Project 2

Max Heap Implementation

CS 241 Winter 2017, Dr. Hao Ji

Taylor Thurlow

tjthurlow@cpp.edu

13 February 2017

**Section 1: Project description**

The program is an implementation of the heap (specifically, the max-heap). Using general types, it supports any comparable data type. The program will provide a simple command-line based interface which gives the user the option to feed random sets of data, or to feed a single set of known integers. It forms a max-heap using the input data in two ways - an inefficient method using a series of insertions, and a different, more optimal method. The program displays relevant information regarding how many “swaps” the algorithms took to fill the max-heap.

**Section 2: Project specification**

In this project, you are going to build a max-heap. You will use an array to implement the heap. Your program should:

- Allow the user to select one of the following two choices (Note that your program needs to implement both choices):
    - (1) test your program with 100 randomly generated integers (no duplicates, positive numbers with proper range)
    - (2) test your program with the following 100 fixed values from 1, 2, 3, ..., and 100.
- Implement both methods of building a max heap
    - Using sequential insertions
    - Using the optimal method
- For both methods, you need to keep track of how many swaps (swapping parent and child) are required to build a heap.
- For choice (1), you need to generate 20 sets of randomly generated integers; compute, print and document (in your project report) the average number of swaps for both methods. Your program should output the average number of swaps for both methods (an average over 20 sets).

- For choice (2), your program should output the first 10 integers in your array and the number of swaps for both methods. Then perform 10 removals on the heap and output the first 10 integers.

**Section 3: Testing methodology**

The majority of the work in implementing a heap is in the reheap function. Though it may be slightly complicated at first, the only legitimate edge case is the first data item being added to the array. In this sense, it's easy to tell if you've broken the first insertion, because everything else is wrong as well. An IDE and debugger were invaluable in this step, stepping through data insertion is a complicated process, and visualizing the data as it moves through the reheap function was extremely valuable. Removing is even easier, because the only removable node is the root, and the majority of the work in removing the node is the reheap itself.

**Section 4: Lessons learned**

Conceptually, the max-heap is not a difficult-to-understand data structure. Because of this, implementation was relatively straightforward. However, due to the complexity of the reheap function, debugging was not. It's been awhile since I had to use a debugger as extensively as I did for this project - more than half of my time was spent testing and debugging code that I had already written.

**Section 5: Analysis of output**

In the case of choice 1, we're working with 20 data sets of 100 randomly generated numbers from 1 to 1000. There are no duplicate entries within each set. The program counts the number of swaps required to obtain a valid max-heap. On average, the smarter method takes around 68 swaps, and the slower method takes around 111. This is roughly a

63% increase in efficiency!  If that wasn't already impressive, this is really the average case - all of our values are randomly generated. With choice 2, we have a known set of 100 integers, in ascending order. Because of the order of the data, the slow method suffers. The program has determined that the smarter method takes exactly 96 swaps to achieve a valid max-heap, and the slower method takes a whopping 480 swaps. Perhaps not coincidentally, this is exactly 5 times as many swaps as the smart method. With the arrangement of the data set, the smarter method really shines and becomes the clear winner between the two.

This should be obvious though. The smart method adds all of the data into the heap, regardless of the values. Obviously this isn't yet a max-heap, so it performs a reheap on the last non-leaf node (the parent of the last leaf node). The reheap function will do everything needed to form a valid max-heap. By contrast, the slow method is essentially the same process, but for one data item at a time. Every time a data element is added, a reheap is performed. The result is a valid max-heap after every addition, but because we know that there is more data to be added, it's a waste of swaps. Lots of the swaps that happen for the first data items are just a waste of time, because the later data items will probably affect what swaps need to take place. By waiting, a lot of time and swaps are saved.

```
$> java Main
Please select a mode:
1) 20 sets of 100 randomly generated integers
2) Fixed integer values 1 to 100
Enter choice: 1

 =====

Average number of smart (fewer reheaps) swaps: 69
Average number of dumb (series of insertions) swaps: 112



$> java Main
Please select a mode:
1) 20 sets of 100 randomly generated integers
2) Fixed integer values 1 to 100
Enter choice: 2

 =====

Number of smart (fewer reheaps) swaps: 96

First 10 entries: 100 95 99 79 94 98 63 71 78 87
After 10 removals: 90 89 63 79 88 55 62 71 78 87

 =====

Number of dumb (series of insertions) swaps: 480

First 10 entries: 100 94 99 77 93 98 61 68 76 84
After 10 removals: 90 89 62 77 88 53 61 68 76 84
```