

CSE 143

Lecture 4

Implementing `ArrayIntList`; Binary Search

reading: 15.1 - 15.3

slides adapted from Marty Stepp and Hélène Martin

<http://www.cs.washington.edu/143/>

Exercise

- Let's write a class that implements a list using an `int[]`
 - We'll call it `ArrayIntList`
 - its behavior:
 - `add(value)`, `add(index, value)`,
 - `get(index)`, `set(index, value)`,
 - `size()`, `isEmpty()`,
 - `remove(index)`,
 - `indexOf(value)`, `contains(value)`,
 - `toString()`,
 - ...
 - The list's *size* will be the number of elements added to it so far.
 - The actual array length ("capacity") in the object may be larger. We'll start with an array of **length 10** by default.

Implementing add

- How do we add to the end of a list?

```
public void add(int value) { // just put the element
    list[size] = value;      // in the last slot,
    size++;                 // and increase the size
}
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

- list.add(**42**);

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	42	0	0	0
<i>size</i>	7									

Implementing add #2

- How do we add to the middle or end of the list?
 - must *shift* elements to make room for the value (*see book 7.3*)

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

– `list.add(3, 42);` **// insert 42 at index 3**

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
<i>value</i>	3	8	9	42	7	5	12	0	0	0
<i>size</i>	7									

– Note: The order in which you traverse the array matters!

add #2 code

```
public void add(int index, int value) {  
    for (int i = size; i > index; i--) {  
        list[i] = list[i - 1];  
    }  
    list[index] = value;  
}
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

- list.add(3, 42);

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	42	7	5	12	0	0	0
<i>size</i>	7	→								

Other methods

- Let's implement the following methods in our list:
 - `get(index)`
Returns the element value at a given index.
 - `set(index, value)`
Sets the list to store the given value at the given index.
 - `size()`
Returns the number of elements in the list.
 - `isEmpty()`
Returns `true` if the list contains no elements; else `false`.
(Why write this if we already have the `size` method?)

Implementing `remove`

- How can we remove an element from the list?

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

- `list.remove(2);` **// delete 9 from index 2**

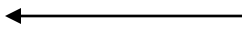
<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	7	5	12	0	0	0	0	0
<i>size</i>	5									

Implementing `remove`, cont.

- Again, we need to shift elements in the array
 - this time, it's a left-shift
 - in what order should we process the elements?
 - what indexes should we process?

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

– `list.remove(2);` **// delete 9 from index 2**

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	7	5	12	0	0	0	0	0
<i>size</i>	5 									

Implementing remove code

```
public void remove(int index) {  
    for (int i = index; i < size; i++) {  
        list[i] = list[i + 1];  
    }  
    size--;  
    list[size] = 0;        // optional (why?)  
}
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

- list.remove(2); // delete 9 from index 2

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	7	5	12	0	0	0	0	0
<i>size</i>	5									

←

Printing an ArrayList

- Let's add a method that allows clients to print a list's elements.

- You may be tempted to write a `print` method:

```
// client code
```

```
ArrayList list = new ArrayList();
```

```
...
```

```
list.print();
```

- Why is this a bad idea? What would be better?

The toString method

- Tells Java how to convert an object into a String

```
ArrayList list = new ArrayList();  
System.out.println("list is " + list);  
                // ("list is " + list.toString());
```

- Syntax:

```
public String toString() {  
    code that returns a suitable String;  
}
```

- Every class has a `toString`, even if it isn't in your code.
 - The default is the class's name and a hex (base-16) number:

```
ArrayList@9e8c34
```

toString solution

// Returns a String representation of the list.

```
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "];";
        return result;
    }
}
```

Searching methods

- Implement the following methods:
 - `indexOf` - returns the first index an element is found, or -1 if not
 - `contains` - returns true if the list contains the given int value
- Why do we need `isEmpty` and `contains` when we already have `indexOf` and `size` ?
 - Adds convenience to the client of our class:

```
// less elegant
```

```
if (myList.size() == 0) {  
    if (myList.indexOf(42) >= 0) {
```

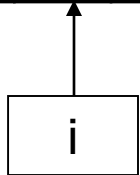
```
// more elegant
```

```
if (myList.isEmpty()) {  
    if (myList.contains(42)) {
```

Sequential search

- **sequential search:** Locates a target value in an array / list by examining each element from start to finish. Used in `indexOf`.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- Notice that the array is sorted. Could we take advantage of this?

Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array / list by successively eliminating half of the array from consideration.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating the binary search process on the array above. The search range is defined by **min** (index 0), **mid** (index 8), and **max** (index 16). The target value **42** is located at index 10.

Arrays.binarySearch

```
// searches an entire sorted array for a given value  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted
```

```
Arrays.binarySearch(array, value)
```

```
// searches given portion of a sorted array for a given value  
// examines minIndex (inclusive) through maxIndex (exclusive)  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted
```

```
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.
 - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)

Using `binarySearch`

```
// index    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index = Arrays.binarySearch(a, 0, 16, 42); // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21); // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:
 - (`insertionPoint` + 1)
 - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
 - To insert the value into the array, negate `insertionPoint` + 1

```
int indexToInsert21 = -(index2 + 1); // 6
```