COP 4530/CGS5425: Data Structures, Algorithms, and Generic Programming
Florida State University, Dept. of Comp. Sci., Fall 2006
Instructor: Breno de Medeiros
TA: Ling Toh

Assignment 5: Priority Queues

Goals:

In this assignment, you will be required to implement a template class for a priority queue object, i.e. you are required to complete the implementation for the `Priority_Queue` class. You will also be required to implement the heap sort algorithm on the priority queue to rearrange the items of a container in descending order.

Requirements for `Priority_Queue` class

A partial interface and implementation of the `Priority_Queue` class is provided in the distributed `priority_queue.h` files.

The underlying `Priority_Queue` data structure is any container type that has its `operator []` overloaded.

A `Priority_Queue` object could be declared as ,

```
Priority_Queue <vector<int> > Q1; //underlying vector stores integer values

//underlying deque stores character values, and uses the greater function
//object for comparison
Priority_Queue <deque<char>, greater<char> > Q2;

//underlyingBasicContainer stores floating-point values and uses the less
//function object for comparison
Priority_Queue <BasicContainer<float>, less<float> > Q3;
```

The *partially* distributed `priority_queue.h` file contain the following implementations:

```
template <typename T>
class LessThan
  {
  public:
    bool operator()( const T &x, const T &y ) { return x < y;}
  };

template <class C, class P = class LessThan<typename C::value_type> >
class Priority_Queue
{

public:

  typedef typename C::value_type value_type;
```

```
  void PushHeap(const value_type& value)
  {
    c.push_back(value);
    RestoreHeap(c.size()-1);
  }


  void PopHeap()
  {
    swap(c[0], c[c.size()-1]);
    c.pop_back();
    RestoreHeap(0);
  }


protected:
  C c;
  P Compare;
};
```

The *full* implementation of the `priority_queue.h` file MUST contain **at least** the following declarations:

`Priority_Queue()`: Constructor for the `Priority_Queue` object.

`Priority_Queue(const C& container)`: Constructor for the `Priority_Queue` object that initializes the container type **`c`** to **`container`**. In addition, this method should also call the **`MakeHeap`** method to ensure that **`c`** adheres to its heap property.

`void Push(const value_type& value)`: Adds item of type **`value`** to priority queue. This method should call the **`PushHeap`** method to add item to the underlying container and maintain its heap property.

`void Pop()`: Removes the item from the top of the priority queue. This method should call the `PopHeap` method to remove the item from the underlying container and maintain its heap property.

`unsigned int Size()`: Returns the size of the priority queue.

`bool Empty()`: Returns `true` if the priority queue is empty and `false` otherwise.

`value_type& Top()`: Returns the element located at the top of the priority queue.

`void Display(std::ostream&, char ofc = '\0') const`: Displays all the elements in the priority queue.

`void MakeHeap()`: Rearranges the elements within the container `c`, in-place so as to maintain the heap property of the priority queue. Must have cost at most O(N). In particular, using generic sorting algorithms such as quicksort, which has cost O(N log N), will be marked down for inefficiency.

`void PushHeap(const value_type & value)`: Adds elements to the heap.

`void PopHeap()`: Removes elements from the back of the heap.

`void RestoreHeap(int index)`: Restores the heap property of the priority queue. Your implementation MUST not exceed O(log N) comparisons to restore the heap.

`template<class C, class P>`
`std::ostream& operator << (std::ostream& os, const Priority_Queue<C,P>& )`: overloaded global **`operator <<`** for the priority queue object.

Note: You may add any supporting methods to the class definition as deemed necessary.

Requirements for `fpq.cpp` client program

The *partially* distributed **`fpq.cpp`** file requires that you complete the template-based **`HeapSort`** method that uses the heap sort algorithm to reassign the container object (2nd parameter) based on the priority queue object (1st parameter).

To compile this file, you will also need to include a copy of your implementation of `BasicContainer.h` and `BasicContainer.cpp` from Assignment 1.

Extra-credit (5 points)

You may add an additional method in the public interface declared as

```
void Erase(int index);
```

This method should allow the user to remove an element located at the position **`index`**. The resulting priority queue must maintain its heap property after a successful removal of the element. Your implementation will be graded for functionality as well as by efficiency.

Download Instructions

To download the partial/sample files, go into the directory where you wish to have the assignment files copied to, and type the command line:

```
cp -r ~cop4530/fall06/partials/proj5/ .
```

The following files have been provided for you:

1. `priority_queue.h.partial` : partial implementation

2. `fucntionTest.cpp :` general functionality test for `Priority_Queue` object.
3. `fpq.cpp:` partial implementation.

Deliverables

1. Your implementation must be entirely contained in the following files, which MUST be named,

   a. `priority_queue.h`
   b. `fpq.cpp`

2. Your program must compile on `linprog.cs.fsu.edu` or `program.cs.fsu.edu`. If you program does not compile on either machines, the grader cannot test your submission.

   To compile in program.cs.fsu.edu, use the commands:

   ```
   g++ -I. fpq.cpp

   g++ -I. functionTest.cpp
   ```

   Points will be deducted for not complying with these requirements.

For instructions on how to submit your project, consult the first assignment, which is posted online. Remember to substitute "4" for "1" in all the command and path names which have the form "projectX" or "projX