CS2040S Lab 4
List ADT

One-Day Assignment 2 – Sort of Sorting

- Need to use stable sort
- Thankfully, Java's Arrays.sort() is stable by default
- Just write a custom comparator to compare by first 2 characters, instead of the entire string

```
// the overridden compare method
public int compare(String a, String b) {
    return a.substr(0, 2).compareTo(b.substr(0, 2));
}
```

Take-Home Assignment 1a – Best Relay Team

- Use a custom class to represent a Runner
  - Contains the 1st leg timing, subsequent leg timings and name
- Use 2 arrays to store the Runners
  - Sort one by 1st leg timing
  - Sort the other by subsequent leg timings
- Pick the first runner A from the 1st leg timing array
- Pick the first 3 runners from the subsequent leg timing array, that are not the same as runner A(since it is possible for the same runner to appear in the front of both arrays
- Repeat the above for the second/third/fourth runner from the 1st leg timing array
- Output the team that gives the best timing among the 4 possible options

Take-Home Assignment 1b – Card Trading

- Start by assuming all cards Anthony owns are sold
  - This would result in the highest possible profit, if no other restrictions exist
- Then, determine the cost for keeping a certain card type as a combo
  - For a card type which Anthony has none of, he needs to buy 2 cards (2 * buy)
  - For a card type which Anthony has one of, he cannot sell that card, and has to buy one more (1 * sell, and 1 * buy)
  - For a card type which Anthony has two of, he cannot sell both cards (2 * sell)
- Sort the cost for all card types, and pick the K smallest elements as the card types to keep. Subtract the sum of these elements from the profit (derived earlier) to get the final answer

Lab 4 – Generics

- Most Java API usage from this point on requires the use of generics
- Essentially a way to define the data type being used, much like arrays
- Eg. String[] versus ArrayList<String>
- Can also be used for your own custom classes
- Eg. Student[] versus ArrayList<Student>
- Note that primitive data types (eg. int, double) cannot be used as generics; you will need to use their relevant Java classes instead
  - Eg. ArrayList<Integer>, ArrayList<Double>

Lab 4 – Lists

- Two kinds of lists tend to be more frequently used in Java
- ArrayList and LinkedList
- Main differences:
  - Accessing elements within the list
    - ArrayList offers O(1) time to access any element in the list
    - LinkedList offers O(1) time to access elements at the front and back of the list only
  - Inserting/deleting elements from the list
    - ArrayList offers O(1) time to insert and delete from the back of the list only
    - LinkedList offers O(1) time to insert and delete from the front or the back of the list
- Inserting/deleting from other positions (apart from the ones mentioned above) runs in about O(n) time for both ArrayList and LinkedList
- For LinkedList accessing any position other than front and back result in O(n) on average

Lab 4 – Lists

- For arrays, a Java class called Vector exists
- This is similar to the ArrayList class, but is “synchronised”
  - Ie. used for multi-threading, by ensuring that only one thread can modify the Vector at a given time
  - However, performing this check makes it slightly slower as a result
- For this module, just the use of ArrayList is sufficient, as we do not cover multi-threading

Lab 4 – ArrayList

Method name	Description	Time
.add(YourClass element)	Adds *element* to the end of the list	O(1)
.add(int index, YourClass element)	Adds *element* to the position at *index*	O(size() - index)
.clear()	Empties the list	O(n)
.contains(Object o)	Checks if *o* is in the list, based off the object's equals() method	O(n)
.get(int index)	Gets the element at *index*	O(1)
.remove(int index)	Removes the element at *index*	O(size() - index)
.size()	Returns the number of elements in the list	O(1)

O(size() - index) is due to the need to shift all elements to the right of *index* when inserting/deleting

Lab 4 – LinkedList

Method name	Description	Time
.add(YourClass element)	Adds *element* to the end of the list	O(1)
.add(int index, YourClass element)	Adds *element* to the position at *index*	O(size() - index) or O(index), whichever is smaller
.clear()	Empties the list	O(n)
.contains(Object o)	Checks if *o* is in the list, based off the object's equals() method	O(n)
.get(int index)	Gets the element at *index*	O(size() - index) or O(index), whichever is smaller
.remove(int index)	Removes the element at *index*	O(size() - index) or O(index), whichever is smaller
.size()	Returns the number of elements in the list	O(1)

Lab 4 – Stacks/Queues

- Special kinds of lists
  - Stacks only allow inserting, accessing and deleting from the top of the stack in O(1) time (first-in last-out)
  - Queues only allow accessing and deleting from the front of queue, and inserting from the back of the queue in O(1) time (first-in first-out)

Lab 4 – Stacks/Queues

- Queue is an interface, so you cannot initialize it using new Queue()
  - Instead, initialise it using a LinkedList: `Queue<YourClass> queue = new LinkedList<YourClass>();`
  - Or as an ArrayDeque (check the documentation)
- Stack is not an interface, so you can use new Stack() directly
  - `Stack<YourClass> stack = new Stack<YourClass>();`
- Note that the Stack class in Java extends the Vector class, so the use of methods such as get(index), and add(index, element) is possible; however, these methods should not be used in a pure stack implementation

Lab 4 – Stack

Method name	Description	Time
.empty()	Checks if the stack is empty	O(1)
.peek()	Retrieves the element at the top of the stack	O(1)
.push(YourClass element)	Adds *element* to the top of the stack	O(1)
.pop()	Removes and retrieves the element at the top of the stack	O(1)

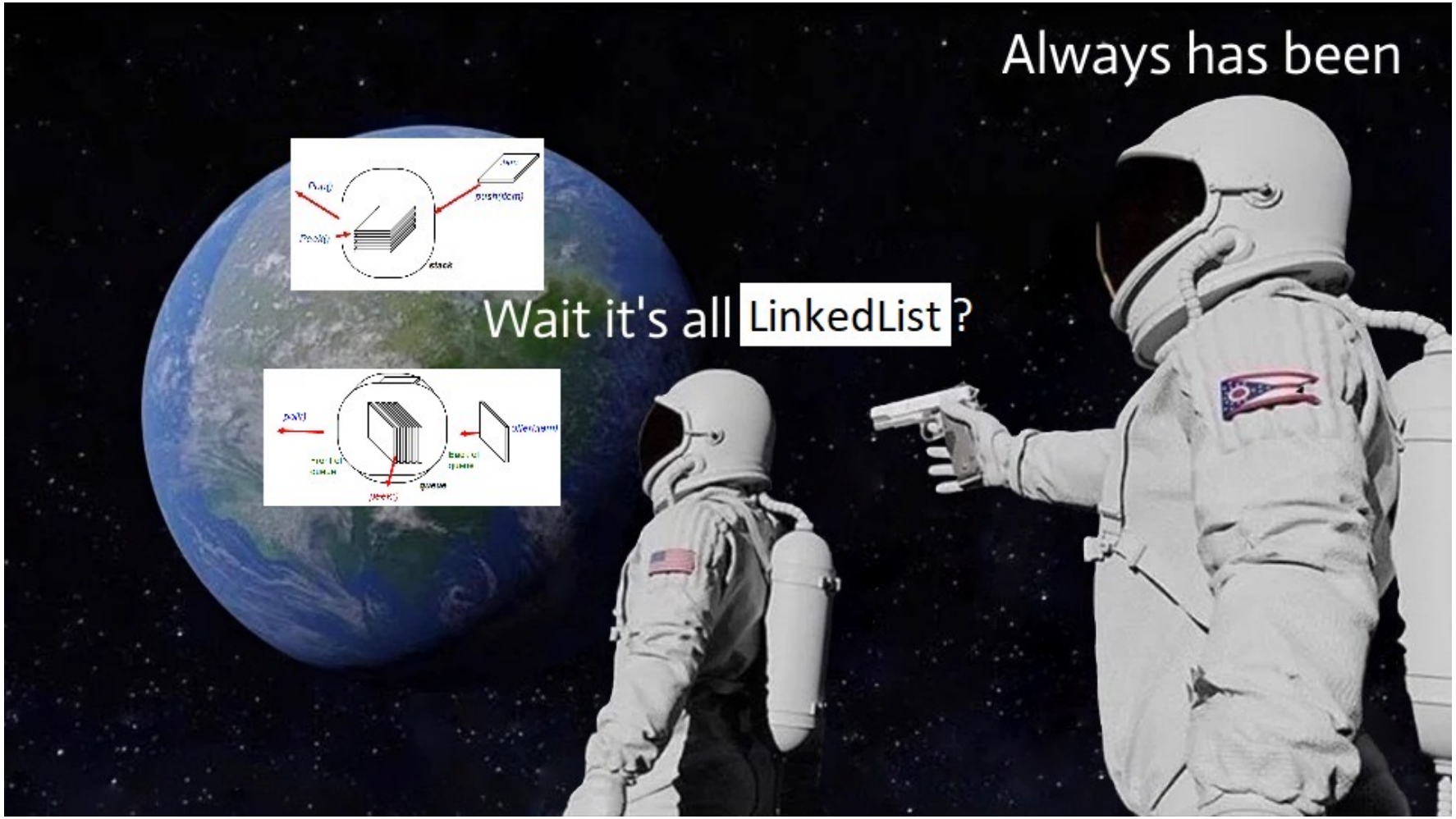
Lab 4 – Queue (using LinkedList)

Method name	Description	Time
.isEmpty()	Checks if the queue is empty	O(1)
.offer(YourClass element)	Adds *element* to the end of the queue	O(1)
.peek()	Retrieves the element at the head of the queue	O(1)
.poll()	Removes and retrieves the element at the head of the queue	O(1)

Lab 4 – LinkedList (again)

- Note that LinkedList supports operations from both Stack (excluding the .empty() method) and Queue
  - However, LinkedList does not extend Stack, so the following is **not** possible:
  - `Stack<YourClass> stack = new LinkedList<YourClass>();`
- As such, it's entirely possible to declare LinkedLists anytime you need a stack or a queue:
  - The top of the stack is the head of the LinkedList
  - The front of the queue is the head of the LinkedList, while the back is the tail of the LinkedList
  - The .peek() method is consistent with both definitions (returns the element at the head of the list)

Lab 4 – LinkedList (again)



Take-Home Assignment 2a – Join Strings

- Given a list of strings, concatenate them together, and output the final string
- Main emphasis of the question is on concatenating strings together
  - Concatenating strings (via str.concat() or the + operator) takes O(k) time, where k is the length of the resulting string (see Tutorial 1, problem 2e)
  - Using StringBuilder/StringBuffer's .append() is slightly better (takes O(m) time, where m is the length of the string that is added on), but still too slow
  - Therefore, simply using the default string methods will not be fast enough
  - Need to find a way to do so in O(1) time when handling queries

Take-Home Assignment 2b – Teque

- Create a custom data structure to support the push_back, push_front, push_middle, and get operations
- Need to ensure all operations run in O(1) time
- This problem will also require the use of buffered IO methods (covered in Lab 2 slides), due to the large input/output sizes (up to 1M lines!)

One-Day Assignment 3 – Coconut Splat

- Simulate a counting-out game
- A player's hand has 4 possible states:
  - 1. Both hands folded together (initial state)
  - 2. Fist
  - 3. Palm down
  - 4. Behind back
- A player's hand moves from state n to state (n + 1) if that hand is touched last
- A player is out of the game if both hands are behind their back
- Find out which player will be the last player standing

One-Day Assignment 3 – Coconut Splat

- State 1 (folded), initial state:
  - Counts as a set of hands (ie. only one syllable) in this form instead of two individual hands
- State 1 (folded) -> State 2 (fist)
  - Split folded hands into 2 fists; counting begins with the first fist on the next round (so it goes to the player's first fist, followed by the player's second fist, followed by the hand of the next player)
- State 2 (fist) -> State 3 (palm down)
  - Change fist into palm down; counting begins with the next hand after this (which could be the player's other hand, or the next player's hand)
- State 3 (palm down) -> State 4 (behind back)
  - Hand is moved behind back; this hand will no longer be counted in subsequent rounds. Counting begins with the next hand after this (which could be the player's other hand, or the next player's hand)

One-Day Assignment 3 – Coconut Splat

- The following 2 slides contain an example of the game simulated for 3 players, with 3 syllables in the rhyme
- The last person standing in this example should be player 2
- In the example, each round starts from the underlined hand/fist/palm

Hand is touched	Hand is touched (last)	Hand changes state	Hand is out of play

Move	Player 1		Player 2		Player 3	
Initial State	Folded Hands		Folded Hands		Folded Hands	
Round 1	Folded Hands		Folded Hands		Folded Hands	
After R1	Folded Hands		Folded Hands		Fist	Fist
Round 2	Folded Hands		Folded Hands		Fist	Fist
After R2	Fist	Fist	Folded Hands		Fist	Fist
Round 3	Fist	Fist	Folded Hands		Fist	Fist
After R3	Fist	Fist	Fist	Fist	Fist	Fist
Round 4	Fist	Fist	Fist	Fist	Fist	Fist
After R4	Fist	Fist	Fist	Fist	Palm Down	Fist
Round 5	Fist	Fist	Fist	Fist	Palm Down	Fist
After R5	Fist	Palm Down	Fist	Fist	Palm Down	Fist
Round 6	Fist	Palm Down	Fist	Fist	Palm Down	Fist
After R6	Fist	Palm Down	Fist	Fist	Behind Back	Fist
Round 7	Fist	Palm Down	Fist	Fist	Behind Back	Fist
After R7	Fist	Behind Back	Fist	Fist	Behind Back	Fist

Hand is touched	Hand is touched (last)	Hand changes state	Hand is out of play

Move	Player 1		Player 2		Player 3		
After R7	Fist	Behind Back	Fist	Fist	Behind Back	Fist	
Round 8	Fist	Behind Back	Fist	Fist	Behind Back	Fist	
After R8	Fist	Behind Back	Fist	Fist	Behind Back	Palm Down	
Round 9	Fist	Behind Back	Fist	Fist	Behind Back	Palm Down	
After R9	Fist	Behind Back	Fist	Palm Down	Behind Back	Palm Down	
Round 10	Fist	Behind Back	Fist	Palm Down	Behind Back	Palm Down	
After R10	Fist	Behind Back	Palm Down	Palm Down	Behind Back	Palm Down	
Round 11	Fist	Behind Back	Palm Down	Palm Down	Behind Back	Palm Down	
After R11	Palm Down	Behind Back	Palm Down	Palm Down	Behind Back	Palm Down	
Round 12	Palm Down	Behind Back	Palm Down	Palm Down	Behind Back	Palm Down	
After R12	Palm Down	Behind Back	Palm Down	Palm Down	Behind Back	Behind Back	<- X_X
Round 13	Palm Down	Behind Back	Palm Down	Palm Down	Behind Back	Behind Back	
After R13	Palm Down	Behind Back	Palm Down	Behind Back	Behind Back	Behind Back	
Round 14	Palm Down	Behind Back	Palm Down	Behind Back	Behind Back	Behind Back	
After R14	Behind Back	Behind Back	Palm Down	Behind Back	Behind Back	Behind Back	

X_X

^_^

Deque (extra)

- A subinterface of Queue
- Supports insertion and removal at both ends, hence a “double-ended queue”
- Initialised with either LinkedList or ArrayDeque
- Can also use it as a Stack or Queue

Deque (extra)

Method name	Description	Time
.isEmpty()	Checks if the deque is empty	O(1)
.peek()	Retrieves, but does not remove, the first element of this deque.	O(1)
.offer(YourClass element)	Inserts the specified element into the **queue** represented by this deque.	O(1)
.poll()	Retrieves and removes the head of the **queue** represented by this deque.	O(1)
.push()	Pushes an element onto the **stack** represented by this deque. (i.e. head)	O(1)
.pop()	Pops an element from the **stack** represented by this deque.	O(1)