CS2040 Tutorial 2 Suggested Solution

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;
}
```

Answer

An ArrayList<String> is a Collection<String>, so all 3 pieces of code work. If a method returns Collection<String>, the caller (calling method) can only use functionality available to any general Collection<String> as that is the datatype of the reference, but not List- or ArrayList-specific methods like get(i), indexOf(i), add(i, elm)

This can be good if you want to guide the caller to just view the result as an unordered bag of names, and discourage access by index

Draw a diagram of how Collection<E>, List<E>, ArrayList<E>, LinkedList<E> are related. Each week where new Java API ADTs / data structures (e.g. Queue<E>, Stack<E>, Set<E>, Map<E>, …) are discussed, add them to your diagram, so you don't need to memorize the operations that each one has. For example, LinkedList<E> has contains() because it is a Collection<E>, has add(int, E) because it is a List<E>, has getFirst() and getLast() because it is a Deque<E>

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

Answer

(a)

For **array-based** implementation, **most of the time** only **1** element is accessed/modified as no shifting is required. However, when the array is already at capacity, the entire underlying array has to be **reallocated** to another array with a larger capacity, which involves copying all **N** elements to the new array

However, when adding a large number of elements to an empty array-based list, the **average** number of elements modified per add operation is **always a constant**

For a linked list, only 1 node is accessed, or 2 nodes if you also count the new node

(b), (c)

Adding to front: In a typical **array-based** implementation, **N** elements from the insertion point onward have to be copied away from the front (to the right) before insertion

Removing from front: Efficiency same as (b), N-1 elements left shifted instead for array-based impl

For a **linked list**, only **1 or 2** nodes are accessed/modified. The second node onwards need not be accessed/modified

(d)

In an **array-based** implementation, only **1** access is needed to move to any index, because arrays have random access. For a doubly linked list with head and tail references, the average number of accesses is **N/4** if your algorithm can choose to start from either end

Therefore, insertion/removal from an array-based list, except at one end, is inefficient
On the other hand, array-based list is good for random access, unlike a linked list

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!]

Answer

In a singly-linked list, to physically reorder/remove a node, the previous node’s next reference needs to be mutated. Modulo can also be used to eliminate repeatedly traversing the list more than once per operation. There is also a need to handle a few special cases depending on the size of the list, and whether head and tail references need to be updated

```
void swap(int index) {
   if (_size < 2) return; // if 0 or 1 nodes, don’t bother

   if (_size == 2) { // if 2 nodes, only need to swap head and tail refs
      ListNode<E> newTail = _head;
      _head = _tail;
      _tail = newTail;
      return;
   } // At this point, the list has >2 nodes

   index %= _size; // ensure that index < size of list first

   // get the 3 desired nodes
   ListNode <E> prev = _tail;
   for (int times = 0; times < index; times++) prev = prev.next;

   ListNode<E> curr = prev.next; // curr now at left node to be swapped
   ListNode<E> succ = curr.next;

   // swap the 2 nodes: Note the order!
   curr.next = succ.next;
   succ.next = curr;
   prev.next = succ;

   if (index == 0) {
      _head = succ; // head incorrect
   } else if (index == _size - 2) {
      _tail = curr; // swap(tail-1), tail incorrect
   } else if (index == _size - 1) {
      _head = curr; // swap(tail), both head & tail swapped
      _tail = succ;
   }
}
```

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*?

Can be done efficiently in O(N) time

The addFirst()/addLast() operations of the TailedLinkedList class do create new nodes, so using any of them is NOT acceptable