CO1

Linked List Applications

A polynomial is of the form: $\sum_{i=0}^{n} \mathrm{ci}x^{i}$

Where, c_i is the coefficient of the i^{th} term and
n is the degree of the polynomial

Some examples are:

$5x^2 + 3x + 1$
$12x^3 - 4x$
$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

Let assume the polynomials in decreasing order of degree.

Polynomial Representation

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$ illustrates in figure

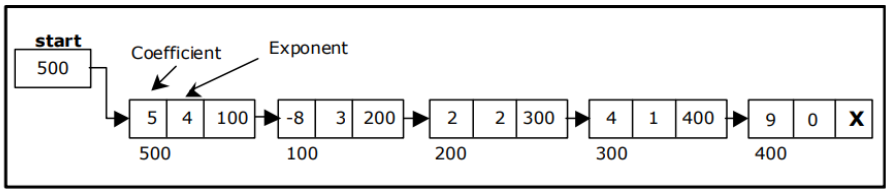
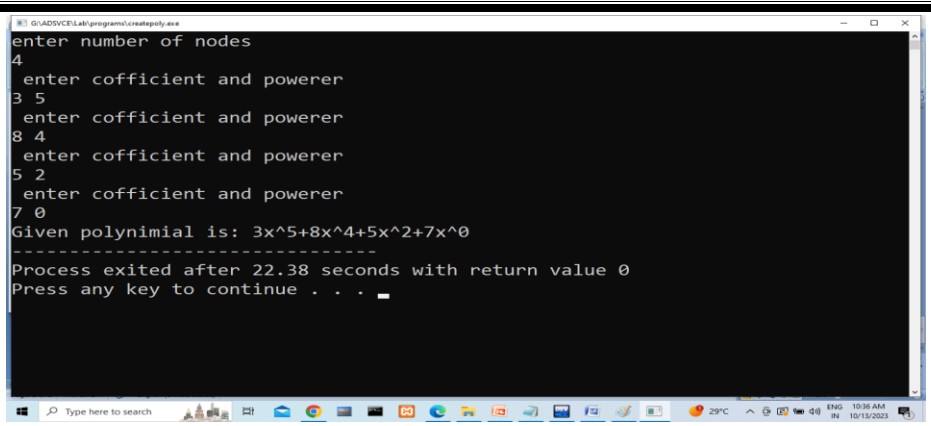


Figure: Representation of polynomial $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$

Write a C program to represent polynomial using linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
 int coeff;
 int power;
 struct node *next;
};
struct node * createpoly (struct node *head)
{
 struct node *newnode;
 int i, n;
```

```
 struct node *temp;
 printf ("enter number of nodes \n");
 scanf ("%d", &n);
 for (i = 1; i <= n; i++)
  {
    newnode = (struct node *) malloc (sizeof (struct node));
    printf (" enter cofficient and powerer\n");
    scanf ("%d", &newnode->coeff);
    scanf ("%d", &newnode->power);
    newnode->next = 0;
    if (head == 0)
          head = newnode;
    else
          {
           temp = head;
           while (temp->next != 0)
             temp = temp->next;


           temp->next = newnode;
          }
  }
 return head;
}
void print (struct node *temp)
{
 while (temp != 0)
  {
    printf ("%dx^%d", temp->coeff, temp->power);
    if (temp->coeff >= 0)
          {
           if (temp->next != NULL)
             printf ("+");
          }
    temp = temp->next;
  }
}
void main()
{
struct node *poly=0,*p;
p=createpoly(poly);
printf("Given polynimial is: ");
print(p);
}
Output
```



Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

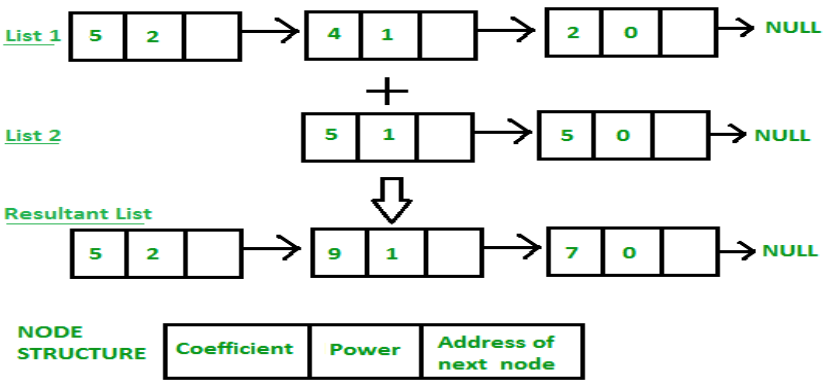
• Read two polynomials

. • Add them.

• Display the resultant polynomial.

Example: Polynomial1: $5x^2+4x+2$

Polynomial2: 5x+5



Algorithm/

1.Declare variables that point to the head of the linked list.

2.Compare the power of the first polynomial of both lists.

I .If it is the same then add their coefficients and push them into the resultant list. Also, increment both the variables so that it points to

the next polynomial.

2.Else, Push the polynomial of the list whose power is greater than the other. Also, increment that particular list.

3. Keep repeating the 2nd step until one of the variables reaches the end of the list.

4. Then check for the remaining data in both of the lists and add it to the resultant list

Write a C program to implement polynomial addition using linked list.

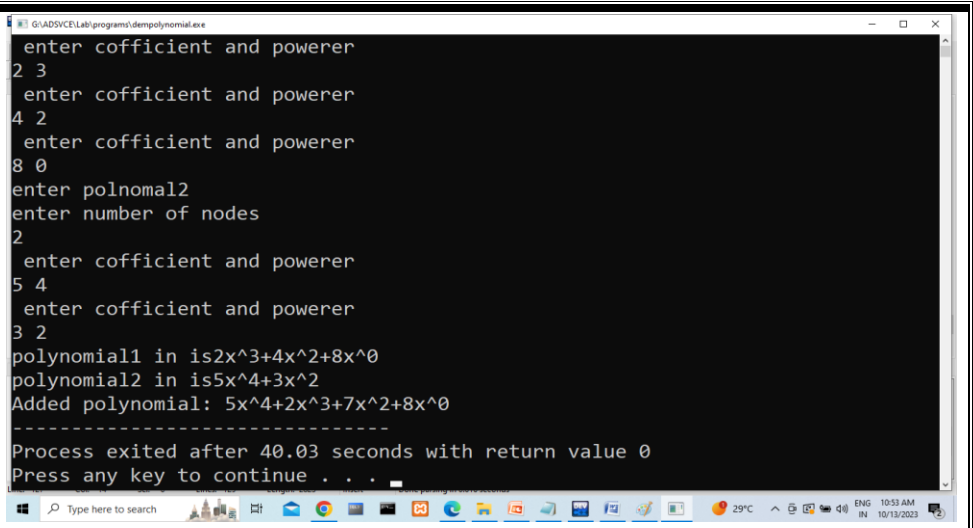
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int coeff;
  int power;
  struct node *next;
};
struct node * createpoly (struct node *head)
{
  struct node *newnode;
  int i, n;
  struct node *temp;
  printf ("enter number of nodes \n");
  scanf ("%d", &n);
  for (i = 1; i <= n; i++)
    {
      newnode = (struct node *) malloc (sizeof
(struct node));
      printf (" enter cofficient and powerer\n");
```

```
      scanf ("%d", &newnode->coeff);
      scanf ("%d", &newnode->power);
      newnode->next = 0;
      if (head == 0)
        head = newnode;
      else
        {
          temp = head;
          while (temp->next != 0)
            temp = temp->next;

          temp->next = newnode;
        }
    }
  return head;
}

void print (struct node *temp)
{
  while (temp != 0)
    {
      printf ("%dx^%d", temp->coeff, temp->power);
      if (temp->coeff >= 0)
        {
          if (temp->next != NULL)
            printf ("+");
        }
      temp = temp->next;
```

```
    }
}

// Function Adding two polynomial numbers
struct node * polyadd (struct node *poly1, struct
node *poly2, struct node *poly)
{
  struct node *p1 = poly1;
  struct node *p2 = poly2;
  struct node *p, *prev=NULL;
  while (p1 != NULL && p2 != NULL)
    {
      p = (struct node *) malloc (sizeof (struct
node));
      p->next=NULL;
      if (p1->power > p2->power)
        {

          p->power = p1->power;
          p->coeff = p1->coeff;
                  p1 = p1->next;
        }
      else if (p1->power < p2->power)
        {
          p->power = p2->power;
          p->coeff = p2->coeff;
          p2 = p2->next;
        }
      else
```

```
        {
          p->power = p1->power;
          p->coeff = p1->coeff + p2->coeff;
          p1 = p1->next;
          p2 = p2->next;
        }
      if (poly == NULL)
          poly = p;
    if (prev != NULL)
       prev->next = p;
           prev = p;
       }
   while (p1 != NULL && p2 == NULL)
    {
        p = (struct node *) malloc (sizeof (struct
node));
      p->power = p1->power;
      p->coeff = p1->coeff;
      p->next = NULL;
      if(prev != NULL) prev->next = p;
      prev = p;
      p1 = p1->next;
    }
  while (p1 == NULL && p2 != NULL)
    {
        p = (struct node *) malloc (sizeof (struct
node));
      p->power = p2->power;
      p->coeff = p2->coeff;
```

```
      p->next = NULL;
      if(prev != NULL) prev->next = p;
      prev = p;
      p2 = p2->next;
    }

  return poly;
}
void main ()
{
  struct node *poly1 = NULL, *poly2 = NULL, *poly
= NULL, *p1;
  printf ("enter polnomal1\n");
  poly1 = createpoly (poly1);
  printf ("enter polnomal2\n");
  poly2 = createpoly (poly2);
  printf ("polynomial1 in is");
  print (poly1);
  printf ("\npolynomial2 in is");
  print (poly2);

  // Function add two polynomial numbers
  p1 = polyadd (poly1, poly2, poly);
  // Display resultant List
  printf ("\nAdded polynomial: ");
  print (p1);
}
```



Multiplication of 2 for Polynomial Multiplication using Linked List

Let's try to understand the problem with the help of examples by referring to the websites to learn to program.

Suppose the given linked lists are:
Poly1: 3×3 + 6×1 – 9
Poly2: 9×3 – 8×2 + 7×1 + 2

- Now, according to the problem statement, we need to multiply these polynomials Poly1 and Poly2.
- So we will multiply each term in Poly1 with every term in Poly2, and then we will add up all the terms with the same power of x such that each term in the final resultant polynomial multiplication using linked list has a different power of x.

output

Resultant Polynomial: 27x6 – 24x5 + 75x4 – 123x3 + 114x2 – 51x1 – 1

Algorithm of Polynomial Multiplication using Linked List in C

Here is the algorithm for polynomial multiplication using linked lists in C:

- First define the node structure to hold the coefficient and exponent values, as well as a pointer to the next node.
- Create a function to create a new node with the specified coefficient and exponent values.
- Create a function to insert a new node into a linked list based on its exponent value.
- Create a function to print the linked list.
- Create a function to multiply two polynomials represented as linked lists.
- Iterate over each term in the first polynomial, multiply it with each term in the second polynomial, and add the resulting term to the result polynomial using the insert_node() function.
- Return the resulting polynomial as a linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int coeff;
  int power;
  struct node *next;
};
struct node * createpoly (struct node *head)
{
  struct node *newnode;
  int i, n;
  struct node *temp;
  printf ("enter number of nodes \n");
  scanf ("%d", &n);
  for (i = 1; i <= n; i++)
    {
      newnode = (struct node *) malloc (sizeof (struct node));
      printf (" enter cofficient and powerer\n");
      scanf ("%d", &newnode->coeff);
      scanf ("%d", &newnode->power);
      newnode->next = 0;
      if (head == 0)
              head = newnode;
      else
              {
                temp = head;
                while (temp->next != 0)
```

```
                temp = temp->next;


                temp->next = newnode;
              }
    }
  return head;
}


void print (struct node *temp)
{
  while (temp != 0)
    {
      printf ("%dx^%d", temp->coeff, temp->power);
      if (temp->coeff >= 0)
              {
                if (temp->next != NULL)
                  printf ("+");
              }
      temp = temp->next;
    }
}
struct node* polymult(struct node *poly1,struct node *poly2,struct node *poly)
{
struct node *p1=poly1;
struct node *p2;
struct node *p3=poly;
struct node *prev;
prev=p3;
```

```
while(p1!=NULL)
{
p2=poly2;
while(p2!=NULL)
{
struct node *newnode;
newnode=(struct node*)malloc(sizeof(struct node *));
newnode->next=NULL;
newnode->power=p1->power+p2->power;
newnode->coeff=p1->coeff*p2->coeff;
if(p3==NULL)
 {

 prev=newnode;
 p3=newnode;
}
else
{
  prev->next=newnode;
  prev=newnode;
}
p2=p2->next;
}
p1=p1->next;
}
return p3;
}
void main ()
```

```
{
  struct node *poly1 = NULL, *poly2 = NULL, *poly = NULL, *p1;
  printf ("enter polnomal1\n");
  poly1 = createpoly (poly1);
  printf ("enter polnomal2\n");
  poly2 = createpoly (poly2);
  printf ("polynomial1 in is");
  print (poly1);
  printf ("\npolynomial2 in is");
  print (poly2);
  // Function add two polynomial numbers
  p1 = polymult(poly1, poly2, poly);
  // Display resultant List
  printf ("\nAdded polynomial: ");
  print (p1);
}
```

Linked List

Introduction

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

void *malloc (number_of_bytes) Since a void * is returned the C standard states that this pointer can be converted to any type.

For example, char *cp; cp = (char *) malloc (100); Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example, int *ip; ip = (int *) malloc (100*sizeof(int));

free() is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is: free (ptr); The advantage of free() is simply memory management when we no longer need a block.

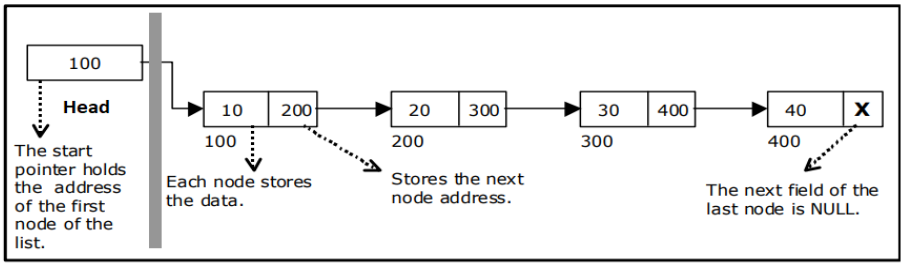
Comparison between array and linked list:

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.

Linked List Concepts:

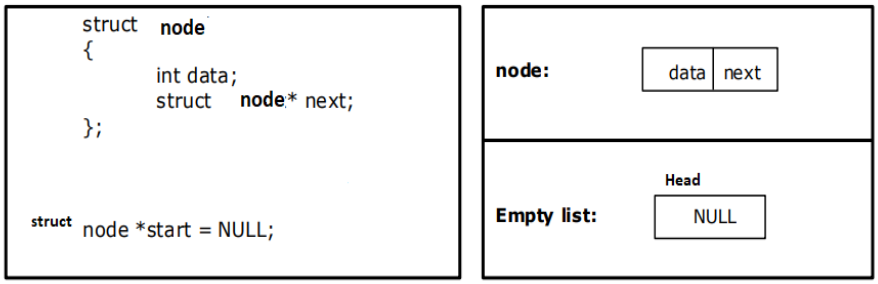
A linked list is a sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. 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.

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the “head” node.



The beginning of the linked list is stored in a "Head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list.

Code can access any node in the list by starting at the Head and following the next pointers.



Circular Singly Linked List

Def: It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called 'Head' pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

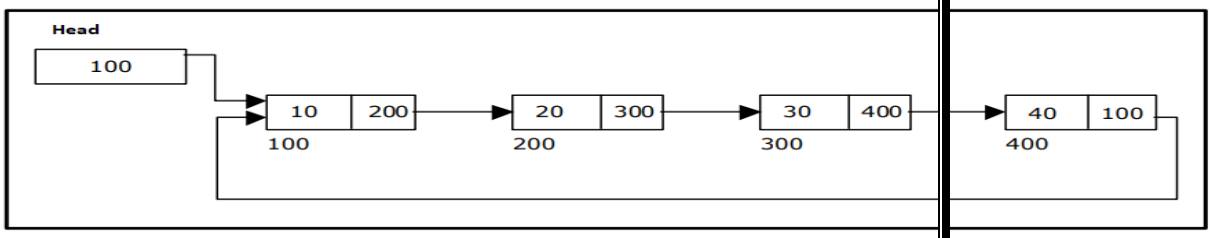


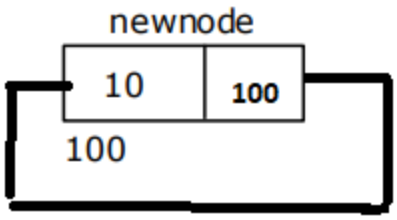
Figure : Circular Linked List

1. **Inserting a node at the beginning:**

The following steps are to be followed to insert a new node at the beginning of the circular list:

• Create a new node using malloc. newnode= (struct node *) malloc(sizeof(struct node));

Insert data and make next head



• If the list is empty, assign new node as head. if (head==NULL)

head= newnode;

newnode -> next = head;

• If the list is not empty,

follow the steps given below:

temp = head;

while(temp -> next != head)

temp= temp -> next;

newnode -> next = head;

head = newnode;

temp -> next = head;(last node pointing to first node)

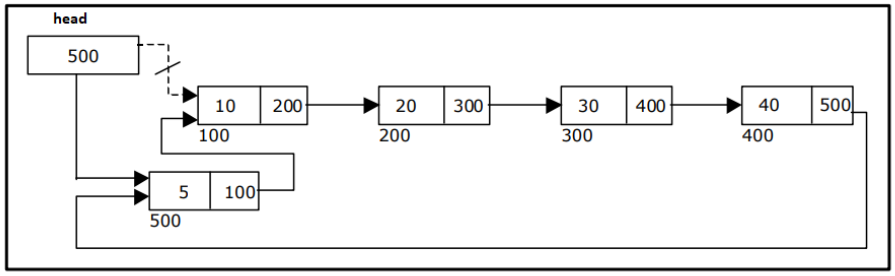
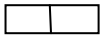


Figure : Inserting a node at the beginning

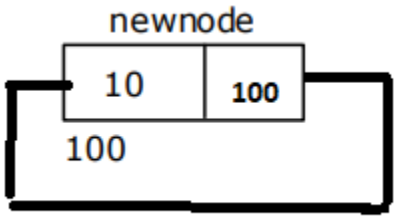
2.Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list

• Create a new node using malloc. newnode= (struct node *) malloc(sizeof(struct node));



Insert data and make next head



• If the list is empty, assign new node as head. if (head==NULL)

head= newnode;

newnode -> next = head;

If the list is not empty follow the steps given below:

temp = head;

while(temp -> next != head)

temp = temp -> next;

temp -> next = newnode;

newnode -> next = head;

Figure shows inserting a node into the circular single linked list at the end.

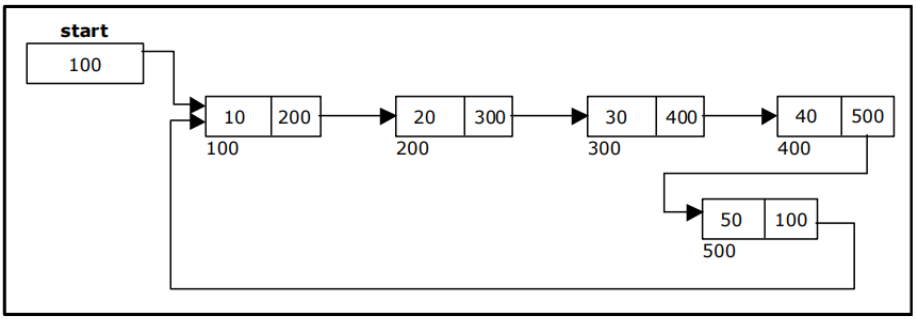


Figure Inserting a node at the end.

3. Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

• If the list is empty, display a message 'Empty List'.

• If the list is not empty, follow the steps given below:

last = temp = head;

while(last -> next != head)

last = last -> next;

head = head -> next;

last -> next = head;

• After deleting the node, if the list is empty then head = NULL.

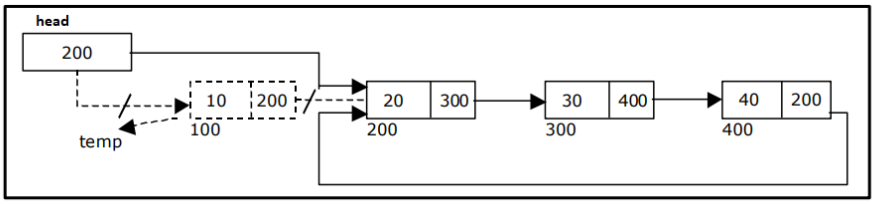


Figure : Deleting a node at beginning.

4. Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

• If the list is empty, display a message 'Empty List'.

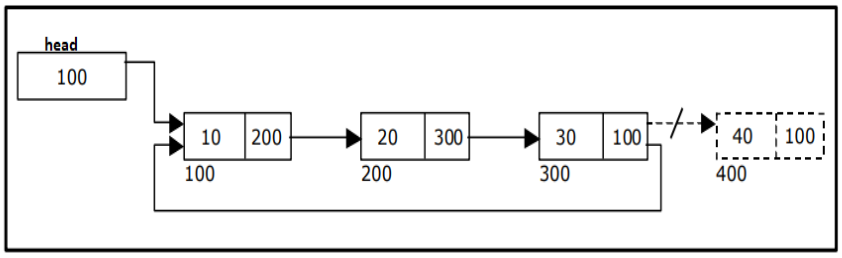
• If the list is not empty, follow the steps given below:

temp = head;

prev = head;

while(temp -> next != head) { prev = temp; temp = temp -> next; } prev -> next = head;

• After deleting the node, if the list is empty then head = NULL.



Deleting a node at the end.

5. Traversing/display a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

• If list is empty then display 'Empty List' message.

• If the list is not empty, follow the steps given below:

temp = head;

do {

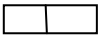
printf("%d ", temp -> data);

```
temp = temp -> next;
 } while(temp != head);
```

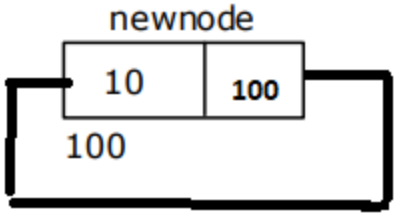
6. Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

• • Create a new node using malloc. newnode= (struct node *) malloc(sizeof(struct node));



Insert data and make next head



• If the list is empty, assign new node as head. if (head==NULL)

head= newnode;

newnode -> next = head;

If the list is not empty follow the steps given below:

• Ensure that the specified position is in between first node and last node.

• Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

```
temp = prev = head;
i=1;
while(i < pos) {
prev = temp; temp = temp -> next;
 i++; }
```

• After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```

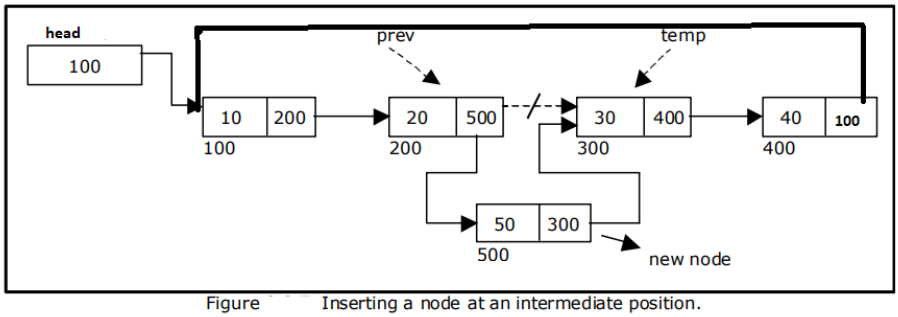


Figure Inserting a node at an intermediate position.

7.Deleting a node at Intermediate position:

The following steps are followed,

- To delete a node from an intermediate position in the list (List must contain more than two node).
- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below.

temp = prev = head;

i=1;

while(i < pos) {

prev = temp; temp = temp -> next; i++; }

prev -> next = temp -> next; free(temp);

}

Pos=2

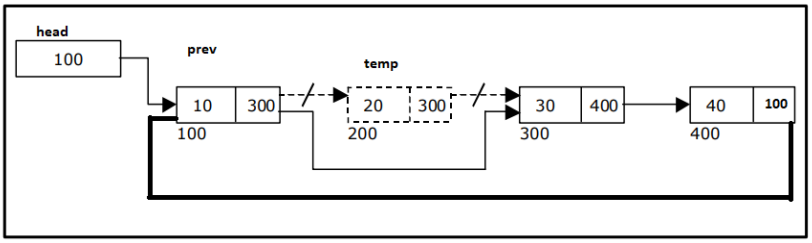


Figure: Deleting 2 node from circular Linked List

Write a c Program to implement circular Linked List and perform insertion and deletion

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=0;
void insert_at_first()
{
```

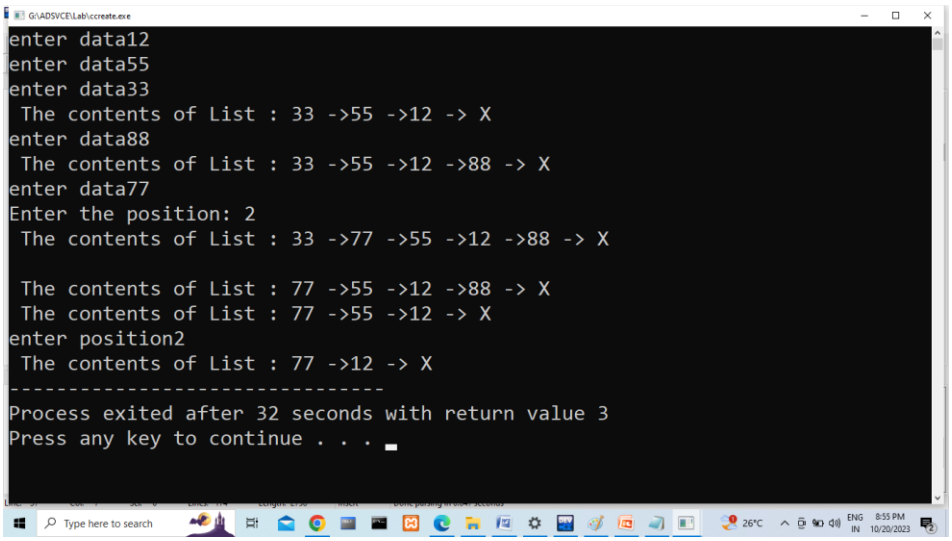
```
struct node *newnode, *temp;
newnode=(struct node *)malloc(sizeof(struct node));
printf("enter data");
scanf("%d",&newnode->data);
if(head == NULL)
 {
 head = newnode;
newnode -> next = head;
 }
 else
 {
  temp = head;
 while(temp -> next != head)
 temp = temp -> next;
 newnode -> next = head;
 head = newnode;
 temp -> next = head;
 }
}
void display()
{
 struct node *temp;
temp = head;
printf("\n The contents of List : ");
if(head == NULL )
 printf("\n Empty List");
 else
 {
do
 {
printf("%d ->", temp -> data);
temp = temp -> next;
} while(temp != head);
 printf(" X ");
  }

}
void insert_last()
{
struct node *newnode, *temp;
newnode=(struct node *)malloc(sizeof(struct node));
printf("enter data");
scanf("%d",&newnode->data);
if(head == NULL )
 {
 head = newnode;

newnode -> next = head;
 }
 else
```

```
 {
 temp = head;
while(temp -> next != head)
temp = temp -> next;
temp -> next = newnode;
newnode -> next = head;
 }
}
void insert_pos()
{
struct node *newnode, *temp, *prev;
int i, pos ;
newnode=(struct node *)malloc(sizeof(struct node));
printf("enter data");
scanf("%d",&newnode->data);
printf("\n Enter the position: ");
 scanf("%d", &pos);
 if(head == NULL )
 {
 head = newnode;
newnode -> next = head;
 }
 else
 {
 temp = head;
 prev = temp;
 i = 1;
 while(i < pos)
 {
 prev = temp;
temp = temp -> next;
 i++;
 }
prev -> next = newnode;
newnode -> next = temp;
}
}
void del_first()
{
struct node *temp, *last;
if(head == NULL)
 printf("\n linked list empty..");
  else
 {
last = temp = head;
while(last -> next != head)
 last = last -> next;
head = head -> next;
last -> next = head;
 free(temp);
```

```
}

}
void delete_last()
{
struct node *temp,*prev;
if(head == NULL)
  printf("\n linked list empty..");
  else
 {
 temp = head;
 prev = head;
while(temp -> next != head)
 {
 prev = temp;
temp = temp -> next;
 }
prev -> next = head;
 free(temp);
}
}
void del_pos()
{
int pos,i;
struct node *prev,*temp;
temp=head;
printf("enter position");
scanf("%d",&pos);
for(i=1;i<pos;i++)
{
prev=temp;
  temp=temp->next;
}
prev->next=temp->next;
free(temp);
}



void main()
{
int n,x,ch;
                insert_at_first();
                insert_at_first();
                insert_at_first();
                display();
                printf("\n");
                insert_last();
                display();
                printf("\n");
```

```
        insert_pos();
        display();
        printf("\n");
        printf("\n");
        del_first();
        display();
        printf("\n");
        delete_last();
        display();
        printf("\n");
        del_pos();
        display();
}
```

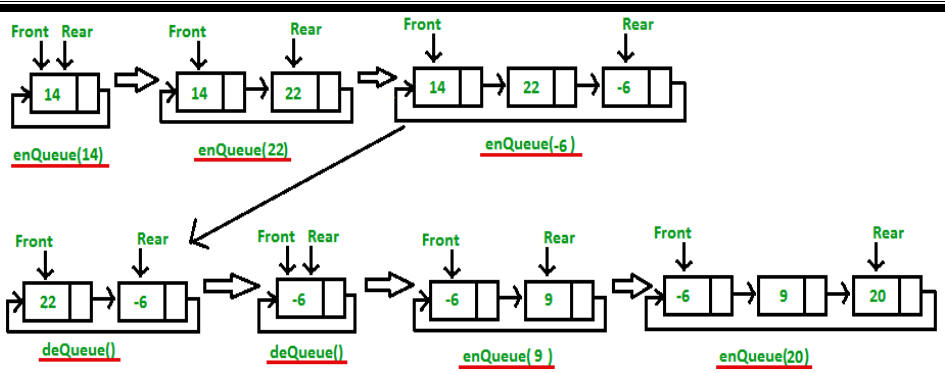
Output:



Circular Queue with Linked List

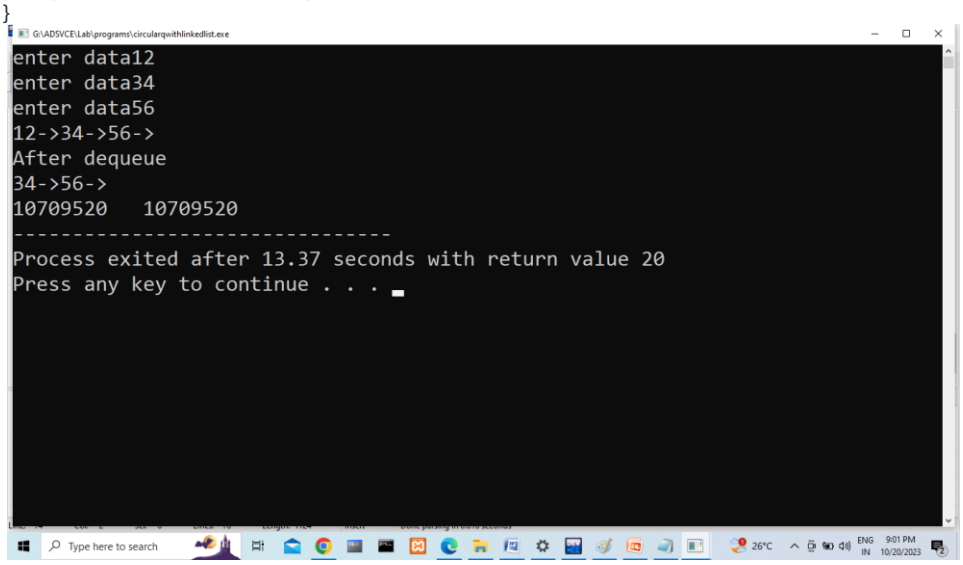
Operations on Circular Queue:

- ***Front:****Get the front item from queue.*
- ***Rear:*** *Get the last item from queue.*
- ***enQueue()*** *This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.*
  1. *Create a new node dynamically and insert value into it.*
  2. *Check if front==NULL, if it is true then front = rear = (newly created node)*
  3. *If it is false then rear=(newly created node) and rear node always contains the address of the front node.*
- ***deQueue()*** *This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.*
  1. *Check whether queue is empty or not means front == NULL.*
  2. *If it is empty then display Queue is empty. If queue is not empty then step 3*
  3. *Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.*




```
   #include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *rear=0,*front=0;
void enqueue()
{
struct node *newnode;
newnode =(struct node*)malloc(sizeof(struct node));
printf("enter data");
scanf("%d",&newnode->data);
newnode->next=0;
if(front==0 && rear==0)
 {
 front=rear=newnode;
  rear->next=front;
}
else
{
rear->next=newnode;
rear=newnode;
rear->next=front;
}
}
void display()
{
struct node *temp;
if(front==0 && rear==0)
 printf("list empty");
 else
 {
 temp=front;
 do
 {
 printf("%d->",temp->data);
 temp=temp->next;
}while(temp!=rear->next);

}
}

void dequeue()
{
struct node *temp;
temp=front;
if(front==0 && rear==0)
  printf("list empty");
  else if(front==rear){
     free(temp);
       front=rear==0;}
       else
       {
         front=front->next;
         rear->next=front;
```

```
        free(temp);
    }
  }


void main()
{
enqueue();
enqueue();
enqueue();
display();
printf("\n");
dequeue();
printf("After dequeue\n");
display();
printf("\n%d   %d",front,rear->next);
}
```



Applications of Circular Queue

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Double Ended Queue

Operations on Deque :

Mainly the following four basic operations are performed on queue :

`insertFront()` : Adds an item at the front of Deque.
`insertRear()` : Adds an item at the rear of Deque.
`deleteFront()` : Deletes an item from front of Deque.
`deleteRear()` : Deletes an item from rear of Deque.

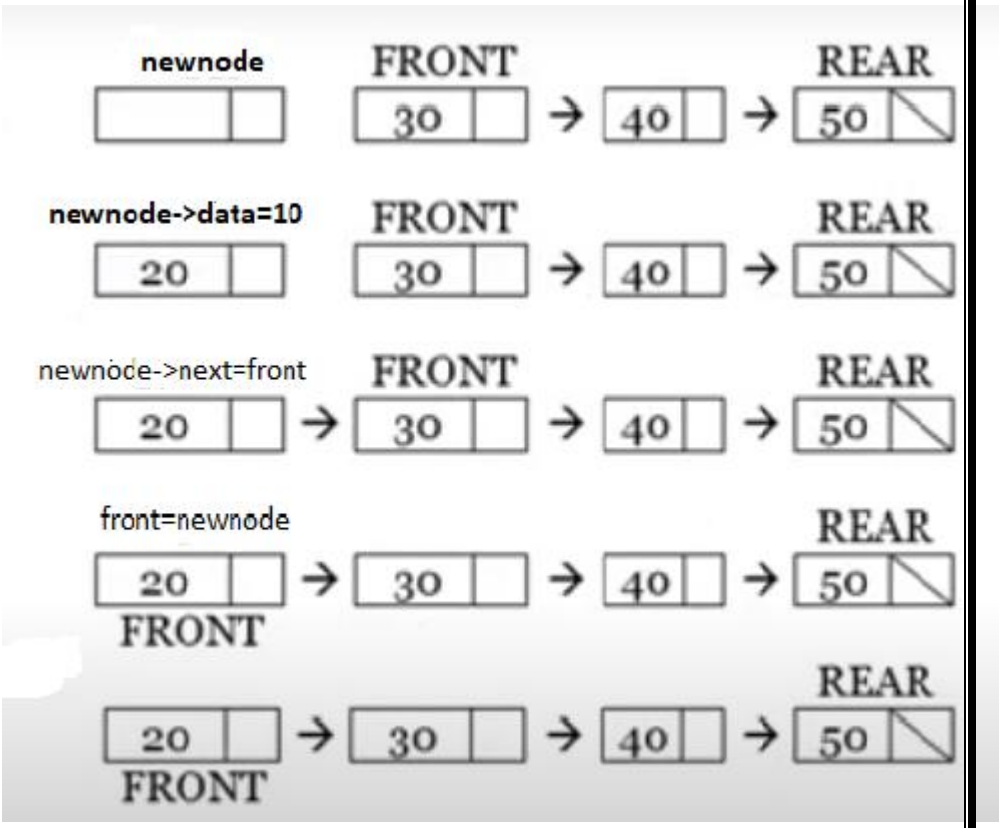
1.enqueuefront

```
void enqueuefront()

{

struct node *newnode;

newnode=(struct node *)malloc(sizeof(struct node));

printf("\n enter data:");

scanf("%d",&newnode->data);

newnode->next=0;

if(front==0 && rear==0)

  front=rear=newnode;

  else

    {

      newnode->next=front;

      front=newnode;

    }

}
```

2.dequeue front

```
void dequeuefront()

{

struct node *temp;

temp=front;

if(front==0 && rear==0)

   printf("de q empty");

   else if(front==rear)

       front=rear=0;
```

```
else

{

front=front->next;

free(temp);}


}
```



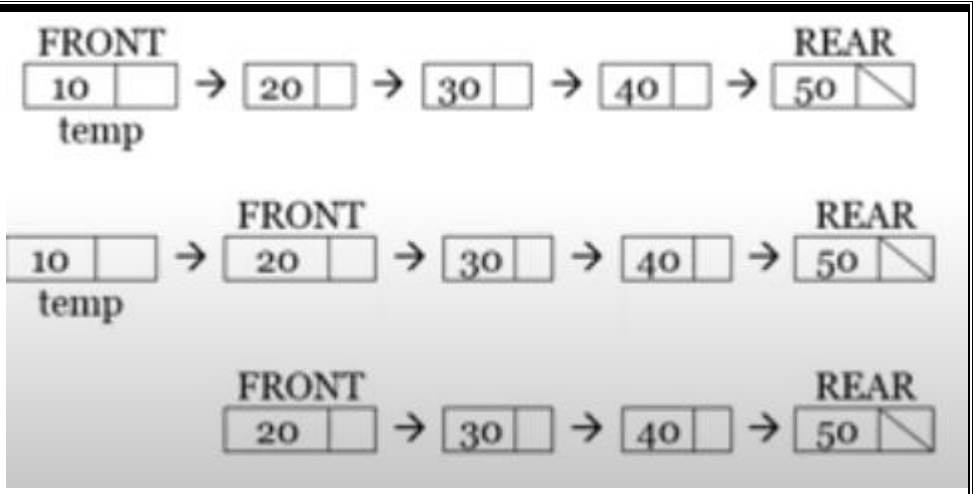
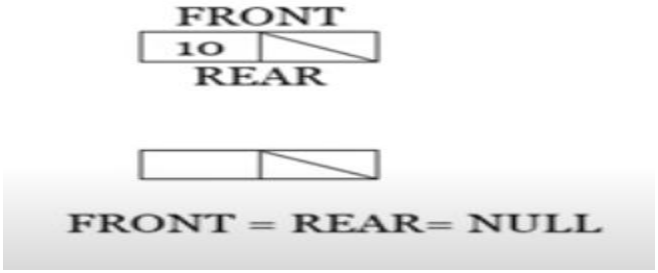


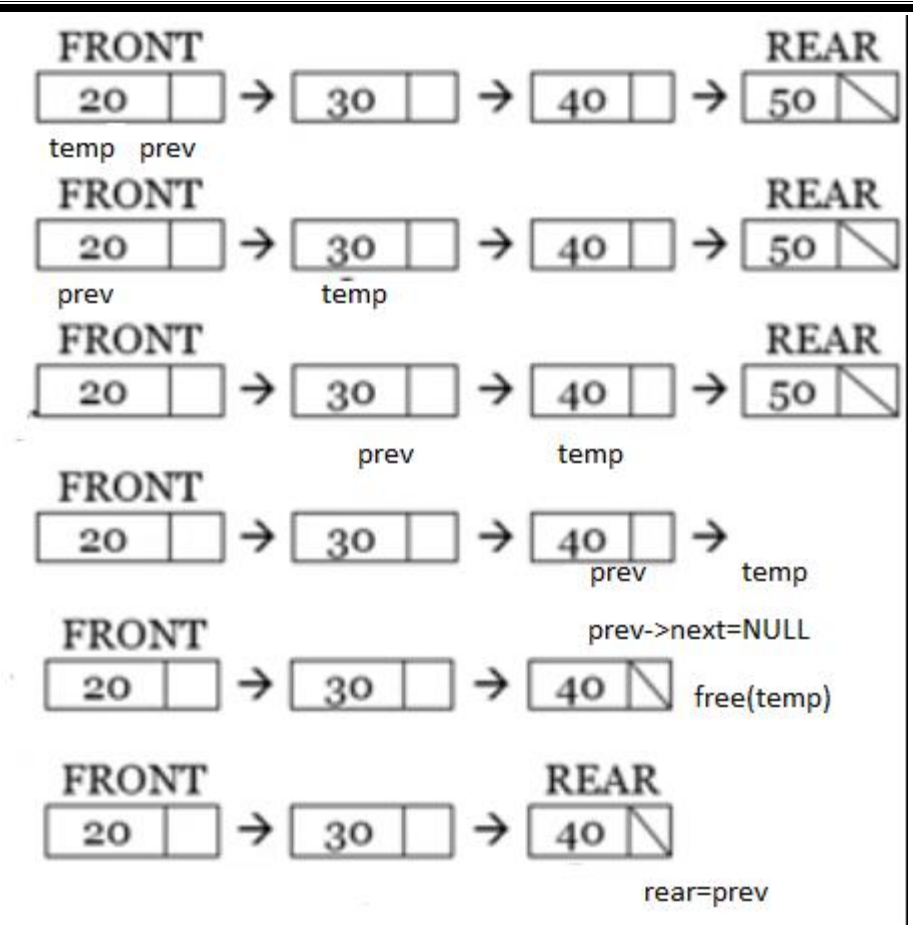
Figure :Dequeue front

```
3.enqueuerear
void enqueuerear()
{
struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
printf("\n enter data:");
scanf("%d",&newnode->data);
newnode->next=0;
if(front==0 && rear==0)
   front=rear=newnode;
   else
   {
    rear->next=newnode;
    rear=newnode;
}
}
```

```
4.dequeuerear
    void dequeuerear()

    {

    struct node *temp,*prev;

    temp=front;

    if(front==0 && rear==0)

      printf("de q empty");
```

```
    else if(front==rear)
       front=rear=0;
       else
       {
        while(temp->next!=NULL){
          prev=temp;
          temp=temp->next;}
          prev->next=NULL;
          rear=prev;
          free(temp);
       }
}
```






```
oid display()
{
struct node *temp;
if(front==0 && rear==0)
   printf("de q empty");
   else
   {
   temp=front;
    while(temp!=NULL){
     printf("%d->",temp->data);
     temp=temp->next;}
 }
}
```

Write a C program to implement deueue operations with linked list.

```
#include<stdio.h>

#include<stdlib.h>
```

```
struct node
{
int data;
struct node *next;
};
struct node *front=0;
struct node *rear=0;
void enqueuerear()
{
struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
printf("\n enter data:");
scanf("%d",&newnode->data);
newnode->next=0;
if(front==0 && rear==0)
  front=rear=newnode;
  else
  {
   rear->next=newnode;
   rear=newnode;
}
}
void enqueuefront()
{
struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
```

```
printf("\n enter data:");
scanf("%d",&newnode->data);
newnode->next=0;
if(front==0 && rear==0)
  front=rear=newnode;
  else
    {
      newnode->next=front;
      front=newnode;
   }
}
void dequeuerear()
{
struct node *temp,*prev;
temp=front;
if(front==0 && rear==0)
  printf("de q empty");
  else if(front==rear)
     front=rear=0;
     else
     {
      while(temp->next!=NULL){
        prev=temp;
         temp=temp->next;}
         prev->next=NULL;
         rear=prev;
```

```
            free(temp);
        }

}
void dequeuefront()
{
struct node *temp;
temp=front;
if(front==0 && rear==0)
   printf("de q empty");
   else if(front==rear)
       front=rear=0;
else
{
front=front->next;
free(temp);}

}
void display()
{
struct node *temp;
if(front==0 && rear==0)
   printf("de q empty");
   else
   {
   temp=front;
```

```
    while(temp!=NULL){
     printf("%d->",temp->data);
     temp=temp->next;}
  }
}
void main()
{
int ch;
while(1)
{
printf("1:enqueuerear 2:enqueuefront 3:dequeuerear 4:dequeuefront\n");
printf("5:display 6:exit\n");
printf("enter your choice");
scanf("%d",&ch);
switch(ch)
{
case 1: enqueuerear();break;
case 2:enqueuefront();break;
case 3:dequeuerear();break;
case 4:dequeuefront();break;
case 5:display();break;
case 6:exit(0);
}
}
}
```

Write a C program to implement dequeue operations using double linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
struct node *prev;
};
struct node *front=0;
struct node *rear=0;
void enqueuerear()
{
struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
printf("\n enter data:");
scanf("%d",&newnode->data);
newnode->next=0;
newnode->prev=0;
if(front==0 && rear==0)
  front=rear=newnode;
    else
  {
   rear->next=newnode;
   newnode->prev=rear;
   rear=newnode;
}
}
void enqueuefront()
{
struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
printf("\n enter data:");
scanf("%d",&newnode->data);
newnode->prev=0;
newnode->next=0;
if(front==0 && rear==0)
  front=rear=newnode;
  else
    {
      newnode->next=front;
      front->prev=newnode;
      front=newnode;
   }
}
void dequeuerear()
{
struct node *temp,*prev;
```

```
temp=rear;
if(front==0 && rear==0)
   printf("de q empty");
   else if(front==rear)
      front=rear=0;
      else
      {
         rear=rear->prev;
         rear->next=0;
         free(temp);
      }

}
void dequeuefront()
{
struct node *temp;
temp=front;
if(front==0 && rear==0)
   printf("de q empty");
   else if(front==rear)
      front=rear=0;
else
{
front=front->next;
front->prev=0;
free(temp);
}
}
void displayforward()
{
struct node *temp;
if(front==0 && rear==0)
   printf("de q empty");
   else
   {
   temp=front;
    while(temp!=NULL){
     printf("%d->",temp->data);
     temp=temp->next;}
 }
}
void displaybackward()
{
struct node *temp;
if(front==0 && rear==0)
   printf("de q empty");
   else
   {
   temp=rear;
    while(temp!=NULL){
```

```
    printf("%d->",temp->data);
    temp=temp->prev;}
 }
}
 void main()
 {
 int ch;
 while(1)
 {
 printf("\n1:enqueuerear 2:enqueuefront 3:dequeuerear 4:dequeuefront\n");
 printf("5:displayforward 6:displaybackword  7:exit\n");
 printf("enter your choice:");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1: enqueuerear();break;
 case 2:enqueuefront();break;
 case 3:dequeuerear();break;
 case 4:dequeuefront();break;
 case 5:displayforward();break;
 case 6:displaybackward();break;
 case 7:exit(0);
}
}
}
```

Prority Queue Using Linked List

- Priority queue in a data structure is an extension of a linear queue that possesses the following properties:
- Every element has a certain priority assigned to it.
- Every element of this queue must be comparable.
- It will delete the element with higher priority before the element with lower priority.
- If multiple elements have the same priority, it does their removal from the queue according to the FCFS principle.

1. insert_with_priority:

- add an element to the queue with an associated priority.

2. pull_highest_priority_element:

- remove the element from the queue that has the highest priority

Syntax for Priority Queue as Linked List

```
struct Node
{
        int data;
```

```
        int priority; // lower value means higher priority
        struct Node* next;
};
```

Write a C program to implement prority queue using liked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
int priority;
struct node *next;
};
struct node *front=0;
// insert method
void insert()
{
   struct node *temp,*newnode;

   newnode = (struct node *)malloc(sizeof(struct node));
   printf("enter data and prority");
   scanf("%d%d",&newnode->data,&newnode->priority );

   // condition to check whether the first element is empty or the element to be inserted has more priority
than the first element
   if( front == NULL || newnode->priority > front->priority )
   {
      newnode->next = front;
      front = newnode;
   }
   else
   {
      temp = front;
      while( temp->next != NULL && temp->next->priority > newnode->priority )
         temp=temp->next;
      newnode->next = temp->next;
      temp->next = newnode;
   }
}
void display()
{
struct node *temp;
if(front==0)
 printf("list empty");
 else
```

```
 {
 temp=front;
 do
 {
 printf("%d|%d->",temp->data,temp->priority);
 temp=temp->next;
   }while(temp!=0);
 }
}

void dequeue()
{
struct node *temp;
temp=front;
if(front==0 )
  printf("list empty");
      else
      {
        front=front->next;
              free(temp);
     }
  }

void main()
{
insert();
insert();
insert();
display();
printf("\n");
dequeue();
printf("After dequeue\n");
display();
insert();
display();
}
```

Applications of Priority Queue in Data Structure

- IP Routing to Find Open Shortest Path First: OSPF is a link-state routing protocol that is used to find the best path between the source and the destination router. This protocol works on the principle of Dijkstra's shortest path algorithm by using a priority queue to track an optimal route.
- Data Compression in WINZIP / GZIP: The Huffman encoding algorithm uses a priority queue to maintain the codes for data contents. They store these codes in a min heap, considering the size of codes as a parameter to decide the priority.
- Used in implementing Prim's algorithm: Prim's algorithm generates a minimum spanning tree from an undirected, connected, and weighted graph. It uses a min priority queue to maintain the order of elements for generating a minimum spanning tree.
- Used to perform the heap sort: When you provide an unsorted array to this algorithm, it converts it into a sorted array. This algorithm uses a min priority queue to generate an order of elements.
- Load balancing and Interrupt handling: Load balancing fields the requests from the users and distributes them to different available servers. Whereas, interrupt handling is a mechanism for handling interrupts in OS. The priority queue is used to manage requests, and it interrupts both functionalities simultaneously.

NON LINEAR DATA STRUCTURES

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. Tree is a very popular non-linear data structure used in wide range of applications. A tree data structure can be defined as follows...

Trees Basic Concepts:

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

Tree data structure is a collection of data (Node) which is organized in **hierarchical structure** recursively

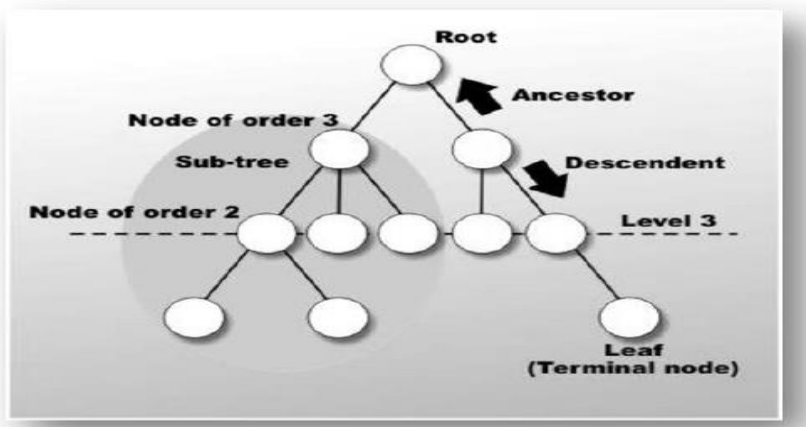
A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

• If T is not empty, T has a special tree called the **root** that has no parent.

• Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.

Tree nodes have many useful properties.

- The **depth** of a node is the length of the path (or the number of edges) from the root to that node.
- The **height** of a node is the longest path from that node to its leafs. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.
- In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

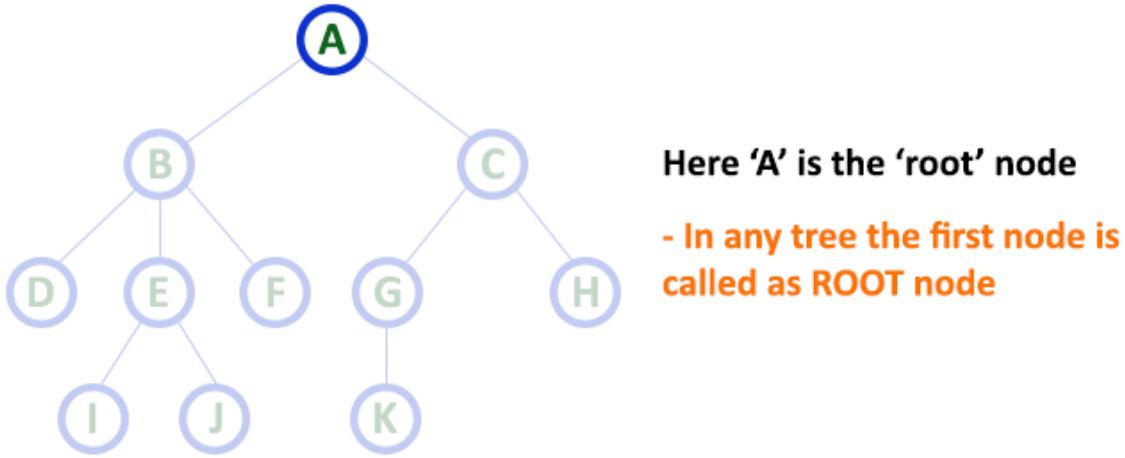


Terminology

In a tree data structure, we use the following terminology...

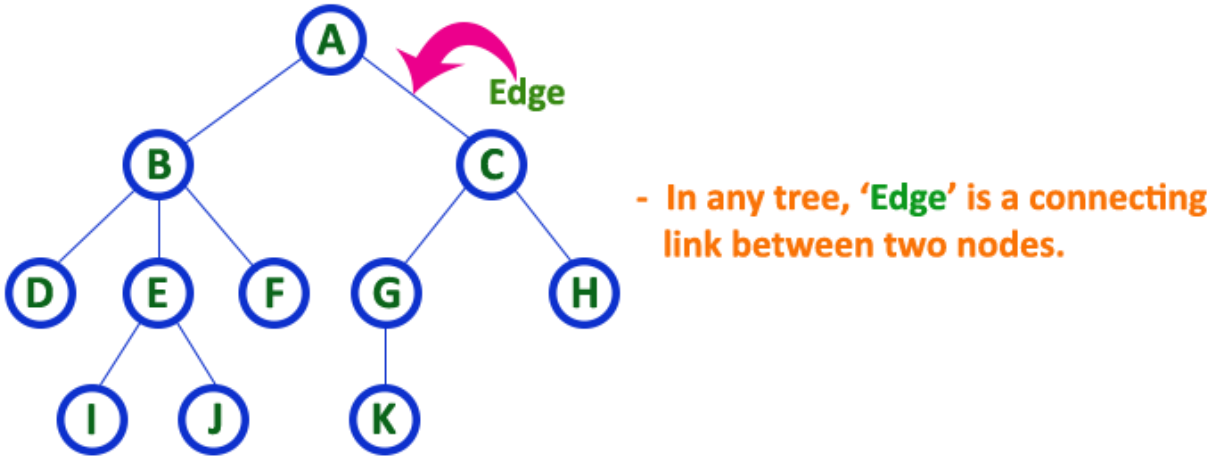
1. Root

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



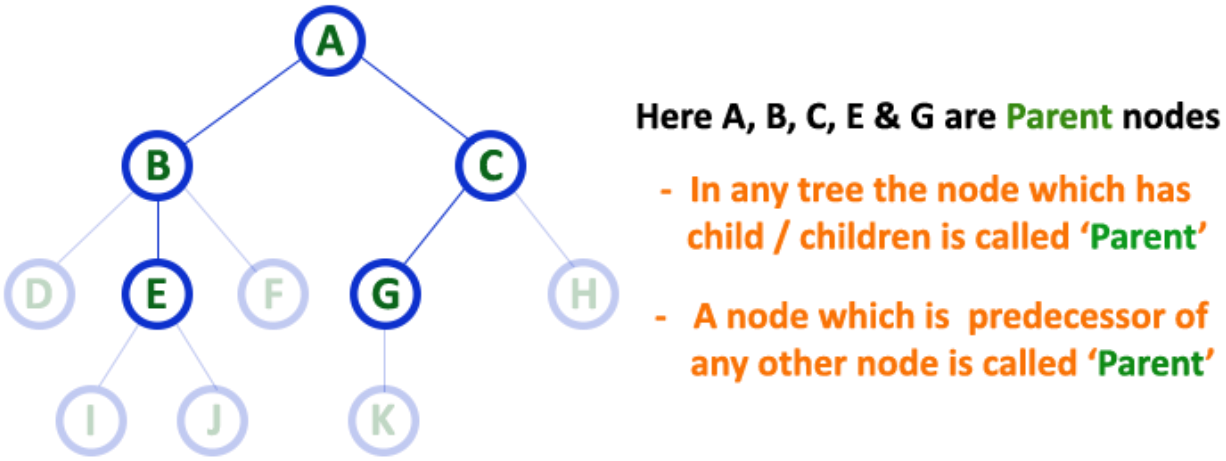
2. Edge

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



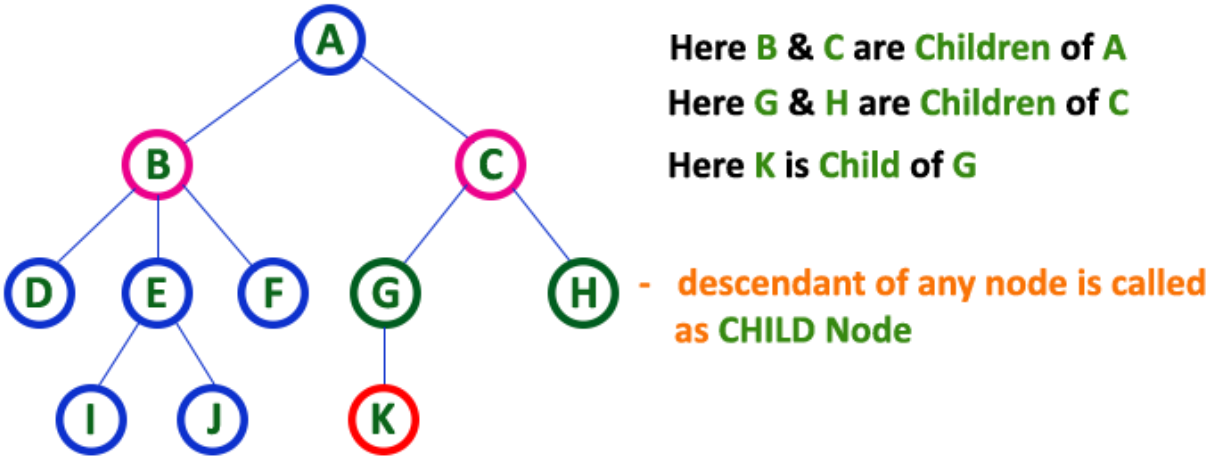
3. Parent

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



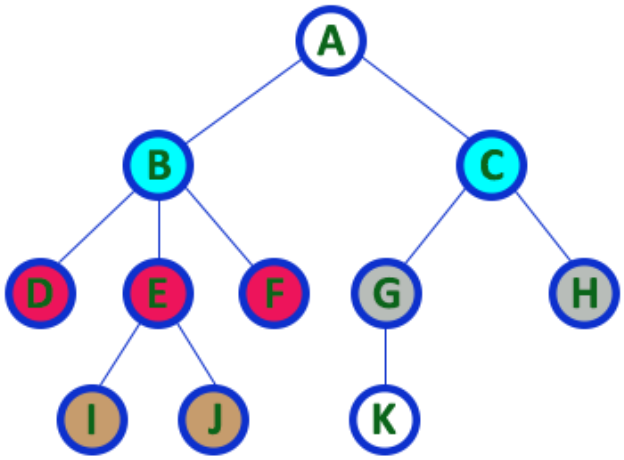
4. Child

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



5. Siblings

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



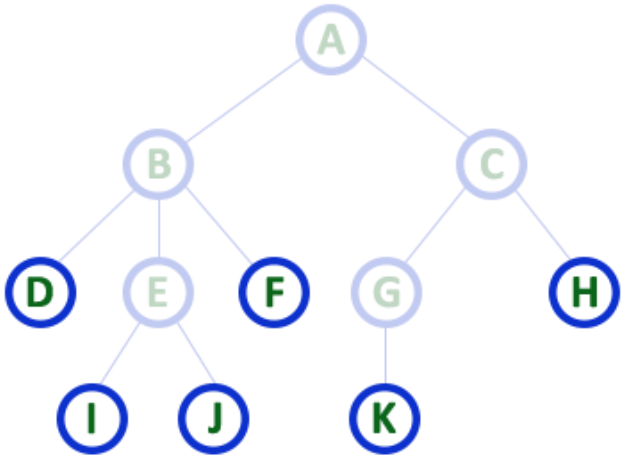
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



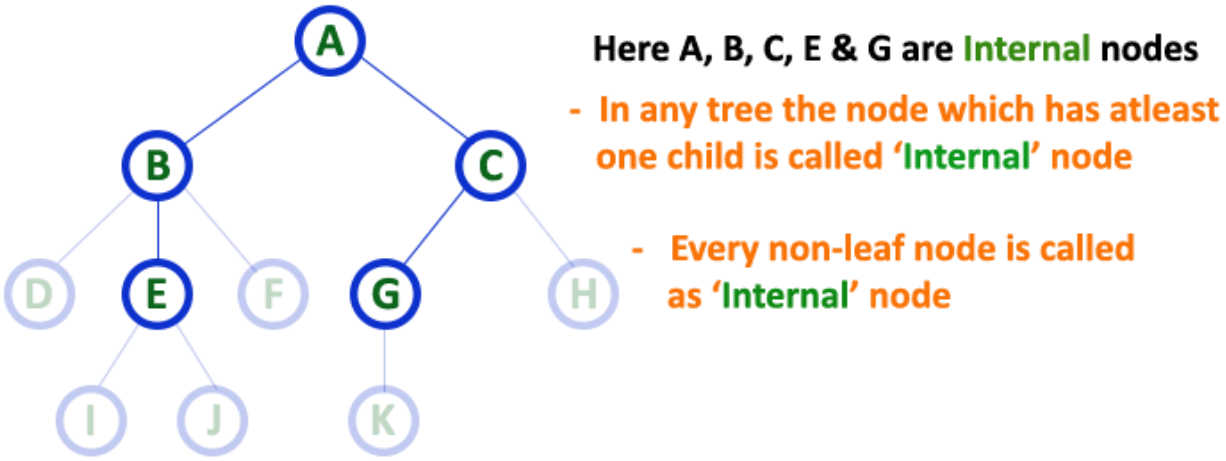
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

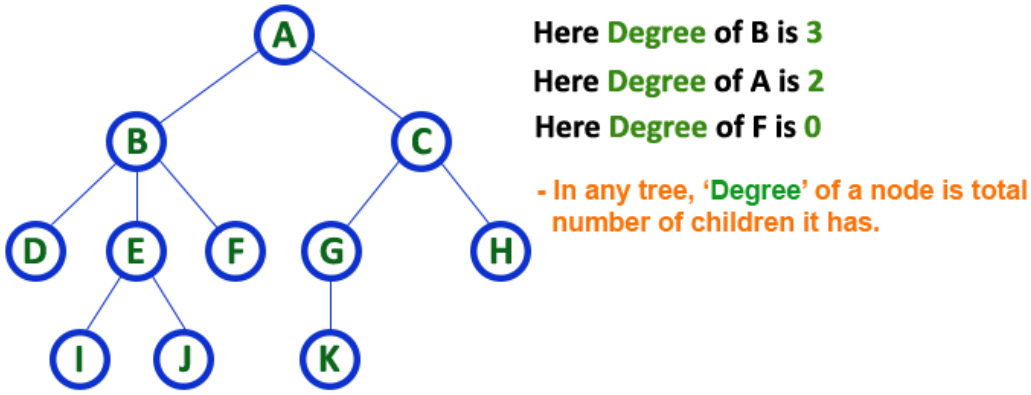
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



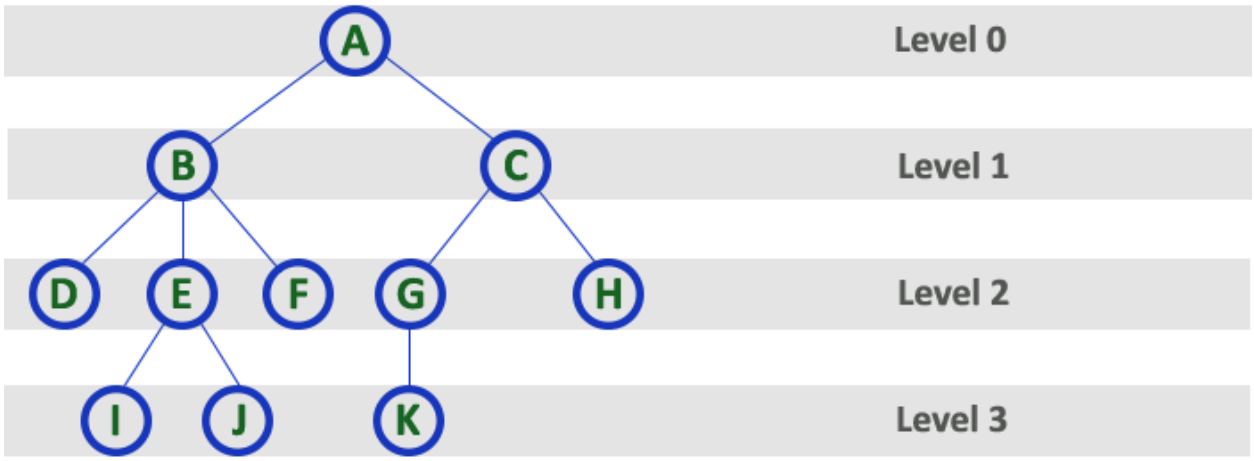
8. Degree

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



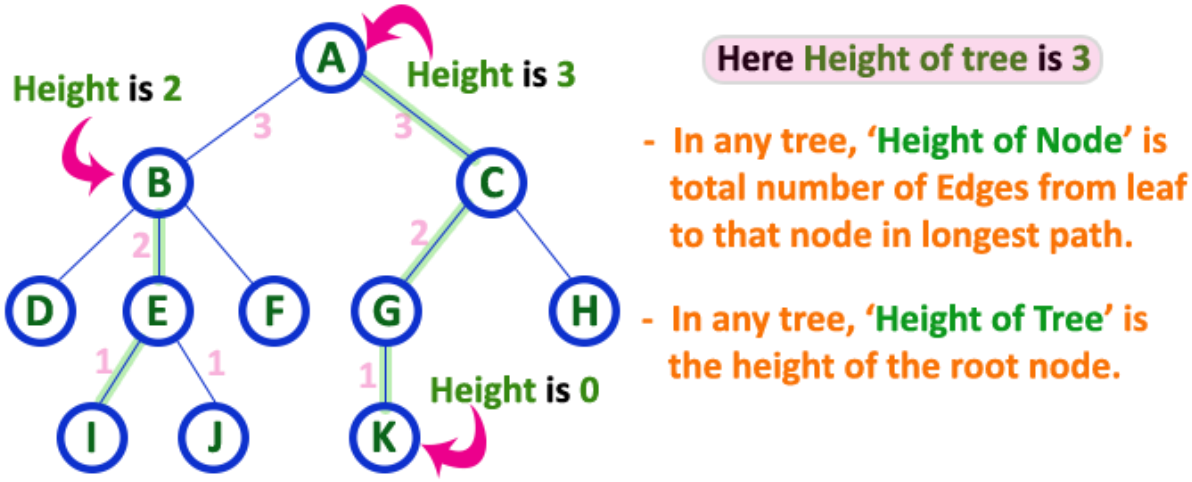
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



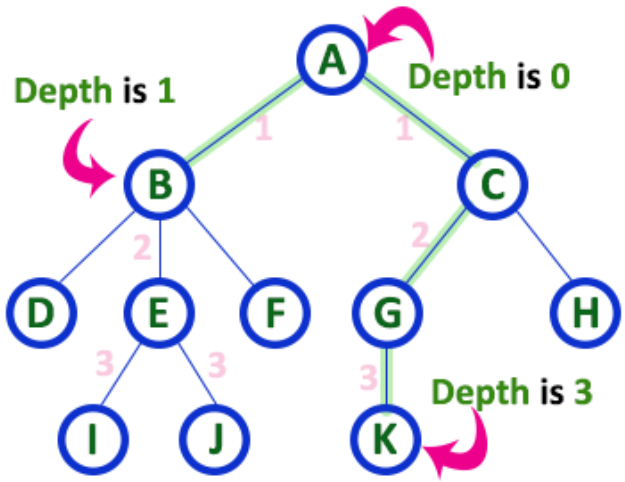
10. Height

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



11. Depth

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

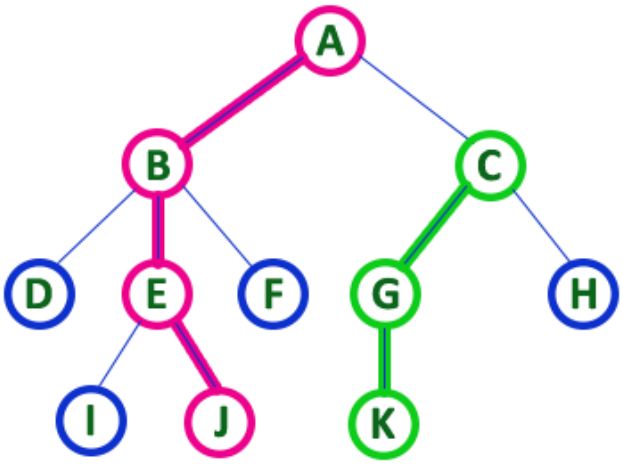


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

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



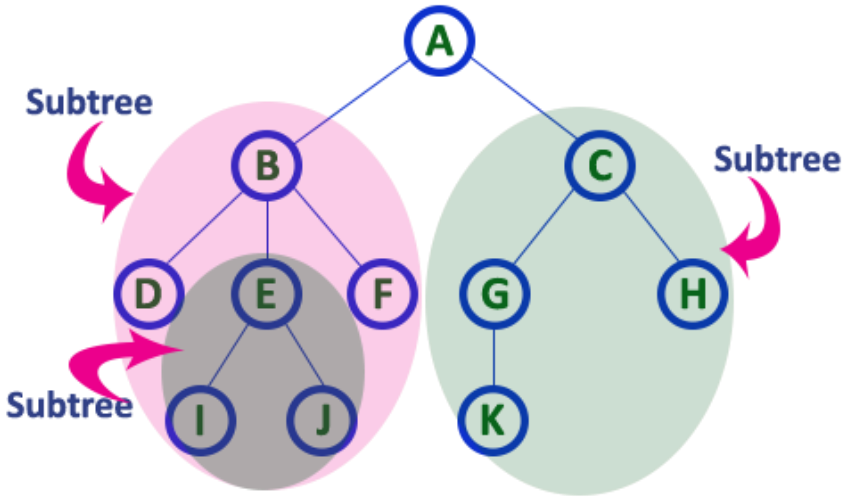
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

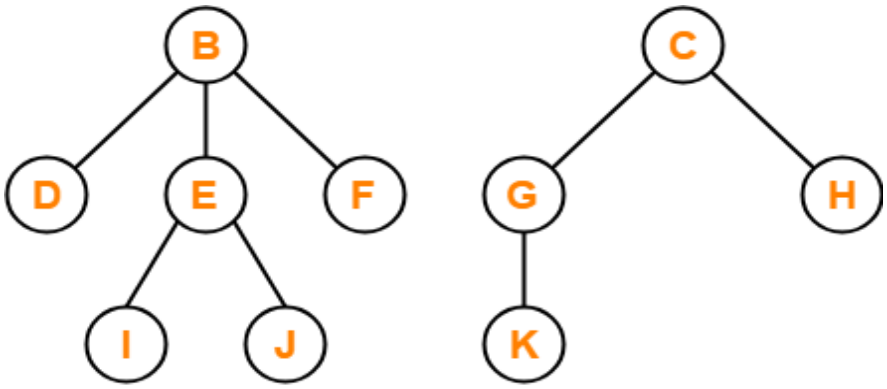
13. Sub Tree

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



14. Forest-

A forest is a set of disjoint trees



Forest

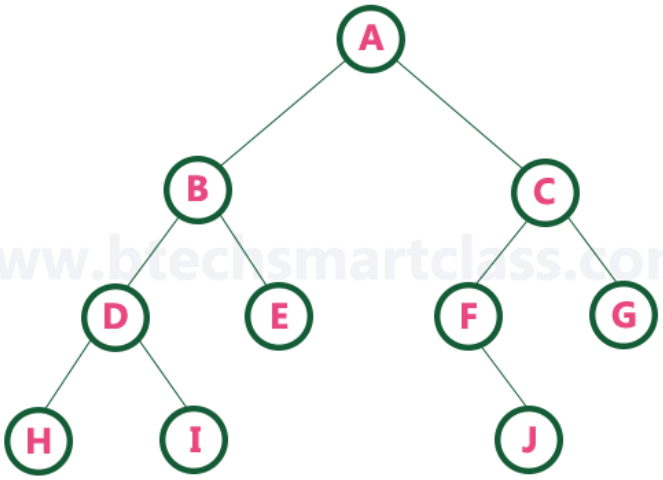
Ancestor and Descendent If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Binary Tree Data structure

- Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

- Example



The maximum number of nodes at any level is 2^n.

Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

    a. Maximum number of leafs = 2^h

    b. Maximum number of nodes = $2^{h+1} - 1$

2. If a binary tree contains m nodes at level l, it contains at most 2^m nodes at level l + 1.

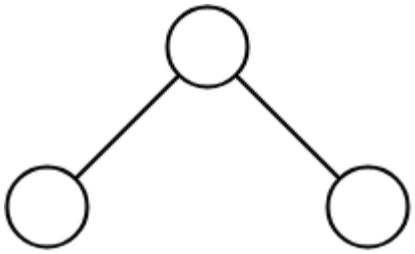
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.

4. The **total number of edges** in a full binary tree **with n node is n - 1.**

There are different types of binary trees and they are...

Unlabeled Binary Tree-

A binary tree is unlabeled if its nodes are not assigned any label.



Unlabeled Binary Tree

$$\text{Number of different Binary Trees possible with 'n' unlabeled nodes} = \frac{{}^{2n}C_n}{n+1}$$

Example-

Consider we want to draw all the binary trees possible with 3 unlabeled nodes.

Using the above formula, we have-

Number of binary trees possible with 3 unlabeled nodes

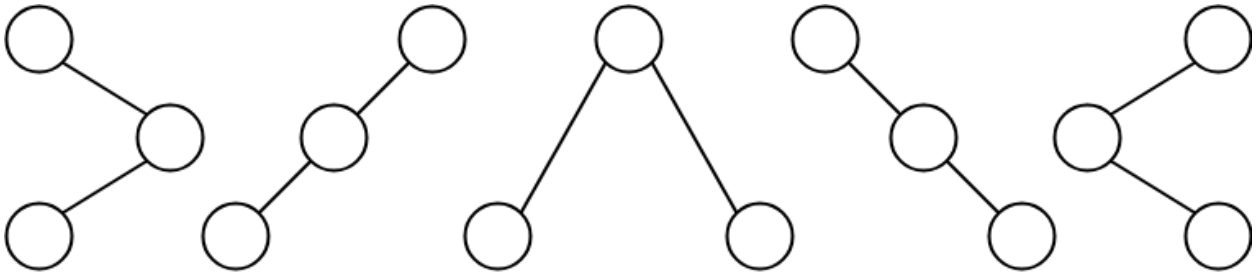
$= {}^{2 \times 3}C_3 / (3 + 1)$

$= {}^{6}C_3 / 4$

$= 5$

Thus,

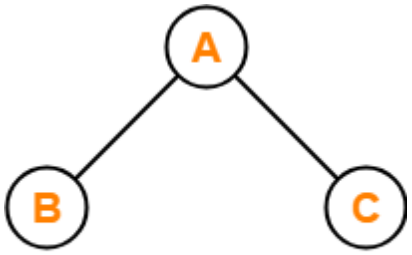
- With 3 unlabeled nodes, 5 unlabeled binary trees are possible.
- These unlabeled binary trees are as follows-



Binary Trees Possible With 3 Unlabeled Nodes

Labeled Binary Tree-

A binary tree is labeled if all its nodes are assigned a label.



Labeled Binary Tree

Number of different Binary Trees possible with 'n' labeled nodes

$$= \frac{{}^{2n}C_n}{n+1} \times n!$$

Example-

Consider we want to draw all the binary trees possible with 3 labeled nodes.

Using the above formula, we have-

Number of binary trees possible with 3 labeled nodes

$= \{ {}^{2 \times 3}C_3 / (3 + 1) \} \times 3!$

$= \{ {}^{6}C_3 / 4 \} \times 6$

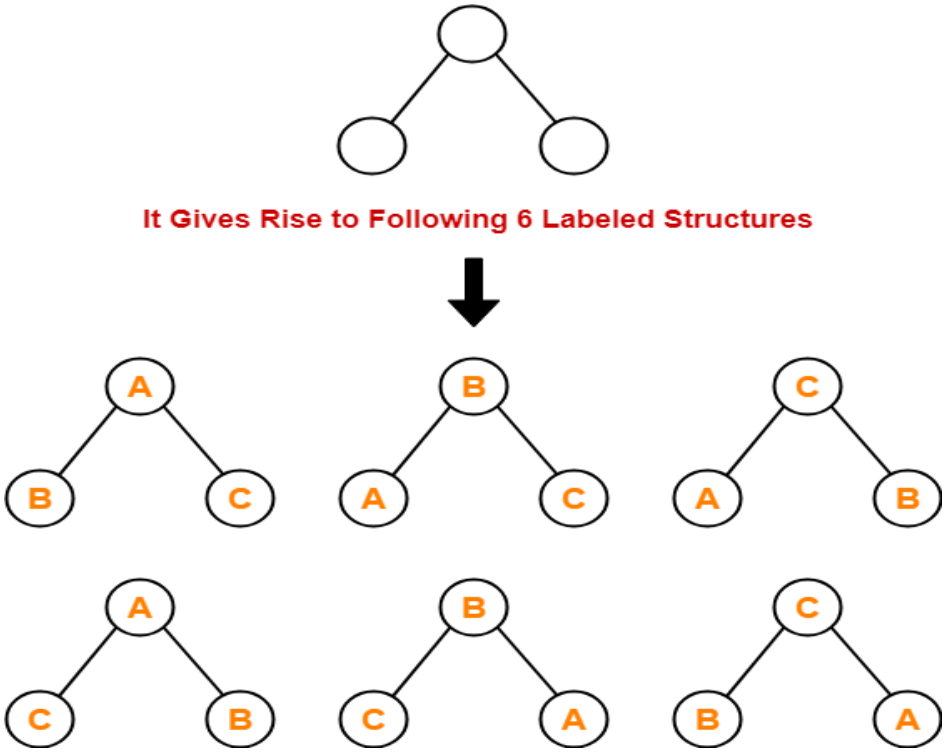
$= 5 \times 6$

$= 30$

Thus,

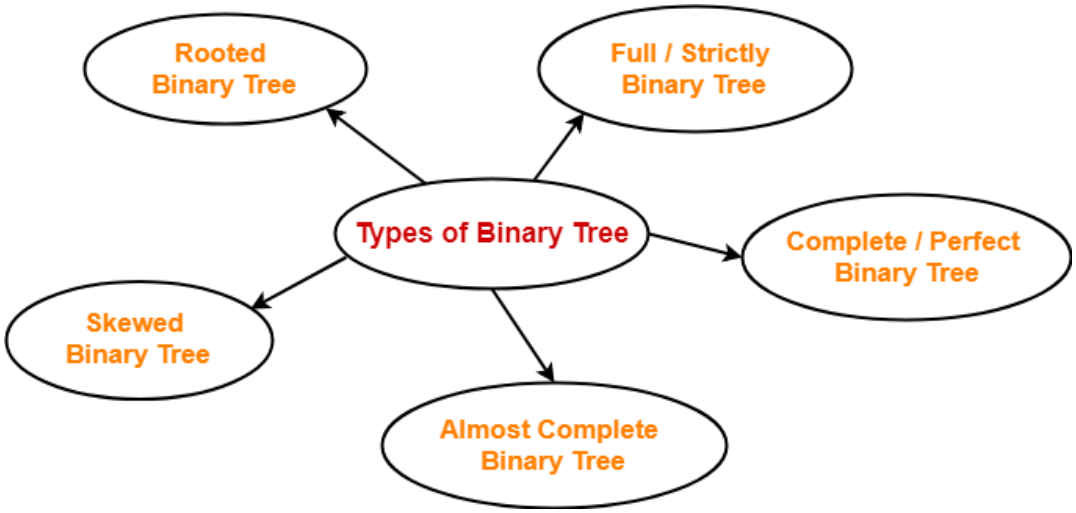
- With 3 labeled nodes, 30 labeled binary trees are possible.

- Each unlabeled structure gives rise to 3! = 6 different labeled structures.



Types of Binary Trees-

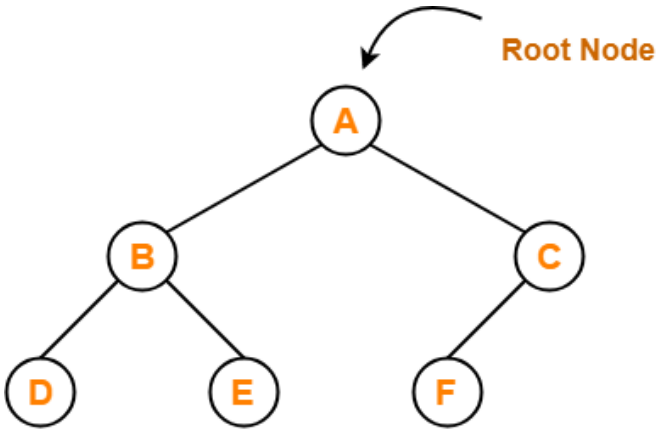
Binary trees can be of the following types-



1. Rooted Binary Tree-

A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

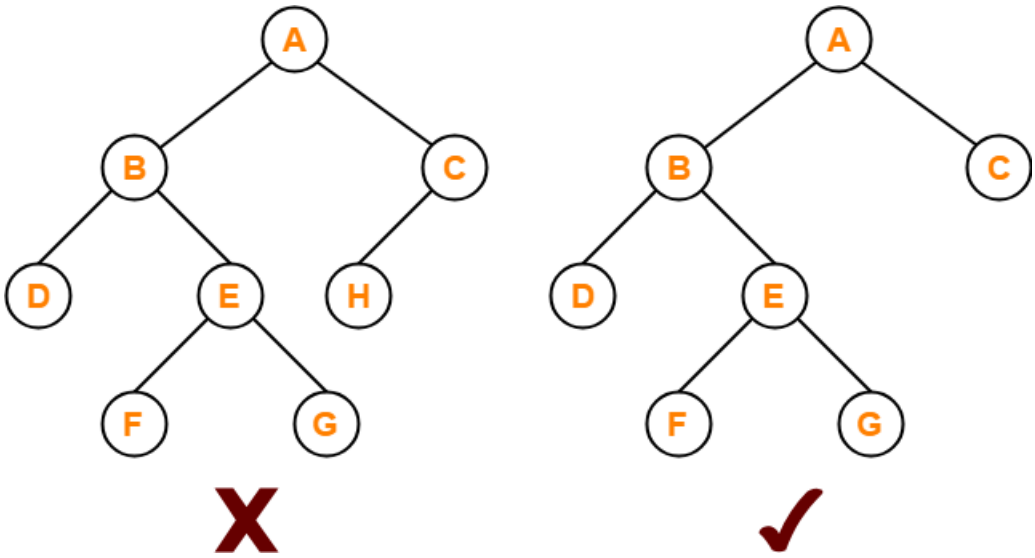
- It has a root node.
- Each node has at most 2 children.



Rooted Binary Tree

2. Full / Strictly Binary Tree-

- A binary tree in which every node has ***either 0 or 2 children*** is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree** or **Proper Binary Tree** or **2-Tree**



Here,

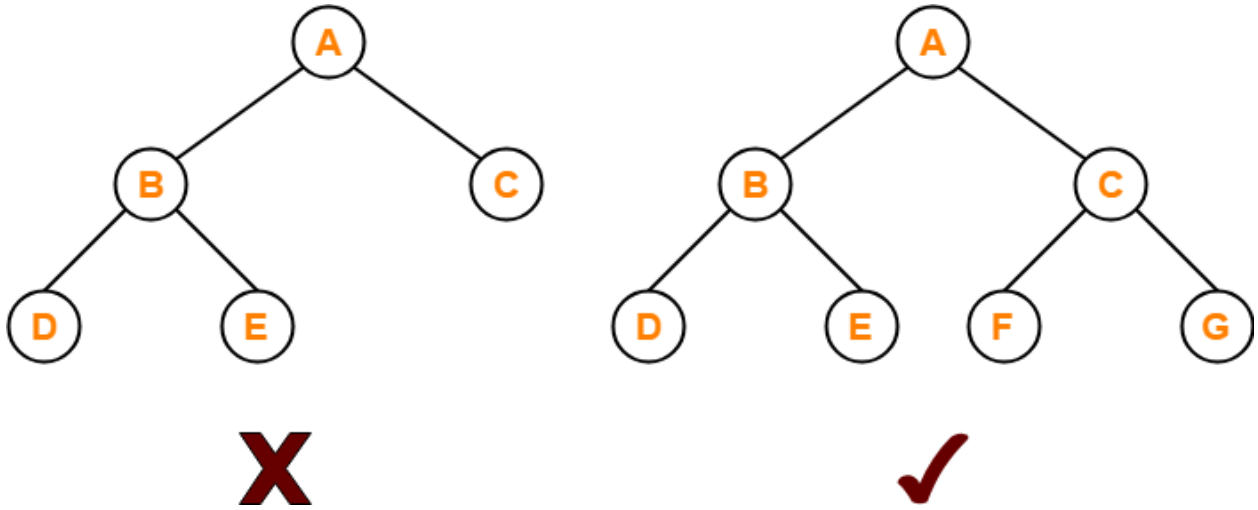
- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

3. Complete / Perfect Binary Tree-

A **complete binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has ***exactly 2 children.***
- All the leaf nodes are at the same level.

Complete binary tree is also called as **Perfect binary tree**.



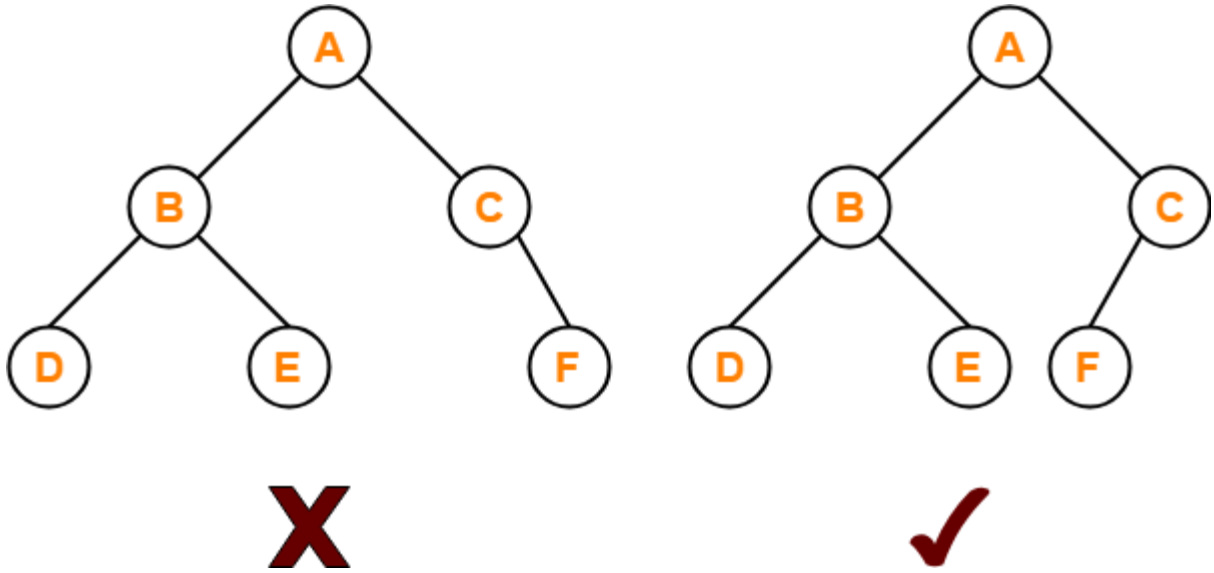
Here,

- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

4. Almost Complete Binary Tree-

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly ***filled from left to right.***



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

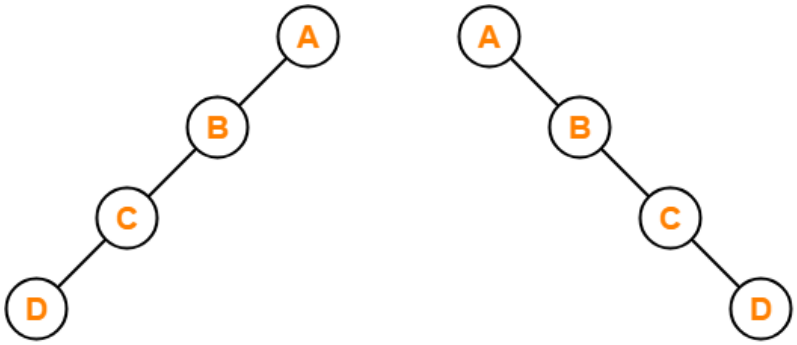
5. Skewed Binary Tree-

A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



Left Skewed Binary Tree Right Skewed Binary Tree

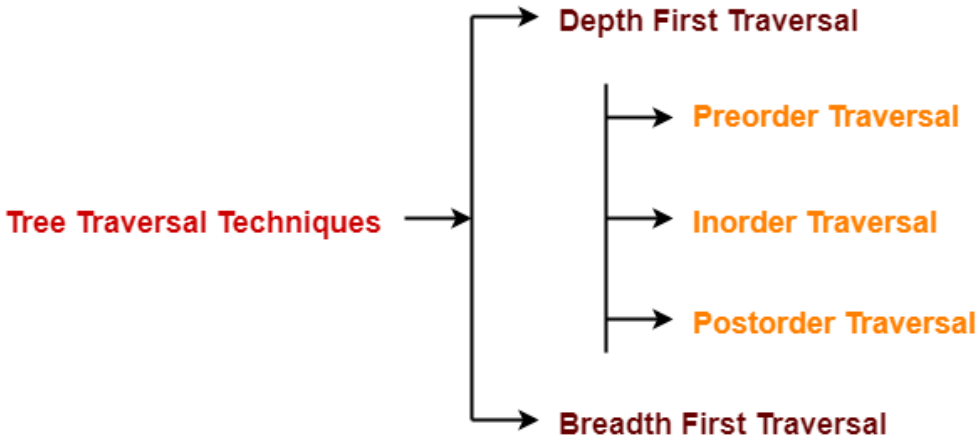
Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

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

Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

Various tree traversal techniques are-



Depth First Traversal-

Following three traversal techniques fall under Depth First Traversal-

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

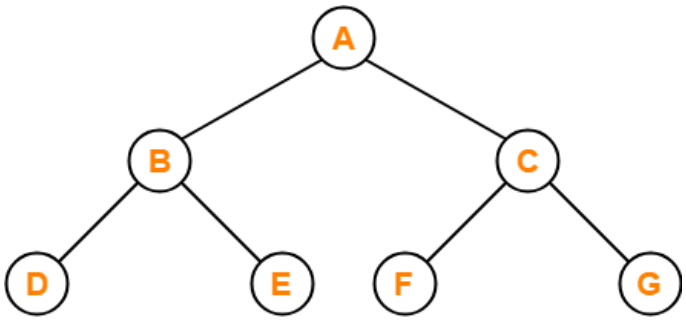
1. Preorder Traversal-

Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

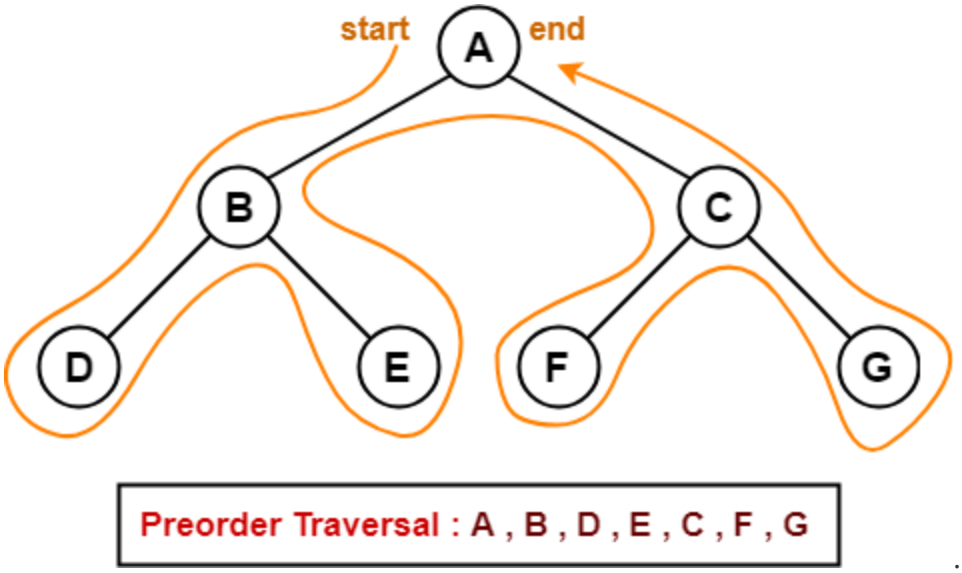
Consider the following example-



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut

Traverse the entire tree starting from the root node keeping yourself to the left.



Applications-

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

2. Inorder Traversal-

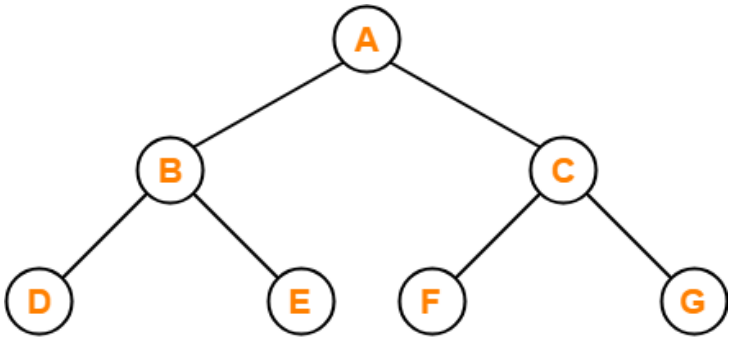
Algorithm-

1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example-

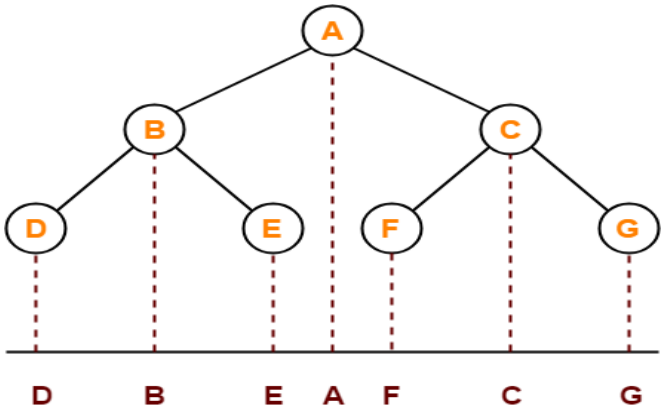
Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

Application-

- Inorder traversal is used to get infix expression of an expression tree.

3. Postorder Traversal-

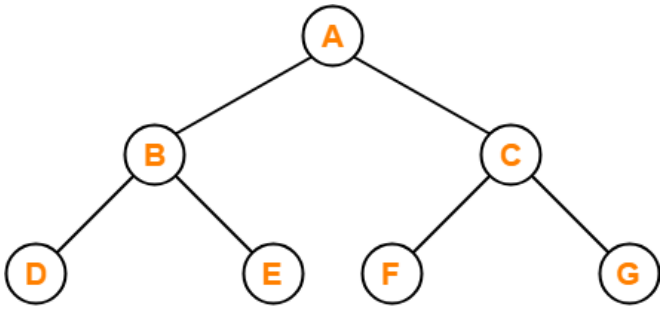
Algorithm

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example-

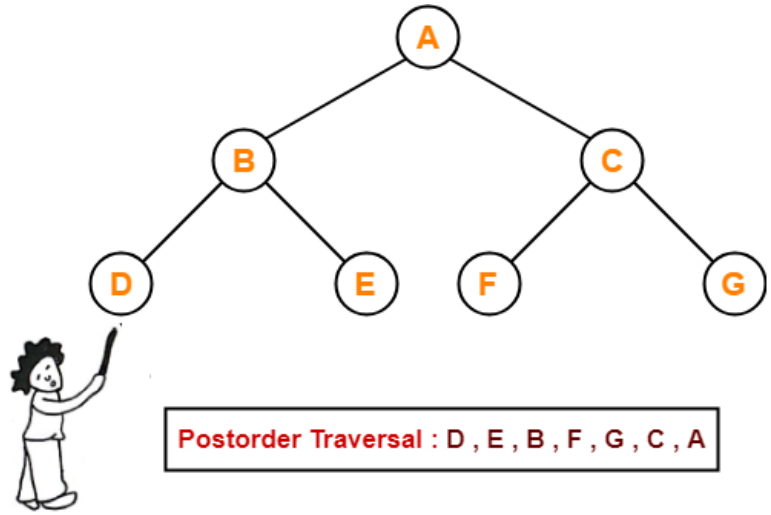
Consider the following example-



Postorder Traversal : D , E , B , F , G , C , A

Postorder Traversal Shortcut

Pluck all the leftmost leaf nodes one by one.



Postorder Traversal : D , E , B , F , G , C , A

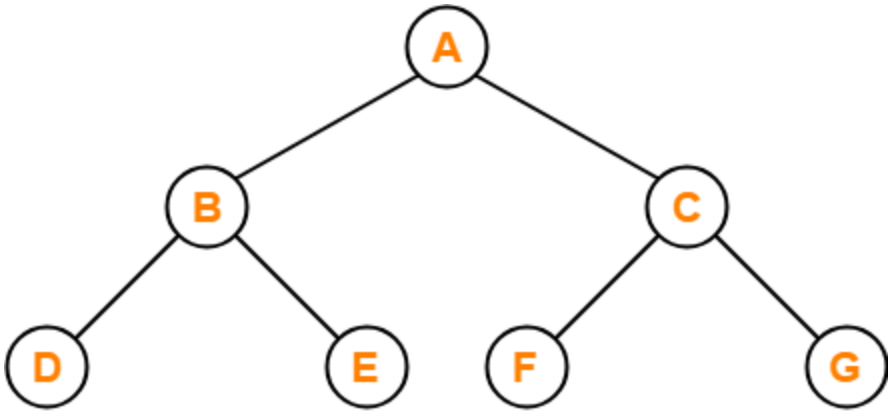
Applications-

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

Breadth First Traversal-

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.

Example-

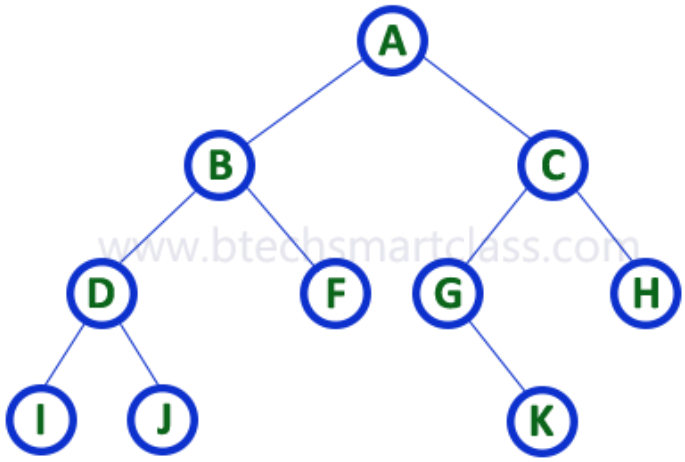


Level Order Traversal : A , B , C , D , E , F , G

Application-

- Level order traversal is used to print the data in the same order as stored in the array representation of a complete binary tree.

Consider the following binary tree...



In-Order Traversal for above example of binary tree is

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

Pre-Order Traversal for above example binary tree is

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

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

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 • Inorder and level order

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:

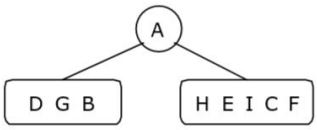
*From Preorder sequence **A** B D G C E H I F, the root is: A*

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:

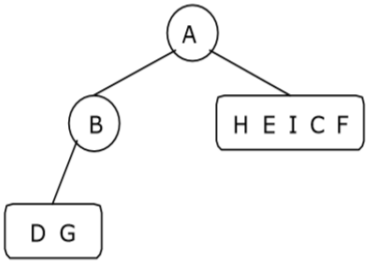


To find the root, left and right sub trees for D G B:

*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:



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

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.

Binary Search Tree

Def: Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

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.

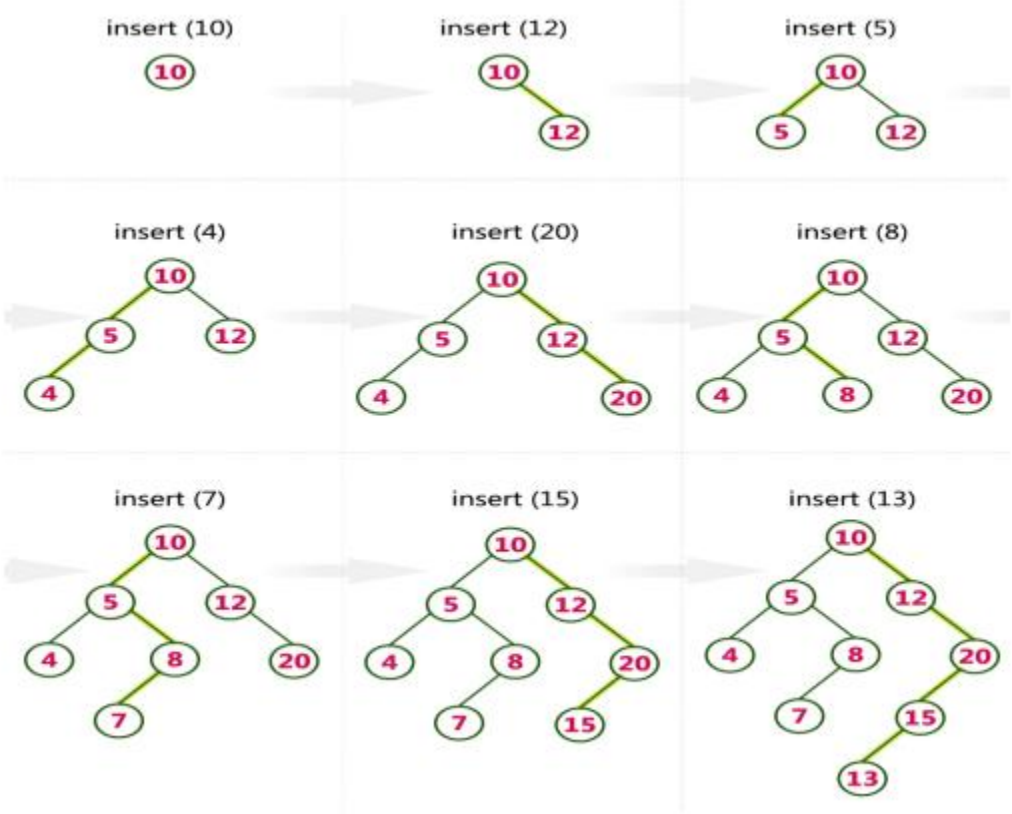
   Insertion Operation in BST

   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

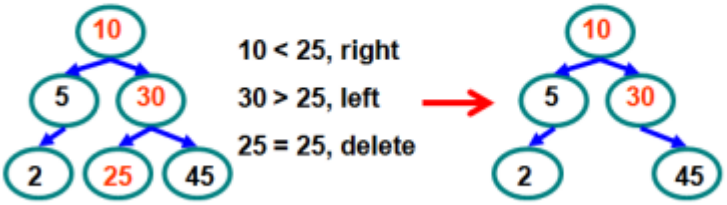
Deletion Operation in BST In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree includes following three cases.
Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting 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.
Example: Construct a Binary Search Tree by inserting the following sequence of numbers...
10,12,5,4,20,8,7,15 and 13
Above elements are inserted into a Binary Search Tree as follows.



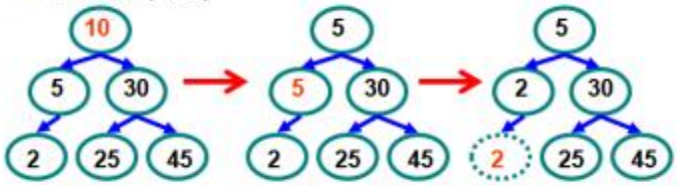
Example Deletion (Leaf)

- Delete (25)



Example Deletion (Internal Node)

- Delete (10)



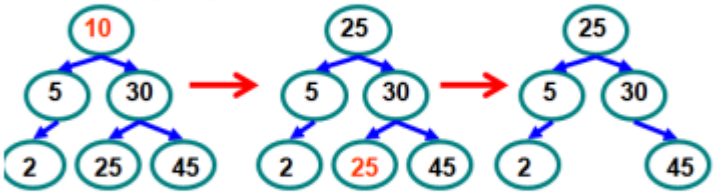
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

```
// Implementation of   BST operations
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *left;
struct node *right;
};

struct node* insert(struct node* root)
{
struct node *newnode,*head=root;
char ch;
do{
newnode=(struct node *)malloc(sizeof(struct node));
printf("enter data");
scanf("%d",&newnode->data);
newnode->left=0;
newnode->right=0;
if(root==0)
root=head=newnode;
else
{
root=head;
```

```
while(1)
 {
if(newnode->data<root->data)
{
if(root->left==NULL)
{
root->left=newnode;
break;
}
root=root->left;
}
if(newnode->data>root->data)
{
if(root->right==NULL)
{
root->right=newnode;
break;
}
root=root->right;
}
}
}
printf("do u want to continue\n");
scanf(" %c",&ch);
}while(ch=='y') ;
```

```
 return head;
}

 //Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) // checking if the root is not null
  {
    inorder(root -> left); // traversing left child
    printf(" %d ", root -> data); // printing data at root
    inorder(root -> right); // traversing right child
  }
}
void search(struct node *root)
{
int flag=0,key;
printf("enter key");
scanf("%d",&key);
while (root != NULL) {
        // pass right subtree as new tree
        if (key > root->data)
            root = root->right;

        // pass left subtree as new tree
        else if (key < root->data)
            root = root->left;
```

```
      else if(key==root->data)
            {flag=1;
          break;}// if the key is found return 1
           }
           if(flag==1)
           printf("key found");
           else
           printf("Not found");
    }
// Find the inorder successor
struct node *minValueNode(struct node *node) {
 struct node *current = node;
 // Find the leftmost leaf
 while (current && current->left != NULL)
 current = current->left;
 return current;
}
// Deleting a node
struct node *deleteNode(struct node *root, int key) {
 // Return if the tree is empty
 if (root == NULL) return root;
 // Find the node to be deleted
 if (key < root->data)
 root->left = deleteNode(root->left, key);
 else if (key > root->data)
```

```
root->right = deleteNode(root->right, key);
else {
// If the node is with only one child or no child
if (root->left == NULL) {
struct node *temp = root->right;
free(root);
return temp;
} else if (root->right == NULL) {
struct node *temp = root->left;
free(root);
return temp;
}
// If the node has two children
struct node *temp = minValueNode(root->right);
// Place the inorder successor in position of the node to be deleted
root->data = temp->data;
// Delete the inorder successor
root->right = deleteNode(root->right, temp->data); }
return root;
}
void main()
{
struct node *q=0,*p;
int x;
p=insert(q);
```

```
printf("Binary search tree inorder traversals\n");

inorder(p);

search(p);

printf("enter the data to be deleted");

scanf("%d",&x);

deleteNode(p,x);

inorder(p);

}
```

Introduction to Graphs

Graph is a non linear data structure. It contains set of points known as nodes (or vertices) and set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

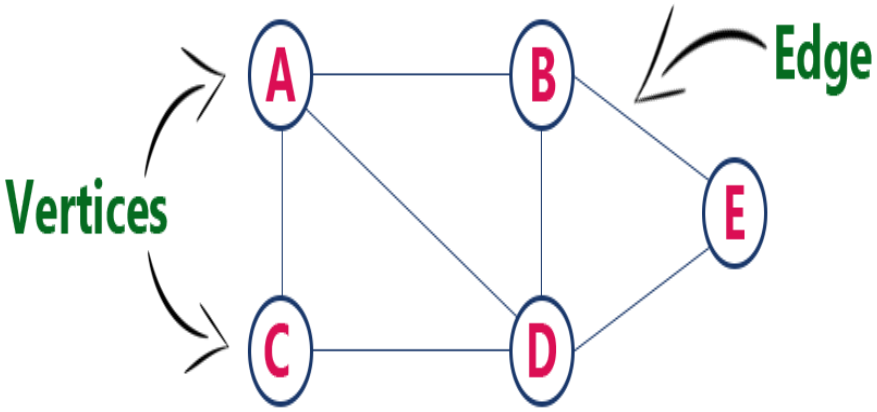
Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

If a edge is directed, its first endpoint is said to be the origin of it.

Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent

If there is a edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is a edge between them.

Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

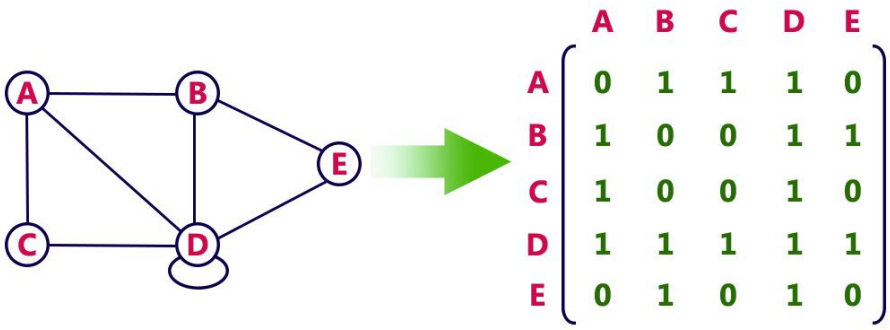
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

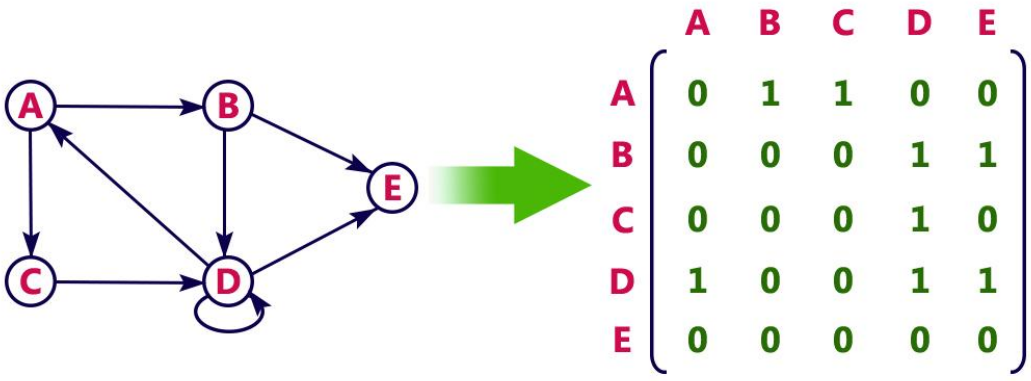
Adjacency Matrix

In this representation, graph is represented using a matrix of size total number of vertices by total number of vertices. That means graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is a edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following **undirected** graph representation...



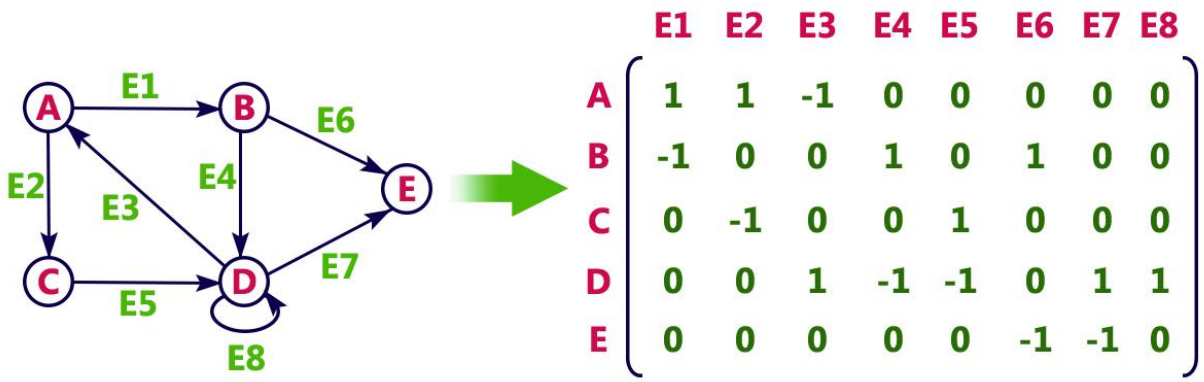
Directed graph representation...



Incidence Matrix

In this representation, graph is represented using a matrix of size total number of vertices by total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as outgoing edge to column vertex and -1 represents that the row edge is connected as incoming edge to column vertex.

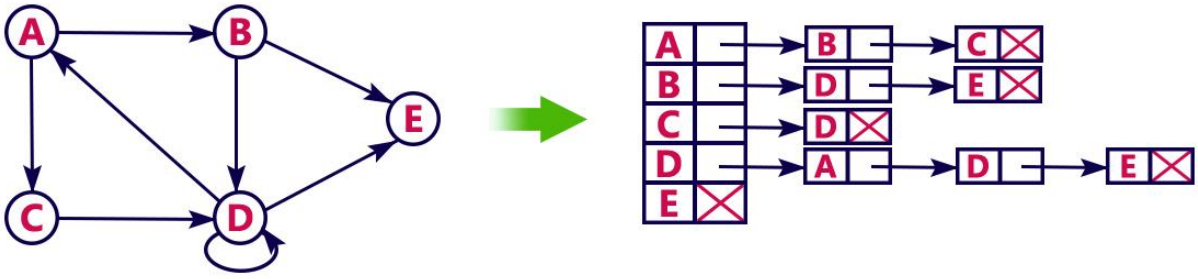
For example, consider the following directed graph representation...



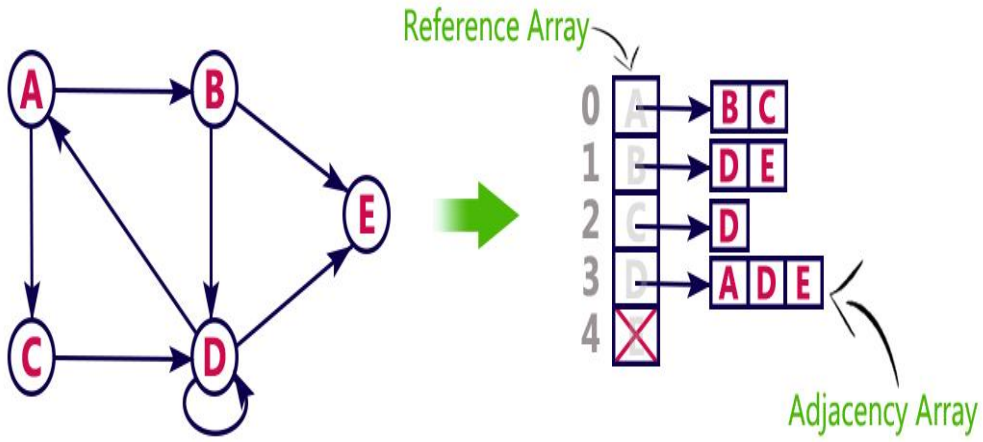
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph Traversal

Graph traversal is a technique used for searching vertex in a graph. The graph traversal is also used to decide the order of vertices to be 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 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)**

BFS (Breadth First Search)

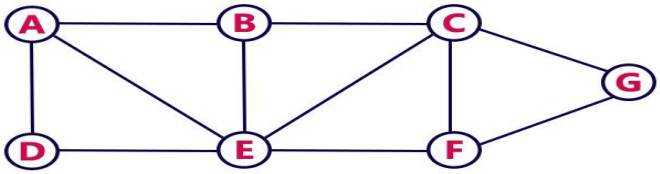
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

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.

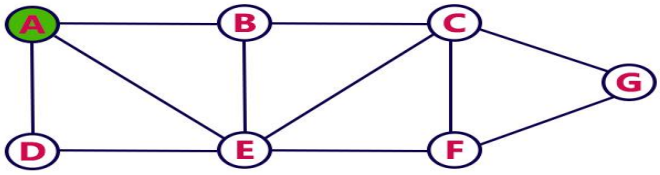
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

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.

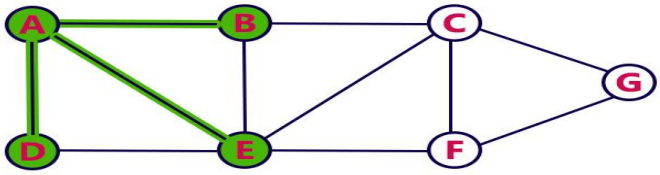


Queue

A						

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..

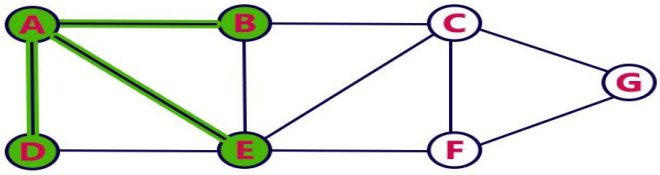


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

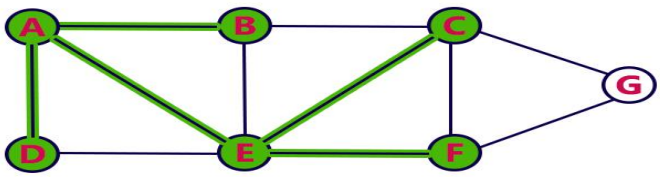


Queue

		E	B			

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.

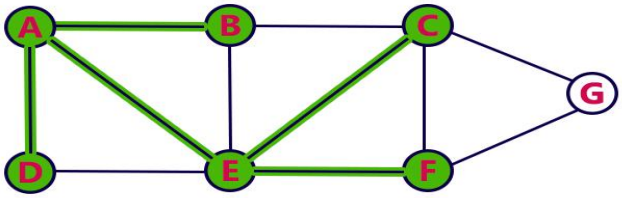


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

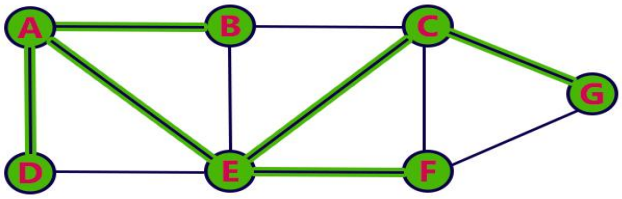


Queue

				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.

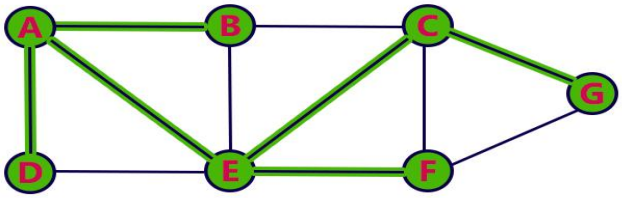


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

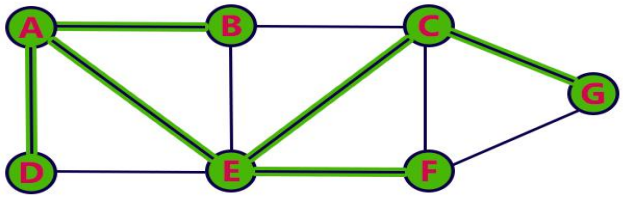


Queue

						G

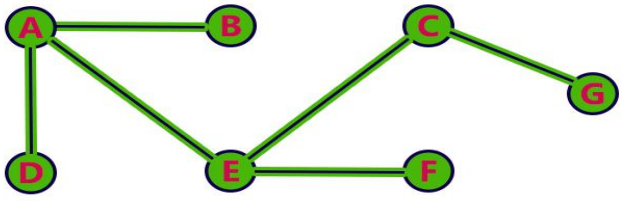
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

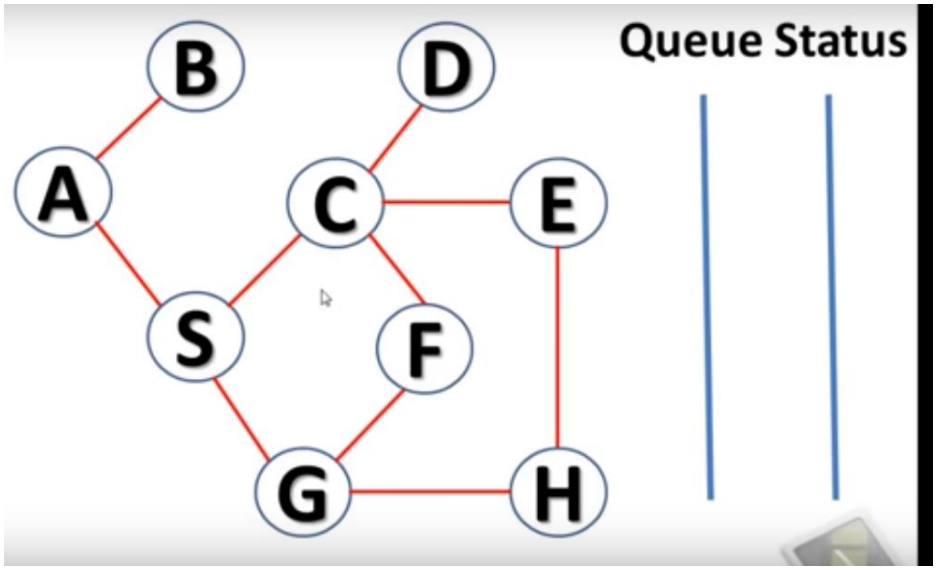


Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

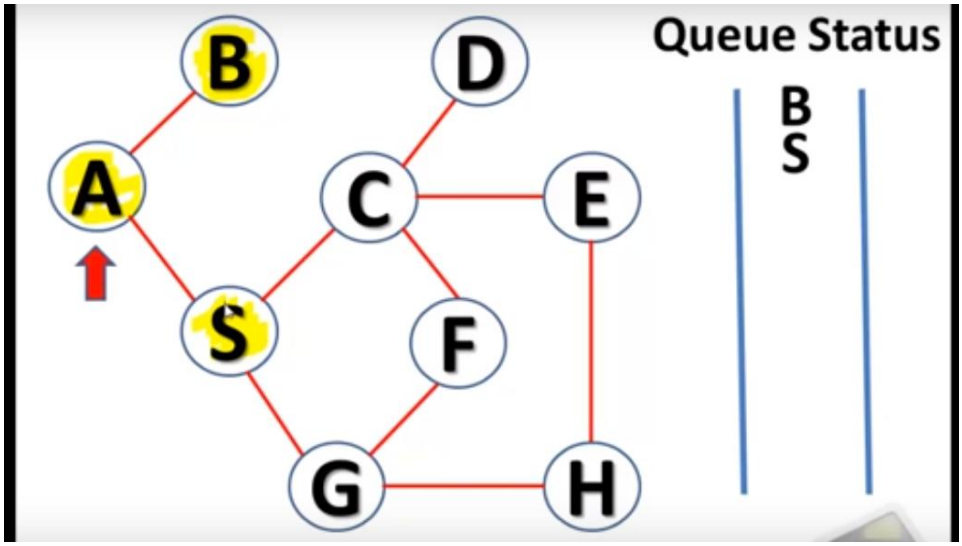


Example 2



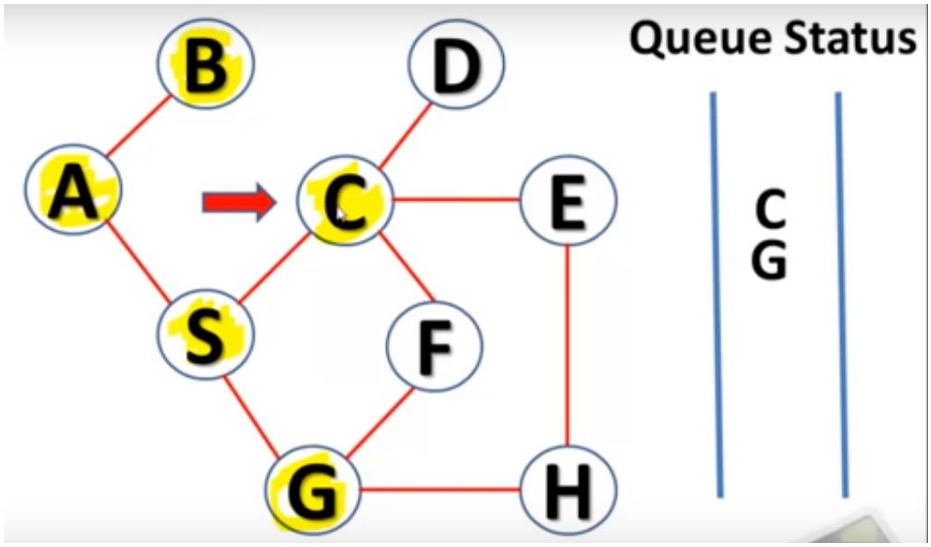
Queue Status :

Output: A



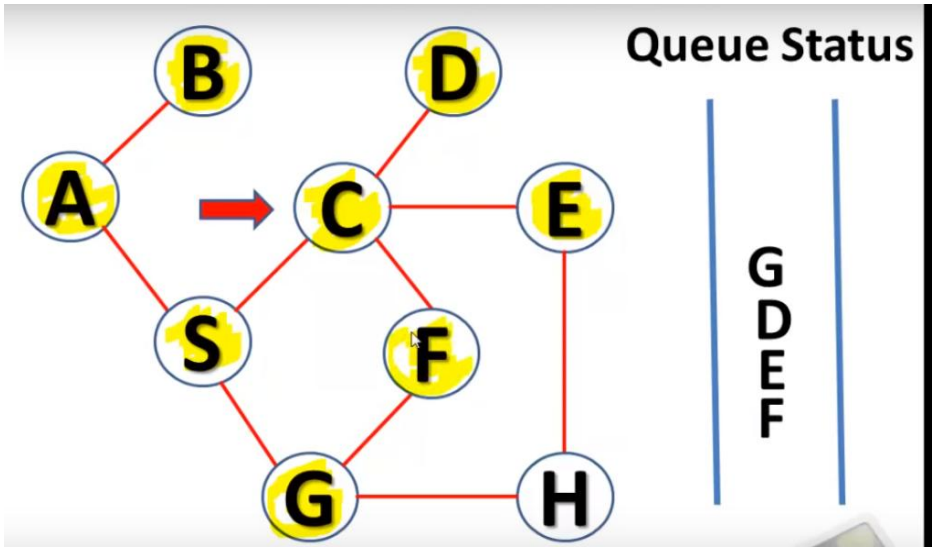
Queue Status : B S

Output: A B S



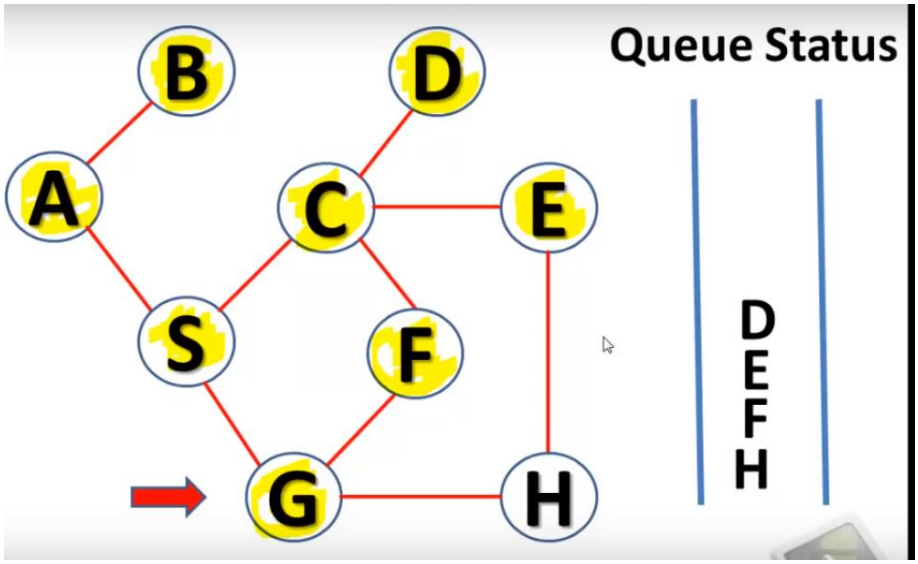
Queue Status : C G

Output: A B S



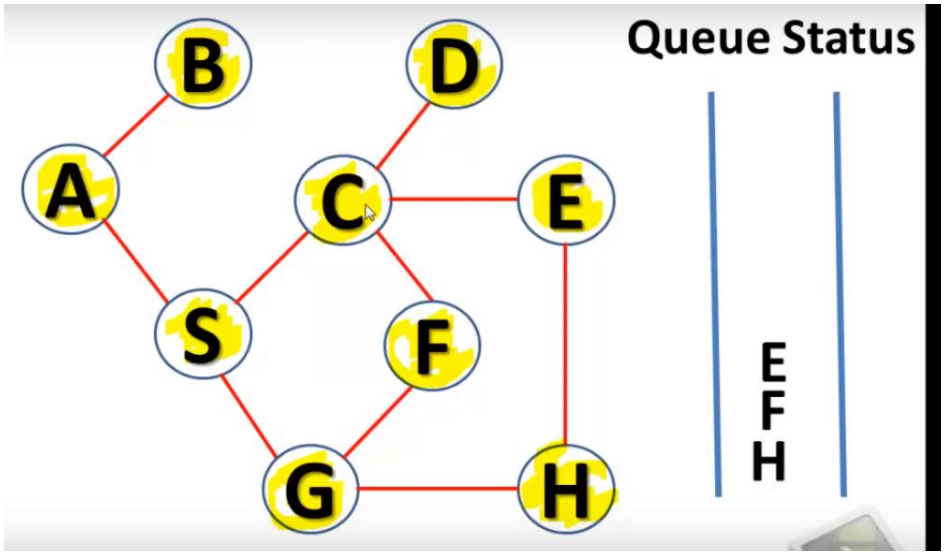
Queue Status : G D E F

Output: A B S C



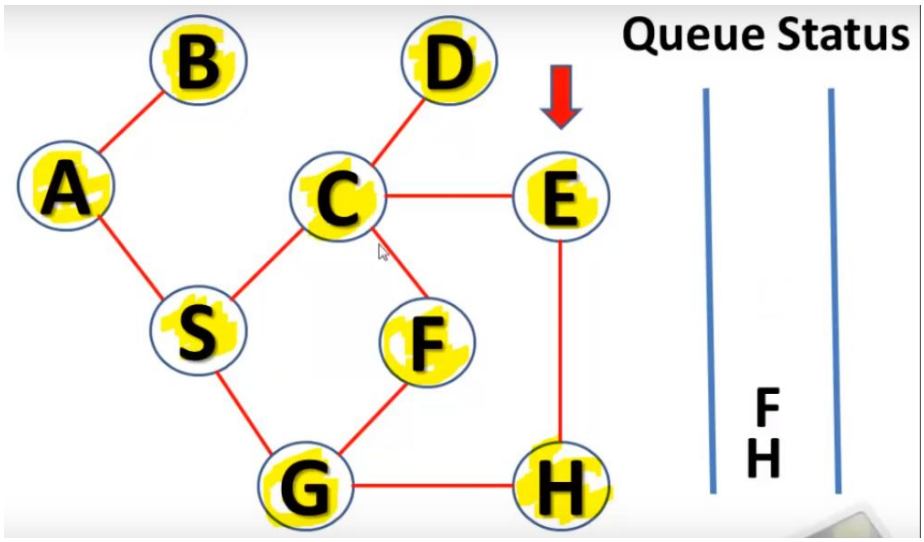
Queue Status : D E F H

Output: A B S C G



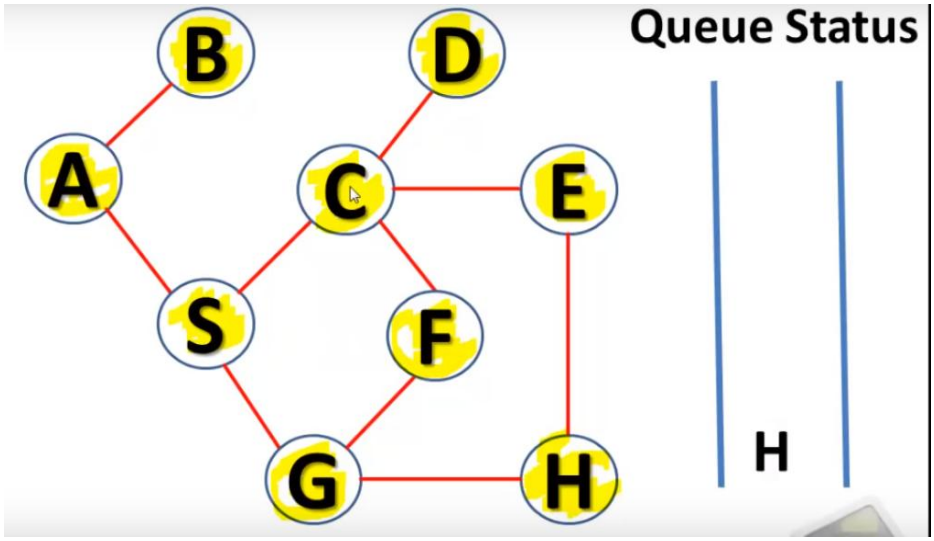
Queue Status : E F H

Output: A B S C G D



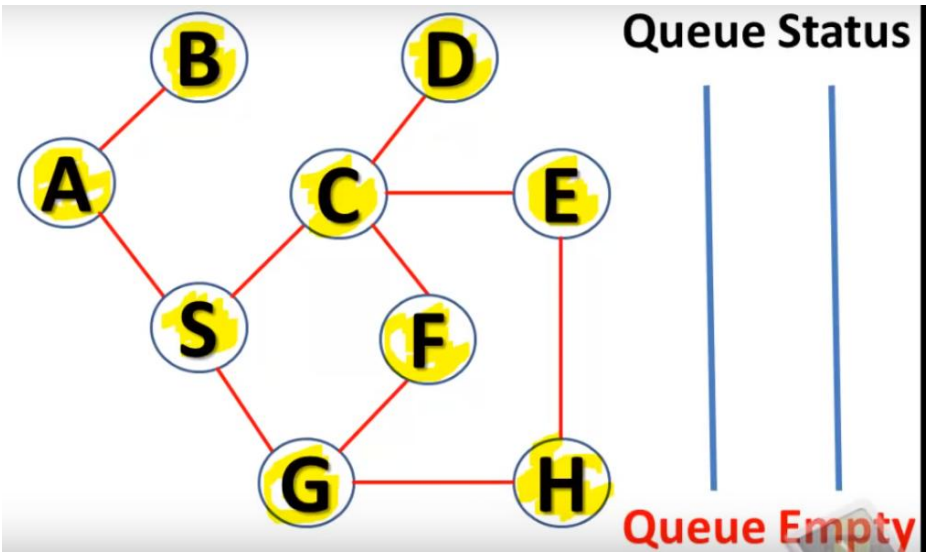
Queue Status : F H

Output: A B S C G D E



Queue Status : H

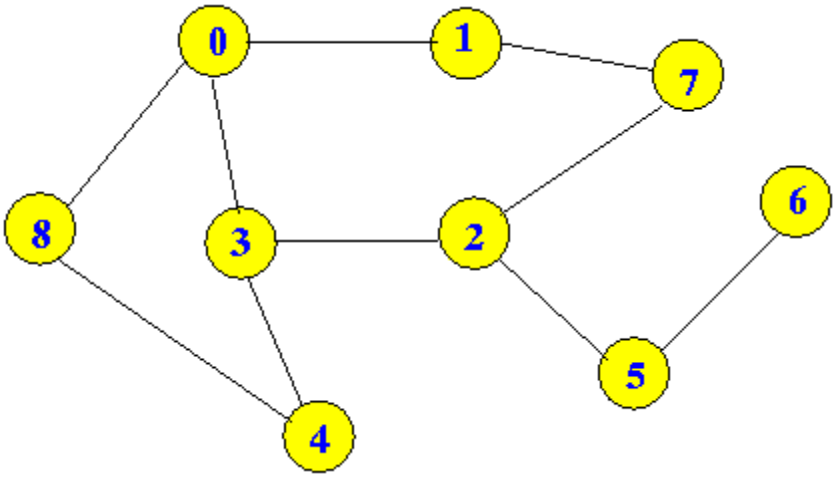
Output: A B S C G D E F



Queue Status :

Output: A B S C G D E F H

http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/bfs.html




```
#include<stdio.h>
int a[20][20], q[20], visited[20], n, i, j, f = 0, r = -1;
```

```
    void bfs(int v) {
    for(i = 1; i <= n; i++)
            if(a[v][i] && !visited[i])
                    q[++r] = i;
    if(f <= r) {
            visited[q[f]] = 1;
            bfs(q[f++]);
      }
  }

    void main() {
      int v;
       printf("\n Enter the number of vertices:");
       scanf("%d", &n);
      for(i=1; i <= n; i++) {
            q[i] = 0;
            visited[i] = 0;
      }
      printf("\n Enter graph data in matrix form:\n");
       for(i=1; i<=n; i++) {
             for(j=1;j<=n;j++) {
                    scanf("%d", &a[i][j]);
             }
       }

      printf("\n Enter the starting vertex:");
      scanf("%d", &v);
       bfs(v);
       printf("\n The node which are reachable are:\n");

       for(i=1; i <= n; i++) {
            if(visited[i])
            printf("%d\t", i);
             else {
                    printf("\n Bfs is not possible. Not all nodes are reachable");
                     break;
             }
       }
}
```

Enter the number of vertices:6

Enter graph data in matrix form:

0 1 1 0 0 0

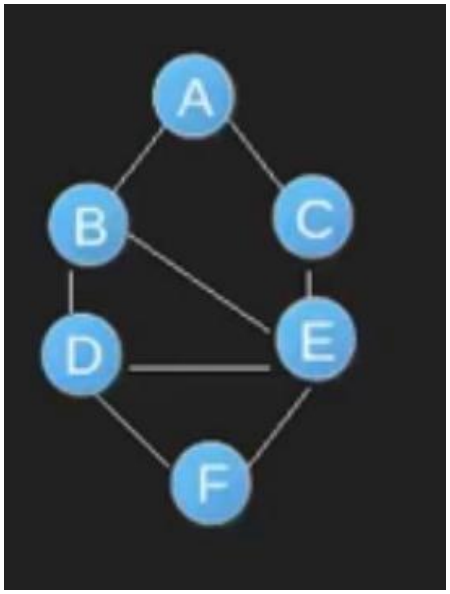
1 0 0 1 1 0

1 0 0 0 1 0

0 1 0 0 1 1

0 1 1 1 0 1

0 0 0 1 1 0



Enter the starting vertex:1

The node which are reachable are:

1 2 3 4 5 6

Graph Traversal - DFS

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...

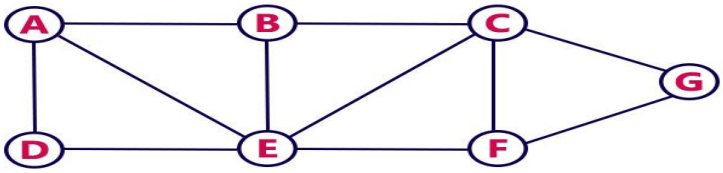
- **Step 1 -** Define a **Stack** of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

- **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.

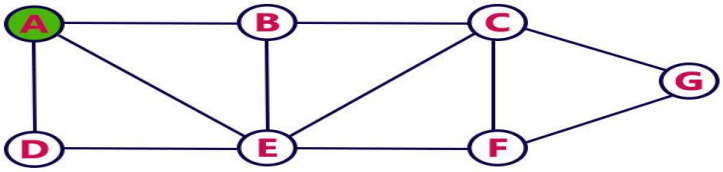
Example

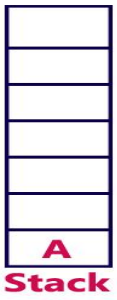
Consider the following example graph to perform DFS traversal



Step 1:

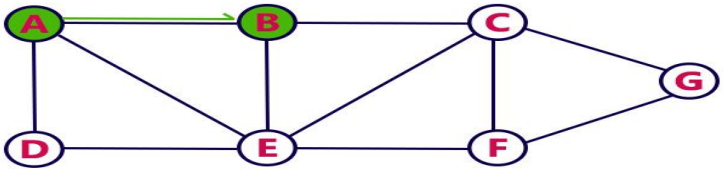
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.





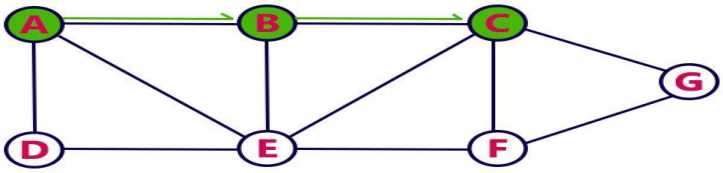
Step 2:

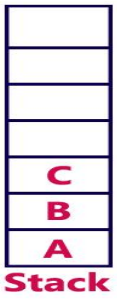
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

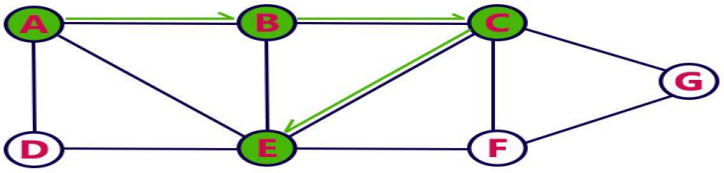
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





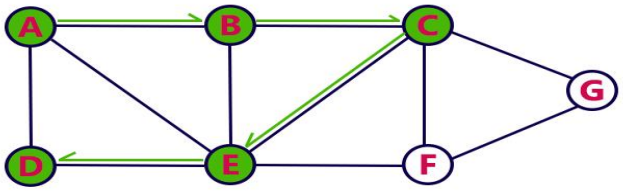
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



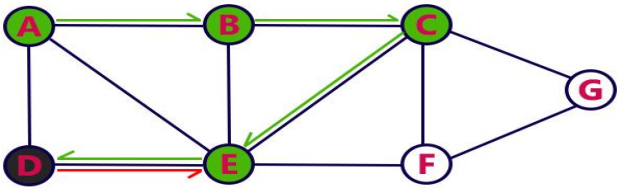
Step 5:

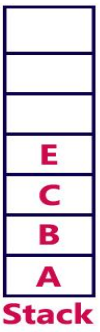
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

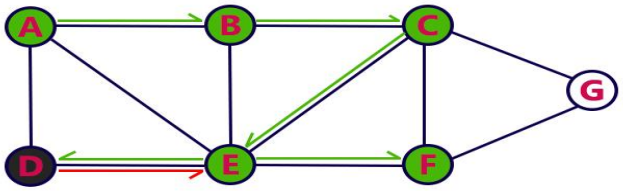
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

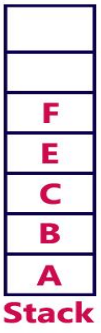




Step 7:

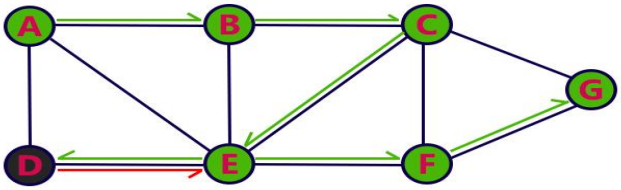
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





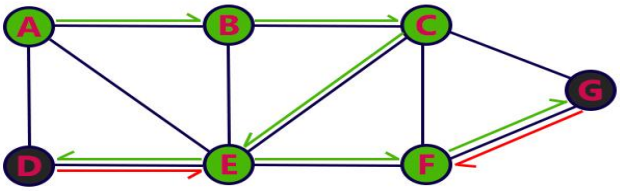
Step 8:

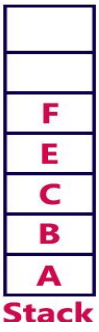
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:

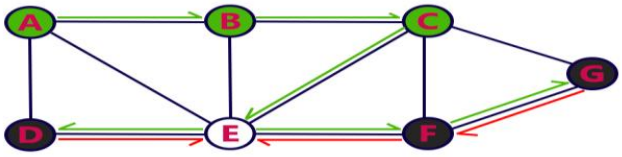
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





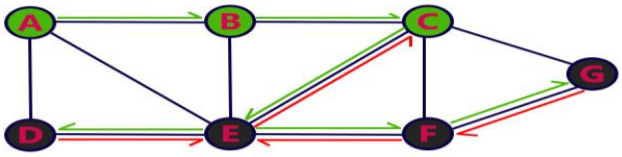
Step 10:

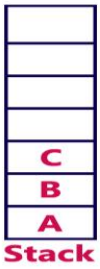
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

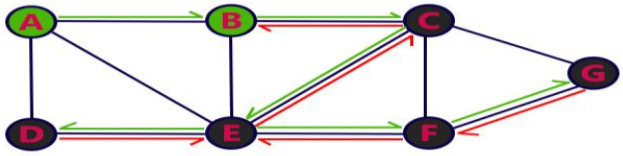
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

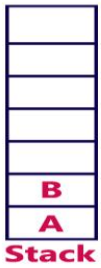




Step 12:

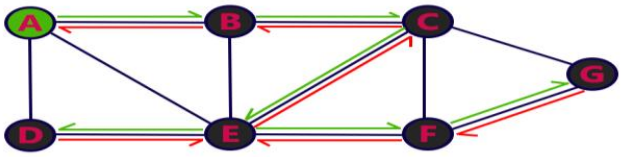
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





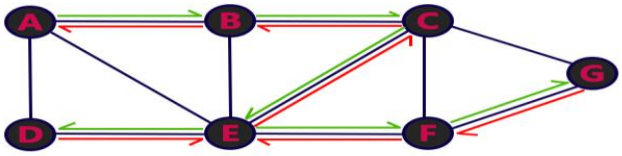
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

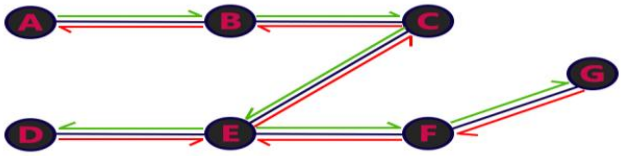


Step 14:

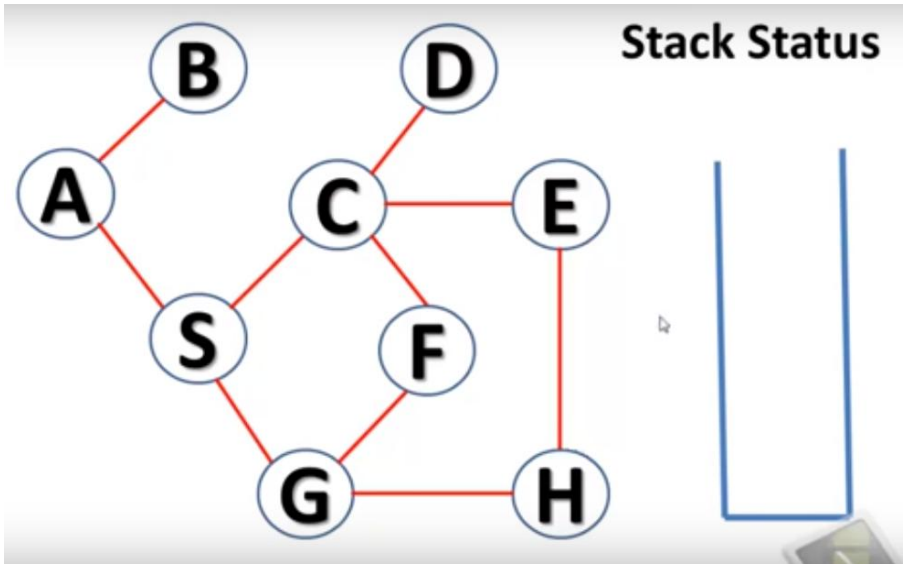
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

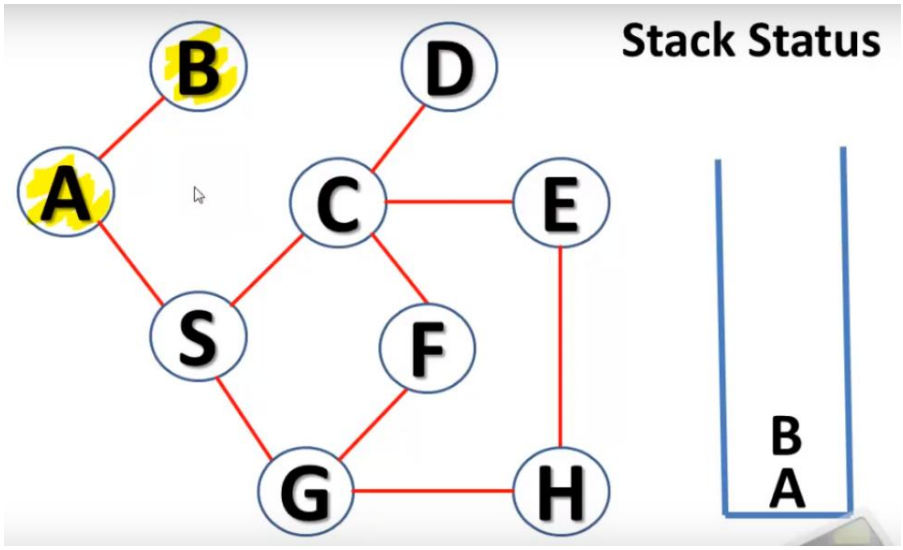


Example



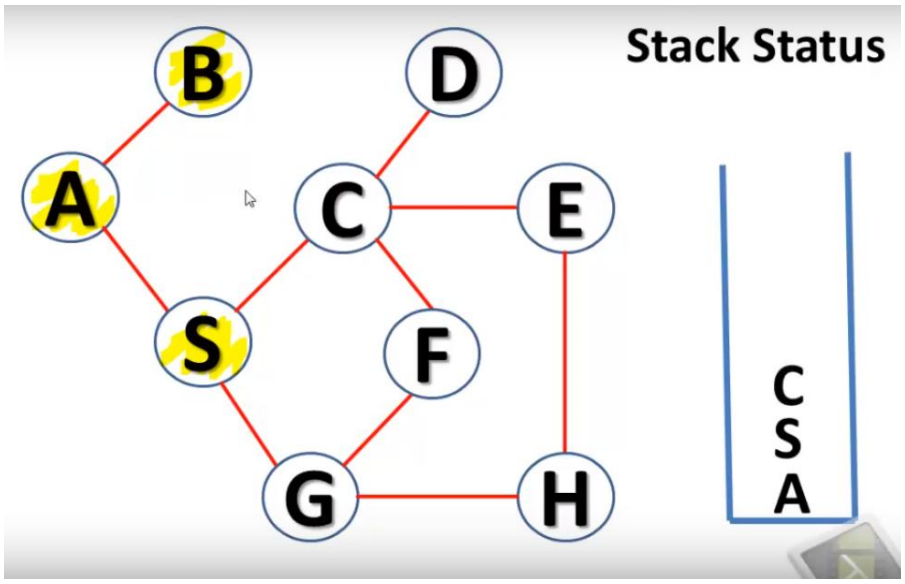
Stack status:

Output:



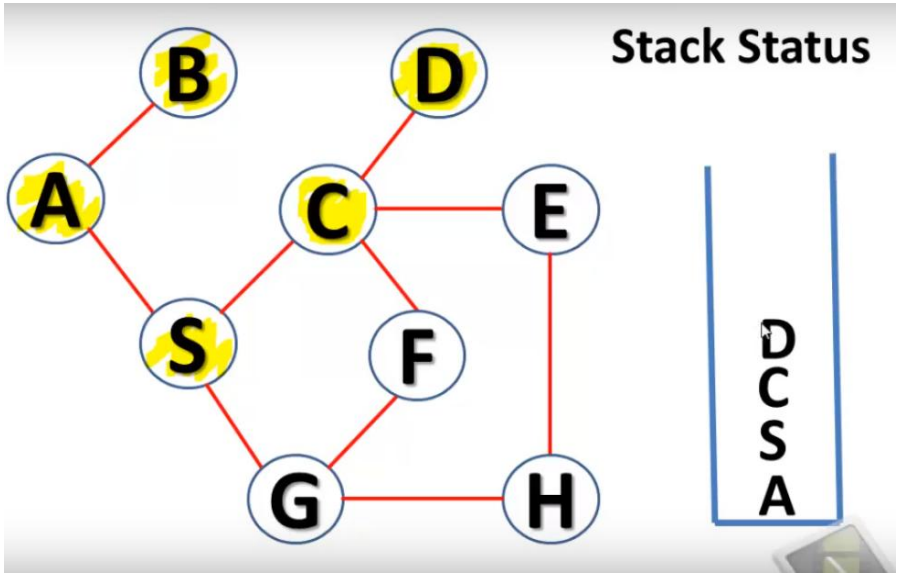
Stack status: A B

Output: A B



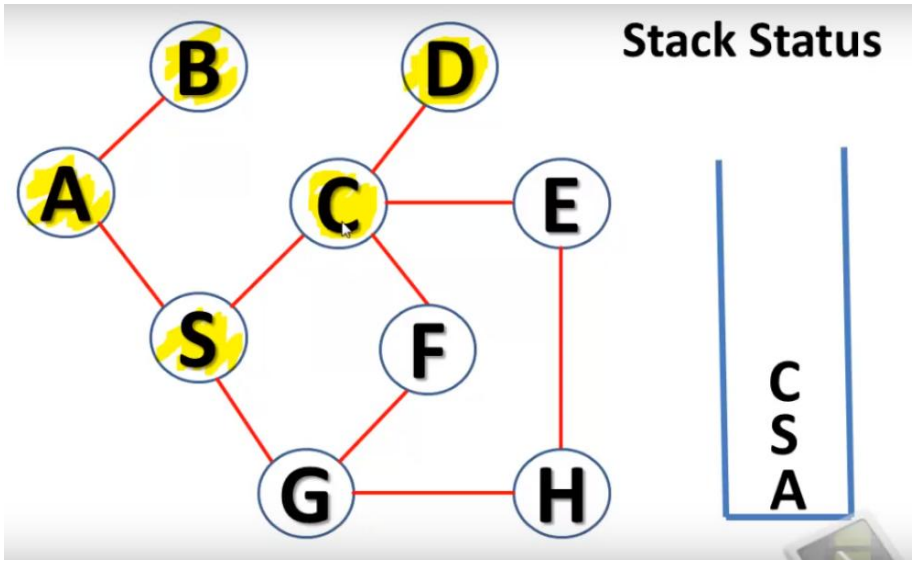
Stack status: A S C

Output: A B S



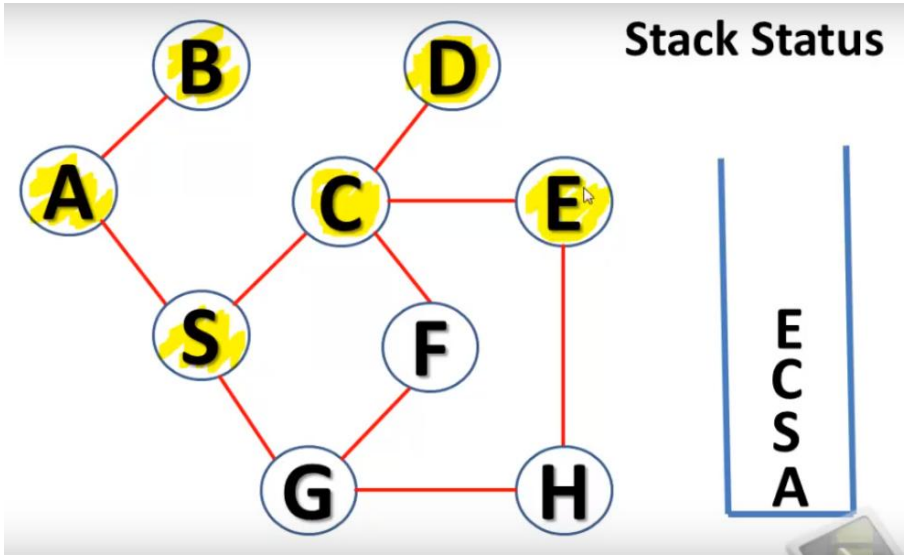
Stack status: A S C D

Output: A B S C



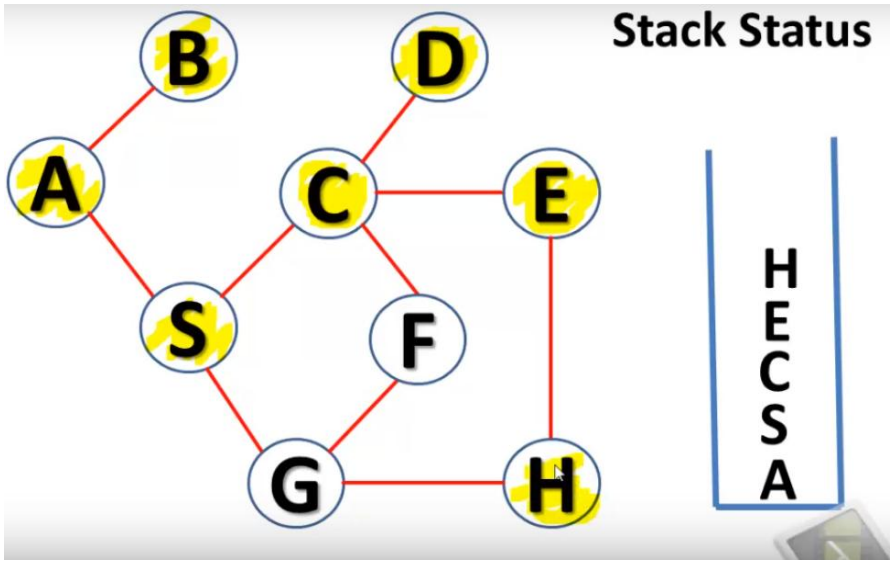
Stack status: A S C

Output: A B S C D



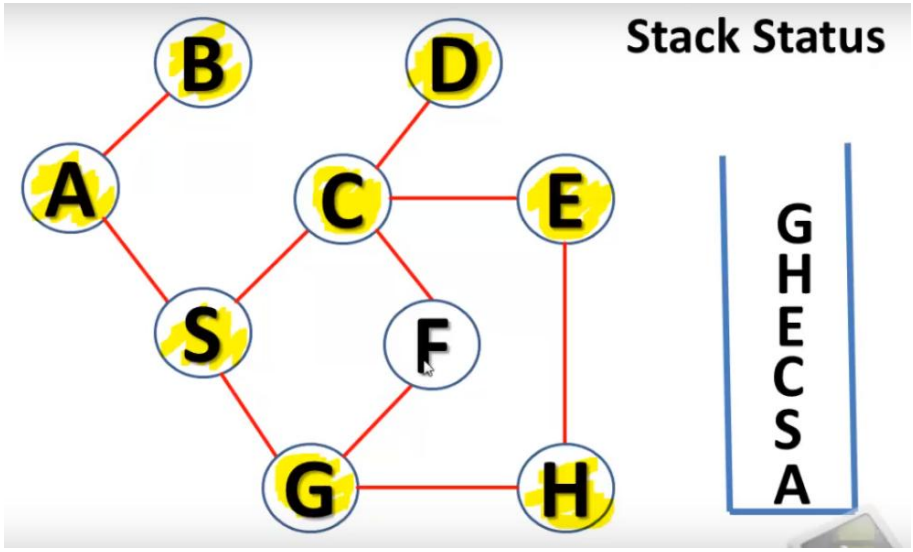
Stack status: A S C E

Output: A B S D E



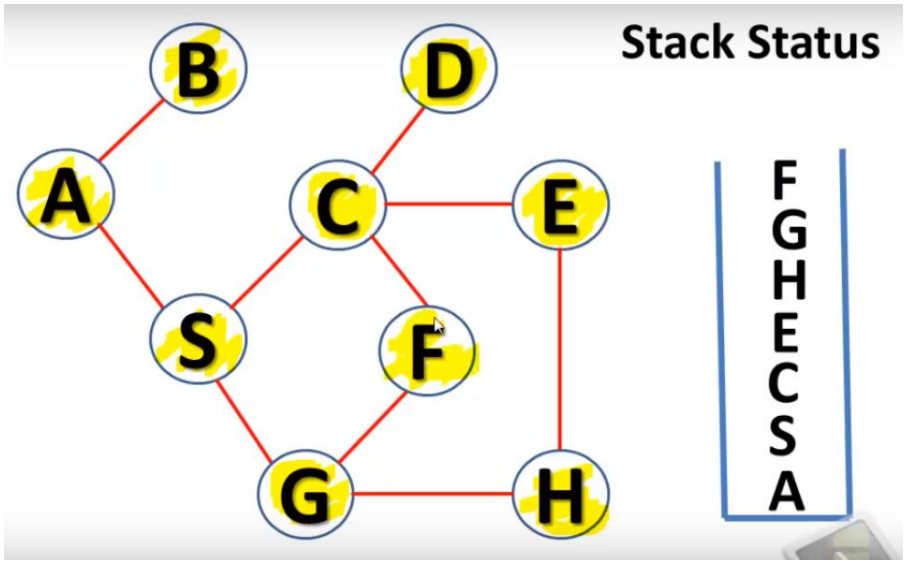
Stack status: A S C E H

Output: A B S D E H



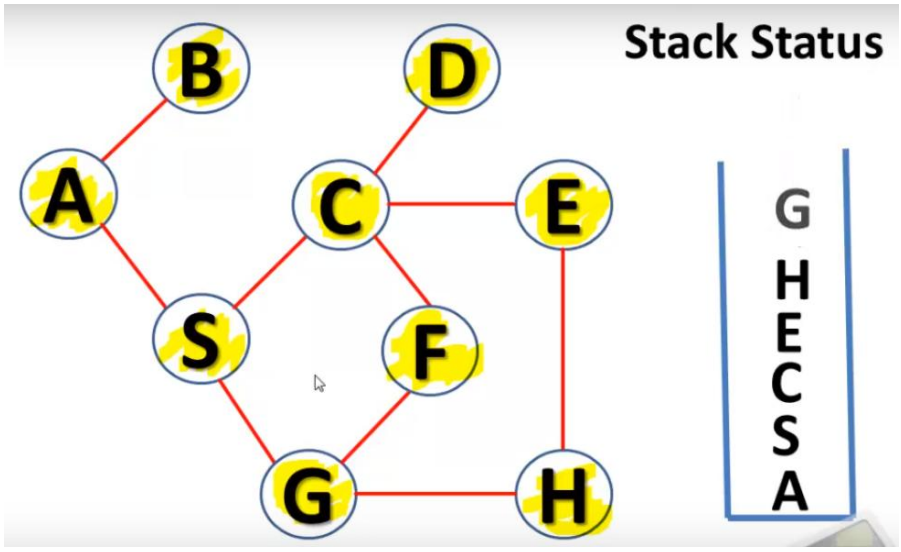
Stack status: A S C E H G

Output: A B S D E H G



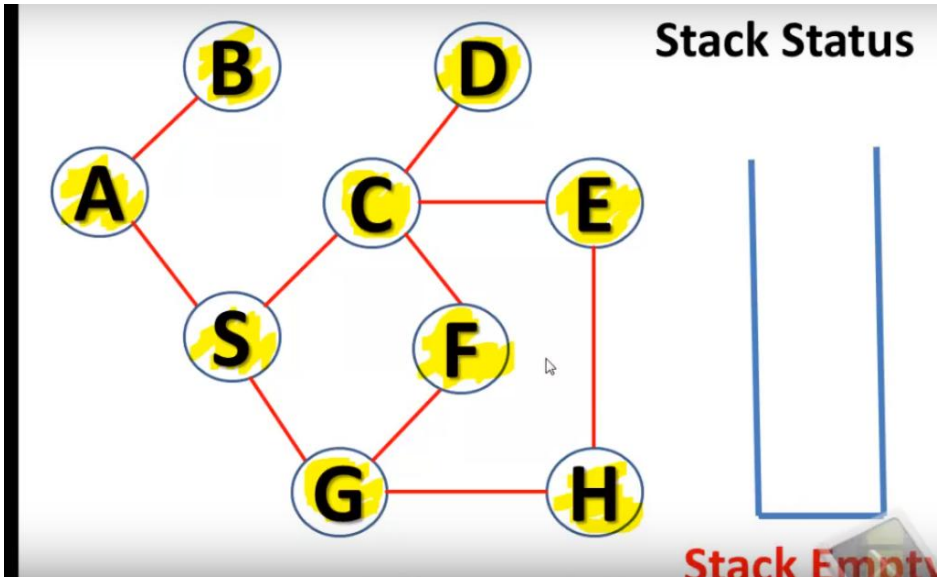
Stack status: A S C E H G F

Output: A B S C D E H G F



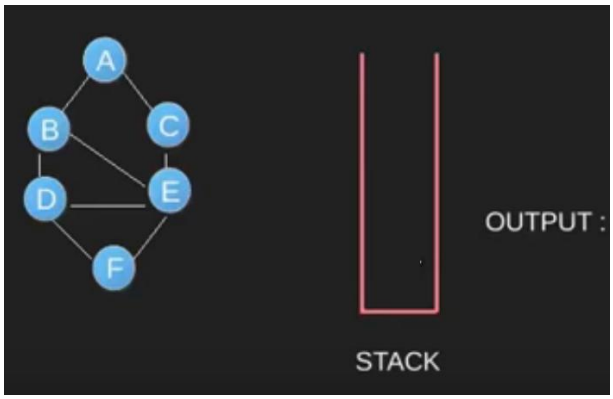
Stack status: A S C E H G

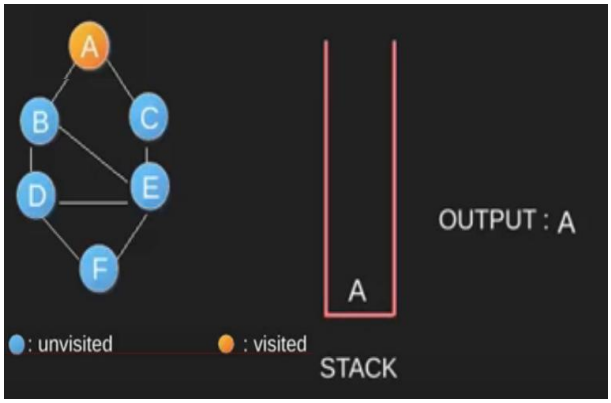
Output: A B S C D E H G F

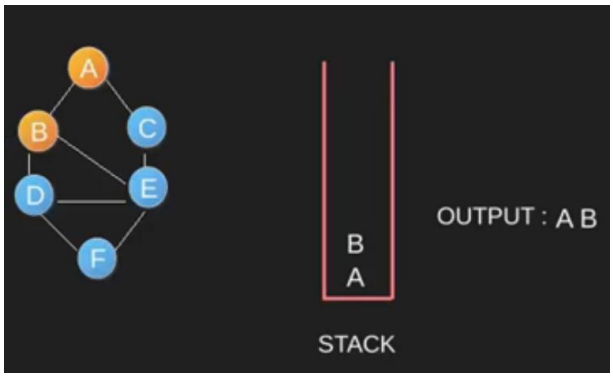


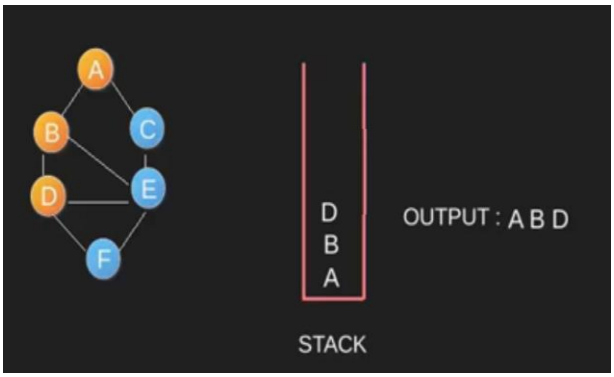
Stack status:

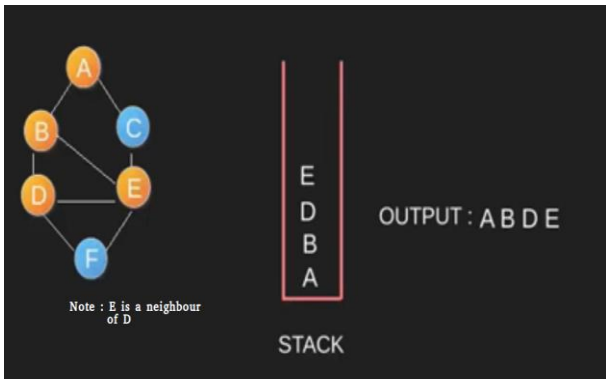
Output: A B S C D E H G F

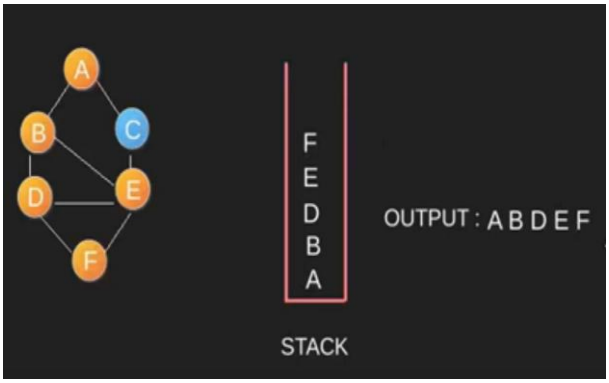


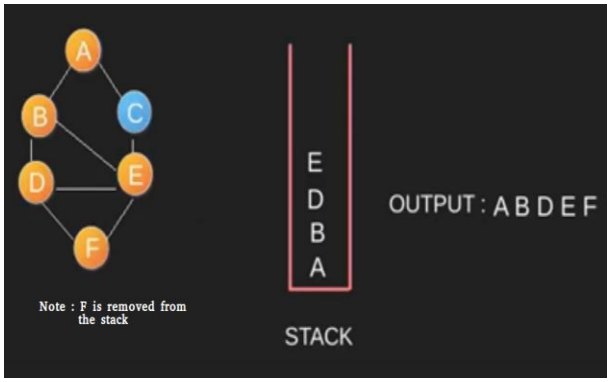


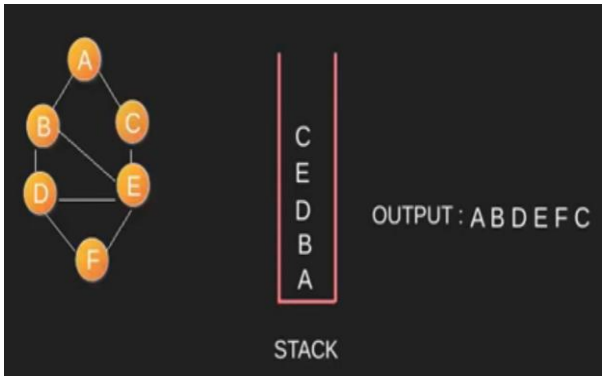


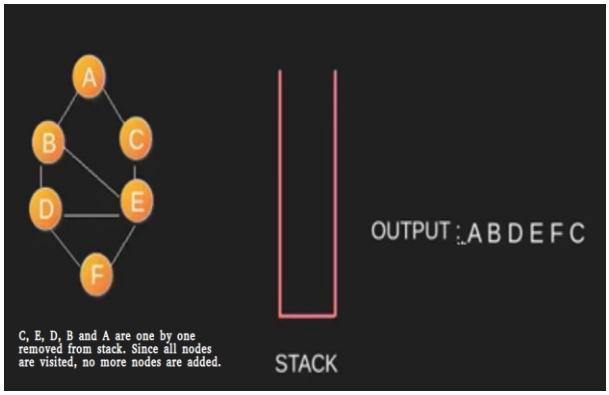












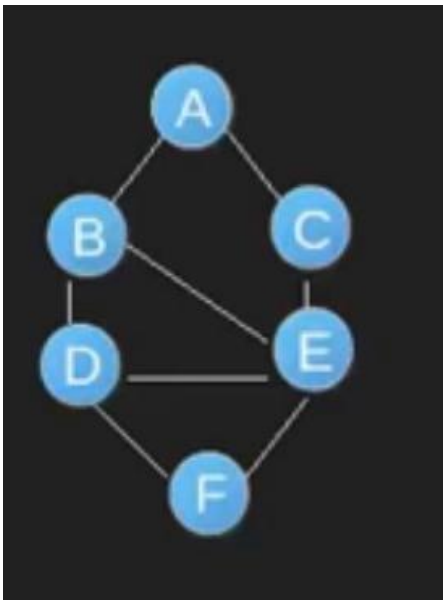
Implement DFS using C programing

```
#include<stdio.h>

 void DFS(int);

int G[10][10],visited[10],n;    //n is no of vertices and graph is sorted in array G[10][10]
```

```
void main()
{
   int i,j;
   printf("Enter number of vertices:");
         scanf("%d",&n);
    //read the adjecency matrix
         printf("\nEnter adjecency matrix of the graph:");
         for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         scanf("%d",&G[i][j]);
    //visited is initialized to zero
  for(i=0;i<n;i++)
      visited[i]=0;
 DFS(0);
}

void DFS(int i)
{
   int j;
   printf("\n%d",i);
   visited[i]=1;
   for(j=0;j<n;j++)
      if(!visited[j]&&G[i][j]==1)
         DFS(j);
```

}

Enter number of vertices:6



Enter adjecency matrix of the graph:

0 1 1 0 0 0

1 0 0 1 1 0

1 0 0 0 1 0

0 1 0 0 1 1

0 1 1 1 0 1

0 0 0 1 1 0

0

1

3

4

2

5

Difference between DFS and BFS

Spanning Tree

Spanning tree can be defined as a sub-graph of connected, undirected graph G that is a tree produced by removing the desired number of edges from a graph. In other words, Spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

Minimum Spanning Tree

There can be weights assigned to every edge in a weighted graph. However, A minimum spanning tree is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains **the least weight among** all other spanning tree of some particular graph.

Shortest path algorithms

In this section, we will discuss the algorithms to calculate the shortest path between two nodes in a graph.

There are two algorithms which are being used for this purpose.

Prim's Algorithm

Kruskal's Algorithm

Prim's Algorithm

- Prim’s Algorithm is greedy algorithm used for finding the Minimum Spanning Tree (MST) of a given graph.
- The graph must be weighted, connected and undirected.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

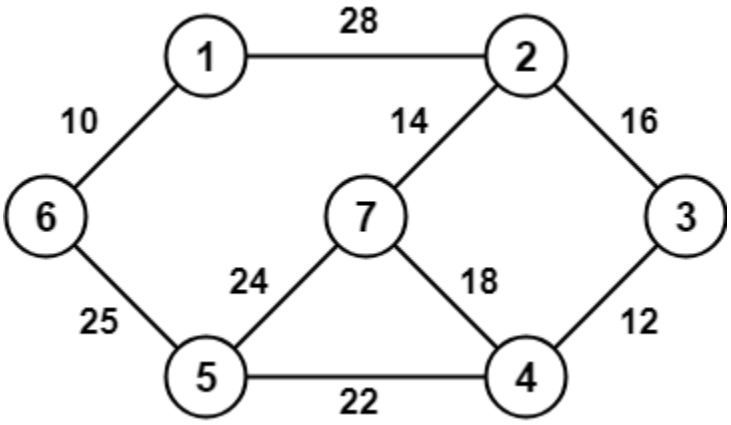
The algorithm is given as follows.

Algorithm

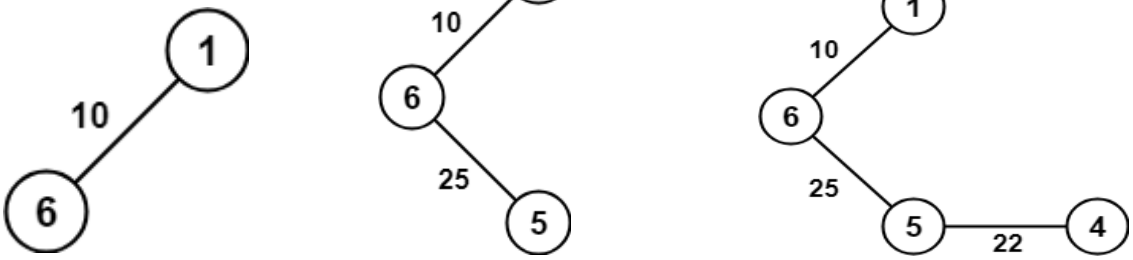
- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight , If including that edge creates a cycle, then reject that edge and look for the next least weight edge.
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T [END OF LOOP]
- **Step 5:** EXIT

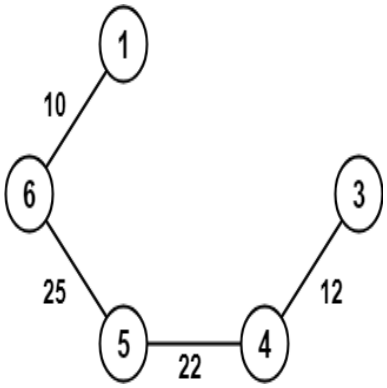
Problem-01:

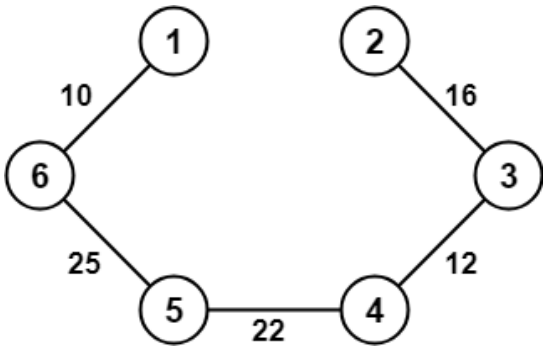
Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

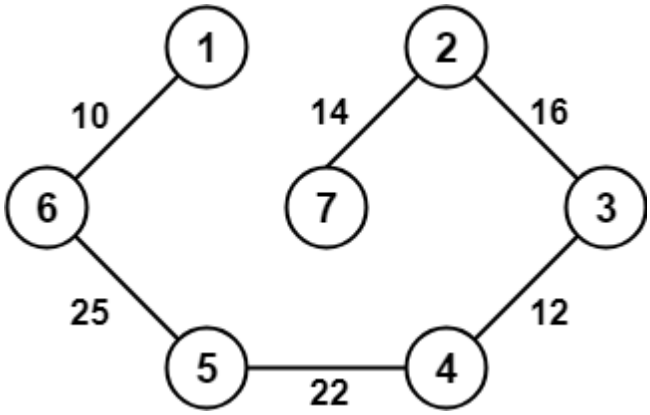


Solution-









Since all the vertices have been included in the MST, so we stop.

Weight of the MST

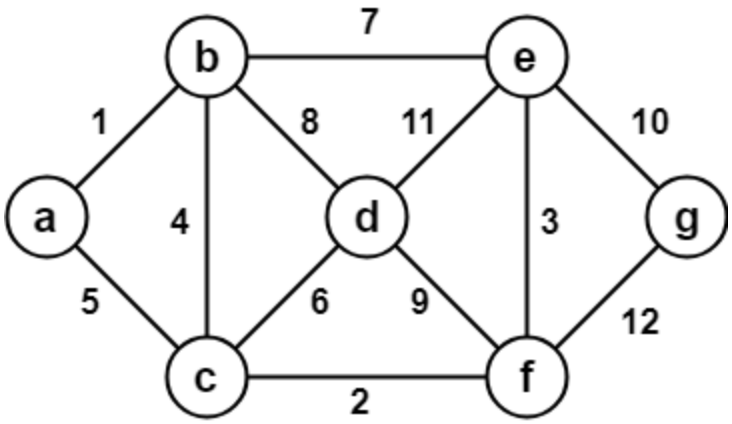
= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

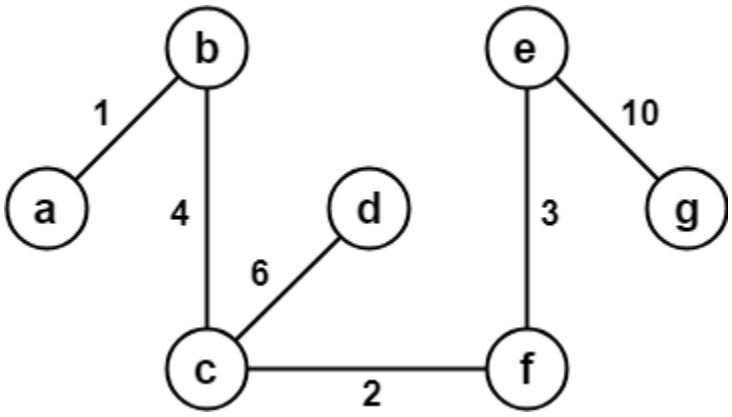
= 99 units

Problem-02:

Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



The Minimum Spanning Tree (MST) obtained by the application of Prim's Algorithm on given graph is-



Weight of the MST

= Sum of all edge weights

= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

```
#include<stdio.h>

int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
        printf("\n Enter the number of nodes:");
        scanf("%d",&n);
```

```
	printf("\n Enter the adjacency matrix:\n");
	for(i=1;i<=n;i++)
		for(j=1;j<=n;j++)
		{
			scanf("%d",&cost[i][j]);
			if(cost[i][j]==0)
			cost[i][j]=999;
		}
	visited[1]=1;
	printf("n");
	while(ne<n)
	{
		for(i=1,min=999;i<=n;i++)
			for(j=1;j<=n;j++)
			if(cost[i][j]<min)
				if(visited[i]!=0)
				{
					min=cost[i][j];
					a=u=i;
					b=v=j;
				}
		if(visited[u]==0 || visited[v]==0)
		{
			printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
			mincost+=min;
			visited[b]=1;
		}
		cost[a][b]=cost[b][a]=999;
	}
	printf("\n Minimun cost=%d",mincost);
}
```

Enter the number of nodes:6

Enter the adjacency matrix:

0 4 4 0 0 0

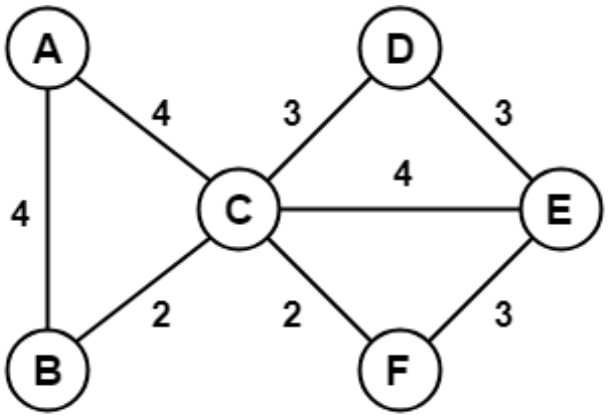
4 0 2 0 0 0

4 2 0 3 4 2

0 0 3 0 3 0

0 0 4 3 0 3

0 0 2 0 3 0



Given Graph

Edge 1:(1 2) cost:4

Edge 2:(2 3) cost:2

Edge 3:(3 6) cost:2

Edge 4: (3,4) cost:3

Edge 5:(4 5) cost:3

Minimun cost=14

Kruskal's Algorithm

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm is given as follows.

Step-01:

Sort all the edges from low weight to high weight.

Step-02:

Take the edge with the lowest weight and use it to connect the vertices of graph.

If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

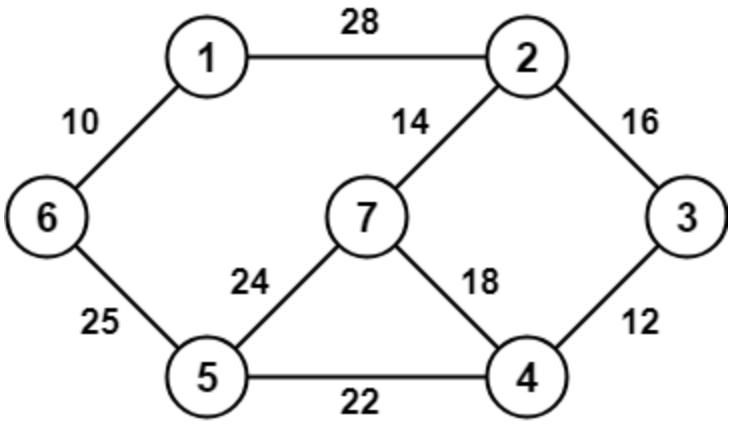
Time Complexity-

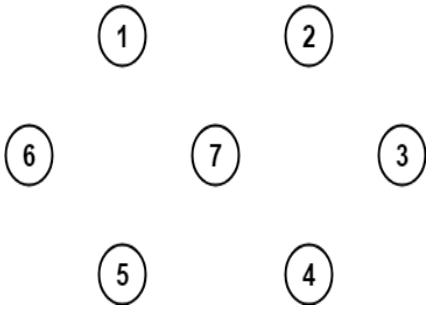
Worst case time complexity of Kruskal's Algorithm = O(ElogE).

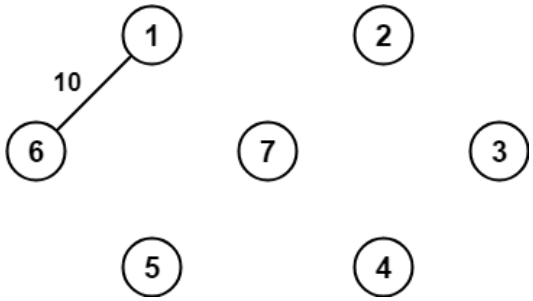
Special Case-

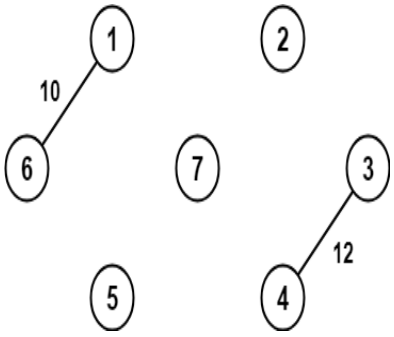
- If the edges are already sorted, then there is no need to construct min heap.
- So, deletion from min heap time is saved.
- In this case, time complexity of Kruskal's Algorithm = O(E + V)

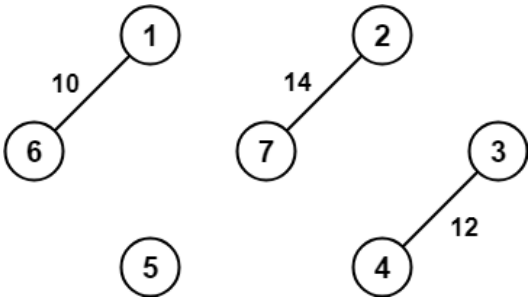
Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

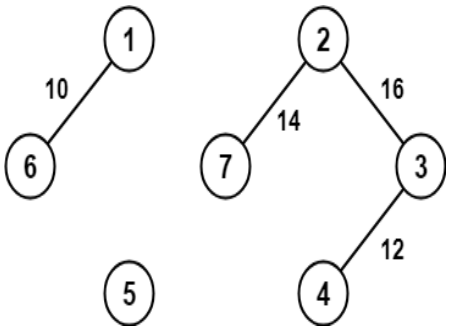

1
2
28
10
14
16
6
7
3
24
18
25
12
5
4
22

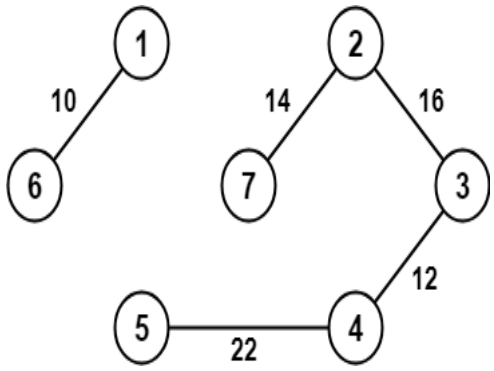

1
2
6
7
3
5
4

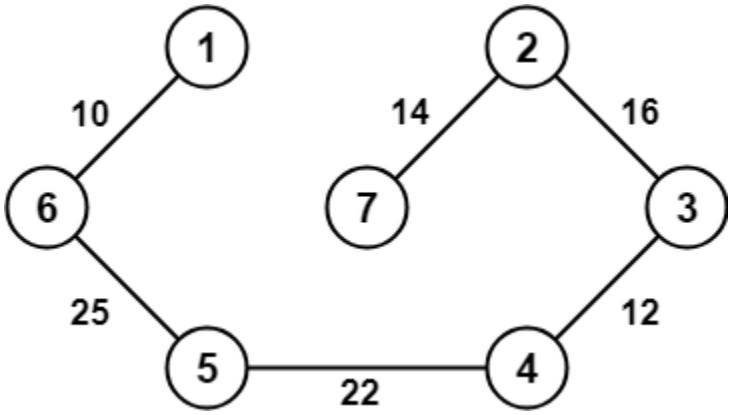

1
10
2
6
7
3
5
4


1
2
10
6
7
3
12
5
4


1
10
2
14
6
7
3
12
5
4







Since all the vertices have been included in the MST, so we stop.

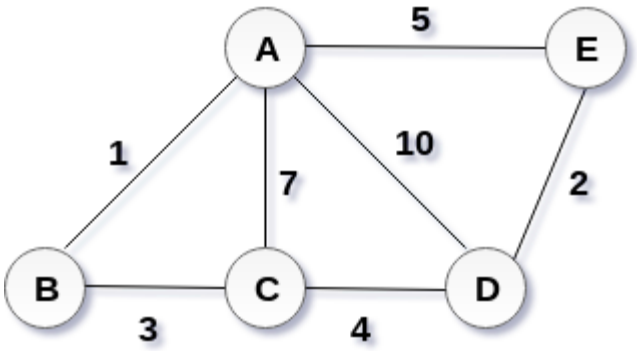
Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Apply the Kruskal's algorithm on the graph given as follows.



Solution:

the weight of the edges given as :

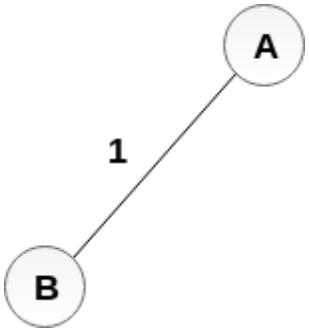
Edge	**AE**	**AD**	**AC**	**AB**	**BC**	**CD**	**DE**
Weight	5	10	7	1	3	4	2

Sort the edges according to their weights.

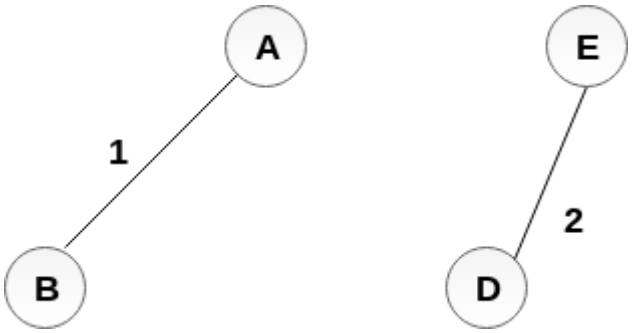
Edge	**AB**	**DE**	**BC**	**CD**	**AE**	**AC**	**AD**
Weight	1	2	3	4	5	7	10

Start constructing the tree;

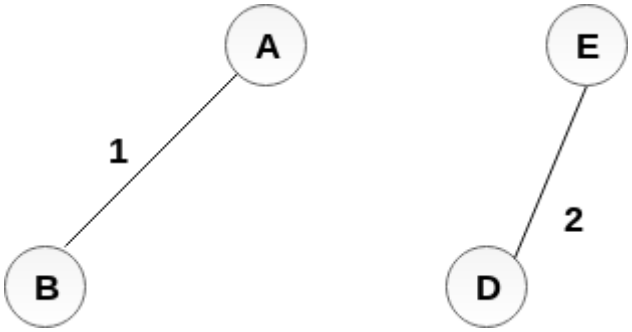
Add AB to the MST;

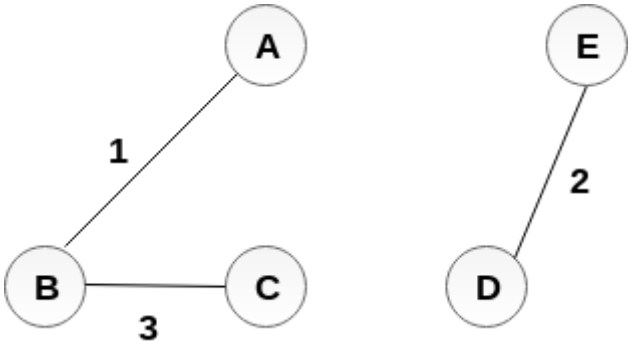


Add DE to the MST;



Add BC to the MST;





The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

Hence, the final MST is the one which is shown in the step 4.

the cost of MST = 1 + 2 + 3 + 4 = 10.

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        printf("\n\n\t Implementation of Kruskal's algorithm\n\n");
        printf("\nEnter the no. of vertices\n");
        scanf("%d",&n);
        printf("\nEnter the cost adjacency matrix\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        scanf("%d",&cost[i][j]);
                        if(cost[i][j]==0)
                        cost[i][j]=999;
                }
        }
        printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
        while(ne<n)
```

```
	{
		for(i=1,min=999;i<=n;i++)
		{
			for(j=1;j<=n;j++)
			{
			if(cost[i][j]<min)
			{
				min=cost[i][j];
				a=u=i;
				b=v=j;
			}
			}
		}
		u=find(u);
		v=find(v);
		if(uni(u,v))
		{
			printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
			mincost +=min;
		}
		cost[a][b]=cost[b][a]=999;
	}
	printf("\n\tMinimum cost = %d\n",mincost);
}

int find(int i)
{
	while(parent[i])
		i=parent[i];
	return i;
}
int uni(int i,int j)
{
	if(i!=j)
	{
		parent[j]=i;
		return 1;
	}
	return 0;
}
```

Enter the no. of vertices

6

Enter the cost adjacency matrix

0 4 4 0 0 0

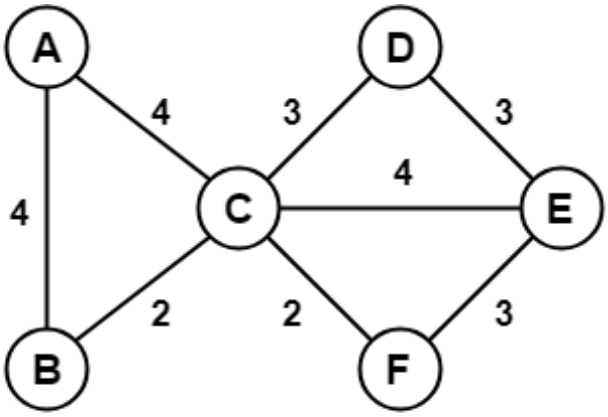
4 0 2 0 0 0

4 2 0 3 4 2

0 0 3 0 3 0

0 0 4 3 0 3

0 0 2 0 3 0



Given Graph

The edges of Minimum Cost Spanning Tree are

1 edge (2,3) =2

2 edge (3,6) =2

3 edge (3,4) =3

4 edge (4,5) =3

5 edge (1,2) =4

Minimum cost = 14

Prim's Algorithm	**Kruskal's Algorithm**
In Prim's Algorithm, the tree that we are making or growing always remains connected.	In kruskal's Algorithm, the tree that we are making or growing usually remains disconnected.
Prim's Algorithm will grow a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm will grow a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.

Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Which algorithm is preferred- Prim's Algorithm or Kruskal's Algorithm?

Solution-

- Kruskal's Algorithm is preferred when the graph is sparse i.e. when there are less number of edges in the graph like E = O(V) or when the edges are already sorted or can be sorted in linear time.
- Prim's Algorithm is preferred when the graph is dense i.e. when there are large number of edges in the graph like $E = O(V^2)$ because we do not have to pay much attention to the cycles by adding an edge as we primarily deal with the vertices in Prim's Algorithm.

Will both Prim's Algorithm and Kruskal's Algorithm always produce the same Minimum Spanning Tree (MST) for any given graph?

Solution-

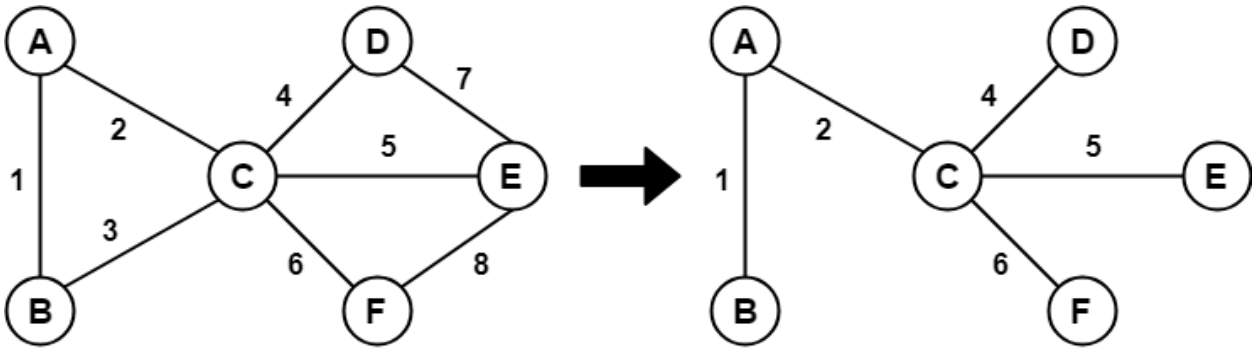
The following two cases are possible-

Case-01: When all the edge weights are distinct-

If all the edge weights are distinct, then both the algorithms are guaranteed to find the same i.e. unique MST.

Example-

Consider the following example-



Minimum Spanning Tree (MST)

(Cost = 18 units)

The application of both the algorithms on the above graph will produce the same MST as shown.

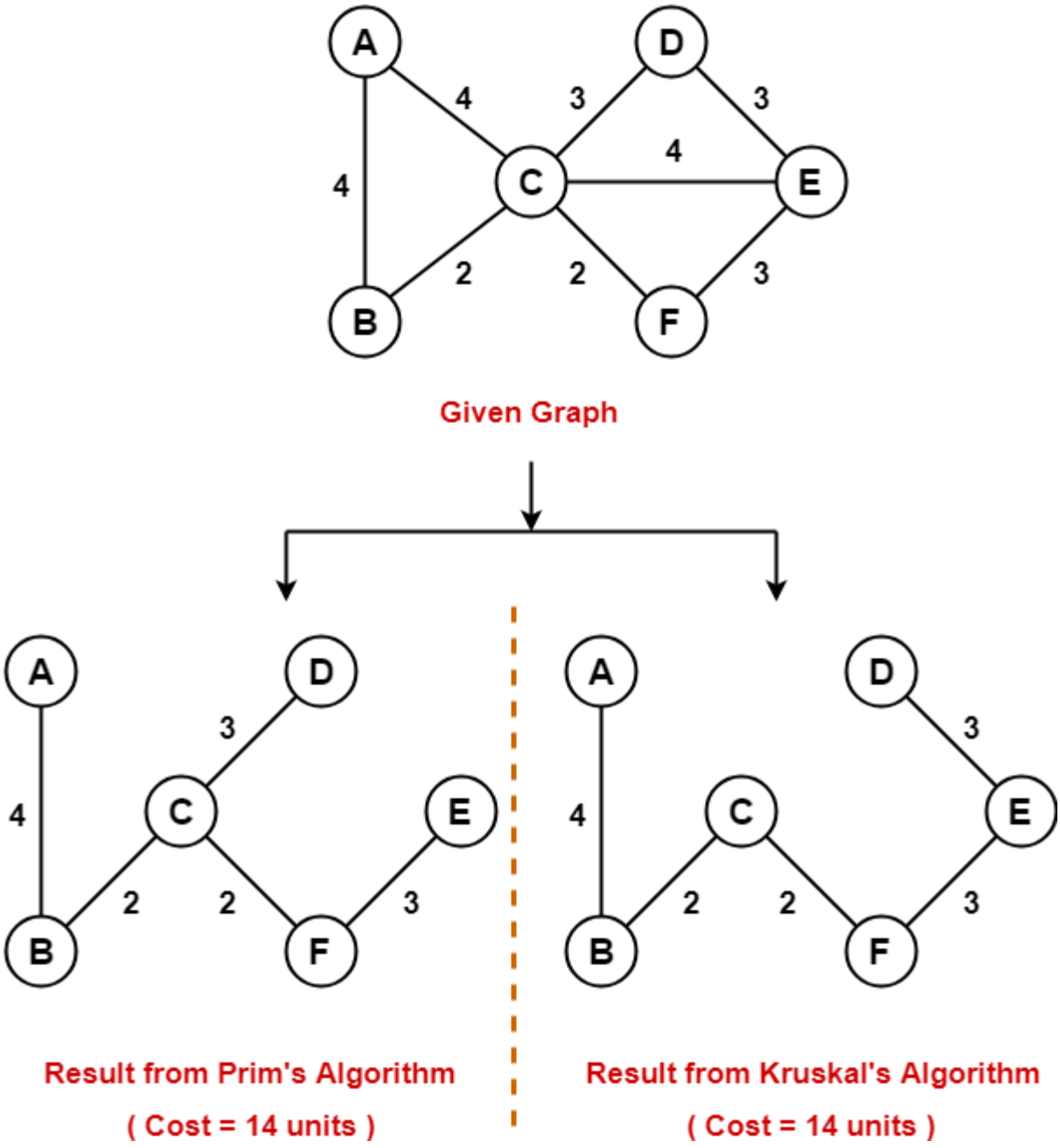
This example clearly illustrates when all the edge weights are distinct, both the algorithms always produces the same MST having the same cost as shown.

Case-02: When all the edge weights are not distinct-

If all the edge weights are not distinct, then both the algorithms may not always produce the same i.e. unique MST but the cost of the MST produced would always be same in both the cases.

Example-

Consider the following example-



This example clearly illustrates when in the given graph, all the edge weights are not distinct, different MSTs could be produced by both the algorithms as shown but the cost of the resulting MSTs from both the algorithms would always be same.

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Algorithm

To solve the problem follow the below idea:

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

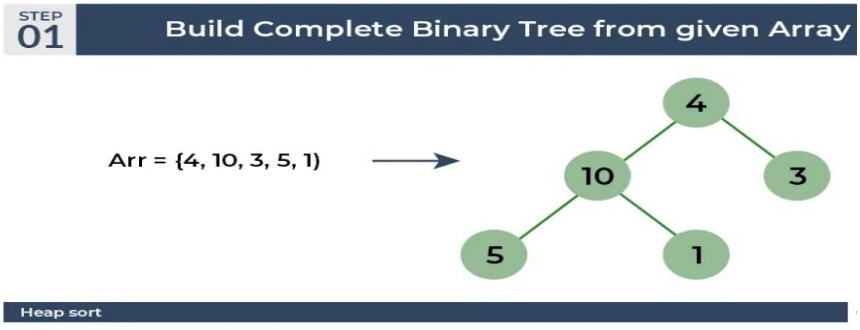
- *Build a heap from the given input array.*
- *Repeat the following steps until the heap contains only one element:*
    - *Swap the root element of the heap (which is the largest element) with the last element of the heap.*
    - *Remove the last element of the heap (which is now in the correct position).*
    - *Heapify the remaining elements of the heap.*
- *The sorted array is obtained by reversing the order of the elements in the input array.*

Detailed Working of Heap Sort

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort.

Consider the array: arr[] = {4, 10, 3, 5, 1}.

***Build Complete Binary Tree:** Build a complete binary tree from the array.*

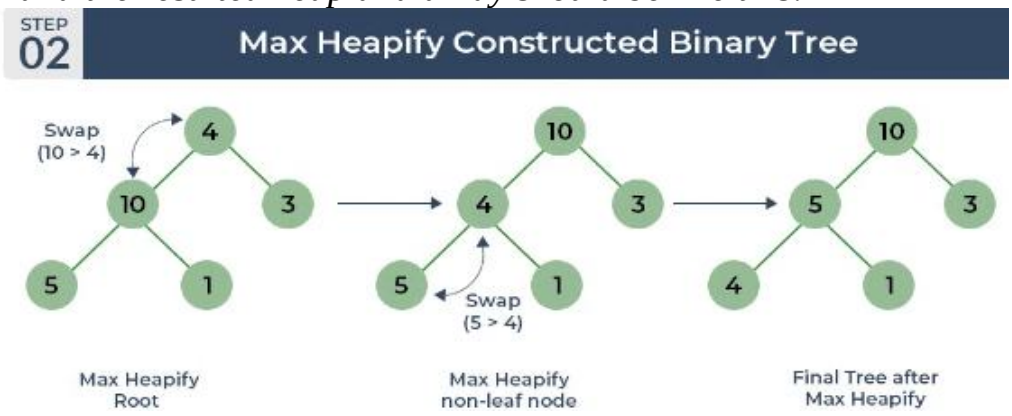


***Transform into max heap:** After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.*

- *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*

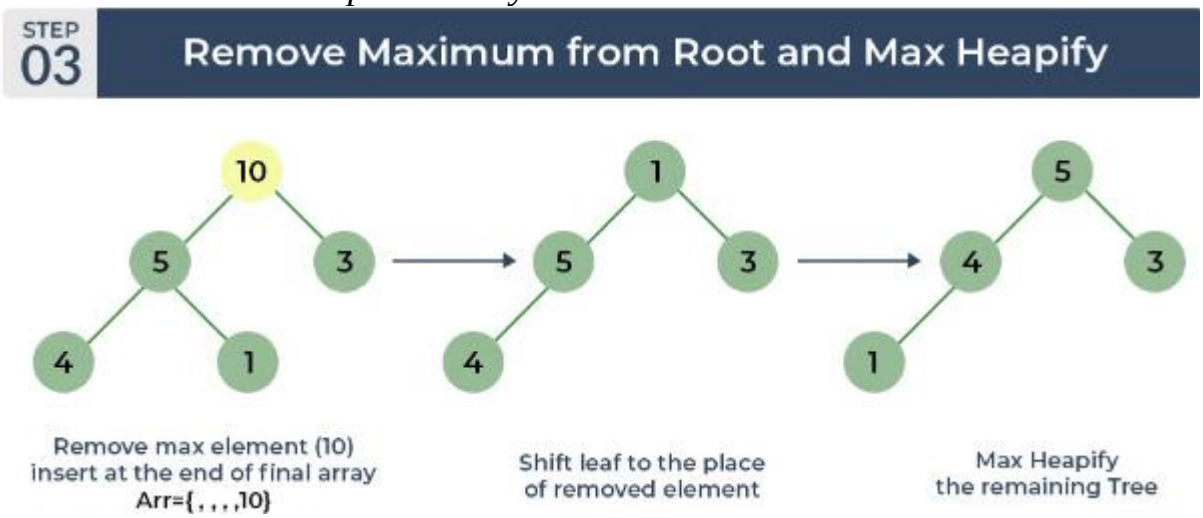
- *Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.*

- *Now, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:*

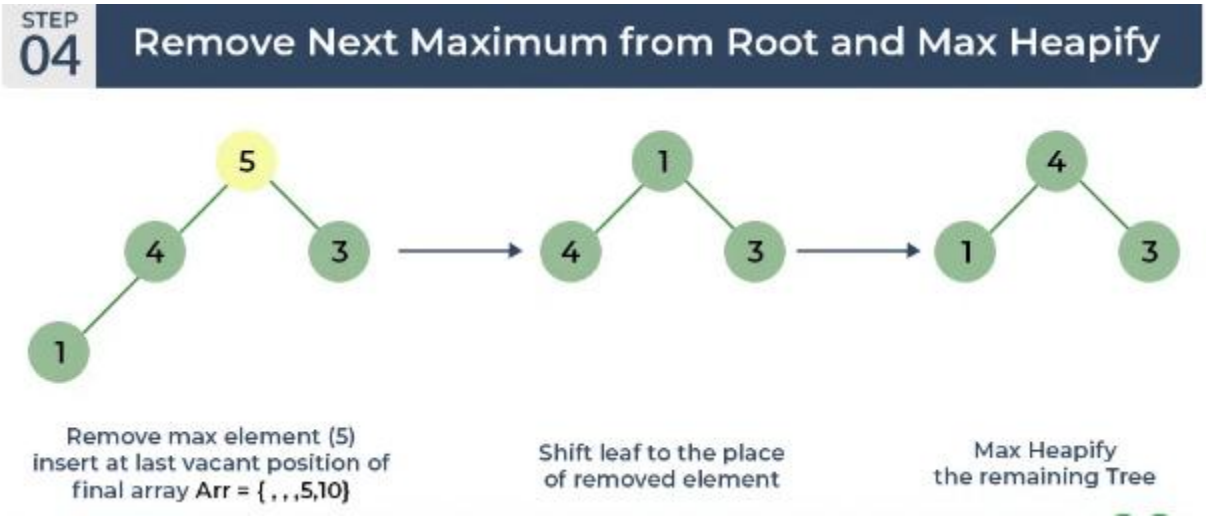


***Perform heap sort:** Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.*

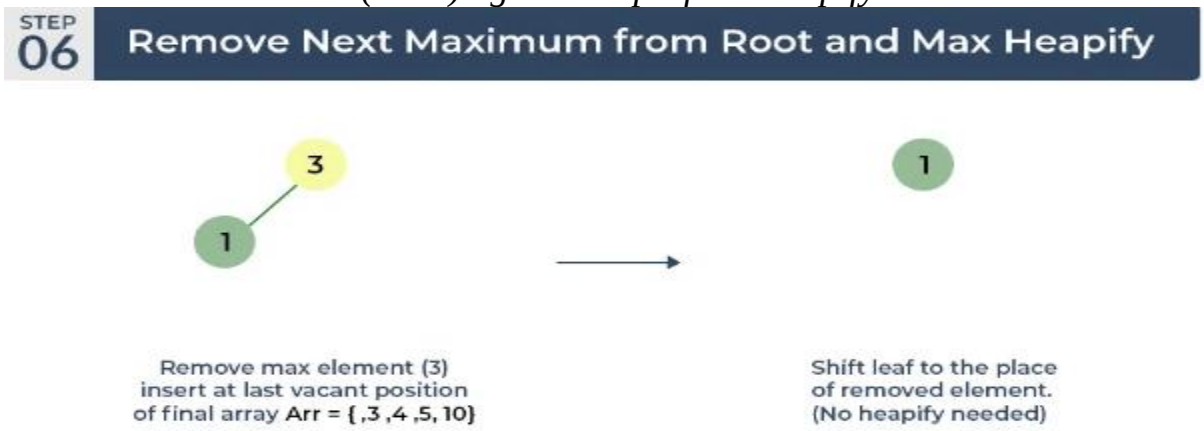
- *Delete the root element **(10)** from the max heap. In order to delete this node, try to swap it with the last node, i.e. **(1).** After removing the root element, again heapify it to convert it into max heap.*
    - *Resulted heap and array should look like this:*



- *Repeat the above steps and it will look like the following:*



- *Now remove the root (i.e. 3) again and perform heapify.*



Now when the root is removed once again it is sorted. and the sorted array will be like ***arr[] = {1, 3, 4, 5, 10}****.*

```
// Heap Sort in C
#include <stdio.h>
// Function to swap the position of two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
```

```
void heapify(int arr[], int N, int i)
{
    // Find largest among root,
    // left child and right child
    // Initialize largest as root
    int largest = i;
    // left = 2*i + 1
    int left = 2 * i + 1;
    // right = 2*i + 2
    int right = 2 * i + 2;
    // If left child is larger than root
    if (left < N && arr[left] > arr[largest])

        largest = left;

    // If right child is larger than largest
    // so far
    if (right < N && arr[right] > arr[largest])

        largest = right;

    // Swap and continue heapifying
    // if root is not largest
    // If largest is not root
    if (largest != i) {

        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{
int i;
    // Build max heap
    for (i = N / 2 - 1; i >= 0; i--)
```

```
            heapify(arr, N, i);

        // Heap sort
        for (i = N - 1; i >= 0; i--) {

            swap(&arr[0], &arr[i]);

            // Heapify root element
            // to get highest element at
            // root again
            heapify(arr, i, 0);
        }
}
// A utility function to print array of size n
void printArray(int arr[], int N)
{
int i;
        for (i = 0; i < N; i++)
            printf("%d ", arr[i]);
        printf("\n");
}
// Driver's code
int main()
{
        int n,i;
        printf("enter size of array");
        scanf("%d",&n);
        int a[n];
        printf("enter elements");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        // Function call
        heapSort(a,n);
        printf("Sorted array is\n");
        printArray(a,n);
}
```

AVL Tree Data structure

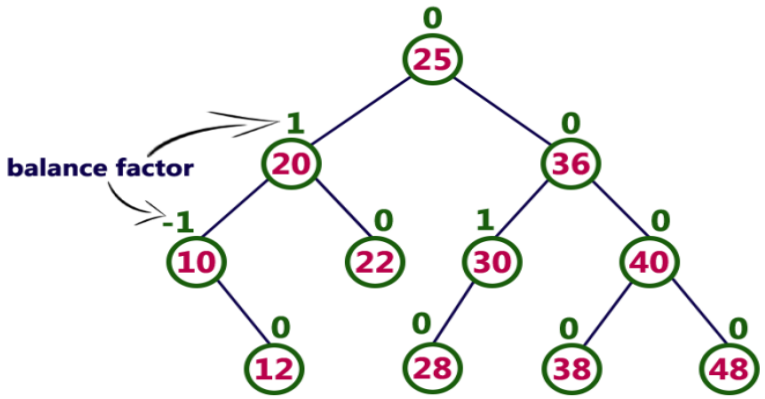
AVL tree is a height balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is ***either -1, 0 or +1***. In other words, 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.

An AVL tree is defined as follows...

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

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right**

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.

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

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)

AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

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

Depending upon the type of insertion, the Rotations are categorized into four categories.

SN	Rotation	Description
1	LL Rotation	The new node is inserted to the left sub-tree of left sub-tree of critical node.
2	RR Rotation	The new node is inserted to the right sub-tree of the right sub-tree of the critical node.
3	LR Rotation	The new node is inserted to the right sub-tree of the left sub-tree of the critical node.
4	RL Rotation	The new node is inserted to the left sub-tree of the right sub-tree of the critical node.

LL Rotations

LL rotation is performed when the node is inserted into the right subtree leading to an unbalanced tree. This is a single left rotation to make the tree balanced again –

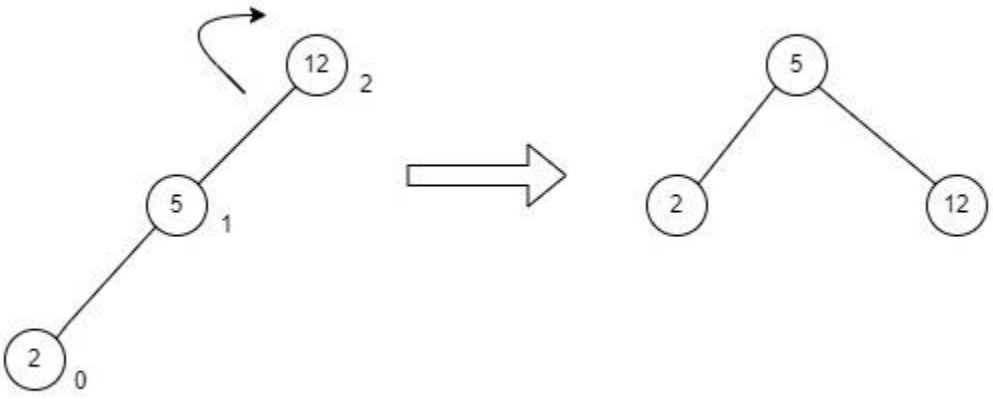


Fig : LL Rotation

The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node.

RR Rotations

RR rotation is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again −

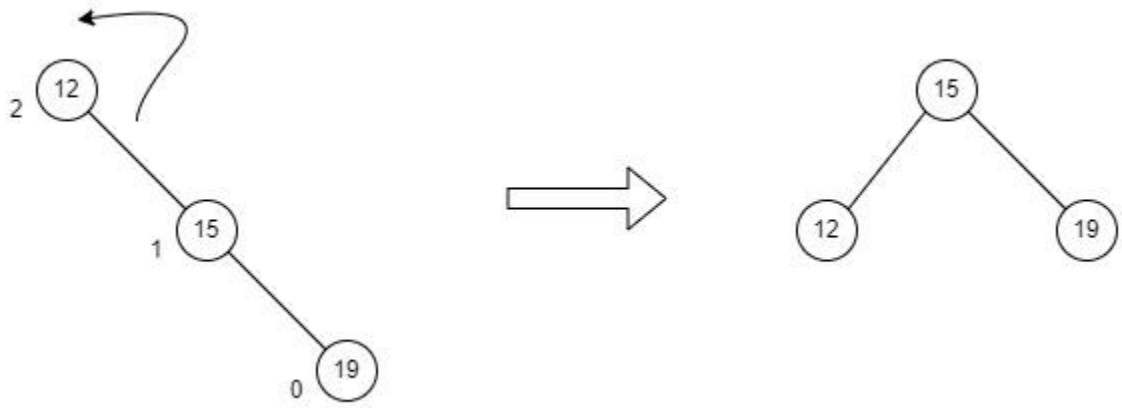


Fig : RR Rotation

The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node.

LR Rotations

LR rotation is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. There are multiple steps to be followed to carry this out.

- Consider an example with “A” as the root node, “B” as the left child of “A” and “C” as the right child of “B”.
- Since the unbalance occurs at A, a left rotation is applied on the child nodes of A, i.e. B and C.

- After the rotation, the C node becomes the left child of A and B becomes the left child of C.
- The unbalance still persists, therefore a right rotation is applied at the root node A and the left child C.
- After the final right rotation, C becomes the root node, A becomes the right child and B is the left child.

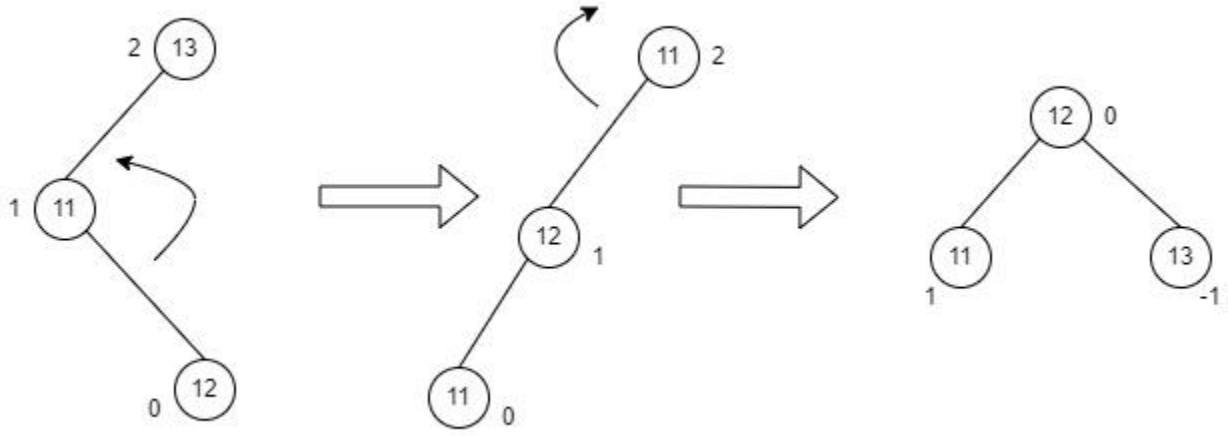


Fig : LR Rotation

RL Rotations

RL rotation is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation. There are multiple steps to be followed to carry this out.

- Consider an example with “A” as the root node, “B” as the right child of “A” and “C” as the left child of “B”.
- Since the unbalance occurs at A, a right rotation is applied on the child nodes of A, i.e. B and C.
- After the rotation, the C node becomes the right child of A and B becomes the right child of C.
- The unbalance still persists, therefore a left rotation is applied at the root node A and the right child C.
- After the final left rotation, C becomes the root node, A becomes the left child and B is the right child.

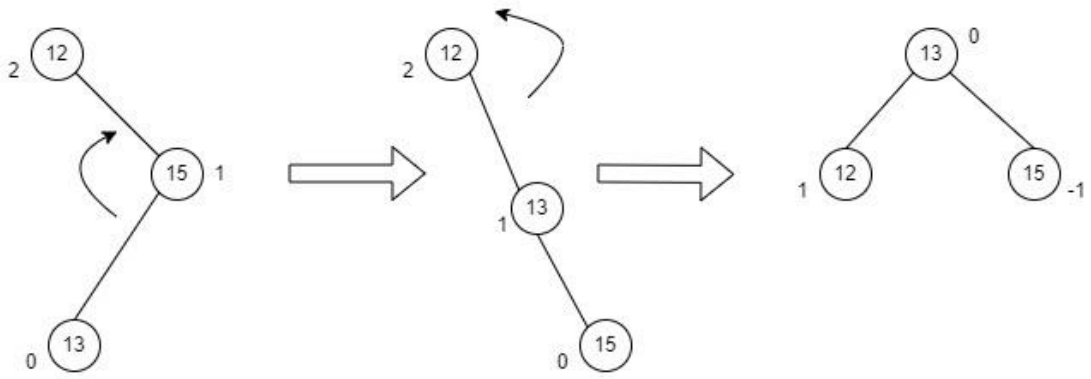


Fig : RL Rotation

Operations on an AVL

Tree 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 AVL tree is similar to search operation in Binary search tree.

AVL –Search Algorithm

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

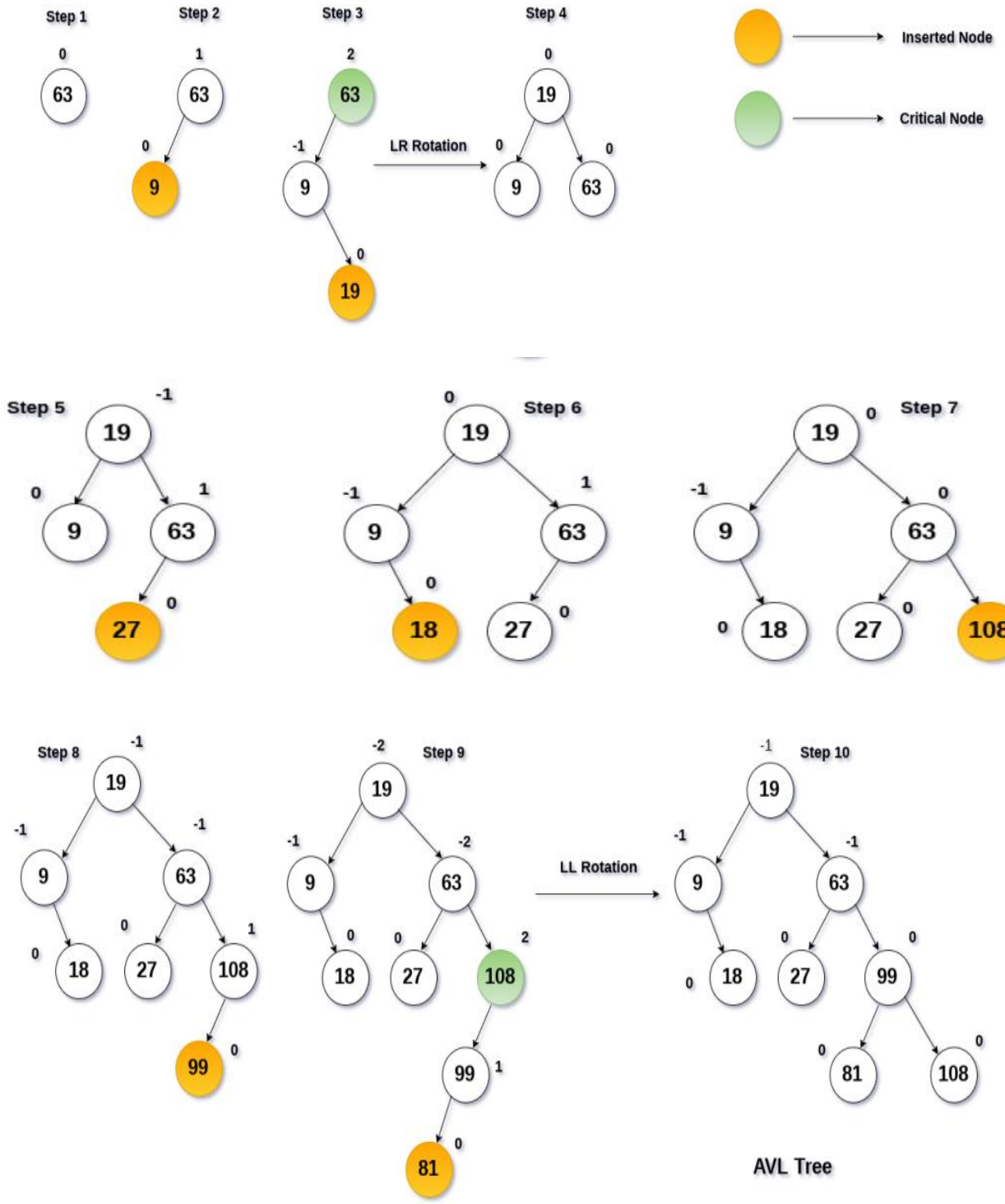
In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows.

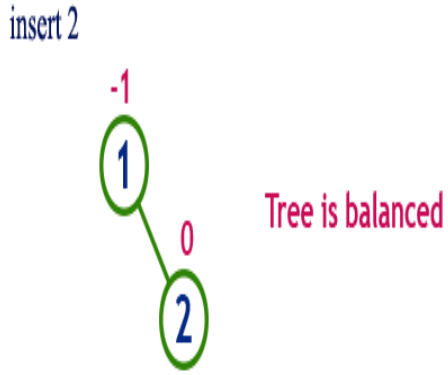
- **Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.**
- **Step 2 - After insertion, check the Balance Factor of every node.**
- **Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.**
- **Step 4 - If the Balance Factor of any 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.**

Example: Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81

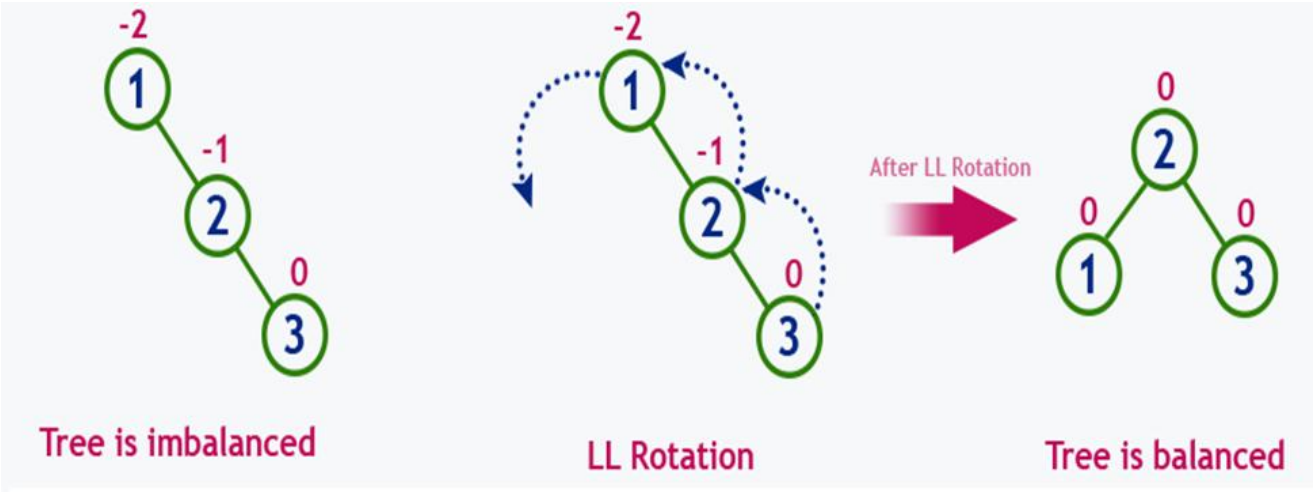
All the elements are inserted in order to maintain the order of binary search tree.

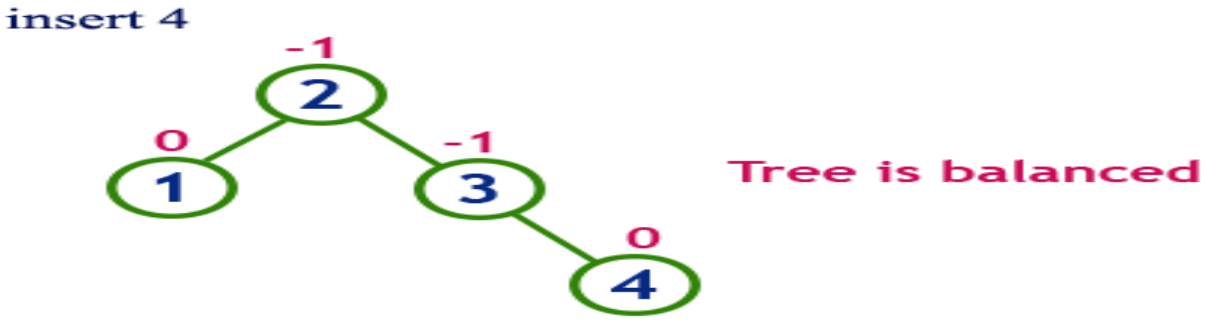

Step 1
Step 2
Step 3
Step 4
LR Rotation
Inserted Node
Critical Node
Step 5
Step 6
Step 7
Step 8
Step 9
Step 10
LL Rotation
AVL Tree

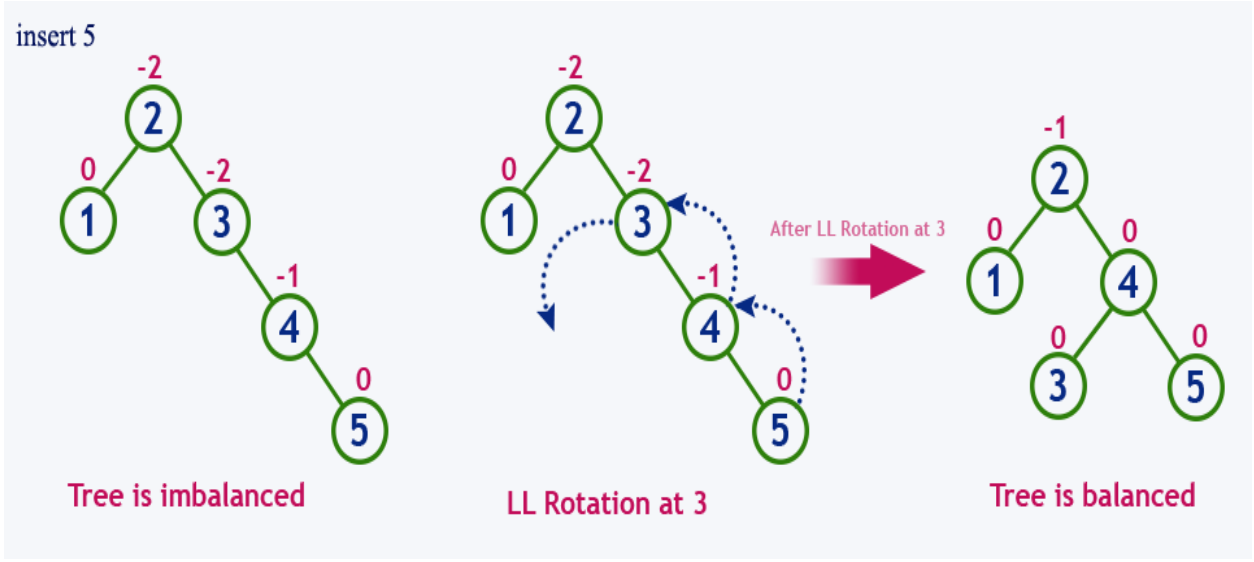
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

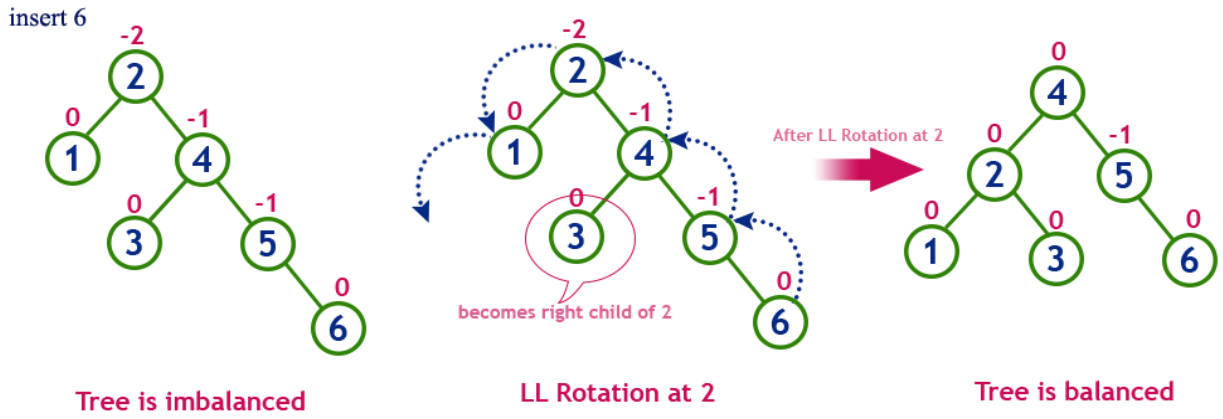


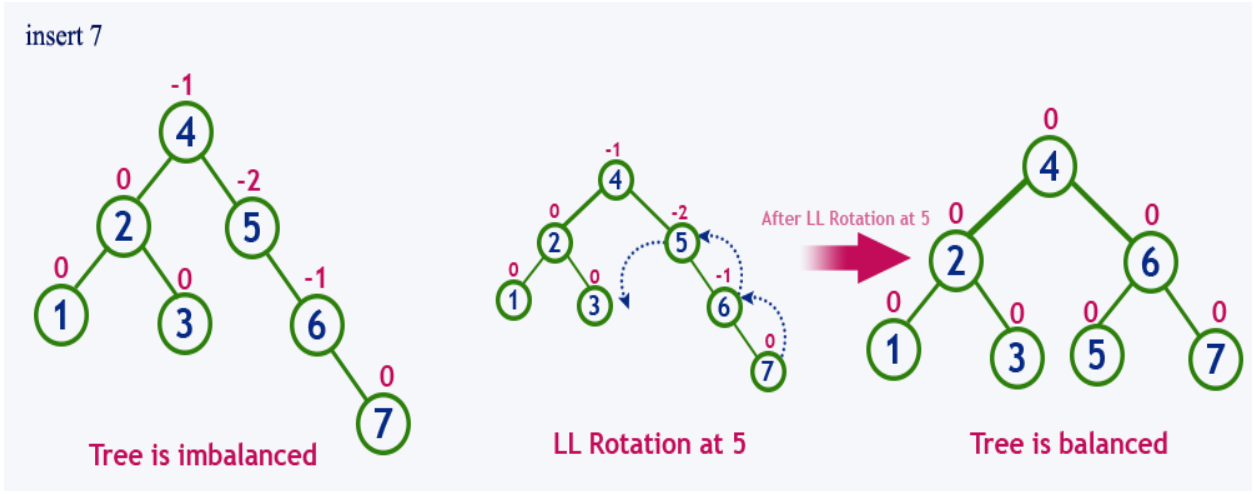
Insert 3

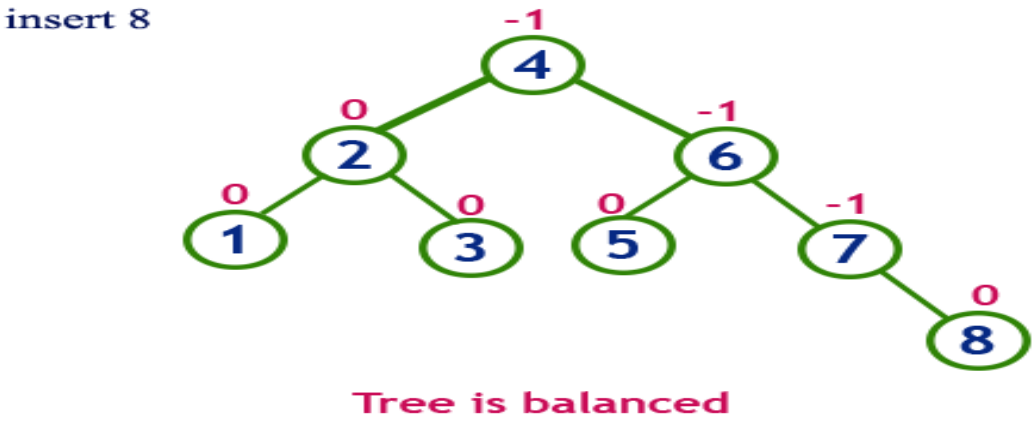




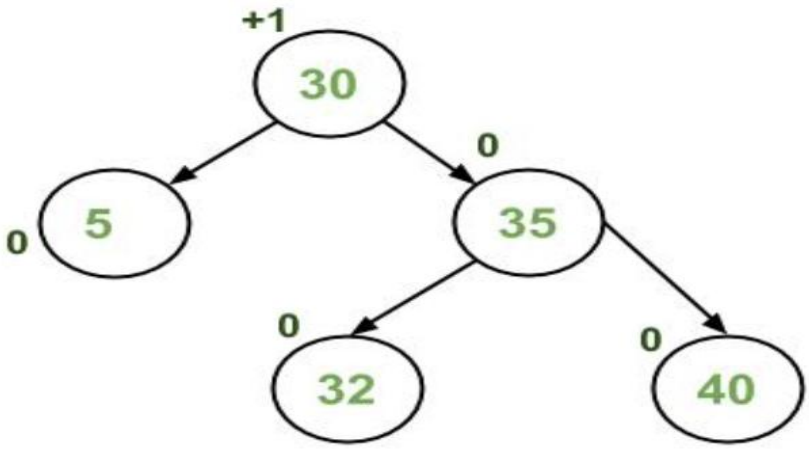

insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced

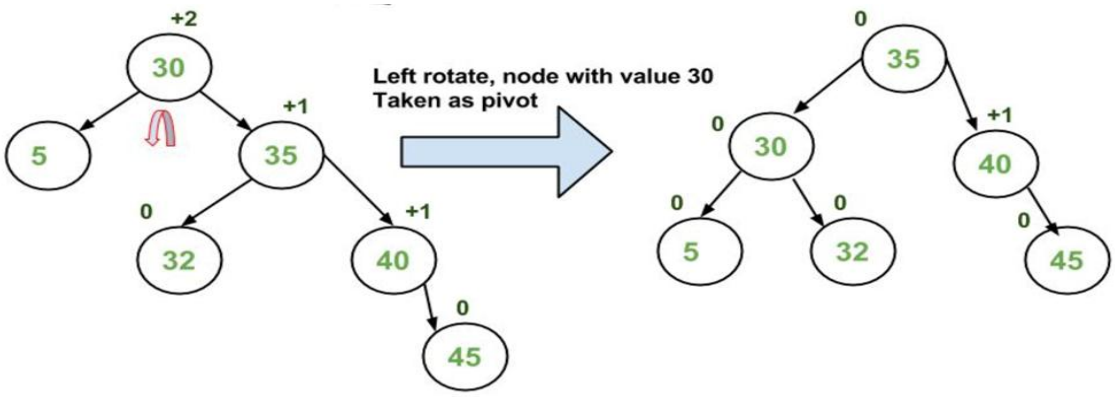

insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced



Example 3: Consider the following AVL tree and insert 45 into AVL tree



Solution:



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.

AVL – Deletion Algorithm

Step1- Find the node to be deleted using **search** operation

Step 2 - **Delete** the node using one of the three BST deletion cases.

Case 1: Deleting a Leaf node

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Step 3 - After deletion, check the **Balance Factor** of every node.

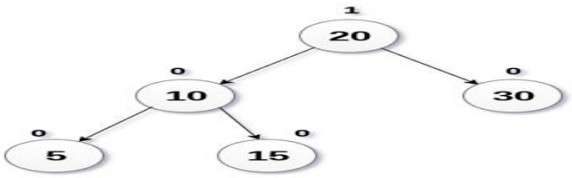
Step 4 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 5 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be **unbalanced**. In this case, perform **suitable Rotation** to make it balanced and go for next operation.

If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

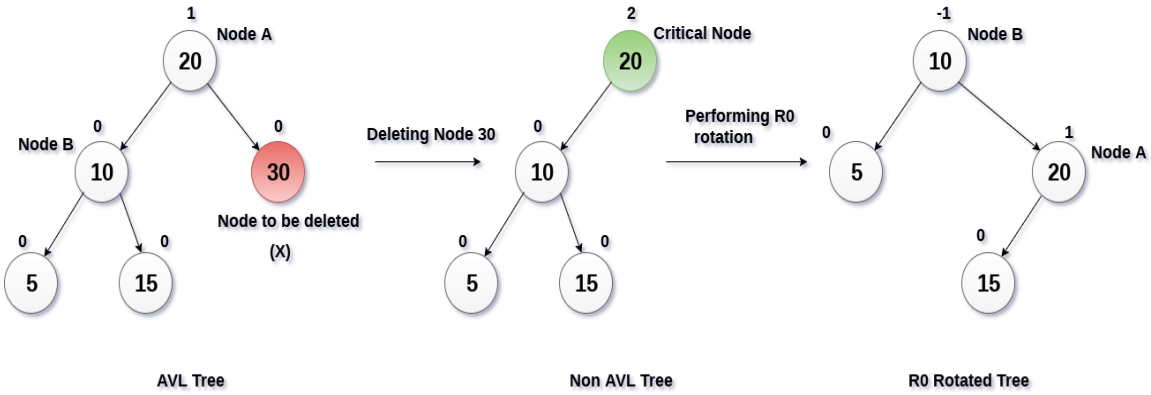
Example:

Delete the node 30 from the AVL tree shown in the following image.

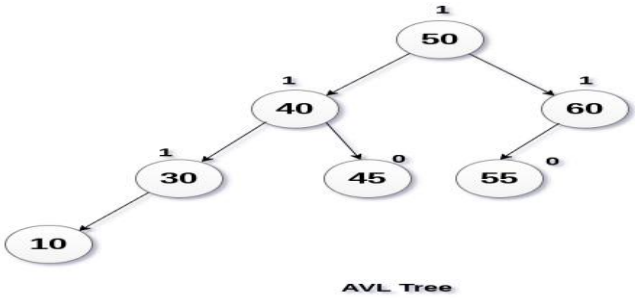


Solution

In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.

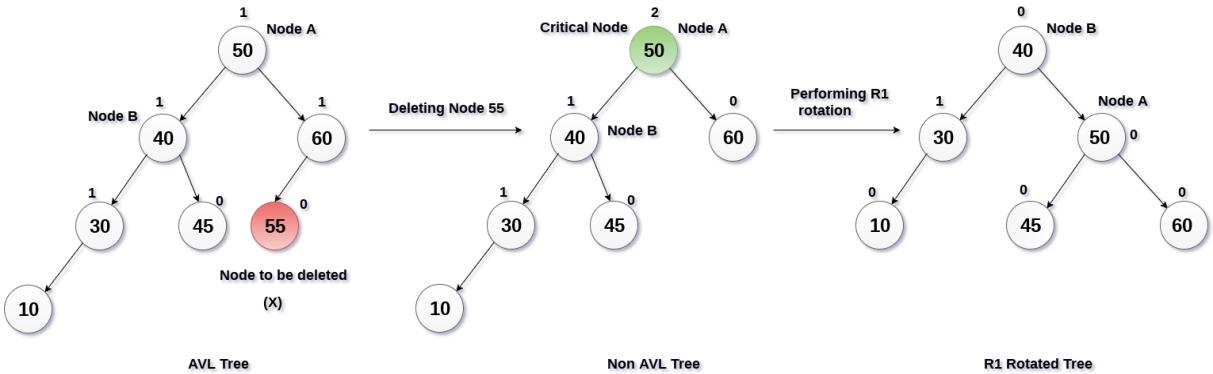


Delete Node 55 from the AVL tree shown in the following image.



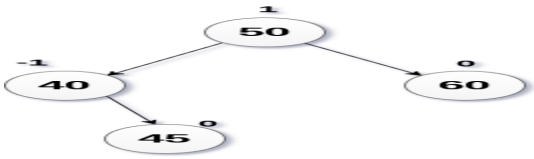
Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).



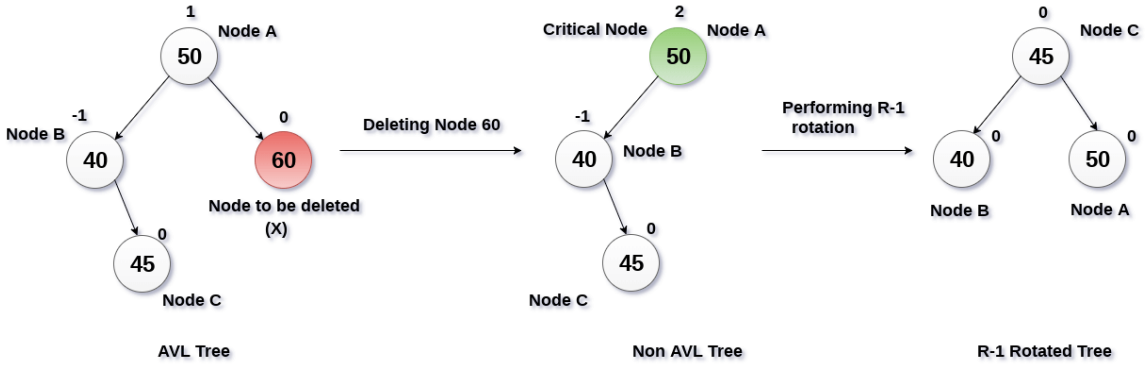
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



Red - Black Tree Data structure

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

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

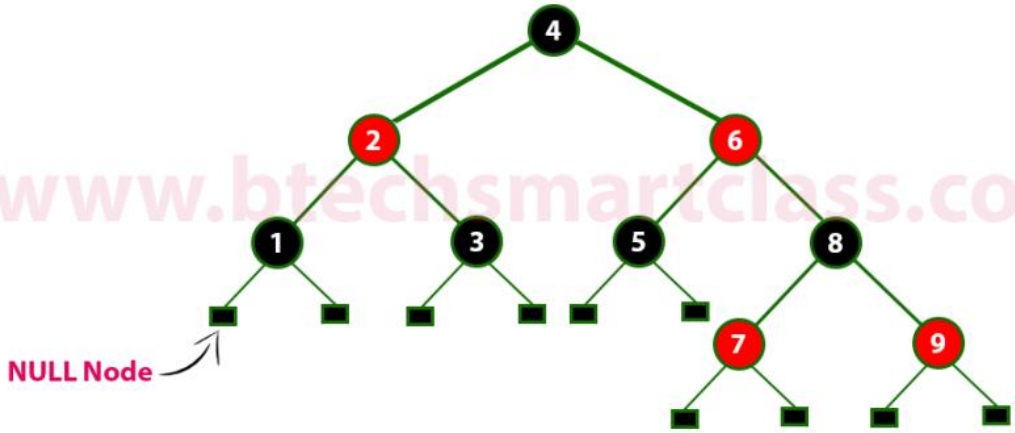
In Red Black Tree, the color of a node is decided based on the properties of Red Black Tree.

Every Red Black Tree has the following properties. Properties of Red Black Tree

- 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 (e.i. NULL node) must be colored BLACK.

Example Following is a Red Black Tree which is created by inserting numbers from 1 to 9.



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

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 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 parent node'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. Check parent is black stop otherwise 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)

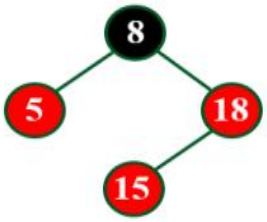
Tree is not Empty. So insert newNode with red color.

insert (5)

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

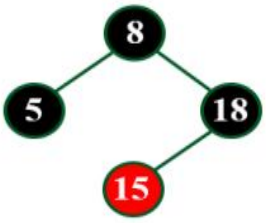
insert (15)

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.

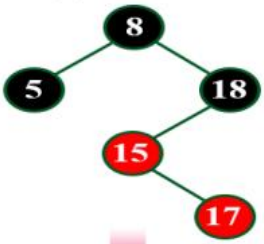
After RECOLOR



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

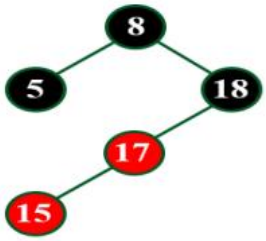
insert (17)

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

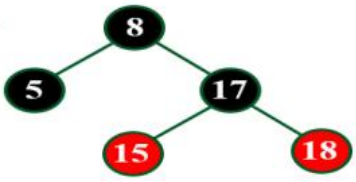


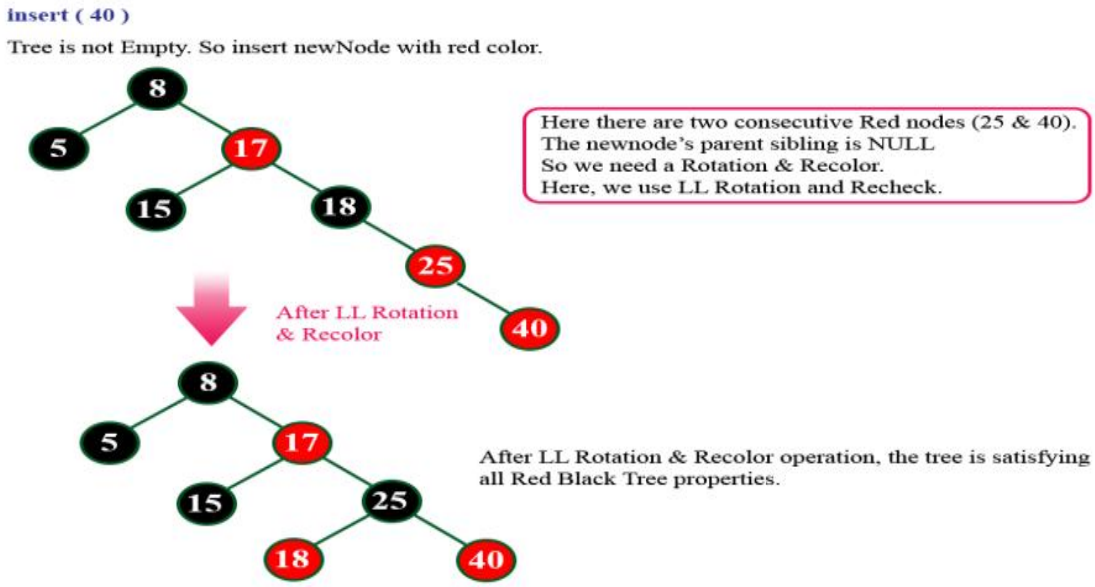
Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

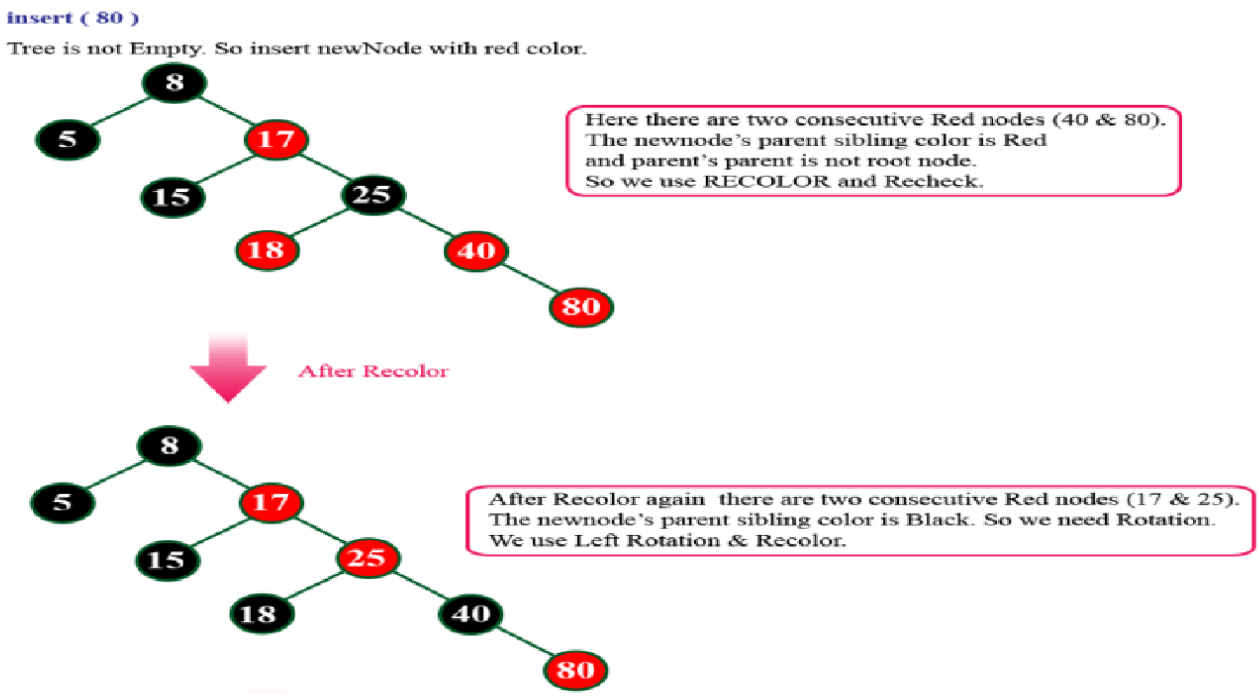
After Left Rotation



After Right Rotation & Recolor







Practice : Construct Red Black Tree which is created by inserting numbers from 1 to 9

Red Black tree deletion

Step 1: Perform BST deletion

step2 : Identify the following cases

Case 1: if node to be deleted is red just delete it

Case 2: if root is DB ,just remove DB

Case 3: if DB 's sibling is black & both it's children are black

-Remove DB and black to parent 'P'

3.1 if p is red , it become black

3.2 if p is black it becomes double black

- make sibling red

- if still DB exists , apply other cases

Case 4: if DB's sibling is red

- swap color's of P and sibling
- perform rotation in DB direction
- reapply cases

case 5: DB's sibling is black, sibling's child who is far from DB is black , but near child to DB is red

- swap color of DB's sibling & sibling's child to who is near to DB
- rotate sibling in opposite direction to DB
- apply case 6

Case 6: DB's sibling is black, far child is red

-swap color of parent & sibling

-rotate parent in DB's direction

- remove DB

-change color of red child to black

Case #	Check condition	Action
1	If node to be delete is a red leaf node	Just remove it from the tree
2	If DB node is root	Remove the DB and root node becomes black.
3	(a) If DB's sibling is black, and (b) DB's sibling's children are black	(a) Remove the DB (if null DB then delete the node and for other nodes remove the DB sign) (b) Make DB's sibling red. (c) If DB's parent is black, make it DB, else make it black
4	If DB's sibling is red	(a) Swap color DB's parent with DB's sibling (b) Perform rotation at parent node in the direction of DB node (c) Check which case can be applied to this new tree and perform that action
5	(a) DB's sibling is black (b) DB's sibling's child which is far from DB is black (c) DB's sibling's child which is near to DB is red	(a) Swap color of sibling with sibling's red child (b) Perform rotation at sibling node in direction opposite of DB node (c) Apply case 6
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red (remember this node)	(a) Swap color of DB's parent with DB's sibling's color (b) Perform rotation at DB's parent in direction of DB (c) Remove DB sign and make the node normal black node (d) Change colour of DB's sibling's far red child to black.

Example 1: Delete 30 from the RB tree in fig. 3

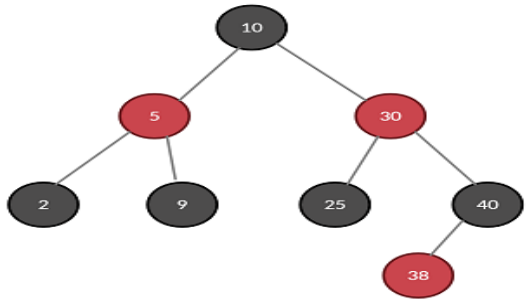


Fig. 3: Initial RB Tree

First search for 30, once found perform BST deleti. For a node with value '30', find either the maximum of the left subtree or a minimum of the right subtree and replace 30 with that value. This is BST deletion (in short).

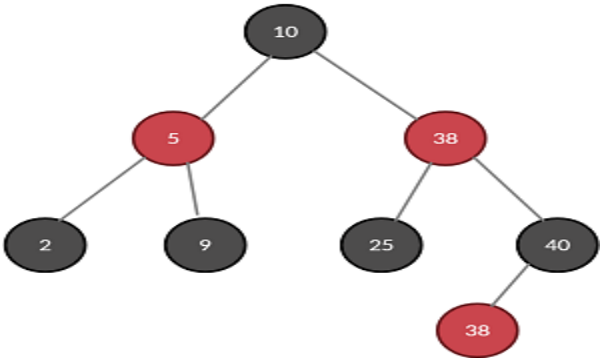


Fig. 4: RB Tree after replacing 30 with min element from right subtree

The resulting RB tree will be like one in fig. 4. Element 30 is deleted and the value is successfully replaced by 38. But now the task is to delete duplicate element 38.

case 1 is satisfied by this tree.

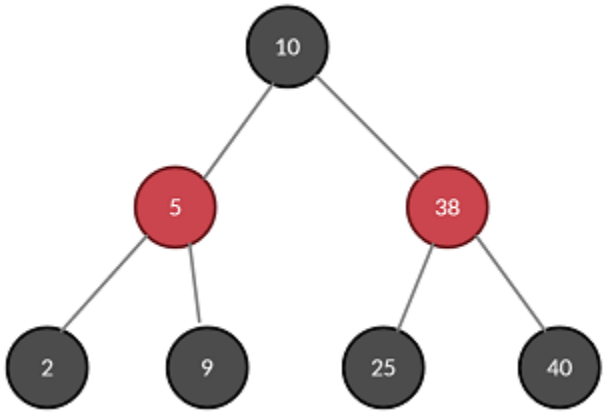


Fig. 5: After removing the red leaf node

Since node with element 38 is a *red* leaf node, remove it and the tree looks like the one in fig. 5.Observe that if you perform correct actions, the tree will still hold all the properties of the RB tree.

Example 2: Delete 15 from RB tree in fig. 6

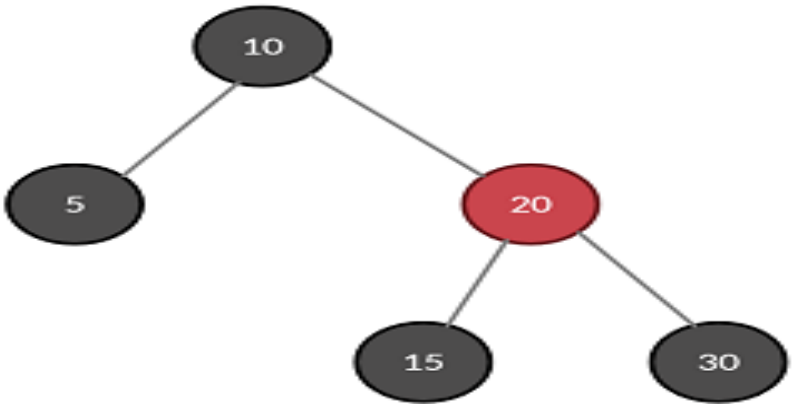


Fig. 6: Initial RB Tree

15 can be removed easily from the tree (BST deletion). In the case of RB trees, if a leaf node is deleted you replace it with a **double black (DB)** nil node (fig. 7). It is represented by a double circle.

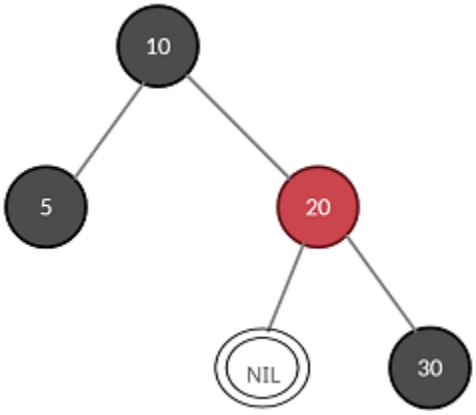


Fig. 7: NIL node added in place of 15

The entire problem is now drilled down to get rid of this bad boy, DB, via some actions. Go back to our rule book (table) and case 3 fits perfectly.

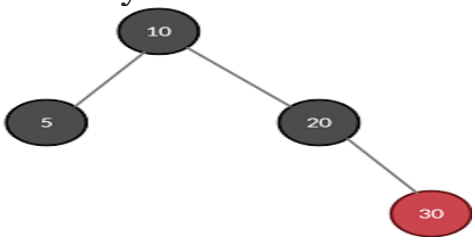


Fig. 8: NIL Node removed after applying actions

In short, remove DB and then swap the color of its sibling with its parent (fig. 8).

Example 3: Delete '15' from fig. 9(A).

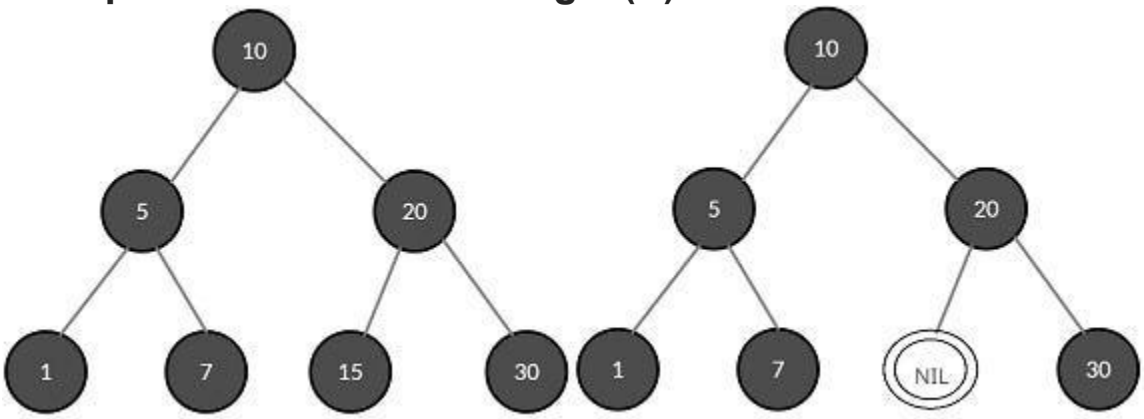


Fig. 9: (A) Initial RB Tree, (B) NIL node added in place of 15

Delete node with value 15 and, as a rule, replace it with DB nil node as shown. Now, DB's sibling is black and sibling's both children are also black (don't forget the hidden NIL nodes!), it satisfies all the conditions of case 3. Here,

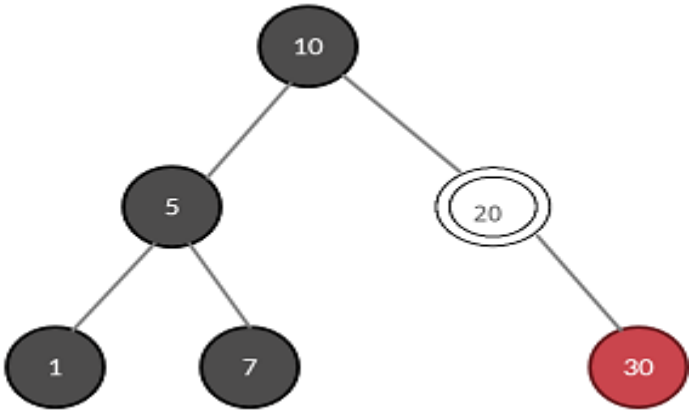


Fig. 10: RB Tree after case 3 is applied

DB's parent is 20

1. DB's parent is *black*
2. DB's sibling is 30

With these points in mind perform the actions and you get an RB tree as in fig. 10.20 becomes DB and hence the problem is not resolved yet. Reapply case 3 (check yourself how conditions match).

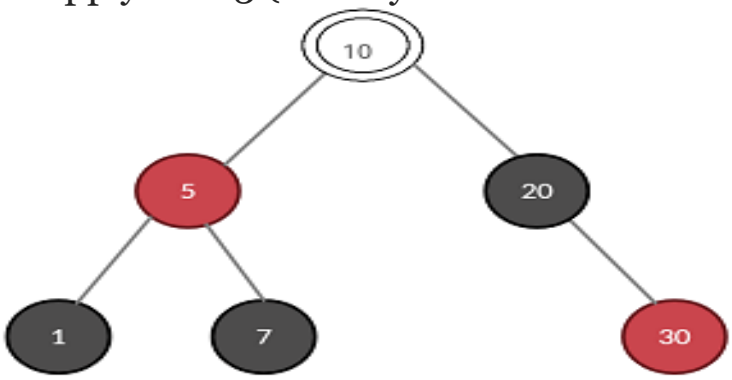


Fig. 11: RB Tree after case 3 is applied

The resulting tree looks like the one in fig. 11.

The DB still exist . Recheck which case will be applicable.

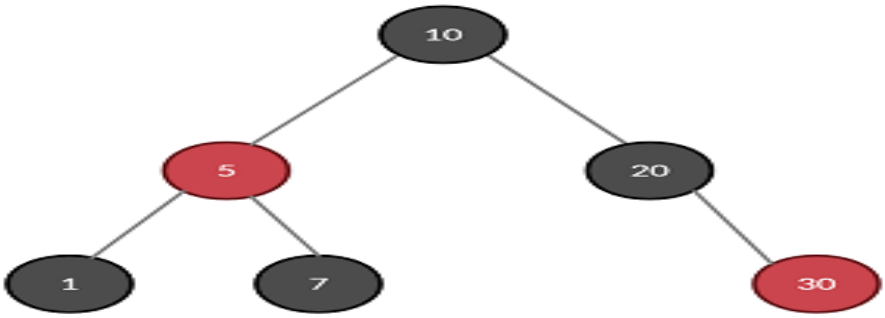


Fig. 12: NIL Node removed after applying actions

It's case 2, the simplest of all!The root resolved DB and becomes a *black* node. And you're done deleting 15 successfully.

Example 4: Delete ‘15’ from fig. 13

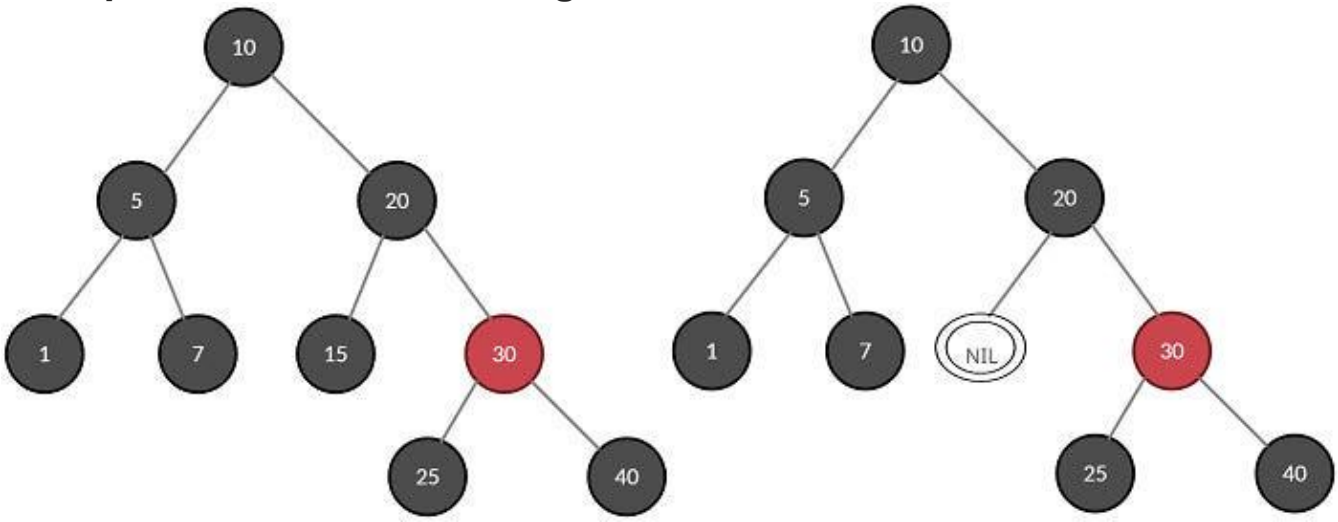


Fig. 13: (A) Initial RB Tree, (B) NIL node added in place of 15First, Search 15 as per BST rules and then delete it. Second, replace deleted node with DB NIL node as shown in fig. 13 (B).

DB’s sibling is *red*. Clearly, case 4 is applicable.

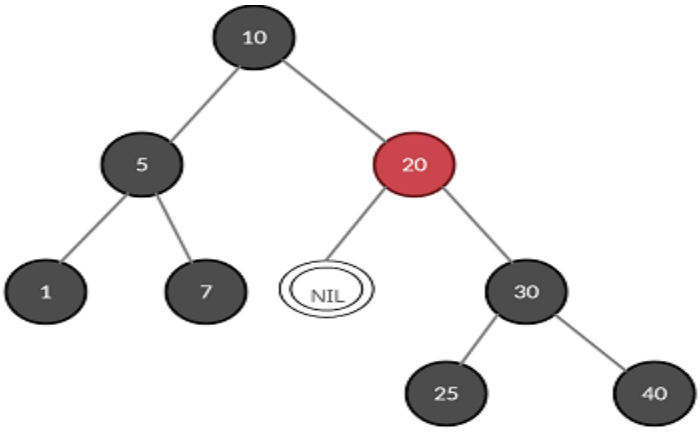


Fig. 14: RB Tree after case 4 is applied

(a) Swap DB’s parent’s color with DB’s sibling’s color.. The tree looks like fig. 14.

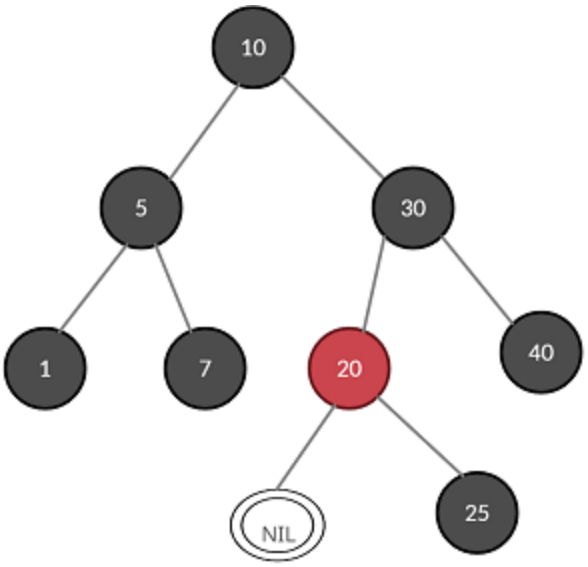


Fig. 15

(b) Perform rotation at parent node in direction of DB. The tree becomes like the one in fig. 15. DB is still there
(c) Check which case can be applied in the current tree. And got it, case 3.

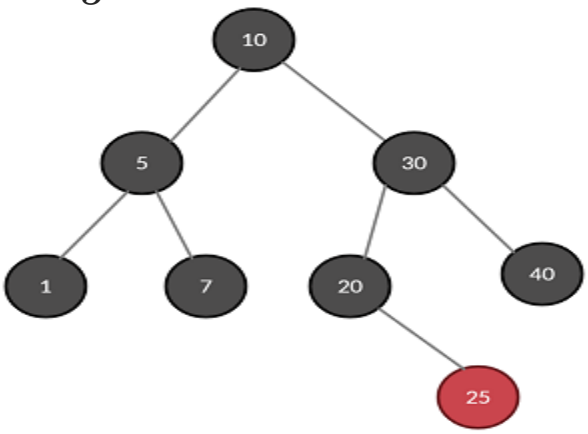


Fig. 16: NIL Node removed after applying actions

(d) Apply case 3 as explained and the RB tree is free from the DB node as shown in fig. 16.

Example 5: Delete ‘1’ from fig. 17

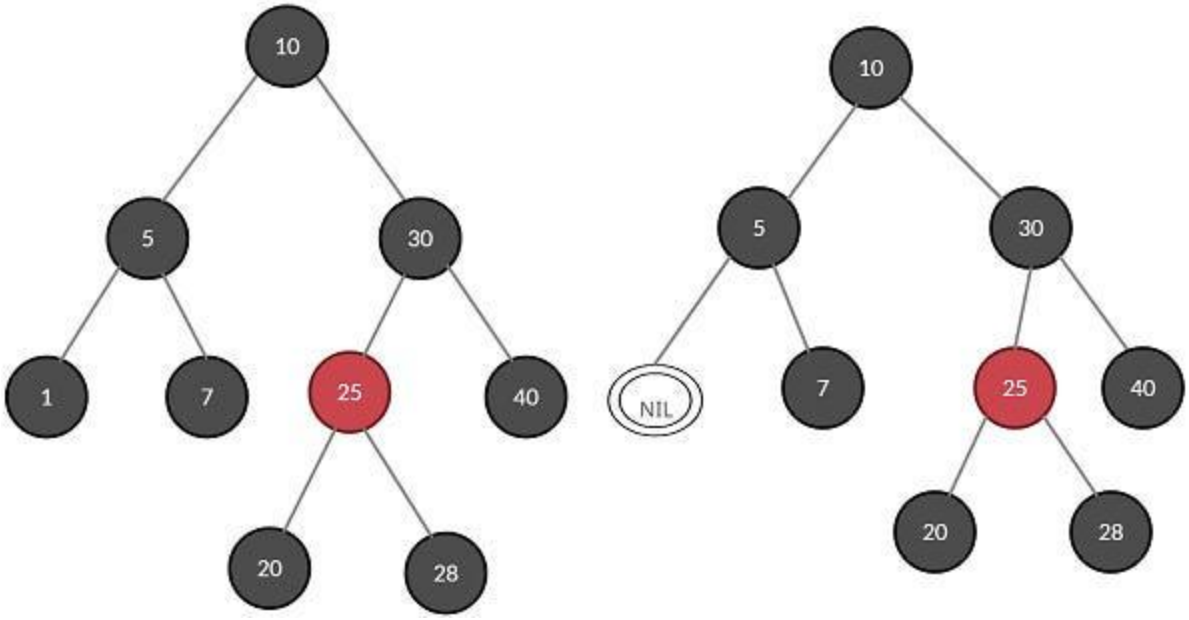


Fig. 17: (A) Initial RB Tree, (B) NIL node added in place of 1

Perform the basic preliminary steps- delete the node with value 1 and replace it with DB NIL node as shown in fig. 17(B). Check for the cases which fit the current tree and it’s case 3(DB’s sibling is *black*).

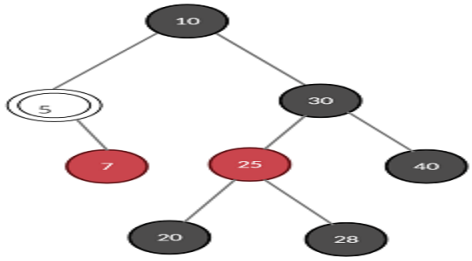


Fig. 18: RB Tree after case 3 is applied

Apply case 3 as and ,we get the tree as shown in fig. 18
Node 5 has now become a *double black* node. Search for cases that can be applied and case 5 seems to fit here (not case 3).

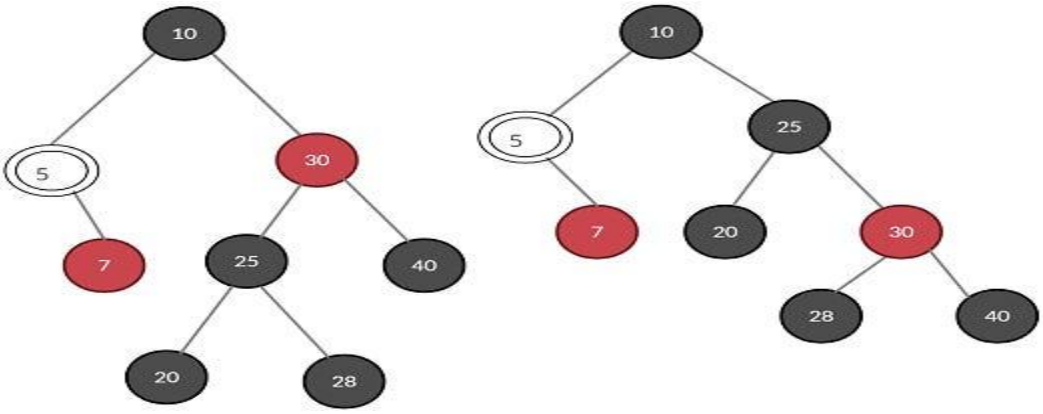


Fig. 19: (A) Tree after swapping colors of 30 & 25 (B) Tree after rotation

Case 5 is applied as follows-
(a) swap colors of nodes 30 and 25 (fig. 19(A))
(b) Rotate at sibling node in the direction opposite to the DB node.
Hence, perform right rotation at node 30 and the tree becomes like fig. 19 (B).
The *double black* node still haunts the tree! Re-check the case that can be applied to this tree and we find that case 6 (don't fall for case 3) seems to fit.

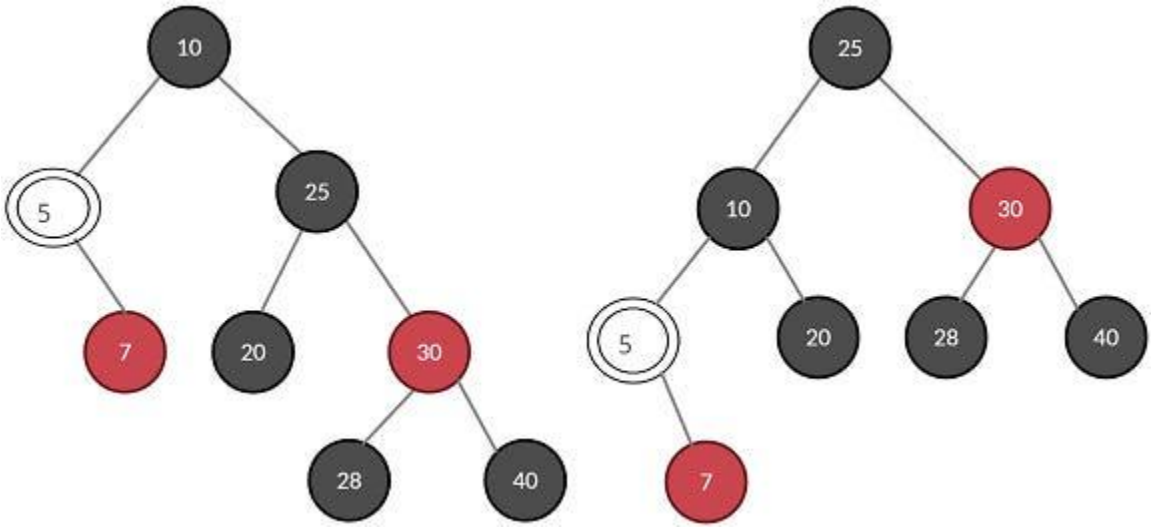


Fig. 20

Apply case 6 as follow-
(a) Swap colors of DB's parent with DB's sibling.

(b) Perform rotation at DB's parent node in the direction of DB (fig, 20(B)).

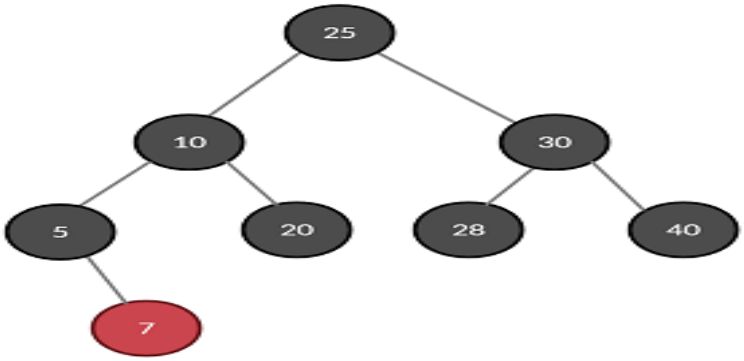


Fig. 21: NIL Node removed after applying actions

(c) Change DB node to *black* node. Also, change the color of DB's sibling's far-*red* child to black and the final RB tree .

Splay Tree Data structure

Splay tree is an another variant of binary search tree. In splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

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.

In splay tree, every operation is performed at root of the tree. All the operations in splay tree are involved with a common operation called **"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 root of the tree. The search operation in 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 (Right Rotation LL)
2. Zag Rotation (Left Rotation RR)
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation(Zig followed by Zag)
6. Zag - Zig Rotation(Zag followed by Zig)

Examples

1. Zig Rotation

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

3.Zag Rotation

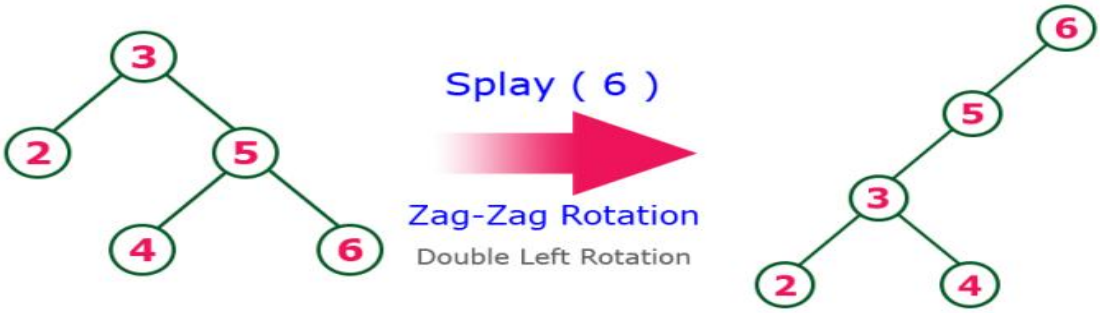
The Zag Rotation in splay tree is similar to the single left rotation (RR) in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

3.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..

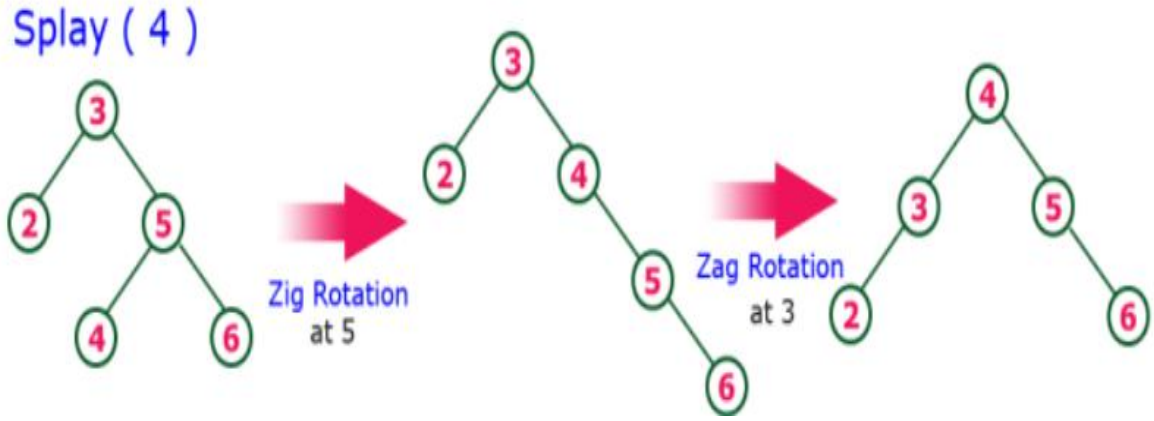
4. Zag-Zag Rotation

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..



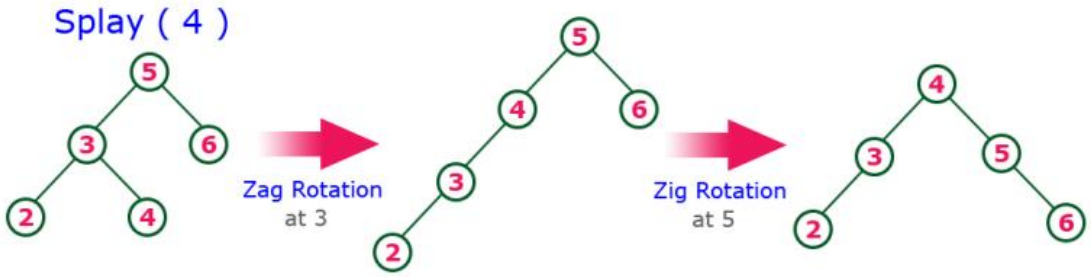
5. Zig-Zag Rotation

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.



6. 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.



Every Splay tree must be a binary search tree but it is need not to be balanced tree

SEARCH OPERATION IN SPLAY TREE

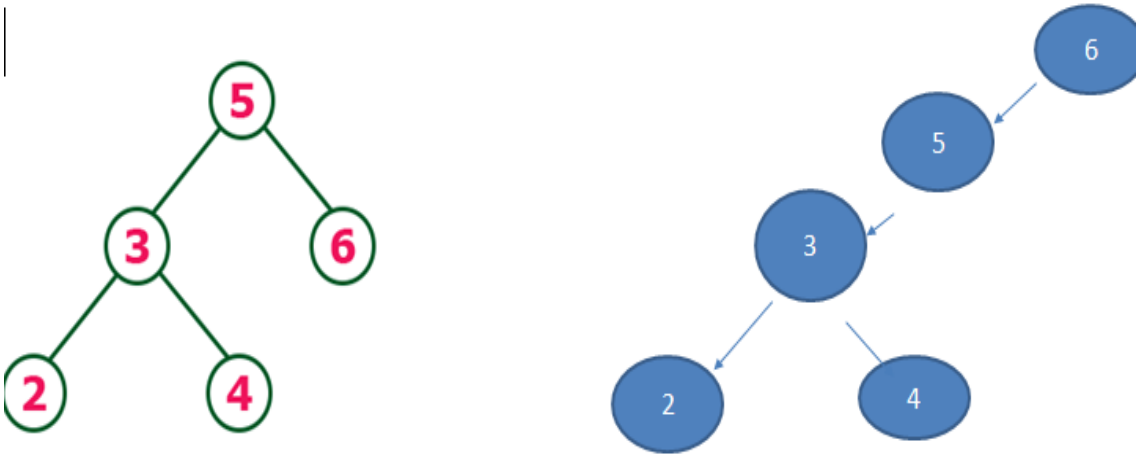
The search operation in Splay tree does the standard Binary Search Tree search.

In addition to search, it also splays (move the node to the root).

If the search is successful, then the node that is found is splayed and becomes the new root. If the search is unsuccessful then the last node traversed in the tree is splayed and becomes the root node.

If the search node is the root node then simply return the root node.

EXAMPLE: SEARCH THE ELEMENT 5 **SEARCH ELEMENT 7**

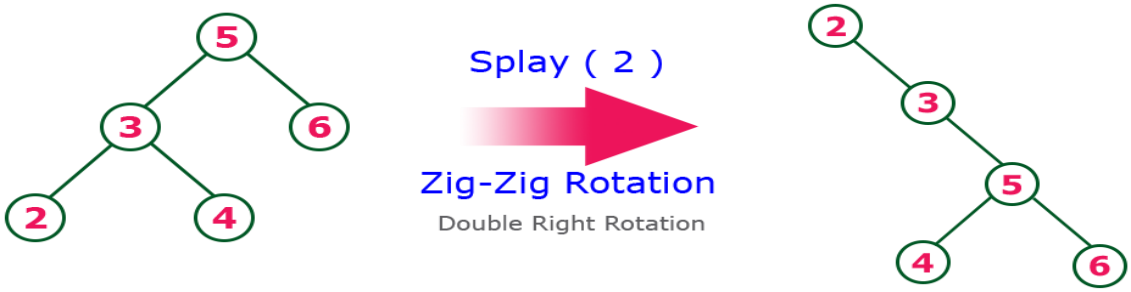


If the search node is the left child of the root node then do the Zig rotation.

EXAMPLE : SEARCH THE ELEMENT 3

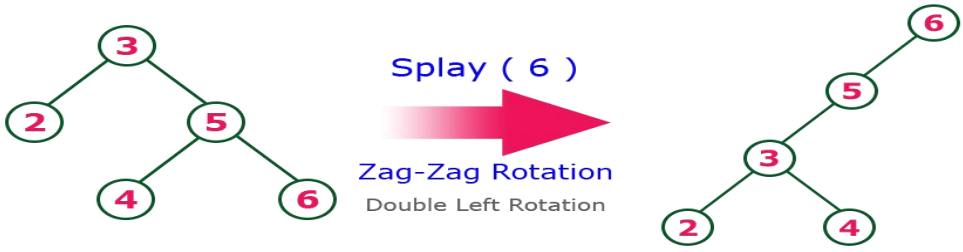
If the search node is present on the left child of its parent node as well as the left child of its grand parent node then we have to perform Zig-Zig notation.

EXAMPLE: SEARCH THE ELEMENT 2



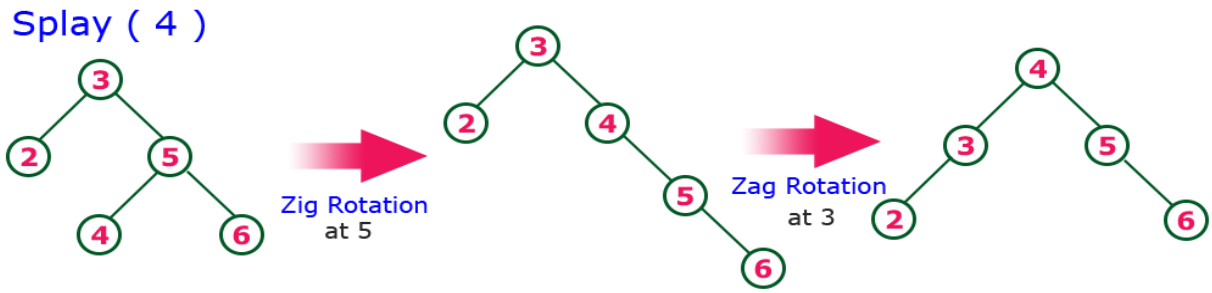
If the search node is present on the right child of the parent node as well as the right child of the grand parent node then we perform Zag-Zag notation.

EXAMPLE : SEARCH THE ELEMENT 6



If the search node is left child of its parent node and parent node is right child of grand parent node then we perform Zig-Zag notation.

EXAMPLE: SEARCH THE ELEMENT 4



Insertion in Splay Trees

The insert operation is similar to BST insert in addition, newly inserted key is made new root.

The insertion operation in Splay tree is performed using following steps.

Step 1 - Check whether tree is Empty.

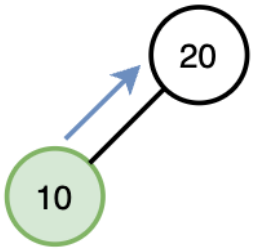
Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.

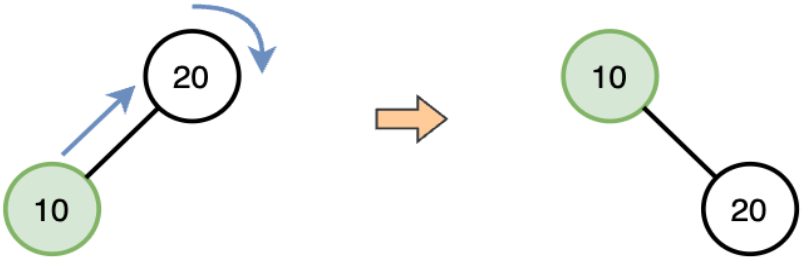
Step 4 - After insertion, Splay the newNode

Example: Insert 20,10,60,40,50 and 30 into splay tree

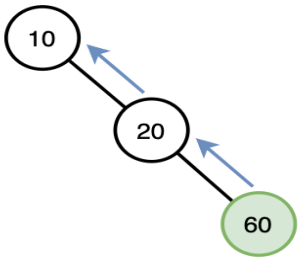
Insert element 20



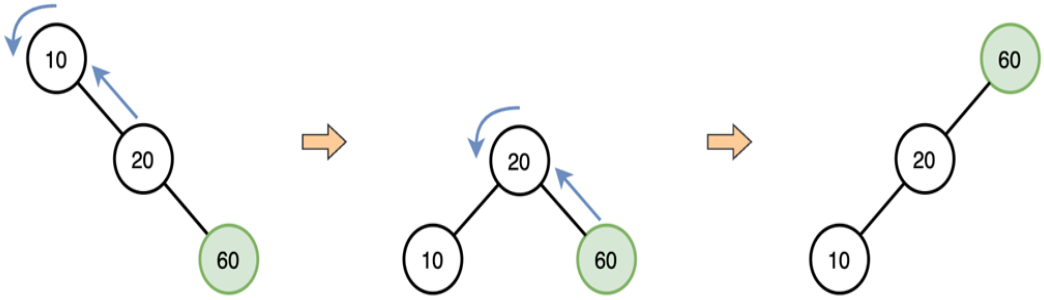
Insert element 10



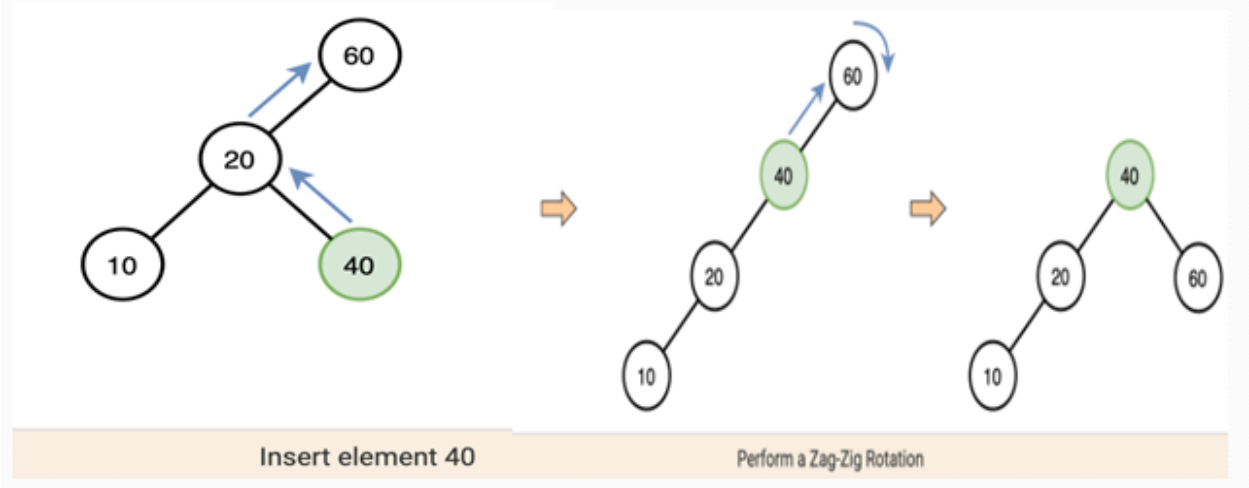
Perform a Zig Rotation from the node (20)

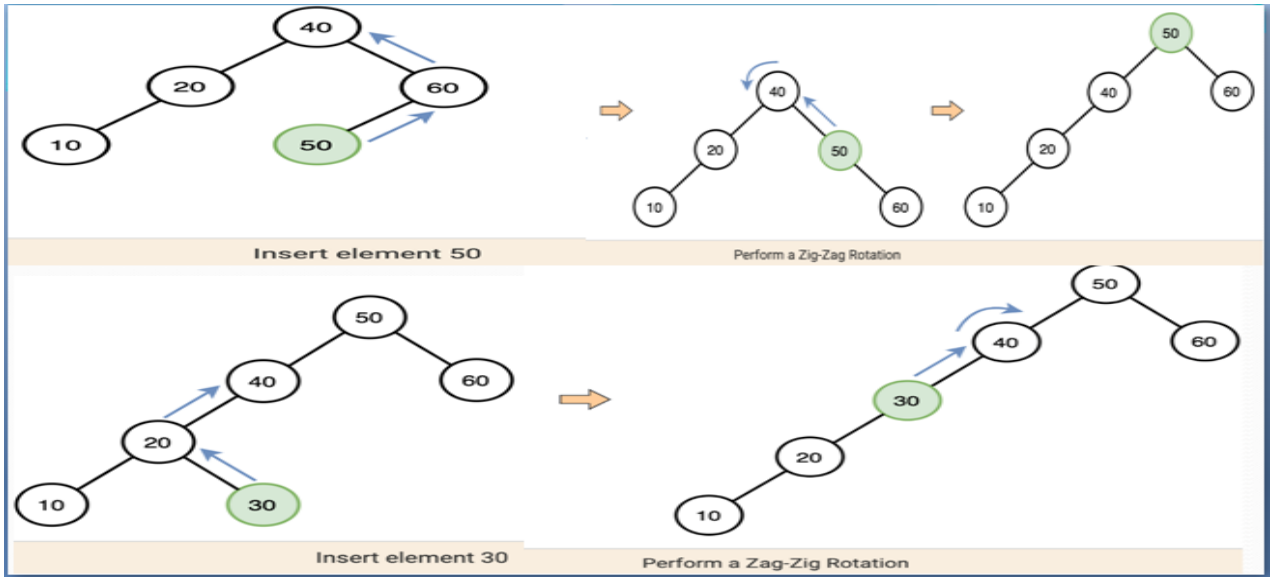


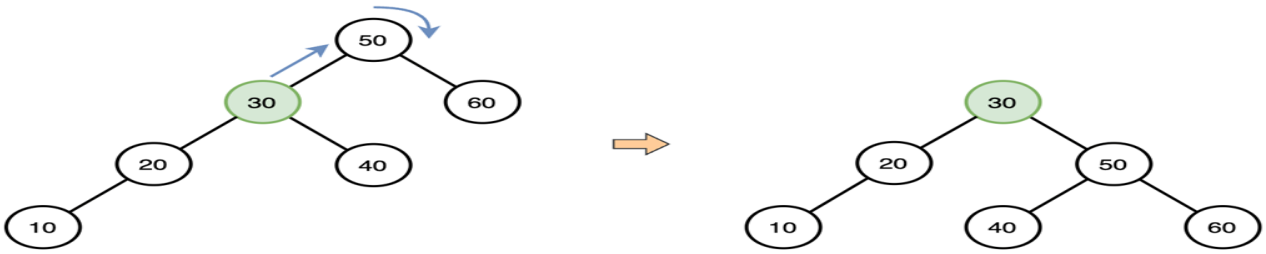
Insert element 60



Perform a Zag-Zag Rotation







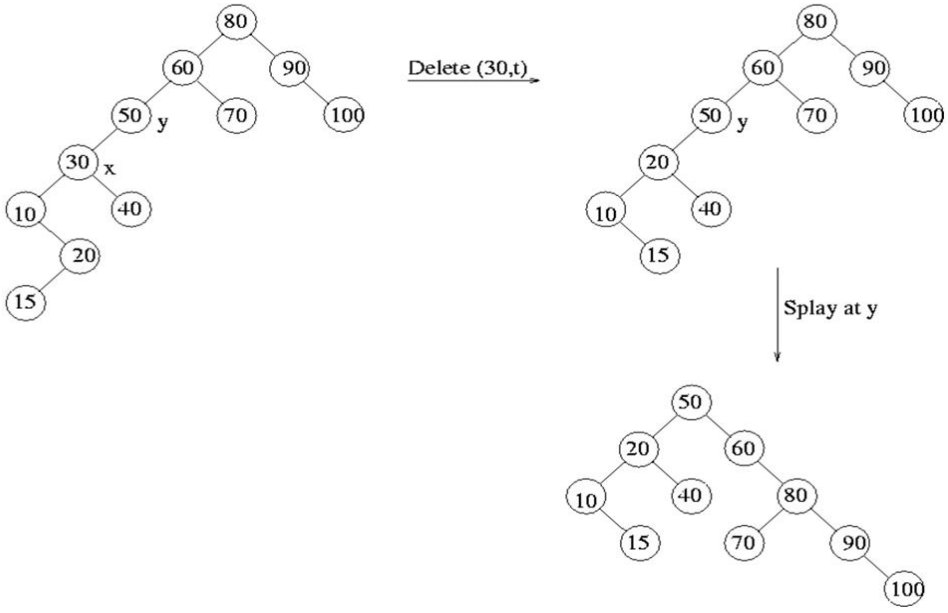
Deletion in Splay Trees

Two ways

a) First Delete using BST, next splay parent of node to be deleted (Bottom-up approach)

b) First Splay node to be deleted, next delete root (split tree), join trees using inorder predecessor or successor (Top-down approach)

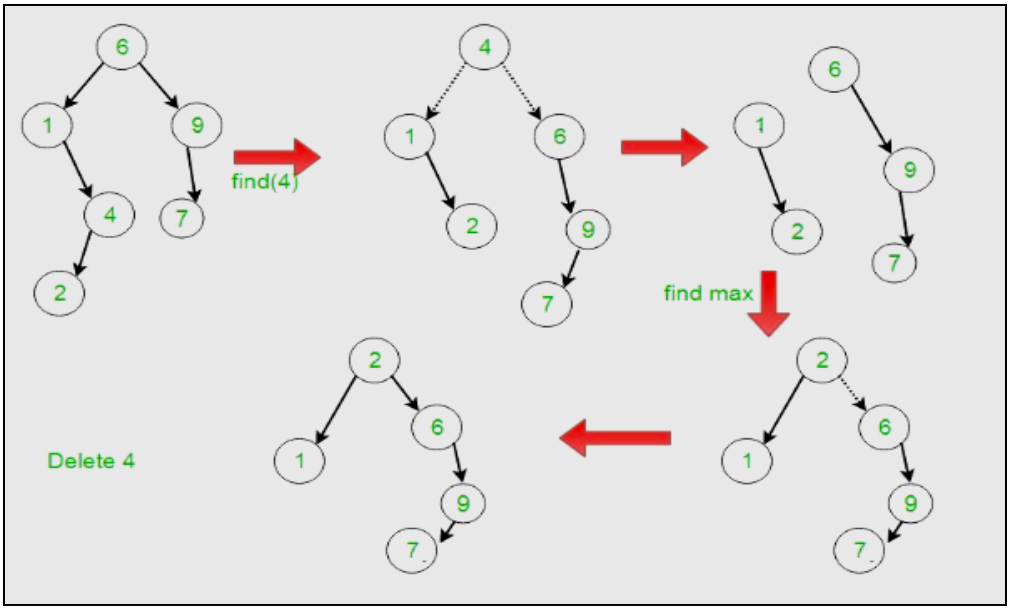
Top down approach is efficient



2.Top down approach

First Splay node to be deleted, next delete root (split tree), join trees using inorder predecessor or successor (Top-down approach)

Top down approach is efficient



B - Tree Data structure

In search trees like binary search tree, AVL Tree, Red-Black tree etc., every node contains only one value (key) and maximum of two children. But there is a special type of search tree called B□Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by Bayer and McCreight with the name **Height Balanced m□way Search Tree.**

Later it was named as B-Tree. B-Tree can be defined as follows.

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

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

Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that

is stored on a disk is a very time consuming process. Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

B-Tree of Order m has the following properties.

Property #1 - All leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

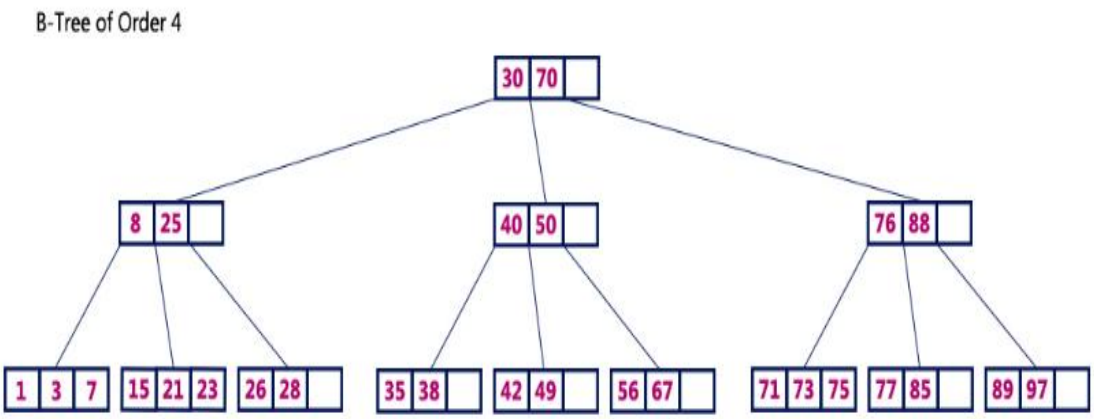
Property #4 - If the root node is a non leaf node, then it must have atleast 2 children.

Property #5 - A non leaf node with n-1 keys must have n number of children.

Property #6 - All the key values in a node must be in Ascending Order.

For example, B-Tree of Order 4 contains maximum of 3 key values in a node and maximum of 4 children for a node.

Example



Order m= 4

- **Min keys = [m/2]-1 = [4/2]-1=1**
- **Max keys = m-1 = 4-1 =3**

Order m=3

- **Min keys = [m/2]-1 = [3/2]-1=1**

Max keys = m-1 = 3-1 =2

Operations on a B-Tree The following operations are performed on a B-Tree...

1. Search 2. Insertion

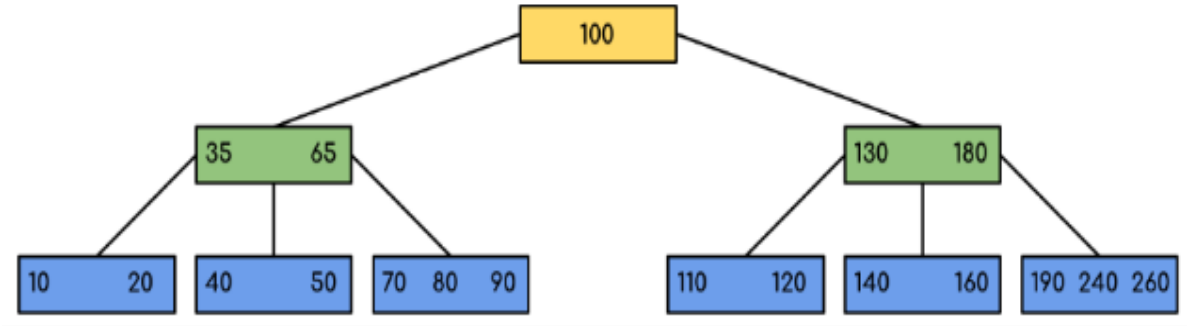
Search Operation

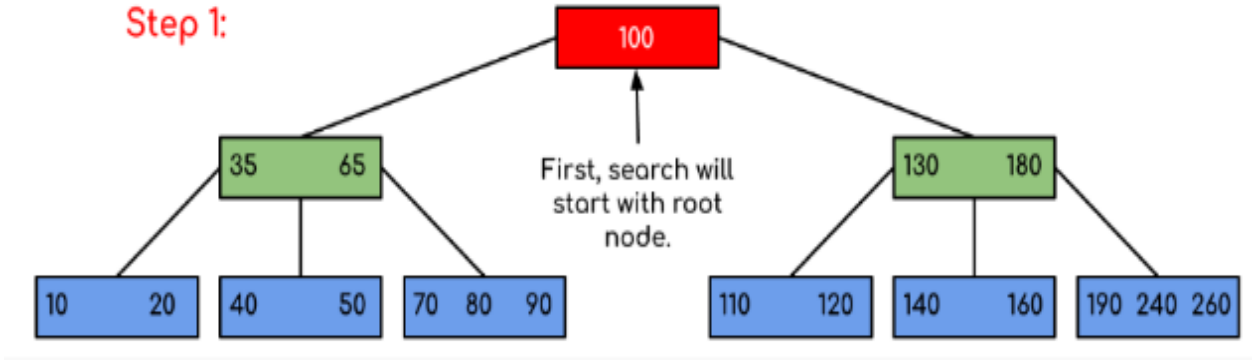
in B-Tree The search operation in B-Tree is similar to search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the

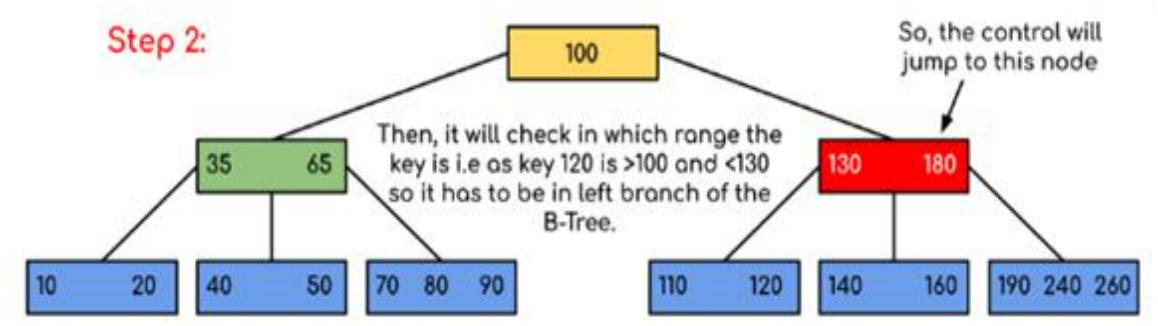
search operation is performed with O(log n) time complexity. The search operation is performed as follows..

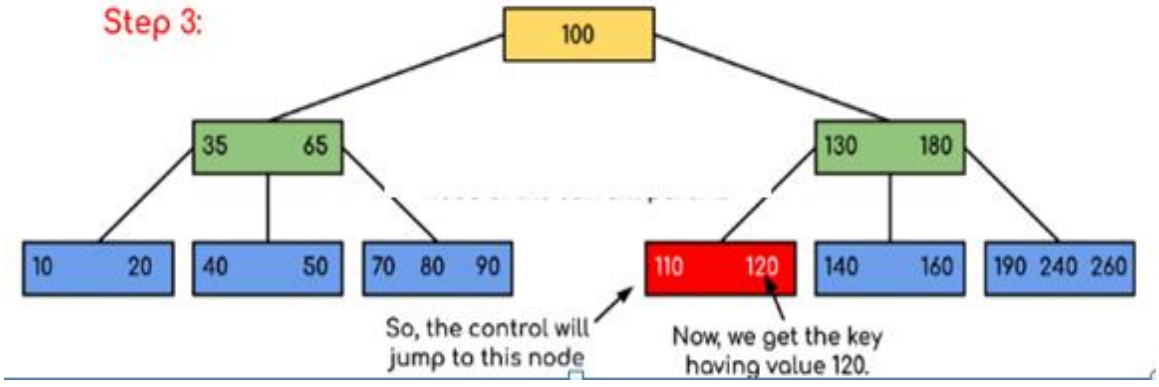
- **Step 1 - Read the search element from the user.**
- **Step 2 - Compare the search element with first key value of root node in the tree.**
- **Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function**
- **Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.**
- **Step 5 - If search element is smaller, then continue the search process in left subtree.**
- **Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.**
- **Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.**

Search 120









Insertion Operation

in B-Tree In a B-Tree, new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

• Step 1 - Check whether tree is Empty.

• Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.

• Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic. • Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

• Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

• Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

insert(1)

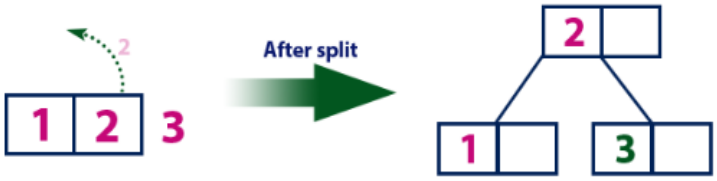
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

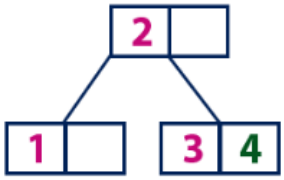
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



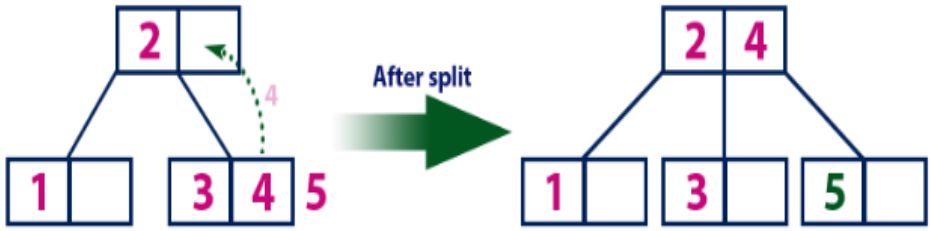
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



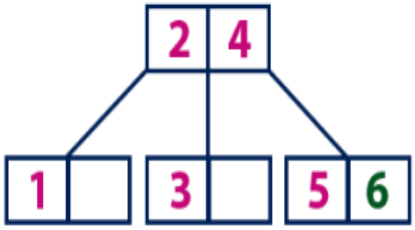
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



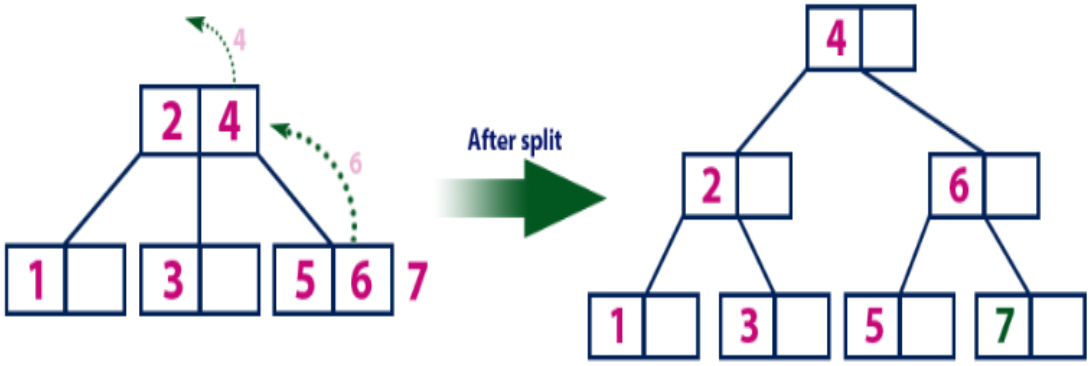
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



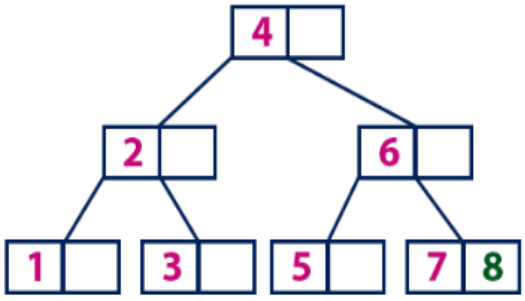
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



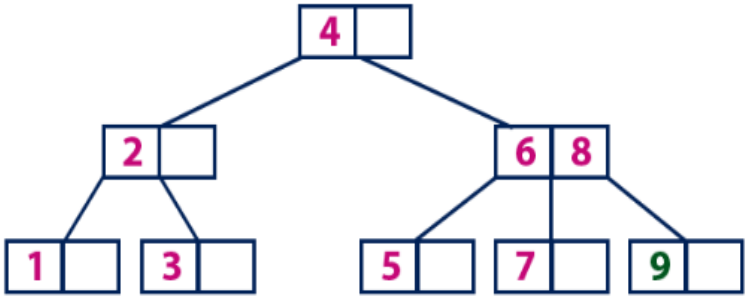
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



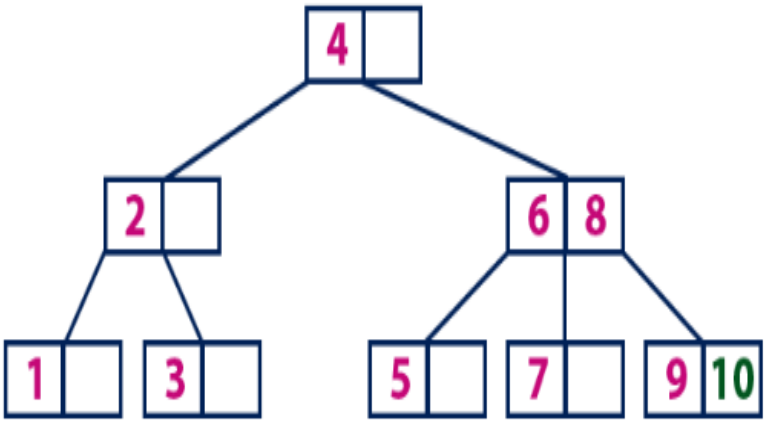
insert(9)

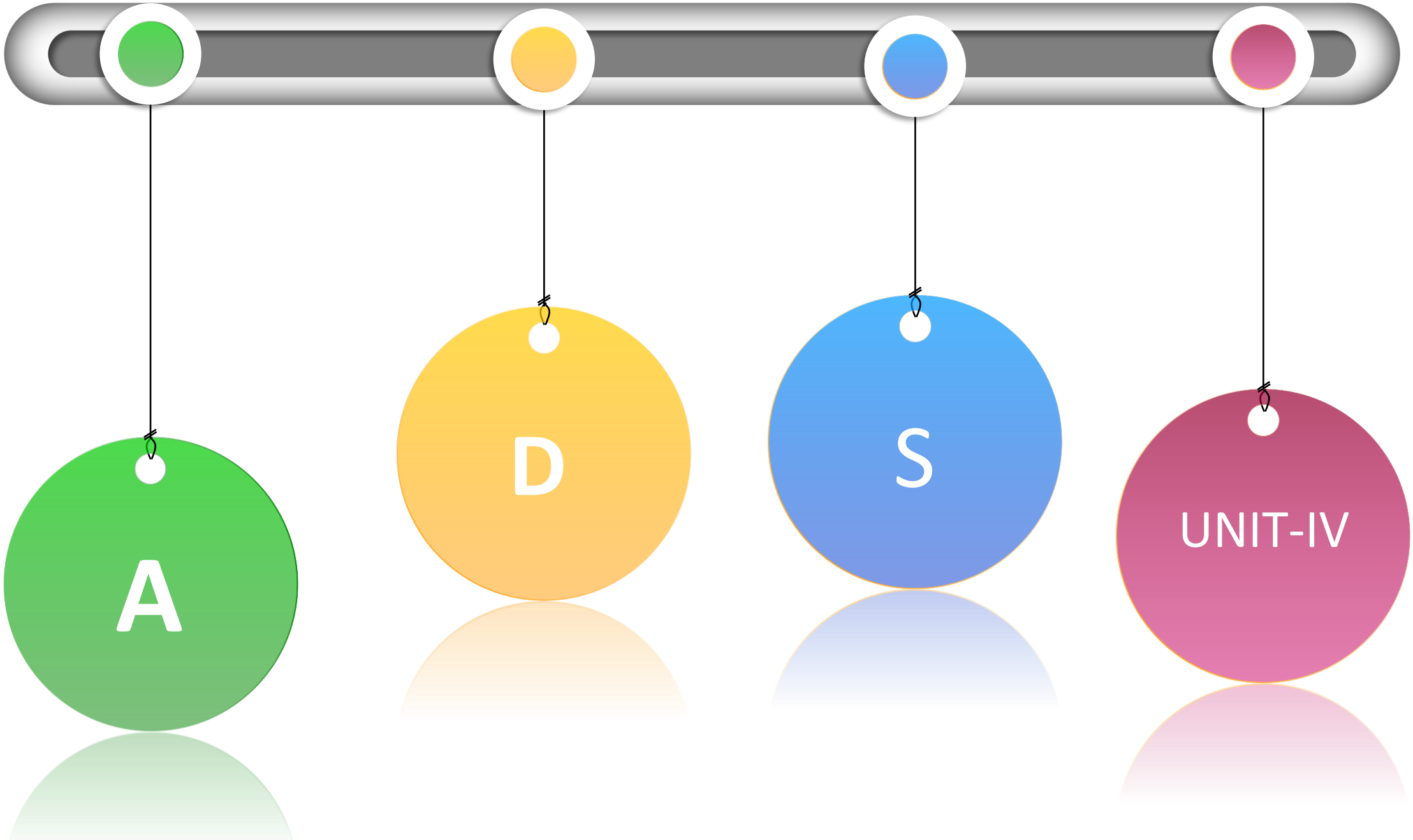
Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.




A
D
S
UNIT-IV

Hashing

Definition:

- Hashing is a technique to **convert a variable length input** into **fixed size output .**
- Hashing is the process of **mapping large amount of data item** to **smaller table** with the help of hashing function.
- The main idea behind the hashing is to **create the (key/hash key) pairs**. If the **key is given**, then the **algorithm** computes the **index at** which the **key value would be stored**.
- Hashing is a well-known technique **to search any particular element** among several elements.
- It **minimizes the number of comparisons** while performing the search.
- Hashing is extremely efficient. The **time** taken by it to perform the search **does not depend** upon the **total number of elements.**
- It completes the search with **constant time complexity O(1).**

Hashing

Basic Terms:

i.Hash table

ii.Key

iii.Hash Function

iv.Hash key

i.Hash table:

-A **Hash table is an array** ,it consists of n- slots, which are **used to store the data items.**

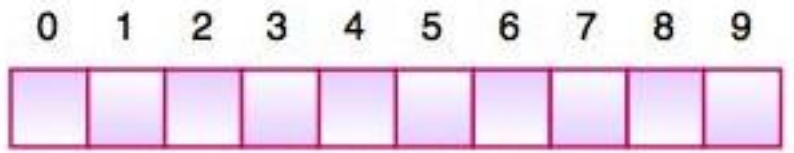


Fig. Hash Table

ii.Key:

-The **items** which we **want to insert** into **hash table** known as keys.

iii.Hash Function:

- Hash Function **is used converts a key to a hash key .**
- Hash function **takes** the key(**data item)** as an **input** and **returns** a small **integer value** as an output **known as Hash key**.
- Using the hash function, we can calculate the address at which the value can be stored.
- There are three ways of calculating the hash function:
    i. **Division method**
    ii. **Folding method**
    iii. **Mid square method**
- In the division method, the hash function can be defined as:

$$h(k) = k\% m;$$

- where m is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$$h(6) = 6\%10 = 6$$

The index is 6 at which the value is stored.

Hash function Methods

i.Division Method:

- Mapping a key K into one of m slots by taking the remainder of K divided by m.

  h (K) = K mod m

- Example: Assume a table has 8 slots (m=8). Using division method, insert the following elements into the hash table. 36, 18, 72, 43, and 6 are inserted in the order.

ii.Mid-Square Method:

- Mapping a key K into one of m slots, by getting the some middle digits from value K^2.

  h(k) = K^2 and get middle ($\log_{10}$ m) digits

Example: Assume the table size of 1000

- 3121 is a key and square of 3121 is 9740641.
- **$\log_{10}$(1000)**=3
- Middle part is 406

Hash function Methods

iii.Folding Method:

- Divide the key K into some sections, besides the last section, have same length. Then, add these sections together.

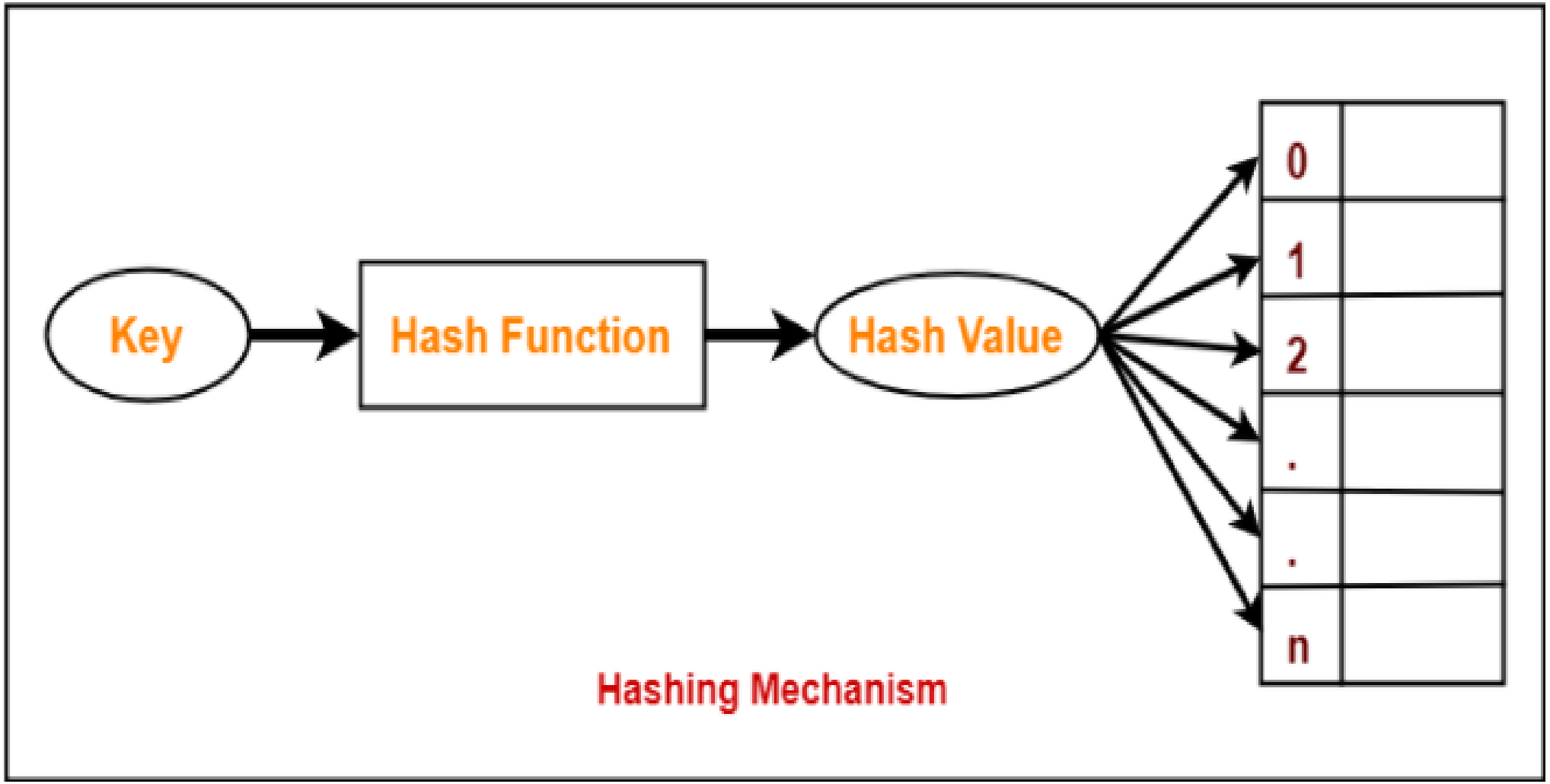
- The task is to fold the key **452378912** into a Hash Table of ten spaces (0 through 9).
- It is given that the key, say **X** is **452378912** and the table size (i.e., **M** = 10).
- Since it can break **X** into three parts in any order. Let's divide it evenly.
- Therefore, a = 452, b = 378, c = 912.
- Now, **H(x) = (a + b + c) mod M** i.e., H(452378912) =(452 + 378 + 912) mod 10 = 1742 mod 10 = **2**.
- Hence, 452378912 is inserted into the table at address **2**.

iv.Hash key:

- **Based on the hash key value**, **data items are inserted** into the hash table.
- **Hash key value** is a special value that **serves as an index** for a **data item**.
- It indicates **where the data item should** be **stored** in **the hash table**.
- **Hash key** value is **generated** using a **hash function.**
- **Hash key value of the data item** is then **used** as an index for **storing** it into the **hash table.**

Hash Key = Key Value % Number of Slots in the Table

H(x)=x % Hash table size



Hashing Mechanism

Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

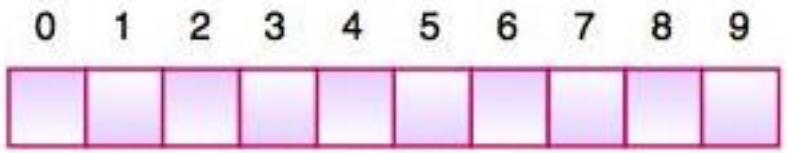


Fig. Hash Table

Hash Key = Key Value % Number of Slots in the Table

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

Fig. Hash Table

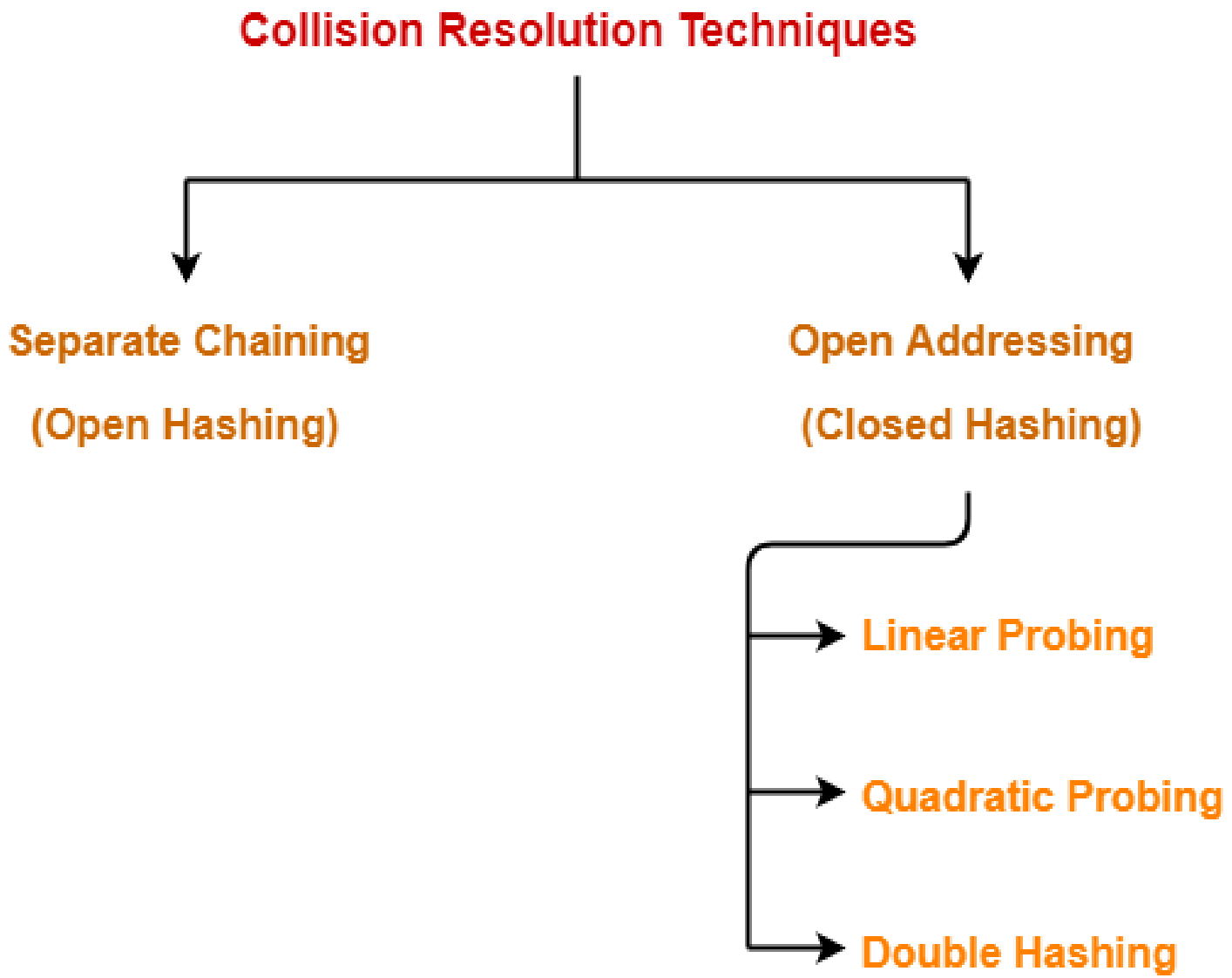
Collision

- There are various types of hash functions are:
- It **depends on the user** which hash function he wants to use.
- Hash function **may return the *same hash value* for two or more keys.**
- if a **hash function produces** the **same hash value** for **multiple elements** ,it is called as a **Collision**.
- **Collision:** No matter what the hash function, there is the possibility that two different keys could resolve to the same hash address. This situation is known as a collision.
- Ex 28,138 hash table size 10
- **h(28)=28%10=8**
- **h(138)=138%10=8**

Hashing Applications

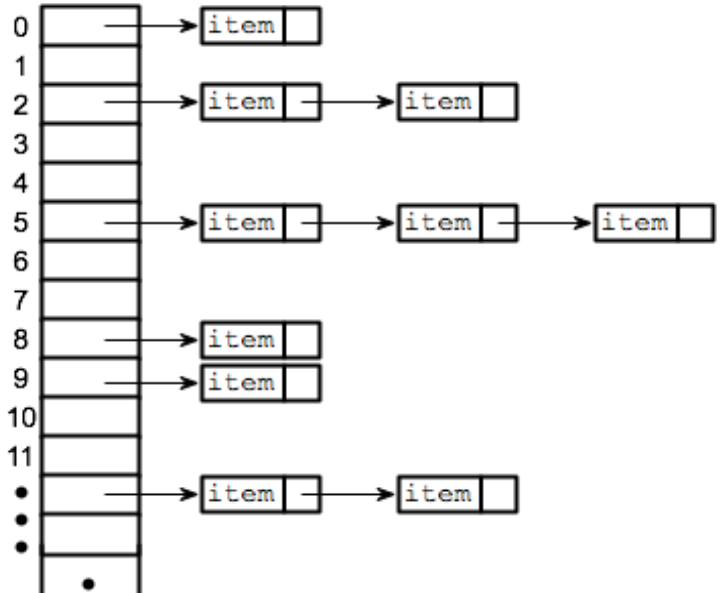
- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Collision Resolution Techniques



1.Separate Chaining

- if a **hash function produces** the **same index** for **multiple elements**, these **elements** are **stored** in the **same index** by using a **linked list**.
- To handle the collision, it **creates a linked list to the slot for which collision occurs.**
- The **new key** is then **inserted in the linked list**.
- **A chain is simply a linked list** of all the elements with the **same hash key.**
- These linked lists to the **slots appear like chains.**
- That is why, this technique is called as **separate chaining**.
- Here all the **keys are stored** in **out side of the hash table.**



1.Separate Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
- Let us consider sequence of keys as {8,21,13,23,14,7} and hash table size 7

Step-1: Insert 8

h(8)=8%7

=1

Step-2: Insert 21

h(21)=21%7

=0

Step-3: Insert 13

h(13)=13%7

=6

Step-4: Insert 23

h(23)=23%7

=2

Step-5: Insert 14

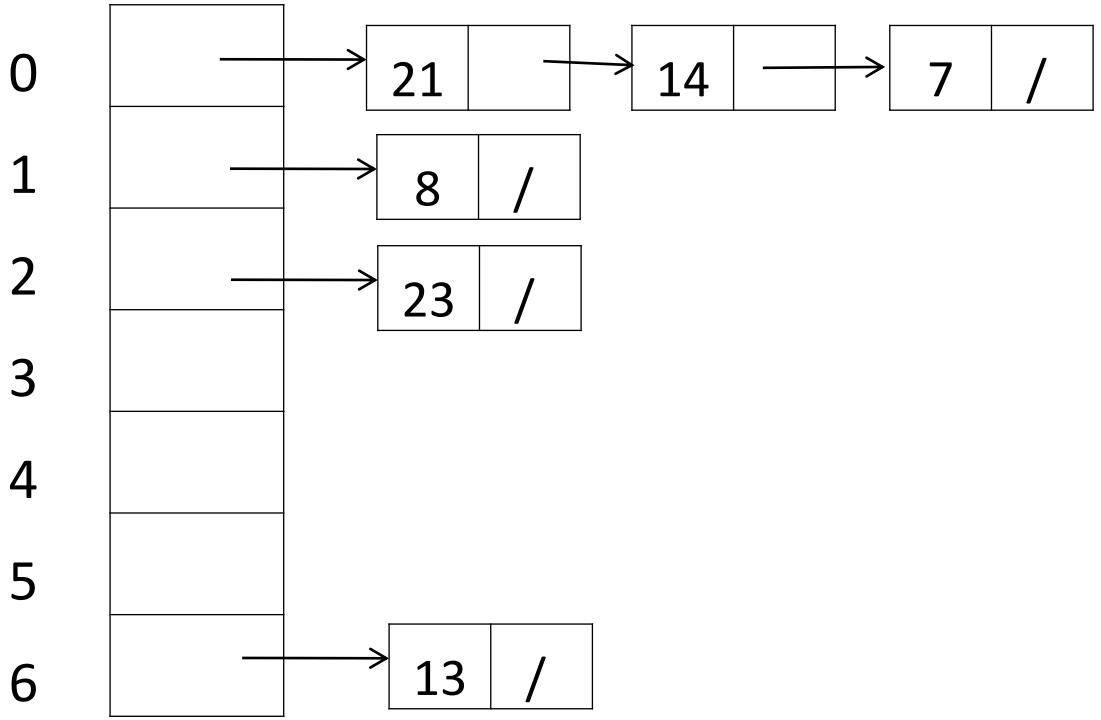
h(14)=14%7

=0

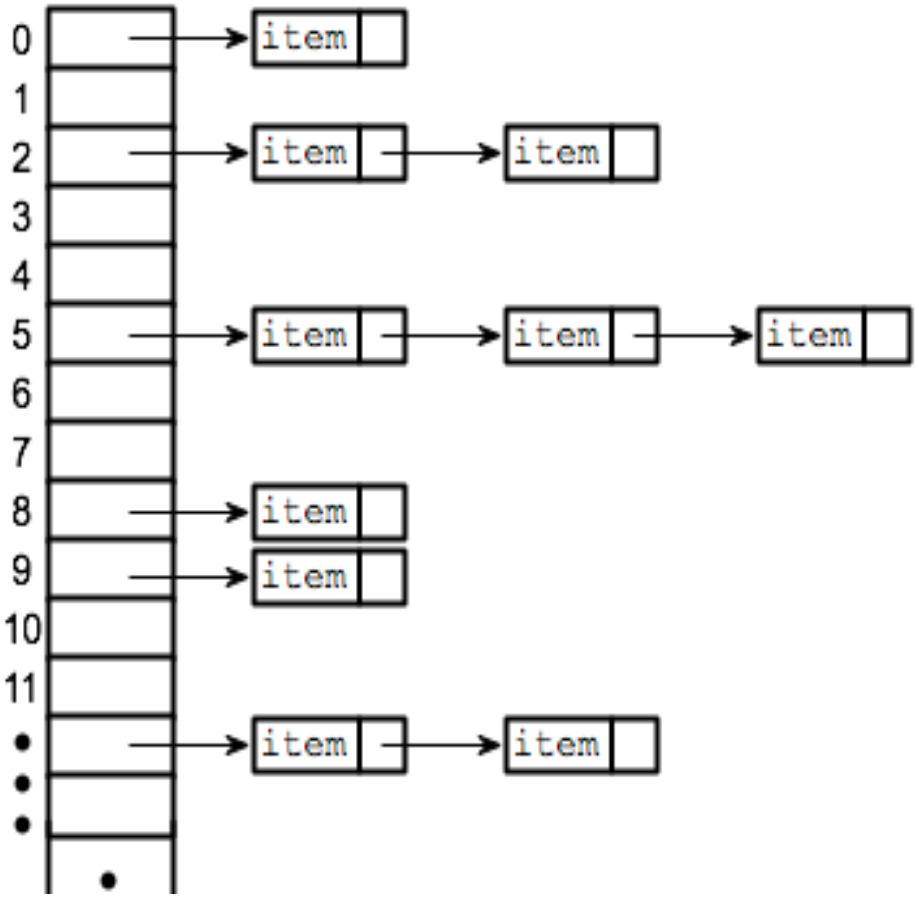
Step-6: Insert 7

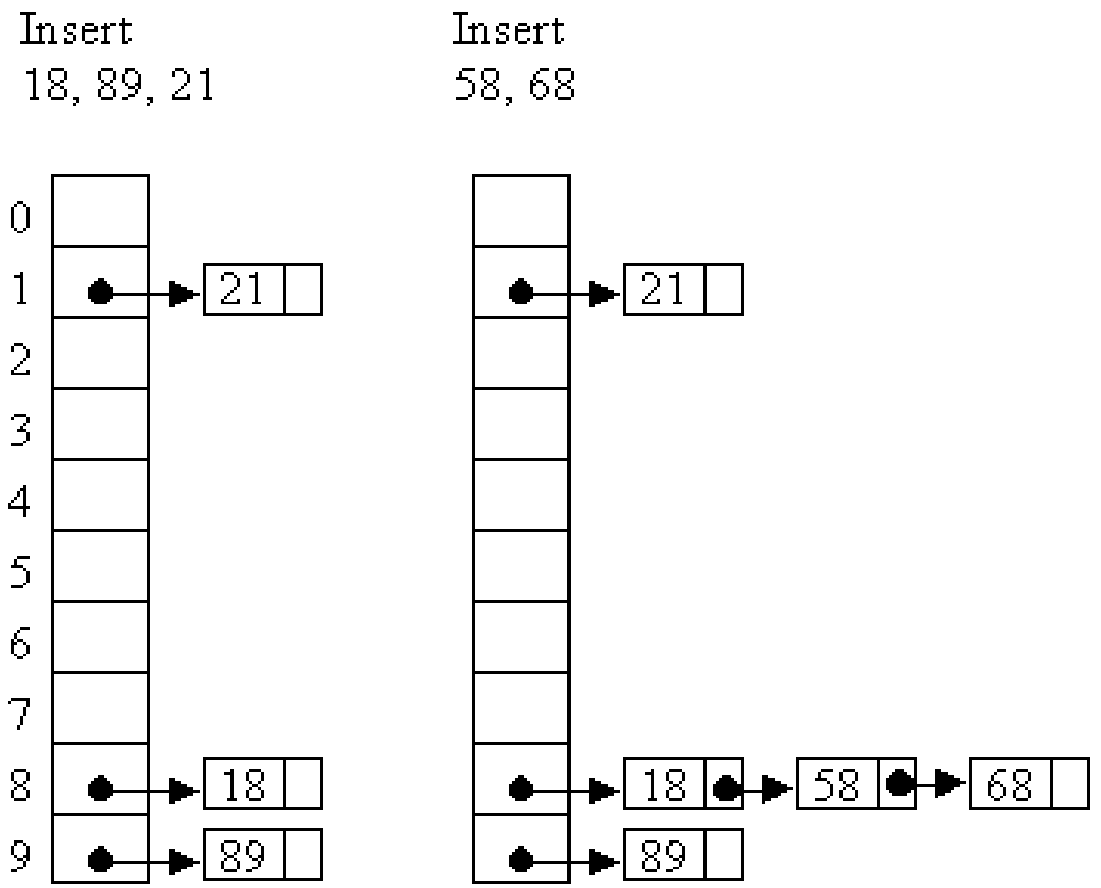
h(7)=7%7

=0



1.Separate Chaining





1.Separate Chaining

Advantages:

- Simple to implement.
- Hash table **never fills up**, we can always add more elements to the chain.
- It is mostly used when it is **unknown how many and how frequently keys** may be inserted or deleted.

Disadvantages:

- **Wastage of Space** (Some Parts of hash table are never used)
- If the **chain becomes long**, then search **time can become O(n)** in the worst case.
- **Uses extra space for links.**

2.Open Addressing

- In open addressing, when a collision occur, **we will search for the empty slot ,** Once an **empty slot is found**, **the key is inserted**.
- We'll use three methods of open addressing, which vary in the method used to find the next empty location.
- These methods are:

  i.Linear Probing **ii.Quadratic Probing** **iii.Double Hashing**

- In open addressing, **all the keys are stored inside the hash table.**
- **No key is stored outside the hash table.**

i)Linear Probing

- When using a linear probing method **the item will be stored in the next available slot** in the table, assuming that the table is not already full.
- This is implemented via **a linear searching** for an empty slot, **from the point of collision.**
- If the **end of table is reached** during the linear search, the **search will from the beginning of the table** and continue from there.

$$h(x)=x \% size$$

$$h^1(x)= [\ h(x) + f(i)\ ] \% size$$

Here $f(i)=i$

$i=0,1,2,3....$

i)Linear Probing

Example:

Assume a table has 8 slots (m=8). Using Linear probing, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.

i=0

Step 1: (36+0) %8=4

Step 2: (18+0) %8=2

Step 3: (72+0) %8=0

Step 4:(43+0)%8=3

Step 5: (6+0)%8=6

Step 6: (10+0) %8=2

Collision

index=3),i=2(index=4),

i=3(index=5

Step 7: 5%8=5 collision 6 so 7

0	72
1	15
2	18
3	43
4	36
5	10
6	6
7	5

index= (key+ i)% tablesize

i=0,1,2,3....

0	
1	
2	
3	
4	
5	
6	
7	

i)Linear Probing

For example,

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

✓If we insert next item 40 in our hash table, it would have a hash value of 0.

h(x)= 40%10

h(x)=0

✓Position 0 is occupied by 70.

✓It becomes a problem, By using Linear probing we will solves this problem

✓ so we look other empty slot to store 40,Using Linear Probing:

Step-1:

$h^1(x) = [\, h(x) + f(i) \,] \,\%\, size$ (since $f(i)=i$ and $i=1$)

=0 + 1% 10

= 1

But, position 1 is occupied by 31

so we look other position to store 40

0	1	2	3	4	5	6	7	8	9
70	31	40	93	54		26		18	

Fig. Hash Table

Step-2:

$h^1(x) = [\, h(x) + f(i) \,] \,\%\, size$ (since $f(i)=i$ and $i=2$)

=0 + 2 % 10

= 2

Position 2 is empty, so 40 is inserted there.

ii). Quadratic Probing

- If the collision occurs, instead of **moving one cell, move i^2 cells from the point of collision**, where i is the number of attempts to resolve the collision.
- It works similar to linear probing but the spacing between the slots is increased

Index = (key + i*i) % tab_size

(i= 0 to tab_size-1)

ii). Quadratic Probing

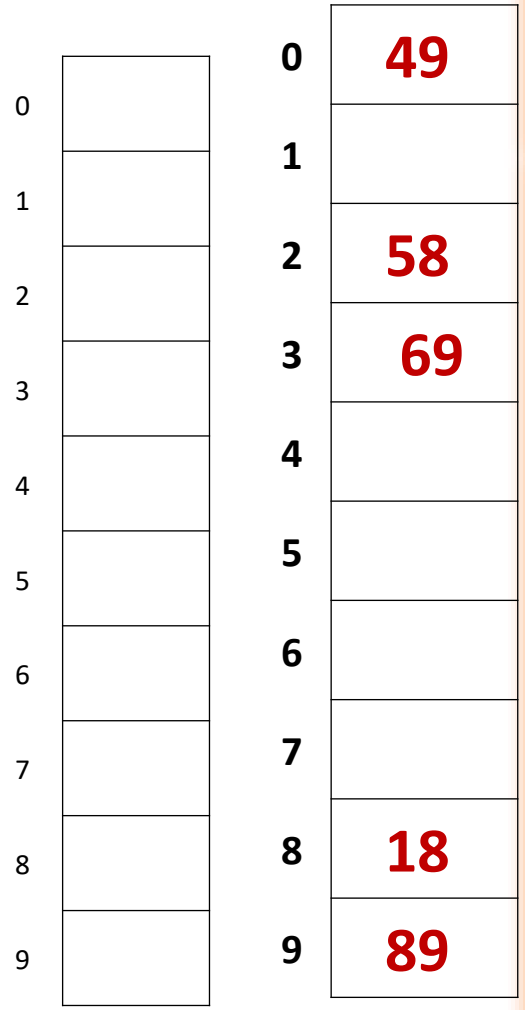
- Example: Assume a table **has 10 slots.**
- **Using Quadratic probing**, insert the following elements in the given order. **89, 18, 49, 58 and 69** are inserted into a hash table.

Step 1: h(89)=89% 10
=9

Step 2: h(18)= 18% 10
=8

Step 3: h(49)= 49% 10
=9 is occupied
so we need to use $h^1(x)$
$h^1(x) = [\, h(x) + f(i) \,] \,\%\, size$
Here $f(i) = i^2$ and $i=1,2,3....$
$h^1(9) = (9 + 1^2) \,\%\, 10$
=0

So, insert 49 into 0 th location.



Step 4: h(58)= 58% 10
=8 is occupied
so we need to use $h^1(x)$
$h^1(x) = [\, h(x) + f(i) \,] \,\%\, size$
Here $f(i) = i^2$ and $i=1,2,3....$
$h^1(58) = (8 + 1^2) \,\%\, 10$
=9 is occupied
$h^1(58) = (8 + 2^2) \,\%\, 10$
=2

So, insert 58 into 2nd location.

Step 5: h(69)= 69% 10
=9 is occupied
so we need to use $h^1(x)$
$h^1(x) = [\, h(x) + f(i) \,] \,\%\, size$
Here $f(i) = i^2$ and $i=1,2,3....$
$h^1(69) = (9 + 1^2) \,\%\, 10$
= 0 is occupied
$h^1(69) = (9 + 2^2) \,\%\, 10$
= 3

So, insert 69 into 3rd location.

Linear Vs Quadratic Probing

- Linear probing
  - Clustering – it causes long probe searches
  - Always find an open spot if one exists
  - It might be a long search but we will find it
- Quadratic Probing
  - Clustering is eliminated
  - In order to guarantee quadratic probes hit, table size must meet these requirements:
    - Be a prime number
    - Never be more than half full

iii. Double Hashing

- To **eliminate secondary clustering**, Double hashing **achieves this by having two hash functions**.
- Double hashing uses the idea of **applying a second hash function** to the key **when a collision occurs.**

$$h(key,i) = [\ h_1(key) + i * h_2(key)\ ] \% tableSize$$
$$for\ i=0,1,\ldots,\ tableSize-1$$

$$h_1(key) = key \% tableSize$$

Common definitions for h2 are :

i. $h_2(key) = 1 + key \% (tableSize - 1)$

ii. $h_2(key) = q - (key \% q)$ where q is a prime less than tableSize

iii. $h_2(key) = q*(key \% q)$ where q is a prime less than tableSize

iii. Double Hashing

- Example: Assume a table has **13 slots.**
- With Double hashing, insert the following elements in the given order.
- **18, 26, 35, 9, 64, 47, 96, 36, and 70** into a hash table

0	1	2	3	4	5	6	7	8	9	10	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
26					18	9			35			

Step 1: h(18,0) = (18%13+0)%13 = 5 (since i=0)

Step 2: h(26,0)=(26% 13+0)% 13 =0

Step 3: h(35,0)= (35% 13+0)%13 = 9

Step 4: h(9,0)= (9% 13+0)%13 = 9 collision

We have to use second hash function

h_2 **(key) = 1 +(key % 12)**

h_2 **(9) = 1+(9%12)**

= 10----------ii

Sub equation ii into main hash function

$h(key,i) = [h_1(key) + i*h_2(key)]\%\ tableSize$

h(9,1) = (9 + 1*10)%13 (since i=1)

=6

So insert 9 into 6th index

With Double hashing, insert the following elements in the given order.18, 26, 35, 9, 64, 47, 96, 36, and 70 into a hash table

0	1	2	3	4	5	6	7	8	9	10	11	12
26					18	9			35			

Step 5: h(64,0) = (64%13+0)%13 = 12

Step 6: h(47,0) = (47%13+0)%13 = 8

Step 7: h(96,0) = (96%13+0)%13 = 5 collision

We have to use second hash function

$h_2(96) = 1 + 96\%12$

=1----------ii

Sub equation ii into main hash function

$h(key,i) = [h_1(key) + i*h_2(key)]\%\ tableSize$

h(96,1) = (5+1*1)%13=6 collision (since i=1)

h(96,2) = (5+2*1)%13=7 (since i=2)

So insert 96 into 7th index

Step 8: h(36,0) = (36% 13+0)% 13 = 10

Step 9: h(70,0) = (70%13)%13 = 5 collision

We have to use second hash function

$h_2(70) = 1 + (70\%12)$

=11----------ii

Sub equation ii into main hash function

$h(key,i) = [h_1(key) + i*h_2(key)]\%\ tableSize$

h(70) = (5+1*11)%13

=3

So insert 70 into 3rd index

iii. Double Hashing

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

0	26
1	
2	
3	70
4	
5	18
6	9
7	96
8	47
9	35
10	36
11	
12	64

Example:

Assume a table has 10 slots. Using Quadratic probing, insert the following elements in the given order.

- 89, 18, 49, 58 and 69 are inserted into a hash table.

iii. Double Hashing

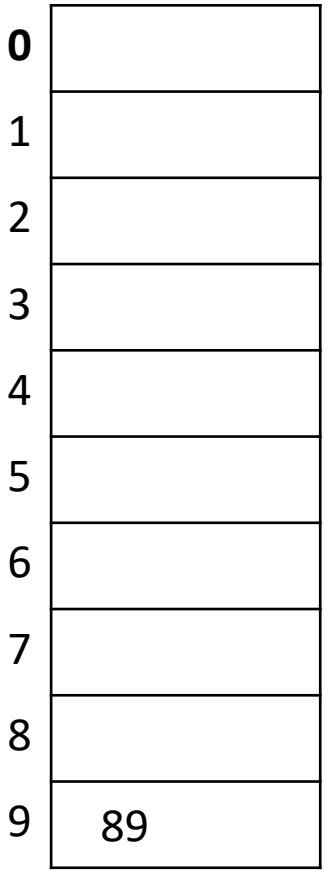
Advantages of Double hashing:

- Resolution sequences for different elements are different even if the first hash function hashes the elements to the same field.
- If the hash functions are chosen appropriately, **insertion never fails if the table has at least one free field.**

Double Hashing

Example 2 :Apply double hashing on sequence of keys 89, 18,49,58 ,69.where R=7 and M=10

Insert 89



Hash function here is key+i *(hash2(key))%TABLE_SIZE

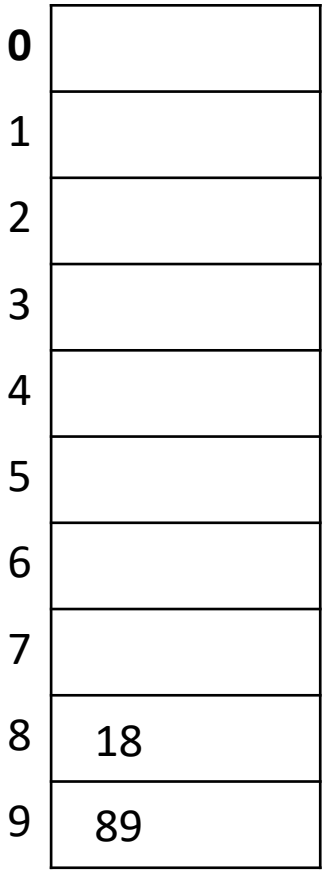
Hash2(key) = R-(key%R) where R is a prime number less than the TABLE_SIZE

Here the Hash function is

(key+i*(R-(key%R)%TABLE_SIZE

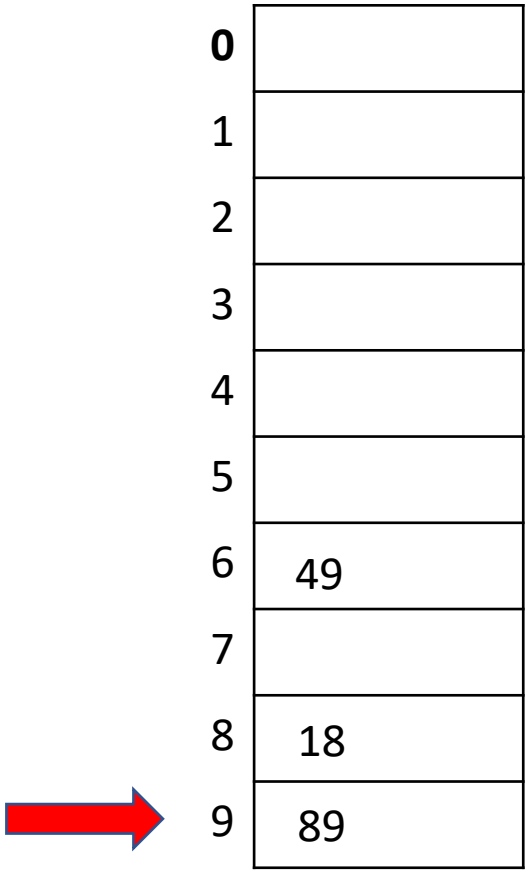
index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (89+0*(7-(89%7)) % 10 = 89%10 = 9
So we insert 89 at index 9

Insert 18



index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (18+0*(7-(18%7)) % 10 = 18%10 = 8
So we insert 18 at index 8

Insert 49

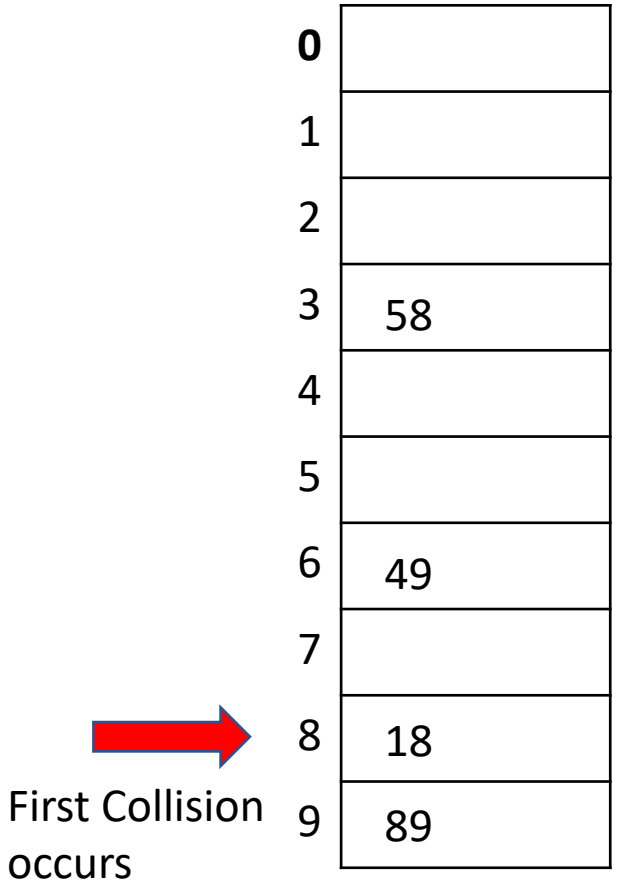


First Collision occurs

index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (49+0*(7-(49%7)) % 10 = 49%10 = 9
So insert 49 at index 9

Index 9 is not free so collision occurs and this is the first collision so i becomes 1.
index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (49+1*(7-(49%7))%10
=(49+1*(7-0))%10
= (49+1*7)%10
=(49+7)%10
=56%10
=6
So insert 49 at index 6

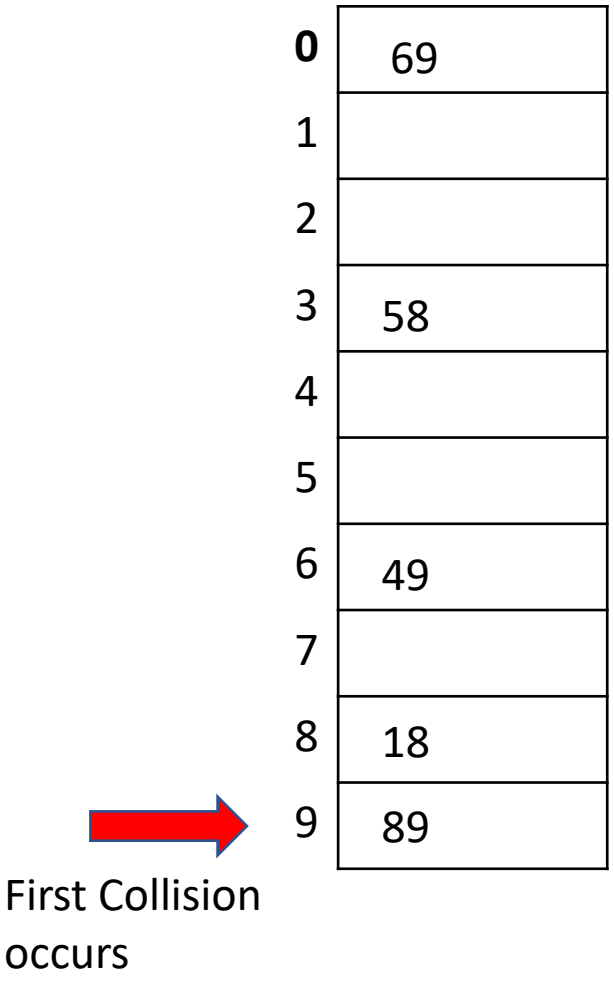
Insert 58



index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (58+0*(7-(58%7)) % 10 = 58%10 = 8
So insert 58 at index 8

index 8 is not free so collision occurs and this is the first collision so i becomes 1.
index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (58+1*(7-(58%7))%10
=(58+1*(7-2))%10
= (58+1*5)%10
=(58+5)%10
=63%10
=3
So insert 58 at index 3

Insert 69



index = (key+i*(R-(key%R))%TABLE_SIZE
Index = (69+0*(7-(69%7)) % 10 = 69%10 = 9
So insert 69 at index 9

index 9 is not free so collision occurs and this is the first collision so i becomes 1.
index = (key+i*(R-(key%R))%TABLE_SIZE
index = (69+1*(7-(69%7))%10
=(69+1*(7-6))%10
= (69+1*1)%10
=(69+1)%10
=70%10
=0
So insert 69 at index 0

Implementation of hash table and perform insert ,delete and search using Linear Probing

Algorithm to insert a value in linear probing

Hashtable is an array of size = TABLE_SIZE

Step 1: Read the value to be inserted

Step 2: let i = 0

Step 3: compute the index at which the value has to be inserted in hash table

index = ((key% TABLE_SIZE) + i) % TABLE_SIZE

Step 4: if there is no element at that index then insert the value at index and STOP

Step 5: If there is already an element at that index

step 4.1: i = i+1

step 6: if i < TABLE_SIZE then go to step 3

Algorithm to search a value in linear probing

Hashtable is an array of size = TABLE_SIZE

Step 1: Read the value to be searched

Step 2: let i = 0

Step 3: compute the index at which the value can be found

index = ((key% TABLE_SIZE) + i) % TABLE_SIZE

Step 4: if the element at that index is same as the search value then print element found and STOP

Step 5: else

step 4.1: i = i+1

step 6: if i < TABLE_SIZE then go to step 3

Hashing using Linear Probing Implementation

```
#include<stdio.h>
#include<stdlib.h>
int count=0;
void insert(int hash[],int m,int key)
{
int i=0,h;
if(count==m) printf("hash table full");
else{
do{
    h=(key+i)%m;
    if(hash[h]==-1){   hash[h]=key ;  count++; break;}
  else     i++; if(i==m-1)    i=0;  }while(i<=m);  }
}
```

```
void search(int hash[],int m,int key)
{
int i=0,h,flag=0,c=0;
   do{
     h=(key+i)%m;
     if(hash[h]==key){ flag=1;break; }
       else{ i++;c++;}
     if(i==m-1)
        i=0;
     }while(c<=m);
 if(flag==1)
  printf("key found at %d position",h);
  else
  printf("key not found ");   }
```

```
void delete(int hash[],int m,int key)
{    int i=0,h,flag=0,c=0;
if(count==0) printf("hash table is empty");
else{
do{     h=(key+i)%m;
if(hash[h]==key){flag=1;
  hash[h]=-1;count--;break;}
  else  {    i++;c++;}
 if(i==m-1)    i=0;  }while(c<=m);}
if(flag==0)
printf("key not found can't delete");  }
```

```
void print(int hash[],int m )
{
int I;  for(i=0;i<m;i++)   printf("hash[%d]- %d\n",i,hash[i]);}
void main()
{  int m,i,ch,key;
printf("enter size of hash table");   scanf("%d",&m); int hash[m];// create hash table with m
for(i=0;i<m;i++   )// intialize table with -1
hash[i]=-1;// empty hash table
while(1){
printf("enter your choice"); printf("1:insert 2:search 3:delete 4:print 5:exit\n");
scanf("%d",&ch);   switch(ch) {
case 1:printf("enter key");   scanf("%d",&key); nsert(hash,m,key);break;
 case 2: printf("enter key");       scanf("%d",&key); search(hash,m,key);break;
      case 3:printf("enter key");       scanf("%d",&key); delete(hash,m,key);break;
       case 4:print(hash,m);break; case 5: exit(0);  } }
}
```

```
#include<stdio.h>
#include<stdlib.h>
#define size 7
struct node
{
    int data;
    struct node *next;
};
struct node *chain[size];
void init()
{
    int i;
    for(i = 0; i < size; i++)
        chain[i] = NULL;
}
```

```
void insert(int value)
{
//create a newnode with value
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = value;
   newNode->next = NULL;
   //calculate hash key
   int key = value % size;
   //check if chain[key] is empty
   if(chain[key] == NULL)
      chain[key] = newNode;
   //collision
   else    {       //add the node at the end of chain[key].
      struct node *temp = chain[key];
      while(temp->next!=NULL)
      {           temp = temp->next;         }
      temp->next = newNode;    }   }
```

```
void print()
{  int i;
    for(i = 0; i < size; i++)
    {  struct node *temp = chain[i];
        printf("chain[%d]-->",i);
        while(temp)
        {   printf("%d -->",temp->data);
            temp = temp->next;  }
        printf("NULL\n");
     }
}
```

```
int main()
{    //init array of list to NULL
    init();
    insert(7);
    insert(0);
    insert(3);
    insert(10);
    insert(4);
    insert(5);
    print();
    return 0;
}
```

Separate chaining Vs Open addressing

Separate chaining	Open addressing
Hash table never fills up	Table may become full
Less sensitive to the hash function or load factors	Requires extra care for to avoid clustering and load factor
Used when it is unknown how many and how frequently keys may be inserted or deleted.	Used when the frequency and number of keys is known.
Wastage of Space (Some Parts of hash table are never used).	A slot can be used even if key doesn't map to it.
Requires extra space and computation for links .	No links hence provides better cache performance

Rehashing

- So, **on an average**, if there are **n entries** and **b is the size** of the array there would be n/b entries on each index. This value **n/b** is called the **load factor** that represents the load that is there on our map.
- when the **load factor increases** to more than its **pre-defined value** then **complexity increases**.
- all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.
- default value of load factor is **0.75**

Rehashing Example

Hash Table with Linear Probing after 13,15,6,24 are inserted

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

0	6
1	15
2	23
3	24
4	
5	
6	13

Load factor=0.6 < 1

After Rehashing

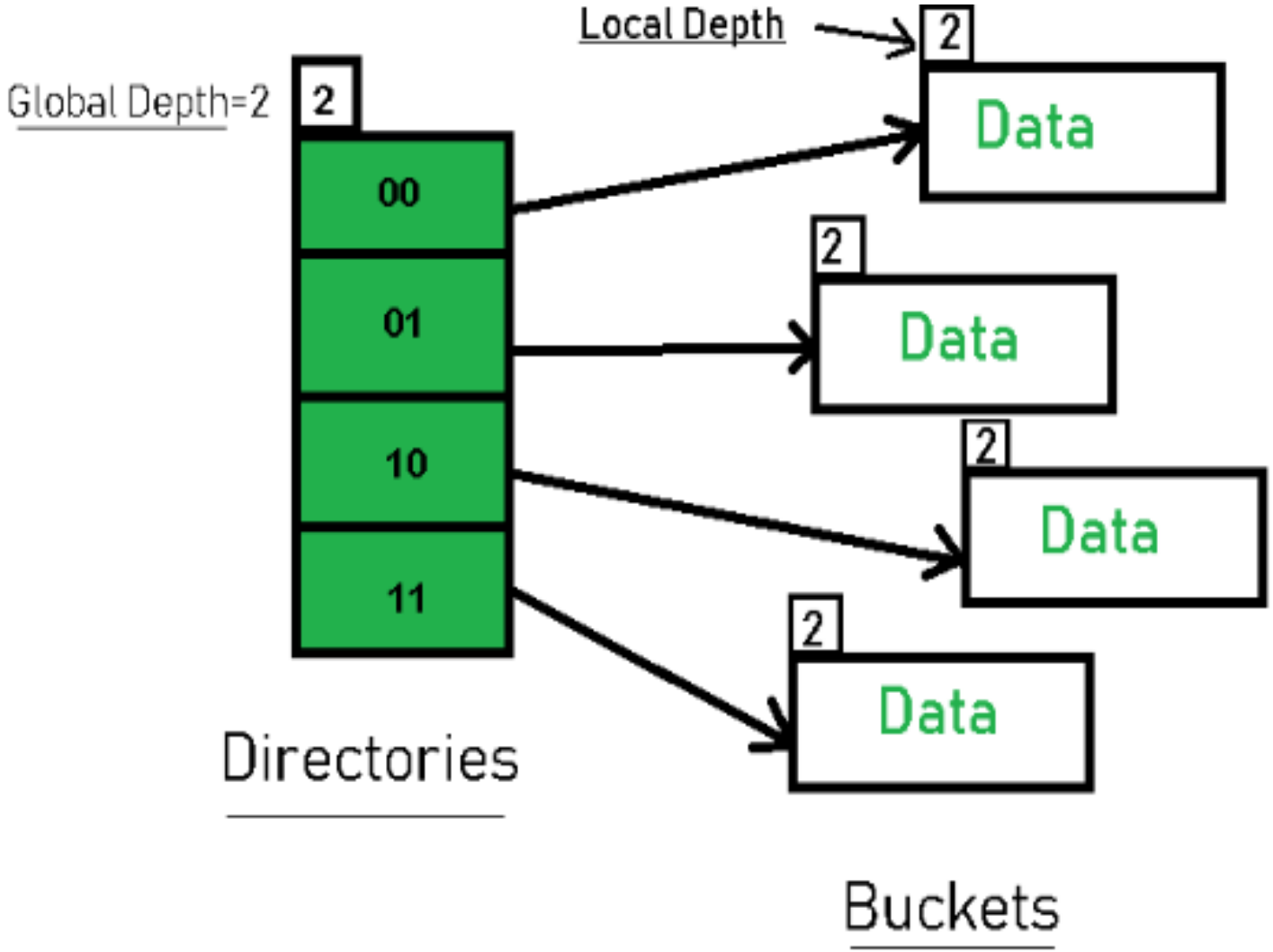
0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing procedure

- For each addition of a new entry to the hash table, check the load factor.
- If it’s greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size (prime)
- Then traverse each element in the old Hash table and call the insert() so as to insert it into the new larger Hash table (bucket array) with new hash function.

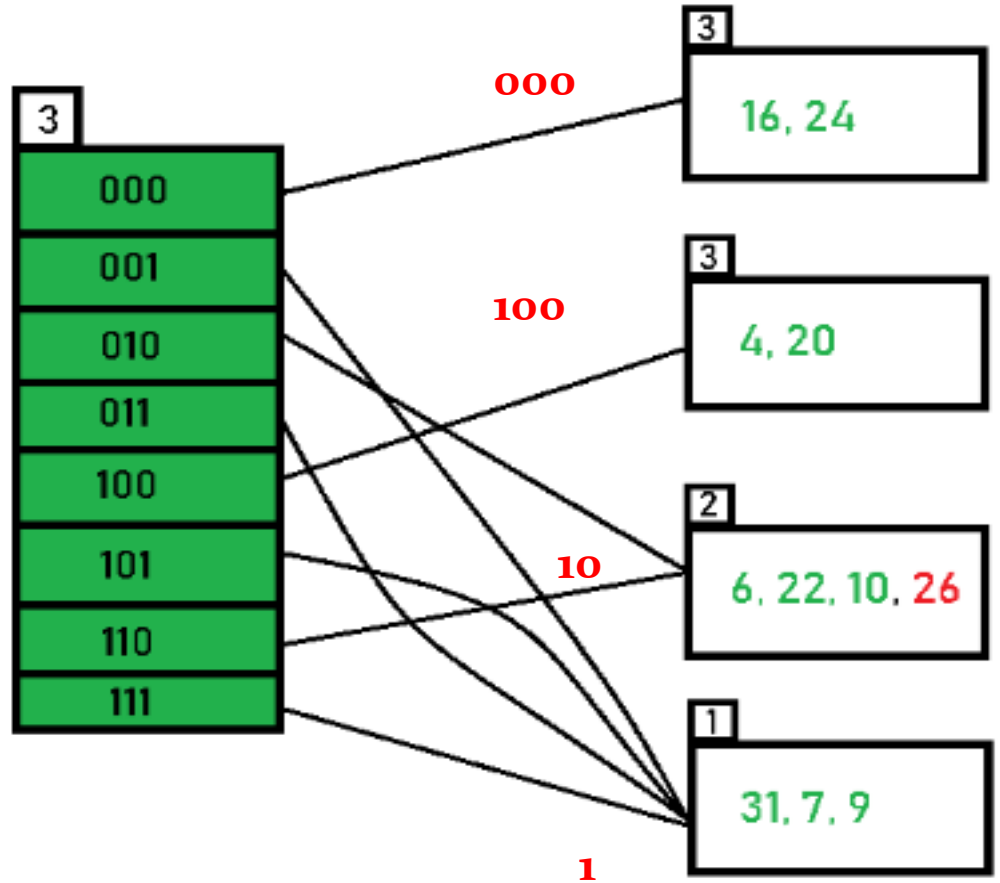
Extendible Hashing

- **Extendible Hashing** is a dynamic hashing method wherein **directories, and buckets.**
- It is a flexible method in which the **hash function** also experiences **dynamic changes**.
- **Directories:** store pointers to the buckets.
- An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The y are used to hash the actual data.
- **Global Depth** = # of bits in directory id.
- **Local Depth**= # of bits used to determine if an
- entry belongs to the bucket.



Extendible Hashing

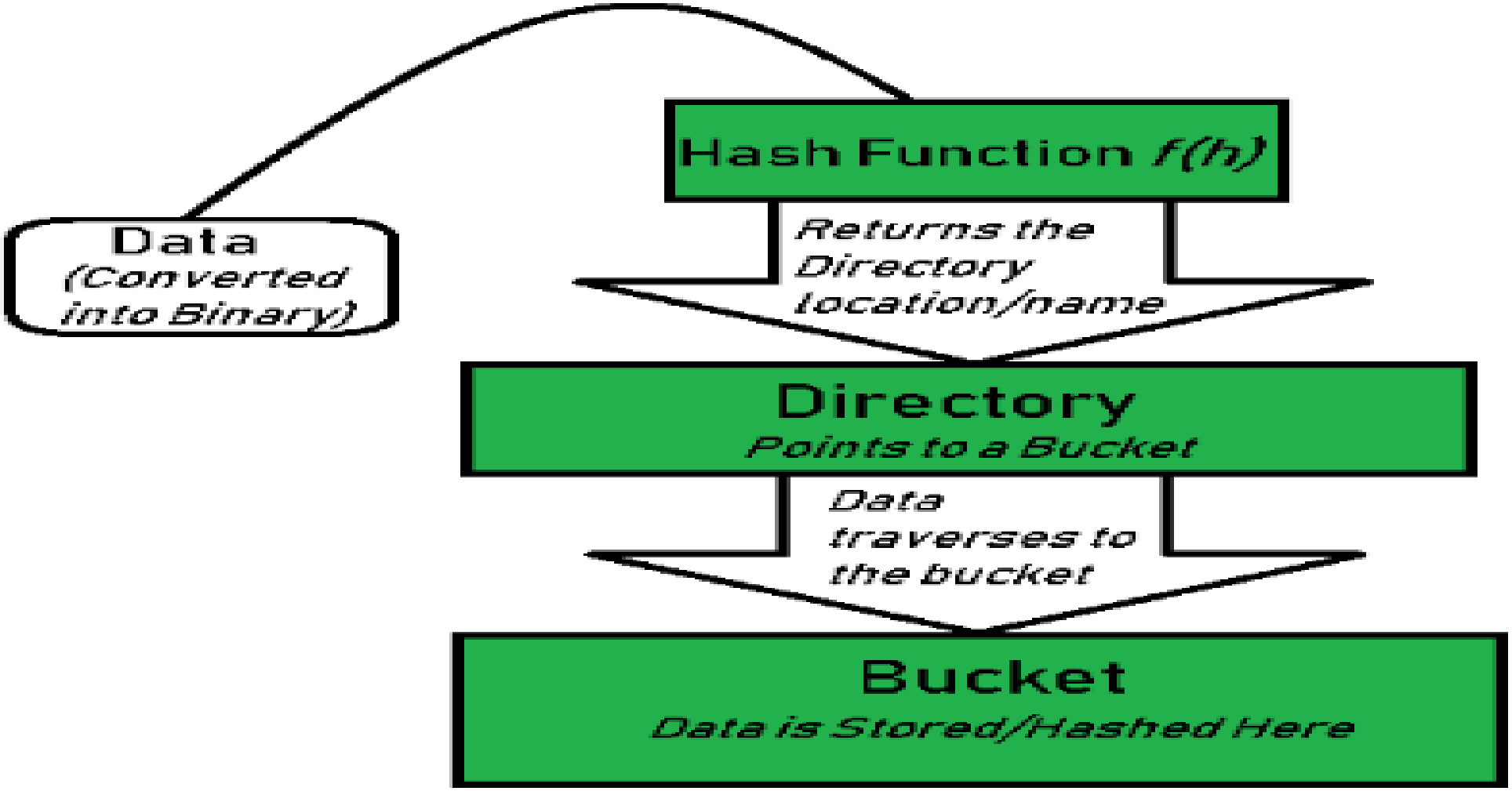
Example



Always ***local depth <= global depth***

A bucket may contain more than one pointers to it if local depth < global depth.

Basic Working of Extendible Hashing



Step 1 – Analyze Data Elements: Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.

Step 2 – Convert into binary format: Convert the data element in Binary form. For string elements, consider the **ASCII** equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.

Step 3 – Check Global Depth of the directory. Suppose the global depth of the Hash-directory is 3.

Step 4 – Identify the Directory: Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. **001**.

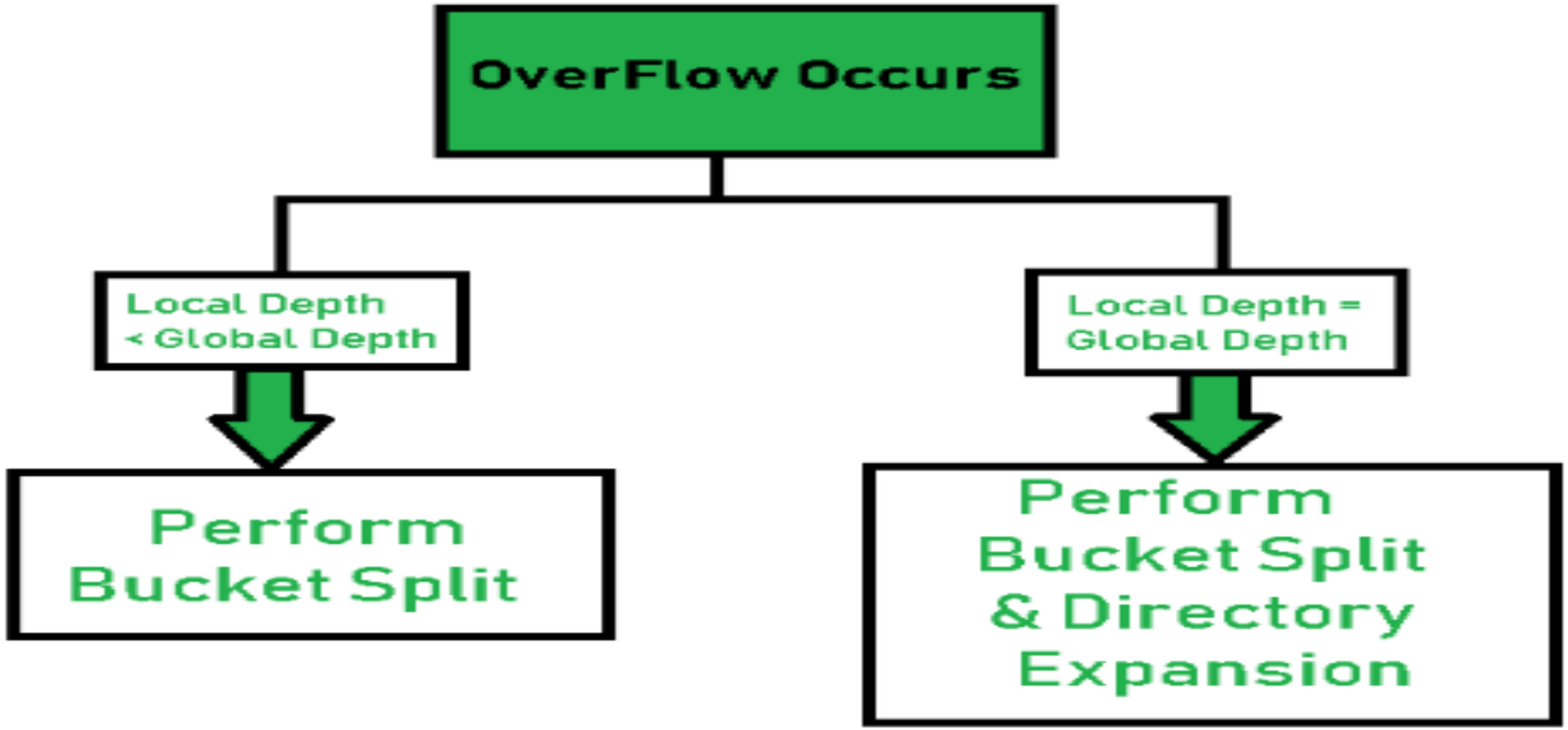
Step 5 – Navigation: Now, navigate to the **bucket** pointed by the directory with directory-id 001.

Step 6 – Insertion and Overflow Check: Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.

Step 7 – Tackling Over Flow Condition during Data Insertion: when Bucket overflows, First, Check if the local depth < or = the global depth. Then choose one of the cases below.

Overflow Handling

- **Case1:** If local depth = global depth, then Directory Expansion, as well as Bucket Split, needs to be performed.
- **Case2:** If local depth < global depth, then only Bucket Split takes place.



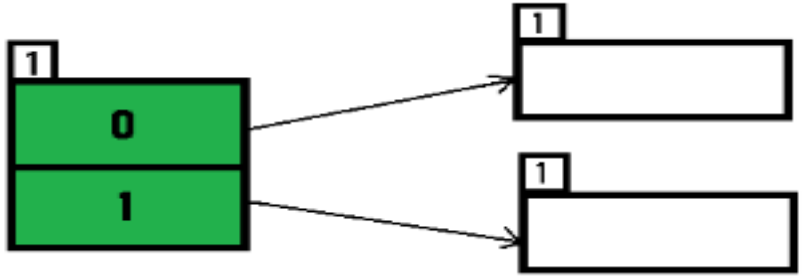
- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 –** The element is successfully hashed.

Example: Now, let us consider an example of hashing the following elements: 16,4,6,22,24,10,31,7,9,20,26.
Bucket Size: 3
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

Binary forms of each of the given numbers

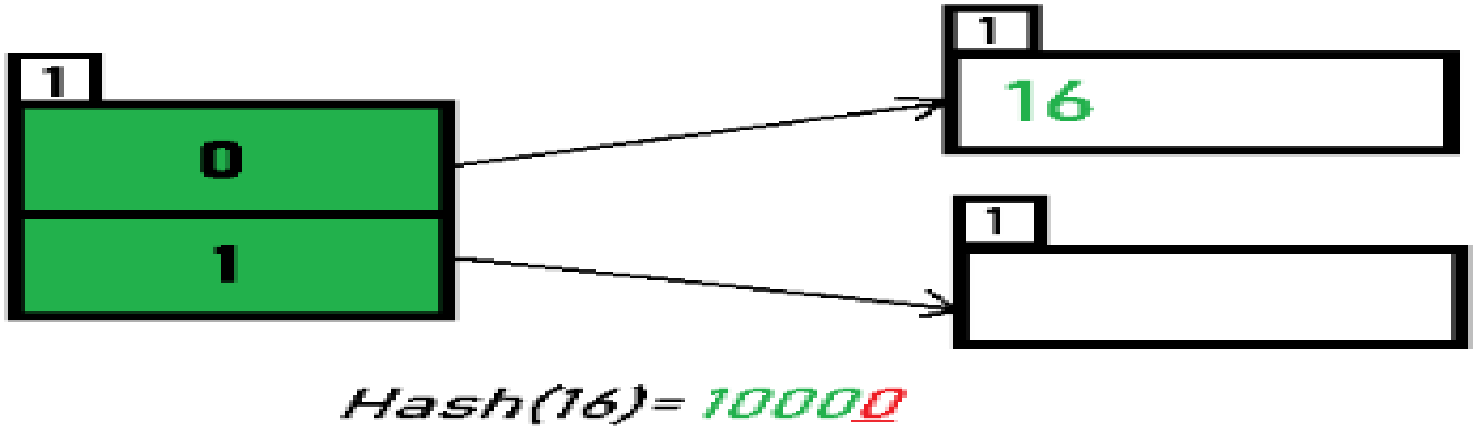
16- 10000
4- 00100
6- 00110
22- 10110
24- 11000
10- 01010
31- 11111
7- 00111
9- 01001
20- 10100
26- 11010

- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



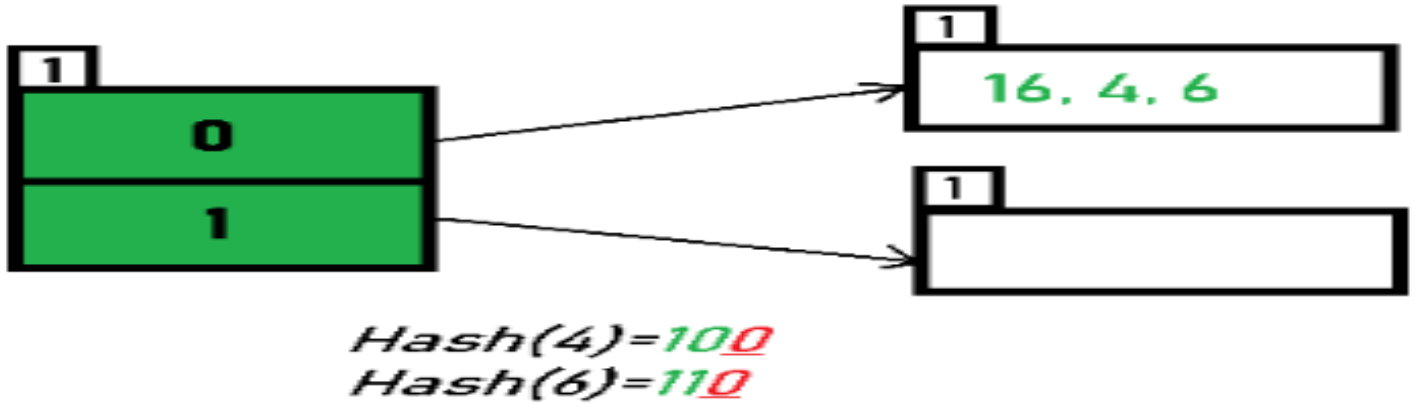
Inserting 16:

The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.

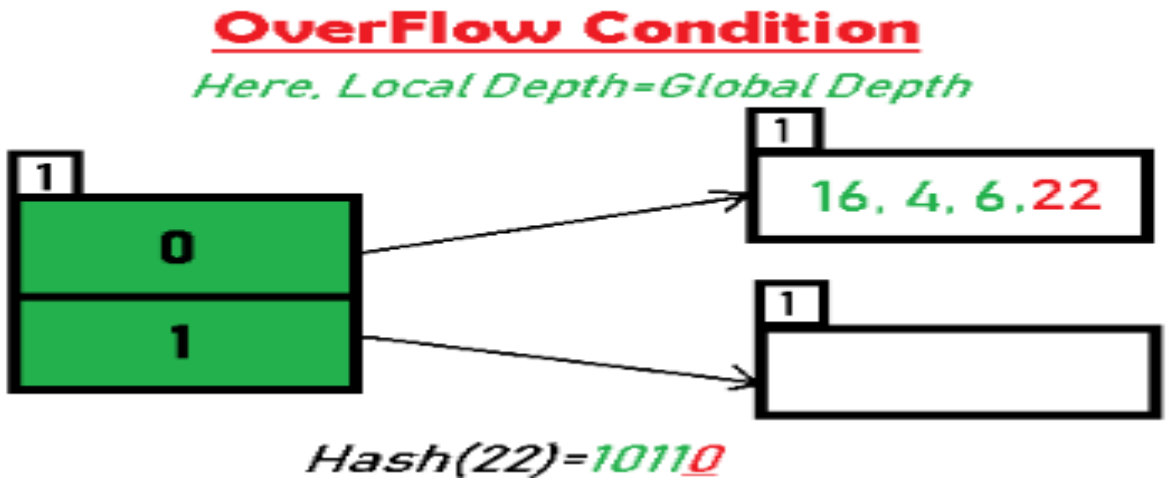


Inserting 4 and 6:

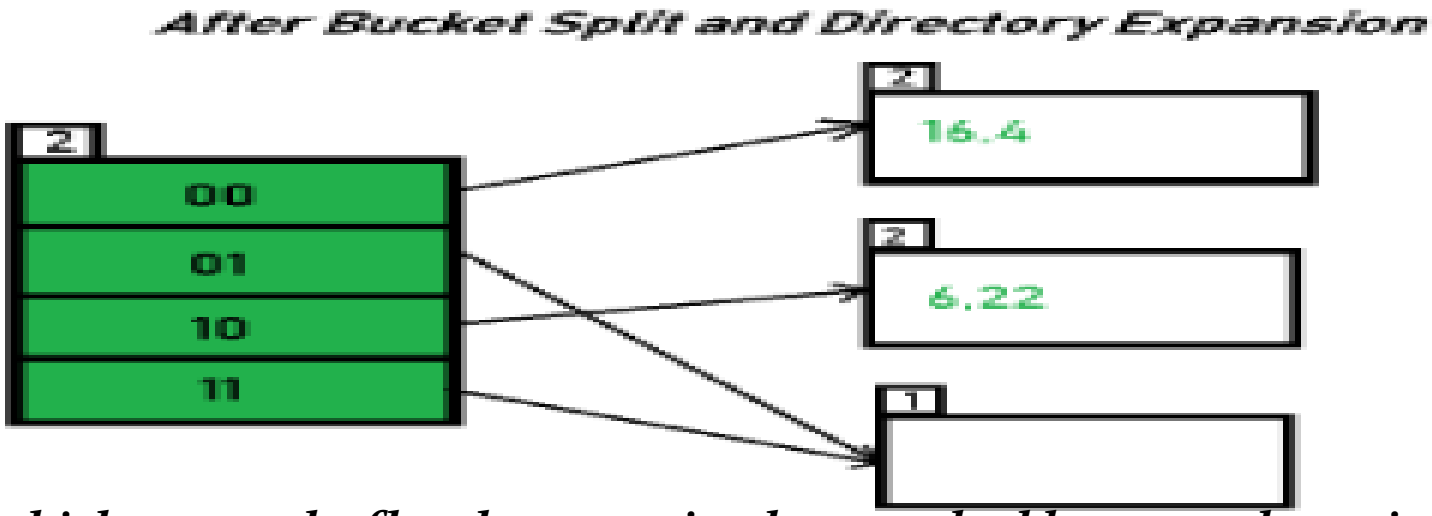
Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



Inserting 22: binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

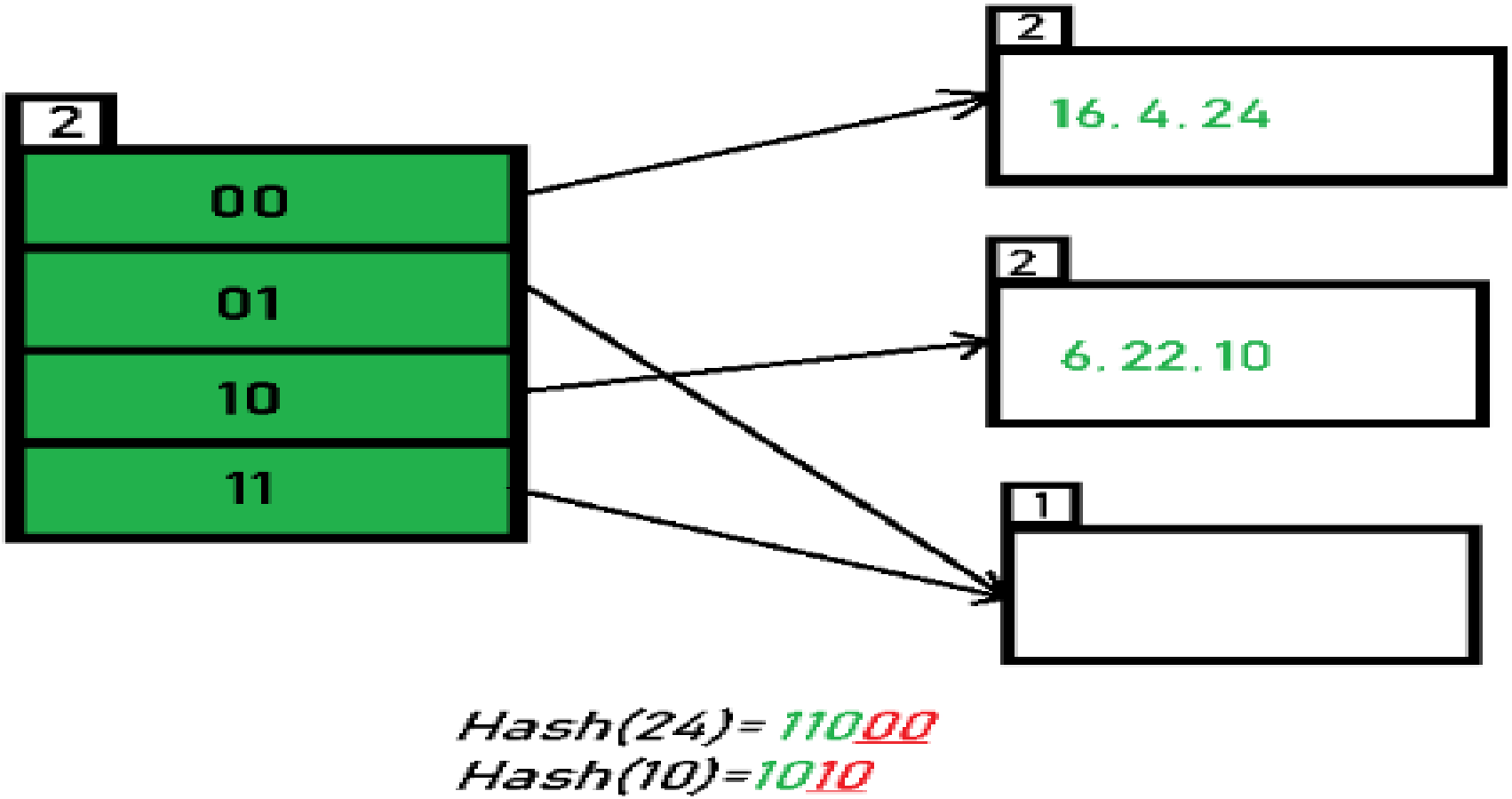


- Apply **Step 7-Case 1**,
- Since Local Depth = Global Depth, the bucket splits and directory expansion takes place.
- Also, rehashing of numbers present in the overflowing bucket takes place after the split.
- since the global depth **is incremented by 1**, now,the global depth is **2**. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

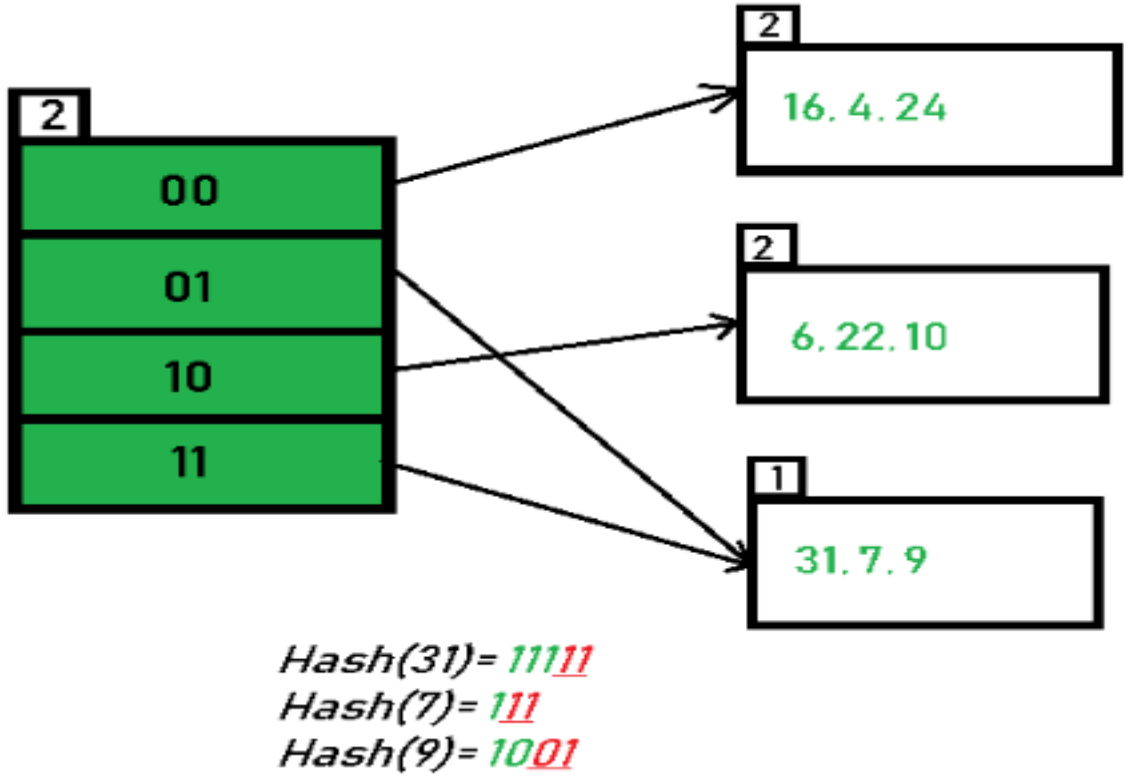


Notice that the bucket which was underflow has remained untouched but 01 and 11 pointing to the same bucket.

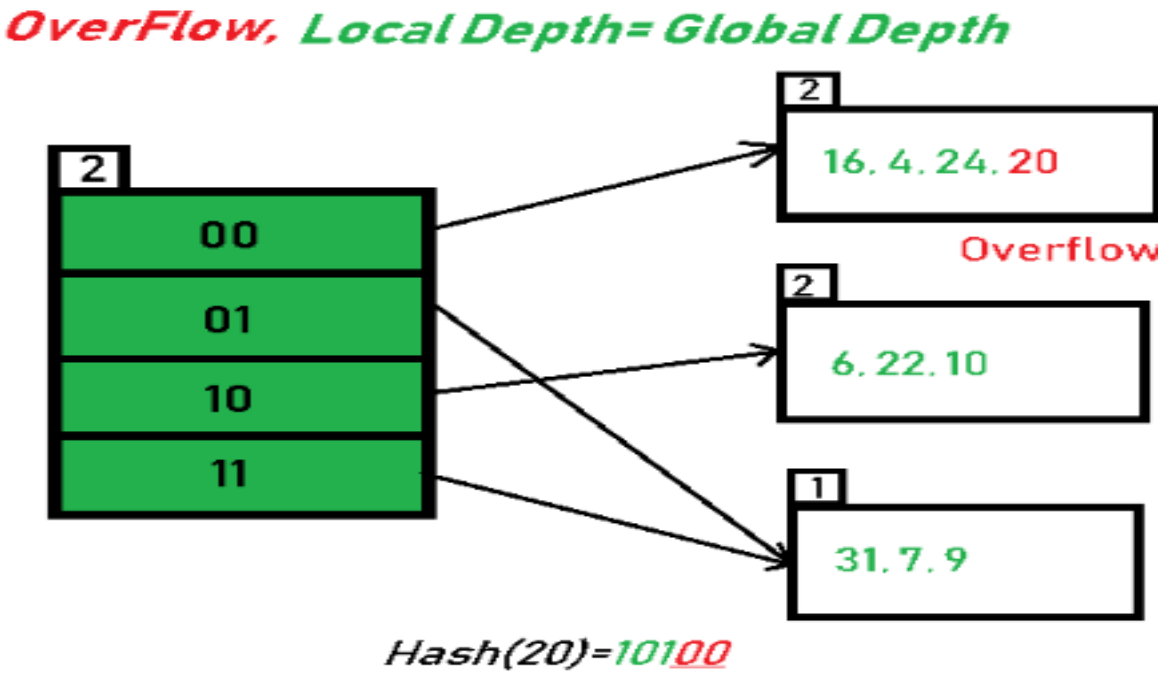
Inserting 24 and 10: 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



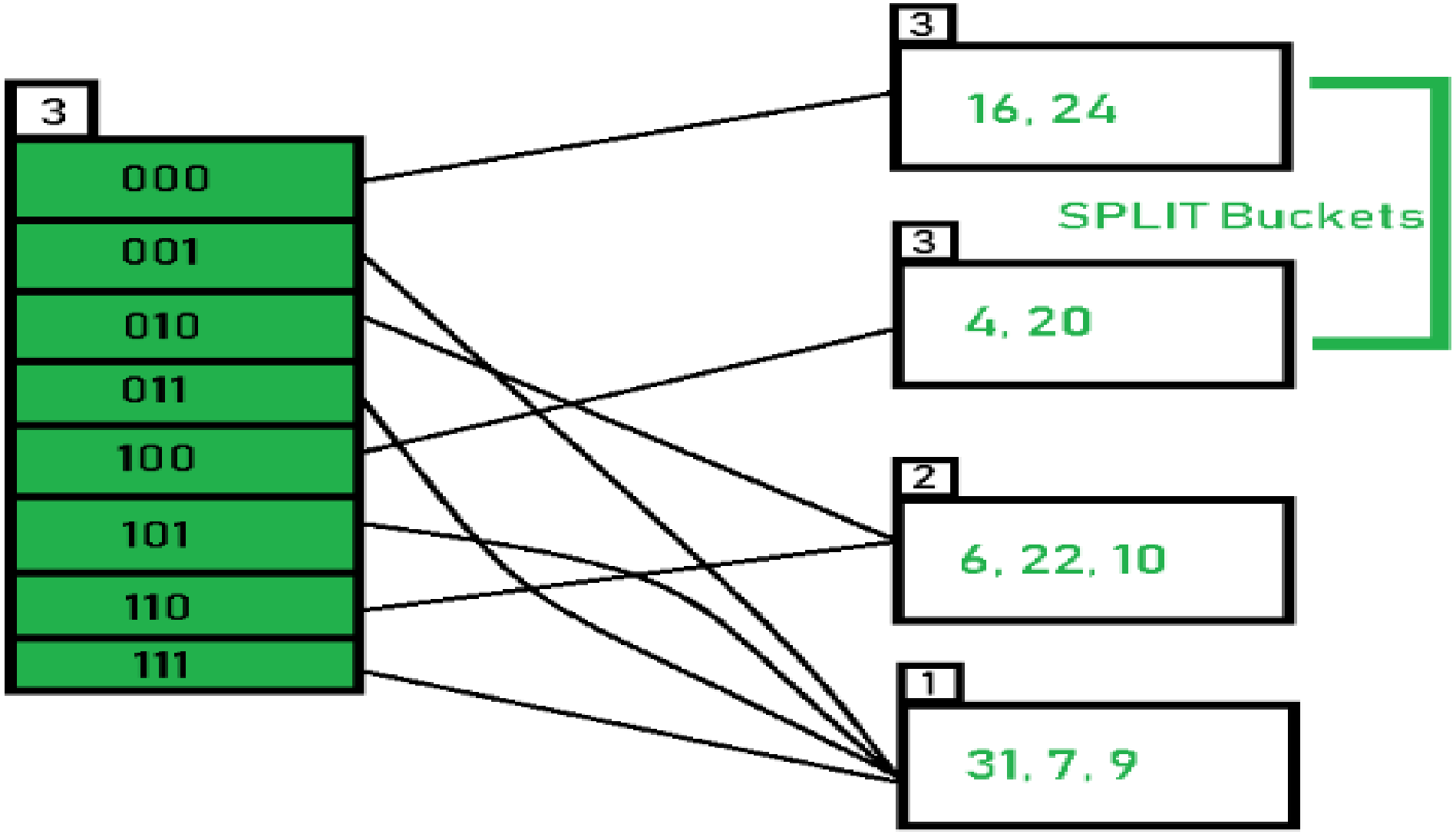
Inserting 31,7,9: All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



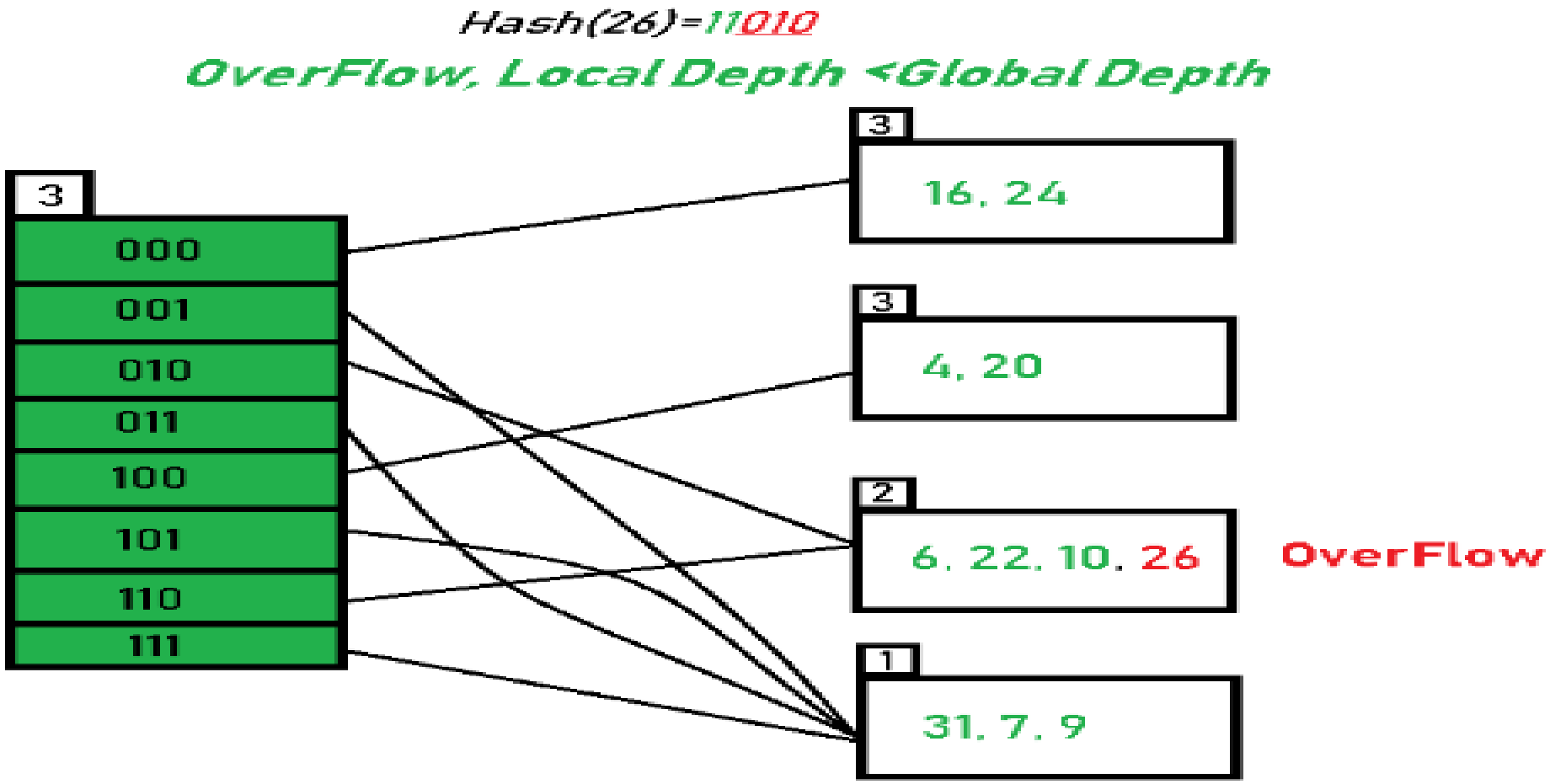
Inserting 20: Insertion of data element 20 (101**00**) will again cause the overflow problem.



20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:

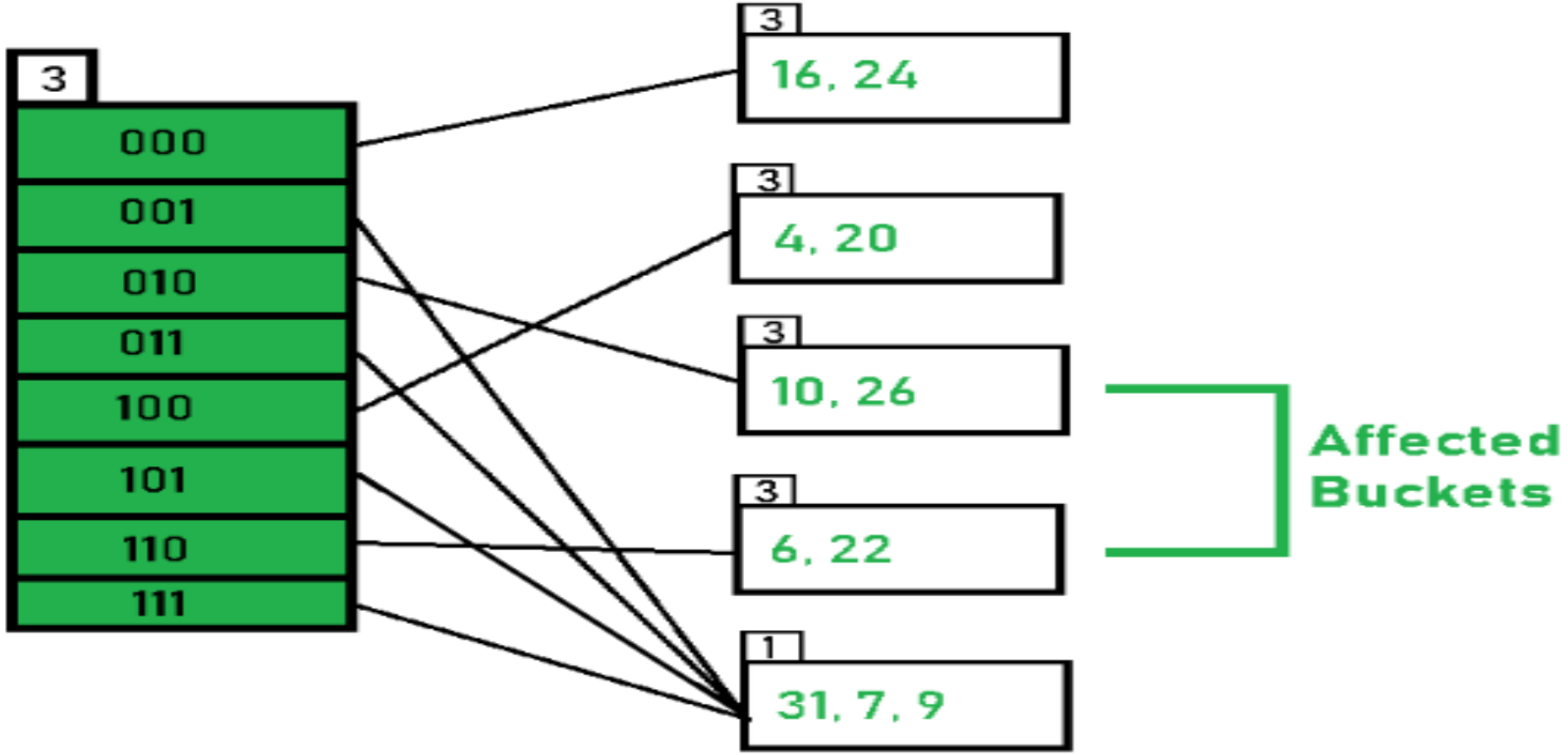

3
000
001
010
011
100
101
110
111
3
16, 24
SPLIT Buckets
3
4, 20
2
6, 22, 10
1
31, 7, 9

Inserting 26: Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.

Finally, the output of hashing the given list of numbers is obtained.



DICTIONARIES

- Dictionary is a collection of Key-Value pair.
- A dictionary is also called a *hash*, a *map*, a *hashmap* in different programming languages
- keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type
- Keys in a dictionary must be unique;
- Example:

```
results = {'Anil' : 7.5,
'Aman' :8.0,
'James' : 8.5,
'Manisha': 7.7,
'Suraj' :8.7,}
```

DICTIONARIES

Basic operations that can be performed on dictionary are:

1. Insertion
2. Deletion
3. Searching of a specific value with the help of key
4. Traverse or Display

Linear List Representation

The dictionary can be represented as a linear list.

The linear list is a collection of key and value.

There are two method of representing linear list.

1. Sorted Array- An array data structure is used to implement the dictionary.

2. Sorted Chain- A linked list data structure is used to implement the dictionary

```
struct Node {
    int key;        // Assuming keys are integers
    int value;      // Assuming values are integers
    struct Node* next;
} *head=NULL, *temp;
```

Search Procedure

```
int search(int key) {

    temp=head
   while (temp != NULL)  do
        if (temp->key == key)
            return temp->value
        temp = temp->next
    return -1}
```

Display Prodecure

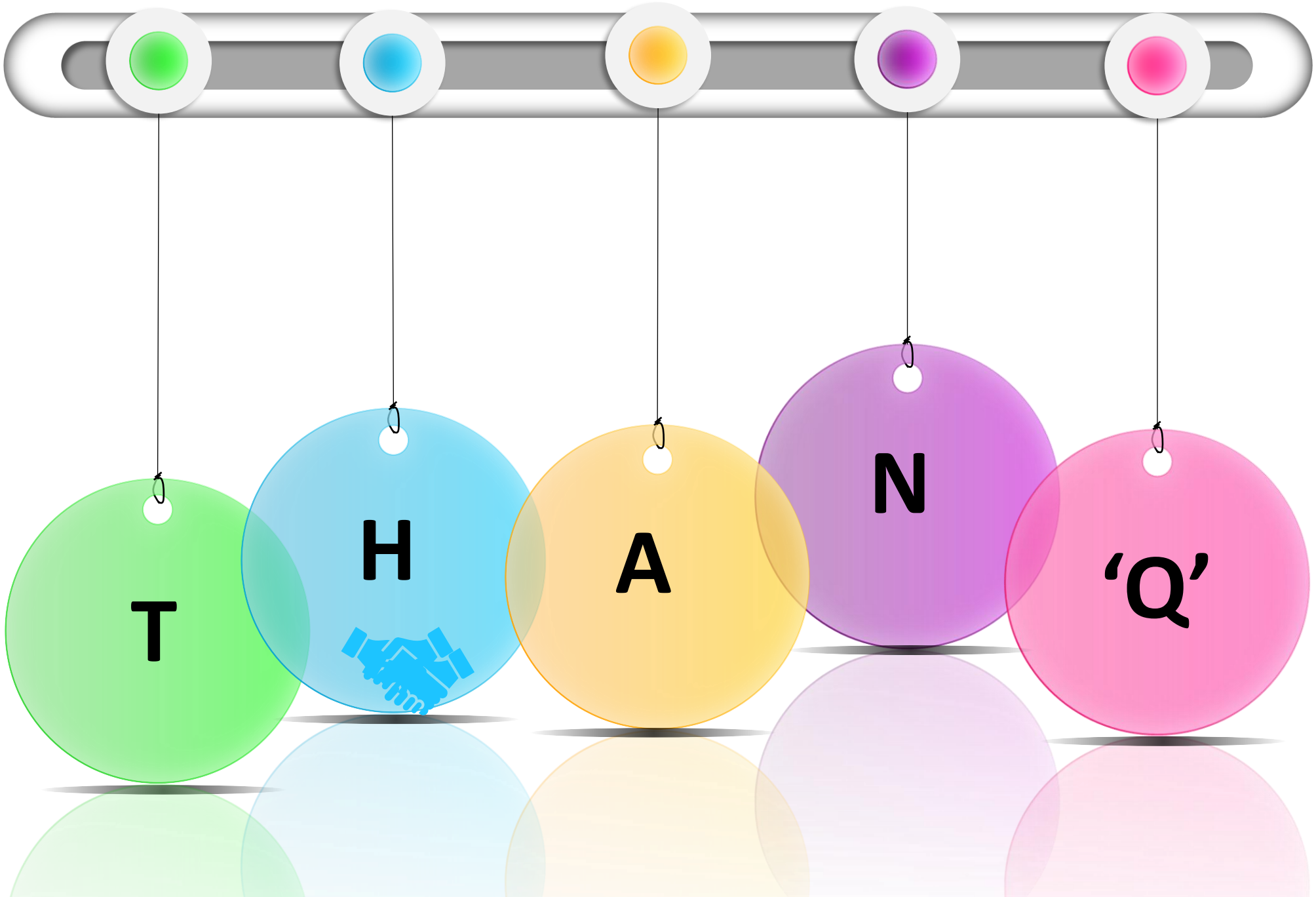
```
display()
{
   temp=head
   while(temp!=NULL) do
      print key and value)
      temp=temp->next
}
```

Deletion Procedure

```
void delete(int key) {
        Traverse and Search for key
        if Key found, delete node
}
```

Insertion Procedure

```
void insert(int key, int value) {
        Traverse and Search for key
    if key not found
        Create a node and read key and value in it
                insert node at end
        else
                Update value
}
```


T
H
A
N
‘Q’

Pattern Matching

Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search results.

Pattern Matching definition: 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.

Applications:

Text editors, Web search engines (e.g. Google), image analysis, Information Retrivel System.

Procedure:

Step 1 : compare the first element of the pattern to be searched ‘p[0]’, with the first element of the string ‘S[0]’ in which to locate ‘p’.

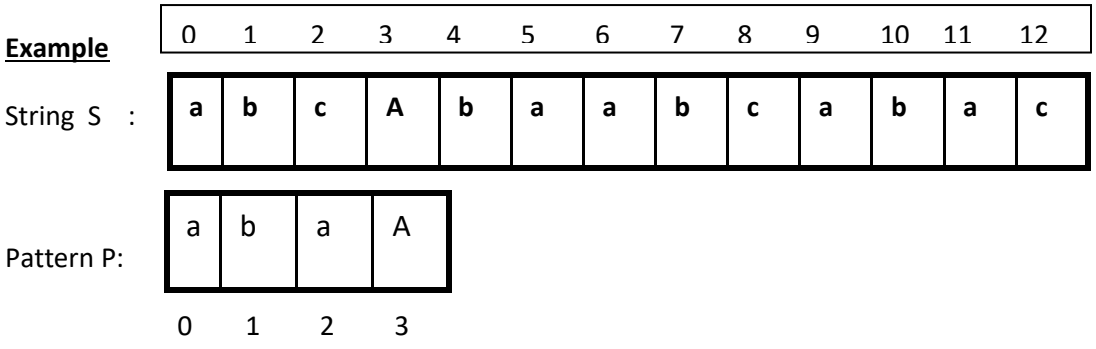
Step 2 : If the first element of ‘p’ matches the first element of ‘S’, then compare the second element of ‘p[1]’ with second element of ‘S[1]’.

Step 3: If match found proceed likewise until entire ‘p’ is found.,Stop process.

Step 4: If a mismatch is found at any position, shift ‘p’ one position to the right and repeat comparison beginning from first element of ‘p’.

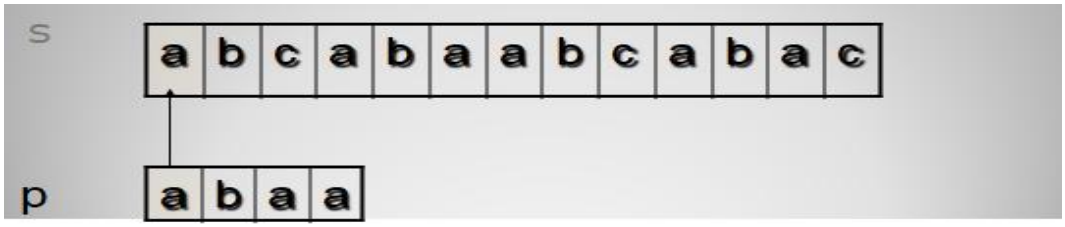
How does the brute force approach work?

Example

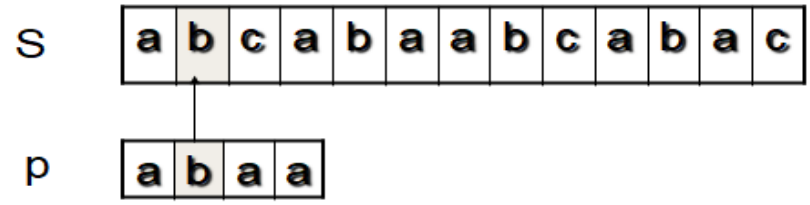


Length of string : n=13, length of pattern m=4

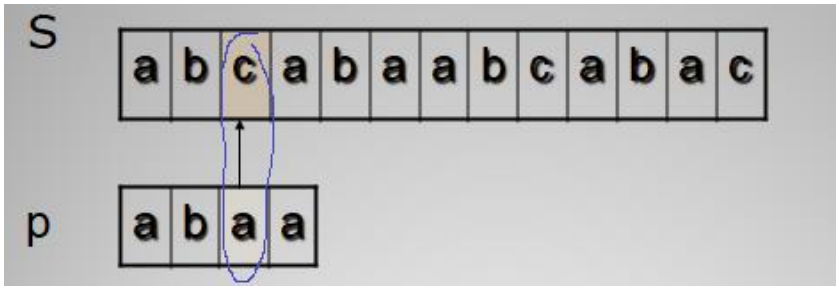
Step 1: compare p[0] with S[0]



Step 2: compare p[1] with s[1]

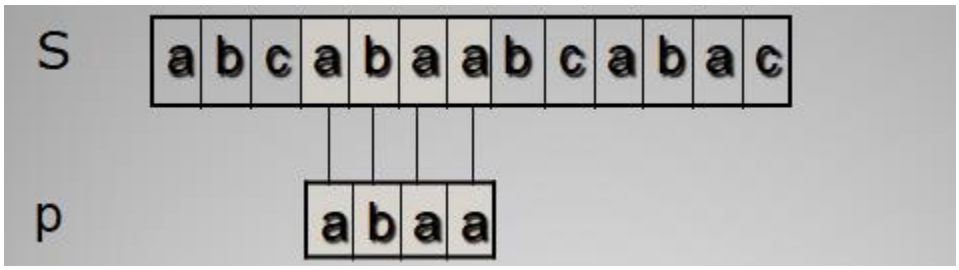


Step 3: compare p[2] with S[2]



Mismatch occurs here…

Since mismatch is detected, shift ‘p’ one position to the left and perform steps to those from step 1 to step 3. At position where mismatch is detected, shift ‘p’ one position to the right and repeat matching procedure.



Finally, a match would be found after shifting ‘p’ three times to the right side. Pattern found at 3 position stop procedure.

Drawbacks of this approach:

1. if ‘m’ is the length of pattern ‘p’ and ‘n’ the length of string ‘S’, the matching time is of the order O(mn). Unnessary comparisions performed in this process.

2. This is a certainly a very slow running algorithm.

3.Time complexity of brute force algorithm is O(mn).

4. Back tracking on the string .

Algorithm/ function brute force

```
int bruteforce(char s[],char p[])
{
int i,j,n,m;
n=strlen(s);
m=strlen(p);
for(i=0;i<=n-m;i++)
{
 for(j=0;j<m;j++)
    {
      if(p[j]!=s[i+j])
```

```
        break;
      }
    if(j==m)
       return i;
}
return -1;
}
```

Program to implement brute force pattern matching algorithm.

```
#include<stdio.h>
#include<string.h>
int bruteforce(char s[],char p[])
{
int i,j,n,m;
n=strlen(s);
m=strlen(p);
for(i=0;i<=n-m;i++)
{
 for(j=0;j<m;j++)
    {
```

```
        if(p[j]!=s[i+j])
          break;
       }
     if(j==m)
        return i;
}
return -1;
}
void main()
{
char s[100],p[10];
int m;
printf("enter the text\n");
gets(s);
printf("enter the pattern\n");
gets(p);
m=bruteforce(s,p);
if(m==-1)
 printf("pattern not found");
```

```
else
printf("pattern found at %d ",m);
}
```

Knuth-Morris-Pratt Algorithm(KMP)

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.

A matching time of O(n) is achieved by avoiding comparisons with elements of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Components of KMP algorithm

- LPS table /failure function,

The failure function for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.

- This information can be used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.
- The KMP Matcher

With string 'S', pattern 'p' and prefix function 'F' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

Following LPS table computing the failure fucnction:

```
Algorithm failure(p)
{
f[0]=0;
i=1,j=0;
While(i<m)
{
      if(p[i]==p[j] {
            f[i]=j+1;
              i++;
              J++;
            }
       else if(j>0)
        j=f[j-1]
      else{
           fi]=0
          i++;
}
```

Example: compute LPS or F array for the pattern ‘p’ below:

0	1	2	3	4	5	6
a	b	a	b	a	c	a

Initially: m = length[p] = 7
f[0] = 0
i = 1, j=0

Step 1: i=1, p[1]=p[0] (a==b) false
f[1]=0 (j==0), f[i]=0

Step 2: i = 2, j = 0,p[2]==p[0],a==a true
f[i]=j+1, f[2]=1,i++,j++

Step 3: i = 3, j = 1,p[3]=p[1],b==b , true
f[i]=j+1, f[3]=1+1=2 ,i++,j++

Step 4: i = 4, j =2,p[4]=p[2] , a==a, true
f[4]=j+1, f[4]=2+1=3 ,i++,j++

Step 5: i = 5, j = 3 , p[5]=p[3] ,b=c , false ,j>0
J=f[j-1]=f[3-1]=f[2]=1

Step 6: i = 5, j = 1 , p[5]=p[1] ,b=c , false ,j>0
J=f[j-1]=f[1-1]=f[0]=0

Step 7: i = 5, j = 0 , p[5]=p[0] ,a=c , false ,j=0
f[5]=0, i++

Step 8: i = 6, j = 0 , p[6]=p[0] ,a=a , true
f[6]=j+1=0+1=1

After final Iteration, the failure function computation is complete

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
f	0	0	1	2	3	0	1

KMP matcher function using f[] /LPS table

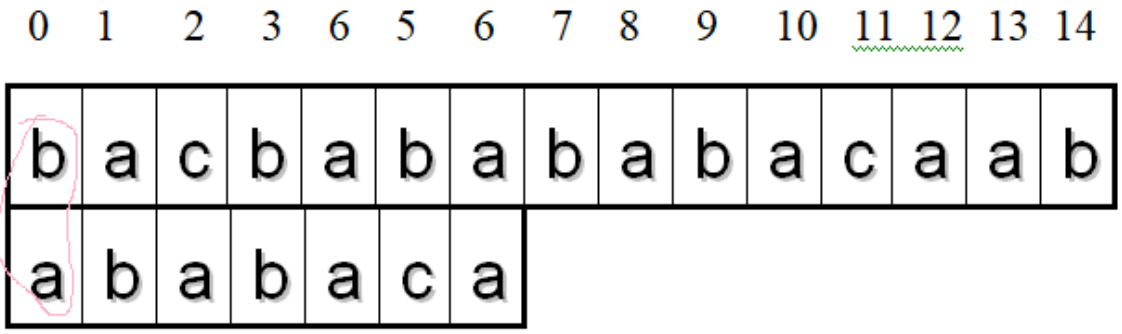
```
int kmp(char s[],char p[])
{
  int i=0,j=0,n,m;
 failure(p);
 n=strlen(s);
 m=strlen(p);
 while(i<n)
```

```
{
 if(s[i]==p[j])
  {
    if(j==m-1)
       return i-j;
       else
         {
          i++;
          j++;
          }
     }
     else    // unmatch occur
      {
          if(j>0)
           j=f[j-1];
           else
              i++;
          }
 }// while
```

return -1;

}

Example : Apply KMP on s=”bacbababacaab” and pattern p=”

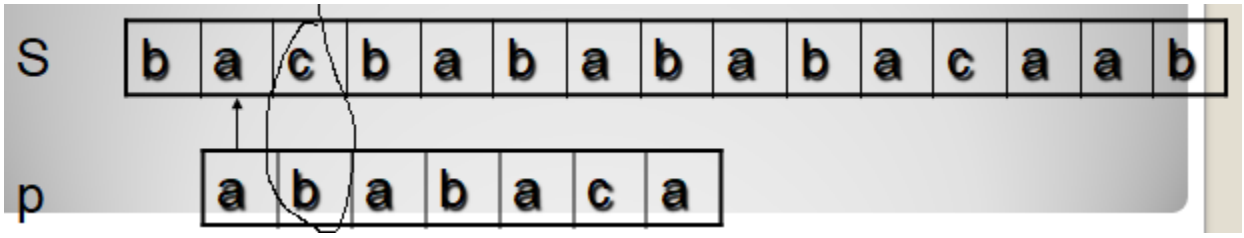


Initially: n = size of S = 15;

m = size of p = 7

Step 1: i = 0, j = 0

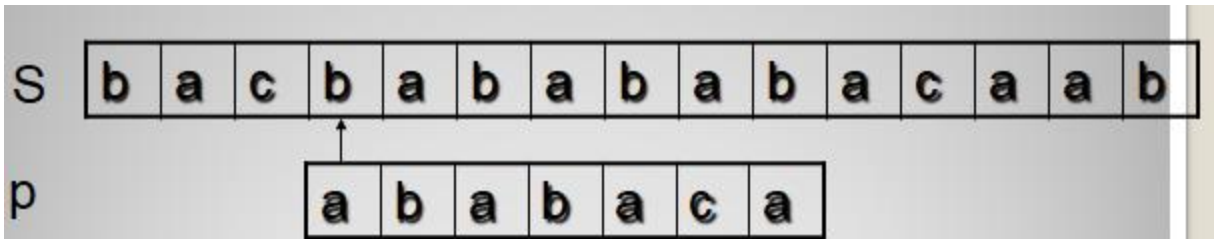
comparing p[0] with S[0] ,a==b ,false,i++



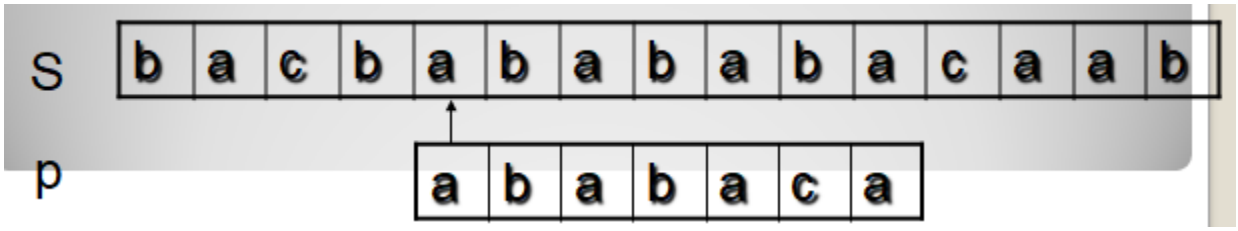
step 2:i=1,j=0 ,compare p[0] with s[1] ,a=a ,true,i++,j++

step 3 i=2, j=1, compare p[1] with s[2] ,b=c, false i++,j>0

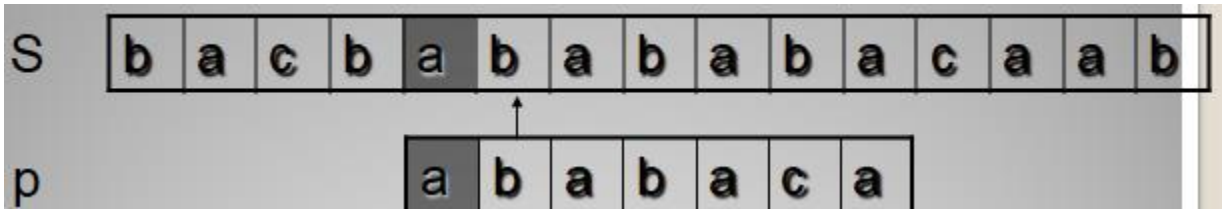
j=f[j-1]=f[0]=0



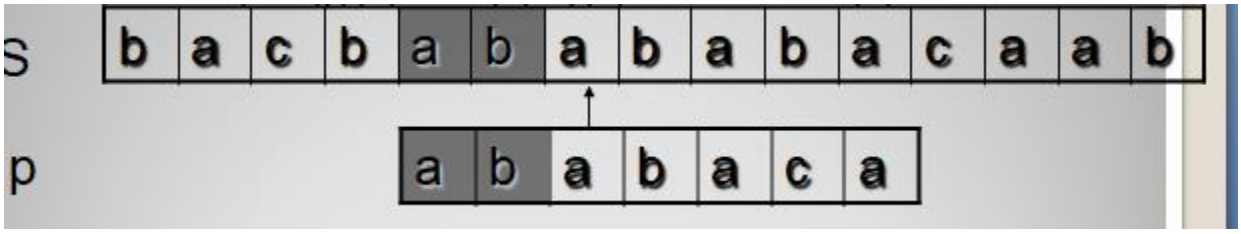
step 4: i=3, j=0, compare p[0] with s[3] ,b=a, false, j=0,i++



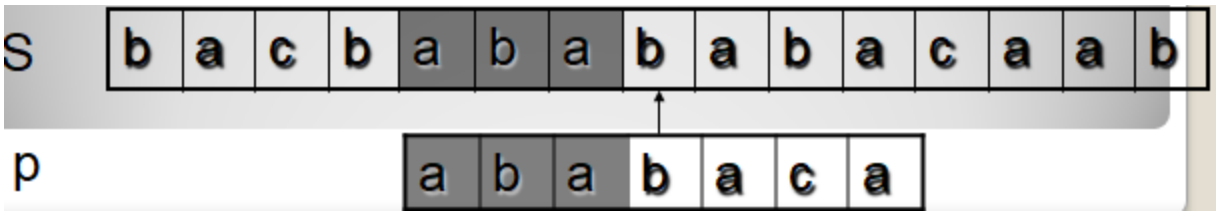
Step 5: i=4,j=0, compare p[0] with s[4] ,a=a, true, j++,i++



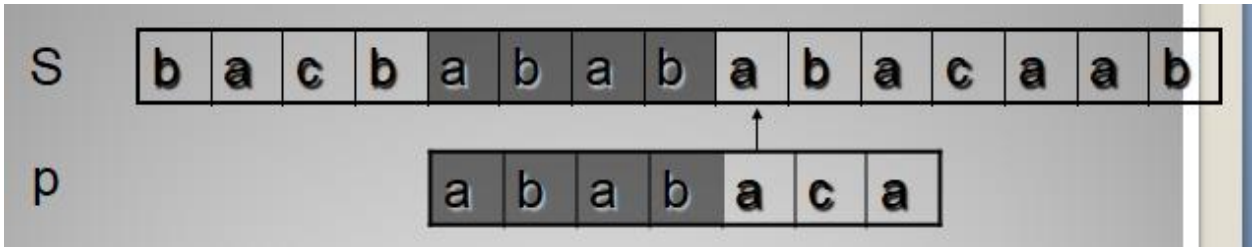
Step 6: i=5,j=1, compare p[1] with s[5] ,b=b, true, j++,i++



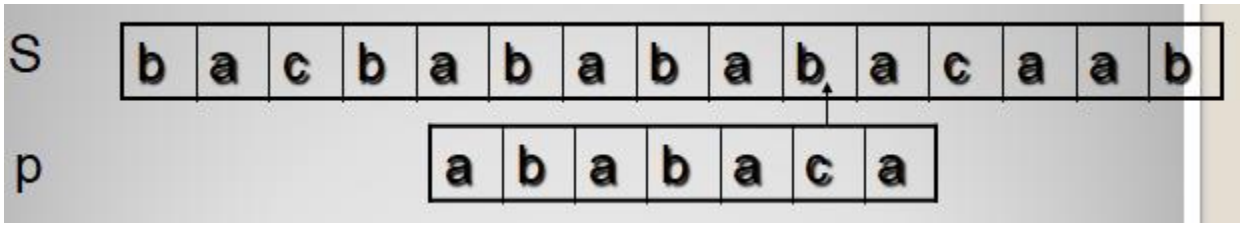
Step 7: i=6,j=2, compare p[2] with s[6] ,a=a, true, j++,i++



Step 8: i=7,j=3, compare p[3] with s[7] ,b=b, true, j++,i++

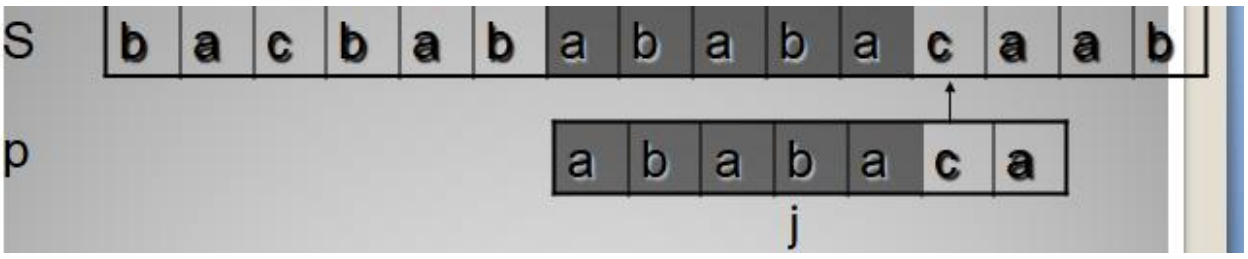


Step 8: i=8,j=4, compare p[4] with s[8] ,a=a, true, j++,i++



Step 8: i=9,j=5, compare p[5] with s[9] ,b=c, false ,

j>0,j=f[j-1] ,j=f[j5-1]=f[4]=3

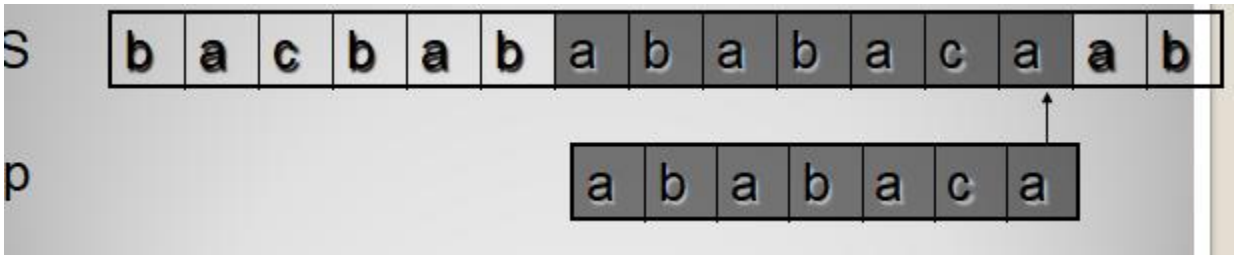


Step 9 : i=9,j=3, compare p[3] with s[9] ,b=b, true, j++,i++

Step 10 : i=10,j=4, compare p[4] with s[10] ,a=a, true, j++,i++

Step 11 : i=11,j=5, compare p[5] with s[11] ,c=c, true, j++,i++

Step 12 : i=12,j=6, compare p[6] with s[12] a=a, true, j=m-1=6



12-6=6

Therefore pattern occurs at 6 position.

Time complexity of KMP =O(m)+O(n)=O(m+n)

Where ‘m’ is length of pattern O(m) is time complexity of construction of LPS

Where ‘n’ is length of pattern O(n) is time complexity of KMP matcher

Write a program to implement KMP algorithm

```
#include<string.h>

#include<stdio.h>

int f[100];

void failure(char p[])

{

int i=1,j=0,m;
```

```
m=strlen(p);
f[0]=0;
while(i<m)
{
  if(p[j]==p[i])
     {
      f[i]=j+1;
      i++;
       j++;
     }
  else if(j>0)
     f[i]=f[j-1];
    else
    {
      f[i]=0;
       i++;
      }
   }//while
}
```

```
int kmp(char s[],char p[])
{
  int i=0,j=0,n,m;
 failure(p);
 n=strlen(s);
 m=strlen(p);
 while(i<n)
 {
  if(s[i]==p[j])
   {
     if(j==m-1)
        return i-j;
        else
          {
            i++;
            j++;
            }
      }
      else
```

```
        {
          if(j>0)
             j=f[j-1];

             else
                i++;
            }
  }// while
    return -1;
}
void main()
{

char p[10],s[100];
printf("enter text\n");
gets(s);
printf("enter pattern\n");
gets(p);
int l=kmp(s,p);
```

```
if(l==-1)
 printf("pattern not found");
else
printf("pattern found at position=%d",l);
}
```

Boyer-Moore pattern matching algorithm

Boyer-Moore is *significantly faster than brute force* for searching English text.

Boyer Moore is a combination of the following two approaches.

1. Bad Character Heuristic
2. Good Suffix Heuristic

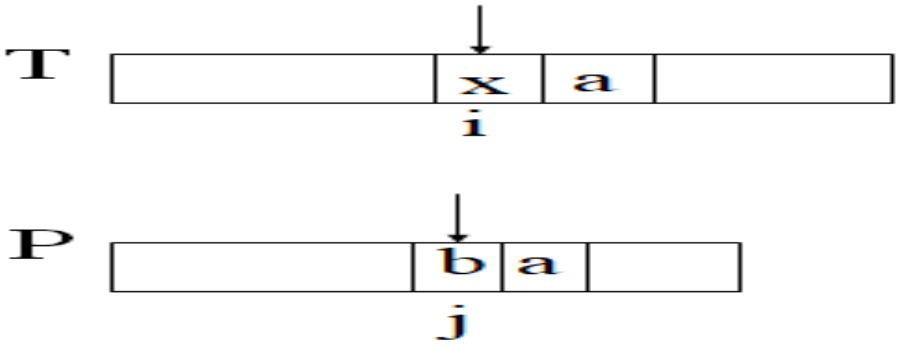
Boyer –Moore using Bad Character approach

The Boyer-Moore pattern matching algorithm is based on two steps.

1 .Back ward alignment: find P in T by moving ***backwards*** **through P**, starting at its end.

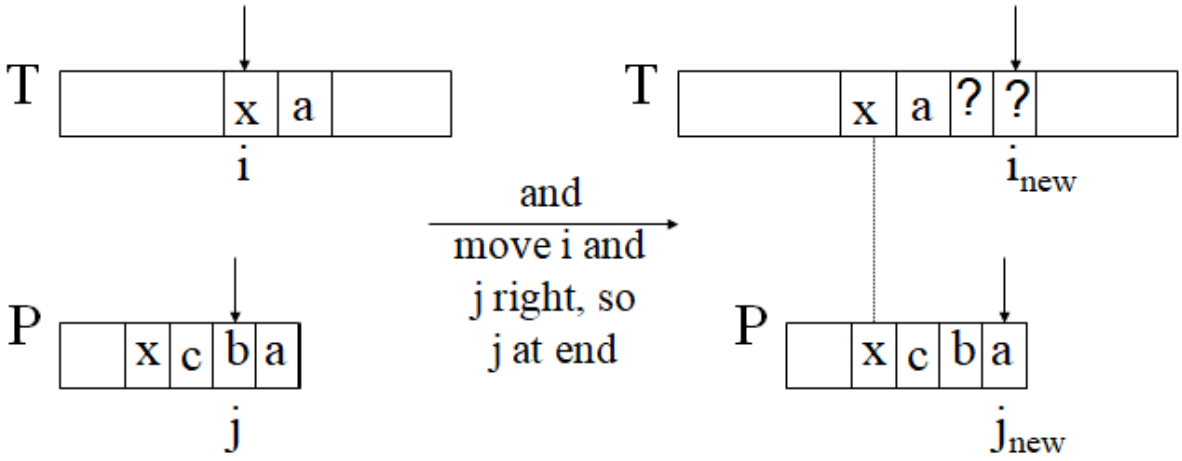
2. Bad Character match : when a mismatch occurs at T[i] == x ,the character in pattern P[j] is not the same as T[i]

There are 3 possible cases, tried in order.



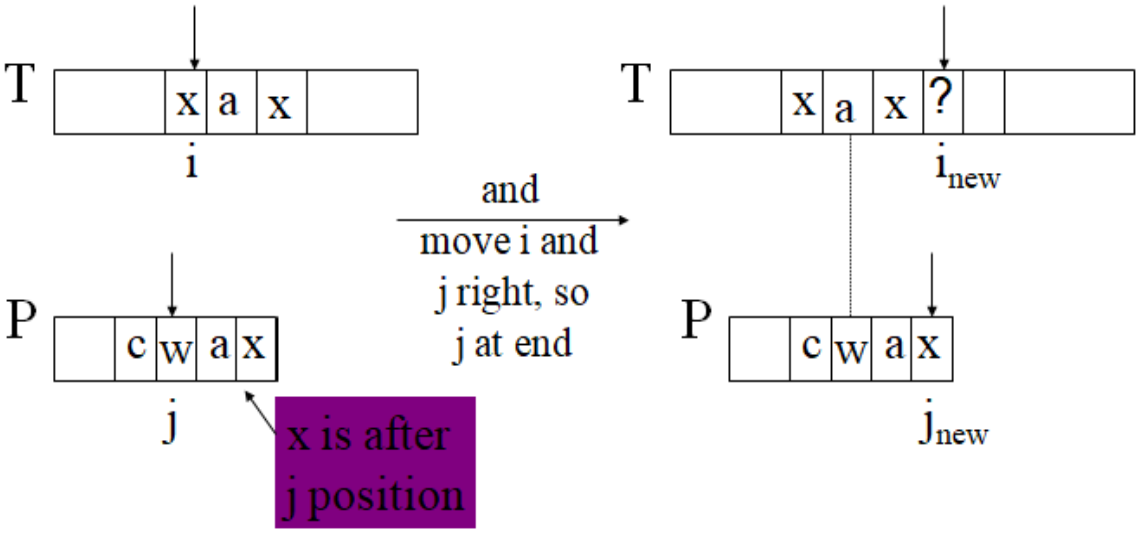
Case 1

If P contains x somewhere, then try to shift P right to align the last occurrence of x in P with T[i].



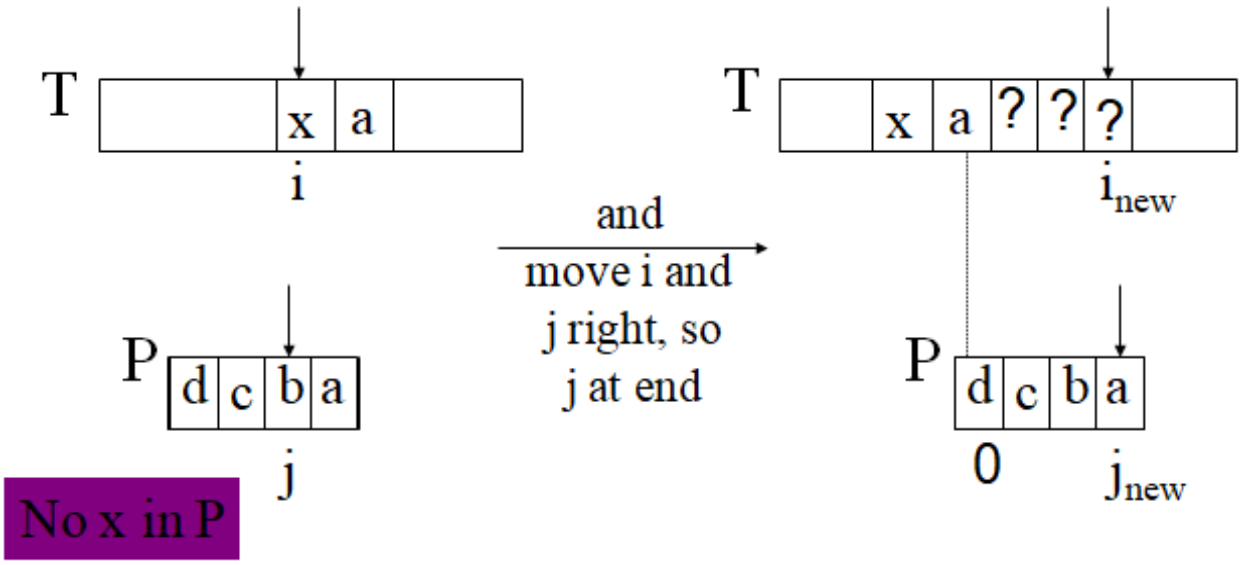
Case 2

If P contains x somewhere, but a shift right to the last occurrence is *not* possible, then *shift P* right by 1 character to T[i+1].

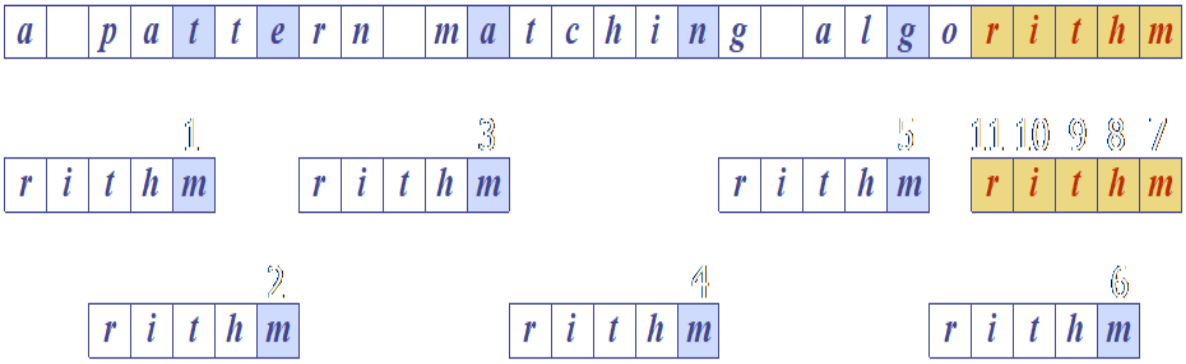


Case 3

If cases 1 and 2 do not apply, then shift P to align P[0] with T[i+1].



Boyer-Moore Example 1: String s, s= "a pattern matching algorithm" and p="rithm"



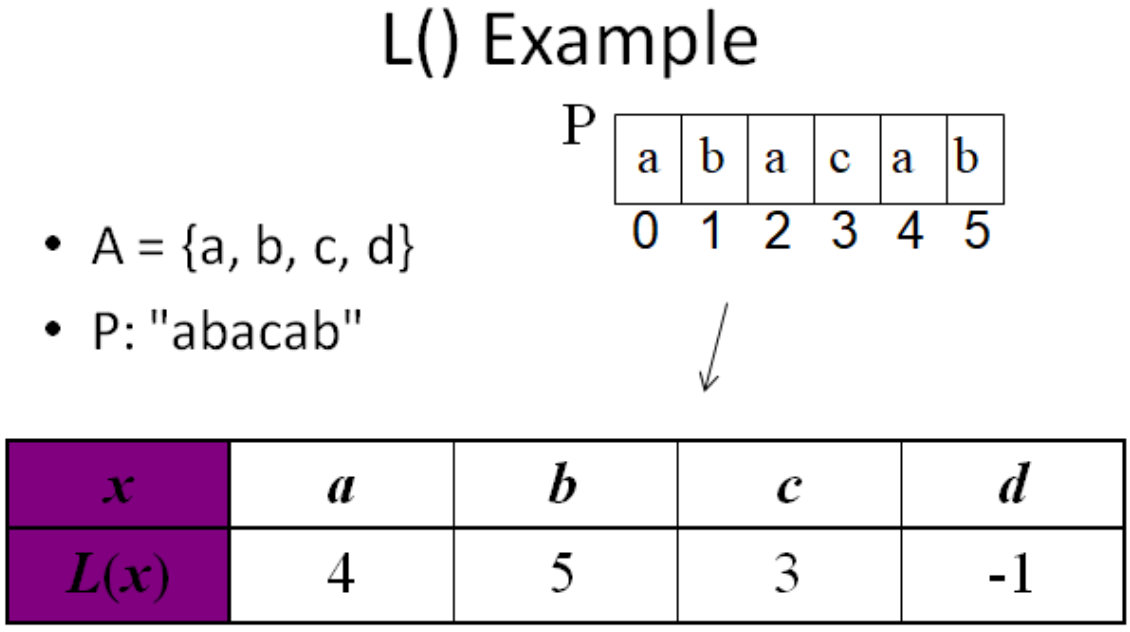
Last Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet A to build a last occurrence function L()

L() maps all the letters in A to integers

- L(x) is defined as: // x is a letter in A
  - the largest index i such that P[i] == x, or
  - -1 if no such index exists

L() Example

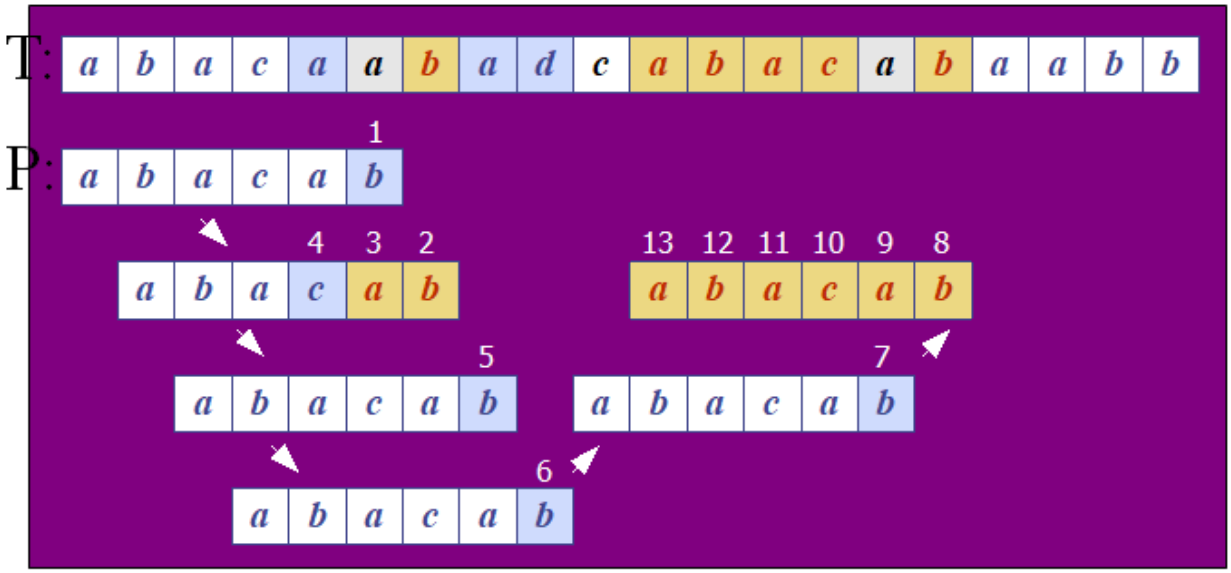


x	*a*	*b*	*c*	*d*
L(x)	4	5	3	-1

L() stores indexes into P[]

- *In Boyer-Moore code, L() is calculated when the pattern P is read in.*
- *Usually L() is stored as an array*

Boyer-Moore Example (2)



x	*a*	*b*	*c*	*d*
L(*x*)	4	5	3	−1

Function L() last occurance

```
void buildLast(char p[])
/* Return array storing index of last    occurrence of each ASCII char in pattern. */
 {
 int i;
    for(i=0; i < 128; i++)
     last[i] = -1; // initialize array
    for (i = 0; i < strlen(p); i++)
           last[p[i]] = i;
        }
       // end of buildLast()
```

Comparision bmMatch()

```
int bmMatch(char s[],char p[])
```

```
{
buildLast(p);
int n = strlen(s);
int m = strlen(p);
int i = m-1;
if (i > n-1)
return -1; // no match
int j = m-1;
       do {
       if (p[j] == s[i])
       if (j == 0)
       return i; // match
      else {
                  i--; j--;
              }
           else { // character jump technique
           int lo = last[s[i]];  //last occ
           i = i + m - min(j, 1+lo);
           j = m - 1;
                }
        } while (i <= n-1);
         return -1; // no match
     } // end of bmMatch()
```

Write a program to implement Boyer-Moore Pattern Matching algorithm

```
#include<stdio.h>
#include<string.h>
int last[128]; // ASCII char set
int min(int a,int b)
{
if(a<b) return a;
else return b;
}
void buildLast(char p[])
 /* Return array storing index of last
  occurrence of each ASCII char in pattern. */
  {
  int i;
     for(i=0; i < 128; i++)
      last[i] = -1; // initialize array
     for (i = 0; i < strlen(p); i++)
            last[p[i]] = i;
         }
        // end of buildLast()
int bmMatch(char s[],char p[])
{
buildLast(p);
```

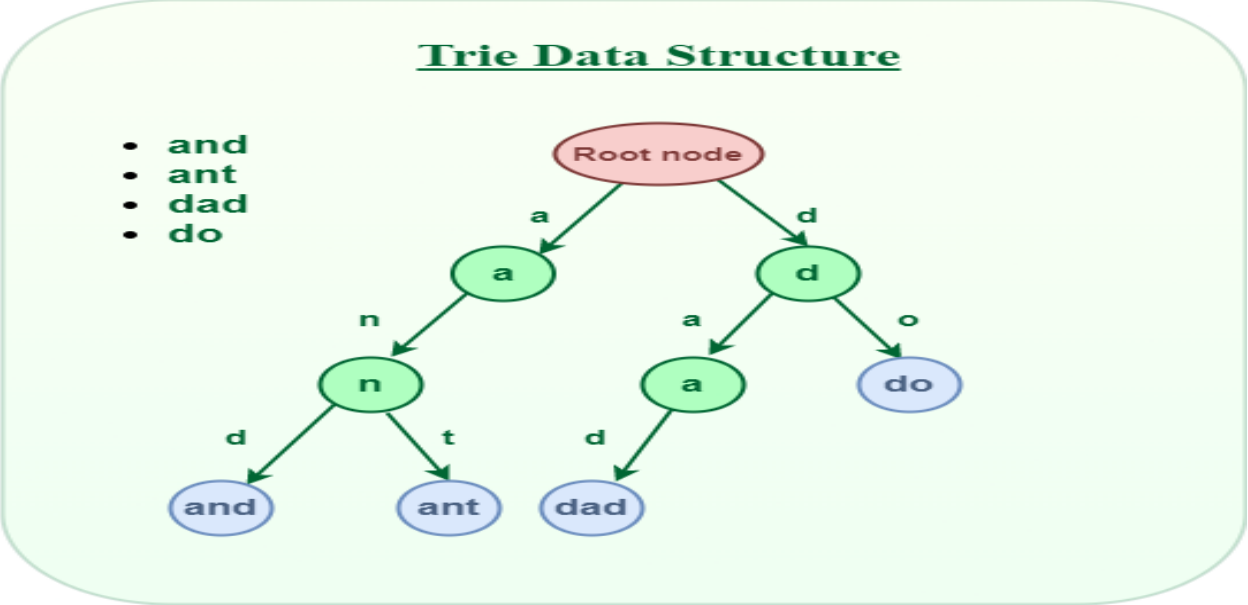
```
int n = strlen(s);
int m = strlen(p);
int i = m-1;
if (i > n-1)
return -1; // no match
int j = m-1;
       do {
       if (p[j] == s[i])
       if (j == 0)
       return i; // match
      else {
                  i--; j--;
              }
            else { // character jump technique
            int lo = last[s[i]];  //last occ
            i = i + m - min(j, 1+lo);
            j = m - 1;
                }
        } while (i <= n-1);
         return -1; // no match
    } // end of bmMatch()
 void main()
  {
```

```
  char s[100],p[50];
   printf("enter text");
   gets(s);
   printf("enter pattern");
   gets(p);
int posn = bmMatch(s, p);
if (posn == -1)
    printf("Pattern not found");
    else
    printf("Pattern starts at position %d",posn);
    }
```

Tries

- A trie (derived from retrieval) is a multiway tree data structure used for storing strings over an alphabet.
- It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.
- The idea is that all strings sharing common prefix should come from a common node.
- The tries are used in spell checking programs.

- A trie is a data structure that supports pattern matching queries in time proportional to the pattern size.
- Properties of the Trie for a set of the string:
- The root node of the trie always represents the null node.
- Each child of nodes is sorted alphabetically.
- Each node can have a maximum of **26** children (A to Z).
- Each node (except the root) can store one letter of the alphabet.



Advantages of tries

- 1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of keys depends upon the height in case of binary search tree.
- 2. Tries take less space when they contain a large number of short strings. As nodes are shared between the keys.
- 3. Tries help with longest prefix matching, when we want to find the key.

Tries are classified into three categories:

- Standard Trie
- Compressed Trie
- Suffix Trie

Standard Trie A standard trie have the following properties:

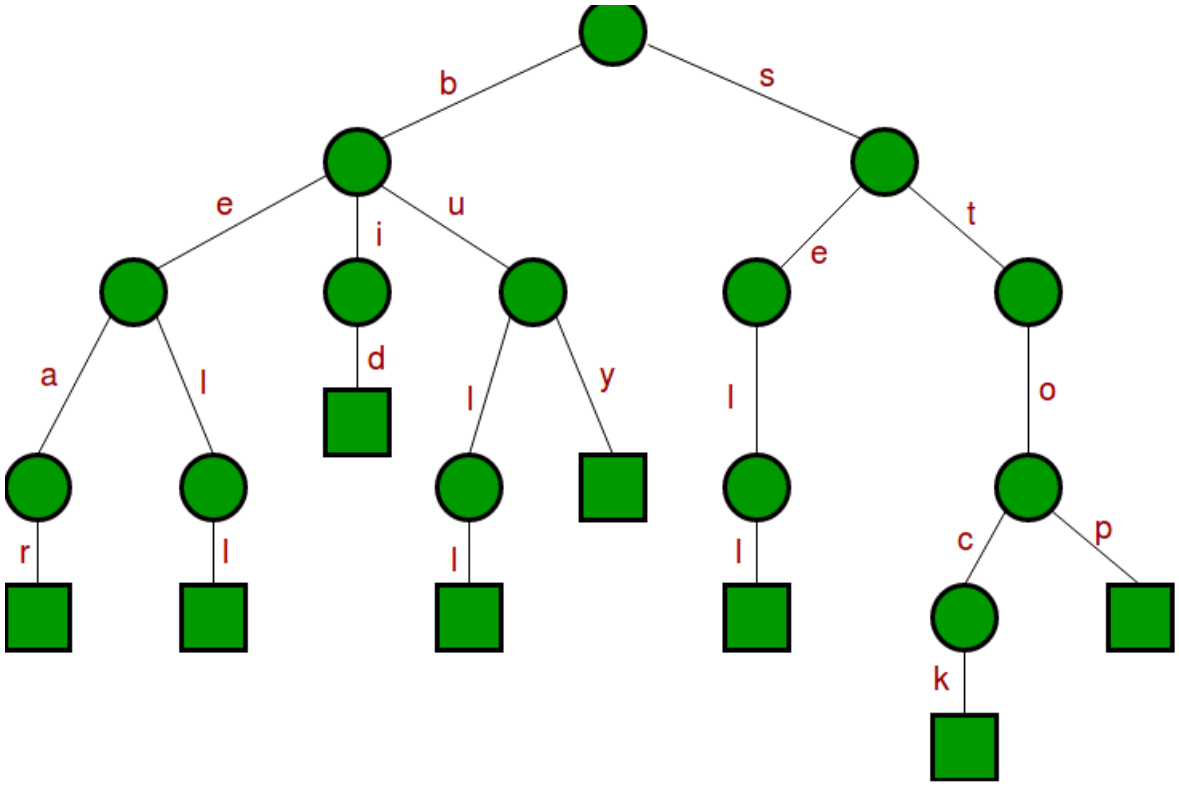
- A Standard Trie has the below structure

```
struct node {
char  data;
struct node* link[26];
```

};

- Each node(except the **root** node) in a standard trie is labeled with a character.
- The children of a node are in alphabetical order.
- Each node or branch represents a possible character of keys or words.
- Each node or branch may have multiple branches.
- The last node of every key or word is used to mark the end of word or node.

Construct standard trie for the following words {bear, bell, bid, bull, buy, sell, stock, stop}

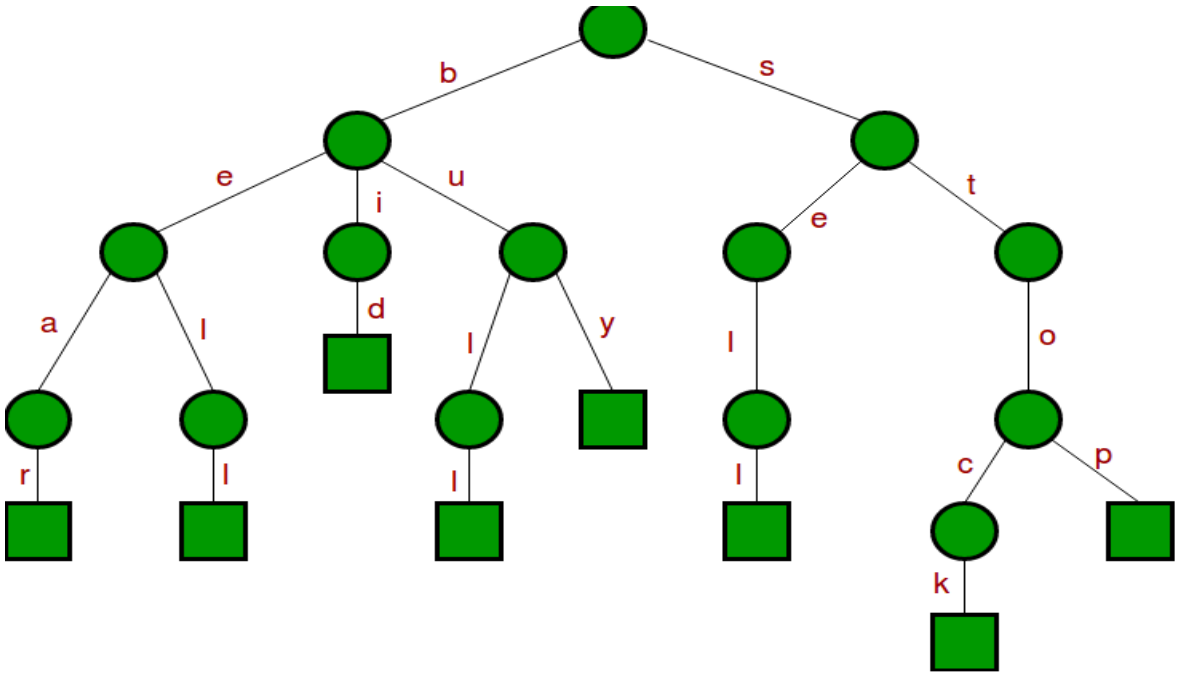


Properties of compressed tries

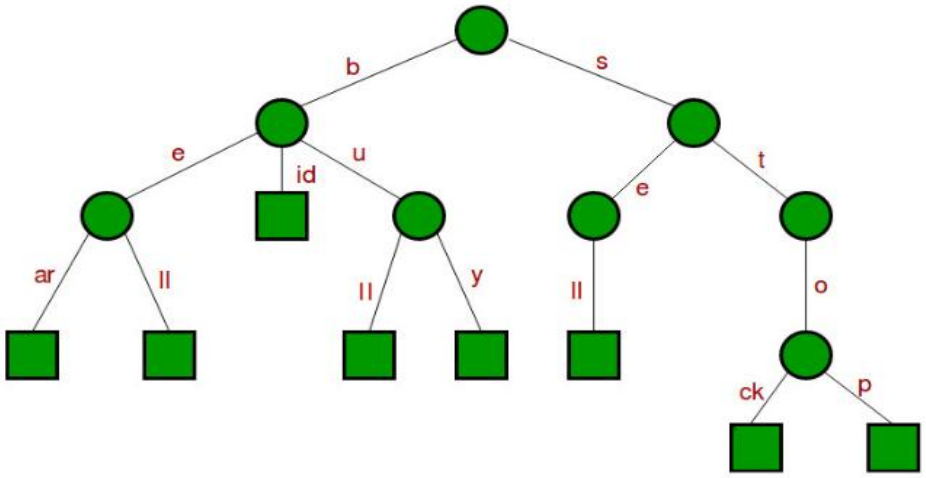
- A Compressed Trie is an advanced version of the standard trie.
- Each nodes(except the **leaf** nodes) have atleast 2 children.
- It is used to achieve space optimization.
- To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.

- It consists of grouping, re-grouping and un-grouping of keys of characters.
- While performing the insertion operation, it may be required to un-group the already grouped characters.
- While performing the deletion operation, it may be required to re-group the already grouped characters.
- A compressed trie T storing s strings(**keys**) has s external nodes and O(s) total number of nodes.

Standard trie



Compressed trie

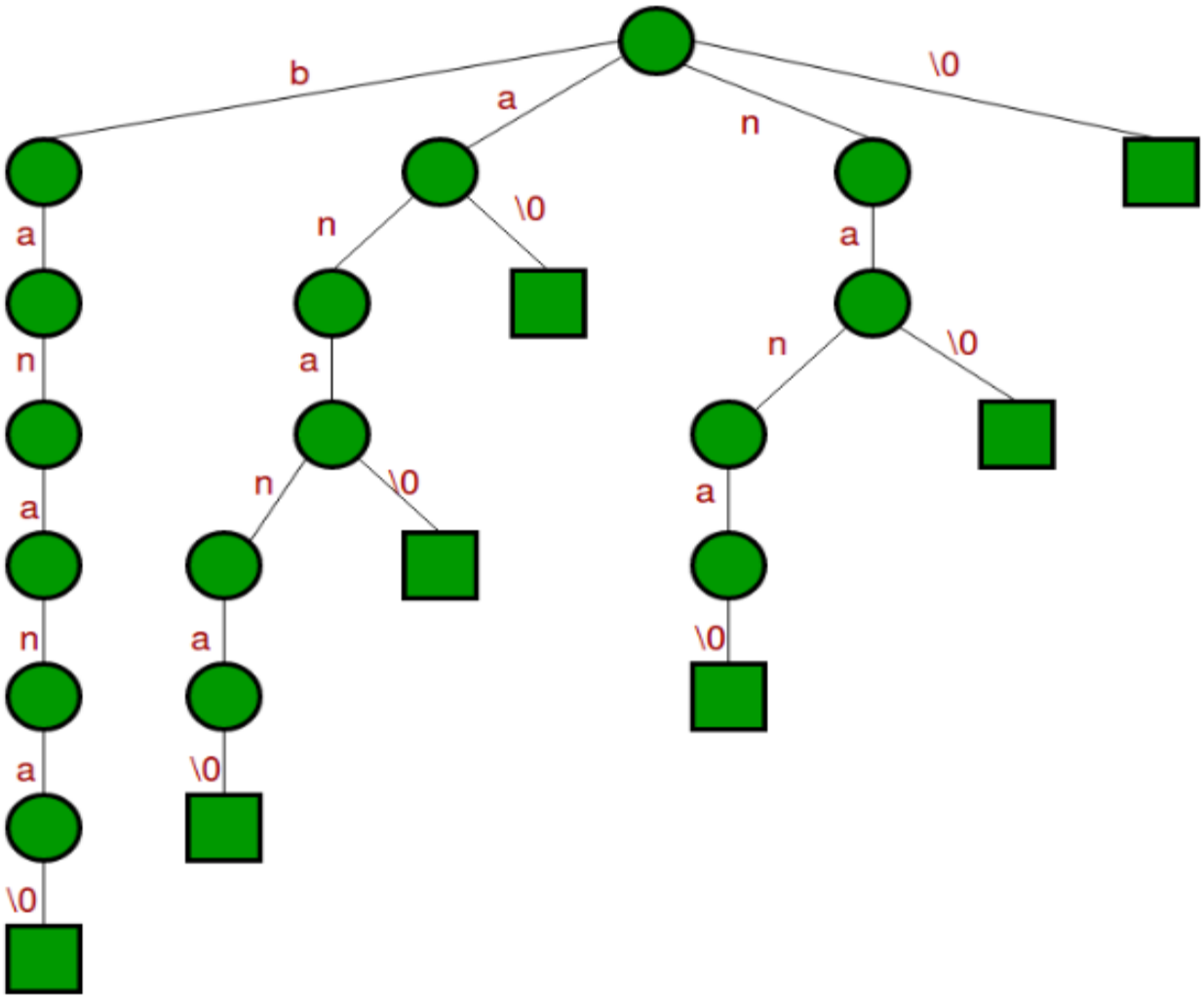


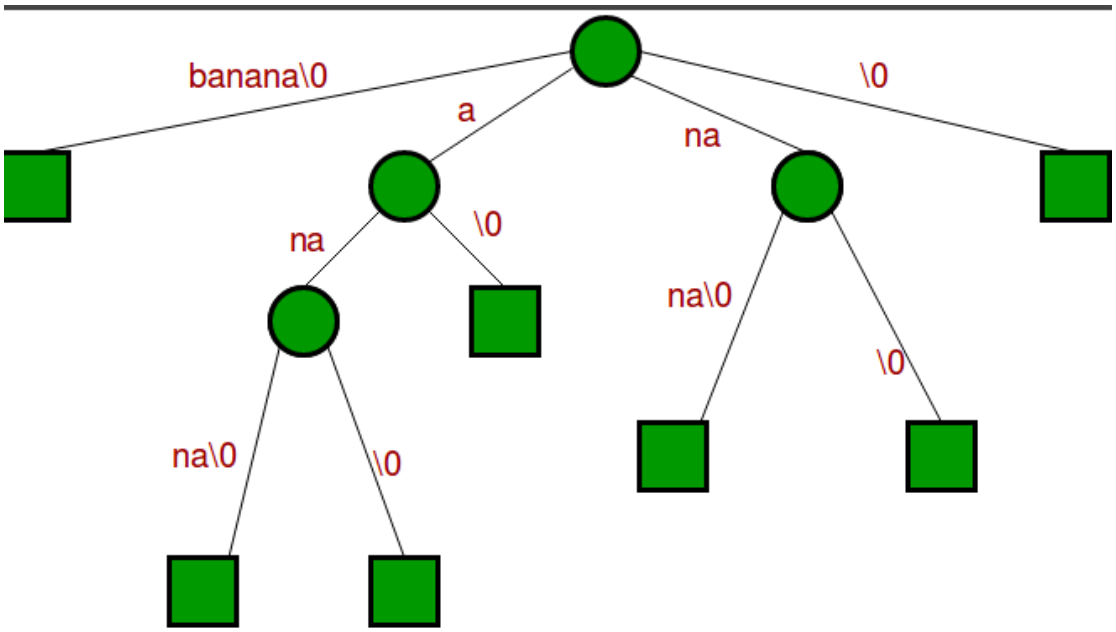
Suffix Trie A Suffix trie have the following properties:

- A Suffix Trie is an advanced version of the compressed trie.
- The most common application of suffix trie is Pattern Matching.
- While performing the insertion operation, both the word and its suffixes are stored.
- A suffix trie is also used in word matching and prefix matching.
- To generate a suffix trie, all the suffixes of given string are considered as individual words.
- Using the suffixes, compressed trie is built.

- Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text
- . 1) Generate all suffixes of given text. 2) Consider all suffixes as individual words and build a compressed trie
- . Let us consider an example text “banana\0” where ‘\0’ is string termination character.
- Following are all suffixes of “banana\0”

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```


b
a
n
\0
a
n
\0
a
n
\0
n
a
n
\0
a
n
\0
a
\0
a
\0
\0


banana\0
a
na
\0
na
\0
na\0
\0
na\0
\0