Review: Continuing from last lecture–memory layouts of lists

WHAT SEQUENCE OF ADDFIRST/ADDLAST CALLS

Consider the following layouts for the list [8, 3, 6, 4] – what program might generate this heap layout?

@1012	**MutableList**(start:@1017)
@1013	Node(item:6, next:@1016)
@1014	Node(item:3, next:@1013)
@1015	Course(name: "CSCI1410", enrollment: 200)
@1016	Node(item:4, next:null)
@1017	Node(item:8, next:@1014)
@1018	

Can follow references to see order of elements in list

ORDER IN HEAP ≠ ORDER IN WHICH OBJECTS WERE CREATED.

[6]

ADDFIRST(6) OR ADDLAST(6)

ADD FIRST(3) [3,6]

ADD LAST(4) [3,6,4]

ADD FIRST(8) [8,3,6,4]

Question: How would this memory layout be different if we were making an *immutable* list with the same sequence of addLast/addFirst calls?

Question: Imagine this list were named `L` in the environment. What sequence of memory objects get visited to compute `L.get(2)` [which should return 6]?

INDEX 0 1 2 3

[8, 3, 6, 4]

Can't just see which element is element 2 by looking at the heap => need to follow the chain of references (many colors or arrows above) to find out!

=> **This means we need to search the whole list, which has linear runtime!**

=> O(N)

Activity: Now imagine the list had the following layout in memory (all the items consecutive and in order). What sequence of memory objects would get visited to compute `L.get(2)`?

[8, 3, 6, 4]

@1012	**ConsecList**
@1013	8
@1014	3
@1015	6
@1016	4
@1017	
@1018	

Where is element 2 in this list?
Just from the picture we can see it's at @1015

Because this implementation has the array elements in consecutive slots, we can figure out element 2's address just by taking the address where the list starts and adding to it:

@1015 = @1012 + 2 + 1

ADDRESS OF ELEMENT 2 — START OF LIST

Therefore, we can implement get(i) by looking up the element at (address of list) + i + 1

=> This just involves adding a constant value to an address => **constant runtime!!**

1 + 1 + 1 => O(1)

Lecture 11: Arrays and ArrayLists

(from last time) Consider the following layouts for the list [8, 3, 6, 4] – what program might generate this heap layout?

Activity: Now imagine the list had the following layout in memory (all the items consecutive and in order). What sequence of memory objects would get visited to compute `L.get(2)`?

	@1012	**ConsecList**
0	@1013	8
1	@1014	3
2	@1015	6
	@1016	4
	@1017	
	@1018	

BASED ON PICTURE, ELEMENT 2 IS AT

@1015 = @1012 + 2 + 1

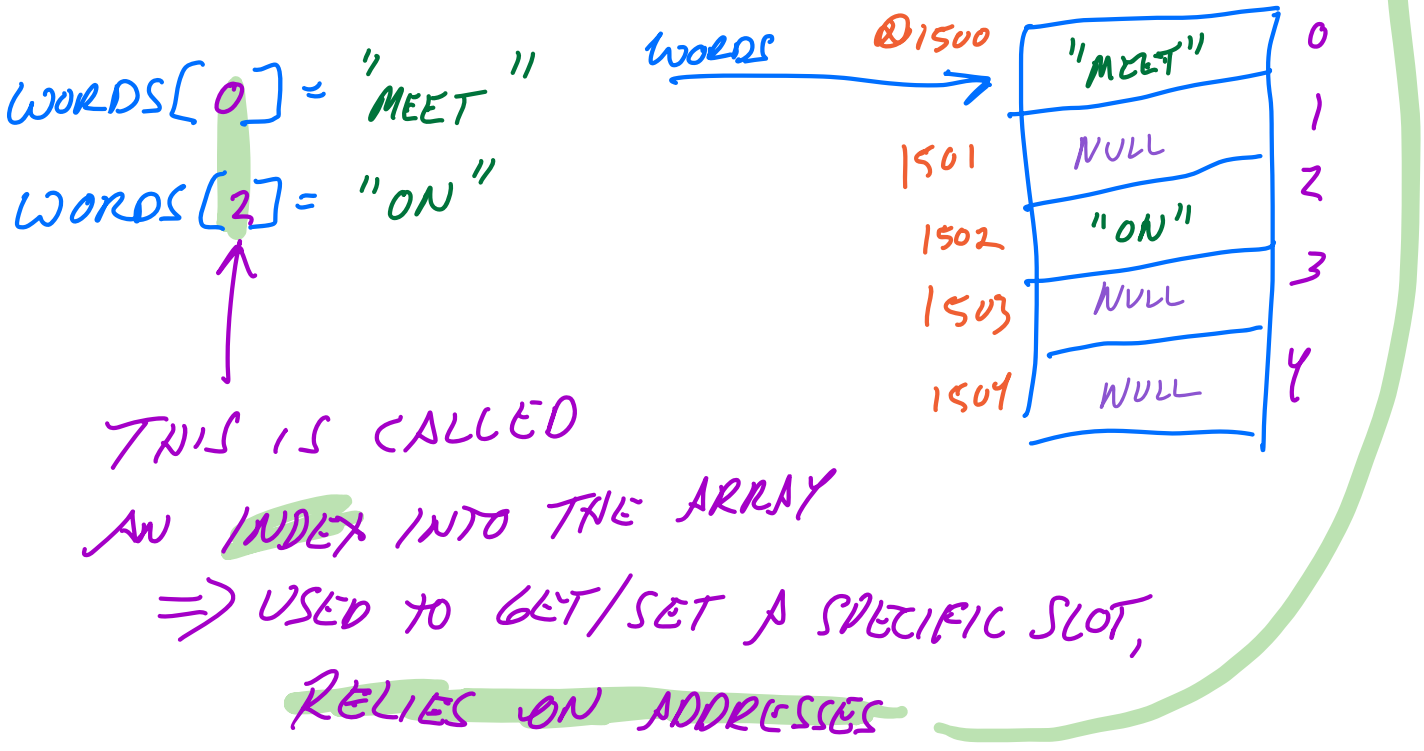
@1015: ELEMENT 2 · @1012: START OF LIST · 2: INDEX · 1: SIZE OF EACH ELEMENT

CONSTANT TIME. ⇒ O(1)

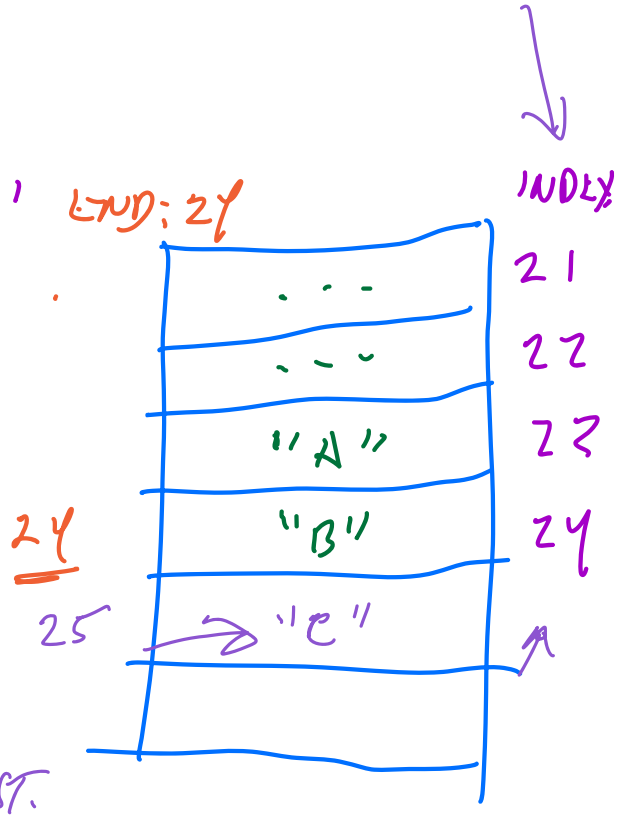
This kind of data structure is called an **array**, which is common to many programming languages. Arrays form the basis of Java's ArrayList (among other types).

AN EXAMPLE:

```
// Make an array with space for 5 strings
String[] words = new String[5];
```

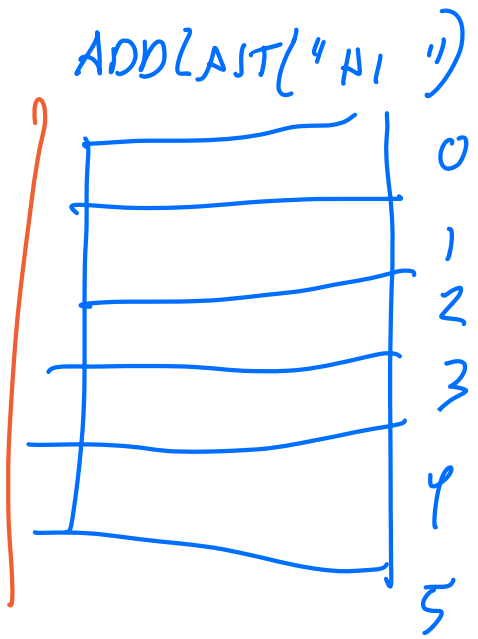


With an array, Java makes all the slots for us ahead of time,
We decide how to use them
Don't need to make nodes to hold objects every time we add something



NOW WOULD WE MAKE ADDFIRST/ADDLAST?

=> CONSIDER WHAT WOULD HAPPEN TO ADD TO THE MIDDLE FIRST.



Adding to a full ArrList

ArrList AL = new ArrList(3)

When we created ArrList, have fixed number of slots

@1012	**ArrList** theArray: @1013 end: 0 eltcount: 2
@1013	"hello"
@1014	"there"
@1015	"brown"
@1016	

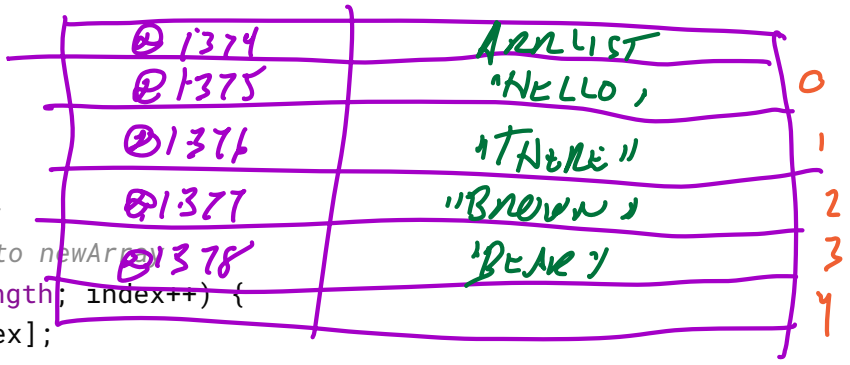
Assume this ArrList is named AL.

Now run `AL.addLast("bear")`

AL.ADDLAST("HELLO")
AL.ADDLAST("THERE")
AL.ADDLAST("BROWN")
AL.ADDLAST("BEAR")

If we want more space, we need to "resize" by getting a new array of larger size, copy everything over, then add new item

WHAT HAPPENS??




```
private void resize(int newSize) {
    // make the new array
    String[] newArray = new String[newSize];
    // copy items from the current theArray to newArray
    for (int index = 0; index < theArray.length; index++) {
        newArray[index] = this.theArray[index];
    }
    // change this.theArray to refer to the new, larger array
    this.theArray = newArray;
}

public void addLast(String newItem) {
    if (this.isFull()) {
        // add capacity to the array
        this.resize(this.theArray.length + 1);
        // now that the array has room, add the item
        this.addLast(newItem);
    } else {
        if (!(this.isEmpty())) {
            this.end = this.end + 1;
        }
        this.eltcount = this.eltcount + 1;
        this.theArray[end] = newItem;
    }
}
```

IF FULL RESIZE

ADDLAST FROM BEFORE

(You don't need to understand all the code here, we just want you to see the shape of it.)