## Homework 2: The Heap Data Structure and Its Variants

**Objective** In this programming homework you are being asked to implement the following data structures:

1. A priority queue based on Min-Heap
2. *Leonardo Heap*, a variant of the Max-Heap

in the `C++` language. You are asked and to create simple experiments to demonstrate the correctness of your implementation and access the performance of these structures. For Leonardo Heap, we will use it to implement a sorting algorithm named *Smooth Sort*.

### Introduction

As discussed in class, the Max-Heap data structure uses of a tree structure (a *complete binary tree* that are stored in an array) to support sorting. As a consequence, the sorting method (*Heap Sort*) developed from it has a good worst case performance. The worst case complexity is found to be $\Theta(n \lg n)$. In addition, the Heap structure can be easily modified for building a priority queue.

In this homework, you are asked to implement a Min-Heap based priority queue and develop a sorting method (*Smooth Sort*) based on a variant of the Heap structure (*Leonardo Heap*). It was shown that the worst case performance of Smooth sort is the same as Heap Sort but *Smooth Sort* tends to perform better when the input is almost sorted. Some analysis suggests that, for some classes of input that almost sorted, the algorithm runs in $O(n)$ time.

In this homework, all data are records with two fields: key (of type int) and value (of type int). We will use the notation `[key value]` to represent a data item. For example, `[5 20]` represents the data item with key = `5` and value = `20`. We will denote a priority queue by $PQ$. You may use the vector class in STL in this assignment.

### I. *A Min-Heap Based Priority Queue*

Develop a priority queue data structure via a Min-Heap. Verify, via a collection of test cases, the functionalities of each of the following operations (1 to 4):

**1.** Insert: insert a data item, say $d$, to $PQ$

**2.** All-Minimum: Return all the data items in $PQ$ with the minimum key (highest priority)

**3.** Decrease-key: decrease the value of an data item's key to a new value

**4.** Extract-all-min: Remove and return all the data items with the smallest key value

As as reference, you may consult Cormen, Chapter 6 Section 5 as a guide for your implementation. Note that some operations are not being defined in the same way as Cormen's text.

### Test Cases

To test that these operations are functioning properly, develop the following test cases $1 - 5$. All test case start with an empty queue. Let $PQ$ denotes the priority queue implemented:

**1.**

1. Insert the following data items to $PQ$ in the given order:

   `[100 1] [99 2] ... [1 100]`

2. Apply Extra-all-min operation to $PQ$ and print the results to the screen
3. Insert the data `[1 0]` to $PQ$
4. Apply Extra-all-min operation to $PQ$ and print the results to the screen

**2.**

1. Insert the following data items to $PQ$ in the given order:

   `[1000 1] [999 2] ... [1 1000]`

2. Apply All-Minimum operation to $PQ$ and print the results to the screen
3. Insert the following data items to $PQ$ in the given order:

   `[1 1] [1 2] ... [1 30]`

4. Apply Extra-all-min operation to $PQ$ and print the results to the screen
5. Apply All-Minimum operation to $PQ$ and print the results to the screen

**3.**

1. Insert 500 randomly generated data items to $PQ$. For each of these data item $[i \quad j]$, $10 \leq i, j \leq 1000$.
2. Apply Extra-all-min operation to $PQ$ 5 times, and print the results to the screen each time right after the Extra-all-min operation is completed
3. Randomly select a data item from the $PQ$. Decrease the key value to 5.
4. Apply the All-Minimum operation to $PQ$ and print the results to the screen

**4.**

1. Insert 500 randomly generated data items to $PQ$. For each of these data item $[i \quad j]$, $10 \leq i, j \leq 1000$.
2. Perform 10 Decrease-key operations to $PQ$. When a Decrease-key operation is performed, the data item should be chosen randomly from the current $PQ$ and the key value should be decreased to a randomly selected value that lies between 0 to 9.
3. Perform 10 Extra-min operations to $PQ$ and print the results to the screen each time such an operation is performed.

**5.**

1. Perform 100 randomly generated *operations* to $PQ$. The random generated operations should include all four operations stated. In addition, when a Decrease key operation is chosen, the data item should be chosen randomly from the current $PQ$ and the key value should be decreased that is less than all the existing key value.

2. Note that your implementation should be able to handle underflow situations and complete the entire test.
3. For every 20 operations performed, print the underlying array/vector of data items to the screen. That is, you should show the status of the underlying array/vector five times in this test case.

**II.** ***Leonardo Heap for Smooth Sorting***

Develop the Leonardo Heap data structure and use it to implement the Smooth Sort algorithm. Create experiments to assess its effectiveness.

**Test Cases** To verify the correctness of your implementation and assess its effectiveness empirically, implement the following test cases $6 - 10$. Let $A$ denotes the list of numbers needs to be sorted. You may represent $A$ via a STL vector.

**6.** A general test:

1. Initially, set $A = [1000, 999, 998, ..., 1]$
2. Apply Smooth Sort to $A$.
3. Print the following data to the screen:

```
The total number of comparisons among elements in A performed = ...
A shortened listing of element of A :
...
```

4. The shortened listing is to print the leading element for every 20 elements in $A$. In this case, we should expect to see

```
[1, 21, 41, 61, ... , 981]
```

on screen.

**7.** For an almost sorted list:

1. Initially generated a almost sorted list by first setting $A$ to be:

$$A = [1, 2, 3, ..., 2000]$$

2. Randomly select 20 distinct numbers that lies between 1 to 2000, say $i_1, \ldots, i_{20}$. Swap $A[i_1]$ and $A[i_2]$, $A[i_3]$ and $A[i_4]$, , ..., $A[i_{19}]$ and $A[i_{20}]$ (Here, I assume the array index starts with 1).
3. Apply Smooth Sort to $A$.
4. Print the following data to the screen:

```
The total number of comparisons among elements in A performed = ...
A shortened listing of element of A:
...
```

5. The shortened listing is to print the leading element for every 20 elements in $A$ on screen.

**7.** For another almost sorted list

1. Initially generated a almost sorted list by first setting $A$ to be:

$$A = [1, 2, 3, ..., 2000]$$

2. Randomly select 60 distinct numbers that lies between 1 to 1000, say $i_1, \ldots, i_{60}$. Permute $A[i_1]$, $A[i_2]$ and $A[i_3]$, Permute $A[i_4]$, $A[i_5]$ and $A[i_6]$, ..., Permute $A[i_{58}]$, $A[i_{59}]$ and $A[i_{60}]$ (Here, I assume the array index starts with 1).
3. Apply Smooth Sort to $A$.
4. Print the following data to the screen:

```
The total number of comparisons among elements in A performed = ...
A shortened listing of element of A:
...
```

5. The shortened listing is to print the leading element for every 20 elements in A on screen.

**9.** For list whose length is a Leonardo number

1. Initially set $A$ to be a list of length 3193, a Leonardo number.
2. Fill the entries of $A$ randomly with numbers selected from 1 to 10000. The numbers may not be all distinct.
3. Apply Smooth Sort to $A$.
4. Print the following data to the screen:

```
The total number of comparisons among elements in A performed = ...
A shortened listing of element of A:
...
```

5. The shortened listing is to print the leading element for every 20 elements in $A$ on screen.

**10.** For list whose length is the sum of two Leonardo numbers

1. Initially set $A$ to be a list of length 5166 the sum of two Leonardo numbers.
2. Fill the entries of $A$ randomly with numbers selected from 1 to 10000. The numbers may not be all distinct.
3. Apply Smooth Sort to $A$.
4. Print the following data to the screen:

```
The total number of comparisons among elements in A performed = ...
A shortened listing of element of A:
...
```

5. The shortened listing is to print the leading element for every 20 elements in $A$ on screen.

**Due Date** Upload your submission on or before 10:00PM (EST Time), 10/26/2016. **24 hour extension rules apply** (See Syllabus for the details). For those who met this criteria, and if the final submissions received after 10:00PM (EST Time), 10/27/2016, the penalty is 25 percent off per day late.

Your submission will contain a collection of `C++` header files (`.h` files), implementation files (.cpp files) and the main program (named as `main.cpp`). They are expected to have adequate comments.

In addition, you will need to submit two text files:

*i*. `readme.txt`:

it should contain a concise description of how the two data structures (i.e. Heap for priority queues and Leonardo Heap) work. Include a concise description regarding your implementation of all the essential operations Acknowledge the reference(s) you use, if any, in this file.

State the expected outcome for each of the required test case. You may include any other relevant information in the readme.txt file to illustrate, explain and support your ideas regarding the implementation submitted.

*ii*. `output.txt`:

it should contain a sample execution result in the ubuntu/g++ environment. This file can be obtained by

```
ubuntu>g++ *.cpp -o hw2
ubuntu>./hw2 > output.txt
```

It should be in a single zip file. It should be named via the following convention:

```
<SU-EMAIL>-<FIRST-Name>-HW2.zip
```

That is, if my SU email address is `abc111@syr.edu` and my first name is Andrew, then I should name my submission as

```
abc111-andrew-HW2.zip
```

Other relevant information regarding the submission process will be posted by our grader within our blackboard site or via our Piazza forum.