Lecture 12 – ArrayLists and Runtime

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

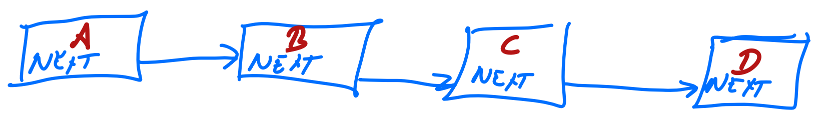
	LinkList	MutableList (Link) (LIKE HW2)	ArrList
size			
addFirst			
addLast			
get(index)	O(N) LINEAR	O(N) LINEAR	O(1) CONSTANT

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

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

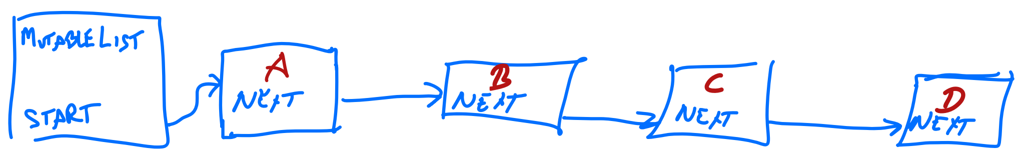
LinkList (or ImmutableList)
- Has a chain of nodes with (at least) a "next" field
- Each node could be at any spot in memory

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)

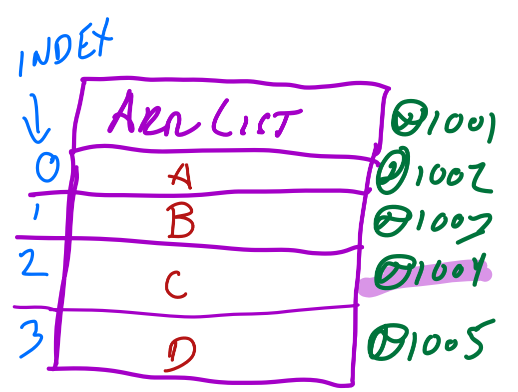


MutableList (like HW2)
- Same "chain" of nodes
- MutableList class has "start" field that points to nodes
- MutableList might have other fields like in HW2 (end, etc.)

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



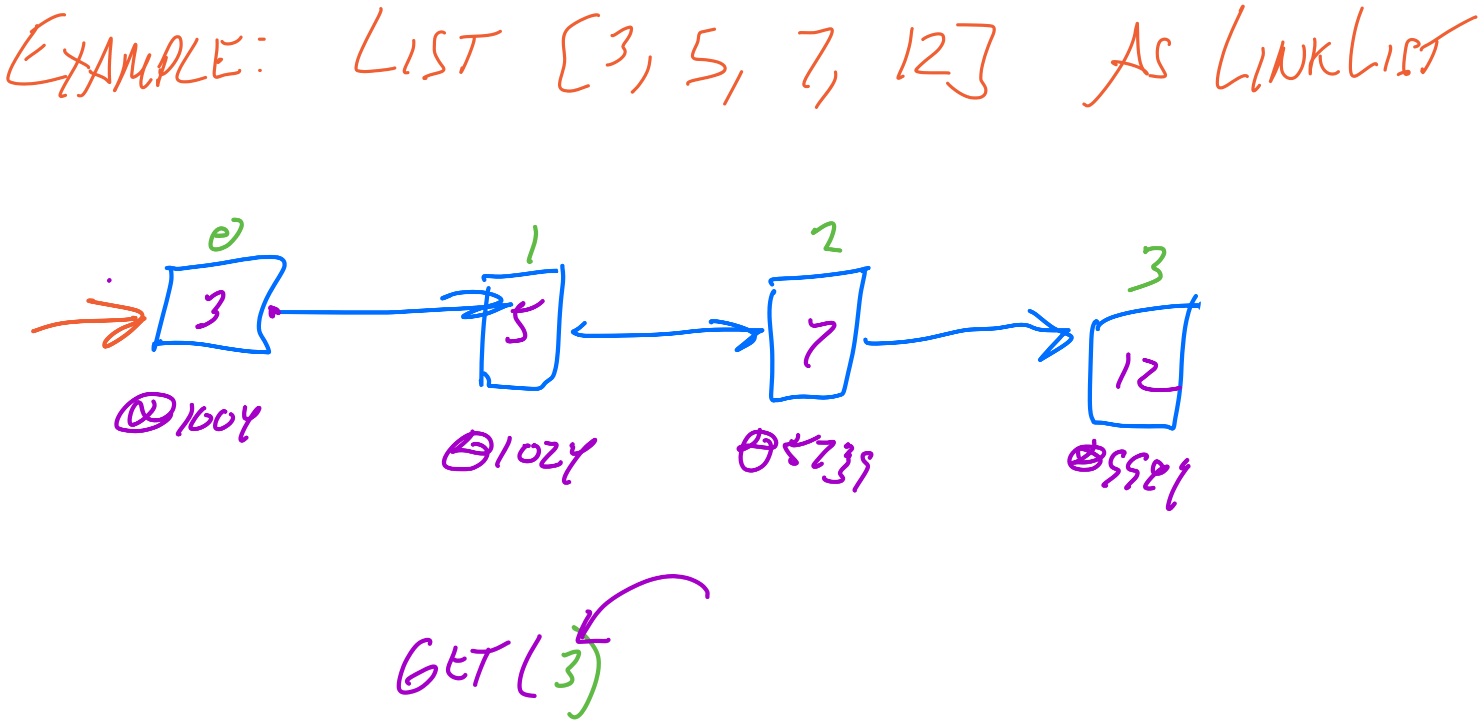
For get() => Since the array elements are always in contiguous memory slots, can look up the i'th element just based on the starting address value.
=> Just add to the starting address => constant time => O(1)

Ex.

GET(2) = @1001 + 2 + 1

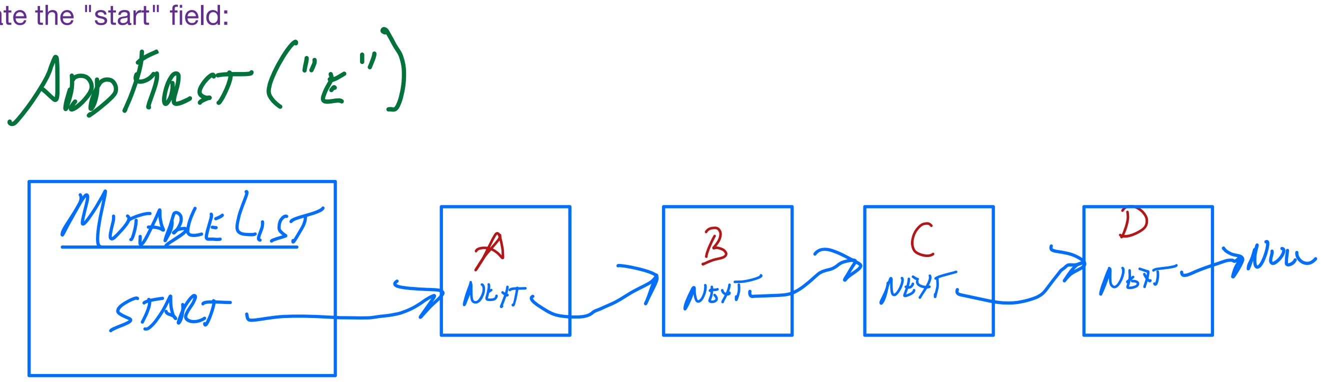
START ↑ INDEX ↑

Q: Why is runtime for get(i) O(N) for a LinkList/MuitableList?
=> No guarantee where nodes are located in heap, so need to follow "next" field in each node to find each element, until we get to the index we want:

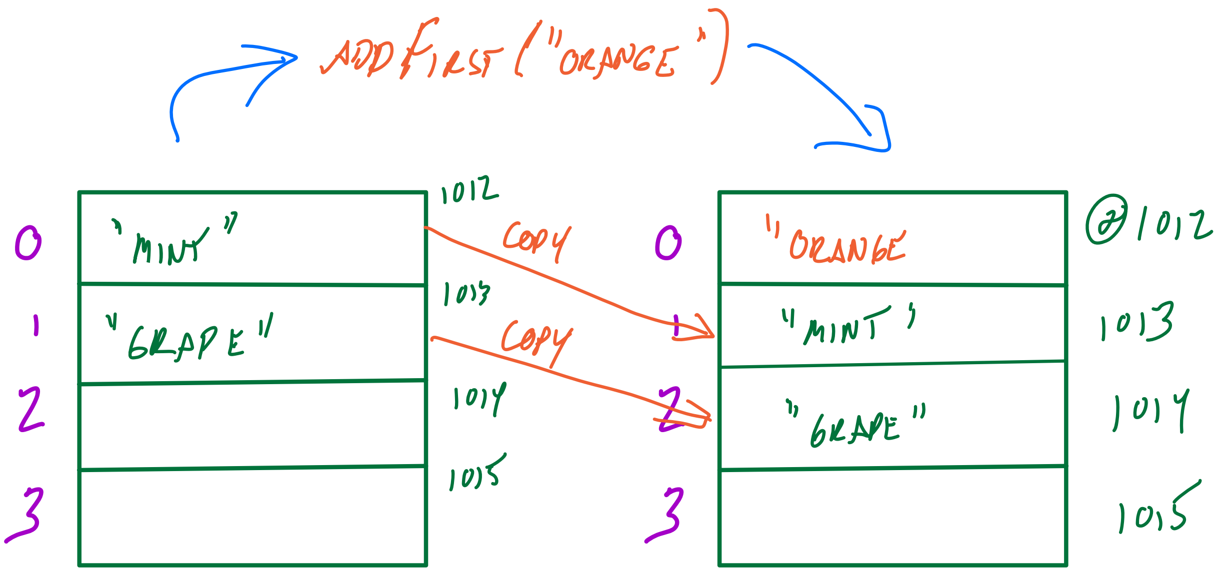


What about addFirst?

On a linked list (eg. MutableList), add first has constant runtime => just need to make a new object and update the "start" field:



Now what about an ArrList?



If you wanted to do this, you'd need to shift all the elements down somehow. The code really isn't that important to us, but in order to maintain our property of everything being laid out contiguously, you'd NEED to rearrange the objects.

So what's the runtime? It's linear!

If your array is really big, this gets expensive.

In practice, it turns out we handle this in the same way we handle adding to a full array, by resizing. So let's talk about that now. But regardless of how the resizing actually happens, I want you to understand that when the list is backed by an array, this copying thing needs to happen.

addFirst on ArrList

Let's say we start with a (non-full) ArrList and we want to add to the beginning. If some elements are already filled-in, addFirst would be a bit complicated

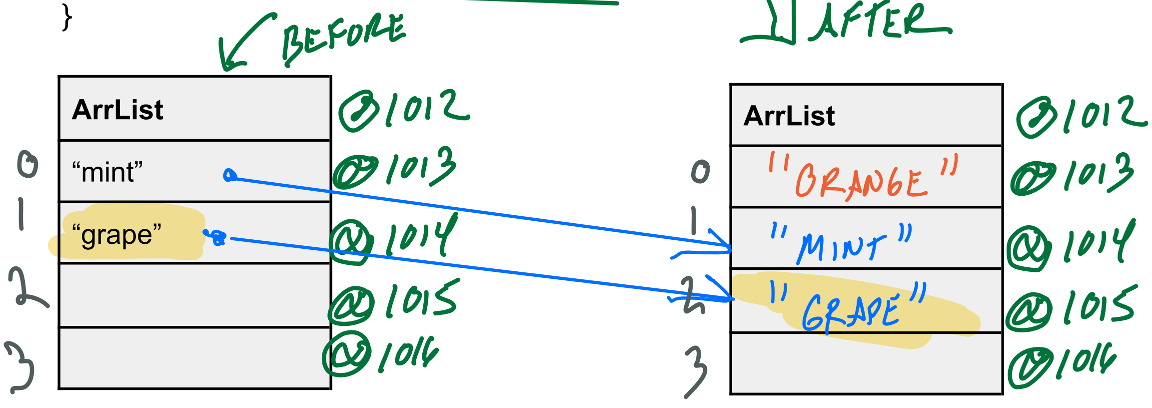
```
public class ArrTest1 {
   ArrList flavors1 = new ArrList(4);
   flavors.addLast("mint");
   flavors.addLast("grape");

   // flavors.addFirst("orange");
}
```

4 SLOTS

What SHOULD happen???

What does this mean for runtime????



Since there might not be slots at the beginning of the array, addFirst would need to shift all of the existing elements down by one slot in order to make room! This would mean moving all elements in the array => O(N) runtime!

Q: how would you move an element in the array?
Example: `arr[2] = arr[1] //  Item at arr[1] ("grape") is now at arr[2]`
(In practice, you'd write this code in a loop to move all the elements, not just one.)

Q: would this make a new array? We could write the code this way, but this isn't required unless the array is already full. In this example, we reuse the same list, just move the elements (so the memory addresses are the same.

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.)

Note for next page: When array is not full, addLast just needs to add one element to the array and increment two fields => constant runtime

```
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		
`flavors.addLast("grape")`	CONST		
`flavors.addLast("lemon")`	LINEAR		
`flavors.addLast("cherry")`	LINEAR		
`flavors.addLast("mango")`	LINEAR		
`flavors.addLast("orange")`			
`flavors.addLast("coffee")`			

With resizing, the runtime for addLast kind of depends on whether array is full or not....

If array is NOT FULL: just need to add an element at index given by this.end (see code on previous page) => runtime is constant

If the array is FULL => runtime is linear because we need to copy all elements

Good default resize policy: each time you need to resize the array, double the size of the underlying array

If you do this => over time the runtime is effectively constant O(1)
=> We call this **amortized runtime:** in this case, the cost of copying N elements is "paid out" over N adds, which makes the runtime effectively constant when measuring over a large number of operations

For details, see the two pages at the end, and the typed notes.

SUPER QUICK intro to exceptions (more on this in a few lectures)

What if we wanted addLast to **throw an error** when the array was full? (This doesn't make much sense in practice, but it's a good example of the concept.)

```
public void addLast(String newItem) {
   if (this.isFull()) {
      // . . .          ERROR
   }
   if (this.isEmpty()) {
       this.end = this.end + 1;
   }
   this.eltcount = this.eltcount + 1;
   this.theArray[this.end] = newItem;
}
```

In the exceptions we've seen so far (eg. IllegalArgumentException), **the program crashes when an exception is thrown**: this is fine in some cases when the program can't possibly continue. But **what if we want to handle the error more gracefully?** What if we want our program to detect when an error occurs, and then do something different to recover from the situation, and **keep running**?

Here's how: First, it's common to create new classes for different types of exceptions in Java that are specific to the error we want to recover from. All exceptions extend the class Exception (or some other subclass of Exception).

```
// This is an exception class, gives the error a name
public class ArrayFullException extends Exception {}
```

Then we throw that exception....

```
public void addLast(String newItem) throws ArrayFullException {
   if (this.isFull()) {
       throw new ArrayFullException(); // Kinda like return, but different
   }
```

We can detect and recover from the error with a **try/catch block,** like this:

```
ArrList arr = new ArrList(2);
arr.addLast("a");
arr.addLast("b");
// try/catch:  do the code in the try
// if it throws this type of exception, run this cod
// (e is a name with error about the exception)
try {
    arr.addLast("c");
} catch (ArrayFullException e) {
    // Do something different (ie, don't crash)
    // Prompt the user, remove element, ...
    // arr.removeLast(...)
}
```

Can put any amount of code here! If an ArrayFullException is thrown, the catch block will be run, and then the program can continue running.

We'll see more with exceptions in a few lectures, but it will help to know about try/catch now!

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	CONST	1
`flavors.addLast("grape")`	CONST	CONST	CONST	1
`flavors.addLast("lemon")`	RESIZE	RESIZE (4)	RESIZE	2 COPY OPS
`flavors.addLast("cherry")`	RESIZE	CONST	CONST	1
`flavors.addLast("mango")`	RESIZE	RESIZE	CONST	1
`flavors.addLast("orange")`	RESIZE	CONST	CONST	1
`flavors.addLast("coffee")`	RESIZE	RESIZE	CONST	1
			RESIZE	4 COPY OPS
			CONST	1
			⋮ 6 MORE	111111
			CONST	1
			RESIZE	8 COPY OPS
			⋮	

Each resize is linear time due to the copy

LINEAR # OF CALLS TO RESIZE, FOR N CALLS TOTAL

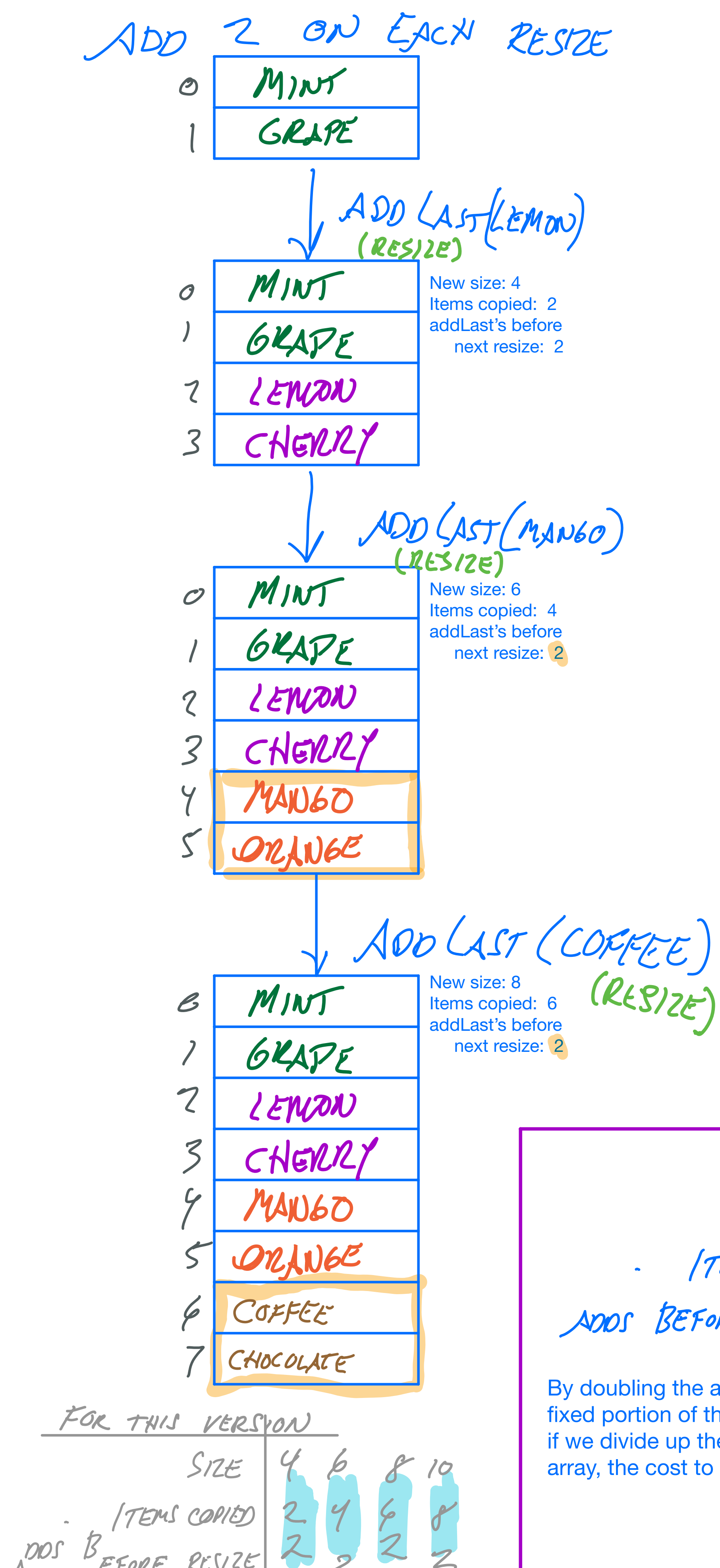
For N calls, resize N/2 times => halved runtime cost => still linear runtime O(N)

What happens in practice (as a general rule)
=> When you resize, double the size of the array

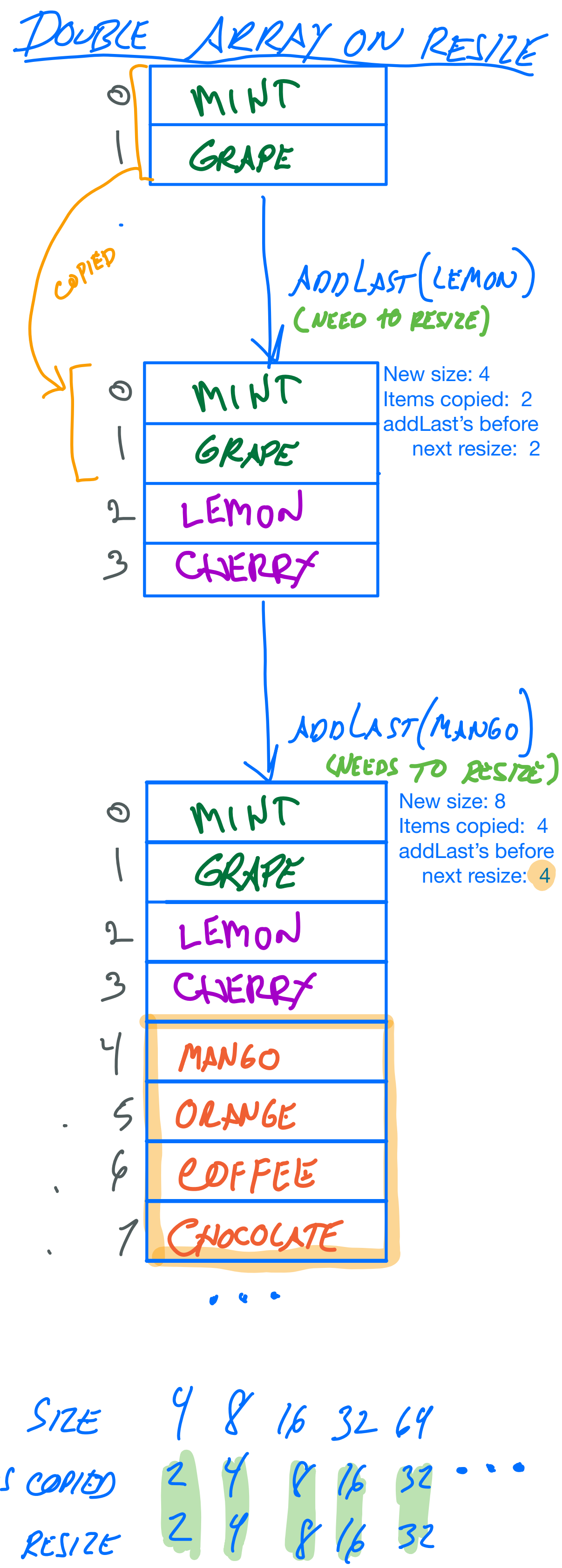
Instead of looking at the worst case for one call (linear), we can look at the cost over all the allocations we'll do to build the whole list… the cost is distributed across all the elements.

The total cost for N calls to addLast is a amortized constant

We can distribute the cost across the elements so each one is charged a constant amount for the copying work.



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



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!