CS2040 Tutorial 2

Week 4, starting 29 Aug 2022

Q1 ADTs

What is the difference between these 3 pieces of code?

```
ArrayList<String> findNames() {
   ArrayList<String> ls = new ArrayList<>();
   // fill ls
   return ls;
}
```

vs

```
List<String> findNames() {
   ArrayList<String> ls = new ArrayList<>();
   // fill ls
   return ls;
}
```

vs

```
Collection<String> findNames() {
   ArrayList<String> ls = new ArrayList<>();
   // fill ls
   return ls;
}
```

Q2 List ADT Implementations

In lectures, we have learned two general List implementations – array-based and reference-based. ArrayList and Vector are array-based list implementations, while LinkedList is a reference-based implementation. Let us compare and contrast the two implementations.

For a list containing N elements, around how many elements would be accessed/modified when:

- **(a)** Adding to end of the list (new index == N / tail)
- **(b)** Adding to front of the list (index == 0 / head)
- **(c)** Removing from front of the list (index == 0 / head)
- **(d)** Getting (accessing) any element, on average (from index == 0 to index == N-1)

Assume the linked list has a tail reference, and is doubly-linked.

Q3 Linked List Operations

When mutating a linked list, we sometimes can:

- create/instantiate **new nodes** containing the desired elements
- manipulate **next pointers**, so that no node is created or removed
- manipulate the **items** (elements) in two of the nodes without rearranging next pointers

Implement a method `swap(int index)` in the `CircularLinkedList<E>` class given to you below, to swap the node at the given index with the next node. The `toString()` method allows you to test your program.

```
class CircularLinkedList<E> {

   int _size;
   ListNode<E> _head, _tail;

   void addFirst(E element) {
      _size++;
      _head = new ListNode<E>(element, _head);
      if (_tail == null) _tail = _head;
      _tail.next = _head;
   }

   public String toString() {
      if (_head == null) return "[]";
      StringBuilder sb = new StringBuilder();
      sb.append("head ->[" + _head.item);
      for (ListNode<E> curr = _head.next; curr != _head; curr=curr.next)
         sb.append(", " + curr.item);
      sb.append("]<- tail");
      return sb.toString();
   }
   void swap(int index) { ... }
}
```

A pre-condition is that `index` will be non-negative. If the index is larger than the size of the list, then the index wraps around. For example, if the list has 13 elements, then `swap(15)` will swap nodes at indexes 2 and 3.

Restriction: You are NOT allowed to:

- create any new nodes
- modify the element in any node

[Hint: Consider all cases, and remember to update the necessary instance variables!]

Question 4 (Online Discussion) – Merging Linked Lists

We are now going to add new functionality **within** the `TailedLinkedList<E>` class:

```
static TailedLinkedList<Integer> merge(
     TailedLinkedList<Integer> left,
     TailedLinkedList<Integer> right) {...}
```

Implement the `merge()` class method **efficiently**. Given two linked lists in which all elements are sorted, create a **new linked list** in which all elements are in **sorted** order. Where there is a draw, always take the element from the `left` list. Although a third linked list object is created, be reminded that **NO new nodes** (node objects) are to be created

Restrictions:

- You are NOT allowed to use additional data structure, but you may maintain a few node references
- You are NOT allowed to create any new nodes

As an example:

- `left` before merge: [**1 3 4 5 5 7**]
  `right` before merge: [**2 2 3 3 5 6**]

- New returned list: [**1 2 2 3 3 3 4 5 5 5 6 7**]
  `left` after merge: []
  `right` after merge: []

As the original lists will be corrupted, and could potentially corrupt the new list, the method should **clean up** the original lists by emptying them. To simplify this question, you may assume both `left` and `right` are non-empty.

Tip: Think of an idea in which you have a few steps that can be repeated many times... How would you merge 2 lists *if they were array-based*?