

CS302ES: DATA STRUCTURES

B.TECH II Year I Sem.

L T P C
3 1 0 4

Prerequisites: A course on “Programming for Problem Solving”.

Course Objectives:

- Exploring basic data structures such as stacks and queues.
- Introduces a variety of data structures such as hash tables, search trees, tries, heaps, graphs.
- Introduces sorting and pattern matching algorithms

Course Outcomes:

- Ability to select the data structures that efficiently model the information in a problem.
- Ability to assess efficiency trade-offs among different data structure implementations or combinations.
- Implement and know the application of algorithms for sorting and pattern matching.
- Design programs using a variety of data structures, including hash tables, binary and general tree structures, search trees, tries, heaps, graphs, and AVL-trees.

UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

UNIT - II

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

UNIT - III

Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

UNIT - IV

Graphs: Graph Implementation Methods. Graph Traversal Methods.

Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort.

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

TEXTBOOKS:

1. Fundamentals of Data Structures in C, 2nd Edition, E. Horowitz, S. Sahni and Susan Anderson Freed, *Universities Press*.
2. Data Structures using C – A. S. Tanenbaum, Y. Langsam, and M.J. Augenstein, *PHI/Pearson Education*.

REFERENCE BOOKS:

1. Data Structures: A Pseudocode Approach with C, 2nd Edition, R. F. Gilberg and B.A. Forouzan, Cengage Learning.

UNIT-I: (INTRODUCTION TO DATA STRUCTURES)

Data structure:

A Data structure is a specialized format for organizing, processing, storing and retrieving the data. It is a method of storing and organizing data efficiently in the memory.

DATA STRUCTURE= DATA ELEMENTS+OPERATIONS+ ALGORITHMS

The operations done on data structures:

1. Data organizing or clubbing.
2. Accessing technique.
3. Manipulating selections for information

Flowchart:

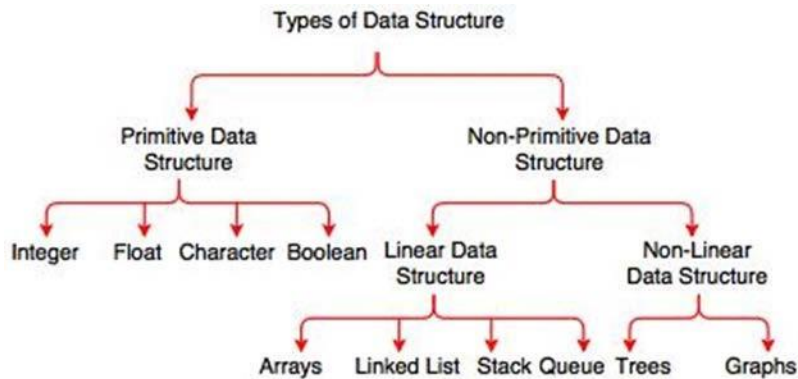


Fig. Types of Data Structure

PRIMITIVE DATA STRUCTURES:

These are directly supported by machines without writing any code.

All primary data types are primitive.

EX; int, float, char...

Non-PRIMITIVE DATA STRUCTURES:

These are not directly supported by machines.

Classified as linear and non-linear.

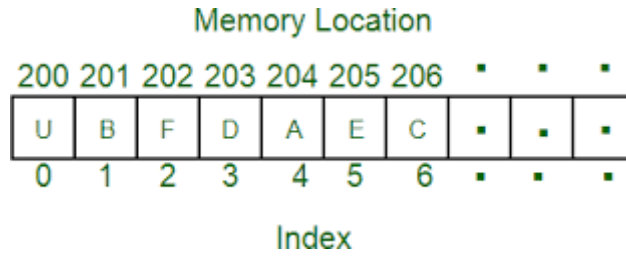
1) Linear data structure:

The data in which the values or information is arranged in a linear fashion is known as “linear data structure”.

List of linear DS: arrays, stacks, queues, and linked lists.

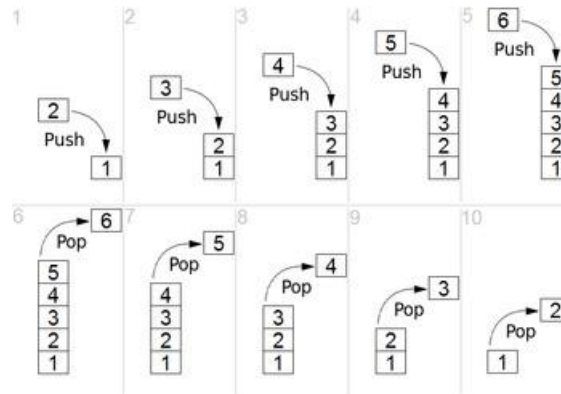
Arrays:

The array is the collection of similar type of data stored at contiguous memory locations.



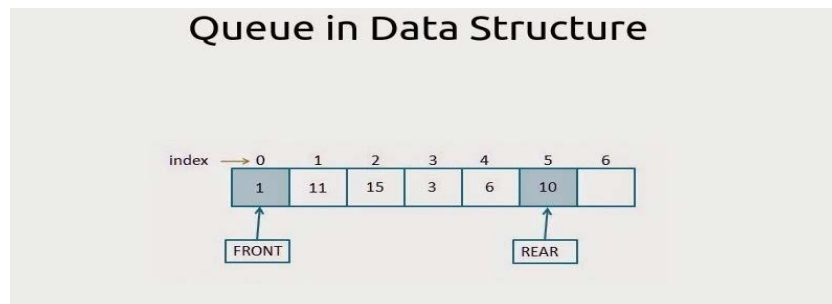
Stacks:

Stack is the non-primitive dynamic linear data structure which follows the F.I.L.O or L.I.F.O principle.



Queues:

Queue is a non-primitive dynamic linear data structure which follows the F.I.F.O or L.I.L.O principle.



Linked lists:

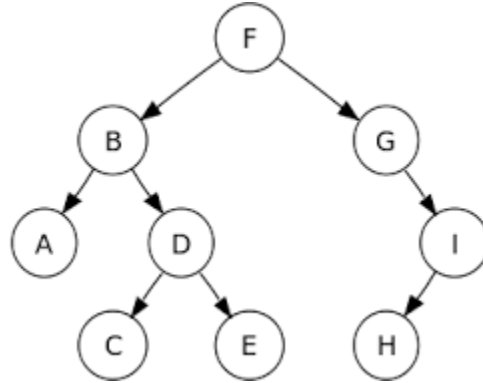
A linked lists is a non-primitive dynamic linear data structure in which the elements are not stored at contiguous memory locations.

2) Nonlinear data structure:

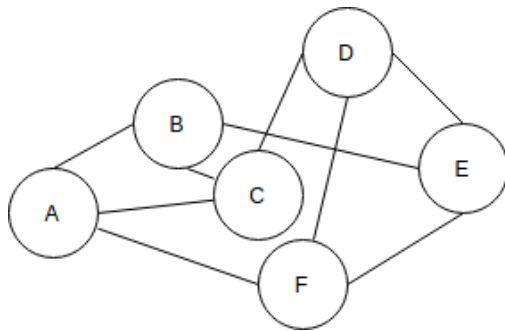
The data structure in which the values in the structure are not arranged in an order is known as “non- linear data structure”. This type is opposite to linear data structure.

List of nonlinear data structures: Trees and Graphs

- **Tree:** Tree is a non-primitive non-linear hierarchical data structure which stores the information naturally in the form of hierarchy style.



Graphs: A Graph is a non-primitive non-linear data structure consisting of nodes and edges.



List of operations:

- 1) **Traversing:** Accessing records exactly once so that certain items in the record may be processed.
- 2) **Searching:** Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.
- 3) **inserting:** Adding the new record to the structure.
- 4) **Deleting:** Removing the record from the structure.
- 5) **Sorting:** Managing the data or record in some logical order.
- 6) **Merging:** Combining the record in two different sorted files into single sorted file.

Applications of data structures:

- 1) **Stack:** Stacks are used to convert infix expression to postfix expression.

In recursion system stack created stores he previous value of recursive function.

2) **Queue:** Queue is used in topological shorting by which appropriate topology to Connect computer and internet.

This is also used in the printer applications.

3) **List:** Lists are used to store huge number of records in sequential order.

4) **Graph:** Graphs are used to establish any network system like LAN, WAN, etc.....

Abstract Data Type (ADT):

Abstraction means representing only the essential things and hiding the implementation details.

The data with its type and set of operations on it (for insertion, handling, manipulation and retrieving it) is referred to as “Abstract Data Type”.

It is a specification of set of data and set of operations that can be performed on data.

Every Data structure has an ADT.

ADT minimizes the dependencies in our code, which helps us to modify our code if needed.

This helps us to hide lower- level details such as representation, execution and limitations from the programmers.

Examples: List, Queue, Stack, String, Tree, Set, etc.....

It consists of

- a) Declaration of data (objects/instances/elements..)
- b) Declaration of operations (functions..)

Example:

ARRAY ADT:
Objects/ Instances/data elements: Int a[]; Int b[]; ...
Operatios: Void read(); Void display(); Void search(); Void reverse() Void insert()

//IMPLEMENTATION OF ARRAY ADT

```
#include<stdio.h>
#include<stdlib.h>
#define max 20
int a[max],n;
void read(int);
void display(int a[], int);
void search(int a[], int);
void reverse(int a[], int);
void merge(int a[], int);
void insert(int a[], int);
main()
{
    int ch,n1;
    system("clear");
    while(1)
    {
        printf("\n_____");
        printf("\n ARRAY ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.READ");
        printf("\n\t2.DISPLAY");
        printf("\n\t3.SEARCH");
        printf("\n\t4.REVERSE");
        printf("\n\t5.INSERT");
        printf("\n\t6.MERGE");
        printf("\n\t7.EXIT");

        printf("\n Enter ur choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:    printf("\n enter size of the array:");
                       scanf("%d",&n);
                       read(n);
                       break;

            case 2: display(a,n);
                       break;
            case 3: search(a,n);
                       break;
            case 4: reverse(a,n);
                       break;
            case 5: insert(a,n);
                       break;
            case 6: merge(a,n);
                       break;
            case 7: exit(0);
                       break;
            default: printf("\n wrong choice\n");
        }
    }
}
```

```

}

void read(int n1)
{
    int i;
    printf("\n Enter array elements:");
    for(i=0;i<n1;i++)
        scanf("%d",&a[i]);
}

void display(int x[], int n1)
{
    int i;
    printf("\n array elements are :");
    for(i=0;i<n1;i++)
        printf("\t%d",x[i]);
}

void search(int a[], int n1)
{
    int i,element,t=0;
    printf("\n Enter element to be searched:");
    scanf("%d",&element);
    for(i=0;i<n;i++)
    {
        if(a[i]==element)
        {
            printf("\n element is found at %d position",i+1);
            t=1;
            break;
        }
    }
    if(t==0)
        printf("\n element not found");
}

void reverse(int a[], int n1)
{
    int i,b[max];
    for(i=0;i<n;i++)
        a[i]=a[n-i-1];

    printf("\n elements in reverse order:");
    display(a,n);
}

void insert(int a[], int n1)
{
    int i,element,pos;
    printf("\n Enter element to be inserted:");
}

```

```

scanf("%d",&element);
    printf("\n Enter the position:");
scanf("%d",&pos);
for(i=n1;i>=pos-1;i--)
{
    a[i]=a[i-1];
}
a[pos-1]=element;
    n++;
    display(a,n);
}

void merge(int a[], int n1)
{
int i,j,b[max],c[max],m;
printf("\n Enter no-of elements in the second array:");
scanf("%d",&m);
printf("\n Enter elements:");
for(i=0;i<m;i++)
    scanf("%d",&b[i]);
for(i=0;i<n1;i++)
    c[i]=a[i];
    for(j=0;j<m;j++)
    {
        c[i]=b[j];
        i++;
    }

    display(c,n+m);
}

```

Stack using arrays:

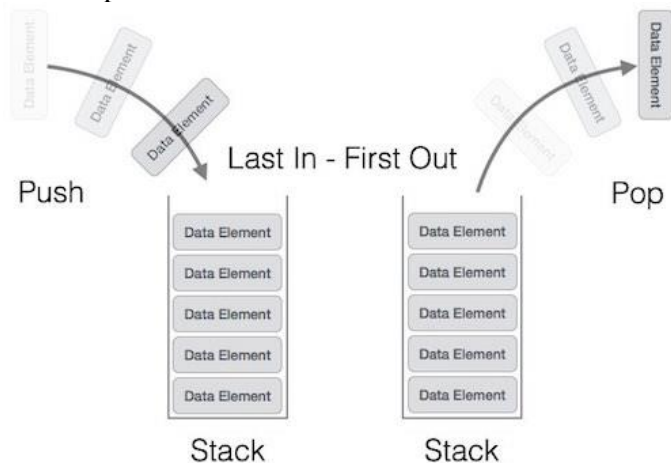
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

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

Implementation of peek() function in C programming language –

Example

```
int peek() {
  return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

    if top equals to MAXSIZE
        return true
    else
        return false
    endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
    if(top == MAXSIZE)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

    if top less than 0
        return true
    else
        return false
    endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

Example

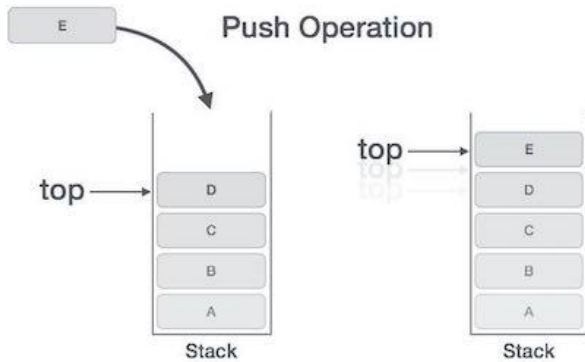
```
bool isempty() {
    if(top == -1)
        return true;
    else
        return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit(OVERFLOW).
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.

- **Step 5** – Returns success.



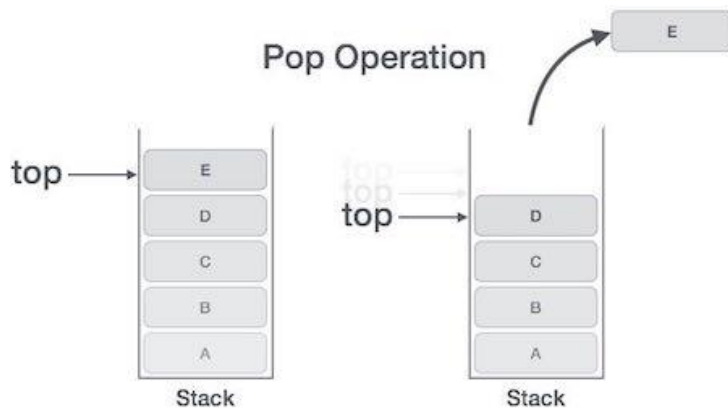
Example

```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit(UNDERFLOW).
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

Implementation of this algorithm in C, is as follows –

Example

```

int pop(int data) {

    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
}

```

Applications of stack:

- 1.** Converting Infix to Postfix expression
- 2.** Postfix expression evaluation
- 3.** Implementing recursion
- 4.** Multi tasking
- 5.** Scheduling algorithms
- 6.** Syntax Parsing
- 7.** Backtracking algorithms

//STACK ADT IMPLEMENTAION USING ARRAYS

```

#include<stdio.h>
#define max 50
#include<stdlib.h>
int top=-1;
int stack[max];
void push();
void pop();
void display();
void main()
{
    int ch;
    system("clear");
    while(1)
    {
        printf("\n_____");
        printf("\n STACK ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.PUSH");
        printf("\n\t2.POP");
        printf("\n\t3.DISPLAY");
        printf("\n\t4.EXIT");
        printf("\n Enter ur choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: push();
                    break;

```



```

        case 2: pop();
                break;
        case 3: display();
                break;
        case 4: exit(0);
                break;
        default: printf("\n wrong choice\n");
    }
}

```

```

void push()
{
    int element;
    if(top==max-1)
        printf("\n STACK OVERFLOW\n");
    else
    {
        printf("\n Enter element to be inserted:");
        scanf("%d",&element);
        top=top+1;
        stack[top]=element;
    }
}

```

```

void pop()
{
    if(top== -1)
        printf("\n STACK UNDERFLOW\n");
    else
    {
        printf("\n deleted element is:%d\n",stack[top]);
        top=top-1;
    }
}

```

```

void display()
{
    int i;
    if(top== -1)
        printf("\n STACK IS EMPTY\n");
    else
    {
        printf("\n STACK ELEMENTS ARE:\n");
    }
}

```

```
for(i=top;i>=0;i--)  
printf("%t%d",stack[i]);  
  
}  
}
```

QUEUE using Arrays:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
  return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull
```

```
    if rear equals to MAXSIZE
        return true
    else
        return false
    endif
```

```
end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
    if(rear == MAXSIZE - 1)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty
    if front is less than MIN OR front is greater than rear
        return true
    else
        return false
    endif
end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

```
bool isempty() {
    if(front < 0 || front > rear)
        return true;
    else
        return false;
}
```

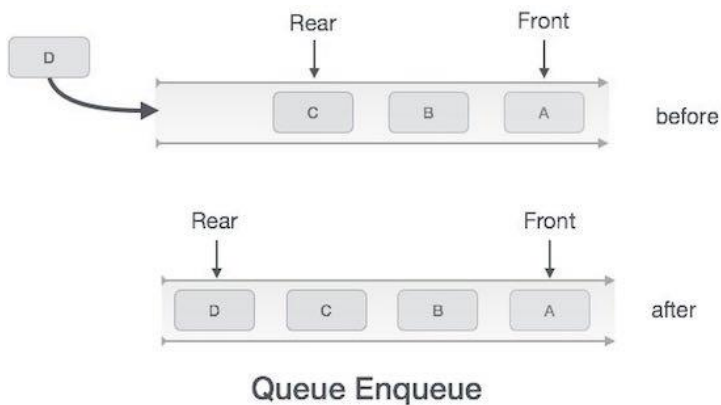
Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.

- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```

procedure enqueue(data)
  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true
end procedure

```

Implementation of enqueue() in C programming language –
Example

```

int enqueue(int data)
  if(isfull())
    return 0;

  rear = rear + 1;
  queue[rear] = data;

  return 1;
end procedure

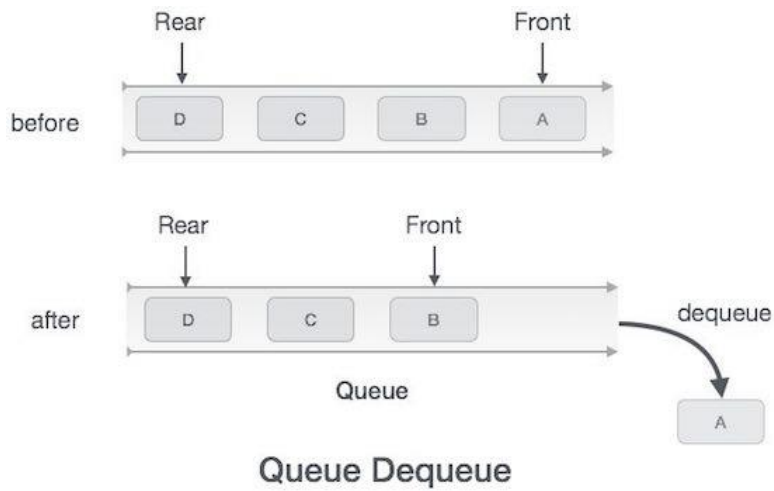
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.

- **Step 5** – Return success.



Algorithm for dequeue operation

```

procedure dequeue
  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true
end procedure

```

Implementation of dequeue() in C programming language –
Example

```

int dequeue() {
  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}

```

Applications:

- Graph traversals
- CPU scheduling
- Disk Scheduling
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.

- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).

//QUEUE ADT IMPLEMENTAION USING ARRAYS

```

#include<stdio.h>
#define max 50
#include<stdlib.h>
int front=-1; int rear=-1;
int queue[max];
void insertion();
void deletion();
void display();
void main()
{
    int ch;
    system("clear");
    while(1)
    {
        printf("\n_____");
        printf("\n QUEUE ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.INSERTION");
        printf("\n\t2.DELETION");
        printf("\n\t3.DISPLAY");
        printf("\n\t4.EXIT");
        printf("\n Enter ur choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: insertion();
                    break;
            case 2: deletion();
                    break;
            case 3: display();
                    break;
            case 4 : exit(0);
                    break;
            default: printf("\n wrong choice\n");
        }
    }
}

void insertion()
{
    int element;
    if(front===-1)

```

```

    front=0;
    if(rear==max-1)
        printf("\n QUEUE OVERFLOW\n");
    else
    {
        printf("\n Enter element to be inserted:");
        scanf("%d",&element);
        rear=rear+1;
        queue[rear]=element;
    }
}

void deletion()
{
    if(front==-1||front>rear)
        printf("\n QUEUE UNDERFLOW\n");
    else
    {
        printf("\n deleted element is:%d\n",queue[front]);
        front=front+1;
    }
}

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

```

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre- allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.
5. Elements may or may not be stored in consecutive memory locations so if we do not have consecutive memory available, even then we can store the data in computer.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.
3. No random access of elements as we do in array by index. We have to access each node sequentially.
4. Reverse traversing is difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	Efficient	efficient
Random access	Efficient	inefficient
Resigning	Inefficient	efficient
Element rearranging	Inefficient	efficient
Overhead per elements	None	1 or 2 links

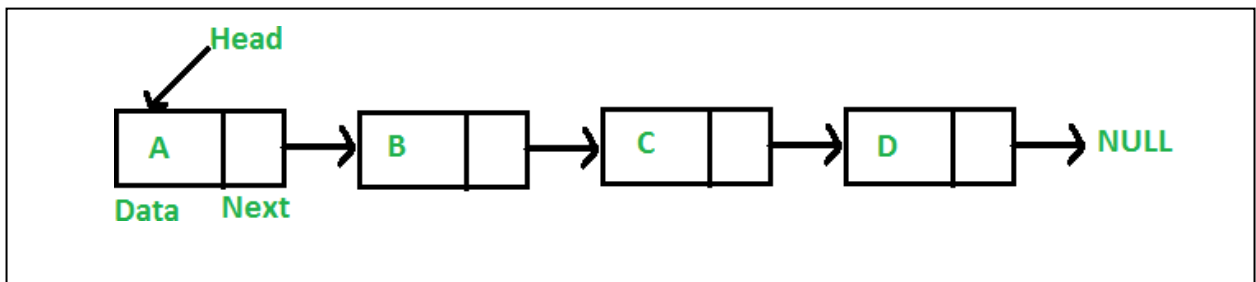
Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.

1. Single Linked List.

Linked List is a non-primitive linear data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain.

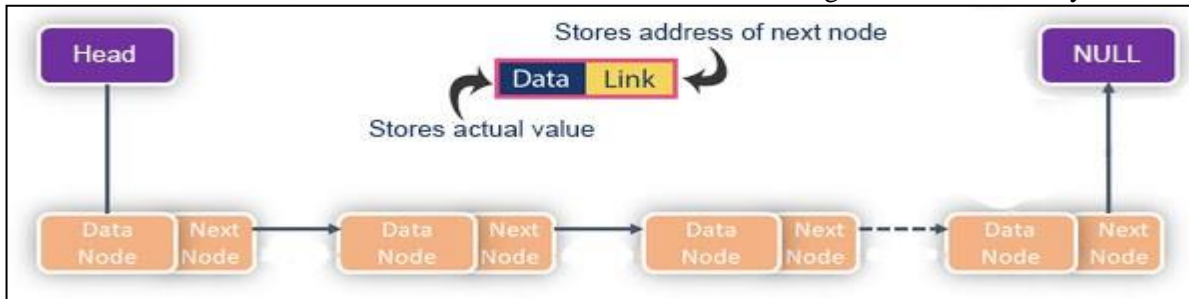


As per the above illustration, following are the important points to be considered:

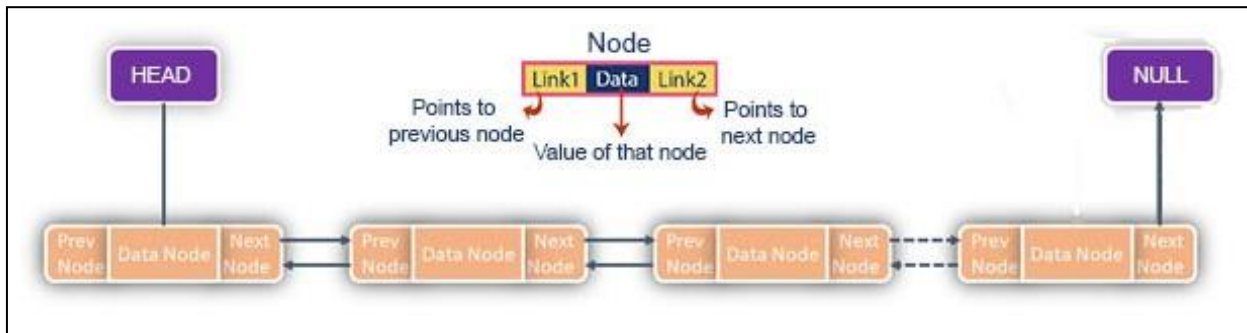
- Linked List contains a reference element (pointer) called head (which points to first node).
- Each node carries a data field and an address field called next.
- Each node is linked with its next node using its next link.
- Last node carries a link as NULL representing the end of the list.

There are three types of linked lists,

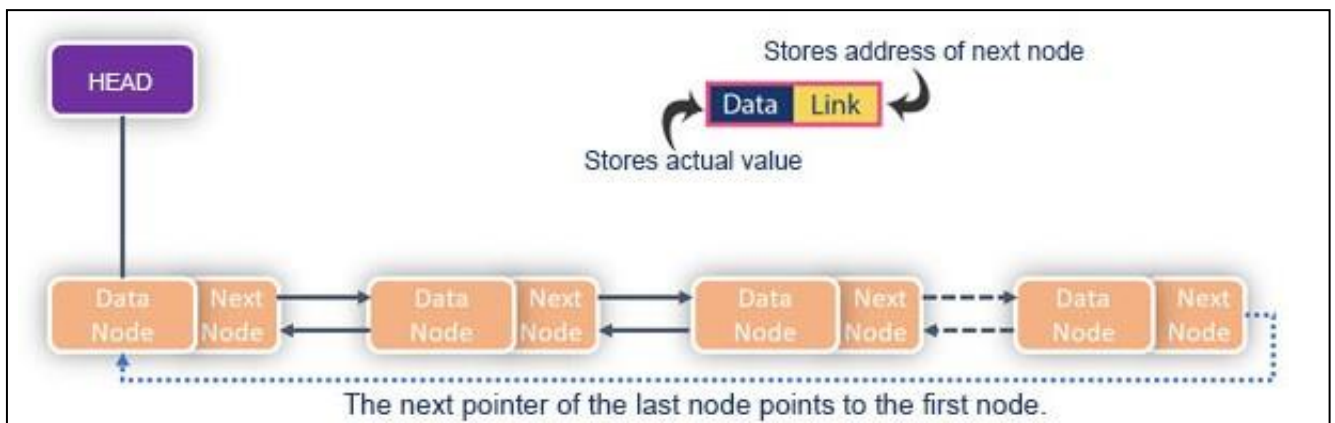
Singly Linked List: Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The link of the last node in the list is NULL, which indicates the end of the list. Item navigation is forward only.



Doubly Linked List: Doubling Linked Lists have two address nodes, in particular, one node serves the purpose of directing to the next node while the other one points to the previous node. A linked list that has only next links is a singly linked list. Adding previous links makes a doubly linked list. Item navigation is both forward and backward.



Circular Linked List: Circular linked list is a linked list where all nodes are connected to form a circle. Simply, every node has the address of next node; putting the address of the first node in the last node makes it circular linked list. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Basic operations carried out in a linked list:

Following are the basic operations supported by a linked list:

1. Insertion:

- Insertion at beginning: - Inserts the new node at the beginning of the linked list.
- Insertion at end: - Inserts the new node at the ending of the linked list.
- Insertion after a given value: - Inserts the new node after a given value.

2. Deletion:

- Deletion at beginning: - Deletes the node at the beginning of the linked list.
- Deletion at end: - Deletes the node at the end of the linked list.
- Deletion of a value: - Deletes the node of given value.

3. Traversing:

- Traversal through a linked list means travelling through every single node of a list and reaching the end of the list. The traversal can be in only one direction in singly linked list and two directions (forward and reverse) in a doubly linked list.

4. Display:

- Displays all the values (data) present in the linked list.

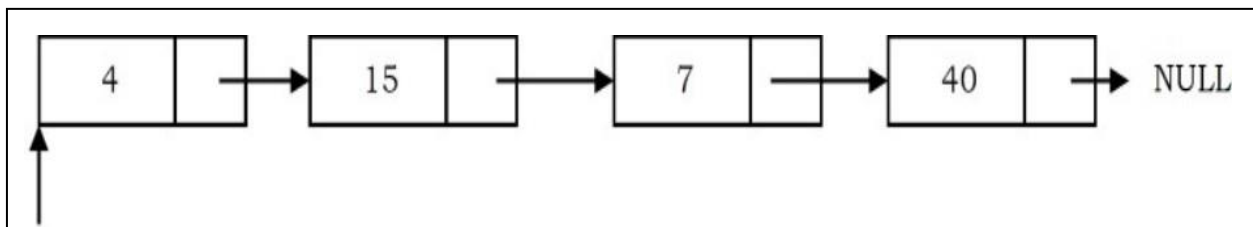
5. Search:

- Searches for the element in the linked list using the given key.

ADDITIONAL INFORMATION:

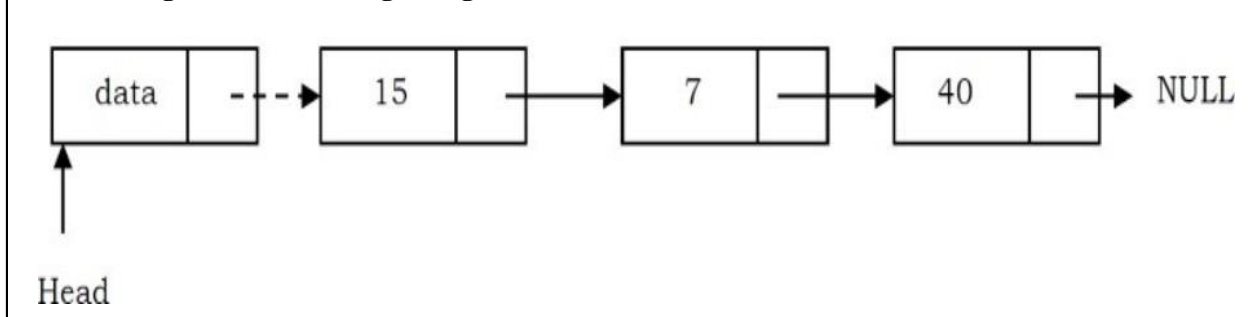
SINGLY LINKED LIST OPERATIONS ILLUSTRATED:

- **Singly linked list:**

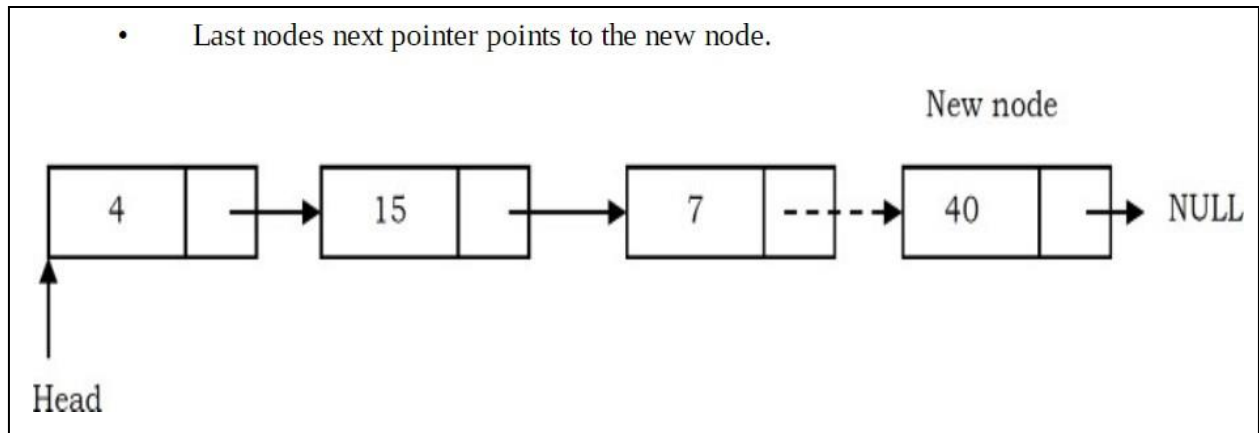
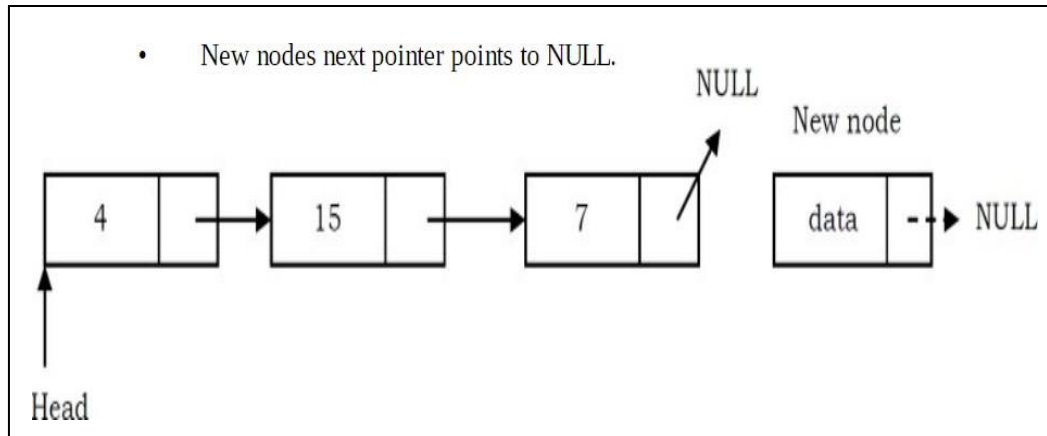


• Update head pointer to point to the new node.

- **Inserting a node at the beginning:**

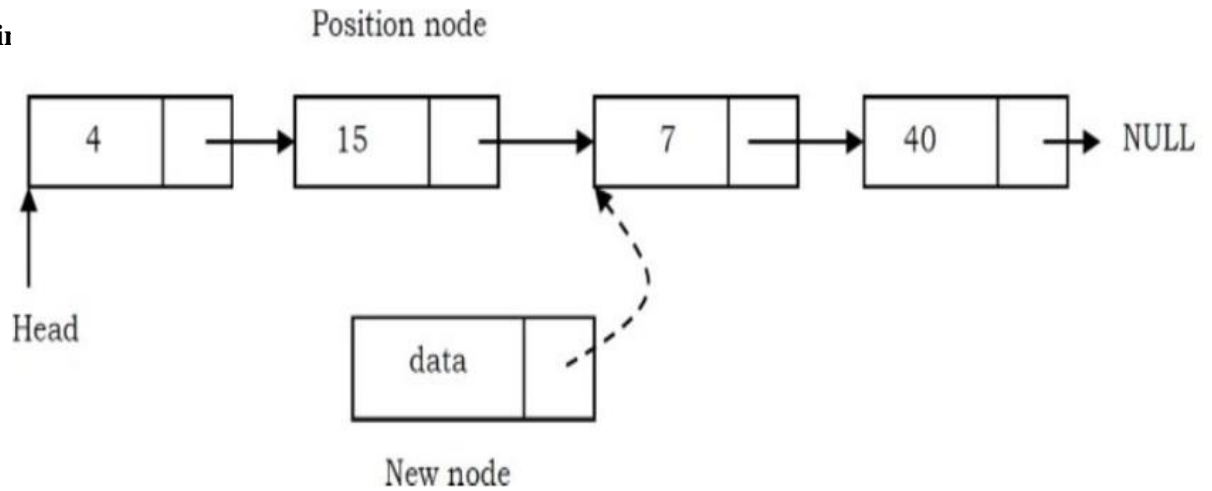


- **Inserting a node at the end:**

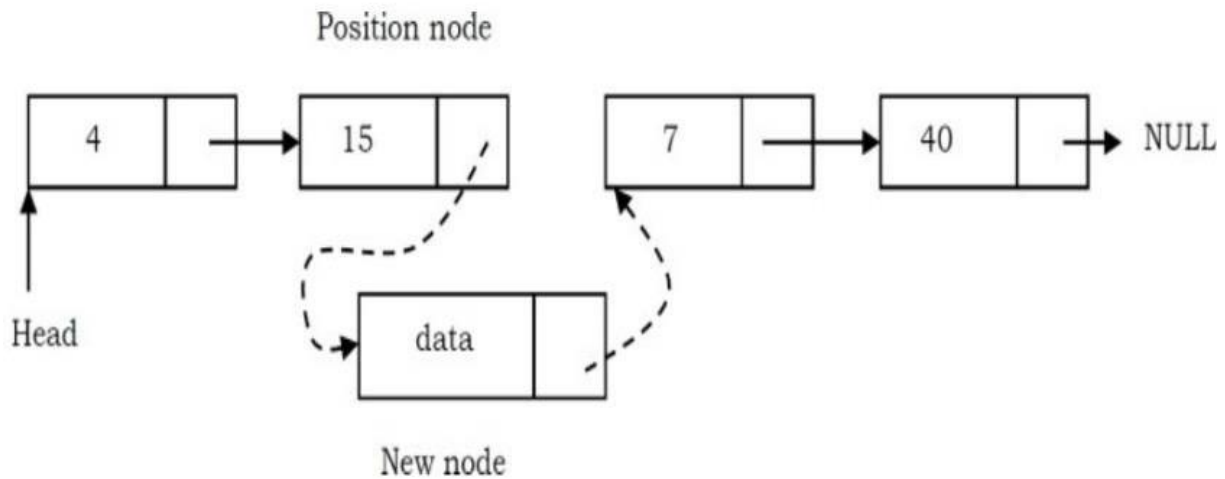


- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.

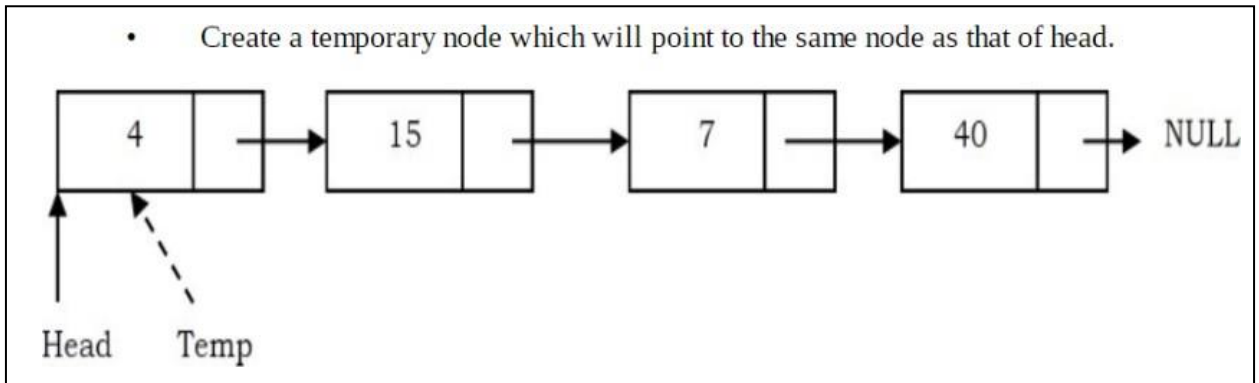
- **Insertion**



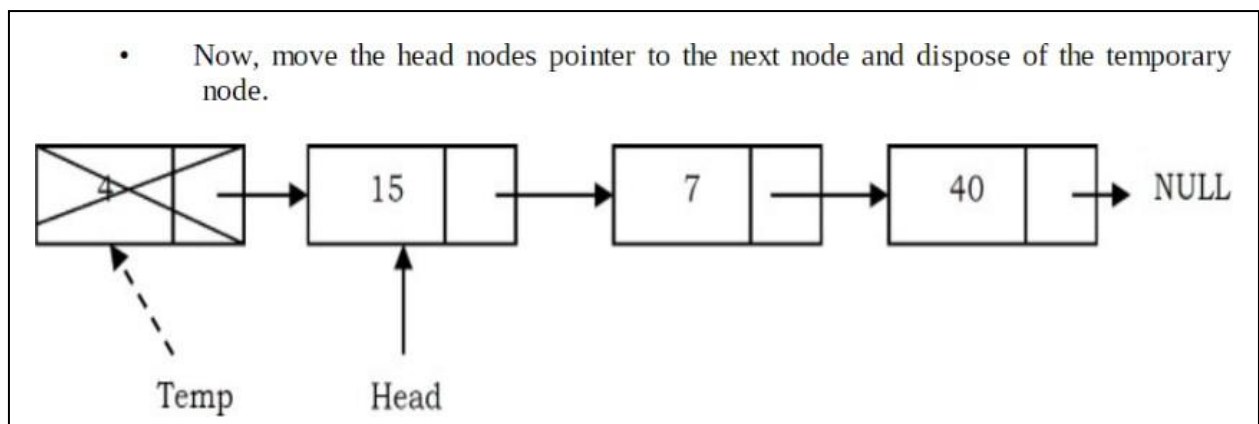
- Position node's next pointer now points to the new node.



- **Deleting the First Node:**

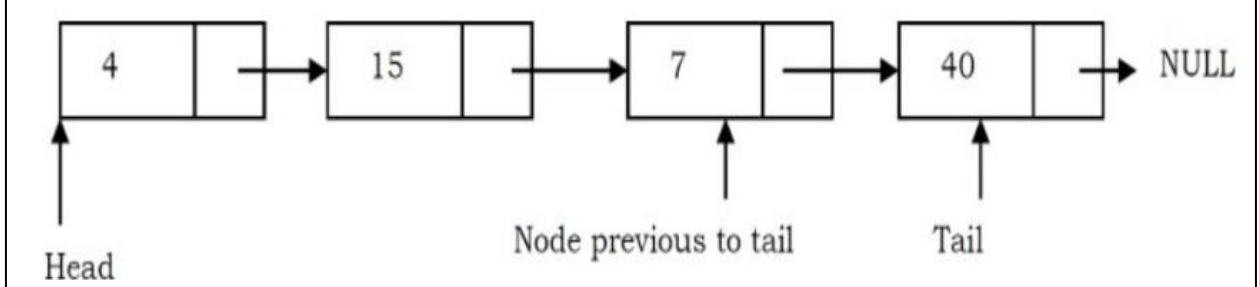


- Now, move the head nodes pointer to the next node and dispose of the temporary node.

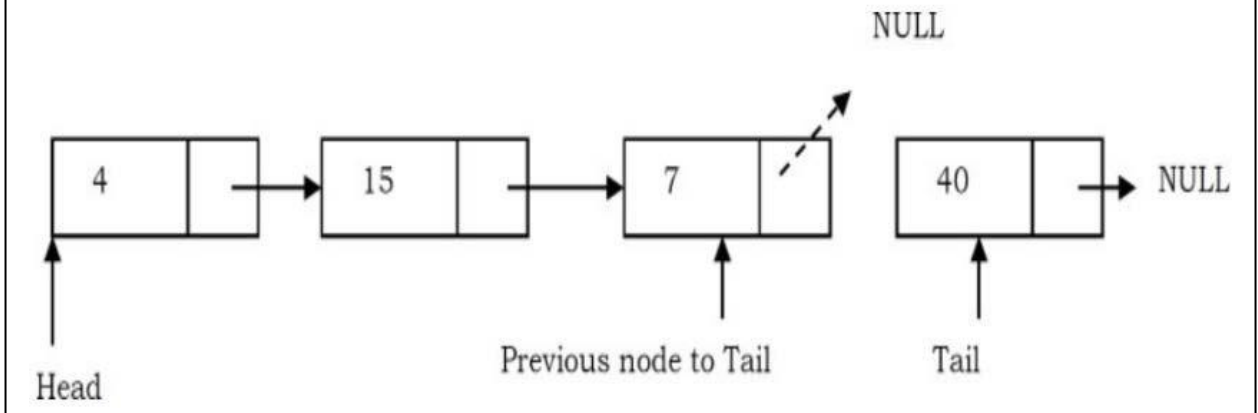


- **Deleting the Last Node in Singly Linked List:**

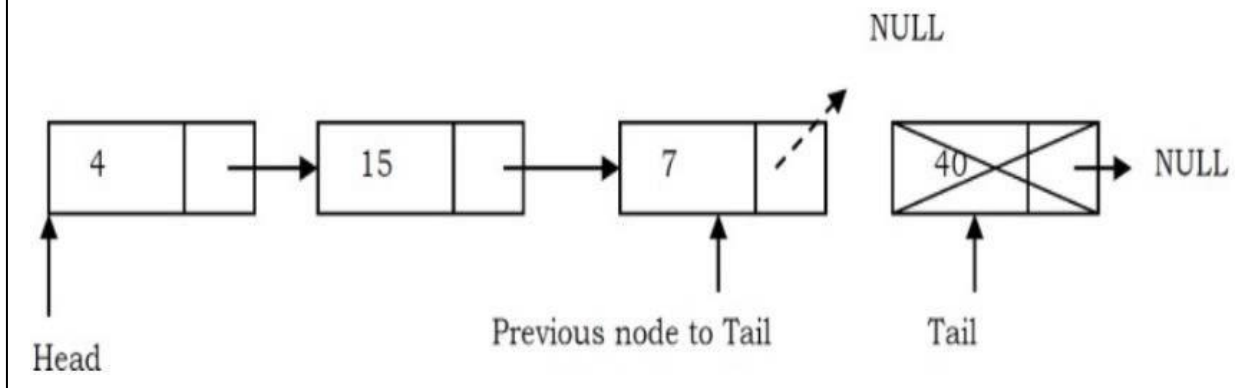
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.



- Update previous node's next pointer with NULL.

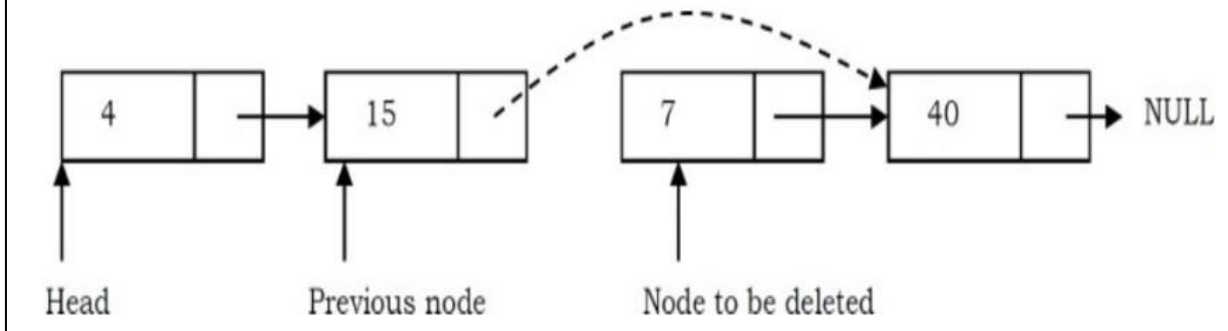


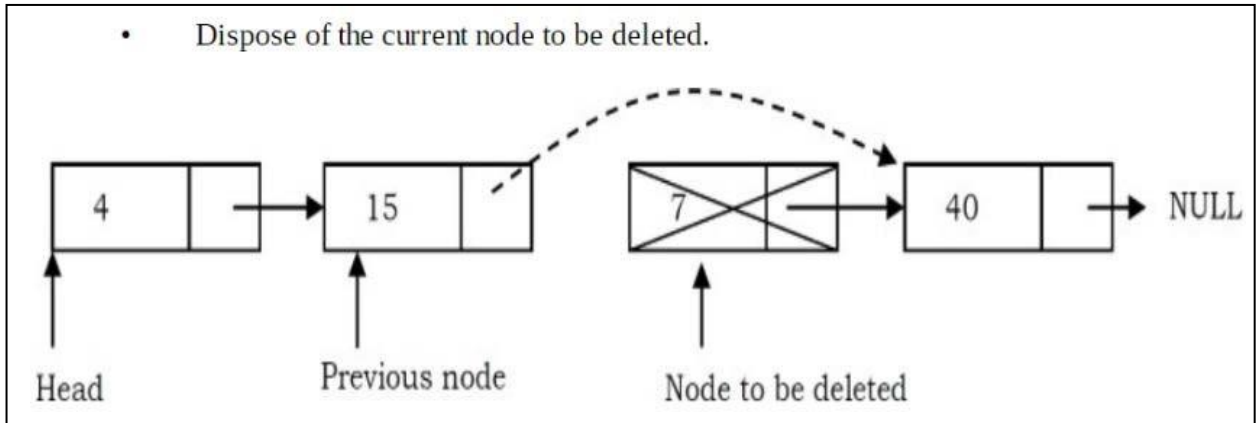
- Dispose of the tail node.



- Deleting an Intermediate Node in Singly Linked List:**

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.





//IMPLEMENTAION OF SINGLE LINKED LIST ADT

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
void delete_info();
void delete_end();
void display();

void main()
{
    int ch;
    system("clear");
    while(1)
    {
        printf("\n_____");
        printf("\n SINGLE LINKED LIST ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.CREATE");
        printf("\n\t2.INSERTION AT THE BEGINNING");
        printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");
        printf("\n\t4.INSERTION AT THE END");
        printf("\n\t5.DELETION AT THE BEGINNING");
        printf("\n\t6.DELETION THE GIVEN INFO:");
    }
}
```

```

printf("\n\t7.DELETION AT THE END");
printf("\n\t8.DISPLAY");
printf("\n\t9.EXIT");
printf("\n Enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
    case 1: create();
            break;
    case 2: insert_begin();
            break;
    case 3: insert_after();
            break;
    case 4: insert_end();
            break;
    case 5: delete_begin();
            break;
    case 6: delete_info();
            break;
    case 7: delete_end();
            break;

    case 8: display();
            break;
    case 9: exit(0);
            break;
    default: printf("\n wrong choice\n");
}
}
}

```

```

void create()
{
    struct node *ptr,*cptr;
    int c;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter first node information:");
    scanf("%d",&ptr->data);
    head=ptr;
    printf("\n Enter 0/1 for more nodes:");
    scanf("%d",&c);
    while(c==1)
    {
        cptr=(struct node*)malloc(sizeof(struct node));
        ptr->next=cptr;
        ptr=cptr;
        printf("\n Enter next node information:");
        scanf("%d",&cptr->data);
        printf("\n Enter 0/1 for more nodes:");
        scanf("%d",&c);
    }
}

```

```

        ptr->next=NULL;
    }

void insert_begin()
{
    struct node *ptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    ptr->next=head;
    head=ptr;
}

void insert_end()
{
    struct node *ptr, *cptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    cptr=head;

    while(cptr->next!=NULL)
        cptr=cptr->next;

    cptr->next=ptr;
    ptr->next=NULL;
}

void insert_after()
{
    struct node *ptr, *cptr;
    int d;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    printf("\n Enter node info after which U want to insert:");
    scanf("%d",&d);
    cptr=head;

    while(cptr->data!=d)
        cptr=cptr->next;

    ptr->next=cptr->next;
    cptr->next=ptr;
}

void delete_begin()
{
    struct node *ptr;

```

```

        if(head==NULL)
        printf("\n LINKED LIST UNDERFLOW\n");
        else
        {
            ptr=head;
            printf("\n deleted element is :%d",ptr->data);
            head=ptr->next;
            free(ptr);
        }
    }

void delete_end()
{
    struct node *ptr, *cptr;
    ptr=head;
    while(ptr->next!=NULL)
    {
        cptr=ptr;
        ptr=ptr->next;
    }
    cptr->next=NULL;
    printf("\n deleted element is :%d",ptr->data);
    free(ptr);
}

void delete_info()
{
    struct node *ptr,*cptr;
    int d;
    if(head==NULL)
    printf("\n LINKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        printf("\n Enter node info to be deleted:");
        scanf("%d",&d);
        while(ptr->data!=d)
        {
            cptr=ptr;
            ptr=ptr->next;
        }
        cptr->next=ptr->next;
        printf("\n deleted element is :%d",ptr->data);
        free(ptr);
    }
}

void display()

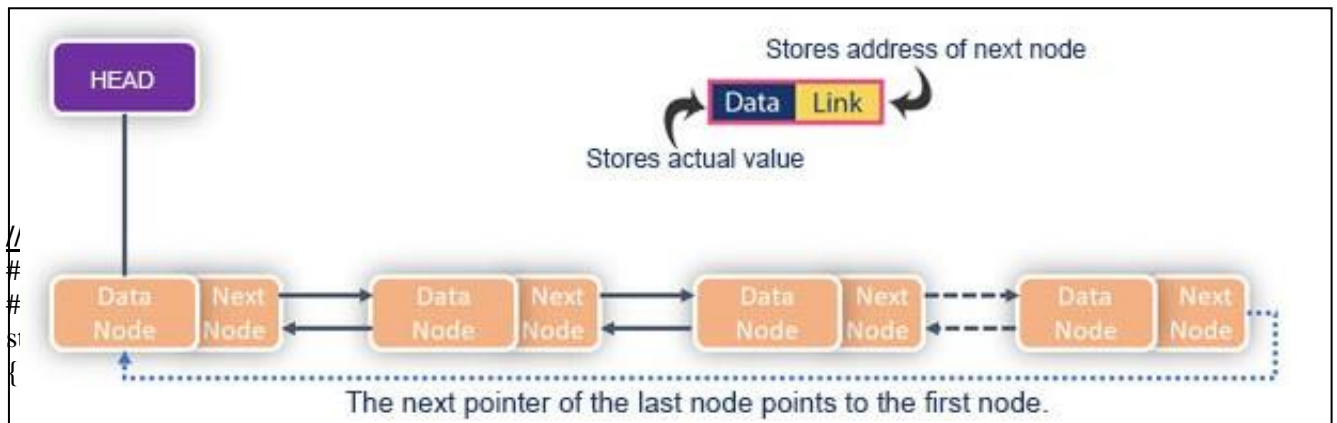
```

```

{
    struct node *ptr;
    ptr=head;
    if(head==NULL)
    printf("\n LINKED LIST IS EMPTY\n");
    else
    {
        while(ptr!=NULL)
        {
            printf(" %d->",ptr->data);
            ptr=ptr->next;
        }
    }
}

```

Circular Linked List: Circular linked list is a linked list where all nodes are connected to form a circle. Simply, every node has the address of next node; putting the address of the first node in the last node makes it circular linked list. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



```

struct node *next;
};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
void delete_info();
void delete_end();
void display();
void main()
{
    int ch;
    system("clear");
    while(1)
    {

```

```

printf("\n_____");
printf("\n SINGLE LINKED LIST ADT OPERATIONS ARE:\n");
printf("_____");
printf("\n\t1.CREATE");
printf("\n\t2.INSERTION AT THE BEGINNING");
printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");
printf("\n\t4.INSERTION AT THE END");
printf("\n\t5.DELETION AT THE BEGINNING");
printf("\n\t6.DELETION OF THE GIVEN INFO:");
printf("\n\t7.DELETION AT THE END");
printf("\n\t8.DISPLAY");
printf("\n\t9.EXIT");
printf("\n Enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
    case 1: create();
            break;
    case 2: insert_begin();
            break;
    case 3: insert_after();
            break;
    case 4: insert_end();
            break;
    case 5: delete_begin();
            break;
    case 6: delete_info();
            break;
    case 7: delete_end();
            break;

    case 8: display();
            break;
    case 9: exit(0);
            break;
    default: printf("\n wrong choice\n");
}
}
}

void create()
{
    struct node *ptr,*cptr;
    int c;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter first node information:");
    scanf("%d",&ptr->data);
    head=ptr;
    printf("\n Enter 0/1 for more nodes:");
    scanf("%d",&c);
    while(c==1)
    {

```

```

        ptr->next=cptr;
        ptr=cptr;
        printf("\n Enter next node information:");
        scanf("%d",&cptr->data);
        printf("\n Enter 0/1 for more nodes:");
        scanf("%d",&c);
    }

    ptr->next=head;
}
void insert_begin()
{
    struct node *ptr,*cptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    cptr=head;
    while(cptr->next!=head)
    {cptr=cptr->next;}

    ptr->next=head;
    head=ptr;
    cptr->next=head;

}
void insert_end()
{
    struct node *ptr, *cptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    cptr=head;

    while(cptr->next!=head)
        cptr=cptr->next;

    cptr->next=ptr;
    ptr->next=head;
}
void insert_after()
{
    struct node *ptr, *cptr;
    int d;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    printf("\n Enter node info after which U want to insert:");
    scanf("%d",&d);
    cptr=head;

```

```

while(cptr->data!=d)
    cptr=cptr->next;

ptr->next=cptr->next;
cptr->next=ptr;
}

void delete_begin()
{
    struct node *ptr,*cptr;
    if(head==NULL)
        printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        cptr=head;
        printf("\n deleted element is :%d",ptr->data);
        while(cptr->next!=head)
        {cptr=cptr->next;}

        head=ptr->next;
        free(ptr);
        cptr->next=head;
    }
}
void delete_end()
{
    struct node *ptr, *cptr;
    if(head==NULL)
        printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        while(ptr->next!=head)
        {
            cptr=ptr;
            ptr=ptr->next;
        }
        cptr->next=head;
        printf("\n deleted element is :%d",ptr->data);
        free(ptr);
    }
}
void delete_info()
{
    struct node *ptr,*cptr;
    int d;
    if(head==NULL)
        printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");
}

```



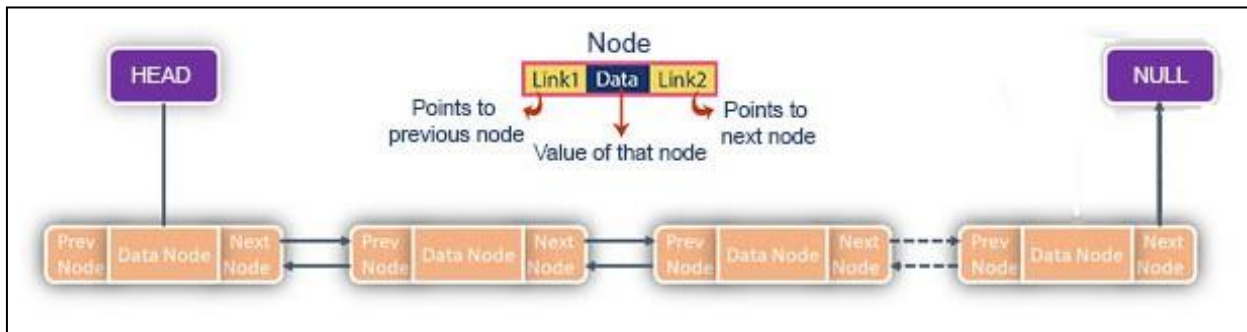
```

else
{
ptr=head;
printf("\n Enter node info to be deleted:");
scanf("%d",&d);
while(ptr->data!=d)
{
cptr=ptr;
ptr=ptr->next;
}
cptr->next=ptr->next;
printf("\n deleted element is :%d",ptr->data);
free(ptr);
}
}

void display()
{
struct node *ptr;
if(head==NULL)
printf("\n LINKED LIST IS EMPTY\n");
else
{
ptr=head;
do
{
printf(" %d->",ptr->data);
ptr=ptr->next;
} while(ptr!=head);
}
}

```

Doubly Linked List: Doubling Linked Lists have two address nodes, in particular, one node serves the purpose of directing to the next node while the other one points to the previous node. A linked list that has only next links is a singly linked list. Adding previous links makes a doubly linked list. Item navigation is both forward and backward.



//IMLEMENTAION OF DOUBLE LINKED LIST ADT

```

#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    int data;
    struct node *next;

};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
void delete_info();
void delete_end();
void display();

void main()
{
    int ch;
    system("clear");
    while(1)
    {
        printf("\n_____");
        printf("\n DOUBLE LINKED LIST ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.CREATE");
        printf("\n\t2.INSERTION AT THE BEGINNING");
        printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");
        printf("\n\t4.INSERTION AT THE END");
        printf("\n\t5.DELETION AT THE BEGINNING");
        printf("\n\t6.DELETION THE GIVEN INFO:");
        printf("\n\t7.DELETION AT THE END");
        printf("\n\t8.DISPLAY");
        printf("\n\t9.EXIT");
        printf("\n Enter ur choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: create();
                    break;
            case 2: insert_begin();
                    break;
            case 3: insert_after();
                    break;
            case 4: insert_end();
                    break;
            case 5: delete_begin();
                    break;
        }
    }
}

```

```

        case 6: delete_info();
                break;
        case 7: delete_end();
                break;

        case 8: display();
                break;
        case 9: exit(0);
                break;
        default: printf("\n wrong choice\n");
    }
}
}

```

```

void create()
{
    struct node *ptr,*cptr;
    int c;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter first node information:");
    scanf("%d",&ptr->data);
    ptr->prev=NULL;
    head=ptr;
    printf("\n Enter 0/1 for more nodes:");
    scanf("%d",&c);
    while(c==1)
    {
        cptr=(struct node*)malloc(sizeof(struct node));
        ptr->next=cptr;
        cptr->prev=ptr;
        ptr=cptr;
        printf("\n Enter next node information:");
        scanf("%d",&cptr->data);
        printf("\n Enter 0/1 for more nodes:");
        scanf("%d",&c);
    }
    ptr->next=NULL;
}

```

```

void insert_begin()
{
    struct node *ptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter node information to be inserted:");
    scanf("%d",&ptr->data);
    ptr->prev=NULL;
    ptr->next=head;
    head=ptr;
}

```

```

void insert_end()
{
    struct node *ptr, *cptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter  node information to be inserted:");
    scanf("%d",&ptr->data);
    cptr=head;

    while(cptr->next!=NULL)
        cptr=cptr->next;

    cptr->next=ptr;
    ptr->prev=cptr;
    ptr->next=NULL;
}

void insert_after()
{
    struct node *ptr, *cptr;
    int d;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter  node information to be inserted:");
    scanf("%d",&ptr->data);
    printf("\n Enter node info after which U want to insert:");
    scanf("%d",&d);
    cptr=head;

    while(cptr->data!=d)
        cptr=cptr->next;

    (cptr->next)->prev=ptr;
    ptr->prev=cptr;
    ptr->next=cptr->next;
    cptr->next=ptr;
}

void delete_begin()
{
    struct node *ptr;
    if(head==NULL)
        printf("\n DOUBLE INKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        printf("\n deleted element is :%d",ptr->data);
        head=ptr->next;
        head->prev=NULL;
        free(ptr);
    }
}

```

```

void delete_end()
{
    struct node *ptr, *cptr;
    if(head==NULL)
        printf("\n DOUBLE INKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        while(ptr->next!=NULL)
        {
            cptr=ptr;
            ptr=ptr->next;
        }
        cptr->next=NULL;
        printf("\n deleted element is :%d",ptr->data);
        free(ptr);
    }
}

void delete_info()
{
    struct node *ptr,*cptr;
    int d;
    if(head==NULL)
        printf("\n DOUBLE LINKED LIST UNDERFLOW\n");
    else
    {
        ptr=head;
        printf("\n Enter node info to be deleted:");
        scanf("%d",&d);
        while(ptr->data!=d)
        {
            cptr=ptr;
            ptr=ptr->next;
        }
        cptr->next=ptr->next;
        (ptr->next)->prev=cptr;
        printf("\n deleted element is :%d",ptr->data);
        free(ptr);
    }
}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head==NULL)

```

```

printf("\n DOUBLE LINKED LIST IS EMPTY\n");
else
{
while(ptr!=NULL)
{
printf(" %d<->",ptr->data);
ptr=ptr->next;
}
}
}

```

STACK IMPLEMENTATION USING LINKED LIST:

//IMLEMENTAION OF STACK ADT USING SINGLE LINKED LIST

```

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

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

void push();
void pop();
void display();

void main()
{
int ch;
system("clear");
while(1)
{
printf("\n_____");
printf("\n STACK ADT USING SLL OPERATIONS ARE:\n");
printf("_____");
printf("\n\t1.PUSH");
printf("\n\t2.POP");
printf("\n\t3.DISPLAY");
printf("\n\t4.EXIT");
printf("\n Enter ur choice:");
scanf("%d",&ch);

switch(ch)
{

case 1: push();
break;

```

```

        case 2: pop();
                break;
        case 3: display();
                break;
        case 4: exit(0);
                break;
        default: printf("\n wrong choice\n");
    }
}

```

```

void push()
{
    struct node *ptr;
    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter element to be inserted:");
    scanf("%d",&ptr->data);
    ptr->next=top;
    top=ptr;
}

```

```

void pop()
{
    struct node *ptr;
    if(top==NULL)
        printf("\n STACK UNDERFLOW\n");
    else
    {
        ptr=top;
        printf("\n deleted element is :%d\n",ptr->data);
        top=ptr->next;
        free(ptr);
    }
}

```

```

void display()
{
    struct node *ptr;
    ptr=top;
    if(top==NULL)
        printf("\n STACK IS EMPTY\n");
    else
    {
        while(ptr!=NULL)
        {
            printf(" %d->",ptr->data);

```



```

                break;
        case 3:    display();
                break;
        case 4:    exit(0);
                break;
        default:  printf("\n wrong choice\n");
    }
}

```

```

void insert()
{
    struct node *ptr;

    ptr= (struct node*)malloc(sizeof(struct node));
    printf("\n Enter  node information:");
    scanf("%d",&ptr->data);
    if(front==NULL)
    {
        front=ptr;
        rear=ptr;
        front->next=NULL;
        rear->next=NULL;
    }
    else
    {
        rear->next=ptr;
        rear=ptr;
        rear->next=NULL;
    }
}

```

```

void del()
{
    struct node *ptr;
    if(front==NULL)
    {
        printf("\n QUEUE UNDERFLOW\n");
        count=1;
    }
    else
    {
        ptr=front;
        printf("\n deleted element is:%d",ptr->data);
        front=ptr->next;
        free(ptr);
    }
}

```

```

void display()

```

```

{
    struct node *ptr;
    ptr=front;
    if(front==NULL)
    printf("\n QUEUE IS EMPTY\n");
    else
    {
        while(ptr!=NULL)
        {
            printf(" %d->",ptr->data);
            ptr=ptr->next;
        }
    }
}

```

Expressions:

OPERATORS +,-, /, %.....*

OPERANDS A,B a,c..., 1 5 6 7 7

INFIX

A+B

A*B+C/R

4*6+3

(4*8)*5+3...

POSTFIX

AB+

AB*CR/+

4 6 * 3 +

4 8 * 5 * 3 +

PREFIX

+AB

+*AB /CR

+ * 4 6 3

+ **4 8 5 3

- 1) Infix expression 2) postfix (reverse polish) 3. Prefix (polish)

Examples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
A+ *BC		
+A *BC		
A +BC*		
A BC*+		

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example



Postfix Expression Evaluation using Stack Data Structure

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

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










Example

Consider the following Expression...

Infix Expression $(5 + 3) * (8 - 2)$

Postfix Expression $5 3 + 8 2 - *$

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	<pre>value1 = pop() value2 = pop() result = value2 + value1 push(result)</pre> 	<pre>value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8)</pre> (5 + 3)
8	push(8) 	(5 + 3)
2	push(2) 	(5 + 3)
-	<pre>value1 = pop() value2 = pop() result = value2 - value1 push(result)</pre> 	<pre>value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6)</pre> (8 - 2) (5 + 3), (8 - 2)
*	<pre>value1 = pop() value2 = pop() result = value2 * value1 push(result)</pre> 	<pre>value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48)</pre> (6 * 8) (5 + 3) * (8 - 2)
\$ End of Expression	result = pop() 	Display (result) 48 As final result

Infix Expression $(5 + 3) * (8 - 2) = 48$

Postfix Expression $5 3 + 8 2 - *$ value is **48**

Infix to Postfix Conversion

Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix).

Example

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

$$D = A + B * C$$

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

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

$$D A B C * + =$$

Infix to Postfix Conversion using Stack Data Structure












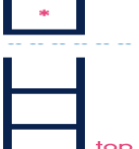

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. **Read all the symbols one by one from left to right in the given Infix Expression.**
2. **If the reading symbol is operand, then directly print it to the result (Output).**
3. **If the reading symbol is left parenthesis '(', then Push it on to the Stack.**
4. **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.**
5. **If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.**

Consider the following Infix Expression...

$$(A + B) * (C - D)$$

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY 	EMPTY
(Push '(' 	EMPTY
A	No operation Since 'A' is OPERAND 	A
+	'+' has low priority than '(' so, PUSH '+' 	A
B	No operation Since 'B' is OPERAND 	A B
)	POP all elements till we reach '(' POP '+' POP '(' 	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*' 	A B +
(PUSH '(' 	A B +
C	No operation Since 'C' is OPERAND 	A B + C
-	'-' has low priority than '(' so, PUSH '-' 	A B + C
D	No operation Since 'D' is OPERAND 	A B + C D
)	POP all elements till we reach '(' POP '-' POP '(' 	A B + C D -
\$	POP all elements till Stack becomes Empty 	A B + C D - *

$A B + C D - *$

The final Postfix Expression is as follows...

$A B + C D - *$

HASHING:

INTRODUCTION

In Chapter 14, we discussed two search algorithms: *linear search* and *binary search*. Linear search has a running time proportional to $O(n)$, while binary search takes time proportional to $O(\log n)$, where n is the number of elements in the array. Binary search and binary search trees are efficient algorithms to search for an element. But what if we want to perform the search operation in time proportional to $O(1)$? In other words, is there a way to search an array in constant time, irrespective of its size?

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
.....
.....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

Figure 15.1 Records of employees

There are two solutions to this problem. Let us take an example to explain the first solution. In a small company of 100 employees, each employee is assigned an Emp_ID in the range 0–99. To store the records in an array, each employee's Emp_ID acts as an index into the array where the employee's record will be stored as shown in Fig. 15.1.

In this case, we can directly access the record of any employee, once we know his Emp_ID, because the array index is the same as the Emp_ID number. But practically, this implementation is hardly feasible.

Let us assume that the same company uses a five-digit `Emp_ID` as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we need an array of size 100,000, of which only 100 elements will be used. This is illustrated in Fig. 15.2.

Key	Array of Employees' Records
Key 00000 → [0]	Employee record with Emp_ID 00000
.....
Key n → [n]	Employee record with Emp_ID n
.....
Key 99998 → [99998]	Employee record with Emp_ID 99998
Key 99999 → [99999]	Employee record with Emp_ID 99999

Figure 15.2 Records of employees with a five-digit `Emp_ID`

It is impractical to waste so much storage space just to ensure that each employee's record is in a unique and predictable location.

Whether we use a two-digit primary key (`Emp_ID`) or a five-digit key, there are just 100 employees in the company. Thus, we will be using only 100 locations in the array. Therefore, in order to keep the array size down to the size that we will actually be using (100 elements), another good option is to use just the last two digits of the key to identify each employee. For example, the employee with `Emp_ID` 79439 will be stored in the element of the array with index 39. Similarly, the employee with `Emp_ID` 12345 will have his record stored in the array at the 45th location.

In the second solution, the elements are not stored according to the *value* of the key. So in this case, we need a way to convert a five-digit key number to a two-digit array index. We need a function which will do the transformation. In this case, we will use the term *hash table* for an array and the function that will carry out the transformation will be called a *hash function*.

HaSH TableS

Hash table is a data structure in which keys are mapped to array positions by a hash function. In the example discussed here we will use a hash function that extracts the last two digits of the key. Therefore, we map the keys to array locations or array indices. A value stored in a hash table can be searched in $O(1)$ time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).

Figure 15.3 shows a direct correspondence between the keys and the indices of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.

However, when the set κ of keys that are actually used is smaller than the universe of keys (\cup), a hash table consumes less storage space. The storage requirement for a hash table is $O(\kappa)$, where κ is the number of keys actually used.

In a hash table, an element with key κ is stored at index $h(\kappa)$ and not κ . It means a hash function h is used to calculate the index at which the element with key κ will be stored. This process of mapping the keys to appropriate locations (or indices) in a hash table is called *hashing*.

Figure 15.4 shows a hash table in which each key from the set κ is mapped to locations generated by using a hash function. Note that keys κ_2 and κ_6 point to the same memory location. This is known as *collision*. That is, when two or more keys map to the same memory location, a collision

is said to occur. Similarly, keys κ_5 and κ_7 also collide. The main goal of using a hash function is to reduce the range of array indices that have to be handled. Thus, instead of having u values, we just need κ values, thereby reducing the amount of storage space required.

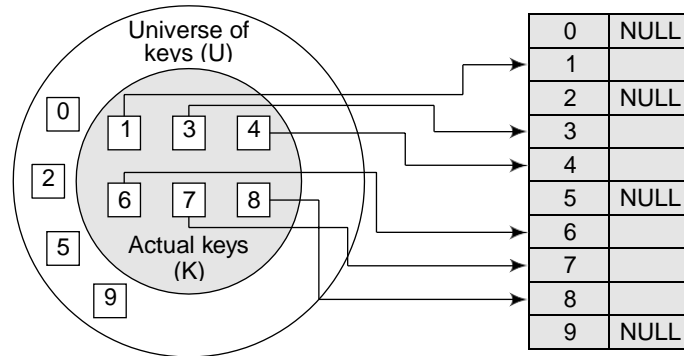


Figure 15.3 Direct relationship between key and index in the array

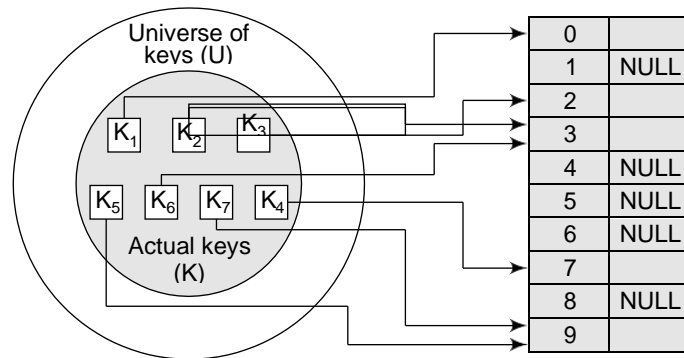


Figure 15.4 Relationship between keys and hash table index

HaSH FUNCTIONS

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table. The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions. In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

In this section, we will discuss the popular hash functions which help to minimize collisions. But before that, let us first look at the properties of a good hash function.

Properties of a Good Hash Function

Low cost The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches. For example, if binary search algorithm can search an element from a sorted table of n items with $\log_2 n$ key comparisons, then the hash function must cost less than performing $\log_2 n$ key comparisons.

Determinism A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value. However, this criteria excludes hash functions that depend

on external variable parameters (such as the time of day) and on the memory address of the object being hashed (because address of the object may change during processing).

Uniformity A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.

DIFFeReNT HaSH FUNCTIONS

In this section, we will discuss the hash functions which use numeric keys. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any of the hash functions given below can be applied to generate the hash value.

Division Method

It is the most simple method of hashing an integer x . This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

$$h(x) = x \bmod M$$

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M .

For example, suppose M is an even number then $h(x)$ is even if x is even and $h(x)$ is odd if x is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values. M should also be not too close to the exact powers of 2. If we have

$$h(x) = x \bmod 2^k$$

then the function will simply extract the lowest k bits of the binary representation of x .

The division method is extremely simple to implement. The following code segment illustrates how to do this:

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

example 15.1 Calculate the hash values of keys 1234 and 5462.

Solution Setting $M = 97$, hash values can be calculated as:

$$\begin{aligned} h(1234) &= 1234 \% 97 = 70 \\ h(5462) &= 5462 \% 97 = 16 \end{aligned}$$

Multiplication Method

The steps involved in the multiplication method are as follows:

Step 1: Choose a constant A such that $0 < A < 1$.

Step 2: Multiply the key k by A .

Step 3: Extract the fractional part of kA .

Step 4: Multiply the result of Step 3 by the size of hash table (m).

Hence, the hash function can be given as:

$$h(k) = \hat{I}_m (kA \bmod 1)^\circ$$

where $(kA \bmod 1)$ gives the fractional part of kA and m is the total number of indices in the hash table.

The greatest advantage of this method is that it works practically with any value of A . Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is

$$A = (\sqrt{5} - 1) / 2 = 0.6180339887$$

example 15.2 Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution We will use $A = 0.618033$, $m = 1000$, and $k = 12345$

$$\begin{aligned} h(12345) &= \hat{I}_{1000} (12345 \times 0.618033 \bmod 1)^\circ \\ &= \hat{I}_{1000} (7629.617385 \bmod 1)^\circ \\ &= \hat{I}_{1000} (0.617385)^\circ \\ &= \hat{I}_{617.385}^\circ \\ &= 617 \end{aligned}$$

Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2 .

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

$$h(k) = s$$

where s is obtained by selecting r digits from k^2 .

example 15.3 Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so $r = 2$.

When $k = 1234$, $k^2 = 1522756$, $h(1234) = 27$

When $k = 5642$, $k^2 = 31832164$, $h(5642) = 21$

Observe that the 3rd and 4th digits starting from the right are chosen.

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k_1, k_2, \dots, k_n where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + \dots + k_r$. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.

example 15.4 Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution

Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

COLLISIONS

As discussed earlier in this chapter, collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called *collision resolution technique*, is applied. The two most popular methods of resolving collisions are:

1. Open addressing
2. Chaining

In this section, we will discuss both these techniques in detail.

Collision Resolution by Open addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table. The hash table contains two types of values: *sentinel values* (e.g., -1) and *data values*. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an OVERFLOW condition.

The process of examining memory locations in the hash table is called *probing*. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h(k) + i] \text{ mod } m$$

Where m is the size of the hash table, $h_i(k) = (k \bmod m + i)$, and i is the probe number that varies from 0 to $m-1$.

Therefore, for a given key k , first the location generated by $[h_0(k) \bmod m]$ is probed because for the first time $i=0$. If the location is free, the value is stored in it, else the second probe generates the address of the location given by $[h_1(k) \bmod m]$. Similarly, if the location is occupied, then subsequent probes generate the address as $[h_2(k) \bmod m]$, $[h_3(k) \bmod m]$, $[h_4(k) \bmod m]$, $[h_5(k) \bmod m]$, and so on, until a free location is found.

Note Linear probing is known for its simplicity. When we have to store a value, we try the slots: $[h_0(k) \bmod m]$, $[h_1(k) \bmod m]$, $[h_2(k) \bmod m]$, $[h_3(k) \bmod m]$, $[h_4(k) \bmod m]$, $[h_5(k) \bmod m]$, and so on, until a vacant location is found.

example 15.5 Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Let $h_i(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

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

Step 1 Key = 72

$$\begin{aligned} h(72, 0) &= (72 \bmod 10 + 0) \bmod 10 \\ &= (2) \bmod 10 \\ &= 2 \end{aligned}$$

Since $\tau[2]$ is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27

$$\begin{aligned} h(27, 0) &= (27 \bmod 10 + 0) \bmod 10 \\ &= (7) \bmod 10 \\ &= 7 \end{aligned}$$

Since $\tau[7]$ is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36

$$\begin{aligned} h(36, 0) &= (36 \bmod 10 + 0) \bmod 10 \\ &= (6) \bmod 10 \\ &= 6 \end{aligned}$$

Since $\tau[6]$ is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key = 24

$$h(24, 0) = (24 \bmod 10 + 0) \bmod 10$$

$$= (4) \bmod 10$$

$$= 4$$

Since $\tau[4]$ is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5

Key = 63

$$h(63, 0) = (63 \bmod 10 + 0) \bmod 10$$

$$= (3) \bmod 10$$

$$= 3$$

Since $\tau[3]$ is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6

Key = 81

$$h(81, 0) = (81 \bmod 10 + 0) \bmod 10$$

$$= (1) \bmod 10$$

$$= 1$$

Since $\tau[1]$ is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	-1	36	27	-1	-1

Step 7

Key = 92

$$h(92, 0) = (92 \bmod 10 + 0) \bmod 10$$

$$= (2) \bmod 10$$

$$= 2$$

Now $\tau[2]$ is occupied, so we cannot store the key 92 in $\tau[2]$. Therefore, try again for the next location. Thus probe, $i = 1$, this time.

Key = 92

$$h(92, 1) = (92 \bmod 10 + 1) \bmod 10$$

$$= (2 + 1) \bmod 10$$

$$= 3$$

Now $\tau[3]$ is occupied, so we cannot store the key 92 in $\tau[3]$. Therefore, try again for the next location. Thus probe, $i = 2$, this time.

Key = 92

$$h(92, 2) = (92 \bmod 10 + 2) \bmod 10$$

$$= (2 + 2) \bmod 10$$

$$= 4$$

Now $\tau[4]$ is occupied, so we cannot store the key 92 in $\tau[4]$. Therefore, try again for the next location. Thus probe, $i = 3$, this time.

Key = 92

$$h(92, 3) = (92 \bmod 10 + 3) \bmod 10$$

$$= (2 + 3) \bmod 10$$

$$= 5$$

Since $\tau[5]$ is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

Step 8

Key = 101

$$\begin{aligned}
 h(101, 0) &= (101 \bmod 10 + 0) \bmod 10 \\
 &= (1) \bmod 10 \\
 &= 1
 \end{aligned}$$

Now $\tau[1]$ is occupied, so we cannot store the key 101 in $\tau[1]$. Therefore, try again for the next location. Thus probe, $i = 1$, this time.

Key = 101

$$\begin{aligned}
 h(101, 1) &= (101 \bmod 10 + 1) \bmod 10 \\
 &= (1 + 1) \bmod 10 \\
 &= 2
 \end{aligned}$$

$\tau[2]$ is also occupied, so we cannot store the key in this location. The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.

Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table.

While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as $O(1)$. If the key does not match, then the search function begins a sequential search of the array that continues until:

- ∑ the value is found, or
- ∑ the search function encounters a vacant location in the array, indicating that the value is not present, or
- ∑ the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may have to make $n-1$ comparisons, and the running time of the search algorithm may take $O(n)$ time. The worst case will be encountered when after scanning all the $n-1$ elements, the value is either present at the last location or not present in the table.

Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Linear probing finds an empty location by doing a linear search in the array beginning from position $h(k)$. Although the algorithm provides good memory caching through good locality of reference, the drawback of this algorithm is that it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place. The performance of linear probing is sensitive to the distribution of input values.

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted into the table at a position which is already occupied, that value is inserted at the end of the cluster, which again increases the length of the cluster. Generally, an insertion is made between two clusters that are separated by one vacant location. But with linear probing, there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the

probes that are required to find a free location and lesser is the performance. This phenomenon is called *primary clustering*. To avoid primary clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h(k) + c_1 i + c_2 i^2] \text{ mod } m$$

where m is the size of the hash table, $h(k) = (k \text{ mod } m)$, i is the probe number that varies from 0 to $m-1$, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key k , first the location generated by $h(k) \text{ mod } m$ is probed. If the location is free, the value is stored in it, else subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number i . Although quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table, the values of c_1, c_2 , and m need to be constrained.

example 15.6 Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

Solution

Let $h(k) = k \text{ mod } m, m = 10$

Initially, the hash table can be given as:

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

We have,

$$h(k, i) = [h(k) + c_1 i + c_2 i^2] \text{ mod } m$$

Step 1 Key = 72

$$\begin{aligned} h(72, 0) &= [72 \text{ mod } 10 + 1 \times 0 + 3 \times 0] \text{ mod } 10 \\ &= [72 \text{ mod } 10] \text{ mod } 10 \\ &= 2 \text{ mod } 10 \\ &= 2 \end{aligned}$$

Since $T[2]$ is vacant, insert the key 72 in $T[2]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27

$$\begin{aligned} h(27, 0) &= [27 \text{ mod } 10 + 1 \times 0 + 3 \times 0] \text{ mod } 10 \\ &= [27 \text{ mod } 10] \text{ mod } 10 \\ &= 7 \text{ mod } 10 \\ &= 7 \end{aligned}$$

Since $T[7]$ is vacant, insert the key 27 in $T[7]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36

$$\begin{aligned}h(36, 0) &= [36 \bmod 10 + 1 \cdot \cancel{0} + 3 \cdot \cancel{0}] \bmod 10 \\ &= [36 \bmod 10] \bmod 10 \\ &= 6 \bmod 10 \\ &= 6\end{aligned}$$

Since $\tau[6]$ is vacant, insert the key 36 in $\tau[6]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key = 24

$$\begin{aligned}h(24, 0) &= [24 \bmod 10 + 1 \cdot \cancel{0} + 3 \cdot \cancel{0}] \bmod 10 \\ &= [24 \bmod 10] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4\end{aligned}$$

Since $\tau[4]$ is vacant, insert the key 24 in $\tau[4]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 Key = 63

$$\begin{aligned}h(63, 0) &= [63 \bmod 10 + 1 \cdot \cancel{0} + 3 \cdot \cancel{0}] \bmod 10 \\ &= [63 \bmod 10] \bmod 10 \\ &= 3 \bmod 10 \\ &= 3\end{aligned}$$

Since $\tau[3]$ is vacant, insert the key 63 in $\tau[3]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81

$$\begin{aligned}h(81, 0) &= [81 \bmod 10 + 1 \cdot \cancel{0} + 3 \cdot \cancel{0}] \bmod 10 \\ &= [81 \bmod 10] \bmod 10 \\ &= 81 \bmod 10 \\ &= 1\end{aligned}$$

Since $\tau[1]$ is vacant, insert the key 81 in $\tau[1]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key = 101

$$\begin{aligned}h(101, 0) &= [101 \bmod 10 + 1 \cdot \cancel{0} + 3 \cdot \cancel{0}] \bmod 10 \\ &= [101 \bmod 10 + 0] \bmod 10 \\ &= 1 \bmod 10 \\ &= 1\end{aligned}$$

Since $\tau[1]$ is already occupied, the key 101 cannot be stored in $\tau[1]$. Therefore, try again for next location. Thus probe, $i = 1$, this time.

$$\begin{aligned}\text{Key} &= 101 \\ h(101, 1) &= [101 \bmod 10 + 1 \cdot \cancel{1} + 3 \cdot \cancel{1}] \bmod 10\end{aligned}$$

$$\begin{aligned}
&= [101 \bmod 10 + 1 + 3] \bmod 10 \\
&= [101 \bmod 10 + 4] \bmod 10 \\
&= [1 + 4] \bmod 10 \\
&= 5 \bmod 10 \\
&= 5
\end{aligned}$$

Since $\tau[5]$ is vacant, insert the key 101 in $\tau[5]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Searching a Value using Quadratic Probing

While searching a value using the quadratic probing technique, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If the desired key value matches with the key value at that location, then the element is present in the hash table and the search is said to be successful. In this case, the search time is given as $O(1)$. However, if the value does not match, then the search function begins a sequential search of the array that continues until:

- ∑ the value is found, or
- ∑ the search function encounters a vacant location in the array, indicating that the value is not present, or
- ∑ the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may take $n-1$ comparisons, and the running time of the search algorithm may be $O(n)$. The worst case will be encountered when after scanning all the $n-1$ elements, the value is either present at the last location or not present in the table.

Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Quadratic probing resolves the primary clustering problem that exists in the linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives a better cache performance.

One of the major drawbacks of quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens, then we will not be able to find an empty location in the table despite the fact that the table is by no means full. In Example 15.6 try to insert the key 92 and you will encounter this problem.

Although quadratic probing is free from primary clustering, it is still liable to what is known as *secondary clustering*. It means that if there is a collision between two keys, then the same probe sequence will be followed for both. With quadratic probing, the probability for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.

Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

Double Hashing

To start with, double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function,

hence the name *double hashing*. In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:

$$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$$

where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$, i is the probe number that varies from 0 to $m-1$, and m' is chosen to be less than m . We can choose $m' = m-1$ or $m-2$.

When we have to insert a key k in the hash table, we first probe the location given by applying $[h_1(k) \bmod m]$ because during the first probe, $i = 0$. If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of $[h_2(k) \bmod m]$ from the previous location. Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Pros and Cons

Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.

example 15.7 Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take $h_1 = (k \bmod 10)$ and $h_2 = (k \bmod 8)$.

Solution

Let $m = 10$

Initially, the hash table can be given as:

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

We have,

$$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$$

Step 1

Key = 72

$$\begin{aligned} h(72, 0) &= [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10 \\ &= [2 + (0 \times 0)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

Since $\tau[2]$ is vacant, insert the key 72 in $\tau[2]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2

Key = 27

$$\begin{aligned} h(27, 0) &= [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10 \\ &= [7 + (0 \times 3)] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

Since $\tau[7]$ is vacant, insert the key 27 in $\tau[7]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3

Key = 36

$$\begin{aligned}h(36, 0) &= [36 \bmod 10 + (0 \text{ \textasciitilde } 36 \bmod 8)] \bmod 10 \\ &= [6 + (0 \text{ \textasciitilde } 4)] \bmod 10 \\ &= 6 \bmod 10 \\ &= 6\end{aligned}$$

Since $\tau[6]$ is vacant, insert the key 36 in $\tau[6]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4

Key = 24

$$\begin{aligned}h(24, 0) &= [24 \bmod 10 + (0 \text{ \textasciitilde } 24 \bmod 8)] \bmod 10 \\ &= [4 + (0 \text{ \textasciitilde } 0)] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4\end{aligned}$$

Since $\tau[4]$ is vacant, insert the key 24 in $\tau[4]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5

Key = 63

$$\begin{aligned}h(63, 0) &= [63 \bmod 10 + (0 \text{ \textasciitilde } 63 \bmod 8)] \bmod 10 \\ &= [3 + (0 \text{ \textasciitilde } 7)] \bmod 10 \\ &= 3 \bmod 10 \\ &= 3\end{aligned}$$

Since $\tau[3]$ is vacant, insert the key 63 in $\tau[3]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6

Key = 81

$$\begin{aligned}h(81, 0) &= [81 \bmod 10 + (0 \text{ \textasciitilde } 81 \bmod 8)] \bmod 10 \\ &= [1 + (0 \text{ \textasciitilde } 1)] \bmod 10 \\ &= 1 \bmod 10 \\ &= 1\end{aligned}$$

Since $\tau[1]$ is vacant, insert the key 81 in $\tau[1]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7

Key = 92

$$\begin{aligned}h(92, 0) &= [92 \bmod 10 + (0 \text{ \textasciitilde } 92 \bmod 8)] \bmod 10 \\ &= [2 + (0 \text{ \textasciitilde } 4)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2\end{aligned}$$

Now $\tau[2]$ is occupied, so we cannot store the key 92 in $\tau[2]$. Therefore, try again for the next location. Thus probe, $i = 1$, this time.

Key = 92

$$h(92, 1) = [92 \bmod 10 + (1 \text{ \textasciitilde } 92 \bmod 8)] \bmod 10$$

$$\begin{aligned}
 &= [2 + (1 \times 4)] \bmod 10 \\
 &= (2 + 4) \bmod 10 \\
 &= 6 \bmod 10 \\
 &= 6
 \end{aligned}$$

Now $\tau[6]$ is occupied, so we cannot store the key 92 in $\tau[6]$. Therefore, try again for the next location. Thus probe, $i = 2$, this time.

$$\begin{aligned}
 &\text{Key} = 92 \\
 h(92, 2) &= [92 \bmod 10 + (2 \times 92 \bmod 8)] \bmod 10 \\
 &= [2 + (2 \times 4)] \bmod 10 \\
 &= [2 + 8] \bmod 10 \\
 &= 10 \bmod 10 \\
 &= 0
 \end{aligned}$$

Since $\tau[0]$ is vacant, insert the key 92 in $\tau[0]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Step 8

$$\begin{aligned}
 &\text{Key} = 101 \\
 h(101, 0) &= [101 \bmod 10 + (0 \times 101 \bmod 8)] \bmod 10 \\
 &= [1 + (0 \times 5)] \bmod 10 \\
 &= 1 \bmod 10 \\
 &= 1
 \end{aligned}$$

Now $\tau[1]$ is occupied, so we cannot store the key 101 in $\tau[1]$. Therefore, try again for the next location. Thus probe, $i = 1$, this time.

$$\begin{aligned}
 &\text{Key} = 101 \\
 h(101, 1) &= [101 \bmod 10 + (1 \times 101 \bmod 8)] \bmod 10 \\
 &= [1 + (1 \times 5)] \bmod 10 \\
 &= [1 + 5] \bmod 10 \\
 &= 6
 \end{aligned}$$

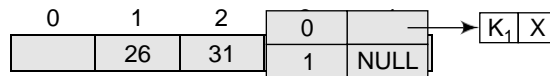
Now $\tau[6]$ is occupied, so we cannot store the key 101 in $\tau[6]$. Therefore, try again for the next location with probe $i = 2$. Repeat the entire process until a vacant location is found. You will see that we have to probe many times to insert the key 101 in the hash table. Although double hashing is a very efficient algorithm, it always requires m to be a prime number. In our case $m=10$, which is not a prime number, hence, the degradation in performance. Had m been equal to 11, the algorithm would have worked very efficiently. Thus, we can say that the performance of the technique is sensitive to the value of m .

Rehashing

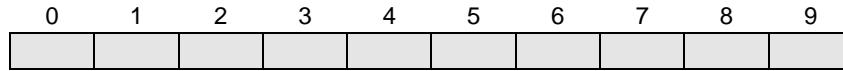
When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

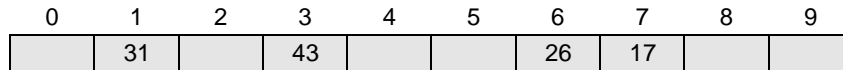
Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently. Consider the hash table of size 5 given below. The hash function used is $h(x) = x \% 5$. Rehash the entries into to a new hash table.



Note that the new hash table is of 10 locations, double the size of the original table.



Now, rehash the key values from the old hash table into the new one using hash function— $h(x) = x \% 10$.



Collision Resolution by Chaining

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l . However, if no key value hashes to l , then location l in the hash table contains NULL. Figure 15.5 shows how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.

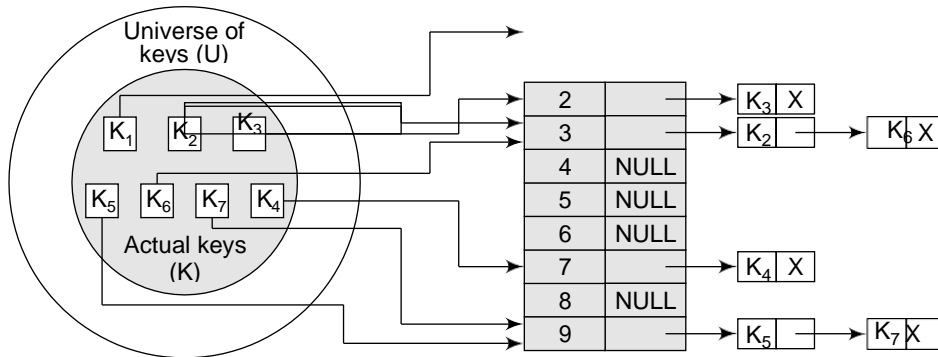


Figure 15.5 Keys being hashed to a chained hash table

Operations on a Chained Hash Table

Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key. Insertion operation appends the key to the end of the linked list pointed by the hashed location. Deleting a key requires searching the list and removing the element.

Chained hash tables with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key. The code for these algorithms is exactly the same as that for inserting, deleting, and searching a value in a single linked list that we have already studied in Chapter 6.

While the cost of inserting a key in a chained hash table is $O(1)$, the cost of deleting and searching a value is given as $O(m)$ where m is the number of elements in the list of that location. Searching and deleting takes more time because these operations scan the entries of the selected location for the desired key.

In the worst case, searching a value may take a running time of $O(n)$, where n is the number of key values stored in the chained hash table. This case arises when all the key values are inserted into the linked list of the same location (of the hash table). In this case, the hash table is ineffective.

Table 15.1 gives the code to initialize a hash table as well as the codes to insert, delete and search a value in a chained hash table.

Table 15.1 Codes to initialize, insert, delete, and search a value in a chained hash table

<p>Structure of the node</p> <pre>typedef struct node_HT { int value; struct node *next; }node;</pre>	<p>Code to insert a value</p> <pre>/* The element is inserted at the beginning of the linked list whose pointer to its head is stored in the location given by h(k). The run- ning time of the insert operation is O(1), as the new key value is always added as the first element of the list irrespective of the size of the linked list as well as that of the chained hash table. */ node *insert_value(node *hash_table[], int val) { node *new_node; new_node = (node *)malloc(sizeof(node)); new_node value = val; new_node next = hash_ table[h(x)]; hash_table[h(x)] = new_node; }</pre>
<p>Code to initialize a chained hash table</p> <pre>/* Initializes m location in the chained hash table. The operation takes a running time of O(m) */ void initializeHashTable (node *hash_ta- ble[], int m) { int i; for(i=0;i<=m;i++) hash_table[i]=NULL;</pre>	

Cont...

Cont....

Code to search a value

```

/* The element is searched in the linked
list whose pointer to its head is stored
in the location given by h(k). If search
is successful, the function returns a pointer
to the node in the linked list; otherwise
it returns NULL. The worst case running
time of the search operation is given as
order of size of the linked list. */
node *search_value(node *hash_table[],
int val)
{
    node *ptr;
    ptr = hash_table[h(x)];
    while ( (ptr!=NULL) && (ptr -> value
!= val))
        ptr = ptr -> next;
    if (ptr->value == val)
        return ptr;
else
    return NULL;
}

```

Code to delete a value

```

/* To delete a node from the linked list whose
head is stored at the location given by h(k)
in the hash table, we need to know the address
of the node's predecessor. We do this using a
pointer save. The running time complexity of
the delete operation is same as that of the
search operation because we need to search the
predecessor of the node so that the node can
be removed without affecting other nodes in the
list. */
void delete_value (node *hash_table[], int val)
{
    node *save, *ptr;
    save = NULL;
    ptr = hash_table[h(x)];
    while ((ptr != NULL) && (ptr value !=val))
    {
        save = ptr;
        ptr = ptr next;
    }
    if (ptr != NULL)
    {
        save next = ptr next;
        free (ptr);
    }
    else
        printf("\n VALUE NOT FOUND");
}

```

example 15.8 Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use $h(k) = k \text{ mod } m$.

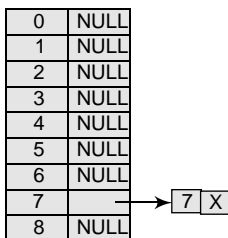
In this case, $m=9$. Initially, the hash table can be given as:

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

Step 1

$$\begin{aligned}
 \text{Key} &= 7 \\
 h(k) &= 7 \text{ mod } 9 \\
 &= 7
 \end{aligned}$$

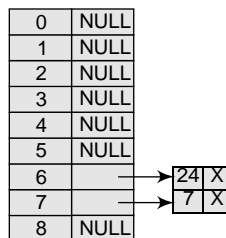
Create a linked list for location 7 and store the key value 7 in it as its only node.



Step 2

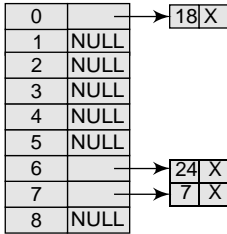
$$\begin{aligned}
 \text{Key} &= 24 \\
 h(k) &= 24 \text{ mod } 9 \\
 &= 6
 \end{aligned}$$

Create a linked list for location 6 and store the key value 24 in it as its only node.



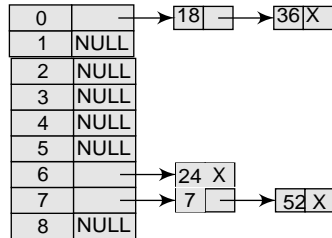
Step 3 Key = 18
 $h(k) = 18 \bmod 9 = 0$

Create a linked list for location 0 and store the key value 18 in it as its only node.



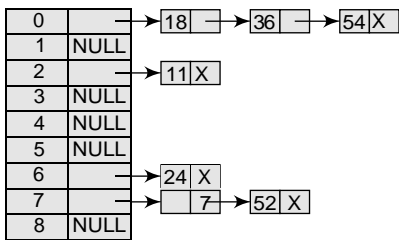
Step 5: Key = 36
 $h(k) = 36 \bmod 9 = 0$

Insert 36 at the end of the linked list of location 0.



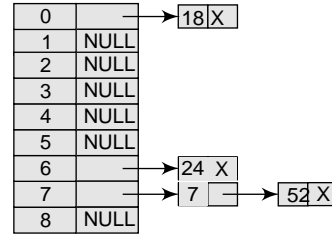
Step 7: Key = 11
 $h(k) = 11 \bmod 9 = 2$

Create a linked list for location 2 and store the key value 11 in it as its only node.



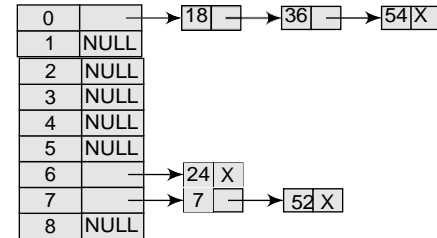
Step 4 Key = 52
 $h(k) = 52 \bmod 9 = 7$

Insert 52 at the end of the linked list of location 7.



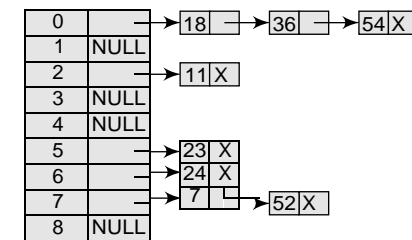
Step 6: Key = 54
 $h(k) = 54 \bmod 9 = 0$

Insert 54 at the end of the linked list of location 0.



Step 8: Key = 23
 $h(k) = 23 \bmod 9 = 5$

Create a linked list for location 5 and store the key value 23 in it as its only node.



Pros and Cons

The main advantage of using a chained hash table is that it remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table. However, with the increase in the number of keys to be stored, the performance of a chained hash table does degrade gradually (linearly). For example, a chained hash table with 1000 memory locations and 10,000 stored keys will give 5 to 10 times less performance as compared to a chained hash table with 10,000 locations. But a chained hash table is still 1000 times faster than a simple hash table.

The other advantage of using chaining for collision resolution is that its performance, unlike quadratic probing, does not degrade when the table is more than half full. This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions.

However, chained hash tables inherit the disadvantages of linked lists. First, to store a key value, the space overhead of the next pointer in each entry can be significant. Second, traversing a linked list has poor cache performance, making the processor cache ineffective.

Bucket Hashing

In closed hashing, all the records are directly stored in the hash table. Each record with a key value k is stored in a location called its home position. The home position is calculated by applying some hash function.

In case the home position of the record with key k is already occupied by another record then the record will be stored in some other location in the hash table. This other location will be determined by the technique that is used for resolving collisions. Once the records are inserted, the same algorithm is again applied to search for a specific record.

One implementation of closed hashing groups the hash table into buckets where m slots of the hash table are divided into b buckets. Therefore, each bucket contains m/b slots. Now when a new record has to be inserted, the hash function computes the home position. If the slot is free, the record is inserted. Otherwise, the bucket's slots are sequentially searched until an open slot is found. In case, the entire bucket is full, the record is inserted into an *overflow bucket*. The overflow bucket has infinite capacity at the end of the table and is shared by all the buckets.

An efficient implementation of bucket hashing will be to use a hash function that evenly distributes the records amongst the buckets so that very few records have to be inserted in the overflow bucket.

When searching a record, first the hash function is used to determine the bucket in which the record can be present. Then the bucket is sequentially searched to find the desired record. If the record is not found and the bucket still has some empty slots, then it means that the search is complete and the desired record is not present in the hash table.

However, if the bucket is full and the record has not been found, then the overflow bucket is searched until the record is found or all the records in the overflow bucket have been checked. Obviously, searching the overflow bucket can be expensive if it has too many records.

PROS AND CONS OF HaSHING

One advantage of hashing is that no extra space is required to store the index as in the case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.

On the other hand, the primary drawback of using the hashing technique for inserting and retrieving data values is that it usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.

All the more, choosing an effective hash function is more of an art than a science. It is not uncommon (in open-addressed hash tables) to create a poor hash function.

aPPLICaTIONS OF HaSHING

Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given here.

Hashing is used for database indexing. Some database management systems store a separate file known as the index file. When data has to be retrieved from a file, the key information is first searched in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

In many database systems, file and directory hashing is used in high-performance file systems. Such systems use two complementary techniques to improve the performance of file access. While

one of these techniques is caching which saves information in the memory, the other is hashing which makes looking up the file location in the memory much quicker than most other methods.

Hashing technique is used to implement compiler symbol tables in C++. The compiler uses a symbol table to keep a record of all the user-defined symbols in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols. Hashing is also widely used for Internet search engines.

Real World Applications of Hashing

CD Databases For CDs, it is desirable to have a world-wide CD database so that when users put their disk in the CD player, they get a full table of contents on their own computer’s screen. These tables are not stored on the disks themselves, i.e., the CD does not store any information about the songs, rather this information is downloaded from the database. The critical issue to solve here is that CDs have no ID numbers stored on them, so how will the computer know which CD has been put in the player? The only information that can be used is the track length, and the fact that every CD is different.

Basically, a big number is created from the track lengths, also known as a ‘signature’. This signature is used to identify a particular CD. The signature is a value obtained by hashing. For example, a number of length of 8 or 10 hexadecimal. digits is made up; the number is then sent to the database, and that database looks for the closest match. The reason being that track length may not be measured exactly.

Drivers Licenses/Insurance Cards Like our CD example, even the driver’s license numbers or insurance card numbers are created using hashing from data items that never change: date of birth, name, etc.

Sparse Matrix A sparse matrix is a two-dimensional array in which most of the entries contain a 0. That is, in a sparse array there are very few non-zero entries. Of course, we can store 2D array as it is, but this would lead to sheer wastage of valuable memory. So another possibility is to store the non-zero elements of the spare matrix as elements in a 1D array. That is by using hashing, we can store a two-dimensional array in a one-dimensional array. There is a one-to-one correspondence between the elements in the sparse matrix and the elements in the array. This concept is clearly visible in Fig. 15.6.

If the size of the sparse matrix is $n \times n$, and there are N non-zero entries in it, then from the coordinates (i, j) of a matrix, we determine an index k in an array by a simple calculation. Thus, we have $k=h(i, j)$ for some function h , called a hash function.

The size of the 1D array is proportional to N . This is far better from the size of the sparse matrix that required storage proportional to $n \times n$. For example, if we have a triangular sparse matrix A , then an entry $A[i, j]$ can be mapped to an entry in the 1D array by calculating the index using the hash function $h(i, j) = i(i-1)/2 + j$.

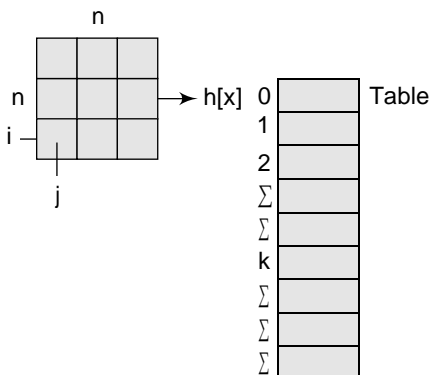


Figure 15.6 Sparse matrix

File Signatures File signatures provide a compact means of identifying files. We use a function, $h[x]$, the file signature, which is a property of the file. Although we can store files by name, signatures provide a compact identity to files.

Since a signature depends on the contents of a file, if any change is made to the file, then the signature will change. In this way, the signature of a file can be used as a quick verification to see if anyone has altered the file, or if it has lost a bit during transmission. Signatures are widely used for files that store marks of students.

Game Boards In the game board for tic-tac-toe or chess, a position in a game may be stored using a hash function.

Graphics In graphics, a central problem is the storage of objects in a scene or view. For this, we organize our data by hashing. Hashing can be used to make a grid of appropriate size, an ordinary vertical–horizontal grid. (Note that a grid is nothing but a 2D array, and there is a one-to-one correspondence when we move from a 2D array to a 1D array.)

So, we store the grid as a 1D array as we did in the case of sparse matrices. All points that fall in one cell will be stored in the same place. If a cell contains three points, then these three points will be stored in the same entry. The mapping from the grid cell to the memory location is done by using a hash function. The key advantage of this method of storage is fast execution of operations like the nearest neighbour search.

Points to rEmEmbEr

- ∑ Hash table is a data structure in which keys are mapped to array positions by a hash function. A value stored in a hash table can be searched in $O(1)$ time using a hash function which generates an address from the key.
- ∑ The storage requirement for a hash table is $O(k)$, where k is the number of keys actually used. In a hash table, an element with key k is stored at index $h(k)$, not k . This means that a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indices) in a hash table is called hashing.
- ∑ Popular hash functions which use numeric keys are division method, multiplication method, mid square method, and folding method.
- ∑ Division method divides x by M and then uses the remainder obtained. A potential drawback of this method is that consecutive keys map to consecutive hash values.
- ∑ Multiplication method applies the hash function given as $h(x) = \hat{I}_m (kA \bmod 1)$.
- ∑ Mid square method works in two steps. First, it finds k^2 and then extracts the middle r digits of the result.
- ∑ Folding method works by first dividing the key value k into parts k_1, k_2, \dots, k_r , where each part has the same number of digits except the last part which may have lesser digits than the other parts, and then obtaining the sum of $k_1 + k_2 + \dots + k_r$. The hash value is produced by ignoring the last carry, if any.
- ∑ Collisions occur when a hash function maps two different keys to the same location. Therefore, a method used to solve the problem of collisions, also called *collision resolution technique*, is applied. The two most popular methods of resolving collisions are: (a) open addressing and (b) chaining.
- ∑ Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values—either sentinel value (for example, -1) or a data value.
- ∑ Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.
- ∑ In linear probing, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h'(k) + i] \bmod m$$
 Though linear probing enables good memory caching, the drawback of this algorithm is that it results in primary clustering.
- ∑ In quadratic probing, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$
 Quadratic probing eliminates primary clustering and provides good memory caching. But it is still liable to secondary clustering.
- ∑ In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:

$$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$$
 The performance of double hashing is very close to the performance of the ideal scheme of uniform hashing. It minimizes repeated collisions and the effects of clustering.
- ∑ When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. So in rehashing, all the entries in the original hash table are moved to the new hash table which is double the size of the original hash table.

Σ In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values where m is the number of elements in the list of that location. While the cost of inserting a key in a chained hash table is $O(1)$, the cost a value may take a running time of $O(n)$.

Exercises

Review Questions

1. Define a hash table.
2. What do you understand by a hash function? Give the properties of a good hash function.
3. How is a hash table better than a direct access table (array)?
4. Write a short note on the different hash functions. Give suitable examples.
5. Calculate hash values of keys: 1892, 1921, 2007, 3456 using different methods of hashing.
6. What is collision? Explain the various techniques to resolve a collision. Which technique do you think is better and why?
7. Consider a hash table with size = 10. Using linear probing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table.
8. Consider a hash table with size = 10. Using quadratic probing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.
9. Consider a hash table with size = 11. Using double hashing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table. Take $h_1 = k \bmod 10$ and $h_2 = k \bmod 8$.
10. What is hashing? Give its applications. Also, discuss the pros and cons of hashing.
11. Explain chaining with examples.
12. Write short notes on:
 - Linear probing
 - Quadratic probing
 - Double hashing

Multiple-choice Questions

1. In a hash table, an element with key k is stored at index
 - (a) k
 - (b) $\log k$
 - (c) $h(k)$
 - (d) k^2
2. In any hash function, M should be a
 - (a) Prime number
 - (b) Composite number
 - (c) Even number
 - (d) Odd number
3. In which of the following hash functions, do consecutive keys map to consecutive hash values?
 - (a) Division method
 - (b) Multiplication method
 - (c) Folding method
 - (d) Mid-square method

4. The process of examining memory locations in a hash table is called
 - (a) Hashing
 - (b) Collision
 - (c) Probing
 - (d) Addressing
5. Which of the following methods is applied in the Berkeley Fast File System to allocate free blocks?
 - (a) Linear probing
 - (b) Quadratic probing
 - (c) Double hashing
 - (d) Rehashing
6. Which open addressing technique is free from clustering problems?
 - (a) Linear probing
 - (b) Quadratic probing
 - (c) Double hashing
 - (d) Rehashing

True or False

1. Hash table is based on the property of locality of reference.
2. Binary search takes $O(n \log n)$ time to execute.
3. The storage requirement for a hash table is $O(k^2)$, where k is the number of keys.
4. Hashing takes place when two or more keys map to the same memory location.
5. A good hash function completely eliminates collision.
6. M should not be too close to exact powers of 2.
7. A sentinel value indicates that the location contains valid data.
8. Linear probing is sensitive to the distribution of input values.
9. A chained hash table is faster than a simple hash table.

Fill in the blanks

1. In a hash table, keys are mapped to array positions by a _____.
2. _____ is the process of mapping keys to appropriate locations in a hash table.
3. In open addressing, hash table stores either of two values _____ and _____.
4. When there is no free location in the hash table then _____ occurs.
5. More the number of collisions, higher is the number of _____ to find free location _____ which eliminates primary clustering but not secondary clustering.
6. _____ eliminates primary clustering but not secondary clustering.

UNIT III: (BINARY SEARCH TREES, AVL TREES, SPLAY TREES & RED BLACK TREES)

TREE: A tree is recursively defined as a set of one or more nodes and branches(edges) where one node is designated as 'root' of the tree and all the remaining nodes can be partitioned into non-empty sets, each of which is a sub-tree with respect to root of the tree. It is a non-linear data structure compared to arrays, linked lists.

Fig. Example of a Tree

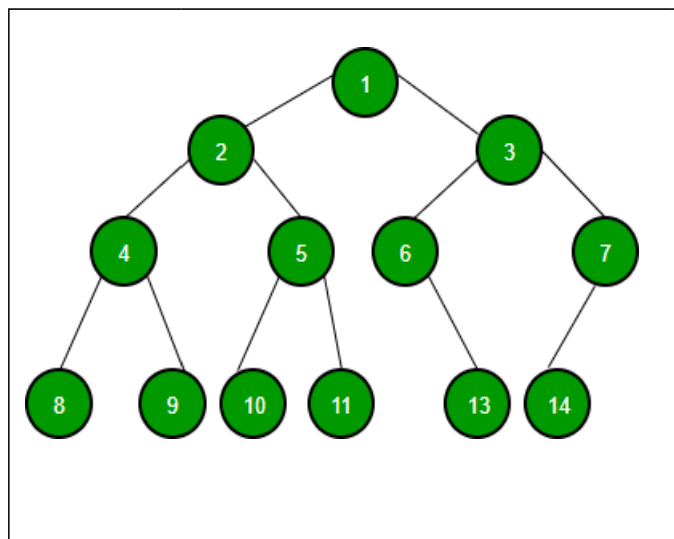


Figure 5.1.1 A Tree and a not a tree

BASIC TERMINOLOGY:

Node: A data unit to store information.

Branch or edge: The Link between two nodes.

Root node: It is the top most node in the tree.

Leaf Node: The node which does not have any children(external node).

Degree of the Node: It is the number of edges connected to that node.

In-Degree: Number of edges arriving to that node.

Out-Degree: Number of edges leaving from that node.

Level: It is the length of the path from the root to that node.

- Root node is said to be at level 0.



Depth or Height of the Tree: It is the maximum level of any node in the tree.

Internal node: A non-leaf node.

Parent: The unique predecessor of a node is termed as its parent.

Child: The successor of a node is termed as its child node.

Siblings: Nodes of the same parent node.

Path: Sequence of consecutive edges is called a path. (1-2-4-8)

Degree of the Tree: Maximum degree of a node in that tree.

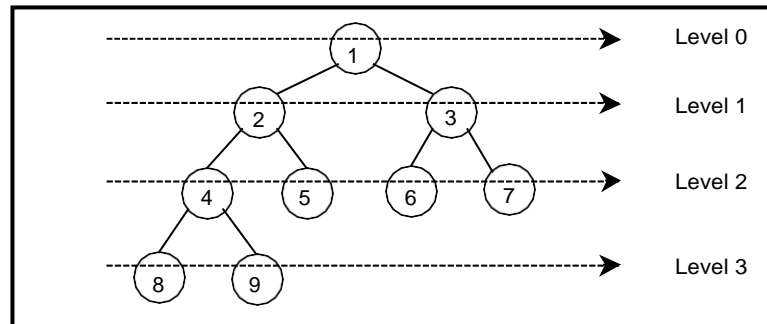


Figure : Levels tree

IMPORTANT POINTS TO BE NOTED:

1. Is Root node considered as internal node?

Any vertex (node) for which there exist one or more children are called as internal vertices, the root of a tree is an internal vertex unless it is the only vertex in the tree.

2. Degree of a node = In-Degree + Out-Degree

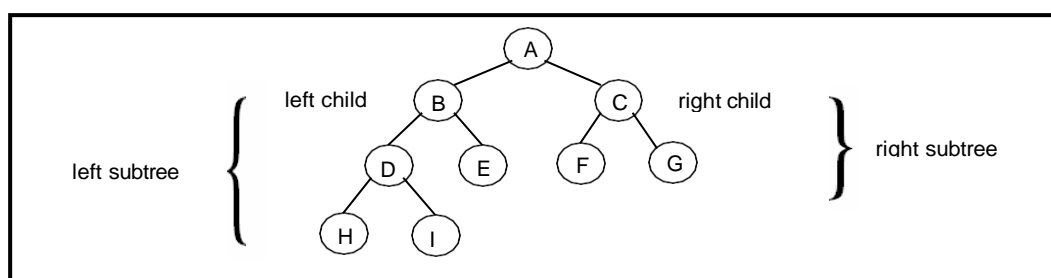
- Any node in a tree has a maximum of in-degree 'one'.
- for any node in a binary tree, the maximum out-degree 'two'.
- root node has in-degree 'zero' .
- leaf node has out-degree 'zero'.

Applications of trees:

1. Used to represent hierarchies.
2. Used to represent simple as well as complex data.
3. Used for implementing other data structures like hash tables, sets, and maps.
4. Used for compiler construction.
5. Used in data base design.
6. Used in file system directories.
7. Used in symbol tables.

BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have **at most two children**. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.



A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

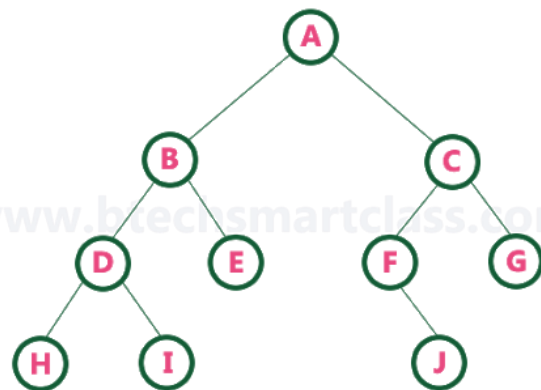
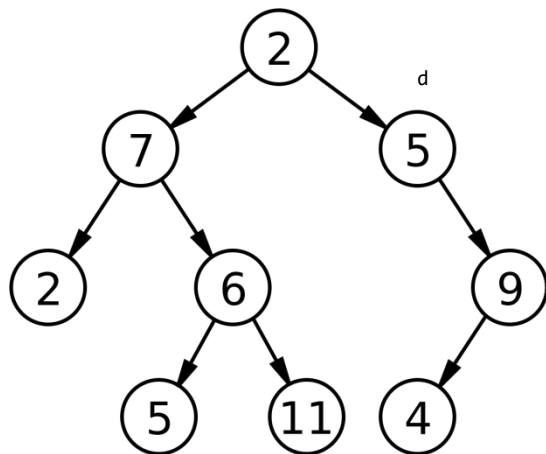
Figure 5.2.1. Binary Tree

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

- For every node in binary tree, In-degree is always 'one'.
Out-degree is always less than or equal to 'two'. Maximum degree of any node is 'three'.
- Every binary tree is a tree, but every tree is not a binary tree.

PROPERTIES OF A BINARY TREE:

1. The maximum number of nodes at level 'L' of a binary tree is 2^L .
2. Maximum number of nodes in a binary tree of height 'h' is $2^{h+1} - 1$.
3. In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.



<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>				<u>J</u>

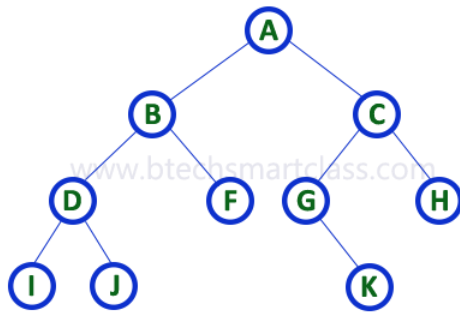
IMPLEMENTATION OF A BINARY TREE USING DOUBLY LINKED LIST:

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation 2. Linked List Representation

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

A node has an index i , its children are found at indices $2i+1$ and $2i+2$, while its parent (if any) is found at index $\text{floor}((i-1)/2)$ (assuming the root of the tree stored in the array at an index zero).

Consider the above example of a binary tree and it is represented as follows...

Root node is at $a[0]$ and its children are at $a[1]$ and $a[2]$ i.e A is at $a[0]$ and

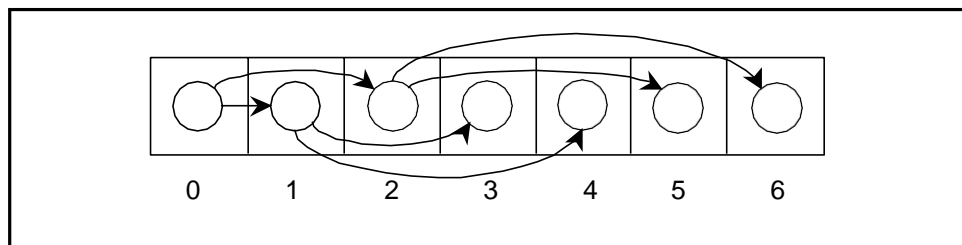
B & C are at $a[1]$ & $a[2]$.

Children of B are at $a[3]$ and $a[4]$

Children of c are at $a[5]$ and $a[6]$



To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.



2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...

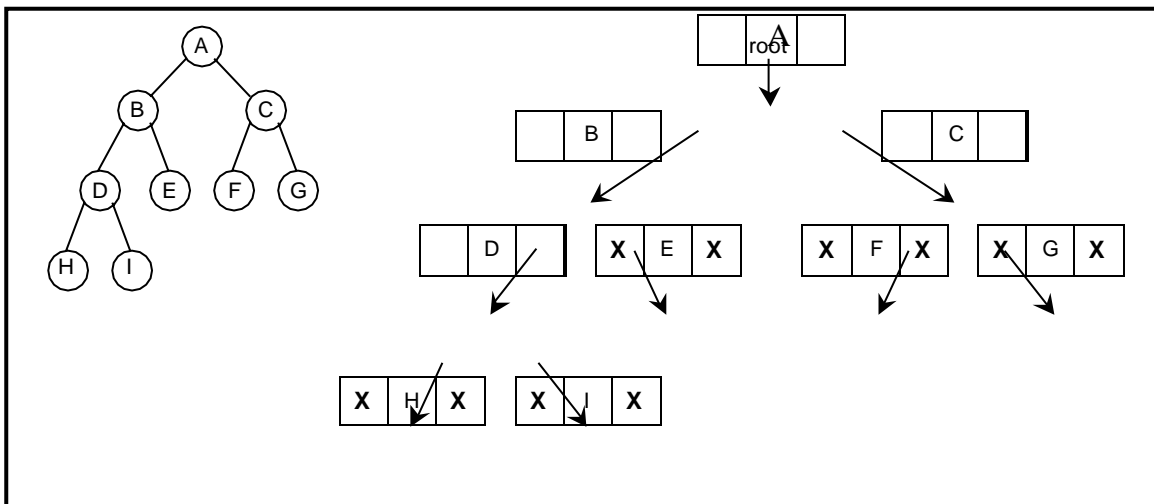
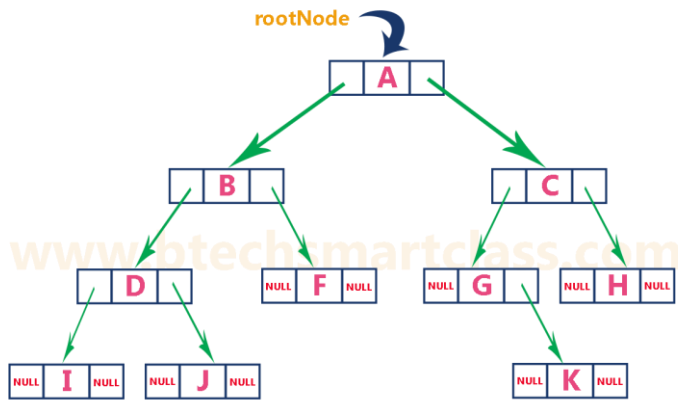


Figure: Linked representation for the binary tree

Typically, creation of a node is done as, A Node contains:

1. Data
2. Pointer to left child
3. Pointer to right child

```

struct node {
    int data;
    struct node *leftChild;
    struct node *rightChild;
};

```

Then respective links are given to the nodes, creating branches between the nodes by using two

self-referencing structure pointers.

TYPES OF BINARY TREES:

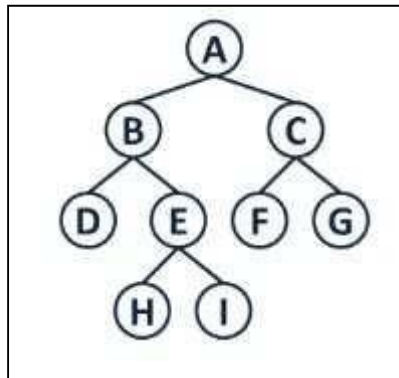
There are different types of binary trees,

1. Full Binary Tree
2. Complete Binary Tree
3. PERFECT

1. FULL OR STRICTLY BINARY TREE

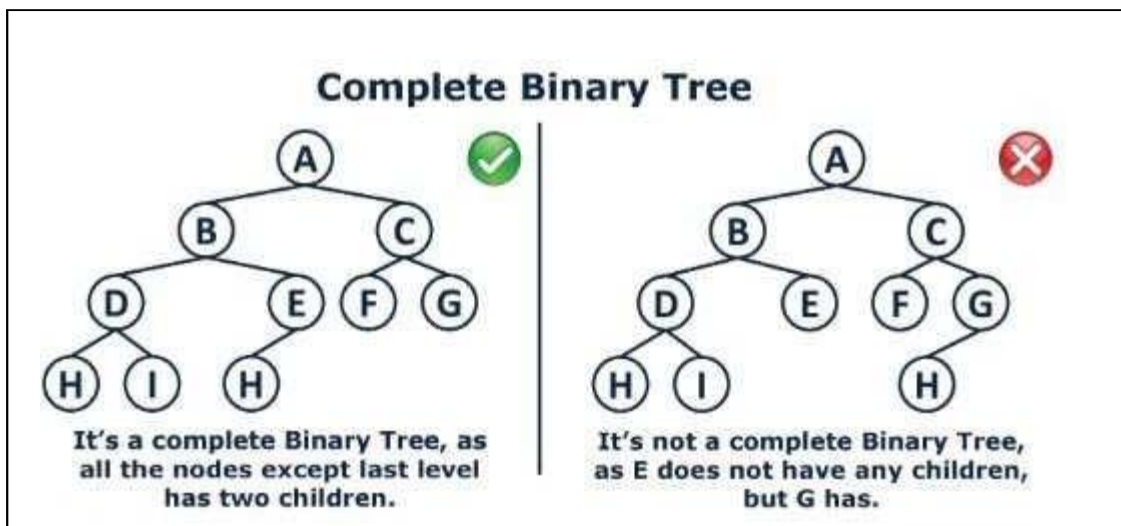
A Binary Tree is said to be a Full or Strictly Binary Tree, If all the nodes other than leaf nodes has 0 or 2 children, then that is Full Binary Tree.

- All the nodes in a Full or Strictly Binary Tree are of out-degree zero or two, never degree one.



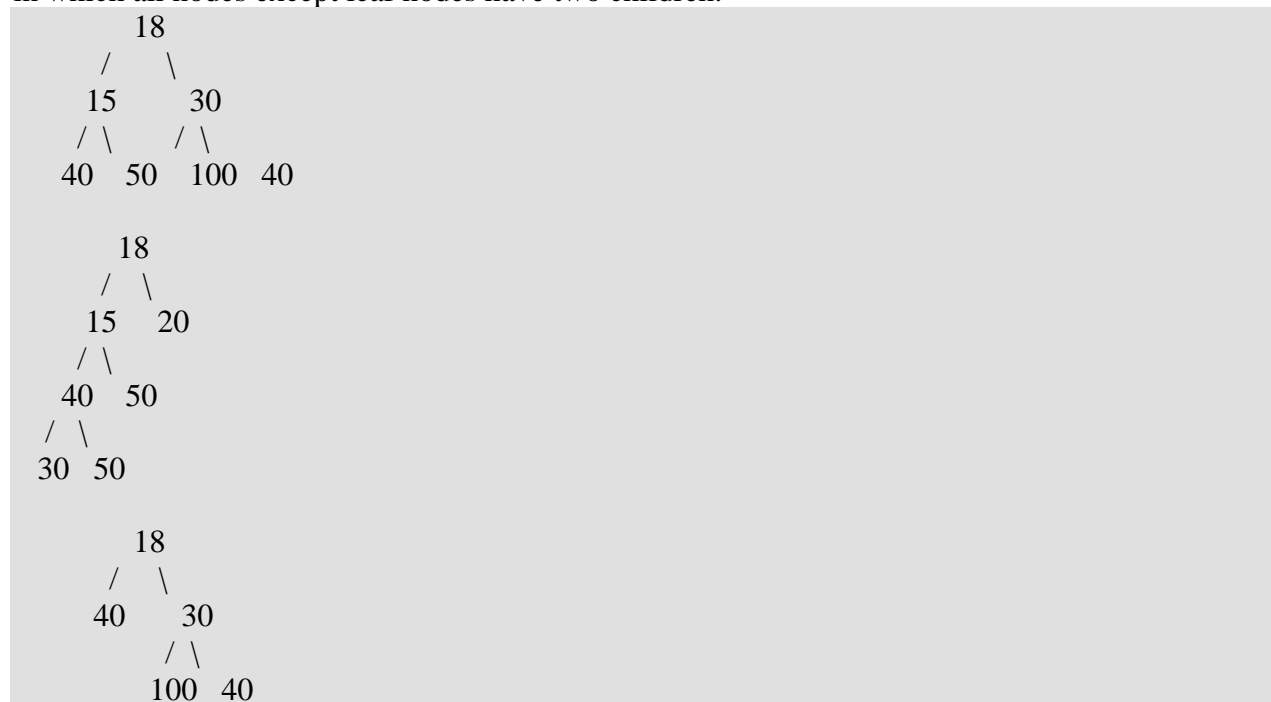
2. COMPLETE BINARY TREE

A Binary Tree is said to be Complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all the nodes as left as possible.



3. Perfect Binary Tree A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

Full Binary Tree A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



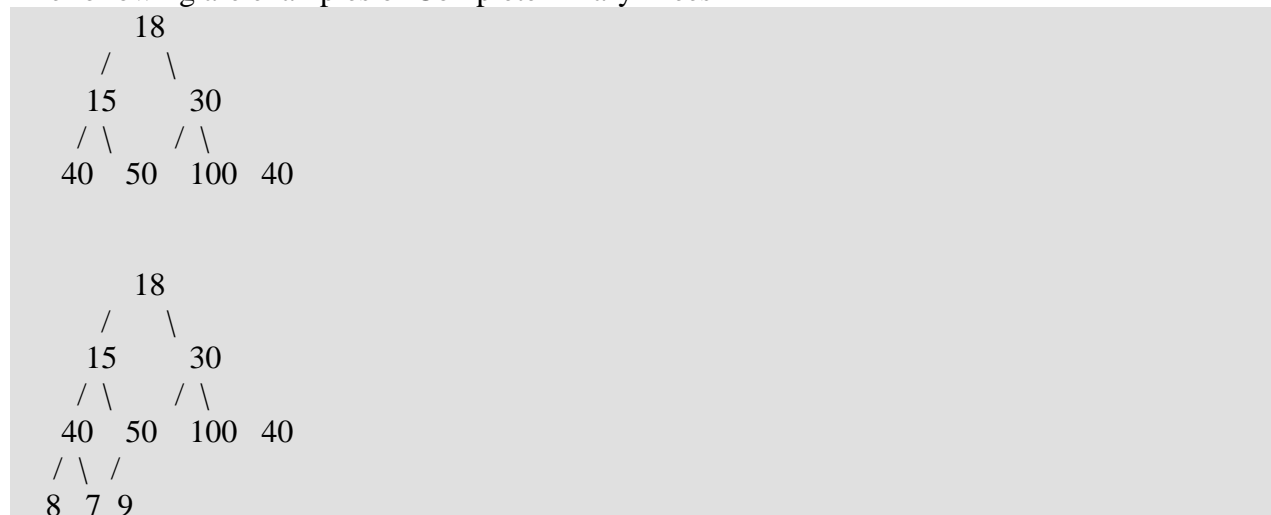
In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1

$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree:

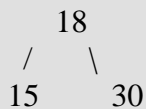
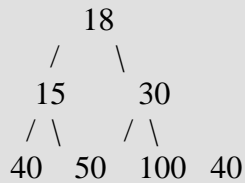
The following are examples of Complete Binary Trees



Practical example of Complete Binary Tree is [Binary Heap](#).

Perfect Binary Tree A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are the examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^{h+1} - 1$ nodes.

Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*(**NODE**, LCHILD, RCHILD)NLR
2. *Inorder* (LCHILD, **NODE**, RCHILD)LNR
3. *Postorder* (LCHILD, RCHILD, **NODE**)LRN

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

Recursive Traversal Algorithms:

Inorder Traversal(LNR):

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```

void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);

        print root -> data;
    }
}
  
```

```

        inorder(root->rchild);
    }
}

```

Preorder Traversal(NRL):

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```

void preorder(node *root)
{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
}

```

Postorder Traversal (LRN):

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

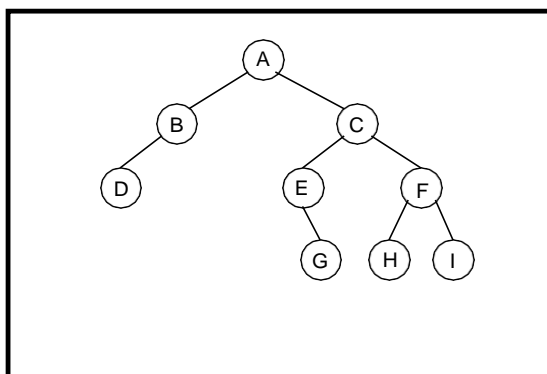
The algorithm for postorder traversal is as follows:

```

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}

```

Traverse the following binary tree in pre, post, inorder and level order.



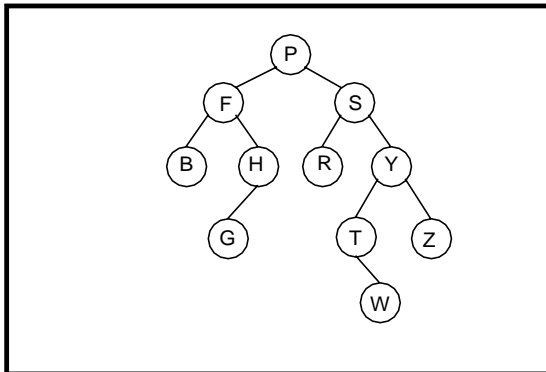
Binary Tree

- Preorder traversal yields: A, B, D, C, E, G, F, H, I
- Postorder traversal yields: D, B, G, E, H, I, F, C, A
- Inorder traversal yields: D, B, A, E, G, C, H, F, I

Pre, Post, Inorder Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



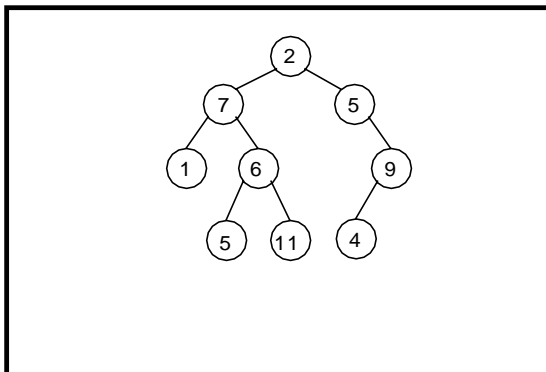
- Preorder traversal yields:
P , F , B , H , G , S , R , Y , T , W , Z
- Postorder traversal yields:
B , G , H , F , R , W , T , Z , Y , S , P
- Inorder traversal yields:
B , F , G , H , P , R , S , T , W , Y , Z

Binary Tree

Pre, Post, Inorder Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



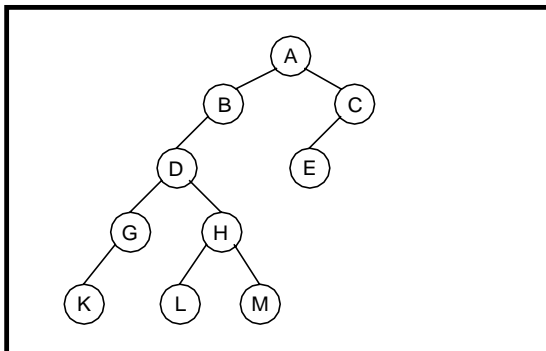
- Preorder traversal yields
2, 7, 1, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
1, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
1, 7, 5, 6, 11, 2, 5, 4, 9

Binary Tree

Pre, Post, Inorder Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



- Preorder traversal yields: A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields: K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Binary Tree

Pre, Post, Inorder Traversing

Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F

Inorder: D G B A H E I C F

Solution:

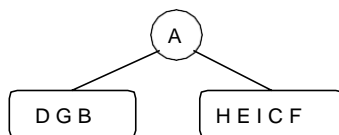
From Preorder sequence A B D G C E H I F, the root is: A

From Inorder sequence D G B A H E I C F, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

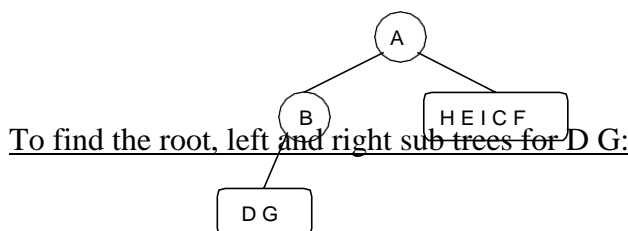


To find the root, left and right sub trees for D G B:

From the preorder sequence B D G, the root of tree is: B

From the inorder sequence D G B, we can find that D and G are to the left of B.

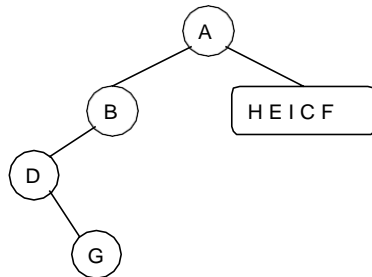
The Binary tree upto this point looks like:



From the preorder sequence **D G**, the root of the tree is: *D*

From the inorder sequence **D G**, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

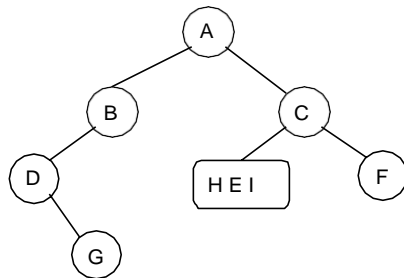


To find the root, left and right sub trees for H E I C F:

From the preorder sequence **C E H I F**, the root of the left sub tree is: *C*

From the inorder sequence **H E I C F**, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

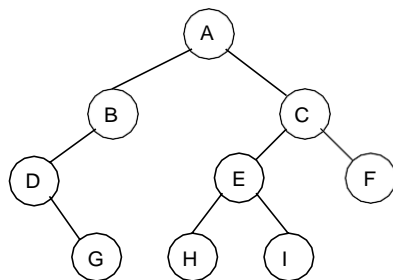


To find the root, left and right sub trees for H E I:

From the preorder sequence **E H I**, the root of the tree is: *E*

From the inorder sequence **H E I**, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F

Postorder: G D B H I E F C A

Solution:

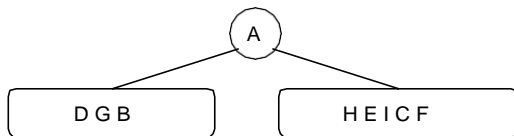
From Postorder sequence G D B H I E F C **A**, the root is: A

From Inorder sequence D G B A H E I C F, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

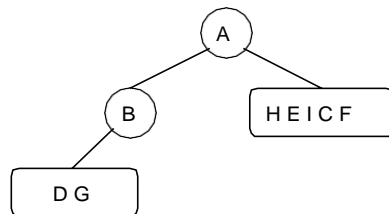


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence D G B, we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

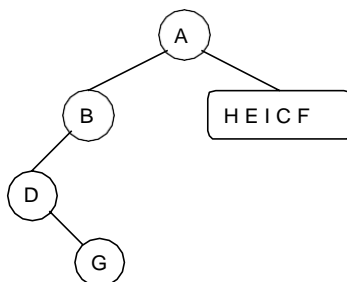


To find the root, left and right sub trees for D G:

From the postorder sequence G D, the root of the tree is: D

From the inorder sequence **D** G, we can find that there is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:

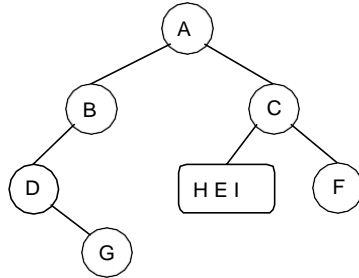


To find the root, left and right sub trees for H E I C F:

From the postorder sequence $H I E F C$, the root of the left sub tree is: C

From the inorder sequence $\underline{H E I} C \underline{F}$, we can find that $H E I$ are to the left of C and F is the right subtree for C .

The Binary tree upto this point looks like:

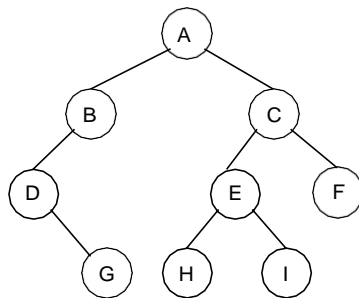


To find the root, left and right sub trees for $H E I$:

From the postorder sequence $H I E$, the root of the tree is: E

From the inorder sequence $\underline{H} \underline{E} \underline{I}$, we can find that H is left subtree for E and I is to the right of E .

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: $n1 n2 n3 n4 n5 n6 n7 n8 n9$

Preorder: $n6 n2 n1 n4 n3 n5 n9 n7 n8$

Solution:

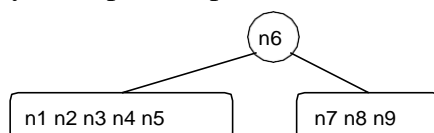
From Preorder sequence $\mathbf{n6} n2 n1 n4 n3 n5 n9 n7 n8$, the root is: $n6$

From Inorder sequence $\underline{n1 n2 n3 n4 n5} \mathbf{n6} \underline{n7 n8 n9}$, we get the left and right sub trees:

Left sub tree is: $n1 n2 n3 n4 n5$

Right sub tree is: $n7 n8 n9$

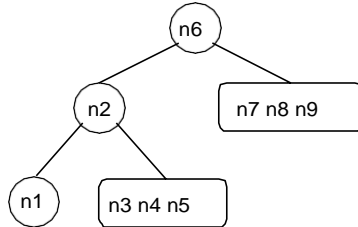
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

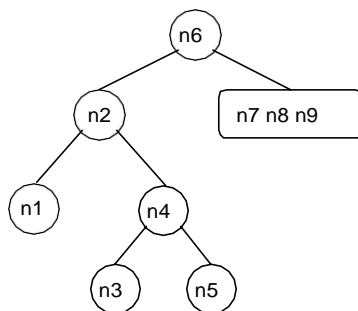


To find the root, left and right sub trees for n3 n4 n5:

From the preorder sequence **n4** n3 n5, the root of the tree is: n4

From the inorder sequence n3 **n4** n5, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

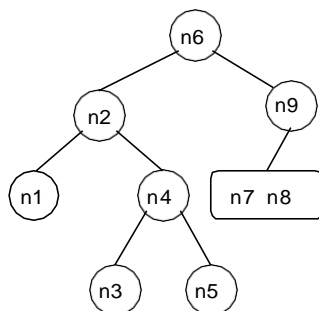


To find the root, left and right sub trees for n7 n8 n9:

From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9

From the inorder sequence n7 n8 **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

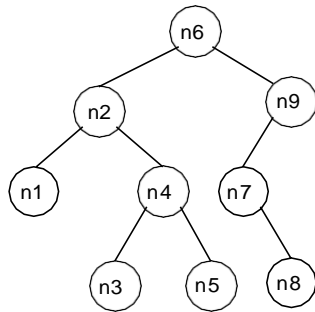


To find the root, left and right sub trees for n7 n8:

From the preorder sequence **n7** n8, the root of the tree is: n7

From the inorder sequence **n7 n8**, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

Solution:

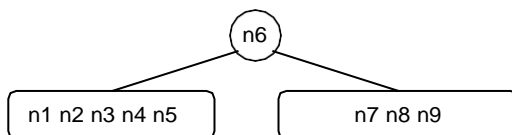
From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, the root is: n6

From Inorder sequence n1 n2 n3 n4 n5 **n6** n7 n8 n9, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:

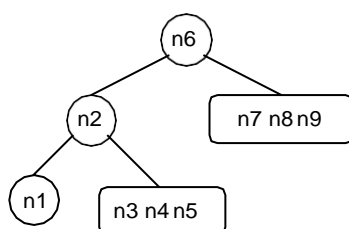


To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the postorder sequence n1 n3 n5 n4 **n2**, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

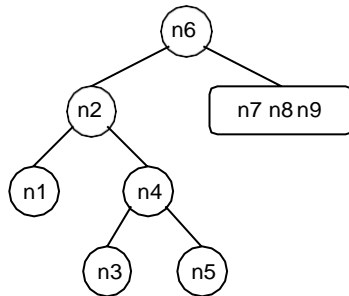
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

*From the postorder sequence n3 n5 **n4**, the root of the tree is: n4*

From the inorder sequence n3 n4 n5, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

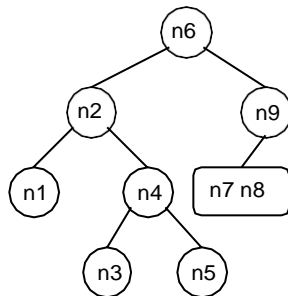


To find the root, left and right sub trees for n7 n8 and n9:

*From the postorder sequence n8 n7 **n9**, the root of the left sub tree is: n9*

From the inorder sequence n7 n8 n9, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

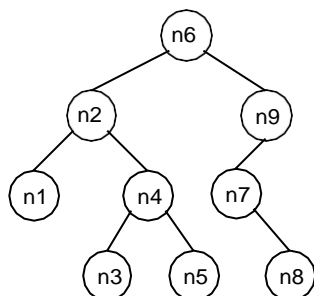
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

*From the postorder sequence n8 **n7**, the root of the tree is: n7*

From the inorder sequence **n7** n8, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



// BINARY TREE ADT IMPLEMENTATION

```
#include<stdio.h>
#include<stdlib.h>
struct node* create();
void preorder(struct node *);
void postorder(struct node *);
void inorder(struct node *);

struct node
{
    int data;
    struct node *left;
    struct node *right;
};

void main()
{
    struct node* root;
    int ch;
    system("clear");
    while(1)
    {
        printf("\n _____");
        printf("\n BINARY TREE ADT OPERATIONS ARE:\n");
        printf("_____");
        printf("\n\t1.CREATE");
        printf("\n\t2.PREORDER");
        printf("\n\t3.INORDER");
        printf("\n\t4.POSTORDER");
        printf("\n\t5.EXIT");
        printf("\n Enter ur choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: root=create();
                    break;
            case 2: printf("\n The preorder traversal of tree is:");
                    preorder(root);
                    break;
            case 3: printf("\n The inorder traversal of tree is:");
                    inorder(root);
                    break;
            case 4: printf("\n The postorder traversal of tree is:");
                    postorder(root);
                    break;
            case 5: exit(0);
                    break;
            default: printf("\n wrong choice\n");
        }
    }
}
```



```

    }

struct node* create()
{
    struct node *p;
    int x;
    printf("enter node data(-1 for no data):");
    scanf("%d",&x);
    if(x==-1)
        return NULL;

    p=(struct node*)malloc(sizeof(struct node));
    p->data=x;
    printf("\nEnter left child of %d:\n",x);
    p->left=create();
    printf("\nEnter right child of %d:\n",x);
    p->right=create();
    return p;
}

void preorder(struct node *t)
{
    if(t!=NULL)
    {
        printf("\n%d",t->data);
        preorder(t->left);
        preorder(t->right);
    }
}

void inorder(struct node *t)
{
    if(t!=NULL)
    {
        inorder(t->left);
        printf("\n%d",t->data);
        inorder(t->right);
    }
}

void postorder(struct node *t)
{
    if(t!=NULL)
    {
        postorder(t->left);
        postorder(t->right);
        printf("\n%d",t->data);
    }
}

```

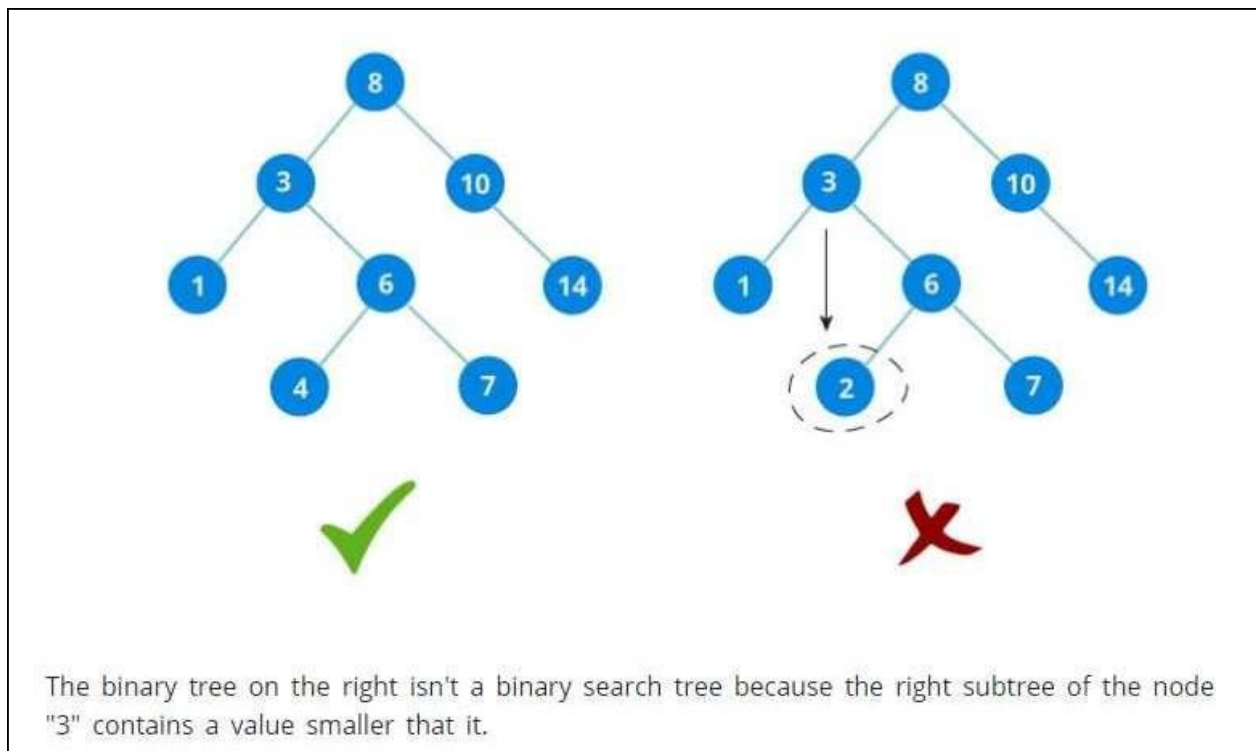
BINARY SEARCH TREES (BST):

Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree. **IT HAS ALL THE PROPERTIES OF A BINARY TREE.**

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- This rule will be recursively applied to all the left and right sub-trees of the root.

Rule of Binary Search Tree: Left Child \leq Root \leq Right Child



NOTE: Every Binary Search Tree is a Binary Tree, But not every Binary Tree is a Binary Search Tree.

Advantages of using Binary Search Tree:

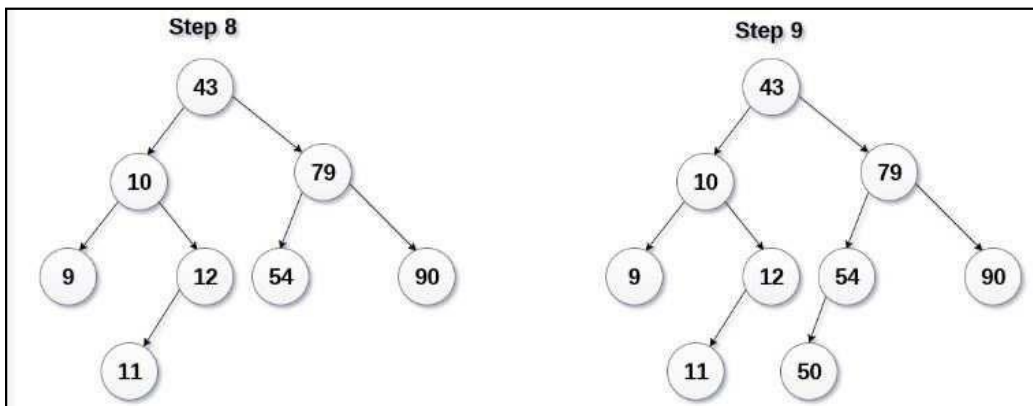
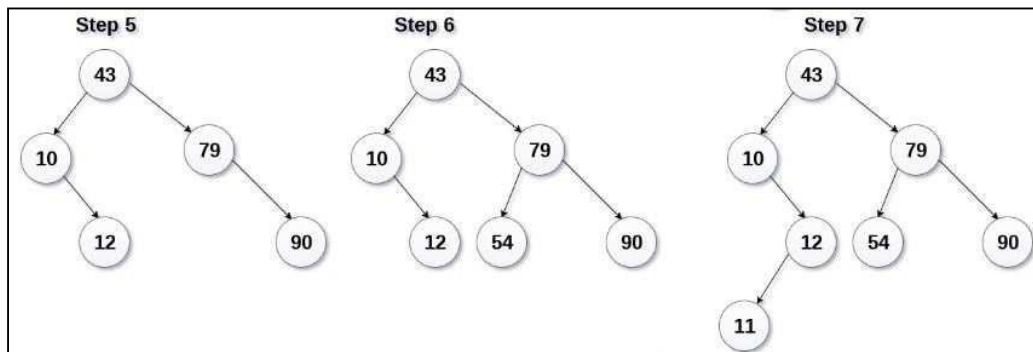
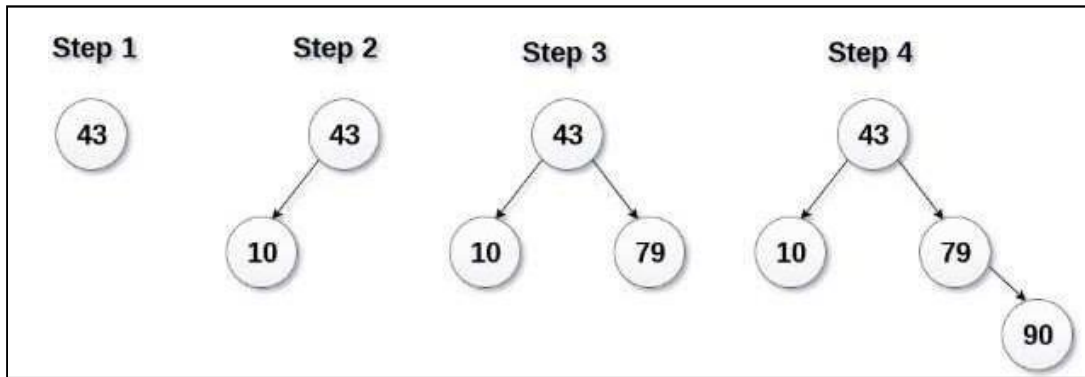
1. Searching become very efficient in a binary search tree since, we skip half of the elements in each step of search process.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

4. It is called a search tree because it can be used to search for the presence of a number in $O(\log(n))$ time.

CREATION OF BINARY SEARCH TREE:

Q. Create the binary search tree using the following data elements. 43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.
4. Continue the process recursively for all the entries.



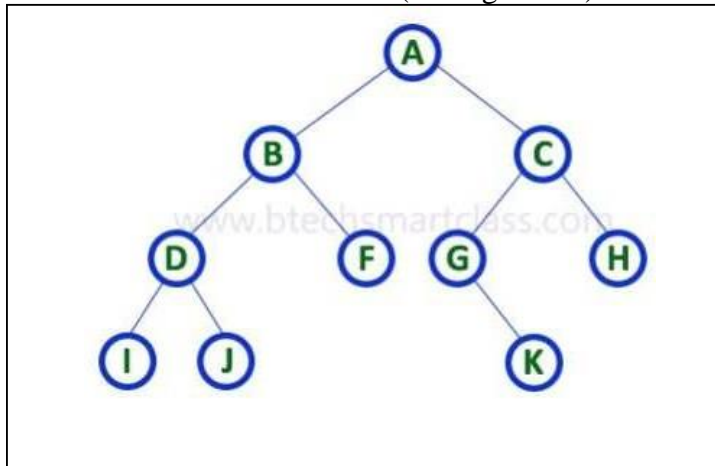
NOTE: If you observe carefully, we can see that the left-most element is the smallest, and right-most element is the largest in a binary search tree.

BINARY SEARCH TREE TRAVERSALS:

Traversal: Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are mainly three types of Traversal techniques for BST:

1. Pre-Order Traversal (root-left-right)
2. In-Order Traversal (left-root-right)
3. Post-Order Traversal (left-right-root) Consider the following binary tree...



1. In - Order Traversal (Left Child - Root - Right Child)

In In-Order traversal, the root node is visited between the left child and right child.

In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (Root - Left Child - Right Child)

In Pre-Order traversal, the root node is visited before the left child and right child nodes.

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

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H.

After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (Left Child - Right Child - Root)

In Post-Order traversal, the root node is visited after left child and right child.

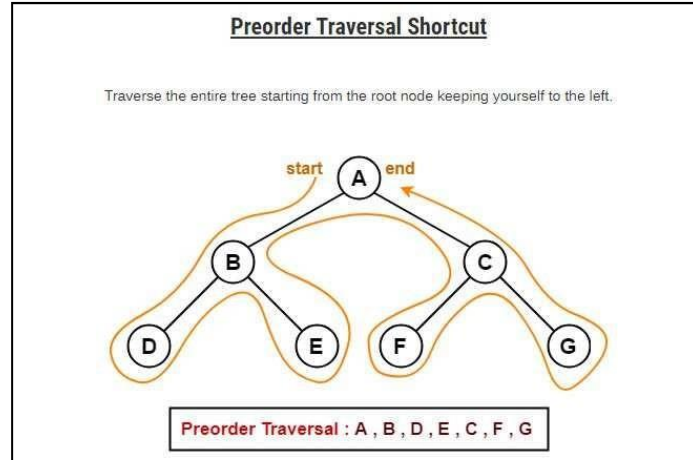
In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

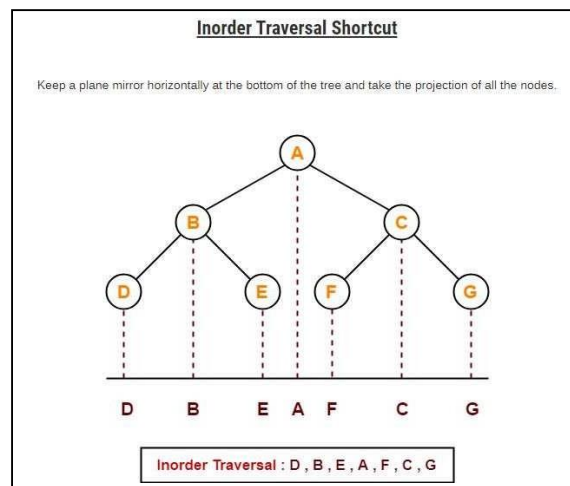
I - J - D - F - B - K - G - H - C - A

IMPORTANT POINTS TO BE NOTED:



- **Application of Pre-Order Traversal:**

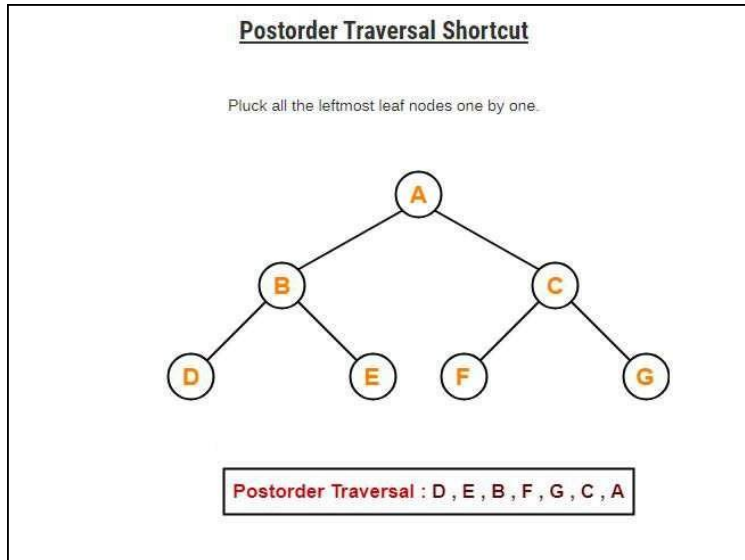
1. used to create a copy of the tree.
2. used to get prefix expression of an expression tree.



- **Application of In-Order Traversal:**

used to get infix expression of an expression tree.

NOTE: In-Order traversal of BST always produces sorted output.



- **Application of Post-Order Traversal:**

1. used to get postfix expression of an expression tree.
2. used to delete the tree. This is because it deletes the children first and then it deletes the parent.

OPERATIONS ON A BINARY SEARCH TREE:

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

Search Operation in BST

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

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6- If search element is larger, then continue the search process in right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
- Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with $O(\log n)$ time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Create a newNode with given value and set its left and right to NULL.
- Step 2 - Check whether tree is Empty.
- Step 3 - If the tree is Empty, then set root to newNode.
- Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
- Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
- Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
- Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Deletion Operation in BST

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

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has only one child then create a link between its parent node and child node.
- Step 3 - Delete the node using free function and terminate the function.

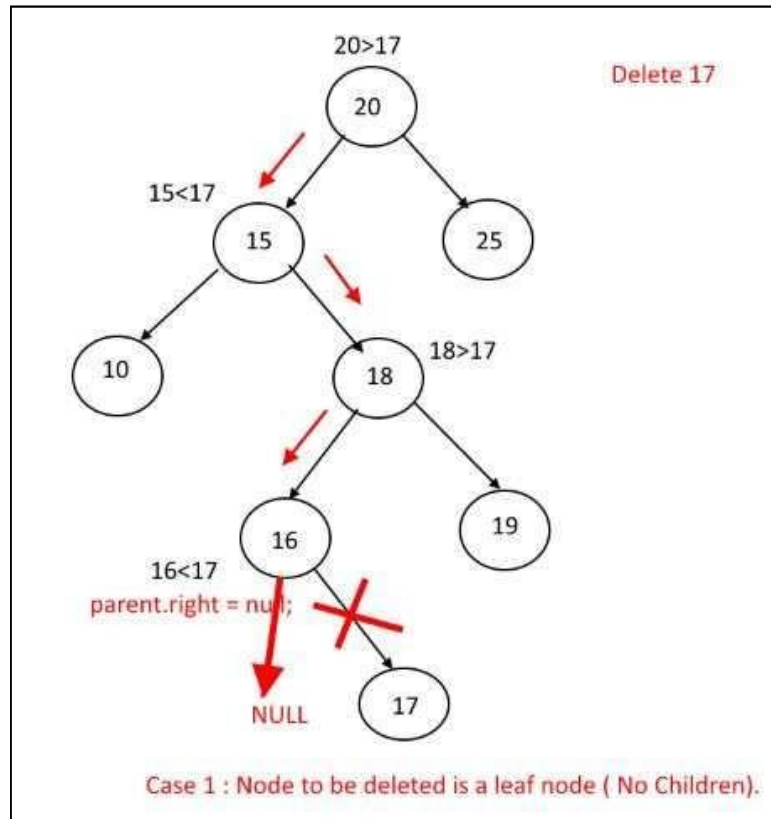
Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

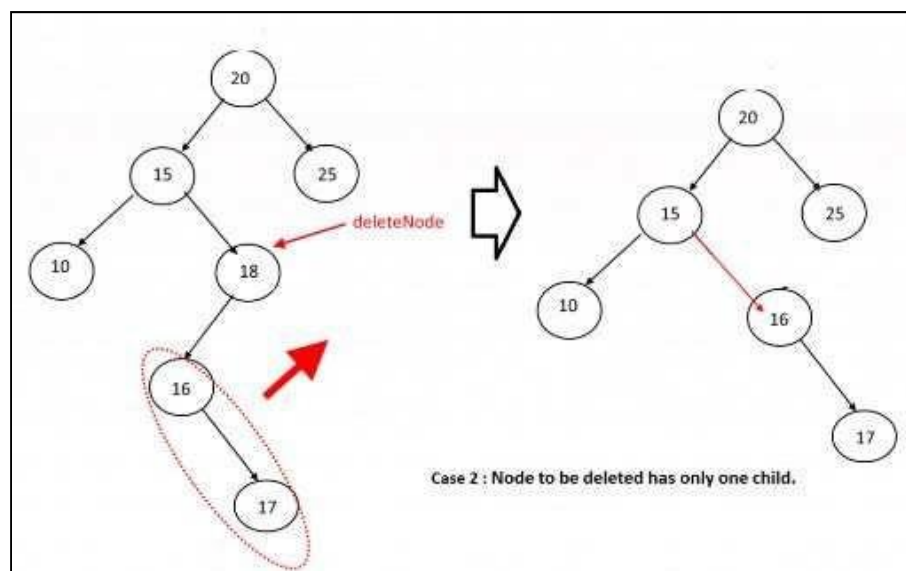
- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
- Step 3 - Swap both deleting node and node which is found in the above step.
- Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
- Step 5 - If it comes to case 1, then delete using case 1 logic.
- Step 6- If it comes to case 2, then delete using case 2 logic.

- Step 7 - Repeat the same process until the node is deleted from the tree.

ILLUSTRATION OF DELETE OPERATION IN BST:



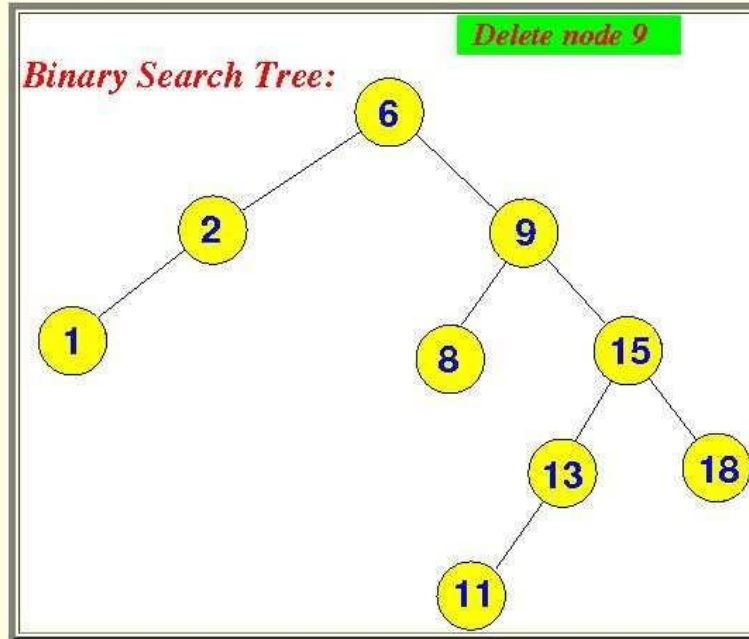
1.



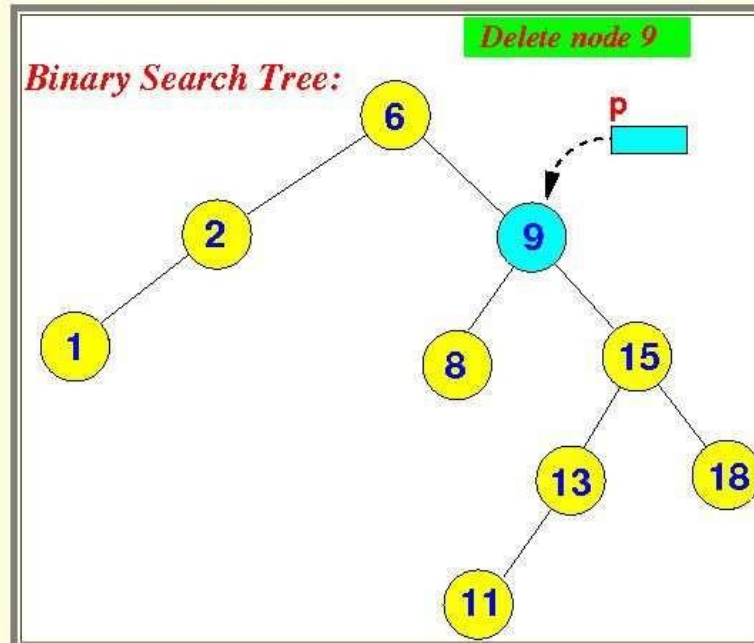
2.

DELETION OF NODE WITH TWO CHILDREN(CASE 3):
EXAMPLE 1:

- Delete **node 9** in the following BST:

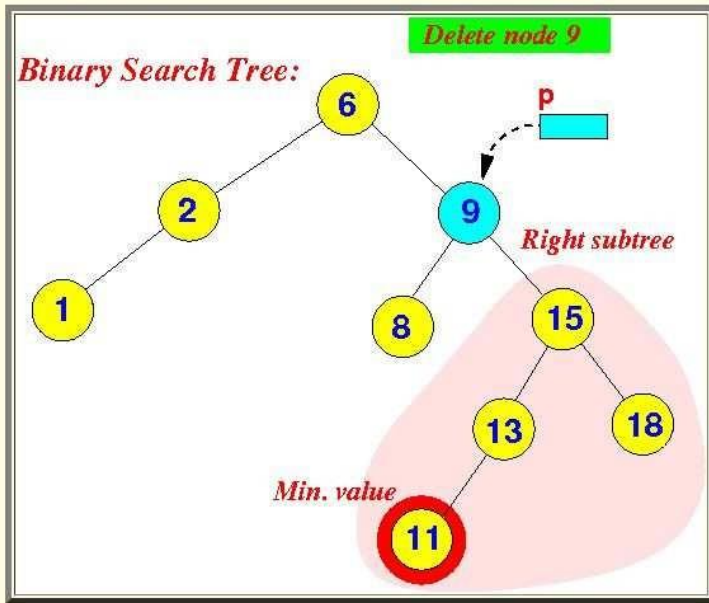


- First, we must **find** the **node** with the value 9:

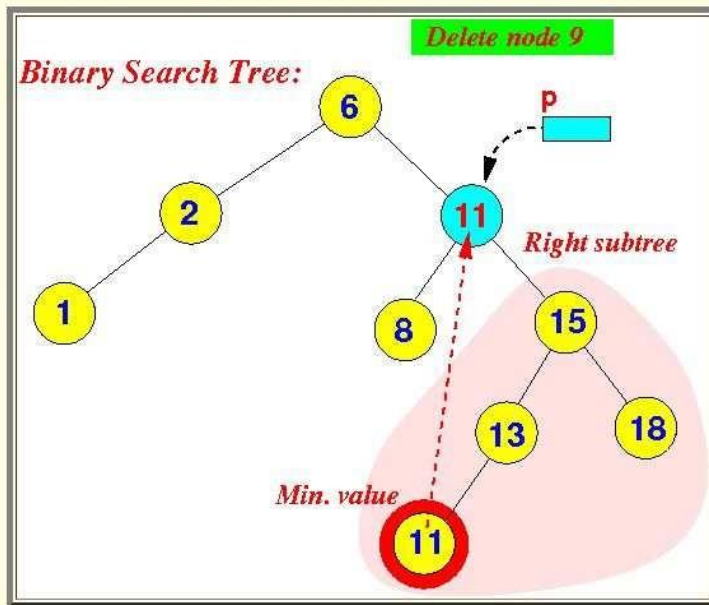


- Next, we find the *successor node* of the node 9.

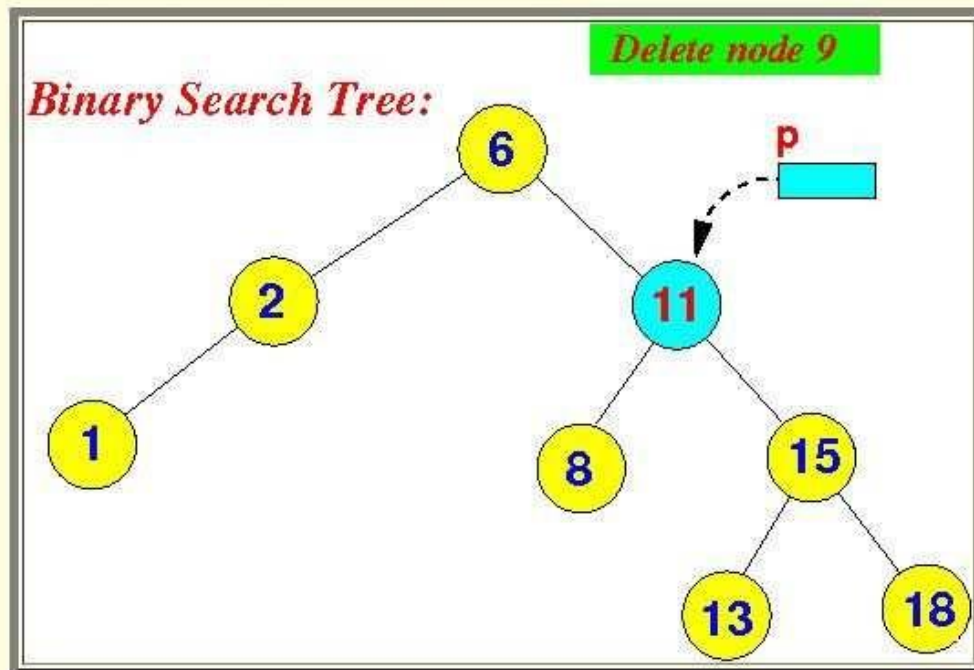
The *successor node* is the node with the *minimum value* in the *right subtree*:



- We copy the content of the *successor node* of the node into the *deletion node* (p):



- The last step is **deleting** the *successor node* from the BST:

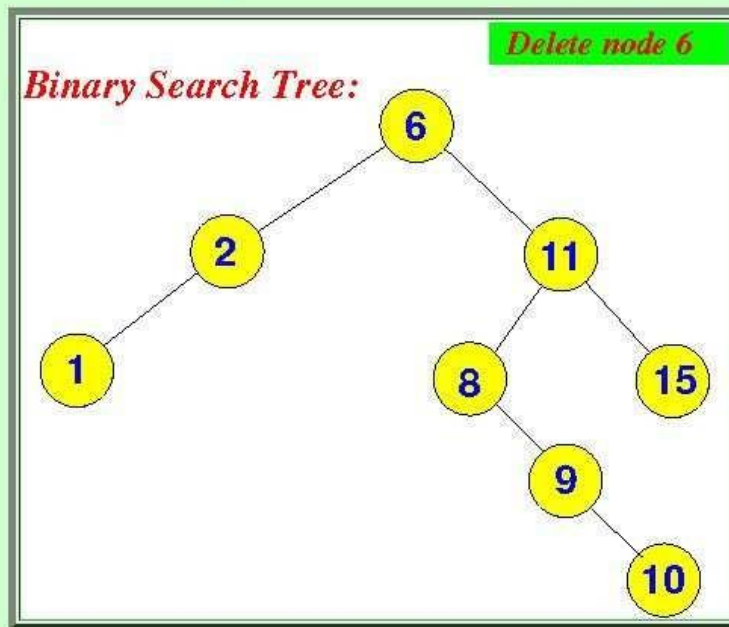


Notice that the tree satisfies the **Binary Search Tree** property:

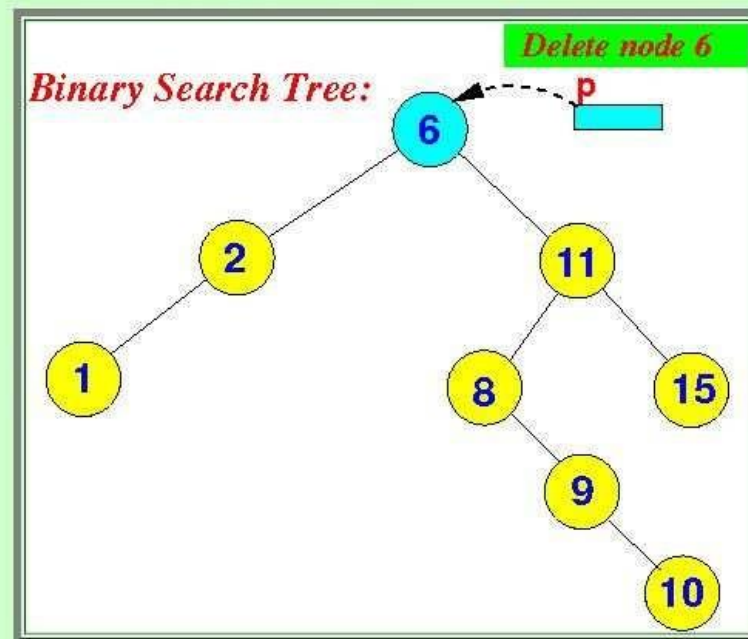
- all nodes in *left subtree* has *smaller values* and
- all nodes in the *right subtree* has *larger values*.

EXAMPLE 2:

- Delete **node 6** in the following BST:



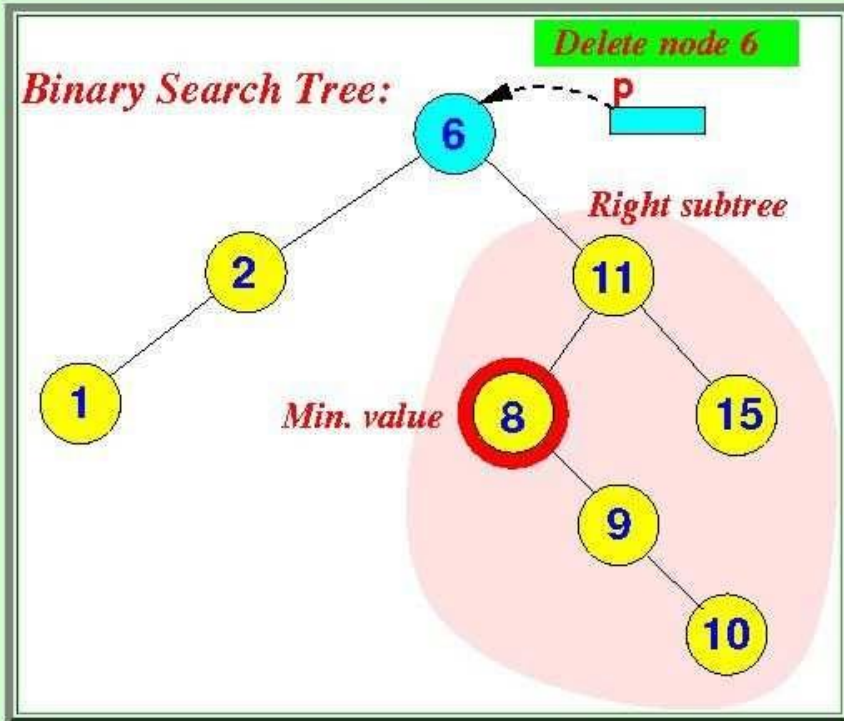
- *First*, we must **find** the **node** with the value 6:



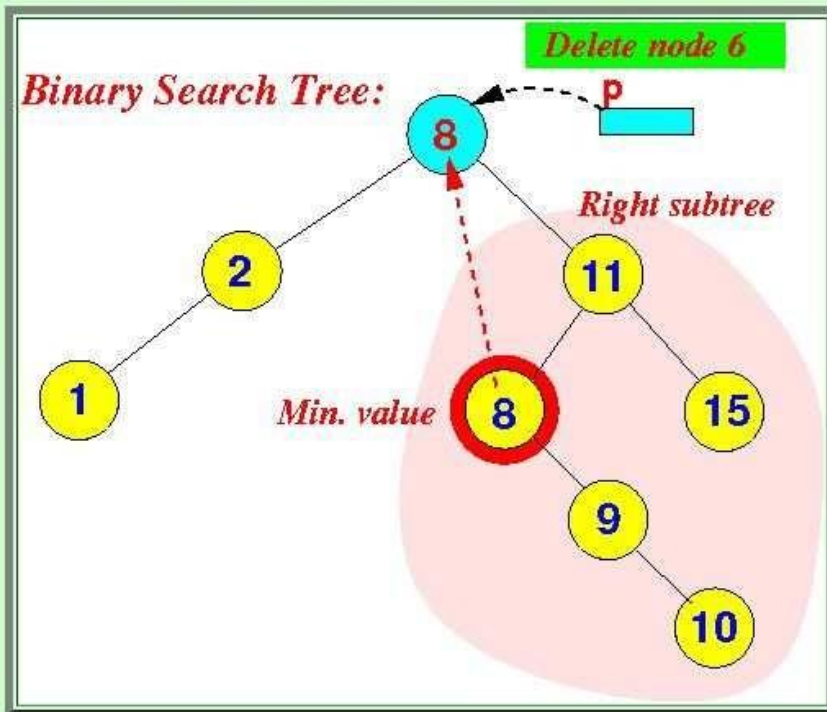
(We use the variable **p** to point to the deletion node)

- Next, we find the *successor node* of the node 6.

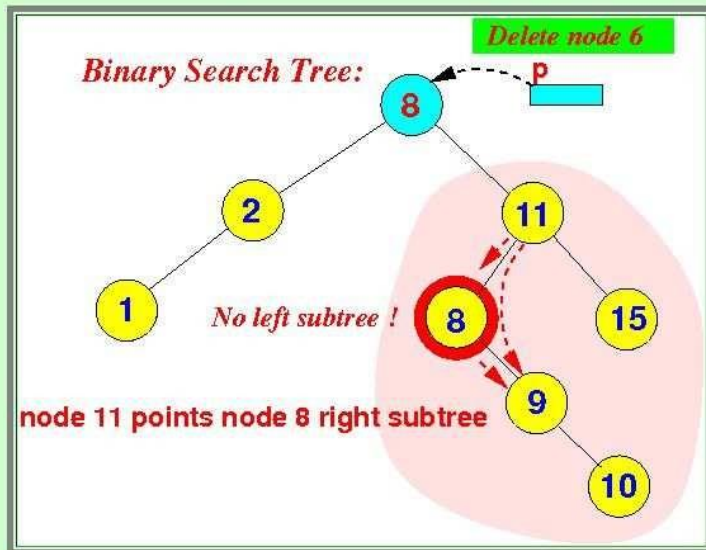
The *successor node* is the node with the *minimum value* in the *right subtree*:



- We copy the content of the *successor node* of the node into the *deletion node* (p):

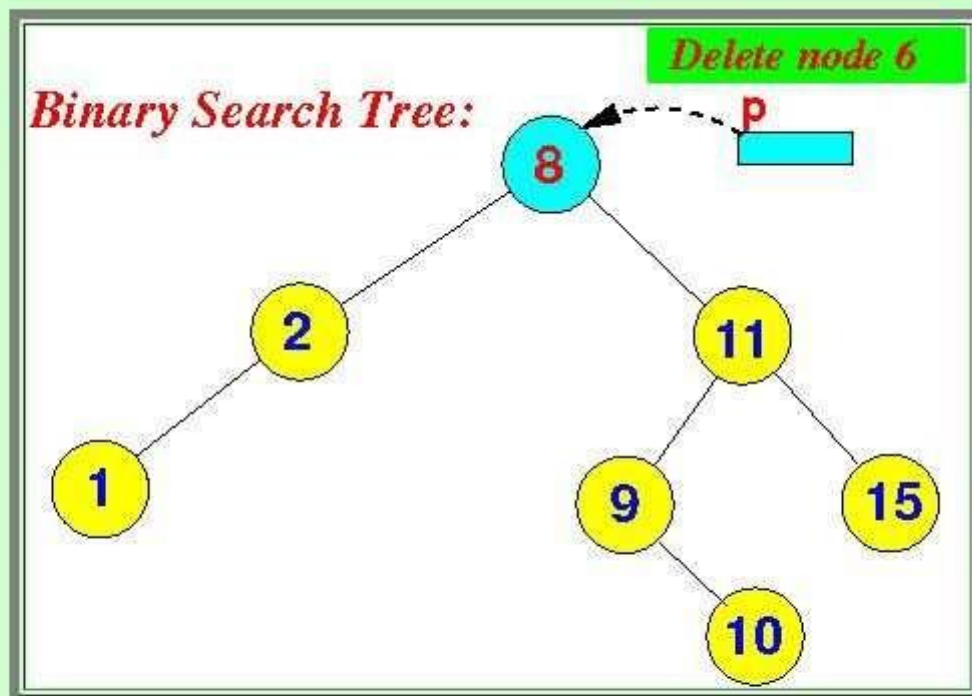


- The last step is deleting the *successor node* from the BST:



Because the *successor node* does not have a *left subtree*, we can delete the *successor node* using the *short-circuit* method (of Case 2)

Result:



Notice that the tree satisfies the **Binary Search Tree** property:

- all nodes in *left subtree* has *smaller values* and
- all nodes in the *right subtree* has *larger values*.

```

//BINARY SEARCH TREE ADT
#include<stdio.h>
#include<stdlib.h>

    struct BSTNode *insertTree(struct BSTNode *p,int key);
    void search(struct BSTNode *root,int key);
    void inorder(struct BSTNode *p);
    void preorder(struct BSTNode *p);
    void postorder(struct BSTNode *p);

struct BSTNode
{
int data;
struct BSTNode *left;
struct BSTNode *right;
};

void main()
{

    struct BSTNode *item,*root=NULL;
    int ch;
    int key;
    while(1)
    {
        printf("\n 1.Insert 2.Search 3 .Traversal 4.Exit \n");
        printf("\nEnter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\nenter element to insert");
                scanf("%d",&key);
                root=insertTree(root,key); break;
            case 2:
                printf("\nenter element to search");
                scanf("%d",&key);
                search(root,key);
                break;
            case 3:
                printf("\nPreorder:");preorder(root);
                printf("\ninorder:");inorder(root);
                printf("\npostorder:");postorder(root);
                break;
            case 4:exit(0);
        }
    }
}

```



```

    }
}

struct BSTNode *insertTree(struct BSTNode *p,int key)
{

    if(p==NULL)
    {
        p=(struct BSTNode*)malloc(sizeof(struct BSTNode));
        p->data=key;
        p->left=p->right=NULL;
    }
    else if(key<p->data)
    p->left=insertTree(p->left,key);
    else
    p->right=insertTree(p->right,key);
    return p;
}

```

```

void search(struct BSTNode *root,int key)
{ int t=0;
    struct BSTNode *p=root;
    while(p!=NULL)
    {
        if(key==p->data)
        {
            t=1;
            printf("\n %d is found",key);
            break;
        }
        else if(key<p->data)
        p=p->left;
        else
        p=p->right;
    }
    if(t==0)
    printf("\n %d not found",key);
}

```

```

void inorder(struct BSTNode *p)
{
    if(p!=NULL)
    {

```

```

        inorder(p->left);
        printf("%d\t",p->data);
        inorder(p->right);
    }
}
void preorder(struct BSTNode *p)
{
    if(p!=NULL)
    {
        printf("%d\t",p->data);
        preorder(p->left);
        preorder(p->right);
    }
}
void postorder(struct BSTNode *p)
{
    if(p!=NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d\t",p->data);
    }
}
}

```

AVL TREE(Adelson-Velsky-Landis):

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

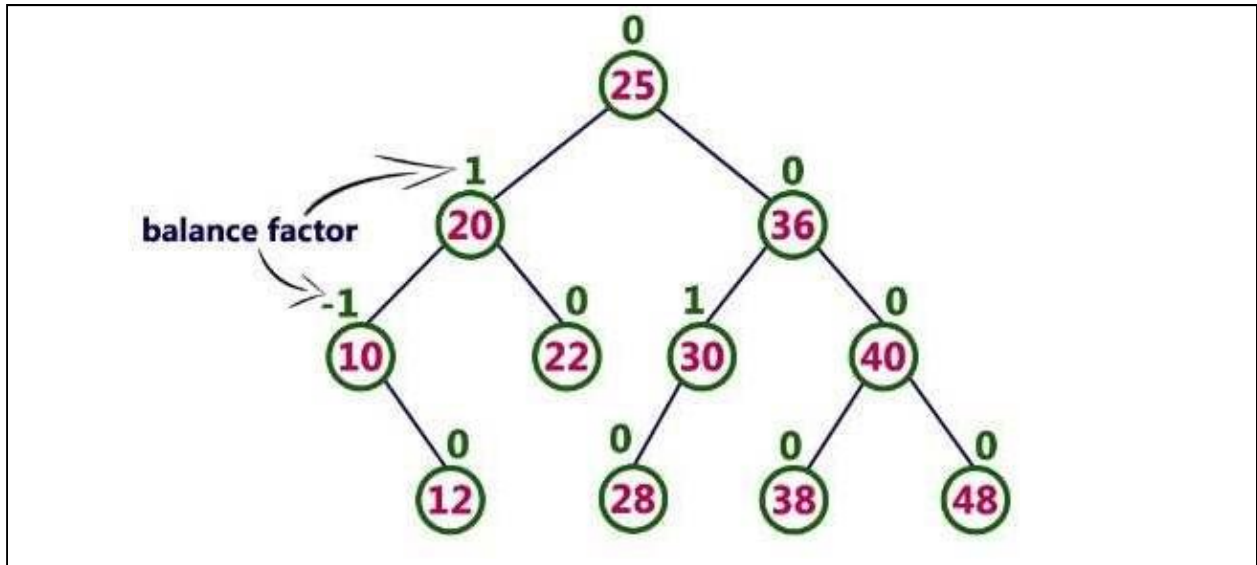
RULE OF AVL TREE:

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Where, **Balance Factor = (Height of left subtree - Height of right subtree)**

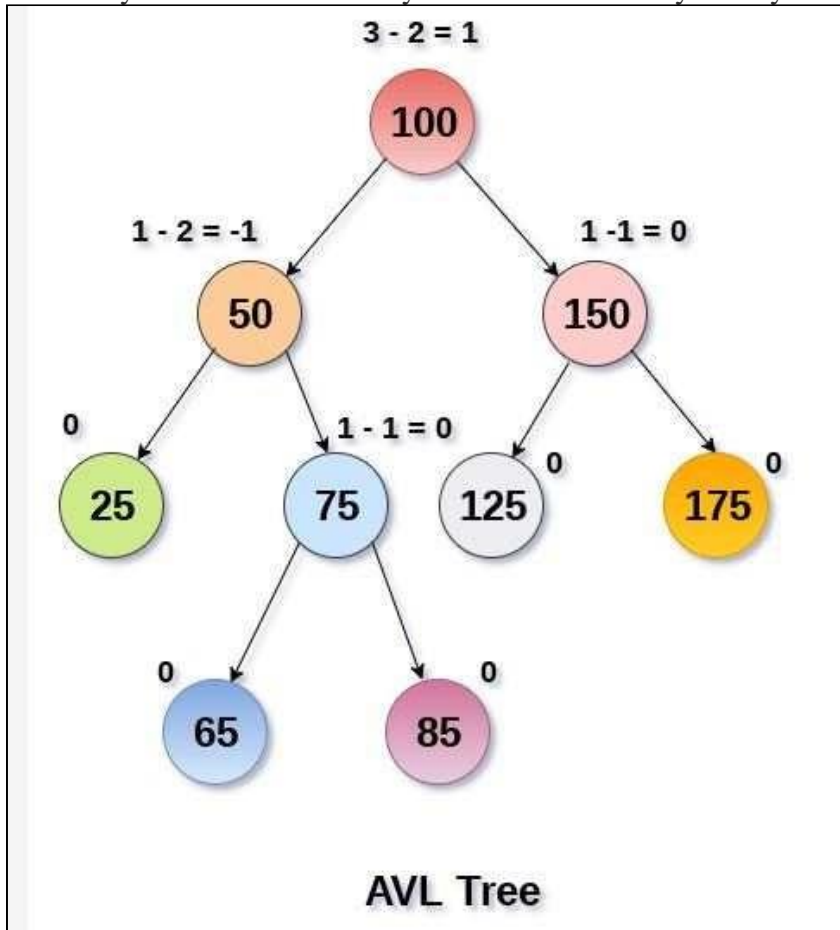
EXAMPLE OF AVL TREE:

The above tree is a binary search tree and every node is satisfying balance factor condition.



So this tree is said to be an AVL tree.

NOTE: Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL



tree.

<https://jamboard.google.com/d/1Zf0TUdTorzaGx-UBXkU0AhZex7T4k-8j27dfmkHWe7I/edit?usp=sharing>

WHY AVL TREES?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree.

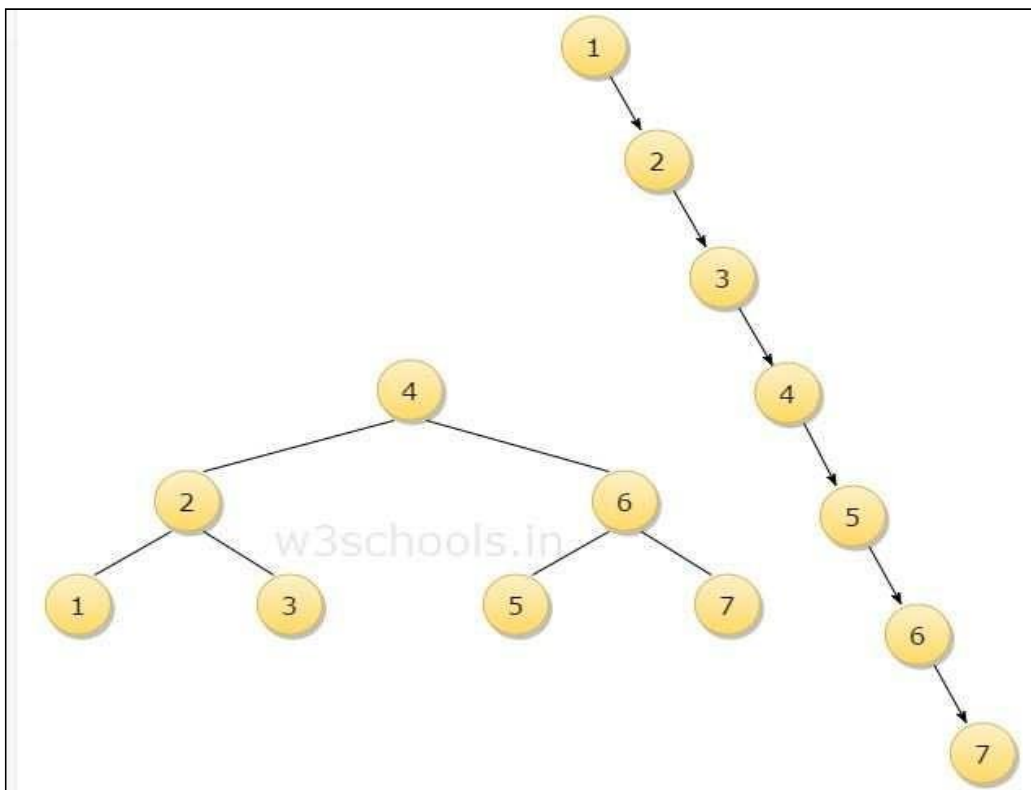
If we make sure that height of the tree remains $O(\text{Log}n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\text{Log}n)$ for all these operations. The height of an AVL tree is always $O(\text{Log}n)$ where n is the number of nodes in the tree

Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity. Let us consider an example:

If you have the following tree having keys 1, 2, 3, 4, 5, 6, 7 and then the binary tree will be like :

To insert a node with a key Q in the binary tree, the algorithm requires seven comparisons, but if you insert the same key in AVL tree, from the above 1st figure, you can see that the algorithm will require three comparisons.

COMPLEXITY

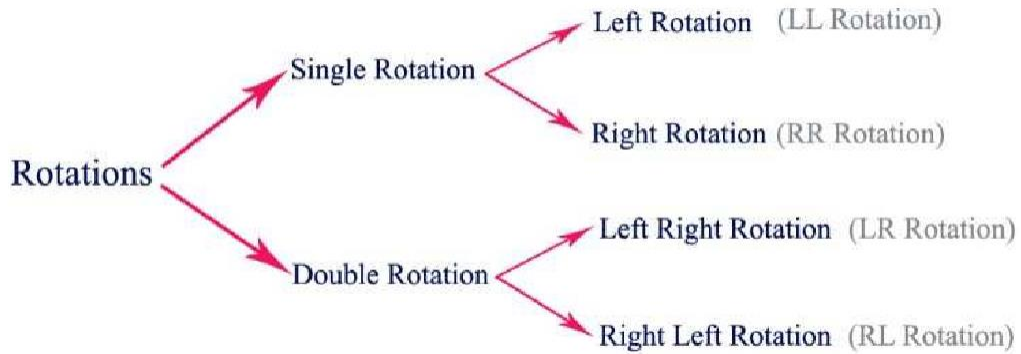


Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

ROTATIONS IN AN AVL TREES:

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

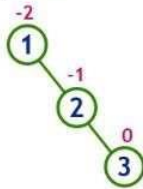
There are **four** rotations and they are classified into **two** types.



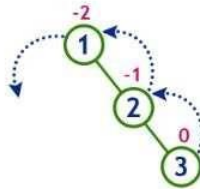
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

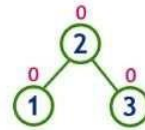
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

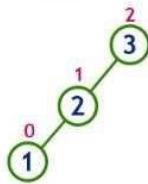


After LL Rotation Tree is Balanced

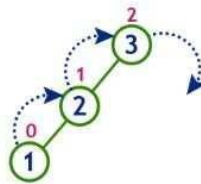
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

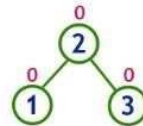
insert 3, 2 and 1



Tree is imbalanced
because node 3 has balance factor 2



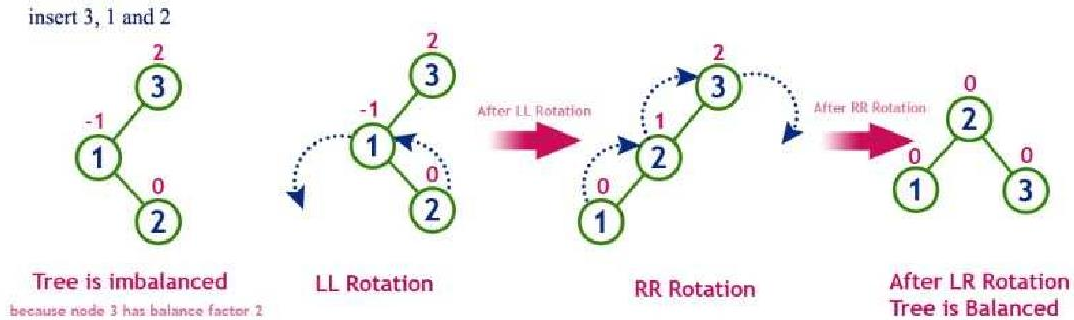
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

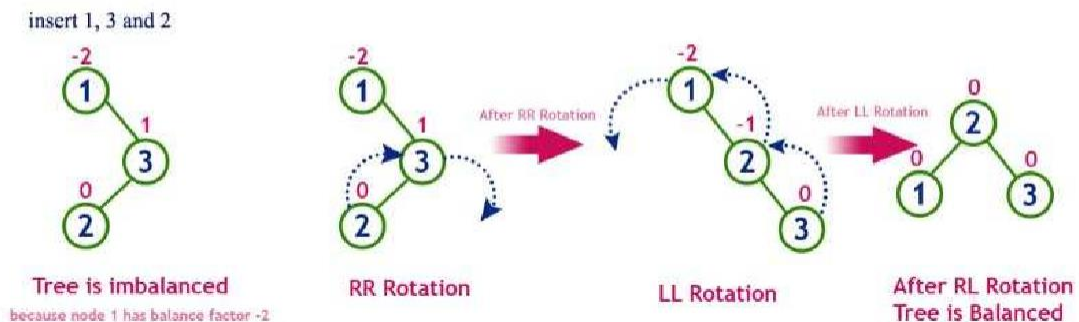
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



OPERATIONS ON AN AVL TREE:

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6 - If search element is larger, then continue the search process in right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with $O(\log n)$ time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the Balance Factor of every node.
- Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
- Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

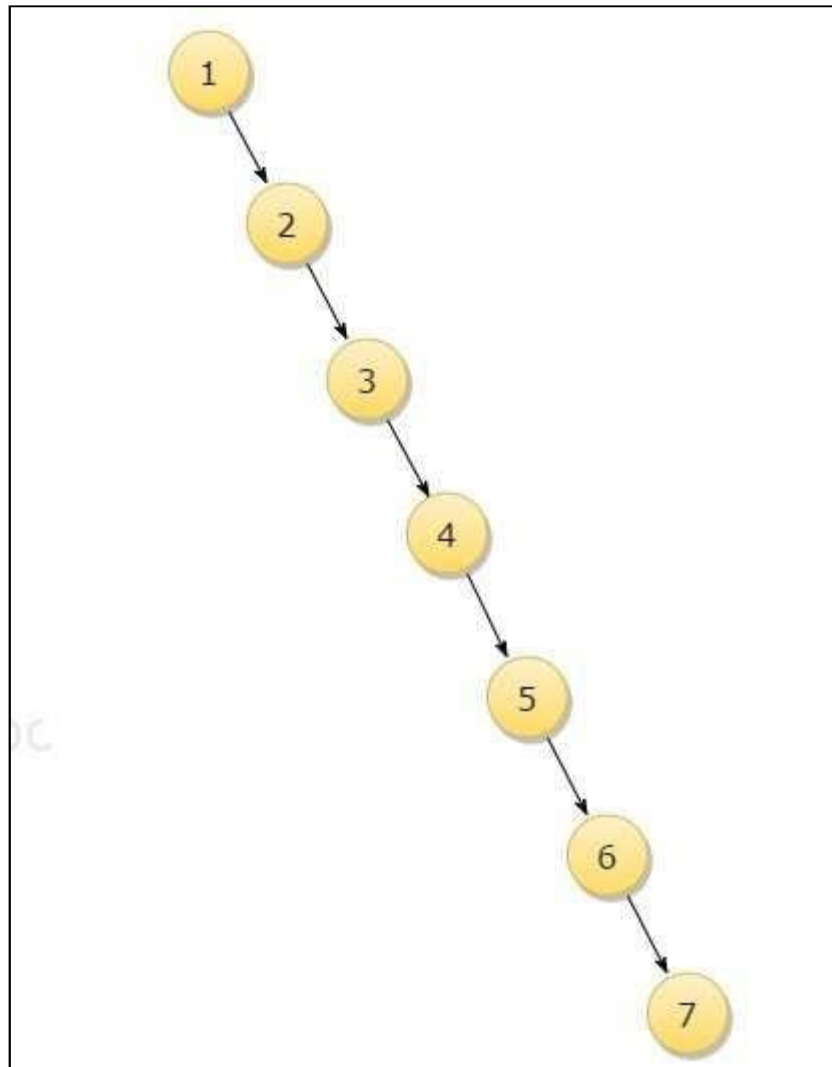
Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance

Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

EXAMPLE PROBLEMS:

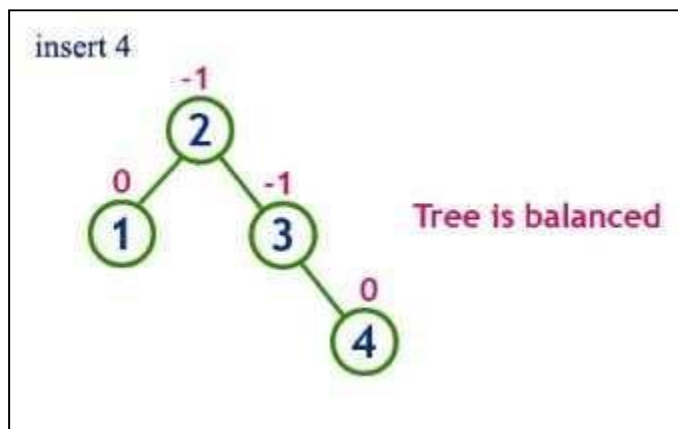
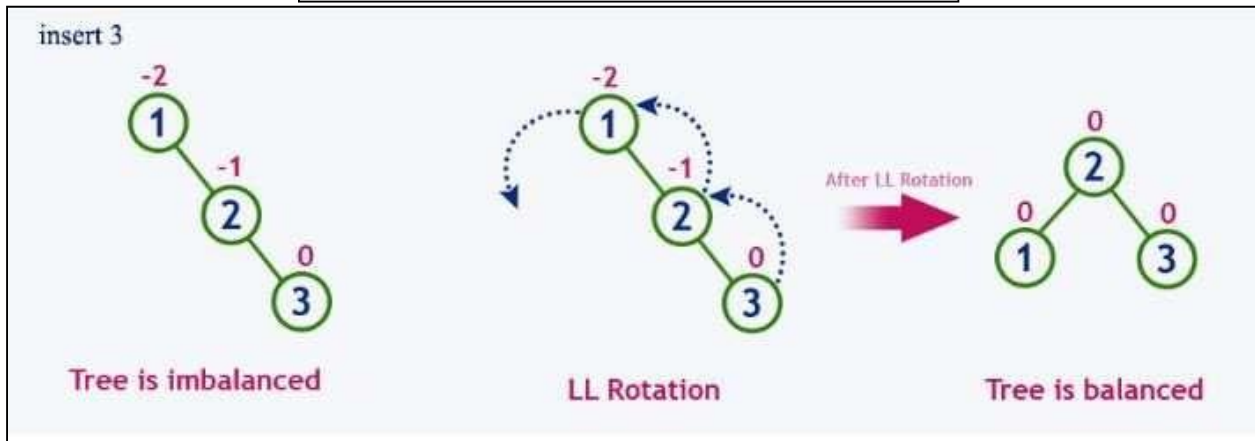
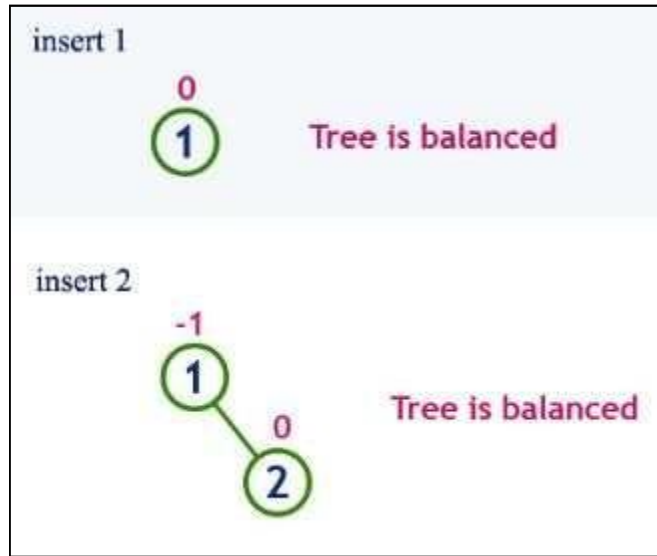
Q1.CONSTRUCT BST - 1,2,3,4,5,6,7.

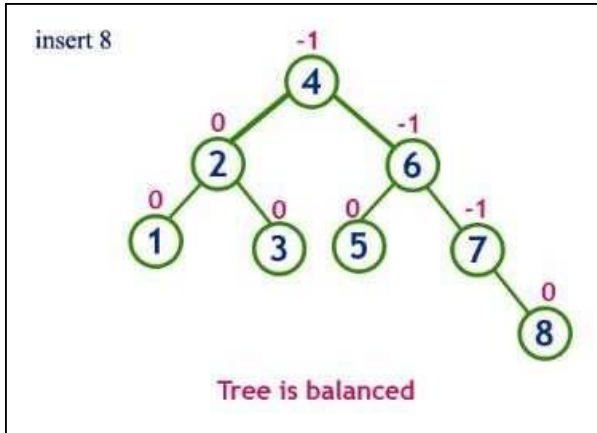
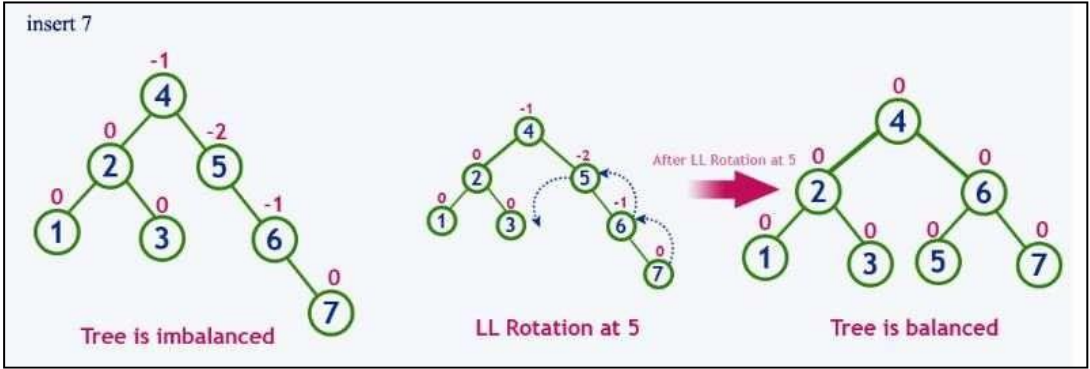
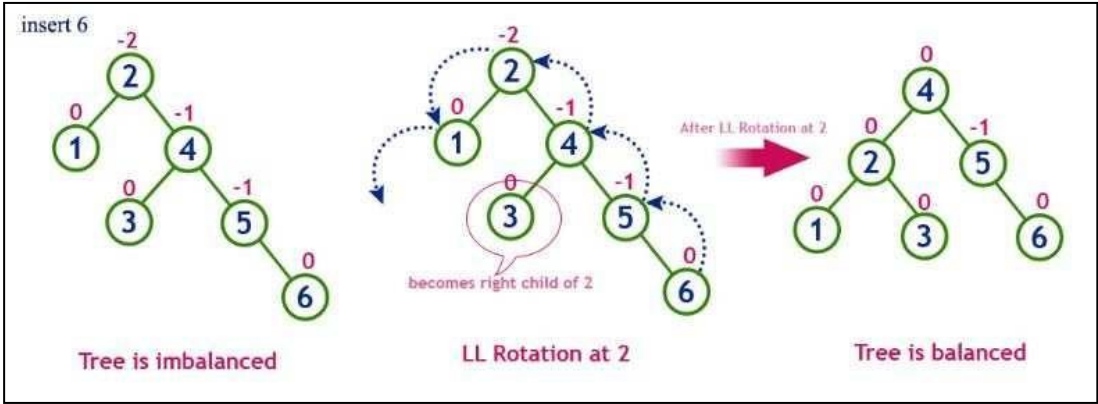
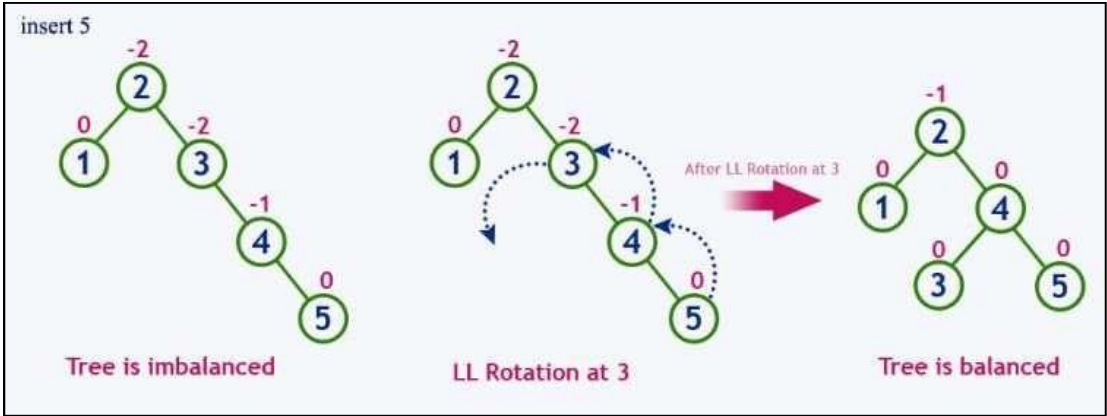


RESULT:

In this case, It becomes a Skew Tree. If we try to search 7, It's gonna take $O(n)$ time. To improve complexity, we'll re-construct this tree(similar) using AVL Tree Properties.

Q2. Construct an AVL Tree by inserting numbers from 1 to 8.



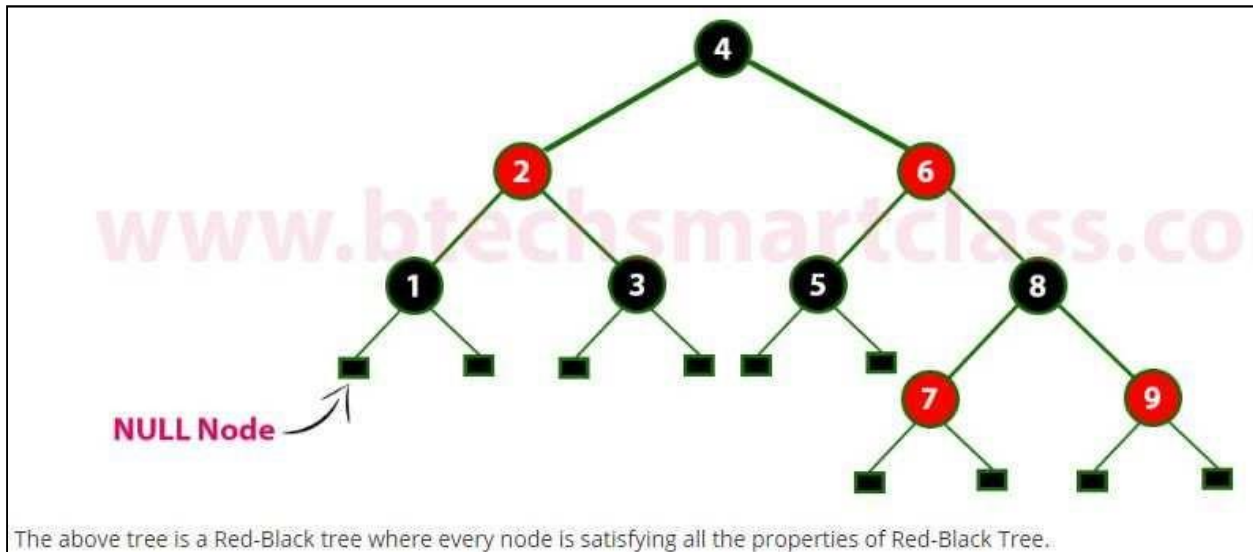


RED-BLACK TREES

Red-Black Tree is a self-balancing Binary Search Tree (BST) in which every node is colored either RED or BLACK.

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

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



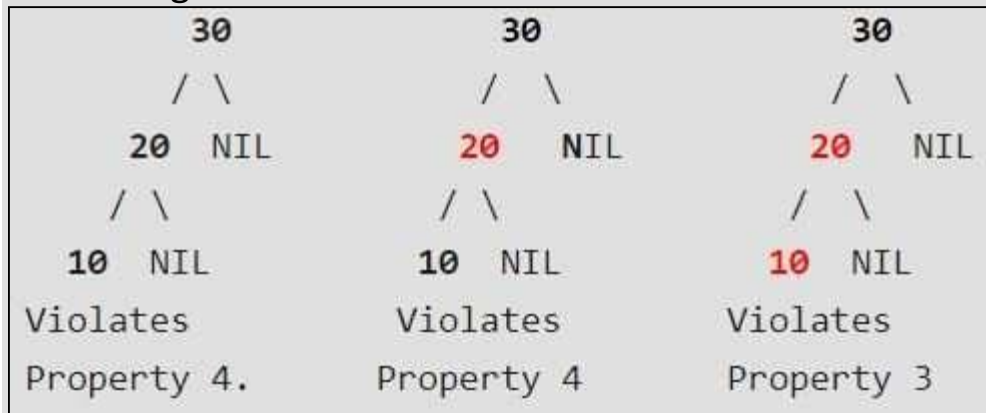
NOTE: Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

How does a Red-Black Tree ensure balance?

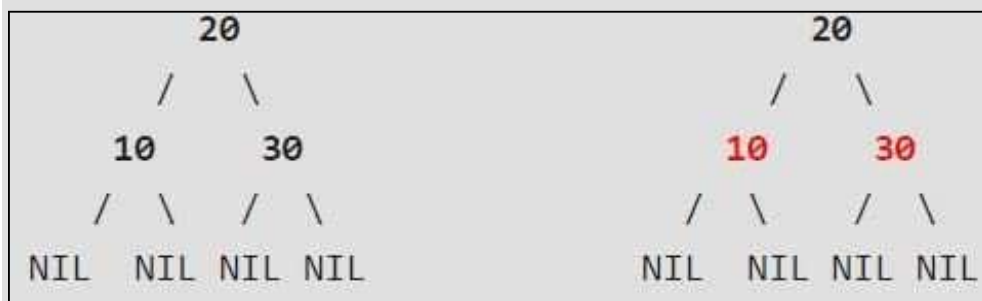
A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red-Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

Comparison with AVL Tree

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red

Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Advantages of Red Black Tree

1. Red black tree are useful when we need insertion and deletion relatively frequent.
 2. Red-black trees are self-balancing so these operations are guaranteed to be $O(\log n)$.
 3. They have relatively low constants in a wide variety of scenarios. **NOTE:** Every Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$
- ### **Applications of Red Black Tree**
1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red Black Tree
 2. It is used to implement CPU Scheduling Linux. **Completely Fair Scheduler** uses it.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

1. **Recolor**
2. **Rotation**
3. **Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

- Step 1 - Check whether tree is Empty.
- Step 2 - If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

- Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.
- Step 4 - If the parent of newNode is Black then exit from the operation.
- Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
- Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.
- Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

EXAMPLE:

Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

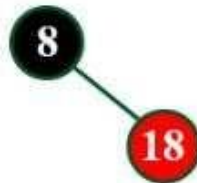
insert (8)

Tree is Empty. So insert newNode as Root node with black color.



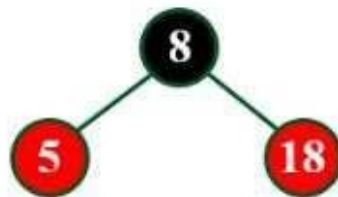
insert (18)

Tree is not Empty. So insert newNode with red color.



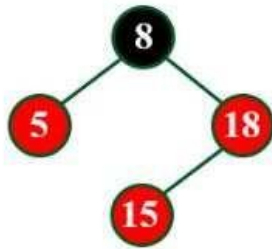
insert (5)

Tree is not Empty. So insert newNode with red color.



insert (15)

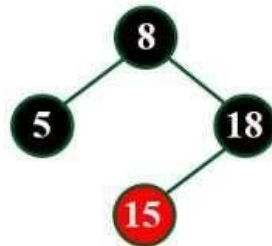
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15). The newnode's parent sibling color is Red and parent's parent is root node. So we use RECOLOR to make it Red Black Tree.



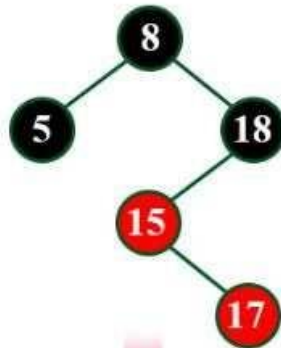
After RECOLOR



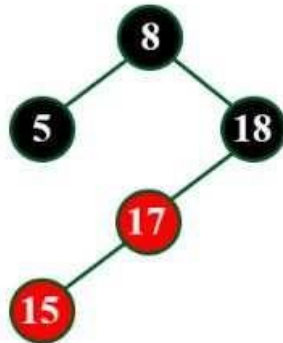
After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (17)

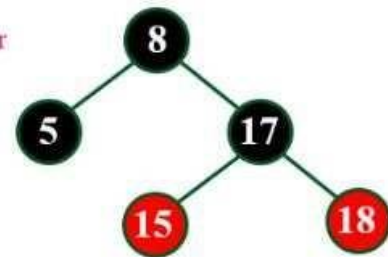
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

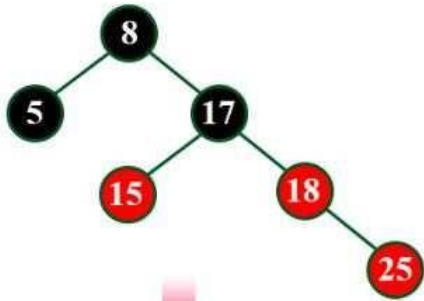


After Right Rotation & Recolor



insert (25)

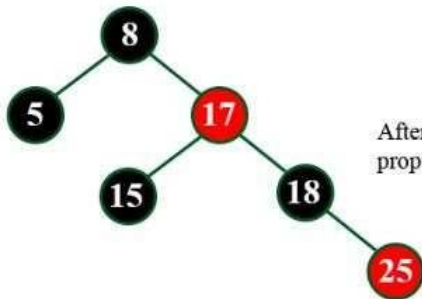
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.



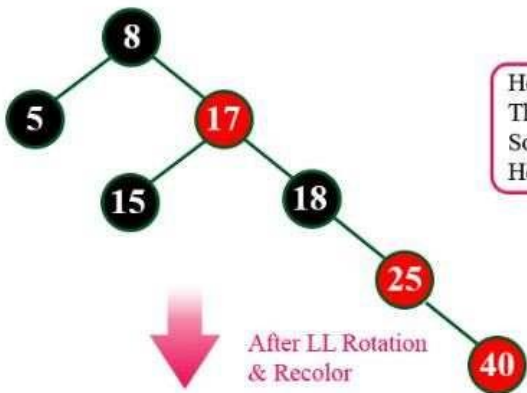
After Recolor



After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

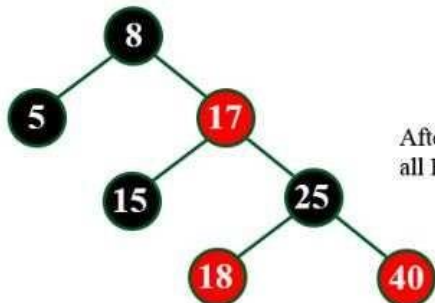
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.



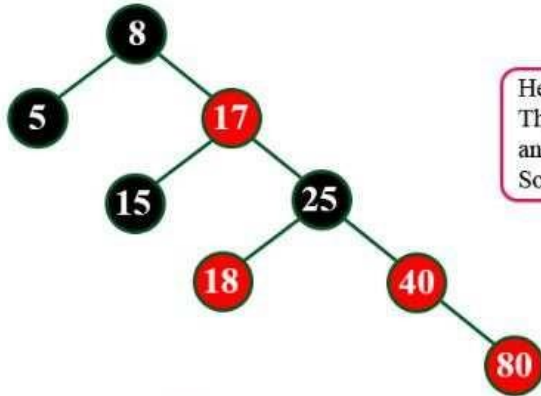
After LL Rotation
& Recolor



After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

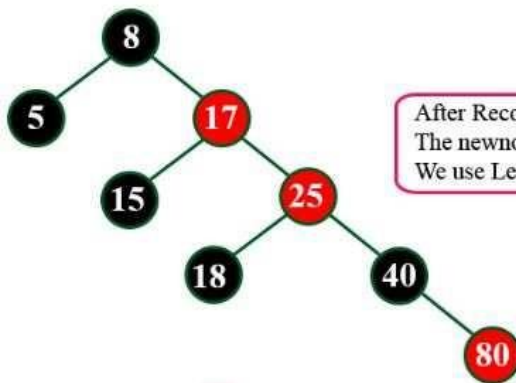
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80). The newnode's parent sibling color is Red and parent's parent is not root node. So we use RECOLOR and Recheck.



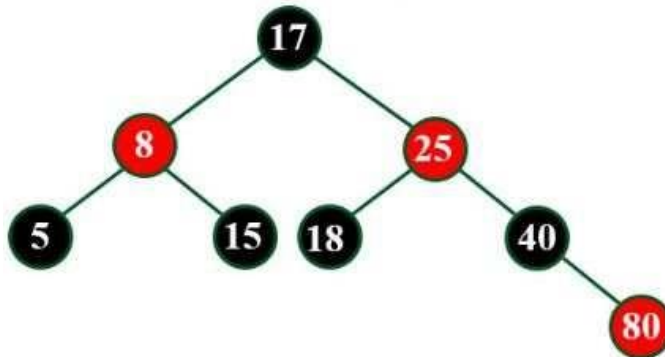
After Recolor



After Recolor again there are two consecutive Red nodes (17 & 25). The newnode's parent sibling color is Black. So we need Rotation. We use Left Rotation & Recolor.



After Left Rotation & Recolor



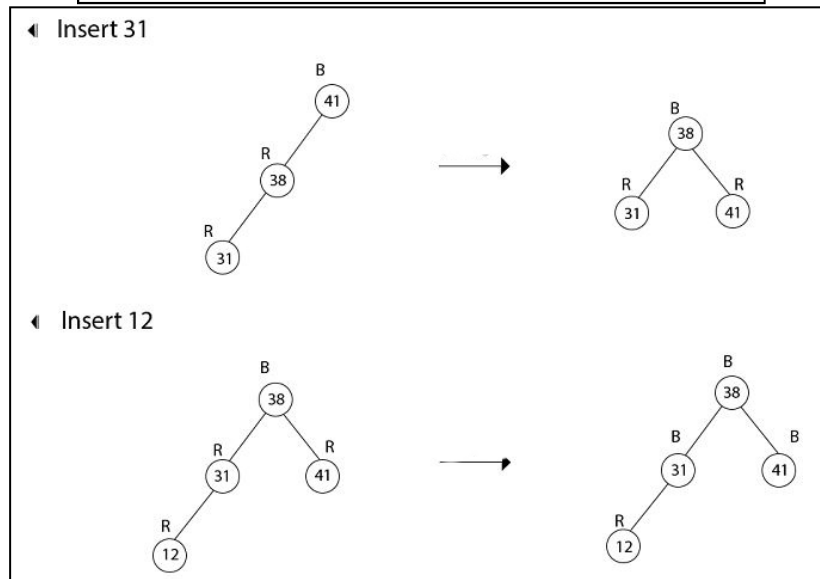
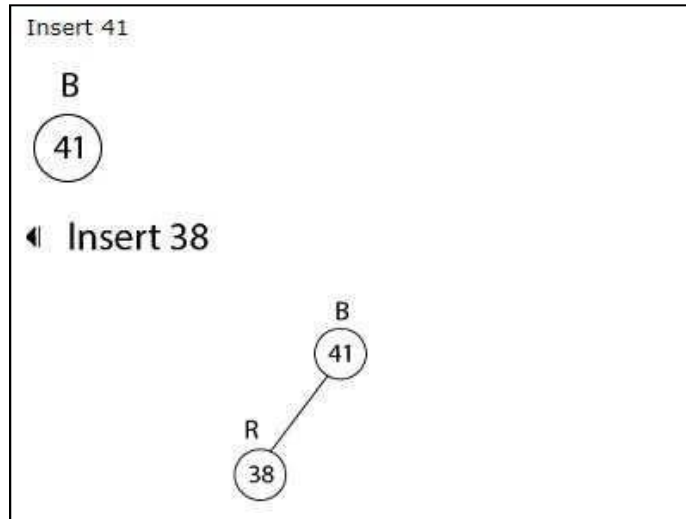
Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

Deletion Operation in Red Black Tree

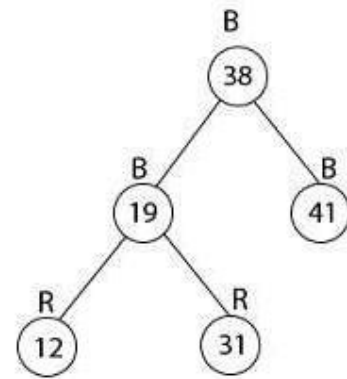
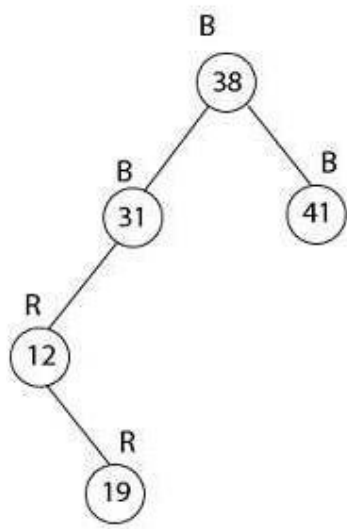
The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red- Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

EXAMPLE:

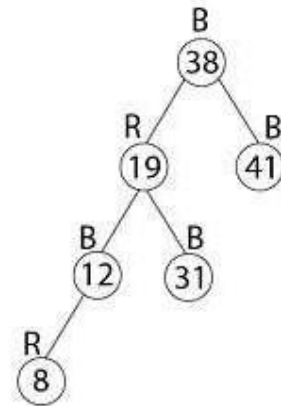
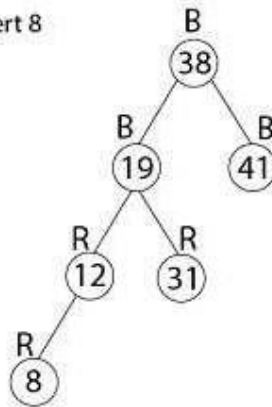
Q. Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.



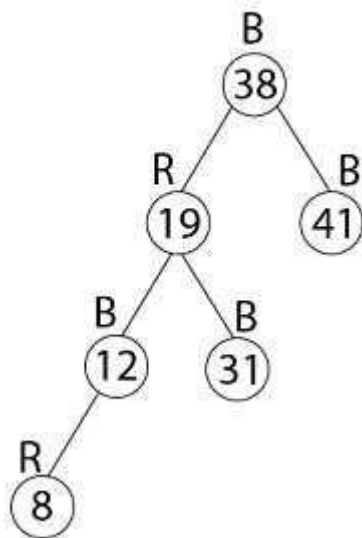
Insert 19



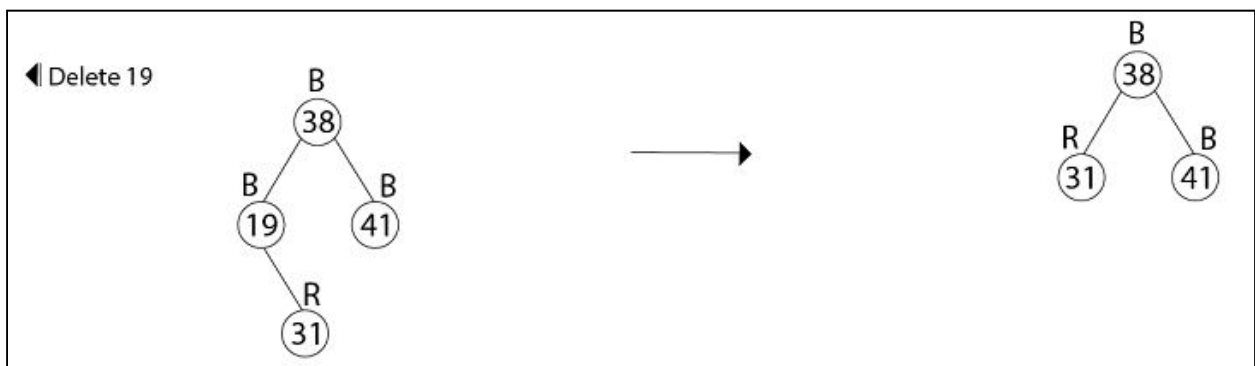
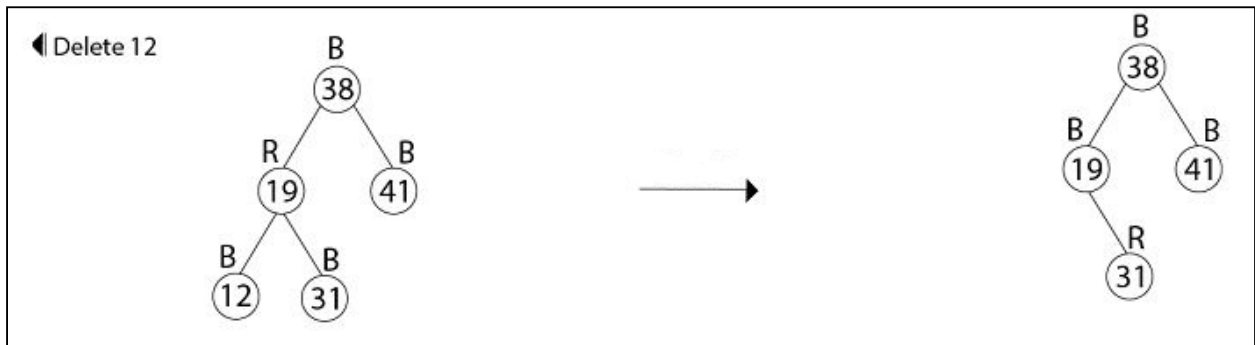
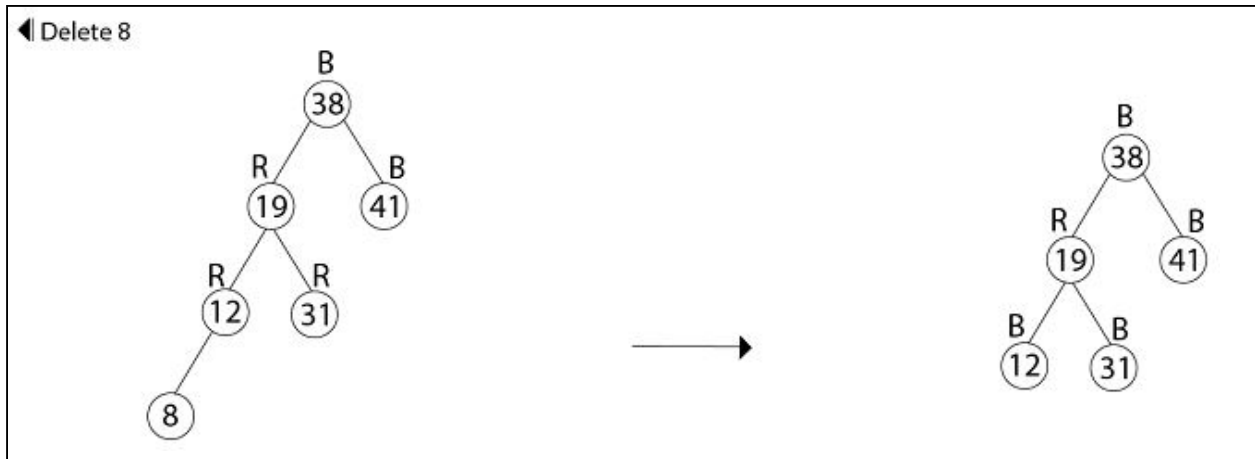
◀ Insert 8

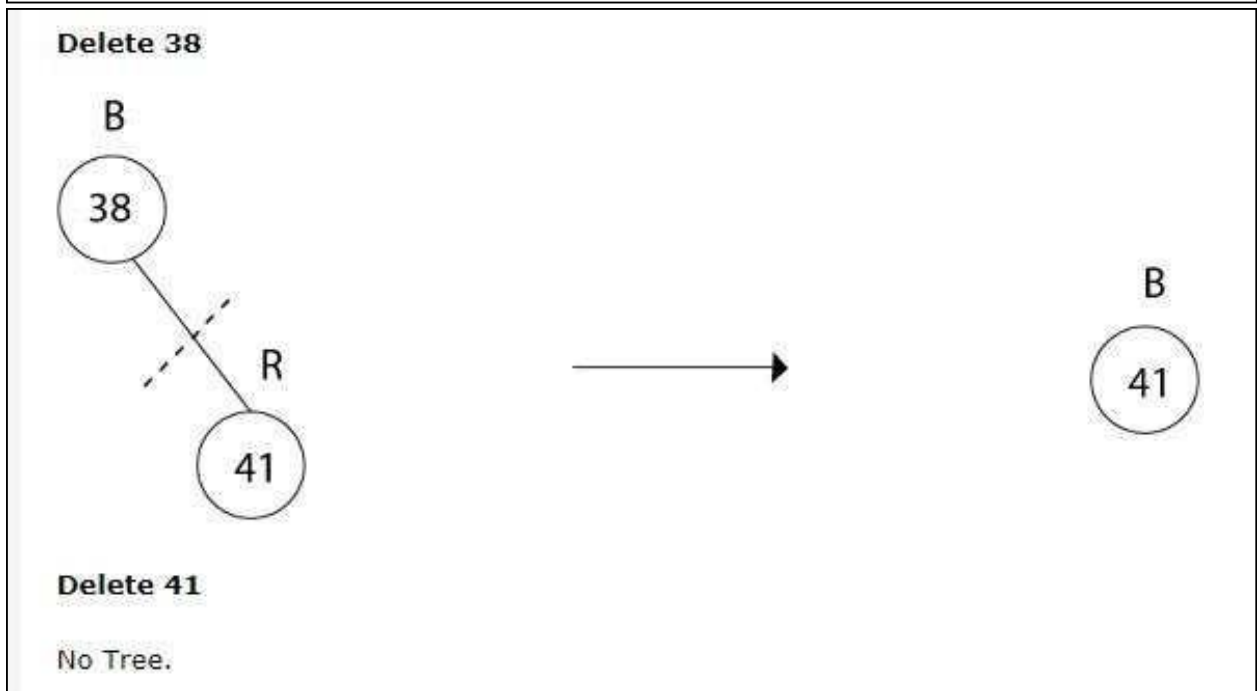
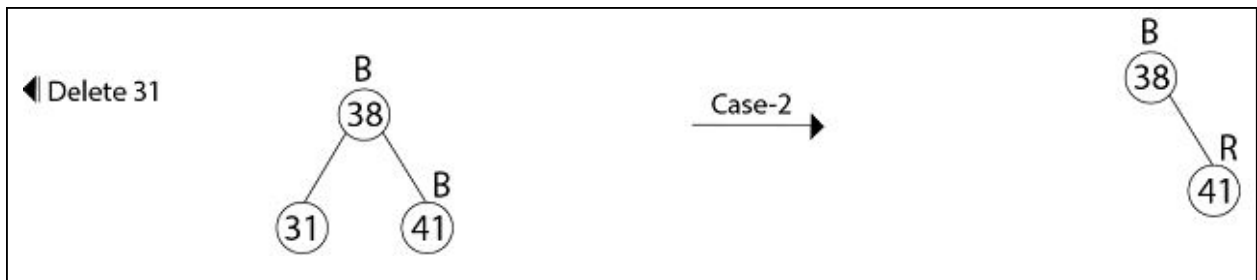


Thus the final tree is



Q. In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.





For more examples of deletion: <https://www.geeksforgeeks.org/red-black-tree-set-3- delete-2/>

SPLAY TREES:

Definition: Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree. The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is $O(n)$. The worst case occurs when the tree is skewed. We can get the worst case time complexity as $O(\log n)$ with AVL and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in $O(1)$ time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in $O(\log n)$ time on average, where n is the number of entries in the tree. Any single operation can take $\Theta(n)$ time in the worst case.

Splaying

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process

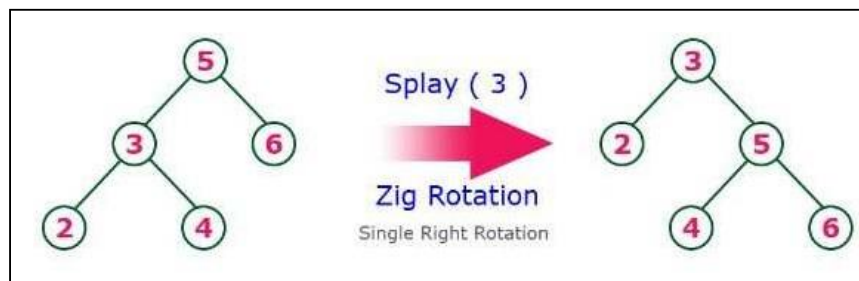
and then splaying that searched element so that it is placed at the root of the tree.
In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

- 1. Zig Rotation
- 2. Zag Rotation
- 3. Zig - Zig Rotation
- 4. Zag - Zag Rotation
- 5. Zig - Zag Rotation
- 6. Zag - Zig Rotation

Zig Rotation

The Zig Rotation in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



Zag Rotation

The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



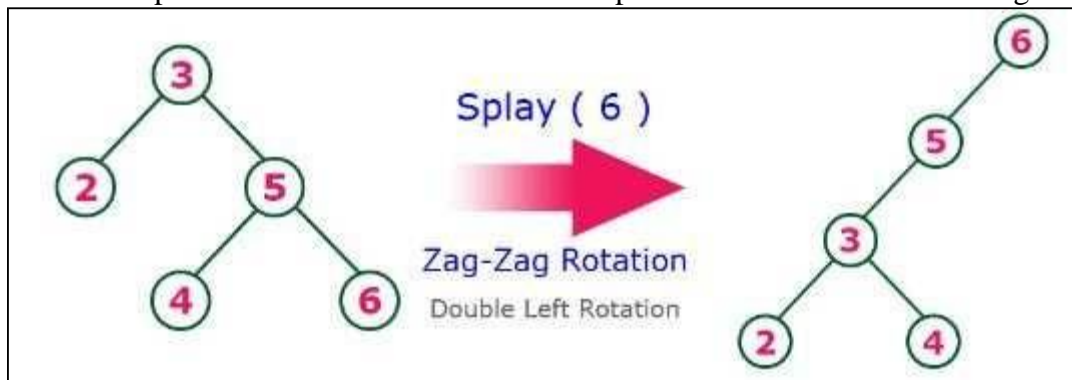
Zig-Zig Rotation

The Zig-Zig Rotation in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



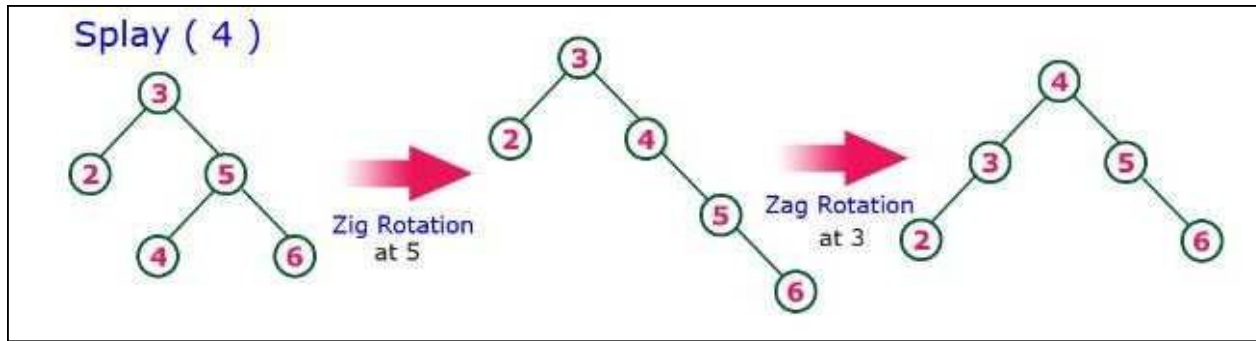
Zag-Zag Rotation

The Zag-Zag Rotation in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



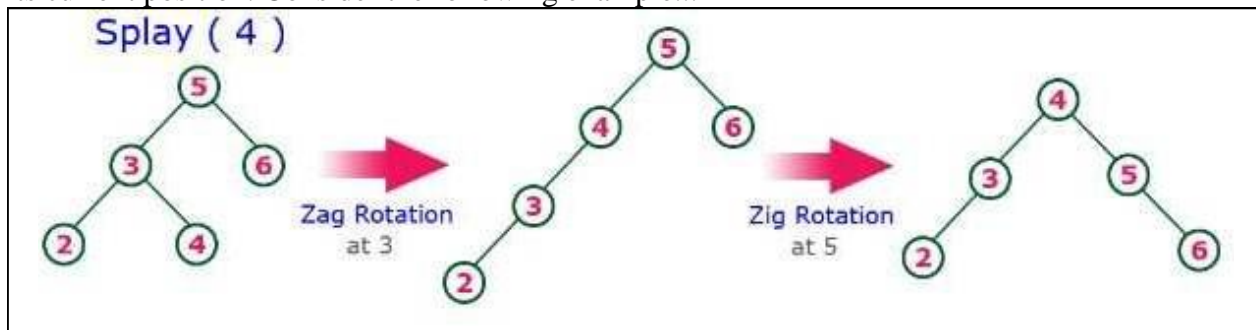
Zig-Zag Rotation

The Zig-Zag Rotation in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



NOTE: Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

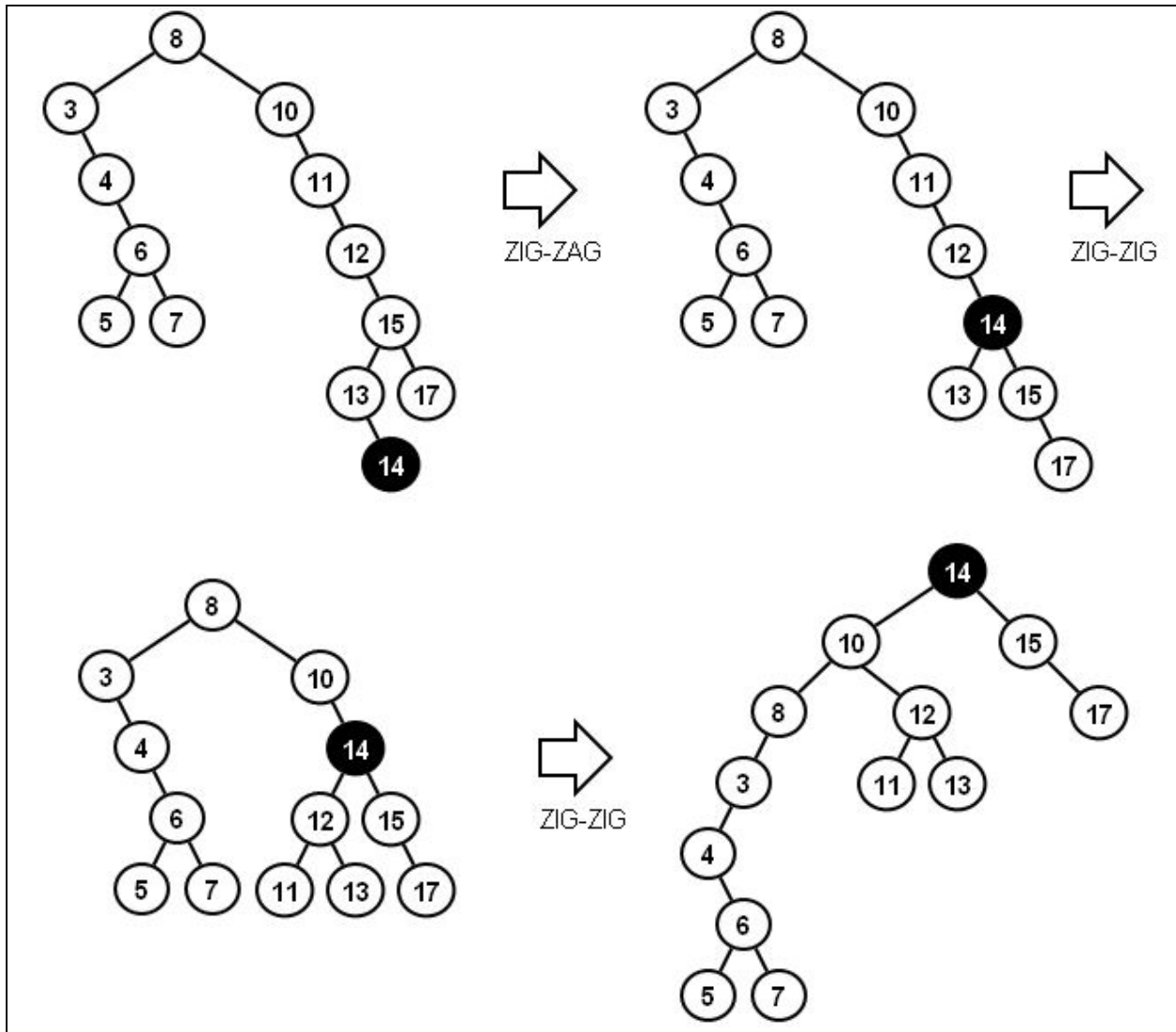
- Step 1 - Check whether tree is Empty.
- Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.
- Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- Step 4 - After insertion, Splay the newNode

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to splay that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

EXAMPLE:

Here's an example We're splaying the 14:



VISUALIZATION LINKS:

BINARY SEARCH TREE: <https://www.cs.usfca.edu/~galles/visualization/BST.html>



AVL TREE: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>



RED BLACK TREE: <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>



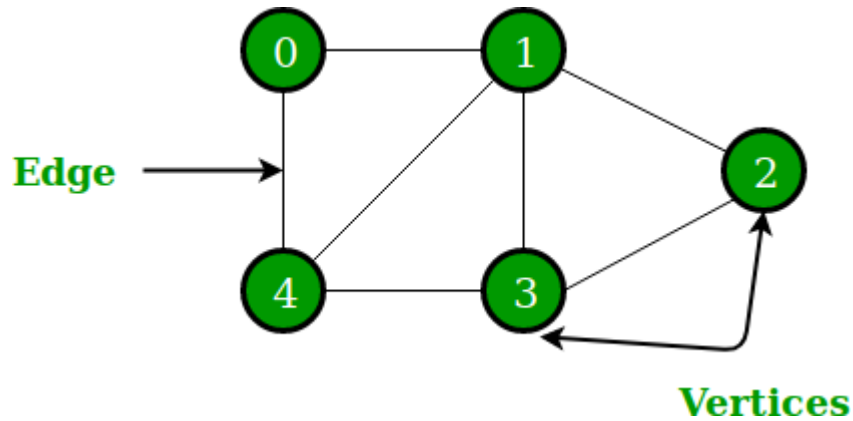
SPLAY TREE: <https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>



Graphs:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



In the above Graph, the set of vertices $V = \{0, 1, 2, 3, 4\}$ and the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$.

Vertex

Each node of the graph is represented as a vertex.

Edge

Edge represents a path between two vertices or a line between two vertices.

Path

Path represents a sequence of edges between the two vertices.

Loop

In a graph, if an edge is drawn from vertex to itself, it is called a loop.



In this example, we have a path from a vertex 'V' to itself. Such can be said as a loop.

Degree of Vertex

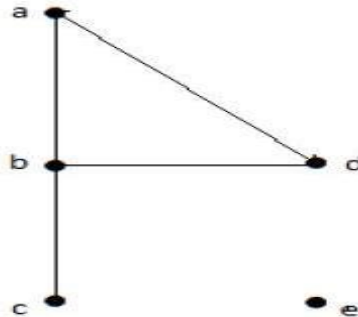
It is the number of vertices adjacent to a vertex V.

Notation – $\text{deg}(V)$

Degree of vertex can be considered under two cases of graphs –

- Undirected Graph
- Directed Graph

Degree of Vertex in an Undirected Graph



In the above Undirected Graph,

- $\text{Deg}(a) = 2$, as there are 2 edges meeting at vertex 'a'.
- $\text{Deg}(b) = 3$, as there are 3 edges meeting at vertex 'b'.
- $\text{Deg}(c) = 1$, as there is 1 edge formed at vertex 'c'
So 'c' is a **pendent vertex**.
- $\text{Deg}(d) = 2$, as there are 2 edges meeting at vertex 'd'.
- $\text{Deg}(e) = 0$, as there are 0 edges formed at vertex 'e'.
So 'e' is an **isolated vertex**.

Degree of Vertex in a Directed Graph

In a directed graph, each vertex has an **in degree** and an **out degree**.

In degree

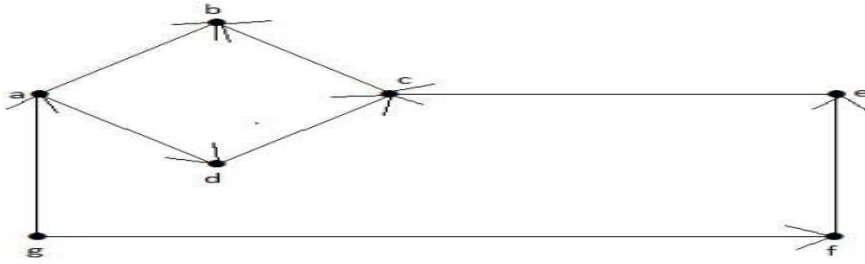
- In degree of vertex V is the number of edges which are coming into the vertex V.
- **Notation** – $\text{deg}^-(V)$.

Out degree

- Out degree of vertex V is the number of edges which are going out from the vertex V.
- **Notation** – $\text{deg}^+(V)$.

Consider the following example:

Take a look at the following directed graph. Vertex 'a' has two edges, 'ad' and 'ab', which are going outwards. Hence its out degree is 2. Similarly, there is an edge 'ga', coming towards vertex 'a'. Hence the in degree of 'a' is 1.



The in degree and out degree of other vertices are shown in the following table –

Vertex	In degree	Out degree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

Pendent Vertex

By using degree of a vertex, we have a two special types of vertices. A vertex with degree one is called a pendent vertex.

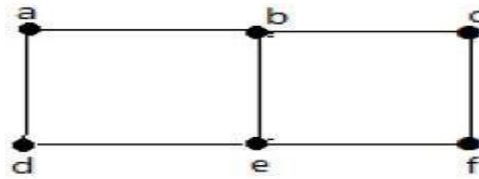
Isolated Vertex

A vertex with degree zero is called an isolated vertex.

Adjacency

- In a graph, two vertices are said to be **adjacent**, if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices.

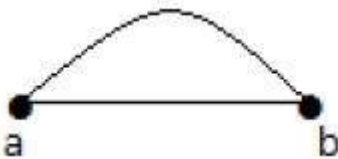
- In a graph, two edges are said to be **adjacent**, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.



In the above graph –

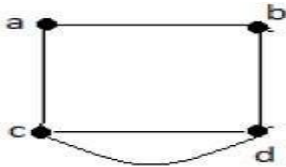
- ‘a’ and ‘b’ are the adjacent vertices, as there is a common edge ‘ab’ between them.
- ‘a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
- ‘ab’ and ‘be’ are the adjacent edges, as there is a common vertex ‘b’ between them.
- ‘be’ and ‘de’ are the adjacent edges, as there is a common vertex ‘e’ between them.

Parallel Edges



In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges.

Multi Graph



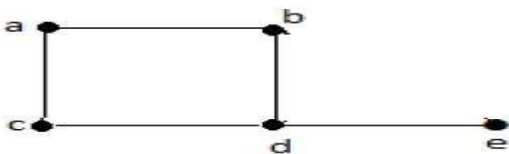
A graph having parallel edges or self-loops or both is known as a Multigraph.

Directed Acyclic Graph:

A graph with no cycles is called a DAG. A graph with no cycles is called a **Forest**.

Degree Sequence of a Graph

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph.



In the above graph, for the vertices {d, a, b, c, e}, the degree sequence is {3, 2, 2, 2, 1}. The below table gives a better understanding. ‘

Vertex	a	b	c	d	e
Connecting to	b,c	a,d	a,d	c,b,e	D
Degree	2	2	2	3	1

Null Graph



A graph **having no edges** is called a Null Graph.

In the above graph, there are three vertices named ‘a’, ‘b’, and ‘c’, but there are no edges among them. Hence it is a Null Graph.

Trivial Graph



A graph **with only one vertex** is called a Trivial Graph.

In the above shown graph, there is only one vertex ‘a’ with no other edges. Hence it is a trivial graph.

Simple Graph

A graph **with no loops** and **no parallel edges** is called a simple graph.

- The maximum number of edges possible in a single graph with ‘n’ vertices is nC_2 where ${}^nC_2 = \frac{n(n-1)}{2}$.
- The number of simple graphs possible with ‘n’ vertices = $2^{{}^nC_2} = 2^{\frac{n(n-1)}{2}}$.

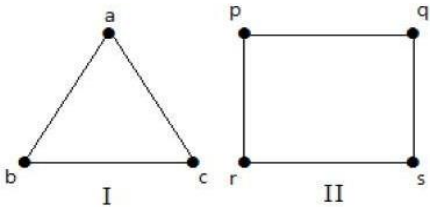
Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

Regular Graph



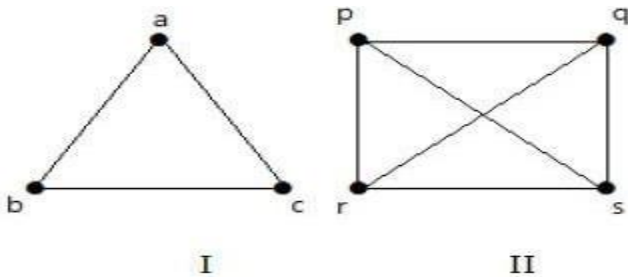
A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is 'k', then the graph is called a 'k-regular graph'.

Complete Graph

A simple graph with 'n' mutual vertices is called a complete graph and it is **denoted by ' K_n '**. In the graph, **a vertex should have edges with all other vertices**, then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



In graph I,

	a	b	c
a	Not Connected	Connected	Connected
b	Connected	Not Connected	Connected
c	Connected	Connected	Not Connected

In graph II,

	p	q	r	s
p	Not Connected	Connected	Connected	Connected
q	Connected	Not Connected	Connected	Connected
r	Connected	Connected	Not Connected	Connected
s	Connected	Connected	Connected	Not Connected

Cycle Graph

A simple graph with 'n' vertices ($n \geq 3$) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.

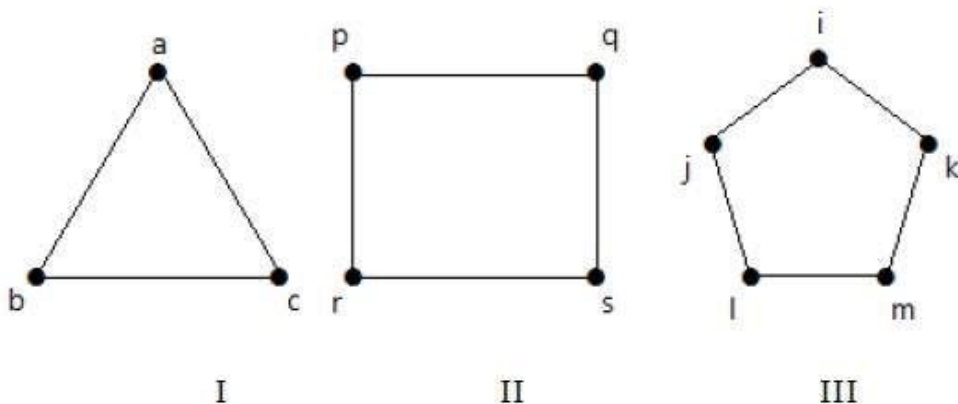
If the **degree of each vertex in the graph is two**, then it is called a Cycle Graph.

Notation – C_n

Example

Take a look at the following graphs –

- Graph I has 3 vertices with 3 edges which is forming a cycle 'ab-bc-ca'.
- Graph II has 4 vertices with 4 edges which is forming a cycle 'pq-qs-sr-rp'.
- Graph III has 5 vertices with 5 edges which is forming a cycle 'ik-km-ml-lj-ji'.



Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

Representation of Graphs

There are mainly two ways to represent a graph –

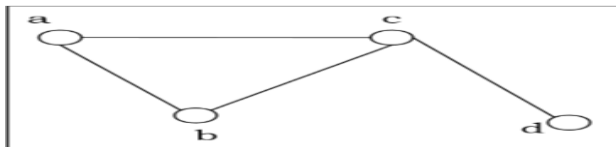
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

An Adjacency Matrix $A [V] [V]$ is a 2D array of size $V \times V$ where V is the number of vertices in an undirected graph. If there is an edge between V_x to V_y then the value of $A [V_x] [V_y] = 1$ and $A [V_y] [V_x] = 1$ (this is because it is an undirected graph and if it is possible to traverse from V_x to V_y , the reverse also is possible), otherwise the value will be zero. And for a directed graph, if there is an edge between V_x to V_y , then the value of $A [V_x] [V_y] = 1$, otherwise the value will be zero.

Adjacency Matrix of an Undirected Graph

Let us consider the following undirected graph and construct the adjacency matrix –

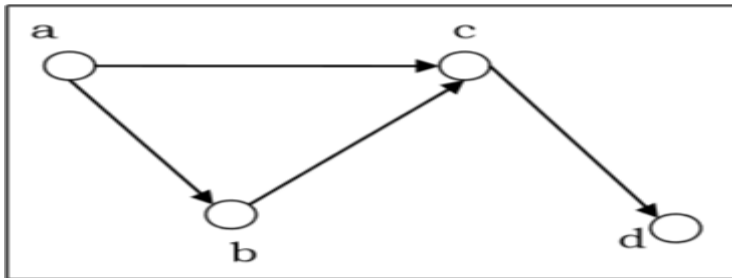


Adjacency matrix of the above undirected graph will be

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency Matrix of a Directed Graph

Let us consider the following directed graph and construct its adjacency matrix –



Adjacency matrix of the above directed graph will be –

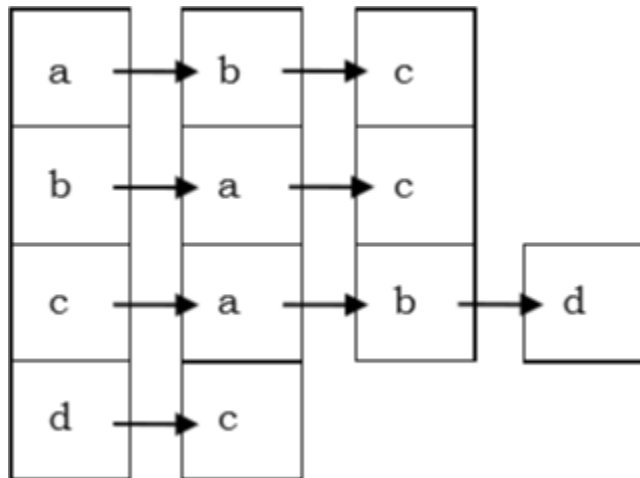
	a	b	c	d
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List

In adjacency list, an array ($A[V]$) of linked lists is used to represent the graph G with V number of vertices. An entry $A[V_x]$ represents the linked list of vertices adjacent to the 'Vx' vertex. The adjacency list of the undirected graph is as shown in the figure below –



Simply considering the same figure as the above undirected graph, an array of list is considered. Now as we can see the Vertex 'a' is connected to vertices 'b' & 'c'. Similarly all the other vertices are connected and they are represented as an array of lists as represented above.

Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows..

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

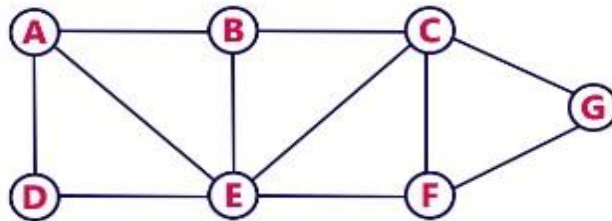
Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

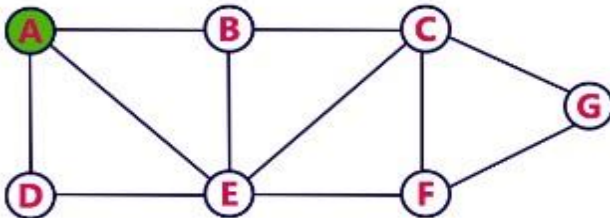
Example:

Consider the following example graph to perform DFS traversal



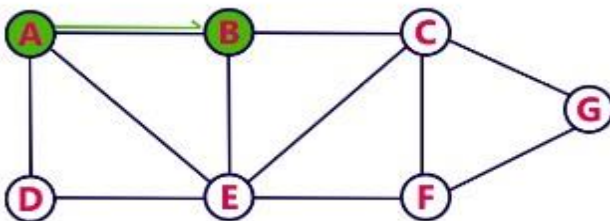
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



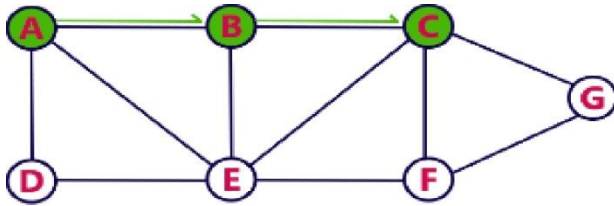
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



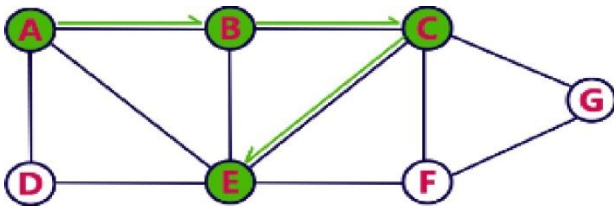
Step 3:

- Visit any adjacent vertex of B which is not visited (C).
- Push C on to the Stack



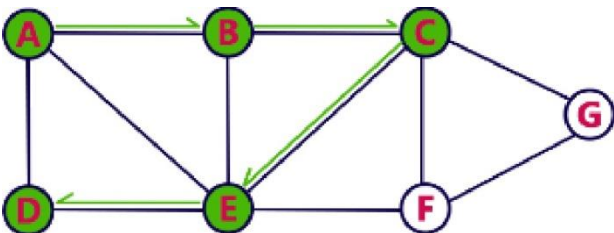
Step 4:

- Visit any adjacent vertex of C which is not visited (E).
- Push E on to the Stack



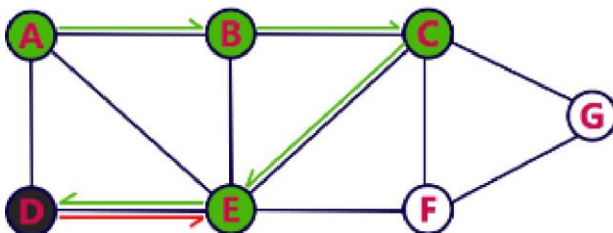
Step 5:

- Visit any adjacent vertex of E which is not visited (D).
- Push D on to the Stack



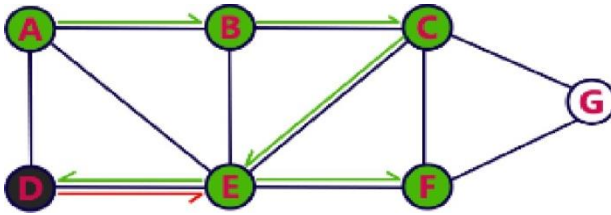
Step 6:

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

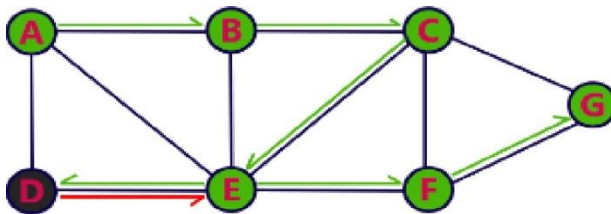
- Visit any adjacent vertex of E which is not visited (F).
- Push F on to the Stack.



Stack

Step 8:

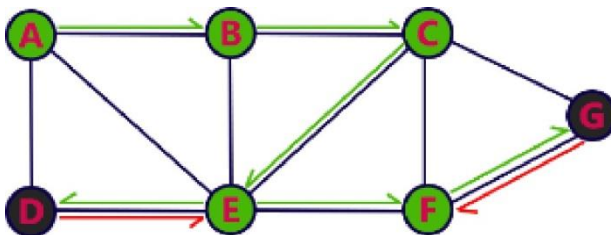
- Visit any adjacent vertex of E which is not visited (G).
- Push G on to the Stack.



Stack

Step 9:

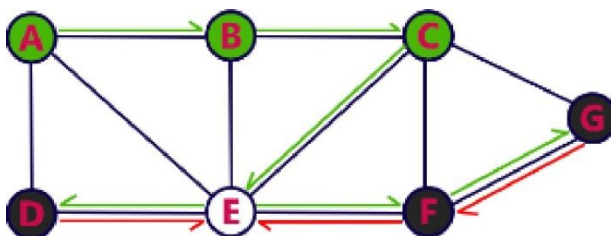
- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.



Stack

Step 10:

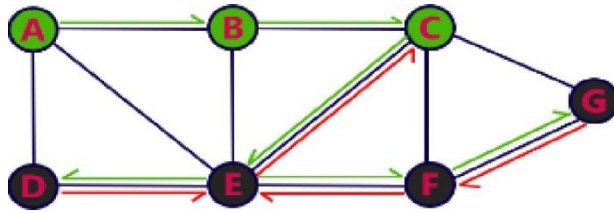
- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.



Stack

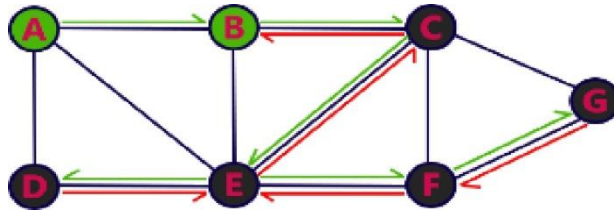
Step 11:

- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



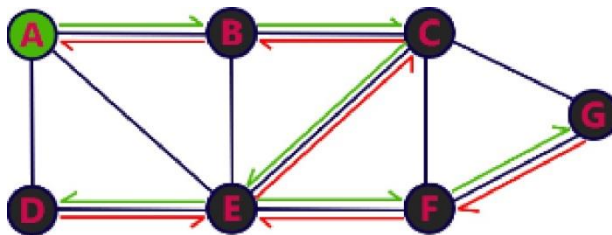
Step 12:

- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack,



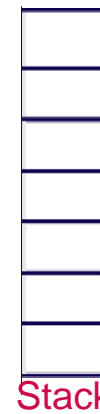
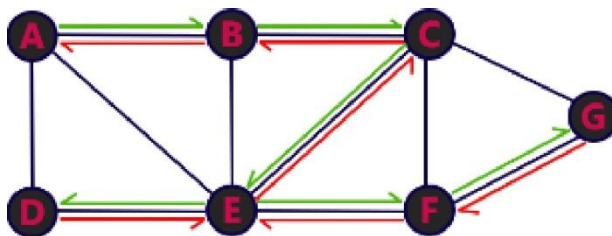
Step 13:

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.

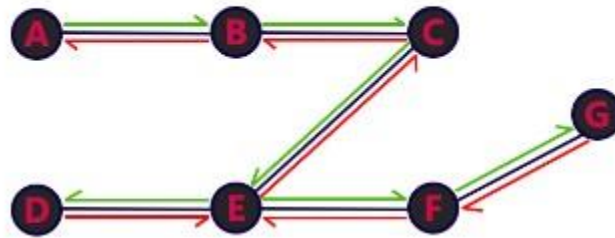


Step 14:

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search)

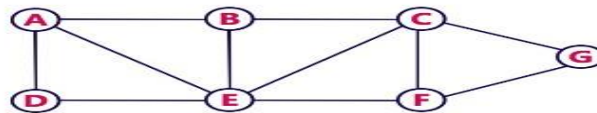
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

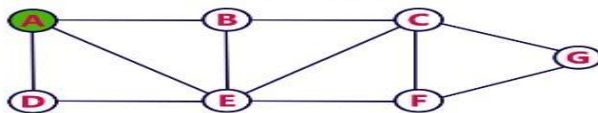
Example:

Consider the following example graph to perform BFS traversal



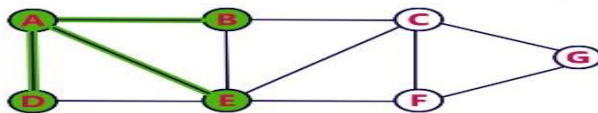
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

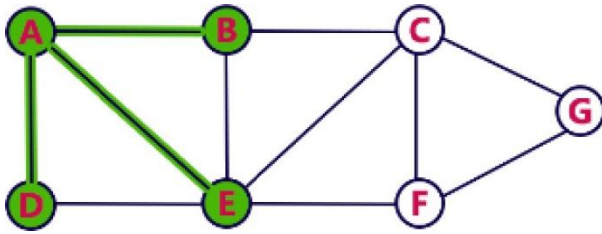


Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



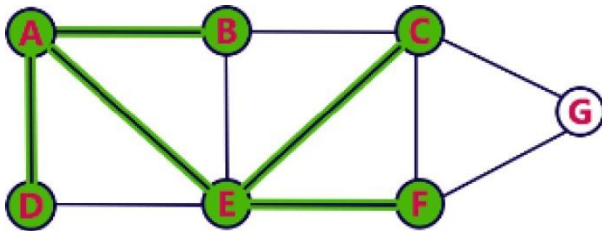
- Visit all adjacent vertices of D which are not visited (there is no vertex).
- Delete D from the Queue.



Queue
EJB

Step 4:

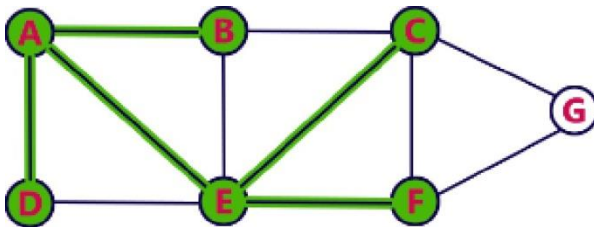
- Visit all adjacent vertices of E which are not visited (C, F).
- Insert newly visited vertices into the Queue and delete E from the Queue.



Queue
QB C F

Step 5:

- Visit all adjacent vertices of B which are not visited (there is no vertex).
- Delete B from the Queue.

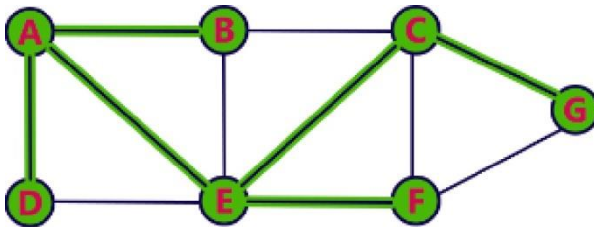


Queue

				C	F	
--	--	--	--	---	---	--

Step 6:

- Visit all adjacent vertices of C which are not visited (G).
- Insert newly visited vertex into the Queue and delete C from the Queue.

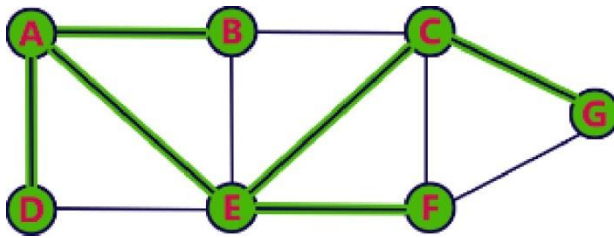


Queue

					F	G
--	--	--	--	--	---	---

Step 7:

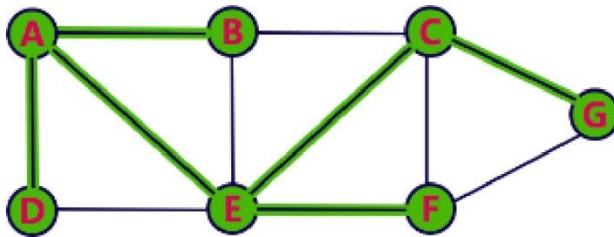
- Visit all adjacent vertices of F which are not visited (there is no vertex).
- Delete F from the Queue.



Queue



- Visit all adjacent vertices of G which are not visited (there is no vertex).
- Delete G from the Queue.

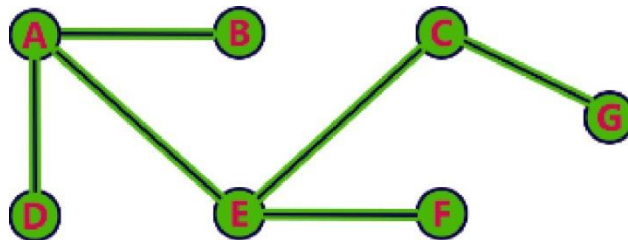


Queue



- Queue became Empty. So, stop the BFS process.

Final result of BFS is a Spanning Tree as shown below...



Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering –

- **Electrical Engineering** – the concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** – Graph theory is used for the study of algorithms. For example,
 - Kruskal's Algorithm
 - Prim's Algorithm
 - Dijkstra's Algorithm
- **Computer Network** – the relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** – the molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** – the parsing tree of a language and grammar of a language uses graphs.
- **General** – Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

UNIT – V (Introduction to Algorithms & Searching and Sorting)

➤ Big O Notation

- Indicates, how hard an algorithm has to work to solve a problem.

Time Complexities of Searching & Sorting Algorithms:

	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n²)	O(n²)	O(n²)
Selection Sort	O(n²)	O(n²)	O(n²)
Insertion Sort	O(n²)	O(n²)	O(n²)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n²)

Linear search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection. Time complexity of linear search is $O(n)$.

Linear Search



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if $i > n$ then go to step 7 Step

3: if $A[i] = x$ then go to step 6 Step 4:

Set i to $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8 Step

7: Print element not found

Step 8: Exit

```
//linear search
#include<stdio.h>
#include<stdlib.h>
```

```
void main()
{
    int a[50],n,i,element,flag=0;
    printf("\n Enter no-of elements:");
    scanf("%d",&n);
    printf("\n Enter array elements:");
```

```

for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\n Enter element to be searched:");
scanf("%d",&element);
for(i=0;i<n;i++)
{
if(a[i]==element)
{
printf("\n element is found at index: %d\n",i);
flag=1;
break;
}
}
if(flag==0)
printf("\n element not found");
}

```

Binary search

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



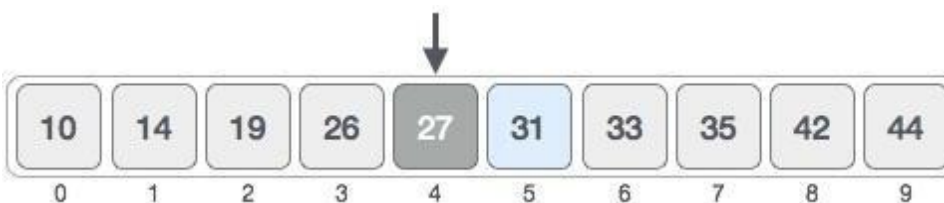
First, we shall determine half of the array by using this formula –

```

n=10;
Low=0;
High=n-1;
mid = (low + high ) / 2

```

Here it is, $(0 + 9) / 2 = 4$ (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again. low

= mid + 1

mid = (low + high) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

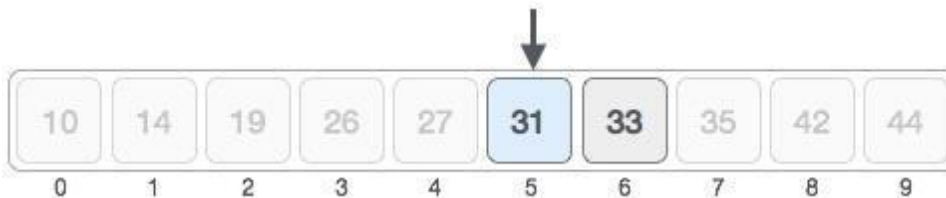


High=mid-1=7-1=6;

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers

Search for x=20

Iteration1`:

L=0 ; h=9; m=4;

Here $x < a[4]$ so $h=m-1=4-1=3$;

No change in l. i.e l=0;

Iteration2`:

L=0; h=3; m=1;

Here $x > a[1]$ so $l = m + 1 = 1 + 1 = 2$;

No change in $h=3$;

Iteration3`:

L=2; h=3; m=2;

Here $x > a[2]$ so $l = m + 1 = 2 + 1 = 3$

No change in $h=3$;

Iteration4`:

L=3; h=3; m=3;

Here $x < a[3]$ so $h = m - 1 = 3 - 1 = 2$;

No change in $l=3$.

Since $L > h$ loop terminates. Here element not found.

//binary search

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

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

```
void main()
```

```
{
```

```
int a[50],n,i,element,flag=0,l,m,h;
```

```
printf("\n Enter no-of elements:");
```

```
scanf("%d",&n);
```

```
printf("\n Enter array elements:");
```

```
for(i=0;i<n;i++)
```

```
scanf("%d",&a[i]);
```

```
printf("\n Enter element to be searched:");
```

```
scanf("%d",&element);
```

```
l=0;h=n-1;
```

```
while(l<=h)
```

```
{
```

```
    m=(l+h)/2;
```

```
    if(element==a[m])
```

```
    {
```

```
        printf("\n element is found at index: %d ",m);
```

```
        flag=1; break;
```

```
    }
```

```
    else if(element<a[m])
```

```
        h=m-1;
```

```
    else
```

```
        l=m+1;
```

```
    }
```

```
    if(flag==0)
```

```
        printf("\n Element not found");
```

```
}
```

BUBBLE SORT

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires $n-1$ passes for sorting. Consider an array

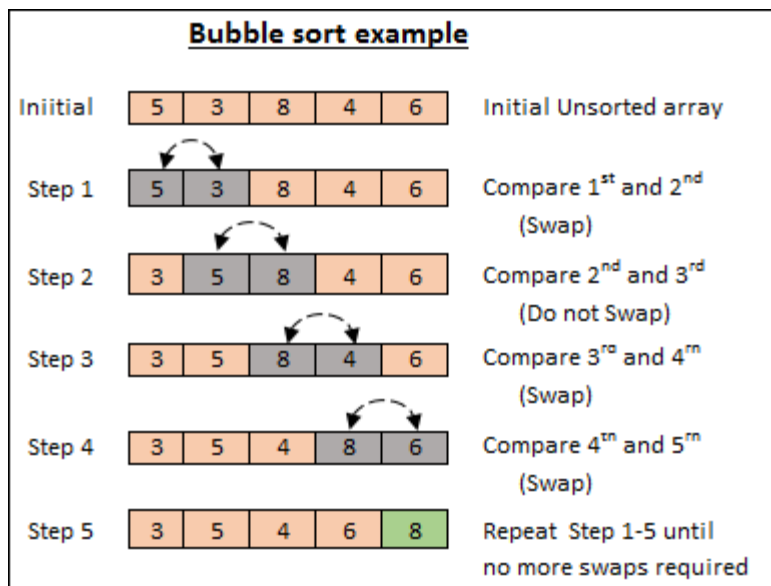
A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

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

first pass:



Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



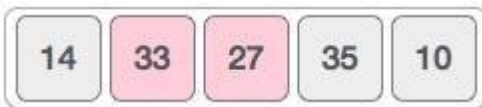
Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



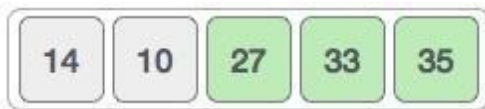
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



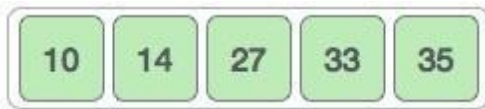
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



45, 23, 32, 10, 16

(J=0; j<n-1; j++)

(J=0; j<n-i-1; j++)

Pass1: i=0

23, 45, 32, 10, 16

23, 32, 45, 10, 16

23, 32, 10, 45, 16

23, 32, 10, 16, 45

Pass2: i=1

23, 32, 10, 16, 45

23, 10, 32, 16, 45

23, 10, 16, 32, 45

~~23, 10, 16, 32, 45~~

Pass3: i=2

10, 23, 16, 32, 45

10, 16, 23, 32, 45

~~10, 16, 23, 32, 45~~

~~10, 16, 23, 32, 45~~

Pass4: i=3

10, 16, 23, 32, 45

~~10, 16, 23, 32, 45~~

~~10, 16, 23, 32, 45~~

~~10, 16, 23, 32, 45~~ (sorted list)

```
// C program for Bubble sort
```

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

```
void main()
```

```
{
```

```
int a[100], i, j, n, temp;
```

```
printf("\nEnter number of elements\n");
```

```
scanf("%d", &n);
```

```
printf("\nEnter elements:");
```

```
for(i=0; i<n; i++)
```

```
scanf("%d", &a[i]);
```

```
printf("\nBefore sorting:\n");
```

```
for(i=0; i<n; i++)
```

```

printf("%d\t", a[i]);

for(i=0;i<(n-1);i++)
{
for(j=0;j<(n-i-1);j++)
{
if (a[j]>a[j+1])
{
temp = a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf("\nSorted list in ascending order:\n");
for(i=0;i<n;i++)
printf("%d\t", a[i]);

}

```

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

The array with **n** elements is sorted by using **n-1** pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap **A[0]** and **A[pos]**. Thus **A[0]** is sorted, we now have **n - 1** elements which are to be sorted.
- In 2nd pas, position **pos** of the smallest element present in the sub-array **A[n-1]** is found. Then, swap, **A[1]** and **A[pos]**. Thus **A[0]** and **A[1]** are sorted, we now left with **n-2** unsorted elements.
- In **n-1**th pass, position **pos** of the smaller element between **A[n-1]** and **A[n-2]** is to be found. Then, swap, **A[pos]** and **A[n-1]**.

Therefore, by following the above explained process, the elements **A[0]**, **A[1]**, **A[2]**,, **A[n- 1]** are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
------	-----	------	------	------	------	------	------

1, i=0	1	2	10	3	90	43	56
2, i=1	2	2	3	10	90	43	56
3, i=2	2	2	3	10	90	43	56
4, i=3	4	2	3	10	43	90	56
5, i=4	5	2	3	10	43	56	90

Sorted A = {2, 3, 10, 43, 56, 90}

Example2:

34 67 21 55 37

Pass1:

I=0, pos=2

Swap(34, 21)

21 67 34 55 37

Pass2:

I=1, pos=2

Swap(67, 34)

21 34 67 55 37

Pass3:

I=2, pos=4

Swap(67, 37)

21 34 37 55 67

Pass 4:

I=3, pos=3

Swap(55, 55)

21 34 37 55 67 (sorted list final)

Algorithm

SELECTION SORT(ARR, N)

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

SMALLEST (ARR, K, N, POS)

- **Step 1:** [INITIALIZE] SET SMALL = ARR[K]
- **Step 2:** [INITIALIZE] SET POS = K

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

```
// C program for selection sort
#include <stdio.h>
void main()
{
    int a[100], i, j, n, temp, pos;

    printf("\nEnter number of elements\n");
    scanf("%d", &n);
    printf("\nEnter elements:");
    for(i=0;i<n;i++)
        scanf("%d", &a[i]);
    printf("\nBefore sorting:\n");
    for(i=0;i<n;i++) printf("%d\t",
        a[i]);

    for(i=0;i<n-1;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[pos])
                pos=j;
        }
        temp=a[i];
        a[i]=a[pos];
        a[pos]=temp;
    }

    printf("\nSorted list in ascending order:\n");
    for(i=0;i<n;i++)
        printf("%d\t", a[i]);

}
```

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k- 1], A[k- 2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Algorithm

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

This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

```
// C program for insertion sort
#include <stdio.h>
void main()
```

```

{
int a[100],i,j,n,temp;

printf("\nEnter number of elements\n");
scanf("%d", &n);
printf("\nEnter elements:");
for(i=0;i<n;i++)
scanf("%d", &a[i]);
printf("\nBefore sorting:\n");
for(i=0;i<n;i++)
printf("%d\t",a[i]);

for(i=1;i<=n-1;i++)
{
temp=a[i];
for(j=i-1;j>=0&& a[j]>temp;j--)
{
a[j+1]=a[j] ;
}
a[j+1]=temp;
}
printf("\nSorted list in ascending order:\n");
for(i=0;i<n;i++)
printf("%d\t",a[i]);

}

```

Example2:

4, 2, 5, 1, 3

Pass 1:

I=1, temp=a[1]=2, j=0
A[1]=a[0]=4 inner loop
A[0]=2 outer loop
2, 4, 5, 1, 3

Pass 2:

I=2, temp=a[2]=5, j=1 Condition
false in inner loop A[2]=5 outer
loop
2, 4, 5, 1, 3

Pass 3:

I=3, temp=a[3]=1, j=2
A[3]=a[2] A[2]=a[1] A[1]=a[0]
inner loop A[0]=1
outer loop 1, 2, 4, 5,
3

Pass 4:

I=4, temp=a[4]=3, j=3
A[4]=a[3] A[3]=a[2]
inner loop
A[2]=3 outer loop
1, 2, 3, 4, 5 (SORTED LIST)

Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
- A Binary Heap is a complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. If the parent node is greater than the children nodes it is called a max heap and if the parent node is smaller than the children nodes it is called a min heap. The heap can be represented by binary tree or array.
- We use an array because Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by $2 * I + 1$ and right child by $2 * I + 2$ (assuming the indexing starts at 0).

Procedure:

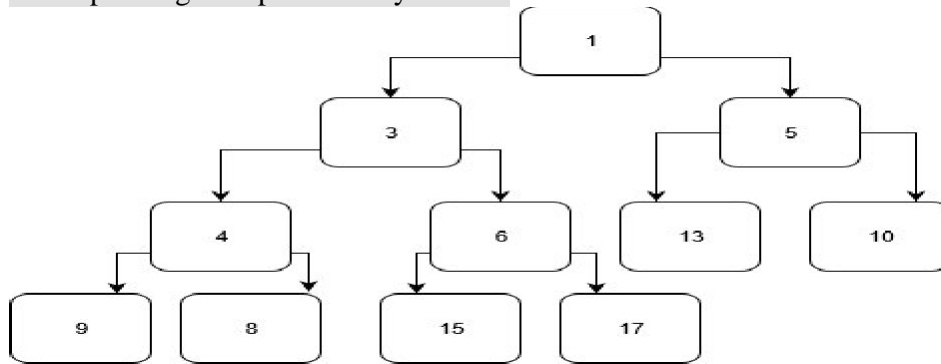
- Initially build a max heap of elements in Array.(Refer 2nd point for what is a Max Heap)
- The root element, that is Array [1], will contain maximum element of Array (since it is a max heap).
- After that, swap this element with the last element of Array and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Building a Max heap:

Consider an array,

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:

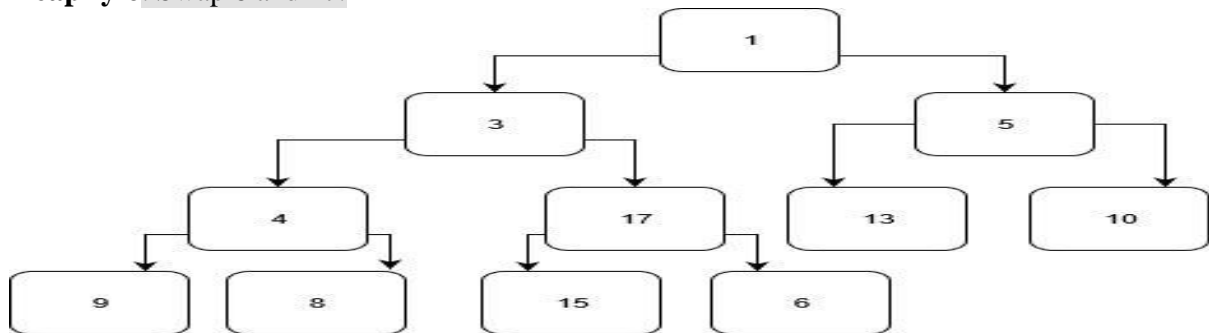


The task to build a Max-Heap from above array

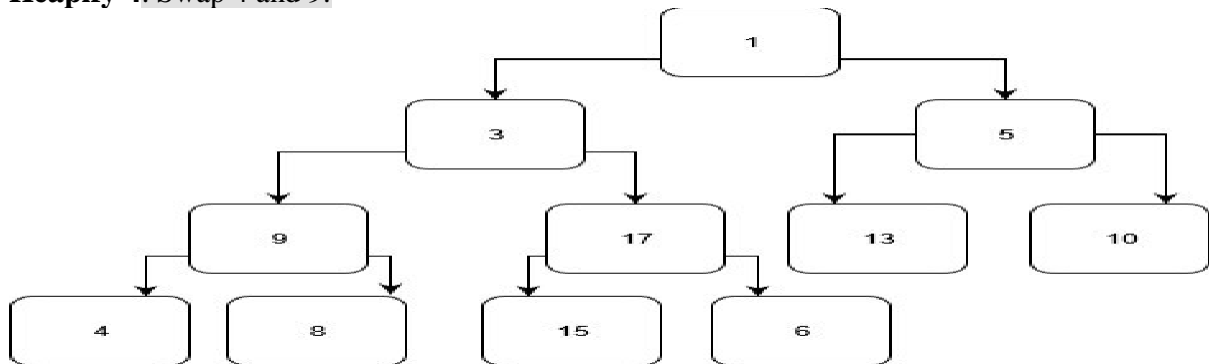
Total Nodes = 11.

To build the heap, heapify only the nodes: [1, 3, 5, 4, 6] in **reverse order** as the rest of the nodes are leaf nodes [13, 10, 9, 8, 15, 17].

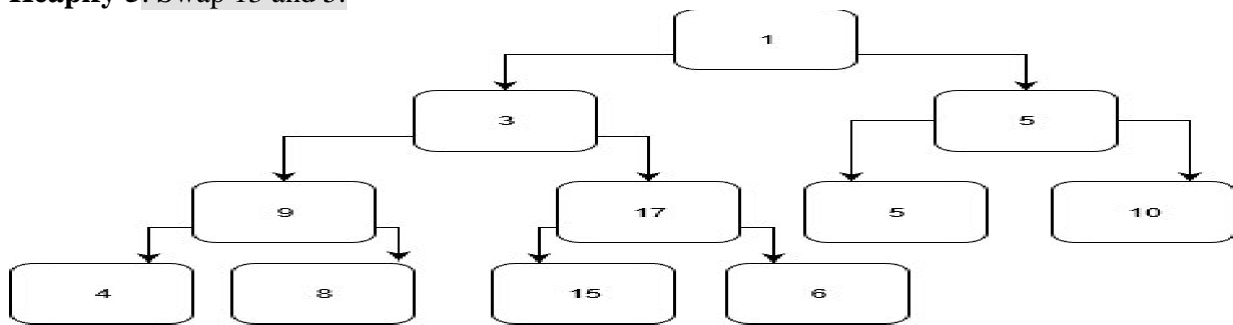
Heapify 6: Swap 6 and 17.



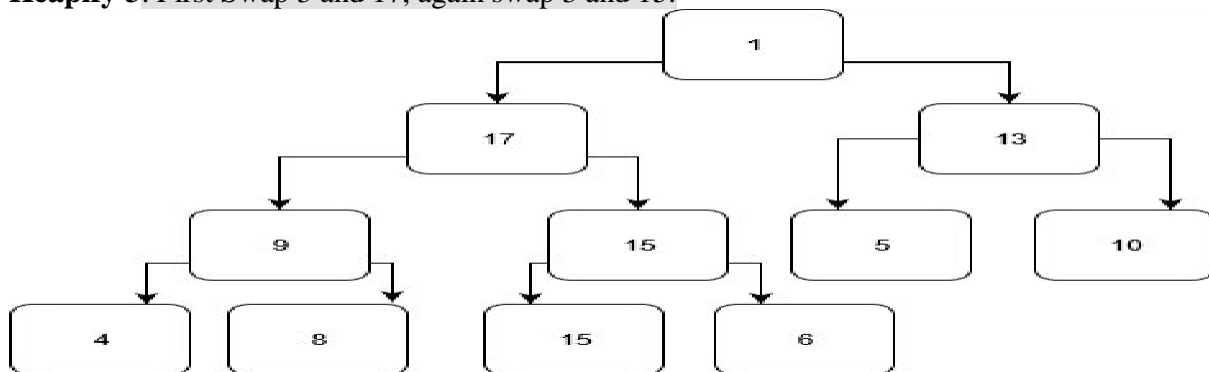
Heapify 4: Swap 4 and 9.



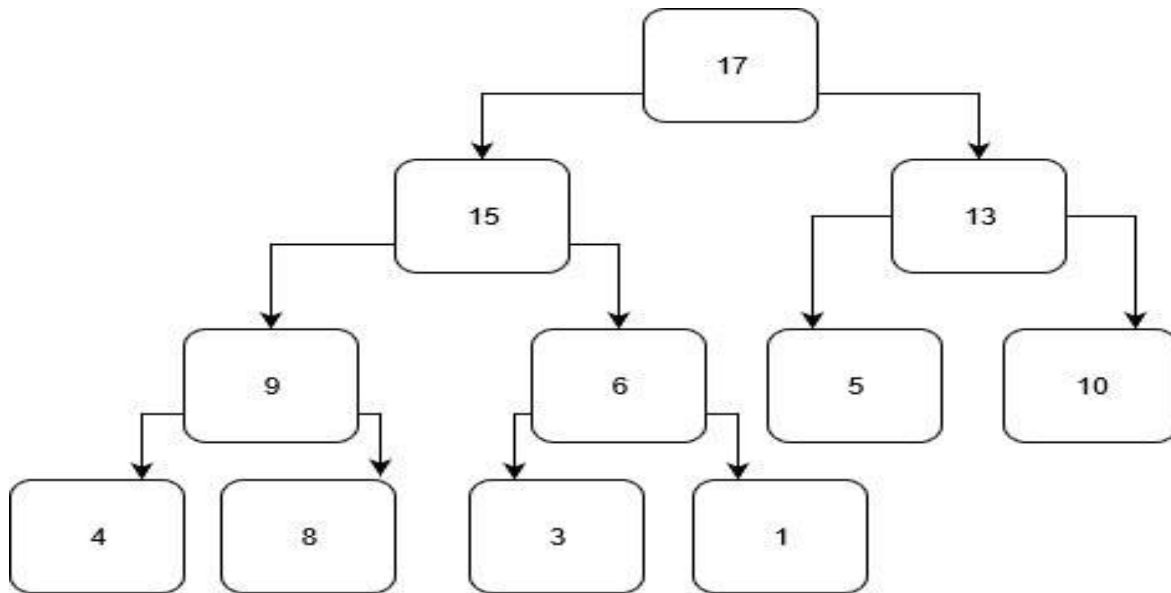
Heapify 5: Swap 13 and 5.



Heapify 3: First Swap 3 and 17, again swap 3 and 15.



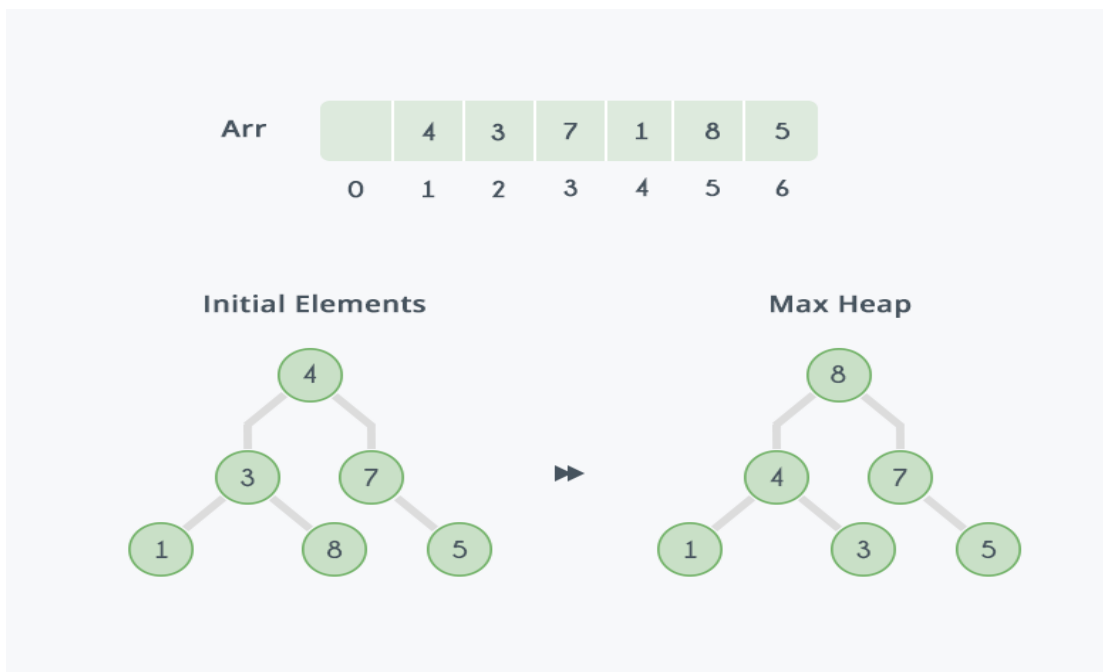
Heapify 1: First Swap 1 and 17, again swap 1 and 15,
Finally swap 1 and 6.



Hence the elements are arranged in a max heap.

Example for heap sort:

In the diagram below, initially there is an unsorted array having 6 elements and then max-heap will be built.



After building max-heap, the elements in the array will be:

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

Step 1: 8 is swapped with 5.

Step 2: 8 is disconnected from heap as 8 is in correct position now.

Step 3: Max-heap is created and 7 is swapped with 3.

Step 4: 7 is disconnected from heap.

Step 5: Max heap is created and 5 is swapped with 1.

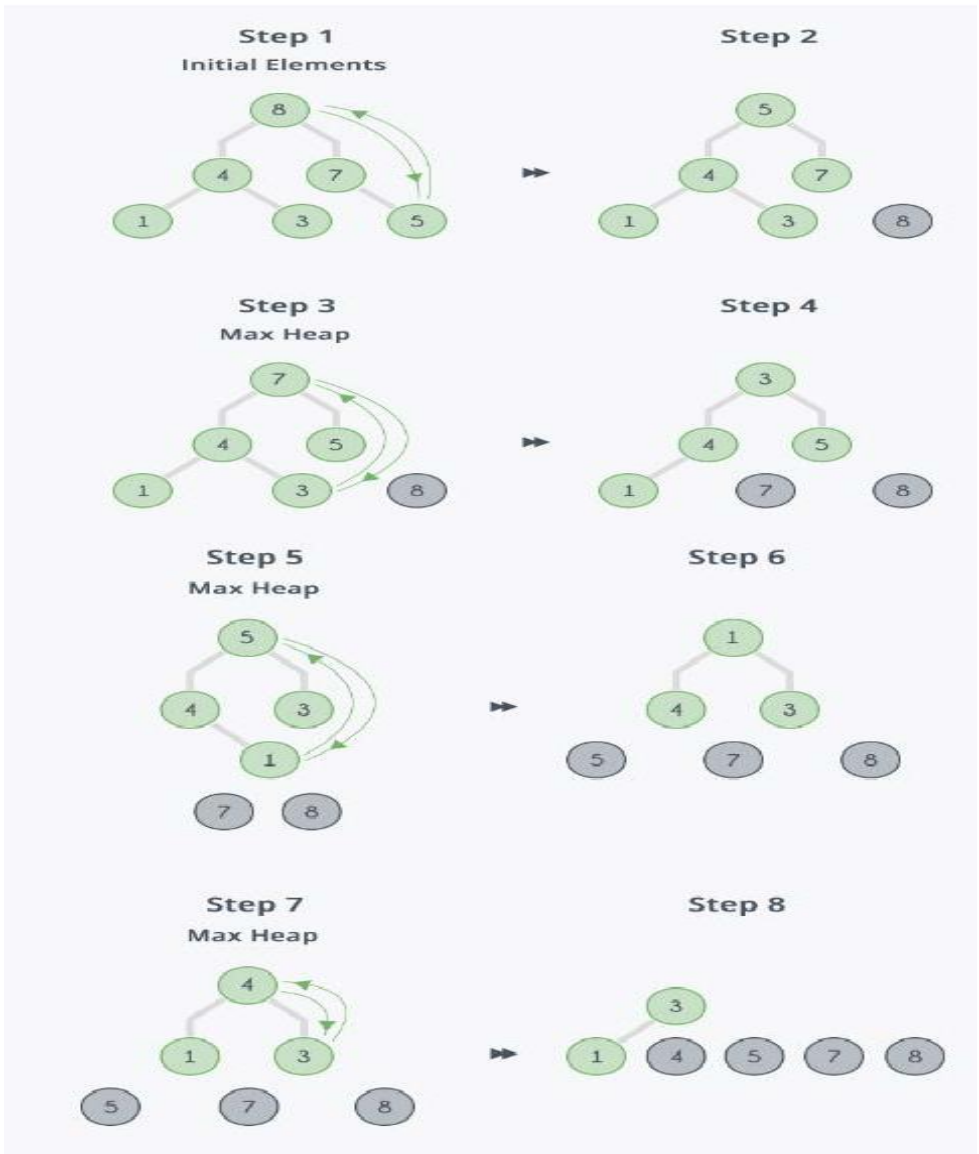
Step 6: 5 is disconnected from heap.

Step 7: Max heap is created and 4 is swapped with 3.

Step 8: 4 is disconnected from heap.

Step 9: Max heap is created and 3 is swapped with 1.

Step 10: 3 is disconnected.



After all the steps, we will get a sorted array.

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

External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Merge Sort

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

Idea:

- Divide the unsorted list into N sub lists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

Time Complexity:

The list of size N is divided into a max of $\log N$ parts, and the merging of all sub lists into a single list takes $O(N)$ time, the worst case run time of this algorithm is $O(N \log N)$.

How Merge Sort Works?

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



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

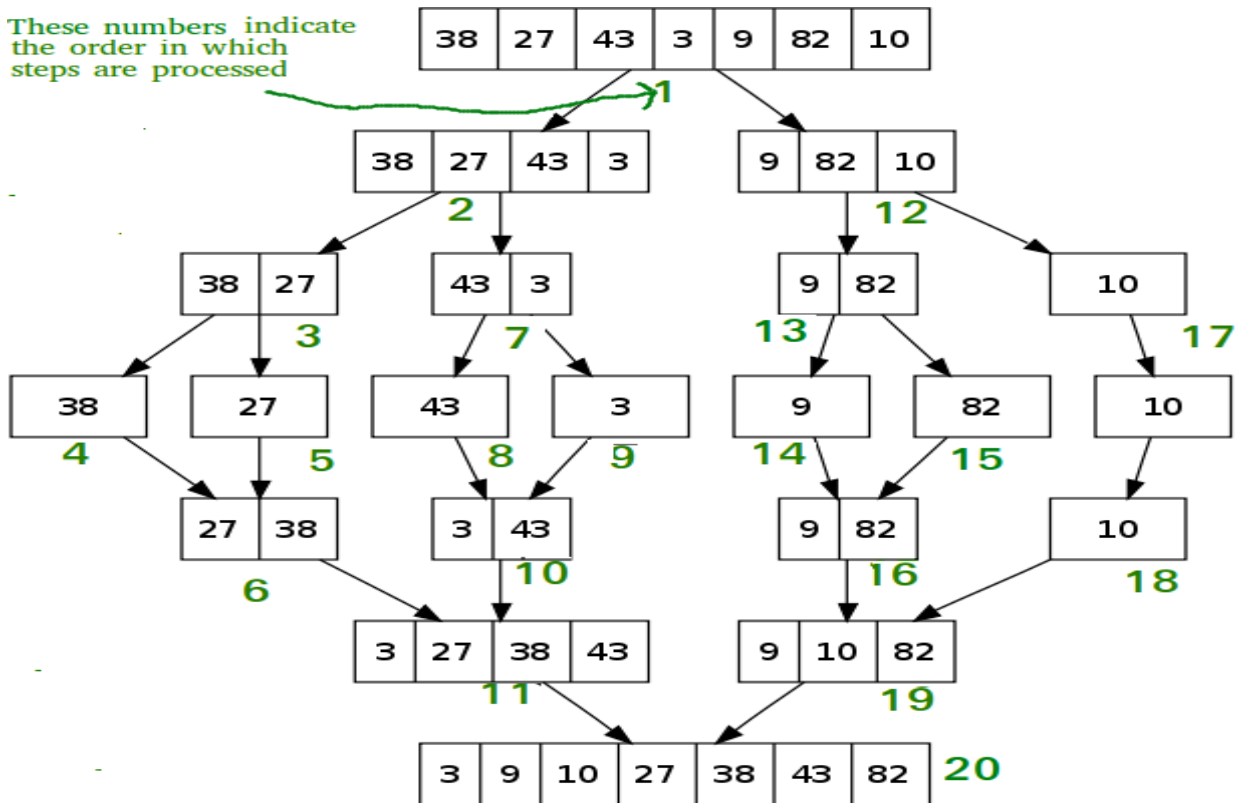


After the final merging, the list should look like this –



Consider another example,

These numbers indicate the order in which steps are processed



Applications of Merge Sort

- Merge sort is useful for sorting linked lists in $O(n \log n)$ time.
- Inversion Count Problem.
- Used in External Sorting.

pattern matching algorithms

(1)

the problem of string matching

Given a string 's', the problem of string matching deals with finding whether a pattern 'p' occurs in 's' and if 'p' does occur then returning position in 's' where 'p' occurs.

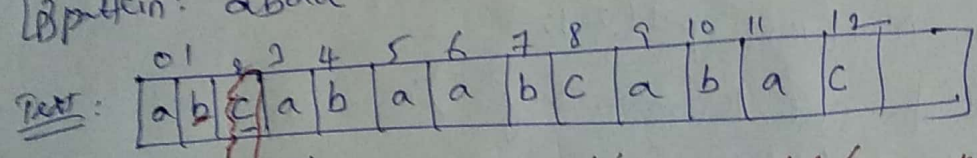
Brute force

time complexity: $O(mn)$

Here, compare the first element of the pattern to be searched 'p' with the first element of the string 's'. If the first element matches, compare the second element of 'p' with the second element of 's'. If match found proceed likewise until entire 'p' is found. If a mismatch is found at any position, shift 'p' to the position to the right and repeat comparisons beginning from first element of 'p'.

[A] Text: abcabaabcabac

[B] pattern: abaa



pattern: a b a a mismatch, shift 'p' to the right (one position)

a b a a mismatch

a b a a

a b a a match found, return the position

drawback: comparisons are more

4 program

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

```
#include <string.h>
```

```
int search(char *t, char *p)
```

```
{ int i, j; int M = strlen(t);
```

```
int N = strlen(p);
```

```
for(i=0; i<=M-N; i++)
```

```
{ for(j=0; j<N; j++)
```

```
if (p[j] != t[i+j])  
break;
```

```
}
```

```
if (j == N)
```

```
return i; }  
if (j != N) printf("pattern found at index %d", i);
```

```
return -1; }  
if (j == -1) printf("pattern not found");
```

```
}
```

```
void main()
```

```
{ char text[100], pat[20];
```

```
printf("Enter the main string:");
```

```
gets(text);
```

```
printf("Enter the pattern:");
```

```
gets(pat);
```

```
search(text, pat);
```

```
if (i != -1)
```

```
printf("pattern not found\n");
```

```
else
```

```
printf("pattern found at index %d\n", i);
```

```
}
```

01 2 3 4 5 6 7 8 9 10
text: abcabac abcd
pat: abaa

M=11, N=4

```
for(i=0; i<=M-N; i++)
```

```
{ for(j=0; j<N; j++)
```

```
p[j] != t[i+j]
```

```
-----  
p[0] = t[0]
```

```
-----  
p[0] = t[1]
```

```
-----  
p[0] = t[2]
```


② KMP Algorithm (Knuth-Morris-Pratt) $O(m+n)$ ①

- It is one of the most popular patterns matching algorithms.
- It was the first linear time complexity algorithm for string matching and used to find a pattern in a Text.
- This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called 'prefix table' to skip characters comparison while matching. It is also known as LPS Table. (Longest proper prefix which is also suffix)

Steps for creating LPS table (prefix table)

- Step
- ① Define a one dimensional array with the size equal to the length of the pattern i.e. $LPS[size]$
 - ② Define variables i & j . set $i=0, j=1$ and $LPS[0]=0$
 - ③ Compare the characters at $pattern[i]$ and $pattern[j]$.
 - ④ if both are matched then set $LPS[i]=i+1$ and increment both i and j by one. goto step ③.
 - ⑤ if both are not matched then check the value of variable i . if it is '0' then set $LPS[j]=0$ and increment j value by one, if it is not '0' then set $i = LPS[i-1]$. goto step ③.
 - ⑥ Repeat above steps until all the values of $LPS[]$ are filled.

Q) create LPS table for the following patterns

pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Sol: Define LPS[] array with size '7'

0	1	2	3	4	5	6

Now, $i=0$, $j=1$ and $LPS[0]=0$

0	1	2	3	4	5	6
0						

$i=0, j=1$

Compare $pattern[0]$ with $pattern[1]$ i.e. 'A' with 'B'
mismatch and $i=0$, $LPS[1]=0$ and increment 'j' by one i.e. $j=2$

0	1	2	3	4	5	6
0	0					

$i=0, j=2$

Compare $pattern[0]$ with $pattern[2]$ i.e. 'A' with 'C'.
mismatch and $i=0$, $LPS[2]=0$ and increment 'j' i.e. $j=3$

0	1	2	3	4	5	6
0	0	0				

$i=0, j=3$

Compare $pattern[0]$ with $pattern[3]$ i.e. 'A' with 'D'.
mismatch and $i=0$, $LPS[3]=0$ and increment 'j' i.e. $j=4$

0	1	2	3	4	5	6
0	0	0	0			

$i=0, j=4$

Compare $pattern[0]$ with $pattern[4]$ i.e. 'A' with 'A', match
 so, $LPS[4]=0+1=1$ and increment both i & j i.e. $i=1, j=5$

0	1	2	3	4	5	6
0	0	0	0	1		

$i=1, j=5$

Compare $pattern[1]$ with $pattern[5]$, 'B' = 'B', match
 so, $LPS[5]=1+1=2$, increment 'i' & 'j'. i.e. $i=2, j=6$

0	1	2	3	4	5	6
0	0	0	0	1	2	

$i=2, j=6$

Compare $pattern[2]$ with $pattern[6]$ i.e. 'C' with 'D'.
mismatch and $i \neq 0$, set $i = LPS[i-1] = LPS[1] = 0$

0	1	2	3	4	5	6
0	0	0	0	1	2	

$i=0$ and $j=6$

Compare $pattern[0]$ with $pattern[6]$ i.e. 'A' with 'D', mismatch
 and $i=0$. so, $LPS[6]=0$ and increment 'j'

LPS[]

0	1	2	3	4	5	6
0	0	0	0	1	2	0

Use of LPS table (a) prefailure when a mismatch has occurred with the first character in the pattern, move the pattern one position right (b)

It is used to decide how many characters are to be skipped for comparison when a mismatch has occurred.

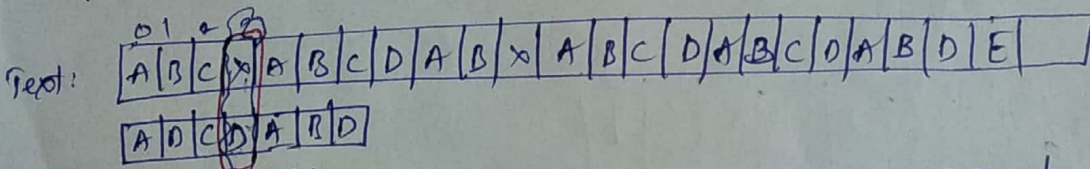
⇒ when a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. if it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. if it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in the pattern with the mismatched character in the text.

⊙ Consider the following text and pattern

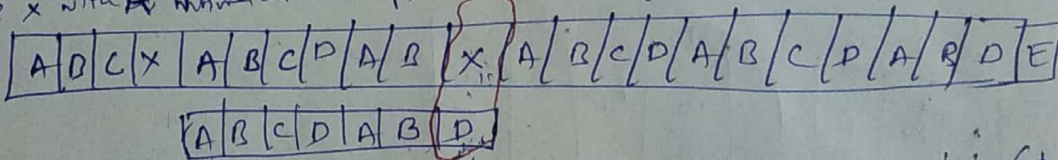
Text: ABC ABCDAB ABCDABCDABDE

pattern: ABCDABD

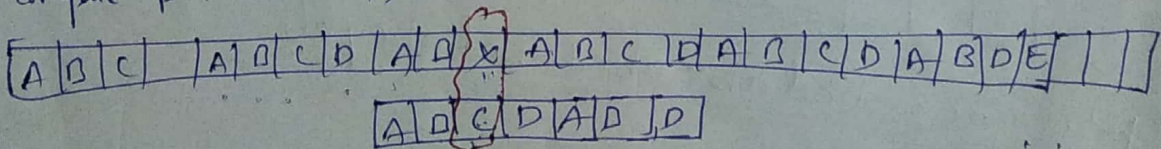
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0



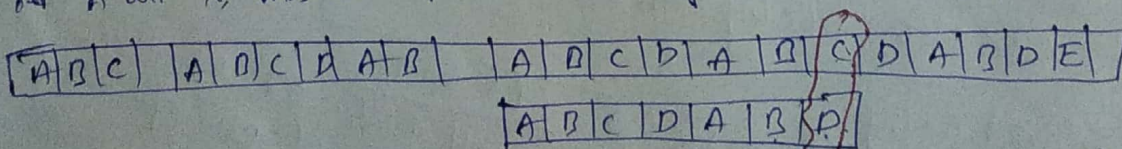
Here mismatch occurred at pattern[3]. So take LPS[2] value i.e '0'. we must compare first character in the pattern with the ~~next~~ character of text. i.e X with A mismatch and LPS[0]=0, move right.



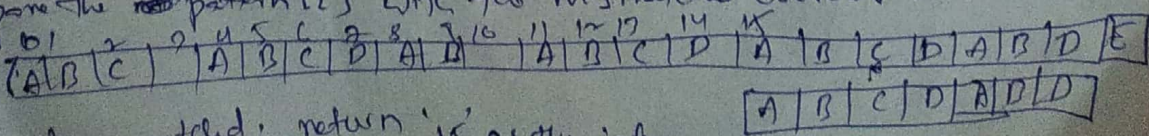
Here mismatch occurred at pattern[6]. So take LPS[5] i.e '2' (≠ 0) we compare pattern[2] with mismatched character in the text



Here, mismatch occurred at pattern[2]. So take LPS[2] i.e '0'. compare pattern[0] with the ~~next~~ ~~next~~ mismatched character in the text i.e 'A' with X, mismatch LPS[0]=0. So, compare pattern[0] with X, move right



mismatch occurred at pattern[6]. So take LPS[5] i.e '2' (≠ 0) compare the ~~next~~ pattern[2] with the mismatched character in the text



compute failure function (or) prefix table for the following strings

- ① abcaby ② aabaabaaa ③ abcdabcy
④ abcdabca

①

	0	1	2	3	4	5
	a	b	c	a	b	y
LPS[]	0	0	0	1	2	0

↑ ↑ ↑ ↑ ↑
i i i i i

initializing
i=0, j=0
LPS[0]=0

②

	0	1	2	3	4	5	6	7	8
	a	a	b	a	a	b	a	a	a
	0	1	0	1	2	3	4	5	2

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
i i i i i i i i

③

	0	1	2	3	4	5	6	7
	a	b	c	d	a	b	c	y
	0	0	0	0	1	2	3	0

④

	0	1	2	3	4	5	6	7
	a	b	c	d	a	b	c	a
	0	0	0	0	1	2	3	1

Text: abcxabcdabxabcdabedabcyab
 pattern: abcdabcy

Text: abxabcabcabyabcd
 pattern: abcaby

Advantages: optimal & fast
 disadvantages: does not work so well as the size of the alphabet increases. By which more chances of mismatch occurs.

$O(m+n)$

③ Boyer-Moore pattern matching Algorithm (11)

It preprocesses the pattern based on two approaches

- ① Bad character rule
- ② Good Suffix rule

At every step, it moves the pattern by the maximum of the moves suggested by the two rules. So, it uses best of the two rules at every step. Unlike the previous pattern searching algorithms, Boyer-Moore algo starts matching from the last character of the pattern.

Bad character rule:

The character of the text which doesn't match with the current character of the pattern is called the Bad character. Upon mismatch, we shift the pattern until-

- Case ① The mismatch becomes a match
- or ② pattern 'p' move past past the mismatched character.

eg: case 1: (The mismatch becomes a match)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Text[]:	G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	A	C	C
Pattern[]:				T	A	T	G	T	G									

↑ mismatch

Start comparison from 5, we got a mismatch at index '3'. Here bad character is 'A'. So we will search for last occurrence of 'A' in the pattern. we got 'A' at index 1. Now we will shift pattern so that 'A' in the pattern get aligned with 'A' in the text. i.e

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Text[]:	G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	A	C	C
Pattern[]:				T	A	T	G	T	G									

↑ mismatch becomes match

eg: case 2: (pattern 'p' move past the mismatched character)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Text[]:	G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	A	C	C
Pattern[]:									T	A	T	G	T	G				

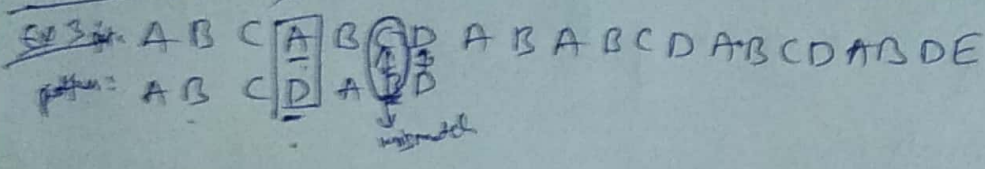
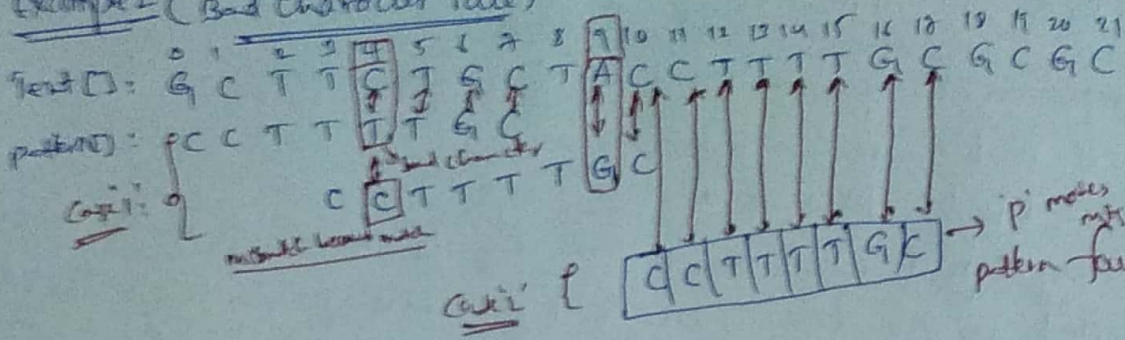
↑ mismatch

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Text[]:	G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	A	C	C
Pattern[]:									T	A	T	G	T	G				

↑ mismatch

Here, we have a mismatch at index 7 i.e 'C'. 'C' does not exist in the pattern, so shift pattern past to the position 7.

Example 2 (Bad character rule)

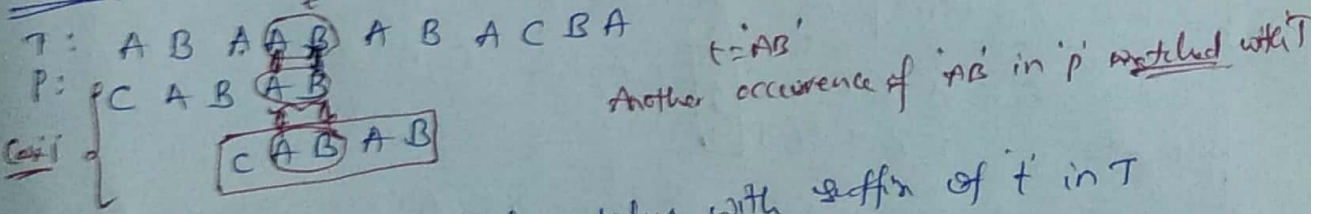


Good suffix rule

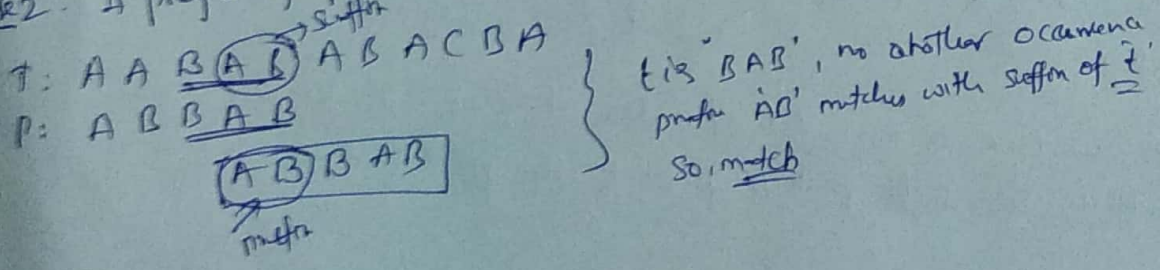
Let t be substring of text T which is matched with substring of pattern p . Now we shift pattern p until:

- Another occurrence of t in p matched with t in T .
- A prefix of p , which matches with suffix of t .
- p moves past t .

Case 1: Another occurrence of t in p matched with t in T



Case 2: A prefix of p , which matches with suffix of t in T



Case 3: 'p' moves past 't'

T: A A C A B A B A C B A
 P: C B A A B

C B A A B

Here, there exist no occurrence of t("AB") in p and also there is no prefix in 'p' which matches with the suffix of t. So, we will shift the 'p' past the 't'.

Ex:

T: C G T G C C T A C T T A C T T A C T T A C T T A C
 P: C T T A C T T A C T

another occurrence of t

C T T A C T T A C } Case 1

T: C G T G C C T A C T T A C T T A C T T A C T T A C
 P: C T T A C T T A C

C T T A C T T A C

prefix of 'p' and suffix of 't'

Ex:

T: G T T A T A Q C T G A T C G C G G C G T A G C G G C G A A
 P: G T A G C G G C G

best of two

steps: bc: 6 steps forward, gs: 0

steps: bc: 0, gs: 2

G T A G C G G C G

G T A G C G G C G

G T A G C G G C G

pattern found

Steps
 bc: 2, gs: 7
 t = G C G G C G
 prefix = G, suffix = G
 match prefix of pattern & suffix of text.