COMP3506/7505 Project

Due: 23rd September 2022

Report Template

	Full Name	Student ID
Details	**Jamie Katsamatsas**	**s4674720**

1. Overview

This document is the ***mandatory*** template which must be used to submit the report section of your project.

This template is automatically synced with Gradescope to identify the location of each section of the report; therefore, it is imperative that the overall format/layout of this document not be modified. Modification of the template **will** result in a penalty being applied.

You are permitted to make changes inside the purple boxes for each question provided however the overall size of the box cannot change. Your report should easily fit within the boxes provided however please be aware the minimum font size allowed is Arial 10pt. If you are exceeding the box size, then this may be a good indication your response is not succinct.

2. Submission

Once you have completed your report this document must be exported as a pdf and uploaded to the Gradescope Report Portal for Part B of this assignment. This document **should not** be uploaded to the autograder.

3. Marking

The report will be hand marked by the teaching team. Information regarding the rubrics used while marking will be made available after grades are released. While this report will indicate the relative weighting of each section of the report, should there be any discrepancy with the official assignment specification, the assignment specification shall take precedent.

4. Plagiarism

The University has strict policies regarding plagiarism. Penalties for engaging in unacceptable behaviour can range from cash fines or loss of grades in a course, through to expulsion from UQ. You are required to read and understand the policies on academic integrity and plagiarism in the course profile (Section 6.1).

If you have any questions regarding acceptable level of collaboration with your peers, please see either the lecturer or your tutor for guidance. Remember that ignorance is not a defence!

5. Task

You are required to complete all sections of this report in line with the programming tasks completed in the project.

Hospital Appointment System (16 Marks)

Hospital 1

1. State the data structure used to store patients and explain why this data structure is the best for the task in hand.

A sparse array is used to store the patients, where every possible 20 min time slot in the day has a unique index in the array (size 30). This is the best for the task at hand as it allows for each 20 min appointment time slot to map to an index allowing fast insertion of the patients. An array has been selected since the problem only requires a limited number of appointments to be possible per day, meaning that once the array has been initialised there is no need to grow the array. Also an array is a good data structure to hold a sequence of items, in this case it is the patients appointments.

2. Describe the algorithm used to order the patients. Briefly explain how it works, from where it is called (from iterator/__iter__ or from addPatient/add_patient) and why it is the best algorithm in comparison with the other known algorithms.

In general the algorithm I have used involves inserting patients into a sparse array at an index that is calculated by their time, thereby maintaining order of the patients time in the array.

When addPatient is called the first bit of logic checks the time on the given patient to check if it lies within the allowed times. This is done by comparing the patient time string with the hardcoded allowed times for the hospital. If the time provided does not fall within “08:00” to “11:40” or “13:00” to “17:40” addPatient returns false. A similar time check is done for the other two hospitals, the only difference is the hardcoded allowed times.

If the time check is successful, then the time string is converted to an index for the appointments array by converting the appointment time and the first hospital time to minutes of the day and dividing the difference by 20mins (seen in timeToIndex()). This results in a possible range of 30 with a time of “08:00” mapping to index 0.

This calculated index is then used to index into the appointments array of size 30. The check for if the time is available is done by checking if the array has a patient in the index their time maps to. If the index contains a null value then you can insert the patient and return true. If the index contains a patient, return false indicating addPatient() failed.

When an iterator is constructed the current index of the iterator is set to 0 which is used to keep track of the iterator location in the array.hasNext() will iterate through the list starting at the currentIndex saved in the iterator up to the number of possible time slots that is saved in the hospital. If hasNext() finds a location in the appointments array that contains a Patient object (not null) it returns true and currentIndex is set to the index of the found patient, else false. Call to next() will return the patient at the currentIndex in the appointments array.

This implementation is preferred over a linked list and dynamic array implementation as it is a simpler implementation and is suitable for when we know we have a fixed number of possible times that won’t grow. Since the amount of possible times is small (30) a small fixed array is a simple suitable algorithm for this hospital.

3. Assuming n to be the number of patients, state the best-case (Ω) and worst-case (O) time complexity of iterator/__iter__ and addPatient/add_patient with respect to n.

addPatient: the best-case and worst case time complexity will be O(1) since all operations in addPatient are constant time:

- Check if time is valid = O(1)
- Check if the appointments array contains a Patient or null at the calculated index = O(1)
- Assign the patient to the appointments array = O(1)

I believe the memory complexity of my solution is also O(1) since for this scenario we know there is a fixed amount of time slots in the day (30) and since only one patient can be in a time slot we will never grow the array. So no matter how large n is the array size will always be 30.

Iterator: I believe the best and worst case time for the iterator is also O(1) as the time for the iterator won't change with respect to n as it will be a constant amount of indexes the iterator has to go over no matter what n is. The appointments list has been initialised to hold all the possible 20min appointments for the day, and the iterator will need to iterate over all the indexes of the appointments array (30 total 20min time slots) regardless of the number of booked appointments.

Hospital 2

1. State the data structure used to store patients and explain why this data structure is the best for the task in hand.

An array has been selected to hold the patients of the hospital 2. An array will allow storage of the patients in a linear sequence that will maintain the order of the time slots. For this task where there may be multiple people for the same time slot an array will be able to hold the patients of the same slot next to each other. An array will allow for fast memory access of the indexes and can grow dynamically when the number of patients increases.

2. Describe the algorithm used to order the patients. Briefly explain how it works, from where it is called (from iterator/__iter__ or from addPatient/add_patient) and why it is the best algorithm in comparison with the other known algorithms.

In general, this algorithm works by inserting patients at the end of the array and then swapping the patients to the left of the new patient until you get to the spot where the patient to the left has a time less than or equal to the new patients. Which maintains order of the patients.

In addPatient() first a check for valid time is done similar to hospital 1. If the time is invalid then then addPatient returns false, else it continues.

Next the number of appointments is checked against the array length. If the appointments array is full then the doubling strategy is applied to resize the array. When the array size is doubled all the elements from the original array are copied over to the new larger array before the addPatient() can continue. The patient is then added to into the next available empty slot of the array so that the array is filled up from the 0th index up to n (number of appointments).

Once the patient is inserted at the next available slot in the array, compare the new patient time with the patient to its left, if the patient to the left has a larger time you swap the indexes of the patients. You continue to compare and swap the patients until you find a patient to the left that has a time either smaller or equal to the patient to the right. This method will maintain order of the patients in the array and maintain order of the patients that have booked for the same time slot.

Since the appointments array is filled from the beginning to the end and the order of patients is maintained the iterator will simply start iterating at the beginning of the array and stop once it has iterated over the number of patients booked and finds a null.

hasNext() checks if the currentIndex of the iterator is less than the numAppointments booked in the system and return true if this is the case and false otherwise. next() will return the patient at appointments[currentIndex] and currentIndex is incremented.

An array implementation is better in this scenario than a linked list since it is simpler to implement than a linked list. A hash table is not suitable since it will provide us a slower iterator than the array method we have implemented as a hash table would be sparse.

3. Assuming n to be the number of patients, state the best-case (Ω) and worst-case (O) time complexity of iterator/__iter__ and addPatient/add_patient with respect to n.

addPatient:

- Valid time check is O(1)
- Check if the array is large enough is O(1)
- Amortised time when growing using the doubling strategy for array resizing is O(1)
- Insertion of the patient at the next available slot is O(1)
- Comparing and swapping the new Patient with the Patients to its left is O(n)

Therefore the best and worst case time complexity is O(n) since the addPatient will complete the same tasks on every call regardless of n.
Iterator:

- hasNext(): check if the currentIndex < number of appointments is O(1)
- next(): returning the next patient in the array if hasNext returns true is O(1)

The iterator best and worst case time complexity is O(n) as it will have to perform the above tasks regardless of n for all values of n.
Memory complexity of this solution is O(n) as the worst case amount of memory used is O(2n)

Hospital 3

1. State the data structure used to store patients and explain why this data structure is the best for the task in hand.

A doubly linked list is selected for hospital 3. The task requires insertion of O(1) and this is enabled by allowing insertion of the patient at the end of the linked list. Linked list has been selected over an array as O(1) insertion was a requirement and although this could be technically possible in an array by using amortised doubling array growth but this is less efficient than using a linked list. Also linked list allows for items to be stored in a linear fashion as is required when we are storing the patients.

2. Describe the algorithm used to order the patients. Briefly explain how it works, from where it is called (from iterator/__iter__ or from addPatient/add_patient) and why it is the best algorithm in comparison with the other known algorithms.

In general this algorithm works by adding patients as nodes to the tail of the doubly linked list that holds all patients. When the iterator is created the linked list is sorted with merge sort and the iterator iterates over the list starting at the head and ending at the tail.

When a patient is added to the system the time is checked for validity similar to hospital 2 and 1.
A new node containing the patient data is created for the doubly linked list. And the patient is then inserted into the linked list by adding the new node to the tail.

When patients are inserted into the linked list they are added to the end meaning that the linked list is not sorted and must then be sorted in the iterator.

When an iterator is created the first thing to occur is mergeSort is called on the doubly linked list containing the patients. Merge sort is selected as it is a stable sort and will maintain the order of the patients that were inserted with the same appointment times.

In the iterator hasNext() will keep track of the current node and check if the current node in the linked list is a null. If the current node is not null, true is returned, otherwise false.

Next() will change the current node of the iterator to be the next node and the patient of the node before the newly set current node is returned.

A doubly linked list that is sorted by merge sort is the best algorithm as it allows for O(1) insertion with minimal memory used in comparison with an array implementation. An array implementation would result in more memory used as the array would be doubling in size when it is full. Additionally when considering insertion a linked list implementation results in O(1) insertion as all you have to do is insert the new patient to the tail which is stored as part of the doubly linked list. Where in an array implementation you would have amortised O(1) insertion when considering doubling the array size on growth, this could be considered slightly less efficient than the O(1) insertion of the linked list. A hash table implementation would result in much more memory and a longer iterator than the linked list implementation.

3. Assuming n to be the number of patients, state the best-case (Ω) and worst-case (O) time complexity of iterator/__iter__ and addPatient/add_patient with respect to n.

For addPatient the best and worst case time complexity is O(1)

- Check of valid time is O(1)
- Add patient to tail of doubly linked list is O(1)

Since the above task are always executed regardless of n addPatient is O(1)

For iterator the best and worst case time complexity is O(nlogn)

- When an iterator is created merge sort is called on the linked list O(nlogn)
- hasNext checks if the current node is null which is O(1)
- next() returns the next node which is O(1)

Since the above tasks are always executed regardless of n, this makes the iterator worst and best case times run in O(nlogn).

The memory complexity of the algorithm will be O(n) as the memory complexity of the list is O(n) and the memory complexity for merge sort of a linked list is O(logn)

Login System (14 Marks)

1. What is the advantage of storing a hash code of the password instead of the plain text password?

Storing the hash of a password increases the security of the system. In a real login system the passwords would be required to be hashed as a security measure. If the system is compromised and an attacker gains access to the data stored they would only have the emails of the users and the hashed passwords.

2. Do we need to store the email in the hash table? If your answer is yes, explain why it is necessary. If your answer is no, explain how you use the email (if you use it at all).

Yes it is necessary to store the email in the hashtable. The email is required for when the hash table is resized. Once a hash table is resized the hashes of all the emails in the original hash table are recalculated in order to find the new locations of the existing emails in the now larger hash table. If you did not store the email in the hash table when you resized the hash table all those original emails would then be in the incorrect position, and you would have no way of recalculating the hash to find the new spots as hashing functions are one way.

3. Which of the following is an example of a collision? Explain your answer for both cases.

   (a) Two users have the same email hash

   (b) Two users have the same password hash

a) If two users have the same email hash this will cause a collision as the email hash is used to find the location of the (email, hashed password) tuple in the hash table. Two users having the same email hash will mean that when the second email is inserted into the hash table it will first land on a spot that is already occupied by the original email. Then you will be required to linear probe in order to find the next available slot to put the second email that had the collision.
b) Two users having the same password hash will not cause a collision as the password hash is not used to find the location of the (email, password hash) tuple in the hash table. However, it may cause some security issues as this means that the users could log into each others accounts as their passwords create the same hash and they could use the other persons email with their password and the system would accept this.

4. What is the type of hash code function being used? Explain why it is suitable for use in this hash table.

The type of hash code function used is polynomial accumulation. The key is an email string and this fits with polynomial accumulation as you can split the email string up into char values and use the ASCII value for each char in the calculation. Polynomial accumulation will help with minimising the chance of collisions occurring which will maximise the insertion time of our algorithm. Polynomial accumulation will also reduce collisions between similar strings.

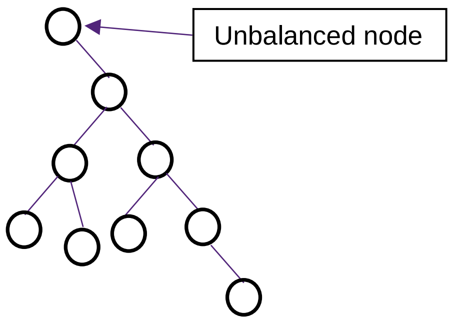
Tree of Symptoms (10 Marks)

1. What is the type of the restructured tree?

Binary search tree. All symptoms of less severity are to the left of a node and all symptoms of higher severity are to the right of a node. This is the same condition as a binary search tree where you must have nodes to the left having a smaller value and the nodes to the right having a larger value.

2. For any given binary tree, is the reconstructed tree **balanced**? If so, explain why. If not, give a counter-example.

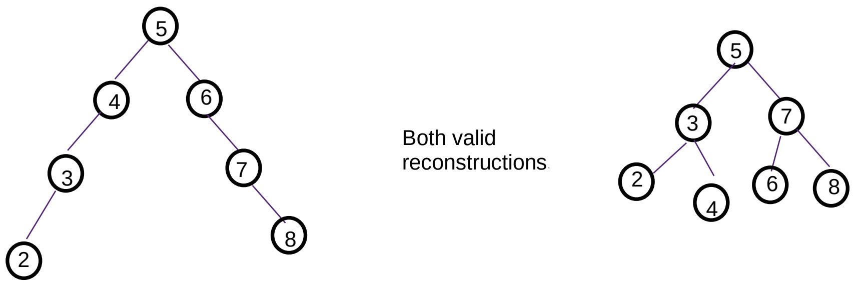
No the reconstructed tree is not balanced. A counter example is you are given a tree that contains the following nodes [1, 2, 3, 4, 5, 6, 7, 8, 9]. You can then reconstruct the tree on the node with severity 1 and when you reconstruct the tree in order to satisfy the conditions you must have all nodes with higher severity to the right. This means that every other node will be to the right with a minimum height of logn + 1 = 4 and the right will have a height of 1 which is unbalanced.



3. For any given binary tree, is the reconstructed tree **unique**? If so, explain why. If not, give a counter-example.

ASSUMPTION: reconstructed tree can be created anyway you want as long as it satisfies the rules of the task.

No the reconstructed tree is not unique. Since the only conditions on the reconstructed tree are that the left side must have severity levels smaller and the right side must contain severity levels higher, this means you have the freedom to construct the left and right trees how you wish. For example from the root you may construct the left tree as a long chain of nodes all linked like a list and you can do the same for the right side. Or you could construct the left and right trees to have the smallest height as possible. Both these trees would still be valid reconstructions.



END OF REPORT

ALIGNMENT TEST BOX
DO NOT EDIT