**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 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.

CODE:

#include<bits/stdc++.h>

using namespace std;

struct Node

{

int data, degree;

Node \*child, \*sibling, \*parent;

};

Node\* newNode(int key)

{

Node \*temp = new Node;

temp->data = key;

temp->degree = 0;

temp->child = temp->parent = temp->sibling = NULL;

return temp;

}

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

{

if (b1->data > b2->data)

swap(b1, b2);

b2->parent = b1;

b2->sibling = b1->child;

b1->child = b2;

b1->degree++;

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,n;

list<Node\*> \_heap;

cout << "Enter number of elements: ";

cin>>n;

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

cout << "Enter value: ";

cin>>key;

\_heap=insert(\_heap,key);

}

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:

