**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 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.

CODE:

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int val, degree;

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

};

Node \*root = NULL;

// link two heaps by making h1 a child

// of h2.

int binomialLink(Node \*h1, Node \*h2)

{

h1->parent = h2;

h1->sibling = h2->child;

h2->child = h1;

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

return 0;

}

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;

}

// merge two Binomial Trees

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

{

if (h1 == NULL)

return h2;

if (h2 == NULL)

return h1;

Node \*res = NULL;

// check degree of both Node i.e.

// which is greater or smaller

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

res = h1;

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

res = h2;

// traverse till if any of heap gets empty

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

{

// if degree of h1 is smaller, increment h1

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

h1 = h1->sibling;

// Link h1 with h2 in case of equal degree

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

{

Node \*sib = h1->sibling;

h1->sibling = h2;

h1 = sib;

}

// if h2 is greater

else

{

Node \*sib = h2->sibling;

h2->sibling = h1;

h2 = sib;

}

}

return res;

}

// union operation on two binomial heap i.e. h1 & h2

Node \*unionBHeaps(Node \*h1, Node \*h2)

{

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

return NULL;

Node \*res = mergeBHeaps(h1, h2);

// Traverse the merged list and set

// values according to the degree of

// Nodes

Node \*prev = NULL, \*curr = res,

\*next = curr->sibling;

while (next != NULL)

{

if ((curr->degree != next->degree) ||

((next->sibling != NULL) &&

(next->sibling)->degree ==

curr->degree))

{

prev = curr;

curr = next;

}

else

{

if (curr->val <= next->val)

{

curr->sibling = next->sibling;

binomialLink(next, curr);

}

else

{

if (prev == NULL)

res = next;

else

prev->sibling = next;

binomialLink(curr, next);

curr = next;

}

}

next = curr->sibling;

}

return res;

}

// insert a Node

void binomialHeapInsert(int x)

{

// Create a new node and do union of

// this node with root

root = unionBHeaps(root, createNode(x));

}

// display the Nodes

void display(Node \*h)

{

while (h)

{

cout << h->val << " ";

display(h->child);

h = h->sibling;

}

}

// reverse a list using recursion.

int revertList(Node \*h)

{

if (h->sibling != NULL)

{

revertList(h->sibling);

(h->sibling)->sibling = h;

}

else

root = h;

return 0;

}

// extract minimum value

Node \*extractMinBHeap(Node \*h)

{

if (h == NULL)

return NULL;

Node \*min\_node\_prev = NULL;

Node \*min\_node = h;

// Find minimum value

int min = h->val;

Node \*curr = h;

while (curr->sibling != NULL)

{

if ((curr->sibling)->val < min)

{

min = (curr->sibling)->val;

min\_node\_prev = curr;

min\_node = curr->sibling;

}

curr = curr->sibling;

}

// If there is a single Node

if (min\_node\_prev == NULL &&

min\_node->sibling == NULL)

h = NULL;

else if (min\_node\_prev == NULL)

h = min\_node->sibling;

// Remove min node from list

else

min\_node\_prev->sibling = min\_node->sibling;

// Set root (which is global) as children

// list of min node

if (min\_node->child != NULL)

{

revertList(min\_node->child);

(min\_node->child)->sibling = NULL;

}

// Do union of root h and children

return unionBHeaps(h, root);

}

// search for an element

Node \*findNode(Node \*h, int val)

{

if (h == NULL)

return NULL;

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

if (h->val == val)

return h;

// Recur for child

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

if (res != NULL)

return res;

return findNode(h->sibling, val);

}

// decrease the value of old\_val to new\_val

void decreaseKeyBHeap(Node \*H, int old\_val,int new\_val)

{

// First check element present or not

Node \*node = findNode(H, old\_val);

// return if Node is not present

if (node == NULL)

return;

// Reduce the value to the minimum

node->val = new\_val;

Node \*parent = node->parent;

// Update the heap according to reduced value

while (parent != NULL && node->val < parent->val)

{

swap(node->val, parent->val);

node = parent;

parent = parent->parent;

}

}

// delete an element

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

{

// Check if heap is empty or not

if (h == NULL)

return NULL;

// Reduce the value of element to minimum

decreaseKeyBHeap(h, val, INT\_MIN);

// Delete the minimum element from heap

return extractMinBHeap(h);

}

// Driver code

int main()

{

// Note that root is global

int key,n,x,y;

cout<<"Enter the number of values to be inserted:\n";

cin>>n;

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

cout<<"Enter the value to be inserted:\n";

cin>>key;

binomialHeapInsert(key);

}

cout << "The heap is:\n";

display(root);

cout<<"\nEnter the number of values to be deleted:\n";

cin>>y;

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

cout<<"\nEnter the value to be deleted:\n";

cin>>x;

root = binomialHeapDelete(root, x);

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

display(root);

}

return 0;

}

OUTPUT:

