#include<iostream>

#include<list>

#include<algorithm>

#include<climits>

using namespace std;

// A Binomial Tree node.

struct Node

{

int data, degree;

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

};

class BHeap

{

list<Node\*> heap;

public:

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 \*mergeBTrees(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. h1 & h2

list<Node \*> Union(list<Node \*> &h1,

list<Node \*> &h2)

{

// \_new to another binomial heap which contain

// new heap after merging h1 & h2

list<Node \*> \_new;

list<Node \*>::iterator it = h1.begin();

list<Node \*>::iterator ot = h2.begin();

while (it != h1.end() && ot != h2.end())

{

// if D(h1) <= D(h2)

if ((\*it)->degree <= (\*ot)->degree)

{

\_new.push\_back(\*it);

it++;

}

// if D(h1) > D(h2)

else

{

\_new.push\_back(\*ot);

ot++;

}

}

// if there remains some elements in h1 binomial heap

while (it != h1.end())

{

\_new.push\_back(\*it);

it++;

}

// if there remains some elements in h2 binomial heap

while (ot != h2.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

void adjust()

{

if (heap.size() <= 1)

return ;

list<Node \*> newheap;

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 = mergeBTrees(\*it1, \*it2);

it2 = heap.erase(it2);

if (it3 != heap.end())

it3++;

}

}

}

// inserting a Binomial Tree into binomial heap

void insertTree(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

heap = Union(heap, temp);

adjust();

}

// inserting a key into the binomial heap

void insert(int key)

{

Node \*temp = newNode(key);

insertTree(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 \*> removeMinTree(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;

}

// return pointer of minimum value Node

// present in the binomial heap

Node \*getMin()

{

list<Node \*>::iterator it = heap.begin();

Node \*temp = \*it;

while (it != heap.end())

{

if ((\*it)->data < temp->data)

temp = \*it;

it++;

}

return temp;

}

// Function to delete the Minimum element of Binomial Heap

void extractMin()

{

list<Node \*> newheap, lo;

Node \*temp;

// temp contains the pointer of minimum value

// element in heap

temp = getMin();

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

newheap.push\_back(\*it);

}

it++;

}

lo = removeMinTree(temp);

heap = Union(newheap, lo);

adjust();

}

// Function to search for an element

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

{

if (it == NULL)

return NULL;

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

if (it->data == val)

return it;

/\*// if value is less than root's data then it is not in this tree

// because it follows Min-Heap property

if(it->data > val)

return NULL;

\*/

// Recur for child

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

if (res != NULL)

return res;

return findNode(it->sibling, val);

}

// Function to decrease the value of old\_val

// to new\_val

void decreaseKey( int old\_val,

int new\_val)

{

// First check element present or not

list<Node \*>::iterator it=heap.begin();

Node \*node = NULL;

while (it != heap.end() && node == NULL)

{

node = findNode(\*it, old\_val);

it++;

}

// return if Node is not present

if (node == NULL)

return;

// Reduce the value to the minimum

node->data = new\_val;

Node \*parent = node->parent;

// Update the heap according to reduced value

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

{

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

node = parent;

parent = parent->parent;

}

}

// Function to delete an element

void Delete(int val)

{

// Reduce the value of element to minimum

decreaseKey(val, INT\_MIN);

// Delete the minimum element from heap

extractMin();

}

// 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 \*>::iterator it;

it = heap.begin();

while (it != heap.end())

{

printTree(\*it);

it++;

}

cout<<"\n";

}

};

// Driver program to test above functions

int main()

{

int ch,key;

BHeap H;

H.insert(10);

H.insert(20);

H.insert(30);

H.insert(40);

H.insert(50);

H.insert(60);

H.insert(70);

H.insert(80);

H.insert(90);

cout<< "Heap elements after insertion:\n";

H.printHeap();

Node \*temp = H.getMin();

cout << "Minimum element of heap "

<< temp->data << "\n";

// Decrease key of 30 to 8

H.decreaseKey(30, 8);

temp = H.getMin();

cout << "Now Minimum element of heap "<< temp->data << "\n";

// Delete minimum element of heap

H.extractMin();

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

H.printHeap();

H.Delete(20);

cout << "Heap after deletion of 20\n";

H.printHeap();

return 0;

}