Lecture 4

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

Last time:

We looked into arrays in Java along with conceptual view of arrays.

1. We saw some methods in `java.util.Arrays` class: `equals()`, `toString()`, `sort()` and `copyOf()`.
2. There are a few other 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 is an 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 based on linear search.

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

ArrayList class supports 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 the JavaDoc of the ArrayList class, you can find that there are many methods that you can use.

The following is a list of the most commonly used ones:

- 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 length of 0
List<Integer> numbers = new ArrayList<Integer>(0);
// add numbers
for (int i = 0; i < 10; i++) numbers.add(i);
System.out.println(numbers);

// delete some numbers
for (int i = numbers.size() – 1; i >= 0; i--) {
     if (numbers.get(i) % 2 == 0) numbers.remove(i);
}
System.out.println(numbers);
```

Look! It dynamically changes the length! 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 in Java 6.

```
/**
 * 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 due to the nature of array; “No holes allowed.” The ArrayList needs to shift many elements in the array EVERY TIME the `remove(int index)` or `remove(Object o)` method is called (except removing the last element).

Also, those remove method calls do not reduce the length of the underlying array, which can be a concern in terms of memory usage.

If you had any issue with this unused memory, you need to manually invoke _____ method to free up 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 we still need to add a new element, we then need to create a new array whose length is twice bigger than that of the original array AND copy all the elements of the old array into the new array.

```
List<Integer> numbers = new ArrayList<Integer>(4);
```

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

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 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	**8**	**3**			
1	6	8	**3**	1	2	8
1	7	8	**3**	1	2	10
1	8	8	**3**	1	2	12
	9		**3**			

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

Amortized analysis (continued)

Revisiting search

****
“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
****

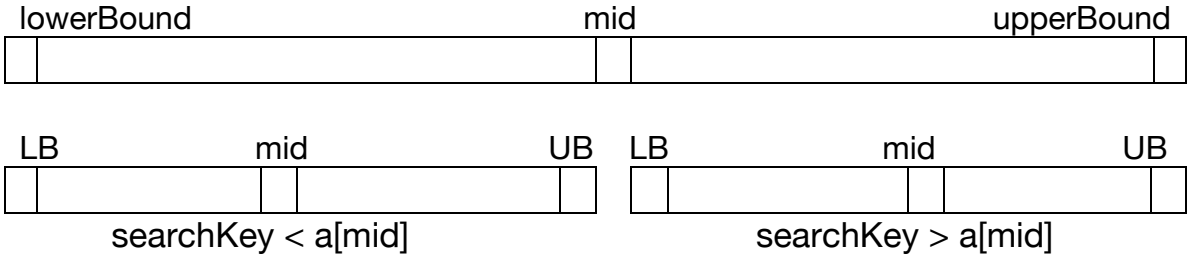
In lecture 3, we saw that searching unordered arrays for a value can be done by linear search algorithm. This would be fine if arrays are small but how about large arrays. Can we improve it?

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
// lowerBound 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 static 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 _________________;
            }

            if (data[mid] < key) {
                  ___________________ = _________________;
            } else {
                  ___________________ = _________________;
            }
      }
}
```

Looks good, right?

We got a bug-free binary search implemented in about 5 minutes, didn't we?

We can say that 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 array (length of 12):
11 → 5 → 2 → 1 → 0 (4 times)

In general, we can split an array in half for $\lfloor \log_2 n \rfloor + 1$ 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) = 50*n

Advantages of Ordered Arrays

It may be useful in situations where searches are frequent but insertions and deletions are not.

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

Unless we keep the order as we insert elements, we would need to think about sorting algorithm and its running time, right? We will take a look at it later.