Data Structure and Algorithm

EG2202CT

Notes

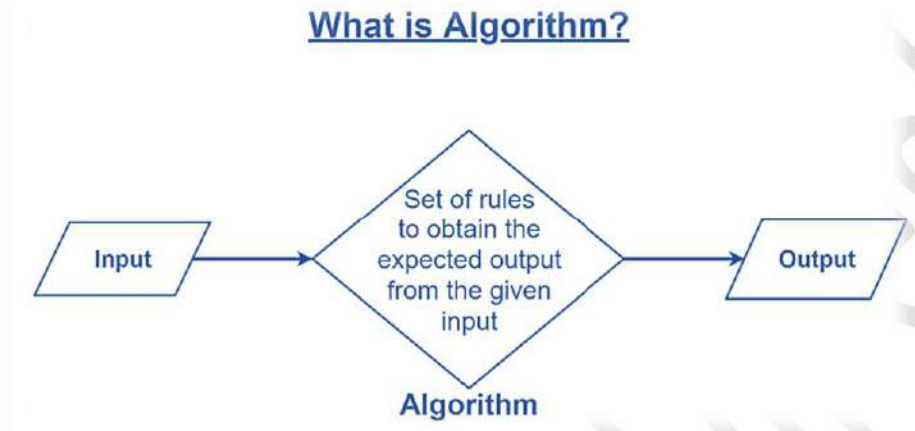
Fourth Semester_(DCOM/IT)

Website :- www.arjun00.com.np

Unit:1 - Introduction to DSA:

1.1 Algorithm and Its types:

An algorithm is a procedure used for solving a problem or performing a computation. Algorithms act as an exact list of instructions that conduct specified actions step by step to get desired output. Algorithms work by following a set of instructions or rules to complete a task or solve a problem. They can be expressed as natural languages, programming languages, pseudocode, flowcharts and control tables.



Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e., every step should do some work.

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm:

It is the simplest approach to a problem. A brute force algorithm is the first approach that comes to finding when we see a problem. More technically it is just like iterating every possibility available to solve that problem.

2. Recursive Algorithm:

A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Randomized Algorithm:

In the randomized algorithm, we use a random number.it helps to decide the expected outcome. The decision to choose the random number so it gives the immediate benefit.

4. Sorting Algorithm:

Sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

5. Hashing Algorithm:
Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

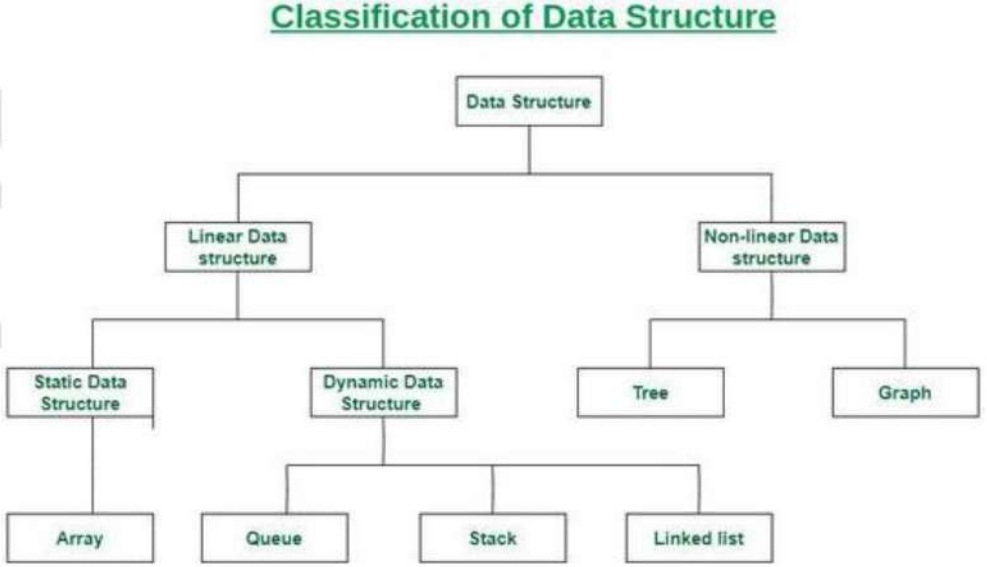
6. Searching Algorithm:
The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data.

1.2 Data Structure and Its types:

Data Structure is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required.

Classification of Data Structure:

Data structure has many different uses in our daily life. There are many different data structures that are used to solve different mathematical and logical problems. Let's look at different data structures that are used in different situations.



Linear data structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
Examples of linear data structures are array, stack, queue, linked list, etc.

- **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
  An example of this data structure is an array.

1. Array Data Structure:

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

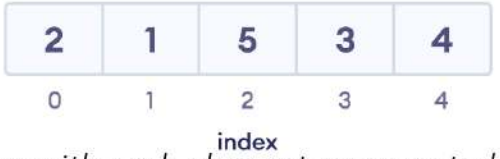
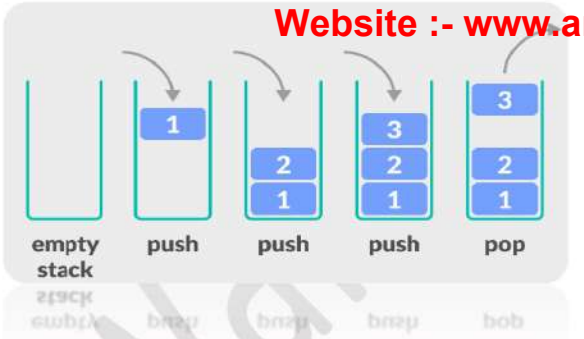


Fig. An array with each element represented by an index

- **Dynamic data structure:** In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  Examples of this data structure are queue, stack, Linked list.

2. Stack:

Stack is a linear data structure that follows a particular order in which the operations are performed. The order is LIFO (Last in first out). Entering and retrieving data is possible from only one end. The entering and retrieving of data is also called push and pop operation in a stack.

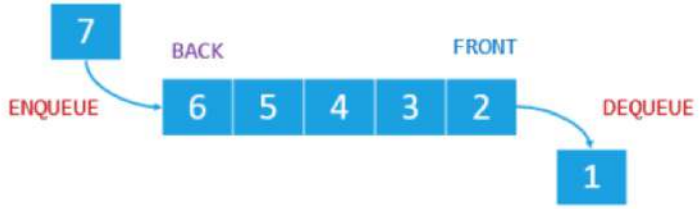


In a stack, operations can be performed only from one end (top here).

- Push: To insert a new element in the Stack.
- Pop: To remove or delete elements from the Stack

3. Queue Data Structure:

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first. It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first



In a queue, addition and removal are performed from separate ends.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data

items and address to the next node.

Fig. A Linked List

Non-linear data structure: Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead, they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Examples of non-linear data structures are trees and graphs.

5. Graph:

A graph is a non-linear data structure that consists of vertices (or nodes) and edges. It consists of a finite set of vertices and set of edges that connect a pair of nodes. The graph is used to solve the most challenging and complex programming problems.

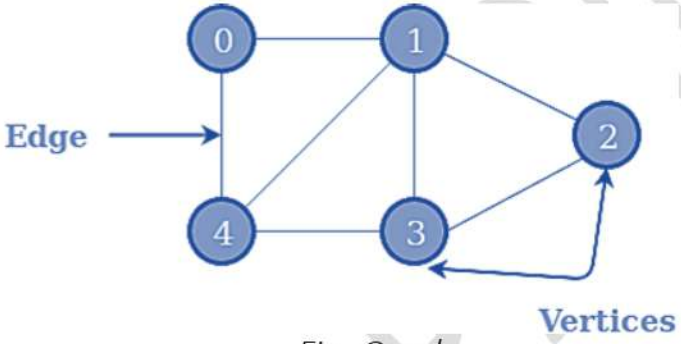


Fig. Graph

6. Tree:

A tree is a non-linear and hierarchical data structure where the elements are arranged in a tree-like structure. In a tree, the topmost node is called the root node. Each node contains some data, and data can be of any type.

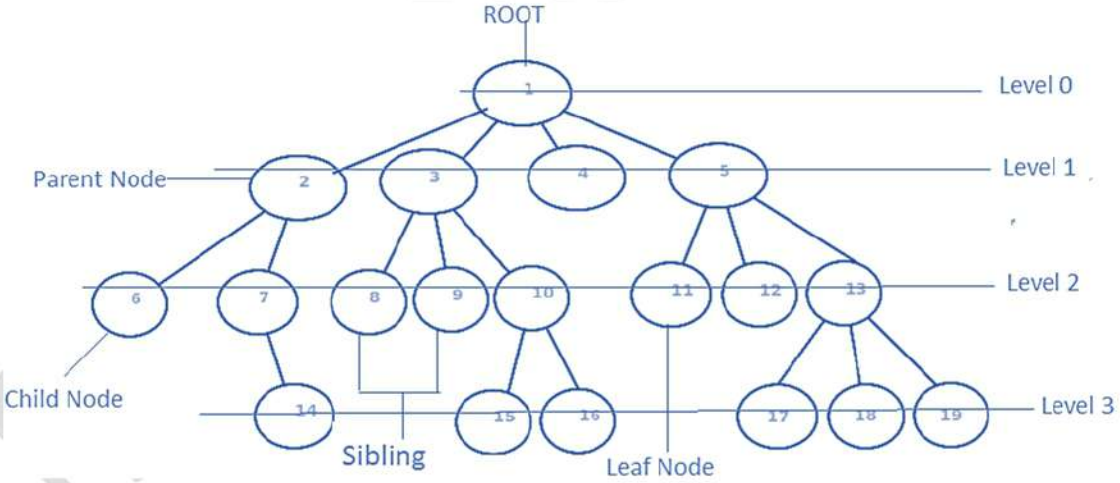


Fig. Tree

1.3 Tools for Algorithm analysis:

1.3.1 Types of analysis Time and space complexity:

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:

- **A Priori Analysis –** This is a theoretical analysis of an algorithm and is done before implementation.

- **A Posterior Analysis –** This is done after the implementation. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

The complexity of an algorithm is a function describing the efficiency of an algorithm in terms of the amount of the data the algorithm must process. There are two main complexity measures of the efficiency for an algorithm:

a. Space Complexity: It represents the amount of memory space requires by the algorithm in its life cycle.
b. Time Complexity: It represent the amount of time required by the algorithm to run to completion.

1.3.2 Asymptotic Notations: Big-O, Big-Ω and Big-θ

Asymptotic Analysis

Asymptotic analysis is the method of algorithm analysis depending upon the time and space factor. Using asymptotic analysis, an algorithm falls under:

- **Worst case:** It defines the input for which the algorithm takes a huge time.
- **Average case:** It takes average time for the program execution.
- **Best case:** It defines the input for which the algorithm takes the lowest time

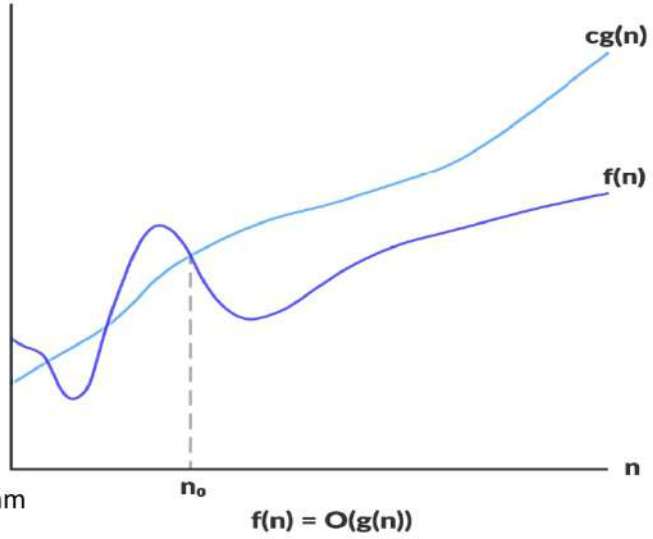
Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

a. O – Big Oh Notation
b. Ω – Big Omega Notation
c. Θ – Theta Notation

a. Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the **worst-case time complexity** or the longest amount of time an algorithm can possibly take to complete.

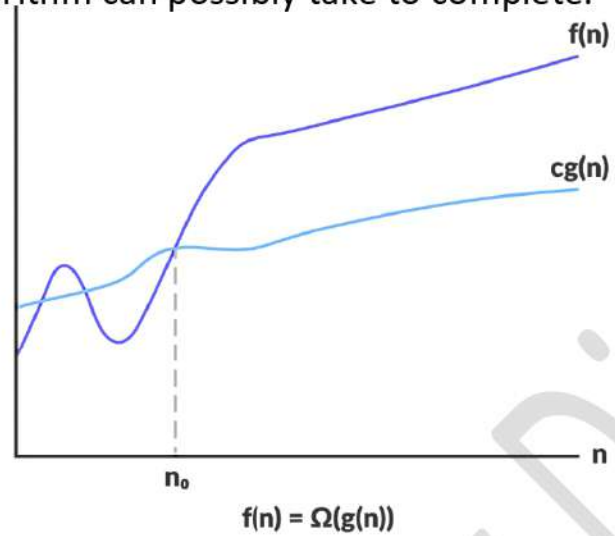


Big-O gives the upper bound of a function.

When we say f(n) is O(g(n)), it means that the running time of the algorithm (f(n)) is at least as small as the running time of another function (g(n)).

b. Big Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the **best-case time complexity** or the best amount of time an algorithm can possibly take to complete.

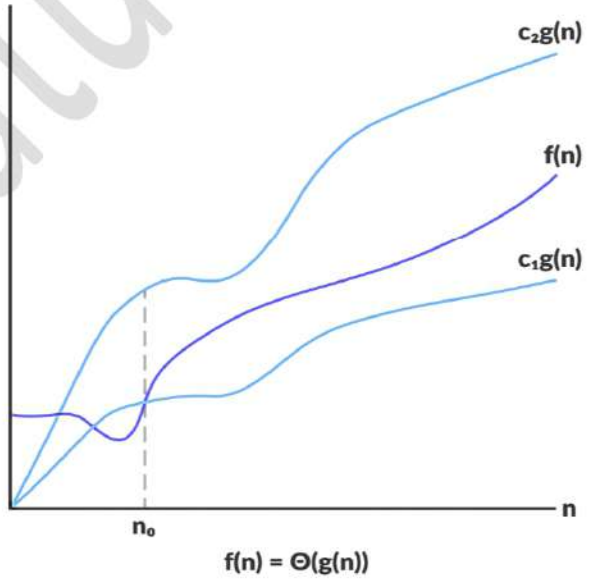


Omega gives the lower bound of a function.

When we say f(n) is Ω (g(n)), it means that the running time of the algorithm (f(n)) is at least as larger as the running time of another function (g(n)).

c. Theta Notation, θ

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case complexity** of an algorithm.



Theta bounds the function within constants factors.

Difference Between Big oh, Big Omega and Big Theta :

S.No.	Big Oh (O)	Big Omega (Ω)	Big Theta (Θ)
1.	It is like (<=) rate of growth of an algorithm is less than or equal to a specific value.	It is like (>=) rate of growth is greater than or equal to a specified value.	It is like (==) meaning the rate of growth is equal to a specified value.
2.	The upper bound of algorithm is represented by Big O notation.	The algorithm's lower bound is represented by Omega notation	The bounding of function from above and below is represented by theta notation.
3.	Big oh (O) – Upper Bound	Big Omega (Ω) – Lower Bound	Big Theta (Θ) – Tight Bound
4.	It is defined as upper bound and upper bound on an algorithm is the most amount of time required (the worst-case performance).	It is defined as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).	It is defined as tightest bound and tightest bound is the best of all the worst- case times that the algorithm can take.
5.	Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0	Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0	Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

Unit:2 – Stack and Queue:

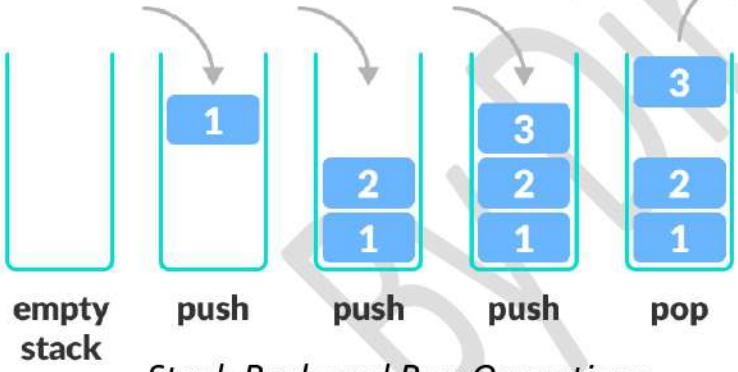
2.1 Stack and Operation

Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.

In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



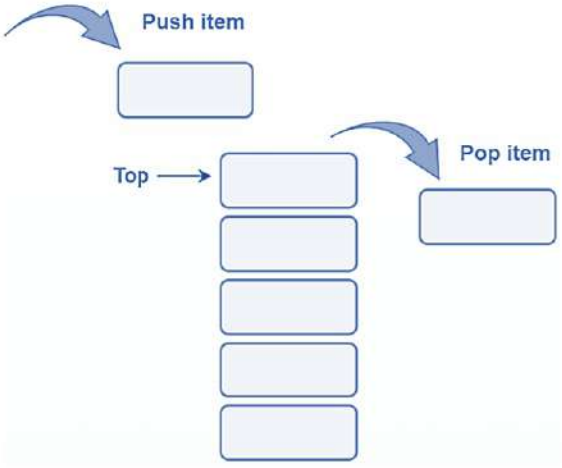
Stack Push and Pop Operations

In the above image, although item 3 was kept last, it was removed first. This is exactly how the **LIFO (Last In First Out)** Principle works.

Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations:

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **size()** returns the size of stack.

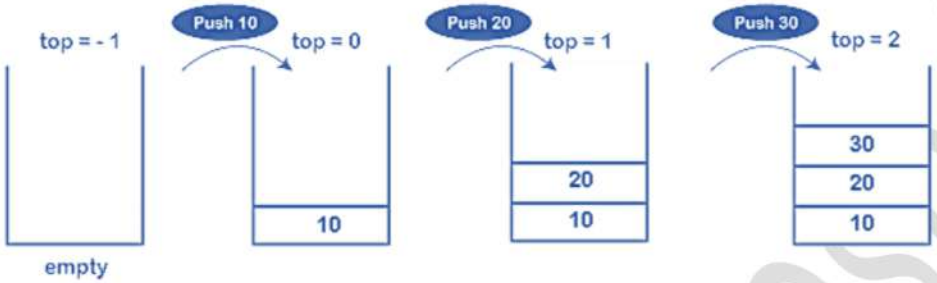


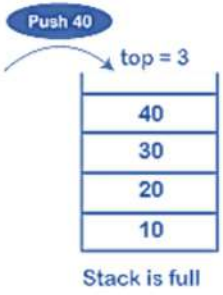
Push:

Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

Algorithm for push:

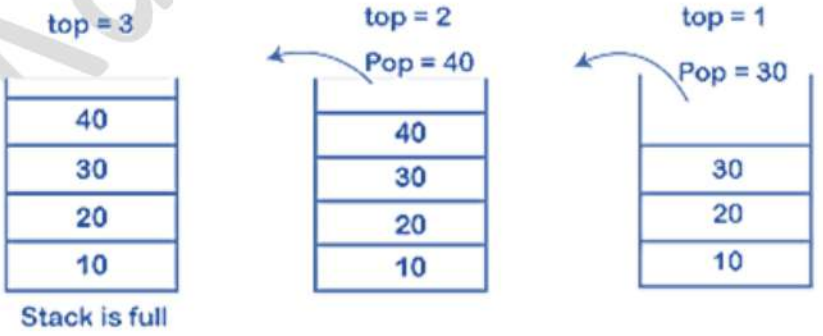
```
begin
   if stack is full
      return
   endif
else
   increment top
   stack[top] assign value
end else
end procedure
```

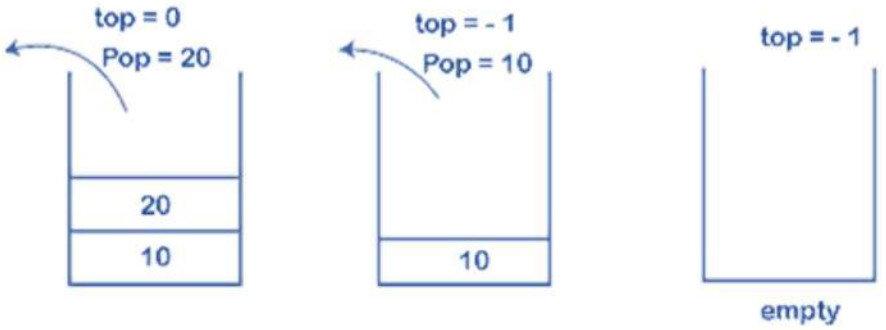




Algorithm for pop:

```
begin
  if stack is empty
     return
  endif
else
  store value of stack[top]
  decrement top
  return value
end else
end procedure
```





Pop:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Top:
Returns the top element of the stack.

Algorithm for Top:

```
begin
 return stack[top]
end procedure
```

isEmpty:
Returns true if the stack is empty, else false.

Algorithm for isEmpty:

```
begin
 if top < 1
   return true
 else
   return false
end procedure
```

2.1.1 Continuous Implementation of Stack:

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called the 'pop' operation. The value of the variable 'top' will be decremented by 1 whenever an item is deleted from the stack. The top-most element of the stack is typically stored in another variable before the 'top' is decremented to access and use the value.

Array implementation of Stack (C Program):

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{
   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("*****Stack operations using array*****");
   printf("\n----------------------------------------------\n");
```

```
while(choice != 4)
{
   printf("Chose one from the below options...\n");
   printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
   printf("\n Enter your choice \n");
   scanf("%d",&choice);
   switch(choice)
   {
      case 1:
      {
         push();
         break;
      }
      case 2:
      {
         pop();
         break;
      }
      case 3:
      {
         show();
         break;
      }
      case 4:
      {
         printf("Exiting....");
         break;
      }
      default:
      {
         printf("Please Enter valid choice ");
      }
   };
}
}
```

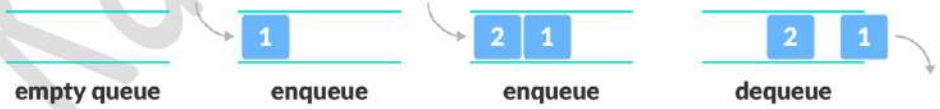
```
void push ()
{
   int val;
   if (top == n )
   printf("\n Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;
   }
}

void pop ()
{
   if(top == -1)
   printf("Underflow");
   else
   top = top -1;
}
void show()
{
   for (i=top;i>=0;i--)
   {
      printf("%d\n",stack[i]);
   }
   if(top == -1)
   {
      printf("Stack is empty");
   }
}
```

2.3 Queue

2.3.1 Definition

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket. In programming terms, putting items in the queue is called enqueue, and removing items from the queue is called dequeue.

Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first.



FIFO Representation of Queue

In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

Basic Operations of Queue

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue:** Add an element to the end of the queue
- **Dequeue:** Remove an element from the front of the queue
- **IsEmpty:** Check if the queue is empty
- **IsFull:** Check if the queue is full

- **Peek:** Get the value of the front of the queue without removing it

2.3.2 Algorithm of Enqueue and Dequeue

1. Enqueue()

The enqueue() is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm

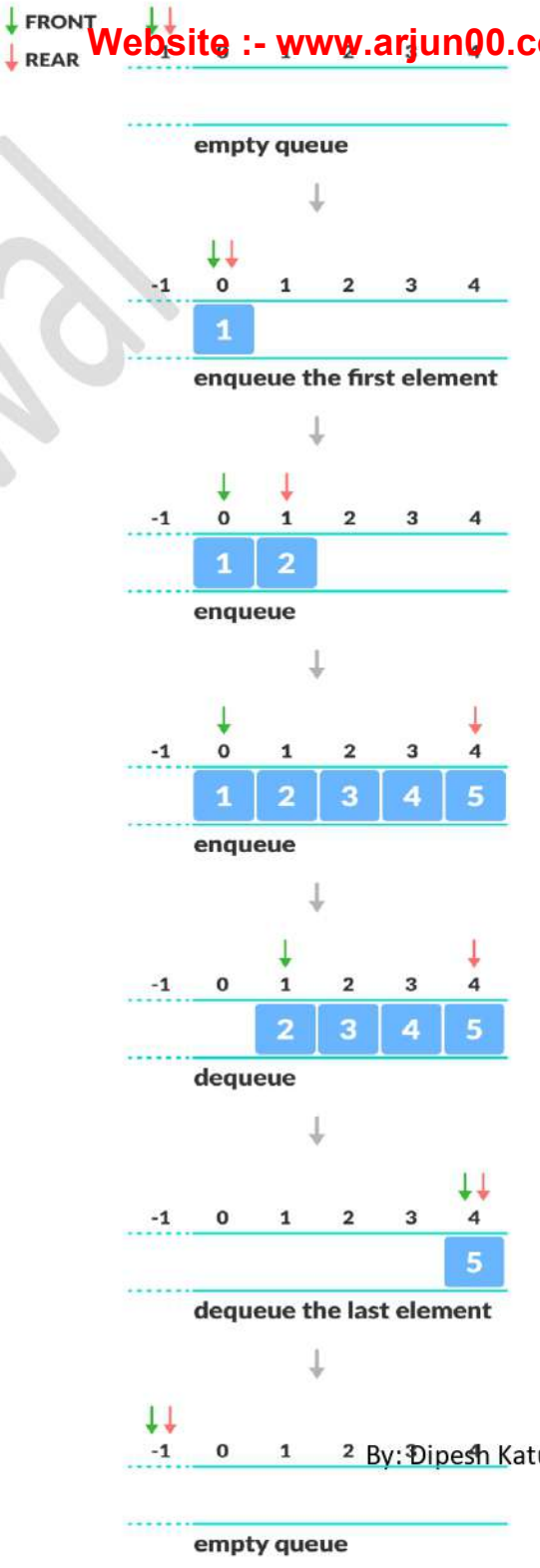
1 – START
2 – Check if the queue is full.
3 – If the queue is full, produce overflow error and exit.
4 – If the queue is not full, increment rear pointer to point the next empty space.
5 – Add data element to the queue location, where the rear is pointing.
6 – return success.
7 – END

2. Dequeue()

The dequeue() is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

1 – START
2 – Check if the queue is empty.
3 – If the queue is empty, produce underflow error and exit.
4 – If the queue is not empty, access the data where front is pointing.
5 – Increment front pointer to point to the next available data element.
6 – Return success.
7 – END



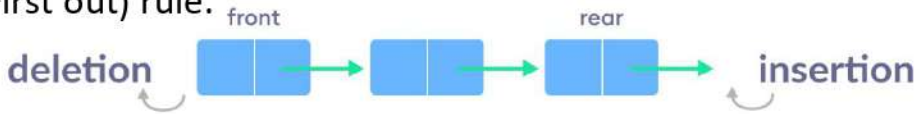
Types of Queues

There are three different types of queues:

1. Simple Queue (or Linear Queue)
2. Circular Queue
3. Priority Queue

1. Simple Queue (or Linear Queue)

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



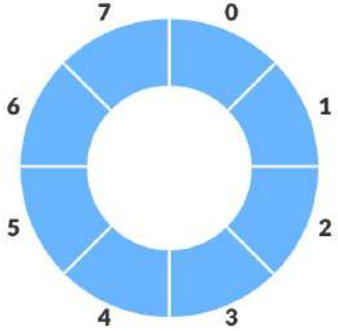
Simple Queue Representation

Implementing Linear Queue (C Program)

```
#include <stdio.h>
#define SIZE 5

void enQueue(int);
void deQueue();
void display();

int items[SIZE], front = -1, rear = -1;
int main() {
  deQueue();
  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);
  enQueue(6);
  display();
  deQueue();
  display();
  return 0;
}

void enQueue(int value) {
  if (rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (front == -1)
      front = 0;
    rear++;
    items[rear] = value;
    printf("\n Inserted -> %d", value);
  }
}

void deQueue() {
  if (front == -1)
    printf("\nQueue is Empty!!");
  else {
    printf("\nDeleted : %d",
items[front]);
    front++;
    if (front > rear)
      front = rear = -1;
  }
}

void display() {
  if (rear == -1)
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements
are:\n");
    for (i = front; i <= rear; i++)
      printf("%d  ", items[i]);
  }
  printf("\n");
}
```

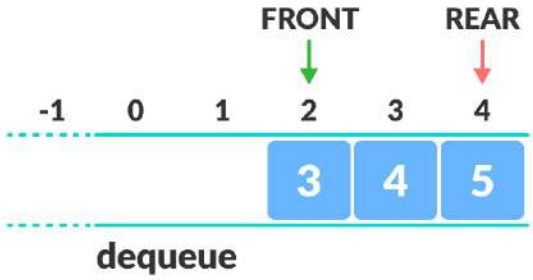
2. Circular Queue

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus, forming a circle-like structure.



Circular queue representation

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Limitation of the regular Queue

Here, indexes 0 and 1 can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

Circular Queue Operations

The circular queue work as follows:

- two pointers **FRONT** and **REAR**
- **FRONT** track the first element of the queue
- **REAR** track the last elements of the queue
- initially, set value of **FRONT** and **REAR** to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of **FRONT** to **0**
- circularly increase the **REAR** index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by **REAR**

2. Dequeue Operation

- check if the queue is empty
- return the value pointed by **FRONT**
- circularly increase the **FRONT** index by **1**
- for the last element, reset the values of **FRONT** and **REAR** to -1

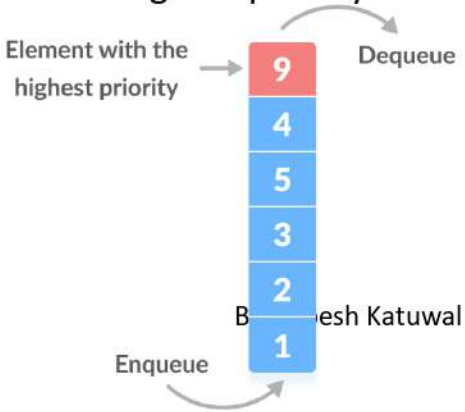
3. Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

However, if elements with the same priority occur, they are served according to their order in the queue.

Difference between Priority Queue and Normal Queue

In a queue, the f**irst-in-first-out** rule is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.



Removing Highest Priority Element =>

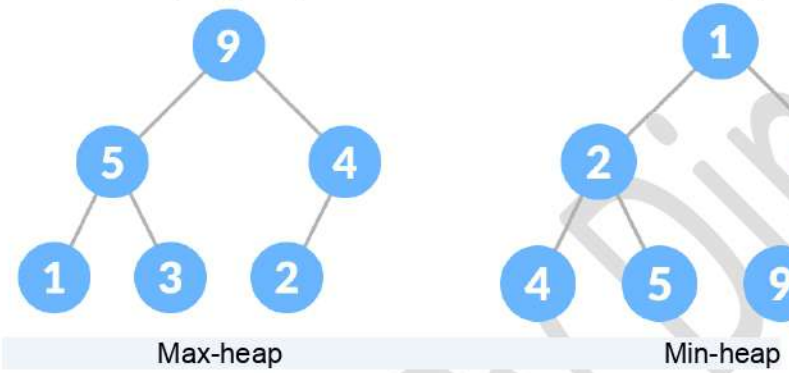
Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, **heap data structure** provides an efficient implementation of priority queues.

Heap Data Structure

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



This type of data structure is also called a **binary heap.**

Some common applications of Queue data structure :

a. Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

b. Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

c. Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

d. Message Buffering: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

e. Event Handling: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

f. Traffic Management: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

g. Operating systems: Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

h. Network protocols: Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.

i. Printer queues: In printing systems, queues are used to manage the order in which print jobs are processed. Jobs

are added to the queue as they are submitted, and the printer processes them in the order they were received.

j. Web servers: Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.

k. Breadth-first search algorithm: The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

2.3.6 Applications of queues

Some applications of queues:

- CPU scheduling- to keep track of processes for the CPU
- Handling website traffic- by implementing a virtual HTTP request queue
- Printer Spooling- to store print jobs
- In routers- to control how network packets are transmitted or discarded
- Traffic management- traffic signals use queues to manage intersections

Here are a few more real-life applications of queues:

- Luggage checking machine
- Vehicles on toll tax bridge
- One way exits
- Patients waiting outside the doctor's clinic
- Phone answering systems
- Cashier line in a store

Unit:3 – List:

3.1 Definition and Structure of link list

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node. For example,

Representation of Linked List

Each node consists:

- A data item.
- An address of another node

3.2 Advantages and Disadvantage of Linked Lists:

Advantages of Linked Lists:

1. **Dynamic Size:** Linked lists can dynamically grow and shrink in size during execution, unlike arrays, which have a fixed size. This flexibility enables efficient memory utilization.
2. **Ease of Insertion and Deletion:** Insertions and deletions in a linked list can be more efficient than arrays, especially in cases where elements need to be added or removed from the middle of the list. This is because it involves only changing the links, rather than shifting elements as in arrays.
3. **No Wastage of Memory:** Linked lists use memory efficiently since they only consume memory based on the number of elements they contain, without requiring a fixed block of memory like arrays.
4. **Dynamic Memory Allocation:** Linked lists allow for dynamic memory allocation, making it easier to manage memory compared to arrays, where resizing can be a costly operation.

Disadvantages of Linked Lists:

1. **Sequential Access:** Accessing an element in a linked list typically requires traversing from the beginning of the list, making sequential access less efficient.
2. **Extra Memory Usage:** Linked lists utilize additional memory for storing pointers or references to the next element, which can be a drawback compared to arrays that solely store data.
3. **No Random Access:** Unlike arrays, linked lists do not support random access.
4. **More Complex Implementation:** Implementing and managing linked lists can be more complex compared to arrays due to the need for proper pointer manipulation and potential issues like memory leaks and pointer errors.

3.3 Operations in singly link list:

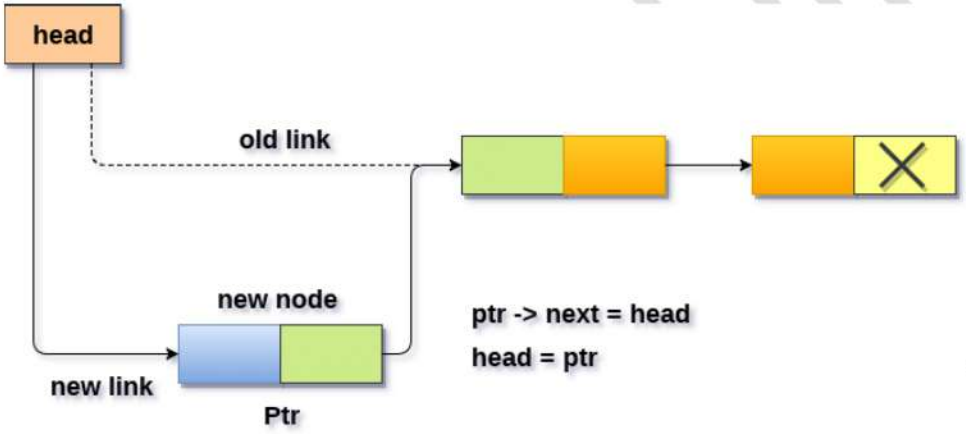
There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

3.3.1 Insertion at the beginning and end, after the node, before the node

Algorithm for Insertion at the Beginning of a Singly Linked List

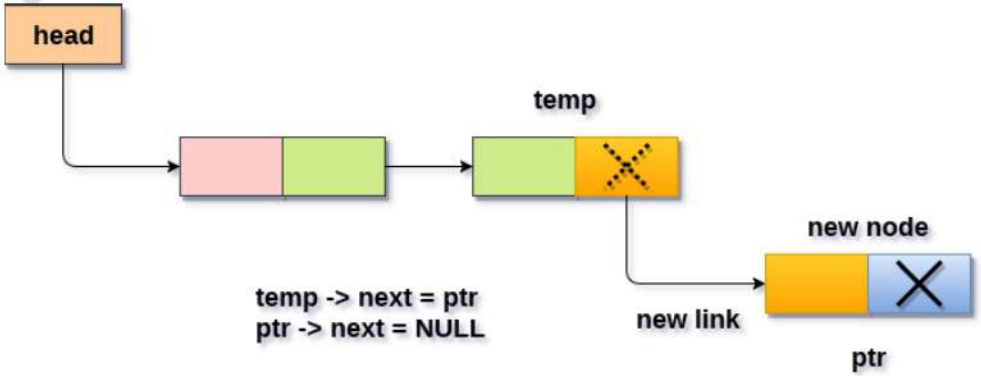
Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 7
[END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT



Algorithm for Insertion at the End of a Singly Linked List

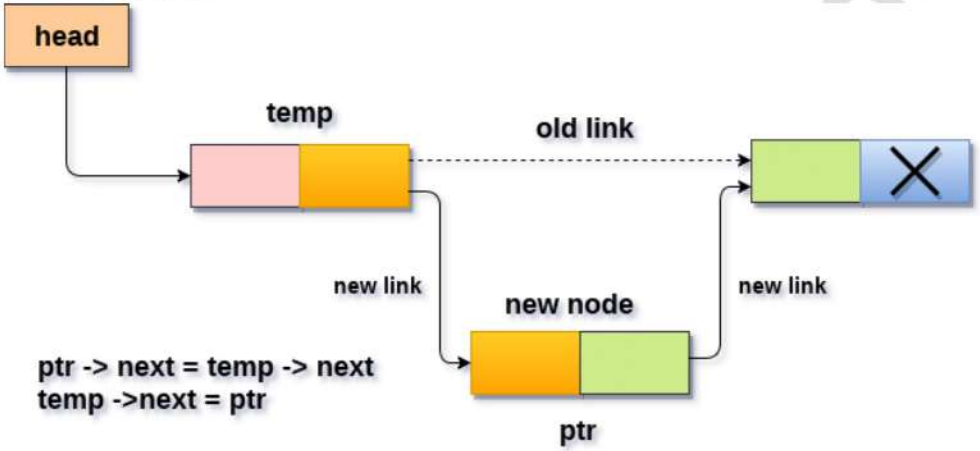
Step 1: IF PTR = NULL Write OVERFLOW
Go to Step 1
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR - > NEXT
Step 4: SET NEW_NODE - > DATA = VAL
Step 5: SET NEW_NODE - > NEXT = NULL
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR - > NEXT != NULL
Step 8: SET PTR = PTR - > NEXT
[END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



Algorithm for Insertion after specified Node

STEP 1: IF PTR = NULL
WRITE OVERFLOW
GOTO STEP 12

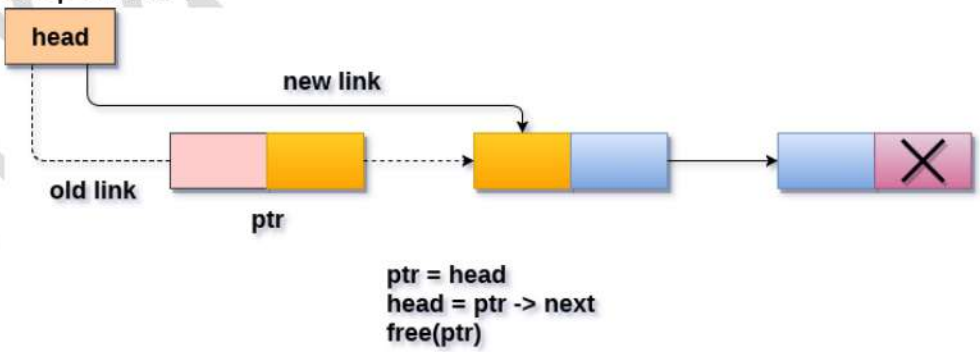
END OF IF
STEP 2: SET NEW_NODE = PTR
STEP 3: NEW_NODE → DATA = VAL
STEP 4: SET TEMP = HEAD
STEP 5: SET I = 0
STEP 6: REPEAT STEP 5 AND 6 UNTIL I
STEP 7: TEMP = TEMP → NEXT
STEP 8: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12
END OF IF
END OF LOOP
STEP 9: PTR → NEXT = TEMP → NEXT
STEP 10: TEMP → NEXT = PTR
STEP 11: SET PTR = NEW_NODE
STEP 12: EXIT



3.3.1 Deletion at the beginning and end, after the node, before the node

Algorithm for Deletion at beginning

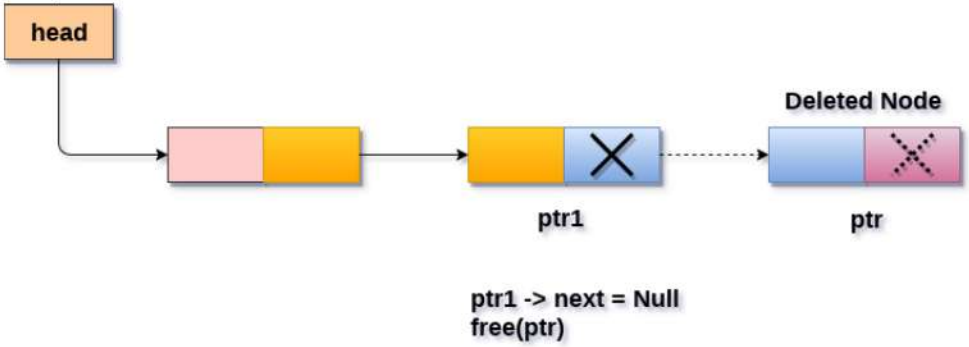
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT



Algorithm for Deletion at end

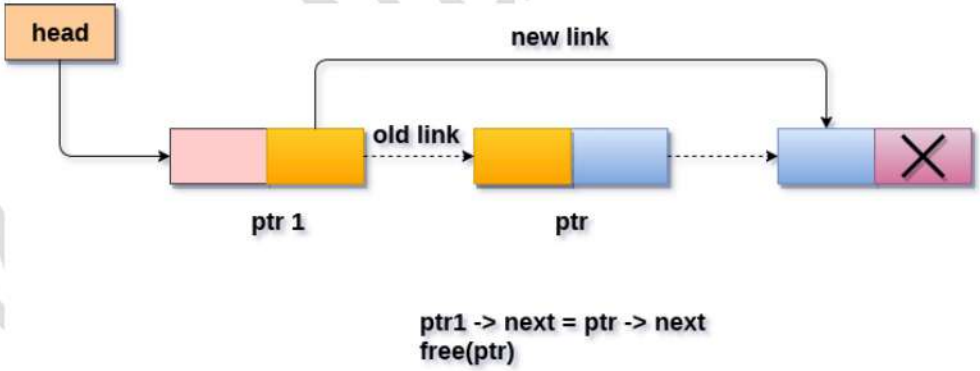
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT!= NULL
Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT



Algorithm for Deletion after the specified node :

STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
GOTO STEP 10
END OF IF
STEP 2: SET TEMP = HEAD
STEP 3: SET I = 0
STEP 4: REPEAT STEP 5 TO 8 UNTIL I
STEP 5: TEMP1 = TEMP
STEP 6: TEMP = TEMP → NEXT
STEP 7: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12
END OF IF
STEP 8: I = I+1
END OF LOOP
STEP 9: TEMP1 → NEXT = TEMP → NEXT
STEP 10: FREE TEMP
STEP 11: EXIT



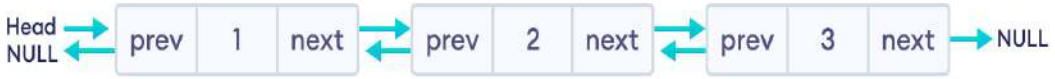
3.4 Doubly Linked List

3.4.1 Definition

A doubly linked list (DLL) is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list.

3.4.2 Structure of doubly linked list:

A doubly linked list is a type of linked list in which each node consists of 3 components:
***prev** - address of the previous node
data - data item
***next** - address of next node



A doubly linked list node

3.4.3 Insertion in doubly linked list

1. Insertion at the Beginning

In this operation, we create a new node with three compartments, one containing the data, the others containing the address of its previous and next nodes in the list. This new node is inserted at the beginning of the list.

Algorithm:

1. START
2. Create a new node with three variables: prev, data, next.
3. Store the new data in the data variable
4. If the list is empty, make the new node as head.
5. Otherwise, link the address of the existing first node to the next variable of the new node, and assign null to the prev variable.
6. Point the head to the new node.
7. END

2. Insertion at the End

In this insertion operation, the new input node is added at the end of the doubly linked list; if the list is not empty. The head will be pointed to the new node, if the list is empty.

Algorithm:

1. START
2. If the list is empty, add the node to the list and point the head to it.
3. If the list is not empty, find the last node of the list.
4. Create a link between the last node in the list and the new node.
5. The new node will point to NULL as it is the new last node.
6. END

3. Add a node after a given node:

We are given a pointer to a node as prev_node, and the new node is inserted after the given node.

Algorithm:

1. Firstly create a new node (say new_node).
2. Now insert the data in the new node.
3. Point the next of new_node to the next of prev_node.
4. Point the next of prev_node to new_node.
5. Point the previous of new_node to prev_node.
6. Change the pointer of the new node's previous pointer to new_node.

4. Add a node before a given node:

Let the pointer to this given node be next_node.

Algorithm:

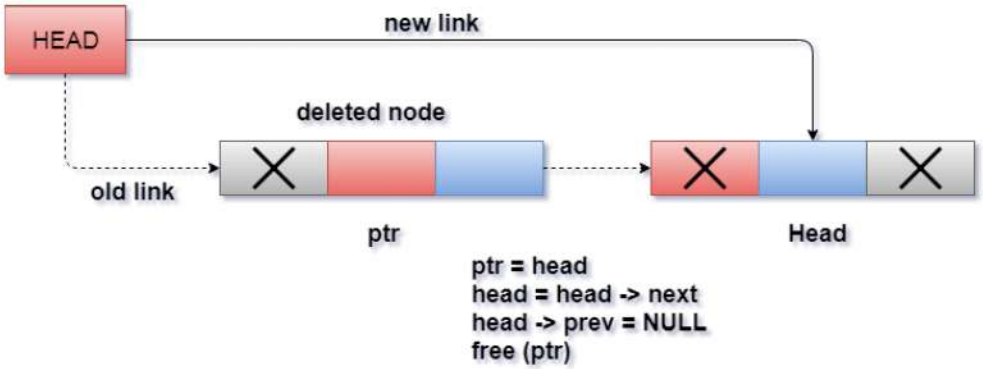
1. Allocate memory for the new node, let it be called new_node.
2. Put the data in new_node.
3. Set the previous pointer of this new_node as the previous node of the next_node.
4. Set the previous pointer of the next_node as the new_node.
5. Set the next pointer of this new_node as the next_node.
6. Now set the previous pointer of new_node.
   - If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node.
   - Else, if the prev of new_node is NULL, it will be the new head node.

3.4.4 Deletion in doubly linked list

1. Deletion at beginning

Algorithm

STEP 1: IF HEAD = NULL
WRITE UNDERFLOW, GOTO STEP 6
STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD → NEXT
STEP 4: SET HEAD → PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT

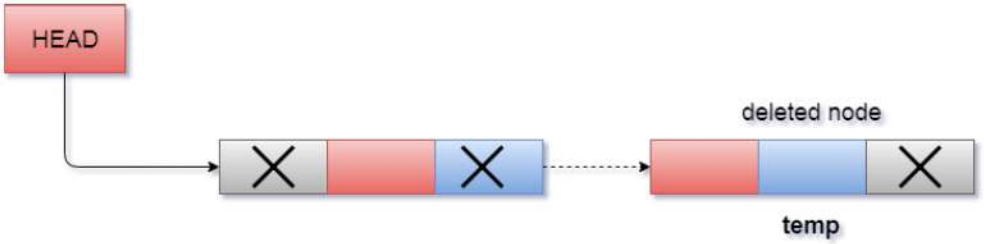


Deletion in doubly linked list from beginning

2. Deletion at the end

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]
Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]
Step 5: SET TEMP ->PREV-> NEXT = NULL
Step 6: FREE TEMP
Step 7: EXIT

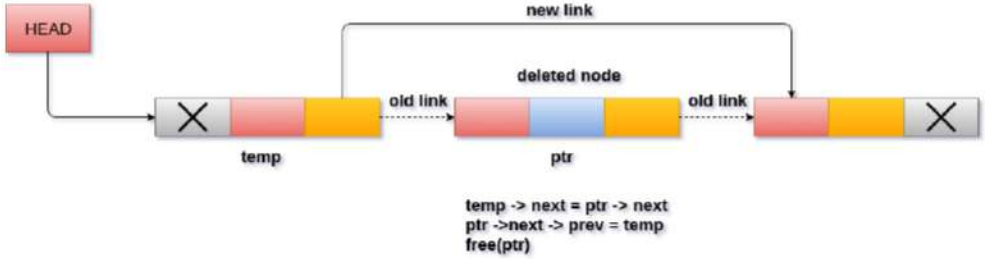


Deletion in doubly linked list at the end

3. Deletion after the specified node

Algorithm

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



Deletion of a specified node in doubly linked list

3.4.5 Advantages and Disadvantages

Advantages of doubly linked list:

1. It allows traversal in both forward and backward directions.
2. It allows easy insertion and deletion of elements at both ends of the list.

Disadvantages of doubly linked list:

1. It requires more memory as compared to a singly linked list due to the extra pointer for the previous node.
2. It requires more operations to maintain the list, such as updating the previous pointer when inserting or deleting elements.

Unit:4 – Recursion:

4.1. Properties of Recursion

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process.

Need of Recursion:

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write.

Properties of recursion:

Some properties of recursion in data structures include:

a) **Base case:** Every recursive function must have a base case, which is a condition that stops the recursion and provides a result. Without a base case, the recursion would continue indefinitely.

b) **Recursive case:** This is the part of the recursive function that calls itself with a modified input. It allows the function to break down the problem into smaller subproblems.

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

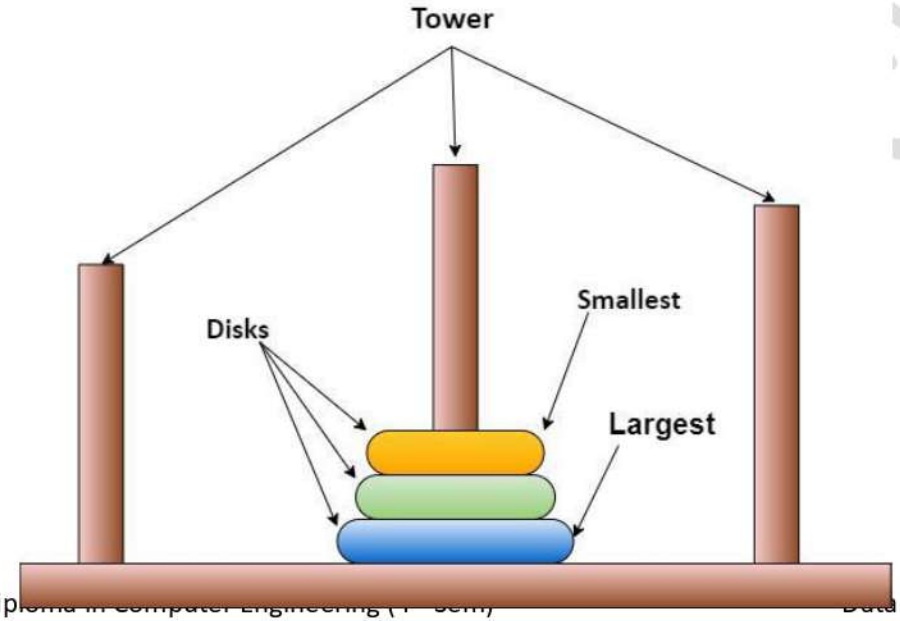
4.2 Recursion VS Iteration

Property	Recursion	Iteration
Definition	Function calls itself	A set of instruction repeatedly executed.
Application	For function	For Loops
Termination	Through the base case, where there will be no function call.	When the termination condition for the iterator cases to be satisfied.

Code size	Smaller code size	Larger code size
Stack	Here the stack is used to store local variables when the function is called.	Stack is not used.
Speed	Execution is slow	Normally, it is faster than recursion.
Memory	Recursion uses more memory as compared to iteration.	Iteration uses less memory as compared to recursion.

4.3 TOH and its solution

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings. These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.



Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Solution:

Here, we are taking 3 disks for solving our TOH problem.

1. Move the first disk from A to C
2. Move the first disk from A to B
3. Move the first disk from C to B
4. Move the first disk from A to C
5. Move the first disk from B to A
6. Move the first disk from B to C
7. Move the first disk from A to C

4.4 Solution of Fibonacci series and Factorial

Fibonacci Series in C (without recursion)

```
1. #include<stdio.h>
2. int main()
3. {
4.  int n1=0,n2=1,n3,i,number;
```

```
5.  printf("Enter the number of elements:");
6.  scanf("%d",&number);
7.  printf("\n%d %d",n1,n2);//printing 0 and 1
8.  for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 ar
    e already printed
9.  {
10.  n3=n1+n2;
11.  printf(" %d",n3);
12.  n1=n2;
13.  n2=n3;
14. }
15.  return 0;
16. }
```

Fibonacci Series using recursion in C

```
1.  #include<stdio.h>
2.  int main(){
3.     int n;
4.     printf("Enter the number of elements: ");
5.     scanf("%d",&n);
6.     printf("Fibonacci Series: ");
7.     printf("%d %d ",0,1);
8.     printFibonacci(n-2);//n-
    2 because 2 numbers are already printed
9.   return 0;
10. }
11.
12. void printFibonacci(int n){
13.    static int n1=0,n2=1,n3;
14.    if(n>0){
15.        n3 = n1 + n2;
16.        n1 = n2;
17.        n2 = n3;
18.        printf("%d ",n3);
19.        printFibonacci(n-1);
20.    }
21. }
```

Factorial Program using loop

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.   int i,fact=1,number;
5.   printf("Enter a number: ");
6.    scanf("%d",&number);
7.     for(i=1;i<=number;i++){
8.        fact=fact*i;
9.    }
10.  printf("Factorial of %d is: %d",number,fact);
11. return 0;
12. }
```

Factorial Program using recursion in C

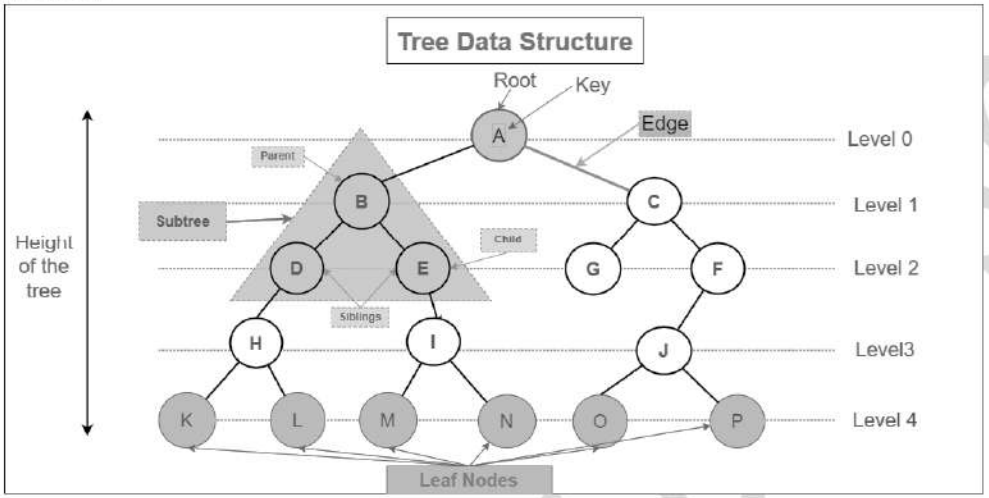
```c
#include<stdio.h>
void main()
{
  int number;
  long fact;
  printf("Enter a number: ");
  scanf("%d", &number);
  fact = factorial(number);
  printf("Factorial of %d is %ld\n", number, fact);
  return 0;
}

long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
```

Unit:5 – Tree:

5.1. Tree Concepts:

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it is called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes.



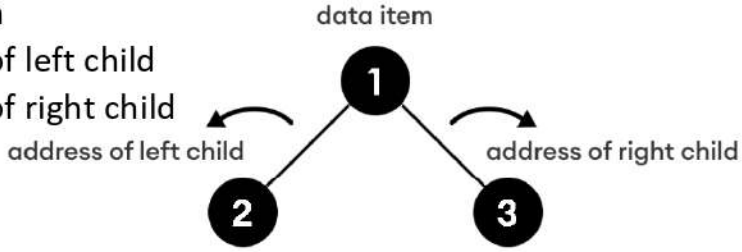
Basic Terminologies in Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P, G} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

5.2 Binary Tree

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

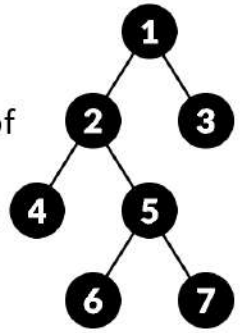
- data item
- address of left child
- address of right child



Types of Binary Tree

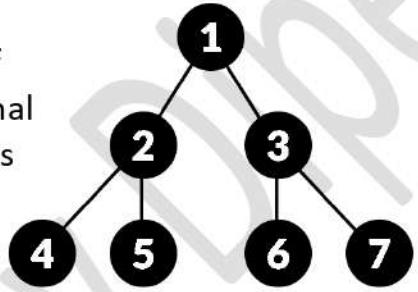
1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



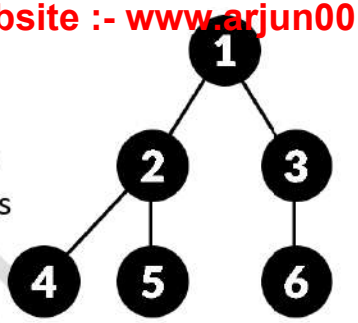
2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two majors differences



- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e., a complete binary tree doesn't have to be a full binary tree.

5.3 Application of Binary Tree

- **Search algorithms:** Binary search algorithms use the structure of binary trees to efficiently search for a specific element.
- **Sorting algorithms:** Binary trees can be used to implement efficient sorting algorithms, such as binary search tree sort and heap sort.
- **Database systems:** Binary trees can be used to store data in a database system, with each node representing a record. This allows for efficient search operations and enables the database system to handle large amounts of data.
- **File systems:** Binary trees can be used to implement file systems, where each node represents a directory or file. This allows for efficient navigation and searching of the file system.
- **Game AI:** Binary trees can be used to implement game AI, where each node represents a possible move in the game. The AI algorithm can search the tree to find the best possible move.

5.4 Node representation

In a binary tree, each node is represented by three parts: data, left pointer, and right pointer. The data part contains the value of the node, while the left and right pointers contain the addresses of the left and right child nodes, respectively. There are two common methods to represent a binary tree: array representation and linked list representation.

- **Array Representation:** In this representation, the root node is stored at the first position of the array, and its left and right children are stored at the second and third positions, respectively. The remaining nodes are stored in consecutive positions in the array, allowing for constant-time access to each node in the tree
- **Linked List Representation:** In this representation, each node consists of three fields. The first field is for storing the left child address, the second field is for storing the actual data, and the third field is for storing the right child address.

5.5 Operation in Binary Tree

1. Insertion

The process of adding the data in the binary search tree is called insert operation.

Algorithm:

1. START
2. If the tree is empty, insert the first element as the root node of the tree. The following elements are added as the leaf nodes.
3. If an element is less than the root value, it is added into the left subtree as a leaf node.
4. If an element is greater than the root value, it is added into the right subtree as a leaf node.
5. The final leaf nodes of the tree point to NULL values as their child nodes.
6. END

2. Search

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree.

Algorithm:

1. START
2. Check whether the tree is empty or not
3. If the tree is empty, search is not possible
4. Otherwise, first search the root of the tree.
5. If the key does not match with the value in the root, search its subtrees.
6. If the value of the key is less than the root value, search the left subtree
7. If the value of the key is greater than the root value, search the right subtree.
8. If the key is not found in the tree, return unsuccessful search.
9. END

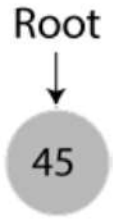
5.6 Tree search:

Create a binary search tree where data elements are: **45, 15, 79, 90, 10, 55, 12, 20, 50**

Steps:

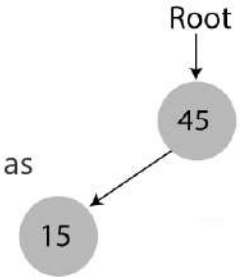
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Step 1 - Insert 45.



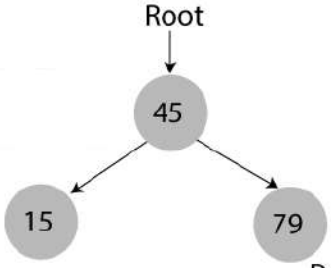
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



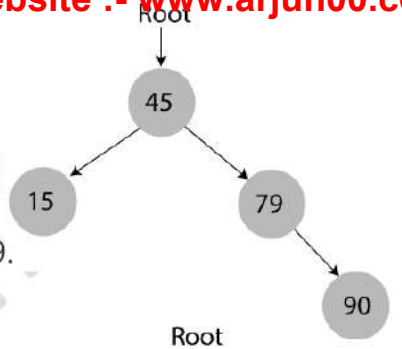
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



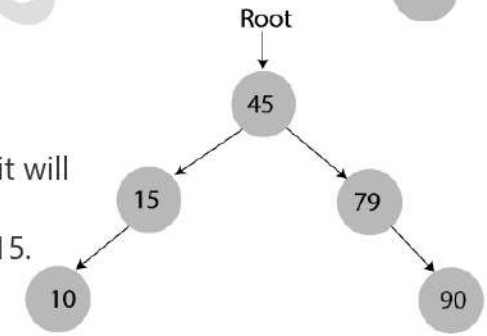
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



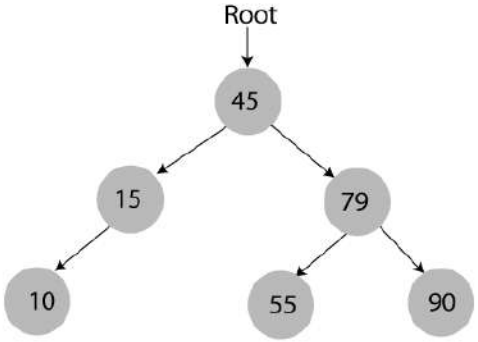
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



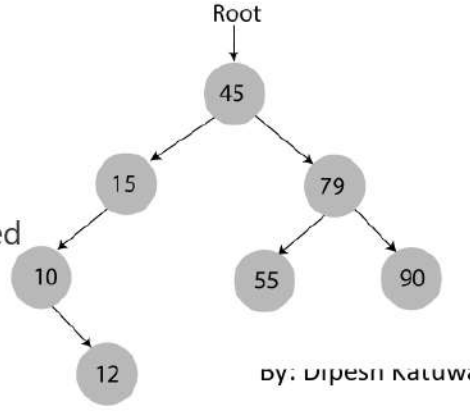
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



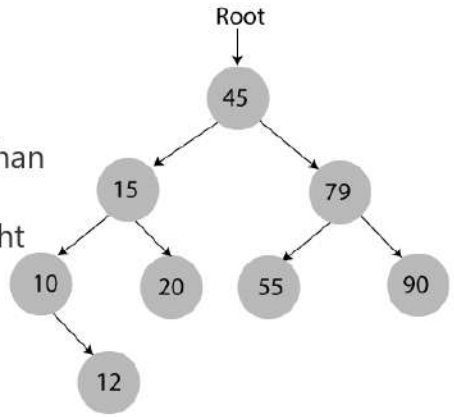
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



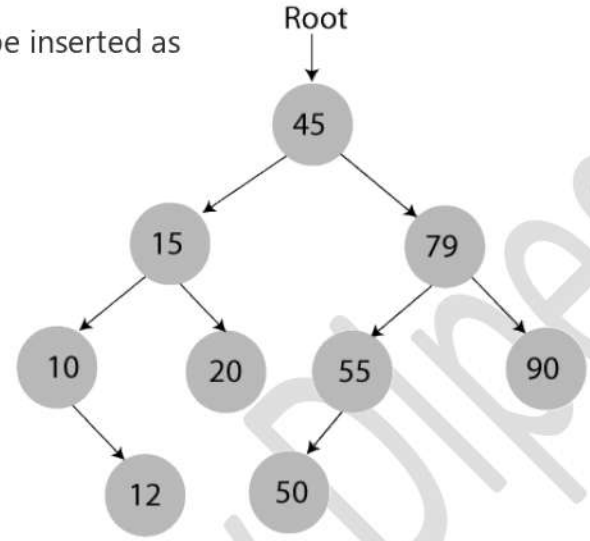
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



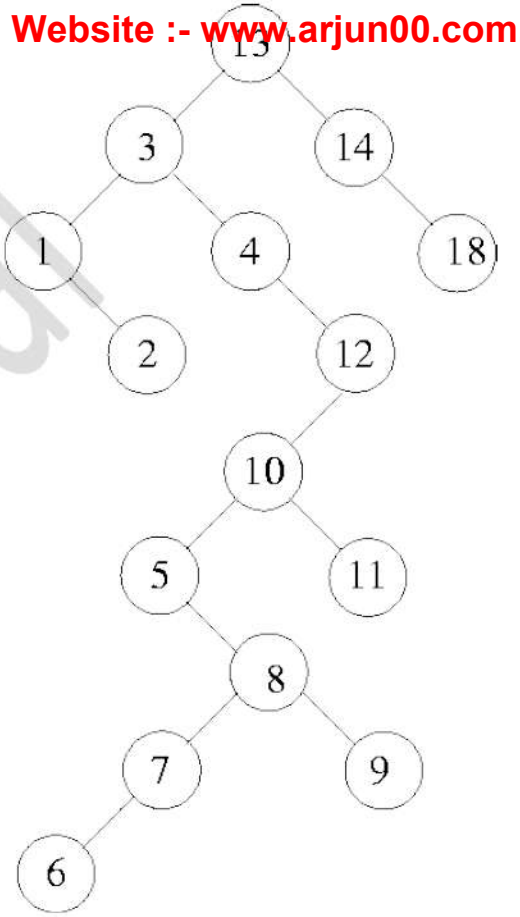
Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.

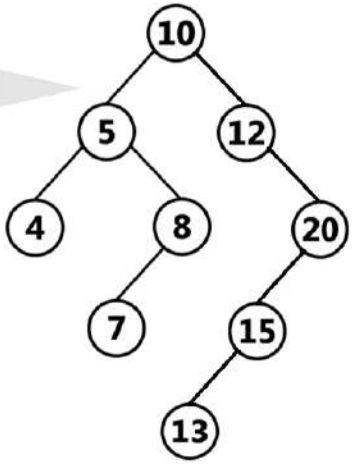


Now, the creation of binary search tree is completed

Create a binary search tree where data elements are:
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



Create a binary search tree where data elements are:
10,12,5,4,20,8,7,15,13



5.7 Tree Traversals

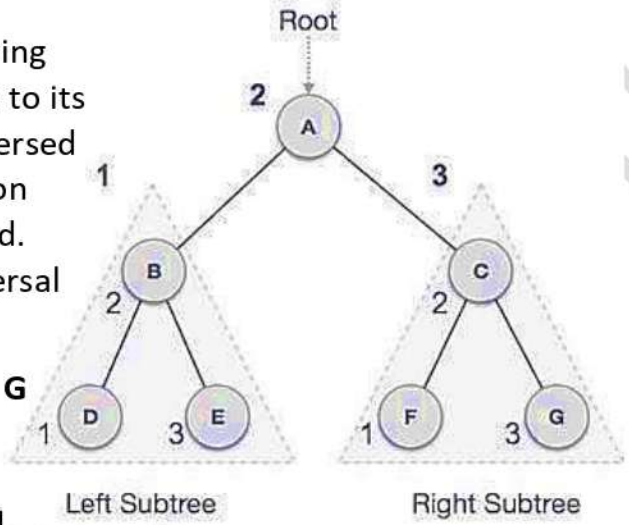
Traversal is a process to visit all the nodes of a tree. Because, all nodes are connected via edges (links) we always start from the root (head) node. We cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

1. In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be



D → B → E → A → F → C → G

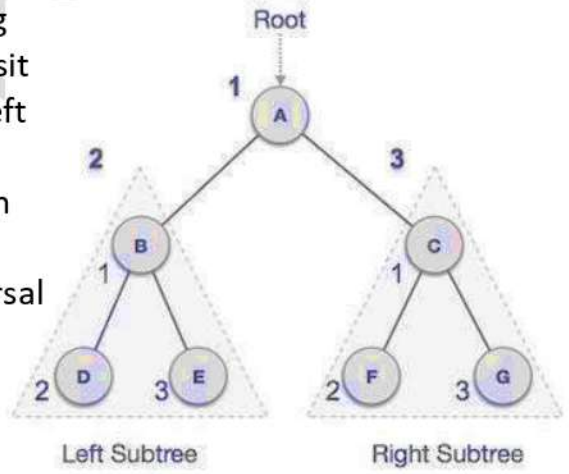
Algorithm:

Until all nodes are traversed
Step 1 – Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 – Recursively traverse right subtree.

2. Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be



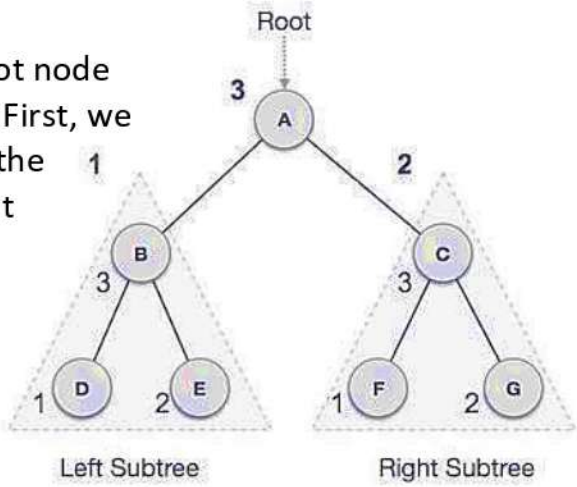
A → B → D → E → C → F → G

Algorithm:

Until all nodes are traversed
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.

3. Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be

D → E → B → F → G → C → A

Algorithm:

Until all nodes are traversed

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

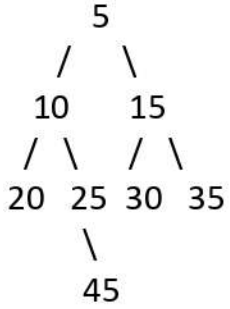
Step 3 – Visit root node.

5.8 Height, level and depth of tree

- The **depth** of a node is the number of edges present in path from the root node of a tree to that node.
- The **height** of a node is the number of edges present in the longest path connecting that node to a leaf node.

Example:

Input: K = 25,



Output:

Depth of node 25 = 2

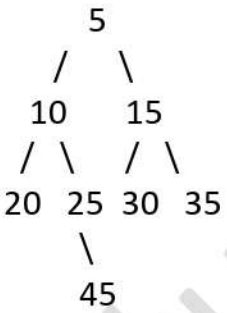
Height of node 25 = 1

Explanation:

The number of edges in the path from root node to the node 25 is 2. Therefore, depth of the node 25 is 2.

The number of edges in the longest path connecting the node 25 to any leaf node is 1. Therefore, height of the node 25 is 1.

Input: K = 10,



Output:

Depth of node 10 = 1

Height of node 10 = 2

5.9 AVL balance tree

5.9.1 Definition

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

5.9.2 Detection of unbalance

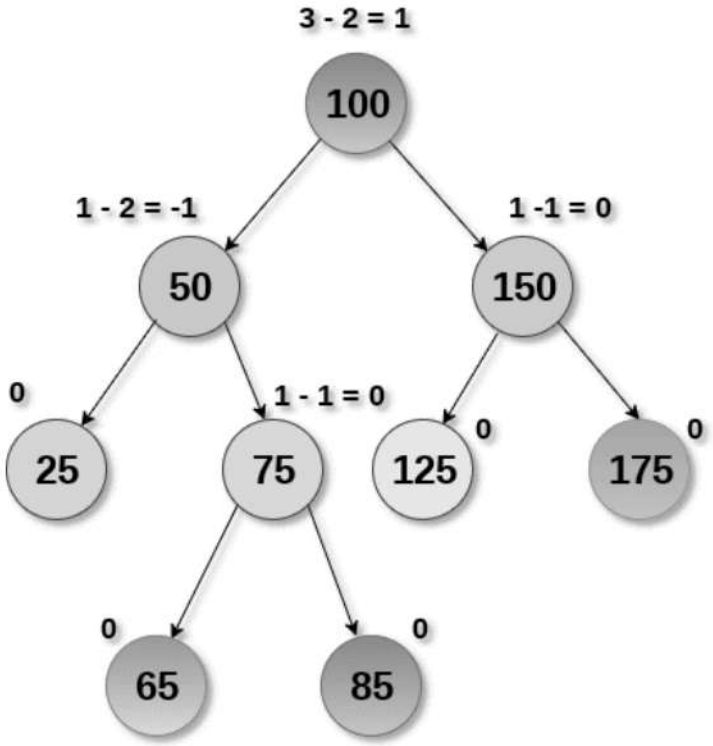
Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

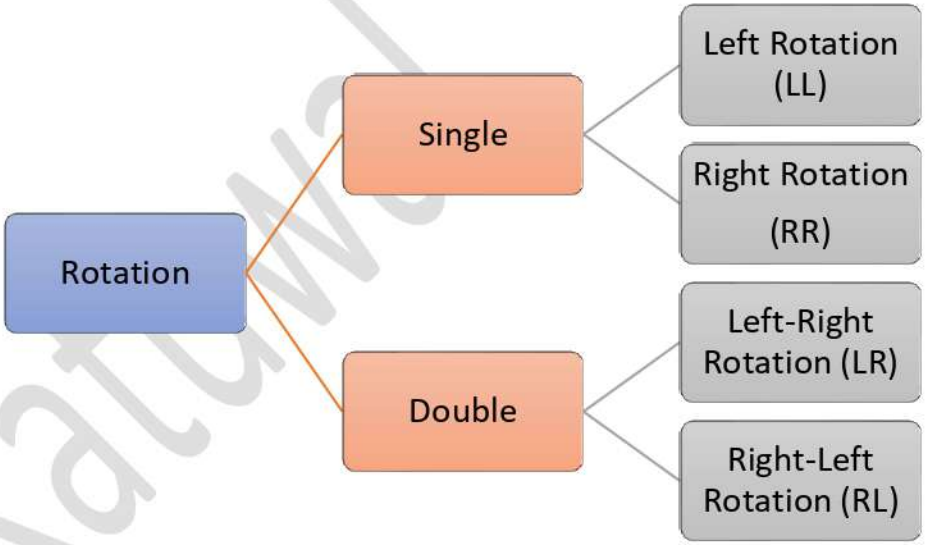
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1.



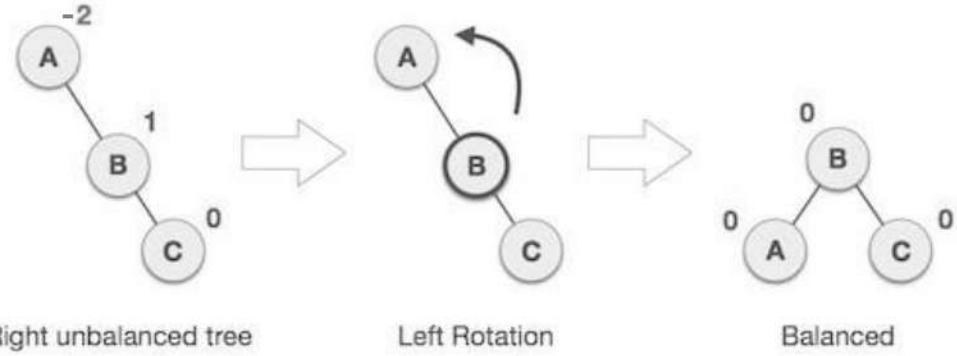
AVL Tree

5.9.2. Single and Double Rotation in Balancing



1. RR Rotation

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2.

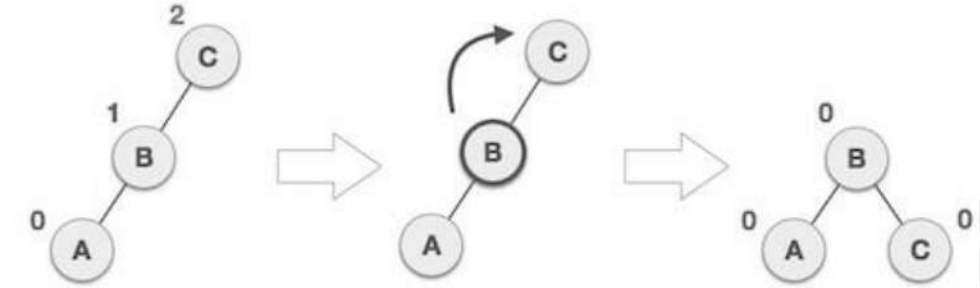


Right unbalanced tree | Left Rotation | Balanced

In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

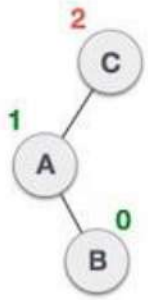
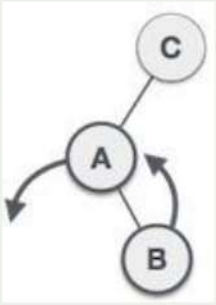
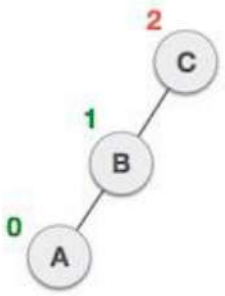
When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

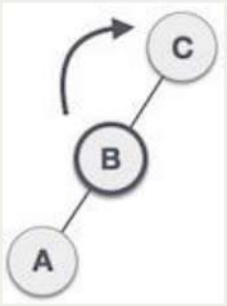
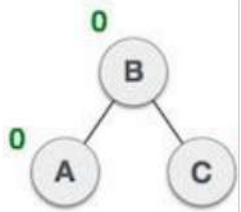


In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

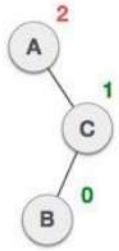
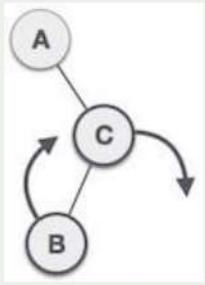
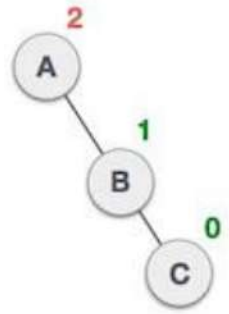
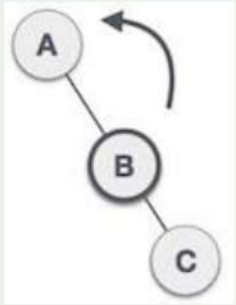
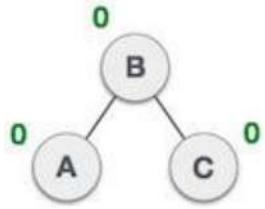
LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node **B** has been inserted into the right subtree of **A** the left subtree of **C**, because of which **C** has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of **C**
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at **A** is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node **C** is still unbalanced, i.e., having balance factor 2, as inserted node **A** is in the left of left of **C**

	Now we perform LL clockwise rotation on full tree, i.e. on node **C**. node **C** has now become the right subtree of node **B**, **A** is left subtree of **B**
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which **A** has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of **A**
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node **A**.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node **A**. node **C** has now become the right subtree of node **B**, and node **A** has become the left subtree of **B**.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Unit:6 – Sorting:

6.1 Definition:

Sorting in data structure refers to the process of arranging data in a specific order, typically in ascending or descending order. This is an essential operation in computer science and is used in various applications such as searching, data analysis, and database management.

6.2 Types of sorting:

Internal Sorting: Internal sorting refers to the sorting of data that can be performed entirely within the computer's main memory. This means that the entire dataset to be sorted can fit into the RAM of the computer. Internal sorting algorithms are designed to work efficiently with the available memory and are typically faster than external sorting algorithms.

Some examples of internal sorting algorithms include:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

External Sorting: External sorting, on the other hand, is used when the dataset to be sorted is too large to fit into the main memory of a computer. In this case, the data is stored on external storage devices such as hard drives or SSDs. External sorting algorithms are designed to minimize the number of input/output operations, as reading and writing to external storage is much slower than accessing data in main memory.

One common technique for external sorting is the use of merge sort, which involves dividing the data into smaller chunks that can fit into memory, sorting these chunks internally, and then merging them back together in the correct order.

Overall, external sorting is more complex and slower than internal sorting due to the additional overhead of reading and writing to external storage devices.

6.3 Algorithm of Bubble Sort:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Algorithm:

1. Start at the beginning of the list.
2. Compare the first two elements. If the first element is greater than the second element, swap them.
3. Move to the next pair of elements, and repeat the process until the end of the list is reached.
4. If a swap was made in the previous pass, repeat the process for the entire list again.

5. Continue this process until no more swaps are needed, which indicates that the list is sorted.

Sort the following element:

13	32	26	35	10

First Pass

Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10

Here, 32 is greater than 13 (32 > 13), so it is already sorted.
Now, compare 32 with 26.

13	32	26	35	10

Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

13	26	32	35	10

Now, compare 32 and 35.

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.

13	26	32	35	10

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

13	26	32	10	35

Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

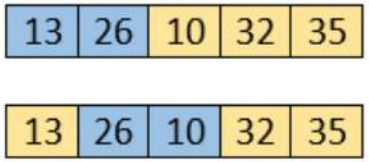
13	26	10	32	35

13	26	10	32	35

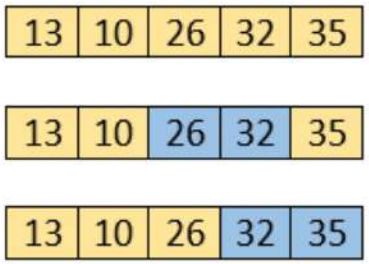
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



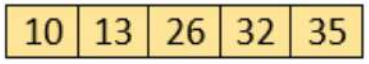
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

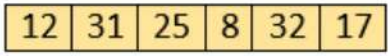
6.4 Algorithm of Insertion Sort:

Insertion Sort is another simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is efficient for small datasets and is often used as part of more complex sorting algorithms.

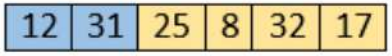
Algorithm:

1. Start with the second element of the array and compare it with the first element. If the second element is smaller, swap them.
2. Move to the third element and compare it with the second and first elements, swapping as necessary to move the third element into its correct position.
3. Continue this process for each subsequent element, comparing it with the elements before it and swapping as needed to place it in its correct position.
4. Repeat until the entire array is sorted.

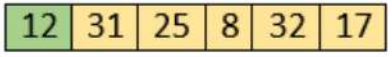
Sort the following elements:



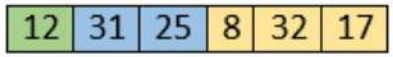
Initially, the first two elements are compared in insertion sort.



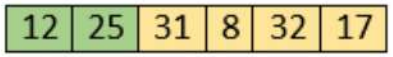
Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



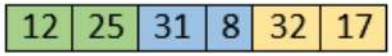
Now, move to the next two elements and compare them.



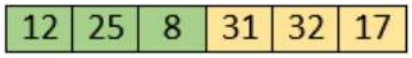
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

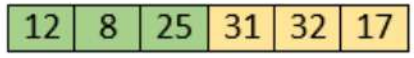


Both 31 and 8 are not sorted. So, swap them.



After swapping, elements 25 and 8 are unsorted.

So, swap them.



Now, elements 12 and 8 are unsorted.

So, swap them too.

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Move to the next elements that are 32 and 17.

17 is smaller than 32. So, swap them.

Swapping makes 31 and 17 unsorted. So, swap them too.

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Now, the array is completely sorted.

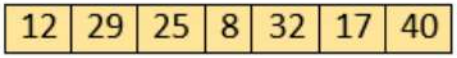
6.5 Algorithm of Selection Sort:

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted portion of the array and swapping it with the first unsorted element.

Algorithm:

1. Find the minimum element in the unsorted portion of the array and swap it with the first unsorted element.
2. Move the boundary between the sorted and unsorted portions of the array one element to the right.
3. Repeat steps 1 and 2 until the entire array is sorted.

Sort the following elements:



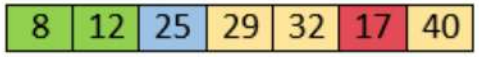
Now, for the first position in the sorted array, the entire array is to be scanned sequentially.
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

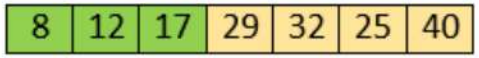
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

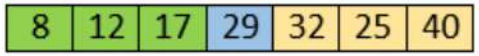
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

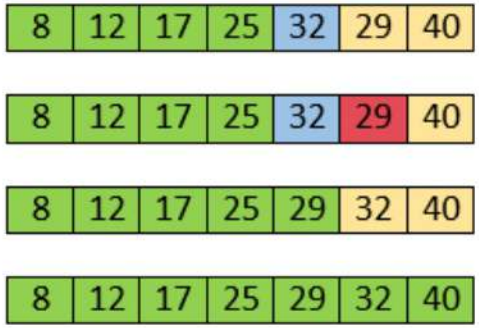
Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

The same process is applied to the rest of the array elements. Now, showing a pictorial representation of the entire sorting process.









Now, the array is completely sorted.

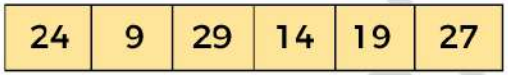
6.6 Algorithm for Quick Sort:

Quick Sort is a widely used efficient sorting algorithm that uses a divide-and-conquer approach to sort an array. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

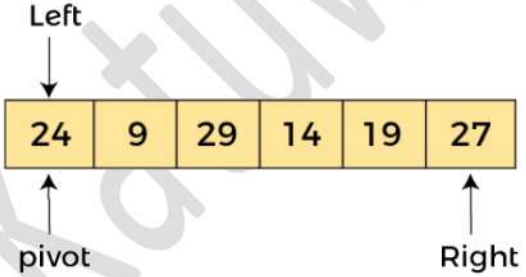
Algorithm:

1. Choose a pivot element from the array. There are various ways to choose the pivot, such as selecting the first element, the last element, the middle element, or using a random element.
2. Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Recursively apply the above steps to the sub-arrays.
4. Combine the sorted sub-arrays to get the final sorted array.

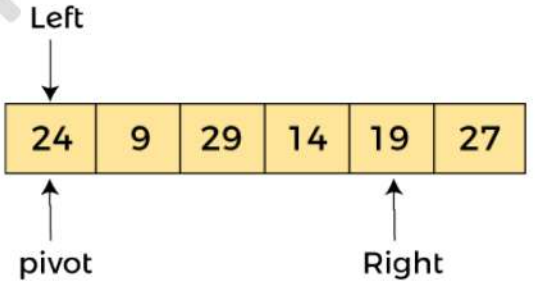
Sort the following elements:



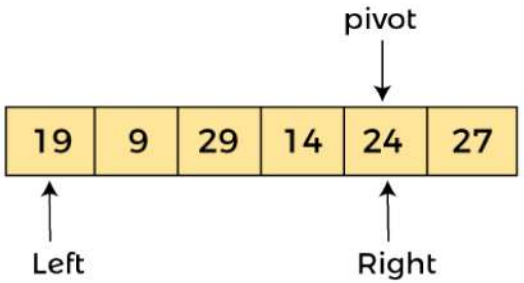
In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24. Since, pivot is at left, so algorithm starts from right and move towards left.



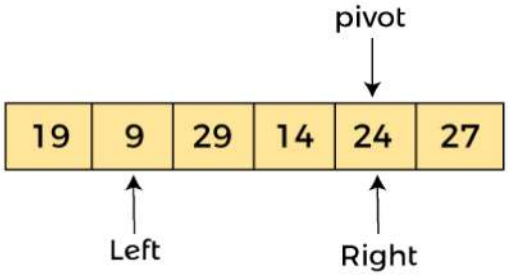
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



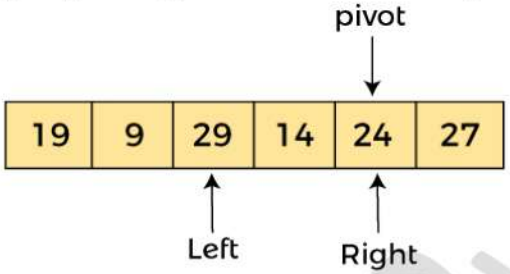
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24. Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



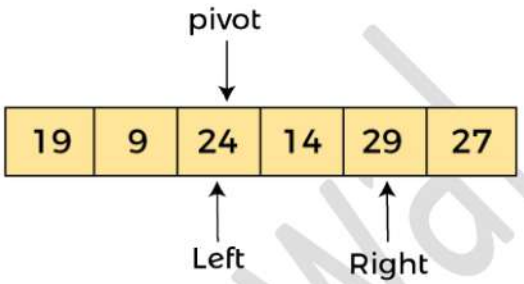
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
As a[pivot] > a[left], so algorithm moves one position to right as -



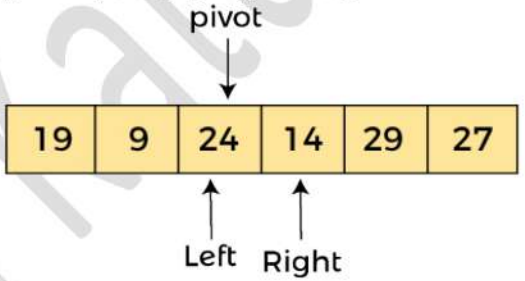
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



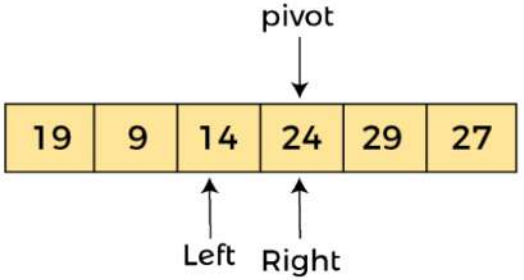
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



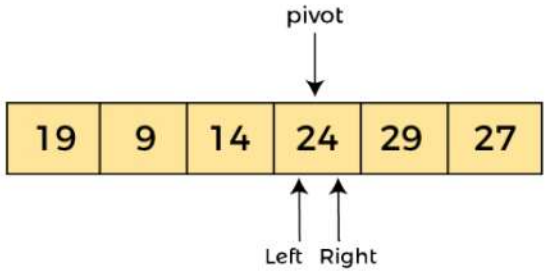
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



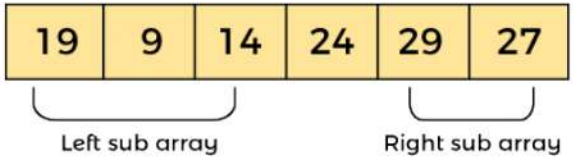
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



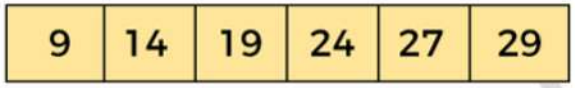
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
Element 24, which is the pivot element is placed at its exact position.
Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be



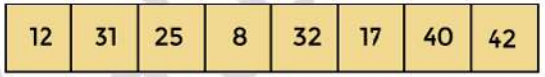
6.7 Algorithm for Merge Sort:

Merge Sort is a popular sorting algorithm that follows the divide-and-conquer approach to sort an array. It works by dividing the array into smaller sub-arrays, sorting them, and then merging them back together.

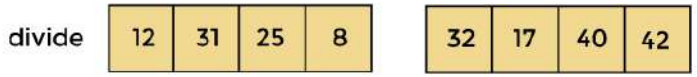
Algorithm:

1. Divide the unsorted array into two halves.
2. Recursively sort the two halves.
3. Merge the sorted halves to produce the final sorted array.

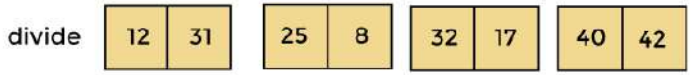
Sort the following elements:



According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.
As there are eight elements in the given array, so it is divided into two arrays of size 4.



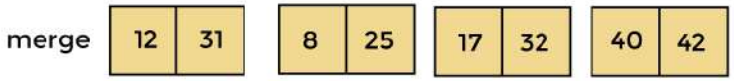
Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



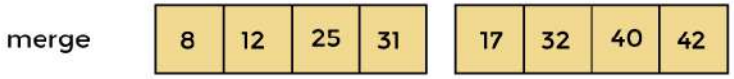
Now, again divide these arrays to get the atomic value that cannot be further divided.

Now, combine them in the same manner they were broken.

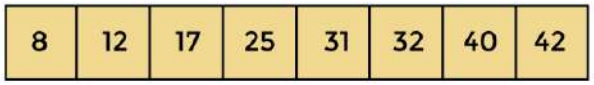
In combining, first compare the element of each array and then combine them into another array in sorted order.
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



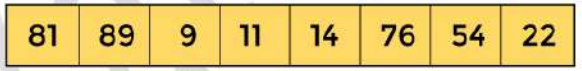
6.8 Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. It is an in-place algorithm, meaning it does not require additional space for sorting.

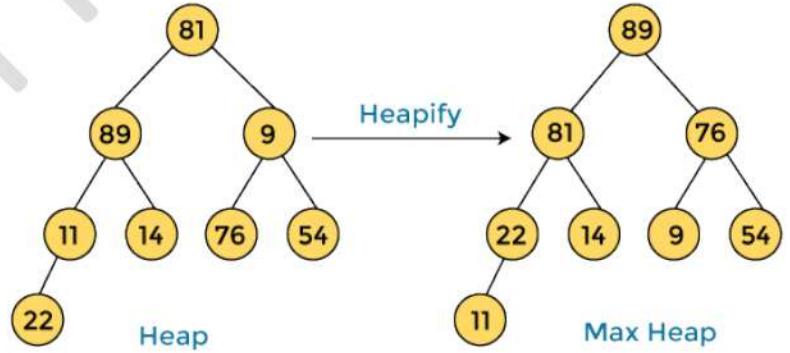
Algorithm:

1. Build a max heap from the input array. This rearranges the elements so that they satisfy the heap property, where the value of each node is greater than or equal to the values of its children (for a max heap).
2. Swap the root element (which is the maximum element in a max heap) with the last element in the heap and reduce the size of the heap by 1.
3. Restore the heap property by performing heapify on the reduced heap.
4. Repeat steps 2 and 3 until the heap is empty.

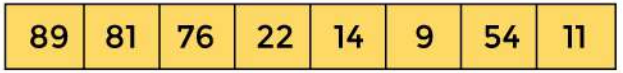
Sort the following elements:



First, we have to construct a heap from the given array and convert it into max heap.

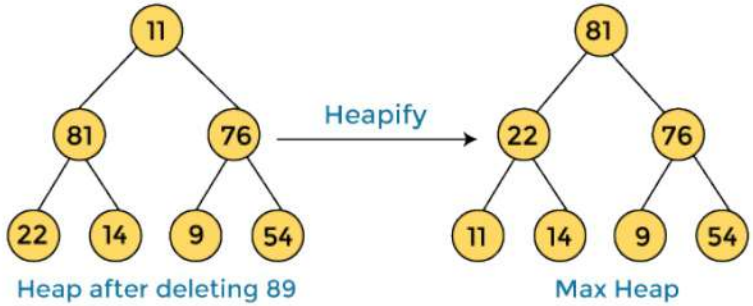


After converting the given heap into max heap, the array elements are -



Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last

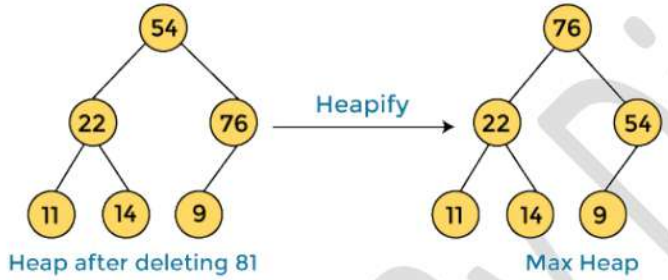
node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89

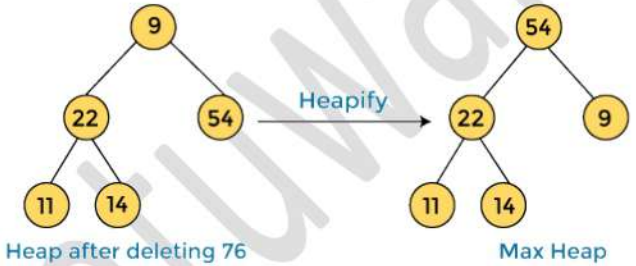
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89

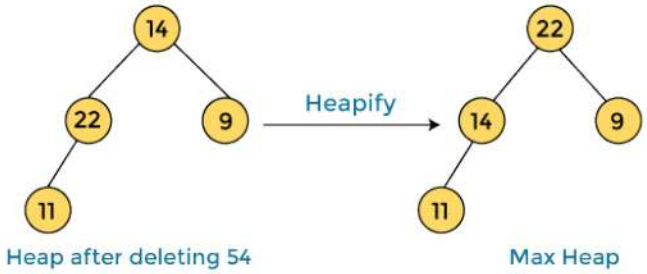
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

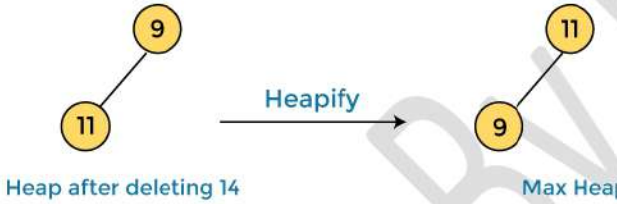
22	14	9	11	54	76	81	89

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -

9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89

Unit:7 – Searching:

Data structure searching is the process of finding a particular item in a data structure such as an array, linked list, or tree. There are various searching algorithms that can be used to efficiently search for an item within a data structure, such as linear search, binary search, and hash tables.

7.1 Sequential Searching

Sequential search, also known as linear search, is a simple searching algorithm that checks every element in a data structure one by one until the desired element is found. It works for both sorted and unsorted data structures.

The steps used in the implementation of Linear Search:

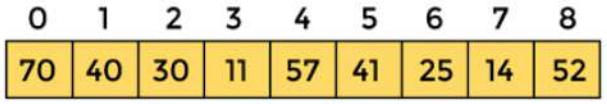
- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -
  - If the element matches, then return the index of the corresponding array element.
  - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1.**

Working of Linear search

Now, let's see the working of the linear search Algorithm.

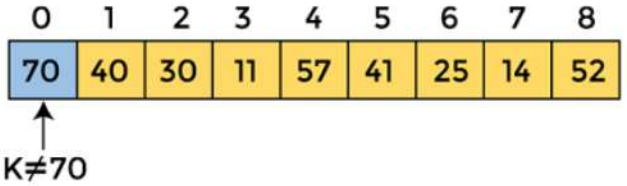
Let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

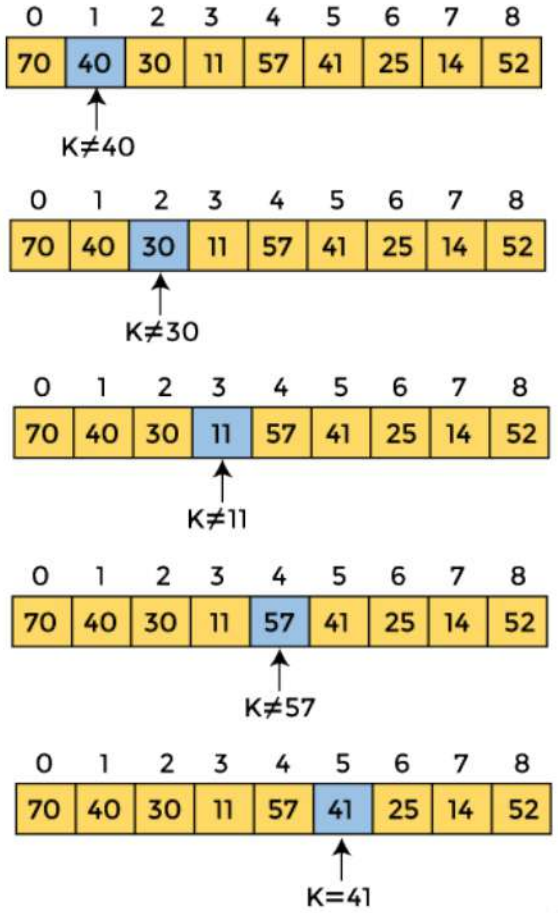


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So, it will return the index of the element matched, Ie. array[5].

7.2 Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.

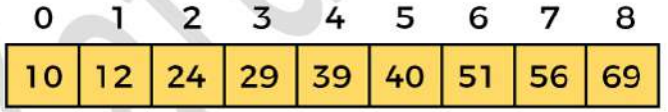
Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

Let's take a sorted array. It will be easy to understand the working of Binary search with an example.

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

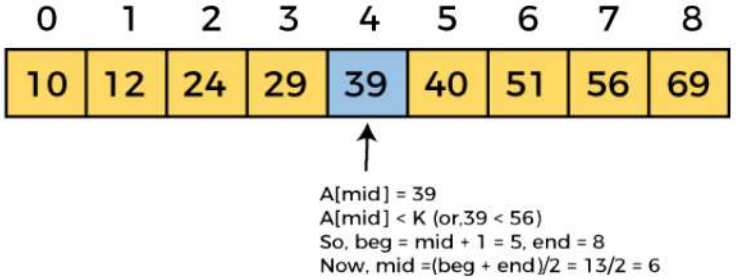
mid = (beg + end)/2

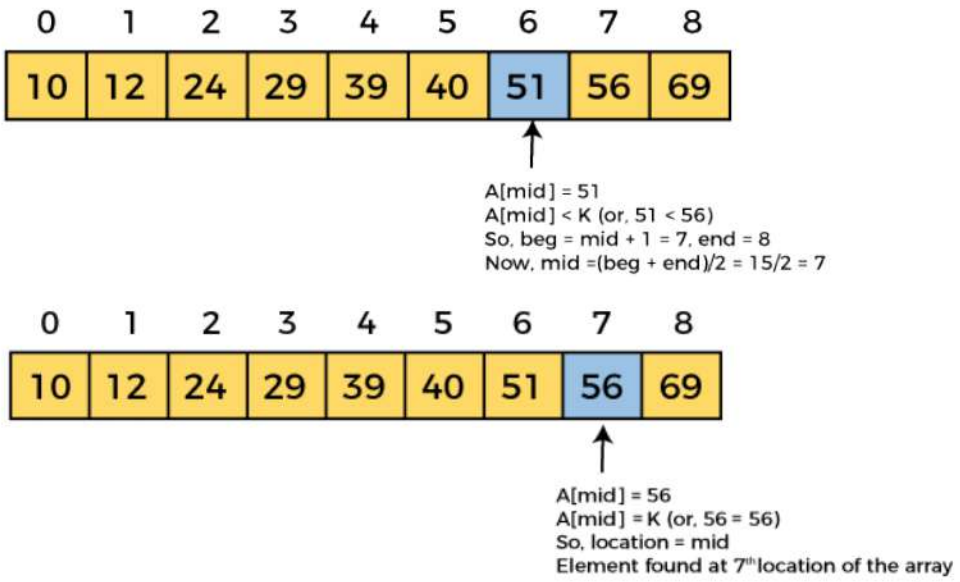
So, in the given array -

beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.





Now, the element to search is found. So, it will return the index of the element matched, ie. array[7].

7.3 Tree Search Algorithm

Discussed in Unit :5

7.4 Hashing

7.4.1 Definition

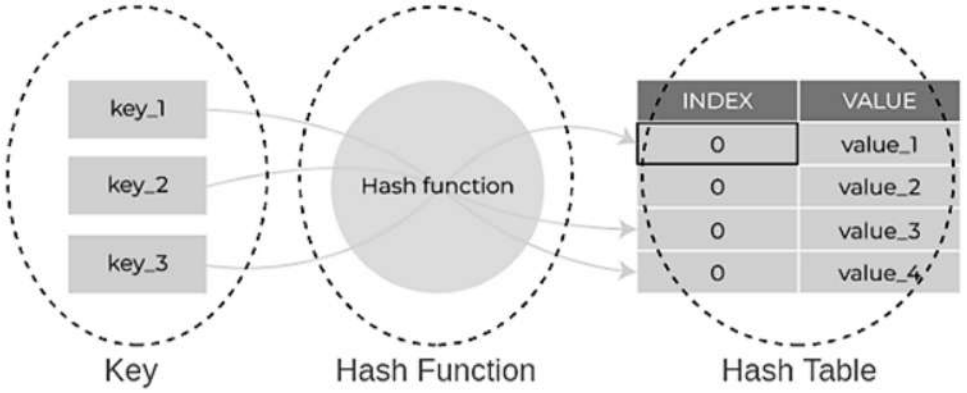
Data structure hashing is a technique used to map data of any size to a fixed-size values, typically for faster access and retrieval. It involves using a hash function to generate a unique value for the input data, which is then used as an index to store or retrieve the data from a data structure such as a hash table.

Hashing is commonly used in computer science for tasks such as indexing, data retrieval, and data storage. It is used in various applications such as databases, file systems, and cryptography.

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



How does Hashing work?

Suppose we have a set of strings {"ab", "cd", "efg"} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly and we are not concerned about the ordering of strings in the table. So, the given set of strings can act as a key and the string itself will act as the value of the string.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
  - "a" = 1,
  - "b"=2, ... , to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
  - *"ab" = 1 + 2 = 3,*
  - *"cd" = 3 + 4 = 7 ,*
  - *"efg" = 5 + 6 + 7 = 18*
- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.
- **Step 5:** So we will then store
  - "ab" in 3 mod 7 = 3,
  - "cd" in 7 mod 7 = 0, and
  - "efg" in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	efg		

Mapping key with indices of array

The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location.

7.4.2 Hash Function

A hashing function in data structure is a function that takes an input (or "key") and returns a unique value (or "hash") that represents that input. This hash value is typically used to index and retrieve data in a hash table, which is a data structure that stores key-value pairs.

Hashing functions are designed to be fast and efficient, and they should ideally produce a unique hash value for each input. However, it is possible for two different inputs to produce the same hash value, which is known as a "collision".

Hash Function Techniques:

a. Truncation method:-

In this method a part of key is considered as address, it can be some rightmost digit or leftmost digit.

Q. apply the truncation method to get the hash index of table size of 100 for following keys.

82394561, 87139465, 83567271, 85943228

Table size = 100
Then,
Take 2 rightmost digits for getting the hash table address.

Key value	address
82394561	61
87139465	65
83567271	71
85943228	28

Example. h(key) = a
h(823994561) = 61

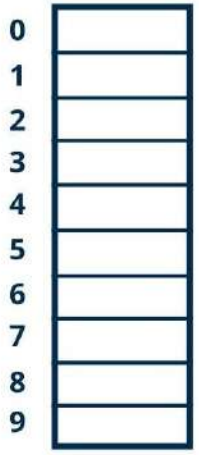
b. Mid square method:-
In this method we square the key , after getting number we take some middle of that number as an address, suppose keys are 4 digits and maximum.
address = 100.

Key value	square	address	
1456	021**19**936	19	h(1456) = 19
1892	035**79**664	79	h(1892) = 79

c. Digital folding method:-
Divide the key-value k into a number of parts i.e., k1, k2, k3, …., kn, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
Add the individual parts.
k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3 = 12 + 34 + 5 = 51 h(12345) = 51

7.4.3 Collision in Hashing
In hashing, collision occurs when two different inputs produce the same hash value. This can happen due to the limited range of hash values and the potentially infinite range of input values.

Example to Understand Collision
In the figure, we have a hash table and the size of the below hash table is 10.



It means this hash table has 10 indexes which are denoted by {0, 1,2,3,4,5,6,7,8,9}

Now we have to insert value **{9,7,17,13,12,8}** into a hash table. So, to calculate slot/index value to store items we will use hashing concept.
And hash function is **h(k) = h(k) mod m**

Step: 1
First Draw an empty hash table of Size 10.
The possible range of hash values will be [0, 9].

Step: 2
First Key to be inserted in the hash table = 9.
h(k) = h(k) mod m
h(9) = 9 mod 10 = 9
So, key 9 will be inserted at index 9 of the hash table

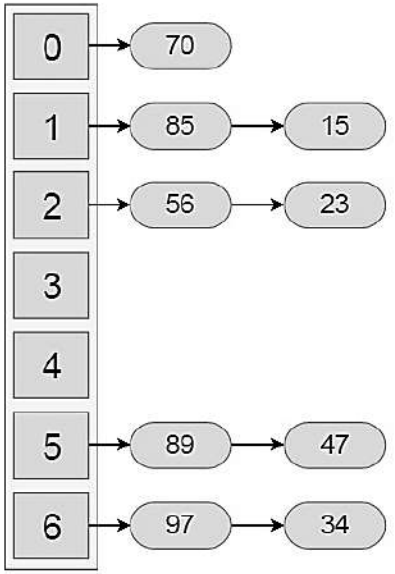
Step: 3
Second Key to be inserted in the hash table = 7.
h(k) = h(k) mod m
h(7) = 7 mod 10 = 7
So, key 7 will be inserted at index 7 of the hash table

Step: 4
Second Key to be inserted in the hash table = 17.
h(k) = h(k) mod m
h(7) = 17 mod 10 = 7
So, key 17 will be inserted at index 7 of the hash table. But here at index 7 already there is a key 7.
So, this is a situation when we can say a collision has occurred.

0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	9

x	h(x) = x mod 7
15	1
47	5
23	2
34	6
85	1
97	6
56	2
89	5
70	0

The Hashing with chaining will be



7.4.4 Collision Resolution Techniques:

There are generally two types of collision resolution techniques:

1. Open hashing.
2. Closed hashing.

1. Open hashing or chaining

Open hashing or more widely known as chaining is one of the simplest approaches to avoid collision in hash tables. In open hashing, each hash table slot, also known as a bucket, contains a linked list of elements that hash to the same slot. The term chaining is used to describe the process because the linked list is used, which is like a chain of elements.

Example:
we have some elements like {15, 47, 23, 34, 85, 97, 65, 89, 70}, Since we have hash table of length 7. Then, our hash function is h(x) = x mod 7.

2. Closed Hashing

Like open hashing, closed hashing is also a technique used for collision resolution in hash tables. Unlike open hashing, where collisions are resolved by chaining elements in separate

chains, closed hashing aims to store all elements directly within the hash table itself without the use of additional data structures like linked lists.

In closed hashing, when a collision occurs, and a hash slot is already occupied, the algorithm probes for the next available slot in the hash table until an empty slot is found.

The probing process can be done in various ways, such as linear probing, quadratic probing, or double hashing.

1. Linear probing:-

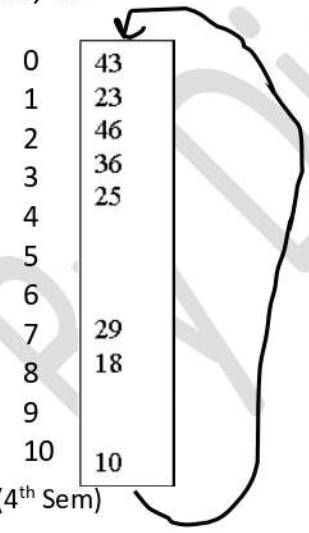
This hashing technique finds the hash key value through hash function and maps the key on the particular position in hash in hash, table. In case if key has same hash address, then it will find the next empty position in the hash table, we take the hash table as circular array. If table size in n, then after n-1 position it will search form zeroth position in the array.

e.g., consider table size 11 & elements are

29, 18, 23, 10, 36, 26, 46, 43

Sol :-

H(29) = 29%11 = 7
H(18) = 18 %11= 7
H(23) = 23%11 = 1
H(10) = 10%11= 10
H(36) = 36%11= 3
H(25) = 25%11= 3
H(46) = 46%11= 2
H(43) = 43%11= 10



Quadratic probing:-

In case of collision,
Rehash functions : (Hash value + 1^2) % size
(Hash value + 2^2) % size & so on.

Example:

table size = 11
given elements = 29, 18, 43, 10, 46, 54
H(29) = 29% 11= 7
H(18) = 18%11 = 7
(Hash value + 1^2) %11
(7 + 1) % 11 = 8
H(43) = 43%11= 10
H(10) = 10 % 11 = 10
(Hash value + 1^2) %11=0
H(46) = 46%11 = 2
H(54) = 54%11 = 10
(10 + 1^2) %11 = 0
(10 + 2^2) % 11= 3

0	10
1	
2	46
3	54
4	
5	
6	
7	29
8	18
9	
10	43

3. Double hashing :-

This technique requires hashing second time in case of collision. Suppose h is a hash key then in case of collision we will again do the hashing of this hash key.
i.e., Hash(h) = h'
First hash function is typically hash1(key) = key % TABLE_SIZE
A popular second hash function is **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.

Formula:

doub_hash= h1(k) + I * h2(k)

h1(k) = k mod tsize

h2(k) = 7- (k mod 7)

example:

elements: 20, 34, 45, 70,56

table size=11

k	h1	h2	doub_hash		
20	9	1	9+0x1=9	no collision	
34	1	1	1+0x1=1	no collision	
45	1	4	1+0x4=1	collision	1+1x4=1+4=5
70	4	7	4+0x7=4	no collision	
56	1	7	1+0x7=1	collision	1+1x7= 1+7=8

Address	element
0	
1	34
2	
3	
4	70
5	45
6	
7	
8	56
9	20
10	

Unit:8 – Graph:

8.1. Components of Graph

A graph is an abstract data type (ADT) that consists of a set of objects that are connected to each other via links. These objects are called vertices and the links are called edges. Usually, a graph is represented as G = {V, E}, where G is the graph space, V is the set of vertices and E is the set of edges. If E is empty, the graph is known as a forest.

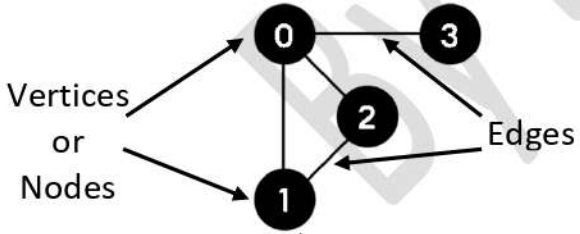
Components of a Graph:

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes.

Edges: Edges are drawn or used to connect two nodes of the graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs.

Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 & 3 are not adjacent because there is no edge between them.

Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

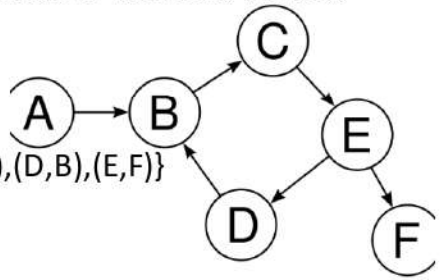


8.2 Directed and Undirected Graph

Directed Graph

When a graph has an ordered pair of vertexes, it is called a directed graph. The edges of the graph represent a specific direction from one vertex to another. When there is an edge representation as (V1, V2), the direction is from V1 to V2. The first element V1 is the initial node or the start vertex. The second element V2 is the terminal node or the end vertex.

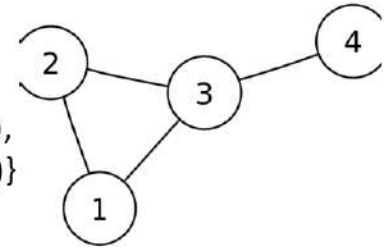
Set of vertices (V): {A, B, C, D, E, F}
Set of edges (E): {(A,B),(B,C),(C,E),(E,D),(D,B),(E,F)}



Undirected Graph:

When a graph has an unordered pair of vertexes, it is an undirected graph. In other words, there is no specific direction to represent the edges. The vertexes connect together by undirected arcs, which are edges without arrows. If there is an edge between vertex A and vertex B, it is possible to traverse from B to A, or A to B as there is no specific direction.

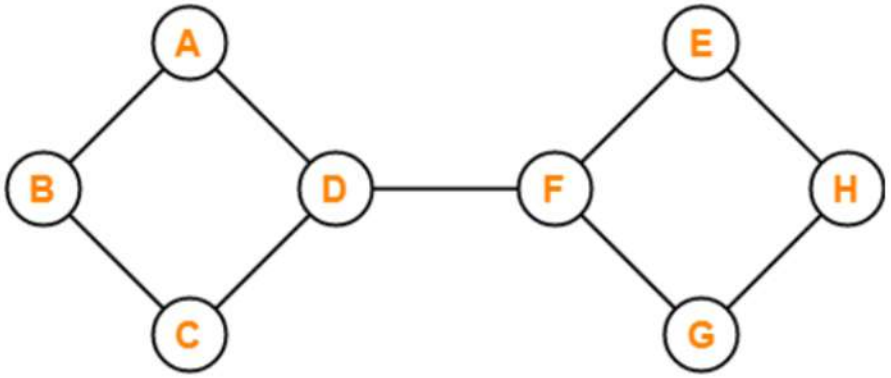
Set of vertices (V): {1, 2, 3, 4}
Set of edges (E): {(1, 2), (2, 1), (2, 3), (3, 2), (1, 3), (3, 1), (3, 4), (4, 3)}



8.3 Connected and Unconnected Graph

Connected Graph:

For a graph to be labelled as a connected graph, there must be at least a single path between every pair of the graph's vertices. In other words, we can say that if we start from one vertex, we should be able to move to any of the vertices that are present in that particular graph, which means there exists at least one path between all the vertices of the graph.

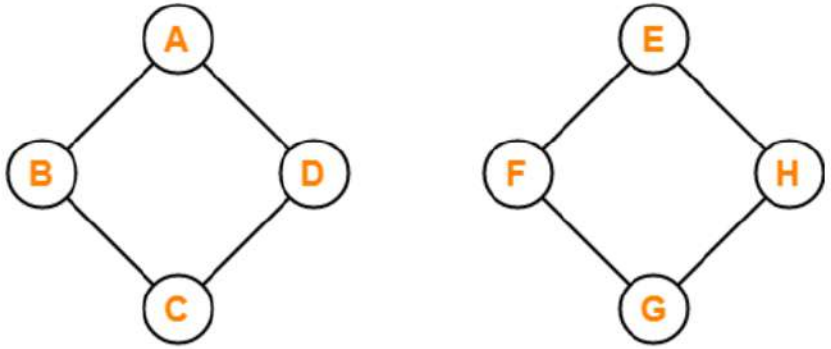


The graph shown above is an example of a connected graph because we start from any one of the vertices of the graph and start moving towards any other remaining vertices of the graph. There will exist at least one path for traversing the graph.

For example, if we begin from vertex B and traverse to vertex H, there are various paths for traversing. One of the paths is **Vertice B -> vertice C -> vertice D -> vertice F -> vertice E -> vertice H.**

Unconnected Graph or Disconnected Graph:

A graph is said to be a disconnected graph where there does not exist any path between at least one pair of vertices. In other words, we can say that if we start from any one of the vertices of the graph and try to move to the remaining present vertices of the graph and there exists not even a single path to move to that vertex, then it is the case of the disconnected graph. If any one of such a pair of vertices doesn't have a path between them, it is called a disconnected graph.



The graph shown above is a disconnected graph. The above graph is called a disconnected graph because at least one pair of vertices doesn't have a path to traverse starting from either node.

For example, a single path between both vertices doesn't exist if we want to traverse from vertex A to vertex G.

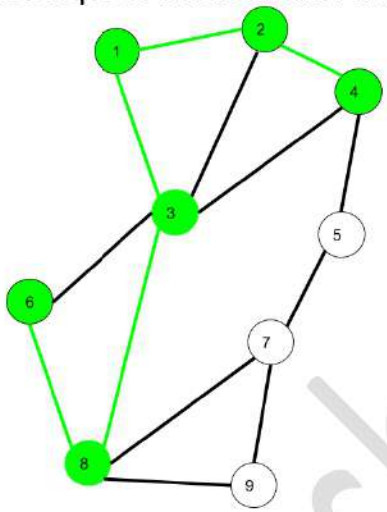
8.4 Path and Cycle

Path:

It is a trail in which neither vertices nor edges are repeated i.e. if we traverse a graph such that we do not repeat a vertex and nor we repeat an edge.

- Vertex not repeated
- Edge not repeated

Here 6->8->3->1->2->4 is a Path



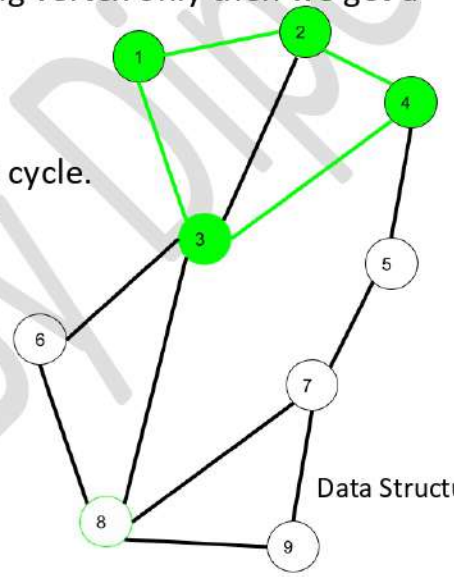
Cycle:

Traversing a graph such that we do not repeat a vertex nor we repeat a edge but the starting and ending vertex must be same i.e. we can repeat starting and ending vertex only then we get a cycle.

- Vertex not repeated
- Edge not repeated

Here 1->2->4->3->1 is a cycle.

Cycle is a closed path. These cannot have repeat anything (neither edges nor vertices). Note that for closed sequences start and end vertices are the only ones that can repeat.



8.5 Adjacency sets and table

In graph theory and data structures, an adjacency set is a way of representing a graph where each vertex is associated with a set of its neighboring vertices. This representation is commonly used for sparse graphs, where the number of edges is much smaller than the number of possible edges.

An adjacency table, on the other hand, is a way of representing a graph as a two-dimensional array or matrix, where the value at position (i, j) represents whether there is an edge between vertex i and vertex j.

8.6 Array based representation

8.7 Linked based representation

Representations of Graph

Here are the two most common ways to represent a graph :

a. Adjacency Matrix
b. Adjacency List

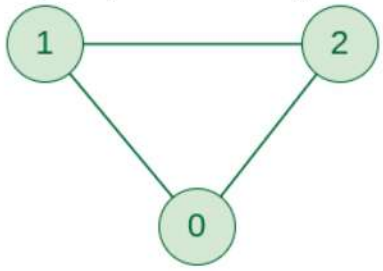
a. Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's).

Let's assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n.

If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
If there is no edge from vertex i to j, mark adjMat[i][j] as 0.

Representation of Undirected Graph to Adjacency Matrix:
The below figure shows an undirected graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases (adjMat[destination] and adjMat[destination]) because we can go either way.



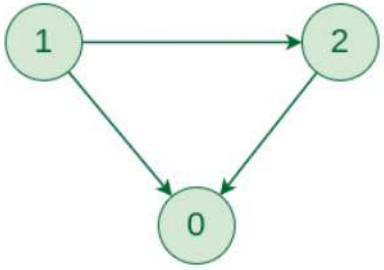
Undirected Graph

	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:
The below figure shows a directed graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 for that particular adjMat[destination].



Directed Graph

	0	1	2
0			
1	1		1
2	1		

Adjacency Matrix

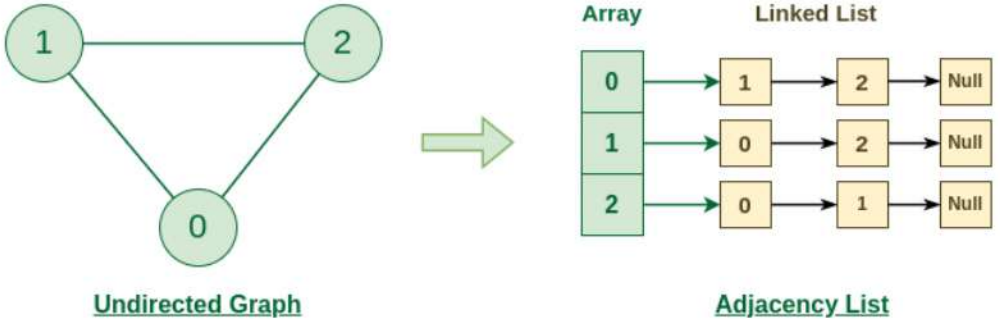
b. Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.

Let's assume there are n vertices in the graph So, create an array of list of size n as adjList[n].

adjList[0] will have all the nodes which are connected (neighbour) to vertex 0.
adjList[1] will have all the nodes which are connected (neighbour) to vertex 1 and so on.

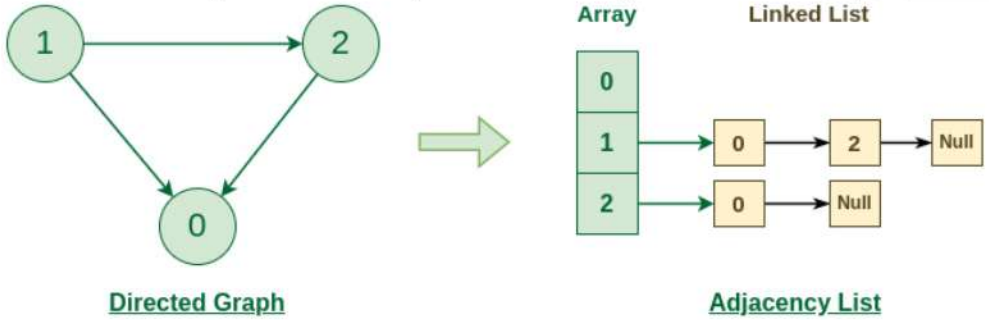
Representation of Undirected Graph to Adjacency list:
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph to Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

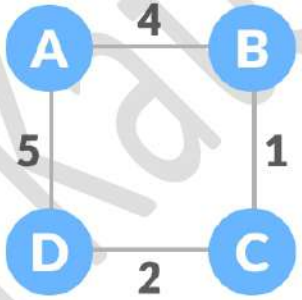
8.8 Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Example of a Spanning Tree

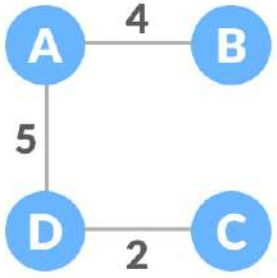
Let's understand the above definition with the help of the example below.

The initial graph is:



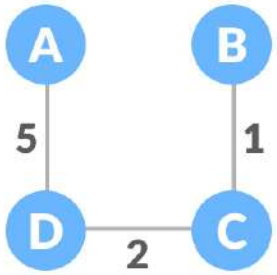
Weighted graph

The possible spanning trees from the above graph are:



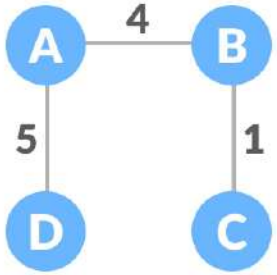
sum = 11

Minimum spanning tree - 1



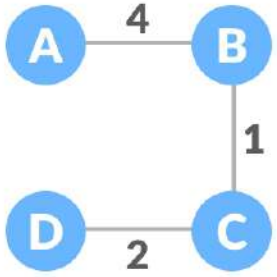
sum = 8

Minimum spanning tree - 2



sum = 10

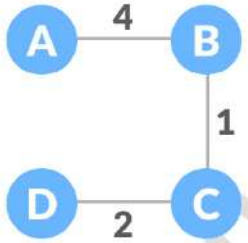
Minimum spanning tree – 3



sum = 7

Minimum spanning tree – 4

The minimum spanning tree from the above spanning trees is:



sum = 7

Minimum spanning tree

8.8.1 Kruskal's Algorithm and Prim's Algorithm

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

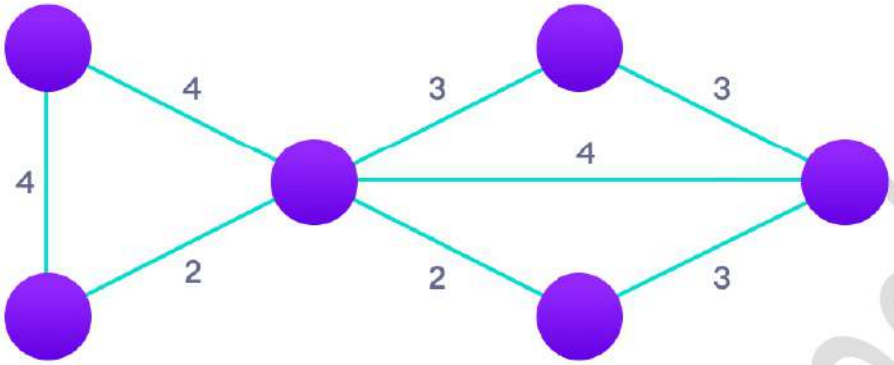
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

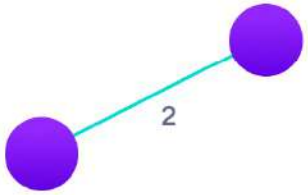
1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



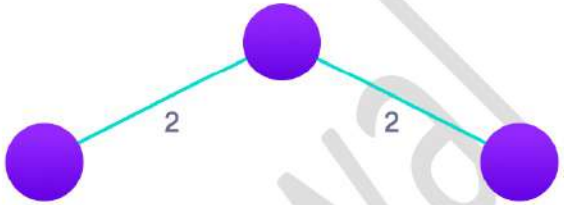
Step: 1

Start with a weighted graph



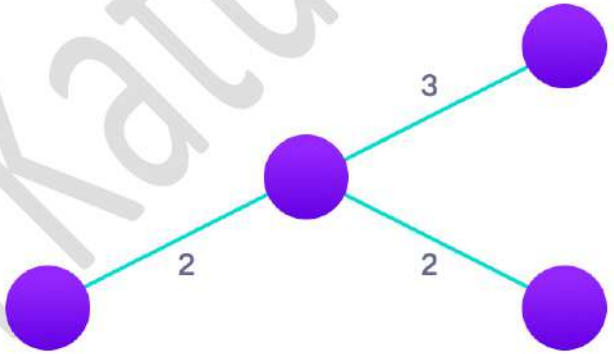
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone.



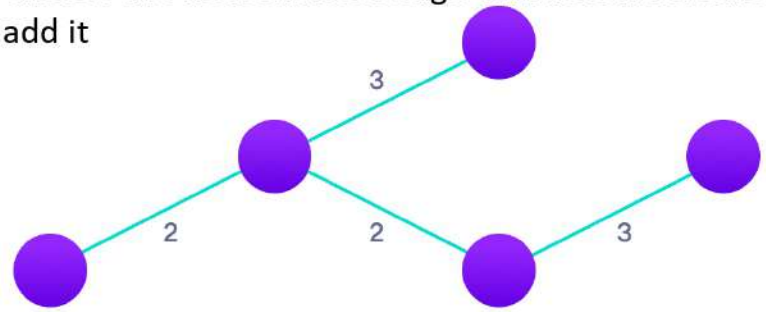
Step: 3

Choose the next shortest edge and add it



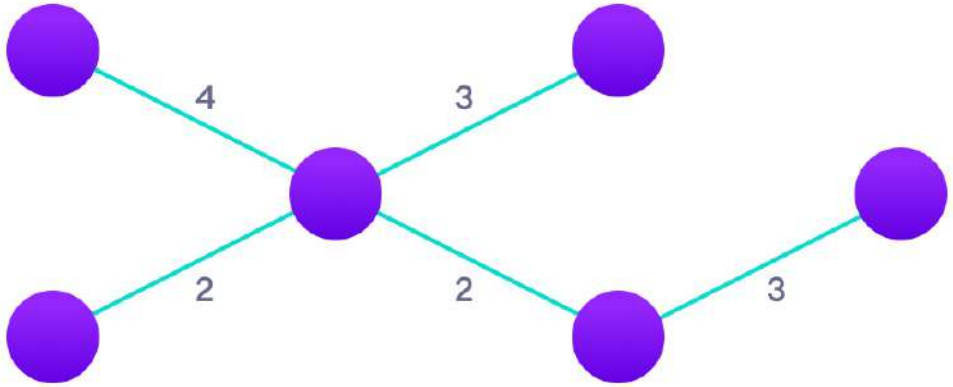
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal's Algorithm Applications:

- In order to layout electrical wiring
- In computer network (LAN connection)

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

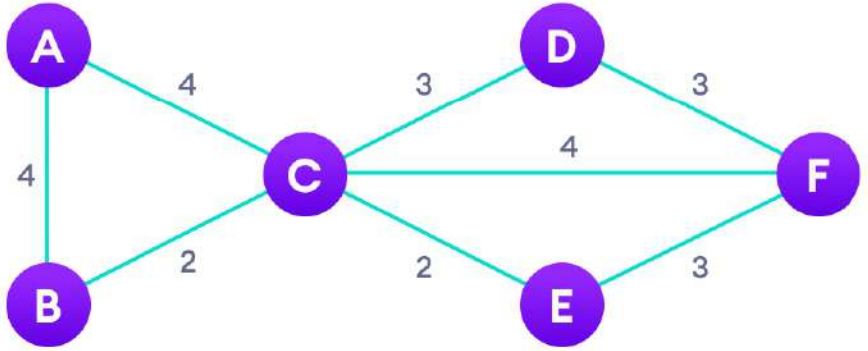
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Example of Prim's algorithm

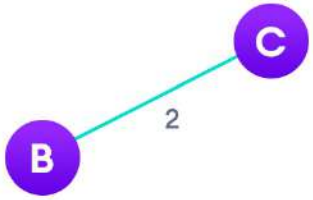


Step: 1

Start with a weighted graph

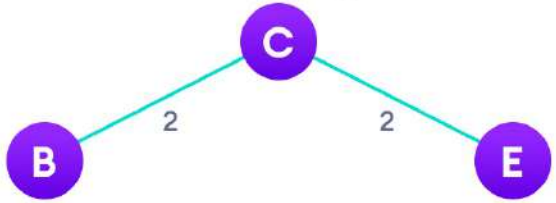
Step: 2

Choose a vertex



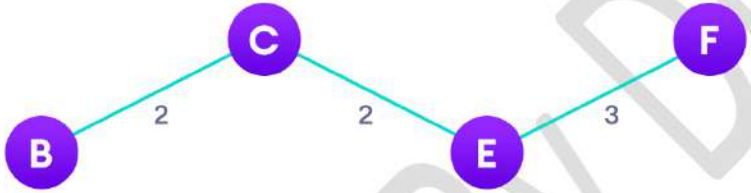
Step: 3

Choose the shortest edge from this vertex and add it



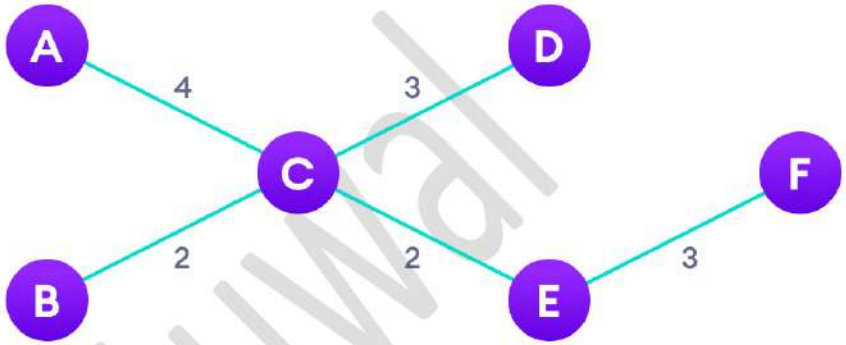
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree