**Question 1**

a) Consider that you are developing software for karaoke centers to host karaoke shows and playing karaoke songs. Besides displaying the latest new songs, the songs in the system are grouped by the gender of the singers, the name of the singer, language, region, musical genre, etc. The system can sort the songs according to the customer’s selection. For example, when a customer is selecting the songs based on title, the titles are sorted ascendingly; when a customer is selecting the songs based on singer, the singers are sorted according to their surnames; and so on. The system allows multiple customers to add songs to their own playlists and then they have to wait for their turn. For every customer, it will allow each person to sing their chosen songs for around 10 minutes. After that, the system will select the next user in line to sing. This will repeat until all songs are played, or the time is up. Any customer can remove any of their selected songs from the playlist at any time, and they can move the song to the front of the playlist as well. The system is able to sync the first 10 songs in the playlist to the customers’ smartphones in real time, so that they know when it is their turn even when they are outside the karaoke room.

Based on the case study above, analyze how list, stack and queue can be applied in the system. Explain with relevant examples.

(i) Propose **ONE (1)** application of the list in the karaoke system with an example. (3 marks)

(ii) Propose **ONE (1)** application of stack in the karaoke system with an example. (3 marks)

(iii) Propose **ONE (1)** application of queue in the karaoke system with an example. (3 marks)

b) An Abstract Data Type (ADT) named **RandomList** is used to generate random items in a list. Refer to the examples below to implement the **RandomString** Java class that outputs random items in a list.

| **/\*Sample main method\*/**  public class MainProject {    public static void main(String[] args){  String[]str = {"red","orange","blue","yellow","green"};  RandomList randString = new RandomString(str);  randString.createRandomList(3);  System.out.println("randString = " + **randString**); }  } |
| --- |

Figure 1.1: Sample main method to generate 3 random Strings

| randString = [green,yellow,yellow] |
| --- |

Figure 1.2: The output of the main method

(i) Write the **RandomList** Java Interface with the relevant abstract method. (3 marks)

| public interface RandomListInterface <T>{  public void createRandomList(int num);  } |
| --- |

**Question 1 b) (Continued)**

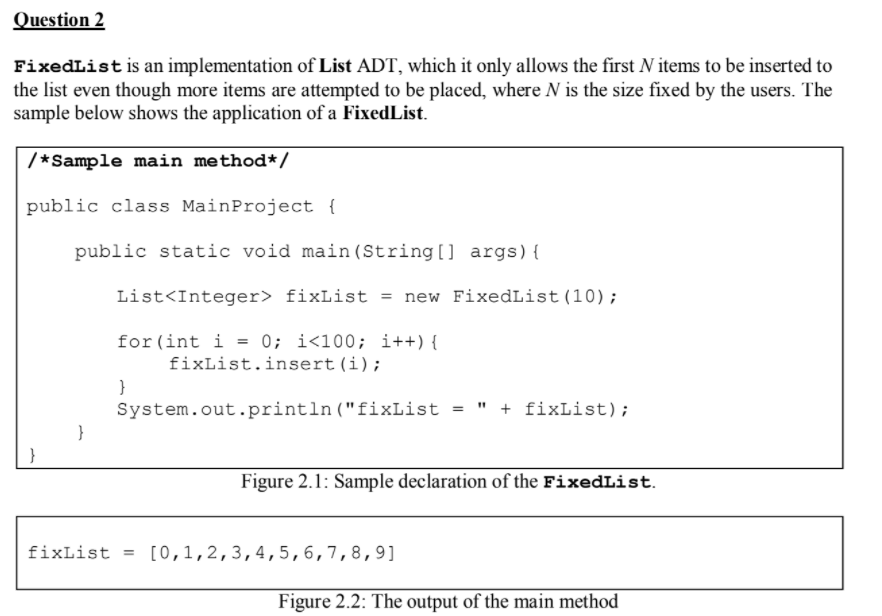
(ii) Complete the **RandomString** Java Class that implements the **RandomList** interface with the following requirements.

| **Requirement** | **Mark** |
| --- | --- |
| **RandomString** Class definition | 2 |
| Declaration of class attributes, which include a list (named **randList**) that stores the required random strings  *Remark: list implementation is not required* | 2 |
| A constructor that accepts a String array, which also initializes all the class attributes | 3 |
| The implementation of the abstract class method as given in Question 1b)(i).  *Remark: you may use any built-in methods or classes to generate the random numbers*. | 6 |

(13 marks)

[Total: 25 marks]

| import java.util.\*;  public class RandomString<T> implements RandomListInterface<T> {  private List<T> randList;  private String[] array;  public RandomString(T[] array) {  randList = new Arraylist<>();  this.array = array;  }  public void createRandomList(int n) {  for (int i = 0; i < n; ++i) {  int randNum = (int)(Math.random() \* 10) % array.length;  randList.add(array[randNum]);  }  public String toString() {  String str = "\n";  for (int i = 0; i < randList.size(); ++i) {  str += randList.get(i) + "\n";  }  return str;  }  } |
| --- |



a) One of the implementations of the **FixedList** is linked list. There are three common types of linked list, i.e. singly linked list, circular linked list and doubly linked list.

(i) Compare and contrast the strengths and limitations of the **THREE (3)** types of linked list if they are going to be applied in the **FixedList** implementation. (12 marks)

| 3 types of linked list implementations:  a. **Singly linked list**  - with a single external head reference  b. **Circular linked list**  - a single external tail reference  - always add at the back , remove from the front (/)  c. **Doubly linked list**  - with head and tail reference  **Compare (similarities) & contrast (differences) aspects**  - efficiency of the remove operations    - efficiency of the add operations    - complexity in implementation  The implementation of singly linked list will be more easy than the circular linked list and the doubly linked list because each of the nodes only need to point to the next and only need one reference point to the first node. For a circular linked list, it needed an external reference point to the last node, and the reference of the last node had to point back to the first node.(When added at the back, the external reference has to point to the new last node. While singly linked list no need to change to point to another node when added at the back. The external reference of a singly linked list only needs to point to the first node, less chances needed to change to point to another node.) For the doubly linked list, every node needed two references, one reference point to the previous node and one reference point to the last node. Doubly linked list also needed external head and tail reference. So the implementation of the doubly linked list is the most complex.  - memory requirements  Compare between singly linked list and doubly linked list, singly linked list will consume lesser memory than doubly linked list because every node only have one reference to the next node while doubly linked list every node have 2 reference which need to point to the previous node and also the next node. |
| --- |

**Circular linked list** (add the back, remove at the back) more efficient

(ii) Select the type of linked list that you think is most suitable to be applied in **FixedList** implementation. Justify your answer. (1 mark)

**Singly linked list.** Singly linked list is the easiest to be implemented, need fewer memory compared to doubly linked list

b) Write the **FixedList** Java Class that implements the **List** interface with the following requirements. You must implement it based on the type of linked list that you have selected in Question 2 a)(ii). You may assume that the **List** interface and the **Node** class have been defined.

| **Requirement** | **Mark** |
| --- | --- |
| **FixedList** Class definition | 1 |
| Declaration of class attributes | 2 |
| A constructor that **accepts the size fixed by the user** | 2 |
| The implementation of the insert method  Remark: You may ignore the toString method. | 7 |

| **Singly linked list**  public FixedList<T> implements List<T> {  private Node firstNode;  private int numberOfEntries;  private int maximumEntries;  public FixedList(int n) {  numberOfEntries = 0;  maximumEntries = n;  }  public void insert(T newEntry) {  if (numberOfEntries != maximumEntries) {  Node currentNode = firstNode;  while (currentNode.next != null)  currentNode = currentNode.next;  currentNode.next = new Node(newEntry);  numberOfEntries++;  }  } |
| --- |

**Question 3**

a) SpeedMovie is a technology and media services provider. It offers streaming service of a wide range of popular TV shows, movies, anime, documentaries and more on thousands of internet connected devices, such as smart TV, mobile device and personal computer. Their customers can watch more than one video at any time. They can also rate the shows that they have watched. Recently SpeedMovie approached your company to help them to improve the search function of their system with advanced features.

You are required to propose **THREE (3)** search functions. The requirements of the proposal of each search function are given below.

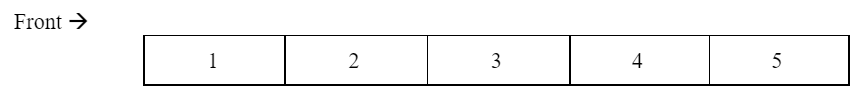
| **Search Function Details** | **Mark** |
| --- | --- |
| Proposed relevant search function with description/example | 2 |
| Proposed implementation of the search function (need to describe the appropriate data structures and search/sort algorithms) | 4 |
| Evaluate the search time efficiency of your proposed implementation using Big O notation. | 1 |
| Proposal of **THREE (3)** search functions | x 3 |

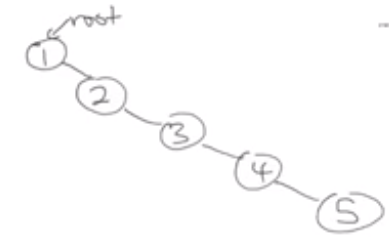
(21 marks)

| **ADT with big O notation**  Hashing O(1)  Binary search tree O(log2 n)  Sorted list O(n) |
| --- |

| **Search function** | **Implementation** | **Big O notation** |
| --- | --- | --- |
| **Search by movie title**  Users only need to enter the keyword.  Eg. Enter “Kingdom”, then the system will display all the movie which with the word of “Kingdom” | **ADT:** **Hash table (Dictionary)**  Users only will enter the movie title and the movie title will be stored as the search key to search and retrive the data. However, there might happen in which one search key contains multiple values so called collision, so **separate chains** need to be implemented to the hash table to store multiple values which belong to one search key location.  In conclusion, hashing will be used as a searching method for the hash table whereas the **sequential search** will be implemented in the separate chain.  ***Notes:*** *If the data in the separate chains is sorted list, binary search can be implemented as a searching method for the separate chains.* | O(1) |
| **Search by TV episodes** | **ADT: Sorted list (Array implementation)**  The search function allows the user to enter a word (part of the name or **title** of the episodes), all the related data that is stored in a **sorted list** from alphabetically A-Z will be searched by using **binary search** algorithms. This is because the episode will normally be sorted which starts from episode 1 until the last episode.  To illustrate, it will divide all the episodes in half and compare the target (episodes searched) with the element at the array’s midpoint. If the episodes searched are matching with the episodes at the middle of the array, it is found at the first time of search. Otherwise, if the value of episodes searched is less than the middle element, it will continue to search the left subarray; else it will continue to search the right subarray until the target episode is found or not found (the subarray of all the episodes is empty). | O(n) |
| **Search by director** | **ADT: Binary search tree**  Users can enter the director name. From the director in the hierarchy, it will divide them in half and use **recursive binary search** to locate the data. To illustrate, it will compare the root node of the data. If the target is not matched, it will continue to search the left root node if the comparison between target and root entry is less than 0 else it will search the right root node. | O(log2 n) |

b) The following array of numbers are being dequeued to build a binary search tree (BST). Identify and explain a potential problem of the resulting BST, and then suggest a solution.





* **Solution 1** : Using AVL tree, automatically balance itself
* **Solution 2** : Randomize the content/data before insert it into the binary tree
* **Solution 3 :** Insert the data start from the middle

**Question 4**

STAR Exam Scheduling System (SESS) is designed to schedule exam papers according to the available sessions, venues, and the number of students. Each exam paper contains a course code, course title, exam date, and a list of programme codes of the students who are taking this exam paper. Figure 4.1 shows part of the **ExamPaper** Java Class, and Figure 4.2 shows one sample data of the **ExamPaper**

object instance.

| public class ExamPaper {    String courseCode;  String courseTitle;  Date examDate;  List<String> programmeCode;  ...  } |
| --- |

Figure 4.1: **ExamPaper** Class

| courseCode | BACS2063 |
| --- | --- |
| courseTitle | Data Structures and Algorithms |
| examDate | 18 December 2020 |
| programmeCode | [RSD2, RIT2, RSF2, REI2, RST2, RIS2, RMM2] |

Figure 4.2: Sample **ExamPaper** instance data

A scheduling problem searches 2,000 exam papers. To increase the efficiency of the search algorithm, you are considering using a hash table to retrieve the exam paper object instances, and then sort the programmeCode.

a) A hash table is an unordered collection of key-value pairs, where each key is unique.

(i) Suggest **ONE (1)** suitable key for the hash table that allows the SESS to retrieve each **ExamPaper** efficiently. (1 mark)

Answer: courseCode

(ii) Based on the key you have identified in Question 4 a) (i), suggest and design an appropriate hash function to convert the selected key into an integer index. Express your answer in pseudocode or code. (4 marks)

| **Computing a hash function consists of 2 steps:**   1. **Get an integer hash code** to represent the object   -courseCode is the best because can get the unique integer (Character.charAt(i))     1. **Compress the hash code** to be within the table’s index range |
| --- |

(iii) One of the challenges of using a hash table is collisions. To avoid collisions, there are several solutions, such as linear probing, quadratic probing, double hashing, and separate chaining. Explain with examples the **THREE (3)** considerations that a developer needs to have when selecting one of these collision resolution methods. Then justify your recommended method for the SESS. (12 + 2 marks)

**Answer Guidelines**: 3 consideration (explain + examples) + Recommendation for collision resolution method + justification

| **Linear probing** | Advantage  -Simple to implement  -able to reach every single location  -utilize all space  Disadvantage  -primary clustering occur : all the object will be group together around the location that the collision occur, long chain  -when try to search the particular entry, it will become sequential search , lead to less efficient O(n)  \*table size is best to become prime number , can avoid collision |
| --- | --- |
| **Quadratic probing** | Advantages  -overcome primary clustering  -not like linear probing going to the next location, quadratic probing will take the h index to do the following calculation to change the probe sequence, for example:  Probe (index + 1^2), (index + 2^2), (index + 3^2), (index + 4^2)  (index + 1), (index + 4), (index + 9), (index + 16)  - The entry will be more distributed  Disadvantages  - cause secondary clustering: clustering occurs further apart , it may also need sequential search to search the object O(n) and lead to increase the search time due to needing more time to compute indices if there are multiple collisions for the same search key.  -may not reach every single of location due to the probe method that mentioned above, may end up revisiting a same location more than one , less efficient |
| **Double hashing** | -use a secondary hash function to increment the step . If collision occurs, will move to the increment step away to probe搜寻 for an available location or to locate the key that you are searching for.  For example :  Key =16  H1(key) = key % 7 = 2  H2(key) = 5 - key % 5 = 4  If location 2 is located by another object, need to go 4 steps away to probe for an available location.  Advantage  Overcome primary and secondary clustering problem  Disadvantages  - If the secondary hash function is complicated, it may take more time to do the computation of the secondary hash function  - Guarantee they can go through every single location in the hash table |
| **Separate chain** | Every location will have a bucket, bucket can be the array list , linked list and so on.  Advantages  -No need to do any primary hashing or double hashing, when new object come just need to add to the bucket, all the object which is hash to a same location will add to the bucket  Disadvantage  - Need external storage to store the bucket (array, linked list) |

Consideration

| **Table size respective to the no. of entries/range of hash indexes.** | If phone number start with 555-\_\_\_\_  0 - 9999  If have a table with 1000 size, it won’t be a big problem, linear probing and quadratic probing still able to apply, no need to worry about the collision  If table size is small, the range of hash number cannot be predicted , number of entries is quite high relative to the table size, then linear probing and quadratic probing are not suitable, double hashing will be more suitable. |
| --- | --- |
| **Efficiency** | Separate chaining no need to increment step away to probe搜寻 for an available location. If you apply the stack method for the bucket, it will be very efficient because every new entry only needs to add to the top of the bucket. But if need to do searching, it need to apply sequential search which lead to O(n)  Therefore, **adding a new entry** will be very efficient [**O(1)**] but **searching** will be less efficient [**O(n)**]. |
| **Memory utilization** | If have a large hash table, separate chaining will lead to a spare hash table, a lot of empty locations in the hash table and need additional memory to allocate the bucket (will be more worse when using array because need to double array may lead to overhead)  If the table size is small, it is good to apply a separate chain. |

In my opinion, separate chaining is the best choice since each of the course will have

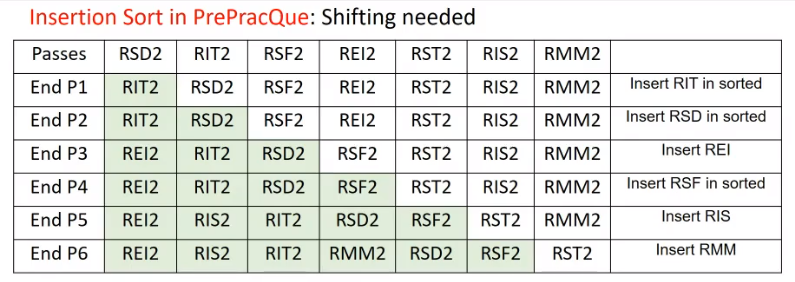
b) Figure 4.3 shows a list of programme codes. Demonstrate the steps of **Insertion Sort** when sorting the programme codes ascendingly. You are required to show the contents after each pass.

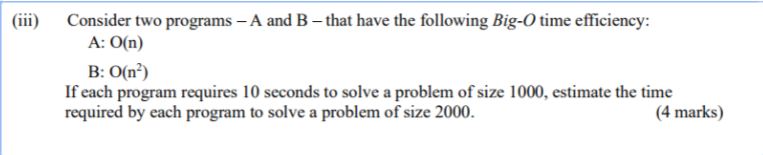
| RSD2 | RIT2 | RSF2 | REI2 | RST2 | RIS2 | RMM2 |
| --- | --- | --- | --- | --- | --- | --- |

Figure 4.3: Sample **programmeCode** data

(6 marks)

[Total: 25 marks]





Program A: **O(n)**

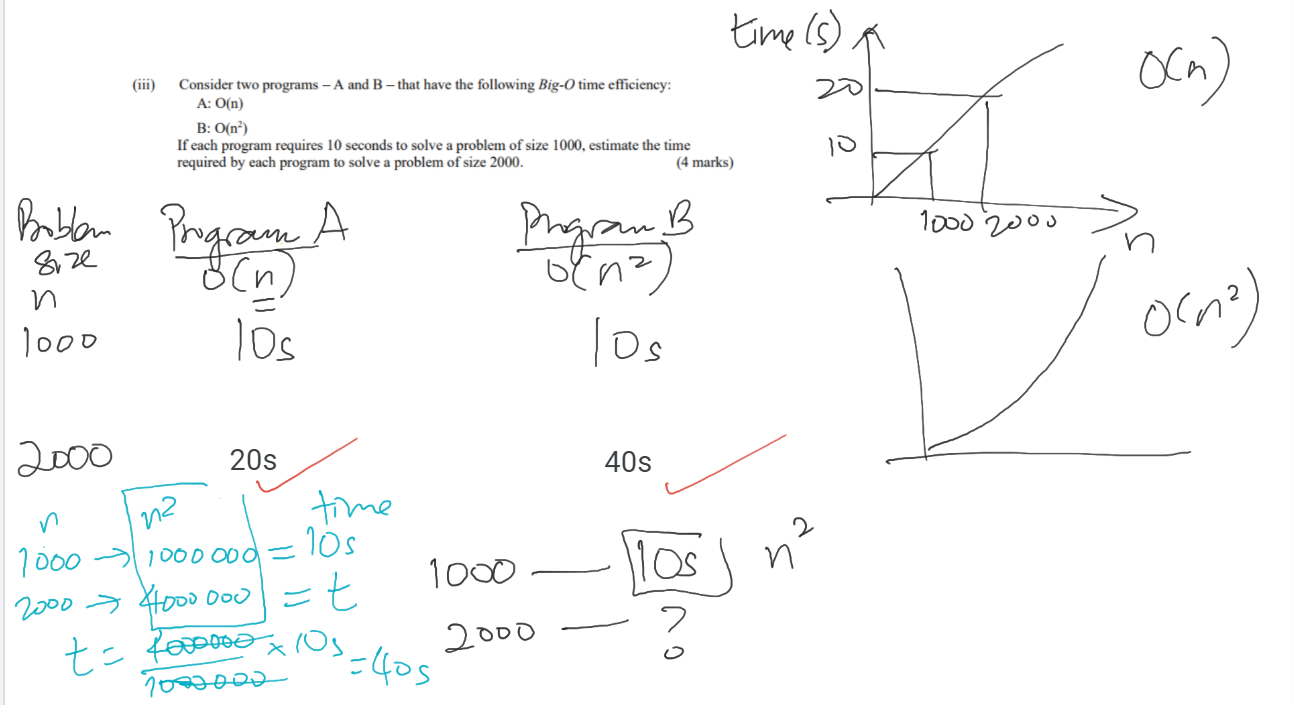
Size 1000 =10

Size 2000=20

Program B: **O(n2)**

Size 1000=10

Size 2000=40

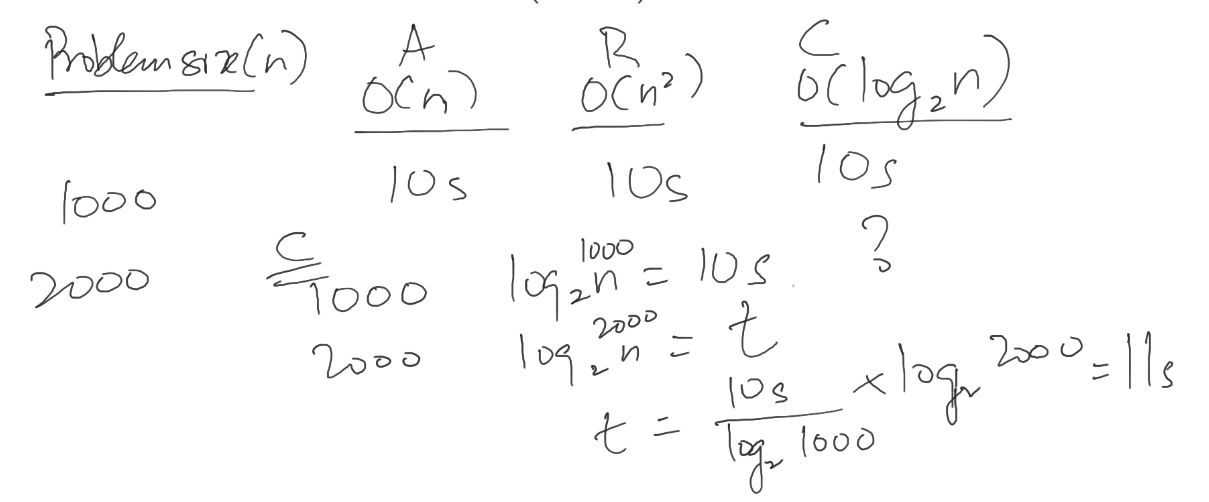


Program C: **O(log2 n)**

Size 1000 = log2 1000=10

Log2 2000 =?

10s/log2 1000 \* log2 2000 = 11s

Calculator can be used for checking