CSCI 140 PA 13 Submission

Due Date: 11/30/2021 Late (date and time):________________

Name(s): Nero Li

The header file for both exercise

Source code below:

```
#ifndef HPQ_H
#define HPQ_H

#include <list>
#include <vector>

template <typename E>
class VectorCompleteTree
{
private:                                                          // member
data
    std::vector<E> V;                                           // tree
contents
public:                                                               //
publicly accessible types
    typedef typename std::vector<E>::iterator Position; // a position in
the tree
protected:                                                        //
protected utility functions
    Position pos(int i)                                         // map an
index to a position
      { return V.begin() + i; }
    int idx(const Position& p) const                    // map a position
to an index
      { return p - V.begin(); }
public:
    VectorCompleteTree() : V(1) {}                          // constructor
    int size() const                                            { return
V.size() - 1; }
    Position left(const Position& p)                    { return
pos(2*idx(p)); }
    Position right(const Position& p)                   { return
pos(2*idx(p) + 1); }
    Position parent(const Position& p)                        { return
pos(idx(p)/2); }
    bool hasLeft(const Position& p) const               { return 2*idx(p)
<= size(); }
    bool hasRight(const Position& p) const              { return 2*idx(p) +
1 <= size(); }
```

```
    bool isRoot(const Position& p) const                { return idx(p) ==
1; }
    Position root()
{ return pos(1); }
    Position last()
{ return pos(size()); }
    void addLast(const E& e)
{ V.push_back(e); }
    void removeLast()
{ V.pop_back(); }
    void swap(const Position& p, const Position& q)     { E e = *q; *q =
*p; *p = e; }
};

template <typename E, typename C>
class HeapPriorityQueue
{
public:
    int size() const;                       // number of elements
    bool empty() const;                     // is the queue empty?
    void insert(const E& e);          // insert element
    const E& min();                             // minimum element
    void removeMin();                       // remove minimum
private:
    VectorCompleteTree<E> T;          // priority queue contents
    C isLess;                                   // less-than comparator
                                              // shortcut for tree position
    typedef typename VectorCompleteTree<E>::Position Position;
};

template <typename E, typename C>           // number of elements
int HeapPriorityQueue<E,C>::size() const
{
    return T.size();
}

template <typename E, typename C>           // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
{
    return size() == 0;
}

template <typename E, typename C>           // minimum element
const E& HeapPriorityQueue<E,C>::min()
{
    return *(T.root());                 // return reference to root
element
}

template <typename E, typename C>           // insert element
void HeapPriorityQueue<E,C>::insert(const E& e)
{
    T.addLast(e);                             // add e to heap
    Position v = T.last();                      // e's position
    while (!T.isRoot(v))                // up-heap bubbling
    {
        Position u = T.parent(v);
        if (!isLess(*v, *u)) break;         // if v in order, we're done
        T.swap(v, u);                           // ...else swap with parent
        v = u;
    }
}

template <typename E, typename C>           // remove minimum
void HeapPriorityQueue<E,C>::removeMin()
{
    if (size() == 1)                        // only one node?
        T.removeLast();                     // ...remove it
    else
    {
        Position u = T.root();              // root position
        T.swap(u, T.last());                // swap last with root
        T.removeLast();                         // ...and remove last
        while (T.hasLeft(u))            // down-heap bubbling
        {
            Position v = T.left(u);
            if (T.hasRight(u) && isLess(*(T.right(u)), *v))
                v = T.right(u);             // v is u's smaller child
            if (isLess(*v, *u))         // is u out of order?
            {
                T.swap(u, v);               // ...then swap
                u = v;
            }
            else break;                     // else we're done
        }
    }
}

#endif
```

Exercise 1 -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:

```
/*  Program: PA_13_exercise_1
    Author: Nero Li
    Class: CSCI 220
    Date: 11/30/2021
    Description:
        Put together heap priority queue and use a test driver to perform
        some operations to confirm it is working correctly. You can use a
        PQ with integer as element. Create two PQ objects – one with
largest
```

```
        value having highest priority and one with lowest value having
        highest priority. Be sure to use a comparator for the PQ.

    I certify that the code below is my own work.

      Exception(s): N/A

*/
#include <iostream>
#include "HeapPriorityQueue.h"

using namespace std;

template <typename E>
class isLess
{
public:
    bool operator()(const E& p, const E& q) const
    {
        return p < q;
    }
};

template <typename E>
class isMore
{
public:
    bool operator()(const E& p, const E& q) const
    {
        return p > q;
    }
};

int main()
{
    HeapPriorityQueue<int, isLess<int>> test1;
    HeapPriorityQueue<int, isMore<int>> test2;

    test1.insert(5);
    test1.insert(4);
    test1.insert(7);
    test1.insert(1);
    cout << test1.min() << ' ';
    test1.removeMin();
    test1.insert(3);
    test1.insert(6);
    cout << test1.min() << ' ';
    test1.removeMin();
    cout << test1.min() << ' ';
    test1.removeMin();
    test1.insert(8);
    cout << test1.min() << ' ';
    test1.removeMin();
    test1.insert(2);
    cout << test1.min() << ' ';
    test1.removeMin();
    cout << test1.min() << ' ';
    test1.removeMin();
    cout << endl;

    test2.insert(5);
    test2.insert(4);
    test2.insert(7);
    test2.insert(1);
    cout << test2.min() << ' ';
    test2.removeMin();
    test2.insert(3);
    test2.insert(6);
    cout << test2.min() << ' ';
    test2.removeMin();
    cout << test2.min() << ' ';
    test2.removeMin();
    test2.insert(8);
    cout << test2.min() << ' ';
    test2.removeMin();
    test2.insert(2);
    cout << test2.min() << ' ';
    test2.removeMin();
    cout << test2.min() << ' ';
    test2.removeMin();
    cout << endl;

    cout << "Modified by: Nero Li\n";
    return 0;
}
```

Input/output below:

```
1 3 4 5 2 6
7 6 5 8 4 3
Modified by: Nero Li
```

Exercise 2 (with extra credit) -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:

```
/*  Program: PA_13_exercise_1
    Author: Nero Li
    Class: CSCI 220
    Date: 11/30/2021
    Description:
        Use your priority queue from exercise 1 to sort data in ascending
order. Sort
        the data file small1k.txt, containing a list of 1,000 integer
values, and output
        the first 5 and last 5 values to the screen (5 values on one line
and at least
        one space between the 2 values). Sort the data file large100k.txt,
containing
        a list of 100,000 integer values, and output the first 5 and last
5 values to
        the screen (5 values on one line and at least one space between
the 2 values).
        For each set of data, collect actual run times in milliseconds and
display to
        the screen as well.

    I certify that the code below is my own work.

      Exception(s): N/A

*/
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include "HeapPriorityQueue.h"

using namespace std;

template <typename E>
class isLess
{
public:
    bool operator()(const E& p, const E& q) const
    {
        return p < q;
    }
};

void func(string str)
{
    HeapPriorityQueue<int, isLess<int>> pq;
    ifstream fin;
    int n{0};
    int i{0};

    fin.open(str, ios::binary);

    auto start = chrono::high_resolution_clock::now();
    while (!fin.eof())
    {
        fin >> n;
        pq.insert(n);
    }

    n = pq.size();

    while (!pq.empty())
    {
        if (i < 5 || i > n - 6)
        {
            cout << pq.min() << ' ';
        }
        if (i == 5 || i == n - 1)
        {
            cout << endl;
        }
        ++i;
        pq.removeMin();
    }
    auto end = chrono::high_resolution_clock::now();
    cout << (chrono::duration_cast<chrono::nanoseconds>(end -
start).count() * (double)1e-6) << " ms" << endl;
}

int main()
{
    func("small1k.txt");
    func("large100k.txt");

    cout << "Modified by: Nero Li\n";
    return 0;
}
```

Input/output below:

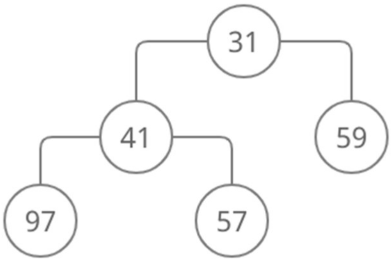
```
7 11 15 39 59
8163 8167 8175 8183 8191
3.9072 ms
1 2 3 4 5
99996 99997 99998 99999 100000
357.077 ms
Modified by: Nero Li
```

Answer for Question 1:

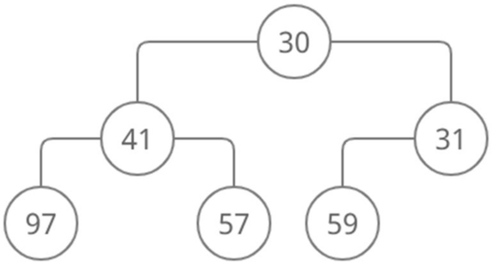
Heap is a binary tree that stores values, if we output the value for the root, based on what comparator we create, we will have a general minimum value or the value that should at the first place. Although we might not get a list that is already ordered, we can still output minimum value as what Priority Queue does. Furthermore, since we are using the binary tree, we can choose avoid sorting all the value. Hence, we saved more time for output the minimum value than a general list priority queue.

Answer for Question 2:

After removeMin():



After insert(30):



Extra credit:

```
Algorithm insert(int n):
    TreePosition p
    Let p point to the element that is the at end of the list or the one with biggest value
    While p != Tree.root:
        If p.element < p.parent.element:
            Swap p.element and p.parent.element
        p = p.parent

Algorithm removeMin():
    TreePosition r = Tree.root
    Elem el = r.element
    TreePosition p
    Let p point to the element that is the at end of the list or the one with biggest value
    Tree.root = p
    While p.hasChild():
        If p.element > p.leftChild.element:
            Swap p and p.leftChild
            p = p.leftChild
        Elif p.element > p.rightChild.element:
            Swap p and p.rightChild
            p = p.rightChild
        Else:
            Break
    Delete r
    Return el
```