VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

ADVANCED DATA STRUCTURES (22CS5PEADS)

Submitted by

DHANUSH S (1BM21CS053)

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
March -June 2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

This is to certify that the Lab work entitled **“ADVANCED DATA STRUCTURES”** carried out by **DHANUSH S (1BM21CS053)**, 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 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Advanced Data Structures Lab - **(22CS5PEADS)** work prescribed for the said degree.

Prof. Namratha M	**Dr. Jyothi S Nayak**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

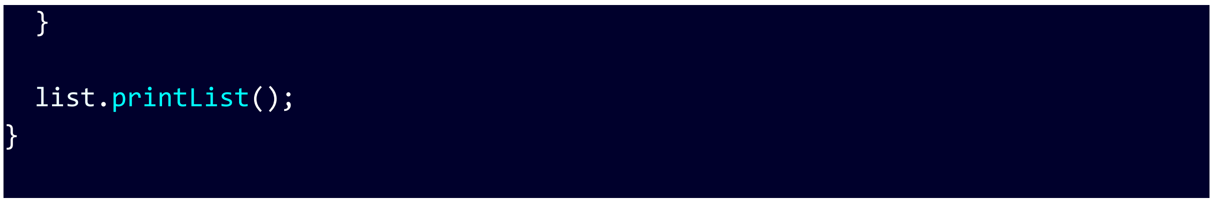
Index Sheet

Course outcomes:

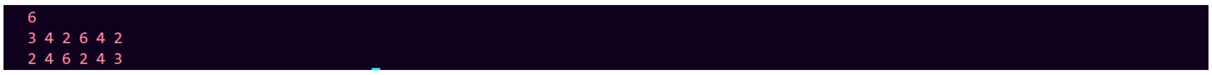
CO1	Apply the concepts of advanced data structures for the given scenario.
CO2	Analyze the usage of appropriate data structures for a given application.
CO3	Design algorithms for performing operations on various advanced data structures.
CO4	Conduct practical experiments to solve problems using an appropriate data structure.

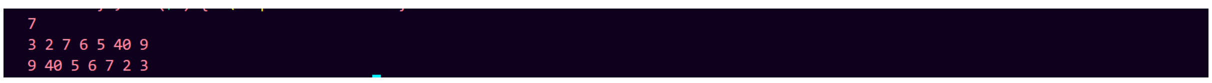
Lab program 1: Write a program to implement the following list: An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory-efficient version of the Doubly Linked List can be created using only one space for the address field with every 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.

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

class Node {
  public:
    int data;
    Node* ptr;

    Node() {
      int data = 0;
      ptr = NULL;
    }

    Node(int data) {
      this->data = data;
      ptr = NULL;
    }

    Node(Node* ptr) {
      data = 0;
      this->ptr = ptr;
    }

    Node(int data, Node* ptr) {
      this->data = data;
      this->ptr = ptr;
    }

};

class XorList {
  private:
    Node* head = NULL;

    Node* XOR(Node* a, Node* b) {
      return (Node*)((uintptr_t)(a) ^ (uintptr_t)(b));
    }

    void movePtr(Node** curr, Node** prevOrNext) {
      Node* next = XOR(*prevOrNext, (*curr)->ptr);
      *prevOrNext = *curr;
      *curr = next;
    }

  public:
    void insertAtBeg(int data) {
      Node* newNode = new Node(data, head);

      if(head != NULL)
        head->ptr = XOR(newNode, head->ptr);

      head = newNode;
    }

    void deleteFromBeg() {
      if(head == NULL)
        return;

      Node* prev = head;
      head = head->ptr;

      if(head != NULL)
        head->ptr = XOR(prev, head->ptr);

      delete prev;
    }


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

      while(curr != NULL) {
        cout << curr->data << " ";
        movePtr(&curr, &prev);
      }

      cout << endl;
    }
};

int main() {
  int n;
  cin >> n;

  XorList list;
  while(n--) {
    int x;
    cin >> x;
    list.insertAtBeg(x);
```


```
  }

  list.printList();
}
```

Output:


```
6
3 4 2 6 4 2
2 4 6 2 4 3
```


```
7
3 2 7 6 5 40 9
9 40 5 6 7 2 3
```

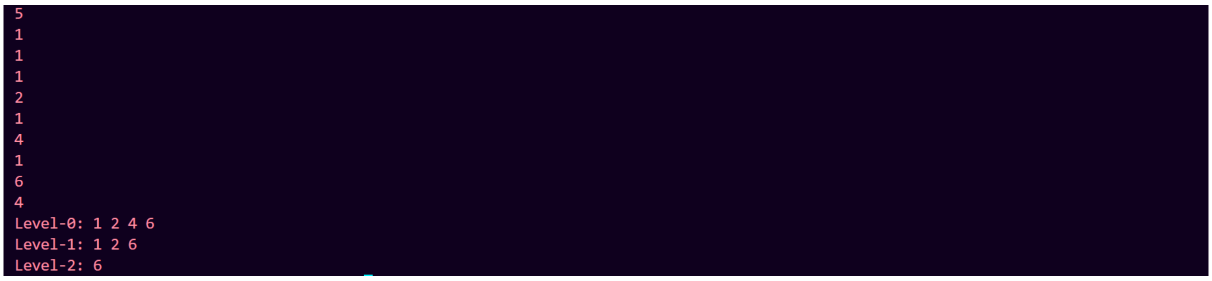
Lab program 2: Write a program to perform insertion, deletion and searching operations on a skip list.

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

class Node
{
public:
    int key;
    int level;
    Node **links;

    Node(int key, int level)
    {
        this->key = key;
        this->level = level;
        this->links = new Node *[level + 1];

        for (int i = 0; i <= level; i++)
            links[i] = NULL;
    }
};

class SkipList
{
private:
    Node *header;
    double P;
    int MAX_LEVEL;
    int currListLevel;

    int getRandomLevel()
    {
        float r = (float)rand() / RAND_MAX;
        int level = 0;

        while (r < P && level <= MAX_LEVEL)
        {
            level++;
            r = (float)rand() / RAND_MAX;
        }

        return level;
    }

    Node *createNewNode(int key)
    {
        int level = getRandomLevel();
        return new Node(key, level);
    }
```

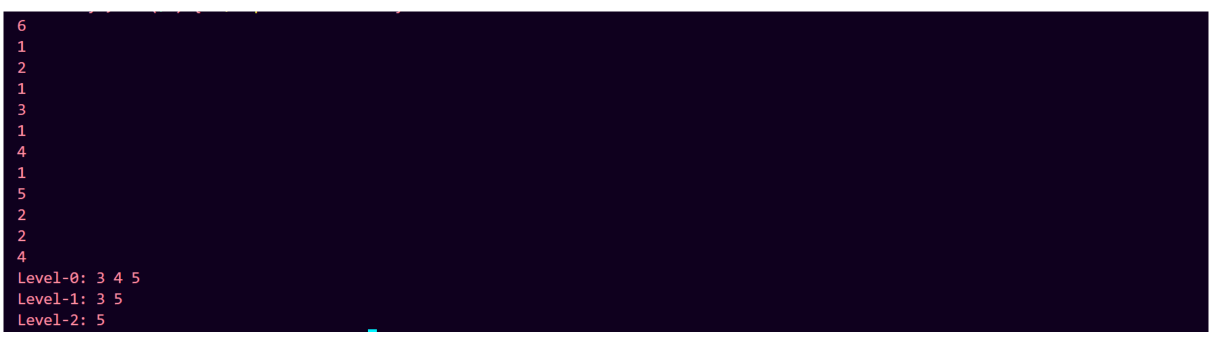
```cpp
public:
    SkipList(int maxLevel, double p)
    {
        this->MAX_LEVEL = maxLevel;
        this->P = p;
        this->header = new Node(-1, MAX_LEVEL);
        this->currListLevel = 0;
    }

    void insertNode(int key)
    {
        Node *curr = header;
        Node *update[MAX_LEVEL + 1];

        for (int i = currListLevel; i >= 0; i--)
        {
            while (curr->links[i] != NULL && curr->links[i]->key < key)
                curr = curr->links[i];
            update[i] = curr;
        }
        curr = curr->links[0];

        // don't insert if key is already present
        if (curr != NULL && curr->key == key)
            return;

        Node *newNode = createNewNode(key);
        if (newNode->level > currListLevel)
        {
            for (int i = currListLevel + 1; i <= newNode->level; i++)
                update[i] = header;
            currListLevel = newNode->level;
        }

        for (int i = 0; i <= newNode->level; i++)
        {
            newNode->links[i] = update[i]->links[i];
            update[i]->links[i] = newNode;
        }
    }

    void deleteNode(int key)
    {
        Node *curr = header;
        Node *update[MAX_LEVEL + 1];

        for (int i = currListLevel; i >= 0; i--)
        {
            while (curr->links[i] != NULL && curr->links[i]->key < key)
```

```
                curr = curr->links[i];
            update[i] = curr;
        }
        curr = curr->links[0];

        // don't delete if node is not present
        if (curr == NULL || curr->key > key)
            return;

        for (int i = 0; i <= curr->level; i++)
            update[i]->links[i] = curr->links[i];

        while (currListLevel > 0 && header->links[currListLevel] == NULL)
            currListLevel--;

        delete curr;
    }

    bool searchNode(int key)
    {
        Node *curr = header;

        for (int i = currListLevel; i >= 0; i--)
            while (curr->links[i] != NULL && curr->links[i]->key < key)
                curr = curr->links[i];

        curr = curr->links[0];

        if (curr != NULL && curr->key == key)
            return true;
        else
            return false;
    }

    void displayList()
    {
        for (int i = 0; i <= currListLevel; i++)
        {
            cout << "Level-" << i << ": ";
            Node *curr = header->links[i];

            while (curr != NULL)
            {
                cout << curr->key << " ";
                curr = curr->links[i];
            }

            cout << endl;
        }
    }
};

int main()
{
    int n;
    cin >> n;

    /*
        Query is of the following type

        1 x -> insert key x
        2 x -> delete key x
        3 x -> search key x
        4   -> display list

    */

    SkipList list(5, 0.5);
    while (n--)
    {
        int query, key;
        cin >> query;

        if (query <= 3)
            cin >> key;

        switch (query)
        {
        case 1:
            list.insertNode(key);
            break;
        case 2:
            list.deleteNode(key);
            break;
        case 3:
            cout << (list.searchNode(key) ? "Found" : "Not Found") << endl;
            break;
        case 4:
            list.displayList();
            break;
        }
    }
}
```

Output:



```
5
1
1
1
2
1
4
1
6
4
Level-0: 1 2 4 6
Level-1: 1 2 6
Level-2: 6
```



```
6
1
2
1
3
1
4
1
5
2
2
4
Level-0: 3 4 5
Level-1: 3 5
Level-2: 5
```

Lab Program 3: Given a Boolean 2D matrix, find the number of islands.

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{1, 0, 0, 1, 1},

{0, 0, 0, 0, 0},

{1, 0, 1, 0, 1}

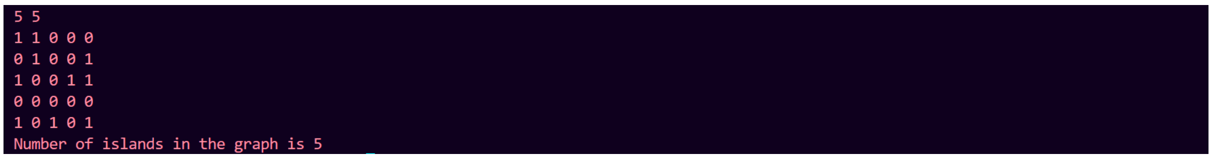
A cell in the 2D matrix can be connected to 8 neighbors.

Use disjoint sets to implement the above scenario.

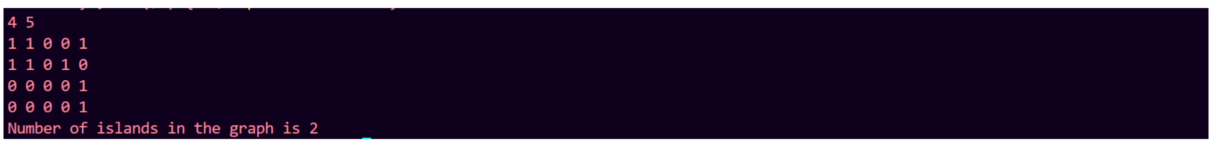
```
#include <bits/stdc++.h>
using namespace std;

class DisJointSet {
public:
    vector<int> rank, parent;
    int n;

    DisJointSet(int n) {
        this->n = n;
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        rank.resize(n, 0);
    }

    int findParent(int node) {
        if (parent[node] == node) return node;
        // path compression code
        return parent[node] = findParent(parent[node]);
    }

    void unionR(int u, int v) {
        int pu = findParent(u);
        int pv = findParent(v);
        if (pu == pv) return;
        if (rank[pu] > rank[pv]) {
            parent[pv] = pu;
        } else if (rank[pv] > rank[pu]) {
            parent[pu] = pv;
        } else {
            // increase the rank of the parent node
            parent[pu] = pv;
            rank[pv]++;
        }
    }
};

int main() {
    int m, n;
    cin >> m >> n;
    vector<vector<int>> matrix(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            cin >> matrix[i][j];
    }

    DisJointSet *ds = new DisJointSet(m * n);

    vector<int> row_offsets = {0, 0, 1, 1, 1, -1, -1, -1};
    vector<int> col_offsets = {1, -1, 0, 1, -1, 0, 1, -1};

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 1) {
                for (int k = 0; k < 8; k++) {
                    int ni = i + row_offsets[k];
                    int nj = j + col_offsets[k];
                    if (ni >= 0 && ni < m && nj >= 0 && nj < n &&
matrix[ni][nj] == 1) {
                        ds->unionR(i * n + j, ni * n + nj);
                    }
                }
            }
        }
    }

    int count = 0;
    vector<int> arr(m * n, 0);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 1 && arr[ds->findParent(i * n + j)] == 0) {
                arr[ds->findParent(i * n + j)] = 1;
                count++;
            }
        }
    }

    cout << "Number of islands in the graph is " << count << endl;
    delete ds;
    return 0;
```

```
}
```

Output:



```
5 5
1 1 0 0 0
0 1 0 0 1
1 0 0 1 1
0 0 0 0 0
1 0 1 0 1
Number of islands in the graph is 5
```



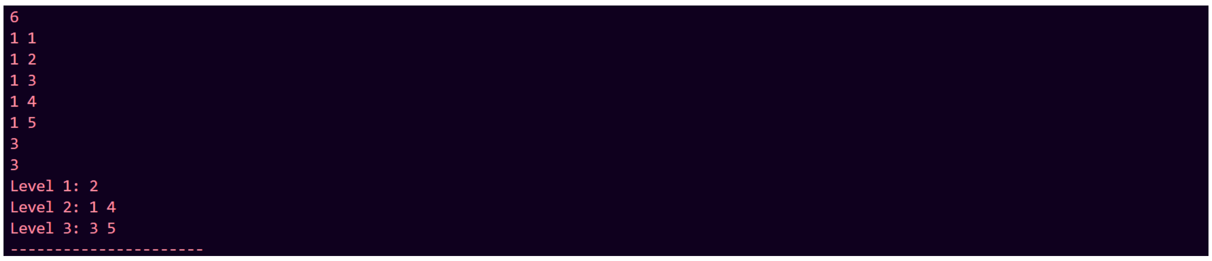
```
4 5
1 1 0 0 1
1 1 0 1 0
0 0 0 0 1
0 0 0 0 1
Number of islands in the graph is 2
```

Lab Program 4: Write a program to perform insertion and deletion operations on AVL trees.

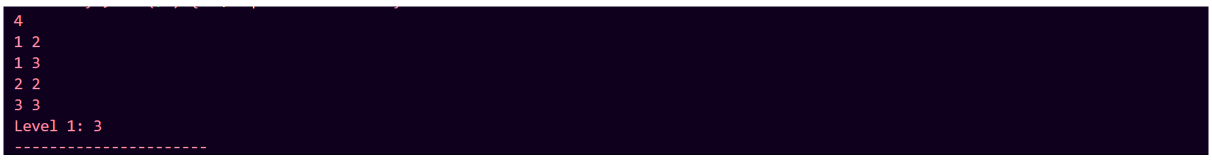
```
#include <bits/stdc++.h>
using namespace std;

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

    Node(int key, Node *left, Node *right, int height)
    {
        this->key = key;
        this->left = left;
        this->right = right;
        this->height = height;
    }
};

class AVLTree
{
private:
    Node *root = nullptr;

    int getHeight(Node *node)
    {
        return node == nullptr ? 0 : node->height;
    }

    Node *leftRotate(Node *node)
    {
        Node *rightChild = node->right;
        Node *subtree = rightChild->left;

        node->right = subtree;
        rightChild->left = node;

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

        return rightChild;
    }

    Node *rightRotate(Node *node)
    {
        Node *leftChild = node->right;
        Node *subtree = leftChild->left;

        node->left = subtree;
        leftChild->right = node;

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

        return leftChild;
    }

    int getBalance(Node *node)
    {
        if (node == nullptr)
            return 0;
        return getHeight(node->left) - getHeight(node->right);
    }

    Node *getMinimumNode(Node *node)
    {
        Node *current = node;
        while (current && current->left != nullptr)
            current = current->left;
        return current;
    }

    Node *getBalancedRoot(Node *root)
    {
        int balance = getBalance(root);

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

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

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

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

    Node *insertNodeInternally(Node *root, int key)
    {
        if (root == nullptr)
            return new Node(key, nullptr, nullptr, 1);

        if (key < root->key)
            root->left = insertNodeInternally(root->left, key);
        else if (key > root->key)
            root->right = insertNodeInternally(root->right, key);
        else
            return root;

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

        if (balance > 1 || balance < -1)
            return getBalancedRoot(root);
        else
            return root;
    }

    Node *deleteNodeInternally(Node *root, int key)
    {
        if (root == nullptr)
            return root;

        if (key < root->key)
            root->left = deleteNodeInternally(root->left, key);
        else if (key > root->key)
            root->right = deleteNodeInternally(root->right, key);
        else
        {
            // root is a leaf node
            if (root->left == nullptr && root->right == nullptr)
            {
                free(root);
                root = nullptr;
            }
            // root has a single child
            else if (root->left == nullptr || root->right == nullptr)
            {
                Node *child = root->left ? root->left : root->right;
```

```
                free(root);
                root = child;
            }
            // root has two children
            else
            {
                Node *successor = getMinimumNode(root->right);
                root->key = successor->key;
                root->right = deleteNodeInternally(root->right,
successor->key);
            }
        }

        if (root == nullptr)
            return root;

        int balance = getBalance(root);

        if (balance > 1 || balance < -1)
            return getBalancedRoot(root);
        else
            return root;
    }

    bool searchNodeInternally(Node *root, int key)
    {
        if (root == nullptr)
            return false;

        if (key < root->key)
            return searchNodeInternally(root->left, key);
        else if (key > root->key)
            return searchNodeInternally(root->right, key);
        else
            return true;
    }

public:
    void insertNode(int key)
    {
        root = insertNodeInternally(root, key);
    }

    void deleteNode(int key)
    {
        root = deleteNodeInternally(root, key);
    }

    void printLevelOrder()
    {
        if (root == nullptr)
        {
            cout << "Tree is empty" << endl;
            return;
        }

        Node *curr;
        queue<Node *> bfs;
        bfs.push(root);
        int level = 1;
        while (!bfs.empty())
        {
            int size = bfs.size();
            cout << "Level " << level << ": ";
            while (size--)
            {
                curr = bfs.front();
                bfs.pop();
                cout << curr->key << " ";

                if (curr->left != nullptr)
                    bfs.push(curr->left);
                if (curr->right != nullptr)
                    bfs.push(curr->right);
            }
            cout << endl;
            level++;
        }
    }
};

int main()
{
    AVLTree tree;

    int n;
    cin >> n;

    while (n--)
    {
        int a, b, c;

        cin >> a;

        if (a <= 3)
            cin >> b;

        switch (a)
        {
        case 1:
            tree.insertNode(b);
            break;

        case 2:
            tree.deleteNode(b);
            break;

        case 3:
            tree.printLevelOrder();
            cout << "----------------------" << endl;
        }
    }
}
```

Output:



```
6
1 1
1 2
1 3
1 4
1 5
3
3
Level 1: 2
Level 2: 1 4
Level 3: 3 5
---------------------
```



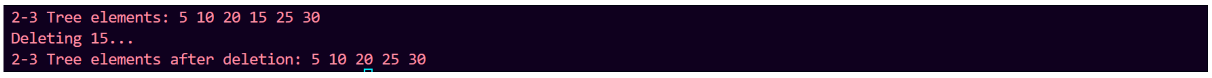
```
4
1 2
1 3
2 2
3 3
Level 1: 3
---------------------
```

Lab Program 5: Write a program to perform insertion and deletion operations on 2-3 trees.

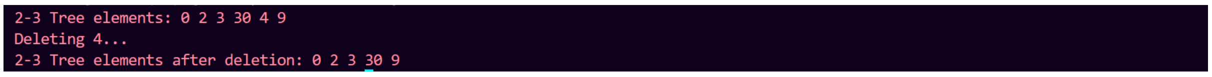
```
#include <iostream>
using namespace std;

class Node {
public:
    int data1;
    int data2;
    Node* left;
    Node* mid;
    Node* right;
    Node* parent;

    Node(int data) {
        data1 = data;
        data2 = -1; // -1 indicates empty
        left = nullptr;
        mid = nullptr;
        right = nullptr;
        parent = nullptr;
    }
};

class TwoThreeTree {
private:
    Node* root;

    Node* insert(Node* node, int data) {
        if (node == nullptr) {
            return new Node(data);
        }

        if (node->data2 == -1) {
            if (data < node->data1) {
                node->data2 = node->data1;
                node->data1 = data;
            } else {
                node->data2 = data;
            }
            return node;
        }

        if (data < node->data1) {
            node->left = insert(node->left, data);
        } else if (data > node->data2) {
            node->right = insert(node->right, data);
        } else {
            node->mid = insert(node->mid, data);
        }

        return node;
    }

    Node* remove(Node* node, int data) {
        if (node == nullptr) {
            return nullptr;
        }

        if (node->data1 == data && node->data2 == -1) {
            delete node;
            return nullptr;
        }

        if (node->data1 == data && node->data2 != -1) {
            node->data1 = node->data2;
            node->data2 = -1;
            return node;
        }

        if (node->data2 == data && node->mid == nullptr) {
            node->data2 = -1;
            return node;
        }

        if (data < node->data1) {
            node->left = remove(node->left, data);
        } else if ((data < node->data2 && data > node->data1) || (node->data1
== data && node->mid != nullptr)) {
            node->mid = remove(node->mid, data);
        } else {
            node->right = remove(node->right, data);
        }

        return node;
    }

    void traverse(Node* node) {
        if (node != nullptr) {
            traverse(node->left);
            cout << node->data1 << " ";
            if (node->data2 != -1) {
                cout << node->data2 << " ";
            }
            traverse(node->mid);
            traverse(node->right);
        }
    }
```

```
public:
    TwoThreeTree() {
        root = nullptr;
    }

    void insert(int data) {
        root = insert(root, data);
    }

    void remove(int data) {
        root = remove(root, data);
    }

    void display() {
        traverse(root);
    }
};

int main() {
    TwoThreeTree tree;
    tree.insert(3);
    tree.insert(30);
    tree.insert(9);
    tree.insert(4);
    tree.insert(0);
    tree.insert(2);

    cout << "2-3 Tree elements: ";
    tree.display();
    cout << endl;

    cout << "Deleting 4...\n";
    tree.remove(4);
    cout << "2-3 Tree elements after deletion: ";
    tree.display();
    cout << endl;

    return 0;
}
```

Output:



```
2-3 Tree elements: 5 10 20 15 25 30
Deleting 15...
2-3 Tree elements after deletion: 5 10 20 25 30
```



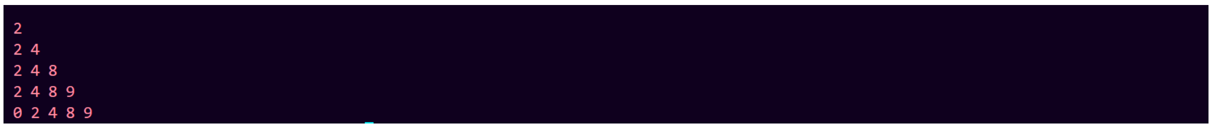
```
2-3 Tree elements: 0 2 3 30 4 9
Deleting 4...
2-3 Tree elements after deletion: 0 2 3 30 9
```

Lab Program 6: Write a program to implement insertion operation on a red-black tree. During insertion, appropriately show how recoloring or rotation operation is used.

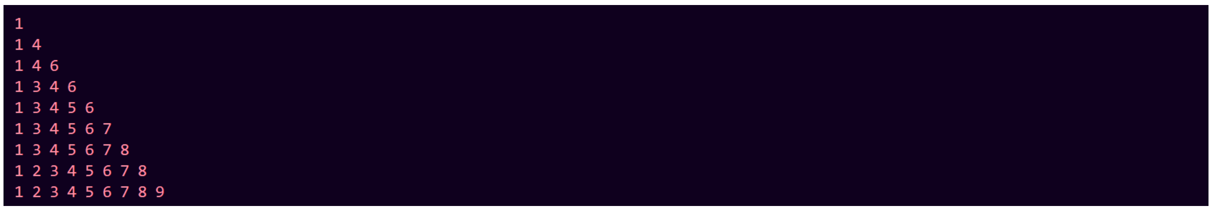
```
#include <bits/stdc++.h>
using namespace std;

class RedBlackTree {
private:

    struct Node {
        int data;
        Node* left;
        Node* right;
        char colour;
        Node* parent;

        Node(int data) : data(data), left(nullptr), right(nullptr),
colour('R'), parent(nullptr) {}
    };

    Node* root;
    bool ll;
    bool rr;
    bool lr;
    bool rl;

    Node* rotateLeft(Node* node) {
        Node* x = node->right;
        Node* y = x->left;
        x->left = node;
        node->right = y;
        node->parent = x;
        if (y != nullptr)
            y->parent = node;
        return x;
    }

    Node* rotateRight(Node* node) {
        Node* x = node->left;
        Node* y = x->right;
        x->right = node;
        node->left = y;
        node->parent = x;
        if (y != nullptr)
            y->parent = node;
        return x;
    }

    Node* insertHelp(Node* root, int data) {
        bool f = false;
        if (root == nullptr)
            return new Node(data);
        else if (data < root->data) {
            root->left = insertHelp(root->left, data);
            root->left->parent = root;
            if (root != this->root) {
                if (root->colour == 'R' && root->left->colour == 'R')
                    f = true;
            }
        } else {
            root->right = insertHelp(root->right, data);
            root->right->parent = root;
            if (root != this->root) {
                if (root->colour == 'R' && root->right->colour == 'R')
                    f = true;
            }
        }

        if (ll) {
            root = rotateLeft(root);
            root->colour = 'B';
            root->left->colour = 'R';
            ll = false;
        } else if (rr) {
            root = rotateRight(root);
            root->colour = 'B';
            root->right->colour = 'R';
            rr = false;
        } else if (rl) {
            root->right = rotateRight(root->right);
            root->right->parent = root;
            root = rotateLeft(root);
            root->colour = 'B';
            root->left->colour = 'R';
            rl = false;
        } else if (lr) {
            root->left = rotateLeft(root->left);
            root->left->parent = root;
            root = rotateRight(root);
            root->colour = 'B';
            root->right->colour = 'R';
            lr = false;
        }

        if (f) {
            if (root->parent->right == root) {
                if (root->parent->left == nullptr ||
root->parent->left->colour == 'B') {
```

```
                    if (root->left != nullptr && root->left->colour == 'R')
                        rl = true;
                    else if (root->right != nullptr && root->right->colour ==
'R')
                        ll = true;
                } else {
                    root->parent->left->colour = 'B';
                    root->colour = 'B';
                    if (root->parent != this->root)
                        root->parent->colour = 'R';
                }
            } else {
                if (root->parent->right == nullptr ||
root->parent->right->colour == 'B') {
                    if (root->left != nullptr && root->left->colour == 'R')
                        rr = true;
                    else if (root->right != nullptr && root->right->colour ==
'R')
                        lr = true;
                } else {
                    root->parent->right->colour = 'B';
                    root->colour = 'B';
                    if (root->parent != this->root)
                        root->parent->colour = 'R';
                }
            }
            f = false;
        }
        return root;
    }

    void inorderTraversalHelper(Node* node) {
        if (node != nullptr) {
            inorderTraversalHelper(node->left);
            std::cout << node->data << " ";
            inorderTraversalHelper(node->right);
        }
    }



public:
    RedBlackTree() : root(nullptr), ll(false), rr(false), lr(false), rl(false)
{}


    void insert(int data) {
        if (root == nullptr) {
            root = new Node(data);
            root->colour = 'B';
        } else
            root = insertHelp(root, data);
    }


    void inorderTraversal() {
        inorderTraversalHelper(root);
    }


};

int main() {
    RedBlackTree t;
    int arr[] = {2 , 4 , 8 , 9 , 0};
    for (int i = 0; i < 5; i++) {
        t.insert(arr[i]);
        std::cout << std::endl;
        t.inorderTraversal();
    }

    return 0;
}
```

Output:



```
2
2 4
2 4 8
2 4 8 9
0 2 4 8 9
```

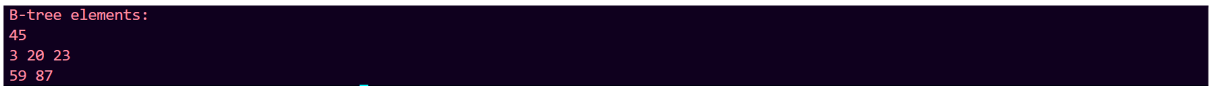


```
1
1 4
1 4 6
1 3 4 6
1 3 4 5 6
1 3 4 5 6 7
1 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

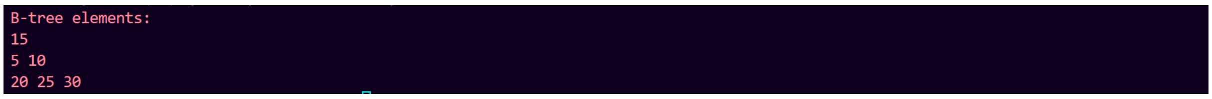
Lab Program 7: Write a program to implement insertion operation on a B-tree.

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

const int MAX_KEYS = 3; // B-tree order

class Node {
public:
    vector<int> keys;
    vector<Node*> children;
    bool leaf;
    Node* parent;

    Node(bool isLeaf) {
        leaf = isLeaf;
        parent = nullptr;
    }
};

class BTree {
private:
    Node* root;

    void splitChild(Node* x, int i) {
        Node* z = new Node(true);
        Node* y = x->children[i];
        z->leaf = y->leaf;

        // Move keys to the new node
        for (int j = 0; j < MAX_KEYS - 1; j++) {
            z->keys.push_back(y->keys[j + MAX_KEYS]);
            y->keys.pop_back();
        }

        // If not leaf, move children too
        if (!y->leaf) {
            for (int j = 0; j < MAX_KEYS; j++) {
                z->children.push_back(y->children[j + MAX_KEYS]);
                y->children.pop_back();
            }
        }

        x->keys.insert(x->keys.begin() + i, y->keys[MAX_KEYS - 1]);
        x->children.insert(x->children.begin() + i + 1, z);
        y->keys.pop_back();
    }
    void insertNonFull(Node* x, int key) {
        int i = x->keys.size() - 1;

        if (x->leaf) {
            x->keys.push_back(0);
            while (i >= 0 && key < x->keys[i]) {
                x->keys[i + 1] = x->keys[i];
                i--;
            }
            x->keys[i + 1] = key;
        } else {
            while (i >= 0 && key < x->keys[i]) {
                i--;
            }
            i++;

            if (x->children[i]->keys.size() == MAX_KEYS * 2 - 1) {
                splitChild(x, i);
                if (key > x->keys[i]) {
                    i++;
                }
            }
            insertNonFull(x->children[i], key);
        }
    }

public:
    BTree() {
        root = nullptr;
    }

    void insert(int key) {
        if (root == nullptr) {
            root = new Node(true);
            root->keys.push_back(key);
        } else {
            if (root->keys.size() == MAX_KEYS * 2 - 1) {
                Node* s = new Node(false);
                s->children.push_back(root);
                splitChild(s, 0);
                root = s;
            }
            insertNonFull(root, key);
        }
    }

    void display(Node* node) {
        if (node != nullptr) {
            for (int i = 0; i < node->keys.size(); i++) {
                cout << node->keys[i] << " ";
            }
            cout << endl;

            if (!node->leaf) {
                for (int i = 0; i < node->children.size(); i++) {
                    display(node->children[i]);
                }
            }
        }
    }

    void display() {
        display(root);
    }
};

int main() {
    BTree tree;
    tree.insert(45);
    tree.insert(3);
    tree.insert(59);
    tree.insert(87);
    tree.insert(23);
    tree.insert(20);

    cout << "B-tree elements: " << endl;
    tree.display();

    return 0;
}
```

Output:



```
B-tree elements:
45
3 20 23
59 87
```



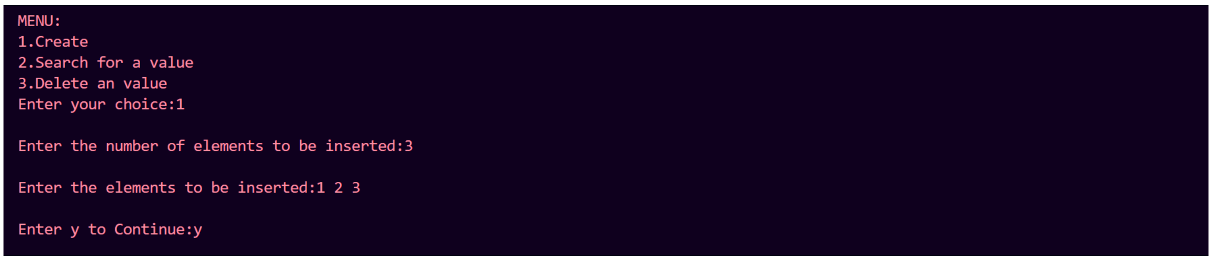
```
B-tree elements:
15
5 10
20 25 30
```

Lab Program 8: Write a program to implement the functions of a Dictionary using Hashing.

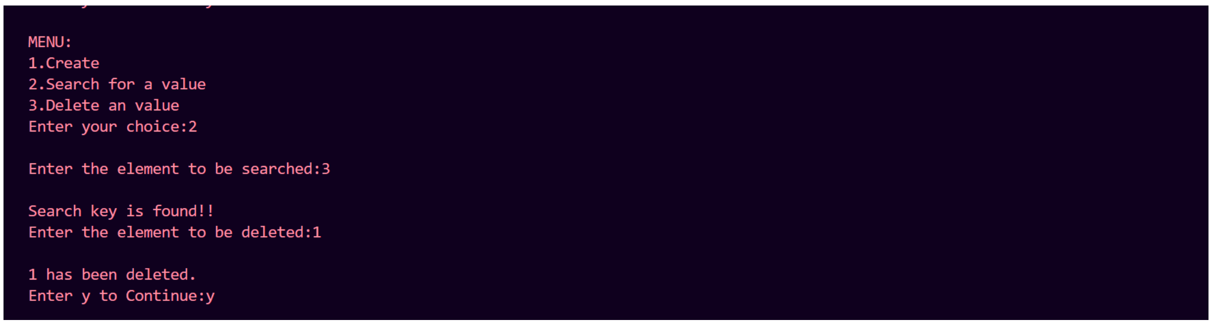
```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
# define max 10
typedef struct list
{
    int data;
    struct list *next;
} node_type;
node_type *ptr[max],*root[max],*temp[max];
class Dictionary
{
public:
    int index;
    Dictionary();
    void insert(int);
    void search(int);
    void delete_ele(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_type*)malloc(sizeof(node_type));
    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];
    }
}
void Dictionary::search(int key)
{
    int flag=0;
    index=int(key%max);
    temp[index]=root[index];
    while(temp[index]!=NULL)
    {
        if(temp[index]->data==key)
        {
            cout<<"\nSearch key is found!!";
            flag=1;
            break;
        }
        else temp[index]=temp[index]->next;
    }
    if (flag==0)
        cout<<"\nsearch key not found.......";
}
void Dictionary::delete_ele(int key)
{
    index=int(key%max);
    temp[index]=root[index];
    while(temp[index]->data!=key && temp[index]!=NULL)
    {
        ptr[index]=temp[index];
        temp[index]=temp[index]->next;
    }
    ptr[index]->next=temp[index]->next;
    cout<<"\n"<<temp[index]->data<<" has been deleted.";
    temp[index]->data=-1;
    temp[index]=NULL;
    free(temp[index]);
}
main()
{
    int val,ch,n,num;
    char c;
    Dictionary d;
    do
    {
        cout<<"\nMENU:\n1.Create";
        cout<<"\n2.Search for a value\n3.Delete an value";
        cout<<"\nEnter your choice:";
        cin>>ch;
        switch(ch)
        {
        case 1:
            cout<<"\nEnter the number of elements to be inserted:";
            cin>>n;
            cout<<"\nEnter the elements to be inserted:";
            for(int i=0; i<n; i++)
            {
                cin>>num;
                d.insert(num);
            }
            break;
        case 2:
            cout<<"\nEnter the element to be searched:";
            cin>>n;
            d.search(n);
        case 3:
            cout<<"\nEnter the element to be deleted:";
            cin>>n;
            d.delete_ele(n);
            break;
        default:
            cout<<"\nInvalid Choice.";
        }
        cout<<"\nEnter y to Continue:";
        cin>>c;
    }
    while(c=='y');
    getch();
}
```

Output:



```
MENU:
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:1 2 3

Enter y to Continue:y
```



```
MENU:
1.Create
2.Search for a value
3.Delete an value
Enter your choice:2

Enter the element to be searched:3

Search key is found!!
Enter the element to be deleted:1

1 has been deleted.
Enter y to Continue:y
```

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 a single key ‘k’, then calls union on H and the new Binomial heap.

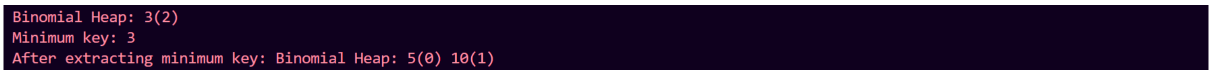
2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.

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.

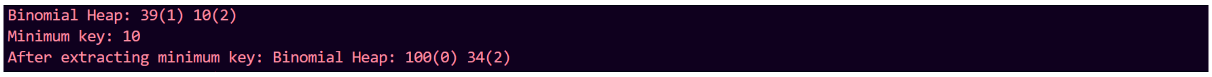
```cpp
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

struct Node {
    int key;
    int degree;
    Node* parent;
    Node* child;
    Node* sibling;
};


Node* createNode(int key) {
    Node* newNode = new Node;
    newNode->key = key;
    newNode->degree = 0;
    newNode->parent = nullptr;
    newNode->child = nullptr;
    newNode->sibling = nullptr;
    return newNode;
}

Node* mergeTrees(Node* tree1, Node* tree2) {
    if (tree1->key > tree2->key)
        swap(tree1, tree2);

    tree2->parent = tree1;
    tree2->sibling = tree1->child;
    tree1->child = tree2;
    tree1->degree++;

    return tree1;
}
```

```
Node* mergeHeaps(Node* heap1, Node* heap2) {
    Node* dummy = createNode(INT_MIN);
    Node* tail = dummy;

    while (heap1 && heap2) {
        if (heap1->degree <= heap2->degree) {
            tail->sibling = heap1;
            heap1 = heap1->sibling;
        } else {
            tail->sibling = heap2;
            heap2 = heap2->sibling;
        }
        tail = tail->sibling;
    }

    tail->sibling = (heap1) ? heap1 : heap2;
    return dummy->sibling;
}

Node* unionHeaps(Node* heap1, Node* heap2) {
    Node* mergedHeap = mergeHeaps(heap1, heap2);

    if (!mergedHeap)
        return nullptr;

    Node* prev_x = nullptr;
    Node* x = mergedHeap;
    Node* next_x = x->sibling;

    while (next_x) {
        if (x->degree != next_x->degree || (next_x->sibling &&
next_x->sibling->degree == x->degree)) {
            prev_x = x;
            x = next_x;
        } else {
            if (x->key <= next_x->key) {
                x->sibling = next_x->sibling;
                mergeTrees(x, next_x);
            } else {
                if (!prev_x)
                    mergedHeap = next_x;
                else
                    prev_x->sibling = next_x;

                mergeTrees(next_x, x);
                x = next_x;
            }
        }
        next_x = x->sibling;
    }

    return mergedHeap;
}

Node* insert(Node* heap, int key) {
    Node* newNode = createNode(key);
    return unionHeaps(heap, newNode);
}

int getMin(Node* heap) {
    int minKey = INT_MAX;
    Node* curr = heap;
    while (curr) {
        if (curr->key < minKey)
            minKey = curr->key;
        curr = curr->sibling;
    }
    return minKey;
}

Node* extractMin(Node* heap) {
    if (!heap)
        return nullptr;

    int minKey = INT_MAX;
    Node* minNode = nullptr;
    Node* prev = nullptr;
    Node* curr = heap;
    Node* prevMin = nullptr;

    while (curr) {
        if (curr->key < minKey) {
            minKey = curr->key;
            minNode = curr;
            prevMin = prev;
        }
        prev = curr;
        curr = curr->sibling;
    }

    if (prevMin)
        prevMin->sibling = minNode->sibling;
    else
        heap = minNode->sibling;

    Node* child = minNode->child;
    Node* prevChild = nullptr;
    while (child) {
        child->parent = nullptr;
        Node* nextChild = child->sibling;
        child->sibling = prevChild;
        prevChild = child;
        child = nextChild;
    }

    return unionHeaps(heap, prevChild);
}

void printHeap(Node* heap) {
    cout << "Binomial Heap: ";
    while (heap) {
        cout << heap->key << "(" << heap->degree << ") ";
        heap = heap->sibling;
    }
    cout << endl;
}


int main() {
    Node* heap = nullptr;

    heap = insert(heap, 100);
    heap = insert(heap, 10);
    heap = insert(heap, 55);
    heap = insert(heap, 34);
    heap = insert(heap, 50);
    heap = insert(heap, 39);

    printHeap(heap);

    cout << "Minimum key: " << getMin(heap) << endl;

    heap = extractMin(heap);
    cout << "After extracting minimum key: ";
    printHeap(heap);

    return 0;
}
```

Output:



```
Binomial Heap: 3(2)
Minimum key: 3
After extracting minimum key: Binomial Heap: 5(0) 10(1)
```



```
Binomial Heap: 39(1) 10(2)
Minimum key: 10
After extracting minimum key: Binomial Heap: 100(0) 34(2)
```

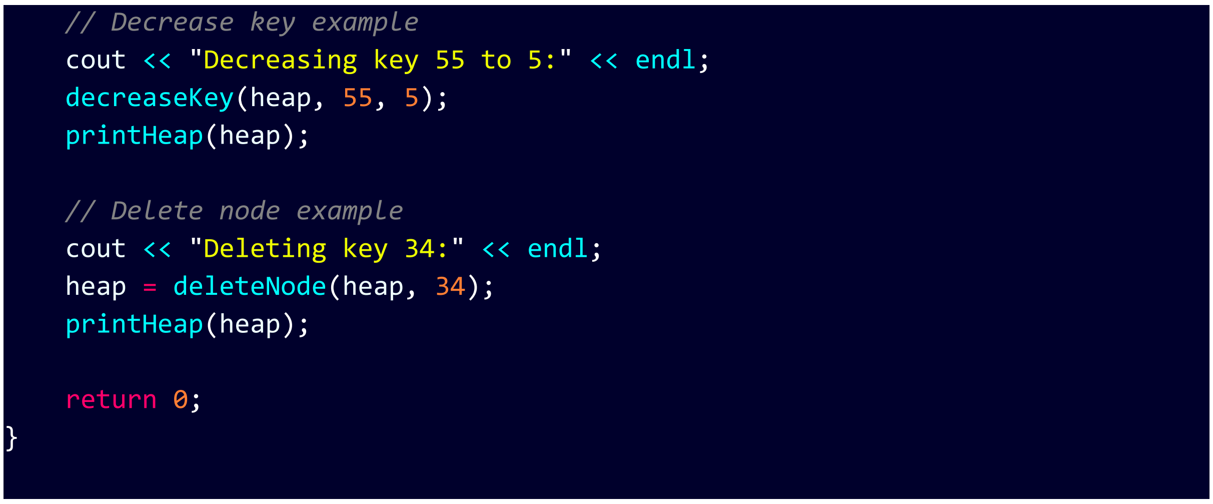
Lab Program 10: Write a program to implement the following functions on a Binomial heap:

1. delete(H): Like Binary Heap, the 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 decreased key with it parent and if the parent's key is more, we swap keys and recur for a parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node.

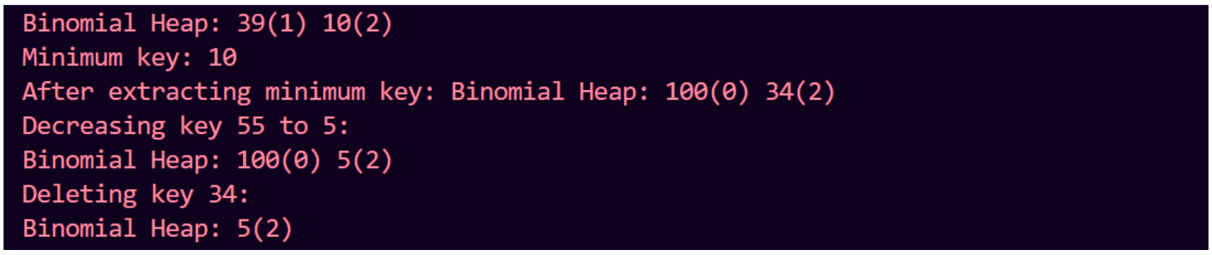
```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

struct Node {
    int key;
    int degree;
    Node* parent;
    Node* child;
    Node* sibling;
};

Node* createNode(int key) {
    Node* newNode = new Node;
    newNode->key = key;
    newNode->degree = 0;
    newNode->parent = nullptr;
    newNode->child = nullptr;
    newNode->sibling = nullptr;
    return newNode;
}


Node* mergeTrees(Node* tree1, Node* tree2) {
    if (tree1->key > tree2->key)
        swap(tree1, tree2);

    tree2->parent = tree1;
    tree2->sibling = tree1->child;
    tree1->child = tree2;
    tree1->degree++;

    return tree1;
}

Node* mergeHeaps(Node* heap1, Node* heap2) {
    Node* dummy = createNode(INT_MIN);
    Node* tail = dummy;

    while (heap1 && heap2) {
        if (heap1->degree <= heap2->degree) {
            tail->sibling = heap1;
            heap1 = heap1->sibling;
        } else {
            tail->sibling = heap2;
            heap2 = heap2->sibling;
        }
        tail = tail->sibling;
    }

    tail->sibling = (heap1) ? heap1 : heap2;
    return dummy->sibling;
}

Node* unionHeaps(Node* heap1, Node* heap2) {
    Node* mergedHeap = mergeHeaps(heap1, heap2);

    if (!mergedHeap)
        return nullptr;

    Node* prev_x = nullptr;
    Node* x = mergedHeap;
    Node* next_x = x->sibling;

    while (next_x) {
        if (x->degree != next_x->degree || (next_x->sibling &&
next_x->sibling->degree == x->degree)) {
            prev_x = x;
            x = next_x;
        } else {
            if (x->key <= next_x->key) {
                x->sibling = next_x->sibling;
                mergeTrees(x, next_x);
            } else {
                if (!prev_x)
                    mergedHeap = next_x;
                else
                    prev_x->sibling = next_x;

                mergeTrees(next_x, x);
                x = next_x;
            }
        }
        next_x = x->sibling;
    }
    return mergedHeap;
}

Node* insert(Node* heap, int key) {
    Node* newNode = createNode(key);
    return unionHeaps(heap, newNode);
}



Node* extractMin(Node* heap) {
    if (!heap)
        return nullptr;

    int minKey = INT_MAX;
    Node* minNode = nullptr;
    Node* prev = nullptr;
    Node* curr = heap;
    Node* prevMin = nullptr;

    while (curr) {
        if (curr->key < minKey) {
            minKey = curr->key;
            minNode = curr;
            prevMin = prev;
        }
        prev = curr;
        curr = curr->sibling;
    }

    if (prevMin)
        prevMin->sibling = minNode->sibling;
    else
        heap = minNode->sibling;

    Node* child = minNode->child;
    Node* prevChild = nullptr;
    while (child) {
        child->parent = nullptr;
        Node* nextChild = child->sibling;
        child->sibling = prevChild;
        prevChild = child;
        child = nextChild;
    }

    return unionHeaps(heap, prevChild);
}
```

```cpp
Node* findNode(Node* heap, int key) {
    if (!heap)
        return nullptr;
    if (heap->key == key)
        return heap;

    Node* res = findNode(heap->child, key);
    if (res)
        return res;

    return findNode(heap->sibling, key);
}

void decreaseKey(Node* heap, int oldKey, int newKey) {
    Node* targetNode = findNode(heap, oldKey);
    if (!targetNode)
        return;

    targetNode->key = newKey;

    while (targetNode->parent && targetNode->key < targetNode->parent->key) {
        swap(targetNode->key, targetNode->parent->key);
        targetNode = targetNode->parent;
    }
}

Node* deleteNode(Node* heap, int keyToDelete) {
    decreaseKey(heap, keyToDelete, INT_MIN);
    return extractMin(heap);
}

void printHeap(Node* heap) {
    cout << "Binomial Heap: ";
    while (heap) {
        cout << heap->key << "(" << heap->degree << ") ";
        heap = heap->sibling;
    }
    cout << endl;
}

int main() {
    Node* heap = nullptr;


    heap = insert(heap, 55);
    heap = insert(heap, 34);
    heap = insert(heap, 50);
    heap = insert(heap, 39);
```

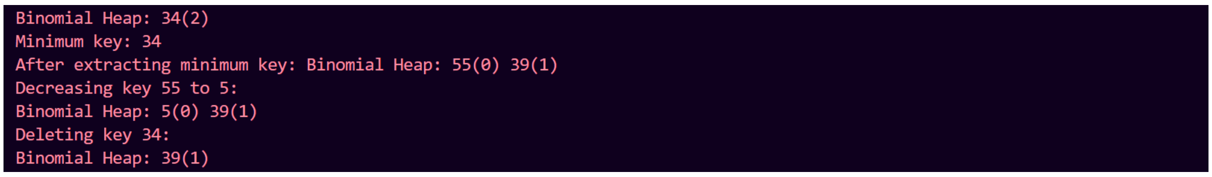


```
    // Decrease key example
    cout << "Decreasing key 55 to 5:" << endl;
    decreaseKey(heap, 55, 5);
    printHeap(heap);

    // Delete node example
    cout << "Deleting key 34:" << endl;
    heap = deleteNode(heap, 34);
    printHeap(heap);

    return 0;
}
```

Output:



```
Binomial Heap: 39(1) 10(2)
Minimum key: 10
After extracting minimum key: Binomial Heap: 100(0) 34(2)
Decreasing key 55 to 5:
Binomial Heap: 100(0) 5(2)
Deleting key 34:
Binomial Heap: 5(2)
```



```
Binomial Heap: 34(2)
Minimum key: 34
After extracting minimum key: Binomial Heap: 55(0) 39(1)
Decreasing key 55 to 5:
Binomial Heap: 5(0) 39(1)
Deleting key 34:
Binomial Heap: 39(1)
```