Department of Computer Science and Software Engineering
Concordia University
COMP 352: Data Structure and Algorithms
Summer 2016
Assignment 2
Due date and time: Friday June 3rd , 2016 by midnight

Written Questions (50 marks):

QA.1

a) Develop a well-documented pseudo code that finds all two elements of the array that sum up exactly to x. The code must display the indices and the values of these elements. For instance, given the following array [25,12,37,48,36,61,15,11,37,19,22,58,28,55] and *x* as 73, your code should find and display something similar to the following (notice that this is just an example. Your solution must not refer to this particular example.):

   All 2-Elements summing up to a value of 73 are:
   Indices 0 & 3 with values 25 & 48
   Indices 1 & 5 with values 12 & 61
   Indices 2 & 4 with values 37 & 36
   Indices 6 & 11 with values 15 & 58
   etc.

b) Briefly justify the motive(s) behind your design.
c) What is the Big-O complexity of your solution? Explain clearly how you obtained such complexity.
d) What is the Big-Ω complexity of your solution? Explain clearly how you obtained such complexity.

QA.2

Given a non-sorted array *A* of *n* integers and an integer value x:

a) Similar to Question 1, develop a well-documented pseudo code that finds all two elements of the array that sum up exactly to x. The code however must be different than the one you had in Question 1 and must use either a stack *S* or a queue *Q* to perform what is needed.
b) Briefly justify the motive(s) behind your design.
c) What is the Big-O complexity of your solution? Explain clearly how you obtained such complexity.
d) What is the Big-Ω complexity of your solution? Explain clearly how you obtained such complexity.
e) What is the Big-O *space* complexity of the utilized stack or queue? Explain your answer.

QA.3

a) Let *A* be an array storing positive integers. Write in pseudocode an algorithm that rearranges the elements of *A* so that the odd elements appear before the even elements. For example, if the input to the algorithm is an array *A* = {8; 7; 6; 11; 4}, and example of a valid output is A = {7; 11; 8; 6; 4}. Your algorithm is not allowed to use any additional container data structures (such as linked lists, sets, etc.) That is the re-arrangement should be done inside the input array. Other outputs, for example *A* = {7; 11; 4; 6; 8}, are also valid, as long as the odd elements appear before the even ones. Compute the time complexity of your algorithm in the worst case. Explain how you computed complexity.

b) Show (no pseudo code) how you can reduce the complexity that is reach a linear time algorithm or better, if the problem allowed extra data structures for storage,

QA.4

a) Draw a single binary tree that gave the following traversals:

Inorder: S A E U Y Q R P D F K L M

Preorder: F A S Q Y E U P R D K L M

b) Assume that the binary tree from Question (a) above is stored in an array-list as a complete binary tree as discussed in class. Specify the contents of such an array-list for this tree.

QA.5

Give an algorithm for computing the depths of all the nodes of a tree T, where *n* is the number of nodes of *T*, in O (n)-time.

Programming Questions (50 marks):

For this question, you are required to design and implement your own version of the *Position* ADT and the *Node List* ADT. You are not allowed to use any of the built-in classes or interfaces provided by Java, which provide similar operations. You are however allowed to use any provided Java methods to find/calculate execution times. To simplify your task, the elements stored in the nodes are assumed to only be integer values. Here are the details of the ADTs:

The Position ADT (or the class implementing it) has one single method as follows:

- ***element( )*:** Returns the element (integer) stored at this position.

Your implementation of the Node List ADT must include at least all the methods indicated below. In addition, this class must use arrays as its underlying supportive structure to perform and achieve what is needed. This array, its implementation and manipulations, should never be seen by the user of the Node List ADT; rather it is only known to the developer of this ADT (you!). In addition, since the number of nodes in this Node List is not known and can dynamically change, your implementation of this array must allow for dynamic expansion of the array size once the Node List becomes 80%, or more, full. An expansion would either double the size of the array, or increase it by a constant amount of 10 additional array elements, based on some configuration as will be explained below. You are free to use any naming convention to assign the names of the positions (i.e. A, B, C, etc. or P1, P2, P3, etc., nonetheless, this has to be scalable).

- ***first():*** Returns the position of the first element; error if list is empty;
- ***last():*** Returns the position of the last element; error if list is empty;
- ***prev(p):*** Returns the position preceding position *p* in the list; error if *p* is first position;
- ***next(p):*** Returns the position following position *p* in the list; error if *p* is last position;
- ***set(p, e):*** Replaces the element at position *p* with *e* , and return the old element at position *p*;
- ***addFirst(e):*** Inserts a new element *e* as the first element and returns the position object;

- ***addLast(e):*** Inserts a new element *e* as the last element and returns the position object;
- ***addBefore(p, e):*** Inserts a new element *e* before position *p* and returns the position object;
- ***addAfter(p, e):*** Inserts a new element *e* after position *p* and returns the position object;
- ***delete(p):*** Removes and return the element at position *p*. This also invalidates that position *p* in the list;
- ***swap(p1, p2):*** Swaps the elements pointed by *p1* and *p2*;
- ***truncate( )***: Truncates the underlying array size to the exact number of current positions in the Node List;
- ***SetExpansionRule ( )***: Accepts a single character as a parameter ('*d*' for doubling the size of the underlying array, or '*c*' for increasing the size of the underlying array by constant value, which is 10). By default, the expansion rule should be set for doubling the size. Further calls to *SetExpansionRule ( )* would change the expansion rule from one rule to another according to the passed parameter.

Particularly, you are required to:

i) Write a pseudo code description of your *Position* and *Node List* ADTs and their interfaces including comments about their assumptions and semantics.

ii) Write a pseudo code of the methods of your Node List ADT (the methods themselves). Keep in mind that Java code is not, and will not be considered as, pseudo code. Your pseudo code must be of a higher and more abstract level.

iii) Indicate clearly what the complexity is, using Big-O notation, for each of your Node List ADT methods. You should provide two different complexities for every method: 1) complexity of the Node List operations in isolation (that is without considering the complexity to manipulate the underlying array), and 2) the complexity of the methods including the manipulation of the underlying array.

iv) Indicate whether or not it is advantageous to have the *truncate( );* method. Explain your answers very clearly.

v) Implement a well-formatted and documented Java source code for the entire Position and Node List ADTs as described above.

vi) Implement a driver code (i.e. a *main*() method) that tests your implementation and the functionality of your ADTs. Your driver must have sufficient test cases to validate your entire implementation.

vii) Additionally, the driver must periodically switch between the two expansion rules. In particular, your code must set the expansion rule to one of the two rules, force multiple expansions and collect sufficient information on how much time was needed for these expansions to take place. The code must then switch to the other expansion rule and collect similar information. In fact, you can switch between the two rules as many times as you wish if this is needed to collect the required information.

viii) In your opinion, which of the two expansion rules is likely to provide better performance? Explain why? Observe the values obtained as a result of switching the expansion rules above, then specify and justify the amortized time for both array expansion schemes given *n add* operations (*addFirst()*, *addLast()*, *addBefore()*, etc.). Which expansion rule has resulted in

better performance? Do the results reflect your initial answer? Explain very clearly why or why not?

ix) In your opinion, would the complexity of the methods (not considering the manipulation of the underlying structure) change if the underlying structure was a linked list instead of an array?

x) In your opinion, would the complexity of the methods (considering the manipulation of the underlying structure) change if the underlying structure was a linked list instead of an array? Explain your answer very clearly.

Both the written part and the programming part must be done either individually or in a team of two (max) students (no groups are permitted). Submit all your answers to written questions in PDF (no scans of handwriting) or text formats only. Please be concise and brief (less than ¼ of a page for each question) in your answers. For the Java programs, you must submit the source files together with the compiled executables. The solutions to all the questions should be zipped together into one .zip or .tar.gz file and submitted via Moodle under Assignment 2_DropBox.