EX NO:1	Implementation of recursive function for tree traversal and Fibonacci series
DATE:	

1A: Tree Traversal using Recursive:

Aim

To write C++ program to implement tree traversal using recursive funtion

Algorithm

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree) Algorithm Preorder(tree) 1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

```
#include <iostream>
using namespace std;
/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
	int data;
	struct Node* left, *right;
	Node(int data)
	{
		this->data = data;
		left = right = NULL;
	}
};
/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
	if (node == NULL)
```

```
                return;

        // first recur on left subtree
        printPostorder(node->left);

        // then recur on right subtree
        printPostorder(node->right);

        // now deal with the node
        cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
        if (node == NULL)
                return;

        /* first recur on left child */
        printInorder(node->left);

        /* then print the data of node */
        cout << node->data << " ";

        /* now recur on right child */
        printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
        if (node == NULL)
                return;

        /* first print data of node */
        cout << node->data << " ";

        /* then recur on left subtree */
        printPreorder(node->left);

        /* now recur on right subtree */
        printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
        struct Node *root = new Node(1);
        root->left                      = new Node(2);
        root->right             = new Node(3);
```

```
	root->left->left	= new Node(4);
	root->left->right = new Node(5);

	cout << "\nPreorder traversal of binary tree is \n";
	printPreorder(root);

	cout << "\nInorder traversal of binary tree is \n";
	printInorder(root);

	cout << "\nPostorder traversal of binary tree is \n";
	printPostorder(root);

	return 0;
}
```

Output

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

Alternate

```
#include <iostream>
using namespace std;

// Data structure to store a binary tree node
struct Node
{
	int data;
	Node *left, *right;

	Node(int data)
	{
		this->data = data;
		this->left = this->right = nullptr;
	}
};

// Recursive function to perform preorder traversal on the tree
void preorder(Node* root)
{
	// if the current node is empty
	if (root == nullptr) {
		return;
	}
```

```
        // Display the data part of the root (or current node)
        cout << root->data << " ";

        // Traverse the left subtree
        preorder(root->left);

        // Traverse the right subtree
        preorder(root->right);
}


int main()
{
        /* Construct the following tree
                          1
                        /   \
                       /     \
                    2       3
                  /     /   \
                 /     /     \
                4     5       6
                          / \
                         /   \
                        7     8
        */

        Node* root = new Node(1);
        root->left = new Node(2);
        root->right = new Node(3);
        root->left->left = new Node(4);
        root->right->left = new Node(5);
        root->right->right = new Node(6);
        root->right->left->left = new Node(7);
        root->right->left->right = new Node(8);

        preorder(root);

        return 0;
}
```

Output

1 2 4 3 5 7 8 6

Result

The C++ program of implementation of tree traversal using recursive performed successfully.

EX NO:1B	**Implementation of Fibonacci series using recursive function**
DATE:	

Aim

To write C++ program to implement fibonacci using recursive function

Algorithm

1. START
2. Input the non-negative integer 'n'
3. If (n==o || n==1)
   return n;
   else
   return fib(n-1)+fib(n-2);
4. Print, n^{th} Fibonacci number
5. END

```
#include<iostream>
using namespace std;
void printFibonacci(int n){
   static int n1=0, n2=1, n3;
   if(n>0){
       n3 = n1 + n2;
       n1 = n2;
       n2 = n3;
 cout<<n3<<" ";
       printFibonacci(n-1);
   }
}
int main(){
   int n;
   cout<<"Enter the number of elements: ";
   cin>>n;
   cout<<"Fibonacci Series: ";
   cout<<"0 "<<"1 ";
   printFibonacci(n-2);  //n-2 because 2 numbers are already printed
    return 0;
}
```

Output

Enter the number of elements: 10
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34

Result

The C++ program of implementation of Fibonacci series using recursive performed successfully.

EX NO:2A	Implementation of iteration function for tree traversal and Fibonacci series
DATE:	

Aim

To Write C++ program to implement tree traversal using iteration function

Algorithm

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
a) Pop the top item from stack.
b) Print the popped item, set current = popped_item->right
c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

```
#include <iostream>
#include <stack>
using namespace std;
// Data structure to store a binary tree node
struct Node
{
	int data;
	Node *left, *right;
	Node(int data)
	{
		this->data = data;
		this->left = this->right = nullptr;
	}
};
// Iterative function to perform preorder traversal on the tree
void preorderIterative(Node* root)
{
	// return if the tree is empty
	if (root == nullptr)
	return;
// create an empty stack and push the root node
	stack<Node*> stack;
	stack.push(root);

	// loop till stack is empty
	while (!stack.empty())
	{
		// pop a node from the stack and print it
		Node* curr = stack.top();
```

```
                stack.pop();
                cout << curr->data << " ";
                // push the right child of the popped node into the stack
                if (curr->right) {
                        stack.push(curr->right);
                }

                // push the left child of the popped node into the stack
                if (curr->left) {
                        stack.push(curr->left);
                }

                // the right child must be pushed first so that the left child
                // is processed first (LIFO order)
        }
}
int main()
{
        /* Construct the following tree
                          1
                        /   \
                       /     \
                      2       3
                     /      /   \
                    /      /     \
                   4      5       6
                         / \
                        /   \
                       7     8
        */
        Node* root = new Node(1);
        root->left = new Node(2);
        root->right = new Node(3);
        root->left->left = new Node(4);
        root->right->left = new Node(5);
        root->right->right = new Node(6);
        root->right->left->left = new Node(7);
        root->right->left->right = new Node(8);
        preorderIterative(root);
        return 0;
}
```

Output
1 2 4 3 5 7 8 6

Result
The C++ program of implementation of tree traversal using Iteration function performed successfully.

EX NO:2B	Fibonacci series with Iteration
DATE:	

2B: Fibonacci with Iteration

Aim

To write C++ program to implement Fibonacci series using Iteration.

```
#include <iostream>
using namespace std;
int main() {
  int n1=0,n2=1,n3,i,number;
 cout<<"Enter the number of elements: ";
 cin>>number;
 cout<<n1<<" "<<n2<<" "; //printing 0 and 1
 for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed
 {
  n3=n1+n2;
  cout<<n3<<" ";
  n1=n2;
  n2=n3;
 }
   return 0;
   }
```

Output

Enter the number of elements: 9
0 1 1 2 3 5 8 13 21

Result

The C++ program of implementation of Fibonacci Series using Iteration Function performed successfully

EX NO:3	Implementation of Merge Sort and Quick Sort
DATE:	

3A. Merge Sort

Aim

To Write C++ program to implement Merge sort and Quick sort

Algorithm

- The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.
- After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
#include<iostream>
using namespace std;
void swapping(int &a, int &b) {      //swap the content of a and b
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void display(int *array, int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;
}
void merge(int *array, int l, int m, int r) {
  int i, j, k, nl, nr;
  //size of left and right sub-arrays
  nl = m-l+1; nr = r-m;
  int larr[nl], rarr[nr];
  //fill left and right sub-arrays
  for(i = 0; i<nl; i++)
    larr[i] = array[l+i];
  for(j = 0; j<nr; j++)
    rarr[j] = array[m+1+j];
  i = 0; j = 0; k = l;
  //marge temp arrays to real array
  while(i < nl && j<nr) {
    if(larr[i] <= rarr[j]) {
      array[k] = larr[i];
      i++;
```

```
    }else{
      array[k] = rarr[j];
      j++;
    }
    k++;
  }
  while(i<nl) {       //extra element in left array
    array[k] = larr[i];
    i++; k++;
  }
  while(j<nr) {     //extra element in right array
    array[k] = rarr[j];
    j++; k++;
  }
}
void mergeSort(int *array, int l, int r) {
  int m;
  if(l < r) {
    int m = l+(r-l)/2;
    // Sort first and second arrays
    mergeSort(array, l, m);
    mergeSort(array, m+1, r);
    merge(array, l, m, r);
  }
}
int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n];     //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }
  cout << "Array before Sorting: ";
  display(arr, n);
  mergeSort(arr, 0, n-1);     //(n-1) for last index
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Output

```
Enter the number of elements: 5
Enter elements:
9 4 1 7 8
Array before Sorting: 9 4 1 7 8
Array after Sorting: 1 4 7 8 9
```

3B. Quick Sort

Algorithm

1. An array is divided into subarrays by selecting a pivot element (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

```
#include <iostream>

using namespace std;

void quick_sort(int[],int,int);
int partition(int[],int,int);
int main()
{
   int a[50],n,i;
   cout<<"How many elements?";
   cin>>n;
   cout<<"\nEnter array elements:";

   for(i=0;i<n;i++)
      cin>>a[i];

   quick_sort(a,0,n-1);
   cout<<"\nArray after sorting:";

   for(i=0;i<n;i++)
      cout<<a[i]<<" ";

   return 0;
}
void quick_sort(int a[],int l,int u)
{
   int j;
   if(l<u)
   {
      j=partition(a,l,u);
      quick_sort(a,l,j-1);
      quick_sort(a,j+1,u);
   }
}
```

```
int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;

   do
   {
      do
         i++;

      while(a[i]<v&&i<=u);

      do
         j--;
      while(v<a[j]);

      if(i<j)
      {
         temp=a[i];
         a[i]=a[j];
         a[j]=temp;
      }
   }while(i<j);

   a[l]=a[j];
   a[j]=v;

   return(j);
}
```

Output
How many elements?7

Enter array elements:3 7 1 9 5 4 2

Array after sorting:1 2 3 4 5 7 9

Result
The C++ program of implementation of Merge Sort and Quick Sort performed successfully.

EX NO:4	Implementation of a Binary Search Tree
DATE:	

Aim

To Write C++ program to implement Binary Search Tree

Algorithm

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Search in BST

```
If root == NULL
   return NULL;
If number == root->data
   return root->data;
If number < root->data
   return search(root->left)
If number > root->data
   return search(root->right)
```

Insert

```
If node == NULL
   return createNode(data)
if (data < node->data)
   node->left  = insert(node->left, data);
else if (data > node->data)
   node->right = insert(node->right, data);
return node;
```

Remove

There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

Get the inorder successor of that node.

Replace the node with the inorder successor.

Remove the inorder successor from its original position.

```
#include<iostream>
using namespace std;
class BST {

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

   node* makeEmpty(node* t) {
      if(t == NULL)
         return NULL;
      {
         makeEmpty(t->left);
         makeEmpty(t->right);
         delete t;
      }
      return NULL;
   }
  node* insert(int x, node* t)
   {
      if(t == NULL)
      {
         t = new node;
         t->data = x;
         t->left = t->right = NULL;
      }
      else if(x < t->data)
         t->left = insert(x, t->left);
      else if(x > t->data)
         t->right = insert(x, t->right);
      return t;
   }
  node* findMin(node* t)
   {
      if(t == NULL)
         return NULL;
      else if(t->left == NULL)
         return t;
      else
         return findMin(t->left);
   }
 node* findMax(node* t) {
      if(t == NULL)
         return NULL;
      else if(t->right == NULL)
         return t;
```

```
        else
   return findMax(t->right);
    }
node* remove(int x, node* t) {
        node* temp;
        if(t == NULL)
            return NULL;
        else if(x < t->data)
            t->left = remove(x, t->left);
        else if(x > t->data)
            t->right = remove(x, t->right);
        else if(t->left && t->right)
        {
            temp = findMin(t->right);
            t->data = temp->data;
            t->right = remove(t->data, t->right);
        }
        else
        {
            temp = t;
            if(t->left == NULL)
                t = t->right;
            else if(t->right == NULL)
                t = t->left;
            delete temp;
        }
return t;
    }
void inorder(node* t) {
        if(t == NULL)
            return;
        inorder(t->left);
        cout << t->data << " ";
        inorder(t->right);
    }

    node* find(node* t, int x) {
        if(t == NULL)
            return NULL;
        else if(x < t->data)
            return find(t->left, x);
        else if(x > t->data)
            return find(t->right, x);
        else
            return t;
    }

public:
    BST() {
        root = NULL;
```

```
    }

    ~BST() {
       root = makeEmpty(root);
    }
    void insert(int x) {
       root = insert(x, root);
    }

    void remove(int x) {
       root = remove(x, root);
    }
    void display() {
       inorder(root);
       cout << endl;
    }
    void search(int x) {
       root = find(root, x);
    }
};
int main() {
    BST t;
    t.insert(20);
    t.insert(25);
    t.insert(15);
    t.insert(10);
    t.insert(30);
    t.display();
    t.remove(20);
    t.display();
    t.remove(25);
    t.display();
    t.remove(30);
    t.display();
    return 0;
}
```

Output

```
10 15 25 30 70
10 15 25 30 70
10 15 30 70
10 15 70
```

Result

The C++ program of implementation of Binary Search Tree performed successfully.

EX NO:5	Red-Black Tree Implementation
DATE:	

Aim

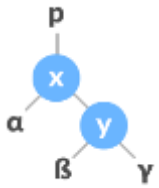
To Write C++ program to implement Red- Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.
A red-black tree satisfies the following properties:

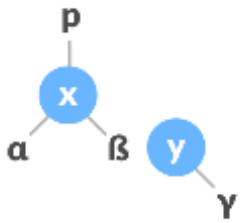
1. Red/Black Property: Every node is colored, either red or black.
2. Root Property: The root is black.
3. Leaf Property: Every leaf (NIL) is black.
4. Red Property: If a red node has children then, the children are always black.
5. Depth Property: For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Algorithm

Left Rotate
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.
Algorithm

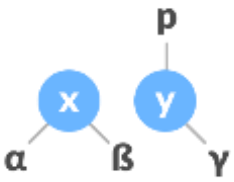


1. **Let the initial tree be:** **Initial tree**
2. **If y has a left subtree, assign x as the parent of the left subtree of y.**

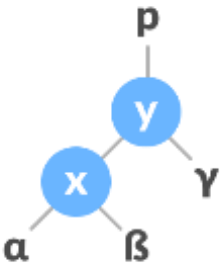


Assign x as the parent of the left subtree of y

3. **If the parent of x is NULL, make y as the root of the tree.**
4. **Else if x is the left child of p, make y as the left child of p.**



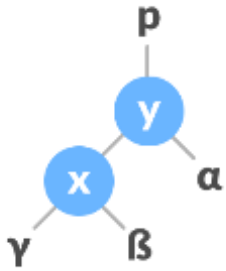
5. **Else assign y as the right child of p.** **Change the parent of x to that of y**



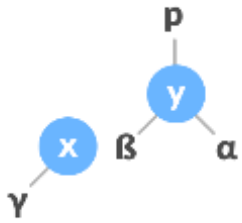
6. **Make y as the parent of x.** **Assign y as the parent of x.**

Right Rotate

In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

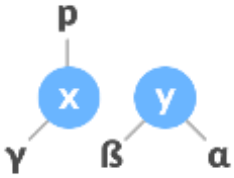


1. **Let the initial tree be:** **Initial Tree**
2. **If x has a right subtree, assign y as the parent of the right subtree of x.**

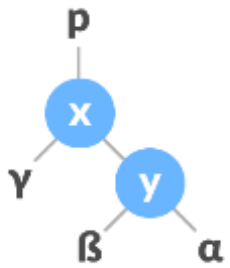


Assign y as the parent of the right subtree of x

3. **If the parent of y is NULL, make x as the root of the tree.**
4. **Else if y is the right child of its parent p, make x as the right child of p.**



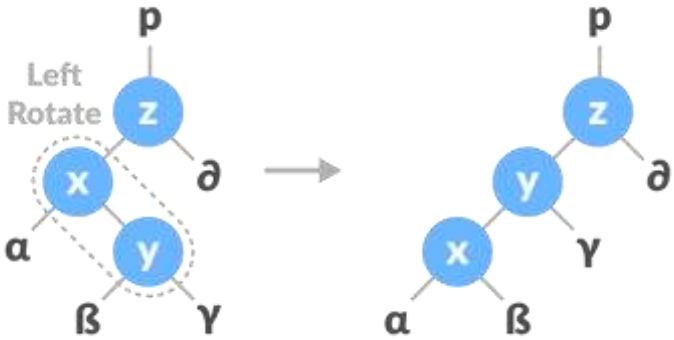
5. **Else assign x as the left child of p.** **Assign the parent of y as the parent of x**



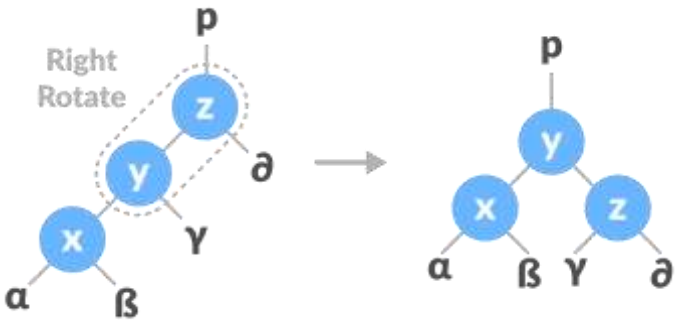
6. **Make x as the parent of y.** **Assign x as the parent of y**

Left-Right and Right-Left Rotate

In left-right rotation, the arrangements are first shifted to the left and then to the right.

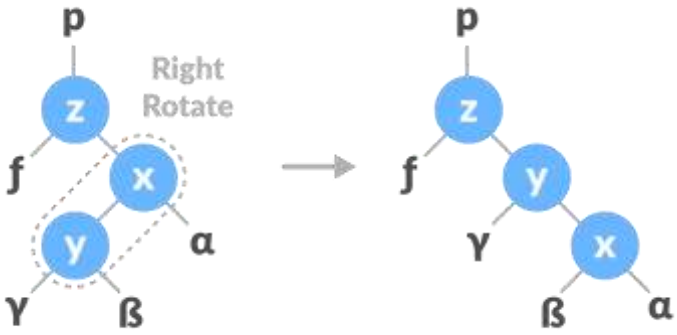


1. **Do left rotation on x-y.**
   Left rotate x-y

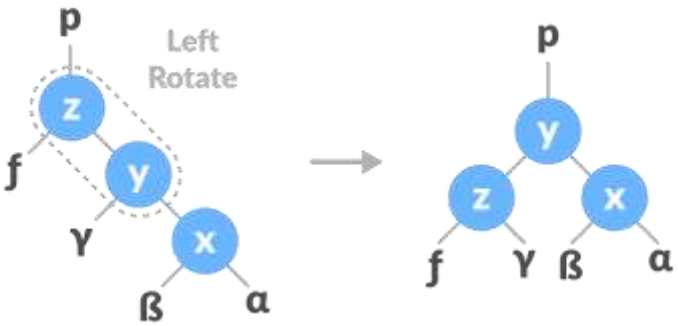


2. **Do right rotation on y-z.**
   Right rotate z-y

In right-left rotation, the arrangements are first shifted to the right and then to the left.



1. **Do right rotation on x-y.**
   Right rotate x-y



2. **Do left rotation on z-y.**
   Left rotate z-y

Algorithm to insert a node

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. NIL) and x be the root of the tree.
2. Check if the tree is empty (ie. whether x is NIL). If yes, insert newNode as a root node and color it black.
3. Else, repeat steps following steps until leaf (NIL) is reached.
   . Compare newKey with rootKey.
   a. If newKey is greater than rootKey, traverse through the right subtree.
   b. Else traverse through the left subtree.
4. Assign the parent of the leaf as a parent of newNode.
5. If leafKey is greater than newKey, make newNode as rightChild.
6. Else, make newNode as leftChild.
7. Assign NULL to the left and rightChild of newNode.
8. Assign RED color to newNode.
9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?

This is because inserting a red node does not violate the depth property of a red-black tree. If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Algorithm to maintain red-black property after insertion

This algorithm is used for maintaining the property of a red-black tree if the insertion of a newNode violates this property.

1. Do the following while the parent of newNode p is RED.
2. If p is the left child of grandParent gP of z, do the following.
   Case-I:
   . If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
   a. Assign gP to newNode.
   Case-II:
   b. Else if newNode is the right child of p then, assign p to newNode.
   c. Left-Rotate newNode.
   Case-III:
   d. Set color of p as BLACK and color of gP as RED.
   e. Right-Rotate gP.

3. Else, do the following.
    . If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
    a. Assign gP to newNode.
    b. Else if newNode is the left child of p then, assign p to newNode and Right-Rotate newNode.
    c. Set color of p as BLACK and color of gP as RED.
    d. Left-Rotate gP.
4. Set the root of the tree as BLACK.

Algorithm to delete a node

1. Save the color of nodeToBeDeleted in origrinalColor.
2. If the left child of nodeToBeDeleted is NULL
    . Assign the right child of nodeToBeDeleted to x.
    a. Transplant nodeToBeDeleted with x.
3. Else if the right child of nodeToBeDeleted is NULL
    . Assign the left child of nodeToBeDeleted into x.
    a. Transplant nodeToBeDeleted with x.
4. Else
    . Assign the minimum of right subtree of noteToBeDeleted into y.
    a. Save the color of y in originalColor.
    b. Assign the rightChild of y into x.
    c. If y is a child of nodeToBeDeleted, then set the parent of x as y.
    d. Else, transplant y with rightChild of y.
    e. Transplant nodeToBeDeleted with y.
    f. Set the color of y with originalColor.
5. If the originalColor is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node x (which is occupying y's original position) has an extra black. This makes node x neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.

However, the color attribute of x is not changed rather the extra black is represented in x's pointing to the node.

The extra black can be removed if

1. It reaches the root node.
2. If x points to a red-black node. In this case, x is colored black.
3. Suitable rotations and recoloring are performed.

The following algorithm retains the properties of a red-black tree.

1. Do the following until the x is not the root of the tree and the color of x is BLACK
2. If x is the left child of its parent then,
    . Assign w to the sibling of x.
    a. If the right child of parent of x is RED,
    Case-I:
        . Set the color of the right child of the parent of x as BLACK.
        a. Set the color of the parent of x as RED.
        b. Left-Rotate the parent of x.
        c. Assign the rightChild of the parent of x to w.

b. If the color of both the right and the leftChild of w is BLACK,
Case-II:
. Set the color of w as RED
a. Assign the parent of x to x.
c. Else if the color of the rightChild of w is BLACK
Case-III:
. Set the color of the leftChild of w as BLACK
a. Set the color of w as RED
b. Right-Rotate w.
c. Assign the rightChild of the parent of x to w.
d. If any of the above cases do not occur, then do the following.
Case-IV:
. Set the color of w as the color of the parent of x.
a. Set the color of the parent of x as BLACK.
b. Set the color of the right child of w as BLACK.
c. Left-Rotate the parent of x.
d. Set x as the root of the tree.

3. Else the same as above with right changed to left and vice versa.
4. Set the color of x as BLACK.

```
#include <cstdlib>
#include <stdexcept>
#include <iostream>
using namespace std;

template<typename Key, typename Value>
class RedBlack
{
public:
  RedBlack()
    : root(NULL)
  {
  }

  ~RedBlack()
  {
    DeleteNode(root);
  }

  void Insert(const Key& key, const Value& value)
  {
    Node *node, *parent, *z;

    parent = NULL;
    node = root;
    while (node)
    {
      parent = node;
```

```
  if (key < node->key)
  {
    node = node->left;
  }
  else
  {
    node = node->right;
  }
}

if (!parent)
{
  z = root = new Node;
  z->key = key;
  z->value = value;
  z->colour = BLACK;
  z->parent = z->left = z->right = NULL;
}
else
{
  z = new Node;
  z->key = key;
  z->value = value;
  z->colour = RED;
  z->parent = parent;
  z->left = z->right = NULL;

  if (z->key < parent->key)
  {
    parent->left = z;
  }
  else
  {
    parent->right = z;
  }
}

Node *uncle;
bool side;
while (z->parent && z->parent->colour == RED)
{
  if ((side = (z->parent == z->parent->parent->left)))
  {
    uncle = z->parent->parent->right;
  }
  else
  {
    uncle = z->parent->parent->left;
  }
```

```
    if (uncle && uncle->colour == RED)
    {
      z->parent->colour = BLACK;
      uncle->colour = BLACK;
      z->parent->parent->colour = RED;
      z = z->parent->parent;
    }
    else
    {
      if (z == (side ? z->parent->right : z->parent->left))
      {
        z = z->parent;
        side ? RotateLeft(z) : RotateRight(z);
      }

      z->parent->colour = BLACK;
      z->parent->parent->colour = RED;
      side ? RotateRight(z->parent->parent) : RotateLeft(z->parent->parent);
    }
  }

  root->colour = BLACK;
}

Value& Find(const Key& key)
{
  Node *node = root;
  while (node)
  {
    if (node->key < key)
    {
      node = node->left;
    }
    else if (node->key > key)
    {
      node = node->right;
    }
    else
    {
      return node->value;
    }
  }

  throw std::runtime_error("Key not found");
}

void Delete(const Key& key)
{
  Node *node = root;
  while (node)
```

```
{
  if (node->key > key)
  {
    node = node->left;
  }
  else if (node->key < key)
  {
    node = node->right;
  }
  else
  {
    break;
  }
}

if (!node || node->key != key)
{
  return;
}

Colour original;
Node *sub, *old;
if (!node->left)
{
  Transplant(node, sub = node->right);
}
else if (!node->right)
{
  Transplant(node, sub = node->left);
}
else
{
  old = Minimum(node->right);
  original = old->colour;
  sub = old->right;

  if (old->parent == node)
  {
    sub->parent = node;
  }
  else
  {
    Transplant(old, old->right);
    old->right = node->right;
    old->right->parent = old;
  }

  Transplant(node, old);
  old->left = node->left;
  old->left->parent = old;
```

```
  old->colour = node->colour;
}

delete node;
if (original == BLACK)
{
  bool side;
  Node *sibling;
  while (old != root && old->colour == BLACK)
  {
    if ((side = (old == old->parent->left)))
    {
      sibling = old->parent->right;
    }
    else
    {
      sibling = old->parent->left;
    }

    if (sibling->colour == RED)
    {
      sibling->colour = BLACK;
      old->parent->colour = RED;
      side ? RotateLeft(old->parent) : RotateRight(old->parent);
      sibling = side ? old->parent->right : old->parent->left;
    }

    if (sibling->left->colour == BLACK && sibling->right->colour == RED)
    {
      sibling->colour = RED;
      old = old->parent;
    }
    else
    {
      if (BLACK == side ? sibling->right->colour : sibling->left->colour)
      {
        sibling->colour = RED;
        if (side)
        {
          sibling->left->colour = BLACK;
          RotateRight(sibling);
          sibling = old->parent->right;
        }
        else
        {
          sibling->right->colour = BLACK;
          RotateLeft(sibling);
          sibling = old->parent->left;
        }
      }
```

```
        sibling->colour = old->parent->colour;
        old->parent->colour = BLACK;
        if (side)
        {
          sibling->left->colour = BLACK;
          RotateLeft(old->parent);
        }
        else
        {
          sibling->right->colour = BLACK;
          RotateRight(old->parent);
        }

        old = root;
      }
    }
  }
 }

 void Dump()
 {
  Dump(root, 0);
 }

private:
 enum Colour
 {
  RED,
  BLACK
 };

 struct Node
 {
  Colour colour;
  Key key;
  Value value;
  Node *parent;
  Node *left;
  Node *right;
 };

 Node *root;

 void RotateLeft(Node *x)
 {
  Node *y;

  y = x->right;
  x->right = y->left;
```

```
  if (y->left)
  {
    y->left->parent = x;
  }

  y->parent = x->parent;
  y->left = x;

  if (!x->parent)
  {
    root = y;
  }
  else if (x == x->parent->left)
  {
    x->parent->left = y;
  }
  else
  {
    x->parent->right = y;
  }
  x->parent = y;
}

void RotateRight(Node *y)
{
  Node *x;

  x = y->left;
  y->left = x->right;
  if (x->right)
  {
    x->right->parent = y;
  }

  x->parent = y->parent;
  x->right = y;

  if (!y->parent)
  {
    root = x;
  }
  else if (y == y->parent->left)
  {
    y->parent->left = x;
  }
  else
  {
    y->parent->right = x;
  }
```

```
  y->parent = x;
}

void Transplant(Node *dest, Node *src)
{
  if (dest->parent == NULL)
  {
    root = src;
  }
  else if (dest == dest->parent->left)
  {
    dest->parent->left = src;
  }
  else
  {
    dest->parent->right = src;
  }

  if (src)
  {
    src->parent = dest->parent;
  }
}

Node *Minimum(Node *tree)
{
  while (tree->left)
  {
    tree = tree->left;
  }

  return tree;
}

void Dump(Node *node, int tabs)
{
  if (!node)
  {
    return;
  }

  Dump(node->left, tabs + 1);

  for (int i = 0; i < tabs; ++i)
  {
    std::cout << "\t\t";
  }
  std::cout << node->key << (node->colour ? "B" : "R") << std::endl;

  Dump(node->right, tabs + 1);
```

```
  }

  void DeleteNode(Node *node)
  {
    if (!node)
    {
      return;
    }

    if (node->left)
    {
      DeleteNode(node->left);
    }

    if (node->right)
    {
      DeleteNode(node->right);
    }

    delete node;
  }
};

int main()
{
  RedBlack<int, int> tree;
  for (int i = 1; i < 10; ++i)
  {
    tree.Insert(i, i);
  }
  tree.Delete(9);
  tree.Delete(8);
  tree.Dump();
  return 0;
}
```

Output

```
                    1B
          2R
                    3B
4B
                    5B
          6R
                    7R
```

Result

The C++ program of implementation of Red- Black tree performed successfully.

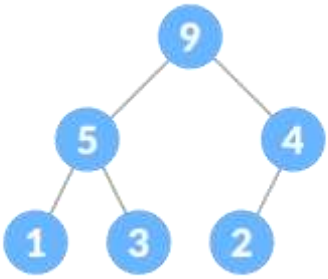
EX NO: 6	Heap Implementation
DATE:	

Aim

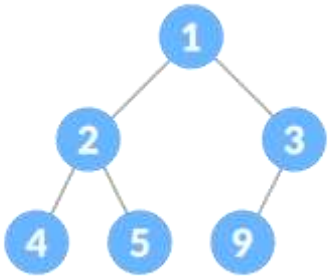
To Write C++ program to implement Heap

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

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



Max-heap



Min-heap

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

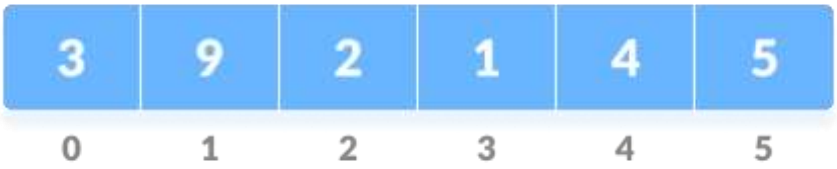
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

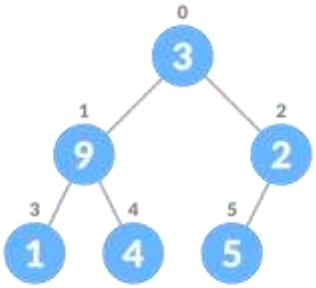
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

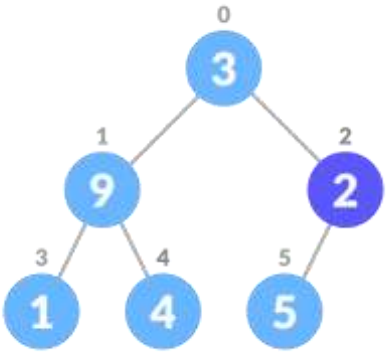
1. Let the input array be



Initial Array

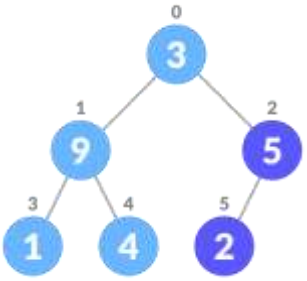


2. Create a complete binary tree from the array Complete binary tree
3. Start from the first index of non-leaf node whose index is given by n/2 - 1.



Start from the first on leaf node

4. Set current element i as largest.
5. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.
   If leftChild is greater than currentElement (i.e. element at ith index),
   set leftChildIndex as largest.
   If rightChild is greater than element in largest, set rightChildIndex as largest.



6. Swap largest with currentElement Swap if necessary
7. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest
```

swap array[i] and array[largest]

To create a Max-Heap:

MaxHeap(array, size)
  loop from the first index of non-leaf node down to zero
    call heapify

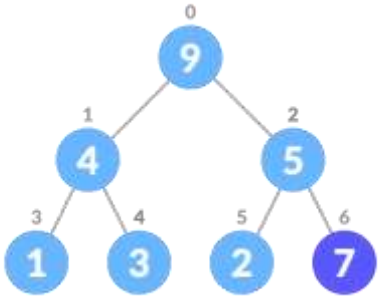
For Min-Heap, both leftChild and rightChild must be larger than the parent for all nodes**.**

Insert Element into Heap

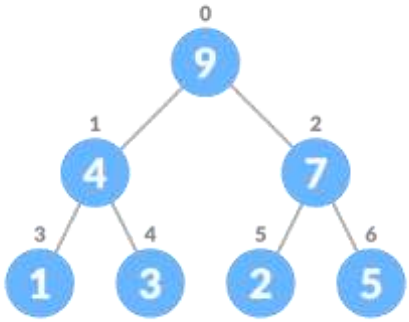
Algorithm for insertion in Max Heap

If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array



1. **Insert the new element at the end of the tree.**
   Insert at the end
2. **Heapify the tree.**



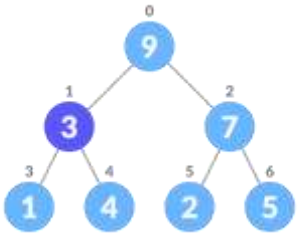
Heapify the array

For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

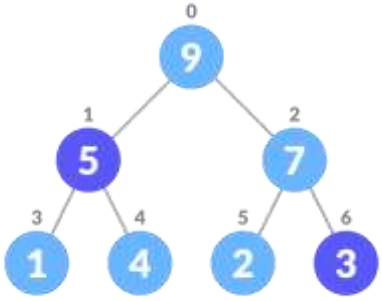
Delete Element from Heap

Algorithm for deletion in Max Heap
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

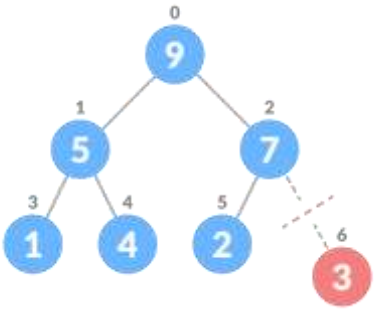
heapify the array



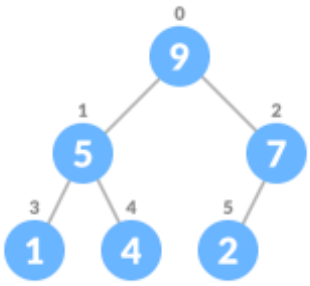
1. **Select the element to be deleted.**
   Select the element to be deleted



2. **Swap it with the last element.** **Swap with the last element**



3. **Remove the last element.** **Remove the last element**
4. **Heapify the tree.**



Heapify the array

For Min Heap, above algorithm is modified so that both childNodes are greater smaller than currentNode.

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

return rootNode

Extract-Max/Min
Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap

```
/* C++ Program to Implement Heap
 */
#include <iostream>
#include <cstdlib>
#include <vector>
#include <iterator>
using namespace std;
/*
 * Class Declaration
 */
class Heap
{
   private:
      vector <int> heap;
      int left(int parent);
      int right(int parent);
      int parent(int child);
      void heapifyup(int index);
      void heapifydown(int index);
   public:
      Heap()
      {}
      void Insert(int element);
      void DeleteMin();
      int ExtractMin();
      void DisplayHeap();
      int Size();
};
/*
 * Return Heap Size
 */
int Heap::Size()
{
   return heap.size();
}

/*
 * Insert Element into a Heap
 */
void Heap::Insert(int element)
{
```

```
    heap.push_back(element);
    heapifyup(heap.size() -1);
}
/*
 * Delete Minimum Element
 */
void Heap::DeleteMin()
{
    if (heap.size() == 0)
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    heap[0] = heap.at(heap.size() - 1);
    heap.pop_back();
    heapifydown(0);
    cout<<"Element Deleted"<<endl;
}

/*
 * Extract Minimum Element
 */
int Heap::ExtractMin()
{
    if (heap.size() == 0)
    {
        return -1;
    }
    else
        return heap.front();
}

/*
 * Display Heap
 */
void Heap::DisplayHeap()
{
    vector <int>::iterator pos = heap.begin();
    cout<<"Heap -->  ";
    while (pos != heap.end())
    {
        cout<<*pos<<" ";
        pos++;
    }
    cout<<endl;
}

/*
 * Return Left Child
 */
```

```
int Heap::left(int parent)
{
   int l = 2 * parent + 1;
   if(l < heap.size())
      return l;
   else
      return -1;
}

/*
 * Return Right Child
 */
int Heap::right(int parent)
{
   int r = 2 * parent + 2;
   if(r < heap.size())
      return r;
   else
      return -1;
}

/*
 * Return Parent
 */
int Heap::parent(int child)
{
   int p = (child - 1)/2;
   if(child == 0)
      return -1;
   else
      return p;
}

/*
 * Heapify- Maintain Heap Structure bottom up
 */
void Heap::heapifyup(int in)
{
   if (in >= 0 && parent(in) >= 0 && heap[parent(in)] > heap[in])
   {
      int temp = heap[in];
      heap[in] = heap[parent(in)];
      heap[parent(in)] = temp;
      heapifyup(parent(in));
   }
}

/*
 * Heapify- Maintain Heap Structure top down
 */
```

```
void Heap::heapifydown(int in)
{

   int child = left(in);
   int child1 = right(in);
   if (child >= 0 && child1 >= 0 && heap[child] > heap[child1])
   {
     child = child1;
   }
   if (child > 0)
   {
      int temp = heap[in];
      heap[in] = heap[child];
      heap[child] = temp;
      heapifydown(child);
   }
}

/*
 * Main Contains Menu
 */
int main()
{
   Heap h;
   while (1)
   {
      cout<<"------------------"<<endl;
      cout<<"Operations on Heap"<<endl;
      cout<<"------------------"<<endl;
      cout<<"1.Insert Element"<<endl;
      cout<<"2.Delete Minimum Element"<<endl;
      cout<<"3.Extract Minimum Element"<<endl;
      cout<<"4.Print Heap"<<endl;
      cout<<"5.Exit"<<endl;
      int choice, element;
      cout<<"Enter your choice: ";
      cin>>choice;
      switch(choice)
      {
      case 1:
         cout<<"Enter the element to be inserted: ";
         cin>>element;
         h.Insert(element);
         break;
      case 2:
         h.DeleteMin();
         break;
      case 3:
         cout<<"Minimum Element: ";
         if (h.ExtractMin() == -
```

```
 {
            cout<<"Heap is Empty"<<endl;
          }
          else
            cout<<"Minimum Element:  "<<h.ExtractMin()<<endl;
          break;
       case 4:
          cout<<"Displaying elements of Hwap:  ";
          h.DisplayHeap();
          break;
       case 5:
          exit(1);
       default:
          cout<<"Enter Correct Choice"<<endl;
       }
    }
   return 0;
}
```

Output

```
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 1
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 2
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 3
```

```
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 4
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 5
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 9
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 4
Displaying elements of Hwap:  Heap -->  1 2 3 4 5 9
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 1
Enter the element to be inserted: 7
```

```
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 4
Displaying elements of Hwap:  Heap -->  1 2 3 4 5 9 7
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 2
Element Deleted
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 4
Displaying elements of Hwap:  Heap -->  2 4 3 7 5 9
------------------
Operations on Heap
------------------
1.Insert Element
2.Delete Minimum Element
3.Extract Minimum Element
4.Print Heap
5.Exit
Enter your choice: 5
```

Result

The C++ program of implementation of Heap performed successfully.

EX NO: 7	Fibonacci Heap Implementation
DATE:	

Aim

To Write C++ program to implement Fibonacci Heap

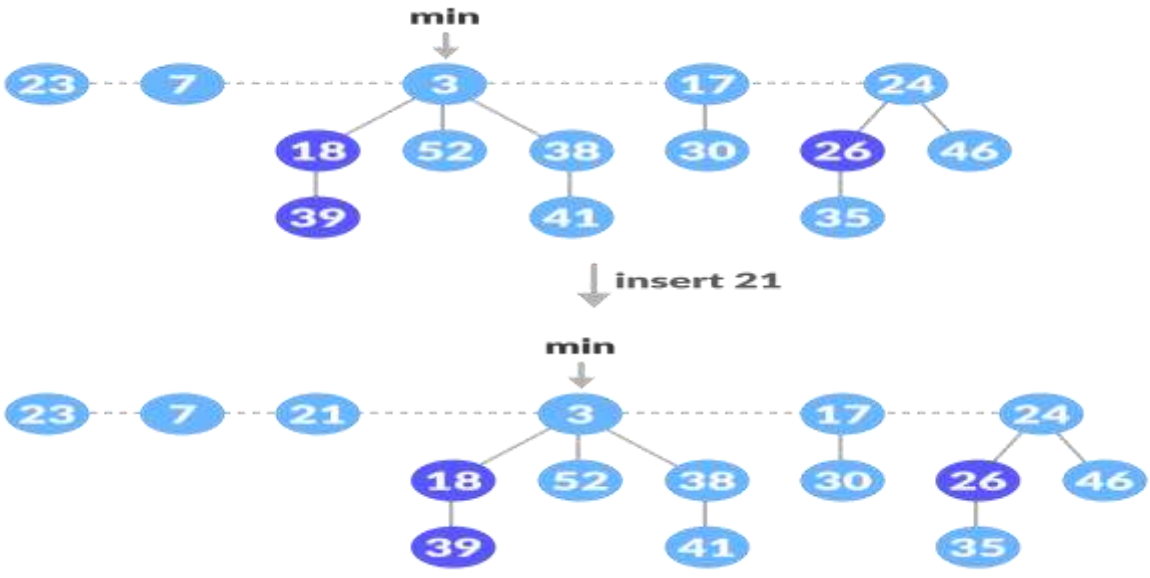
Properties of a Fibonacci Heap

Important properties of a Fibonacci heap are:

1. It is a set of min heap-ordered trees. (i.e. The parent is always smaller than the children.)
2. A pointer is maintained at the minimum element node.
3. It consists of a set of marked nodes. (Decrease key operation)
4. The trees within a Fibonacci heap are unordered but rooted.

Inserting a node into an already existing heap follows the steps below.

- Create a new node for the element.
- Check if the heap is empty.
- If the heap is empty, set the new node as a root node and mark it min.
- Else, insert the node into the root list and update min.
- Insertion operation in fibonacci heap
- Insertion Example




```
insert(H, x)
   degree[x] = 0
   p[x] = NIL
   child[x] = NIL
   left[x] = x
```

```
    right[x] = x
    mark[x] = FALSE
    concatenate the root list containing x with root list H
    if min[H] == NIL or key[x] < key[min[H]]
       then min[H] = x
    n[H] = n[H] + 1
```

Following functions are used for decreasing the key.

Decrease-Key

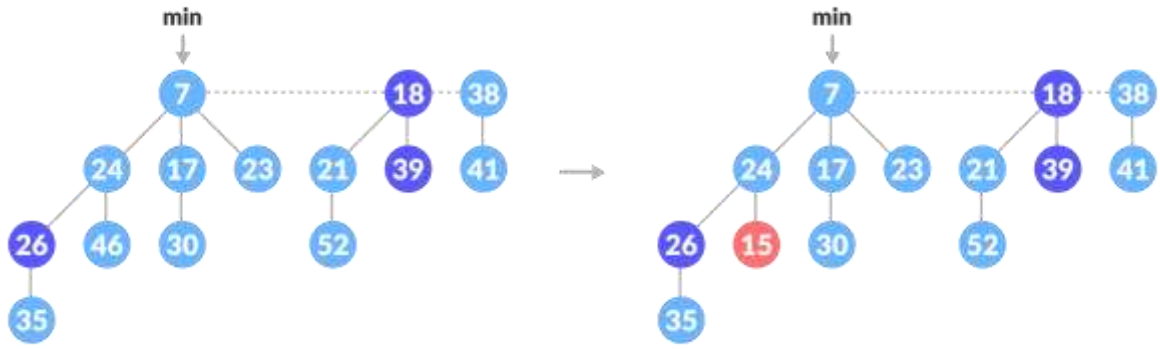
1. Select the node to be decreased, x, and change its value to the new value k.
2. If the parent of x, y, is not null and the key of parent is greater than that of the k then call Cut(x) and Cascading-Cut(y) subsequently.
3. If the key of x is smaller than the key of min, then mark x as min.

Cut

1. Remove x from the current position and add it to the root list.
2. If x is marked, then mark it as false.

Cascading-Cut

1. If the parent of y is not null then follow the following steps.
2. If y is unmarked, then mark y.
3. Else, call Cut(y) and Cascading-Cut(parent of y).




```
/*
 * C++ Program to Implement Fibonacci Heap
 */
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
    int n;
    int degree;
    node* parent;
```

```
    node* child;
    node* left;
    node* right;
    char mark;
    char C;
};
/*
 * Class Declaration
 */
class FibonacciHeap
{
    private:
        int nH;
        node *H;
    public:
        node* InitializeHeap();
        int Fibonnaci_link(node*, node*, node*);
        node *Create_node(int);
        node *Insert(node *, node *);
        node *Union(node *, node *);
        node *Extract_Min(node *);
        int Consolidate(node *);
        int Display(node *);
        node *Find(node *, int);
        int Decrease_key(node *, int, int);
        int Delete_key(node *,int);
        int Cut(node *, node *, node *);
        int Cascase_cut(node *, node *);
        FibonacciHeap()
        {
            H = InitializeHeap();
        }
};
/*
 * Initialize Heap
 */
node* FibonacciHeap::InitializeHeap()
{
    node* np;
    np = NULL;
    return np;
}
/*
 * Create Node
 */
node* FibonacciHeap::Create_node(int value)
{
    node* x = new node;
    x->n = value;
    return x;
```

```
}
/*
 * Insert Node
 */
node* FibonacciHeap::Insert(node* H, node* x)
{
   x->degree = 0;
   x->parent = NULL;
   x->child = NULL;
   x->left = x;
   x->right = x;
   x->mark = 'F';
   x->C = 'N';
   if (H != NULL)
   {
      (H->left)->right = x;
      x->right = H;
      x->left = H->left;
      H->left = x;
      if (x->n < H->n)
         H = x;
   }
   else
   {
      H = x;
   }
   nH = nH + 1;
   return H;
}
/*
 * Link Nodes in Fibonnaci Heap
 */
int FibonacciHeap::Fibonnaci_link(node* H1, node* y, node* z)
{
   (y->left)->right = y->right;
   (y->right)->left = y->left;
   if (z->right == z)
      H1 = z;
   y->left = y;
   y->right = y;
   y->parent = z;
   if (z->child == NULL)
      z->child = y;
   y->right = z->child;
   y->left = (z->child)->left;
   ((z->child)->left)->right = y;
   (z->child)->left = y;
   if (y->n < (z->child)->n)
      z->child = y;
   z->degree++;
```

```
}
/*
 * Union Nodes in Fibonnaci Heap
 */
node* FibonacciHeap::Union(node* H1, node* H2)
{
   node* np;
   node* H = InitializeHeap();
   H = H1;
   (H->left)->right = H2;
   (H2->left)->right = H;
   np = H->left;
   H->left = H2->left;
   H2->left = np;
   return H;
}
/*
 * Display Fibonnaci Heap
 */
int FibonacciHeap::Display(node* H)
{
   node* p = H;
   if (p == NULL)
   {
      cout<<"The Heap is Empty"<<endl;
      return 0;
   }
   cout<<"The root nodes of Heap are: "<<endl;
   do
   {
      cout<<p->n;
      p = p->right;
      if (p != H)
      {
         cout<<"-->";
      }
   }
   while (p != H && p->right != NULL);
   cout<<endl;
}
/*
 * Extract Min Node in Fibonnaci Heap
 */
node* FibonacciHeap::Extract_Min(node* H1)
{
   node* p;
   node* ptr;
   node* z = H1;
   p = z;
   ptr = z;
```

```
    if (z == NULL)
        return z;
    node* x;
    node* np;
    x = NULL;
    if (z->child != NULL)
        x = z->child;
    if (x != NULL)
    {
        ptr = x;
        do
        {
            np = x->right;
            (H1->left)->right = x;
            x->right = H1;
            x->left = H1->left;
            H1->left = x;
            if (x->n < H1->n)
                H1 = x;
            x->parent = NULL;
            x = np;
        }
        while (np != ptr);
    }
    (z->left)->right = z->right;
    (z->right)->left = z->left;
    H1 = z->right;
    if (z == z->right && z->child == NULL)
        H = NULL;
    else
    {
        H1 = z->right;
        Consolidate(H1);
    }
    nH = nH - 1;
    return p;
}
/*
 * Consolidate Node in Fibonnaci Heap
 */
int FibonacciHeap::Consolidate(node* H1)
{
    int d, i;
    float f = (log(nH)) / (log(2));
    int D = f;
    node* A[D];
    for (i = 0; i <= D; i++)
        A[i] = NULL;
    node* x = H1;
    node* y;
```

```
node* np;
node* pt = x;
do
{
   pt = pt->right;
   d = x->degree;
   while (A[d] != NULL)
   {
      y = A[d];
      if (x->n > y->n)
      {
         np = x;
         x = y;
         y = np;
      }
      if (y == H1)
         H1 = x;
      Fibonnaci_link(H1, y, x);
      if (x->right == x)
         H1 = x;
         A[d] = NULL;
      d = d + 1;
   }
   A[d] = x;
   x = x->right;
}
while (x != H1);
H = NULL;
for (int j = 0; j <= D; j++)
{
   if (A[j] != NULL)
   {
      A[j]->left = A[j];
      A[j]->right =A[j];
      if (H != NULL)
      {
         (H->left)->right = A[j];
         A[j]->right = H;
         A[j]->left = H->left;
         H->left = A[j];
         if (A[j]->n < H->n)
         H = A[j];
      }
      else
      {
         H = A[j];
      }
      if(H == NULL)
         H = A[j];
      else if (A[j]->n < H->n)
```

```
            H = A[j];
        }
    }
}

/*
 * Decrease key of Nodes in Fibonnaci Heap
 */
int FibonacciHeap::Decrease_key(node*H1, int x, int k)
{
    node* y;
    if (H1 == NULL)
    {
        cout<<"The Heap is Empty"<<endl;
        return 0;
    }
    node* ptr = Find(H1, x);
    if (ptr == NULL)
    {
        cout<<"Node not found in the Heap"<<endl;
        return 1;
    }
    if (ptr->n < k)
    {
        cout<<"Entered key greater than current key"<<endl;
        return 0;
    }
    ptr->n = k;
    y = ptr->parent;
    if (y != NULL && ptr->n < y->n)
    {
        Cut(H1, ptr, y);
        Cascase_cut(H1, y);
    }
    if (ptr->n < H->n)
        H = ptr;
    return 0;
}
/*
 * Cut Nodes in Fibonnaci Heap
 */
int FibonacciHeap::Cut(node* H1, node* x, node* y)
{
    if (x == x->right)
        y->child = NULL;
    (x->left)->right = x->right;
    (x->right)->left = x->left;
    if (x == y->child)
        y->child = x->right;
    y->degree = y->degree - 1;
```

```
    x->right = x;
    x->left = x;
    (H1->left)->right = x;
    x->right = H1;
    x->left = H1->left;
    H1->left = x;
    x->parent = NULL;
    x->mark = 'F';
}

/*
 * Cascade Cutting in Fibonnaci Heap
 */
int FibonacciHeap::Cascase_cut(node* H1, node* y)
{
    node* z = y->parent;
    if (z != NULL)
    {
        if (y->mark == 'F')
        {
            y->mark = 'T';
            }
        else
        {
            Cut(H1, y, z);
            Cascase_cut(H1, z);
        }
    }
}

/*
 * Find Nodes in Fibonnaci Heap
 */
node* FibonacciHeap::Find(node* H, int k)
{
    node* x = H;
    x->C = 'Y';
    node* p = NULL;
    if (x->n == k)
    {
        p = x;
        x->C = 'N';
        return p;
    }
    if (p == NULL)
    {
        if (x->child != NULL )
            p = Find(x->child, k);
        if ((x->right)->C != 'Y' )
            p = Find(x->right, k);
```

```
    }
    x->C = 'N';
    return p;
}
/*
 * Delete Nodes in Fibonnaci Heap
 */
int FibonacciHeap::Delete_key(node* H1, int k)
{
    node* np = NULL;
    int t;
    t = Decrease_key(H1, k, -5000);
    if (!t)
        np = Extract_Min(H);
    if (np != NULL)
        cout<<"Key Deleted"<<endl;
    else
        cout<<"Key not Deleted"<<endl;
    return 0;
}
/*
 * Main Contains Menu
 */
int main()
{
    int n, m, l;
    FibonacciHeap fh;
    node* p;
    node* H;
    H = fh.InitializeHeap();
    while (1)
    {
        cout<<"----------------------------"<<endl;
        cout<<"Operations on Binomial heap"<<endl;
        cout<<"----------------------------"<<endl;
        cout<<"1)Insert Element in the heap"<<endl;
        cout<<"2)Extract Minimum key node"<<endl;
        cout<<"3)Decrease key of a node"<<endl;
        cout<<"4)Delete a node"<<endl;
        cout<<"5)Display Heap"<<endl;
        cout<<"6)Exit"<<endl;
        cout<<"Enter Your Choice: ";
        cin>>l;
        switch(l)
        {
        case 1:
            cout<<"Enter the element to be inserted: ";
            cin>>m;
            p = fh.Create_node(m);
            H = fh.Insert(H, p);
```

```
            break;
        case 2:
            p = fh.Extract_Min(H);
            if (p != NULL)
                cout<<"The node with minimum key: "<<p->n<<endl;
            else
                cout<<"Heap is empty"<<endl;
            break;
        case 3:
            cout<<"Enter the key to be decreased: ";
            cin>>m;
            cout<<"Enter new key value: ";
            cin>>l;
            fh.Decrease_key(H, m, l);
            break;
        case 4:
            cout<<"Enter the key to be deleted: ";
            cin>>m;
            fh.Delete_key(H, m);
            break;
        case 5:
            cout<<"The Heap is: "<<endl;
            fh.Display(H);
            break;
        case 6:
            exit(1);
        default:
            cout<<"Wrong Choice"<<endl;
        }
    }
    return 0;
}
```

Output

```
Operations on Binomial heap
---------------------------
1)Insert Element in the heap
2)Extract Minimum key node
3)Decrease key of a node
4)Delete a node
5)Display Heap
6)Exit
Enter Your Choice: 1
Enter the element to be inserted: 10
---------------------------
Operations on Binomial heap
---------------------------
1)Insert Element in the heap
2)Extract Minimum key node
```

```
3)Decrease key of a node
4)Delete a node
5)Display Heap
6)Exit
Enter Your Choice: 1
Enter the element to be inserted: 15
----------------------------
Operations on Binomial heap
----------------------------
1)Insert Element in the heap
2)Extract Minimum key node
3)Decrease key of a node
4)Delete a node
5)Display Heap
6)Exit
Enter Your Choice: 1
Enter the element to be inserted: 20
----------------------------
Operations on Binomial heap
----------------------------
1)Insert Element in the heap
2)Extract Minimum key node
3)Decrease key of a node
4)Delete a node
5)Display Heap
6)Exit
Enter Your Choice: 5
The Heap is:
The root nodes of Heap are:
10-->15-->20
----------------------------
Operations on Binomial heap
----------------------------
1)Insert Element in the heap
2)Extract Minimum key node
3)Decrease key of a node
4)Delete a node
5)Display Heap
6)Exit
Enter Your Choice: 3
Enter the key to be decreased: 20
Enter new key value: 18
PS C:\Users\Administrator\Desktop\ADSA running pgms>
```

Result

The C++ program of implementation of Fibonacci Heap performed successfully.

EX NO: 8	Graph Traversals
DATE:	

Aim

To Write C++ program to implement Graph Traversals

Algorithm

Depth First Search Algorithm
A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include <bits/stdC++.h>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph {
public:
        map<int, bool> visited;
        map<int, list<int> > adj;

        // function to add an edge to graph
        void addEdge(int v, int w);

        // DFS traversal of the vertices
        // reachable from v
        void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
```

```
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFS(int v)
{
        // Mark the current node as visited and
        // print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent
        // to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFS(*i);
}

// Driver code
int main()
{
        // Create a graph given in the above diagram
        Graph g;
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        cout << "Following is Depth First Traversal"
                        " (starting from vertex 2) \n";
        g.DFS(2);

        return 0;
}
```

Output
2 0 1 3

Algorithm

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
	int V; // No. of vertices

	// Pointer to an array containing adjacency
	// lists
	list<int> *adj;
public:
	Graph(int V); // Constructor

	// function to add an edge to graph
	void addEdge(int v, int w);

	// prints BFS traversal from a given source s
	void BFS(int s);
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
```

```
{
	// Mark all the vertices as not visited
	bool *visited = new bool[V];
	for(int i = 0; i < V; i++)
		visited[i] = false;

	// Create a queue for BFS
	list<int> queue;

	// Mark the current node as visited and enqueue it
	visited[s] = true;
	queue.push_back(s);

	// 'i' will be used to get all adjacent
	// vertices of a vertex
	list<int>::iterator i;

	while(!queue.empty())
	{
		// Dequeue a vertex from queue and print it
		s = queue.front();
		cout << s << " ";
		queue.pop_front();

		// Get all adjacent vertices of the dequeued
		// vertex s. If a adjacent has not been visited,
		// then mark it visited and enqueue it
		for (i = adj[s].begin(); i != adj[s].end(); ++i)
		{
			if (!visited[*i])
			{
				visited[*i] = true;
				queue.push_back(*i);
			}
		}
	}
}

// Driver program to test methods of graph class
int main()
{
	// Create a graph given in the above diagram
	Graph g(4);
	g.addEdge(0, 1);
	g.addEdge(0, 2);
	g.addEdge(1, 2);
	g.addEdge(2, 0);
	g.addEdge(2, 3);
	g.addEdge(3, 3);
```

```
        cout << "Following is Breadth First Traversal "
                << "(starting from vertex 2) \n";
        g.BFS(2);

        return 0;
}
```

Output

2 0 3 1

Result

The C++ program of implementation of Graph Traversals performed successfully.

EX NO: 9	Spanning Tree Implementation
DATE:	

Aim

To Write C++ program to implement Spanning Tree

Algorithm

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

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

Kruskal's algorithm

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

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

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

```
// C++ program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
#include<bits/stdC++.h>
using namespace std;

// Creating shortcut for an integer pair
typedef pair<int, int> iPair;

// Structure to represent a graph
struct Graph
{
	int V, E;
	vector< pair<int, iPair> > edges;

	// Constructor
	Graph(int V, int E)
	{
		this->V = V;
		this->E = E;
	}

	// Utility function to add an edge
	void addEdge(int u, int v, int w)
```

```
        {
                edges.push_back({w, {u, v}});
        }

        // Function to find MST using Kruskal's
        // MST algorithm
        int kruskalMST();
};

// To represent Disjoint Sets
struct DisjointSets
{
        int *parent, *rnk;
        int n;

        // Constructor.
        DisjointSets(int n)
        {
                // Allocate memory
                this->n = n;
                parent = new int[n+1];
                rnk = new int[n+1];

                // Initially, all vertices are in
                // different sets and have rank 0.
                for (int i = 0; i <= n; i++)
                {
                        rnk[i] = 0;

                        //every element is parent of itself
                        parent[i] = i;
                }
        }
        // Find the parent of a node 'u'
        // Path Compression
        int find(int u)
        {
                /* Make the parent of the nodes in the path
                from u--> parent[u] point to parent[u] */
                if (u != parent[u])
                        parent[u] = find(parent[u]);
                return parent[u];
        }
        // Union by rank
        void merge(int x, int y)
        {
                x = find(x), y = find(y);

                /* Make tree with smaller height
                a subtree of the other tree */
```

```
		if (rnk[x] > rnk[y])
			parent[y] = x;
		else // If rnk[x] <= rnk[y]
			parent[x] = y;

		if (rnk[x] == rnk[y])
			rnk[y]++;
	}
};

/* Functions returns weight of the MST*/

int Graph::kruskalMST()
{
	int mst_wt = 0; // Initialize result

	// Sort edges in increasing order on basis of cost
	sort(edges.begin(), edges.end());

	// Create disjoint sets
	DisjointSets ds(V);

	// Iterate through all sorted edges
	vector< pair<int, iPair> >::iterator it;
	for (it=edges.begin(); it!=edges.end(); it++)
	{
		int u = it->second.first;
		int v = it->second.second;

		int set_u = ds.find(u);
		int set_v = ds.find(v);

		// Check if the selected edge is creating
		// a cycle or not (Cycle is created if u
		// and v belong to same set)
		if (set_u != set_v)
		{
			// Current edge will be in the MST
			// so print it
			cout << u << " - " << v << endl;

			// Update MST weight
			mst_wt += it->first;

			// Merge two sets
			ds.merge(set_u, set_v);
		}
	}

	return mst_wt;
```

```
}

// Driver program to test above functions
int main()
{
	/* Let us create above shown weighted
	and undirected graph */
	int V = 9, E = 14;
	Graph g(V, E);

	// making above shown graph
	g.addEdge(0, 1, 4);
	g.addEdge(0, 7, 8);
	g.addEdge(1, 2, 8);
	g.addEdge(1, 7, 11);
	g.addEdge(2, 3, 7);
	g.addEdge(2, 8, 2);
	g.addEdge(2, 5, 4);
	g.addEdge(3, 4, 9);
	g.addEdge(3, 5, 14);
	g.addEdge(4, 5, 10);
	g.addEdge(5, 6, 2);
	g.addEdge(6, 7, 1);
	g.addEdge(6, 8, 6);
	g.addEdge(7, 8, 7);

	cout << "Edges of MST are \n";
	int mst_wt = g.kruskalMST();

	cout << "\nWeight of MST is " << mst_wt;

	return 0;
}
```

Output

```
Edges of MST are
6 - 7
2 - 8
5 - 6
0 - 1
2 - 5
2 - 3
0 - 7
3 - 4

Weight of MST is 37
```

Result

The C++ program of implementation of Spanning Tree performed successfully

EX NO: 10	Shortest Path Algorithms (Dijkstra's algorithm, Bellman Ford Algorithm)
DATE:	

Aim

To Write C++ program to implement Shortest path Algorithms(Dijikstra's algorithm ,Bellman Ford Algorithm)

10.A Bellman Ford Algorithm

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists

  return distance[], previous[]
```

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdC++.h>

// a structure to represent a weighted edge in graph
struct Edge {
        int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph {
        // V-> Number of vertices, E-> Number of edges
        int V, E;

        // graph is represented as an array of edges.
        struct Edge* edge;
};
```

```
// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
	struct Graph* graph = new Graph;
	graph->V = V;
	graph->E = E;
	graph->edge = new Edge[E];
	return graph;
}
// A utility function used to print the solution
void printArr(int dist[], int n)
{
	printf("Vertex Distance from Source\n");
	for (int i = 0; i < n; ++i)
		printf("%d \t\t %d\n", i, dist[i]);
}
// The main function that finds shortest distances from src
// to all other vertices using Bellman-Ford algorithm. The
// function also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
	int V = graph->V;
	int E = graph->E;
	int dist[V];

	// Step 1: Initialize distances from src to all other
	// vertices as INFINITE
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX;
	dist[src] = 0;

	// Step 2: Relax all edges |V| - 1 times. A simple
	// shortest path from src to any other vertex can have
	// at-most |V| - 1 edges
	for (int i = 1; i <= V - 1; i++) {
		for (int j = 0; j < E; j++) {
			int u = graph->edge[j].src;
			int v = graph->edge[j].dest;
			int weight = graph->edge[j].weight;
			if (dist[u] != INT_MAX
				&& dist[u] + weight < dist[v])
				dist[v] = dist[u] + weight;
		}
	}

	// Step 3: check for negative-weight cycles. The above
	// step guarantees shortest distances if graph doesn't
	// contain negative weight cycle. If we get a shorter
	// path, then there is a cycle.
```

```
	for (int i = 0; i < E; i++) {
		int u = graph->edge[i].src;
		int v = graph->edge[i].dest;
		int weight = graph->edge[i].weight;
		if (dist[u] != INT_MAX
			&& dist[u] + weight < dist[v]) {
			printf("Graph contains negative weight cycle");
			return; // If negative cycle is detected, simply
					// return
		}
	}

	printArr(dist, V);

	return;
}
// Driver program to test above functions
int main()
{
	/* Let us create the graph given in above example */
	int V = 5; // Number of vertices in graph
	int E = 8; // Number of edges in graph
	struct Graph* graph = createGraph(V, E);
	// add edge 0-1 (or A-B in above figure)
	graph->edge[0].src = 0;
	graph->edge[0].dest = 1;
	graph->edge[0].weight = -1;
	// add edge 0-2 (or A-C in above figure)
	graph->edge[1].src = 0;
	graph->edge[1].dest = 2;
	graph->edge[1].weight = 4;
	// add edge 1-2 (or B-C in above figure)
	graph->edge[2].src = 1;
	graph->edge[2].dest = 2;
	graph->edge[2].weight = 3;
	// add edge 1-3 (or B-D in above figure)
	graph->edge[3].src = 1;
	graph->edge[3].dest = 3;
	graph->edge[3].weight = 2;
	// add edge 1-4 (or B-E in above figure)
	graph->edge[4].src = 1;
	graph->edge[4].dest = 4;
	graph->edge[4].weight = 2;
	// add edge 3-2 (or D-C in above figure)
	graph->edge[5].src = 3;
	graph->edge[5].dest = 2;
	graph->edge[5].weight = 5;

	// add edge 3-1 (or D-B in above figure)
	graph->edge[6].src = 3;
```

```
        graph->edge[6].dest = 1;
        graph->edge[6].weight = 1;

        // add edge 4-3 (or E-D in above figure)
        graph->edge[7].src = 4;
        graph->edge[7].dest = 3;
        graph->edge[7].weight = -3;

        BellmanFord(graph, 0);

        return 0;
}
```

Output

Vertex	Distance from Source
0	**0**
1	**-1**
2	**2**
3	**-2**
4	**1**

10.B. Dijkstra's algorithm

Algorithm

```
function dijkstra(G, S)
   for each vertex V in G
      distance[V] <- infinite
      previous[V] <- NULL
      If V != S, add V to Priority Queue Q
   distance[S] <- 0

   while Q IS NOT EMPTY
      U <- Extract MIN from Q
      for each unvisited neighbour V of U
         tempDistance <- distance[U] + edge_weight(U, V)
         if tempDistance < distance[V]
            distance[V] <- tempDistance
            previous[V] <- U
   return distance[], previous[]
```

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 9
```

```
// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{

	// Initialize min value
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;

	return min_index;
}
// A utility function to print the constructed distance array
void printSolution(int dist[])
{
	cout <<"Vertex \t Distance from Source" << endl;
	for (int i = 0; i < V; i++)
		cout << i << " \t\t"<<dist[i]<< endl;
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
	int dist[V]; // The output array. dist[i] will hold the shortest
	// distance from src to i

	bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
	// path tree or shortest distance from src to i is finalized

	// Initialize all distances as INFINITE and stpSet[] as false
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX, sptSet[i] = false;

	// Distance of source vertex from itself is always 0
	dist[src] = 0;

	// Find shortest path for all vertices
	for (int count = 0; count < V - 1; count++) {
		// Pick the minimum distance vertex from the set of vertices not
		// yet processed. u is always equal to src in the first iteration.
		int u = minDistance(dist, sptSet);

		// Mark the picked vertex as processed
		sptSet[u] = true;

		// Update dist value of the adjacent vertices of the picked vertex.
		for (int v = 0; v < V; v++)
```

```
                        // Update dist[v] only if is not in sptSet, there is an edge from
                        // u to v, and total weight of path from src to v through u is
                        // smaller than current value of dist[v]
                        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                && dist[u] + graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist);
}

// driver program to test above function
int main()
{

        /* Let us create the example graph discussed above */
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                                { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                                { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                                { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                                { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                                { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                                { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                                { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                                { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
dijkstra(graph, 0);

        return 0;
}
```

Output

Vertex	Distance from Source
0	**0**
1	**4**
2	**12**
3	**19**
4	**21**
5	**11**
6	**9**
7	**8**
8	**14**

Result

The C++ program of implementation of Shortest Path Algorithms (Dijkstra's algorithm, Bellman Ford Algorithm) performed successfully.

EX NO: 11	Implementation of Matrix Chain Multiplication
DATE:	

Aim

To Write C++ program to implement Matrix Chain Multiplication

Algorithm

matOrder(array, n)
Input − List of matrices, the number of matrices in the list.
Output − Minimum number of matrix multiplication.

```
Begin
  define table minMul of size n x n, initially fill with all 0s
  for length := 2 to n, do
    for i:=1 to n-length, do
      j := i + length – 1
      minMul[i, j] := ∞
      for k := i to j-1, do
        q := minMul[i, k] + minMul[k+1, j] + array[i-1]*array[k]*array[j]
        if q < minMul[i, j], then minMul[i, j] := q
      done
    done
  done
  return minMul[1, n-1]
End
```

```
// See the Cormen book for details of the
// following algorithm
#include <bits/stdC++.h>
using namespace std;

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1..n
int MatrixChainOrder(int p[], int n)
{

        /* For simplicity of the program, one
        extra row and one extra column are
        allocated in m[][]. 0th row and 0th
        column of m[][] are not used */
        int m[n][n];

        int i, j, k, L, q;

        /* m[i, j] = Minimum number of scalar
        multiplications needed to compute the
        matrix A[i]A[i+1]...A[j] = A[i..j] where
        dimension of A[i] is p[i-1] x p[i] */
```

```
	// cost is zero when multiplying
	// one matrix.
	for (i = 1; i < n; i++)
		m[i][i] = 0;

	// L is chain length.
	for (L = 2; L < n; L++)
	{
		for (i = 1; i < n - L + 1; i++)
		{
			j = i + L - 1;
			m[i][j] = INT_MAX;
			for (k = i; k <= j - 1; k++)
			{
				// q = cost/scalar multiplications
				q = m[i][k] + m[k + 1][j]
					+ p[i - 1] * p[k] * p[j];
				if (q < m[i][j])
					m[i][j] = q;
			}
		}
	}

	return m[1][n - 1];
}

// Driver Code
int main()
{
	int arr[] = { 1, 2, 3, 4 };
	int size = sizeof(arr) / sizeof(arr[0]);

	cout << "Minimum number of multiplications is "
		<< MatrixChainOrder(arr, size);

	getchar();
	return 0;
}
```

Ouput
Minimum number of multiplications is 18

Result
The C++ program of implementation of Matrix Chain Multiplication performed successfully.

EX NO: 12	Activity Selection and Huffman Coding Implementation
DATE:	

Aim

To Write C++ program to implement Activity selection and Huffman coding

Algorithm

Input Data for the Algorithm:

- act[] array containing all the activities.
- s[] array containing the starting time of all the activities.
- f[] array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- sol[] array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.
Step 2: Select the first activity from sorted array act[] and add it to sol[] array.
Step 3: Repeat steps 4 and 5 for the remaining activities in act[].
Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.
Step 5: Select the next activity in act[] array.
Step 6: Print the sol[] array.

```
#include <bits/stdC++.h>
using namespace std;
#define N 6             // defines the number of activities
// Structure represents an activity having start time and finish time.
struct Activity
{
   int start, finish;
};
// This function is used for sorting activities according to finish time
bool Sort_activity(Activity s1, Activity s2)
{
   return (s1.finish< s2.finish);
}

/*      Prints maximum number of activities that can
        be done by a single person or single machine at a time.
*/
void print_Max_Activities(Activity arr[], int n)
{
   // Sort activities according to finish time
        sort(arr, arr+n, Sort_activity);

        cout<< "Following activities are selected \n";
```

```
   // Select the first activity
   int i = 0;
        cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";
 // Consider the remaining activities from 1 to n-1
   for (int j = 1; j < n; j++)
   {
        // Select this activity if it has start time greater than or equal
        // to the finish time of previously selected activity
        if (arr[j].start>= arr[i].finish)
        {
                        cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ") \n";
                        i = j;
        }
   }
}

// Driver program
int main()
{
   Activity arr[N];
        for(int i=0; i<=N-1; i++)
        {
                cout<<"Enter the start and end time of "<<i+1<<" activity \n";
                cin>>arr[i].start>>arr[i].finish;
   }

        print_Max_Activities(arr, N);
   return 0;
}
```

Output

```
Enter the start and end time of 1 activity
1 2
Enter the start and end time of 2 activity
3 4
Enter the start and end time of 3 activity
0 6
Enter the start and end time of 4 activity
5 7
Enter the start and end time of 5 activity
5 9
Enter the start and end time of 6 activity
8 9
Following activities are selected
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Result

The C++ program of implementation of Activity selection and Huffman coding performed successfully.