**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**ADAVANCED DATA STRUCTURES**

***Submitted by***

**DINESH KUMAR G(1BM20CS043)**

**V SEM , A Sec**

**Under the Guidance of**

**Dr. Prasad G R**

**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 2022-Feb 2023**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “**ADVANCED** **DATA STRUCTURES**” carried out by **DINESH KUMAR G (1BM20CS043),** who is a bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Advanced Data structures Lab **- (20CS5PEADS)** work prescribed for the said degree.

**Dr. Prasad G R**           **Dr. Jyothi S Nayak**

Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

`

**Index**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiment Title** | **Page No.** |
| 1 | IMPLEMENTATION OF XOR LINKED LIST | 4 |
| 2 | IMPLEMENTATION OF SKIP LIST | 9 |
| 3 | IMPLEMENTATION OF DISJOINT SETS TO FIND NUMBER OF ISLANDS | 17 |
| 4 | IMPLEMENTATION OF OPERATIONS ON AVL TREES | 21 |
| 5 | IMPLEMENTATION OF OPERATIONS ON 2-3 TREES | 29 |
| 6 | IMPLEMENTATION OF INSERTION OPERATION IN RED-BLACK TREES | 43 |
| 7 | IMPLEMENTATION OF INSERTION OPERATION ON B TREES | 51 |
| 8 | IMPLEMENTATION OF DICTIONARY FUNCTIONS USING HASHING | 58 |
| 9 | IMPLEMENTATION OF FOLLOWING OPERATIONS ON A BINOMIAL HEAP   1. Insertion 2. Determining minimum key 3. Extracting minimum key | 63 |
| 10 | IMPLEMENTATION OF FOLLOWING OPERATIONS ON A BINOMIAL HEAP   1. Delete a key 2. Decrease a key | 71 |

**LAB PROGRAM 1:**

**Implementation of XOR linked list**

#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 insertBeg(Node \*\*head, int data)

{

Node \*new\_node = new Node();

new\_node->data = data;

new\_node->npx = \*head;

if (\*head != NULL)

(\*head)->npx = XOR(new\_node, (\*head)->npx);

\*head = new\_node;

}

void insertEnd(Node \*\*head, int data)

{ Node \*new\_node = new Node();

new\_node->data = data;

if (\*head == NULL)

{

new\_node->npx = \*head;

\*head = new\_node;

}

else

{

Node \*curr = \*head;

Node \*prev = NULL;

Node \*next;

while (XOR(prev, curr->npx) != NULL)

{

next = XOR(prev, curr->npx);

prev = curr;

curr = next;

}

new\_node->npx = curr;

curr->npx = XOR(prev, new\_node);

}

}

void printList(Node \*head)

{

Node \*curr = head;

Node \*prev = NULL;

Node \*next;

cout << "The elements are \n";

while (curr != NULL)

{

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

next = XOR(prev, curr->npx);

prev = curr;

curr = next;

}

}

Node \*deleteEnd(Node \*head)

{

if (!head)

return NULL;

Node \*prev = NULL, \*curr = head, \*next = XOR(prev, curr->npx);

while (next)

{

prev = curr;

curr = next;

next = XOR(prev, curr->npx);

}

prev->npx = XOR(prev->npx, curr);

free(curr);

return head;

}

Node \*deleteBeg(Node \*head)

{

if (!head)

return NULL;

Node \*next = XOR(head->npx, NULL);

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

free(head);

return next;

}

int main()

{

Node \*head = NULL;

int beg, end;

cout << "enter the number of elements to insert at begining \n";

cin >> beg;

while (beg--)

{

int val;

cout<<"enter element\n";

cin >> val;

insertBeg(&head, val);

}

cout << "enter the number of elements to insert at end \n";

cin >> end;

while (end--)

{

int val;

cout<<"enter element\n";

cin >> val;

insertEnd(&head, val);

}

printList(head);

cout << "\n enter the number of elements to delete at begining \n";

cin >> beg;

while (beg--)

{ head = deleteBeg(head); }

cout << "enter the number of elements to delete at end \n";

cin >> end;

while (end--)

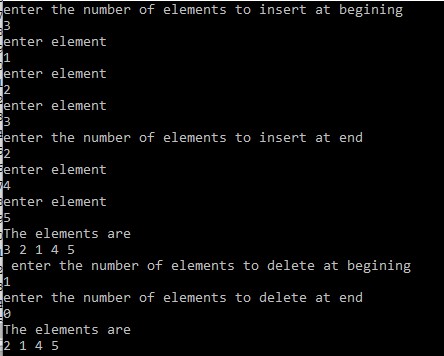
{ head = deleteEnd(head); }

printList(head);

return 0;

}

**OUTPUT**



**LAB PROGRAM 2:**

**Implementation of Skip list**

#include <bits/stdc++.h>

using namespace std;

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 SkipList

{

int MAXLVL;

float P;

int level;

Node \*header;

public:

SkipList(int, float);

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;

};

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;

}

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 and 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";

}

};

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 and 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()

{

SkipList lst(3, 0.5);

srand((unsigned)time(0));

int choice, n;

cout << "1.Insert Element" << endl;

cout << "2.Delete Element" << endl;

cout << "3.Search Element" << endl;

cout << "4.Display List " << endl;

cout << "5.Exit " << endl;

while (1)

{ cout << "Enter your choice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter the element to be inserted: ";

cin >> n;

lst.insertElement(n);

break;

case 2:

cout << "Enter the element to be deleted: ";

cin >> n;

lst.deleteElement(n);

break;

case 3:

cout << "Enter the element to be searched: ";

cin >> n;

lst.searchElement(n);

case 4:

cout << "The List is: ";

lst.displayList();

break;

case 5:

exit(1);

break;

default:

cout << "Wrong Choice" << endl;

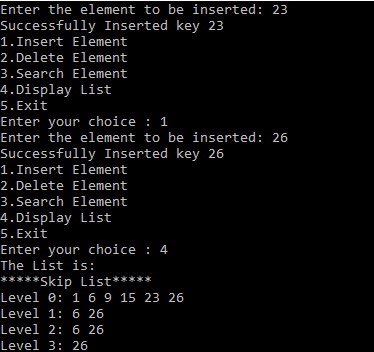
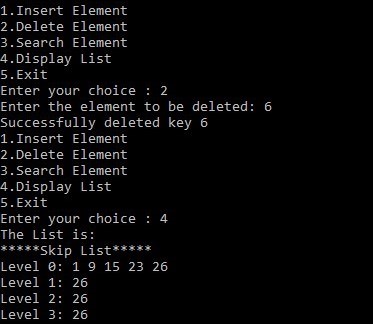
}

}

return 0;

}

**OUTPUT**

**LAB PROGRAM 3:**

**Implementation of Disjoint sets to find the number of islands in a given matrix**

#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;

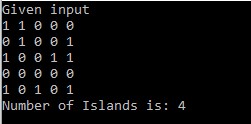
}

cout << "Number of Islands is: "

<< countIslands(a) << endl;

}

**OUTPUT**



**LAB PROGRAM 4:**

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

#include<bits/stdc++.h>

#include<queue>

#include<unordered\_map>

using namespace std;

class Node

{

public:

int key;

Node \*left;

Node \*right;

int height;

};

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 levelorder\_newline(struct Node \*v){

queue <struct Node \*> q;

struct Node \*cur;

q.push(v);

q.push(NULL);

while(!q.empty()){

cur = q.front();

q.pop();

if(cur == NULL && q.size()!=0){

cout<<"\n";

q.push(NULL);

continue;

}

if(cur!=NULL){

cout<<" "<<cur->key;

if (cur->left!=NULL){

q.push(cur->left);

}

if (cur->right!=NULL){

q.push(cur->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 << "\n----Tree----\n";

levelorder\_newline(root);

cout<<endl;

cout<<"Enter the element to be deleted"<<endl;

cin>>del;

root = deleteNode(root, del);

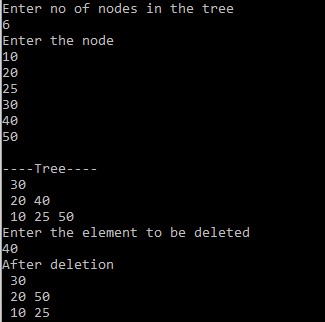
cout <<"After deletion \n";

levelorder\_newline(root);

return 0;

}

**OUTPUT**



**LAB PROGRAM 5:**

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

#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 ch;

int n,k;

cout<<"2-3 Trees"<<"\n";

cout<<"1.Insert\n2.Delete\n3.Display\n";

while(1)

{

cout<<"Enter your choice: ";

cin>>ch;

switch(ch)

{ case 1: 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);

}

break;

case 2: cout<<"Enter the key to be deleted"<<endl;

cin>>k;

t.remove(k);

break;

case 3: cout << "Traversal of tree constructed is\n";

t.traverse();

break;

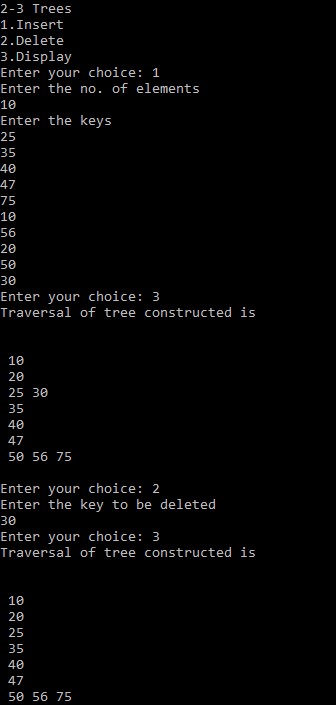
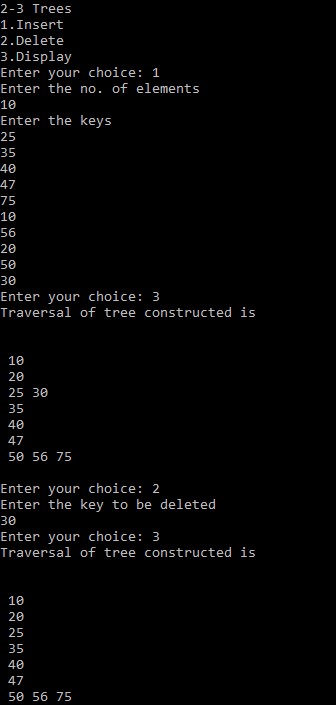
default: exit(0);

}

}

}

**OUTPUT**

**LAB PROGRAM 6:**

**Write a program to perform insertion operation on red-black tree.**

#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 <<"Element: "<<temp->data << " ";

cout<<"Color: "<<temp->color<<" \n";

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 << "\n\nLevel Order Traversal of Created Tree\n";

cout<<"1-Black 0-Red\n";

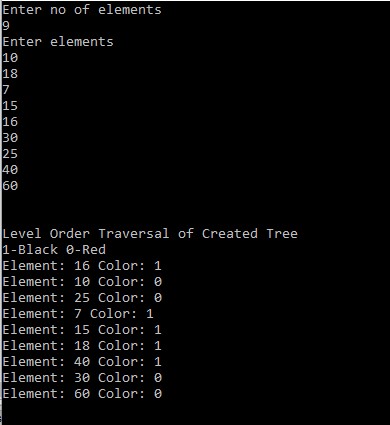
tree.levelOrder();

cout<<endl;

return 0;

}

**OUTPUT**



**LAB PROGRAM 7:**

**Write a program to perform insertion operation on a B-tree.**

#include<iostream>

using namespace std;

struct BTree{

int \*d;

BTree \*\*child\_ptr;

bool l;

int n;

};

struct BTree \*r = NULL, \*np = NULL, \*x = NULL;

BTree\* init(){

int i;

np = new BTree;

np->d = new int[6];//order 6

np->child\_ptr = new BTree \*[7];

np->l = true;

np->n = 0;

for (i = 0; i < 7; i++) {

np->child\_ptr[i] = NULL;

}

return np;

}

void traverse(BTree \*p){

cout<<endl;

int i;

for (i = 0; i < p->n; i++) {

if (p->l == false) {

traverse(p->child\_ptr[i]);

}

cout << " " << p->d[i];

}

if (p->l == false) {

traverse(p->child\_ptr[i]);

}

cout<<endl;

}

void sort(int \*p, int n){

int i, j, t;

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

for (j = i; j <= n; j++) {

if (p[i] >p[j]) {

t = p[i];

p[i] = p[j];

p[j] = t;

}

}

}

}

int split\_child(BTree \*x, int i) {

int j, mid;

BTree \*np1, \*np3, \*y;

np3 = init();//create new node

np3->l = true;

if (i == -1) {

mid = x->d[2];//find mid

x->d[2] = 0;

x->n--;

np1 = init();

np1->l= false;

x->l= true;

for (j = 3; j < 6; j++) {

np3->d[j - 3] = x->d[j];

np3->child\_ptr[j - 3] = x->child\_ptr[j];

np3->n++;

x->d[j] = 0;

x->n--;

}

for (j = 0; j < 6; j++) {

x->child\_ptr[j] = NULL;

}

np1->d[0] = mid;

np1->child\_ptr[np1->n] = x;

np1->child\_ptr[np1->n + 1] = np3;

np1->n++;

r = np1;

} else {

y = x->child\_ptr[i];

mid = y->d[2];

y->d[2] = 0;

y->n--;

for (j = 3; j <6 ; j++) {

np3->d[j - 3] = y->d[j];

np3->n++;

y->d[j] = 0;

y->n--;

}

x->child\_ptr[i + 1] = y;

x->child\_ptr[i + 1] = np3;

}

return mid;

}

void insert(int a) {

int i, t;

x = r;

if (x == NULL) {

r = init();

x = r;

} else {

if (x->l== true && x->n == 6) {

t = split\_child(x, -1);

x = r;

for (i = 0; i < (x->n); i++) {

if ((a >x->d[i]) && (a < x->d[i + 1])) {

i++;

break;

} else if (a < x->d[0]) {

break;

} else {

continue;

}

}

x = x->child\_ptr[i];

} else {

while (x->l == false) {

for (i = 0; i < (x->n); i++) {

if ((a >x->d[i]) && (a < x->d[i + 1])) {

i++;

break;

} else if (a < x->d[0]) {

break;

} else {

continue;

}

}

if ((x->child\_ptr[i])->n == 6) {

t = split\_child(x, i);

x->d[x->n] = t;

x->n++;

continue;

} else {

x = x->child\_ptr[i];

}

}

}

}

x->d[x->n] = a;

sort(x->d, x->n);

x->n++;

}

int main() {

int i, n, t;

cout<<"enter the no of elements to be inserted\n";

cin>>n;

cout<<"enter the elements\n";

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

cin>>t;

insert(t);

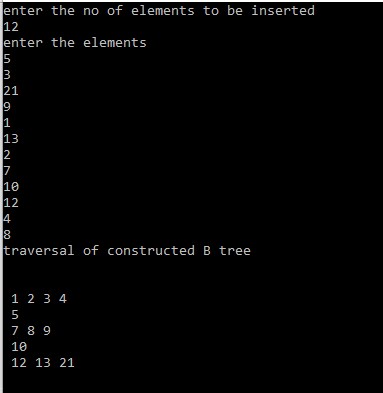
}

cout<<"traversal of constructed B tree\n";

traverse(r);

}

**OUTPUT**



**LAB PROGRAM 8:**

**Implementation of Dictionary using hashing**

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

# define max 10

typedef struct list

{

int data;

struct list \*next;

} node\_type;

node\_type \*ptr[max],\*root[max],\*temp[max];

class Dictionary

{

public:

int index;

Dictionary();

void insert(int);

void search(int);

void delete\_ele(int);

};

Dictionary::Dictionary()

{

index=-1;

for(int i=0; i<max; i++)

{

root[i]=NULL;

ptr[i]=NULL;

temp[i]=NULL;

}

}

void Dictionary::insert(int key)

{

index=int(key%max);

ptr[index]=(node\_type\*)malloc(sizeof(node\_type));

ptr[index]->data=key;

if(root[index]==NULL)

{

root[index]=ptr[index];

root[index]->next=NULL;

temp[index]=ptr[index];

}

else

{

temp[index]=root[index];

while(temp[index]->next!=NULL)

temp[index]=temp[index]->next;

temp[index]->next=ptr[index];

}

}

void Dictionary::search(int key)

{

int flag=0;

index=int(key%max);

temp[index]=root[index];

while(temp[index]!=NULL)

{

if(temp[index]->data==key)

{

cout<<"\nSearch key is found";

flag=1;

break;

}

else temp[index]=temp[index]->next;

}

if (flag==0)

cout<<"\nsearch key not found.......\n";

}

main()

{

int val,ch,n,num;

char c;

Dictionary d;

while(1)

{

cout<<"MENU:\n1.Create";

cout<<"\n2.Search for a value\n";

cout<<"\nEnter your choice:";

cin>>ch;

switch(ch)

{

case 1:

cout<<"\nEnter the number of elements to be inserted:";

cin>>n;

cout<<"\nEnter the elements to be inserted:";

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

{

cin>>num;

d.insert(num);

}

break;

case 2:

cout<<"\nEnter the element to be searched:";

cin>>n;

d.search(n);

break;

default:

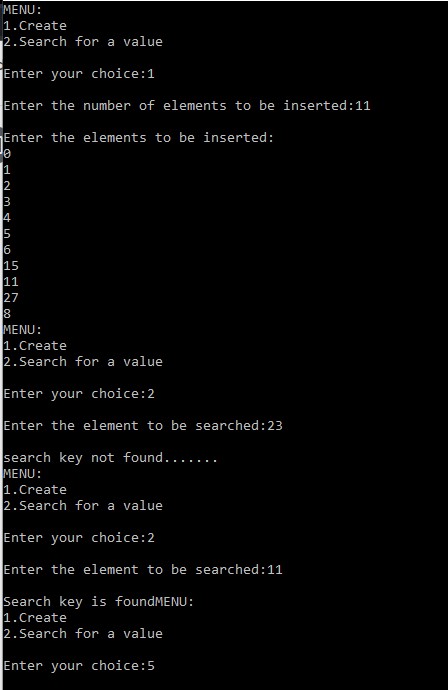
cout<<"\nInvalid Choice.";

}

}

}

**OUTPUT**



**LAB PROGRAM 9:**

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

1. insert(H, k)

2. getMin(H)

3. extractMin(H)

#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++;

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())

{

\_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;

}

list<Node\*> insertATreeInHeap(list<Node\*> \_heap, Node \*tree)

{

list<Node\*> temp;

temp.push\_back(tree);

temp = unionBionomialHeap(\_heap,temp);

return adjust(temp);

}

list<Node\*> removeMinFromTreeReturnBHeap(Node \*tree) {

list<Node\*> heap;

Node \*temp = tree->child;

Node \*lo;

while (temp){

lo = temp;

temp = temp->sibling;

lo->sibling = NULL;

heap.push\_front(lo);

}

return heap;

}

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

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);

new\_heap = adjust(new\_heap);

return new\_heap;

}

void printTree(Node \*h){

while (h){

cout << h->data << " ";

printTree(h->child);

h = h->sibling;

}

}

void printHeap(list<Node\*> \_heap){

list<Node\*> ::iterator it;

it = \_heap.begin();

while (it != \_heap.end())

{

printTree(\*it);

it++;

}

}

int main(){

int ch,key;

list<Node\*> \_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);

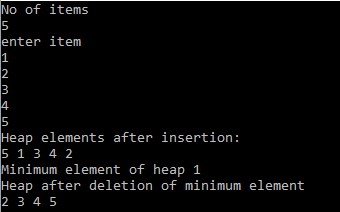
cout << "Heap after deletion of minimum element\n";

printHeap(\_heap);

return 0;

}

**OUTPUT**



**LAB PROGRAM 10:**

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

1. delete(H)

2. decreaseKey(H)

#include <iostream>

#include <cstdlib>

using namespace std;

struct node

{

int n;

int degree;

node\* parent;

node\* child;

node\* sibling;

};

class BinomialHeap

{

private:

node \*H;

node \*Hr;

int count;

public:

node\* Initializeheap();

int Binomial\_link(node\*, node\*);

node\* Create\_node(int);

node\* Union(node\*, node\*);

node\* Insert(node\*, node\*);

node\* Merge(node\*, node\*);

node\* Extract\_Min(node\*);

int Revert\_list(node\*);

int Display(node\*);

node\* Search(node\*, int);

int Decrease\_key(node\*, int, int);

int Delete(node\*, int);

BinomialHeap()

{

H = Initializeheap();

Hr = Initializeheap();

int count = 1;

}

};

node\* BinomialHeap::Initializeheap()

{

node\* np;

np = NULL;

return np;

}

int BinomialHeap::Binomial\_link(node\* y, node\* z)

{

y->parent = z;

y->sibling = z->child;

z->child = y;

z->degree = z->degree + 1;

}

node\* BinomialHeap::Create\_node(int k)

{

node\* p = new node;

p->n = k;

return p;

}

node\* BinomialHeap::Insert(node\* H, node\* x)

{

node\* H1 = Initializeheap();

x->parent = NULL;

x->child = NULL;

x->sibling = NULL;

x->degree = 0;

H1 = x;

H = Union(H, H1);

return H;

}

node\* BinomialHeap::Union(node\* H1, node\* H2)

{

node \*H = Initializeheap();

H = Merge(H1, H2);

if (H == NULL)

return H;

node\* prev\_x;

node\* next\_x;

node\* x;

prev\_x = NULL;

x = H;

next\_x = x->sibling;

while (next\_x != NULL)

{

if ((x->degree != next\_x->degree) || ((next\_x->sibling != NULL)

&& (next\_x->sibling)->degree == x->degree))

{

prev\_x = x;

x = next\_x;

}

else

{

if (x->n <= next\_x->n)

{

x->sibling = next\_x->sibling;

Binomial\_link(next\_x, x);

}

else

{

if (prev\_x == NULL)

H = next\_x;

else

prev\_x->sibling = next\_x;

Binomial\_link(x, next\_x);

x = next\_x;

}

}

next\_x = x->sibling;

}

return H;

}

node\* BinomialHeap::Merge(node\* H1, node\* H2)

{

node\* H = Initializeheap();

node\* y;

node\* z;

node\* a;

node\* b;

y = H1;

z = H2;

if (y != NULL)

{

if (z != NULL)

{

if (y->degree <= z->degree)

H = y;

else if (y->degree > z->degree)

H = z;

}

else

H = y;

}

else

H = z;

while (y != NULL && z != NULL)

{

if (y->degree < z->degree)

{

y = y->sibling;

}

else if (y->degree == z->degree)

{

a = y->sibling;

y->sibling = z;

y = a;

}

else

{

b = z->sibling;

z->sibling = y;

z = b;

}

}

return H;

}

int BinomialHeap::Display(node\* H)

{

if (H == NULL)

{

cout<<"The Heap is empty"<<endl;

return 0;

}

cout<<"The root nodes are: "<<endl;

node\* p;

p = H;

while (p != NULL)

{

cout<<p->n;

if (p->sibling != NULL)

cout<<"-->";

p = p->sibling;

}

cout<<endl;

}

node\* BinomialHeap::Extract\_Min(node\* H1)

{

Hr = NULL;

node\* t = NULL;

node\* x = H1;

if (x == NULL)

{

cout<<"Nothing to Extract"<<endl;

return x;

}

int min = x->n;

node\* p = x;

while (p->sibling != NULL)

{

if ((p->sibling)->n < min)

{

min = (p->sibling)->n;

t = p;

x = p->sibling;

}

p = p->sibling;

}

if (t == NULL && x->sibling == NULL)

H1 = NULL;

else if (t == NULL)

H1 = x->sibling;

else if (t->sibling == NULL)

t = NULL;

else

t->sibling = x->sibling;

if (x->child != NULL)

{

Revert\_list(x->child);

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

}

H = Union(H1, Hr);

return x;

}

int BinomialHeap::Revert\_list(node\* y)

{

if (y->sibling != NULL)

{

Revert\_list(y->sibling);

(y->sibling)->sibling = y;

}

else

{

Hr = y;

}

}

node\* BinomialHeap::Search(node\* H, int k)

{

node\* x = H;

node\* p = NULL;

if (x->n == k)

{

p = x;

return p;

}

if (x->child != NULL && p == NULL)

p = Search(x->child, k);

if (x->sibling != NULL && p == NULL)

p = Search(x->sibling, k);

return p;

}

int BinomialHeap::Decrease\_key(node\* H, int i, int k)

{

int temp;

node\* p;

node\* y;

node\* z;

p = Search(H, i);

if (p == NULL)

{

cout<<"Invalid choice of key"<<endl;

return 0;

}

if (k > p->n)

{

cout<<"Error!! New key is greater than current key"<<endl;

return 0;

}

p->n = k;

y = p;

z = p->parent;

while (z != NULL && y->n < z->n)

{

temp = y->n;

y->n = z->n;

z->n = temp;

y = z;

z = z->parent;

}

cout<<"Key reduced successfully"<<endl;

}

int BinomialHeap::Delete(node\* H, int k)

{

node\* np;

if (H == NULL)

{

cout<<"\nHEAP EMPTY!!!!!";

return 0;

}

Decrease\_key(H, k, -1000);

np = Extract\_Min(H);

if (np != NULL)

cout<<"Node Deleted Successfully"<<endl;

}

int main()

{

int n, m, l, i;

BinomialHeap bh;

node\* p;

node \*H;

H = bh.Initializeheap();

char ch;

cout<<"1)Insert Element in the heap"<<endl;

cout<<"3)Decrease key of a node"<<endl;

cout<<"4)Delete a node"<<endl;

cout<<"5)Display Heap"<<endl;

cout<<"6)Exit"<<endl;

while (1)

{

cout<<"Enter Your Choice: ";

cin>>l;

switch(l)

{

case 1:

cout<<"Enter the element to be inserted: ";

cin>>m;

cout<<"\n";

p = bh.Create\_node(m);

H = bh.Insert(H, p);

break;

case 3:

cout<<"Enter the key to be decreased: ";

cin>>m;

cout<<"Enter new key value: ";

cin>>l;

bh.Decrease\_key(H, m, l);

break;

case 4:

cout<<"Enter the key to be deleted: ";

cin>>m;

bh.Delete(H, m);

break;

case 5:

cout<<"The Heap is: "<<endl;

bh.Display(H);

break;

case 6:

exit(1);

default:

cout<<"Wrong Choice";

}

}

return 0;

}

**OUTPUT**

