VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

ADAVANCED DATA STRUCTURES

Submitted by

VIJAYA VERMA(1BM20CS187)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Oct 2022-Feb 2023

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**ADVANCED DATA STRUCTURES**" carried out by **VIJAYA VERMA (1BM20CS187),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Advanced Data structures Lab **- (20CS5PEADS)** work prescribed for the said degree.

Name of the Lab-Incharge
Designation
Department of CSE
BMSCE, Bengaluru

Namratha M
Assistant professor
Department of CSE
BMSCE, Bengaluru

`

Index Sheet

LAB PROGRAM 1:

Given a boolean 2D matrix, find the number of islands . A group of connected 1s forms an island. A cell in the 2D matrix can be connected to 8 neighbours.Use disjoint sets to implement .

Complete program code-C++

```
#include<stdio.h>
#include<stdlib.h>
#include<vector>
#include<iostream>
using namespace std;
void count_no_islands(vector<vector<int> > &M, int r, int c);
void dfs(vector<vector<int> > &M, int i, int j, int r, int c);

int num = 0;
int main(){
   int r,c,ele;
   cout<<"Enter dimensions of matrix : "<<endl;
   cin>>r>>c;
   vector <vector<int> > M(r);
   cout<<"Enter the matrix : "<<endl;
   for(int i=0;i<r;i++){
      for(int j=0;j<c;j++){
         cin>>ele;
         M[i].push_back(ele);
      }
   }
   cout<<"matrix : "<<endl;
   for(int i=0;i<r;i++){
      for(int j=0;j<c;j++){
         cout<<M[i][j]<<" ";
      }
      cout<<endl;
   }
   count_no_islands(M,r,c);
   cout<<"No of islands = "<<num<<endl;
}

void count_no_islands(vector<vector<int> > &M, int r, int c){
   for(int i=0;i<r;i++){
      for(int j=0;j<c;j++){
         if(M[i][j]==1){
```

```
            // cout<<"I="<<i<<" J="<<j<<endl;
            num+=1;
            M[i][j]=0;
            dfs(M,i-1,j-1,r,c);
            dfs(M,i-1,j,r,c);
            dfs(M,i-1,j+1,r,c);
            dfs(M,i,j-1,r,c);
            dfs(M,i,j+1,r,c);
            dfs(M,i+1,j-1,r,c);
            dfs(M,i+1,j,r,c);
            dfs(M,i+1,j+1,r,c);
        }
    }
  }
}

void dfs(vector<vector<int> >&M, int i, int j, int r, int c){
  // cout<<"i="<<i<<" j="<<j<<endl;
  if(i<0 || j<0 || i==r || j==c)
    return;
  if(M[i][j]==1){
    M[i][j]=0;
    dfs(M,i-1,j-1,r,c);
    dfs(M,i-1,j,r,c);
    dfs(M,i-1,j+1,r,c);
    dfs(M,i,j-1,r,c);
    dfs(M,i,j+1,r,c);
    dfs(M,i+1,j-1,r,c);
    dfs(M,i+1,j,r,c);
    dfs(M,i+1,j+1,r,c);
  }
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
Enter dimensions of matrix :
5
5
Enter the matrix :
1 0 0 1 0
1 1 0 0 0
0 0 0 1 1
1 1 0 0 0
0 0 0 0 1
matrix :
1 0 0 1 0
1 1 0 0 0
0 0 0 1 1
1 1 0 0 0
0 0 0 0 1
No of islands = 5
```

LAB PROGRAM 2:

Write a program to implement the following list:
An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

Complete program code-C/C++

```
#include<stdio.h>
#include<stdlib.h>
#include <cinttypes>
#include<iostream>

using namespace std;

class Node {
   public : int data;
   Node* xnode;
};

Node* Xor(Node* x, Node* y)
{
   return reinterpret_cast<Node*>(
      reinterpret_cast<uintptr_t>(x)
      ^ reinterpret_cast<uintptr_t>(y));
}

void insert(Node** head_ref, int data)
{

   Node* new_node = new Node();
   new_node -> data = data;

   new_node -> xnode = *head_ref;

   if (*head_ref != NULL) {
```

```
        (*head_ref)
            -> xnode = Xor(new_node, (*head_ref) -> xnode);
    }
    *head_ref = new_node;
}

void printList(Node* head)
{
    Node* curr = head;
    Node* prev = NULL;
    Node* next;

    cout << "The nodes of Linked List are: \n";

    while (curr != NULL) {

        cout << curr -> data << " ";

        next = Xor(prev, curr -> xnode);

        prev = curr;
        curr = next;
    }
}

int main()
{
    Node* head = NULL;
    insert(&head, 10);
    insert(&head, 100);
    insert(&head, 1000);
    insert(&head, 10000);

    printList(head);

    return (0);
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
The nodes of Linked List are:
10000 1000 100 10
```

LAB PROGRAM 3:

Write a program to perform insertion, deletion and searching operations on a skip list.

Complete program code-C/C++

```
#include <stdio.h>
#include<stdlib.h>
#include<iostream>
#define INF 1e9
using namespace std;

struct Node {
   int key;
   Node *prev, *next, *above, *below;

   Node() : key(-INF), prev(NULL), next(new Node(INF)), above(NULL), below(NULL) {}
   Node(int key) : key(key), prev(NULL), next(NULL), above(NULL), below(NULL) {}
};

class SkipList {
   Node* head;
   int height;

   int getRandomLevel(int height) {
      return rand() % height + 1;
   }

  public:
   SkipList(int height) : height(height) {
      head = new Node();
      Node* curr = head;
      for (int i = 0; i < height; ++i) {
         curr->below = new Node();
         curr->next->below = curr->below->next, curr->next->prev = curr;
         curr->below->above = curr;
         curr = curr->below;
         curr->next->above = curr->above->next;
      }
   }

   Node* search(int key) {
```

```
    Node* curr = head;
    while (curr->below) {
        curr = curr->below;
        while (curr->next && curr->next->key <= key)
            curr = curr->next;
    }
    return curr;
}

void insert(int key) {
    Node *prev = search(key), *node = new Node(key), *below = NULL;
    int level = getRandomLevel(height);
    while (level--) {
        node->next = prev->next, node->prev = prev;
        prev->next->prev = node, prev->next = node;
        node->above = level ? new Node(key) : NULL, node->below = below;
        below = node, node = node->above;
        while (level && !prev->above)
            prev = prev->prev;
        prev = prev->above;
    }
}

bool remove(int key) {
    Node* node = search(key);
    if (node->key != key)
        return false;
    while (node) {
        node->prev->next = node->next, node->next->prev = node->prev;
        Node* toDelete = node;
        node = node->above;
        delete toDelete;
    }
    return true;
}

void displayList() {
    Node* level = head;
    int height = SkipList::height;
    while (level->below) {
        level = level->below;
        Node* curr = level;
        cout << height-- << ": ";
        while (curr) {
            cout << (curr->key == INF ? "INF" : (curr->key == -INF ? "-INF" : to_string(curr-
>key))) << ' ';
```

```
            curr = curr->next;
        }
        cout << '\n';
    }
  }
};

int main() {
  srand(time(0));

  cout << "Enter height: ";
  int height;
  cin >> height;

  SkipList sl(height);

  int choice, n;
  bool run = true;

  cout << "1.Insert Element" << endl;
  cout << "2.Delete Element" << endl;
  cout << "3.Search Element" << endl;
  cout << "4.Display List" << endl;
  cout << "5.Exit" << endl;
  while (run) {
    cout << "Enter your choice: ";
    cin >> choice;
    switch (choice) {
      case 1:
        cout << "Enter the element to be inserted: ";
        cin >> n;
        sl.insert(n);
        cout << n << " inserted.";
        break;
      case 2:
        cout << "Enter the element to be deleted: ";
        cin >> n;
        if (sl.remove(n))
          cout << n << " removed.";
        else
          cout << n << " not found.";
        break;
      case 3:
        cout << "Enter the element to be searched: ";
        cin >> n;
        if (sl.search(n)->key == n)
```

```
            cout << n << " found.";
        else
            cout << n << " not found.";
        break;

    case 4:
        cout << "The List is: " << endl;
        sl.displayList();
        break;
    case 5:
        run = false;
        break;
    default:
        cout << "Wrong Choice";
    }
    cout << '\n';
  }

  return 0;
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
Enter height: 5
1.Insert Element
2.Delete Element
3.Search Element
4.Display List
5.Exit
Enter your choice: 1
Enter the element to be inserted: 34
34 inserted.
Enter your choice: 1
Enter the element to be inserted: 2
2 inserted.
Enter your choice: 1
Enter the element to be inserted: 67
67 inserted.
Enter your choice: 1
Enter the element to be inserted: 90
90 inserted.
Enter your choice: 1
Enter the element to be inserted: 33
33 inserted.
```

```
Enter your choice: 4
The List is:
5: -INF 67 INF
4: -INF 67 90 INF
3: -INF 2 33 67 90 INF
2: -INF 2 33 67 90 INF
1: -INF 2 33 34 67 90 INF

Enter your choice: 2
Enter the element to be deleted: 33
33 removed.
Enter your choice: 4
The List is:
5: -INF 67 INF
4: -INF 67 90 INF
3: -INF 2 67 90 INF
2: -INF 2 67 90 INF
1: -INF 2 34 67 90 INF

Enter your choice: 5
```

LAB PROGRAM 4:

Write a program to perform insertion and deletion operations on 2-3 trees.

Complete program code-C/C++

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

class Node {
    int *keys;
    int t;
    Node **C;
    int n;
    bool leaf;

   public:
    Node(bool _leaf);
    void traverse();
    Node *search(int k);
    int findKey(int k);
    void insertNonFull(int k);
    void splitChild(int i, Node *y);
    void remove(int k);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
    int getPred(int idx);
    int getSucc(int idx);
    void fill(int idx);
    void borrowFromPrev(int idx);
    void borrowFromNext(int idx);
    void merge(int idx);
    friend class Tree;
};

class Tree {
    Node *root;
    int t;

   public:
    Tree() {
        root = NULL;
        t = 2;
    }

    void traverse() {
```

```
        if (root != NULL)
            root->traverse();
    }

    Node *search(int k) {
        return (root == NULL) ? NULL : root->search(k);
    }

    void insert(int k);
    void remove(int k);
};

Node::Node(bool leaf1) {
    t = 2;
    leaf = leaf1;
    keys = new int[2 * t - 1];
    C = new Node *[2 * t];
    n = 0;
}

int Node::findKey(int k) {
    int idx = 0;
    while (idx < n && keys[idx] < k)
        ++idx;
    return idx;
}

void Node::remove(int k) {
    int idx = findKey(k);
    if (idx < n && keys[idx] == k) {
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    } else {
        if (leaf) {
            cout << "The key " << k << " is does not exist in the tree\n";
            return;
        }
        bool flag = ((idx == n) ? true : false);
        if (C[idx]->n < t)
            fill(idx);
        if (flag && idx > n)
            C[idx - 1]->remove(k);
        else
            C[idx]->remove(k);
```

```
    }
    return;
}

void Node::removeFromLeaf(int idx) {
    for (int i = idx + 1; i < n; ++i)
        keys[i - 1] = keys[i];
    n--;
    return;
}

void Node::removeFromNonLeaf(int idx) {
    int k = keys[idx];
    if (C[idx]->n >= t) {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    } else if (C[idx + 1]->n >= t) {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx + 1]->remove(succ);
    } else {
        merge(idx);
        C[idx]->remove(k);
    }
    return;
}

int Node::getPred(int idx) {
    Node *cur = C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];
    return cur->keys[cur->n - 1];
}

int Node::getSucc(int idx) {
    Node *cur = C[idx + 1];
    while (!cur->leaf)
        cur = cur->C[0];
    return cur->keys[0];
}

void Node::fill(int idx) {
    if (idx != 0 && C[idx - 1]->n >= t)
        borrowFromPrev(idx);
    else if (idx != n && C[idx + 1]->n >= t)
```

```
        borrowFromNext(idx);
    else {
        if (idx != n)
            merge(idx);
        else
            merge(idx - 1);
    }
    return;
}

void Node::borrowFromPrev(int idx) {
    Node *child = C[idx];
    Node *sibling = C[idx - 1];
    for (int i = child->n - 1; i >= 0; --i)
        child->keys[i + 1] = child->keys[i];
    if (!child->leaf) {
        for (int i = child->n; i >= 0; --i)
            child->C[i + 1] = child->C[i];
    }
    child->keys[0] = keys[idx - 1];
    if (!child->leaf)
        child->C[0] = sibling->C[sibling->n];
    keys[idx - 1] = sibling->keys[sibling->n - 1];
    child->n += 1;
    sibling->n -= 1;
    return;
}

void Node::borrowFromNext(int idx) {
    Node *child = C[idx];
    Node *sibling = C[idx + 1];
    child->keys[(child->n)] = keys[idx];
    if (!(child->leaf))
        child->C[(child->n) + 1] = sibling->C[0];
    keys[idx] = sibling->keys[0];

    for (int i = 1; i < sibling->n; ++i)
        sibling->keys[i - 1] = sibling->keys[i];
    if (!sibling->leaf) {
        for (int i = 1; i <= sibling->n; ++i)
            sibling->C[i - 1] = sibling->C[i];
    }
    child->n += 1;
    sibling->n -= 1;
    return;
}
```

```
void Node::merge(int idx) {
   Node *child = C[idx];
   Node *sibling = C[idx + 1];
   child->keys[t - 1] = keys[idx];
   for (int i = 0; i < sibling->n; ++i)
      child->keys[i + t] = sibling->keys[i];
   if (!child->leaf) {
      for (int i = 0; i <= sibling->n; ++i)
         child->C[i + t] = sibling->C[i];
   }
   for (int i = idx + 1; i < n; ++i)
      keys[i - 1] = keys[i];
   for (int i = idx + 2; i <= n; ++i)
      C[i - 1] = C[i];
   child->n += sibling->n + 1;
   n--;
   delete (sibling);
   return;
}

void Tree::insert(int k) {
   if (root == NULL) {
      root = new Node(true);
      root->keys[0] = k;
      root->n = 1;
   } else {
      if (root->n == 2 * t - 1) {
         Node *s = new Node(false);
         s->C[0] = root;
         s->splitChild(0, root);
         int i = 0;
         if (s->keys[0] < k)
            i++;
         s->C[i]->insertNonFull(k);
         root = s;
      } else
         root->insertNonFull(k);
   }
}

void Node::insertNonFull(int k) {
   int i = n - 1;
   if (leaf == true) {
      while (i >= 0 && keys[i] > k) {
         keys[i + 1] = keys[i];
```

```
            i--;
        }
        keys[i + 1] = k;
        n = n + 1;
    } else {
        while (i >= 0 && keys[i] > k)
            i--;
        if (C[i + 1]->n == 2 * t - 1) {
            splitChild(i + 1, C[i + 1]);
            if (keys[i + 1] < k)
                i++;
        }
        C[i + 1]->insertNonFull(k);
    }
}

void Node::splitChild(int i, Node *y) {
    Node *z = new Node(y->leaf);
    z->n = t - 1;
    for (int j = 0; j < t - 1; j++)
        z->keys[j] = y->keys[j + t];
    if (y->leaf == false) {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j + t];
    }
    y->n = t - 1;
    for (int j = n; j >= i + 1; j--)
        C[j + 1] = C[j];

    C[i + 1] = z;
    for (int j = n - 1; j >= i; j--)
        keys[j + 1] = keys[j];

    keys[i] = y->keys[t - 1];
    n = n + 1;
}

void Node::traverse() {
    int i;
    for (i = 0; i < n; i++) {
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }
    if (leaf == false)
        C[i]->traverse();
```

```
}

Node *Node::search(int k) {
   int i = 0;
   while (i < n && k > keys[i])
      i++;
   if (keys[i] == k)
      return this;

   if (leaf == true)
      return NULL;
   return C[i]->search(k);
}

void Tree::remove(int k) {
   if (!root) {
      cout << "The tree is empty\n";
      return;
   }

   root->remove(k);
   if (root->n == 0) {
      Node *tmp = root;
      if (root->leaf)
         root = NULL;
      else
         root = root->C[0];

      delete tmp;
   }
   return;
}

int main() {
   Tree t;
   cout << "1. Insert\n2. Delete\n3. Display Tree\n4. Exit" << endl;
   int choice, node;
   do {
      cout << "Enter choice: ";
      cin >> choice;
      switch (choice) {
         case 1:
            cout << "Enter node: ";
            cin >> node;
            t.insert(node);
            break;
```

```
            case 2:
                cout << "Enter node to remove: ";
                cin >> node;
                t.remove(node);
                break;
            case 3:
                cout << "Tree is \n";
                t.traverse();
                cout << endl;
                break;
            case 4:
                return 0;
            default:
                cout << "Enter valid choice!" << endl;
        }
    } while (choice != 4);
    return 0;
}
```

outputs.

```
/tmp/swXE7OwvJ0.o
1. Insert
2. Delete
3. Display Tree
4. Exit
Enter choice: 1
Enter node: 34
Enter choice: 1
Enter node: 56
Enter choice: 1
Enter node: 22
Enter choice: 1
Enter node: 45
Enter choice: 3
Tree is
 22 34 45 56
Enter choice: 2
Enter node to remove: 34
Enter choice: 3
Tree is
 22 45 56
Enter choice:
```

LAB PROGRAM 5:

Write a program to perform insertion and deletion operations on AVL trees

Complete program code-C/C++

```
#include <stdio.h>
#include<stdlib.h>
#include<iostream>
using namespace std;

class Node {
  public:
   int key;
   Node *left;
   Node *right;
   int height;
};

int max(int a, int b);

int height(Node *N) {
   if (N == NULL)
      return 0;
   return N->height;
}

int max(int a, int b) {
   return (a > b) ? a : b;
}

Node *newNode(int key) {
   Node *node = new Node();
   node->key = key;
   node->left = NULL;
   node->right = NULL;
   node->height = 1;

   return (node);
}

Node *rightRotate(Node *y) {
   Node *x = y->left;
```

```
    Node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

int getBalance(Node *N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

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

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

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
```

```
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

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

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

    return current;
}

Node *deleteNode(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) || (root->right == NULL)) {
            Node *temp = root->left ? root->left : root->right;

            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
```

```
            free(temp);
        } else {
            Node *temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == NULL)
        return root;

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void preOrder(Node *root) {
    if (root != NULL) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main() {
    Node *root = NULL;
    int n, choice, val;
```

```
    while (true) {
        cout << "1 - Insert, 2 - Delete, 3 - Display: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter element to be inserted: ";
                cin >> val;
                root = insert(root, val);
                break;
            case 2:
                cout << "Enter element to be deleted: ";
                cin >> val;
                root = deleteNode(root, val);
                break;
            case 3:
                cout << "AVL Tree: " << endl;
                preOrder(root);
                cout << "\n\n";
                break;
            default:
                exit(0);
        };
    }

    return 0;
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
1 - Insert, 2 - Delete, 3 - Display: 1
Enter element to be inserted: 34
1 - Insert, 2 - Delete, 3 - Display: 1
Enter element to be inserted: 2
1 - Insert, 2 - Delete, 3 - Display: 1
Enter element to be inserted: 32
1 - Insert, 2 - Delete, 3 - Display: 1
Enter element to be inserted: 67
1 - Insert, 2 - Delete, 3 - Display: 3
AVL Tree:
32 2 34 67

1 - Insert, 2 - Delete, 3 - Display: 2
Enter element to be deleted: 2
1 - Insert, 2 - Delete, 3 - Display: 3
AVL Tree:
34 32 67
```

LAB PROGRAM 6:

Write a program to implement insertion operation on a B-tree

Complete program code-C++

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

class Node {
    int *keys;
    int t;
    Node **C;
    int n;
    bool leaf;

  public:
    Node(int tt, bool lleaf);

    void insertNonFull(int k);
    void splitChild(int i, Node *y);
    void traverse();

    friend class BTree;
};

class BTree {
    Node *root;
    int t;

  public:
    BTree(int tt) {
        root = NULL;
        t = tt;
    }

    void traverse() {
        if (root) root->traverse();
    }

    void insert(int k);
};

Node::Node(int tt, bool lleaf) {
```

```
    t = tt;
    leaf = lleaf;

    keys = new int[2 * t - 1];
    C = new Node *[2 * t];

    n = 0;
}

void Node::traverse() {
    int i;
    for (i = 0; i < n; i++) {
        if (leaf == false) C[i]->traverse();
        cout << " " << keys[i];
    }

    if (leaf == false) C[i]->traverse();
}

void BTree::insert(int k) {
    if (root == NULL) {
        root = new Node(t, true);
        root->keys[0] = k;
        root->n = 1;
    } else {
        if (root->n == 2 * t - 1) {
            Node *s = new Node(t, false);

            s->C[0] = root;

            s->splitChild(0, root);

            int i = 0;
            if (s->keys[0] < k) i++;
            s->C[i]->insertNonFull(k);

            root = s;
        } else
            root->insertNonFull(k);
    }
}

void Node::insertNonFull(int k) {
    int i = n - 1;

    if (leaf == true) {
```

```
        while (i >= 0 && keys[i] > k) {
            keys[i + 1] = keys[i];
            i--;
        }

        keys[i + 1] = k;
        n = n + 1;
    } else {
        while (i >= 0 && keys[i] > k) i--;

        if (C[i + 1]->n == 2 * t - 1) {
            splitChild(i + 1, C[i + 1]);

            if (keys[i + 1] < k) i++;
        }
        C[i + 1]->insertNonFull(k);
    }
}

void Node::splitChild(int i, Node *y) {
    Node *z = new Node(y->t, y->leaf);
    z->n = t - 1;

    for (int j = 0; j < t - 1; j++)
        z->keys[j] = y->keys[j + t];

    if (y->leaf == false)
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j + t];

    y->n = t - 1;
    for (int j = n; j >= i + 1; j--)
        C[j + 1] = C[j];

    C[i + 1] = z;

    for (int j = n - 1; j >= i; j--)
        keys[j + 1] = keys[j];

    keys[i] = y->keys[t - 1];
    n = n + 1;
}

int main() {
    int o;
    cout << "Enter B Tree Order: ";
```

```
    cin >> o;
    BTree t(o);

    int k, ele;
    cout << "Number of elements to insert: ";

    cin >> k;
    cout << "Enter " << k << " elements \n";
    while (k--) {
        cin >> ele;
        t.insert(ele);
    }

    cout << "The B-tree is: ";
    {
        // * In a different block because it doesn't end with a newline
        t.traverse();
        cout << endl;
    }
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
Enter B Tree Order: 3
Number of elements to insert: 6
Enter 6 elements
34 2 89 7 5 3
The B-tree is:  2 3 5 7 34 89

|
```

LAB PROGRAM 7:

Write a program to implement insertion operation on a red black tree. During insertion, appropriately show how recolouring or rotation operation is used

Complete program code-C/C++

```
#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 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;
   }
```

```
// For balancing the tree after insertion
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;
}

 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;
}

// Inserting a node
void insert(int key) {
    NodePtr node = new Node;
    node->parent = nullptr;
    node->left = TNULL;
    node->right = TNULL;
    node->color = 1;
    node->data = key;

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

int main() {
    RedBlackTree bst;
    int n;
    cout << "Enter Number of Nodes: ";
    cin >> n;

    while (n--) {
        int k;
        cin >> k;
        bst.insert(k);
    }

    bst.printTree();
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
Enter Number of Nodes: 5
5 45 89 23 2
R----45(BLACK)
   L----5(BLACK)
|  L----2(RED)
   |  R----23(RED)
   R----89(BLACK)
```

LAB PROGRAM 8:

Write a program to implement functions of Dictionary using Hashing.

Complete program code-C/C++

```
#include <stdlib.h>

#include <iostream>
#define YELLOW "\033[33m"
#define RESET "\033[0m"

using namespace std;

#define max 10

typedef struct list {
   int data;
   struct list *next;
} node;
node *ptr[max], *root[max], *temp[max];

class Dictionary {
  public:
   int index;
   Dictionary();
   void insert(int);
   int search(int);
   void del(int);
};

Dictionary::Dictionary() {
   index = -1;
   for (int i = 0; i < max; i++) {
      root[i] = NULL;
      ptr[i] = NULL;
      temp[i] = NULL;
   }
}

void Dictionary::insert(int key) {
   index = int(key % max);
   ptr[index] = (node *)malloc(sizeof(node));
```

```
    ptr[index]->data = key;
    if (root[index] == NULL) {
        root[index] = ptr[index];
        root[index]->next = NULL;
        temp[index] = ptr[index];
    } else {
        temp[index] = root[index];
        while (temp[index]->next != NULL) temp[index] = temp[index]->next;
        temp[index]->next = ptr[index];
    }
}
int Dictionary::search(int key) {
    int flag = 0;
    index = int(key % max);
    temp[index] = root[index];
    while (temp[index] != NULL) {
        if (temp[index]->data == key) {
            flag = 1;
            break;
        } else
            temp[index] = temp[index]->next;
    }
    return flag;
}
void Dictionary::del(int key) {
    index = int(key % max);
    temp[index] = root[index];
    if (search(key) == 0) {
        cout << "Key not found in dictionary. Did not delete anything" << endl;
        return;
    }
    while (temp[index]->data != key && temp[index] != NULL) {
        ptr[index] = temp[index];
        temp[index] = temp[index]->next;
    }
    ptr[index]->next = temp[index]->next;
    cout << temp[index]->data << " has been deleted." << endl;
    temp[index]->data = -1;
    temp[index] = NULL;
    free(temp[index]);
}
int main() {
    int val, ch, n, num;
    char c;
    Dictionary d;
    do {
```

```
    cout << YELLOW << "1. Create\n2. Search for a value\n3. Delete an value" << RESET;
    cout << "\nEnter your choice: ";
    cin >> ch;
    switch (ch) {
      case 1:
        cout << "Enter the number of elements to be inserted: ";
        cin >> n;
        cout << "Enter the elements to be inserted: " << endl;
        for (int i = 0; i < n; i++) {
          cin >> num;
          d.insert(num);
        }
        break;
      case 2:
        cout << "Enter the element to be searched: ";
        cin >> n;
        if (d.search(n) == 1)
          cout << "Search key found" << endl;
        else
          cout << "Search key not found" << endl;
        break;
      case 3:
        cout << "Enter the element to be deleted: ";
        cin >> n;
        d.del(n);
        break;
      default:
        cout << "Invalid Choice." << endl;
    }
    cout << "Continue? (y/n): ";
    cin >> c;
  } while (c == 'y');
  return 0;
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 1
Enter the number of elements to be inserted: 5
Enter the elements to be inserted:
34 67 45 22 55
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 2
Enter the element to be searched: 22
Search key found
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 3
Enter the element to be deleted: 22
22 has been deleted.
Continue? (y/n): n
```

LAB PROGRAM 9:

Write a program to implement the following functions on a Binomial heap:
1. insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.

3. extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial

Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap.

Complete program code-C++

```
#include <stdlib.h>

#include <iostream>
#define YELLOW "\033[33m"
#define RESET "\033[0m"

using namespace std;

#define max 10

typedef struct list {
    int data;
    struct list *next;
} node;
node *ptr[max], *root[max], *temp[max];

class Dictionary {
   public:
    int index;
    Dictionary();
```

```
    void insert(int);
    int search(int);
    void del(int);
};

Dictionary::Dictionary() {
    index = -1;
    for (int i = 0; i < max; i++) {
        root[i] = NULL;
        ptr[i] = NULL;
        temp[i] = NULL;
    }
}

void Dictionary::insert(int key) {
    index = int(key % max);
    ptr[index] = (node *)malloc(sizeof(node));
    ptr[index]->data = key;
    if (root[index] == NULL) {
        root[index] = ptr[index];
        root[index]->next = NULL;
        temp[index] = ptr[index];
    } else {
        temp[index] = root[index];
        while (temp[index]->next != NULL) temp[index] = temp[index]->next;
        temp[index]->next = ptr[index];
    }
}
int Dictionary::search(int key) {
    int flag = 0;
    index = int(key % max);
    temp[index] = root[index];
    while (temp[index] != NULL) {
        if (temp[index]->data == key) {
            flag = 1;
            break;
        } else
            temp[index] = temp[index]->next;
    }
    return flag;
}
void Dictionary::del(int key) {
    index = int(key % max);
    temp[index] = root[index];
    if (search(key) == 0) {
        cout << "Key not found in dictionary. Did not delete anything" << endl;
```

```
        return;
    }
    while (temp[index]->data != key && temp[index] != NULL) {
        ptr[index] = temp[index];
        temp[index] = temp[index]->next;
    }
    ptr[index]->next = temp[index]->next;
    cout << temp[index]->data << " has been deleted." << endl;
    temp[index]->data = -1;
    temp[index] = NULL;
    free(temp[index]);
}
int main() {
    int val, ch, n, num;
    char c;
    Dictionary d;
    do {
        cout << YELLOW << "1. Create\n2. Search for a value\n3. Delete an value" << RESET;
        cout << "\nEnter your choice: ";
        cin >> ch;
        switch (ch) {
            case 1:
                cout << "Enter the number of elements to be inserted: ";
                cin >> n;
                cout << "Enter the elements to be inserted: " << endl;
                for (int i = 0; i < n; i++) {
                    cin >> num;
                    d.insert(num);
                }
                break;
            case 2:
                cout << "Enter the element to be searched: ";
                cin >> n;
                if (d.search(n) == 1)
                    cout << "Search key found" << endl;
                else
                    cout << "Search key not found" << endl;
                break;
            case 3:
                cout << "Enter the element to be deleted: ";
                cin >> n;
                d.del(n);
                break;
            default:
                cout << "Invalid Choice." << endl;
        }
```

```
        cout << "Continue? (y/n): ";
        cin >> c;
    } while (c == 'y');
    return 0;
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 1
Enter the number of elements to be inserted: 3
Enter the elements to be inserted:
23 45 34
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 2
Enter the element to be searched: 45
Search key found
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 3
Enter the element to be deleted: 34
```

LAB PROGRAM 10:

Write a program to implement the following functions on a Binomial heap:

1. delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

2. decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node.

Complete program code-C++

```
#include <stdlib.h>

#include <iostream>
#define YELLOW "\033[33m"
#define RESET "\033[0m"

using namespace std;

#define max 10

typedef struct list {
    int data;
    struct list *next;
} node;
node *ptr[max], *root[max], *temp[max];

class Dictionary {
   public:
    int index;
    Dictionary();
    void insert(int);
    int search(int);
    void del(int);
};

Dictionary::Dictionary() {
    index = -1;
```

```
    for (int i = 0; i < max; i++) {
        root[i] = NULL;
        ptr[i] = NULL;
        temp[i] = NULL;
    }
}

void Dictionary::insert(int key) {
    index = int(key % max);
    ptr[index] = (node *)malloc(sizeof(node));
    ptr[index]->data = key;
    if (root[index] == NULL) {
        root[index] = ptr[index];
        root[index]->next = NULL;
        temp[index] = ptr[index];
    } else {
        temp[index] = root[index];
        while (temp[index]->next != NULL) temp[index] = temp[index]->next;
        temp[index]->next = ptr[index];
    }
}
int Dictionary::search(int key) {
    int flag = 0;
    index = int(key % max);
    temp[index] = root[index];
    while (temp[index] != NULL) {
        if (temp[index]->data == key) {
            flag = 1;
            break;
        } else
            temp[index] = temp[index]->next;
    }
    return flag;
}
void Dictionary::del(int key) {
    index = int(key % max);
    temp[index] = root[index];
    if (search(key) == 0) {
        cout << "Key not found in dictionary. Did not delete anything" << endl;
        return;
    }
    while (temp[index]->data != key && temp[index] != NULL) {
        ptr[index] = temp[index];
        temp[index] = temp[index]->next;
    }
    ptr[index]->next = temp[index]->next;
```

```
    cout << temp[index]->data << " has been deleted." << endl;
    temp[index]->data = -1;
    temp[index] = NULL;
    free(temp[index]);
}
int main() {
    int val, ch, n, num;
    char c;
    Dictionary d;
    do {
        cout << YELLOW << "1. Create\n2. Search for a value\n3. Delete an value" << RESET;
        cout << "\nEnter your choice: ";
        cin >> ch;
        switch (ch) {
            case 1:
                cout << "Enter the number of elements to be inserted: ";
                cin >> n;
                cout << "Enter the elements to be inserted: " << endl;
                for (int i = 0; i < n; i++) {
                    cin >> num;
                    d.insert(num);
                }
                break;
            case 2:
                cout << "Enter the element to be searched: ";
                cin >> n;
                if (d.search(n) == 1)
                    cout << "Search key found" << endl;
                else
                    cout << "Search key not found" << endl;
                break;
            case 3:
                cout << "Enter the element to be deleted: ";
                cin >> n;
                d.del(n);
                break;
            default:
                cout << "Invalid Choice." << endl;
        }
        cout << "Continue? (y/n): ";
        cin >> c;
    } while (c == 'y');
    return 0;
}
```

OUTPUT

```
/tmp/swXE7OwvJ0.o
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 1
Enter the number of elements to be inserted: 3
Enter the elements to be inserted:
43 23 1
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 2
Enter the element to be searched: 2
Search key not found
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 2
Enter the element to be searched: 1
Search key found
Continue? (y/n): y
1. Create
2. Search for a value
3. Delete an value
Enter your choice: 3
Enter the element to be deleted: 1
1 has been deleted.
Continue? (y/n):
```