CS 61B Spring 2021

Linked Lists & Arrays

Topical Review Section 2: February 7, 2021

1 Linked List Practice

Here's a basic `SLList` class we've defined. Assume the `SLList` constructor is properly implemented and creates a sentinel with a placeholder value. Use `SLList` to answer the following parts.

```
public class SLList {
    private class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int item, IntNode next) {
            this.item = item;
            this.next = next;
        }
    }

    private IntNode sentinel;
    private int size;

    public void addFirst(int x) {
        this.sentinel.next = new IntNode(x, this.sentinel.next);
        this.size += 1;
    }
}
```

(a) Implement `addLast(int x)`, a method of `SLList` that creates a new `IntNode` and adds it to the back of our `SLList`

```
public void addLast(int x) {
    IntNode pointer = this.sentinel;
    while (pointer.next != null) {
        pointer = pointer.next;
    }
    pointer.next = new IntNode(x, null);
    this.size += 1;
}
```

(b) Notice that this is quite slow for long `SLLists`, why? How can we change `SLList` to make this faster?

We must iterate through the entire list every time we want to insert an element at the back. We can implement a Doubly Linked List class, known as `DLList`, which should be able to insert elements at both the front and back very quickly.

(c) Let's create a Doubly Linked List class. The `DLList` should be able to support a fast insertion at both the front and back of the list. Assume the `DLList` constructor is already implemented and creates a sentinel node with a placeholder value properly. Also assume `sentinel.next` points to the first node in the list, and `sentinel.prev` points to the last node. Fill in the blanks below:

```
public class DLList {
    private class IntNode {
        public int item;
        public IntNode next;
        public IntNode prev;
        public IntNode(int item, IntNode next, IntNode previous) {
            this.item = item;
            this.next = next;
            this.prev = prev;
        }
    }

    private IntNode sentinel;
    private int size;

    public void addFirst(int x) {
        this.size += 1;
        IntNode oldFront = this.sentinel.next;
        IntNode newNode = new IntNode(x, oldFront, this.sentinel);
        this.sentinel.next = newNode;
        oldFront.prev = newNode;
    }

    public void addLast(int x) {
        this.size += 1;
        IntNode oldBack = this.sentinel.prev;
        IntNode newNode = new IntNode(x, this.sentinel, oldBack);
        this.sentinel.prev = newNode;
        oldBack.next = newNode;
    }

}
```

(d) Implement `destructiveReverse`, a method of `DLList` that destructively reverses the values of our `DLList`. For example, if our list is $1 \leftrightarrow 3 \leftrightarrow 5 \leftrightarrow 7$, then `destructiveReverse` should modify the list to be $7 \leftrightarrow 5 \leftrightarrow 3 \leftrightarrow 1$. `destructiveReverse` should modify **values only**, not pointers.

```
public void destructiveReverse() {
    if (this.size == 0) {
        return;
    }
    IntNode lPointer = this.sentinel.next;
    IntNode rPointer = this.sentinel.prev;
    int lIndex = 0;
    int rIndex = this.size - 1;
    while (lIndex < rIndex && lPointer != null && rPointer != null) {
        int temp = lPointer.item;
        lPointer.item = rPointer.item;
        rPointer.item = temp;
        lIndex += 1;
        rIndex -= 1;
        rPointer = rPointer.prev;
        lPointer = lPointer.next;
    }
}
```

2 ArrayLists

Use the following class structure to answer the following parts below.

```
1  public class AList {
2    private int[] items;
3    private int size;
4    private int FACTOR = 2;
5
6    public AList() {
7      items = new int[100];
8      size = 0;
9    }
10
11   public int getLast() {
12     return items[size - 1];
13   }
14
15   public int get(int i) {
16     return items[i];
17   }
18
19   public int size() {
20     return size;
21   }
22
23   private void resize(int capacity) {
24     int[] a = new int[capacity];
25     System.arraycopy(items, 0,
26                      a, 0, size);
27     items = a;
28   }
29 }
```

(a) Implentation the `removeLast(int x)` method that "removes" and returns the int value at the end of the `AList` by setting it to zero. You do not have to resize down in this implementation.

```
public int removeLast() {
    int returnItem = get(size - 1);
    items[size - 1] = 0;
    size -= 1;
    return returnItem;
}
```

(b) Finish the implentation of the `addLast(int x)` method that adds an int at the end of the `AList` (index=size). The method should take into account the case when items has no more space available and increase the capacity of items by a factor of FACTOR. Feel free to use any helper methods available in the code

above.

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size * FACTOR);
    }
    items[size] = x;
    size += 1;
}
```

(c) Your friend would love to use your AList class for Proj0 of his SC16p class at UCLA. However, he needs your AList class to have a method that allows him to remove and return values at specific indices. Since you go to the Number 1 public university in the United States, he requests you to implement `remove(int index)` which removes and returns the element at the index. Assume index is in [0, size) and that the method in part a works as intended.

```
public int remove(int index) {
    int element = get(index);
    for (int i = index + 1; i < size; i++) {
        items[i - 1] = get(i);
    }
    removeLast();
    return element;
}
```

3 ArrayLists vs LinkedLists

Consider the following scenarios. Choose between a LinkedList or an ArrayList implementation, and explain your reasoning.

(a) Keeping a list of the current stock of products in a supermarket where each stock item is numbered.
An ArrayList would work better since it is expected that we will need to index into different elements of the list frequently to view the current stock of a product.

(b) Managing a list of unprocessed orders at a fast food restaurant.
A LinkedList can be easily used to remove the first (oldest) order after it has been processed and add new orders to the end of the list, which is what we need. An ArrayList would be inefficient because every time the first entry (oldest order) is removed, all the next entries need to be shifted up by one index value.

(c) Keeping track of the grades you have for each class as you progress through the semester.
An ArrayList would allow you to easily index and modify the values as your grade changes throughout the semester. (Hopefully going up!)