The methods of the ADTs that we have studied are in the attached pdf file.

For the following algorithm design questions, you can use any of the ADT's and algorithms that we studied in class as building blocks, i.e., any of the ADT's listed in the attached pdf file or (if/when appropriate) you can call any algorithms such as InsertionSort, MergeSort, QuickSort, BinarySearch, etc. However, since BucketSort and RadixSort are not standard (number of buckets and number of digits vary), you would have to write them yourself as part of the algorithm.

To enter your solution, click on the "Show Rich-Text Editor" link below, then click on the "maximize" icon (found in the editor header on bottom line, second from the right and you will see an icon with four arrows pointing out; click on this icon). This will give the ability to do full screen editing where you can add your pseudo-code algorithm.

1. (a) **[25]** Design a pseudo code function, **isSubset(A, B)** that determines true or false whether or not a Sequence **A** of integers is a subset of the integers in another Sequence B, i.e., isSubset returns true if and only if every element in A is also a member of B. For example, suppose A=(7,5,12,4,3) and B=(7,5,2,4,3,10,12), then isSubset would return true because every member of A is also a member of B. However, if A=(7,5,20,12,4) and B=(7,5,2,4,3,10,12), then isSubset would return false because 20 is a member of A but is not a member of B.

(b) [**5**] What is the time complexity of your algorithm? Justify your answer.

2. (a) **[25]** Design a pseudo code function, **removeDups(L)** that removes the duplicate elements in a List **L** of integers i.e., the output must not have any duplicate elements in it. Furthermore, the input **List L must not be altered in any way**; thus you need to create a new separate output List so you do not change the input List L. For example, suppose L=(7,5,2,5,7,4,3,2,3), then a valid output would be (7,5,2,4,3) with no duplicate integers. The order of the output does not matter; thus, for example, the output could be sorted, i.e., (2,3,4,5,7) would also be a correct output. However, again the input **L** must NOT be changed, i.e., it must remain (7,5,2,5,7,4,3,2,3). **Hint**: note that all of our sorts modify the input so just calling Sort(L) would modify L and thus would NOT meet the specification.

(b) [**5**] What is the time complexity of your algorithm? Justify your answer.

3. (a) Using the EulerTour template attached, override the methods so the EulerTour template finds the position/node ***p*** containing the smallest element in the tree. **Hint**: return ***p*** NOT the element.

(b) [**5**] What is the time complexity of your algorithm? Justify your answer.

```
class FindMin extends EulerTour {  // insert your code here
        findMin(T) {
            // and here
        }
}
```

Question3.pdf 12 KB

4. (a) **[20-25]** Design a **recursive** pseudo code function, **findTwoLargest(L)**, that finds the two largest integers in a List **L** of integers that could contain duplicates. List **L** must NOT be modified. The output must be an array in which the first element, at index 0, is the largest and the second element of the array is the second largest. For example, suppose L=(7,5,2,5,7,4,3,2,3), then the output would be the array containing [7,5]. Note that there are two 7's, but the output is [7,5], i.e., the two largest must be unique (different). However, again the input **L** must NOT be changed, i.e., it must remain (7,5,2,5,7,4,3,2,3).

Hint: note that **findTwoLargest(L)** is NOT going to be recursive, but one or more of its helpers must be recursive to receive full credit.

[20] If you do not feel comfortable writing a recursive algorithm, then an iterative version is worth up to 20 points as long as you traverse the list using methods first(), after(p), isLast(p), etc.

[20] Similarly, if you are having trouble handling the duplicates, then assume that L does not contain duplicates.

(b) [**5**] What is the time complexity of your algorithm? Justify your answer.

ADTs-cs421-algs.pdf 11 KB

You are given an algorithm A with running time O(n^5) in every case (best case, worst case, and average case) and algorithm B with running time O(n) in every case. Very briefly describe why A might be faster than B on some inputs. Your answer must be based on the definition of βιγ–O. Your answer **cannot** have anything to do with memory limitations, disk accesses, or paging (i.e., memory is unbounded, disk accesses take 1 unit of time, etc.).