Project 4, Fake Min-Max Heaps

Due: Tuesday, May 1, 8:59pm

Objectives

The objective of this programming assignment is to familiarize you with the binary heap data structure and for you to gain some experience designing templated C++ code.

Background

The conceit for this project is that you've been asked by your tech-ignorant manager to implement a min-max heap. This data structure allows you to maintain a collections of items, where you can:

- insert an item
- delete the smallest item
- delete the largest item

all in $O(\log n)$ time. Real min-max heaps (e.g., Project 3 from the Fall 2013 CMSC 341), are quite difficult to implement, and chock-full of strange special cases (q.v., Case 4). This is not something you want to be told to implement from scratch!

So, you concoct a plan to implement a *fake* min-max heap. Your brilliant, lazy idea is to just store every item in two heaps: a min-heap an a max-heap. Presumably insertion would still be in $O(\log n)$ time, since you just insert twice. You could delete the smallest item from the min-heap or the largest item from the max-heap in $O(\log n)$ time. You just have to store everything twice, but memory is cheap, right? There's just one small problem: when you delete the smallest item from the min-heap, its "twin" in the max-heap must also be removed, and vice versa. Removing anything but the smallest in a min-heap, or the largest in a max-heap, is not easy: heaps do not support search. You'd have to link the corresponding values in the two heaps somehow so when one it is removed from one of the heaps, it can be found int the other one and removed quickly from there, also.

Since your boss is not exactly a genius in C++, you figure you can probably get away with this ruse --- you just can't be too obvious about it. For example, you can't have two separately-defined classes, one for a min-heap and one for a max-heap. Also, having `if` statements that check a flag and then run the code for a min-heap versus a max-heap would be way too obvious. You plan to make a templated class that isn't specifically a min-heap or max-heap by design, but instead by construction, and you plan to instantiate it twice inside your min-max heap, in one case acting as a min-heap, and in the other as a max-heap.

So, to sum it up: you are going to create a binary heap class that can be configured at construction time to act like either a min-heap or max-heap ("if it quacks like a duck..."). Then, you are going to create a fake min-max heap class, which internally implements min-max behavior by just maintaining two internal heap instances: a min-heap and a max-heap.

Oh, and by the way, your boss wants you to make the entire min-max tree templated, so that users can store any type of item. Sound fun yet?

Assignment

Note: Running time is one of the most important considerations in the implementation of a data structure. Programs that produce the desired output but exceed the required running times are considered wrong implementations and will receive substantial deductions during grading.

Your assignment is to design and implement a templated C++ class `MinMaxHeap` that supports insert, deleteMin and deleteMax in $O(\log n)$ time. You are given general strategies and some specific requirements, but most of the class design and implementation is up to you.

The general requirements are:

1. `MinMaxHeap` must support insert, deleteMin and deleteMax in $O(\log n)$ time. The restriction here is that searching for a key in a heap takes linear time. So, you can't delete the root of one heap and look for its twin in the other heap. That will take too long.
2. You may not use STL container classes. (Strings are OK.)
3. Your templated min-max heap must work with any class that supports assignment and comparison. (See test programs.)
4. You must have one implementation of a heap class that can be used as a min-heap and a max-heap. The code must use a function pointer to compare items. The behavior of this comparison function is what determines whether the heap is a min-heap or a max-heap.

General Strategies

1. You will need a wrapper class `MinMaxHeap` that has the functions used by the main program and a separate `Heap` class that does the "bubbling up" and "trickling down" to maintain a heap. A `MinMaxHeap` object will have two `Heap` objects --- one for the min-heap and one for the max-heap.
2. Each heap has to "know" that it is part of a min-max heap. It just doesn't "know" if it is the min-heap or the max-heap.
3. Each item in either the min-heap or the max-heap has to keep track of its "twin" in the other heap. The easiest way to do this is to store the index of the twin in the other heap. For example, consider the following output from the `dump()` function of a small min-max heap. The line

   ```
   Heap[2] = (6,5)
   ```

   means that key value 6 is stored in index 2 of the min-heap. The second value 5 indicates that key 6 is in index 5 of the max-heap. This confirmed by the output:

   ```
   Heap[5] = (6,2)
   ```

   in the max-heap. That is, the 5th item does indeed hold the key value 6. It also knows that in the min-heap, key value 6 is stored in index 2. You can confirm for each item in each heap that the index of the item's twin in the other heap is recorded properly. You can also confirm that the min-heap is heap ordered correctly with the smallest item in the root and the max-heap is heap ordered correctly with the largest itme int the root.

   ```
   ... MinMaxHeap::dump() ...

   ------------Min Heap------------
   size = 9, capacity = 25
   Heap[1] = (2,6)
   Heap[2] = (6,5)
   Heap[3] = (5,8)
   Heap[4] = (10,3)
   Heap[5] = (9,9)
   Heap[6] = (7,7)
   ```

```
Heap[7] = (16,1)
Heap[8] = (12,2)
Heap[9] = (11,4)

------------Max Heap------------
size = 9, capacity = 25
Heap[1] = (16,7)
Heap[2] = (12,8)
Heap[3] = (10,4)
Heap[4] = (11,9)
Heap[5] = (6,2)
Heap[6] = (2,1)
Heap[7] = (7,6)
Heap[8] = (5,3)
Heap[9] = (9,5)
--------------------------------
```

4. Every time an item is moved in one of the heaps, say in the bubbling up process after insertion, you have to update the item's twin so the twin "knows" the item's new position. (I.e., after moving an item, a dump should still be consistent as described above.) Look at the rest of the output from first test program for examples: test1.txt.
5. Each heap needs access to the internals of the other heap. So, in the `MinMaxHeap` constructor you have to create two heaps and then give each `Heap` a pointer to the other `Heap`.
6. You'll have to make templated stand-alone functions (i.e., not member functions) that compare two heap items and says which one is bigger or which one is smaller. You can assume that the type/class used to instantiate the templates supports the six comparison operators: <, <=, ==, !=, >= and >. The `MinMaxHeap` constructor should give a pointer to one of these comparison functions to the appropriate `Heap` object. That is what determines if the `Heap` is a max heap or a min heap.
7. The `MinMaxHeap` class has to coordinate insertion and deletion between the two heaps. For example, suppose that during insertion, you insert in the min-heap first and then insert in the max-heap. When you insert in the min-heap, it does not yet have a twin in the max-heap. After inserting in the min-heap, the `MinMaxHeap` insertion function has to know the location of the item in the array for the min-heap. This index has to be recorded with the item's twin in the max-heap. Then, the opposite has to happen: the newly inserted item's location in the max-heap must be recorded with its twin in the min-heap.
   A similar coordination must also take place for deletion.
8. Your heaps have to support deleting an item given its index. (I.e., you have to be able to delete nodes other than the root.) The removed item should be replaced by the last item in the heap, as we've seen when the root is removed. However, when we fix the heap, we may have to bubble up or trickle down. (Re-read the last sentence.)

Specific Requirements

1. Your class definitions must be in a single file called `MinMaxHeap.h` (case sensitive). Client programs that use the `MinMaxHeap` class should be able to do so simply using:

   ```
   #include "MinMaxHeap.h"
   ```

   All of the template implementations should be in a file called `MinMaxHeap.cpp` (case sensitive).
2. The `MinMaxHeap` class must have a constructor with the signature:

   ```
   template <typename T>
   MinMaxHeap<T>::MinMaxHeap(int capacity)  ;
   ```

The constructor must create a `MinMaxHeap` object capable of holding `capacity` items. The object created by the constructor must be able to accept insertion and deletion calls right after the constructor is done.

3. The `MinMaxHeap` class must have a copy constructor with the signature:

   ```
   template <typename T>
   MinMaxHeap<T>::MinMaxHeap(const MinMaxHeap<T>& other)  ;
   ```

   The constructor must create a copy of the `MinMaxHeap` object given in the parameter. The copied object must have its own allocated memory.

4. The `MinMaxHeap` class must have a destructor with the signature:

   ```
   template <typename T>
   MinMaxHeap<T>::~MinMaxHeap()  ;
   ```

   The destructor must deallocate any dynamically allocated memory.

5. The `MinMaxHeap` class must have an overloaded assignment operator with the signature:

   ```
   template <typename T>
   const MinMaxHeap<T>& MinMaxHeap<T>::operator=(const MinMaxHeap<T>& rhs)  ;
   ```

   The assignment operator must deallocate memory in the host object and copy the `rhs` `MinMaxHeap` into the host.

6. Your compiled code must run without memory leaks.

7. The `MinMaxHeap` class must have a member function called `size` that returns the number of items in the min-max heap. The `size()` function must have the signature:

   ```
   template <typename T>
   int MinMaxHeap<T>::size() ;
   ```

   The `size()` function must run constant time to receive full credit.

8. The `MinMaxHeap` class must have a member function called `insert` that adds the item given in the parameter to the min-max heap. The `insert()` function must have the signature:

   ```
   template <typename T>
   void MinMaxHeap<T>::insert(const T& data) ;
   ```

   If min-max heap cannot hold anymore items (i.e., it has reached capacity), then `insert()` should throw a `out_of_range` exception.
   The `insert()` function must run in $O(\log n)$ time to receive full credit.

9. The `MinMaxHeap` class must have a member function called `deleteMin` that removes the smallest item in the min-max heap and returns the key value of the deleted item. The `deleteMin()` function must have the signature:

   ```
   template <typename T>
   T MinMaxHeap<T>::deleteMin() ;
   ```

   If the heap is empty, the `deleteMin()` function should throw a `out_of_range` exception.
   The `deleteMin()` function must run in $O(\log n)$ time to receive full credit.

10. The `MinMaxHeap` class must have a member function called `deleteMax` that removes the largest item in the min-max heap and returns the key value of the deleted item. The `deleteMax()` function must have the signature:

```
    template <typename T>
    T MinMaxHeap<T>::deleteMax() ;
```

If the heap is empty, the `deleteMax()` function should throw a `out_of_range` exception.
The `deleteMax()` function must run in $O(\log n)$ time to receive full credit.

11. The `MinMaxHeap` class must have a member function called `dump` that prints out the contents of the min-max heap including the positions of each key in the min-heap and the max-heap along with recorded position of its "twin". See formatting examples in the sample output below. The `dump()` function must have the signature:

```
    template <typename T>
    void MinMaxHeap<T>::dump() ;
```

12. The `MinMaxHeap` class must have two member functions used for grading called `locateMin` and `locateMax`. These functions should have the prototype:

```
    template <typename T>
    void MinMaxHeap<T>::locateMin(int pos, T& data, int& index) ;

    template <typename T>
    void MinMaxHeap<T>::locateMax(int pos, T& data, int& index) ;
```

The function `locateMin` has two reference parameters for "return values". The `data` parameter should hold the value of the item in position `pos` of the min-heap. The `index` parameter should hold the location of the item's "twin" in the other heap.
The `locateMax` function have the analogous requirements.

Your code must run without segmentation fault and without memory leaks. For grading purposes, memory leaks are considered as bad as segmentation faults. This is because many segmentation faults are cause by poorly written destructors. A program with an empty destructor might avoid some segmentation faults but will leak memory horribly. Thus, not implementing a destructor or not deleting unused memory must incur a penalty that is equivalent to a segmentation fault.

Test Programs

Here are sample driver programs to test your implementation. *Passing these tests do not mean you will receive 100% on your project. It does not guarantee that you will pass tests used in grading. You should make additional tests of your own!*

- Test `MinMaxHeap` with `int` data. A `sanityCheck()` function is provided. Compile using:

```
    g++ test1.cpp
```

  Driver program: test1.cpp
  Sample Output: test1.txt

- Test `MinMaxHeap` with the `string` data. A `sanityCheck()` function is provided. Compile using:

```
    g++ test2.cpp
```

  Driver program: test2.cpp
  Sample output: test2.txt

- Test `MinMaxHeap` with the `NoCopyString` class. A `sanityCheck()` function is provided. Compile using:

```
    g++ test3.cpp NoCopyString.cpp
```

  Driver program: test3.cpp,
  `NoCopyString` class: NoCopyString.h, NoCopyString.cpp,

Sample output: test3.txt

- Bigger test of `MinMaxHeap` with `int` data. A `sanityCheck()` function is provided. The keys in the `MinMaxHeap` are also checked against an equivalent STL `multiset`. Only `deleteMin()` is checked because STL `multiset::erase()` doesn't want to erase reverse iterators. Compile using:

  ```
  g++ test4.cpp
  ```

  Driver program: test4.cpp
  Sample output: test4.txt

- Same as previous test, except it checks `deleteMax()` instead of `deleteMin()`. Compile using:

  ```
  g++ test5.cpp
  ```

  Driver program: test5.cpp
  Sample output: test5.txt

- Test copy constructor, destructor and assignment operator Should test this with valgrind
  Driver program: test6.cpp
  Sample output: test6.txt

Implementation Notes

Here we list some recommendations and point out some traps and pitfalls.

- Review C++ template syntax before starting. Remember that you should

  ```
  #include "MinMaxHeap.cpp"
  ```

  at the bottom of the header file. Also, the .cpp file must also be guarded with `#ifndef`.

- Just start at index 1 for the heaps. The "wasted" storage in the 0-th index could be useful (hint, hint).

- Beware of off-by-one errors since the 0-th item of the heap array is not used. A heap with capacity for 100 items needs 101 entries indexed 0 thru 100.

- After you delete an item from a heap, the "twin" object has an invalid index for its twin.

- There is a special case for deleting the last item of a heap. This is because we would usually swap in the last item to replace the item that is deleted. However, if are deleting the last item, then we would be swapping into an invalid location.

- When a `MinMaxHeap` is copied, you must make sure that in the new `MinMaxHeap`, the new min-heap and the new max-heap are referencing each other and not the max-heap and min-heap of the original `MinMaxHeap`. The same situation applies for assignment.

What to Submit

You must submit the following files to the `proj4` directory.

- `MinMaxHeap.h`
- `MinMaxHeap.cpp`
- `Driver.cpp`

The `Driver.cpp` program should include tests showing the parts of your project that work correctly.

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using this Unix command command.

```
    cp MinMaxHeap.h MinMaxHeap.cpp Driver.cpp ~/cs341proj/proj4/
```