VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

ADVANCE DATA STRUCTURES

LAB REPORT

For

Lab Cycle 2

Submitted by

PRITHVI.J(1BM19CS122)

Under the Guidance of

Dr. LATHA N.R

Assistant Professor, BMSCE

in partial fulfilment 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

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 Advance Data Structures Lab for Cycle 2 (CIE 2) carried out by, PRITHVI.J (1BM19CS122) who are Bonafede students of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visveswaraiah Technological University, Belgaum during the year 2021-2022. The Lab report has been approved as it satisfies the academic requirements in respect of ADVANCE DATA STRUCTURES (20CS5PCADS) work prescribed for the said degree.

Signature of the Guide

Dr. LATHA N.R

Assistant Professor

BMSCE, Bengaluru

Signature of the HOD

Dr. Umadevi V

Associate Prof.& Head, Dept. of CSE

BMSCE, Bengaluru

Index

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

```
#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
{
	Node *root;
	public:
		void rotateLeft(Node *&, Node *&);
		void rotateRight(Node *&, Node *&);
		void fixViolation(Node *&, Node *&);
	RBTree() { root = NULL; }
	void insert(int &n);
	void inorder();
	void levelOrder();
};

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

	inorderHelper(root->left);
	cout << root->data << ":";
	if(root->color == 1)
		cout<<"Black ";
```

```
        else
                cout<<"Red ";
        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 << ":";
                if(temp->color == 1)
                        cout<<"Black ";
                else
                        cout<<"Red ";
        q.pop();

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

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

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(int &data)
{
	Node *pt = new Node(data);
	root = BSTInsert(root, pt);
	fixViolation(root, pt);
}

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

int main()
{
	RBTree tree;
	int n, key;
	cout<<"Enter the no. of elements:"<<endl;
	cin>>n;
	cout<<"Enter the elements:"<<endl;
	for(int i=0; i<n; i++)
```

```
        {
                cin>>key;
                tree.insert(key);
                cout << "Level Order Traversal after inserting "<<key<<" : "<<endl;
                tree.levelOrder();
        }
        cout<<endl;
        cout << "Inoder Traversal of Created Tree"<<endl;
        tree.inorder();
        cout<<endl;



        return 0;
}
```

Output:

```
/*

Enter the no. of elements:
4
Enter the elements:
4 8 6 9
Level Order Traversal after inserting 4 :
4:Black
Level Order Traversal after inserting 8 :
4:Black 8:Red
Level Order Traversal after inserting 6 :
6:Black 4:Red 8:Red
Level Order Traversal after inserting 9 :
6:Black 4:Black 8:Black 9:Red

Inoder Traversal of Created Tree
4:Black 6:Black 8:Black 9:Red

*/
```

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

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

class BTreeNode
{
	int *keys;
	BTreeNode **child;
	int t;
	int n;
	bool leaf;
	public:
		BTreeNode(int t, bool leaf);
		void traverse();
		void insertNonFull(int k);
		void splitChild(int i, BTreeNode *y);
		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];
	child = new BTreeNode *[2*t];
	n = 0;
}
```

```
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->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 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(child[i+1]->n == 2*t-1)
		{
			splitChild(i+1, child[i+1]);
			if(keys[i+1]<k)
				i++;
		}
```

```
            child[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->child[j] = y->child[j+t];
        }
        y->n = t-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[t-1];

        n = n + 1;
}

void BTreeNode::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;
}
```

```
int main()
{
	int d;
	cout<<"Enter the degree: ";
	cin>>d;
	BTree t(d);
	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<<endl;
	return 0;

}
```

Output:

```
/*

Enter the degree: 5
Enter the no. of elements
8
Enter the keys
10 11 12 13 14 15 16 17
Traversal of tree constructed is
10 11 12 13 14 15 16 17

*/
```

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

```
#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:

```
/*

1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 1
Enter element to be inserted: 9
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: 18
Enter key at which element to be inserted: 5
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 1
Enter element to be inserted: 21
Enter key at which element to be inserted: 7
```

```
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 1
Enter element to be inserted: 29
Enter key at which element to be inserted: 6
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 21
No element found at key 21
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 7
Element at key 7 : 21
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 3
Enter key of the element to be deleted: 7
Element Deleted
1.Insert
2.Search
3.Delete
4.Exit
Enter your choice: 4
PS D:\codes\Practice Code\C++\lab>


*/
```

4. Write a program to implement the following functions on a Binomial heap:
    1. insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
    2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.
    3. extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap.

```
#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++;
        }
        cout<<endl;
}

int main()
{
        int ele,n;
        list<Node*> _heap;
        cout<<"Enter the no. of elements:"<<endl;
        cin>>n;
        cout<<"Enter the elements to be inserted:"<<endl;
        for(int i=0; i<n; i++)
        {
                cin>>ele;
                _heap = insert(_heap,ele);
        }

        cout << "Heap elements after insertion:"<<endl;
        printHeap(_heap);
```

```
        Node *temp = getMin(_heap);
        cout << "Minimum element of heap: "<< temp->data <<endl;

        _heap = extractMin(_heap);
        cout << "Heap after deletion of minimum element: "<<endl;
        printHeap(_heap);

        return 0;
}
```

Output:

```
/*

Enter the no. of elements:
10
Enter the elements to be inserted:
15 69 89 56 20 31 14 25 4 16
Heap elements after insertion:
4 16 14 15 56 89 69 20 31 25
Minimum element of heap: 4
Heap after deletion of minimum element:
16 14 15 56 89 69 20 31 25

*/
```

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

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

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

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

Node *root = NULL;

void 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;
        }
}

void 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 ele,n;
        cout<<"Enter the no. of elements:"<<endl;
        cin>>n;
        cout<<"Enter the elements to be inserted:"<<endl;
        for(int i=0; i<n; i++)
```

```
	{
		cin>>ele;
		binomialHeapInsert(ele);
	}

	cout << "Heap elements after insertion:"<<endl;
	display(root);
	cout<<endl;

	cout << "Enter the number of nodes to be deleted: ";
	cin >> n;
	cout<<"Enter the elements to be deleted: "<<endl;
	for(int i=0; i<n; i++)
	{
		cin>>ele;
		root = binomialHeapDelete(root,  ele);
		cout << "After deleting " << ele << ": \n";
		display(root);
		cout << endl;
	}

	return 0;
}
```

Output:

```
/*

Enter the no. of elements:
10
Enter the elements to be inserted:
14 25 63 26 85 96 56 75 21 222
Heap elements after insertion:
21 222 14 56 85 96 75 26 63 25
Minimum element of heap: 14
Heap after deletion of minimum element:
25 21 56 85 96 75 26 63 222

*/
```