CSP2348 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:

- 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 are required to do not only algorithm complexity analysis (to part of the questions), but also an implementation using either Python or Java programming language. (*Note*: if you wish to use a programming language other than Python or Java, please let your tutor know beforehand, and you may be required to demonstrate your work in the end).

- 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. However if you really prefer to work individually, you may do so but no workload is to be reduced.

- Algorithm complexity analysis is one of the most important skills covered by this unit. It can be done in either or both of *theoretical algorithm analysis* and *experimental studies*. The theoretical analysis is generally conducted by applying asymptotic theory/methods, such as in O-notation, etc., to the algorithm's complexity function, which reflects the number of 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.

- The first four questions are closely related, requiring array-based searching and/or sorting algorithm design and/or implementation skills.

  Q1 requests you design and implement an array-based sorting algorithm using specific sorting strategy. Q2~Q4 are for theoretical analysis and experimental study for array-based sorting algorithms. Part of the questions also requires algorithm analysis using big-O notation.
  The 5th question requires you design and implement part of the application scenario using SLL to represent the required information. Part of the codes has been completed. You are requested to complete the rest.
  The 6th question requires you modify an existing algorithm and then convert it into Python (or Java) code to implement specific application scenarios using binary tree data structure.

Due: **Monday 27th May 2019 @ 9:00 am**
(i.e., last teaching week, see unit **Schedule**)

Value: **20%** (of unit mark)

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.
	Introduction 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 (in pseudo codes); (ii) algorithm analysis (if applicable); and, (iii) screen shots (of execution of the codes, 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 be attached in body of the report (they should be saved as separate runnable 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 10 pages (excluding references, snapshots and diagrams). 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 (or Java) 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 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 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 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 two sort algorithms:

(1) Bubble sort algorithm and analysis

Like *Insertion-sort* and *Selection-sort* algorithms, the *Bubble-sort* is one of the key elementary array-sorting algorithms. Its sorting strategy 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, do the following (assuming you are given an array of *n* integers):

- (i) Write the bubble sort algorithm (in pseudo code), which takes an array as input; Once this is done,
  - (a) analyse your algorithm using O-notation by two ways, i.e.,
    - by counting the number of comparisons; and
    - by counting the number copying operations;
  - (b) convert your algorithm to Python (or Java) code, and test your code using an array of some 20 elements.
- (ii) Observe that, after the scanning the array for *k* (*k* = 1, 2, …) times, the top-*k* greatest values will be in their correct positions (of the sorted array). As such, when scanning the array for the (*k*+1)th time, it is not necessary to scan the full array (but a sub-array of the first *n-k* elements only). This way the total number of comparisons could be reduced. Based on this observation, modify your algorithm completed in step (i), and re-do (i)(a) and (i) (b).

(2) Heap sort algorithm and analysis

For a given array, A[0, …, *n*-1] of *n* integers (or any other comparable elements), the pseudocode of *Heap sort* algorithm consists of two steps:

```
a) Build a heap based on the elements in A[ ] ;
b) for i = n-1 down to 1 ~~2~~
   { swap the element at root with the element in position i;
     restore the heap property for sub-array[0, …, i-1];
   }
```

You are required to

- Re-write pseudocode of the *heap sort* algorithm by refining the statement "restore the heap property for sub-array[0, …, i-1]" in the above pseudocode;
- Analyse your algorithm using O-notation.
- 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 the one 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	$\sim n^2/2$	$\sim 2n$	$O(n^2)$	
Insertion	$\sim n^2/4$	$\sim n^2/4$	$O(n^2)$	
Merge	$\sim n \log_2 n$	$\sim 2n \log_2 n$	$O(n \log_2 n)$	
Quick	*BC : $\sim n \log_2 n$ *WC: $\sim n^2/2$	*BC :$\sim 2n/3 \log_2 n$ *WC: 0	*BC : $O(n \log_2 n)$ *WC: $O(n^2)$	*BC: $O(\log_2 n)$ *WC: $O(n)$
Heap				

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

Q3: Algorithm analysis by *experimental studies*

Produce Python (or Java) code/s to complete the following task/s:

For a given *n* value (e.g., for *n* = 200, 400, 800, 1000, 2000, respectively),

(a) randomly generate an array, *A*, of size *n;* 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					
Heap					

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					
Heap					

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

(1) Write Python (or 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) Heap 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 to 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 (or 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 (or 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 (4) & (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).

<u>Q4</u>: 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 (e.g., 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 (or running time) is put to the first place and that with worst complexity (or running time) 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*

Refer to lab practice code in Module 5. The *unit_list* class uses an SLL to represent a list of a 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).
For simplicity, you may assume that all marks be in integer.
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,
(1) expand the class by implementing the following two Python functions (or Java methods):

a) Create/generate a 2nd SLL that holds the same amount of student information as the original SLL, but the information stored in the new SLL is in the *reverse* order of the original SLL (i.e., in descending order by the student_ID); and
based on the student information stored in the new SLL, print student's results, in format of
(ID, A1_mark, A2_mark, exam_mark, *total_mark*, *Grade*).
where
total_mark = A1_mark + A2_mark + exam_mark,
and
Grade is calculated based on ECU's grading criteria:

• total_mark >=80:	HD
• 70 <= total_mark <80:	D
• 60 <= total_mark <70:	CR
• 50 <= total_mark <60:	C
• total_mark <50:	N

b) Deletion of a student's record:

When being given a student ID (as input), the function/method searches the SLL and deletes the student record if it exists in the SLL, while keeping all other student information in the SLL unchanged.

(2) analyse the above methods using O-notation.

Q6: Binary tree traversal Application

Refer to the Lab code/s in Module 06. It prints the *Pre-order, In-order* and *Post-order* traversal sequences of a given binary search tree.

Your Task:

Modify the code so that it would:
a) Accept a list of integers (as input), and insert them, one by one, into an (initially) empty BST;
b) Print the *Pre-order, In-order* and *Post-order* traversal sequences of the given binary tree (*already implemented*);

c) Print all leaf nodes (of the tree) only;
d) Print non-leaf nodes (of the tree) only;
e) Prompt user enter an integer, then search the integer from the BST: if found, print the depth of that node in the BST otherwise output "ERROR: not found!"
f) Calculate and print the depth (also called *height*) of the BST.

Requirement: implement b), c), d), e) and f) in separate methods, and then call them in main method.

Test your code using the following list of integers:
50, 76, 60, 15, 30, 74, 18, 9, 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 the sort algorithms	**20**	
	Q2: Theoretical summary of sort algorithm complexities	**5**	
	Q3: Algorithm analysis by experimental studies	**25**	
	Q4: Sequence array sorting algorithms based on their complexities	**5**	
	Over all (Q1 ~ Q4)	**5**	
Q5	Linked-list Programming	**15**	
	Running? Outputs? & others		
Q6	Algorithm design & Analysis; Algorithms implementation for method b), c), d) e) & f); Running? Outputs? & others	**15**	
Report	Presented as per format requirement?	**10**	
	Submitted as per submission requirement? etc.		
Total mark of 100 which is then converted to 20% of unit mark		**100 (/20)**	

<u>The END of the Assignment Description</u>