Questions

1. The `add(i, e)` method says that, when the array is full (`size == a.length`), it should first copy all the elements to a 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 pseudocode for such a 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 pseudocode for such a 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.

   ```cpp
   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.

   In the lecture, 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. (More challenging – this is the one mentioned in the lecture)

   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:

   ```cpp
   for i = 1 to n
       add( e_i )
   ```
where \( e_i \) refers to the \( i \)th 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 \( \text{add}(e_i) \) 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.
Answers

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 < size; j++)
      b[j] = a[j]
}

   a = b
   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
}