**COSC 3320 Assignment 2**

**1.** Assume you are given two linear lists of size n each; consider the problem of determining whether any element of one list is an element of the other (not value, **element**!!).

**(a)** Derive a lower bound for this problem.

A lower bound for this problem **could be O(n)** if we make a few assumptions. First, we already know that both lists are of same size n, but now lets also assume that both lists are sorted. Next, let us assume that both lists have the same values and neither have duplicate values. This important because if the two lists have values {1, 2 , 3, 4} and {1, 1, 3, 4} the first element in the first list would have to be compared to both the first and second element in the second list because they share the same values. Also, if the lists do not have to have the same values, we would have to parse the second list until we find a matching value from the first list such as in {6, 7, 8, 9, 10, 11} and {1, 2, 3, 4, 5, 10}. Because one value can be stored in multiple elements but an element can not store multiple values, if we make the above assumptions we will have two lists that would look something like {1, 2, 3, 4} and {1, 2, 3, 4} which means we only make one comparison for each element. And because there are n elements there would be n comparisons, hence O(n). Otherwise if the only thing that we know about the lists are that they are both of size n, we would need to check each element in the second list for each element in the first list which would be n\*n, or **O(n^2)**.

**(b)** Design an algorithm for this problem. Derive its time complexity. It should be as close to your lower bound as possible.

For each element in the first list, we need to find each element in the second list that has a matching value. Then if the two elements have the same value, change the value of one element and compare again. If the two elements have different values now, then they are in fact two different elements and the next element in the second list needs to be compared. Otherwise, if they still have the same values, then changing the value of the element in the first list also changed the value of the element in the second list, which means the two lists are sharing an element.

For x in list1

For y in list2

If x == y // shared value found

X++ // change value of element from list1

If x == y // check if changing value of x also changed value of y

return 1 //shared element found!

Continue

Continue

Return 0 //no shared elements

Because of the two nested for loops of size n, **the time complexity of this algorithm O(n^2)**

### 2. On-line median

In class, we discussed the on-line kth-largest problem. We solved it, using an augmented AVL-tree structure, with the following characteristics:

Insert(x) in time and space O(log2n) where n is the number of elements in the structure at this time (i. e., the number of Insert operations, minus the number of Delete operations, up until now).

Delete(x) in time and space O(log2n) where n is the number of elements in the structure at this time (i. e., the number of Insert operations, minus the number of Delete operations, up until now).

Find(k) in time O(log2n) and space O(1) where n is the number of elements in the structure at this time (i. e., the number of Insert operations, minus the number of Delete operations, up until now).

Suppose that instead of doing the Find(k) operation, with k an arbitrary positive integer that can vary from one Find to the next, we replace it by

Find(⎡n/4⎤)

where n is the number of all elements that are currently stored in the structure.

Can you devise a data structure and algorithms for

Insert(x)

Delete(x)

Find(⎡n/4⎤)

which improve over the Find(k) approach discussed in class. (Obviously, that approach will still apply, so we know that all three operations can certainly be done in time and space O(log2n); however, the question for you to solve is: Can you do better??).

Carefully formulate your data structure, outline the three algorithms in some detail, and determine with care the time and space complexities of your three algorithms.

(If your structures/algorithms are based on standard structures/algorithms, emphasize in what way yours are different. Do not repeat everything!)

**NOTE TO SELF:**  make a structure where the insert(x) and delete(x) functions keep the n/4 node at the root. Insert and delete will probably take longer but Find will be instant O(1).

Make a tree such that the root is always the ⎡n/4⎤th largest value, where the left and right subtrees are independent AVL trees. The left subtree will contain values less than (or equal to) the root value, and the right subtree will contain values larger than the root value.

Insert:

If value <= root:

insert value in left subtree as normal

If number of nodes % 4 == 0: // else the root doesn’t need to change

Move root into right subtree

Make largest value in left subtree the new root

Increase number of nodes

Else If value > root:

Insert value into right subtree as normal

If number of nodes % 4 != 0: // else the root doesn’t need to change

Move root back into left subtree

Make smallest value in right subtree the new root

Increase number of nodes

Delete:

If value <= root:

Normal delete and balance left subtree up to root (if root was deleted than the largest value from the left subtree will be the root at this point)

Decrease number of nodes

If number of nodes % 4 == 0: // else the root doesn’t need to change

Move root back into left subtree

Make smallest value in right subtree the new root

Else If value > root:

Normal delete and balance right subtree up to, but not including the root.

Decrease number of nodes

If number of nodes % 4 != 0: // else the root doesn’t need to change

Move root into right subtree

Make largest value in left subtree the new root

Find:

Return root value

Summery:

The insert and delete functions are changed to keep the ⎡n/4⎤th largest value at the root with two self-balancing AVL trees where the right subtree will always contain ⎡# of nodes/4⎤-1 number of nodes that are a higher value than the root. For example, if values {0, 1, 2, 3} are inserted into the tree, then the root would be ‘3’ with no values in the right subtree, and the number of nodes will be incremented to equal 4. If the list of values to be inserted was {0, 1, 2, 3, 4} then ‘4’ will be inserted into the right subtree. ‘3’ will stay the root because number nodes mod 4 would be equal to 0, and then the number of nodes will be incremented to equal 5. The delete function is an inverse of insert and they both make sure that the the ⎡n/4⎤th largest value at the root by making sure the root is larger than the left and smaller than right and making sure that only ⎡# of nodes/4⎤-1 nodes are in the right tree at all times. This Makes insert and delete have more operations than normal but still grow with a O(log2n) time complexity; However, now the Find function has constant time complexity O(1) because the key value is always at the root independent of n.

### 3. Multiplying rectangular matrices

Assume that every possible way of evaluating a sequence of n such matrices (every possible way of placing parentheses) is equally likely. Design and algorithm that determines the average amount of work (in terms of scalar multiplications) for a given sequence of n matrices. Precisely define what is “average work”! Determine its time and space complexity.

# Programming

**Remember: For each of these three programs, you are expected to write a report that uses that program as a research tool**

**4**. Write a program, using your favorite computer (under some operating system, supporting VMM) and your favorite programming language, to implement the algorithm on p. 126 for n=16, 64, 256, 1024, 4096, and 16384, and for two values for m, m=1 677 721 600 and m=13 421 772 800 (that is, m does not depend on n). Determine the timings for your twelve instances. **Carefully discuss and interpret your results!** What should be the computational complexity of the twelve runs?

**5**. Conduct the following experiment that should provide information about the use of garbage collection on your specific computing platform: Implement insertion and deletion for AVL trees, except instead of having as the content I(N) of the node N a single integer val, let it consist of that integer val (to govern the insertion into its appropriate location in the search tree) plus a large matrix of size M. Furthermore, choose M as follows: If val = 0 mod 3, then M = 220; if val = 1 mod 3, then M = 219+218; if val = 2 mod 3, then M = 218+217 (these values should guarantee that fragmentation of the available memory will occur quite rapidly). Now randomly choose a large number, perhaps 100,000, of values between 0 and 299 for insertion and deletion, making sure that your tree never contains more than 50 nodes. (If your compiler is very clever, it may be necessary to assign values to some of the array elements – to ensure that the compiler is unable to conclude that the array is not needed since it is never used.) Measure the time each of the insertions and deletions takes. Since your tree never has more than 50 nodes, its height cannot exceed 6 (since an AVL tree of height 7 must have at least 54 nodes); consequently, the complexity of the insertion and deletion operations is quite small. However, the repeated insertions and deletions, together with the size of the matrices in the nodes created, should result in extensive memory fragmentation, which in turn should engage garbage collection and subsequently memory compaction in a major way.

**6**. Design a program that illustrates the influence of virtual memory management on execution.

Specifically, for a computer platform that uses VMM, determine the size of the active memory set and the access characteristics of the components involved in the VMM (size of page, access times, etc.). Then write a synthetic program that uses a relatively small amount of data for extensive computations. In more detail, if the size of the active memory set is M, have your program load a data set of size C into the cache and carry out a number of operations (involving this data set) that is several orders larger than C. Determine and plot the timings for C= 0.5**.**M, 0.6**.**M, 0.7**.**M, 0.8**.**M, 0.9 M, 0.95**.**M, 0.99**.**M, 1.0**.**M, 1.01**.**M, 1.1**.**M, 1.5**.**M, 2**.**M, 5**.**M, 10**.**M, and 50**.**M. Pay attention to the replacement policy of the VMM and structure your computations so that you can be certain that thrashing occurs for C>M.

**Points: 1: 15 2: 20 3: 15 4: 18 5: 16 6: 16**