Lecture 4

ArrayList and Binary Search
An idea of dynamic array and faster search

Last time:

We saw arrays in Java along with conceptual view of arrays.

1. There are a few important methods with java.util.Arrays class : equals(), toString(), sort() and copyOf()
2. There are a few different ways of cloning arrays, using for loop, System.arraycopy() or clone().
3. Arrays might be a good choice when:
    - The amount of data is reasonably small
    - The amount of data is predictable in advance
4. This restriction is posed by its limitation in terms of length. It has only one immutable field and, once it is created, the length is fixed and cannot be changed.
5. As a result, an array can only hold a certain number of items.
6. Search operation on an unordered array is linear search.

In many cases, it would be great if we could increase or decrease the size of an array on demand.

ArrayList class does support an idea of a dynamic array that grows and shrinks (?) on demand to have flexible number of elements in the array.

Now, let's take a look at java.util.ArrayList class!

Looking at ArrayList class's Java doc, you can find that there are 20 methods that you can call.

The followings are the methods that are commonly used.

- add(object) : adds a new element to the end
- add(index, object) : inserts a new element at the specified index
- set(index, object) : replaces an existing element at the specified index with the new element.
- get(index) : returns the element at the specified index.
- remove(index) : deletes the element at the specified index.
- size() : returns the number of elements.

Example of a few methods' usage

```
// Initialize an arraylist with initial size of 0
ArrayList<Integer> number = new ArrayList<Integer>(0);
// add numbers
for(int i=0; i<10; i++) number.add(i);
System.out.println(number);

// delete some numbers
for(int i=0; i<number.size(); i++)
     if(number.get(i)%2==0) number.remove(i);
System.out.println(number);
```

Look! It dynamically changes the size! How?
The answer is quite simple. The ArrayList used to implement the doubling-up policy. (In Java 6, there has been a change to be (oldCapacity*3)/2 +1). Here is the code snippet of how it is done in Java.

```
/**
 * Appends the specified element to the end of this list.
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacity(size+1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

```
/**
 * Increases the capacity of this <tt>ArrayList</tt> instance, if
 * necessary, to ensure that it can hold at least the number of
elements
 * specified by the minimum capacity argument.
 *
 * @param   minCapacity   the desired minimum capacity
 */
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity*3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

However, the removal in Java is not that efficient at all due to the array's nature; "No holes allowed." The ArrayList needs to shift all of the elements in the array EVERY TIME the remove(int index) or remove(Object o) method is called.

Also, those remove methods do not reduce the size of the underlying array, which can be a concern in terms of memory usage. But, good news is that arrays only hold references to each object and that is not big.

If you had any issue with this unused memory, you need to manually call trimToSize() method to free the memory.

Running time complexity analysis of add(E e) method : adding a new element to the end of an array: Amortized Analysis

For the sake of simplicity, let's assume that we are doubling up the array. As we saw in the last lecture, adding a new element at the end of an array takes constant time: O(1). However, when the array is full and still need to add a new element, we then need to create a new array whose length is twice bigger than the original array and copy all of the elements into the new array.

```
ArrayList<Integer> number = new ArrayList<Integer>(4);
```

	Running time	# of elements	Array length
number.add(1)	1	1	4
number.add(2)	1	2	4
number.add(3)	1	3	4
number.add(4)	1	4	4
number.add(5)	**5**	**5**	**8**
number.add(6)	1	6	8
number.add(7)	1	7	8
number.add(8)	1	8	8
number.add(9)	**9**	**9**	**16**

From the table above, you can see that adding 5 and 9 would take longer than the other calls. But, all of the other calls take constant time. We can use so called banker's (accounting) method to show amortized analysis.

	Running time	# of elements	Array length	**Allocated dollars**	Cost	Saved dollars	Balance
	1	1	4	**3**	1	2	2
	1	2	4	**3**	1	2	4
	1	3	4	**3**	1	2	6
	1	4	4	**3**	1	2	8
	5	**5**	**8**	**3**	**5**		
	1	6	8	**3**	1	2	8
	1	7	8	**3**	1	2	10
	1	8	8	**3**	1	2	12
	9	**9**	**16**	**3**	**9**		

We can conclude that add(E e) method has constant amortized time by looking at allocated dollars per operation.

Amortized analysis (continued)

Revisiting search on arrays.

****
"I've assigned [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert [its] description into a program in the language of their choice; a high-level pseudo code was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found). I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962."

—Jon Bentley, Programming Pearls (1st edition), pp.35–36
****

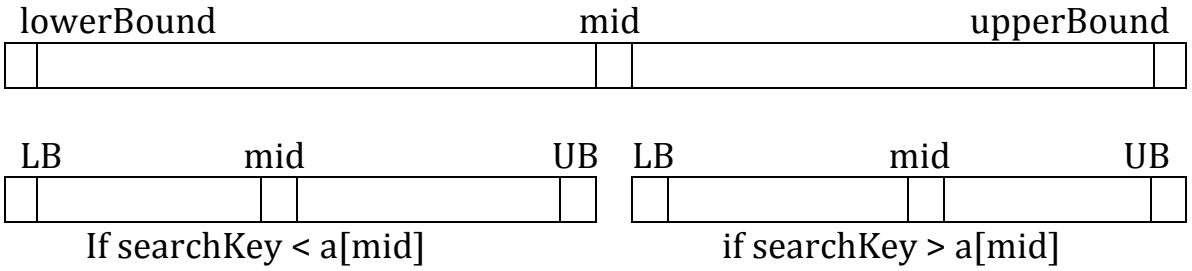
In lecture 3, we saw that searching on unordered arrays can be done by linear search algorithm. This would be fine if arrays are small but how about large arrays. Can we improve it?

Let's play the Guess-a-Number Game first.

Let's look into details about how this can be implemented!

Prerequisite for the binary search: The array should already be ordered.

<Conceptual View>



Search for 6 in a given ordered array of [1,2,3,4,5,6,7,8,9]

We need three variables:

```
int upperBound = array.length-1; // value is 8
int lowerBound = 0;
int mid = (upperBound+lowerBound)/2; // value is 4
```

Compare 6 with the middle element at index 4, which is 5. Since 6 > 5, we set:

```
// upperBound value not changed. (8)
int lowerBound = mid+1; // now set to be 5
int mid = (upperBound+lowerBound)/2; // value is 6
```

and start a new iteration. Compare 6 with the middle element at index 6, which is 7. Since 6 < 7, we set:

```
int upperBound = mid-1; // now set to be 5
// upperBound value not changed. (5)
int mid = (upperBound+lowerBound)/2; // value is 5
```

and start a new iteration. Compare 6 with the middle element. Bingo!

How many comparisons do we need with linear search?

How many comparisons do we need with binary search?

Example for you!

Search for -1 in a given array [-34,-23,-9,-1,5,7,8,9,34,68,88,99]

Set initial variables:
upperBound = _____; //
lowerBound = 0; // 0
mid = (upperBound+lowerBound)/2; //

<Implementation View>

```
public int binarySearch(int[] data, int key) {
      int lowerBound = 0;
      int upperBound = data.length-1;
      int mid;

      while(true) {
            if(____________________________) return -1;
            mid = ____________________________________;
            if(data[mid] == key) return _______________;
            else {
                  if(data[mid] < key)
                        __________________ = _________________;
                  else
                        __________________ = _________________;
            }
      }
}
```

Looks good, right!

We got a bug-free binary search implemented in about 5 minutes.

We are smarter than those professional programmers at Bell Labs and IBM.

Are you sure?

Time Complexity of Binary Search
Remember that we consider the worst-case scenario!

What is the worst-case?

How many times do we need to split the array in half before we cannot split anymore?

For the previous example of an array (length of 12):
11 → 5 → 2 → 1 → 0 (4 times)

In general, we can split an array in half $\lfloor \log_2 N \rfloor + 1$ times before it becomes empty

Thus, when there are 12 elements, we need 4 comparisons (or splits):
$\lfloor \log_2 12 \rfloor + 1 = 4$

As a result, we can say that binary search requires O(log N) time on a sorted array with n elements.

Number of elements	Number of comparisons
15	4
31	5
63	6
127	7
255	8
511	9
1,023	10
…	…
1,000,000	20

As you can see, finding an element in an array of a million elements requires 20 comparisons. Only 20 comparisons! This is SIGNIFICANT!

Practical Advice: Pay attention to constant!
Big O notation does not necessarily tell the whole story.

What to choose from the followings?
$T(N) = N \log N$
$U(N) = 50N$

Advantages of Ordered Arrays

It is useful in situations in which searches are frequent but insertions and deletions are not.

We talked about the prerequisite for binary search; the array is already sorted.

Well, we need to think about sorting and its running time then, right?
We will take a look at it later!