Extra exercises – Week 5

Array Lists and Linked Lists

Questions on Array Lists

1. The `add(i, e)` method says that, when the array is full (`size == a.length`), it should first copy all the elements to bigger array, and then insert the new element at slot `i` in the new array. This is slightly inefficient since adding the element could be done *while* copying. Write out such a (more efficient) method. Assume the underlying array is `a[ ]`. (This is a good exercise for practicing indices.)

2. If you just want to add an element to a list and you don't care where the element goes, then the simplest approach is to add the element at the end of the list. Or, you just might *want* to add an element at the end of a list. Either way, the method is `add(e)`. Write out the method. Assume the underlying array is `a[ ]`.

3. Give an algorithm for reversing all the elements of an array list which uses a constant amount of additional space (other than the array itself). That is, you are not allowed to use a new array for your solution.

   The main idea is to use a 'swap' method which you should be familiar with from COMP 202.

   ```
   swap(j, k){
     tmp   =  a[j]
     a[j]  =  a[k]
     a[k]  =  tmp
   }
   ```

4. Give a $O(N)$ algorithm for removing the first instance of a given object `e` in a list, assuming the list is represented as an array list and the size of the list is N. That is, give an algorithm for `remove(e)`. In your answer, you can use methods given in the lecture notes.

   In the lecture notes, I presented a `remove(i)` algorithm which removes the element at index i in the list. Here I'm asking for a `remove(e)` algorithm which removes the first instance of object in the list, if at least one instance is present.

5. An important property of arrays is that they have constant time access. This property follows from the fact that array slots all have the same size and the address of any slot is the address of the first slot in the array plus some multiple of the array index, where the multiple is the *constant* amount of memory used by each slot.

   What about an array of strings, in which the strings have possibly different lengths? Does this contradict the property just mentioned? Does one still have constant time access to an element in an array of strings?

6. Suppose we wish to make an array list, and we have n elements that we would like to add. Let's start with an underlying array of length 1. (This makes the math easier.) We then do this:

   ```
   for i = 1 to n
      add( e_i )
   ```

   where `e_i` refers to the ith element in the list. Here I am assuming that these elements already exist somewhere else and I am just adding them to this new array list data structure. Note that the `add( e_i)` operation adds to the end of the current list.

   (a) How much work is needed to do this? In particular, how many times will we fill up a smaller array and have to create a new array of double the array size? How many copies do we need to make in total from each full small array to each new larger (2x) array? It requires just a bit of math to answer this question, and it is math we will see several times in this course.

   (b) What is the advantage or disadvantage of this doubling scheme, instead of just using a huge array to start?

   (c) Java's ArrayList class increases the new array by 50% when it needs more space. Suppose we fill up the array k times, i.e. we have to expand it k times. What is the length of the array you end up with. Ignore the rounding off errors for simplicity.

Questions on Linked Lists

1. See the code: `http://www.cim.mcgill.ca/~langer/250/LinkedList_Exercises.zip`. You will need to put this code into the correct packages.
   The `stub` code contains the questions. The non-`stub` code contains the solutions.

   (a) Fill in the missing code of the following methods in the `SLinkedList_stub.java` class :

   - add(int i, E element)
   - getIndexOf(E e)
   - remove(int i)
   - getNode(int i)

   (b) (More challenging) Fill in the code of the method `reverse()` which reverses the order of elements in a *singly linked list*. The idea of the method is to reverse the order of the nodes (not reversing the elements), by changing `next` references so that they go in the opposite direction in the list. The `head` and `tail` references must be swapped too.

   The idea of the solution is to iterate from the `head` node to the `tail` node. While doing so, partition of the nodes into two lists, namely (1) the (reversed) nodes up to the current node and (2) the not-yet-reversed nodes beyond the current node. The heads of the two lists are `headList1` and `headList2`.

   You may find it helpful to visualize the linked list by drawing boxes (nodes) and arrows, as done in the lectures. Doing it in your head is likely to be too difficult.

2. Implement the following methods in the `DLinkedList_stubs.java` class:

   - `remove(int i)`

     This method first calls `getNode(int i)` which returns a reference to a node. `getNode(int i)` was discussed in detail in lecture 6. `remove(i)` then removes this node from the list, and this is the part you need to implement.

   - `addBefore(E e, DNode<E> node)`

     This is a private helper method which is called by various add methods.

- `reverse()`

  i.e. same as in Questions 2 and 3, but now with a doubly linked list.

3. Consider the Java code:

```
public void display( LinkedList<E>  list ){
   for (int i = 0;  i < list.size(); i++){
       System.out.println( list.get(i).toString() );
   }
}
```

   How does the number of steps of this method depend on `size`, the number of elements in the list?

   (a) Consider the case that the `get(i)` method starts at the front of the list.

   (b) Consider the case that the `get(i)` method will start from the tail of the list in the case that i is greater than `size`/2.

4. Can you have a loop in a singly linked list? That is, if you follow the `next` references, then can you reach a node that you have already visited (and hence loop around infinitely many times if you keep advancing by following the `next` reference) ?

5. Suppose you have a reference to a node in a singly linked list and this node is not the last one in the list.

   (a) How could you remove the element at this node from the list, while maintaining a proper linked list data structure? Note that this would reduce the number of nodes by 1. Your solution should not require looking for this node by starting at the head of the list, but rather you must do it in constant time i.e. O(1). The solution is just a few lines of code. Don't peek!

   (b) How could you insert an element into the list at the position that comes *before* the element at this given node?

Solutions to questions on Array Lists

1.
```
 if (size == length){
    b = new array with a bigger length (say twice as big)
    for (int j=0; j < i; j++)
       b[j] = a[j]

    b[i] = e           //  adding e to slot i

    for (int j = i; j < size; j++)
       b[j+1] = a[j]          // indices are  shifted

    a = b
    size = size + 1
 }
```

2.
```
if (size == length){
   //  same as above,  make a bigger array and copy into it
   b = new array with a bigger length (say twice as big)
   for (int j=0; j < i; j++)
       b[j] = a[j]
}
a[size] = e          //  insert into now empty slot
size = size + 1
```

3. The algorithm then swaps the first and the last, the second and second last, etc.

```
reverseArrayList() {
   for (i = 0; i < size/2; i++){
      swap(i, size-1-i)
}
```

If the list has a odd number of elements, then it doesn't touch the middle one, which is fine. For example, consider the case $i = 13$. It swaps 0,1,2,3,4,5 with 12,11,10,9,8,7, respectively, and doesn't touch 6.

4. The following algorithm loops through the list and examines each element at most once. Hence it takes time proportional to N in the worst case, and so we say it is $O(N)$.

Note that the code below is pseudocode! In Java, you might want to check if the object "equals" e, in the Java sense of equals.

```
remove( e ) {
  i = 0
  found = false
  while ((i < size) and (found == false)){
     if a[i] == e
        found = true
        return remove(i) //  this method was discussed in the lecture
     else
        i++    //  means  i = i + 1
  }
  print("Could not find it.")
}
```

5. The slots in the "array of strings" don't contain strings. Rather they contain references to strings, that is, they contain addresses of strings. The references themselves are all the same size. The strings may be different size and each is located somewhere (at some address) in memory. Bottom line: the slots are all the same size, so you get constant time access to any slot, regardless of how many slots there are.

6. (a) If you double the array capacity k times (starting with capacity 1), then you end up with array with 2^k slots. So let's say $n = 2^k$, i.e. $k = \log_2 n$. That means we double it $k = \log_2 n$ times.

   When we double the size of the underlying array and then fill it, we need to copy the elements from the smaller array to the lower half of the bigger array (and then eventually fill the upper half of the bigger array with new elements). When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 (and then add the new element in the array of size 2). When $k = 2$ and we create a new array of size 4, and then copy 2 elements (and then eventually add the 2 new elements). The number of copies from smaller to larger arrays when we double $k - 1$ times is:

   $$1 + 2 + 4 + 8 + \cdots + 2^{k-1}.$$

   Note that the last term is 2^{k-1} since we are copying into the lower half of an array with capacity $n = 2^k$.

   The above expression is a geometric series

   $$1 + x + x^2 + x^3 + \cdots + x^{k-1} \;=\; \frac{2^x - 1}{x - 1}$$

   with $x = 2$, and so the right side is $2^k - 1$ or $n - 1$. So the number of copies from smaller to larger arrays is $2^k - 1$ or $n - 1$. So using the doubling array length scheme, the amount of work you need to do in order to add n items to an array is about twice what you would need to do if the capacity of the array was at least n at the beginning!

   (b) The advantage to using the doubling scheme rather than initializing the array to be huge is that you don't need to know in advance what the number of elements in the list will be.

   (c) When an ArrayList is declared and initialized, memory space for 10 elements is created by default. If we increase our array capacity k times, then, we end up with an array of $10 * \left(\frac{3}{2}\right)^k$ slots. Ignoring rounding off errors (caused since an array length can't be fractional) for simplicity and solving for k, we get, $n = 10 * \left(\frac{3}{2}\right)^k$.

Solutions to questions on Linked Lists

1. See the `SLinkedList.java` file.

   For the `reverse()` method, see the figures and description here:
   `http://www.cim.mcgill.ca/~langer/250/E3-slinkedlist-reverse.pdf`

2. See the `DLinkedList.java` file.

3. (a) If the `get(i)` method always starts at the front of the list, then it requires `i` steps to reach node `i`. So the total number of steps when calling `get(i)` for `i` in 0 to N-1 is

   $$(1 + 2 + 3 + ... + N) \;=\; \frac{N(N+1)}{2}$$

   which is $O(N^2)$. You might ask whether the sum should be

   $$(0 + 1 + 2 + 3 + ... + N - 1) \;=\; \frac{N(N-1)}{2}$$

   and that would be fine. It is also $O(N^2)$.

(b) For any index i the first half of the list, it takes i steps to get to the node. So the number of steps total for nodes in the first half of the list is:

$$(1+2+3+...+\frac{N}{2}) \;=\; \frac{\frac{N}{2}(\frac{N}{2}+1)}{2}$$

For nodes in the second half of the list, we start from the tail instead of the head, but the idea is the same, so it takes

$$(1+2+3+...+\frac{N}{2}) \;=\; \frac{\frac{N}{2}(\frac{N}{2}+1)}{2}$$

steps in total to reach those nodes. Thus, in total the number of steps is the sum of the above, or

$$\frac{N}{2}(\frac{N}{2}+1).$$

This is about twice as fast as using the inefficient `getNode()` method, but it is still $O(N^2)$.

4. Linked list data structures do allow for a loop, in the sense that there is nothing stopping the `next` field of some node from referencing a node earlier in the list. However, in this case, the data structure will not be a "list", in the sense of having a well defined ordering from 0, 1, ..., size -1. If the linked list class is properly implemented, then the methods should not allow this to happen.

5. (a) Let `cur` be the reference to the given node.

```
cur.element = cur.next.element   //   Copy the element at the next node
//   back to the current one.
cur.next = cur.next.next         //   Skip over the next node.
```

This reduces the size of the list by 1 and removes the element that had been at the current node. The node that was removed still references the next node. That's easy to fix by inserting the following in a suitable place (see comment).

```
tmp      = cur.next     //  Insert after the first instruction above.
tmp.next = null         //  Insert after the second instruction above.
```

(b) Here the idea is similar. We insert a node after the current node and let that node's next field point to the same not as `cur.next`

```
 tmp  = new node
 tmp.next = cur.next
```

Then, change the next field of the current node to point to the new node. Finally, move the elements to their appropriate nodes.

```
   cur.next = tmp
   tmp.element = cur.element
   cur.element = new element // the one to be added
```