ADSA_12212070

Advanced Data Structures

```
Name:    Debatreya Das
Roll No: 12212070
Class:   CS-A4
```

Experiments

Quick Sort

CODE

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

int partition(int left, int right, vector<double> &arr){
    int pivotVal = arr[left];
    int i = left+1, j = right;
    while(i<=j){
        // Move i ->
        while(i <= right and arr[i] < pivotVal){
            i++;
        }
        // Move j <-
        while(j > left and arr[j] >= pivotVal){
            j--;
```

```
        }
        // Where the i and j stop (swap if i < j)
        if(i<j){
            // swap(arr[i], arr[j]);
            double temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // After the loop stops replace arr[left] with arr[j] and return j;
    // swap(arr[left], arr[j]);
    double temp = arr[left];
    arr[left] = arr[j];
    arr[j] = temp;

    // Print the Array
    cout<<"After Partition: ";
    for(double f: arr){
        cout<<f<<" ";
    }
    cout<<endl;
    return j; // pivot = j;
}

void quick_sort(int left, int right, vector<double> &arr){
    // If Left pointer has crossed the Right pointer: RETURN
    cout<<"Left: "<<left<<" Right: "<<right<<endl;
    if(left>=right){
        return;
    }
    // Else: FIND PIVOT
    int pivot = partition(left, right, arr);
    cout<<"\tPivot: "<<pivot<<endl;
    // Now only pivot is at its correct position.
    // QUICKSORT the two parts separately.
    quick_sort(left, pivot-1, arr);
    cout<<"After Left Part: ";
    for(double f: arr){
        cout<<f<<" ";
    }
    cout<<endl;
    quick_sort(pivot+1, right, arr);
    cout<<"After Right Part: ";
    for(double f: arr){
        cout<<f<<" ";
    }
    cout<<endl;
}

int main(){
    int n;
    cout<<"Enter the size of the Array: ";
    cin>>n;
    vector<double> arr(n, 0);
    cout<<"Enter the elements of the Array: ";
    for(int i=0; i<n; i++){
```

```
        cin>>arr[i];
    }
    // Quick Sort
    quick_sort(0, arr.size()-1, arr);
    // Print the Sorted Array
    cout<<"Sorted Array: ";
    for(double f: arr){
        cout<<f<<" ";
    }
    cout<<endl;
    return 0;
}

// Testcase:
// 8
// 12.4 12.5 19.9 10.67 15.8 23.9 90 51.34
```

OUTPUT (with all in-between steps)

```
Enter the size of the Array: 8
Enter the elements of the Array: 12.4 12.5 19.9 10.67 15.8 23.9 90 51.34
Left: 0 Right: 7
After Partition: 10.67 12.4 19.9 12.5 15.8 23.9 90 51.34
        Pivot: 1
Left: 0 Right: 0
After Left Part: 10.67 12.4 19.9 12.5 15.8 23.9 90 51.34
Left: 2 Right: 7
After Partition: 10.67 12.4 15.8 12.5 19.9 23.9 90 51.34
        Pivot: 4
Left: 2 Right: 3
After Partition: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
        Pivot: 3
Left: 2 Right: 2
After Left Part: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
Left: 4 Right: 3
After Right Part: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
After Left Part: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
Left: 5 Right: 7
After Partition: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
        Pivot: 5
Left: 5 Right: 4
After Left Part: 10.67 12.4 12.5 15.8 19.9 23.9 90 51.34
Left: 6 Right: 7
After Partition: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
        Pivot: 7
Left: 6 Right: 6
After Left Part: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
Left: 8 Right: 7
After Right Part: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
After Right Part: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
After Right Part: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
After Right Part: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90

Sorted Array: 10.67 12.4 12.5 15.8 19.9 23.9 51.34 90
```

COMPLEXITY

- **Time Complexity**:
  - Best and Average Case: $(O(nlogn))$
  - Worst Case: $(O(n^2))$
- **Space Complexity**:
  - Average Case: $(O(logn))$
  - Worst Case: $(O(n))$

B-Tree

CODE

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

//Node protoType Functions
template<class T,int Order>
struct Node
{
    //Node *Parent;
    int NumbersOfKeys;//number of the actual keys
    int order;
    int position=-1;//to allocate value in the appropriate place
    T* keys=new T[order];
    Node** childs=new Node*[order];

    Node (int order);
    int Insert (T value);
    Node* split (Node* node, T* value);
    void Print () ;
    void PrintUtil (int height,bool checkParent);
    int getHeight () ;
    ~Node ();
};
/////////////////////////////////////////////////////////////

//Node implementation
template <class T,int Order>
Node<T,Order>::Node (int order)
{
    this->order = order;
    this->NumbersOfKeys = 0;
}



template <class T,int Order>
int Node<T,Order>::Insert (T value)
{

    //if the node is leaf
    if (this->childs[0] == NULL)
    {
```

```
        this->keys[++this->position] = value;
        ++this->NumbersOfKeys;
        //arrange the keys array after put new value in node
        for(int i=this->position; i>0 ; i--)
        {
            if (this->keys[i] < this->keys[i-1]) std::swap(this->keys[i],this->keys[i-
1]);
        }
    }
    //if the node is not leaf
    else
    {
        //count to get place of child to put the value in it
        int i=0;
        for(; i<this->NumbersOfKeys>0 && value > this->keys[i];)
        {
            i++;
        }
        //Check if the child is full to split it
        int check=this->childs[i]->Insert(value);
        //if node full
        if(check)
        {
            T mid;
            int TEMP = i;
            Node<T,Order> *newNode = split(this->childs[i], &mid); //Splitted Node to
store the values and child that greater than the midValue
            //allocate midValue in correct place
            for(; i<this->NumbersOfKeys>0 && mid > this->keys[i];)
            {
                i++;
            }

            for (int j = this->NumbersOfKeys; j > i ; j--) this->keys[j] = this->keys[j
- 1];
            this->keys[i] = mid;

            ++this->NumbersOfKeys;
            ++this->position;

            //allocate newNode Splitted in the correct place
            int k;
            for (k = this->NumbersOfKeys; k > TEMP + 1; k--)
                this->childs[k] = this->childs[k - 1];
            this->childs[k] = newNode;
        }

    }
    if(this->NumbersOfKeys == this->order) return 1;//to split it
    else return 0;
}

template <class T,int Order>
Node<T,Order>* Node<T,Order>::split (Node *node, T *med) //mid to store value of mid and
use it in insert func
{
```

```
    int NumberOfKeys = node->NumbersOfKeys;
    Node<T,Order> *newNode = new Node<T,Order>(order);
    //Node<T,Order> *newParentNode = new Node<T,Order>(order);
    int midValue = NumberOfKeys / 2;
    *med = node->keys[midValue];
    int i;
    //take the values after mid value
    for (i = midValue + 1; i < NumberOfKeys; ++i)
    {
        newNode->keys[++newNode->position] = node->keys[i];
        newNode->childs[newNode->position] = node->childs[i];
        ++newNode->NumbersOfKeys;
        --node->position;
        --node->NumbersOfKeys;
        node->childs[i] = NULL;
    }
    newNode->childs[newNode->position+1] = node->childs[i];
    node->childs[i] = NULL;

    --node->NumbersOfKeys; //because we take mid value...
    --node->position;
    return newNode;
}


template <class T,int Order>
void Node<T,Order>::Print ()
{
    int height = this->getHeight(); //number of levels  -> log (n)
    for (int i = 1; i <= height; ++i) //50 levels maximum
    {
        //O(n)
        if(i==1)PrintUtil(i,true);
        else PrintUtil(i,false);
        cout<<endl;
    }
    cout<<endl;
}

template <class T,int Order>
void Node<T,Order>::PrintUtil (int height,bool checkRoot)
{
    //to print all values in the level
    if (height==1 || checkRoot)
    {
        for (int i = 0; i < this->NumbersOfKeys; i++){
             if(i==0) cout << "|";
             cout<< this->keys[i];
             if(i!=this->NumbersOfKeys-1) cout<<"|";
             if(i==this->NumbersOfKeys-1) cout << "|"<<" ";
        }

    }

    else
    {
```

```cpp
        for (int i = 0; i <= this->NumbersOfKeys; i++){
            this->childs[i]->PrintUtil(height-1,false);
            //cout<<endl<<" ";
        }

    }

}

template <class T,int Order>
int Node<T,Order>::getHeight ()
{
    int COUNT=1;
    Node<T,Order>* Current=this;//current point to root
    while(true){
            //is leaf
        if(Current->childs[0] == NULL){
            return COUNT;
        }
        Current=Current->childs[0];
        COUNT++;
    }
}

//Deallocation
template <class T,int Order>
Node<T,Order>::~Node ()
{
    delete[]keys;
    for (int i = 0; i <= this->NumbersOfKeys; ++i)
        delete this->childs[i];
}
//////////////////////////////////////////////////////////////


//BTree protoType Function
template <class T,int Order>
class BTree
{
private:
    Node<T,Order> *Root;
    int order;
    int count=0;//to count number of elements

public:
    BTree ();
    void Insert (T value);
    void Print () const;
    ~BTree ();
};
//////////////////////////////////////////////////////////////

//BTree implementation
template <class T,int Order>
BTree<T,Order>::BTree()
{
```

```
    this->order = Order;
    this->Root  = NULL;
}

template <class T,int Order>
void BTree<T,Order>::Insert (T value)
{
    count++;
    //if Tree is empty
    if (this->Root == NULL)
    {
        this->Root = new Node<T,Order>(this->order);
        this->Root->keys[++this->Root->position]=value;
        this->Root->NumbersOfKeys=1;
    }
    //if tree not empty
    else
    {
        int check=Root->Insert(value);
        if(check){
            T mid;
            Node<T,Order> *splittedNode = this->Root->split(this->Root, &mid);
            Node<T,Order> *newNode = new Node<T,Order>(this->order);
            newNode->keys[++newNode->position]=mid;
            newNode->NumbersOfKeys=1;
            newNode->childs[0] = Root;
            newNode->childs[1] = splittedNode;
            this->Root = newNode;
        }
    }
}


template <class T,int Order>
void BTree<T,Order>::Print () const
{
    if (Root != NULL)
        Root->Print();
    else cout<<"The B-Tree is Empty"<<endl;
}


template <class T,int Order>
BTree<T,Order>::~BTree ()
{
    delete Root;
}


////////////////////////////////////////////////////////////
int main ()
{

    // Construct a BTree of order 3, which stores int data
    cout<<"BTree of order 3, which stores int data"<<endl;
    BTree<int,3> t1;
```

```
    vector<int> v = {1,5,0,4,3,2};
    for (int i = 0; i < v.size(); i++){
        cout<<"Inserting "<<v[i]<<endl;
        t1.Insert(v[i]);
    }
    t1.Print();
    cout<<endl;

    cout<<"BTree of order 5, which stores char data"<<endl;
    BTree<char,5> t;
    vector<char> v2 =
{'G','I','B','J','C','A','K','E','D','S','T','R','L','F','H','M','N','P','Q'};
    for (int i = 0; i < v2.size(); i++){
        cout<<"Inserting "<<v2[i]<<endl;
        t.Insert(v2[i]);
    }
    t.Print();
    cout<<endl;
    return 0;
}
```

OUTPUT

```
BTree of order 3, which stores int data
Inserting 1
Inserting 5
Inserting 0
Inserting 4
Inserting 3
Inserting 2
|1|4|
|0| |2|3| |5|


BTree of order 5, which stores char data
Inserting G
Inserting I
Inserting B
Inserting J
Inserting C
Inserting A
Inserting K
Inserting E
Inserting D
Inserting S
Inserting T
Inserting R
Inserting L
Inserting F
Inserting H
Inserting M
Inserting N
Inserting P
Inserting Q
|K|
|C|G| |N|R|
```

```
|A|B| |D|E|F| |H|I|J| |L|M| |P|Q| |S|T|
```

AVL Tree

CODE

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

class TreeNode {
public:
    int data;
    TreeNode *left, *right;

    TreeNode(int x = -1) {
        this->data = x;
        this->left = NULL;
        this->right = NULL;
    }
};

class BST {
public:
    TreeNode* root;

    BST(int data = -1) {
        this->root = (data == -1) ? NULL : new TreeNode(data);
    }

    TreeNode* insert(TreeNode* root, int data) {
        if (root == NULL) {
            return new TreeNode(data);
        }
        if (data < root->data) {
            root->left = insert(root->left, data);
        } else if (data > root->data) {
            root->right = insert(root->right, data);
        } else {
            // Value already exists in BST
            cout << "BST already has this value" << endl;
            return root;
        }
        return root;
    }

    void BFS() {
        if (this->root == NULL) {
            cout << "Tree is empty" << endl;
            return;
        }
        queue<TreeNode*> q;
        q.push(this->root);
        q.push(NULL);
```

```cpp
        while (!q.empty()) {
            TreeNode *temp = q.front();
            q.pop();
            if (temp == NULL) {
                cout << endl;
                if (!q.empty()) q.push(NULL);
            } else {
                cout << temp->data << " ";
                if (temp->left != NULL) q.push(temp->left);
                if (temp->right != NULL) q.push(temp->right);
            }
        }
    }

    TreeNode* search(int val) {
        TreeNode *temp = this->root;
        while (temp != NULL) {
            if (temp->data == val) return temp;
            if (temp->data > val) {
                temp = temp->left;
            } else {
                temp = temp->right;
            }
        }
        return NULL;
    }

    TreeNode* inorderSuccessor(TreeNode* node) {
        if (node == NULL) return NULL;
        node = node->right;
        while (node && node->left != NULL) {
            node = node->left;
        }
        return node;
    }

    TreeNode* deleteNode(TreeNode* root, int val) {
        if (root == NULL) {
            return root;
        }
        if (val < root->data) {
            root->left = deleteNode(root->left, val);
        } else if (val > root->data) {
            root->right = deleteNode(root->right, val);
        } else {
            if (root->left == NULL) {
                TreeNode* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == NULL) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            } else {
                TreeNode* temp = inorderSuccessor(root);
                root->data = temp->data;
```

```cpp
                root->right = deleteNode(root->right, temp->data);
            }
        }
        return root;
    }
};

class AVL : public BST {
public:
    AVL(int data = -1) {
        this->root = (data == -1) ? NULL : new TreeNode(data);
    }

    int height(TreeNode* root) {
        if (root == NULL) return 0;
        return 1 + max(height(root->left), height(root->right));
    }

    int balanceFactor(TreeNode* root) {
        if (root == NULL) return 0;
        return height(root->left) - height(root->right);
    }

    TreeNode* leftRotate(TreeNode* x) {
        TreeNode* y = x->right;
        TreeNode* T2 = y->left;
        y->left = x;
        x->right = T2;
        return y;
    }

    TreeNode* rightRotate(TreeNode* y) {
        TreeNode* x = y->left;
        TreeNode* T2 = x->right;
        x->right = y;
        y->left = T2;
        return x;
    }

    TreeNode* insert(TreeNode* root, int data) {
        if (root == NULL) return new TreeNode(data);

        if (data < root->data) {
            root->left = insert(root->left, data);
        } else if (data > root->data) {
            root->right = insert(root->right, data);
        } else {
            return root;
        }

        int bf = balanceFactor(root);

        // Left Left Case
        if (bf > 1 && data < root->left->data) {
            return rightRotate(root);
        }
```

```cpp
        // Right Right Case
        if (bf < -1 && data > root->right->data) {
            return leftRotate(root);
        }

        // Left Right Case
        if (bf > 1 && data > root->left->data) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        // Right Left Case
        if (bf < -1 && data < root->right->data) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }

    TreeNode* insert(int data) {
        this->root = insert(this->root, data);
        return this->root;
    }

    TreeNode* deleteNode(TreeNode* root, int val) {
        root = BST::deleteNode(root, val);

        if (root == NULL) return root;

        int bf = balanceFactor(root);

        // Left heavy
        if (bf > 1 && balanceFactor(root->left) >= 0) {
            return rightRotate(root);
        }
        if (bf > 1 && balanceFactor(root->left) < 0) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        // Right heavy
        if (bf < -1 && balanceFactor(root->right) <= 0) {
            return leftRotate(root);
        }
        if (bf < -1 && balanceFactor(root->right) > 0) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }

    TreeNode* deleteNode(int val) {
        this->root = deleteNode(this->root, val);
```

```
        return this->root;
    }
};


int main(){
    AVL avl(10);
    cout<<"Initial AVL"<<endl;
    avl.BFS();
    cout<<endl;
    vector<int> v = {1, 2, 3, 5, 6, 7, 8, 11, 12, 13, 15, 18, 20, 22};
    // Inserting elements in AVL
    for (int i = 0; i < v.size(); i++) {
        cout<<"Inserting "<<v[i]<<endl;
        avl.insert(v[i]);
        // Level Order Traversal of AVL
        avl.BFS();
        cout<<endl;
    }
    // Deleting elements from AVL
    for (int i = 0; i < v.size(); i++) {
        cout<<"Deleting "<<v[i]<<endl;
        avl.deleteNode(v[i]);
        // Level Order Traversal of AVL
        avl.BFS();
        cout<<endl;
    }
    return 0;
}
```

OUTPUT

```
Initial AVL
10

Inserting 1
10
1

Inserting 2
2
1 10

Inserting 3
2
1 10
3

Inserting 5
2
1 5
3 10

Inserting 6
5
2 10
```

```
1 3 6

Inserting 7
5
2 7
1 3 6 10

Inserting 8
5
2 7
1 3 6 10
8

Inserting 11
5
2 7
1 3 6 10
8 11

Inserting 12
5
2 10
1 3 7 11
6 8 12

Inserting 13
5
2 10
1 3 7 12
6 8 11 13

Inserting 15
10
5 12
2 7 11 13
1 3 6 8 15

Inserting 18
10
5 12
2 7 11 15
1 3 6 8 13 18

Inserting 20
10
5 15
2 7 12 18
1 3 6 8 11 13 20

Inserting 22
10
5 15
2 7 12 20
1 3 6 8 11 13 18 22

Deleting 1
```

```
10
5 15
2 7 12 20
3 6 8 11 13 18 22

Deleting 2
10
5 15
3 7 12 20
6 8 11 13 18 22

Deleting 3
10
5 15
7 12 20
6 8 11 13 18 22

Deleting 5
10
7 15
6 8 12 20
11 13 18 22

Deleting 6
10
7 15
8 12 20
11 13 18 22

Deleting 7
15
10 20
8 12 18 22
11 13

Deleting 8
15
10 20
12 18 22
11 13

Deleting 11
15
10 20
12 18 22
13

Deleting 12
15
10 20
13 18 22

Deleting 13
15
10 20
18 22
```

```
Deleting 15
18
10 20
22

Deleting 18
20
10 22

Deleting 20
22
10

Deleting 22
10
```

Red-Black Tree

CODE

```cpp
#include <iostream>
#include <queue>

using namespace std;

// Node colors
enum Color { RED, BLACK };

class Node {
public:
    int data;
    Color color;
    Node* left, *right, *parent;

    Node(int data) {
        this->data = data;
        left = right = parent = nullptr;
        color = RED; // New nodes are always red
    }
};

class RedBlackTree {
private:
    Node* root;

    // Helper functions
    void rotateLeft(Node* &root, Node* &x) {
        Node* y = x->right;
        x->right = y->left;

        if (y->left != nullptr)
            y->left->parent = x;
```

```cpp
    y->parent = x->parent;

    if (x->parent == nullptr)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

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

void rotateRight(Node* &root, Node* &x) {
    Node* y = x->left;
    x->left = y->right;

    if (y->right != nullptr)
        y->right->parent = x;

    y->parent = x->parent;

    if (x->parent == nullptr)
        root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;

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

void fixInsert(Node* &root, Node* &pt) {
    Node* parent_pt = nullptr;
    Node* grand_parent_pt = nullptr;

    while ((pt != root) && (pt->color != BLACK) && (pt->parent->color == RED)) {
        parent_pt = pt->parent;
        grand_parent_pt = pt->parent->parent;

        // Parent is left child of grandparent
        if (parent_pt == grand_parent_pt->left) {
            Node* uncle_pt = grand_parent_pt->right;

            // Uncle is RED (Case 1)
            if (uncle_pt != nullptr && uncle_pt->color == RED) {
                grand_parent_pt->color = RED;
                parent_pt->color = BLACK;
                uncle_pt->color = BLACK;
                pt = grand_parent_pt;
            }
            else {
                // pt is right child of parent (Case 2)
                if (pt == parent_pt->right) {
                    rotateLeft(root, parent_pt);
```

```
                    pt = parent_pt;
                    parent_pt = pt->parent;
                }

                // pt is left child of parent (Case 3)
                rotateRight(root, grand_parent_pt);
                swap(parent_pt->color, grand_parent_pt->color);
                pt = parent_pt;
            }
        }
        // Parent is right child of grandparent
        else {
            Node* uncle_pt = grand_parent_pt->left;

            // Uncle is RED (Case 1)
            if (uncle_pt != nullptr && uncle_pt->color == RED) {
                grand_parent_pt->color = RED;
                parent_pt->color = BLACK;
                uncle_pt->color = BLACK;
                pt = grand_parent_pt;
            }
            else {
                // pt is left child of parent (Case 2)
                if (pt == parent_pt->left) {
                    rotateRight(root, parent_pt);
                    pt = parent_pt;
                    parent_pt = pt->parent;
                }

                // pt is right child of parent (Case 3)
                rotateLeft(root, grand_parent_pt);
                swap(parent_pt->color, grand_parent_pt->color);
                pt = parent_pt;
            }
        }
    }

    root->color = BLACK;
}

void levelOrderHelper(Node* root) {
    if (root == nullptr)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        while (size--) {
            Node* temp = q.front();
            cout << temp->data << "(" << (temp->color == RED ? "R" : "B") << ") ";
            q.pop();

            if (temp->left != nullptr)
                q.push(temp->left);
```

```
            if (temp->right != nullptr)
                q.push(temp->right);
        }
        cout << endl;  // Move to the next level
    }
}

void inorderHelper(Node* root) {
    if (root == nullptr)
        return;

    inorderHelper(root->left);
    cout << root->data << "(" << (root->color == RED ? "R" : "B") << ") ";
    inorderHelper(root->right);
}

void preorderHelper(Node* root) {
    if (root == nullptr)
        return;

    cout << root->data << "(" << (root->color == RED ? "R" : "B") << ") ";
    preorderHelper(root->left);
    preorderHelper(root->right);
}

void postorderHelper(Node* root) {
    if (root == nullptr)
        return;

    postorderHelper(root->left);
    postorderHelper(root->right);
    cout << root->data << "(" << (root->color == RED ? "R" : "B") << ") ";
}

Node* searchHelper(Node* root, int key) {
    if (root == nullptr || root->data == key)
        return root;

    if (key < root->data)
        return searchHelper(root->left, key);

    return searchHelper(root->right, key);
}

void deleteNodeHelper(Node* &root, Node* v) {
    Node* u = BSTreplace(v);
    bool uvBlack = ((u == nullptr || u->color == BLACK) && (v->color == BLACK));
    Node* parent = v->parent;

    if (u == nullptr) {
        // v is a leaf node
        if (v == root) {
            root = nullptr;
        } else {
            if (uvBlack) {
```

```
                fixDoubleBlack(root, v);
            } else {
                if (sibling(v) != nullptr)
                    sibling(v)->color = RED;
            }

            if (v == parent->left) {
                parent->left = nullptr;
            } else {
                parent->right = nullptr;
            }
        }
        delete v;
        return;
    }

    if (v->left == nullptr || v->right == nullptr) {
        // v has one child
        if (v == root) {
            v->data = u->data;
            v->left = v->right = nullptr;
            delete u;
        } else {
            if (v == parent->left) {
                parent->left = u;
            } else {
                parent->right = u;
            }
            delete v;
            u->parent = parent;
            if (uvBlack) {
                fixDoubleBlack(root, u);
            } else {
                u->color = BLACK;
            }
        }
        return;
    }

    // v has two children, swap data with successor and delete successor
    swapValues(u, v);
    deleteNodeHelper(root, u);
}

Node* successor(Node* x) {
    Node* current = x;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
}


Node* BSTreplace(Node* x) {
    if (x->left != nullptr && x->right != nullptr)
        return successor(x->right);
```

```
        if (x->left == nullptr && x->right == nullptr)
            return nullptr;
        return (x->left != nullptr) ? x->left : x->right;
    }

    void fixDoubleBlack(Node* &root, Node* x) {
        if (x == root)
            return;

        Node* siblingNode = sibling(x);
        Node* parent = x->parent;

        if (siblingNode == nullptr) {
            fixDoubleBlack(root, parent);
        } else {
            if (siblingNode->color == RED) {
                parent->color = RED;
                siblingNode->color = BLACK;
                if (siblingNode == parent->left) {
                    rotateRight(root, parent);
                } else {
                    rotateLeft(root, parent);
                }
                fixDoubleBlack(root, x);
            } else {
                if (hasRedChild(siblingNode)) {
                    if (siblingNode->left != nullptr && siblingNode->left->color == RED)
{
                        if (siblingNode == parent->left) {
                            siblingNode->left->color = siblingNode->color;
                            siblingNode->color = parent->color;
                            rotateRight(root, parent);
                        } else {
                            siblingNode->left->color = parent->color;
                            rotateRight(root, siblingNode);
                            rotateLeft(root, parent);
                        }
                    } else {
                        if (siblingNode == parent->left) {
                            siblingNode->right->color = parent->color;
                            rotateLeft(root, siblingNode);
                            rotateRight(root, parent);
                        } else {
                            siblingNode->right->color = siblingNode->color;
                            siblingNode->color = parent->color;
                            rotateLeft(root, parent);
                        }
                    }
                    parent->color = BLACK;
                } else {
                    siblingNode->color = RED;
                    if (parent->color == BLACK) {
                        fixDoubleBlack(root, parent);
                    } else {
                        parent->color = BLACK;
                    }
```

```
                }
            }
        }
    }

    Node* sibling(Node* node) {
        if (node->parent == nullptr)
            return nullptr;
        if (node == node->parent->left)
            return node->parent->right;
        return node->parent->left;
    }

    bool hasRedChild(Node* node) {
        return (node->left != nullptr && node->left->color == RED) ||
               (node->right != nullptr && node->right->color == RED);
    }

    void swapValues(Node* u, Node* v) {
        int temp = u->data;
        u->data = v->data;
        v->data = temp;
    }

public:
    RedBlackTree() { root = nullptr; }

    // Insert function
    void insert(const int &data) {
        Node* pt = new Node(data);

        root = bstInsert(root, pt);

        fixInsert(root, pt);
    }

    // Utility function to insert in BST
    Node* bstInsert(Node* root, Node* pt) {
        if (root == nullptr)
            return pt;

        if (pt->data < root->data) {
            root->left = bstInsert(root->left, pt);
            root->left->parent = root;
        }
        else if (pt->data > root->data) {
            root->right = bstInsert(root->right, pt);
            root->right->parent = root;
        }

        return root;
    }

    // Search function
    Node* search(int key) {
        return searchHelper(root, key);
```

```
    }

    // Traversal functions
    void levelOrder() { levelOrderHelper(root); }
    void inorder() { inorderHelper(root); }
    void preorder() { preorderHelper(root); }
    void postorder() { postorderHelper(root); }

    // Delete function
    void deleteNode(int data) {
        Node* nodeToDelete = searchHelper(root, data);
        if (nodeToDelete == nullptr) {
            cout << "Node not found in the tree.\n";
            return;
        }
        deleteNodeHelper(root, nodeToDelete);
    }

};

int main(){
    RedBlackTree tree;
    vector<int> arr = {3, 7, 12, 15, 20, 25, 40, 45, 50, 60};

    // Inserting elements in the tree
    cout<<"Inserting elements in the tree:\n";
    for (int i = 0; i < arr.size(); i++){
        cout << "Inserting " << arr[i] << ":\n";
        tree.insert(arr[i]);
        tree.levelOrder();
    }

    // Deleting elements from the tree
    cout<<"Deleting elements from the tree:\n";
    for (int i = 0; i < arr.size(); i++){
        cout << "Deleting " << arr[i] << ":\n";
        tree.deleteNode(arr[i]);
        tree.levelOrder();
    }

    return 0;
}
```

OUTPUT

```
Inserting elements in the tree:

Inserting 3:
3(B)
Inserting 7:
3(B)
7(R)
Inserting 12:
7(B)
3(R) 12(R)
Inserting 15:
```

```
7(B)
3(B) 12(B)
15(R)
Inserting 20:
7(B)
3(B) 15(B)
12(R) 20(R)
Inserting 25:
7(B)
3(B) 15(R)
12(B) 20(B)
25(R)
Inserting 40:
7(B)
3(B) 15(R)
12(B) 25(B)
20(R) 40(R)
Inserting 45:
15(B)
7(R) 25(R)
3(B) 12(B) 20(B) 40(B)
45(R)
Inserting 50:
15(B)
7(R) 25(R)
3(B) 12(B) 20(B) 45(B)
40(R) 50(R)
Inserting 60:
15(B)
7(B) 25(B)
3(B) 12(B) 20(B) 45(R)
40(B) 50(B)
60(R)


Deleting elements from the tree:

Deleting 3:
25(B)
15(B) 45(B)
7(B) 20(B) 40(B) 50(B)
12(R) 60(R)
Deleting 7:
25(B)
15(B) 45(B)
12(B) 20(B) 40(B) 50(B)
60(R)
Deleting 12:
25(B)
15(B) 45(R)
20(R) 40(B) 50(B)
60(R)
Deleting 15:
25(B)
20(B) 45(R)
40(B) 50(B)
```

```
60(R)
Deleting 20:
45(B)
25(B) 50(B)
40(R) 60(R)
Deleting 25:
45(B)
40(B) 50(B)
60(R)
Deleting 40:
50(B)
45(B) 60(B)
Deleting 45:
50(B)
60(R)
Deleting 50:
60(B)
Deleting 60:
```

Fibonacci Heap

CODE

```
#include <bits/stdc++.h>
#define INF 987654321
using namespace std;
typedef long long lld;
typedef unsigned long long llu;

struct FibNode
{
    int key;
    bool marked;
    int degree;
    FibNode *b, *f, *p, *c;

    FibNode()
    {
        this -> key = 0;
        this -> marked = false;
        this -> degree = 0;
        this -> b = this -> f = this -> p = this -> c = NULL;
    }

    FibNode(int key)
    {
        this -> key = key;
        this -> marked = false;
        this -> degree = 0;
        this -> b = this -> f = this -> p = this -> c = NULL;
    }
};

class FibHeap
```

```
{
    FibNode *min;
    int N;

public:
    FibHeap();
    FibHeap(FibNode*);
    bool isEmpty();
    void insert(FibNode*);
    void merge(FibHeap*);
    FibNode* first();
    FibNode* extractMin();
    void decreaseKey(FibNode*, int);
    void Delete(FibNode*);
};

FibHeap::FibHeap()
{
    this -> min = NULL;
    this -> N = 0;
}

FibHeap::FibHeap(FibNode *n)
{
    this -> min = n;
    n -> b = n -> f = n;
    n -> p = n -> c = NULL;

    this -> N = 1;
}

bool FibHeap::isEmpty()
{
    return (this -> min == NULL);
}

void FibHeap::insert(FibNode *n)
{
    this -> merge(new FibHeap(n));
}

void FibHeap::merge(FibHeap *h)
{
    this -> N += h -> N;
    if (h -> isEmpty()) return;
    if (this -> isEmpty())
    {
        this -> min = h -> min;
        return;
    }
    FibNode *first1 = this -> min;
    FibNode *last1 = this -> min -> b;
    FibNode *first2 = h -> min;
    FibNode *last2 = h -> min -> b;
    first1 -> b = last2;
    last1 -> f = first2;
```

```
    first2 -> b = last1;
    last2 -> f = first1;
    if (h -> min -> key < this -> min -> key) this -> min = h -> min;
}

FibNode* FibHeap::first()
{
    return this -> min;
}

FibNode* FibHeap::extractMin()
{
    FibNode *ret = this -> min;
    this -> N = this -> N - 1;

    if (ret -> f == ret)
    {
        this -> min = NULL;
    }
    else
    {
        FibNode *prev = ret -> b;
        FibNode *next = ret -> f;
        prev -> f = next;
        next -> b = prev;
        this -> min = next; // Not necessarily a minimum. This is for assisting with the
merge w/ min's children.
    }

    if (ret -> c != NULL)
    {
        FibNode *firstChd = ret -> c;
        FibNode *currChd = firstChd;

        do
        {
            currChd -> p = NULL;
            currChd = currChd -> f;
        } while (currChd != firstChd);

        if (this -> isEmpty())
        {
            this -> min = firstChd;
        }
        else
        {
            FibNode *first1 = this -> min;
            FibNode *last1 = this -> min -> b;
            FibNode *first2 = firstChd;
            FibNode *last2 = firstChd -> b;
            first1 -> b = last2;
            last1 -> f = first2;
            first2 -> b = last1;
            last2 -> f = first1;
        }
    }
```

```
    if (this -> min != NULL)
    {
        int maxAuxSize = 5 * (((int)log2(this -> N + 1)) + 1);
        FibNode **aux = new FibNode*[maxAuxSize + 1];
        for (int i=0;i<=maxAuxSize;i++) aux[i] = NULL;
        int maxDegree = 0;
        FibNode *curr = this -> min;

        do
        {
            FibNode *next = curr -> f;
            int deg = curr -> degree;
            FibNode *P = curr;
            while (aux[deg] != NULL)
            {
                FibNode *Q = aux[deg];
                aux[deg] = NULL;

                if (P -> key > Q -> key)
                {
                    FibNode *tmp = P;
                    P = Q;
                    Q = tmp;
                }

                Q -> p = P;
                if (P -> c == NULL)
                {
                    P -> c = Q;
                    Q -> b = Q -> f = Q;
                }
                else
                {
                    FibNode *last = P -> c -> b;
                    last -> f = Q;
                    Q -> b = last;
                    P -> c -> b = Q;
                    Q -> f = P -> c;
                }

                deg++;
                P -> degree = deg;
            }
            aux[deg] = P;
            if (deg > maxDegree) maxDegree = deg;
            curr = next;
        } while (curr != this -> min);


        FibNode *previous = aux[maxDegree];
        this -> min = previous;
        for (int i=0;i<=maxDegree;i++)
        {
            if (aux[i] != NULL)
            {
```

```
                previous -> f = aux[i];
                aux[i] -> b = previous;
                if (aux[i] -> key < this -> min -> key) this -> min = aux[i];
                previous = aux[i];
            }
        }

    }

    return ret;
}

void FibHeap::decreaseKey(FibNode *n, int newKey)
{
    // Precondition: newKey < n -> key
    n -> key = newKey;

    FibNode *curr = n;
    if (curr -> p != NULL)
    {
        if (curr -> key < curr -> p -> key)
        {
            FibNode *parent = curr -> p;
            curr -> marked = false;

            curr -> p = NULL;
            if (curr -> f == curr) parent -> c = NULL;
            else
            {
                FibNode *prev = curr -> b;
                FibNode *next = curr -> f;
                prev -> f = next;
                next -> b = prev;
                if (parent -> c == curr) parent -> c = prev;
            }
            parent -> degree = parent -> degree - 1;

            FibNode *last = this -> min -> b;
            last -> f = curr;
            curr -> b = last;
            this -> min -> b = curr;
            curr -> f = this -> min;

            if (curr -> key < this -> min -> key) this -> min = curr;

            while (parent -> p != NULL && parent -> marked)
            {
                curr = parent;
                parent = curr -> p;
                curr -> marked = false;

                curr -> p = NULL;
                if (curr -> f == curr) parent -> c = NULL;
                else
                {
                    FibNode *prev = curr -> b;
```

```
                      FibNode *next = curr -> f;
                      prev -> f = next;
                      next -> b = prev;
                      if (parent -> c == curr) parent -> c = prev;
                  }
                  parent -> degree = parent -> degree - 1;

                  FibNode *last = this -> min -> b;
                  last -> f = curr;
                  curr -> b = last;
                  this -> min -> b = curr;
                  curr -> f = this -> min;

              }
              if (parent -> p != NULL) parent -> marked = true;
          }
      }
      else if (n -> key < this -> min -> key) this -> min = n;
}

void FibHeap::Delete(FibNode *n)
{
    this -> decreaseKey(n, -INF);
    this -> extractMin();
}

int main()
{
    FibHeap *fh = new FibHeap();

    FibNode *x = new FibNode(11);
    FibNode *y = new FibNode(5);

    fh -> insert(x);
    fh -> insert(y);
    fh -> insert(new FibNode(3));
    fh -> insert(new FibNode(8));
    fh -> insert(new FibNode(4));

    fh -> decreaseKey(x, 2);
    fh -> Delete(y);

    while (!fh -> isEmpty())
    {
        printf("%d ", fh -> extractMin() -> key);
    }
    printf("\n");
    return 0;
}

// Output: 2 3 4 8
```

Complexity:

- $O(1)$ for insert, first and merge

- $O(1)$ amortized for decreaseKey
- $(O(logN))$ amortized for extractMin and delete

Binomial Heap

CODE

```
#include <bits/stdc++.h>
#define INF 987654321
#define MAX_N 100002
using namespace std;
typedef long long lld;
typedef unsigned long long llu;

struct BinNode
{
    int key;
    int degree;
    BinNode *f, *p, *c;

    BinNode()
    {
        this->key = 0;
        this->degree = 0;
        this->f = this->p = this->c = NULL;
    }

    BinNode(int key)
    {
        this->key = key;
        this->degree = 0;
        this->f = this->p = this->c = NULL;
    }
};

class BinHeap
{
    BinNode *roots;
    BinNode *min;
    void linkTrees(BinNode *, BinNode *);
    BinNode *mergeRoots(BinHeap *, BinHeap *);

public:
    BinHeap();
    BinHeap(BinNode *);
    bool isEmpty();
    void insert(BinNode *);
    void merge(BinHeap *);
    BinNode *first();
    BinNode *extractMin();
    void decreaseKey(BinNode *, int);
    void Delete(BinNode *);
};

BinHeap::BinHeap()
{
```

```
    this->roots = NULL;
}

BinHeap::BinHeap(BinNode *x)
{
    this->roots = x;
}

bool BinHeap::isEmpty()
{
    return (this->roots == NULL);
}

void BinHeap::insert(BinNode *x)
{
    this->merge(new BinHeap(x));
}

void BinHeap::linkTrees(BinNode *y, BinNode *z)
{
    // Precondition: y -> key >= z -> key
    y->p = z;
    y->f = z->c;
    z->c = y;
    z->degree = z->degree + 1;
}

BinNode *BinHeap::mergeRoots(BinHeap *x, BinHeap *y)
{
    BinNode *ret = new BinNode();
    BinNode *end = ret;

    BinNode *L = x->roots;
    BinNode *R = y->roots;
    if (L == NULL)
        return R;
    if (R == NULL)
        return L;
    while (L != NULL || R != NULL)
    {
        if (L == NULL)
        {
            end->f = R;
            end = end->f;
            R = R->f;
        }
        else if (R == NULL)
        {
            end->f = L;
            end = end->f;
            L = L->f;
        }
        else
        {
            if (L->degree < R->degree)
            {
```

```
                end->f = L;
                end = end->f;
                L = L->f;
            }
            else
            {
                end->f = R;
                end = end->f;
                R = R->f;
            }
        }
    }
    return (ret->f);
}

void BinHeap::merge(BinHeap *bh)
{
    BinHeap *H = new BinHeap();
    H->roots = mergeRoots(this, bh);

    if (H->roots == NULL)
    {
        this->roots = NULL;
        this->min = NULL;
        return;
    }

    BinNode *prevX = NULL;
    BinNode *x = H->roots;
    BinNode *nextX = x->f;
    while (nextX != NULL)
    {
        if (x->degree != nextX->degree || (nextX->f != NULL && nextX->f->degree == x-
>degree))
        {
            prevX = x;
            x = nextX;
        }
        else if (x->key <= nextX->key)
        {
            x->f = nextX->f;
            linkTrees(nextX, x);
        }
        else
        {
            if (prevX == NULL)
                H->roots = nextX;
            else
                prevX->f = nextX;
            linkTrees(x, nextX);
            x = nextX;
        }
        nextX = x->f;
    }

    this->roots = H->roots;
```

```
    this->min = H->roots;
    BinNode *cur = this->roots;
    while (cur != NULL)
    {
        if (cur->key < this->min->key)
            this->min = cur;
        cur = cur->f;
    }
}

BinNode *BinHeap::first()
{
    return this->min;
}

BinNode *BinHeap::extractMin()
{
    BinNode *ret = this->first();

    // delete ret from the list of roots
    BinNode *prevX = NULL;
    BinNode *x = this->roots;
    while (x != ret)
    {
        prevX = x;
        x = x->f;
    }
    if (prevX == NULL)
        this->roots = x->f;
    else
        prevX->f = x->f;

    // reverse the list of ret's children
    BinNode *revChd = NULL;
    BinNode *cur = ret->c;
    while (cur != NULL)
    {
        BinNode *next = cur->f;
        cur->f = revChd;
        revChd = cur;
        cur = next;
    }

    // merge the two lists
    BinHeap *H = new BinHeap();
    H->roots = revChd;
    this->merge(H);

    return ret;
}

void BinHeap::decreaseKey(BinNode *x, int newKey)
{
    // Precondition: x -> key > newKey
    x->key = newKey;
    BinNode *y = x;
```

```
    BinNode *z = y->p;
    while (z != NULL && y->key < z->key)
    {
        // swap contents
        swap(y->key, z->key);

        y = z;
        z = y->p;
    }

    if (y->key < this->min->key)
        this->min = y;
}

void BinHeap::Delete(BinNode *x)
{
    decreaseKey(x, -INF);
    extractMin();
}

int main()
{
    BinHeap *bh = new BinHeap();

    BinNode *x = new BinNode(11);
    BinNode *y = new BinNode(5);

    bh->insert(x);
    bh->insert(y);
    bh->insert(new BinNode(3));
    bh->insert(new BinNode(8));
    bh->insert(new BinNode(4));

    bh->decreaseKey(x, 2);

    while (!bh->isEmpty())
    {
        printf("%d ", bh->extractMin()->key);
    }
    printf("\n");
    return 0;
}

// Output: 2 3 4 5 8
```

Complexity:
- $(O(1))$ for first
- $(O(logn))$ for insert, merge, extractMin, decreaseKey and delete

KMP

CODE

```
#include <iostream>
#include <vector>
```

```cpp
#include <string>
using namespace std;

// LPS (Longest Prefix Suffix) array
void computeLPSArray(const string& pattern, vector<int>& lps) {
    int length = 0; // length of the previous longest prefix suffix
    lps[0] = 0;     // lps[0] is always 0
    int i = 1;

    // Loop calculates lps[i] for i = 1 to pattern.size() - 1
    while (i < pattern.size()) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            // If mismatch after length matches
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    // Print the lps array
    cout << "LPS array: ";
    for (int i = 0; i < lps.size(); i++) {
        cout << lps[i] << " ";
    }
    cout << endl;
}

// KMP pattern matching
void KMPSearch(const string& text, const string& pattern) {
    int n = text.size();
    int m = pattern.size();

    // lps[] -> It will hold the longest prefix suffix values for pattern
    vector<int> lps(m);
    computeLPSArray(pattern, lps);

    cout<<"KMPSearch: "<<endl;

    int i = 0; // index for text[]
    int j = 0; // index for pattern[]

    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
```

```
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {

    string text, pattern;

    // Taking input of text and pattern from the user
    cout << "Enter the text: ";
    getline(cin, text);
    cout << "Enter the pattern: ";
    getline(cin, pattern);

    // KMP search
    KMPSearch(text, pattern);

    // Printing the index and character of the text for better understanding
    cout<<"Character and corresponding index of the text: "<<endl;
    for(int i=0; i<text.size(); i++){
        cout<<text[i]<<" "<<i<<endl;
    }
    cout<<endl;

    return 0;
}

// Testcase:
// bacbababaabcbababacaabcbababacababacababaca
// ababaca
```

OUTPUT

```
Enter the text: bacbababaabcbababacaabcbababacababacababaca
Enter the pattern: ababaca
LPS array: 0 0 1 2 3 0 1
KMPSearch:
Pattern found at index 13
Pattern found at index 24
Pattern found at index 30
Pattern found at index 36
Character and corresponding index of the text:
b 0
a 1
c 2
b 3
a 4
b 5
a 6
```

```
b 7
a 8
a 9
b 10
c 11
b 12
a 13
b 14
a 15
b 16
a 17
c 18
a 19
a 20
b 21
c 22
b 23
a 24
b 25
a 26
b 27
a 28
c 29
a 30
b 31
a 32
b 33
a 34
c 35
a 36
b 37
a 38
b 39
a 40
c 41
a 42
```

COMPLEXITY

- **Time Complexity**: $(O(n+m))$
- **Space Complexity**: $(O(m))$ due to LPS array.

Rabin-Karp

CODE

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

// Rabin-Karp algorithm for pattern matching
void rabinKarp(string text, string pattern, int q) {
    int d = 256; // Number of characters in the input alphabet
    int n = text.length();
    int m = pattern.length();
    int p = 0; // Hash value for pattern
```

```cpp
    int t = 0; // Hash value for text
    int h = 1;

    // The value of h would be "pow(d, m-1) % q"
    for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
    }

    // Calculate the hash value of the pattern and first window of text
    for (int i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    // Slide the pattern over text one by one
    for (int i = 0; i <= n - m; i++) {
        // Check the hash values of current window of text and pattern
        if (p == t) {
            // Check for characters one by one
            bool match = true;
            for (int j = 0; j < m; j++) {
                if (text[i + j] != pattern[j]) {
                    match = false;
                    break;
                }
            }

            // If the hash values and characters match
            if (match) {
                cout << "Pattern found at index " << i << endl;
            }
        }

        // Calculate hash value for next window of text
        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;

            // Convert negative value of t to positive
            if (t < 0) {
                t = (t + q);
            }
        }
    }
}

int main() {
    string text, pattern;
    int q;

    // Taking user input
    cout << "Enter the text: ";
    getline(cin, text);
    cout << "Enter the pattern to search: ";
    getline(cin, pattern);
    cout << "Enter the prime number (q): ";
    cin >> q;
```

```
    // Rabin-Karp algorithm
    rabinKarp(text, pattern, q);

    return 0;
}

// Test case:
// Enter the text: AABAACAADAABAABA
// Enter the pattern to search: AABA
// Enter the prime number (q): 101
```

OUTPUT

```
Enter the text: AABAACAADAABAABA
Enter the pattern to search: AABA
Enter the prime number (q): 101
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

COMPLEXITY

- **Time Complexity**:
  - Avgerage Case: $(O(n + m))$
  - Worst Case: $(O((n - m + 1) * m))$
- **Space Complexity**: $(O(1))$.

m-way Tree

CODE

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

struct mnode{
    vector<int> key;
    vector<mnode*> next;

    mnode(int m,int key){
        for(int i = 0;i<m;i++){
            this->next.push_back(NULL);
        }
        this->key.push_back(key);
    }
};


class mway{
    int m,h;
    mnode* root;
public:
    mway(int m){
        root = NULL;
```

```
        this->m = m;
    }

    mnode* insertion(int item,mnode* ptr){
        if(root == NULL){
            root = new mnode(m,item);
            return root;
        }
        if(ptr == NULL){
            ptr = new mnode(m,item);
            return ptr;
        }
        mnode* newmnode = new mnode(m,item);
        if(ptr->key.size() == m-1){
            for(int i =0 ;i< m-1;i++){
                if(item < ptr->key[i]){
                    ptr->next[i] = insertion(item,ptr->next[i]);
                    break;
                }
                else if(item > ptr->key[m-2]){
                    ptr->next[m-1] = insertion(item,ptr->next[m-1]);
                    break;
                }
            }
        }
        else{
            ptr->key.push_back(item);
            sort(ptr->key.begin(),ptr->key.end());
        }
        return ptr;
    }

    int findsuc(mnode* ptr){
        if(ptr->next[0]==NULL){
            return ptr->key[0];
        }
        else{
            return findsuc(ptr->next[0]);
        }
    }
    mnode* deletion(int item,mnode*ptr){
        if(root == NULL){
            cout<<"\nempty tree.";
            return root;
        }
        if(ptr == NULL){
            return ptr;
        }

        for(int i = 0 ; i < ptr->key.size() ; i++){
            if(ptr->key[i] == item){
                if(ptr->next[i+1] != NULL){
                    int suc = findsuc(ptr->next[i+1]);
                    ptr->key[i] = suc;
                    ptr->next[i+1] = deletion(suc,ptr->next[i+1]);
                    return ptr;
```

```
                }
                else{
                    if(i < ptr->key.size()-1){
                        for(int j = i;j < ptr->key.size()-1;j++){
                            ptr->key[j] = ptr->key[j+1];
                            ptr->next[j+1] = ptr->next[j+2];
                        }
                    }
                    ptr->key.pop_back();
                    ptr->next[ptr->key.size()] = NULL;
                    return ptr;
                }
            }
        }
        if(item < ptr->key[0]){
            ptr->next[0] = deletion(item,ptr->next[0]);
            return ptr;
        }
        for(int i =0;i<ptr->next.size();i++){
            if(i != ptr->key.size()-1){
                if(item > ptr->key[i] && item < ptr->key[i+1]){
                    ptr->next[i+1] = deletion(item,ptr->next[i+1]);
                    return ptr;
                }
            }
            else{
                if(item > ptr->key[i]){
                    ptr->next[i+1] = deletion(item,ptr->next[i+1]);
                    return ptr;
                }
            }
        }
    }

    void del(int item){
        mnode * ptr = deletion(item,root);
        if(ptr == NULL){
            cout<<"\nitem not found";
            return;
        }
        else{
            cout<<"\nitem found and deleted.";
        }
        return;
    }

    void traversal(){
        queue<mnode*> q1;
        if(root == NULL){
            cout<<"\ntree is empty.";
            return;
        }
        q1.push(root);
        q1.push(NULL);
        while(!q1.empty()){
            mnode * ptr = q1.front();
```

```
            q1.pop();
            if(ptr == NULL){
                cout<<"\n";
                continue;
            }
            for(int i =0;i<ptr->key.size();i++){
                cout<<ptr->key[i]<<"  ";
            }
            for(int i =0;i<ptr->key.size()+1;i++){
                q1.push(ptr->next[i]);
            }
        }
        return;
    }

    int height(){
        h = 0;
        queue<mnode*> q;
        mnode * ptr = root;
        q.push(ptr);
        if(root == NULL)
            return 0;
        int nodecount;

        while(!q.empty()){
            nodecount=q.size();
            for(int i =0;i<nodecount;i++){
                ptr = q.front();
                q.pop();
                for( int j = 0;j < ptr->next.size();j++){
                    if(ptr->next[j] != NULL)
                        q.push(ptr->next[j]);
                }
            }
            h++;
        }
        cout<<"\nheight is:"<<h;
        return h;
    }

    vector<mnode*> a;
    void getlevel(mnode * ptr,int level){
        if (level == 0) {
                        a.push_back(ptr);
                }
                else {
                        if (ptr != NULL) {
                for(int i =0;i<m;i++){
                                getlevel(ptr->next[i], level - 1);
                }
                        }
                        else {
                for(int i =0;i<m;i++){
                                getlevel(NULL, level - 1);
                }
                        }
```

```
                }
    }

    void ins(int item){
        insertion(item,root);
    }

    void traverse(){
        traversal();
    }

};

int main(){
    // Insertion
    mway t1(5);

    cout<<"Insertion\n";
    vector<int> ins = {50,60,80,30,35,58,59,63,70,73,96,52,54,61,62,57,55,56,53};
    for(auto i:ins){
        cout<<"\nInserting "<<i;
        t1.ins(i);
        // t1.traverse();
    }

    // Traversal
    cout<<"\nTraversal\n";
    t1.traverse();
    cout<<"\n";

    // Height
    t1.height();

    // Deletion
    cout<<"Deletion\n";
    vector<int> del = {63, 62, 96, 52};
    for(auto i:del){
        cout<<"\nDeleting "<<i;
        t1.del(i);
        t1.traverse();
    }
    return 0;
}
```

OUTPUT

```
Insertion

Inserting 50
Inserting 60
Inserting 80
Inserting 30
Inserting 35
Inserting 58
Inserting 59
Inserting 63
```

```
Inserting 70
Inserting 73
Inserting 96
Inserting 52
Inserting 54
Inserting 61
Inserting 62
Inserting 57
Inserting 55
Inserting 56
Inserting 53

Traversal
30  50  60  80
35  52  54  58  59  61  63  70  73  96
53  55  56  57
62

height is:3

Deletion

Deleting 63
item found and deleted.
30  50  60  80
35  52  54  58  59  61  70  73  96
53  55  56  57
62


Deleting 62
item found and deleted.
30  50  60  80
35  52  54  58  59  61  70  73  96
53  55  56  57

Deleting 96
item found and deleted.
30  50  60  80
35  52  54  58  59  61  70  73
53  55  56  57

Deleting 52
item found and deleted.
30  50  60  80
35  53  54  58  59  61  70  73
55  56  57
```

Splay Tree

CODE

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

```
typedef long long lld;
typedef unsigned long long llu;

struct TreeNode
{
    int key;
    TreeNode* parent;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int key)
    {
        this -> key = key;
        this -> parent = NULL;
        this -> left = NULL;
        this -> right = NULL;
    }
};

class SplayTree
{
    TreeNode *root;

    void zig(TreeNode*);
    void zig_zig(TreeNode*);
    void zig_zag(TreeNode*);

    void splay(TreeNode*);

public:
    SplayTree();
    SplayTree(TreeNode*);
    TreeNode* find(int);
    void insert(int);
    void Delete(int);
    void inOrderPrint(bool);
};

void SplayTree::zig(TreeNode *x)
{
    TreeNode *p = x -> parent;
    if (p -> left == x)
    {
        TreeNode *A = x -> left;
        TreeNode *B = x -> right;
        TreeNode *C = p -> right;

        x -> parent = NULL;
        x -> right = p;

        p -> parent = x;
        p -> left = B;

        if (B != NULL) B -> parent = p;
    }
```

```
    else
    {
        TreeNode *A = p -> left;
        TreeNode *B = x -> left;
        TreeNode *C = x -> right;

        x -> parent = NULL;
        x -> left = p;

        p -> parent = x;
        p -> right = B;

        if (B != NULL) B -> parent = p;
    }
}

void SplayTree::zig_zig(TreeNode *x)
{
    TreeNode *p = x -> parent;
    TreeNode *g = p -> parent;
    if (p -> left == x)
    {
        TreeNode *A = x -> left;
        TreeNode *B = x -> right;
        TreeNode *C = p -> right;
        TreeNode *D = g -> right;

        x -> parent = g -> parent;
        x -> right = p;

        p -> parent = x;
        p -> left = B;
        p -> right = g;

        g -> parent = p;
        g -> left = C;


        if (x -> parent != NULL)
        {
            if (x -> parent -> left == g) x -> parent -> left = x;
            else x -> parent -> right = x;
        }

        if (B != NULL) B -> parent = p;

        if (C != NULL) C -> parent = g;
    }
    else
    {
        TreeNode *A = g -> left;
        TreeNode *B = p -> left;
        TreeNode *C = x -> left;
        TreeNode *D = x -> right;

        x -> parent = g -> parent;
```

```
        x -> left = p;

        p -> parent = x;
        p -> left = g;
        p -> right = C;

        g -> parent = p;
        g -> right = B;

        if (x -> parent != NULL)
        {
            if (x -> parent -> left == g) x -> parent -> left = x;
            else x -> parent -> right = x;
        }

        if (B != NULL) B -> parent = g;

        if (C != NULL) C -> parent = p;
    }
}

void SplayTree::zig_zag(TreeNode *x)
{
    TreeNode *p = x -> parent;
    TreeNode *g = p -> parent;
    if (p -> right == x)
    {
        TreeNode *A = p -> left;
        TreeNode *B = x -> left;
        TreeNode *C = x -> right;
        TreeNode *D = g -> right;

        x -> parent = g -> parent;
        x -> left = p;
        x -> right = g;

        p -> parent = x;
        p -> right = B;

        g -> parent = x;
        g -> left = C;

        if (x -> parent != NULL)
        {
            if (x -> parent -> left == g) x -> parent -> left = x;
            else x -> parent -> right = x;
        }

        if (B != NULL) B -> parent = p;

        if (C != NULL) C -> parent = g;
    }
    else
    {
        TreeNode *A = g -> left;
        TreeNode *B = x -> left;
```

```
        TreeNode *C = x -> right;
        TreeNode *D = p -> right;

        x -> parent = g -> parent;
        x -> left = g;
        x -> right = p;

        p -> parent = x;
        p -> left = C;

        g -> parent = x;
        g -> right = B;

        if (x -> parent != NULL)
        {
            if (x -> parent -> left == g) x -> parent -> left = x;
            else x -> parent -> right = x;
        }

        if (B != NULL) B -> parent = g;

        if (C != NULL) C -> parent = p;
    }
}

void SplayTree::splay(TreeNode *x)
{
    while (x -> parent != NULL)
    {
        TreeNode *p = x -> parent;
        TreeNode *g = p -> parent;
        if (g == NULL) zig(x);
        else if (g -> left == p && p -> left == x) zig_zig(x);
        else if (g -> right == p && p -> right == x) zig_zig(x);
        else zig_zag(x);
    }
    this -> root = x;
}

SplayTree::SplayTree()
{
    this -> root = NULL;
}

SplayTree::SplayTree(TreeNode *rt)
{
    this -> root = rt;
}

TreeNode* SplayTree::find(int x)
{
    TreeNode *ret = NULL;
    TreeNode *curr = this -> root;
    TreeNode *prev = NULL;
    while (curr != NULL)
    {
```

```
        prev = curr;
        if (x < curr -> key) curr = curr -> left;
        else if (x > curr -> key) curr = curr -> right;
        else
        {
            ret = curr;
            break;
        }
    }
    if (ret != NULL) splay(ret);
    else splay(prev);
    return ret;
}

void SplayTree::insert(int x)
{
    if (root == NULL)
    {
        root = new TreeNode(x);
        return;
    }
    TreeNode *curr = this -> root;
    while (curr != NULL)
    {
        if (x < curr -> key)
        {
            if (curr -> left == NULL)
            {
                TreeNode *newNode = new TreeNode(x);
                curr -> left = newNode;
                newNode -> parent = curr;
                splay(newNode);
                return;
            }
            else curr = curr -> left;
        }
        else if (x > curr -> key)
        {
            if (curr -> right == NULL)
            {
                TreeNode *newNode = new TreeNode(x);
                curr -> right = newNode;
                newNode -> parent = curr;
                splay(newNode);
                return;
            }
            else curr = curr -> right;
        }
        else
        {
            splay(curr);
            return;
        }
    }
}
```

```
TreeNode* subtree_max(TreeNode *subRoot)
{
    TreeNode *curr = subRoot;
    while (curr -> right != NULL) curr = curr -> right;
    return curr;
}

TreeNode* subtree_min(TreeNode *subRoot)
{
    TreeNode *curr = subRoot;
    while (curr -> left != NULL) curr = curr -> left;
    return curr;
}

void SplayTree::Delete(int x)
{
    TreeNode *del = find(x);
    TreeNode *L = del -> left;
    TreeNode *R = del -> right;
    if (L == NULL && R == NULL)
    {
        this -> root = NULL;
    }
    else if (L == NULL)
    {
        TreeNode *M = subtree_min(R);
        splay(M);
    }
    else if (R == NULL)
    {
        TreeNode *M = subtree_max(L);
        splay(M);
    }
    else
    {
        TreeNode *M = subtree_max(L);
        splay(M);
        M -> right = R;
        R -> parent = M;
    }
    delete del;
}

void printTree(TreeNode *root, bool brackets)
{
    if (root == NULL)
    {
        if (brackets) printf("{}");
        return;
    }
    if (brackets) printf("{");
    if (root -> left != NULL) printTree(root -> left, brackets);
    if (root != NULL) printf(" %c ", root -> key);
    if (root -> right != NULL) printTree(root -> right, brackets);
    if (brackets) printf("}");
}
```

```cpp
void SplayTree::inOrderPrint(bool brackets)
{
    printTree(this -> root, brackets);
}

int main()
{
    SplayTree *sTree = new SplayTree();
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('D');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('I');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('N');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('O');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('S');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('A');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('U');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('R');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> Delete('I');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> insert('Z');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    sTree -> Delete('S');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");
```

```
    sTree -> insert('S');
    sTree -> inOrderPrint(true);
    printf("\n---------------------------------------------------------------\n");

    return 0;
}
```

OUTPUT

```
{}
---------------------------------------------------------------
{ D }
---------------------------------------------------------------
{{ D } I }
---------------------------------------------------------------
{{{ D } I } N }
---------------------------------------------------------------
{{{{ D } I } N } O }
---------------------------------------------------------------
{{{{{ D } I } N } O } S }
---------------------------------------------------------------
{ A {{{ D { I }} N { O }} S }}
---------------------------------------------------------------
{{{ A {{ D { I }} N { O }}} S } U }
---------------------------------------------------------------
{{ A {{{ D { I }} N } O }} R {{ S } U }}
---------------------------------------------------------------
{{ A } D {{{ N } O } R {{ S } U }}}
---------------------------------------------------------------
{{{ A } D {{{{ N } O } R { S }} U }} Z }
---------------------------------------------------------------
{{{ A } D {{ N } O }} R {{ U } Z }}
---------------------------------------------------------------
{{{{ A } D {{ N } O }} R } S { U { Z }}}
---------------------------------------------------------------
```

Complexity: $O(logN)$ amortized for all operations