Data Structures and Algorithms: Assignment 1

Due: 6th April - 100 marks possible (20% of final grade)

When submitting, zip up your entire Netbeans project (**make sure you include a .jar file of Virus Simulation project**) into a folder with the following structure:

lastname_firstname_studentID.zip (using your own name and ID)

and submit to the Assignment 1 zipped folder on AUT Canvas BEFORE the due date and time. Late submissions will incur a penalty. Antiplagiarism software will be used on all assignment submissions, so make sure you submit **YOUR OWN** work and **DO NOT SHARE** your answer with others. Any breaches will be reported and will result in an automatic fail and possible removal from the course.

Question 1) Linked List and Bracket Evaluator (50%)

The purpose of this question is to utilise a linked list to create an application to efficiently evaluate whether opening and closing bracket and brace pairs match up inside any given string in O(*n*) time, where *n* is the length of the string. All other content in the string can be ignored.

Linked list must be built and you must use your linked list to implement the application.

Part 1 Linked list (28%)

Node Class (5%)

Create a Node Class which has data and linker parts. Data part stores any type of data and node can be linked to each other together by its linker part.

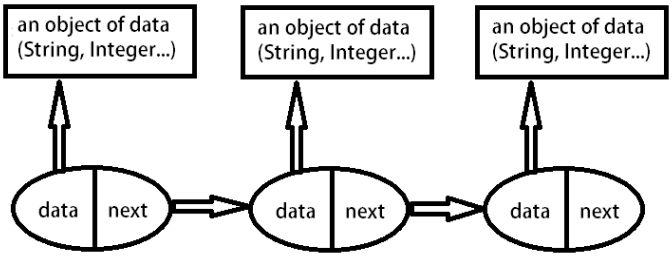
Node class has a generic object. The generic object points to any types of data (the data type may be String, Integer, Float or Character).

Node Class has a node object named "**next**" which points to a next node.

Node Class has an "**equals**" method. "equals" method takes a Node object in and returns true if the argument (node)'s data equals to this node's data. Otherwise, it returns false.

Node Class has an "**compareTo**" method. "compareTo" method takes a Node object in and returns an int 0 if the argument node's data equals to this node's data; an int value less than 0 returns if this node's data is numerically less than the argument node's data or this node's data is alphabetically less than the argument node's data. Otherwise, it returns an int value greater than 0.

Node <E>
+ data : E + next : Node
+ Node() + equals(Node node) : boolean + compareTo(Node node) : int



LinkedList Class (23%)

Create a LinkedList Class which builds and manages a linked list. **Extra 10 marks will be given if no loops are used.**

LinkedList Class has a Node object named "**head**" to point to the head of a linked list.

LinkedList Class has an int variable named "**size**" to store the size of a linked list (number of nodes).

LinkedList Class has an "**add**" method. It takes a generic object to create a node and add to the end of a linked list

LinkedList Class has an "**addInOrder**" method. It takes a generic object to create a node and add to a linked list following the ascending numerical order if the generic object is a number. It follows alphabetical order if the generic object is a char or String.

LinkedList Class has a "**contains**" method. It takes a node and returns true if the linked list contains this node. Otherwise, it returns false.

LinkedList Class has an "**getNode**" method. It takes an index of a node to be retrieved and returns the node.

LinkedList Class has an "**getData**" method. It takes an index of a node to be retrieved and returns the node's data.

LinkedList Class has a "**printLinkedList**"method. It prints all the contains of the linked list to the console.

LinkedList Class has a "**remove**" method. It takes a node as an argument and remove a node which has the same data as the argument node from the linked list.

LinkedList Class has a "**remove**" method. It takes an index of a node to be removed and removes the node.

LinkedList Class has a "**removeFromHead**" method. It removes the first node from the linked list.

LinkdedList Class has a "**removeFromTail**" method. It removes the last node from the linked list.

LinkedList <E>
+ size : int - head : Node
+LinkedList() + add(E data) : void + addInOrder(E data) : void + contains(Node node) : Boolean + getNode(int index): Node + getData(int index): E + printLinkedList(): void + remove(Node node) : void + remove(int index): void + removeFromHead() : Node + removeFromTail(): Node

Part 2 Queue and Stack(10%)

Queue Class (5%)

Queue class manages a linked list as a queue.

Queue class has an "**enqueue**" method. It takes a generic object and enqueues the object to a queue.

Queue class has a "**dequeue**" method. It dequeues and returns the node from a queue.

Queue class has a "**getSize**" method. It returns the size of the queue.

Queue class has a "**printQueue**" method. It prints the contains of the queue to the console.

Queue <E>
- queue : LinkedList<E>
+ Queue() + enqueue(E data) : void + dequeue() : E + getSize() : int + printQeue() : void

Stack Class (5%)

Stack class manager a linked list as a stack.

Stack class has an "**push**" method. It takes a generic object and push the object to a stack.

Stack class has a "**pop**" method. It pops and returns the node from a stack.

Stack class has a "**getSize**" method. It returns the size of the stack.

Stack class has a "**printStack**" method. It prints the contains of the stack to the console.

Stack <E>
- stack : LinkedList<E>
+ Stack() + push(E data) : void + pop() : E + getSize() : int + printStack() : void

Part 3 Bracket Evaluator (12%)

DataAnalysis Class

DataAnalysis class checks whether a list of data is symmetrical (you can use Queue and Stack Classes).

DataAnalysis class has an "**bracketEvaluator**" method. It returns true whether opening and closing bracket and brace pairs match up inside any given string.

DataAnalysis
- data : E [] - queue : Queue <E> - stack : Stack <E>
+ DataAnalysis(E[] data) + bracketEvaluator() : boolean

You can add more fields or methods if you need.

Question 2) Thread: a mobile phone virus transmission simulation (50%)

You need to design a virus transmission simulation. You are going to draw some icons instead of mobile phones in your simulation. The number of mobile phones can be increased during the simulation by clicking a key from a keyboard. Each phone runs as a thread.

A virus starts in a random phone, and it transmits to the others in the range of 20 pixels. Infected phone needs to take to a repair shop to get repaired. **Only one phone can be repaired at a time.** You need to design and write "synchronized" algorithm to handle it. If the phone cannot be repaired within 500 frames, the phone will die (the phone is removed from the simulation).

Design YOUR OWN GUI of the simulation.

- Phone's behaviours
    - You may use the linked list from question 1 to store all the phones or you may use array to store all the phones.
    - An "up" arrow key is used to increase the number of phones in the simulation.
    - Different colours for three different states of a phone (health phone, infected phone and moving to repair shop). You choose your own colour for the phones.
    - Press "v" key to set one phone to get virus randomly.
    - The virus transmits through network within a certain distance (20 pixels in this simulation)
    - After a phone get infection, simulation counts down the life span of the phone from 500 to 0. When it gets to 0, the phone will be removed from the simulation.

- Synchronized
    - You can synchronize your block of code or method.
    - Only one phone is in the repair shop at a time. (**NO** race conditions)
    - Answer the following question as comments in your Phone class (put question and your answer on the first line of your code).
      Question: "Which object(s) have you chosen for the synchronize? Why?"