VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

ADVANCED DATA STRUCTURES (20CS5PEADS)

Submitted by

SUDESHNA BHUSHAN (1BM19CS189)
5D

Under the Guidance of
Prof. Namratha M
Assistant Professor,
Department of CSE, BMSCE

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Oct-2021 to Jan-2022

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 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.

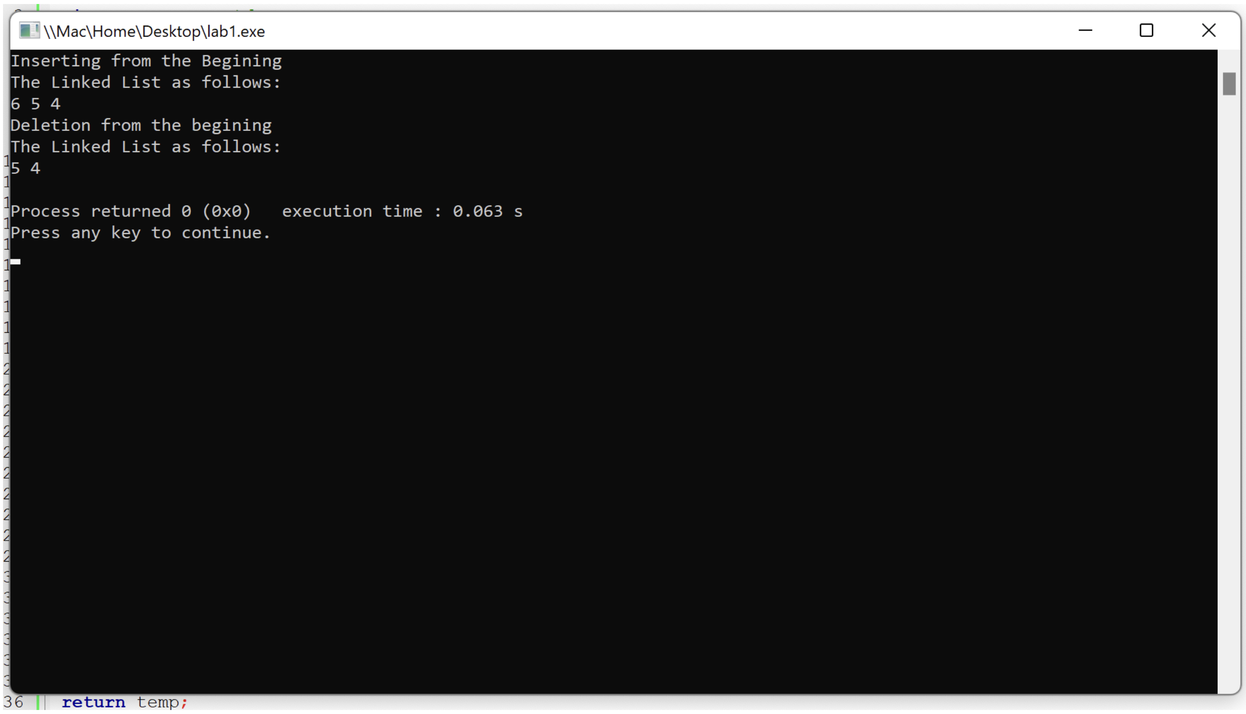
PROGRAM:

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

class node
{
 public:
 int data;
 node *npx;
};

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

void insert(node **head,int key)
{
```

```
 node *new_node=new node();
 new_node->data=key;
 new_node->npx=*head;
 if(*head!=NULL)
 {
 (*head)->npx=XOR(new_node,(*head)->npx);
 }
*head=new_node;
}

node *deleteb(node *head)
{
 if(head == NULL)
  return NULL;
 node *temp=XOR(head->npx,NULL);
 temp->npx=XOR(head,temp->npx);
 free(head);
 return temp;

}

void print(node *head)
{
 node *curr=head;
 node *prev=NULL;
 node *next;
 cout<<"The Linked List as follows: "<<endl;
 while(curr!=NULL)
```

```
 {
  cout<<curr->data<<" ";
 next=XOR(prev,curr->npx);
 prev=curr;
 curr=next;
 }
}

int main()
{
 node *head=NULL;
 cout<<"Inserting from the Begining"<<endl;
  insert(&head,4);
 insert(&head,5);
 insert(&head,6);
 print(head);
 cout<<endl;
 cout<<"Deletion from the beginning"<<endl;
 head=deleteb(head);
 print(head);
 cout<<endl;
 return 0;
}
```

OUTPUT:-



LAB PROGRAM 2:

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

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>


#define SKIPLIST_MAX_LEVEL 6
```

PROGRAM:

```
typedef struct snode {
    int key;
    int value;
    struct snode **forward;
} snode;
```

```
typedef struct skiplist {
    int level;
    int size;
    struct snode *header;
} skiplist;

skiplist *skiplist_init(skiplist *list) {
    int i;
    snode *header = (snode *) malloc(sizeof(struct snode));
    list->header = header;
    header->key = INT_MAX;
    header->forward = (snode **) malloc(
            sizeof(snode*) * (SKIPLIST_MAX_LEVEL + 1));
    for (i = 0; i <= SKIPLIST_MAX_LEVEL; i++) {
        header->forward[i] = list->header;
    }

    list->level = 1;
    list->size = 0;

    return list;
}

static int rand_level() {
    int level = 1;
    while (rand() < RAND_MAX / 2 && level < SKIPLIST_MAX_LEVEL)
        level++;
    return level;
```

```
}

int skiplist_insert(skiplist *list, int key, int value) {
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    int i, level;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }
    x = x->forward[1];

    if (key == x->key) {
        x->value = value;
        return 0;
    } else {
        level = rand_level();
        if (level > list->level) {
            for (i = list->level + 1; i <= level; i++) {
                update[i] = list->header;
            }
            list->level = level;
        }

        x = (snode *) malloc(sizeof(snode));
        x->key = key;
        x->value = value;
```

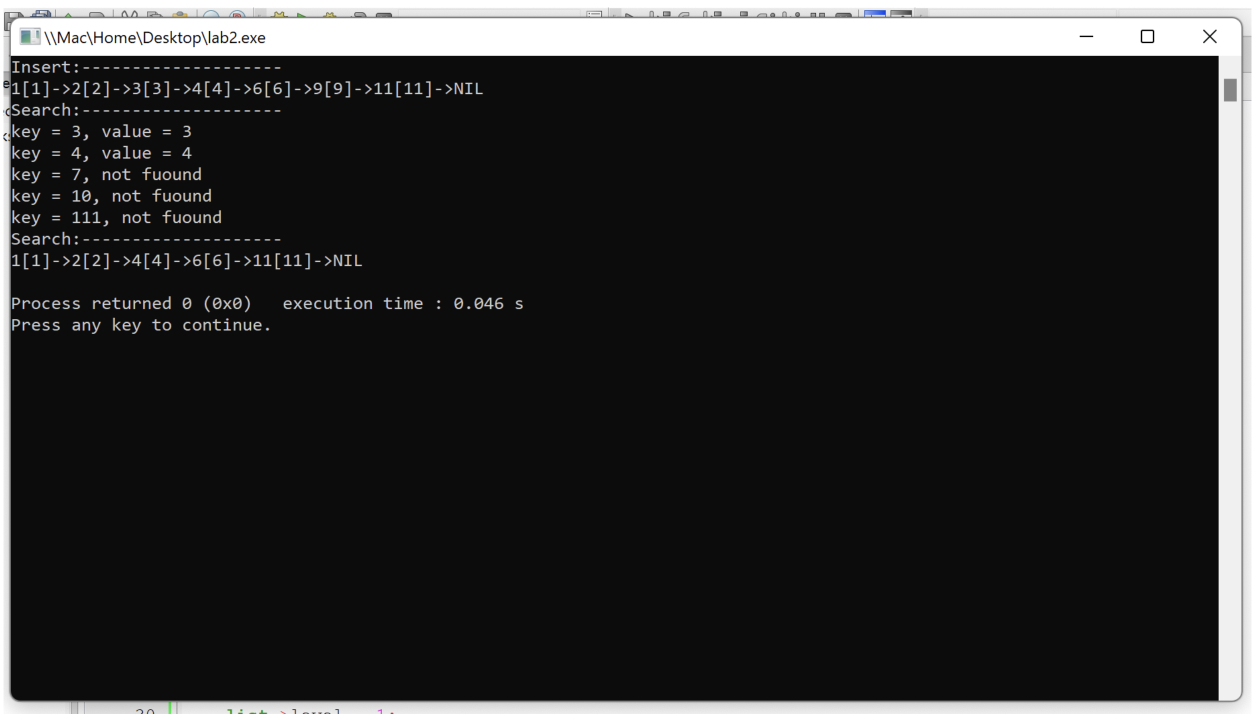
```
        x->forward = (snode **) malloc(sizeof(snode*) * (level + 1));
        for (i = 1; i <= level; i++) {
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
    }
    return 0;
}

snode *skiplist_search(skiplist *list, int key) {
    snode *x = list->header;
    int i;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
    }
    if (x->forward[1]->key == key) {
        return x->forward[1];
    } else {
        return NULL;
    }
    return NULL;
}

static void skiplist_node_free(snode *x) {
    if (x) {
        free(x->forward);
        free(x);
```

```
    }
}

int skiplist_delete(skiplist *list, int key) {
    int i;
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }

    x = x->forward[1];
    if (x->key == key) {
        for (i = 1; i <= list->level; i++) {
            if (update[i]->forward[i] != x)
                break;
            update[i]->forward[1] = x->forward[i];
        }
        skiplist_node_free(x);

        while (list->level > 1 && list->header->forward[list->level]
                == list->header)
            list->level--;
        return 0;
    }
    return 1;
```

```
}

static void skiplist_dump(skiplist *list) {
    snode *x = list->header;
    while (x && x->forward[1] != list->header) {
        printf("%d[%d]->", x->forward[1]->key, x->forward[1]->value);
        x = x->forward[1];
    }
    printf("NIL\n");
}

int main() {
    int arr[] = { 3, 6, 9, 2, 11, 1, 4 }, i;
    skiplist list;
    skiplist_init(&list);

    printf("Insert:--------------------\n");
    for (i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        skiplist_insert(&list, arr[i], arr[i]);
    }
    skiplist_dump(&list);

    printf("Search:--------------------\n");
    int keys[] = { 3, 4, 7, 10, 111 };

    for (i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {
        snode *x = skiplist_search(&list, keys[i]);
        if (x) {
```

```
            printf("key = %d, value = %d\n", keys[i], x->value);
        } else {
            printf("key = %d, not fuound\n", keys[i]);
        }
    }

    printf("Search:--------------------\n");
    skiplist_delete(&list, 3);
    skiplist_delete(&list, 9);
    skiplist_dump(&list);

    return 0;
}
```

OUTPUT:-



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 neighbours.

Use disjoint sets to implement the above scenario.

PROGRAM:

```
#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)
    {

        return find(parent[x]);
    }

    return 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;
        }
    }
};
```

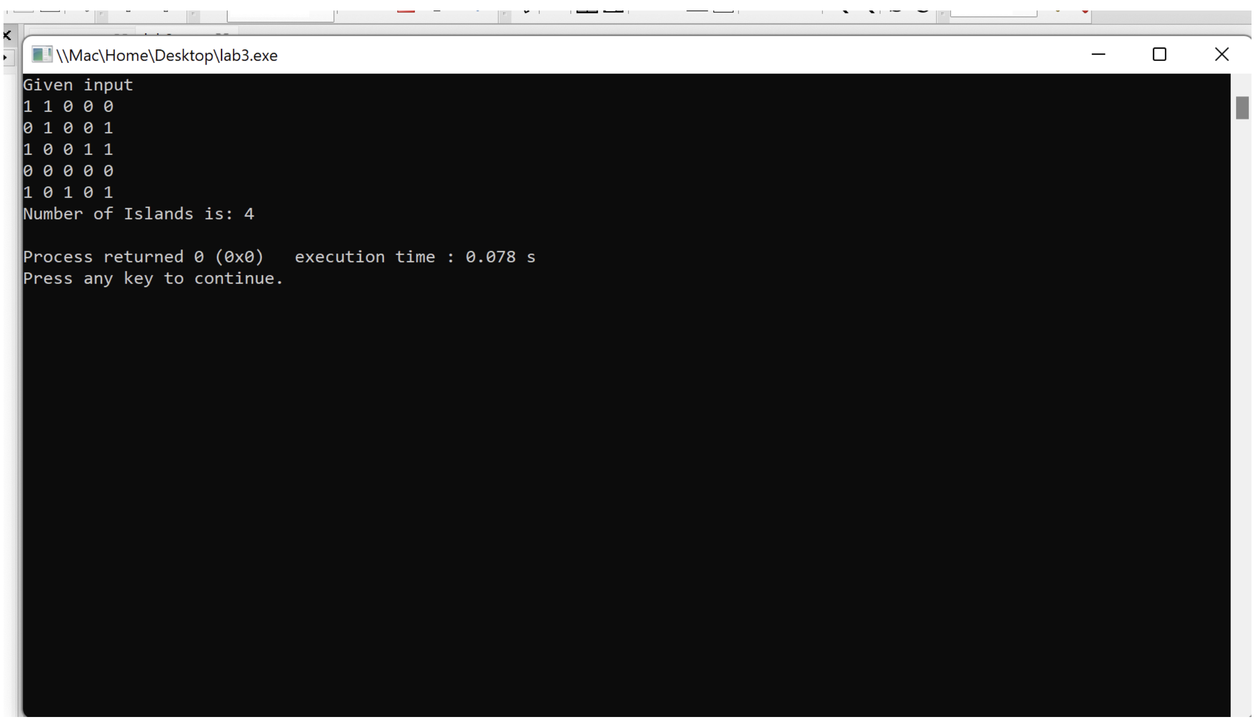
```
int countIslands(vector<vector<int>>a)
{
    int n = a.size();
    int m = a[0].size();

    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;



            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);
        }
    }

    int *c = new int[n * m];
    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]++;
                }

                else
                    c[x]++;
            }
        }
    }
    return numberOfIslands;
}

int main(void)
{
    vector<vector<int>>a = {{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<<"Given input"<<endl;
     for(int i=0;i<a.size();i++)
     {
```

```
        for(int j=0;j<a[0].size();j++)
        {
        cout<<a[i][j]<<" ";
         }
         cout<<endl;
        }

      cout << "Number of Islands is: "
          << countIslands(a) << endl;
}
```

OUTPUT:



LAB PROGRAM 4:

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

PROGRAM:

```
#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;
    node->right = NULL;
    node->height = 1;
    return(node);
}


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


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


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

```
                height(x->right)) + 1;


    return x;
}


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


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


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


    return y;
}


int getBalance(Node *N)
```

```
{
   if (N == NULL)
      return 0;
   return height(N->left) -
         height(N->right);
}

Node* insert(Node* node, int key)
{

   if (node == NULL)
      return(newNode(key));

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


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


   int balance = getBalance(node);
```

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


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


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


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


    return node;
}
```

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


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

    return current;
}


Node* deleteNode(Node* root, int key)
{


    if (root == NULL)
        return root;


    if ( key < root->key )
        root->left = deleteNode(root->left, key);


    else if( key > root->key )
        root->right = deleteNode(root->right, key);
```

```
else
{

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


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

      free(temp);
   }
   else
   {

      Node* temp = minValueNode(root->right);


      root->key = temp->key;
```

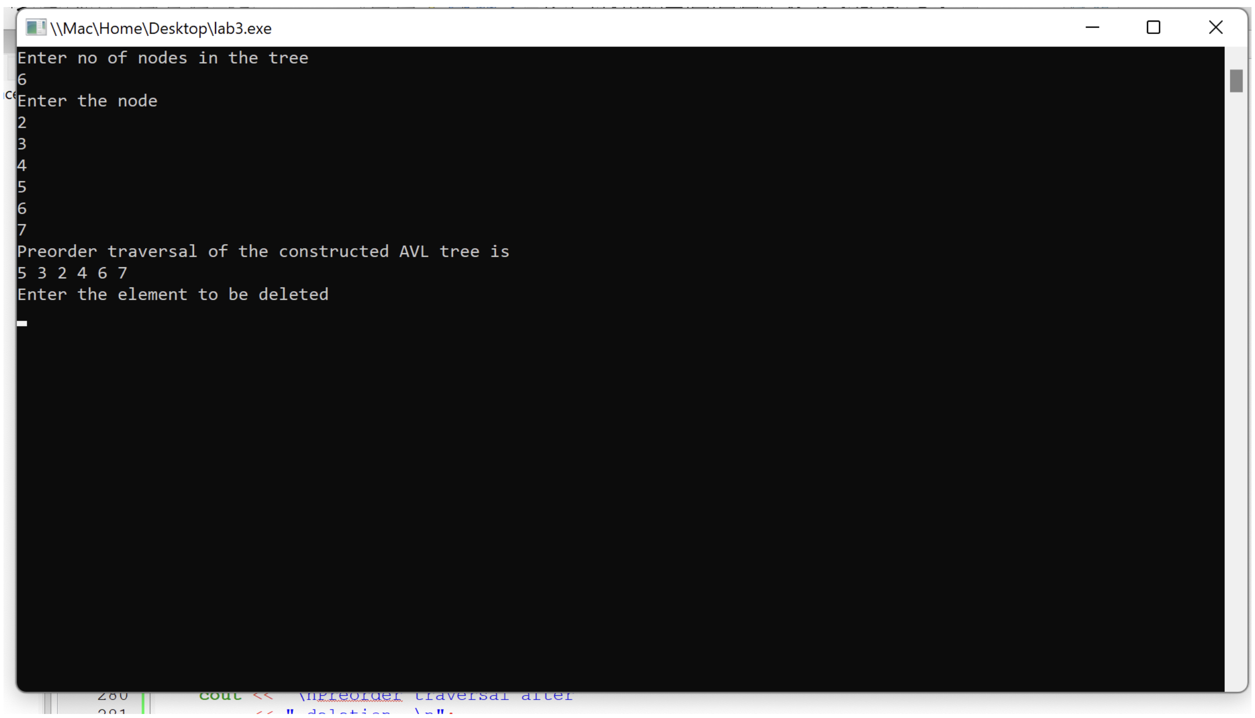
```
        root->right = deleteNode(root->right,
                                 temp->key);
    }
}

if (root == NULL)
return root;

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

int balance = getBalance(root);

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

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

```
        return rightRotate(root);
    }


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


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

    return root;
}


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

```
}


int main()
{
Node *root = NULL;

   int n,in,del;
   cout<<"Enter no of nodes in the tree"<<endl;
   cin>>n;
   cout<<"Enter the node"<<endl;
   for(int i=0;i<n;i++)
   {

    cin>>in;
    root = insert(root, in);
  }



    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);
   cout<<endl;
    cout<<"Enter the element to be deleted"<<endl;
    cin>>del;
    root = deleteNode(root, del);
```

```
    cout << "\nPreorder traversal after"
         << " deletion  \n";
    preOrder(root);

    return 0;
}
```

OUTPUT:



LAB PROGRAM 5:

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

PROGRAM:

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

```
class TreeNode
{
	int *keys;
	TreeNode **child;
	int n;
	bool leaf;
	public:
		TreeNode(bool leaf);
		void traverse();
		int findKey(int k);
		void insertNonFull(int k);
		void splitChild(int i, TreeNode *y);
		void remove(int k);
		void removeFromLeaf(int idx);
		void removeFromNonLeaf(int idx);
		int getPred(int idx);
		int getSucc(int idx);
		void fill(int idx);
		void borrowFromNext(int idx);
		void borrowFromPrev(int idx);
		void merge(int idx);
		friend class Tree;
};

class Tree
{
	TreeNode *root = NULL;
	public:
```

```
        /*Tree(){
            root = NULL;
        }*/
        void traverse()
        {
            if(root != NULL)
                root->traverse();
        }
        void insert(int k);
        void remove(int k);
};

TreeNode::TreeNode(bool leaf1)
{
    leaf = leaf1;
    keys = new int[3];
    child = new TreeNode *[4];
    n = 0;
}

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

```
void Tree::insert(int k)
{
	if(root == NULL)
	{
		root = new TreeNode(true);
		root->keys[0] = k;
		root->n = 1;
	}
	else{
		if(root->n == 3)
		{
			TreeNode *s = new TreeNode(false);
			s->child[0] = root;
			s->splitChild(0, root);
			int i = 0;
			if(s->keys[0]<k)
				i++;
			s->child[i]->insertNonFull(k);
			root = s;
		}
		else
			root->insertNonFull(k);
	}
}

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

```
    if(leaf == true)
    {
        while(i>=0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }
        keys[i+1] = k;
        n = n + 1;
    }
    else{
        while(i>=0 && keys[i]>k)
            i--;
        if(child[i+1]->n == 3)
        {
            splitChild(i+1, child[i+1]);
            if(keys[i+1]<k)
                i++;
        }
        child[i+1]->insertNonFull(k);
    }
}

void TreeNode::splitChild(int i, TreeNode *y)
{
    TreeNode *z = new TreeNode(y->leaf);
    z->n = 1;
    z->keys[0] = y->keys[2];
```

```
    if(y->leaf == false)
    {
        for(int j=0; j<2; j++)
            z->child[j] = y->child[j+2];
    }
    y->n = 1;
    for(int j=n; j>=i+1; j--)
        child[j+1] = child[j];
    child[i+1] = z;
    for (int j = n-1; j >= i; j--)
    keys[j+1] = keys[j];

    keys[i] = y->keys[1];

    n = n + 1;
}

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

```
                child[i]->traverse();

        cout<<endl;
}

void TreeNode::remove(int k)
{
        int idx = findKey(k);
        if(idx<n && keys[idx] == k)
        {
                if(leaf)
                        removeFromLeaf(idx);
                else
                        removeFromNonLeaf(idx);
        }
        else
        {
                if(leaf)
                {
                        cout<<"The key doesn't exist"<<endl;
                        return;
                }
                bool flag = ((idx==n)?true : false);
                if(child[idx]->n < 2)
                        fill(idx);
                if(flag && idx>n)
                        child[idx-1]->remove(k);
                else
```

```
                    child[idx]->remove(k);
        }
        return;
}

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

void TreeNode::removeFromNonLeaf(int idx)
{
        int k = keys[idx];
        if(child[idx]->n >=2)
        {
                int pred = getPred(idx);
                keys[idx] = pred;
                child[idx]->remove(pred);
        }
        else if(child[idx+1]->n >= 2)
        {
                int succ = getSucc(idx);
                keys[idx] = succ;
                child[idx+1]->remove(succ);
        }
```

```
        else{
                merge(idx);
                child[idx]->remove(k);
        }
        return;
}

int TreeNode::getPred(int idx)
{
        TreeNode *curr = child[idx];
        while(!curr->leaf)
                curr = curr->child[curr->n];
        return curr->keys[curr->n-1];
}

int TreeNode::getSucc(int idx)
{
        TreeNode *curr = child[idx+1];
        while (!curr->leaf)
                curr = curr->child[0];
        return curr->keys[0];
}

void TreeNode::fill(int idx)
{
        if(idx!=0 && child[idx-1]->n>=2)
        borrowFromPrev(idx);
```

```
	else if (idx!=n && child[idx+1]->n>=2)
	borrowFromNext(idx);
	else
	{
	if (idx != n)
			merge(idx);
	else
			merge(idx-1);
	}
	return;
}

void TreeNode::borrowFromPrev(int idx)
{


	TreeNode *c=child[idx];
	TreeNode *sibling=child[idx-1];

	for (int i=c->n-1; i>=0; --i)
	c->keys[i+1] = c->keys[i];

	if (!c->leaf)
	{
	for(int i=c->n; i>=0; --i)
	    c->child[i+1] = c->child[i];
	}
	c->keys[0] = keys[idx-1];
```

```
        if(!c->leaf)
        c->child[0] = sibling->child[sibling->n];

        keys[idx-1] = sibling->keys[sibling->n-1];

        c->n += 1;
        sibling->n -= 1;

        return;
}

void TreeNode::borrowFromNext(int idx)
{

        TreeNode *c=child[idx];
        TreeNode *sibling=child[idx+1];

        c->keys[(c->n)] = keys[idx];

        if (!(c->leaf))
        c->child[(c->n)+1] = sibling->child[0];

        keys[idx] = sibling->keys[0];

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

        if (!sibling->leaf)
```

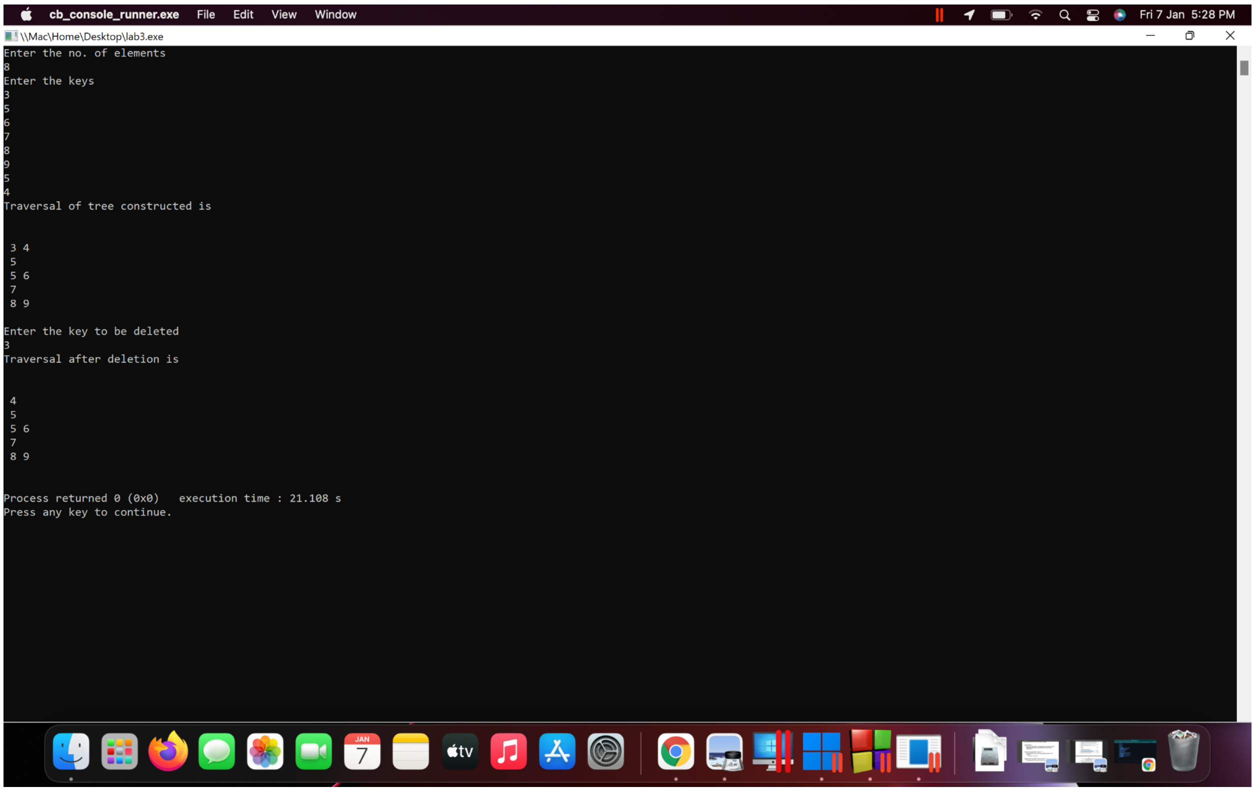
```
	{
	for(int i=1; i<=sibling->n; ++i)
			sibling->child[i-1] = sibling->child[i];
	}

	c->n += 1;
	sibling->n -= 1;

	return;
}

void TreeNode::merge(int idx)
{
	TreeNode *c = child[idx];
	TreeNode *sibling = child[idx+1];

	c->keys[1] = keys[idx];

	for (int i=0; i<sibling->n; ++i)
	c->keys[i+2] = sibling->keys[i];

	if (!c->leaf)
	{
	for(int i=0; i<=sibling->n; ++i)
			c->child[i+2] = sibling->child[i];
	}

	for (int i=idx+1; i<n; ++i)
```

```
        keys[i-1] = keys[i];

        for (int i=idx+2; i<=n; ++i)
        child[i-1] = child[i];

        c->n += sibling->n+1;
        n--;

        delete(sibling);
        return;
}

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

        root->remove(k);

        if (root->n==0)
        {
        TreeNode *tmp = root;
        if (root->leaf)
                        root = NULL;
        else
```

```
                    root = root->child[0];

        delete tmp;
        }
        return;
}



int main()
{
        Tree t;
        int n,k;
        cout<<"Enter the no. of elements"<<endl;
        cin>>n;
        cout<<"Enter the keys"<<endl;
        for(int i=0; i<n; i++)
        {
                cin>>k;
                t.insert(k);
        }
        cout << "Traversal of tree constructed is\n";
        t.traverse();
        cout<<"Enter the key to be deleted"<<endl;
        cin>>k;
        t.remove(k);
        cout<<"Traversal after deletion is"<<endl;
        t.traverse();
```

```
	return 0;

}
```

OUTPUT:



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.

PROGRAM:

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


enum Color {RED, BLACK};
```

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


   Node(int data)
   {
     this->data = data;
     left = right = parent = NULL;
     this->color = RED;
   }
};


class RBTree
{
private:
   Node *root;
protected:
   void rotateLeft(Node *&, Node *&);
   void rotateRight(Node *&, Node *&);
   void fixViolation(Node *&, Node *&);
public:

   RBTree() { root = NULL; }
```

```
    void insert(const int &n);
    void inorder();
    void levelOrder();
};


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

    inorderHelper(root->left);
    cout << root->data << "  ";
    inorderHelper(root->right);
}


Node* BSTInsert(Node* root, Node *pt)
{

    if (root == NULL)
       return pt;



    if (pt->data < root->data)
    {
        root->left  = BSTInsert(root->left, pt);
        root->left->parent = root;
```

```
    }
    else if (pt->data > root->data)
    {
        root->right = BSTInsert(root->right, pt);
        root->right->parent = root;
    }


    return root;
}


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

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

    while (!q.empty())
    {
        Node *temp = q.front();
        cout << temp->data << "  ";
        q.pop();

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

```
        if (temp->right != NULL)
            q.push(temp->right);
    }
}

void RBTree::rotateLeft(Node *&root, Node *&pt)
{
    Node *pt_right = pt->right;

    pt->right = pt_right->left;

    if (pt->right != NULL)
        pt->right->parent = pt;

    pt_right->parent = pt->parent;

    if (pt->parent == NULL)
        root = pt_right;

    else if (pt == pt->parent->left)
        pt->parent->left = pt_right;

    else
        pt->parent->right = pt_right;

    pt_right->left = pt;
    pt->parent = pt_right;
```

```
}

void RBTree::rotateRight(Node *&root, Node *&pt)
{
    Node *pt_left = pt->left;

    pt->left = pt_left->right;

    if (pt->left != NULL)
        pt->left->parent = pt;

    pt_left->parent = pt->parent;

    if (pt->parent == NULL)
        root = pt_left;

    else if (pt == pt->parent->left)
        pt->parent->left = pt_left;

    else
        pt->parent->right = pt_left;

    pt_left->right = pt;
    pt->parent = pt_left;
}


void RBTree::fixViolation(Node *&root, Node *&pt)
```

```
{
   Node *parent_pt = NULL;
   Node *grand_parent_pt = NULL;

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

      parent_pt = pt->parent;
      grand_parent_pt = pt->parent->parent;



      if (parent_pt == grand_parent_pt->left)
      {

         Node *uncle_pt = grand_parent_pt->right;



         if (uncle_pt != NULL && uncle_pt->color ==
                                             RED)
         {
            grand_parent_pt->color = RED;
            parent_pt->color = BLACK;
            uncle_pt->color = BLACK;
            pt = grand_parent_pt;
         }

         else
```

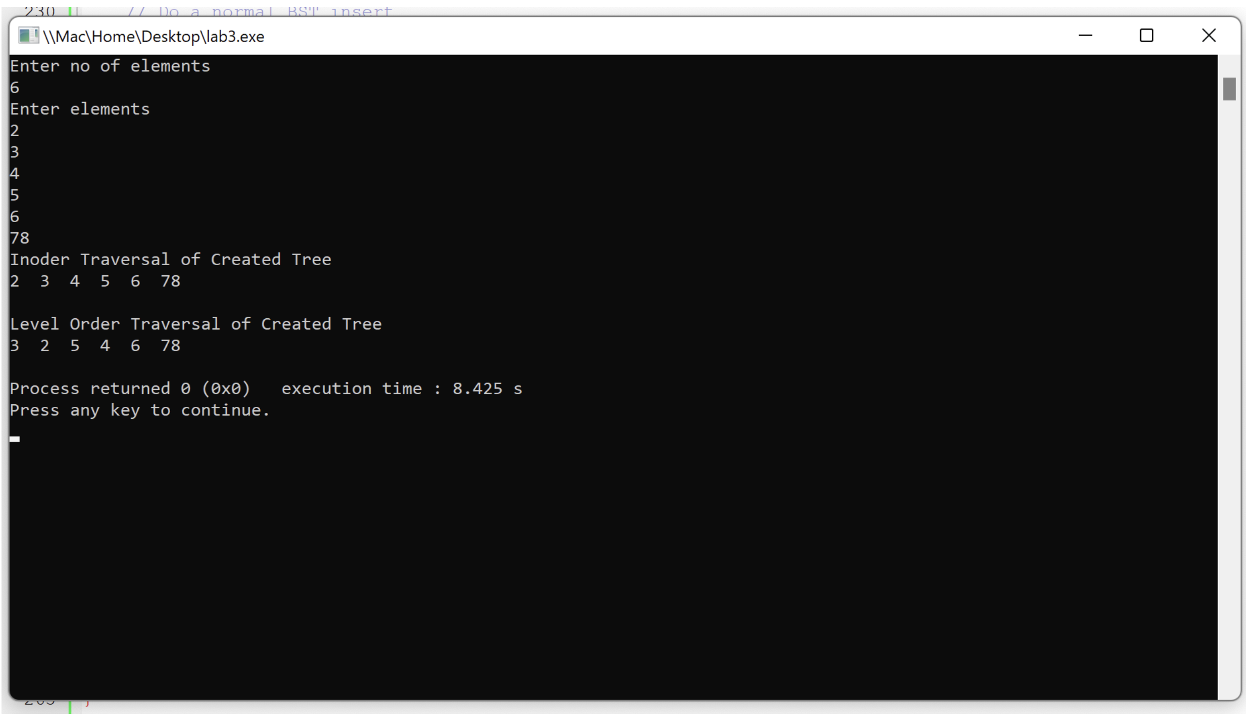
```
    {

        if (pt == parent_pt->right)
        {
            rotateLeft(root, parent_pt);
            pt = parent_pt;
            parent_pt = pt->parent;
        }


        rotateRight(root, grand_parent_pt);
        swap(parent_pt->color,
                   grand_parent_pt->color);
        pt = parent_pt;
    }
}


else
{
    Node *uncle_pt = grand_parent_pt->left;

    if ((uncle_pt != NULL) && (uncle_pt->color ==
                                          RED))
    {
        grand_parent_pt->color = RED;
        parent_pt->color = BLACK;
        uncle_pt->color = BLACK;
```

```
            pt = grand_parent_pt;
        }
        else
        {

            if (pt == parent_pt->left)
            {
                rotateRight(root, parent_pt);
                pt = parent_pt;
                parent_pt = pt->parent;
            }


            rotateLeft(root, grand_parent_pt);
            swap(parent_pt->color,
                     grand_parent_pt->color);
            pt = parent_pt;
        }
    }
  }

  root->color = BLACK;
}


void RBTree::insert(const int &data)
{
  Node *pt = new Node(data);
```

```
    // Do a normal BST insert
    root = BSTInsert(root, pt);

    // fix Red Black Tree violations
    fixViolation(root, pt);
}


void RBTree::inorder()     {  inorderHelper(root);}
void RBTree::levelOrder()  {  levelOrderHelper(root); }


int main()
{
    RBTree tree;
  int n,num;
  cout<<"Enter no of elements"<<endl;
  cin>>n;
  cout<<"Enter elements"<<endl;
  for(int i=0;i<n;i++)
  {
    cin>>num;
    tree.insert(num);
   }

    cout << "Inoder Traversal of Created Tree\n";
    tree.inorder();
```

```
    cout << "\n\nLevel Order Traversal of Created Tree\n";
    tree.levelOrder();
    cout<<endl;

    return 0;
}
```

OUTPUT:



LAB PROGRAM 7:

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

PROGRAM:

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

```
class BTreeNode
{
    int *keys;
    int t;
    BTreeNode **C;
    int n;
    bool leaf;
public:
    BTreeNode(int _t, bool _leaf);


    void insertNonFull(int k);



    void splitChild(int i, BTreeNode *y);


    void traverse();



friend class BTree;
};
```

```
class BTree
{
   BTreeNode *root;
   int t;
public:

   BTree(int _t)
   {  root = NULL;  t = _t; }



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






   void insert(int k);
};


BTreeNode::BTreeNode(int t1, bool leaf1)
{

   t = t1;
   leaf = leaf1;
```

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



    n = 0;
}

void BTreeNode::traverse()
{

    int i;
    for (i = 0; i < n; i++)
    {

        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }



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

```
void BTree::insert(int k)
{

    if (root == NULL)
    {

        root = new BTreeNode(t, true);
        root->keys[0] = k;
        root->n = 1;
    }
    else
    {

        if (root->n == 2*t-1)
        {

            BTreeNode *s = new BTreeNode(t, false);



            s->C[0] = root;



            s->splitChild(0, root);



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

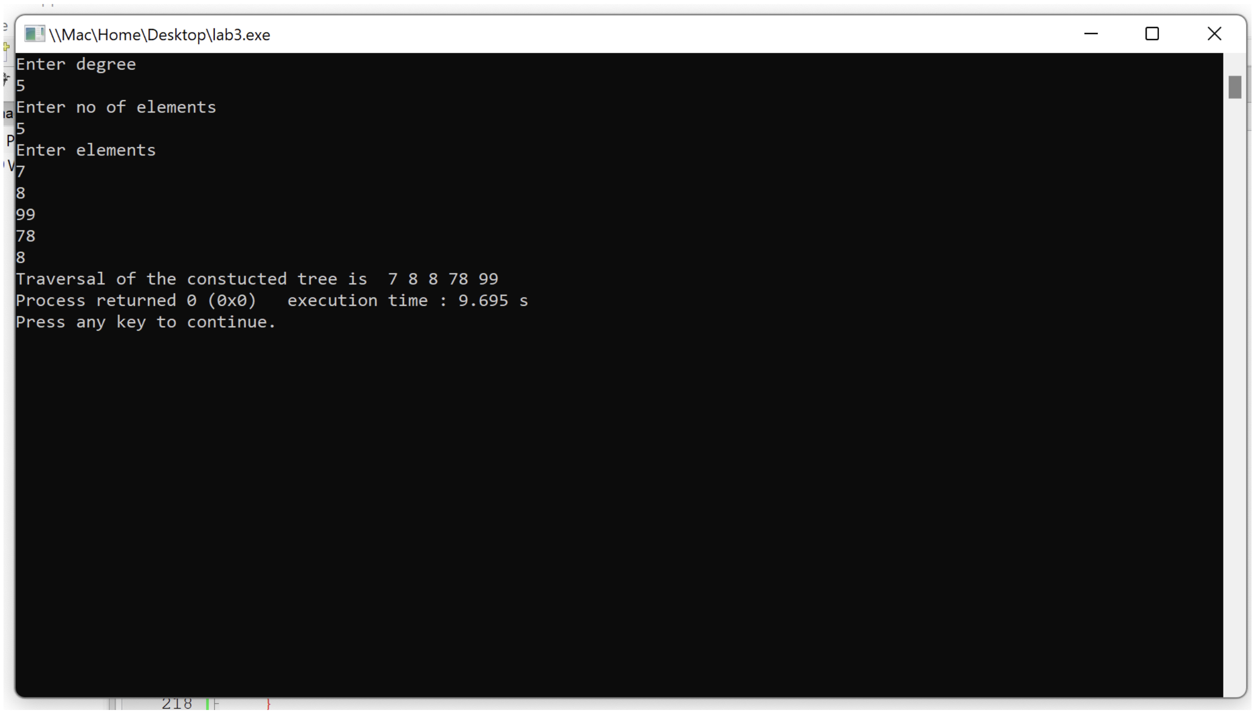
```
            s->C[i]->insertNonFull(k);


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


void BTreeNode::insertNonFull(int k)
{

    int i = n-1;


    if (leaf == true)
    {

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


        keys[i+1] = k;
```

```
        n = n+1;
    }
    else
    {

        while (i >= 0 && keys[i] > k)
            i--;


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

            splitChild(i+1, C[i+1]);


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


void BTreeNode::splitChild(int i, BTreeNode *y)
{

    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;
```

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


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


y->n = t - 1;


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


C[i+1] = z;


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


keys[i] = y->keys[t-1];
```

```
    n = n + 1;
}


int main()
{
    int degree,n,num;
    cout<<"Enter degree"<<endl;
    cin>>degree;
    BTree t(degree);
    cout<<"Enter no of elements"<<endl;
    cin>>n;
    cout<<"Enter elements"<<endl;
    for(int i=0;i<n;i++)
    {
      cin>>num;
    t.insert(num);
    }


    cout << "Traversal of the constucted tree is ";
    t.traverse();


    return 0;
}
```

OUTPUT:



```
\\Mac\Home\Desktop\lab3.exe
Enter degree
5
Enter no of elements
5
Enter elements
7
8
99
78
8
Traversal of the constucted tree is  7 8 8 78 99
Process returned 0 (0x0)   execution time : 9.695 s
Press any key to continue.
```

LAB PROGRAM 8:

Write a program to implement functions of Dictionaries using Hashing.

PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;
const int Table_size = 200;
class HashTableEntry {
  public:
    int k;
    int v;
    HashTableEntry(int k, int v) {
      this->k= k;
```

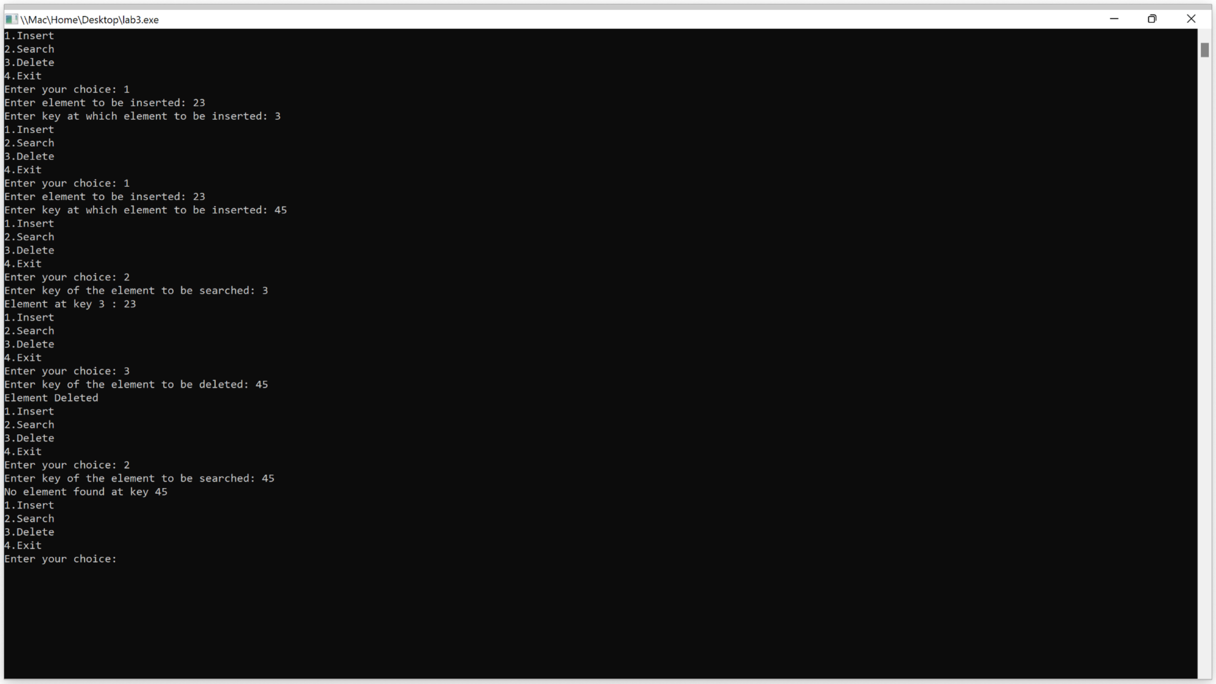
```
      this->v = v;
    }
};
class HashMapTable {
  private:
    HashTableEntry **t;
  public:
    HashMapTable() {
      t = new HashTableEntry * [Table_size];
      for (int i = 0; i< Table_size; i++) {
        t[i] = NULL;
      }
    }
    int hashFunc(int k) {
      return k % Table_size;
    }
    void insert(int k, int v) {
      int h = hashFunc(k);
      while (t[h] != NULL && t[h]->k != k) {
        h = hashFunc(h + 1);
      }
      if (t[h] != NULL)
        delete t[h];
      t[h] = new HashTableEntry(k, v);
    }
    int search(int k) {
      int h = hashFunc(k);
      while (t[h] != NULL && t[h]->k != k) {
```

```
    h = hashFunc(h + 1);
  }
  if (t[h] == NULL)
    return -1;
  else
    return t[h]->v;
}
void deleteEle(int k) {
  int h = hashFunc(k);
  while (t[h] != NULL) {
    if (t[h]->k == k)
      break;
    h = hashFunc(h + 1);
  }
  if (t[h] == NULL) {
    cout<<"No Element found at key "<<k<<endl;
    return;
  } else {
    delete t[h];
  }
  cout<<"Element Deleted"<<endl;
}
~HashMapTable() {
  for (int i = 0; i < Table_size; i++) {
    if (t[i] != NULL)
      delete t[i];
      delete[] t;
  }
```

```
    }
};
int main() {
  HashMapTable hash;
  int k, v;
  int c;
  while (1) {
    cout<<"1.Insert"<<endl;
    cout<<"2.Search"<<endl;
    cout<<"3.Delete"<<endl;
    cout<<"4.Exit"<<endl;
    cout<<"Enter your choice: ";
    cin>>c;
    switch(c) {
      case 1:
        cout<<"Enter element to be inserted: ";
        cin>>v;
        cout<<"Enter key at which element to be inserted: ";
        cin>>k;
        hash.insert(k, v);
      break;
      case 2:
        cout<<"Enter key of the element to be searched: ";
        cin>>k;
        if (hash.search(k) == -1) {
          cout<<"No element found at key "<<k<<endl;
          continue;
        } else {
```

```
            cout<<"Element at key "<<k<<" : ";
            cout<<hash.search(k)<<endl;
          }
        break;
        case 3:
          cout<<"Enter key of the element to be deleted: ";
          cin>>k;
          hash.deleteEle(k);
        break;
        case 4:
          exit(1);
        default:
          cout<<"\nEnter correct option\n";
      }
    }
  return 0;
}
```

OUTPUT:



```
\\Mac\Home\Desktop\lab3.exe
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 1
Enter element to be inserted: 23
Enter key at which element to be inserted: 3
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 1
Enter element to be inserted: 23
Enter key at which element to be inserted: 45
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 3
Element at key 3 : 23
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 3
Enter key of the element to be deleted: 45
Element Deleted
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 45
No element found at key 45
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice:
```

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 roots 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

PROGRAM:

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


// A Binomial Tree node.
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;
}

// This function merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
   // Make sure b1 is smaller
   if (b1->data > b2->data)
      swap(b1, b2);

   // We basically make larger valued tree
   // a child of smaller valued tree
   b2->parent = b1;
   b2->sibling = b1->child;
   b1->child = b2;
   b1->degree++;
```

```
    return b1;
}

// This function perform union operation on two
// binomial heap i.e. l1 & l2
list<Node*> unionBionomialHeap(list<Node*> l1,
                               list<Node*> l2)
{
    // _new to another binomial heap which contain
    // new heap after merging l1 & l2
    list<Node*> _new;
    list<Node*>::iterator it = l1.begin();
    list<Node*>::iterator ot = l2.begin();
    while (it!=l1.end() && ot!=l2.end())
    {
        // if D(l1) <= D(l2)
        if((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }
        // if D(l1) > D(l2)
        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }
```

```
    // if there remains some elements in l1
    // binomial heap
    while (it != l1.end())
    {
        _new.push_back(*it);
        it++;
    }

    // if there remains some elements in l2
    // binomial heap
    while (ot!=l2.end())
    {
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}

// adjust function rearranges the heap so that
// heap is in increasing order of degree and
// no two binomial trees have same degree in this heap
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 only one element remains to be processed
    if (it2 == _heap.end())
        it1++;

    // If D(it1) < D(it2) i.e. merging of Binomial
    // Tree pointed by it1 & it2 is not possible
    // then move next in heap
    else if ((*it1)->degree < (*it2)->degree)
    {
        it1++;
        it2++;
        if(it3!=_heap.end())
```

```
            it3++;
        }

        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are same
        // in heap
        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
            it1++;
            it2++;
            it3++;
        }

        // if degree of two Binomial Tree are same in heap
        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)
{
    // 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);
}

// removing minimum key element from binomial heap
// this function take Binomial Tree as input and return
// binomial heap after
// removing head of that tree i.e. minimum element
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;
}

// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
    Node *temp = newNode(key);
    return insertATreeInHeap(_head,temp);
}

// return pointer of minimum value Node
// present in the binomial heap
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++;
```

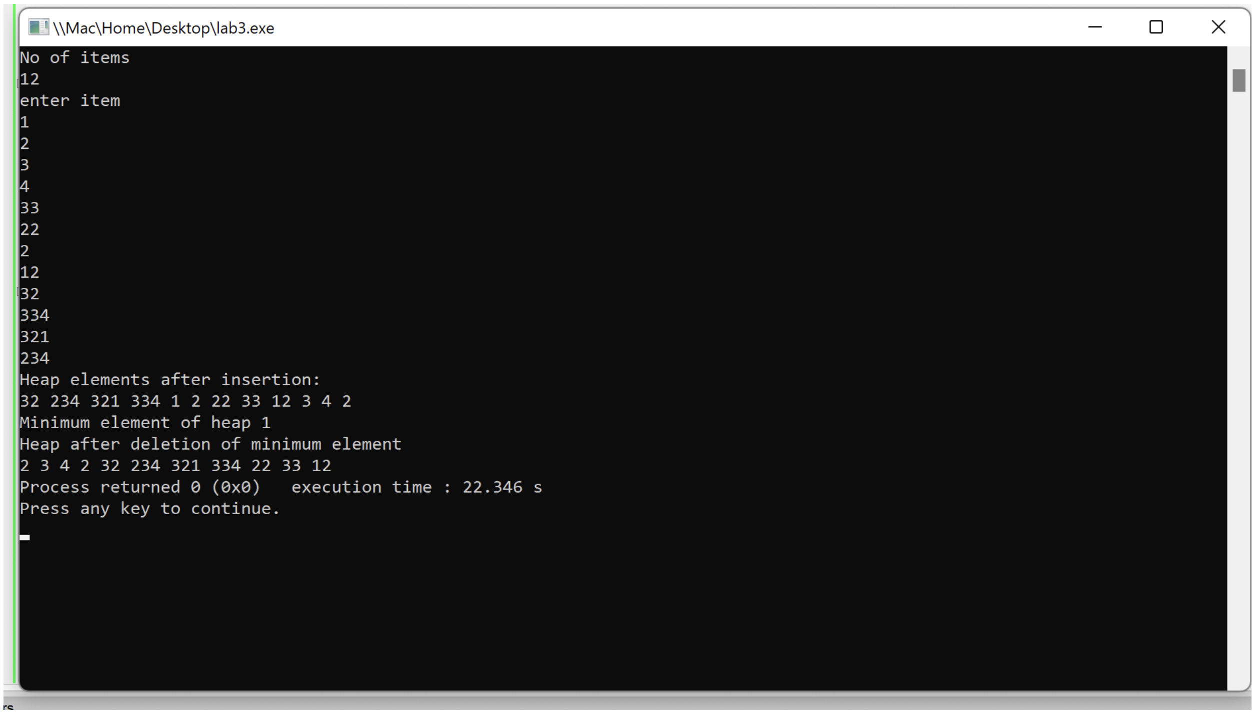
```
    }
    return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
    list<Node*> new_heap,lo;
    Node *temp;

    // temp contains the pointer of minimum value
    // element in heap
    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        if (*it != temp)
        {
            // inserting all Binomial Tree into new
            // binomial heap except the Binomial Tree
            // contains minimum element
            new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBionomialHeap(new_heap,lo);
    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;

   // Insert data in the heap
   int i,n,in;
   cout<<"No of items"<<endl;
   cin>>n;
   cout<<"enter item"<<endl;
   for(i=0;i<n;i++)
   {
    cin>>in;
   _heap = insert(_heap,in);
   }

   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;
}
```

OUTPUT:



LAB PROGRAM 10:

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

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

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

PROGRAM:

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


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


Node *root = NULL;

int binomialLink(Node *h1, Node *h2)
{
    h1->parent = h2;
    h1->sibling = h2->child;
    h2->child = h1;
    h2->degree = h2->degree + 1;
}

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;
}

Node *mergeBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;



    Node *res = NULL;


    if (h1->degree <= h2->degree)
        res = h1;
```

```
else if (h1->degree > h2->degree)
    res = h2;



while (h1 != NULL && h2 != NULL)
{
    if (h1->degree < h2->degree)
        h1 = h1->sibling;



    else if (h1->degree == h2->degree)
    {
        Node *sib = h1->sibling;
        h1->sibling = h2;
        h1 = sib;
    }



    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);



    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;
}

void binomialHeapInsert(int x)
{
    root = unionBHeaps(root, createNode(x));
}

void display(Node *h)
{
    while (h)
    {
        cout << h->val << " ";
        display(h->child);
        h = h->sibling;
    }
}
```

```
int revertList(Node *h)
{
    if (h->sibling != NULL)
    {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
    }
    else
        root = h;
}

Node *extractMinBHeap(Node *h)
{
    if (h == NULL)
       return NULL;

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

    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 (min_node_prev == NULL &&
    min_node->sibling == NULL)
    h = NULL;

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



else
    min_node_prev->sibling = min_node->sibling;
```

```
    if (min_node->child != NULL)
    {
        revertList(min_node->child);
        (min_node->child)->sibling = NULL;
    }

    return unionBHeaps(h, root);
}


Node *findNode(Node *h, int val)
{
    if (h == NULL)
      return NULL;



    if (h->val == val)
        return h;



    Node *res = findNode(h->child, val);
    if (res != NULL)
```

```
        return res;

    return findNode(h->sibling, val);
}



void decreaseKeyBHeap(Node *H, int old_val,
                      int new_val)
{
    Node *node = findNode(H, old_val);

    if (node == NULL)
        return;

    node->val = new_val;
    Node *parent = node->parent;

    while (parent != NULL && node->val < parent->val)
    {
        swap(node->val, parent->val);
        node = parent;
        parent = parent->parent;
```

```
    }
}


Node *binomialHeapDelete(Node *h, int val)
{

    if (h == NULL)
        return NULL;



    decreaseKeyBHeap(h, val, INT_MIN);



    return extractMinBHeap(h);
}

int main()
{
   int n,temp,del;
   cout<<"Enter no of elements in the heap"<<endl;
   cin>>n;
```

```
cout<<"Enter elements"<<endl;
for(int i=0;i<n;i++)
{
 cin>>temp;
 binomialHeapInsert(temp);
 }

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

 cout<<"Enter element to be deleted"<<endl;
 cin>>del;

 root = binomialHeapDelete(root, del);

 cout << "After deletion, the heap is:\n";

 display(root);
 cout<<endl;

 return 0;
```

}

OUTPUT:

\\Mac\Home\Desktop\lab3.exe

```
Enter no of elements in the heap
13
Enter elements
12
21
2
122
32
21
222
111
45
32
67
7
8
The heap is:
8 7 32 45 67 2 21 111 222 32 12 21 122
Enter element to be deleted
45
After deletion, the heap is:
8 67 2 21 111 222 32 12 21 122

Process returned 0 (0x0)   execution time : 28.297 s
Press any key to continue.
```