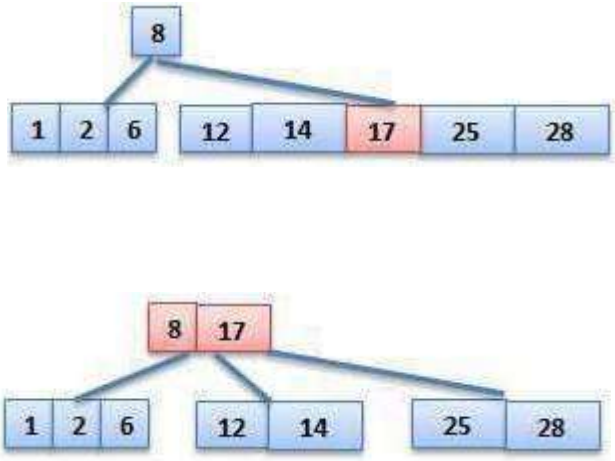
1) Insert 1

2) insert 12

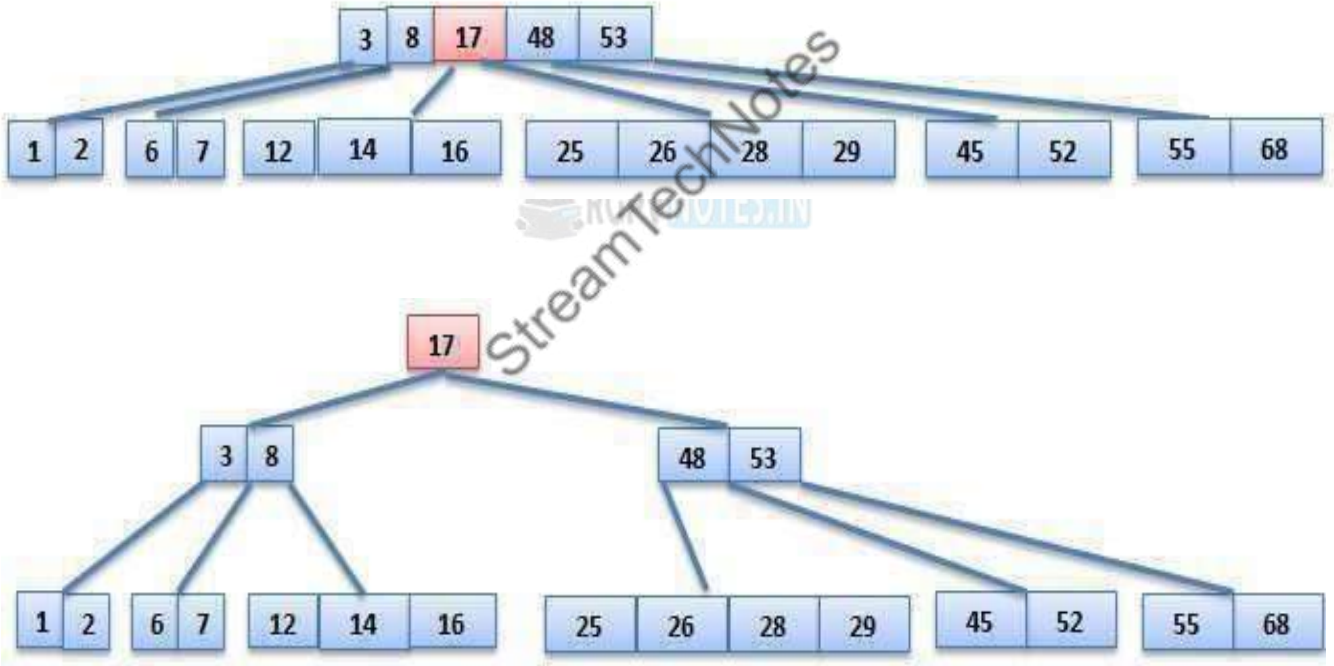
3) insert 8,2 and 25



4) insert 6,14,28,17



5) insert 7,52,16,48,68,3,26,29,53,55,45.



Introduction to red-black tree

Red - Black Tree is another varient of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows:- Red Black Tree is a Binary Search Tree in which every node is colored eigther RED or BLACK.

In a Red Black Tree the color of a node is decided based on the Red Black Tree properties. Every Red Black Tree has the following properties.

Properties of Red Black Tree

- Property #1: Red - Black Tree must be a Binary Search Tree.
- Property #2: The ROOT node must colored BLACK.
- Property #3: The children of Red colored node must colored BLACK. (There should not be two consecutive RED nodes).
- Property #4: In all the paths of the tree there must be same number of BLACK colored nodes.
- Property #5: Every new node must inserted with RED color.
- Property #6: Every leaf (e.i. NULL node) must colored BLACK

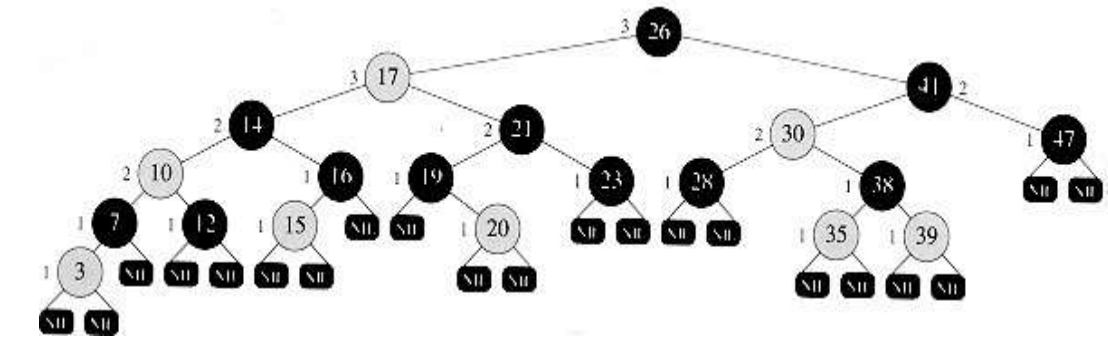


Figure 3.24 Red Black Tree

Introduction to B+ tree

A B+ tree is a data structure often used in the implementation of database indexes. Each node of the tree contains an ordered list of keys and pointers to lower level nodes in the tree. These pointers can be thought of being between each of the keys. To search for or insert an element into the tree, one loads up the root node, finds the adjacent keys that the searched-for value is between, and follows the corresponding pointer to the n node in the tree. Recursing eventually leads to the desired value or the conclusion that the value is not present

B+ trees use some clever balancing techniques to make sure that all of the leaves are always on the same leve of the tree, that each node is always at least half full (rounded) of keys, and (therefore) that the height of the tr is always at most ceiling(log(k)) of base ceiling(n/2) where k is the number of values in the tree and n is the maximum number of pointers (= maximum number of nodes + 1) in each block. This means that only a small number of pointer traversals is necessary to search for a value if the number of keys in a node is large. This is crucial in a database because the B+ tree is on disk.

Properties of a B+ Tree

The root node points to at least two nodes.

1. All non-root nodes are at least half full.
2. For a tree of order m, all internal nodes have m-1 keys and m pointers.
3. A B+-Tree grows upwards.
4. A B+-Tree is balanced.
5. Sibling pointers allow sequential searching.

Introduction to multi-way Tree

A multiway tree is a tree that can have more than two children
A multiway tree of order m88 (or an **m-way tree) is one in which a tree can have m children.
An m-way search tree is a m-way tree in which:

- Each node has m children and m-1 key fields
- The keys in each node are in ascending order.
- The keys in the first i children are smaller than the ithith key
- The keys in the last m-i children are larger than the ithith key
- In a binary search tree, m=2. So it has one value and two sub trees.
- The figure above is a m-way search tree of order 3.
- M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update.
- However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance.
- In m-way search tree, each sub-tree is also a m-way search tree and follows the same rules.
- An extension of a multiway search tree of order m is a B-tree of order m.
- This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

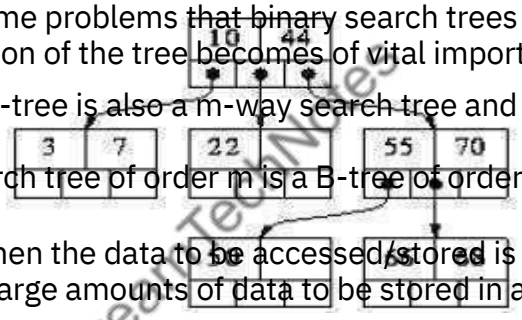


Figure 3.25 Multi Way Tree

Unit 4

Syllabus:

Graphs: Introduction, Classification of graph: Directed and Undirected graphs, etc, Representation, Graph Traversal: Depth First Search (DFS), Breadth First Search (BFS), Graph algorithm: Minimum Spanning Tree (MST)- Kruskal, Prim's algorithms. Dijkstra's shortest path algorithm; Comparison between different graph algorithms, Application of graphs.

Graphs:

A graph is a mathematical abstraction used to represent "connectivity information". A graph consists of vertices and edges that connect them, e.g.
A graph G = (V, E) is:
a set of vertices V and a set of edges E = { (u, v): u and v are vertices }.

Two types of graphs:

- Undirected graphs: the edges have no direction.
- Directed graphs: the edges have direction.

Classification of graph:

Undirected graphs:

Graph is a data structure that consists of following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge.
The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.
Following is an example of an undirected graph with 5 vertices.

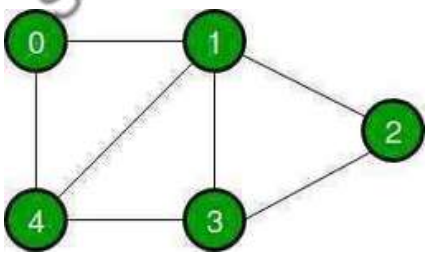


Figure 4.1 Undirected Graph

Directed graphs:

A **directed graph** is a Data Structure containing a vertex set V and an arc set A, where each arc (or edge, or link) is an ordered pair of vertices (or nodes, or sommets). The arcs may be thought of as arrows, each one starting at one vertex and pointing at precisely one other.

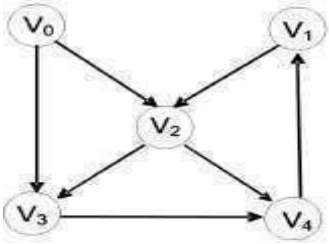


Figure 4.2 Directed Graph

Graph Traversal:
Depth First Search (DFS):

Depth First Search (DFS) algorithm traverses a graph in a deathward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

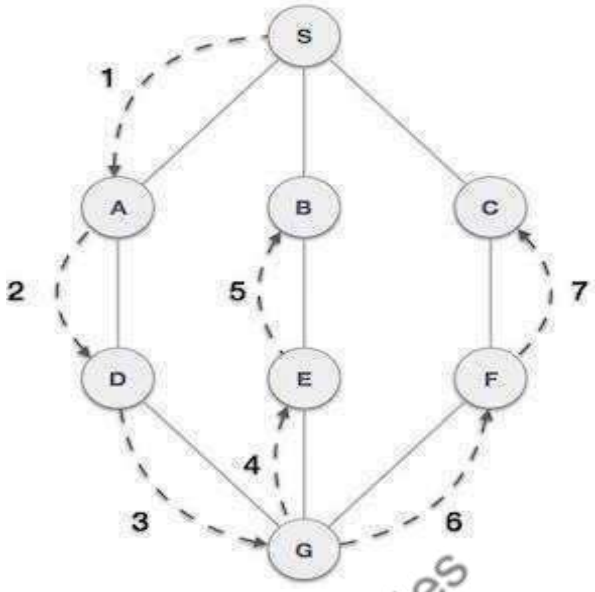
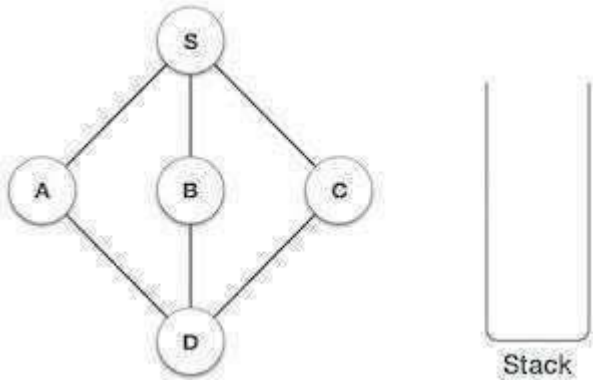
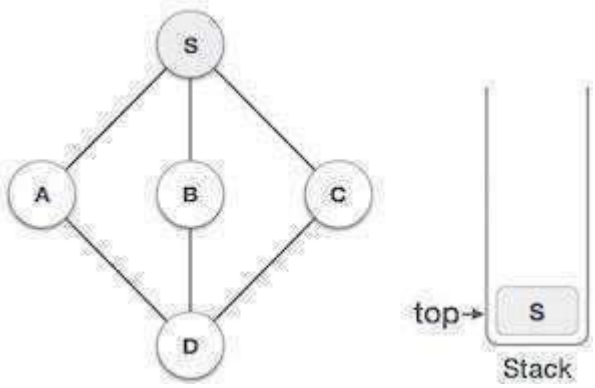
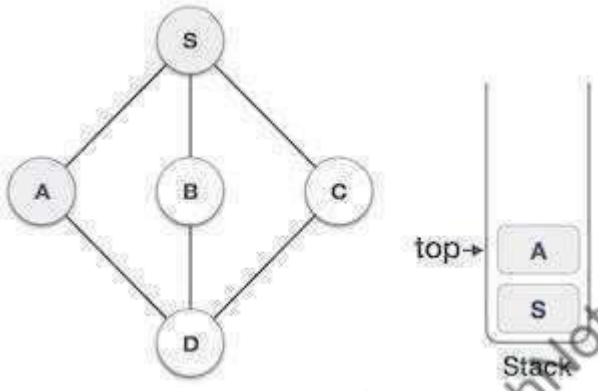
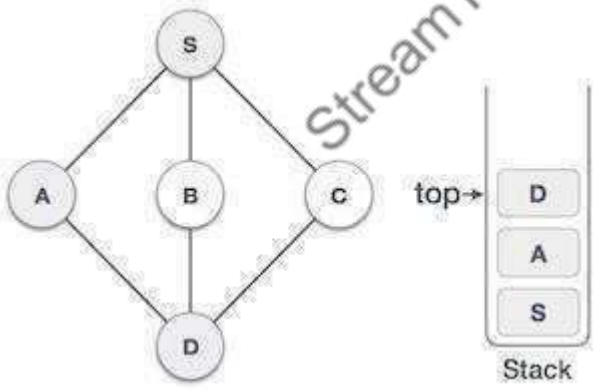


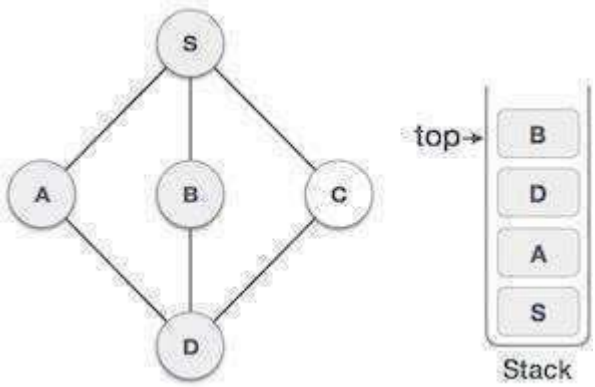
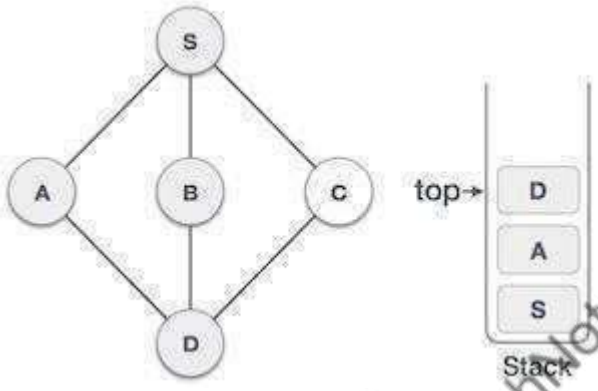
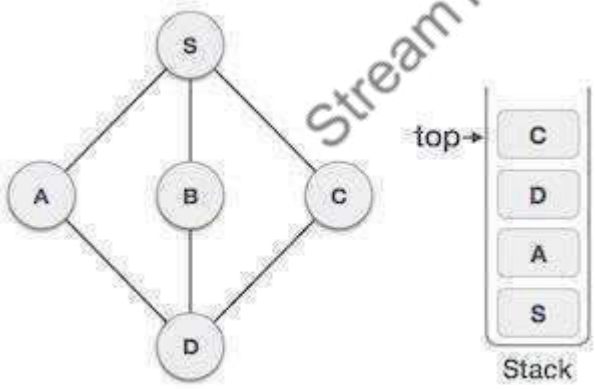
Figure 4.3: DFS Example

As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5		We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Breadth First Search (BFS):

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

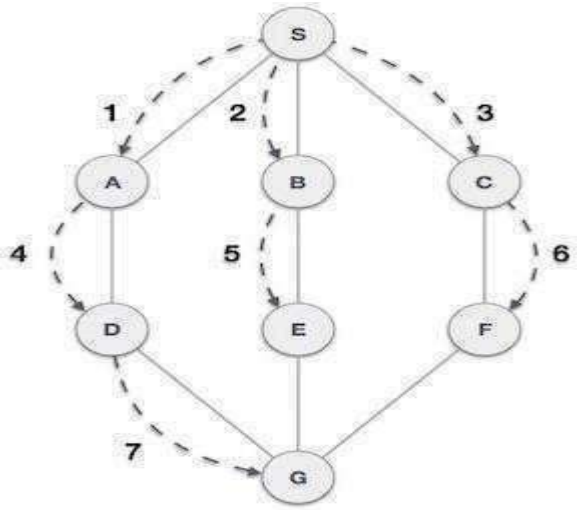
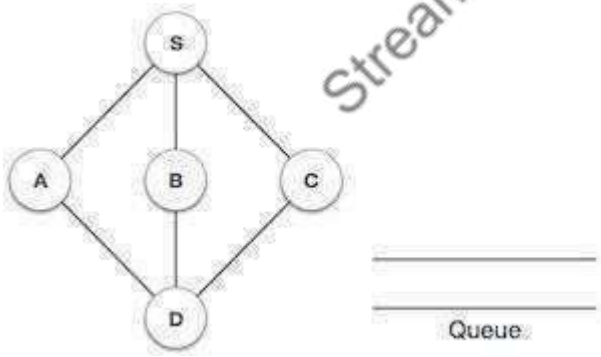
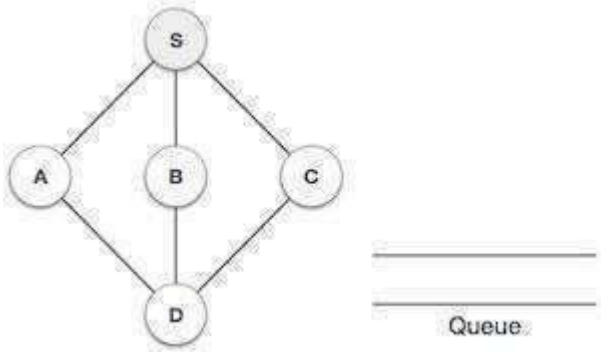
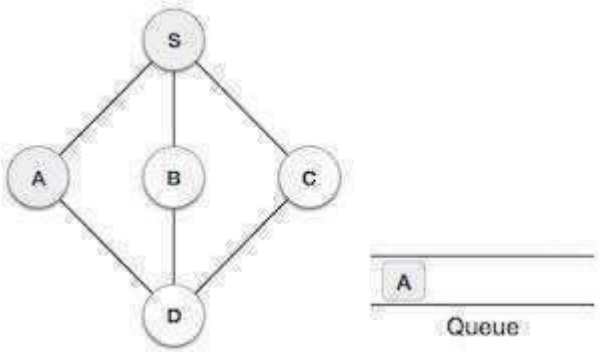
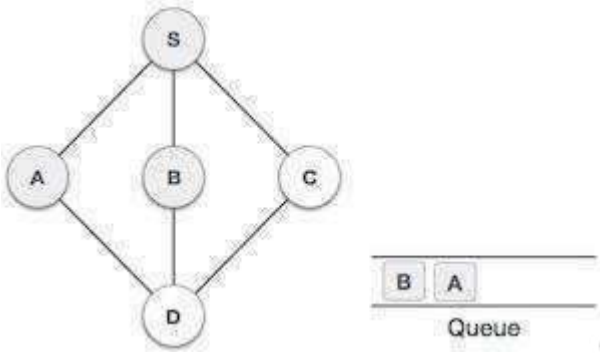
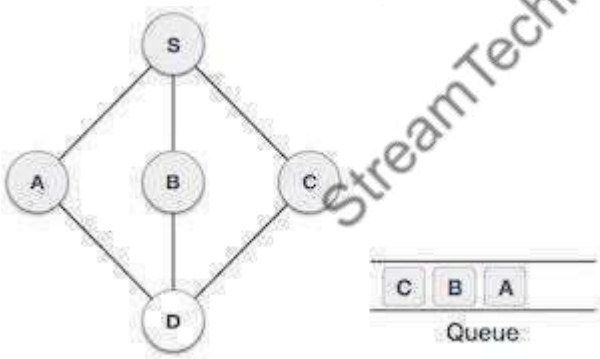
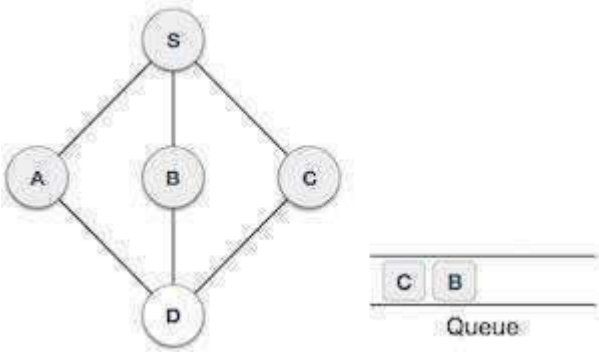


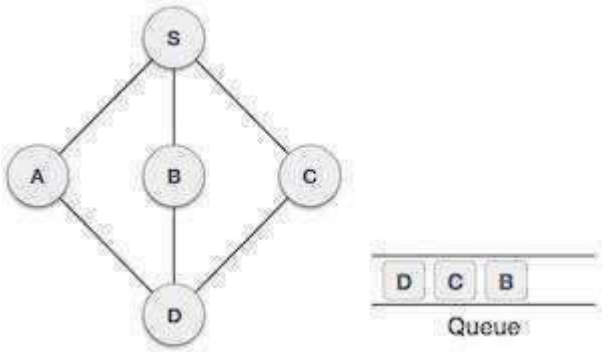
Figure 4.4: BFS Example

As in the example given above, BFS algorithmtraversesfromAtoBtoE to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S**(starting node), and mark it as visited.

3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

Graph algorithm:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

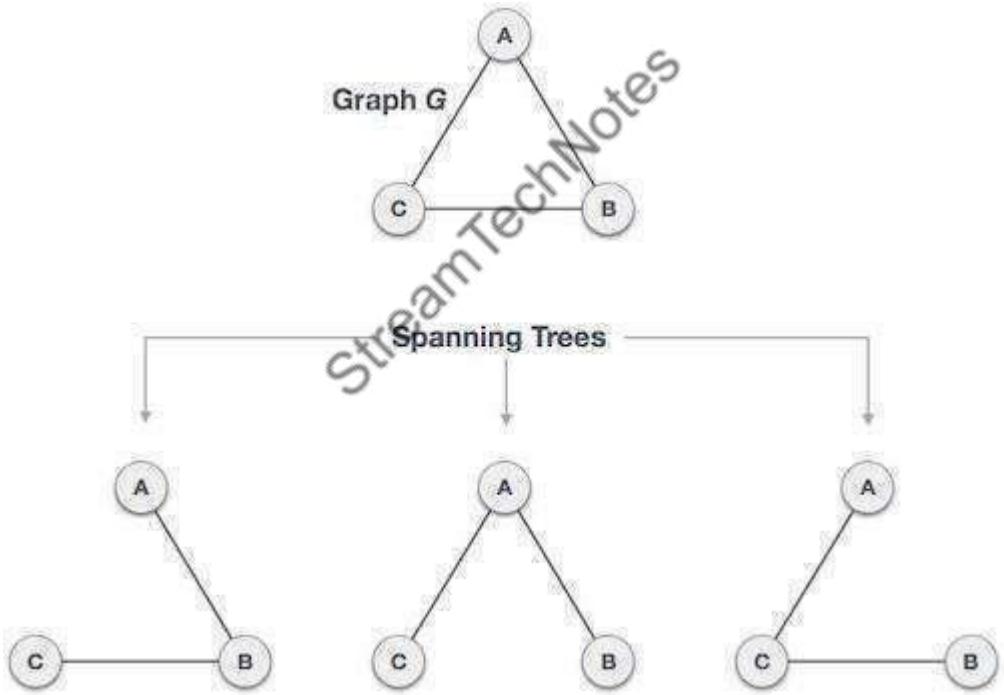


Figure 4.5: Graph and Spanning Trees

We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

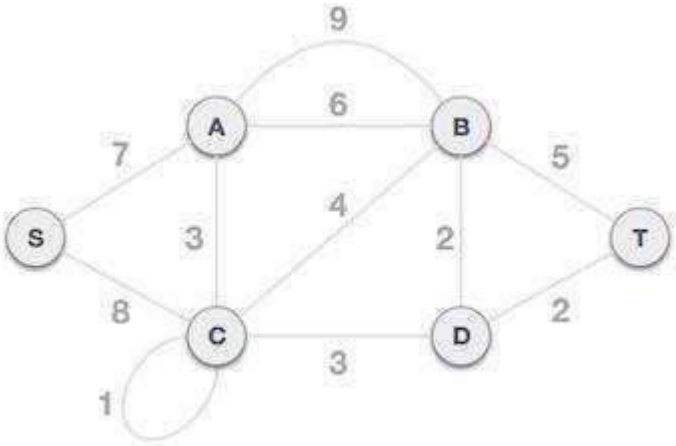
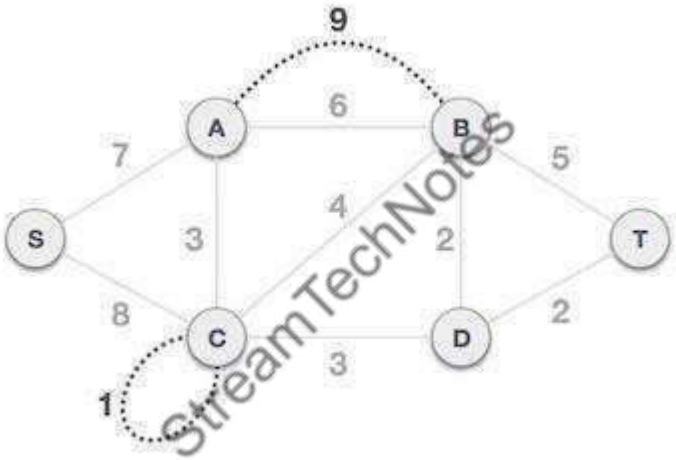
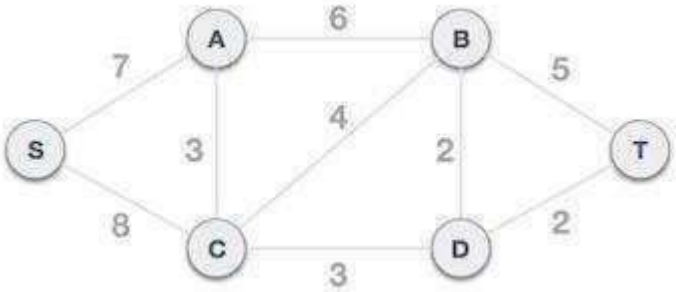


Figure 4.6: Kruskal's Algorithm Example

Step 1 - Remove all loops and Parallel Edges
Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.

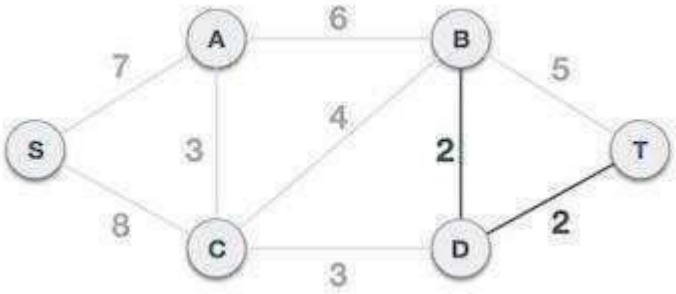


Step 2 - Arrange all edges in their increasing order of weight
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

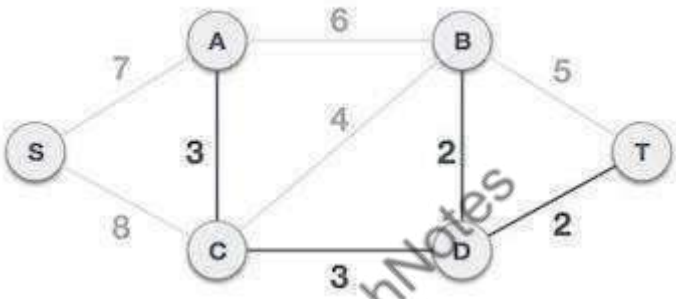
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

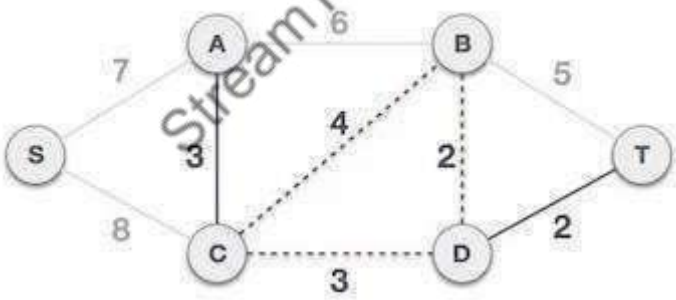
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



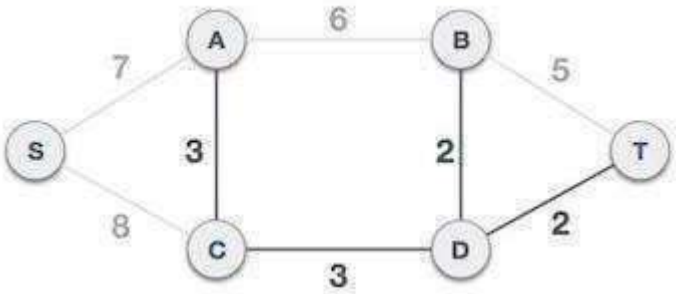
The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
Next cost is 3, and associated edges are A,C and C,D. We add them again −



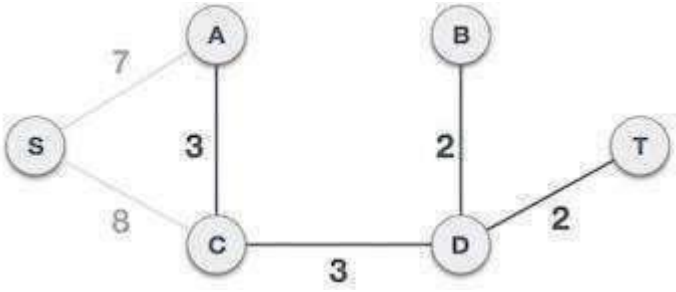
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



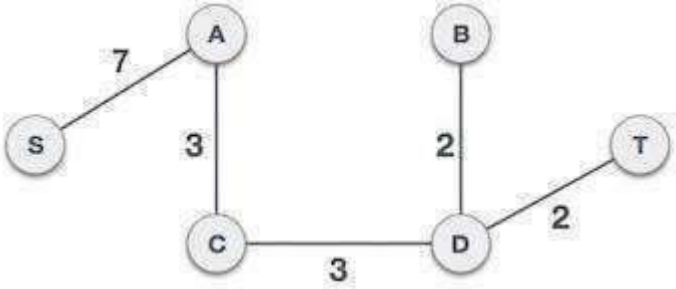
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Algorithm:

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −

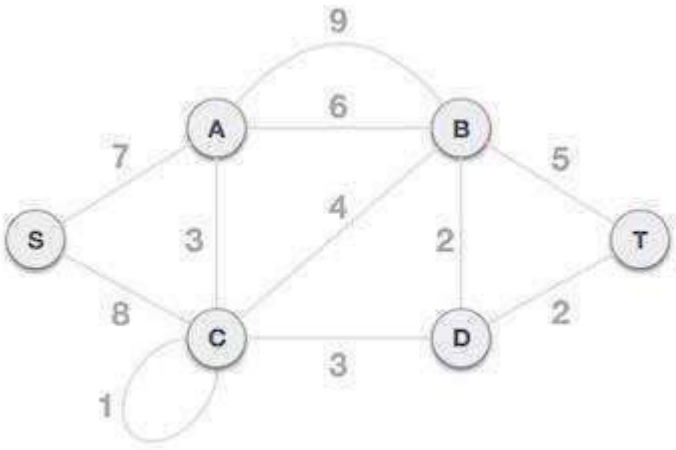
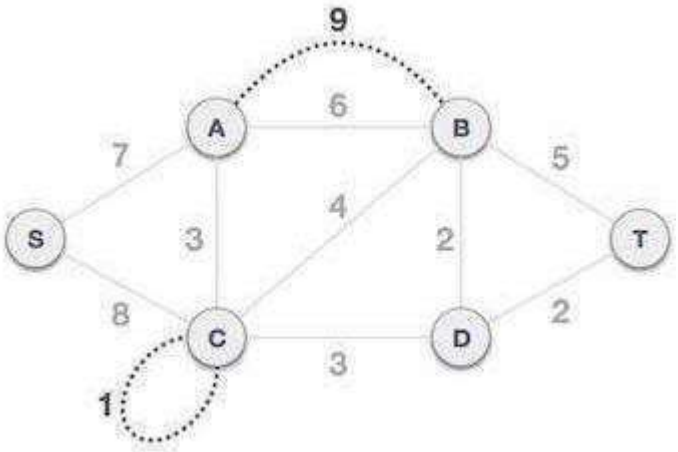
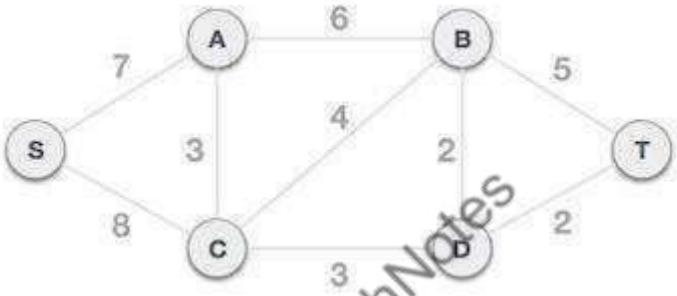


Figure 4.6: Prim's Algorithm Example

Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

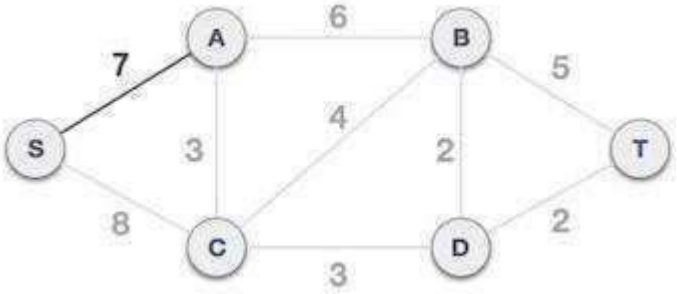


Step 2 - Choose any arbitrary node as root node

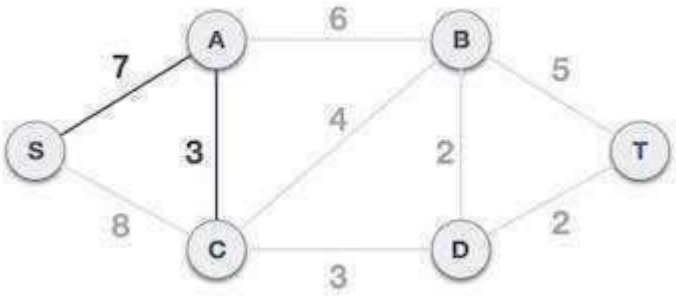
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

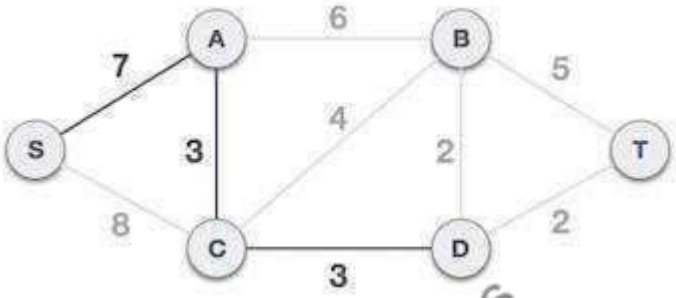
After choosing the root node **S**, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



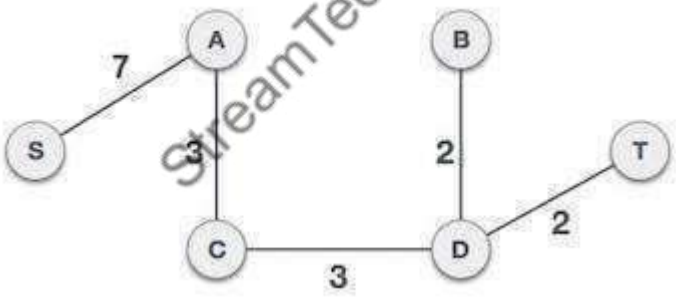
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Dijkstra's Shortest Path Algorithm:

Dijkstra's algorithm can be used to determine the shortest path from one node in a graph to every other node within the same graph data structure, provided that the nodes are reachable from the starting node. This algorithm will continue to run until all of the reachable vertices in a graph have been visited, which means that we could run Dijkstra's algorithm, find the shortest path between any two reachable nodes, and then save the results somewhere. Once we run Dijkstra's algorithm just *once*, we can look up our results from our algorithm again and again — without having to actually run the algorithm itself.

The abstracted rules to solve the algorithm are as follows:

1. Every time that we set out to visit a new node, we will choose the node with the smallest known distance/cost to visit first.
2. Once we've moved to the node we're going to visit, we will check each of its neighboring nodes.
3. For each neighboring node, we'll calculate the distance/cost for the neighboring nodes by summing the cost of the edges that lead to the node we're checking from the starting vertex.

4. Finally, if the distance/cost to a node is *less than* a known distance, we'll update the shortest distance that we have on file for that vertex.

Example: Find the shortest path in the following multistage graph-

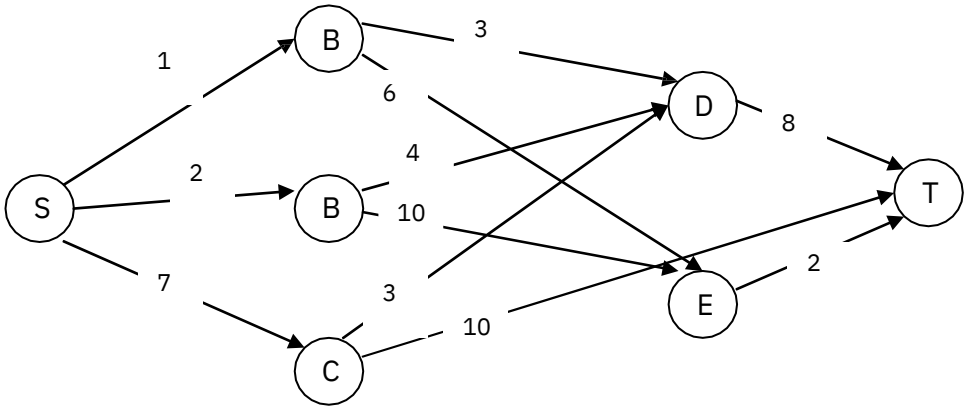


Figure 4.7: Prim's Algorithm Example

There is a single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

d(S, T)=min {1+d(A, T),2+d(B,T),7+d(C,T)} …(1)

We will compute d(A,T), d(B,T) and d(C,T).

d(A,T)=min{3+d(D,T),6+d(E,T)} …(2)

d(B,T)=min{4+d(D,T),10+d(E,T)} …(3)

d(C,T)=min{3+d(E,T),d(C,T)} …(4)

Now let us compute d(D,T) and d(E,T).

d(D,T)=8

d(E,T)=2 backward vertex=E

Let us put these values in equations (2), (3) and (4)

d(A,T)=min{3+8, 6+2}

d(A,T)=8 A-E-T

d(B,T)=min{4+8,10+2}

d{B,T}=12 A-D-T

d(C,T)=min(3+2,10)

d(C,T)=5 C-E-T

d(S,T)=min{1+d(A,T), 2+d(B,T), 7+d(C,T)}

=min{1+8, 2+12,7+5}

=min{9,14,12}

d(S,T)=9 S-A-E-T

The path with minimum cost is S-A-E-T with the cost 9.

Forward approach

d(S,A)=1

d(S,B)=2

d(S,C)=7

d(S,D)=min{1+d(A,D),2+d(B,D)}

=min{1+3,2+4}

d(S,D)=4

d(S,E)=min{1+d(A,E), 2+d(B,E),7+d(C,E)}

=min {1+6,2+10,7+3}

=min {7,12,10}

d(S,E)=7 i.e. Path S-A-E is chosen.

d(S,T)=min{d(S,D)+d(D,T),d(S,E),d(E,T),d(S,C)+d(C,T)}

=min {4+8,7+2,7+10}

d(S,T)=9 i.e. Path S-E, E-T is chosen.

The minimum cost=9 with the path S-A-E-T.

Using dynamic approach programming strategy, the multistage graph problem is solved. This is because in multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.

Comparison between different graph algorithms:

The first distinction is that Dijkstra's algorithm solves a different problem than Kruskal and Prim. Dijkstra solves the shortest path problem (from a specified node), while Kruskal and Prim finds a minimum-cost spanning tree.

For any graph, a spanning tree is a collection of edges sufficient to provide exactly one path between every pair of vertices. This restriction means that there can be no circuits formed by the chosen edges. A minimum-cost spanning tree is one which has the smallest possible total weight (where weight represents cost or distance). There might be more than one such tree, but Prim and Kruskal are both guaranteed to find one of them.

For a specified vertex (say X), a shortest path tree is a spanning tree such that the path from X to any other vertex is as short as possible (i.e., has the minimum possible weight).

Prim and Dijkstra "grow" the tree out from a starting vertex. In other words, they have a "local" focus; at each step, we only consider those edges adjacent to previously chosen vertices, choosing the cheapest option which satisfies our needs. Meanwhile, Kruskal is a "global" algorithm, meaning that each edge is (greedily) chosen from the entire graph.

To find a minimum-cost spanning tree:

- **Kruskal (global approach):** At each step, choose the cheapest available edge *anywhere* which does not violate the goal of creating a spanning tree.
- **Prim (local approach):** Choose a starting vertex. At each successive step, choose the cheapest available edge attached to any previously chosen vertex which does not violate the goal of creating a spanning tree.

To find a shortest-path spanning tree:

- **Dijkstra:** At each step, choose the edge attached to any previously chosen vertex (the local aspect) which makes the total distance from the starting vertex (the global aspect) as small as possible, and

does not violate the goal of creating a spanning tree

Minimum-cost trees and shortest-path trees are easily confused, as are the Prim and Dijkstra algorithms that solve them. Both algorithms "grow out" from the starting vertex, at each step choosing an edge which connects a vertex Y which is in the tree to a vertex Z which is not. However, while Prim chooses the cheapest such edge, Dijkstra chooses the edge which results in the shortest path from X to Z.

Application of graphs:

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: To tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. Transportation networks: In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Utility graphs: The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

4. Document link graphs: The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. Protein-protein interactions graphs: Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

6. Network packet traffic graphs: Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. Scene graphs: In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

8. Finite element meshes: In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a

building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements form a graph that is called a finite element mesh.

9. Robot planning: Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

10. Neural networks: Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

11. Graphs in quantum field theory: Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

12. Semantic networks: Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

13. Graphs in epidemiology: Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

14. Graphs in compilers: Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

15. Constraint graphs: Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

16. Dependence graphs: Graphs can be used to represent dependences or precedence among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

Data Structure Lecture Notes

Unit -5

SORTING:

INTRODUCTION:

Sorting is any process of arranging items systematically, and has two common, yet distinct meanings:
1. **ordering:** arranging items in a sequence ordered by some criterion;
2. **Categorizing**: grouping items with similar properties.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Need for sorting:

The most common uses of sorted sequences are:

- Making lookup or search efficient.
- Making merging of sequences efficient.
- Enable processing of data in a defined order.

Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory –** The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary –** The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

SORT METHODS:

There are many types of sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

- Bubble sort
- Insertion sort
- Selection sort
- Quick sort
- Merge sort
- Heap sort
- Shell sort
- Radix sort

BUBBLE SORT:

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Bubble sort compares all the element one by one and sort them based on their values. If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times. It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

QUICK SORT:

~~Quick sort is~~ a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of o(n2), where n is the number of items.
Quick sort is a divide-and-conquer sorting algorithm in which division is dynamically carried out.

The steps of quick sort are as follows:

Divide: rearrange the elements and split the array into two sub arrays and an element in between such that so that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.
Conquer: recursively sort the two subarrays.

Algorithm:

```
Quicksort0 (A, p, r)
{
if (p< r)
{
q = Partition(A, p, r);
QuicksortO (A, p, q − 1);
Quicksort0 (A, q + 1, r);
}
}

Partition(A, p, r)
{
x = A[r] ;
i C p – 1;
for j C p to r − 1 do
{
if (A[j] ≤ x)
{
i C i + 1 ;
Exchange {A[i] and A[j] }
}
Exchange (A[i + 1] and A[r])
}
return i + 1;
}
```


Given a subarray A[p .. r] such that p ≤ r − 1, this subroutine rearranges the input subarray into two subarrays, A[p .. q − 1] and A[q + 1 .. r], so that

- each element in A[p .. q − 1] is less than or equal to A[q] and
- each element in A[q + 1 .. r] is greater than or equal to A[q]

Then the subroutine outputs the value of q. Use the initial value of A[r] as the “pivot,” in the sense that the keys are compared against it. Scan the keys A[p .. r − 1] from left to right and flush to the left all the keys that are greater than or equal to the pivot.

During the for-loop i + 1 is the position at which the next key that is greater than or equal to the pivot should go to.

Example:

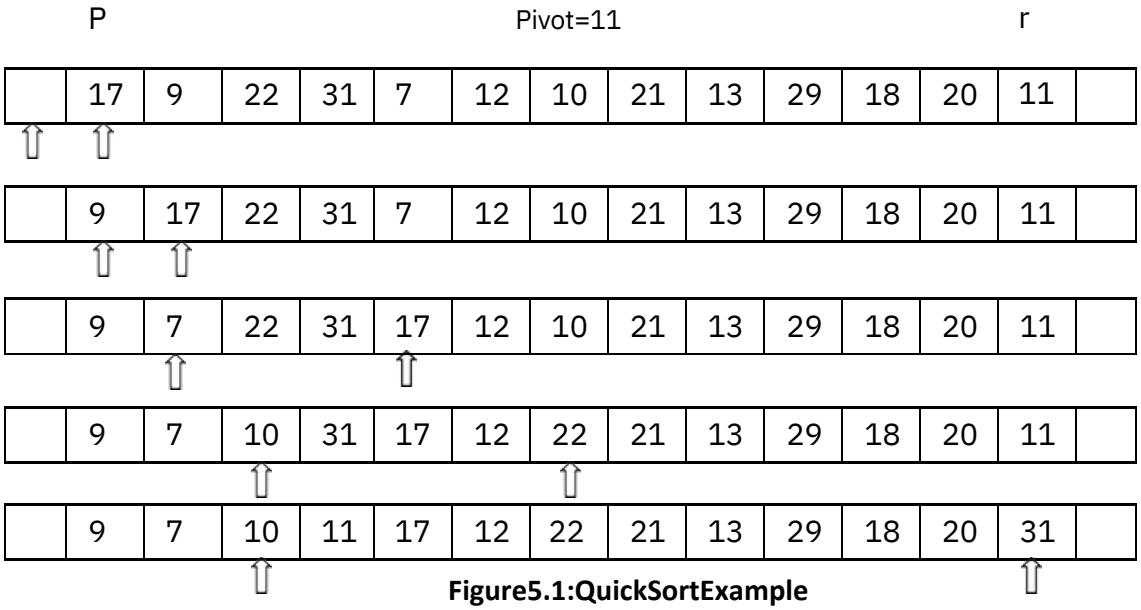


Figure5.1:QuickSortExample

SELECTION SORT:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where **n** is the number of items.

Example:

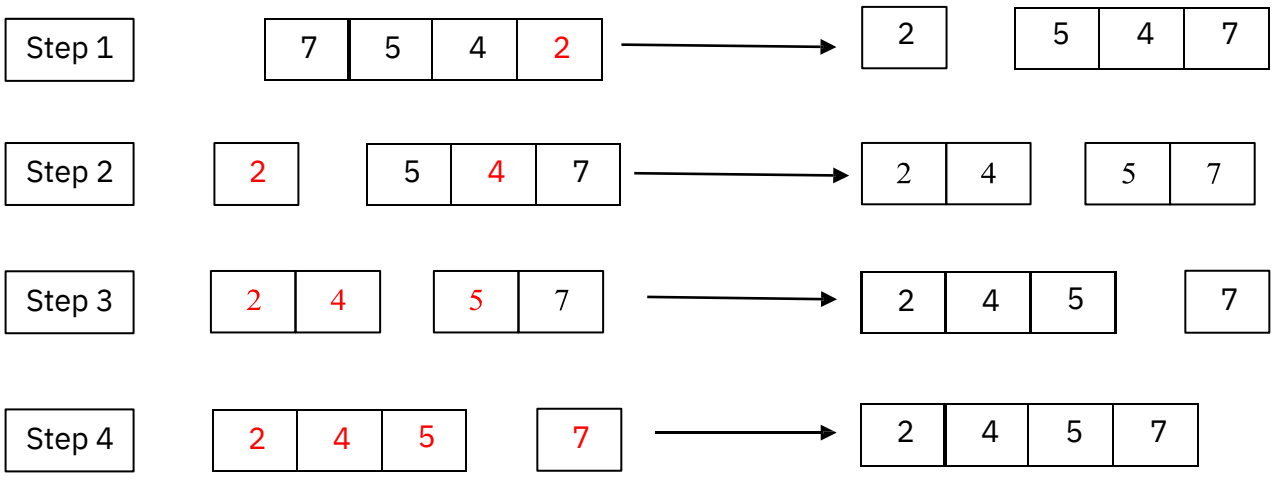


Figure5.2: Selection Sort

Algorithm:

Step 1 − Set MIN to location O
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

HEAP SORT:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Binary Heap:

• A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

• A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

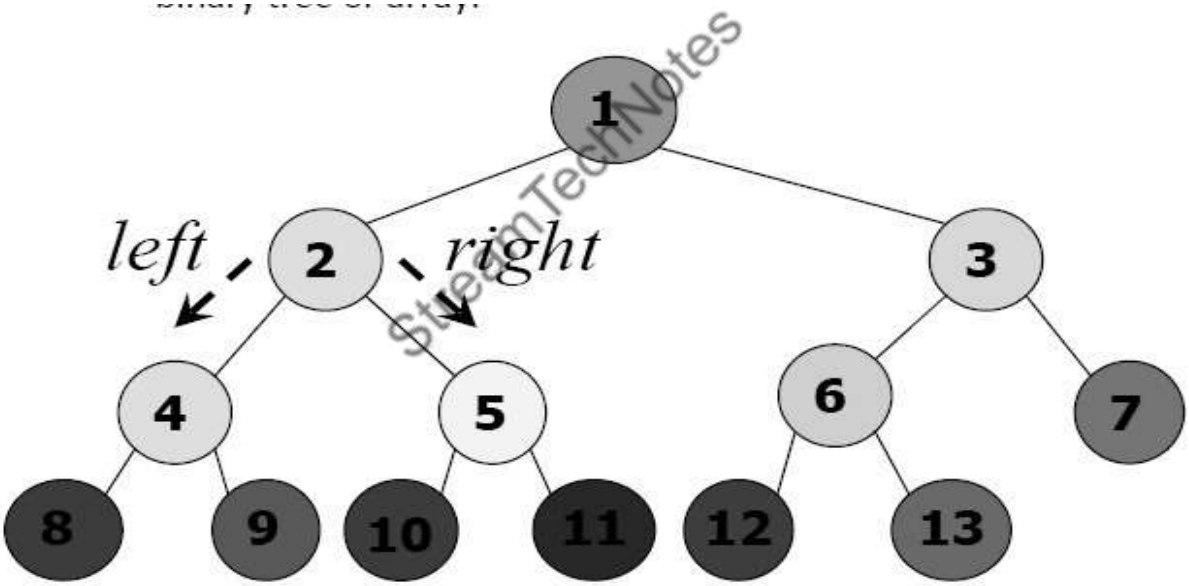


Figure 5.3: Binary Heap

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Representation of Binary Heaps:

• An array *A* that represents a heap is an object with two attributes:
• *length*[*A*], which is the number of elements in the array
• *heap-size*[*A*], the number of elements in the heap stored within array *A*.

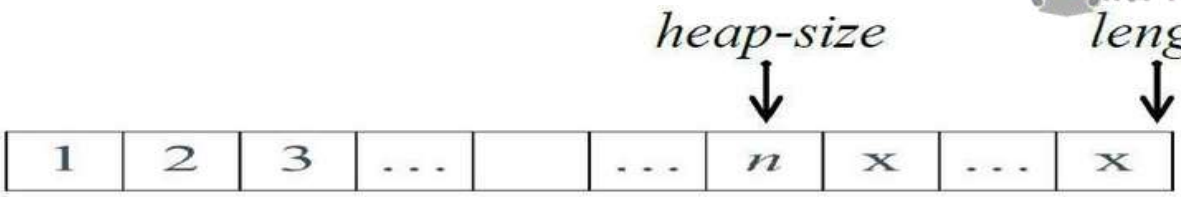


Figure 5.4: Representation of Binary Heap

Properties of Binary Heaps

• If a heap contains *n* elements, its height is lg2*n.*

• In a max-heaps

For every non-root node *i*, *A*[PARENT(*i*)+ ≥ []*A i*

a min-heaps

For every non-root node *i*, *A*[PARENT(*i*)+ ≤ []*A i*

Example:

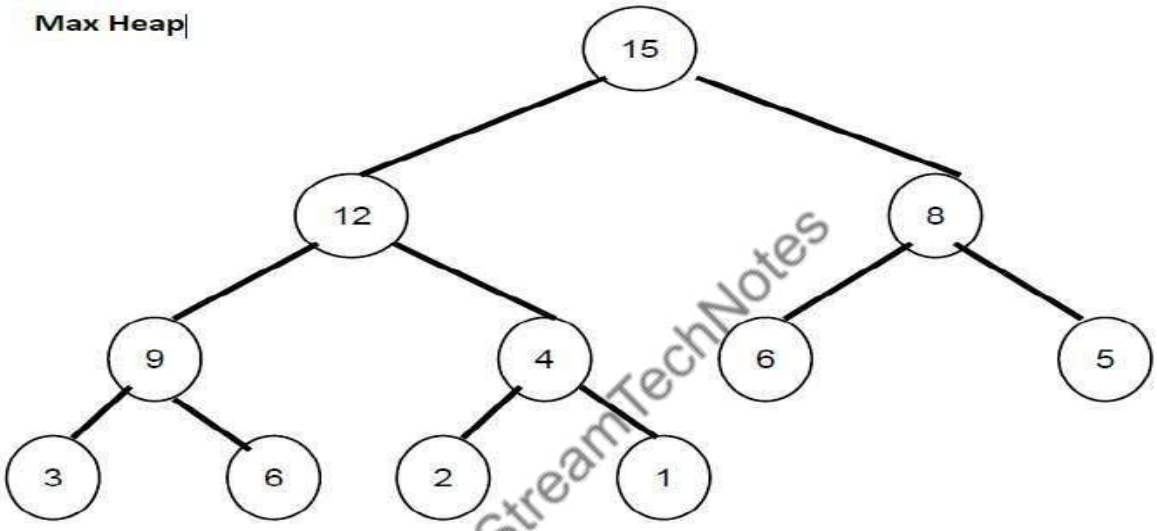


Figure 5.5: Max Heap

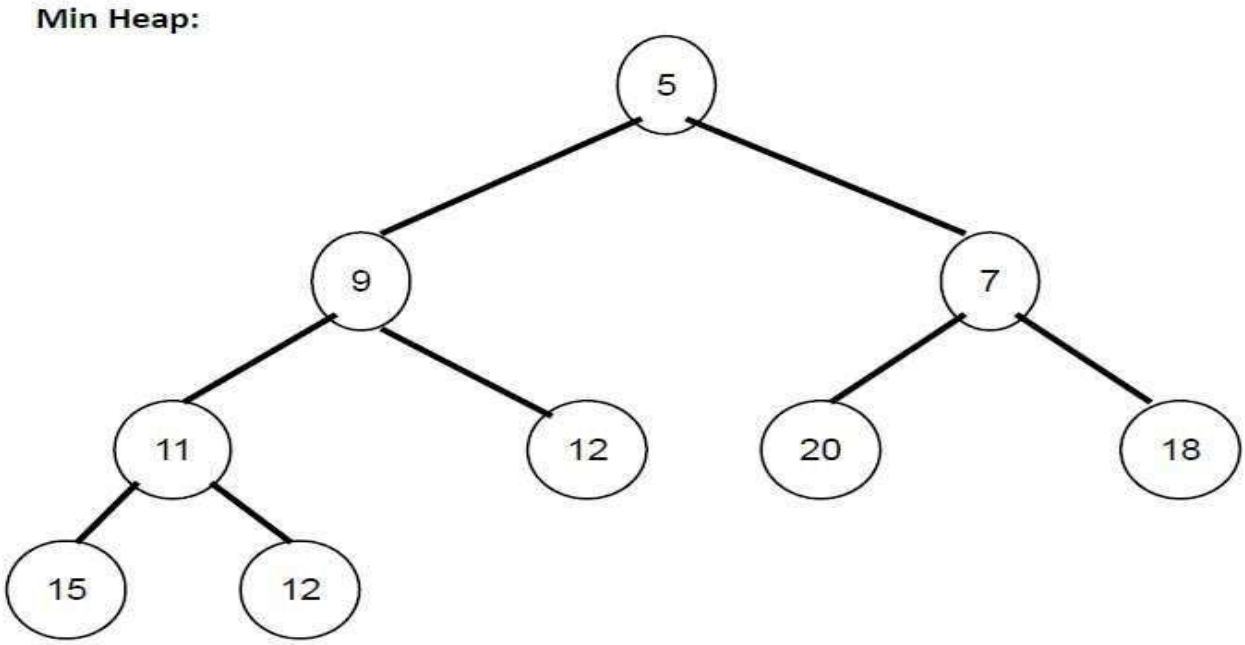


Figure 5.6: Min Heap

INSERTION SORT:

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less. Like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is a Stable sorting, as it does not change the relative order of elements with equal keys.

HowInsertion SortingWorks:

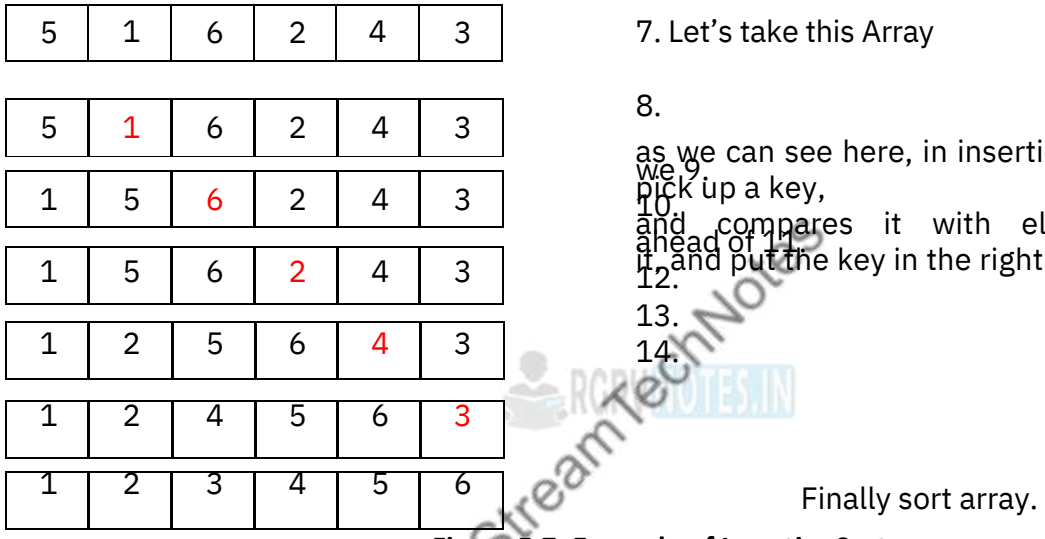


Figure 5.7: Example of InsertionSort

SHELL SORT:

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort.

This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n), where n is the number of items.

The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

Example: In the Figure 3.3 This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the

list shownin Figure 3.4 Althoughthislistisnotcompletelysorted,somethingveryinterestinghas happened.Bysortingthesublists,wehavemovedtheitemsclosertowheretheyactuallybelong.

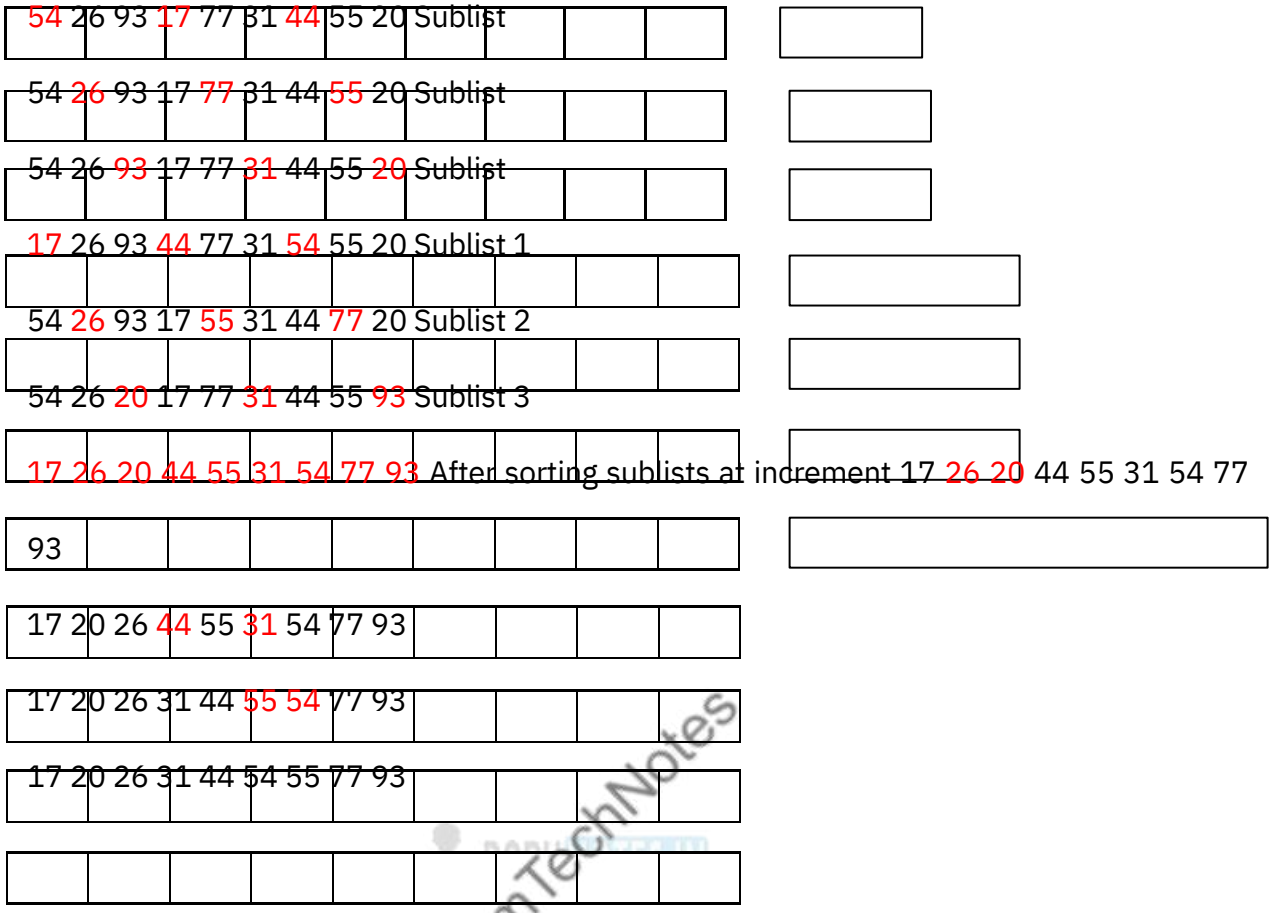


Figure5.8:ExampleofShellSort

Figure 5.8 showsafinalinsertionsortusinganincrementofone;inotherwords,astandardinsertion sort. Note thatbyperformingtheearliersublistsorts,wehavenowreducedthetotalnumberof shifting operationsnecessarytoputthelistinitsfinalorder.Forthiscase,weneedonlyfourmore shifts to completetheprocess.

Here are somekeypointsofshellsortalgorithm–

- ShellSortisacomparisonbasedsorting.
- TimecomplexityofShellSortdependsongapsequence.Itsbestcasetimecomplexityis O(n* logn) and worst case is O(n* log n). Time complexity of Shell sort is generally assumed to be neartoO(n)andlessthanO(n)asdeterminingitstimecomplexityisstillanopenproblem.
- Thebestcaseinshellsortiswhenthearrayisalreadysorted.Thenumberofcomparisonsis less.
- It isanin-placesortingalgorithmasitrequiresnoadditionalscratchspace.
- ShellSortisunstablesortasrelativeorderofelementswithequalvaluesmaychange.
- It isbeenobservedthatshellsortis5timesfasterthanbubblesortandtwicefasterthan insertionsortitsclosestcompetitor.
- Therearevariousincrementsequencesorgapsequencesinshellsortwhichproducevarious complexitybetweenO(n)andO(n).

MERGE SORT:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

• Divide the unsorted list into NN sublists, each containing 11 element.

• Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. NN will now convert into N/2N/2 lists of size 2.

• Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that mergesortfirstdivides thewholearray iteratively intoequalhalves unlessthe atomic values are achieved.Wesee herethat anarray of8itemsisdividedintotwoarrays ofsize4.

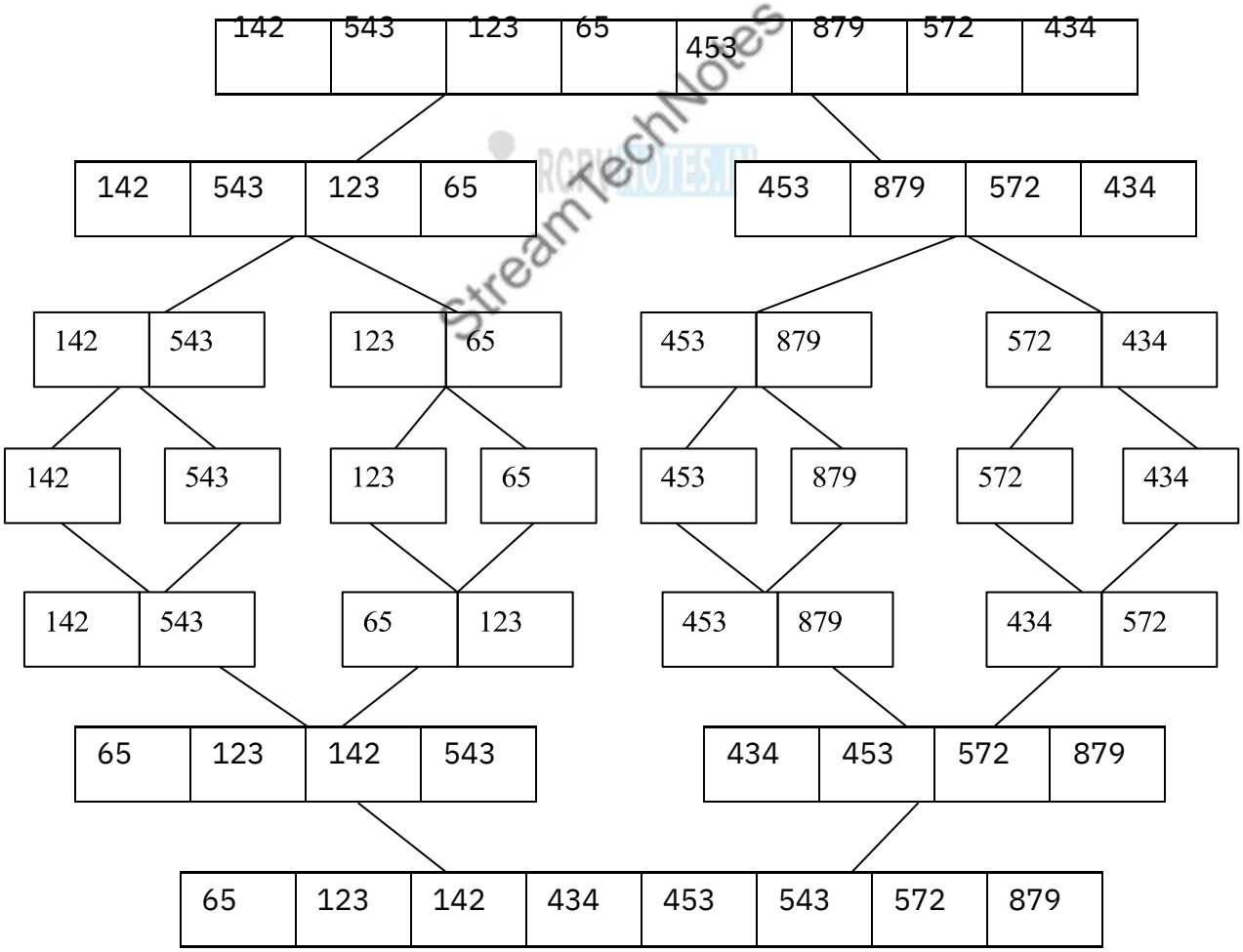


Figure 5.9: Merge Sort

RADIX SORT:

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the digits of the numbers.

We sort the numbers from the least significant digit to most significant digit.

Consider the following 9 numbers:

493, 812, 715, 710, 195, 437 ,582, 340, 385

We should start sorting by comparing and ordering the **one's** digits:

Digit Sublist	
0 340 710	
1	
2 812 582	
3 493	
4	
5 715 195 385	
6	
7 437	
8	
9	

Figure 5.10: Radix Sort Example (comparing based on one's digits)

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit Sublist	
0	
1 710 812 715	
2	
3 437	
4 340	
5	
6	
7	
8 582 385	
9 493 195	

Figure 5.11: Radix Sort Example (comparing based on ten's digits)

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit Sublist	
0	
1 195	
2	
3 340 385	
4 437 493	
5 582	
6	
7 710 715	
8 812	
9	

Figure 5.12: Radix Sort Example(comparing based on hundred's digits)

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

COMPARISON OF VARIOUS SORTING TECHNIQUES

	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort		O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	Stops after reaching a sorted array
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quick Sort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

Table 5.1: Sorting Comparison Table

SEARCHING:

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

BASIC SEARCH TECHNIQUES:

SEQUENTIAL SEARCH OR LINEAR SEARCH:

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Figure 5.13: Linear Search

BINARY SEARCH:

The Binary search technique is a search technique which is based on Divide & Conquer strategy. The entered array must be sorted for the searching, then we calculate the location of mid element by using formula mid= (Beg + End)/2, here Beg and End represent the initial and last position of array. In this technique we compare the Key element to mid element. So there May be three cases:-

- If array[mid] = = Key (Element found and Location is Mid)
- If array[mid] > Key ,Then set End = mid-1.(continue the process)
- If array [mid] < Key, Then set Beg=Mid+1. (Continue the process)

Binary Search Algorithm

1. [Initialize segment variable] set beg=LB,End=UB and Mid=int(beg+end)/2.
2. Repeat step 3 and 4 while beg<=end and Data[mid] != item.
3. If item< data[mid] then set end=mid-1
1. Else if Item>data[mid] then set beg=mid+1[end of if structure]

4. Set mid= int(beg+end)/2.[End of step 2 loop]
5. If data[mid]=item then set Loc= Mid.
2. Else set loc=null[end of if structure]
6. Exit.

COMPARISON OF SEARCH METHODS:

- The major difference between linear search and binary search is that binary search takes less time to search an element from the sorted list of elements. So it is inferred that efficiency of binary search method is greater than linear search.
- Another difference between the two is that there is a prerequisite for the binary search, i.e., the elements must be sorted while in linear search there is no such prerequisite.

HASHING & INDEXING:

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the hash key. Here, hash key is a value which provides the index value where the actual data is likely to store in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function.
Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure. Using hash table concept insertion, deletion and search operations are accomplished in constant time. Generally, every hash table make use of a function, which we'll call the **hash function** to map the data into the hash table.
Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.
Basic concept of hashing and hash table is shown in the following figure.

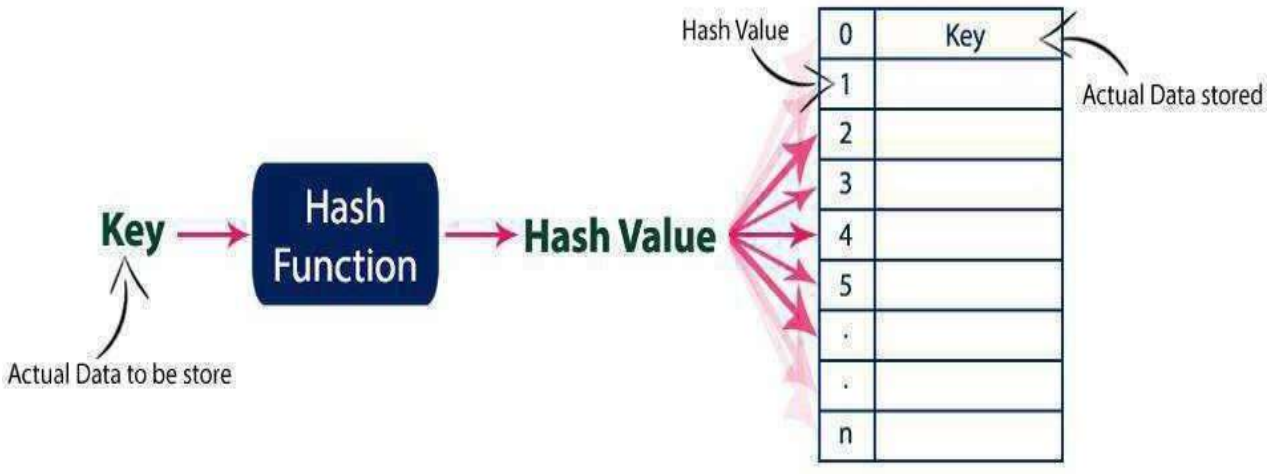


Figure 5.14: Hashing Concept

CASE STUDY:

APPLICATION OF VARIOUS DATA STRUCTURES IN OPERATING SYSTEM:

Some of the application of data structure associating in accomplishing operating System task like:

Stack:

• A stack is useful for the compiler/operating system to store local variables used inside a function block, so that they can be discarded once the control comes out of the function block.

• A stack can be used as an "undo" mechanism in text editors; this operation has been accomplished by keeping all text changes in a stack.

• Stacks are useful in backtracking, which is a process when you need to access the most recent data element in a series of elements.

• An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation: 1 + ((2 + 3) * 4 + 5)*6

Queue:

• Queues can be used to store the interrupts in the operating system.

• It is used by an application program to store the incoming data.

• Queue is used to process synchronization in Operating System

• Queues are used for CPU job scheduling and in disk scheduling.

Linked lists:

• Linked lists are used in dynamic Memory Management tasks like allocation and releasing memory at run time.

• Linked lists are used in Symbol Tables for balancing parenthesis and in representing Sparse Matrix.

• The cache in your browser that allows you to hit the BACK button where a linked list of URLs can be implemented.

Tree:

• An operating system maintains a disk's file system as a tree, where file folders act as tree nodes. The tree structure is useful because it easily accommodates the creation and deletion of folders and files.

• Trees are used to represent phrase structure of sentences, which is crucial to language processing programs. Java compiler checks the grammatical structures of Java program by reading the program's words and attempting to build the program's parse tree. If successfully constructed the parse tree is used as a guide to help the Java compiler in order to generate the byte code that one finds in program's class file.

Queue:

• The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B.

• Simultaneous execution of jobs problem between set of processors and set of jobs can be easily solved with graphs.

APPLICATIONS OF DATA STRUCTURES FOR DATA BASE MANAGEMENT SYSTEM:

The data structures employed in a DBMS context are B-trees, buffer trees, quad trees, R-trees, interval trees, hashing etc. Data Structures for Query Processing high-level input query expressed in a declarative language called SQL the parser scans, parses, and validates the query.