CS 104 (Fall 2013) — Assignment 7
Due: 11/05/2013, 11:59am

BitBucket directory name for this homework (case sensitive): HW7

(1) Review Chapters 13, 14, 15, 16, and 17 from the textbook. (Yes, we covered a lot of material.)

(2) In class, we saw how to implement a Priority Queue using a Heap, which was built on top of a complete binary tree. The choice of making the tree binary was kind of arbitrary, and here, you will explore some of the implications of making it d-ary for other values of d. So for the purpose of this question, all trees we talk about will be complete d-ary.

(a) Precisely state the Heap Property for complete d-ary trees.

(b) One of the advantages of using complete binary trees was that it made it easy to store the tree in an array and do simple calculations to find the index of the parent and the children of a given node i. Give formulas for calculating the parent and the children when the tree is d-ary. (Specify clearly whether the root of your tree is in array index 0 or 1; you can make it work either way, but you should let us know which one you do.)

(c) As we saw in class, the advantage of a complete binary tree was that the height is $\log_2(n)$, which means that insertions and deletions run in $O(\log n)$. What is the height of a complete d-ary tree with n nodes? Show us the calculations that arrive at your answer.

(d) Based on your calculation of tree heights, analyze the running times of the natural generalizations of `trickleUp` and `trickleDown` to d-ary tree in big-$O/\Omega/\Theta$ notation. Use this analysis to analyze the running times for `add`, `remove`, and `peek`.

Your running time will be a function of two variables, d and n. There is nothing magical about big-O notation only working for one variable — you can use it with functions of many variables as well.

(e) Based on your running time calculations, which operations get faster, and which get slower, as you increase d?

(3) Implement a template Priority Queue as a Heap based on a complete d-ary tree. Your implementation should allow the user to choose whether the root of the tree contains the largest or smallest element. Your implementation should provide at least the following functions:

```
T & peek () const;  /* returns the highest-priority element in the heap.
                       Throws an exception if the heap is empty. */
void remove (); /* removes the highest-priority element from the heap.
                   Throws an exception if the heap is empty. */
void add (const T & item);
                /* adds the item to the heap. */

Heap (int d, bool maxHeap);
     /* constructs a new empty Min or Max heap which will store its elements in a d-ary tree.
        If maxHeap is true, then the root should always contain the largest element.
        If maxHeap is false, then the root should always contain the smallest element. */
~Heap ();       /* a destructor */
```

For the implementation, your Heap should be stored in a `List<T>`, using your own implementation from Homework 4. (Using the C++ `vector<T>` class will lose some points.) You can implement it

"as-A" `List<T>`, or "has-A" `List<T>`. Both will receive full credit, though perhaps "as-A" is even more natural.

To implement the comparison functionality (to determine priority between pairs of elements), you should assume that the type `T` has overloaded the "`<`" and "`>`" operators.

(a) In Homework 4, you implemented four different versions of `List<T>`. Which of them is best to use for what you want to do here? Give a detailed justification for your choice.

(b) Now go ahead and implement the Heap. It may make a lot of sense to implement a private function `swap` for swapping two elements of the array, to follow the logic of the algorithm more closely.

(4) Above, you did a theoretical analysis of the impact of the tree's degree on the running time. Now, let's see how practice agrees with theory, using your own implementation. Refresh yourself on how to do timing measurements.

(a) Build a Heap of integers using a d-ary tree, and insert $n = 10000$ randomly generated numbers into it, and measure the time this takes. Do this for at least 10 values of d ranging from $d = 2$ to $d = 100$, roughly evenly spread. Give a plot of the running time for these insertions as a function of d.

(b) Next, we measure the time for removals. Again, build a heap of $n = 10000$ random integers inserted into a d-ary tree. Measure the time it takes to remove them one by one from the heap. Take care *not* to measure the time for inserting them. Give a plot of the running time for these removals as a function of d.

(c) Describe to what extent what you see in the plot agrees or disagrees with what your theoretical calculations predicted.

(5) Just to get to play a little with your Heap, build the following simple interactive main function. The user will be asked repeatedly whether to add or remove a student record. Each record consists of two strings: the name of the student, and their student ID. These should be stored inside a `struct` or `class`.

The program will then add the entered data item to the heap, or remove the alphabetically first person from the heap (depending on what the user asked). After each operation, the program also outputs the current alphabetically first student on the heap and his/her student ID.

(6) Chocolate Problem (2 bars): Same rules as in the past.

Sokoban is a fun single-player puzzle game. It involves a player moving around in a level that's a rectangular grid, and trying to push boxes to their designated locations. If you don't know the rules, check out `http://en.wikipedia.org/wiki/Sokoban`.

This problem is about writing a solver for Sokoban, with only a *single box and its destination*. You can define your own way of specifying a level, which will presumably somehow mark the walls, empty squares, the box location, the box destination, and the person's location.

The intermediate state of a game is characterized by two things: the box's position (in (x, y) coordinates) and the person's position (also in (x, y) coordinates). From one state, you can get to other states based on walking in one of the four directions, which may involve pushing a box. The total cost of a sequence is measured first by the number of pushes; then, ties are broken by the number of walks.

Your goal is to write a solver that will find the cheapest solution for the level it reads (according to the metric given above). In solving the puzzle, you should maintain a priority queue of all states you have found out you can reach. You start with only one state (your start state) in the priority queue. Then, in each step, you should take a state, explore all possible moves, and add them into the priority queue with their costs. Of course, you should never add a state that has already been explored in the past, as it cannot be any cheaper now.