CSP2348D Data Structures

Assignment 2: A Mini Team Project

- Algorithm Design, Analysis & Implementation

Objectives from the Unit Outline

- Analyse complexity and performance of their associated algorisms.
- Apply abstract data types to programming practices.
- Describe the concept, application, and specification of an ADT and employ classes to encapsulate ADTs.
- Describe the general principles of algorithm complexity and performance.

General Information:

- Algorithm complexity analysis is one of the most important skills covered by this unit. It can be done in either or both of theoretical analysis and experimental studies. The theoretical algorithm analysis is generally conducted by applying asymptotic theory/methods, such as in O-notations, etc., to the algorithm's complexity function, which reflects the number of the basic operations the algorithm has to execute over the input size, therefore to estimate the growth rate of the function. On the other hand, the experimental studies require implementation of the algorithm/s, thus to reveal the growth trend of the complexity function/s.

- This is a mini programming project, requiring **up to two students** to work as a team to complete it. It consists of six questions. You will be required to not only do algorithm complexity analysis (to part of the questions), but also implement the tasks using either Python or Java programming language. (Note: if you wish to use a programming language other than Python/Java, please let your tutor know, and you may be required to demonstrate your work in the end).

- The first four questions are closely related. The first question requests you design algorithms to implement specific sorting strategies. The second to forth questions are for theoretical and experimental analysis for sorting algorithms. Part of the question also requires algorithm analysis using big-O notation. The 5th question requires you design and implement part of the application scenario using singly-linked list (SLL) to store the required information. Part of the codes has been completed. You are requested to complete the rest. The 6th question requires you modify some algorithms and then convert them into Python/Java code to implement specific application scenarios using binary tree data structure.

- It is your responsibility to form your team/group. Please send the details of your team members (i.e., the student IDs and names) to your tutor by the end of week 9, if your team has more than one member (if you really prefer to work individually, you may do so but no workload is to be reduced).

Due: Friday in Week 11 @ 5:00 pm

(i.e., 2nd last teaching week, see unit **Schedule**)

Value: 20%

Main assignment document format requirement:

Must contain	**Cover page** Must show assignment title, student IDs and name/s of your team, due date etc.
	Executive Summary (optional) This should represent a snapshot of the entire report that your tutor will browse through and should contain the most vital information needed.
	Table of Contents (ToC) This must accurately reflect the content of your report and should be generated automatically in Microsoft Word with clear page numbers.
	I**ntroduction** Introduce the report, define its scope and state any assumption; Use in-text citation/reference where appropriate.
	Main body The report should contain (but not limited to): • Understanding of concepts/techniques involved in the report. • Any strategies used to solve the problems (e.g., an approach to develop a solution?) • The questions being solved/answered, e.g., Q1&Q3: must have: (i) key algorithm/s (pseudo codes); (ii) algorithm analysis (if applicable); (iii) screen shots (of execution of the converted Python/Java code/s, at least one screenshot per function); Q5: algorithms and analysis using O-notation; screen snapshots of execution of the code (at least one snapshot per method to show the running result of your code); Q6: key screen snapshots of execution of the code (at least one snapshot for b), c) and d)); • Comment/Discussion or a critique of the solution developed /achieved, etc. • No Python/Java code needs to be attached in body of the report (they should be saved as separate running codes and submitted as separate files).
	Conclusion Outcomes or main works done in this assignment.
	References A list of end-text reference, if any, formatted according to the ECU requirements using the APA format (Web references are also OK).
Other requirement	The report should be no more than 6 pages (excluding references, snapshots and diagrams) for individual assignment, and no more than 10 pages for team work. The text must use font **Times New Roman** and **be not smaller than 12pt**.

Submission Instructions

- Submit your Assignment 2 via Blackboard electronic assessment submission facility. For a detailed submission procedure, please refer to "How to submit your Assignment online**"** item in the Assessment section of this unit website.
- Your submission should include the assignment main document (i.e., a report) and your Python/Java classes' source codes. The main assignment document must be in

report style, in Word (or PDF) format (see detailed format requirement above). Pages must be numbered. Your Python/Java source code file/s must be runnable.

- One submission per team - Your submission should be in a single compressed file (.zip), which contains your mini project report and Python/Java source code file/s. Please name the zip file as
  <team-leader's Student ID>_< team-leader's full name>_A2_CSP2348.zip.
- Remember to keep the copy of your assignment. No hard copy is required.
- Your attention is drawn to ECU rules governing cheating and referencing. In general, cheating is the inclusion of the unacknowledged work of another person. Plagiarism (presenting other people's work and ideas as your own) will result in a Zero mark.
- ECU rules/policies will apply for late submission of this assignment.

Background Information

Arrays, linked lists and binary trees are typical data structures that are frequently used in many applications. While an array is a static data structure, linked lists and binary trees are among the most important yet simplest dynamic data structures. Many applications employ these data structures to model the applications scenarios.

An array is a random-access structure. An array component can be accessed using a unique index in constant time. This property makes array the most effective data structure in term of component access; therefore the most frequently used data structure.

A linked list is a sequential data structure, consisting of a sequence of nodes connected by links. Each node contains a single element, together with links to one or both neighbouring nodes (depending on whether or not it is an SLL or DLL). To access a linked list node, you must search it through the header of the linked list. Linked list is one of the simplest yet the most important dynamic data structures.

A binary tree is a non-sequential dynamic data structure. It consists of a header, plus a number of nodes connected by links in a hierarchical way. The header contains a link to a node designated as the *root* node. Each node contains an element, plus links to at most two other nodes (called its *left* child and *right* child, respectively). Tree elements can only be accessed by way of the header.

This assignment focuses on algorithm design, analysis, and implementation using array, SLL and binary tree data structures. While all questions are of equal importance, pay more attention to Q5 and Q6 as these questions enable you practise using typical linked list and binary tree based algorithms (e.g., SLL creation and data/link manipulation, binary tree traversals and their applications, etc.).

(Go to next page)

Tasks

Q1: design, analyse and write code/s to implement sort algorithms:

(1) Bubble sort algorithm and analysis.

Bubble-sort is one of the simplest elementary array-sorting algorithms. The principle of it is as below:
For a given array of *n* (comparable) elements, the algorithm scans the array n-1 times, where, in each scan, it compares the current element with the next one and swaps them if they are not in the required order (e.g., in *ascending order*, as usual). Based on this principle, you are required to

- Write the bubble sort algorithm (in pseudo code), which takes an array as a parameter;
- Analyse your algorithm using O-notation, i.e., by counting the number of comparisons (and also by counting the number copying operation);
- Convert your algorithm to a Python or Java code, and test your code using an array of some 20 elements.

(2) BST-based sort algorithm and analysis

For a given array of *n* (comparable) elements, the *BST-based sort* algorithm consists of two steps:

a) Insert all array elements into an (initially) empty BST;
b) Generate an *in-order* traversal sequence of the BST (as the sorting result).

You are required to

c) Write pseudo code to implement the *BST-based sort* algorithm, which takes an array as parameter;
- Analyse your algorithm using O-notation.
- Referencing to the codes for BST related algorithms, convert your algorithm to a Python or Java code, and test your code using an array of some 20 elements.

Q2: Theoretical summary of sort algorithm complexities

Review all arrays-based sorting algorithms that you have learnt so far (including those you've done in Q1), and fill in the missed complexity related data in Table 1, which has been partially completed (note the complexity data shown in Table 1 is of *Average case* unless noted otherwise). While no detailed algorithm analysis is required for this question, you are requested to fill in as detailed as you can.

Table 1: array sorting algorithm complexity (by theoretical analysis)

Sorting Algorithm	**No. of Comparison**	**No. of copies**	**Time complexity**	**Space complexity**
Bubble				
Selection	$\sim n^2/2$	$\sim 2n$	$O(n^2)$	$O(1)$
Insertion	$\sim n^2/4$	$\sim n^2/4$	$O(n^2)$	$O(1)$

Merge	*~ n log_2n*	*~ 2n log_2n*	*O(n log_2n)*	*O(n)*
Quick	**BC : ~ n log_2n* **WC: ~ n^2/2*	**BC :~ 2n/3 log_2n* **WC: 0*	**BC : O(n log_2n)* **WC: O(n^2)*	**BC: O(log_2n)* **WC: O(n)*
BST-base				

Notes: *: BC – best case; WC- worst case

Q3: Algorithm analysis by *experimental studies*

Produce Python or Java code/s to

(a) randomly generate an array, A, of size *n* (for *n* = 200, 400, 800, 1000, 2000); and
(b) sort the array A using array-based sorting algorithms/strategies – you are requested to modify the related array sorting algorithms such that they not only sort the array, but also count the total number of comparisons and, in addition, record the running time (say, in *ms*) for each run.

Collect data from the above running outputs and complete the following Table 2 and Table 3.

Table 2: Algorithm analysis by experimental studies: The average number of comparisons for sorting arrays of *n* integers (over 10 runs).

Sorting Algorithm	*n=200*	*n=400*	*n=800*	*n=1000*	*n=2000*
Bubble					
Selection					
Insertion					
Merge					
Quick					
BST-based					

Table 3: Algorithm analysis by experimental studies: The average running time (say, in *ms*) for sorting arrays of *n* integers (over 10 runs).

Sorting Algorithm	*n=200*	*n=400*	*n=800*	*n=1000*	*n=2000*
Bubble					
Selection					
Insertion					
Merge					
Quick					
BST-based					

To assist you prepare and achieve the solution/s, the following steps are recommended:

(1) Write Python/Java codes to implement the following array-based algorithms that can be used to sort an array of *n* integers:
   a) Bubble sort
   b) Selection sort
   c) Insertion sort
   d) Merge sort
   e) Quick sort
   f) BST-based sort

   (Note that most of those array-sorting algorithms have been implemented and tested in the lab sessions or implemented in Q1).
(2) For each of the above codes, make necessary modification on it so that it not only sorts the array, but also counts/outputs the number of comparisons, and records running time, for each run;
(3) Write Python/Java code that, for a given number of *n*,
   i). randomly generates an array A of *n* integers; and
   ii). sorts the array A, using *Bubble* sort (*Selection* sort, *Insertion* sort, *Merge* sort, *Quick* sort, *BST-based* sort, respectively), and outputs the required data (i.e., number of comparisons and running time);
(4) For *n* = 200, run the Python/Java codes completed in step (3), and collect the output data in this case.
(5) Repeat step (4) for 10 times (i.e., 10 runs), and calculate the average number of comparisons, and average running time, over 10 runs. Fill the average number (of comparisons over 10 runs) in the first column of Table 2, and fill the average running time (of sorting the array over 10 runs) in the first column of Table 3;
(6) Repeat step (5) for *n* = 400, 800, 1000 and 2000, and fill in the rest columns of Table 2 and Table 3, respectively (Note: For comparison purpose, make sure you use the exact same array data for all 6 sorting algorithms/codes).

Q4: Sequencing array sorting algorithms based on their complexities

Based on the tasks completed in Q2 and Q3,
(1) Compare data from Table 1, Table 2 and Table 3, and make a summary/comment/s on whatever you find from the algorithm analysis activity (i.e., the relationship between theoretical complexity analysis and that by experimental studies);
(2) Put these sorting algorithms (i.e., *Bubble, Selection*, *Insertion*, *Merge*, *Quick*, and *BST-based* sort) in a sequence based on their complexities (e.g., the algorithm with the best complexity is put to the first place and that with highest complexity to the last).

If you were given a task that needs to sort an array, which of the above algorithm/strategy may you prefer to do the sorting? Explain your reason/s.

Q5: ***Linked-list Programming:*** SLL deletion and reverse operations

A unit_list class is to be given in Workshop 05. The class uses an SLL to represent a list of student assessment results of a unit. Each SLL node contains a record of one student's assessment results. That is, it stores a student ID followed by the marks of three assessment components (A1_mark, A2_mark, and exam_mark), all are in integers. For convenience, student information stored in the SLL is in ascending order by the student_ID field. The class given shows the technique of traversing an SLL, searching an SLL and inserting a node into the SLL.

Your Task:

Starting from the unit_list class given, you are requested to expand the class by implementing the following two Python functions, or Java methods:

(1) Deletion of a student's record:

When being given a student ID (as input), the function/method searches the SLL to see if the student with that ID existed in the SLL or not. If it exists, the function/method deletes the student record (i.e., the node in the SLL) and keeps all other student information in the SLL unchanged.

Note that the method requires no return (i.e., *void* return as in Java). As such, you should make sure that the first node of the SLL is not physically deleted from the SLL, even if it might be logically deleted (why?).

(2) Display/print the student information (ID, individual mark items, and *total*) in descending order on the Student ID field:

This method displays/prints all student information in descending order on student ID field. Please note that, to keep the unit_class work properly you should not destroy the original SLL (why?). To do this, you may create a new SLL which is the reverse of the original SLL (or if you have to destroy the original SLL, make sure your recover it afterwards).

(3) Analyse the above methods using O-notation.

Q6: Binary tree traversal: Print leaf and non-leaf node and tree-depth

A Python/Java code is to be tested in TreeTest.py (or TreeTest.java) in Module 6. It prints the *Pre-order, In-order* and *Post-order* traversal sequences of a given binary tree.

Your Task:

Modify it so that it would:

a) print the *Pre-order, In-order* and *Post-order* traversal sequences of the given binary tree (*already implemented*);
b) print all leaf nodes (of the tree) only;
c) print non-leaf nodes (of the tree) only;
d) print the depth (also called *height*) of the tree.

Test your code by creating a more complicated BST, for example, using the following list of integers:

50, 76, 60, 30, 74, 18, 16, 98, 87, 40, 80, 46, 42, 43, 45, 41

==========

Indicative Marking Guide:

	Description	Allocated Marks (%)	Marks achieved & Comments
Q1~Q4	Q1: design, analyse and write code/s to implement sort algorithms	**20**	
	Q2: Theoretical summary of sort algorithm complexities	**5**	
	Q3: Algorithm analysis by experimental studies	**20**	
	Q4: Sequencing array sorting algorithms based on their complexities	**5**	
	Documentation	**5**	
	Over all	**5**	
Q5	Linked-list Programming Running? Outputs? & others	**15**	
Q6	Algorithm design & Analysis Algorithms / Pseudocodes for method b) method c) method d) Running? Outputs? & others	**15**	
Report	Presented as per format requirement? Submitted as per submission requirement?	**10**	
Total mark of 100 which is then converted to 20% of unit mark		**100 (/20)**	

The END of the Assignment Description