CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2019
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MW 2:30-3:45
Handout 11: Quiz 1 Solutions — Daniel 313

1. True/False. Please circle T (true) or F (false):

F A skip list made of n elements has height $O(n^2)$.

F Take a binary tree representing an arbitrary sequence (the variant of binary search tree we studied in lab 5, which we interact with using rank-based access instead of value-based access). If we pick some node x in the tree and repeatedly rotate x with its parent until x becomes the root, this can potentially change what is printed out during an in-order traversal of the tree.

F If an operation runs in $O(n)$ time and an algorithm calls this operation n times, then the algorithm must run in $\Omega(n^2)$ time.

T If we insert the integers $1, 2, \ldots, n$ into an AVL tree, this takes $O(n \log n)$ time.

F If we insert the integers $1, 2, \ldots, n$ into a splay tree, this takes $\Omega(n \log n)$ time.

T If we `push` n elements successively onto a stack and then call `pop` n times, the elements will be output in reverse order.

T The worst-case running time of randomized quicksort applied to an already-sorted array is $\Theta(n^2)$.

T If a C++ class has a private integer member variable x, then a public function in the class can return a pointer to x, allowing x to be changed by a function that is not a class member.

T Inserting n elements into a hash table of size $\sqrt{n}$ (using chaining to resolve collisions, and not checking if duplicates are present) can be done in $O(n)$ time.

T A in-order traversal of an n-element binary search tree always takes $O(n)$ time, even if the tree is not balanced.

2. Heap. Suppose you insert the numbers $5, 4, 3, 2, 1$ in that order into a standard "min" binary heap (a heap designed for easy extraction of the minimum element) that is initially empty. Please write down the contents of the array representing the heap afterward: **1 2 4 5 3**

3. TreePath. Please complete the following function that takes a pointer to the root of a balanced binary search tree and an integer x returns a pointer to the head node of a newly-allocated linked list representing the path in the tree from the root down to x. The figure above shows an example. If x does not exist in the tree, the function should return NULL.

```
ListNode *getpath(TreeNode *root, int x)
{
  if (root == NULL) return NULL;
  if (x == root->key) return new ListNode(x, NULL);
  ListNode *L = getpath(x < root->key ? root->left : root->right, x);
  return L ? new ListNode(root->key, L) : NULL;
}
```

4. Similar Elements. You are given pointers to the roots of two binary search trees T_1 and T_2, each containing $n/2$ elements. Please describe, preferably in English rather than code, an efficient algorithm for determining whether there is some element x in T_1 and some element y in T_2 such that x and y differ in value by at most 7.

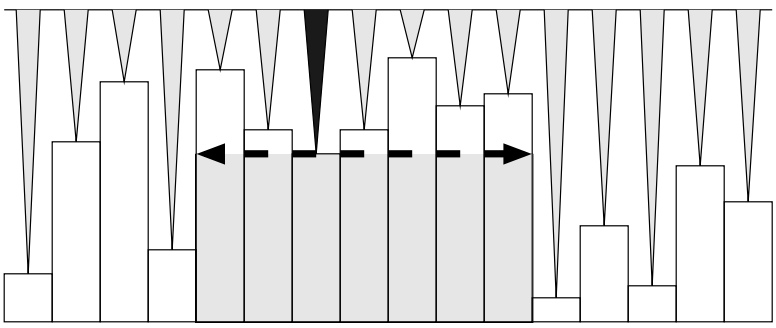
The best solution, running in $O(n)$ worst-case time, is probably to merge the inorder traversals of T_1 and T_2, checking along the way if two adjacent elements in the merged sequence come from different trees and have difference at most 7. Alternatively, we can use hashing to get $O(n)$ anticipated time: traverse T_1 and insert its contents into a hash table (this takes $O(n)$ time). Then traverse T_2 and for each element y in T_2, do a hash lookup on $y-7, y-6, \ldots, y+6, y+7$. This is only $O(1)$ hash lookups per element in T_2, so we anticipate this taking also just $O(n)$ time.

5. Black Boxes. (a) Describe how you can figure out, via some manner of computational testing, how to differentiate merge sort from insertion / bubble sort and insertion sort from bubble sort.

Run each program on length-n reverse-sorted arrays for increasing values of n. Insertion and bubble sort will scale quadratically in running time, which should be quite apparent versus the $\Theta(n \log n)$ scaling of merge sort (e.g., running on 1 million elements, merge sort will terminate within a second or so while the others will run for a very long time). To differentiate insertion sort and bubble sort, run each program on a length-n array containing $2, 3, 4, \ldots, n, 1$. This causes insertion sort to run in $\Theta(n)$ time and bubble sort to run in $\Theta(n^2)$ time, so they will be similarly easy to differentiate as n grows large.

6. Building. What is the largest rectangle you can fit inside the bar graph described by an array. Please describe an efficient algorithm for determining this maximum possible area and give its running time.

This can be solved in $O(n \log n)$ time using a trivial adaptation of the lab 6 sweep line algorithm for the longest line of sight problem, since it is essentially the same problem just "upside down":



The longest lines of sight to the left and right of each triangle define the width of a prospective rectangle (the one whose height is defined by the bar coinciding with the tip of the triangle). We take the best of all n such prospective rectangles as our answer, since we know the optimal rectangle is constrained above by the tip of some triangle. As a note, if you want something fun to think about, there is also an $O(n)$ solution that works by building a treap from the n bars – as a hint, the BST key is the x location of a bar and the heap key is the bar's height.