COMP3506/7505 Project

Due: 23rd September 2022

Report Template

	Full Name	Student ID
Details	**Hugo Burton**	**s4698512 (46985123)**

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.

Hospital 1 features a constraint where at most one patient can be booked in at the same time slot. Intuitively, this guides the use of an array – a data structure where one element can be placed at each index.

An array data structure was chosen here as it allows for constant time random lookups. This is very useful when calling the addPatient function since where we add a patient will be in a random spot of the array each time. The array data structure also has simple methods and therefore, is far less complex in its implementation.

One disadvantage I will mention with using arrays is that we don't know the number of appointments before runtime. While we do know there can only be one appointment per 20 minutes between 8am and 5pm, excluding 12pm to 1pm, it would not be a good idea to allocate memory for all of these possible appointment slots beforehand as this is wasted memory. Instead, the initial array is of size 1 and this is doubled when the array is full. That is the array jumps size 1, 2, 4, 8, 16, …, 128 etc. More about this in (2).

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.

The addPatient function can be separated into two distinct sections with the first being managing the array size. Since array sizes are fixed at runtime, and we don't want to allocate memory for all possible appointments (as discussed in (1)), we begin the program with a small appointments array of size 1 and double it each time it is full. A class variable stores the number of appointments currently in the array, named appts_load. When this is one less than the length of the current appointments array, a new array is created with double the length of the existing one. Existing appointments are then copied into the new array at the same index (maintaining the sorted property discussed below).

In comparison to using a linked list, this does take slightly more time for the average case, however, it also requires less memory since we don't need to store pointers to the previous and next patients, as a linked list would require. Here, it is a trade off between time and memory complexity.

The second part of the addPatient function handles inserting appointments at the correct index to maintain a sorted property by increasing appointment time. We start from the last appointment in the array and iterate downwards. Each existing patient is compared with the new patient we are trying to insert. If a duplicate is found, an error is returned as per the spec, however, if the new patient is found to be greater than an existing patient, we know their appointment should succeed that patient. That is, if we are at index j, the new patient should be inserted at index j + 1. But first we must store the patient already at index j + 1. By iterating from j + 2 upwards, we then shift each appointment along one index. We know here we won't overflow the array since its size was checked prior to adding the new Patient.

Secondly, we will discuss the iterator method, which is very simple due to the data structure chosen and its property of always being sorted. Within the iterator class, an integer index which stores the current patient we are up to is stored. Upon calling the interator next method, the index variable is incremented and the next patient in the array is accessed. Since patients are stored in ascending order of their appointment time, no additional calculation is required upon calling the iterator method. This allows it to run in constant O(1) time. More details in (3).

Constant time is the fastest any iterator function can run, hence this implementation of the iterator method is the best in comparison with other possible algorithms.

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.

Add Patient

The best case for adding patients will occur when patients' appointments are added in ascending order by time of appointment. It will achieve $O(1)$ constant time complexity because the elements of the array never need to be shifted to maintain the ordering. I will note, when the array is full, its size does need to be doubled. As a result, the best case for addPatient is actually only **amortised $\boldsymbol{\Omega(1)}$ time**.

The worst case for adding patients will occur when patients' appointments are added in descending order by time of appointment. Here, patients will need to be shifted along one spot for each addPatient method call resulting in an $O(n)$ time complexity. Here, the array still needs to be doubled in size when full, and hence we say in the worst case, addPatient runs in **amortised $\boldsymbol{O(n)}$ time**.

Iterator

As the array is already sorted, the iterator function will run in constant time, $O(1)$, (independent of n) in the worst case and best case. It's algorithm is discussed in (2). We can also say it has a big-θ bound of $\boldsymbol{\theta(1)}$.

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 data structure was also chosen for Hospital 2 as it features one main advantage over a linked list – random access. As the constraint for addPatient was constrained to linear time (O(n)), with an array, it is possible maintain a sorted property at all times.

An array was used over a hash map-like data structure as it allows for its elements to contain duplicates but remain in a stable order. For example, with a hash map, if two patients booked the same time, this can be dealt with through collision handling, however, it also increases time complexity. Whereas with an array data structure, if a patient is being added at the same time as an existing patient, they are simply inserted in the next index position. This property reduces the complexity for handling collisions/duplicates while maintaining a stable order. Here, by stable order, I mean priority is always given to existing patients in the case of a duplicate.

Note: I am aware hash maps aren't allowed for this question, but I'm referring to if one were to implement their own hash function (similar to the LoginSystem question)

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.

The algorithm to order the patients is handled in the addPatient method. Upon adding a patient, they are first checked for a valid appointment time (i.e. within hospital hours). Then, similarly to Hospital 1, the array is doubled in size if is full. The algorithm iterates from the end of the array downwards and inserts the new patient at the index after the appointment right before it. In the case of a duplicate appointment time, the new patient is inserted at the index immediately succeeding the existing appointment to ensure a stable priority ordering. For any appointments from the inserted index upwards, they are shifted along one position similarly to hospital 1. As we checked the size of the array before operating on it, we can be sure the array will never overflow.

By doubling the size of the array when full, rather than incrementing its size by one, the amortised time complexity for handling the array can remain at $O(1)$. This then allows for the ordering to occur at the add patient step while still achieving a linear time complexity. The ordering part of the addPatient method is the best for these conditions because it will always achieve a linear time complexity in the worst case, and average case. The algorithm also never requires auxiliary arrays to be allocated during sorting (as merge sort does for example).

By always maintaining a sorted property, it isn't required to check the entire array upon adding each Patient. Rather we just need to find where to insert the new patient and shift appointments after that along by 1 position. In a way, it's almost like insertion sort but without the need to traverse the array for each element in the array (i.e. $O(n^2)$).

As a result, the iterator method is very simple. A simple integer pointer to the current index is maintained and upon calling the next method, the value at the current index is returned, followed by the pointer being incremented. This can be classified as constant time: $O(1)$.

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.

Add Patient

Similarly to Hospital 1, the best case for adding patients will occur when patients' appointments are added in ascending order by time of appointment. It will achieve **amortised $\boldsymbol{O(1)}$** time complexity because we are adding in ascending order, though the array does still need to be doubled in size every 2^n calls.

The worst case for adding patients will occur when patients' appointments are added in descending order by time of appointment. Here, patients will need to be shifted along one spot for each addPatient method call resulting in an $O(n)$ time complexity. Here, the array still needs to be doubled in size when full, and hence we say in the worst case, addPatient runs in **amortised $\boldsymbol{O(n)}$ time**.

Iterator

As the array is already sorted, the iterator function will run in constant time, $O(1)$, (independent of n) in the worst case and best case. It's algorithm is discussed in (2). We can also say it has a big-θ bound of $\boldsymbol{\theta(1)}$**.**

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.

I implemented Hospital 3 using a linked list data structure. A custom class, called Appointments, was created and stored the patient for that appointment as well as a pointer to the next patient (i.e. a classic singly-linked list).

A linked list allows for $O(1)$ insertions, as the question requested, and since we aren't required to delete patients (i.e. where random access would be important), it makes the most sense to use this.

I will note, initially I implemented this question using an array and doubled the size of the array when it was full. However, I realised this only yields an amortised time complexity of $O(1)$, rather than an overall time complexity. However, in doing so, the sorting algorithm used in the iterator does become far more complicated here, as discussed in (2).

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.

The addPatient method is very simple in this implementation of Hospital 3. The class keeps track of pointers to the head and tail. In the base case where they both point to null, the head is updated to point to the first new patient, but never changed again. For all subsequent patients, the tail is updated to point to the most recently added patient. Additionally, the patient at the previous tail is updated so its next value points to the new tail. Here, we are creating a singly-linked list with pointers from each patient to the next. The tail patient will point to a null value, signifying the end of the linked-list.

The iterator method must then sort this linked list and it does so using a modified version of merge sort. Merge sort works by recursively splitting the array in halves until single elements are reached. It then merges these elements back together at each level of the binary tree (if we think about it as a tree). However, in a linked list, how can we possibly split it in half? This was achieved using two pointers with one iterating double the speed of the other. That is, pointer A would call its .next() method once per iteration, while pointer B would call .next().next() per iteration. When B reached a null value (i.e. the end of the linked list, we know pointer A will be halfway traversed in the list. We define variables middle and nextOfMiddle to hold these respective values and set the element after the middle to null so that the next layer of recursion knows that is the end of the sub-list (i.e. imitating a subarray) but without the need to allocate memory for additional arrays.

For each level of the merge step, the head of each sub-list is set to the minimum head of the two child lists. From here, each next pointer is assigned in ascending order of appointment times. Importantly, when duplicates times occur, the merge algorithm is sure to keep these in stable order to maintain priority for bookings made first. Finally, we are left with the same linked list, but with the head pointing to the first appointment and each next pointer pointing to the next patient in ascending order of their appointment time.

The merge sort algorithm adapted for linked lists is ideal here because it runs in $O(n \log n)$ time in all cases. Other than pointers to handle splitting the list, it doesn't require additional linked lists to be allocated during the sorting algorithm. Granted this implementation doesn't have a linear best case time complexities as some other algorithms do, however, this can be offset as it doesn't require additional memory to be allocated. It is also a quite complex implementation of the adapted merge sort, however, with some clever tricks, detailed above, it does pay off in terms of efficiency.

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.

As per the spec, the addPatient method will **always** run in constant, $\boldsymbol{O(1) = \Omega(1)}$ time because patients simple get appended to the tail of the linked list, using a class tail pointer which is constantly updated. This does, however, mean the sorting needs to happen in the iterator method

No matter the initial ordering of the linked list, the merge sort algorithm (adapted for my linked list data structure) needs to split the array into a left and right half until single elements are reached, as well as merge each subarray back together in the merge step. We can express the time function recursively as

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Where the $2T\left(\frac{n}{2}\right)$ represents the two halves of each array with $\frac{n}{2}$ elements and the $O(n)$ accounts for positioning each patient in the list as it is merged back together. We can see this will become

Hence the iterator method runs in $\boldsymbol{O(n \log n) = \Omega(n \log n)}$ **time in the best, worst and average cases**.

Login System (14 Marks)

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

Security: If the database gets hacked, it’s usually quite difficult for a hacker to reverse this (to gain the original password). Hence even if they had the hash, they won’t necessarily be able to log into users’ accounts.

The hash is also stored as a number which is a datatype easier to handle than strings (array of characters). It allows for users to have obscure characters in their passwords (so long as they have a corresponding ascii code) which allows for a more secure login than otherwise possible.

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).

In my implementation, storing the email in the hash table is required for two main reasons: resizing of the table and collisions. The email is still used as the index for the hash table, however, consider the case where the table is tripled in size. Then the compression function changes and so all existing email hashes need to be recalculated. Given this function is only possible in one direction, it is required the original email also be stored in the value of the hash table.

I achieved this by implementing my own data type for the value stored at each hash index, called Tuple, which consists of a String email as well as the int passwordHash. Here, the email is stored in plain text, however, the password is stored as a Hash (as alluded to in (1)) for security reasons.

Storing the email as part of the values is also important for collision handling. It is inevitable with enough users that collisions will occur, hence we cannot rely on the compressed hash of the email to be the only check made during the lookup. For example, consider the case where both users A and B’s compressed email hash is 1. However, linear probing has alleviated the collision by storing user A at index 1 and user B at index 2. Now consider the scenario when user A and user B happen to have the same password hash. If user B were to log in, and we only checked against the password, they would be logging in to user A’s account!

To prevent this, we must also check the plain email against what is stored in the database; only using the compressed email hash as a lookup to achieve a constant time complexity. While such an above scenario is unlikely, if we are linear probing for a long time before finding the correct user, it is conceivable two users might have the same password hash – even if they don’t have the same plain text password! Hence we must design login systems that are fail proof.

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) Two users with the same email will cause a collision. In fact, it is possible for two users with different emails to have the same hash (with approximately 1/M probability) due to the compression function. In the data structure, a custom hash map has been implemented which handles collisions using linear probing. When a user is added where another user already exists at that index, the algorithm will iterate from that hashed index until it finds a **null** or **sentinel** value. The new user is then inserted here. If the array is not full (i.e. contains null and/or sentinel values), we are guaranteed to find a place to add a user in all cases.

    When the array becomes full, upon inserting the next user, a new array triple the size is allocated and all existing users have their hash function recalculated. The new hash (with compression) will be different as we will need to find the modular value of the hash code with a new M value. This will ensure we can use the entire new array (not just the first third). The password hash, however, remains the same as

(b) Two users can have the same password hash without causing a collision. This is because the password is stored as the value for the hash table, rather than the key.

    In the case where two users have the same password, it still isn't possible for either user to log into the other user's account because the data structure also stores the email which can be cross-checked.

    It's worth noting that since the password hash doesn't use compression, two users would have to have the exact same password (in plain text) for their hashes to be the same.

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

In this implementation, a polynomial accumulation hash code function is used here. It is suitable here for many reasons. As we can be sure the strings being hashed (emails and passwords) are relatively short, we will never encounter an overflow (i.e. where the hash code is too big to be stored in a Java integer).

A polynomial accumulation type hash is also very simple to implement and calculate. Calculating a hash code for any given string only requires simple multiplication and addition. In my implementation, I calculated this value iteratively by first converting all characters to integers (ascii), then summing each integer multiplied by the **c** value to the power of the length of the array minus the character's position in the array minus 1. In mathematical notation, we could write this as

$$h(s_0, s_1, \dots, s_{n-1}) = \sum_{i=0}^{n-1} a_i \cdot c^{n-i-1}$$

For emails, $mod\ M$ of this value was then taken.

Due to the array being a fixed size, when it overflows, it is important for the the hash function to be simple to compute, given we would need to recalculate it for each email. This does, however, mean each email also needs to be stored in plain text so its compressed hash can be recalculated.

To avoid collisions for email hashes, a linear probing method was used here. Linear probing is quite a good fit here because it achieves expected constant O(1) lookup times for user's emails and passwords. In the case where there is a collision, and linear probing needs to occur, typically only a few indexes need to be checked before the user is found (or not found in the case of a null). In doing so, another advantage is no additional memory needs to be allocated.

With a login system, users need to be able to be deleted as well as added. In the case of a linear probing approach as we see here, simply marking a value of the hash table as SENTINEL is quite easy here and can be achieved in constant time. There is no need for pointers to be remapped as in a separate chaining approach.

Tree of Symptoms (10 Marks)

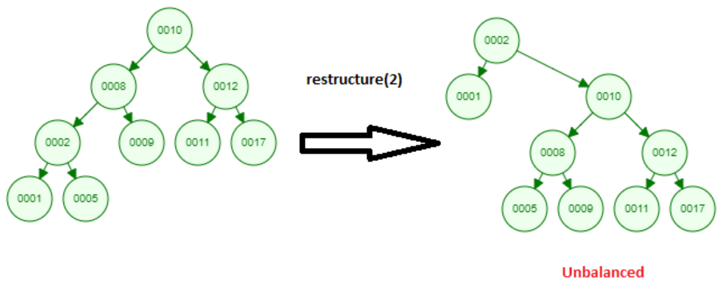
1. What is the type of the restructured tree?

When the tree has been successfully restructured, it can be named a **splay tree** because within the restructuring process, the node satisfying severity >= threshold is to be the root node of the tree (e.g. a frequently accessed node). Splay trees are a *subset of binary trees* with the additionally property that the root node is specified to improve search performance on the tree. A binary search tree is a binary tree where the left and right child of each node is less than and greater than its parent respectively. In this implementation, the spec also specifies no duplicates – that is, no symptoms can have the same level of severity.

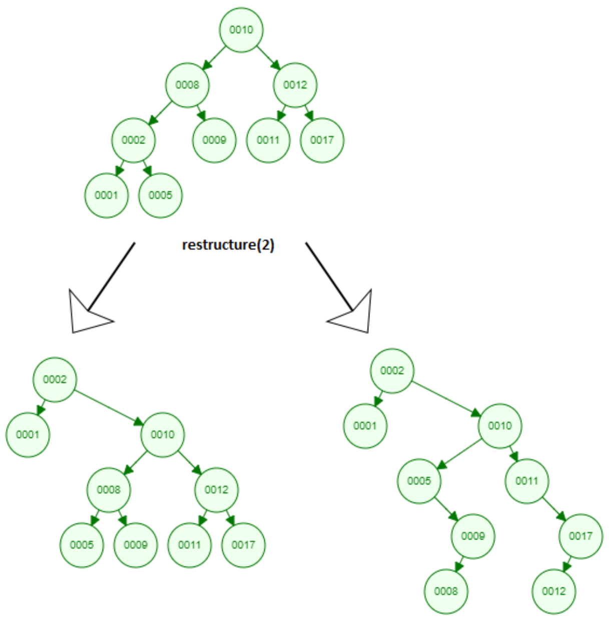
2. For any given binary tree, is the reconstructed tree **balanced**? If so, explain why. If not, give a counter-example.

For any given tree, the reconstructed tree is **not** necessarily balanced, although there may be some cases where it is. This is due to the root node being specified as an argument to the restructure method. The restructuring process will then ensure the node with the minimum severity above the specified threshold is placed as the root node of the tree.

As this node isn't necessarily the middle of the tree (i.e. the node that would allow the tree to be balanced), it is possible for the left and right descendants from the root to differ in height. For example, I have illustrated below an example tree restructured with severity 2. Note: the left tree needn't be a binary search tree, but it happens to be in this example. It can be seen in the restructured tree, the left subtree (root 1) and right subtree (root 10) are both balanced, however, the entire tree itself is not balanced. There is a height difference greater than one between node 1 and node 10. I will note, the spec doesn't specifically mention for the subtrees to be balanced, however, this is how it is implemented in my solution.



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



Following the specifications on the task sheet, it is possible to have multiple distinct reconstructed trees with the same root node from the same initial tree. For example, we could have. Notice the two trees at the bottom have the same in-order traversal, however, are clearly distinct.

While it is possible to have distinct restructured trees, it is worth noting in my implementation, this would not occur as restructuring always follows the same deterministic algorithm. Specifically, if the tree at the top was restructured with argument two, the output would be the tree in the bottom left, where each subtree (depth 1) is balanced. Hence, we could say, the restructured tree is unique if and only if it is balanced.

END OF REPORT

ALIGNMENT TEST BOX
DO NOT EDIT