Data Structures and Algorithms: Assignment 01

Due: Monday 3rd May 2021, 11:59pm

100 marks possible – worth 20% of final grade

When submitting, zip up your entire Netbeans project into a folder with the following structure:

lastname_firstname_studentID.zip (using your own name and ID)

and submit to the Assignment 1 folder on Blackboard BEFORE the due date and time. Late submissions will incur a penalty. Anti-plagiarism 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) – **Fish Tank (20 marks)**

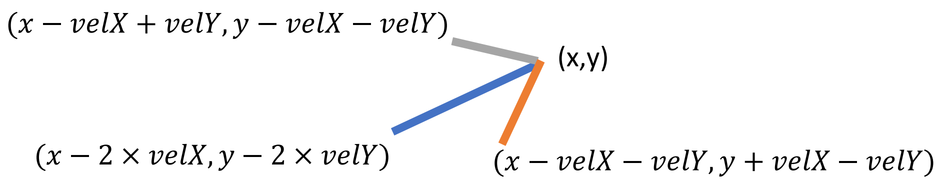
Part A: Using the UML below create a class that represents a *fish* swimming in a tank. A *Fish* runs in a separate thread while it is still alive and has an *x,y* position. A fish moves within its world from (0,0) to (world_width, world_height) pixel coordinates, increasing its position by suitable *dx,dy* movement values. It will also periodically generate small randomized noise factors which are added or subtracted to its *dx, dy* movement values after several movements (but still keeping dx and dy below an upper threshold). A fish is also randomly given a *size* during instantiation within some min and max value. A fish is instantiated with a reference to a *FishShoal* collection.

```
Fish
----------------------------------------
- x, y : double
- dx, dy : double
- size : double
- isAlive : boolean
+ world_width, world_height : int
- colour : Color[]
- shoal : FishShoal
----------------------------------------
+ Fish(shoal : FishShoal)
+ run(): void
+ getX(): double
+ getY(): double
+ getSize() : double
+ kill() : void
- move() : void
+ eat(target : Fish) : void
+ draw(g : Graphics) : void
```

As a fish moves around the world it should check to see if it can eat another fish by passing itself to the *FishShoal* method *canEat*, and obtaining a reference to a *target* fish available to eat (or null if it cannot eat a fish). The *eat* method is used to feed this fish with the target parameter fish, in which case it kills the target fish (stopping its thread from running) and increases its size (up to a maximum). If a fish dies it should remove itself from the shoal.

A three coloured fish can be drawn as follows:

$$speed = \sqrt{dx^2 + dy^2} \qquad velX = \frac{size \times dx}{2 \times speeed} \qquad velY = \frac{size \times dy}{2 \times speeed}$$


$(x - velX + velY, y - velX - velY)$
(x,y)
$(x - 2 \times velX, y - 2 \times velY)$
$(x - velX - velY, y + velX - velY)$

Part B: Using the UML diagram below, create a class called *FishShoal* which represents a shoal of fish by encapsulating a *List* of *Fish* objects. It has methods to *remove* and *add* fish to the shoal and *size* to obtain the number of fish in the shoal. It has a method *drawShoal* which draws all fish in the shoal by calling the *draw* method of each Fish. The method *canEat* is used to check if the parameter fish can eat any other fish in the shoal. The two basic rules for the parameter fish to eat another fish in the fish shoal is as follows:

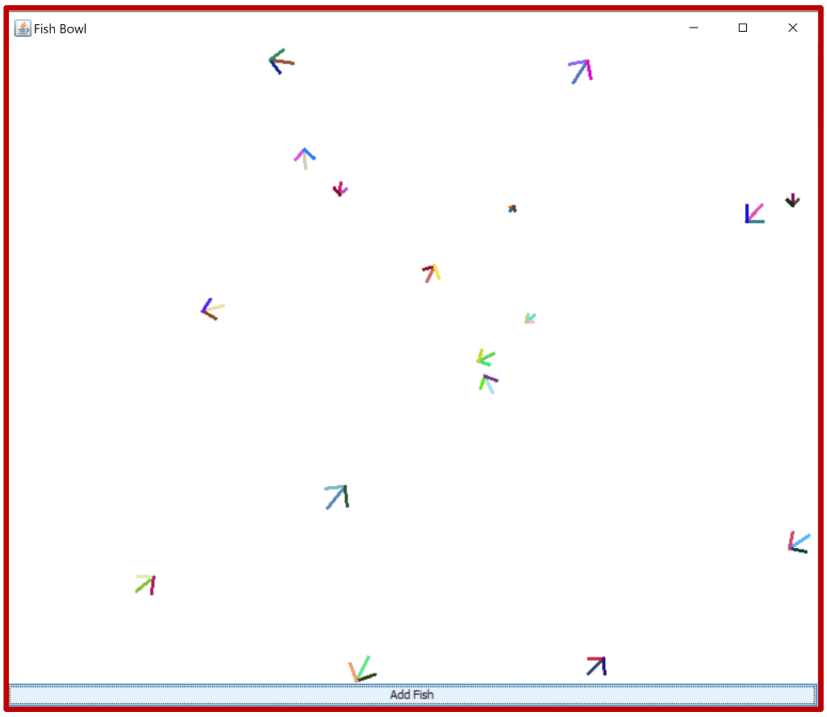
- The parameter fish must be 40% bigger in size than a target fish
- The parameter Fish position must be within range of the target fish position, the range is determined by taking an average of the size of the two fish.

If the parameter fish can eat a fish in the shoal the *canEat* method will return reference to that fish otherwise it will return *null* if the parameter fish cannot eat.

FishShoal
- fishList : List<Fish>
+ FishShoal() + add(fish : Fish) : void + remove(fish : Fish) : void + drawShoal(g : Graphics) : void + canEat(fish : Fish) : Fish

Part C: Create a GUI called FishTank, which can be used to draw all *Fish* in a *FishShoal* collection. It should have a button which can add a single fish in the shoal and start it running as a thread. Carefully think about what changes need to be made in both *FishShoal* and *Fish* to **prevent any race conditions** from occurring. Feel free to tweak movement variables and fish sizes as necessary to ensure a nice smooth GUI.

Here is a screenshot showing several Fish moving around the world.



Question 2) – Bag implementations (20 marks)

The following UML class diagram represents an *abstract data type* for a **Bag** data structure.

```
<<interface>>
Bag<E>

+ add(item : E) : boolean
+ grab() : E
+ remove(item : E) : boolean
+ size() : int
+ capacityRemaining() : int
+ isFull() : boolean
+ isEmpty() : boolean
+ clear() : void
+ iterator() : Iterator<E>
+ toArray() : E[]
```

Once instantiated, a *bag* has a fixed capacity and does not need to keep elements in any specific order. The only way to get an *item* in the bag is to *grab* it, which gives back an item in the bag at random but does not remove it. It also has many other useful methods; *add*, for adding a new item but only if it can fit it; *remove*, which removes a specific item from the bag; *clear*, which removes all items from the bag; *iterator*, which allows items to be iterated over; *isFull* and *isEmpty* which return true if the bag is full or empty; *size*, giving the number of items in the bag; *capacityRemaining*, returns how many items can still be added to bag before it is full; and *toArray* which returns an array of the same type of all items left in the bag.

Create **two** implementing classes of *Bag*. Once called *ArrayBag* which implements a bag with an underlying **array**, the other called *LinkedBag* which implements a bag with an underlying **doubly linked list**. Include two constructors for each, one which instantiates a bag with a fixed capacity of 10, the other which allows a using program to specify capacity. Include a *toString* method with each implementation and throw any suitable exceptions where needed.

Create a suitable *test* class which can be used to thoroughly test both your bag implementations.

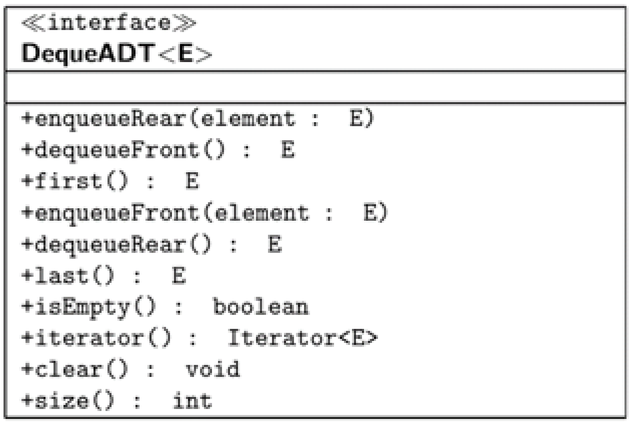
Question 3) – Bracket Sorter (10 marks)

Design and create a class that uses an algorithm and an appropriate data structure used 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. The pairs are: () {} <> []

Examples:
String: "{((2 x 5)+(3*-2 + 5))}" - will evaluate successfully
String: "{ (2 x 5)+(3*-2 + 5))}" - will not as one of the opening brackets (missing
String: "List<List>" - will evaluate successfully
String: "List<List " - will not as missing > closing angle brace
String: "{(<>){}{…}(e(e)e){hello}}" - will evaluate
String: "{(< eeeek>>){}{…} e(e)e){hello} " - will not evaluate for multiple reasons

Question 4) Deque implementation – 15 marks

A *deque* (pronounced deck) or double-ended queue, is a linear collection that allows insertion and deletion at both ends. The below interface called *DequeADT<E>* available from Blackboard, is an abstract data type which describes operations that a deque should have.



```
≪interface≫
DequeADT<E>

+enqueueRear(element :  E)
+dequeueFront() :  E
+first() :  E
+enqueueFront(element :  E)
+dequeueRear() :  E
+last() :  E
+isEmpty() :  boolean
+iterator() :  Iterator<E>
+clear() :  void
+size() :  int
```

- It has *enqueue* and *dequeue* methods for adding or removing an element from the front and rear of the deque.
- *first* and *last* for obtaining the elements (but not removing) at the front and rear of the deque.
- *isEmpty* for checking whether any elements are in the deque.
- *clear* for clearing out all elements in deque.
- *size* for returning the number of elements currently in the deque.
- *iterator* which can be used to contiguously iterate over the elements in the deque, going from the front of the deque to the rear.

Implement the *Deque* abstract data type using an **array,** called *ArrayDeque<E>*. Implement the data structure yourself, do not use extra Java *Collection* classes.

For efficiency *first*, *last*, *enqueueRear*, *enqueueFront*, *dequeueRear*, *dequeueFront*, *clear*, *size*, and *isEmpty* should all be *O(1)* operations. Feel free to add any necessary private helper methods and a suitable *toString* which returns a string representation for your deque implementation.

Further enhance your classes to make sure all methods are safe from unexpected events using appropriate exception handling (example, if user code tries to remove something from the deque when empty, it should throw an appropriate exception). Create a test class with a suitable *main* method which effectively tests all operations on one or both *Deque* implementations.

Question 5) File Sorting – 20 marks

There is often a big difference between an algorithm's performance in internal memory and its performance in external storage. For example, quick sort is considered very efficient for collections held in memory but is less efficient with elements stored in a file, whereas merge sort is more efficient with sorting files.

Suppose the lines of a large text file are to be sorted, but the file is too large to all fit in memory at one time. Prepare a class called FileSorter whose constructor accepts an integer limit (a maximum for the number of Strings that will be held in memory at a time), and input and output text File objects.

The algorithm firstly performs the split stage, repeatedly reading each string from the input file line by line, breaking the text file apart with at most limit strings (where a String is on its own single line in the file). Once the limit has been reached, the algorithm should then quicksort the elements in memory then output them back to a smaller temporary file (you may use an existing class for the quicksort routine). The split stage continues until all lines of the input file are read in. After the split stage, the algorithm then performs the merge stage. The merge stage uses merge sort to read files together two at a time (deleting the temporary files), until a single sorted file remains, the output file.

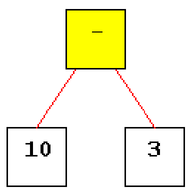
The FileSorter should be able to run as a separate thread, so any gui program remains responsive to the user. Add any necessary methods which can be used to notify the user of the progress in the sorting algorithm.

Create a class called FileSorterGUI which can be used to run FileSorting tasks, it should have a button to queue up a new task and another button to execute the task at the front of the queue. It should also display progress bars to update the user on the progress of the currently running task.

Question 6) Expression Tree – 15 marks

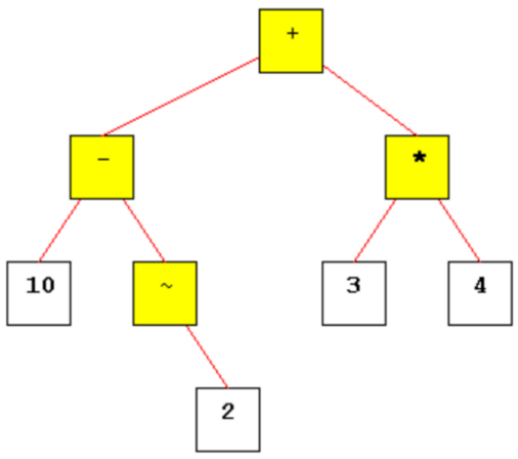
An *expression tree* is a binary tree representing mathematical expressions built from postfix notation and converting to infix notation. Each internal node in the tree can be an operator (mathematical operations), whereas leaf nodes are operands (numbers or values). An operator can calculate an expression from the output of its left child and right child if it is binary operator, or simply taking the expression from its right child if it's a unary operator.

Eg: The posfix notation input **10 3 -** would produce the expression tree:



This would evaluate to the infix notation (10 - 3) = 7

Eg: The postfix notation of **10 2 ~ - 3 4 * +** as input would produce the expression tree:



This would evaluate to the infix notation: : ((10 - ~ 2) + (3 * 4)) = 21.5

The abstract class *ExpNode*, available on Blackboard and described in the UML below, keeps an expression *symbol* as a String as well as links to its left and right subtrees. The *toString* method simply prints out the symbol as a String which could either be a mathematical operator or a number.

<<abstract>> **ExpNode**
#symbol : String +leftChild : ExpNode +rightChild : ExpNode
+ ExpNode(symbol : String) *+ evaluate() : double* + toString() : String

Create concrete subclasses of *ExpNode* called *OperatorNode* and *OperandNode*. Their constructors should accept a String *symbol* used to represent the operand {which is also converted to a double value} or operator {binary operators; multiplication (*) division (/), subtraction(-), addition(+) and a unary operator; inverse (~)}, throwing an appropriate exception if the symbol is invalid. The subclasses must override the abstract method *evaluate* which for an operand will simply return its double value. However, for an operator node it must evaluate the correct expression by evaluating the left subtree and right subtree depending on the operator symbol (care should be taken for the unary inverse operator).

Create another class called ExpressionTreeBuilder using the UML below. Note all methods should be declared static.

ExpressionTreeBuilder
+ buildExpressionTree(postfixStrings : String[]) : ExpNode *+ countNodes() : int* *+ toInfixString(node : ExpNode) : String*

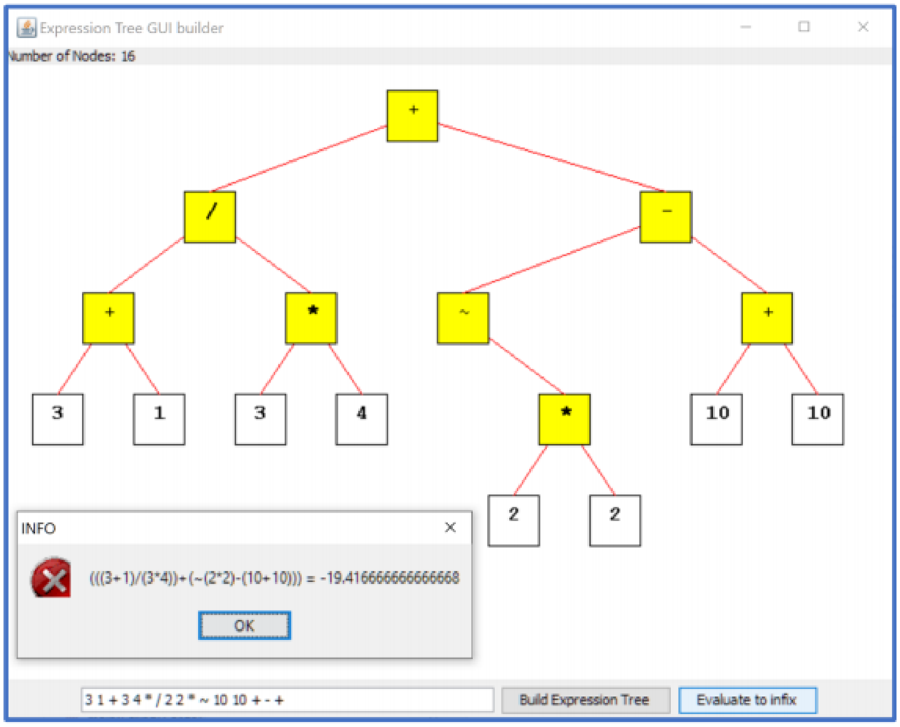
The method *buildExpressionTree* which builds an expression tree comprised of operator and operand nodes by reading symbols from the postfix input array. The following pseudo code may be very useful:

```
Create Stack S, Create parent ExpNode P;
for(symbol in expression)
    if symbol is an operand
        push(operand) onto S
    if symbol is operator
        set P as new Operator
        pop two Nodes (or only one if unary operator) and set as P.left and P.right
        push(P) onto S
return P.
```

The method *toInfixString* which can be used to recursively convert a binary expression tree into infix notation using brackets.

The method *countNodes* should recursively count the number of nodes down the left and right subtrees from the parameter *ExpNode*.

Obtain the partially complete class *ExpressionTreeGUI* from Blackboard. The class can display an expression tree, drawing the tree from the root node (if not null) down. It has an input text field to enter a post fix expression (separated by spaces) and associated buttons. One button is used to build the tree, the other to evaluate the tree, outputting the infix notation string, and the resulting evaluated output. It also needs to update the JLabel to count the number of nodes in the tree. Note, all drawing and GUI components are already done for you, you simply need to just wire it in to work with ExpressionTreeBuilder and ExpNode plus subclasses. Your working GUI should look like something like this once tree is built and the evaluate to infix button pressed.



Number of Nodes: 16

(((3+1)/(3*4))+(~(2*2)-(10+10))) = -19.416666666666668

3 1 + 3 4 * / 2 2 * ~ 10 10 + - +

Make any necessary modifications to handle any unexpected user input such as invalid symbols and incorrect postfix formula.