CS 132 - Intro to Computer Science II - Spring 2005
WEEK #12 LAB EXERCISE and Homework #9

Week #12 Lab Exercise due: Wednesday, April 13th, END of lab
HW #9 due: Wednesday, April 20th, 1:00 pm

Purpose: to implement heap and heapsort; to become more familiar with an array-based implementation of a complete binary tree

WEEK #12 LAB EXERCISE

On the public course web page, you will find a file **heap.h**, the .h file for a **heap** template class. This heap template class includes, as a private data field, an instance of a **complete_tree**, a template class whose .h and .template files are also available from the course web page. **complete_tree** is an array-based implementation of a complete binary tree.

Answer the following questions on paper; after coming up with your initial answers, you may discuss them with another student before getting your answers checked, if you wish. When you are ready, put your name on the "Next:" list on the board so that your answers can be checked.

1. Consider the implementation of the **complete_tree** template class provided along with this assignment.

(a) List the names of the **accessor/observer** methods provided by **complete_tree**.

_____

_____

_____

(b) List the names of the **modifier** methods provided by **complete_tree**.

_____

_____

_____

(c) List the **types AND names** of the private **data fields** used within **complete_tree**.

_____

_____

(d) What private methods are declared within **complete_tree**? List their names.

_____

_____

(e) Give the formula that this **complete_tree** implementation uses for determining the index of the **parent** of the current node (assuming the index of the current node is stored within **current_index**).

_____

(f) Give the formula that this **complete_tree** implementation uses for determining the index of the **left child** of the current node (assuming the index of the current node is stored within **current_index**).

_____

(g) Give the formula that this **complete_tree** implementation uses for determining the index of the **right child** of the current node (assuming the index of the current node is stored within **current_index**).

---

2. Now consider the provided file **heap.h**, the declaration of the **heap** template class that you will be implementing for part of HW #9 (as you'll see below).

(a) List the names of the **accessor/observer** methods to be provided by **heap**.

(b) List the names of the **modifier** methods to be provided by **heap**.

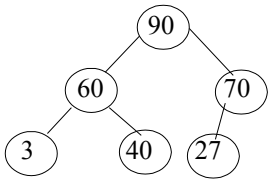
(c) List the **types AND names** of the private **data fields** that will be part of each **heap** instance.

(d) What private methods are to be declared within **heap**? List their names.

---

3. In lecture, we discussed the basic algorithms for inserting into and removing from a heap. For this problem, you are going to show that you understand these basic algorithms.

Consider the following heap:



(a) Consider what should happen in order for **100** to be added to this heap.

On the back of one of this handout's pages or on a separate sheet of paper, write **3(a)** and then **draw** the stages that this heap goes through during the process of adding **100** to this heap. (You must give a snapshot of each change to this heap that occurs in the process of adding 100 to it. The final picture will how the heap "ends up".)

(b) ASSUME that **(a)** has been done. Now we want to add **75** to the heap.

On the back of one of this lab assignment's pages or on a separate sheet of paper, write **3(b)** and then **draw** the stages that this heap goes through during the process of adding **75** to this heap. Again, you must give a "snapshot" of each change to the heap that occurs.

(c) ASSUME that **(a)** and **(b)** have been done. Now add **14** to the heap, writing **3(c)** and then **drawing** the stages that the heap goes through during this process

(d) ASSUME that **(a) - (c)** have been done. Now **REMOVE** the the maximum value in this heap, writing **3(d)** and then **drawing** the stages that the heap goes through during this process.

What value is returned by this **remove_max** operation? _____

(e) ASSUME that **(a) - (d)** have been done. Now, remove the maximum value in this heap again, writing **3(e)** and then **drawing** the stages that the heap goes through during this process.

What value is returned by this **remove_max** operation? _____

(f) ASSUME that **(a) - (e)** have been done. Now, remove the maximum value in this heap one more time, writing **3 (f)** and then **drawing** the stages that the heap goes through during this process.

What value is returned by this **remove_max** operation? _____

When you are happy with all of your answers, put your name on the "Next:" list to get your work checked over. Your work must be completed and checked over before the end of lab.

HOMEWORK #9:

1. You have already looked a bit at the provided file **heap.h** (during the lab exercise). You should see that there is a file **test_heap.cpp**, a partial tester for the heap template class. Make copies of **heap.h**, **test_heap.cpp**, **complete_tree.h**, and **complete_tree.template** to a directory on cs-server.

Implement **heap.template** for this **heap.h**. When you are done, compile **test_heap.cpp** and you can partially test your heap implementation.

When you are happy with your **heap.template**, run:

```
test_heap > 132hw09_1_out
```

...and submit your **heap.template** and **132hw09_1_out**.

2. What can you do with a heap? Well, use it to implement **heapsort**, for one thing!

Write a **template** function **heapsort** (in file **heapsort.template**) that expects an array of items and its size as its parameters; it modifies the passed array as a result, using the **heapsort** algorithm described in lecture (and in the course text) to modify the passed array so that its contents are in ascending sorted order. (As a template function, the full function opening comment block --- complete with a suitable collection of examples! --- is expected, of course.)

(remember the basic pseudocode for heapsort of an array's elements ---
add all of the elements of the array into a heap,
while the heap is not empty, remove the heap's root, reheap, and add it to the array in the "next" spot)

Of course, you'll have to **adapt** the basic pseudocode based on the operations of this particular **heap** class. And, your function **must** use the provided **heap.h** and your **heap.template** from Problem #1.

Write a **test_heapsort.cpp** that tests your template function (including testing all of your examples, following the expected course testing standards...), and when you are happy with your code run:

```
test_heapsort > 132hw09_2_out
```

...and submit your resulting **heapsort.template**, **test_heapsort.cpp**, and **132hw09_out**.

(Note: even if your **heap.template** is not up-to-snuff, you will receive substantial partial **credit** for problem #2 if you turn in a version of heapsort.template and test_heapsort.cpp that DO run when I test them with my version of heapsort.template; that is, if you are comfortable with heap.h's contents, you should still be able to write heapsort and its tester, even if you cannot actually execute them yet...)

And, when you are satisfied with all of the above, submit them using **~st10/132submit** on cs-server.

heap.template
132hw09_1_out.
heapsort.template
test_heapsort.cpp
132hw09_2_out