*(**For a preview on exceptions**, see the end of these notes)*

Lecture 12 – ArrayLists and Runtime

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

(LIKE HW2)

	LinkList	MutableList (Link)	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…

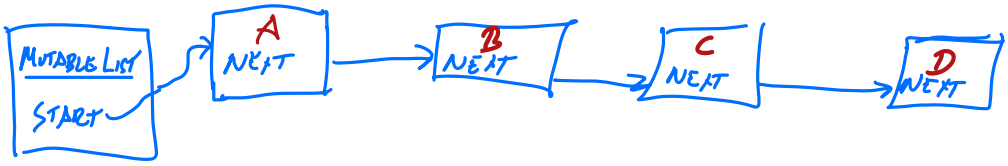
SAY WE HAVE LIST WITH STRINGS ["A","B","C","D"]

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)

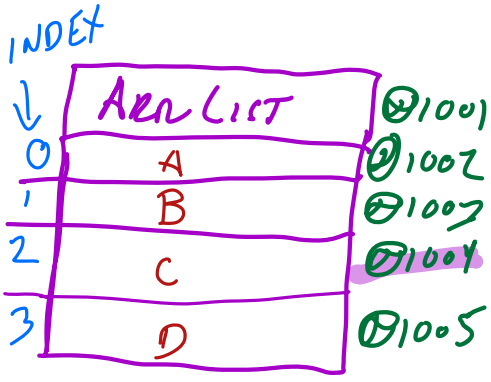
MutableList (like HW2)
- Same "chain" of nodes
- MutableList class has "start" field that points to nodes
- MutableList might have other fields like in HW2 (end, etc.)

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 all the elements

=> If the double the size of the array on each resize
Runtime to addLast to array becomes constant O(1)
when looking over a large number of adds ("amortized constant")

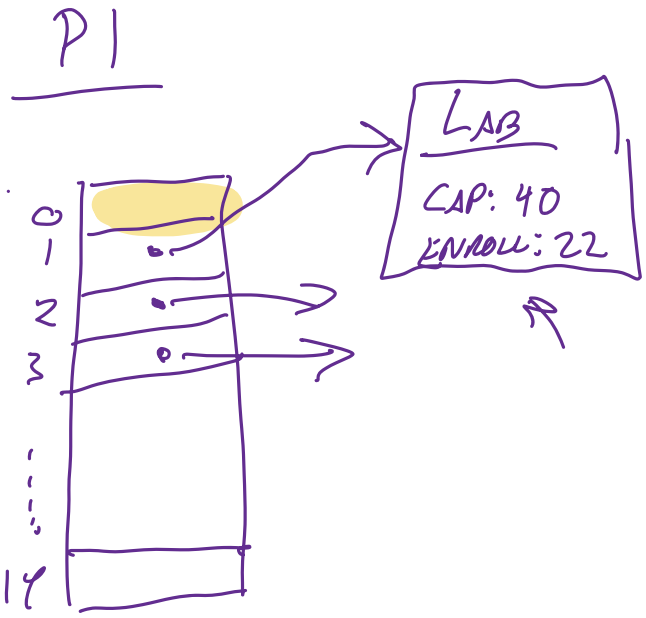


Activity: Three Design Exercises

Design problem #1: A professor is trying to manage enrollments for several lab sections (numbered 0-1 through 14). For each lab, the professor needs to store the capacity of the room and the number of students in the lab. Propose specific data structures to organize this information.

Design problem #2: A department is trying to manage enrollments in several courses (numbered 1000 through 1999). For each course, the department needs to store the capacity of the room and the number of students in the course. Propose specific data structures to organize this information.

Design problem #3: A professor is trying to manage enrollments for multiple lab sections, each labeled with the day of week and start time (such as Mon 8-10, Tues 4-6, etc). For each lab, the professor needs to store the room where the lab is meeting.

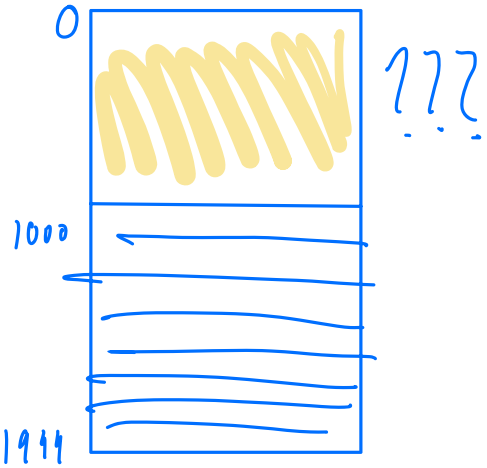


With an array-like structure, can access a specific lab with constant time lookup get(1)
=> *Could use an Array of ArrayList for this (see next page for discussion of differences)*

If use use a LinkedList, get(i) => O(N)

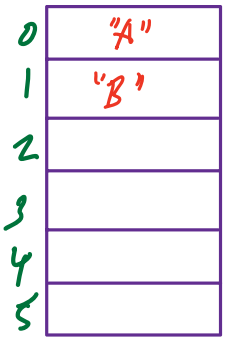
P2. Same idea here, but if the course numbers start at 1000 we have some issues:
- Do we leave 999 empty slots at the start of the array? This could waste memory

Could write a helper get(i) that subtracts 1000 from the course number...
=> But what if we wanted to look up courses by something other than the number (eg. a name, like P3)?
=> Next lecture!



Array vs. ArrayList?

A plain Array:
- Fixed slots in memory at contiguous addresses

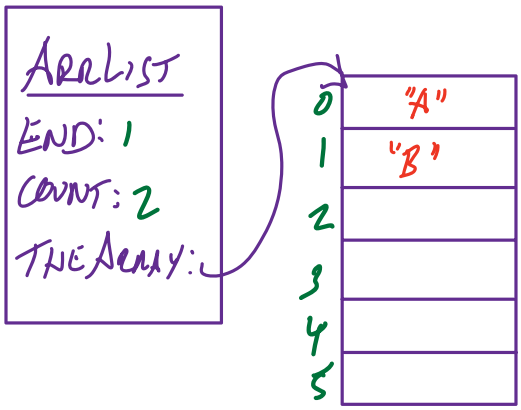


Use like this:

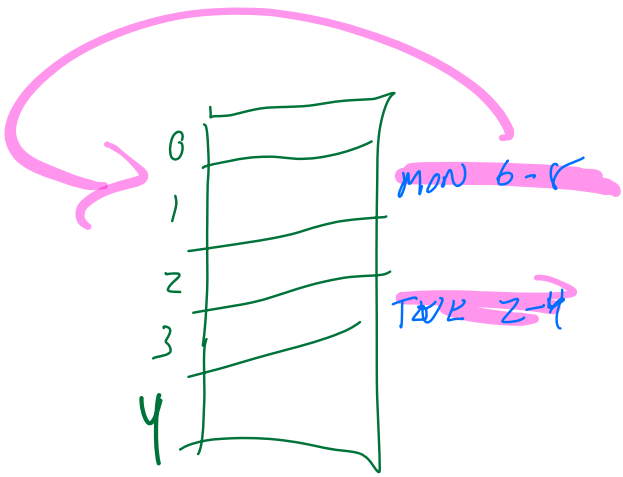
```
String[] arr = new String[5];
arr[1] = "hi";
```

- All you can do is access the different elements (in constant time)

ArrayList (ArrList)
- Class that contains an array, has methods to perform operations like a list (eg. addFirst, addLast, etc.)
- Our version: ArrList; Java's version: ArrayList
- When array becomes full, creates a new array and copies over all the elements
- Contains EXACTLY ONE underlying array
  - All elements are always stored contiguously in memory
  - After copying, old array is unused (and Java cleans it up)



But what if we want to access array elements by a name, instead of a numeric index???
=> Next lecture!



Preview on exceptions: As we work with more code, we'll see more examples of Java's exceptions. Exceptions are ways for the program to deal with errors. So far, all the exceptions we've seen have been for errors that just cause the program to quit.

However, what if we want to handle errors more gracefully? For example, if the user enters a bad input, perhaps we could display an error and prompt them again?

To continue from the ArrList example, let's consider a version that throws an exception when the array is full--this doesn't make sense for an ArrList, but let's pretend this is the behavior we want... How could we implement this?

```
public class ArrayFullException extends Exception {
    public ArrayFullException() {
    }
}
```

To represent specific errors, can make a new class extending class Exception.
This creates a "checked" exception, which is the default type in Java.
(All the exceptions we've seen before this are called "unchecked" exceptions--more on this later.)

```
public void addLast(String newItem) throws ArrayFullException {
   if (this.isFull()) {
      throw new ArrayFullException();
   }
   // . . . rest of addLast code . . .
```

To "throw" an exception, we use the "throw" keyword (as we've been doing in prior assignments).

A new twist: when we throw a checked exception, we need to add an annotation to the method header to tell Java that this method throws an exception

Example 1

```
public static void exceptionExample1() {
   ArrList arr = new ArrList(2);

   try {
       arr.addLast("a");
       arr.addLast("b");
       arr.addLast("c"); // Will fail
   } catch (ArrayFullException e) {
       System.out.println("array was full!");

   }
   System.out.println("Array has:  " + arr);
}
```

To "handle" an exception, we can use a try/catch block:
- Java runs the code in the "try"
- If an exception (here, ArrayFullException) is thrown, the code in the catch block will run.
- Then, the program continues running after the catch block

Example: this program would print something like...
"array was full!"
"Array has: [a, b]"

Example 2

```
public static void exceptionExample2() {
   ArrList arr = new ArrList(2);
   arr.addLast("a");
   arr.addLast("b");
}
```

Error! Must add try/catch, or throws annotation

Note: **Java requires the programmer to catch checked exceptions**: if Java can't find code to catch the exception, it's an error! Two options:
- Add a try/catch (like example 1)
- Add a "throws" annotation to the method (which forces the method that called this one to deal with the exception instead)

=> Which option you pick depends on where in the program makes the best sense to handle the error (eg. where to prompt the user, or undo an operation) => we'll talk more about this from a design perspective soon!

So what's the deal with checked vs. unchecked exceptions?

Checked exceptions: Java enforces the programmer to catch the exception somewhere
- This is the default type in Java (exception extends class "Exception")
- In general, should be used for exceptions where the program shouldn't crash (ie, when there's some possibility to recover from the error, display an error message, or do something other than totally crashing the program

Unchecked exceptions: work in the same way, except Java doesn't enforce the programmer to catch them.
- Declared just like checked exceptions, except they extend the class RuntimeException
- Used for errors that shouldn't occur during normal operation (ie, if the program has no bugs), or errors where there's nothing to do except exit the program
- All of the exceptions we've seen so far are unchecked exceptions, including: NullPointerException, IndexOutOfBoundsException, IllegalArgumentException, ...

We'll learn a lot more about exceptions and their differences in a few lectures, so no need to worry too much about this for now! We just want you to know the terminology, and so you know there's a difference between the exceptions you're seeing now, and the exceptions you've been throwing in earlier assignments.