CSE 330 LABORATORY -- Week 9, Spring 2018

Instructor: Kerstin Voigt

In this lab we will implement a class called BinaryHeap which will form (in some sense already "is") the ADT PriorityQueue. You will have discussed the details of priority queue data structures in the lecture and you will have learned about the surprising fact that certain binary tree structures ("complete" binary trees) can be implemented in terms of linear vector. Naturally, if a hierarchical structure can be represented without the overhead of having to maintain special pointers to parent and child nodes, then this is the more efficient avenue to follow.

Exercise 1: Copy the code below into a file BinaryHeap.h. Notice that a choice was made to define the member functions with the larger bodies *outside* the scope of class BinaryHeap. It is good to practice this alternative layout of your C++ code, and some would say that moving the bulk of the function definitions outside of the class is preferred. Here, we will not argue, but simply explore this other way of doing things …

```
// BinaryHeap.h
// May 2018, after Weiss, Data Structures textbook

#ifndef BINARYHEAP_H_
#define BINARYHEAP_H_

#include <cassert>
#include "Vector.h"   // Yes, we are using our Vector …
using namespace std;


template <typename C>
class BinaryHeap
{
 public:
  BinaryHeap()
  : heap(100), currentSize(0){}

  BinaryHeap(int capacity)
    : heap(capacity), currentSize(0) {}

  BinaryHeap(const Vector<C>&);

  bool isEmpty() const {return currentSize == 0;}

  int size() const {return currentSize;}

  const C & findMin() const {return heap[1];}

  void insert(const C &);
  void insert(C &&);
  void deleteMin();
  void deleteMin(C &);
  void makeEmpty()
  {
    while (!heap.isEmpty())
      heap.pop_back();
    currentSize = 0;
  }

 private:

  int currentSize;
  Vector<C> heap;

  void buildHeap()
  {
    for (int i = currentSize/2; i > 0; i--)
      percolateDown(i);
  }

  void percolateDown(int);
  void printHeap(int,int) const;
};

template <typename C>
BinaryHeap<C>::BinaryHeap(const Vector<C>& items)
: heap(items.size() + 10), currentSize(items.size())
{
  for (int i = 0; i < items.size(); i++)
    heap[i + 1] = items[i];
  buildHeap();
}

template <typename C>
void BinaryHeap<C>::insert(const C& x)
{
  if (currentSize == heap.size()-1)
    heap.resize(heap.size()*2);

  int hole = ++currentSize;
  C copy = x;

  heap[0] = std::move(copy);

  for (; x < heap[hole/2]; hole /= 2)
    heap[hole] = std::move(heap[hole/2]);
  heap[hole] = std::move(heap[0]);
}

template <typename C>
void BinaryHeap<C>::insert(C&& x)
{
  if (currentSize == heap.size()-1)
    heap.resize(heap.size()*2);

  int hole = ++currentSize;
  C copy = x;

  heap[0] = std::move(copy);
  for (; x < heap[hole/2]; hole /= 2)
    heap[hole] = std::move(heap[hole/2]);
  heap[hole] = std::move(heap[0]);
}

template <typename C>
void BinaryHeap<C>::deleteMin()
{
  assert(!isEmpty());
  heap[1] = std::move(heap[currentSize--]);
  percolateDown(1);
}

template <typename C>
void BinaryHeap<C>::deleteMin(C & minItem)
{
  assert(!isEmpty());
  minItem = std::move(heap[1]);
  heap[1] = std::move(heap[currentSize--]);
  percolateDown(1);
}

template <typename C>
void BinaryHeap<C>::percolateDown(int hole)
{
  int child;
  C tmp = std::move(heap[hole]);

  for (; hole * 2 <= currentSize; hole = child)
    {
      child  = hole * 2;
      if (child != currentSize && heap[child + 1] < heap[child])
         child++;

      if (heap[child] < tmp)
         heap[hole] = std::move(heap[child]);
      else
         break;
    }
  heap[hole] = std::move(tmp);
}

#endif
```

Exercise 2: Put our class BinaryHeap to the test by running the following int main(). Copy the following into a file BinaryHeapMain.cpp.

```
// BinaryHeapMain.cpp
// May 2018

#include <iostream>
#include "BinaryHeap.h"
using namespace std;
```

```
int main()
{
  BinaryHeap<int> mypq;
  int howmany;
  int next;

  cout << "How many items to store? ";
  cin >> howmany;
  cout << endl;
  for (int i = 1; i <= howmany; i++)
    {
      cout << "Item: ";
      cin >> next;
      cout << endl;
      mypq.insert(next);
    }
  cout << endl;

  //mypq.printHeap();
  cout << endl << endl;

  cout << "Breaking down the PQ top-down ..." << endl;
  while (!mypq.isEmpty())
    {
      mypq.deleteMin(next);
        cout << "removed: " << next << endl;
      //mypq.printHeap();
      //cout << endl << endl;
    }
  cout << endl;
  return 0;
}
```

Exercise 3: It there is time left, enhance your program with the ability to print out the internal tree structure of an instance of Binary Heap. To this end, consider implementing a member function BinaryHeap<C>::printHeap(). How would you do this? The task is has some analogy to the printing of the internal structure or BinarySearchTree …

Credit for this lab: (1) After working diligently on the above, sign up on the signup sheet. There is nothing to be handed in for this lab, but today's implementation will be the basis for our last upcoming homework assignment.