CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2020
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
Handout 10: Quiz 1 Coding Part. 30 possible points. — Flour 132

You can use any resources you want for this quiz except other humans. You must upload your solutions to handin by the end of your lab session; files with upload timestamps beyond the end of lab will not be counted. All code must run on the School of Computing machines. it will be compiled with `-std=c++11`. Unless otherwise noted, de-allocation of memory is not necessary for any of your solutions.

1 Almost Sorted (10 points)

You are given a linked list of n distinct integers that would be in sorted order if not for one "out of place" element. That is, removing one element and re-inserting in a new location in the list would be all that is necessary to restore the list to sorted order. Please write a function that takes a pointer to the head of such a list and re-orders its elements properly so they are now in sorted order. Your function should return a pointer to the head of the resulting list, and it should be neutral with respect to memory allocation (i.e,. the total amount of memory allocated after the function returns should be the same as before the function was called). For full credit, your code should run in $O(n)$ time. A recursive solution is fine (i.e., you are absolved of the responsibility of worrying about stack size for this problem). Starter code is provided: `prob1_starter.cpp`. Note that your solution should work on other examples beyond just the one used by the starter code.

2 Level by Level (10 points)

For this problem, you are given an n-node binary search tree with distinct keys that are all short strings (at most 25 characters, each a lower case letter in the range a . . . z). You should print out the contents of the tree *level by level*: the first line of output should be the root's key, the next line should be the children of the root in sorted order, separated by a space, the next line should be the grandchildren of the root in sorted order, separated by spaces, and so on (if there are, say, only 3 grandchildren, you would only print three things on this line). You should print out as many lines of output as there are levels in the tree.

The input to your program (see `prob2_input.txt` for an example) consists of $n-1$ lines, each with two strings a and b, indicating that the node with key a is the parent of the node with the key b in the binary search tree (there are only $n-1$ lines instead of n lines because the root node has no parent). The tree is guaranteed to be reasonably well balanced, and it is guaranteed to satisfy the standard BST property, where nodes in the left subtree of a node are alphabetically smaller, and nodes in the right subtree are alphabetically larger.

For partial credit on this problem, you may do any of the following:

- Build a reasonably-balanced binary search tree on all the distinct strings provided in your input, even if it is not the exact shape of the binary search tree specified by the parent-child relationships in the input.
- Write code that prints a binary search tree out level-by-level, even if you are not able to run this on the binary search tree provided in the input.
- Build the binary search tree specified by the input file, even if you aren't able to print it out level-by-level.

3 Probing Predictions (10 points)

Consider a hash table of size n that uses sequential *probing* to resolve collisions. That is, the elements of the hash table are stored in an array $A[0 \ldots n-1]$, and inserting a new element x involves checking locations $A[h(x)]$, $A[h(x)+1]$, $A[h(x)+2]$, and so on until an empty location is finally found, wrapping around if we ever scan past the end of the table (empty locations are marked as such during initialization, say by storing -1 in those locations). The hash table size here is never changed, even when it becomes quite full, so the size always remains n.

A hash function h for you to use is provided in the starter code `prob3_starter.cpp`, although your solution to this problem should work correctly and efficiently no matter what hash function is being used. You may assume the hash function outputs an integer in the range $0 \ldots n-1$. You may also set n equal to 1 million for this exercise, since that's the value of n used by the provided hash function, although your should design your code so that it would be easy to test on other values of n as well.

Suppose you were to insert the integers $1, 2, 3, \ldots, n$ in this order into the hash table described above. Let p_i denote the number of locations in the table that are probed (examined) when inserting the value i into the table. For example, if you try to insert 7 into the table and $A[h(7)]$ is already occupied but $A[h(7)+1]$ is available, then $p_7 = 2$ (and afterwards, $A[h(7)+1]$ would contain 7).

Please write a program that quickly computes the *average* and also the *maximum* of the p_i's. Faster code will earn you more points. You will notice that if you simply simulate the operations of inserting $1 \ldots n$ using the hash function provided[1], this will not be particularly fast, since the hash function provided has been intentionally designed to cause a large number of collisions.

[1] It's worth noting that your code technically does not need to actually perform the operations of inserting $1 \ldots n$. Rather, it only needs to be able to *predict* how many locations would be probed by these operations.