NC STATE UNIVERSITY

Array Based Lists

Reading
Zybooks 13.1 & 13.2

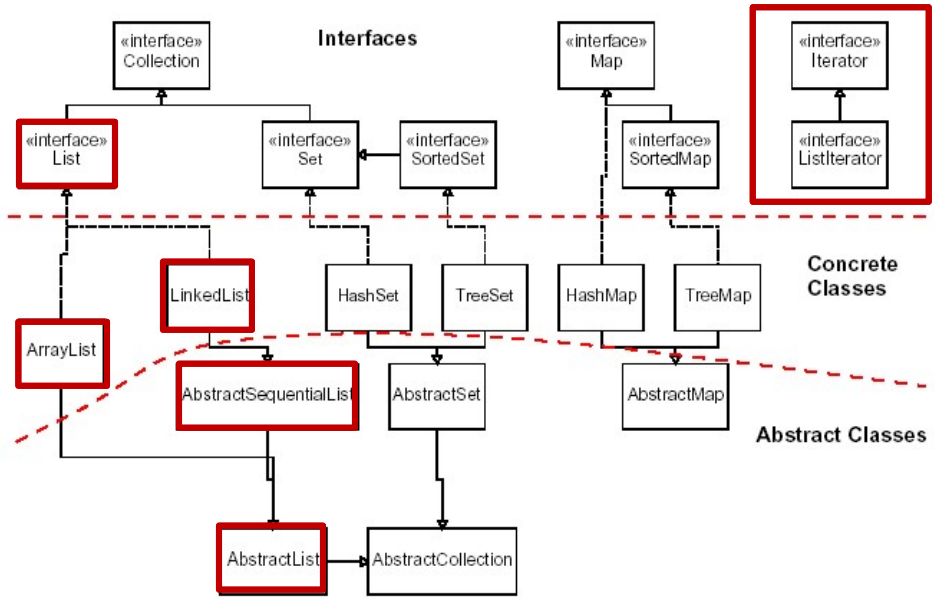
2

NC STATE UNIVERSITY

Collections

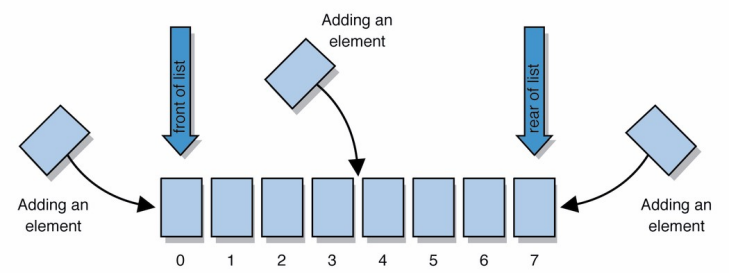
- Data structures – stores elements in a manner that makes it easy for a client to work with the elements
- Specific collections are specialized for particular types of operations
  - Ordering
  - Duplicates
  - Add/remove
- Common set of operations: add, remove, get, contains, size

3



Lists

- List: an ordered collection of elements
  - Accessible by a zero-based index
  - Size: number of elements in the list
  - Elements may be added to *an empty list*, at the *front*, *middle*, and *back*



NC STATE UNIVERSITY

Scenario

- Suppose we had a file of integers, representing grades, counts, etc.
- We want to calculate metrics and manipulate the list of ints
  - Average, min, max, quartiles, print them in reverse, remove even numbers, etc.
- We would use an ArrayList to solve
  - "Automatically" grows
  - Simplifies working with an array
- But how does it really work? How is the ArrayList class implemented?

6

NC STATE UNIVERSITY

Working with ArrayList

An ArrayList is working with an array. But how?

- Constructing

```
ArrayList<Integer> list =
                new ArrayList<Integer>();
```

- Adding an element to the end of the list

```
list.add(42);
```

- Adding an element at an index

```
list.add(3, 42); //(index, element)
```

- Removing an element

```
list.remove(3); //(index)
```

- Getting an element from an index

```
list.get(3); //(index)
```

- Constructing

```
int [] list = new int[?];
//how many elements?
```

- Adding an element to the end of the list

```
list[?] = 42;
//what index is the end of the list?
```

- Adding an element at an index

```
list[3] = 42;
//what happens to the rest of the list?
```

- Removing an element

```
int v = list[3]; //rest of the list?
```

- Getting an element from an index

```
int v = list[3];
```


7

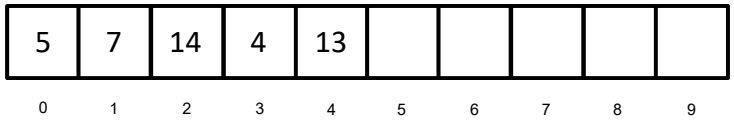
NC STATE UNIVERSITY

Partially Filled Arrays

- ArrayLists use the concept of partially filled arrays to simplify implementation of common client functionality
  - There are no gaps in values
  - Values are stored between indices 0 and size-1, inclusive

```
int [] list = new int[10]; //capacity is 10
int size = 0; //size is the number of spots used
list[size++] = 5;
list[size++] = 7;
list[size++] = 14;
list[size++] = 4;
list[size++] = 13;
```

size 5



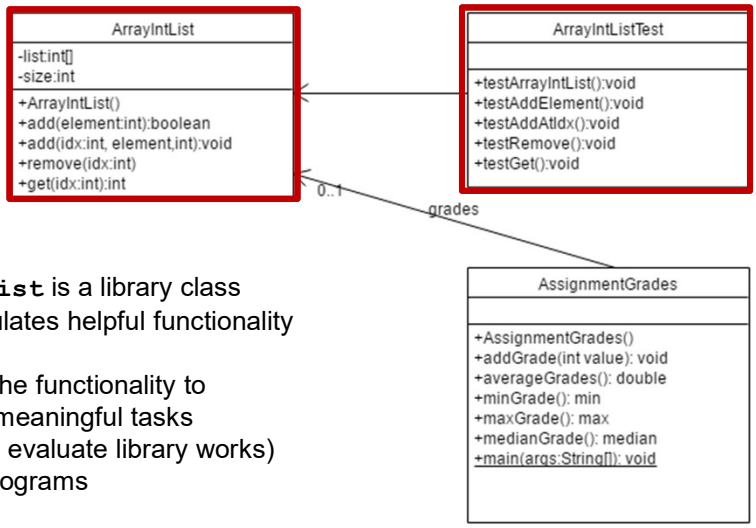
NC STATE UNIVERSITY

ArrayList (of ints)

- Abstracts the details about how to interact with the *partially filled array* as a *list*
- Encapsulates the array and size
- Provides operations to manipulate the array

```
public class ArrayIntList {
   private int [] list;
   private int size;
   public ArrayIntList() { … }
   public boolean add(int element) { … }
   public void add(int idx, int element) { … }
   public int remove(int idx) { … }
   public int get(int idx) { … }
   public int size() { … }
}
```

ArrayIntList Design



- **`ArrayIntList`** is a library class
  - Encapsulates helpful functionality
- Client uses the functionality to accomplish meaningful tasks
  - Tests (to evaluate library works)
  - Client programs

Testing Lists

- Test each List method
  - Empty list
  - Front of the list
  - Middle of the list
  - Back of the list
- After each operation
  - Length is correct
  - Contents are correct (are the elements in the right order?)
- Then test the methods in combination
  - maybe `add` usually works, but fails after you call `remove`
  - what happens if I call `add` then `size`? `remove` then `toString`?
  - make multiple calls; maybe `size` fails the second time only

Testing `ArrayIntList()`

- Start by testing the constructor
  - Should create an empty list of integers

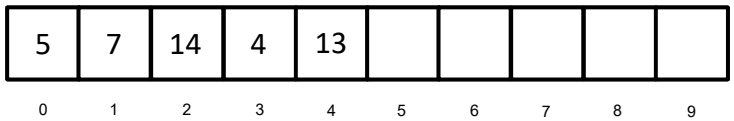
```
public class ArrayIntListTest {
    @Test
    public void testArrayIntList() {
        ArrayIntList l = new ArrayIntList();
        assertEquals(0, l.size());
    }
}
```


Testing `ArrayIntList.add(element)`

- Test adding elements to the end of the list
  - Check size AND that elements are in the right order after the add

```
list.add(13);
```

size 4 5



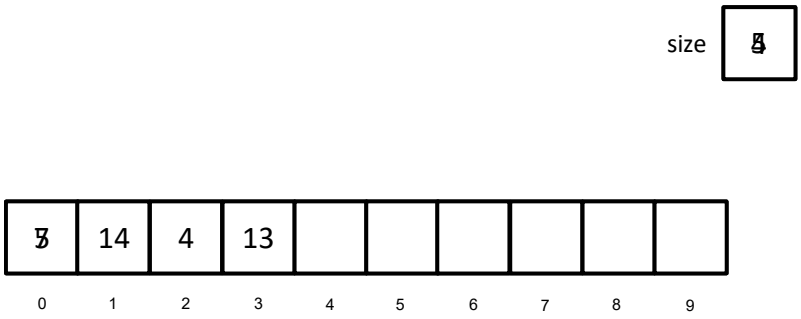
Implementation to Pass Test

- Implement **`add()`**
  - **`size`** is the index where the new last element will go
  - Don't forget to increment size!
- Implement **`size()`** & **`get()`**
  - Straightforward methods & useful for testing other functionality
  - Implement the main functionality
    - you can ignore error checking/handling… for now
    - We will test **`get()`** a bit more thoroughly later

Testing `ArrayIntList.add(idx, e)`

- How do we add to the front and middle of the list?

```
list.add(0, 5);
```

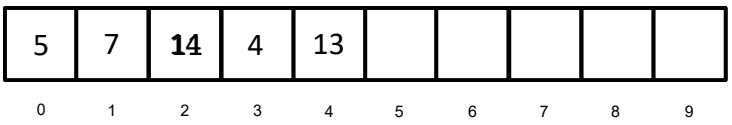


Testing `ArrayIntList.add(idx, e)`

- How do we add to the front and middle of the list?

```
list.add(0, 5);
list.add(2, 11);
```

size | 6



Implementation to Pass Test

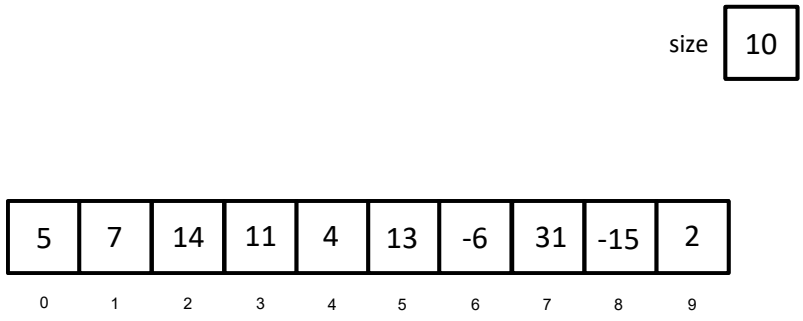
- Implement `add(idx, element)`
  - Right shift – order of the elements you shift matters!

NC STATE UNIVERSITY

Size vs. Capacity

- What happens when a client tries to add an element at capacity+1?

```
list.add(21);
```



NC STATE UNIVERSITY

Reaching Capacity

- What should happen when we reach capacity?
  - Depends on the requirements
- Maybe there is a hard cap
  - Throw an exception
- Maybe the array should "automatically" grow
  - Resize the array
    - Create a new array of larger size – typically double
    - Copy values from old array into new array
    - Update field `list` to refer to the new array

Update to add() Methods

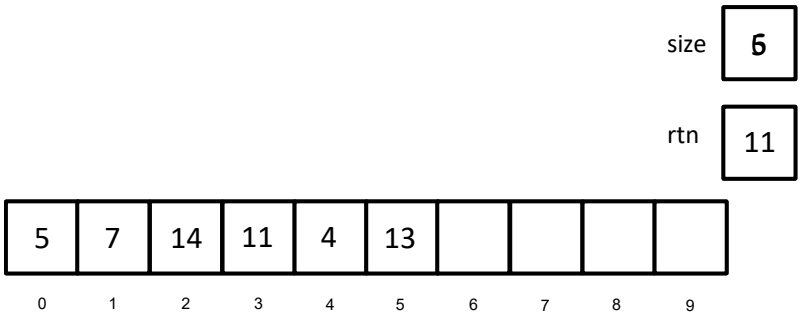
- `add()` methods should check capacity and increase if needed

```
public boolean add(int element) {
    ensureCapacity(size + 1);
    …
}
public void add(int idx, int element) {
    ensureCapacity(size + 1);
    …
}
```


Testing remove(idx)

- How do we remove an element from the list?

```
int v = list.remove(3);
```



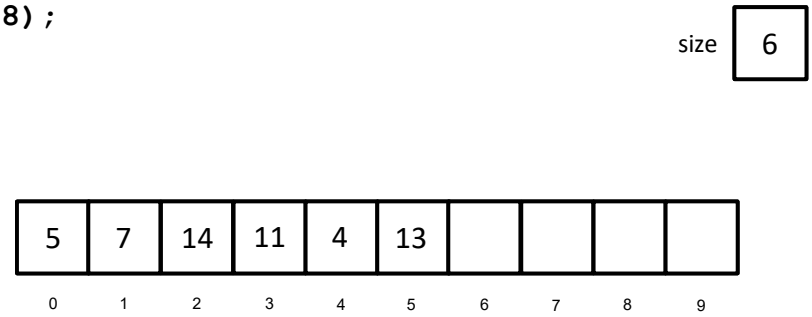
Implementation to Pass Test

- Implement **`remove(idx)`**
  - Left shift – order of the elements you shift matters!

Size vs. Capacity

- What happens if a client tried to access an element in a partially-filled array that is past the size but within the capacity (bounds) of the array?

```
list.get(8);
```



NC STATE UNIVERSITY

Preconditions & Postconditions

- **Precondition:** something that your method assumes is true before starting execution
  - In documentation
  - Enforce?
- **Postcondition:** something that you promise will be true after execution of your method
  - Assuming the precondition is met
  - Test to make sure this is true!

NC STATE UNIVERSITY

Bad Precondition Check

- What is wrong with handling a violation of a precondition in the following manner?

```
//Returns the element at the given index
public int get(int idx) {
  if (idx < 0 || idx >= size) {
     System.out.println("Bad index! " + idx);
     return -1;
  }
  …
}
```

Enforcing the Precondition

- Throw an exception to the client!

```
//Returns the element at the given index
public int get(int idx) {
  if (idx < 0 || idx >= size) {
     throw new IndexOutOfBoundsException();
  }
  …
}
```


Convenience Methods

- Useful methods for the client
  - **`indexOf()`** – returns the first index of the element if found and -1 otherwise
  - **`isEmpty()`** – returns true if size is 0
  - **`contains()`** – returns true if the element is in the list

- Why are they useful to the client?

```
if (list.size() == 0) { … }

if (list.isEmpty()) { … }
```

Implementing your own Lists

- We've reached the point in the semester where using the Java Collections Framework may not be allowed
  - Depends on the project
    - Project 1 – OK
    - Project 2 – NOT OK
    - Labs – NO
- You'll start implementing your own custom lists in the labs!