CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2017
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 12:30-1:45
Handout 9: Midterm Quiz Solutions (In-Class Part) — McAdams 119

---

1. True/False (1 point each).

T		The height of an AVL tree on n elements is always $\Omega(\log n)$.
	F	If an algorithm runs in $\Omega(n)$ time, then it runs in $O(n^2)$ time.
T		If an algorithm runs in $\Theta(n^3)$ time, then it runs in $O(n^3)$ time and also $\Omega(n^3)$ time.
T		A queue can be implemented so that the operations of inserting an element at the front and removing an element from the back both take $O(1)$ time.
	F	In a network with n webpages, each iteration of Pagerank will run in $O(n)$ time (here an "iteration" is what we performed 50 times on homework #1).

2. Running Time (3 points). For each of the code samples below, please write the running time as a function of n using $\Theta()$ notation. **Answers:** $\Theta(n)$, $\Theta(n^2)$, $\Theta(n \log n)$.

```
// CODE EXCERPT 1:
for (int i=0; i<n; i++)
  cout << "Hello world\n";
```

```
// CODE EXCERPT 2:
for (int i=0; i<n; i++)
  for (int j=i+1; j<n; j+=2)
    cout << "Hello world\n";
```

```
// CODE EXCERPT 3:
for (int i=0; i<n; i++)
  for (int j=i; j>0; j/=2)
    cout << "Hello world\n";
```

3. Verify (4 points). Given two length-n integer arrays A and B, please describe, in English, a fast algorithm that can verify if B contains the sorted contents of A.

A simple and correct algorithm is to just sort A in $O(n \log n)$ time (say, by inserting A's contents into a balanced binary search tree and doing an in-order traversal) and then compare the result with B element-by-element in $O(n)$ time. However, we can do slightly better, by checking that (i) B is sorted, and (ii) B contains the same elements as A. We can check (i) in $O(n)$ time by simply scanning through B, verifying that each element is no smaller than the element before it. We can check (ii) in $O(n)$ expected time by inserting the elements in A into a (universal) hash table (inserting multiple copies of a value if there are duplicates present), then scanning B and removing its elements from the hash table. If we remove all the elements successfully, then B contained the same elements as A.

4. Missing Element (4 points). Suppose you are given a balanced binary search tree T_1 containing n distinct strings, and another balanced binary search tree T_2 containing all but one of the strings in T_1. Please describe a fast algorithm for determining the missing element – appearing in T_1 but not T_2.

There are many nice solutions here. For example, one could find the inorder traversal sequence from T_1 and T_2 in $O(n)$ time and then in $O(n)$ time walk through and compare these element-by-element. For an even faster approach, let x be the root of T_1. Call *find*(x) in T_2 ($O(\log n)$ time). If absent, we are done. Otherwise, remove x from T_2 and insert it at the root of T_2 as in lab 4 (also $O(\log n)$ time). Now T_1 and T_2 have the same root, so we can compare subtree sizes. It must either be the case that T_2's left subtree is one element smaller than T_1's left subtree (so T_2's left subtree contains the missing element), or the same for their right subtrees. Recurse on whichever side contains this discrepancy. Since we spend $O(\log n)$ time and then recurse one step down the tree (and the tree is balanced, having height $O(\log n)$), total running time is $O(\log^2 n)$ (in expectation, since our trees are randomly balanced).

5. Tree Copy (4 points). Please fill in the function below so it creates a duplicate copy of an entire balanced binary search tree.

```
Node *copy_tree(Node *root)
{
  if (root == NULL) return NULL;
  return new Node(root->key, root->size, copy_tree(root->left), copy_tree(root->right));
}
```

6. Debugging (5 points). Corrected versions below:

```
int sum(int n) // Was int &n, but this wouldn't have compiled
```

```
int find_max(Node *root)
{
  if (root->right == NULL) return root->key;
  return find_max(root->right);  // only need to recurse right, not left
 }
```

```
// Without passing S by reference (or with a pointer), it will go out of
// scope twice and therefore its destructor will try to free its hash
// table twice, causing a crash.  So we should use
void insert_hello_world(Stringset &S)
// or
void insert_hello_world(Stringset *S)
```

```
// A[i][j][k] is of type int, not int *, so assigning it to a new int
// will not compile.  Something like this is more appropriate:
A[i][j][k] = 0;
```