#include <iostream> //console input and output

#include <string> //using the string data type

#include <cstring> //for using c\_str()

using namespace std;

struct Node

{

int priority; //priority of person in queue

char name[20];

};

class priorityQueue

{

private:

Node x[100];

int endIndex;

//Max Heap is a heap in which the parent is always

//smaller than all of its children.

//Max Heap Priority Queue is a priority queue which

//is a max heap before an enqueue or a dequeue operation

//is started. In carrying out any of these operations,

//its max heap property may be temporarily disturbed.

//However, at the completion of each of these operation,

//the priority queue must be a max heap again.

void maxreheapifyUpward(Node [], int);

void maxreheapifyDownward(Node [], int);

int findLargeChildIndex(int, int);

public:

Node userInput;

priorityQueue();

//This method will be used to enqueue an item to the priority queue.

//This method will receive a customer record as a parameter and

//queue it to the priority queue

void penque(int, string);

//Pre-condition: This method will be called only when the que is

//NOT empty. This method will be used to deque an item from the

//priority queue

Node pdeque();

//This method will return true or 1 if the queue is empty.

//Otherwise, it will return false or 0. This method can be

//called to ensure that the queue is not empty before

//calling the dequeue method above

bool isEmpty();

};

priorityQueue::priorityQueue() { endIndex = -1; } //constructor

void priorityQueue::penque(int p, string n)

{

endIndex++;

x[endIndex].priority = p;

strcpy(x[endIndex].name, n.c\_str());

maxreheapifyUpward(x,endIndex);

}

Node priorityQueue::pdeque()

{

Node returnValue = x[0];

x[0] = x[endIndex];

endIndex--;

maxreheapifyDownward(x,endIndex);

return returnValue;

}

bool priorityQueue::isEmpty() { return endIndex < 0; }

void priorityQueue::maxreheapifyUpward (Node x [], int endIndex)

{

int parentIndex;

int childIndex = endIndex;

while (childIndex > 0 )

{

parentIndex = childIndex/2;

if (x[childIndex].priority <= x[parentIndex].priority) break;

else

{

//swap values at child and at parent.

Node p = x[childIndex];

x[childIndex] = x[parentIndex];

x[parentIndex] = p;

//Update child to parent childIndex = parentIndex;

childIndex = parentIndex;

}

}

}

void priorityQueue::maxreheapifyDownward (Node x [], int endIndex)

{

int parentIndex = 0;

int largeChildIndex;

while (parentIndex < endIndex)

{

largeChildIndex = findLargeChildIndex(parentIndex, endIndex);

if (largeChildIndex<0 || x[largeChildIndex].priority <= x[parentIndex].priority) break;

else

{

Node p = x[largeChildIndex];

x[largeChildIndex] = x[parentIndex];

x[parentIndex] = p;

parentIndex = largeChildIndex;

}

}

}

int priorityQueue::findLargeChildIndex(int parentIndex, int endIndex)

{

int lChildIndex = (2 \* parentIndex) + 1; //locate index of left child

int rChildIndex = (2 \* parentIndex) + 2; //locate index of right child

if (rChildIndex <= endIndex && lChildIndex <= endIndex)

{

if(x[lChildIndex].priority >= x[rChildIndex].priority) return lChildIndex;

else return rChildIndex;

}

else if (lChildIndex <= endIndex) return lChildIndex;

return -1;

}

int main()

{

priorityQueue recordList;

while(recordList.userInput.priority >= 0)

{

cout << "Enter the queue number: ";

cin >> recordList.userInput.priority;

if (recordList.userInput.priority == -1) break;

cout << "Enter the persons name: ";

cin >> recordList.userInput.name;

recordList.penque(recordList.userInput.priority, recordList.userInput.name);

}

while(!recordList.isEmpty())

{

recordList.userInput = recordList.pdeque();

cout << recordList.userInput.priority << " " << recordList.userInput.name << endl;

}

}

