VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

ADVANCED DATA STRUCTURES (22CS5PEADS)

Submitted by

NISHANT KUMAR (1BM21CS117)

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 NISHANT KUMAR(1BM21CS117), 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 2023-24. 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. Rajeshwari Madli   **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 structure 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 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.

Code:

```
#include <cstdint>
#include <iostream>


struct Node
    {int data;


    Node* both;
};


class XORLinkedList
{private:
    Node* head;
    Node* tail;


    Node* XOR(Node* a, Node* b);


public:


    XORLinkedList();
```

```cpp
    void insert_at_head(int data);

    void insert_at_tail(int data);

    void delete_from_head();

    void delete_from_tail();

    void print_list();
};

XORLinkedList::XORLinkedList()
{
    head = tail = nullptr;

}

Node* XORLinkedList::XOR(Node* a, Node* b)
{
    return (

        Node*)((uintptr_t)(a) ^ (uintptr_t)(b));
}

void XORLinkedList::insert_at_head(int data)
```

```cpp
{
    Node* new_node = new Node();
    new_node->data = data;
    new_node->both = XOR(nullptr, head);

    if (head)
        { head-
        >both
            = XOR(new_node, XOR(head->both, nullptr));
    }
    else {
        tail = new_node;
    }

    head = new_node;
}

void XORLinkedList::insert_at_tail(int data)
{
    Node* new_node = new Node();
    new_node->data = data;
    new_node->both = XOR(tail, nullptr);

    if (tail)
        { tail-
        >both
            = XOR(XOR(tail->both, nullptr), new_node);
    }
    else {
        head = new_node;
```

```
    }
```

```
    }
```

```cpp
        tail = new_node;

}


void XORLinkedList::delete_from_head()

{

    if (head) {

        Node* next = XOR(head->both, nullptr);

        delete head;

        head = next;


        if (next) {

            next->both = XOR(next->both, head);

        }

        else {


            tail = nullptr;

        }

    }

}


void XORLinkedList::delete_from_tail()

{

    if (tail) {

        Node* prev = XOR(tail->both, nullptr);

        delete tail;

        tail = prev;


        if (prev) {
```

```cpp
            prev->both = XOR(prev->both, tail);
        }
        else {


            head = nullptr;
        }
    }
}


void XORLinkedList::print_list()
{
    Node* current = head;
    Node* prev = nullptr;
    while (current) {
        std::cout << current->data << " ";
        Node* next = XOR(prev, current->both);
        prev = current;
        current = next;
    }
    std::cout << std::endl;
}


int main()
{
    XORLinkedList list;
    list.insert_at_head(10);
    list.insert_at_head(20);
    list.insert_at_tail(30);
    list.insert_at_tail(40);
```
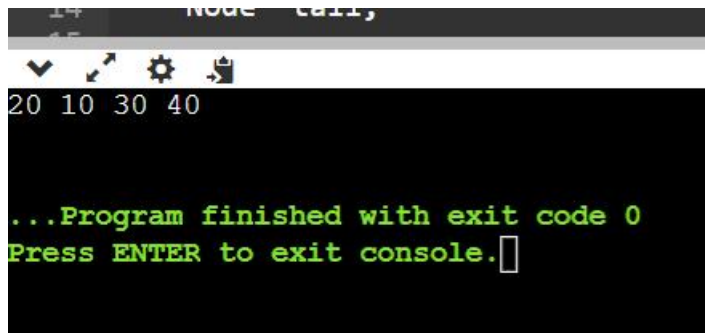
```
    list.print_list();

    return 0;

}
```

Output:

Output_1:

Output_2:

Lab Program 2:

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

Code:

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

// Class to implement node
class Node {
public:
    int key;
    Node** forward;
    Node(int, int);
};

Node::Node(int key, int level)
    {this->key = key;
    forward = new Node*[level+1];
    memset(forward, 0, sizeof(Node*) * (level+1));
}

// Class for Skip list
class SkipList {
    int MAXLVL;
    float P;
    int level;
    Node* header;
public:
    SkipList(int, float);
```

```cpp
    int randomLevel();

    Node* createNode(int, int);

    void insertElement(int);

    void deleteElement(int);

    void searchElement(int);

    void displayList();
};


SkipList::SkipList(int MAXLVL, float P)
    {this->MAXLVL = MAXLVL;
    this->P = P;
    level = 0;
    header = new Node(-1, MAXLVL);
}


int SkipList::randomLevel() {
    float r = (float)rand() / RAND_MAX;
    int lvl = 0;
    while (r < P && lvl < MAXLVL)
        {lvl++;
        r = (float)rand() / RAND_MAX;
    }
    return lvl;
}


Node* SkipList::createNode(int key, int level)
    {Node* n = new Node(key, level);
    return n;
}
```

```cpp
void SkipList::insertElement(int key)
  {Node* current = header;
  Node* update[MAXLVL + 1];
  memset(update, 0, sizeof(Node*) * (MAXLVL + 1));


  for (int i = level; i >= 0; i--) {
     while (current->forward[i] != NULL && current->forward[i]->key < key)
        current = current->forward[i];
     update[i] = current;
  }


  current = current->forward[0];


  if (current == NULL || current->key != key)
     {int rlevel = randomLevel();


     if (rlevel > level) {
        for (int i = level + 1; i < rlevel + 1; i++)
           update[i] = header;
        level = rlevel;
     }


     Node* n = createNode(key, rlevel);


     for (int i = 0; i <= rlevel; i++) {
        n->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = n;
     }
```

```cpp
        cout << "Successfully Inserted key " << key << "\n";
    }
}


void SkipList::deleteElement(int key)
    {Node* current = header;
    Node* update[MAXLVL + 1];
    memset(update, 0, sizeof(Node*) * (MAXLVL + 1));


    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != NULL && current->forward[i]->key < key)
            current = current->forward[i];
        update[i] = current;
    }


    current = current->forward[0];


    if (current != NULL && current->key == key)
        {for (int i = 0; i <= level; i++) {
            if (update[i]->forward[i] != current)
                break;
            update[i]->forward[i] = current->forward[i];
        }


        while (level > 0 && header->forward[level] == 0)
            level--;
        cout << "Successfully deleted key " << key << "\n";
    }
}
```

```cpp
void SkipList::searchElement(int key)
  {Node* current = header;


  for (int i = level; i >= 0; i--) {
    while (current->forward[i] && current->forward[i]->key < key)
      current = current->forward[i];
  }


  current = current->forward[0];


  if (current && current->key == key)
    cout << "Found key: " << key << "\n";
}


void SkipList::displayList() {
  cout << "\n*****Skip List*****" << "\n";
  for (int i = 0; i <= level; i++) {
    Node* node = header->forward[i];
    cout << "Level " << i << ": ";
    while (node != NULL) {
      cout << node->key << " ";
      node = node->forward[i];
    }
    cout << "\n";
  }
}


int main() {
```

```cpp
    srand((unsigned)time(0));

    SkipList lst(3, 0.5);

    lst.insertElement(3);
    lst.insertElement(6);
    lst.insertElement(7);
    lst.insertElement(9);
    lst.insertElement(12);
    lst.insertElement(19);
    lst.insertElement(17);
    lst.insertElement(26);
    lst.insertElement(21);
    lst.insertElement(25);
    lst.displayList();

    lst.searchElement(19);
    lst.deleteElement(19);
    lst.displayList();
}
```
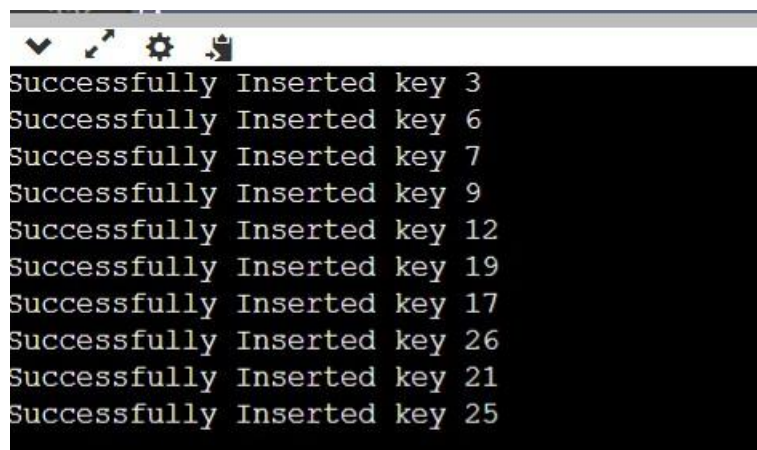
Output:

Output_1:


```
Successfully Inserted key 3
Successfully Inserted key 6
Successfully Inserted key 7
Successfully Inserted key 9
Successfully Inserted key 12
Successfully Inserted key 19
Successfully Inserted key 17
Successfully Inserted key 26
Successfully Inserted key 21
Successfully Inserted key 25
```

Output_2:

```
*****Skip List*****
Level 0: 3 6 7 9 12 17 19 21 25 26
Level 1: 12 17 19 21 25 26
Level 2: 19 25 26
Level 3: 26
Found key: 19
Successfully deleted key 19

*****Skip List*****
Level 0: 3 6 7 9 12 17 21 25 26
Level 1: 12 17 21 25 26
Level 2: 25 26
Level 3: 26


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 3:

Given a boolean 2D matrix, find the number of islands.Use disjoint sets to implement the above scenario.

Code:

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


class DisjointUnionSets
  { vector<int> rank,
  parent;int n;


public:
  DisjointUnionSets(int n)
    {rank.resize(n);
    parent.resize(n);
    this->n = n;
    makeSet();
  }


  void makeSet() {
    for (int i = 0; i < n; i++)
      parent[i] = i;
  }


  int find(int x) {
    if (parent[x] != x) {
      parent[x] = find(parent[x]);
    }
    return parent[x];
  }
```

```
    void Union(int x, int y)
      {int xRoot = find(x);
      int yRoot = find(y);


      if (xRoot == yRoot)
         return;


      if (rank[xRoot] < rank[yRoot])
         parent[xRoot] = yRoot;
      else if (rank[yRoot] < rank[xRoot])
         parent[yRoot] = xRoot;
      else {
         parent[yRoot] = xRoot;
         rank[xRoot] = rank[xRoot] + 1;

      }
   }
};


int countIslands(vector<vector<int>> a)
   {int n = a.size();
   int m = a[0].size();


   DisjointUnionSets* dus = new DisjointUnionSets(n * m);


   for (int j = 0; j < n; j++) {
      for (int k = 0; k < m; k++)
         {if (a[j][k] == 0)
            continue;
```

```cpp
        if (j + 1 < n && a[j + 1][k] == 1)
            dus->Union(j * m + k, (j + 1) * m + k);
        if (j - 1 >= 0 && a[j - 1][k] == 1)
            dus->Union(j * m + k, (j - 1) * m + k);
        if (k + 1 < m && a[j][k + 1] == 1)
            dus->Union(j * m + k, j * m + k + 1);
        if (k - 1 >= 0 && a[j][k - 1] == 1)
            dus->Union(j * m + k, j * m + k - 1);
        if (j + 1 < n && k + 1 < m && a[j + 1][k + 1] == 1)
            dus->Union(j * m + k, (j + 1) * m + k + 1);
        if (j + 1 < n && k - 1 >= 0 && a[j + 1][k - 1] == 1)
            dus->Union(j * m + k, (j + 1) * m + k - 1);
        if (j - 1 >= 0 && k + 1 < m && a[j - 1][k + 1] == 1)
            dus->Union(j * m + k, (j - 1) * m + k + 1);
        if (j - 1 >= 0 && k - 1 >= 0 && a[j - 1][k - 1] == 1)
            dus->Union(j * m + k, (j - 1) * m + k - 1);
    }
}

vector<int> c(n * m, 0);
int numberOfIslands = 0;
for (int j = 0; j < n; j++) {
    for (int k = 0; k < m; k++)
        {if (a[j][k] == 1) {
            int x = dus->find(j * m + k);
            if (c[x] == 0) {
                numberOfIslands++;
                c[x]++;
```

```cpp
        } else {
            c[x]++;
        }
      }
    }
  }
  return numberOfIslands;
}


int main()
  { vector<vector<int>> a1 = {
    {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}
  };
  cout << "Number of Islands is: " << countIslands(a1) << endl;


  vector<vector<int>> a2 = {
    {1, 0, 0, 1, 1},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 1, 1},
    {1, 1, 0, 0, 0},
    {0, 0, 1, 1, 1}
  };
  cout << "Number of Islands is: " << countIslands(a2) << endl;


  return 0;
```
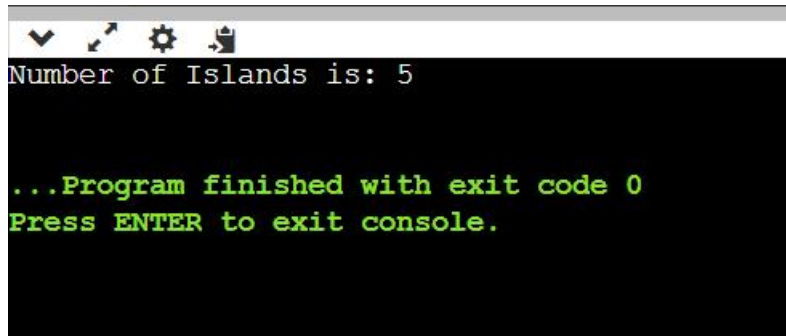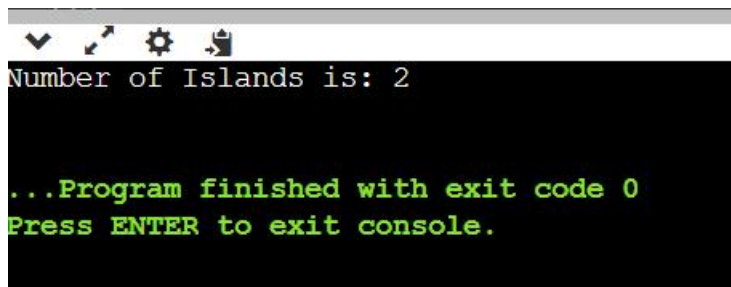
}

Output:

Output_1:

```
Number of Islands is: 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Output_2:

```
Number of Islands is: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 4:

Code:

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

```c
    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);
```

```c
}

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) {
```

```c
    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)
```

```cpp
    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;
```

```cpp
    root = insert(root, 9);

    root = insert(root, 5);

    root = insert(root, 10);

    root = insert(root, 0);

    root = insert(root, 6);

    root = insert(root, 11);

    root = insert(root, -1);

    root = insert(root, 1);

    root = insert(root, 2);


    cout << "Preorder traversal of the constructed AVL tree is \n";

    preOrder(root);


    root = deleteNode(root, 10);


    cout << "\nPreorder traversal after deletion of 10 \n";

    preOrder(root);


    return 0;
}
```
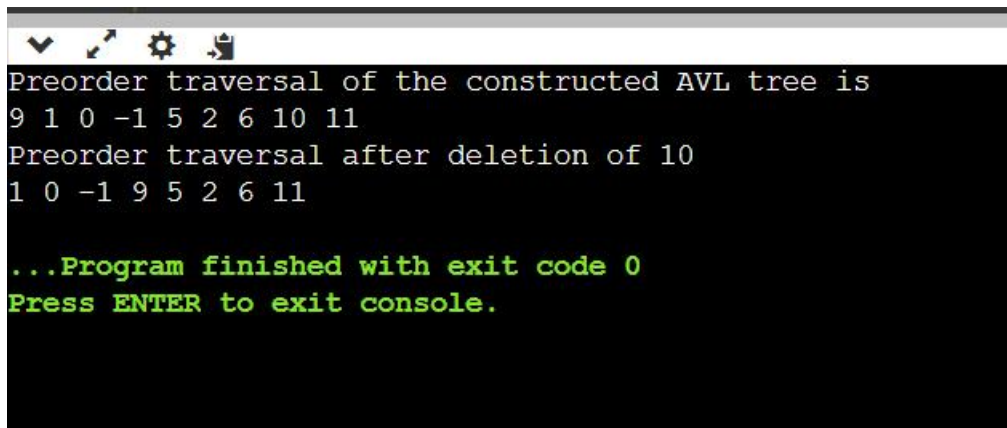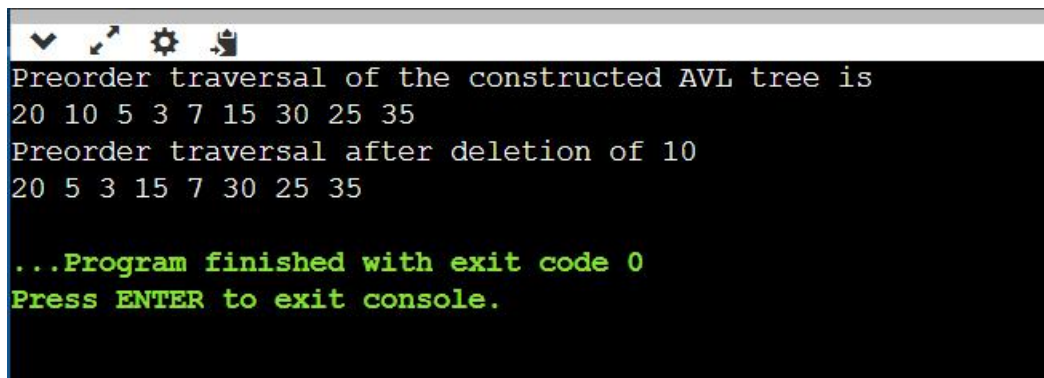
Output:

Output_1;

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11

...Program finished with exit code 0
Press ENTER to exit console.
```

Output_2:


```
Preorder traversal of the constructed AVL tree is
20 10 5 3 7 15 30 25 35
Preorder traversal after deletion of 10
20 5 3 15 7 30 25 35

...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 5:

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

Code:

```cpp
#include <iostream>

using namespace std;

class Node
{public:
    int data1;
    int data2;
    Node* left;
    Node* middle;
    Node* right;
    Node(int data) {
        data1 = data;
        data2 = -1;
        left = middle = right = nullptr;
    }
};

class TwoThreeTree
{private:
    Node* root;

    Node* insert(Node* root, int data)
        {if (root == nullptr) {
            root = new Node(data);
```

```cpp
        return root;
    }


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


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


    return root;
}


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


    if (data < root->data1) {
```

```cpp
        root->left = remove(root->left, data);
    } else if (data > root->data1 && (root->data2 == -1 || data < root->data2))
        {root->middle = remove(root->middle, data);
    } else if (data > root->data1 && (root->data2 != -1 && data == root->data2))
        {root->right = remove(root->right, data);
    } else {
        if (root->left == nullptr && root->middle == nullptr && root->right == nullptr)
            {delete root;
            return nullptr;
        }

        if (root->left == nullptr && root->middle != nullptr && root->right != nullptr)
            {root->data1 = root->data2;
            root->data2 = -1;
            root->left = root->middle;
            root->middle = root->right;
            root->right = nullptr;
        } else if (root->left != nullptr && root->middle == nullptr && root->right != nullptr)
{
            root->data1 = root->left->data2;
            root->left->data2 = -1;
            root->middle = root->right;
            root->right = nullptr;
        } else if (root->left != nullptr && root->middle != nullptr && root->right == nullptr)
{
            root->data2 = -1;
            root->right = root->middle;
            root->middle = nullptr;
        } else {
            Node* temp = findMin(root->right);
```

```cpp
            root->data1 = temp->data1;

            root->right = remove(root->right, temp->data1);

        }

    }


    return root;

}


Node* findMin(Node* node)

    {Node* current = node;

    while (current->left != nullptr)

        current = current->left;

    return current;

}


public:

    TwoThreeTree()

        {root = nullptr;

    }


    void insert(int data) {

        root = insert(root, data);

    }


    void remove(int data) {

        root = remove(root, data);

    }


    void printInorder(Node* node) {
```

```cpp
        if (node == nullptr)
            return;
        printInorder(node->left);
        cout << node->data1 << " ";
        if (node->data2 != -1)
            cout << node->data2 << " ";
        printInorder(node->middle);
        printInorder(node->right);
    }


    void printInorder()
        {printInorder(root)
        ;
    }
};


int main()
    { TwoThreeTree
    tree;tree.insert(10);
    tree.insert(20);
    tree.insert(5);
    tree.insert(6);
    tree.insert(12);


    cout << "Inorder traversal of the 2-3 tree: ";
    tree.printInorder();
    cout << endl;


    tree.remove(6);
    cout << "Inorder traversal after removing 6: ";
```

```
    tree.printInorder();

    cout << endl;


    return 0;
}
```
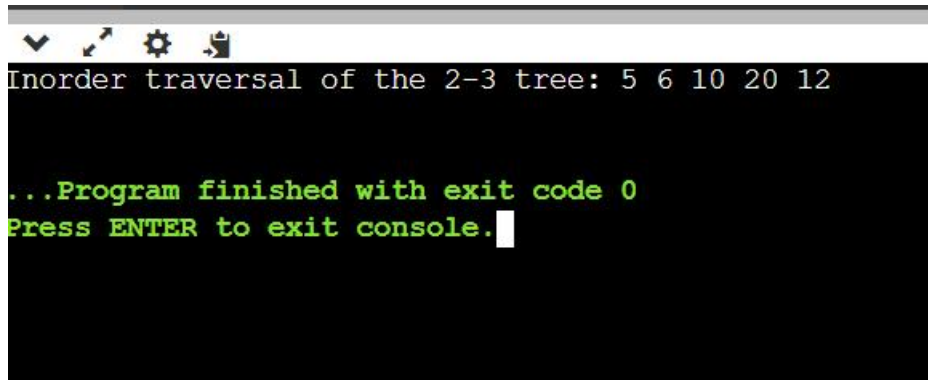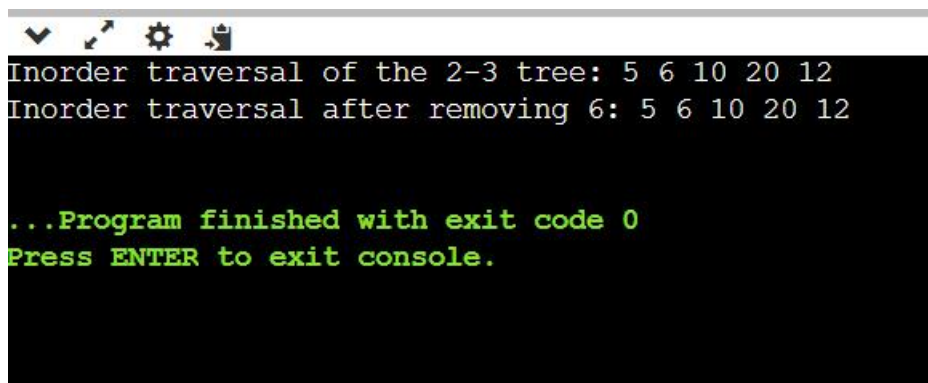
Outputs:

Output_1:


```
Inorder traversal of the 2-3 tree: 5 6 10 20 12



...Program finished with exit code 0
Press ENTER to exit console.
```

Output_2:


```
Inorder traversal of the 2-3 tree: 5 6 10 20 12
Inorder traversal after removing 6: 5 6 10 20 12


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 6:

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

Code:

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

enum Color { RED, BLACK };

struct Node
   { int data;
   Color color;
   Node *left, *right, *parent;

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

class RedBlackTree
{private:
   Node *root;

   void rotateLeft(Node *&root, Node *&ptr)
      {Node *ptr_right = ptr->right;
      ptr->right = ptr_right->left;
```

```cpp
        if (ptr->right != nullptr)

            ptr->right->parent = ptr;



        ptr_right->parent = ptr->parent;



        if (ptr->parent == nullptr)

            root = ptr_right;

        else if (ptr == ptr->parent->left)

            ptr->parent->left = ptr_right;

        else

            ptr->parent->right = ptr_right;



        ptr_right->left = ptr;

        ptr->parent = ptr_right;

    }


    void rotateRight(Node *&root, Node *&ptr)

        {Node *ptr_left = ptr->left;

        ptr->left = ptr_left->right;



        if (ptr->left != nullptr)

            ptr->left->parent = ptr;



        ptr_left->parent = ptr->parent;



        if (ptr->parent == nullptr)

            root = ptr_left;

        else if (ptr == ptr->parent->left)

            ptr->parent->left = ptr_left;
```

```cpp
    else
      ptr->parent->right = ptr_left;


    ptr_left->right = ptr;

    ptr->parent = ptr_left;

}


void fixInsertRBTree(Node *&root, Node *&ptr)
  {Node *parent_ptr = nullptr;

  Node *grand_parent_ptr = nullptr;


  while (ptr != root && ptr->color != BLACK && ptr->parent->color == RED)
    {parent_ptr = ptr->parent;

    grand_parent_ptr = ptr->parent->parent;


    if (parent_ptr == grand_parent_ptr->left)
      { Node *uncle_ptr = grand_parent_ptr->right;


      if (uncle_ptr != nullptr && uncle_ptr->color == RED)
        {grand_parent_ptr->color = RED;

        parent_ptr->color = BLACK;

        uncle_ptr->color = BLACK;

        ptr = grand_parent_ptr;

      } else {
        if (ptr  ==  parent_ptr->right)
          {            rotateLeft(root,
          parent_ptr);ptr = parent_ptr;

          parent_ptr = ptr->parent;

        }
```

```cpp
            rotateRight(root, grand_parent_ptr);

            swap(parent_ptr->color, grand_parent_ptr->color);

            ptr = parent_ptr;

        }

    } else {

        Node *uncle_ptr = grand_parent_ptr->left;


        if (uncle_ptr != nullptr && uncle_ptr->color == RED)

            {grand_parent_ptr->color = RED;

            parent_ptr->color = BLACK;

            uncle_ptr->color = BLACK;

            ptr = grand_parent_ptr;

        } else {

            if (ptr == parent_ptr->left)

                { rotateRight(root,

                parent_ptr);ptr = parent_ptr;

                parent_ptr = ptr->parent;

            }


            rotateLeft(root, grand_parent_ptr);

            swap(parent_ptr->color, grand_parent_ptr->color);

            ptr = parent_ptr;

        }

    }

}


root->color = BLACK;

}
```

```cpp
public:
  RedBlackTree() : root(nullptr) {}

  void insert(const int data) {
    Node *new_node = new Node(data);
    if (root == nullptr) {
      root = new_node;
      root->color = BLACK;
      return;
    }

    Node *current = root;
    Node *parent = nullptr;
    while (current != nullptr) {
      parent = current;
      if (new_node->data < current->data)
        current = current->left;
      else
        current = current->right;
    }

    new_node->parent = parent;
    if (parent == nullptr)
      root = new_node;
    else if (new_node->data < parent->data)
      parent->left = new_node;
    else
      parent->right = new_node;
```

```cpp
        fixInsertRBTree(root, new_node);
    }

    void inorder(Node *node)
      {if (node == nullptr)
          return;

      inorder(node->left);
      cout << node->data << " ";
      inorder(node->right);
    }

    void inorder()
      {inorder(root)
      ;
    }
};

int main()
    { RedBlackTree
    rbTree;

    // Inserting elements
    rbTree.insert(7);
    cout << "Inserted: 7" << endl;
    cout << "Inorder Traversal of the tree: ";
    rbTree.inorder();
    cout << endl;
```

```
rbTree.insert(10);
```

```cpp
    cout << "Inserted: 10" << endl;

    cout << "Inorder Traversal of the tree: ";

    rbTree.inorder();

    cout << endl;


    return 0;
}
```
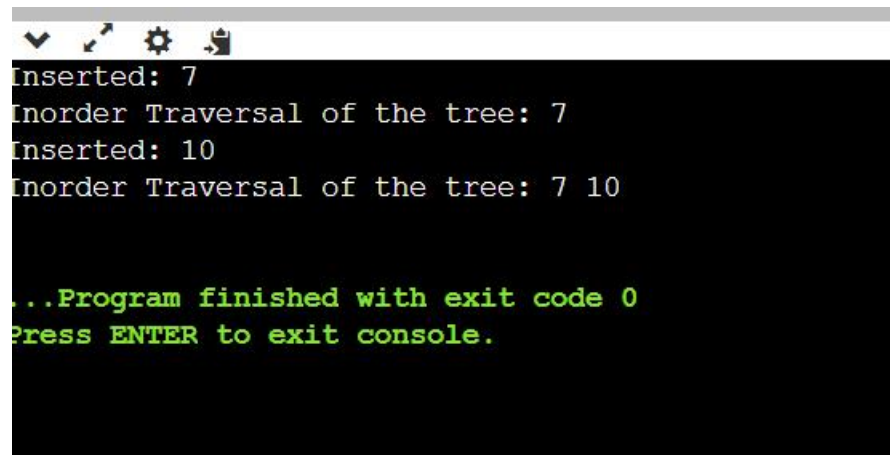
Outputs:

Output_1:


```
Inserted: 7
Inorder Traversal of the tree: 7
Inserted: 10
Inorder Traversal of the tree: 7 10


...Program finished with exit code 0
Press ENTER to exit console.
```

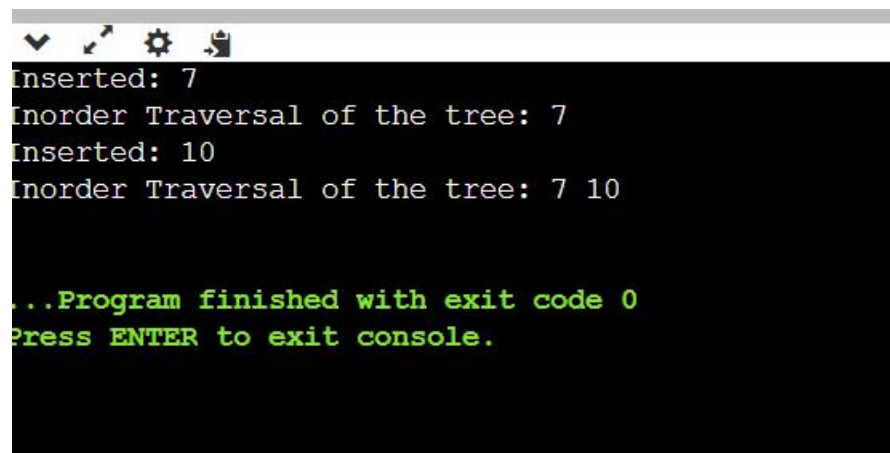Output_2:


```
Inserted: 7
Inorder Traversal of the tree: 7
Inserted: 10
Inorder Traversal of the tree: 7 10


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 7:

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

Code:

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

const int MAX_KEYS = 3; // Maximum number of keys in a node

struct BTreeNode {
    int keys[MAX_KEYS];
    BTreeNode* children[MAX_KEYS + 1];
    int num_keys;
    bool is_leaf;

    BTreeNode()
        { num_keys =
        0;is_leaf =
        true;
        for (int i = 0; i < MAX_KEYS + 1; ++i)
            children[i] = nullptr;
    }
};

void insert(BTreeNode* root, int key) {
    if (root->num_keys == MAX_KEYS) {
        // Split root
        BTreeNode* new_root = new BTreeNode();
        new_root->is_leaf = false;
        new_root->children[0] = root;
```

```cpp
        // Split the old root and move one key to the new root
        BTreeNode* new_child = new BTreeNode();
        new_child->is_leaf = root->is_leaf;
        new_child->num_keys = MAX_KEYS / 2;
        for (int i = 0; i < new_child->num_keys; ++i)
            { new_child->keys[i] = root->keys[MAX_KEYS / 2 +
            i];
            root->keys[MAX_KEYS / 2 + i] = 0; // Clear the old key
        }
        root->num_keys = MAX_KEYS / 2;


        new_root->keys[0] = new_child->keys[0];
        new_root->children[1] = new_child;


        root = new_root;
    }


    // Insert key into root or its child
    if (root->is_leaf) {
        // Find the position to insert the new key
        int i = root->num_keys - 1;
        while (i >= 0 && root->keys[i] > key)
            {root->keys[i + 1] = root->keys[i];
            i--;
        }
        root->keys[i + 1] = key;
        root->num_keys++;
    } else {
        // Find the child to insert the new key
        int i = root->num_keys - 1;
```

```cpp
        while (i >= 0 && root->keys[i] > key)
            i--;


        // Check if child is full
        if (root->children[i + 1]->num_keys == MAX_KEYS)
            insert(root->children[i + 1], key);
        else
            insert(root->children[i + 1], key);
    }
}


void printBTree(BTreeNode* root)
    {if (root) {
        for (int i = 0; i < root->num_keys; ++i)
            {cout << root->keys[i] << " ";
        }
        cout << endl;
        if (!root->is_leaf) {
            for (int i = 0; i <= root->num_keys; ++i)
                {printBTree(root->children[i]);
            }
        }
    }
}


int main() {
    BTreeNode* root = new BTreeNode();


    insert(root, 10);
```
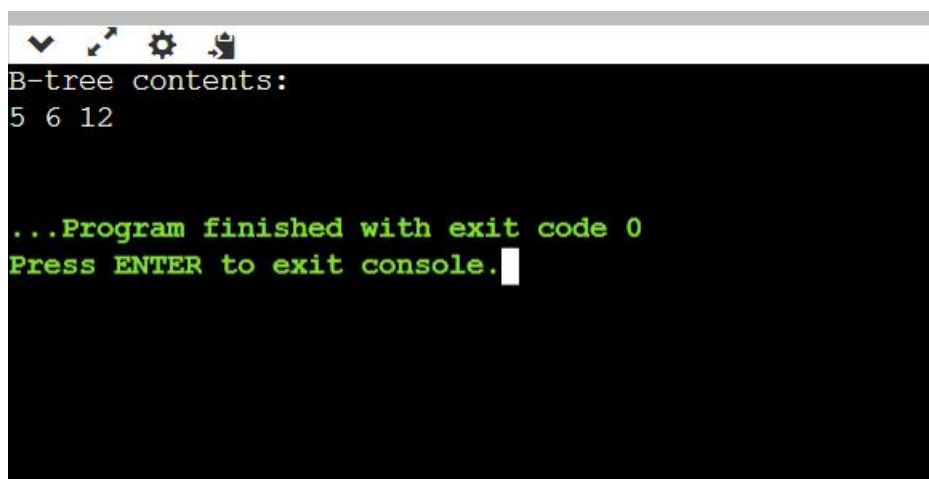
```cpp
    insert(root, 20);

    insert(root, 5);

    insert(root, 6);

    insert(root, 12);


    cout << "B-tree contents:" << endl;

    printBTree(root);


    return 0;

}
```
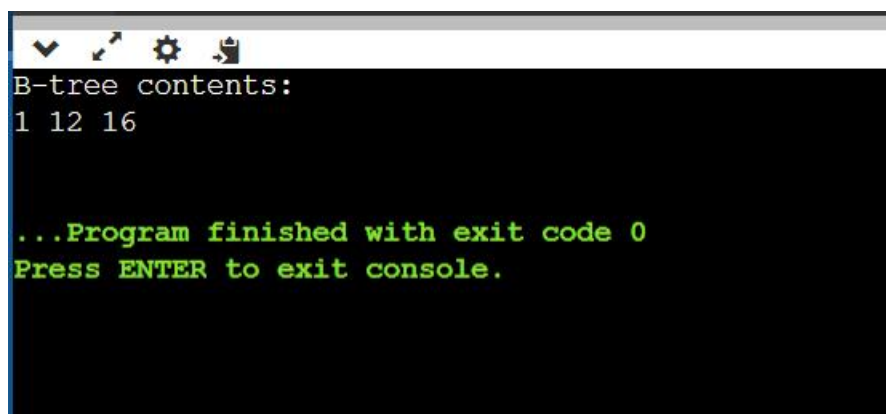
Outputs:

Output_1:

Output_2:

Lab Program 8:

Write a program to to implement functions of Dictionary using Hashing.

Code:

```cpp
#include <iostream>

#include <vector>

#include <list>


using namespace std;


// Class for the dictionary
class Dictionary { private:
    static const int TABLE_SIZE = 10;
    vector<list<pair<int, string>>> table;


    // Hash function
    int hashFunction(int key)
        { return key % TABLE_SIZE;
    }


public:
    // Constructor
    Dictionary() {
        table.resize(TABLE_SIZE);
    }


    // Function to insert a key-value pair
    void insert(int key, const string& value)
        {int index = hashFunction(key);
        table[index].push_back(make_pair(key, value));
```

```cpp
    }

    // Function to search for a value given a key
    string search(int key) {
        int index = hashFunction(key);
        for (auto& entry : table[index]) {
            if (entry.first == key)
                {return entry.second;
            }
        }
        return "Key not found";
    }
};

int main() {
    Dictionary dictionary;

    // Insert some key-value pairs
    dictionary.insert(1, "Apple");
    dictionary.insert(11, "Banana");
    dictionary.insert(21, "Orange");

    // Search for some keys
    cout << "Value for key 1: " << dictionary.search(1) << endl;
    cout << "Value for key 11: " << dictionary.search(11) << endl;
    cout << "Value for key 21: " << dictionary.search(21) << endl;
    cout << "Value for key 2: " << dictionary.search(2) << endl;

    return 0;
}
```
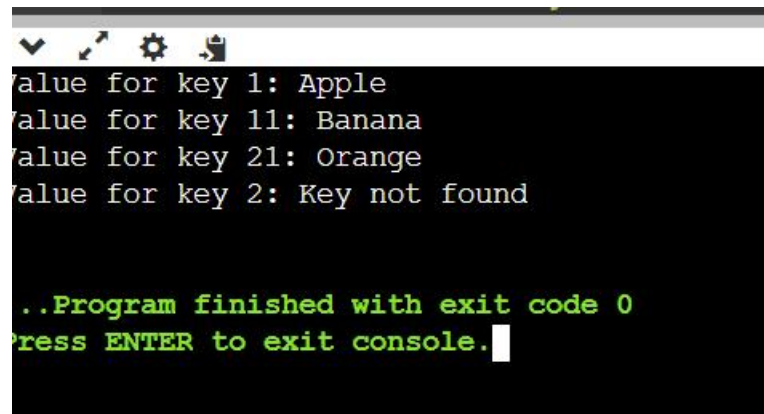
Outputs:
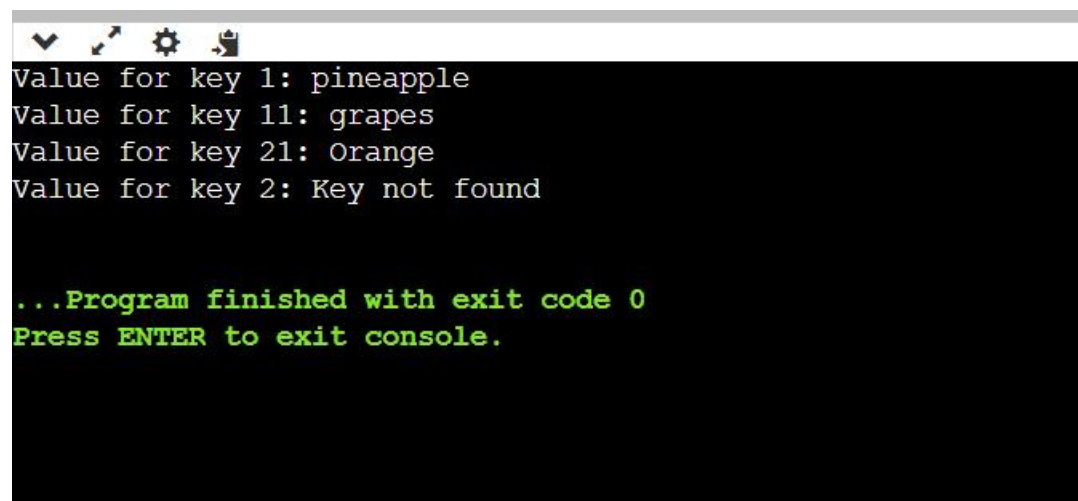
Output_1:


```
Value for key 1: Apple
Value for key 11: Banana
Value for key 21: Orange
Value for key 2: Key not found


..Program finished with exit code 0
Press ENTER to exit console.
```

Output_2:


```
Value for key 1: pineapple
Value for key 11: grapes
Value for key 21: Orange
Value for key 2: Key not found


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 9:

Write a program to implement the following functions on a Binomial heap:Insert(), getMin(), extractMin().

Code:

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

struct Node
{
  int data, degree;
  Node *child, *sibling, *parent;
};

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

Node* mergeBinomialTrees(Node *b1, Node *b2)
{

  if (b1->data > b2->data)
    swap(b1, b2);

  b2->parent = b1;
  b2->sibling = b1->child;
  b1->child = b2;
  b1->degree++;
```

```cpp
 return b1;
}

list<Node*> unionBionomialHeap(list<Node*> l1,
          list<Node*> l2)
{
 list<Node*> _new;
 list<Node*>::iterator it = l1.begin();
 list<Node*>::iterator ot = l2.begin();
 while (it!=l1.end() && ot!=l2.end())
  {

   if((*it)->degree <= (*ot)->degree)
    {
     _new.push_back(*it);
     it++;
    }

   else
    {
     _new.push_back(*ot);
     ot++;
    }
  }

   while (it != l1.end())
  {
   _new.push_back(*it);
   it++;
  }
  while (ot!=l2.end())
  {
```

```cpp
        _new.push_back(*ot);
      ot++;
    }
  return _new;
}


list<Node*> adjust(list<Node*> _heap)
{
  if (_heap.size() <= 1)
    return _heap;
  list<Node*> new_heap;
  list<Node*>::iterator it1,it2,it3;
  it1 = it2 = it3 = _heap.begin();

  if (_heap.size() == 2)
  {
    it2 = it1;
    it2++;
    it3 = _heap.end();
  }
  else
  {
    it2++;
    it3=it2;
    it3++;
  }
  while (it1 != _heap.end())
  {

    if (it2 == _heap.end())
      it1++;

    else if ((*it1)->degree < (*it2)->degree)
```

```
    {
     it1++;
     it2++;
     if(it3!=_heap.end())
       it3++;
    }


      else if (it3!=_heap.end() &&
      (*it1)->degree == (*it2)->degree &&
      (*it1)->degree == (*it3)->degree)
    {
     it1++;
     it2++;
     it3++;
    }


    else if ((*it1)->degree == (*it2)->degree)
    {
     Node *temp;
     *it1 = mergeBinomialTrees(*it1,*it2);
     it2 = _heap.erase(it2);
     if(it3 != _heap.end())
       it3++;
    }
  }
  return _heap;
}

// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
        Node *tree)
{
```

```cpp
    // creating a new heap i.e temp
    list<Node*> temp;

    // inserting Binomial Tree into heap
    temp.push_back(tree);

    // perform union operation to finally insert
    // Binomial Tree in original heap
    temp = unionBionomialHeap(_heap,temp);

    return adjust(temp);
}

list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;

    // making a binomial heap from Binomial Tree
    while (temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}

list<Node*> insert(list<Node*> _head, int key)
{
```

```cpp
  Node *temp = newNode(key);
  return insertATreeInHeap(_head,temp);
}
Node* getMin(list<Node*> _heap)
{
  list<Node*>::iterator it = _heap.begin();
  Node *temp = *it;
  while (it != _heap.end())
  {
    if ((*it)->data < temp->data)
      temp = *it;
    it++;
  }
  return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
  list<Node*> new_heap,lo;
  Node *temp;
  temp = getMin(_heap);
  list<Node*>::iterator it;
  it = _heap.begin();
  while (it != _heap.end())
  {
    if (*it != temp)
    {
      new_heap.push_back(*it);
    }
    it++;
  }
  lo = removeMinFromTreeReturnBHeap(temp);
  new_heap = unionBionomialHeap(new_heap,lo);
```

```cpp
    new_heap = adjust(new_heap);
    return new_heap;
}


// print function for Binomial Tree
void printTree(Node *h)
{
  while (h)
  {
    cout << h->data << " ";
    printTree(h->child);
    h = h->sibling;
  }
}


// print function for binomial heap
void printHeap(list<Node*> _heap)
{
  list<Node*> ::iterator it;
  it = _heap.begin();
  while (it != _heap.end())
  {
    printTree(*it);
    it++;
  }
}



// Driver program to test above functions
int main()
{
  int ch,key;
  list<Node*> _heap;
```

```cpp
// Insert data in the heap
_heap = insert(_heap,10);
_heap = insert(_heap,20);
_heap = insert(_heap,30);

cout << "Heap elements after insertion:\n";
printHeap(_heap);

Node *temp = getMin(_heap);
cout << "\nMinimum element of heap "
  << temp->data << "\n";
_heap = extractMin(_heap);
cout << "Heap after deletion of minimum element\n";
printHeap(_heap);
return 0;
}
```

Outputs:

Output_1:

Output_2:

Lab Program 10:

Write a program to implement the following functions on a Binomial heap: delete() and decrease().

Code:

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

using namespace std;


// Structure of Node
struct Node {

        int val, degree;

        Node *parent, *child, *sibling;

};
Node* root = NULL;


// link two heaps by making h1 a child
// of h2.
int binomialLink(Node* h1, Node* h2)
{

        h1->parent = h2;

        h1->sibling = h2->child;

        h2->child = h1;

        h2->degree = h2->degree + 1;

}


// create a Node
Node* createNode(int n)
{

        Node* new_node = new Node;

        new_node->val = n;

        new_node->parent = NULL;
```

```
        new_node->sibling = NULL;

        new_node->child = NULL;

        new_node->degree = 0;

        return new_node;

}


// This function merge two Binomial Trees

Node* mergeBHeaps(Node* h1, Node* h2)

{

        if (h1 == NULL)

                return h2;

        if (h2 == NULL)

                return h1;


        // define a Node

        Node* res = NULL;


        // check degree of both Node i.e.

        // which is greater or smaller

        if (h1->degree <= h2->degree)

                res = h1;


        else if (h1->degree > h2->degree)

                res = h2;


        // traverse till if any of heap gets empty

        while (h1 != NULL && h2 != NULL) {

                // if degree of h1 is smaller, increment h1

                if (h1->degree < h2->degree)
```

```
                          h1 = h1->sibling;


               // Link h1 with h2 in case of equal degree
               else if (h1->degree == h2->degree) {

                          Node* sib = h1->sibling;

                          h1->sibling = h2;

                          h1 = sib;

               }


               // if h2 is greater
               else {

                          Node* sib = h2->sibling;

                          h2->sibling = h1;

                          h2 = sib;

               }
        }
        return res;
}
Node* unionBHeaps(Node* h1, Node* h2)

{

        if (h1 == NULL && h2 == NULL)

                return NULL;


        Node* res = mergeBHeaps(h1, h2);


        // Traverse the merged list and set

        // values according to the degree of

        // Nodes

        Node *prev = NULL, *curr = res, *next = curr->sibling;
```

```
        while (next != NULL) {
                if ((curr->degree != next->degree)
                        || ((next->sibling != NULL)
                                && (next->sibling)->degree
                                        == curr->degree)) {
                        prev = curr;
                        curr = next;
                }

                else {
                        if (curr->val <= next->val) {
                                curr->sibling = next->sibling;
                                binomialLink(next, curr);
                        }
                        else {
                                if (prev == NULL)
                                        res = next;
                                else
                                        prev->sibling = next;
                                binomialLink(curr, next);
                                curr = next;
                        }
                }
                next = curr->sibling;
        }
        return res;
}


// Function to insert a Node
```

```cpp
void binomialHeapInsert(int x)
{
        // Create a new node and do union of
        // this node with root
        root = unionBHeaps(root, createNode(x));
}

// Function to display the Nodes
void display(Node* h)
{
        while (h) {
                cout << h->val << " ";
                display(h->child);
                h = h->sibling;
        }
}

// Function to reverse a list
// using recursion.
int revertList(Node* h)
{
        if (h->sibling != NULL)
                { revertList(h-
                >sibling);
                (h->sibling)->sibling = h;
        }
        else
                root = h;
}
```

```
// Function to extract minimum value
Node* extractMinBHeap(Node* h)
{
        if (h == NULL)
                return NULL;


        Node* min_node_prev = NULL;
        Node* min_node = h;


        // Find minimum value
        int min = h->val;
        Node* curr = h;
        while (curr->sibling != NULL) {
                if ((curr->sibling)->val < min)
                        { min = (curr->sibling)-
                        >val;min_node_prev =
                        curr; min_node = curr-
                        >sibling;
                }
                curr = curr->sibling;
        }


        // If there is a single Node
        if (min_node_prev == NULL && min_node->sibling == NULL)
                h = NULL;


        else if (min_node_prev == NULL)
                h = min_node->sibling;


        // Remove min node from list
```

```
        else

                min_node_prev->sibling = min_node->sibling;


        // Set root (which is global) as children

        // list of min node

        if (min_node->child != NULL)

                { revertList(min_node->child);

                (min_node->child)->sibling = NULL;

        }

        else

                root = NULL;

        delete min_node;

        // Do union of root h and children

        return unionBHeaps(h, root);

}


// Function to search for an element

Node* findNode(Node* h, int val)

{

        if (h == NULL)

                return NULL;


        // check if key is equal to the root's data

        if (h->val == val)

                return h;


        // Recur for child

        Node* res = findNode(h->child, val);

        if (res != NULL)
```

```
                return res;


        return findNode(h->sibling, val);

}


// Function to decrease the value of old_val
// to new_val
void decreaseKeyBHeap(Node* H, int old_val, int new_val)
{
        // First check element present or not
        Node* node = findNode(H, old_val);


        // return if Node is not present
        if (node == NULL)
                return;


        // Reduce the value to the minimum
        node->val = new_val;
        Node* parent = node->parent;


        // Update the heap according to reduced value
        while (parent != NULL && node->val < parent->val)
                {swap(node->val, parent->val);
                node = parent;
                parent = parent->parent;
        }
}


// Function to delete an element
```

```cpp
Node* binomialHeapDelete(Node* h, int val)
{
        // Check if heap is empty or not
        if (h == NULL)
                return NULL;


        // Reduce the value of element to minimum
        decreaseKeyBHeap(h, val, INT_MIN);


        // Delete the minimum element from heap
        return extractMinBHeap(h);
}


// Driver code
int main()
{
        // Note that root is global
        binomialHeapInsert(10);
        binomialHeapInsert(20);
        binomialHeapInsert(30);
        binomialHeapInsert(40);
        binomialHeapInsert(50);


        cout << "The heap is:\n";
        display(root);


        // Delete a particular element from heap
        root = binomialHeapDelete(root, 10);
```

```
        cout << "\nAfter deleting 10, the heap is:\n";


        display(root);


        return 0;
}
```
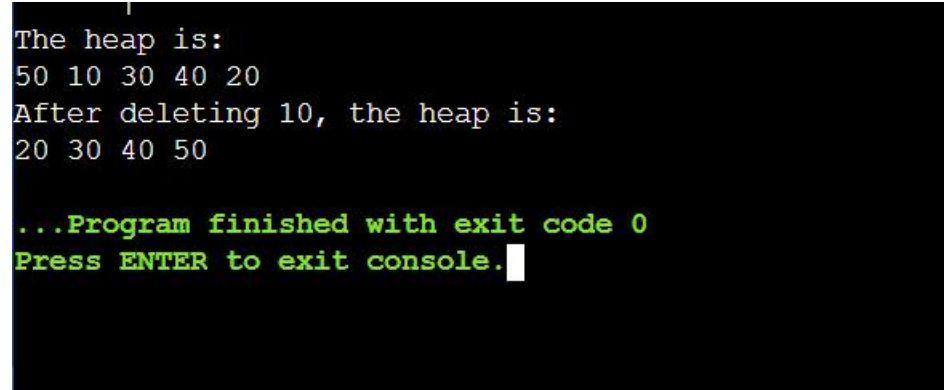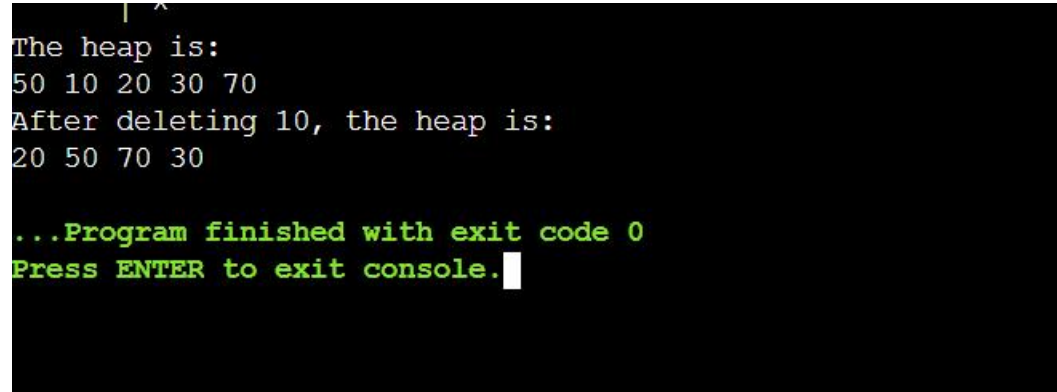
Outputs:

Output_1:

```
The heap is:
50 10 30 40 20
After deleting 10, the heap is:
20 30 40 50

...Program finished with exit code 0
Press ENTER to exit console.
```

Output_2:

```
The heap is:
50 10 20 30 70
After deleting 10, the heap is:
20 50 70 30

...Program finished with exit code 0
Press ENTER to exit console.
```