Lecture 12 – ArrayLists and Runtime

Summarize Worst-Case Runtimes (in terms of number of elements in the list)

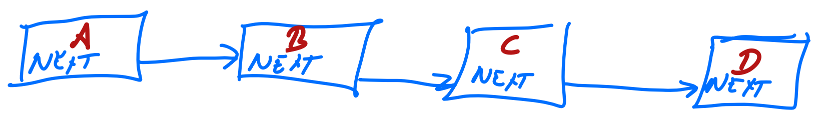
	LinkList	MutableList (Link) (LIKE HW2)	ArrList
size			
addFirst			
addLast			
get(index)	O(N) LINEAR	O(N) LINEAR	O(1) CONSTANT

So far we've seen three ways to look at lists…

SAY WE HAVE LIST WITH STRINGS ["A","B","C","D"]

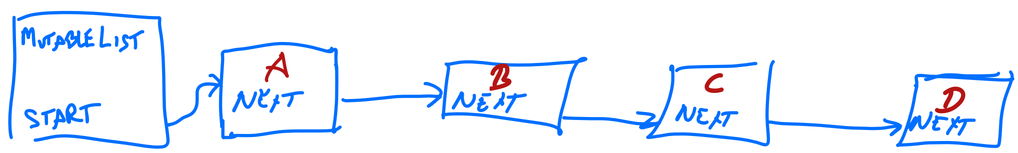
LinkList (or ImmutableList)
- Has a chain of nodes with (at least) a "next" field
- Each node could be at any spot in memory

For get() => Need to follow "chain" of nodes (or Links) to get a specific item
=> Linear runtime over the size of the list => O(N)



MutableList (like HW2)
- Same "chain" of nodes
- MutableList class has "start" field that points to nodes
- MutableList might have other fields like in HW2 (end, etc.)

For get() => same as LinkList => O(N)



ArrList (ArrayList in Java)
- Relies on arrays: at start, reserve a fixed number of consecutive memory slots
- When array is full, resize by creating a new array and copying over all elements

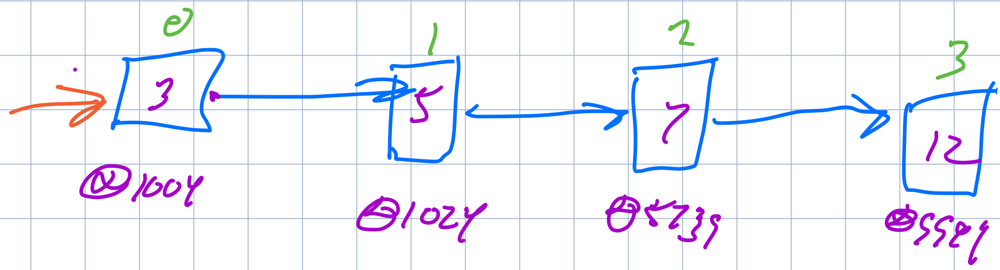
For get() => Since the array elements are always in contiguous memory slots, can look up the i'th element just based on the starting address value.
=> Just add to the starting address => constant time => O(1)

Ex.

GET(2) = @1001 + 2 + 1

(@1001 = START, 2 = INDEX)

[3, 5, 7, 12]



GET(3)

Runtime of AddLast/AddFirst with Resizing

```
public class ArrList {
    String[] theArray;  // the underlying array that stores the elements
    int eltcount;       // how many elements are in the array
    int end;            // the last USED slot in the array

    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[this.end] = newItem;
        }
    }
```

For now, we make a new array 1 larger than the previous one each time we resize.
We could call this the "resize policy" (This isn't a very good one, we'll learn a practical one soon.)

```
public class ArrTest {
    ArrList flavors = new ArrList(2);
    flavors.addLast("mint")
    flavors.addLast("grape")
    new Course("cs1410", 200)
    flavors.addLast("lemon")
    flavors.addLast("cherry")
}
```

---

environment

flavors → @1221

@1221	**ArrList** theArray: @1222 end: 1 eltcount: 2
@1222	"mint"
@1223	"grape"
@1224	Course("cs1410", 200)
@1225	
@1226	
@1227	
@1228	

What's the worst case runtime of addLast?

It's a big more nuanced before, because it depends on if the array is full:

- If array is full => resize => linear time operation (copy)
- If array is not full => constant time (add to a slot)

=> As developers, we want to think about how often we "pay the cost" of resizing

How many resizes get done across N calls to addLast? How does this affect runtime?

```
ArrList flavors = new ArrList(2);
```

	Resize by 1	Resize by 2	Resize by double
`flavors.addLast("mint")`	CONST	CONST	
`flavors.addLast("grape")`	CONST	CONST	
`flavors.addLast("lemon")`	RESIZE	RESIZE (4)	
`flavors.addLast("cherry")`	RESIZE	CONST	
`flavors.addLast("mango")`	RESIZE	RESIZE	
`flavors.addLast("orange")`	RESIZE	CONST	
`flavors.addLast("coffee")`	RESIZE	RESIZE	

SEE NEXT PAGE

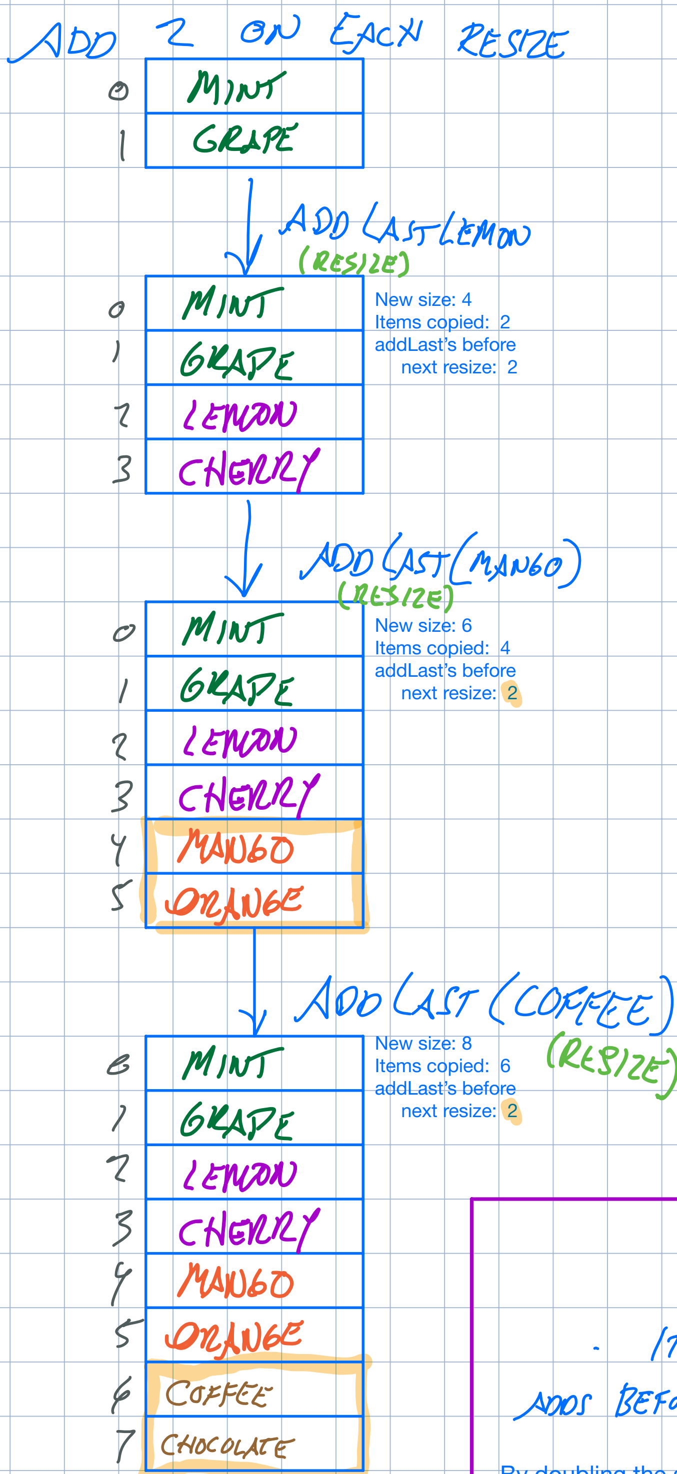
Each resize is linear time due to the copy

For N calls, resize N/2 times => halved runtime cost => still linear runtime O(N)

Overall, It’s helpful to think about the amortized cost, which is the runtime across multiple calls to a method.

What happens in practice (as a general rule)

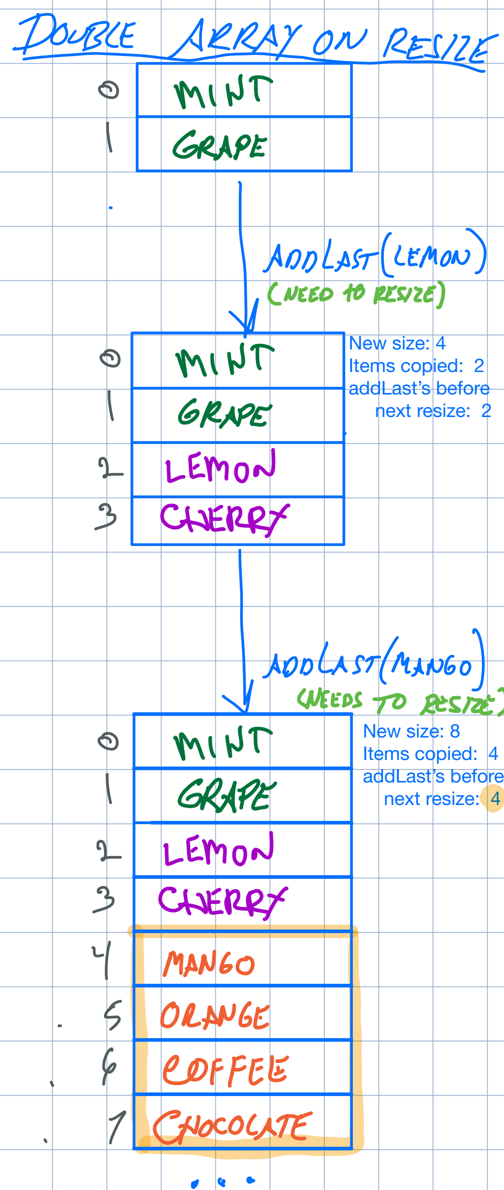
=> When you resize, double the size of the array



FOR THIS VERSION

SIZE	4	6	8	10
ITEMS COPIED	2	4	6	8
ADDS BEFORE RESIZE	2	2	2	2

Number of copies still grows linearly as array size grows => linear runtime! => O(N)



SIZE	4	8	16	32	64	
ITEMS COPIED	2	4	8	16	32	...
ADDS BEFORE RESIZE	2	4	8	16	32	

By doubling the array each time we resize, we pay effectively pay a fixed portion of the cost equal to the number of items we add. Thus, if we divide up the total cost of copying over all elements in the array, the cost to add is constant!