COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 4

Dr. Joe Anderson

Due: 25 February

1 Objectives

In this lab you will focus on the following objectives:

1. Use the heap data structure to implement a priority queue.
2. Review implementations of heaps and binary trees in `c++`.

2 Tasks

1. Put your code in a folder to be zipped and turned in at the end.
2. Create a template class called `HeapQ` which will model a heap-based priority queue.
   (a) Demonstrate the usage of your queue with a set of people, represented by only their name as a `string`.
   (b) The class should use an array of `struct` objects to hold the entire heap, so you will also need private variables to track the heap size and current size of the array. For example, you can use

```
template <class T>
struct HeapObj {
  T data;
  int key;
};
```

   and in the `HeapQ` class, maintain a dynamic array, `HeapObj<T>* arr`.
   (c) Include the following `public` Heap operations, as discussed during lecture (and see the appendix):
      i. `Extract-Max`, but you may alias with something like `Dequeue` if you like.
      ii. `Peek`, to see what data is the current top of the queue.
      iii. `Insert(T obj, int priority)` to insert a new item into the heap.
      iv. `Print` to display the current contents of the heap, along with key data, for testing/visualization purposes. You may assume in this method that `class T` has an overloaded `<<` operator.
   (d) Your heap should have a `private` method for `Increase-Key` (private because outside code will not know the array positions), which increases the key of a node at a particular location.
   (e) Your heap should also have private methods to expand the internal array if needed by the user.
   (f) For simplicity, you may assume that all priorities will be positive numbers.

(g) Consult with the instructor if you feel that you need or want to add extra functionality to the class.

3. Write a `main` function in a separate file to show tests of all aspects of your priority queue.
4. Include a `Makefile` to build your code.
5. Include a `README` file to document your code, any interesting design choices you made, and **answer the following questions completely and thoroughly**:
   (a) Summarize your approach to the problem, and how your code addresses the abstractions needed.
   (b) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the input size? Be sure to vary the parameters enough to use the observations to answer the next questions!
   (c) How does the absolute timing of different algorithms scale with the input? Use the data collected to rectify this with the theoretical time complexity, e.g. what non-asymptotic function of n mostly closely matches the timings that you observe as n grows?
   (d) Describe 2-3 different larger application areas where a priority queue would be helpful. Explain and give some justification why it would be a better choice than other data structures.
   (e) How could the code be improved in terms of usability, efficiency, and robustness?

3 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment. Upload your project files to MyClasses in a single `.zip` file. Also turn in files reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

4 Bonus

(10 pts) Implement an array randomization algorithm to take an n element array and permute it by inserting each element into a priority queue with a random priority between 1 and n^3. The algorithm should then successively call EXTRACT-MAX to get the elements in a random order. Come up with some metrics to compare this to simply randomly swapping different elements for some fixed number of iterations (e.g. simply swapping two random elements, 1000 times). What can you say/determine about the probability that some of the priorities are duplicated, when randomly assigned from 1 to n^3? Try computing an upper bound on the probability of getting any collisions.

A Algorithm Reference

The following are implementations that use a MAX-HEAP as the underlying structure. Keep in mind that for your lab, the elements of A will be structures, so you will need to use `.key` and `.data` to interface with the user's data versus the heap keys.

Algorithm 1 Insert(A, x)

1: // This function adds a new heap item and calls Increase-Key to put in the correct position
2: $A.heap\_size = A.heap\_size + 1$
3: $A[A.heap\_size] = -\infty$ // Or some value less than every integer in the heap
4: Increase-Key$(A, A.heap_size, k)$

Algorithm 2 Increase-Key(A, i, k)

```
1: // Increases the priority of the item at location i in the heap, then
2: // does a "reverse" Max-Heapify to move it upward accordingly
3: if k < A[i] then
4:     error "new key smaller than old key!"
5:     return
6: end if
7: A[i] = k
8: while i > 1 and A[Parent(i)] < A[i] do
9:     Swap(A[i], A[Parent(i)])
10:    i = Parent(i)
11: end while
```