

**THIRUMALAI ENGINEERING COLLEGE**  
**KILAMBI,KANCHIPURAM-631551**



**MASTER OF COMPUTER APPLICATIONS**  
**ADVANCED DATA STRUCTURES AND ALGORITHMS**  
**LABORATORY**  
**(MC4111)**

**NAME** : \_\_\_\_\_

**REG.NUMBER** : \_\_\_\_\_

**SEMESTER** : \_\_\_\_\_

**YEAR** : \_\_\_\_\_

**THIRUMALAI ENGINEERING COLLEGE**  
**KILAMBI, KANCHIPURAM-631551**  
**DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS**



**BONAFIDE CERTIFICATE**

This is to certify that this is a bonafide work done by  
Mr./Miss. \_\_\_\_\_ Reg.No. \_\_\_\_\_ For  
the **MC4111- ADVANCED DATA STRUCTURES AND ALGORITHMS**  
**LABORATORY** as a part of **MASTER OF COMPUTER APPLICATIONS**  
course in **Thirumalai Engineering College**, Kanchipuram during the year of 2021-  
2022. The Record is found to be completed and satisfactory.

**Staff In-Charge**

**Head of the Department**

Submitted for the Practical Examination held on

**Internal Examiner**

**External Examiner**

## LIST OF EXPERIMENTS

<b>S. NO.</b>	<b>DATE</b>	<b>NAME OF THE EXPERIMENT</b>	<b>PAGE. NO</b>	<b>SIGN</b>
1.		<b>IMPLEMENTATION OF RECURSIVE FUNCTION USING</b> <b>1.A)TREE TRAVERSAL</b> <b>1.B)FIBONACCI</b>		
2.		<b>IMPLEMENTATION OF ITERATIVE FUNCTION USING</b> <b>2.A)TREE TRAVERSAL</b> <b>2.B)FIBONACCI</b>		
3.		<b>IMPLEMENTATION OF</b> <b>3.A)MERGE SORT</b> <b>3.B)QUICK SORT</b>		
4.		<b>IMPLEMENTATION OF A BINARY SEARCH TREE</b>		
5.		<b>RED-BLACK TREE IMPLEMENTATION</b>		
6.		<b>HEAP IMPLEMENTATION</b> <b>6.A)MAXIMUM HEAP</b> <b>6.B)MINIMUM HEAP</b>		
7.		<b>FIBONACCI HEAP IMPLEMENTATION</b>		

<b>8.</b>		<b>GRAPH TRAVERSAL</b> <b>8.A)BFS</b> <b>8.B)DFS</b>		
<b>9.</b>		<b>SPANNING TREE IMPLEMENTATION</b> <b>9.A)KRUSKAL ALGORITHM</b> <b>9.B)PRIM'S ALGORITHM</b>		
<b>10.</b>		<b>SHORTEST PATH ALGORITHM</b> <b>10.A)DIJIKSTRA ALGORITHM</b> <b>10.B)BELLMAN FORD ALGORITHM</b>		
<b>11.</b>		<b>IMPLEMENTATION OF MATRIX CHAIN MULTIPLICATION</b>		
<b>12.</b>		<b>12.A)IMPLEMENTATION OF ACTIVITY SELECTION</b> <b>12.B)IMPLEMENTATION OF HUFFMAN CODING IMPLEMENTATION</b>		

**EXERCISE NO: 1**

**DATE:**

**IMPLEMENTATION OF RECURSIVE FUNCTION FOR TREE TRAVERSAL AND FIBONACCI**

**1.A) RECURSIVE FUNCTION USING TREE TRAVERSAL**

**AIM:**

Write a program to implement recursive function for Tree traversal

**ALGORITHM:**

Step 1: 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)

Step 2: 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)

Step 3: 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.

**PROGRAM:**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node *left, *right;
};

Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
```

```

}

void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    printPostorder(node->left);

    printPostorder(node->right);

    cout << node->data << " ";
}

void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);

    cout << node->data << " ";

    printInorder(node->right);
}

void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    cout << node->data << " ";

    printPreorder(node->left);

    printPreorder(node->right);
}

int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);

```

```

    root->left->right = newNode(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
PS E:\>

```

### RESULT:

Thus the above program for implementation of Tree traversal using Recursive function as been executed successfully.

## 1.B) RECURSIVE FUNCTION USING FIBONACCI

### AIM:

Write a program to implement recursive function for Fibonacci series.

### ALGORITHM:

Step 1: START

Step 2: Input the non-negative integer 'n'

    If (n==0 || n==1)

        return n;

    else

        return fib(n-1)+fib(n-2);

    Print, nth Fibonacci number

Step 3: END

### PROGRAM:

```
#include <iostream>
using namespace std;
int fib(int x) {
    if((x==1)||(x==0)) {
        return(x);
    }else {
        return(fib(x-1)+fib(x-2));
    }
}
int main() {
    int x , i=0;
    cout << "Enter the number of terms of series : ";
    cin >> x;
    cout << "\nFibonacci Series : ";
    while(i < x) {
        cout << " " << fib(i);
        i++;
    }
    return 0;
}
```



**OUTPUT:**

```
Enter the number of terms of series : 10  
Fibonnaci Series : 0 1 1 2 3 5 8 13 21 34  
PS E:\> █
```

**RESULT:**

Thus the above program for implementation of Fibonacci series using Recursion as been executed successfully.

**EXERCISE NO: 2**

**DATE:**

**IMPLEMENTATION OF ITERATION FOR TREE TRAVERSAL AND FIBONACCI**

**2.A) ITERATIVE FUNCTION USING TREE TRAVERSAL**

**AIM:**

Write a program to implement iterative function for Tree traversal.

**ALGORITHM:**

Step 1: 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)

Step 2: 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)

Step 3: 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.

**PROGRAM:**

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

struct Node {
    int data;
    struct Node *left, *right;
};

Node* newNode(int data)
{
```

```

        Node* temp = new Node;
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
    }

void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    printPostorder(node->left);

    printPostorder(node->right);

    cout << node->data << " ";
}

void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);

    cout << node->data << " ";

    printInorder(node->right);
}

void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    cout << node->data << " ";

    printPreorder(node->left);

    printPreorder(node->right);
}

int main()

```

```

{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(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
PS E:\>

```

#### RESULT:

Thus the above program for implementation of Tree traversal using Iterative method as been executed successfully.

## 2.B) ITERATIVE FUNCTION USING FIBONACCI

### AIM:

Write a program to implement iterative function for Fibonacci Series.

### ALGORITHM:

Step 1: The sequence starts with 0 and 1, and we find the following number by summing the two numbers that occur before it.

Step 2: In the case of the third number in the sequence, the two preceding numbers are 0 and 1, so the third number is  $(0 + 1) = 1$ .

Step 3: In the case of the fourth number in the sequence, the two preceding numbers are 1 and 1, so the fourth number is  $(1 + 1) = 2$ .

Step 4: Each number in the Fibonacci sequence is a Fibonacci number,  $\text{fib}(\text{num})$ .

### PROGRAM:

```
#include <iostream>
using namespace std;
void fib(int num) {
    int x = 0, y = 1, z = 0;
    for (int i = 0; i < num; i++) {
        cout << x << " ";
        z = x + y;
        x = y;
        y = z;
    }
}
int main() {
    int num;
    cout << "Enter the number : ";
    cin >> num;
    cout << "\n\nThe fibonacci series : " ;
    fib(num);
    return 0;
}
```

### OUTPUT :

```
Enter the number : 10  
  
The fibonacci series : 0 1 1 2 3 5 8 13 21 34  
PS E:\> █
```

### RESULT:

Thus the above program for implementation of Fibonacci series using Iterative method as been executed successfully.

**EXERCISE NO: 3**

**DATE:**

## **IMPLEMENTATION OF MERGE SORT AND QUICK SORT**

### **3.A) IMPLEMENTATION OF MERGE SORT**

**AIM:**

Write a program to sort the values using merge sort .

**ALGORITHM:**

Step 1: First, we considered an array Hello[10, 3, 7, 1, 15, 14, 9, 22] in this array there are total 8 elements

Step 2: As we saw earlier merge sort uses the divide and conquer approach to sort the elements. We found m which lies in the middle of our array and divided our array from the middle where  $m = (a + n)/2$  'a' is the index of the leftmost element and n is the index of the rightmost element of our array.

Step 3: After the first division, we have 2 parts consisting of 4 elements each. Let's look at the first half [10, 3, 7, 1].

Step 4: We divide [10, 3, 7, 1] in 2 parts [10, 3] and [7, 1]. After that we divide them further into [10], [3], [7], [1]. Further division is not possible as we can't calculate m. a list containing a single element is always considered sorted.

Step 5: How does merging take place? Let's find out. First [10] and [3] is compared and merged in ascending order [3, 10] in the same way we get [1, 7]

Step 6: After that, we compare [3, 10] and [1, 7]. Once compared we merge them in ascending order and We get [1, 3, 7, 10].

Step 7: [15, 14, 9, 2] is also divided and combined in a similar way to form [9, 14, 15, 22].

Step 8: In the last step we compare and combine [1, 3, 7, 10] [9, 14, 15, 22] to give us our sorted array i.e [1, 3, 7, 9, 10, 14, 15, 22].

**PROGRAM:**

```
#include<stdlib.h>

#include<iostream>

using namespace std;

void merge(int a[], int Firstindex, int m, int Lastindex);

void mergeSort(int a[], int Firstindex, int Lastindex)
{
    if (Firstindex < Lastindex)
    {
        int m = Firstindex + (Lastindex - Firstindex)/2;

        mergeSort(a, Firstindex, m);
        mergeSort(a, m+1, Lastindex);

        merge(a, Firstindex, m, Lastindex);
    }
}

void merge(int a[], int Firstindex, int m, int Lastindex)
{
    int x;
    int y;
    int z;
    int sub1 = m - Firstindex + 1;
    int sub2 = Lastindex - m;

    int First[sub1];

    int Second[sub2];

    for (x = 0; x < sub1; x++)
        First[x] = a[Firstindex + x];
    for (y = 0; y < sub2; y++)
        Second[y] = a[m + 1 + y];

    x = 0;
    y = 0;
    z = Firstindex;
    while (x < sub1 && y < sub2)
    {
```



```

        if (First[x] <= Second[y])
        {
            a[z] = First[x];
            x++;
        }
        else
        {
            a[z] = Second[y];
            y++;
        }
        z++;
    }
    while (x < sub1)
    {
        a[z] = First[x];
        x++;
        z++;
    }
    while (y < sub2)
    {
        a[z] = Second[y];
        y++;
        z++;
    }
}

int main()
{
    int size;
    cout<<"Enter the size of the Array that is to be sorted: "; cin>>size;
    int Hello[size],i;
    cout<<"Enter the elements of the array one by one:";
    for(i=0; i<size; i++) cin>>Hello[i];
    mergeSort(Hello, 0, size - 1);
    cout<<"The Sorted List is: ";
    for(i=0; i<size; i++)
    {
        cout<<Hello[i]<<" ";
    }
    return 0;
}

```

### OUTPUT :

```
Enter the size of the Array that is to be sorted: 8
Enter the elements of the array one by one:10
3
7
1
15
14
9
22
The Sorted List is 1 3 7 9 10 14 15 22
PS D:\c++> |
```

### RESULT:

Thus the above program for implementation of Merge Sort Algorithm as been executed successfully.

### 3.B) IMPLEMENTATION OF QUICK SORT

#### AIM:

Write a program to sort the values using Quick sort..

#### ALGORITHM :

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

Step 2: 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.

Step 3: The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

Step 4: At this point, elements are already sorted. Finally, elements are combined to form a sorted array

#### PROGRAM :

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

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void printArray(int array[], int size) {
    int i;
    for (i = 0; i < size; i++)
        cout << array[i] << " ";
    cout << endl;
}

int partition(int array[], int low, int high) {
    int pivot = array[high];
```

```

int i = (low - 1);

for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

        i++;

        swap(&array[i], &array[j]);
    }
}

swap(&array[i + 1], &array[high]);

return (i + 1);
}

void quickSort(int array[], int low, int high) {
    if (low < high) {

        int pi = partition(array, low, high);

        quickSort(array, low, pi - 1);

        quickSort(array, pi + 1, high);
    }
}

int main() {
    int data[] = {8, 7, 6, 1, 0, 9, 2};
    int n = sizeof(data) / sizeof(data[0]);

    cout << "Unsorted Array: \n";
    printArray(data, n);

    quickSort(data, 0, n - 1);

    cout << "Sorted array in ascending order: \n";
    printArray(data, n);
}

```

## OUTPUT :

```
Unsorted Array:  
8 7 6 1 0 9 2  
Sorted array in ascending order:  
0 1 2 6 7 8 9  
PS D:\c++> |
```

## RESULT:

Thus the above program for implementation of Quick Sort Algorithm as been executed successfully.

**EXERCISE NO: 4****DATE:****IMPLEMENTATION OF A BINARY SEARCH TREE****AIM:**

Write a program to implementing and deleting a node from tree by using binary search tree

**ALGORITHM:**

Step 1: In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree

Step 2: In the second case, the node to be deleted has a single child node. In such a case follow the steps below:

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

Step 3: 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.

**PROGRAM:**

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

```
struct node {
    int key;
    struct node *left, *right;
};
```

```
struct node *newNode(int item) {
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```

}

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

        cout << root->key << " -> ";

        inorder(root->right);
    }
}

struct node *insert(struct node *node, int key) {
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}

struct node *minValueNode(struct node *node) {
    struct node *current = node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}

struct node *deleteNode(struct node *root, int key) {
    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        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;
}

struct node *temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);
}
return root;
}

int main() {
    struct node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    cout << "Inorder traversal: ";
    inorder(root);

    cout << "\nAfter deleting 10\n";
    root = deleteNode(root, 10);
    cout << "Inorder traversal: ";
    inorder(root);
}

```

## OUTPUT:

```

Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 14 ->
After deleting 10
Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 14 ->
PS D:\c++>

```

## RESULT:

Thus the above program for implementation of Binary Search Tree as been executed successfully.



**EXERCISE NO: 5****DATE:****IMPLEMENTATION OF RED-BLACK TREE****AIM:**

Write a program to implementing Red-Black Tree and delete a node from it.

**ALGORITHM:**

Step 1: Save the color of nodeToBeDeleted in originalColor.

Step 2: If the left child of nodeToBeDeleted is NULL

- Assign the right child of nodeToBeDeleted to x.
- Transplant nodeToBeDeleted with x.

Step 3: Else if the right child of nodeToBeDeleted is NULL

- Assign the left child of nodeToBeDeleted into x.
- Transplant nodeToBeDeleted with x.

Step 4: Else

- Assign the minimum of right subtree of nodeToBeDeleted into y.
- Save the color of y in originalColor.
- Assign the rightChild of y into x.
- If y is a child of nodeToBeDeleted, then set the parent of x as y.
- Else, transplant y with rightChild of y.
- Transplant nodeToBeDeleted with y.
- Set the color of y with originalColor.

Step 5: If the originalColor is BLACK, call DeleteFix(x).

Step 6: After deletion check whether the property of Red-black tree is satisfied or not, If not then do the appropriate rotation and recolor to fix it.

**PROGRAM:**

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

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

```

Node *parent;
Node *left;
Node *right;
int color;
};
typedef Node *NodePtr;

class RedBlackTree {
private:
NodePtr root;
NodePtr TNULL;

void initializeNULLNode(NodePtr node, NodePtr parent) {
    node->data = 0;
    node->parent = parent;
    node->left = nullptr;
    node->right = nullptr;
    node->color = 0;
}

void preOrderHelper(NodePtr node) {
    if (node != TNULL) {
        cout << node->data << " ";
        preOrderHelper(node->left);
        preOrderHelper(node->right);
    }
}

void inOrderHelper(NodePtr node) {
    if (node != TNULL) {
        inOrderHelper(node->left);
        cout << node->data << " ";
        inOrderHelper(node->right);
    }
}

void postOrderHelper(NodePtr node) {
    if (node != TNULL) {
        postOrderHelper(node->left);
        postOrderHelper(node->right);
        cout << node->data << " ";
    }
}

NodePtr searchTreeHelper(NodePtr node, int key) {
    if (node == TNULL || key == node->data) {
        return node;
    }
    if (key < node->data) {

```

```

    return searchTreeHelper(node->left, key);
}
return searchTreeHelper(node->right, key);
}
void deleteFix(NodePtr x) {
    NodePtr s;
    while (x != root && x->color == 0) {
        if (x == x->parent->left) {
            s = x->parent->right;
            if (s->color == 1) {
                s->color = 0;
                x->parent->color = 1;
                leftRotate(x->parent);
                s = x->parent->right;
            }
            if (s->left->color == 0 && s->right->color == 0) {
                s->color = 1;
                x = x->parent;
            } else {
                if (s->right->color == 0) {
                    s->left->color = 0;
                    s->color = 1;
                    rightRotate(s);
                    s = x->parent->right;
                }
                s->color = x->parent->color;
                x->parent->color = 0;
                s->right->color = 0;
                leftRotate(x->parent);
                x = root;
            }
        } else {
            s = x->parent->left;
            if (s->color == 1) {
                s->color = 0;
                x->parent->color = 1;
                rightRotate(x->parent);
                s = x->parent->left;
            }
        }
        if (s->right->color == 0 && s->right->color == 0) {
            s->color = 1;
            x = x->parent;
        } else {
            if (s->left->color == 0) {
                s->right->color = 0;
                s->color = 1;
            }
        }
    }
}

```

```

        leftRotate(s);
        s = x->parent->left;
    }

    s->color = x->parent->color;
    x->parent->color = 0;
    s->left->color = 0;
    rightRotate(x->parent);
    x = root;
}
}
}
x->color = 0;
}
void rbTransplant(NodePtr u, NodePtr v) {
    if (u->parent == nullptr) {
        root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    v->parent = u->parent;
}
void deleteNodeHelper(NodePtr node, int key) {
    NodePtr z = TNULL;
    NodePtr x, y;
    while (node != TNULL) {
        if (node->data == key) {
            z = node;
        }
        if (node->data <= key) {
            node = node->right;
        } else {
            node = node->left;
        }
    }
    if (z == TNULL) {
        cout << "Key not found in the tree" << endl;
        return;
    }
    y = z;
    int y_original_color = y->color;
    if (z->left == TNULL) {
        x = z->right;
        rbTransplant(z, z->right);
    }

```

```

} else if (z->right == TNULL) {
    x = z->left;
    rbTransplant(z, z->left);
} else {
    y = minimum(z->right);
    y_original_color = y->color;
    x = y->right;
    if (y->parent == z) {
        x->parent = y;
    } else {
        rbTransplant(y, y->right);
        y->right = z->right;
        y->right->parent = y;
    }
    rbTransplant(z, y);
    y->left = z->left;
    y->left->parent = y;
    y->color = z->color;
}
delete z;
if (y_original_color == 0) {
    deleteFix(x);
}
}
void insertFix(NodePtr k) {
    NodePtr u;
    while (k->parent->color == 1) {
        if (k->parent == k->parent->parent->right) {
            u = k->parent->parent->left;
            if (u->color == 1) {
                u->color = 0;
                k->parent->color = 0;
                k->parent->parent->color = 1;
                k = k->parent->parent;
            } else {
                if (k == k->parent->left) {
                    k = k->parent;
                    rightRotate(k);
                }
                k->parent->color = 0;
                k->parent->parent->color = 1;
                leftRotate(k->parent->parent);
            }
        } else {
            u = k->parent->parent->right;

```

```

        if (u->color == 1) {
            u->color = 0;
            k->parent->color = 0;
            k->parent->parent->color = 1;
            k = k->parent->parent;
        } else {
            if (k == k->parent->right) {
                k = k->parent;
                leftRotate(k);
            }
            k->parent->color = 0;
            k->parent->parent->color = 1;
            rightRotate(k->parent->parent);
        }
    }
    if (k == root) {
        break;
    }
}
root->color = 0;
}

void printHelper(NodePtr root, string indent, bool last) {
    if (root != TNULL) {
        cout << indent;
        if (last) {
            cout << "R----";
            indent += "  ";
        } else {
            cout << "L----";
            indent += "|  ";
        }
    }
    string sColor = root->color ? "RED" : "BLACK";
    cout << root->data << "(" << sColor << ")" << endl;
    printHelper(root->left, indent, false);
    printHelper(root->right, indent, true);
}

public:
RedBlackTree() {
    TNULL = new Node;
    TNULL->color = 0;
    TNULL->left = nullptr;
    TNULL->right = nullptr;
    root = TNULL;
}

void preorder() {

```

```

    preOrderHelper(this->root);
}
void inorder() {
    inOrderHelper(this->root);
}
void postorder() {
    postOrderHelper(this->root);
}
NodePtr searchTree(int k) {
    return searchTreeHelper(this->root, k);
}
NodePtr minimum(NodePtr node) {
    while (node->left != TNULL) {
        node = node->left;
    }
    return node;
}
NodePtr maximum(NodePtr node) {
    while (node->right != TNULL) {
        node = node->right;
    }
    return node;
}
NodePtr successor(NodePtr x) {
    if (x->right != TNULL) {
        return minimum(x->right);
    }
    NodePtr y = x->parent;
    while (y != TNULL && x == y->right) {
        x = y;
        y = y->parent;
    }
    return y;
}

NodePtr predecessor(NodePtr x) {
    if (x->left != TNULL) {
        return maximum(x->left);
    }

    NodePtr y = x->parent;
    while (y != TNULL && x == y->left) {
        x = y;
        y = y->parent;
    }
    return y;
}

```

```

}
void leftRotate(NodePtr x) {
    NodePtr y = x->right;
    x->right = y->left;
    if (y->left != TNULL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
        this->root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
void rightRotate(NodePtr x) {
    NodePtr y = x->left;
    x->left = y->right;
    if (y->right != TNULL) {
        y->right->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
        this->root = y;
    } else if (x == x->parent->right) {
        x->parent->right = y;
    } else {
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
void insert(int key) {
    NodePtr node = new Node;
    node->parent = nullptr;
    node->data = key;
    node->left = TNULL;
    node->right = TNULL;
    node->color = 1;

    NodePtr y = nullptr;
    NodePtr x = this->root;

```



```

while (x != TNULL) {
    y = x;
    if (node->data < x->data) {
        x = x->left;
    } else {
        x = x->right;
    }
}

node->parent = y;
if (y == nullptr) {
    root = node;
} else if (node->data < y->data) {
    y->left = node;
} else {
    y->right = node;
}

if (node->parent == nullptr) {
    node->color = 0;
    return;
}

if (node->parent->parent == nullptr) {
    return;
}

insertFix(node);
}

NodePtr getRoot() {
    return this->root;
}

void deleteNode(int data) {
    deleteNodeHelper(this->root, data);
}

void printTree() {
    if (root) {
        printHelper(this->root, "", true);
    }
}

};

int main() {

```

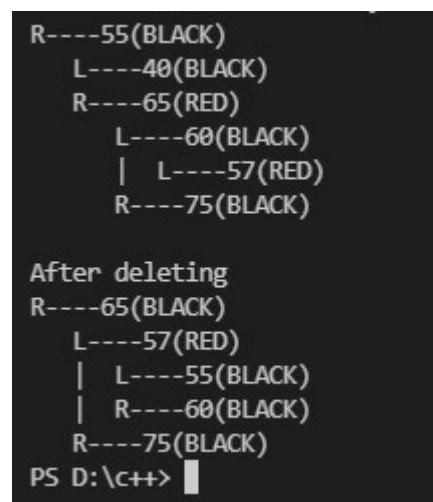
```

RedBlackTree bst;
bst.insert(55);
bst.insert(40);
bst.insert(65);
bst.insert(60);
bst.insert(75);
bst.insert(57);

bst.printTree();
cout << endl
<< "After deleting" << endl;
bst.deleteNode(40);
bst.printTree();
}

```

### OUTPUT:



```

R----55(BLACK)
  L----40(BLACK)
    R----65(RED)
      L----60(BLACK)
        |  L----57(RED)
          R----75(BLACK)

After deleting
R----65(BLACK)
  L----57(RED)
    |  L----55(BLACK)
      |  R----60(BLACK)
        R----75(BLACK)
PS D:\c++>

```

### RESULT:

Thus the above program for implementation of Red-BlackTree as been executed successfully.

**EXERCISE NO: 6**

**DATE:**

## **IMPLEMENTATION OF HEAP**

### **6.A) MAXIMUM HEAP IMPLEMENTATION**

**AIM:**

Write a program to implementing the Maximum Heap.

**ALGORITHM:**

Step 1: Build a max heap from the given data such that the root is the highest element of the heap.

Step 2: Remove the root i.e. the highest element from the heap and replace or swap it with the last element of the heap.

Step 3: Then adjust the max heap, so as to not to violate the max heap properties (heapify).

Step 4: The above step reduces the heap size by 1.

Step 5: Repeat the above three steps until the heap size is reduced to 1.

**PROGRAM :**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
using namespace std;

struct PriorityQueue
{
private:
    vector<int> A;
    int PARENT(int i) {
        return (i - 1) / 2;
    }
};
```

```

    }
    int LEFT(int i) {
        return (2*i + 1);
    }

    int RIGHT(int i) {
        return (2*i + 2);
    }
    void heapify_down(int i)
    {
        int left = LEFT(i);
        int right = RIGHT(i);
        int largest = i;

        if (left < size() && A[left] > A[i]) {
            largest = left;
        }
        if (right < size() && A[right] > A[largest]) {
            largest = right;
        }
        if (largest != i)
        {
            swap(A[i], A[largest]);
            heapify_down(largest);
        }
    }
    void heapify_up(int i)
    {
        if (i && A[PARENT(i)] < A[i])
        {
            swap(A[i], A[PARENT(i)]);

            heapify_up(PARENT(i));
        }
    }

public:
    unsigned int size() {
        return A.size();
    }

    bool empty() {
        return size() == 0;
    }

```

```

void push(int key)
{
    A.push_back(key);

    int index = size() - 1;
    heapify_up(index);
}
void pop()
{
    try {
        if (size() == 0)
        {
            throw out_of_range("Vector<X>::at() : "
                               "index is out of range(Heap underflow)");
        }
        A[0] = A.back();
        A.pop_back();

        heapify_down(0);
    }

    catch (const out_of_range &oor) {
        cout << endl << oor.what();
    }
}
int top()
{
    try {
        if (size() == 0)
        {
            throw out_of_range("Vector<X>::at() : "
                               "index is out of range(Heap underflow)");
        }

        return A.at(0);        // or return A[0];
    }
    catch (const out_of_range &oor) {
        cout << endl << oor.what();
    }
}

};

int main()
{
    PriorityQueue pq;

```

```

    pq.push(3);
    pq.push(2);
    pq.push(15);
    cout << "Size is " << pq.size() << endl;
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    pq.push(5);
    pq.push(4);
    pq.push(45);
    cout << endl << "Size is " << pq.size() << endl;
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    cout << endl << boolalpha << pq.empty();
    pq.top();
    pq.pop();

    return 0;
}

```

## OUTPUT:

```

Size is 3
15 3
Size is 4
45 5 4 2
true
Vector<X>::at() : index is out of range(Heap underflow)
Vector<X>::at() : index is out of range(Heap underflow)
PS D:\c++> 

```

## RESULT:

Thus the above program for implementation of Maximum Heap as been executed successfully.

## 6.B) MINIMUM HEAP IMPLEMENTATION

### AIM:

Write a program to implementing the Minimum Heap.

### ALGORITHM:

Step 1: Build a min heap from the given data such that the root is the lowest element of the heap.

Step 2: Remove the root i.e. the least element from the heap and replace or swap it with the last element of the heap.

Step 3: Then adjust the min heap, so as to not to violate the min heap properties (heapify).

Step 4: The above step reduces the heap size by 1.

Step 5: Repeat the above three steps until the heap size is reduced to 1.

### PROGRAM:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
using namespace std;

struct PriorityQueue
{
private:
    vector<int> A;

    int PARENT(int i) {
        return (i - 1) / 2;
    }

    int LEFT(int i) {
        return (2*i + 1);
    }

    int RIGHT(int i) {
```

```

        return (2*i + 2);
    }
    void heapify_down(int i)
    {

        int left = LEFT(i);
        int right = RIGHT(i);
        int smallest = i;

        if (left < size() && A[left] < A[i]) {
            smallest = left;
        }

        if (right < size() && A[right] < A[smallest]) {
            smallest = right;
        }

        if (smallest != i)
        {
            swap(A[i], A[smallest]);
            heapify_down(smallest);
        }
    }

    void heapify_up(int i)
    {
        if (i && A[PARENT(i)] > A[i])
        {
            swap(A[i], A[PARENT(i)]);

            heapify_up(PARENT(i));
        }
    }

public:
    unsigned int size() {
        return A.size();
    }

    bool empty() {
        return size() == 0;
    }

    void push(int key)

```



```

    {
        A.push_back(key);

        int index = size() - 1;
        heapify_up(index);
    }
void pop()
{
    try {
        if (size() == 0)
        {
            throw out_of_range("Vector<X>::at() : "
                               "index is out of range(Heap underflow)");
        }

        A[0] = A.back();
        A.pop_back();

        heapify_down(0);
    }
    catch (const out_of_range &oor) {
        cout << endl << oor.what();
    }
}
int top()
{
    try {
        if (size() == 0)
        {
            throw out_of_range("Vector<X>::at() : "
                               "index is out of range(Heap underflow)");
        }

        return A.at(0);
    }
    catch (const out_of_range &oor) {
        cout << endl << oor.what();
    }
}

};

int main()
{
    PriorityQueue pq;
    pq.push(3);
    pq.push(2);

```

```

    pq.push(15);

    cout << "Size is " << pq.size() << endl;

    cout << pq.top() << " ";
    pq.pop();

    cout << pq.top() << " ";
    pq.pop();

    pq.push(5);
    pq.push(4);
    pq.push(45);

    cout << endl << "Size is " << pq.size() << endl;
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    cout << pq.top() << " ";
    pq.pop();
    cout << endl << boolalpha << pq.empty();
    pq.top();
    pq.pop();
    return 0;
}

```

## OUTPUT:

```

Size is 3
2 3
Size is 4
4 5 15 45
true
Vector<X>::at() : index is out of range(Heap underflow)
Vector<X>::at() : index is out of range(Heap underflow)
PS D:\c++> 

```

## RESULT:

Thus the above program for implementation of Minimum Heap as been executed successfully.

**EXERCISE NO: 7****DATE:****IMPLEMENTATION OF FIBONACCI HEAP****AIM:**

Write a program to implement Fibonacci in Heap.

**ALGORITHM:**

Step 1: Delete the min node.

Step 2: Set the min-pointer to the next root in the root list.

Step 3: Create an array of size equal to the maximum degree of the trees in the heap before deletion.

Step 4: Do the following (steps 5-7) until there are no multiple roots with the same degree.

Step 5: Map the degree of current root (min-pointer) to the degree in the array.

Step 6: Map the degree of next root to the degree in array.

Step 7: If there are more than two mappings for the same degree, then apply union operation to those roots such that the min-heap property is maintained (i.e. the minimum is at the root).

**PROGRAM:**

```
#include <cmath>
#include <cstdlib>
#include <iostream>

using namespace std;

struct node {
    int n;
    int degree;
    node *parent;
    node *child;
    node *left;
    node *right;
    char mark;

    char C;
};
```

```

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(); }
};

node *FibonacciHeap::InitializeHeap() {
    node *np;
    np = NULL;
    return np;
}

node *FibonacciHeap::Create_node(int value) {
    node *x = new node;
    x->n = value;
    return x;
}

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;
}

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++;
}

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;
}

int FibonacciHeap::Display(node *H) {

```

```

node *p = H;
if (p == NULL) {
    cout << "Empty Heap" << endl;
    return 0;
}
cout << "Root Nodes: " << endl;

do {
    cout << p->n;
    p = p->right;
    if (p != H) {
        cout << "-->";
    }
} while (p != H && p->right != NULL);
cout << endl;
}

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;
}

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;
            }
        }
    } while (pt != NULL);

    A[d] = x;
}

```

```

        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];
}
}
}

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;
}

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';
}

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);
    }
}
}

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;
}

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;
}

int main() {
    int n, m, l;
    FibonacciHeap fh;

```

```

node *p;
node *H;
H = fh.InitializeHeap();
p = fh.Create_node(7);
H = fh.Insert(H, p);
p = fh.Create_node(3);
H = fh.Insert(H, p);
p = fh.Create_node(17);
H = fh.Insert(H, p);
p = fh.Create_node(24);
H = fh.Insert(H, p);

fh.Display(H);

p = fh.Extract_Min(H);
if (p != NULL)
    cout << "The node with minimum key: " << p->n << endl;
else
    cout << "Heap is empty" << endl;

m = 26;
l = 16;
fh.Decrease_key(H, m, l);

m = 16;
fh.Delete_key(H, m);
}

```

## OUTPUT:

```

Root Nodes:
3-->7-->17-->24
The node with minimum key: 3
Node not found in the Heap
Node not found in the Heap
Key not Deleted
PS D:\C++>

```

## RESULT:

Thus the above program for implementation of Fibonacci Heap as been executed successfully.

**EXERCISE NO: 8**

**DATE:**

## **IMPLEMENTATION OF GRAPH TRAVERSALS**

### **8.A) DEPTH FIRST SEARCH IMPLEMENTATION**

**AIM:**

Write a program to find shortest path of graph traversals using DFS algorithm.

**ALGORITHM:**

Step 1: A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

Step 2: The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Step 3: Start by putting any one of the graph's vertices on top of a stack.

Step 4: Take the top item of the stack and add it to the visited list.

Step 5: 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.

Step 6: Keep repeating steps 2 and 3 until the stack is empty.

**PROGRAM:**

```
#include <iostream>
#include <list>
using namespace std;
class DFSGraph
{
int V;
list<int> *adjList;
```

```

void DFS_util(int v, bool visited[]); public:
DFSGraph(int V)
{
    this->V = V;
    adjList = new list<int>[V];
}
void addEdge(int v, int w){
adjList[v].push_back(w);
}

void DFS();
};
void DFSGraph::DFS_util(int v, bool visited[])
{
    visited[v] = true;
    cout << v << " ";
    list<int>::iterator i;
    for(i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if(!visited[*i])
            DFS_util(*i, visited);
}
void DFSGraph::DFS()
{
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFS_util(i, visited);
}
int main()
{
    DFSGraph gdfs(5);
    gdfs.addEdge(0, 1);
    gdfs.addEdge(0, 2);
    gdfs.addEdge(0, 3);
    gdfs.addEdge(1, 2);
    gdfs.addEdge(2, 4);
    gdfs.addEdge(3, 3);
    gdfs.addEdge(4, 4);
    cout << "Depth-first traversal for the given graph:"<<endl;
    gdfs.DFS();
    return 0;
}

```

**OUTPUT:**

```
Depth-first traversal for the given graph:  
0 1 2 4 3  
PS D:\c++>
```

**RESULT:**

Thus the above program for implementation of DFS algorithm as been executed successfully

## 8.B) BREADTH FIRST SEARCH

### AIM:

Write a program to find shortest path of graph traversals using BFS algorithm.

### ALGORITHM:

Step 1: A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

Step 2: The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Step 3: Start by putting any one of the graph's vertices at the back of a queue.

Step 4: Take the front item of the queue and add it to the visited list.

Step 5: 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.

Step 6: 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:

```
#include<iostream>
#include <list>

using namespace std;

class Graph
{
    int V;
```

```

        list<int> *adj;
public:
    Graph(int V);

    void addEdge(int v, int w);

    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);
}

void Graph::BFS(int s)
{
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    list<int> queue;

    visited[s] = true;
    queue.push_back(s);

    list<int>::iterator i;

    while(!queue.empty())
    {
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])

```



```

        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

int main()
{
    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 :

```

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
PS D:\c++>

```

### RESULT:

Thus the above program for implementation of BFS algorithm as been executed successfully

**EXERCISE NO: 9**

**DATE:**

## **IMPLEMENTATION OF SPANNING TREE**

### **9.A) KRUSKAL ALGORITHM IMPLEMENTATION**

**AIM:**

Write a program to find the minimum spanning tree using Kruskal algorithm.

**ALGORITHM:**

Step 1: Sort all the edges from low weight to high

Step 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.

Step 3: Keep adding edges until we reach all vertices.

**PROGRAM:**

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

#define edge pair<int, int>

class Graph {
private:
    vector<pair<int, edge>> G;
    vector<pair<int, edge>> T;
    int *parent;
    int V;
public:
    Graph(int V);
    void AddWeightedEdge(int u, int v, int w);
    int find_set(int i);
    void union_set(int u, int v);
    void kruskal();
};
```

```

    void print();
};
Graph::Graph(int V) {
    parent = new int[V];

    for (int i = 0; i < V; i++)
        parent[i] = i;

    G.clear();
    T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
    G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {

    if (i == parent[i])
        return i;
    else

        return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
    parent[u] = parent[v];
}
void Graph::kruskal() {
    int i, uRep, vRep;
    sort(G.begin(), G.end()); // increasing weight
    for (i = 0; i < G.size(); i++) {
        uRep = find_set(G[i].second.first);
        vRep = find_set(G[i].second.second);
        if (uRep != vRep) {
            T.push_back(G[i]); // add to tree
            union_set(uRep, vRep);
        }
    }
}
void Graph::print() {
    cout << "Edge : "
    << " Weight" << endl;
    for (int i = 0; i < T.size(); i++) {
        cout << T[i].second.first << " - " << T[i].second.second << " : "
        << T[i].first;
        cout << endl;
    }
}

```

```

int main() {
    Graph g(6);
    g.AddWeightedEdge(0, 1, 4);
    g.AddWeightedEdge(0, 2, 4);
    g.AddWeightedEdge(1, 2, 2);
    g.AddWeightedEdge(1, 0, 4);
    g.AddWeightedEdge(2, 0, 4);
    g.AddWeightedEdge(2, 1, 2);
    g.AddWeightedEdge(2, 3, 3);
    g.AddWeightedEdge(2, 5, 2);
    g.AddWeightedEdge(2, 4, 4);
    g.AddWeightedEdge(3, 2, 3);
    g.AddWeightedEdge(3, 4, 3);
    g.AddWeightedEdge(4, 2, 4);
    g.AddWeightedEdge(4, 3, 3);
    g.AddWeightedEdge(5, 2, 2);
    g.AddWeightedEdge(5, 4, 3);
    g.kruskal();
    g.print();
    return 0;
}

```

### OUTPUT:

```

Edge : weight
1 - 2 : 2
2 - 5 : 2
2 - 3 : 3
3 - 4 : 3
0 - 1 : 4
PS D:\c++> 

```

### RESULT:

Thus the above program for implementation of Kruskal algorithm as been executed successfully.

## 9.B) PRIM'S ALGORITHM IMPLEMENTATION

### AIM:

Write a program to find the minimum spanning tree using Prim's algorithm.

### ALGORITHM:

Step 1: We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

Step 2: Initialize the minimum spanning tree with a vertex chosen at random.

Step 3: Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree

Step 4: Keep repeating step 2 until we get a minimum spanning tree

### PROGRAM:

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

#define INF 9999999

#define V 5

int G[V][V] = {
    {0, 9, 75, 0, 0},
    {9, 0, 95, 19, 42},
    {75, 95, 0, 51, 66},
    {0, 19, 51, 0, 31},
    {0, 42, 66, 31, 0}};

int main() {
    int no_edge;

    int selected[V];
```

```

memset(selected, false, sizeof(selected));

no_edge = 0;

selected[0] = true;

int x;
int y;

cout << "Edge"
<< " : "
<< "Weight";
cout << endl;
while (no_edge < V - 1) {

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
        if (selected[i]) {
            for (int j = 0; j < V; j++) {
                if (!selected[j] && G[i][j]) {
                    if (min > G[i][j]) {
                        min = G[i][j];
                        x = i;
                        y = j;
                    }
                }
            }
        }
    }
    cout << x << " - " << y << " : " << G[x][y];
    cout << endl;
    selected[y] = true;
    no_edge++;
}

return 0;
}

```

**OUTPUT:**

```
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51
PS D:\c++>
```

**RESULT:**

Thus the above program for implementation of Prim's algorithm as been executed successfully.

**EXERCISE NO: 10**

**DATE:**

**IMPLEMENTATION OF DIJKSTRA AND BELLMAN FORD  
ALGORITHM**

**10.A) IMPLEMENTATION OF DIJKSTRA ALGORITHM**

**AIM:**

Write a program to find the shortest path using Dijkstra Algorithm.

**ALGORITHM:**

Step 1: First of all, we will mark all vertex as unvisited vertex

Step 2: Then, we will mark the source vertex as 0 and all other vertices as infinity

Step 3: Consider source vertex as current vertex

Step 4: Calculate the path length of all the neighboring vertex from the current vertex by adding the weight of the edge in the current vertex

Step 5: Now, if the new path length is smaller than the previous path length then replace it otherwise ignore it

Step 6: Mark the current vertex as visited after visiting the neighbor vertex of the current vertex

Step 7: Select the vertex with the smallest path length as the new current vertex and go back to step 4.

Step 8: Repeat this process until all the vertex are marked as visited.

**PROGRAM:**

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

int miniDist(int distance[], bool Tset[])
{
```



```

int minimum=INT_MAX,ind;

for(int k=0;k<6;k++)
{
    if(Tset[k]==false && distance[k]<=minimum)
    {
        minimum=distance[k];
        ind=k;
    }
}
return ind;
}

void DijkstraAlgo(int graph[6][6],int src)
{
    int distance[6];
    bool Tset[6];

    for(int k = 0; k<6; k++)
    {
        distance[k] = INT_MAX;
        Tset[k] = false;
    }

    distance[src] = 0;

    for(int k = 0; k<6; k++)
    {
        int m=miniDist(distance,Tset);
        Tset[m]=true;
        for(int k = 0; k<6; k++)
        {
            if(!Tset[k] && graph[m][k] && distance[m]!=INT_MAX &&
distance[m]+graph[m][k]<distance[k])
                distance[k]=distance[m]+graph[m][k];
        }
    }
    cout<<"Vertex\t\tDistance from source vertex"<<endl;
    for(int k = 0; k<6; k++)
    {
        char str=65+k;
        cout<<str<<"\t\t"<<distance[k]<<endl;
    }
}

```

```

int main()
{
    int graph[6][6]={
        {0, 1, 2, 0, 0, 0},
        {1, 0, 0, 5, 1, 0},
        {2, 0, 0, 2, 3, 0},
        {0, 5, 2, 0, 2, 2},
        {0, 1, 3, 2, 0, 1},
        {0, 0, 0, 2, 1, 0}};
    DijkstraAlgo(graph,0);
    return 0;
}

```

### OUTPUT:

Vertex	Distance from source vertex
A	0
B	1
C	2
D	4
E	2
F	3
PS D:\>	

### RESULT:

Thus the above program for implementation of Dijkstra algorithm as been executed successfully.

## 10.B) IMPLEMENTATION OF BELLMAN FORD

### AIM:

Write a program to find the shortest path using Bellman Ford Algorithm.

### ALGORITHM:

Step 1: This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array `dist[]` of size  $|V|$  with all values as infinite except `dist[src]` where `src` is source vertex.

Step 2: This step calculates shortest distances. Do following  $|V|-1$  times where  $|V|$  is the number of vertices in given graph.

.....a) Do following for each edge `u-v`

.....If `dist[v] > dist[u] + weight of edge uv`, then update `dist[v]`

.....`dist[v] = dist[u] + weight of edge uv`

Step 3: This step reports if there is a negative weight cycle in graph. Do following for each edge `u-v`

.....If `dist[v] > dist[u] + weight of edge uv`, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

### PROGRAM:

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
void BellmanFord(int graph[][3], int V, int E,  
                 int src)
```

```
{
```

```
    int dis[V];
```

```
    for (int i = 0; i < V; i++)
```

```
        dis[i] = INT_MAX;
```

```
    dis[src] = 0;
```

```
    for (int i = 0; i < V - 1; i++) {
```

```
        for (int j = 0; j < E; j++) {
```

```
            if (dis[graph[j][0]] != INT_MAX && dis[graph[j][0]] + graph[j][2] <
```

```

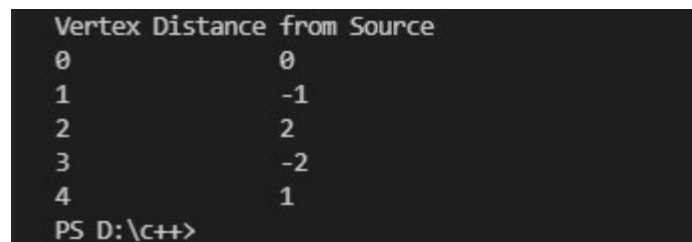
                                dis[graph[j][1]])
                                dis[graph[j][1]] =
                                dis[graph[j][0]] + graph[j][2];
                                }
                                }
                                for (int i = 0; i < E; i++) {
                                    int x = graph[i][0];
                                    int y = graph[i][1];
                                    int weight = graph[i][2];
                                    if (dis[x] != INT_MAX &&
                                        dis[x] + weight < dis[y])
                                        cout << "Graph contains negative"
                                            " weight cycle"
                                            << endl;
                                    }
                                    cout << "Vertex Distance from Source" << endl;
                                    for (int i = 0; i < V; i++)
                                        cout << i << "\t\t" << dis[i] << endl;
                                }
                                int main()
                                {
                                    int V = 5;
                                    int E = 8;

                                    int graph[][3] = { { 0, 1, -1 }, { 0, 2, 4 },
                                                            { 1, 2, 3 }, { 1, 3, 2 },
                                                            { 1, 4, 2 }, { 3, 2, 5 },
                                                            { 3, 1, 1 }, { 4, 3, -3 } };

                                    BellmanFord(graph, V, E, 0);
                                    return 0;
                                }

```

## OUTPUT:



```

Vertex Distance from Source
0          0
1         -1
2          2
3         -2
4          1
PS D:\c++>

```

## RESULT:

Thus the above program for implementation of Bellman Ford algorithm as been executed successfully.

**EXERCISE NO: 11****DATE:****IMPLEMENTATION OF MATRIX CHAIN MULTIPLICATION****AIM:**

Write a program to perform the operations of Matrix Chain Multiplication.

**ALGORITHM:**

Step 1:  $m[1,1]$  tells us about the operation of multiplying matrix A with itself which will be 0. So fill all the  $m[i,i]$  as 0.

Step 2:  $m[1,2]$  We are multiplying two matrices A and B. Operations will be  $5*4*6=120$ .

Step 3:  $m[2,3]$  We are multiplying B and C now. Operations will be  $4*6*2=48$ ,  $m[3,4]$  can be also filled in the same way

Step 4: Now,  $m[1,3]$ - We are considering three matrices A, B, and C. Now there are two ways to do this.

➤ **A(BC)**

$$m[1,1]+m[2,3]+5*4*2=0+48+40=88$$

➤ **(AB)C**

$$m[1,2]+m[3,3]+5*6*2=120+0+60=180$$

✓ We select the minimum from above and fill it in the table.

Step 5: Similarly,  $m[2,4]$  can be found out.

Step 6:  $m[1,4]$  can be deduced as:

$$m[1,4]=\min(m[1,1]+m[2,4]+5*4*7, m[1,2]+m[3,4]+5*6*7, m[1,3]+m[4,4]+5*2*7)$$

$$m[1,4]=\min(0+104+140, 120+84+210, 88+0+70)$$

$$m[1,4]=158$$

Step 7: So, the formula which we deduced from our observations above is:

$$m[i,j]=\min(m[i,k]+m[k+1,j]+d(i-1)*d(k)*d(j)) \text{ (d represents order of matrix)}$$

**PROGRAM:**

```
#include<bits/stdc++.h>
using namespace std;
```

```

int MatrixChainOrder(int p[], int n)
{
    int m[n][n];

    int i, j, k, L, q;

    for (i=1; i<n; i++)
        m[i][i] = 0;

    for (L=2; L<n; L++) //filling half table only
    {
        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 = 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];
}

int main()
{
    int arr[] = {5,4,6,2,7};
    int size = 5;
    cout<<MatrixChainOrder(arr,size)<<" operations";

}

```

## OUTPUT:

```

158 operations
PS D:\c++> 

```

## RESULT:

Thus the above program for implementation of Matrix Chain Multiplication as been executed successfully.

**EXERCISE NO: 12**

**DATE:**

**IMPLEMENTATION OF ACTIVITY SELECTION AND HUFFMAN CODING**

**12.A) ACTIVITY SELECTION IMPLEMENTATION**

**AIM:**

Write a program to perform Activity Selection Algorithm.

**ALGORITHM:**

Step 1: First, we need to sort the activities in ascending order according to their finishing time.

Step 2: Then, select the first activity from the sorted array and print it.

Step 3: Then, do following for remaining activities in the sorted array.

Step 4: Check, if the starting time of this activity is greater than or equal to the finishing time of previously selected activity then select this activity and print it.

**PROGRAM:**

```
#include<bits/stdc++.h>
using namespace std;
bool comp(pair<int,int>i,pair<int,int>j){
    return i.second<j.second;
}
int main() {
    int n;
    cout<<"enter the number of elements:"<<endl;
    cin>>n;
    map<pair<int,int>,int>m;
    vector<pair<int,int>>vec(n);
    cout<<" Enter the starting time of activity:"<<endl;
    for(int i=0;i<n;i++){
        cin>>vec[i].first;
    }
    cout<<"Enter the finishing time of activity:"<<endl;
    for(int i=0;i<n;i++){
```

```

        cin>>vec[i].second;
    }
    for(int i=0;i<n;i++){
        m[vec[i]]=i;
    }
    sort(vec.begin(),vec.end(),comp);
    vector<int>v;
    vector<int>::iterator i;
    v.push_back(m[vec[0]]);
    pair<int,int>current=vec[0];
    for(int j=1;j<n;j++){
        if(vec[j].first>current.second){
            v.push_back(m[vec[j]]);
            current=vec[j];
        }
    }
    cout<<"Order in which the activity take place"<<endl;
    for(i=v.begin();i!=v.end();i++){
        cout<<*i+1<<" ";
    }
    cout<<endl;
    return 0;
}

```

#### OUTPUT:

```

enter the number of elements:
6
Enter the starting time of activity:
1
3
0
5
8
5
Enter the finishing time of activity:
2
4
6
7
9
9
Order in which the activity take place
1 2 4 5
PS D:\C++>

```

#### RESULT:

Thus the above program for implementation of Activity Selection as been executed successfully.



## 12.B) HUFFMAN CODING IMPLEMENTATION

### AIM:

Write a program to perform Huffman Coding Algorithm.

### ALGORITHM:

Step 1: Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

Step 2: Extract two nodes with the minimum frequency from the min heap.

Step 3: Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

Step 4: Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

### PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;

struct MinHeapNode {
    char data;

    unsigned freq;

    MinHeapNode *left, *right;

    MinHeapNode(char data, unsigned freq)
    {
        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

struct compare {
```

```

        bool operator()(MinHeapNode* l, MinHeapNode* r)
        {
            return (l->freq > r->freq);
        }
};

void printCodes(struct MinHeapNode* root, string str)
{
    if (!root)
        return;

    if (root->data != '$')
        cout << root->data << ": " << str << "\n";

    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

void HuffmanCodes(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

    while (minHeap.size() != 1) {

        left = minHeap.top();
        minHeap.pop();

        right = minHeap.top();
        minHeap.pop();

        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        minHeap.push(top);
    }
}

```

```

    }

    printCodes(minHeap.top(), "");
}

int main()
{
    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}

```

## OUTPUT:

```

f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
PS D:\c++>

```

## RESULT:

Thus the above program for implementation of Huffman Coding as been executed successfully.