University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 3: this Friday @ 11:59 pm
  - Lab 2: next Monday @ 11:59 pm
  - Programming Assignment 1: Friday Oct. 7th
- Please include all instructors when sending private messages on Piazza, if possible
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- Code efficiency
  - How to determine running time of an algorithm without running it?
  - Count the number of executed basic operations
    - **as a function of the input size**
  - Determine the order of growth of the runtime function
    - Ignore lower order terms
    - Ignore constant factors
    - Big-Oh approximation

Muddiest Points

- **Q: in what case would you specifically want to use a linked list?**
- **Q: Is a linked list usually more or less efficient than an unlinked one?**
- Linked chains grow and shrink in size based on the actual number of data items.
- Arrays are more rigid in the sense that they need to be allocated contiguously.
- If the actual number of used data items is static, that is, doesn't change widely throughout the runtime of the application, an array would be better (more space efficient)
- Otherwise, use a linked chain
-

Muddiest Points

- **Q: is all the memory needed for every reference variable in an array allocated when the array is created, or when each index is filled? i.e. Does a newly formed (empty) array take up the same space in memory as a filled array.**
- Yes! The reference variables inside an array are allocated when the array is created.

Muddiest Points

- **Q: I am still confused by how memory is allocated with a partially filled array. Do the objects within the array determine this or the reference variable type of the array.**
- String[10] uses the same memory as Integer[10], ArrayBag[10], Square[10], …
- Each has 10 reference variables, and all reference variables have the same size (e.g. 4 bytes)

Muddiest Points

- **Q: Clarification on why a linked bag takes exactly double the space than an arraybag.**
- **Q: Why is it that linked chains will always take up 100% more memory than arrays?**
- A linked chain takes exactly double the space of a **full** array. Each node in the chain has one extra reference variable, which is the next field
- **Q: What if the data fields contained in each node are different than those contained in an array?**
- A: The size of the data objects doesn't affect the size of the array not the chain node.
- Both contain reference variables and all reference variables are the same size.

Muddiest Points

- **Q: Big-Oh runtime is very confusing to me. Are there easy ways to practice and master this material?**
- A: I will prepare a list of examples on determining the Big-Oh approximations of various functions.
- **What are some examples for the different growth rate functions?**
- 1, log log n, $\log^2 n$, n, n log n, n^2, 2^n, n!

Muddiest Points

- **Q: How does one "lose the chain" when incorrectly removing nodes in a chain?**
- If we change what firstNode points to before saving that in another variable.
- Incorrect way to remove first node:

```
firstNode = newNode;

newNode.next = firstNode;
```

- Correct way:

```
newNode.next = firstNode;

firstNode = newNode;
```

Muddiest Points

- **Q: The big oh notation, how does it actual works**
- A: Big-Oh is an approximation tool.
- $5n^2 + 30\ n + 100 = O(n^2)$
- It breaks a function down to its **order of growth**, how fast is the rate of function value increase when input increases
- **Q: I still don't quite get the big O notion. Like why isn't 2^cn not O(2^n)?**
- $2^{cn} = 2^{(c-1)n}\ 2^n$
- cannot be expressed as a *constant x* 2^n

Muddiest Points

- **Q: How would you update the pointer in the linked list to something in the center of the list.**
- We make an outside pointer point to a node in the center by traversing the list starting from its first node.

Muddiest Points

- **Q: Why does big o matter?**
- Because it extracts the order of growth of a function. In algorithm analysis we care more about the order of growth of runtime than about the exact runtime values.

Muddiest Points

- **Q: Calculating the number of executed steps of an algorithm**
- Watch for loops and determine the number of loop iterations

Muddiest Points

- **Q: I still don't fully understand the remove implementation for a linked bag. Would it not be the same, if not more efficient, by traversing the linked list and removing there as replacing the middle element with the first element and removing the first element? Both require a traversal, which would be O(n) but replacing the middle Node's data with the first Node will be an extra operation.**
- You are right! Removing a node from middle of a LinkedBag can be done by:
  - cutting it out of the chain and
  - by replacing its data with firstNode.data
  - both are O(n) because they both require chain traversal
  - cutting the node out is a bit more complicated than simply removing the first node.

Today's Agenda

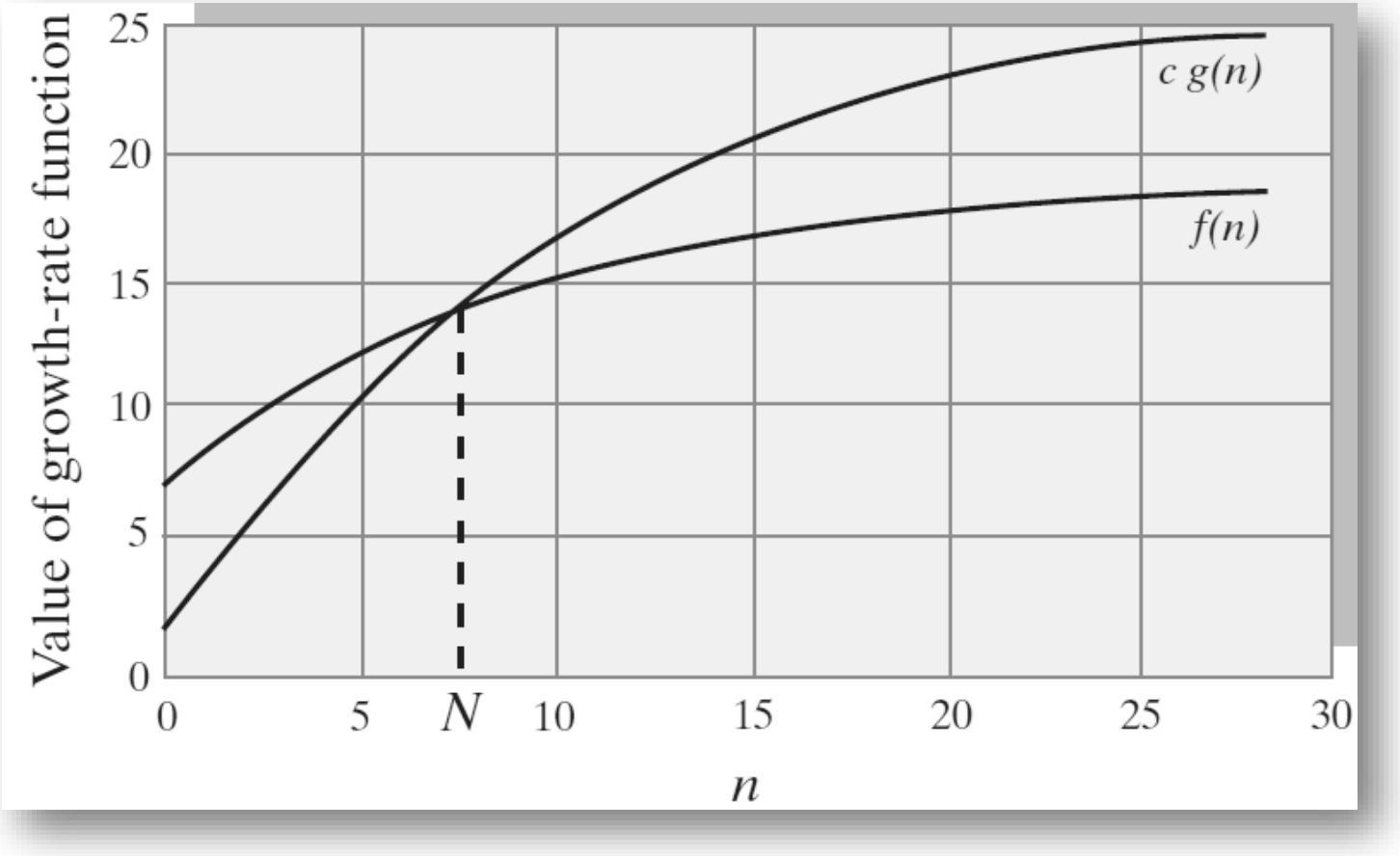
- Big-Oh Approximation
- ADT List
  - Fixed-size array implementation: ArrayList

Big Oh Notation

- A function f(n) is of order at most g(n)
- That is, f(n) is O(g(n))—if
  - A positive real number c and positive integer N exist …
  - Such that $f(n) \leq c * g(n)$ for all $n \geq N$
  - That is, $c * g(n)$ is an upper bound on $f(n)$ when n is sufficiently large

Big Oh Notation

- An illustration of the definition of Big Oh



Big Oh Notation

- Identities for Big Oh Notation

The following identities hold for Big Oh notation:

$O(k\,g(n)) = O(g(n))$ for a constant k

$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$

$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$

$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n))$

$O(\max(g_1(n), g_2(n), \ldots, g_m(n)) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is $4n^2 + 50n - 10$,

$O(4n^2 + 50n - 10) = O(4n^2)$ by ignoring the smaller terms

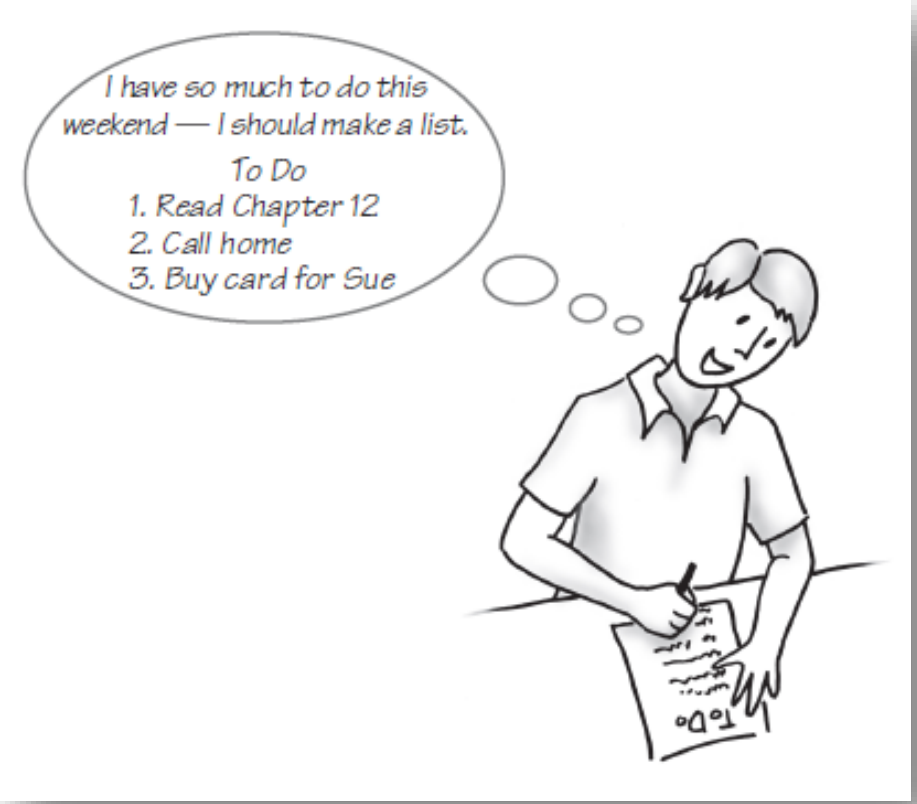
$= O(n^2)$ by ignoring the constant multiplier

Complexities of Program Constructs

Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are g_1 and g_2, respectively	$O(condition) + \max(O(g_1), O(g_2))$
A loop that iterates m times and has a body whose growth-rate function is g	$m \times O(g(n))$

Lists

A to-do list



Specifications for the ADT List

- `add (newEntry)`
- `add (newPosition, newEntry)`
- `remove(givenPosition)`
- `clear()`
- `replace( givenPosition, newEntry)`

`getEntry( givenPosition)`

`toArray()`

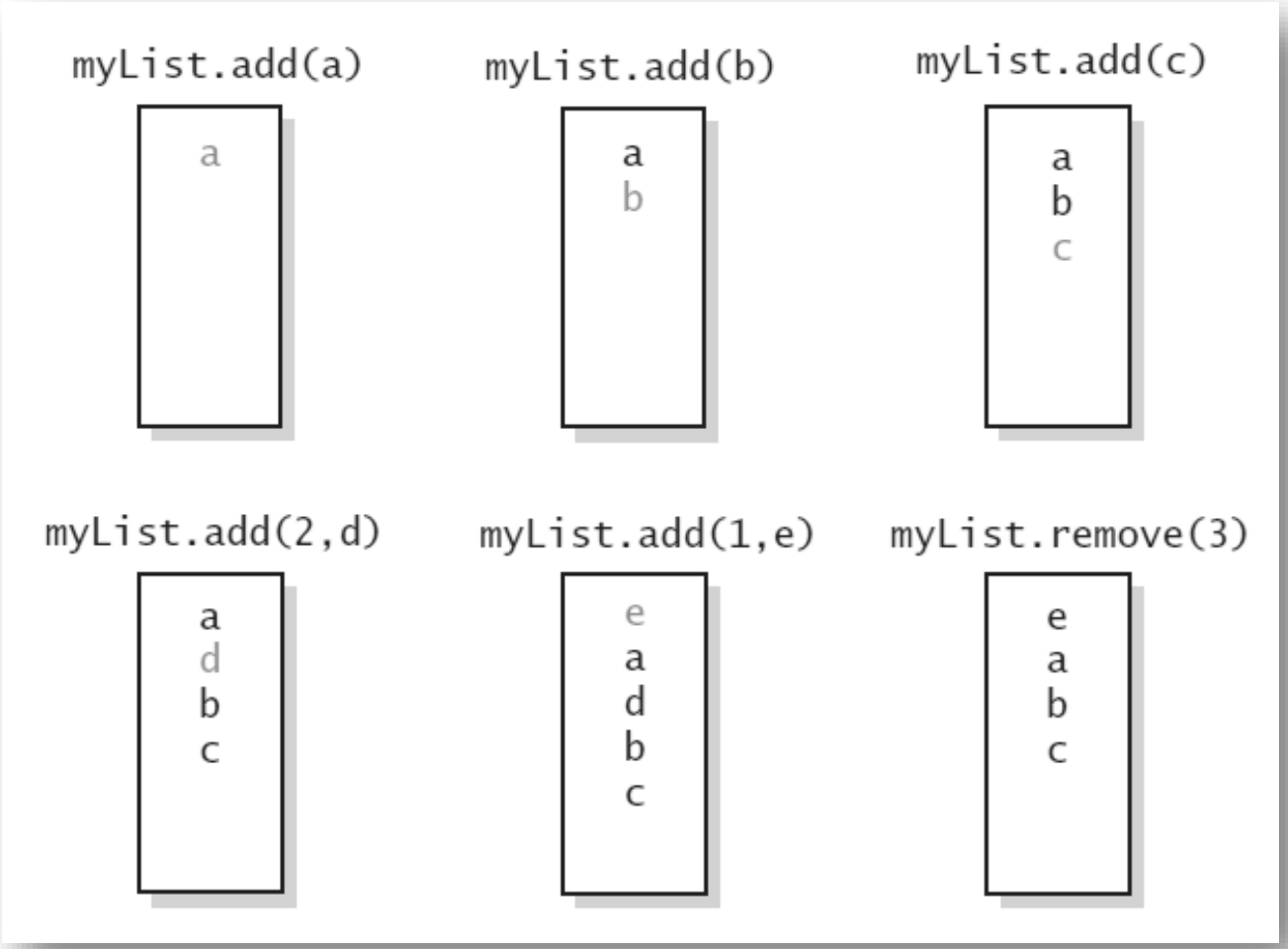
`contains(anEntry)`

`getLength()`

`isEmpty()`

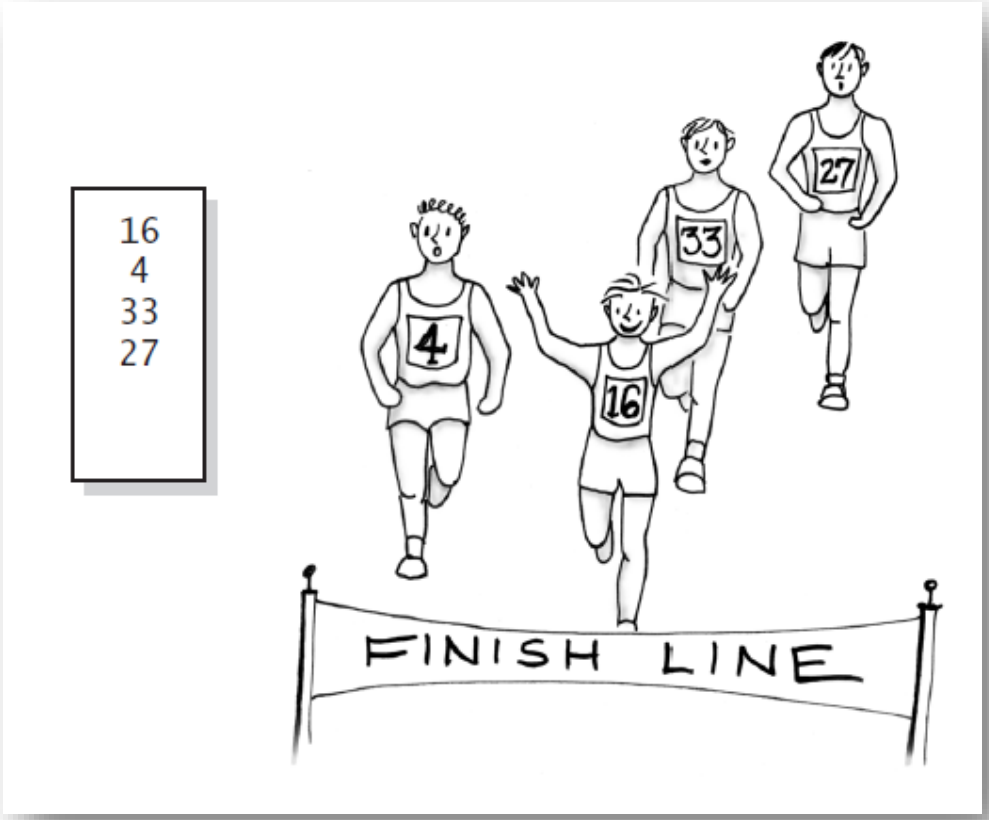
Specifications for the ADT List

The effect of ADT list operations
on an initially empty list



Using the ADT List

A list of numbers that identify runners in the order in which they finished a race



Using the ADT List

A client of a class
that implements **`ListInterface`**

```
1  public class ListClient
2  {
3     public static void main(String[] args)
4     {
5        testList();
6     } // end main
7
8     public static void testList()
9     {
10       ListInterface<String> runnerList = new AList<>();
11 //  runnerList has only methods in ListInterface
12
13       runnerList.add("16"); // Winner
14       runnerList.add(" 4"); // Second place
15       runnerList.add("33"); // Third place
16       runnerList.add("27"); // Fourth place
17       displayList(runnerList);
18    } // end testList
19
20    public static void displayList(ListInterface<String> list)
```

Using the ADT List

A client of a class
that implements **`ListInterface`**

```
19
20    public static void displayList(ListInterface<String> list)
21    {
22       int numberOfEntries = list.getLength();
23       System.out.println("The list contains " + numberOfEntries +
24                          " entries, as follows:");
25
26       for (int position = 1; position <= numberOfEntries; position++)
27          System.out.println(list.getEntry(position) +
28                             " is entry " + position);
29
30       System.out.println();
31    } // end displayList
32 } // end ListClient
```

Output

```
The list contains 4 entries, as follows:
16 is entry 1
 4 is entry 2
33 is entry 3
27 is entry 4
```

Java Class Library:The Interface `List`

Method headers from the interface `List`

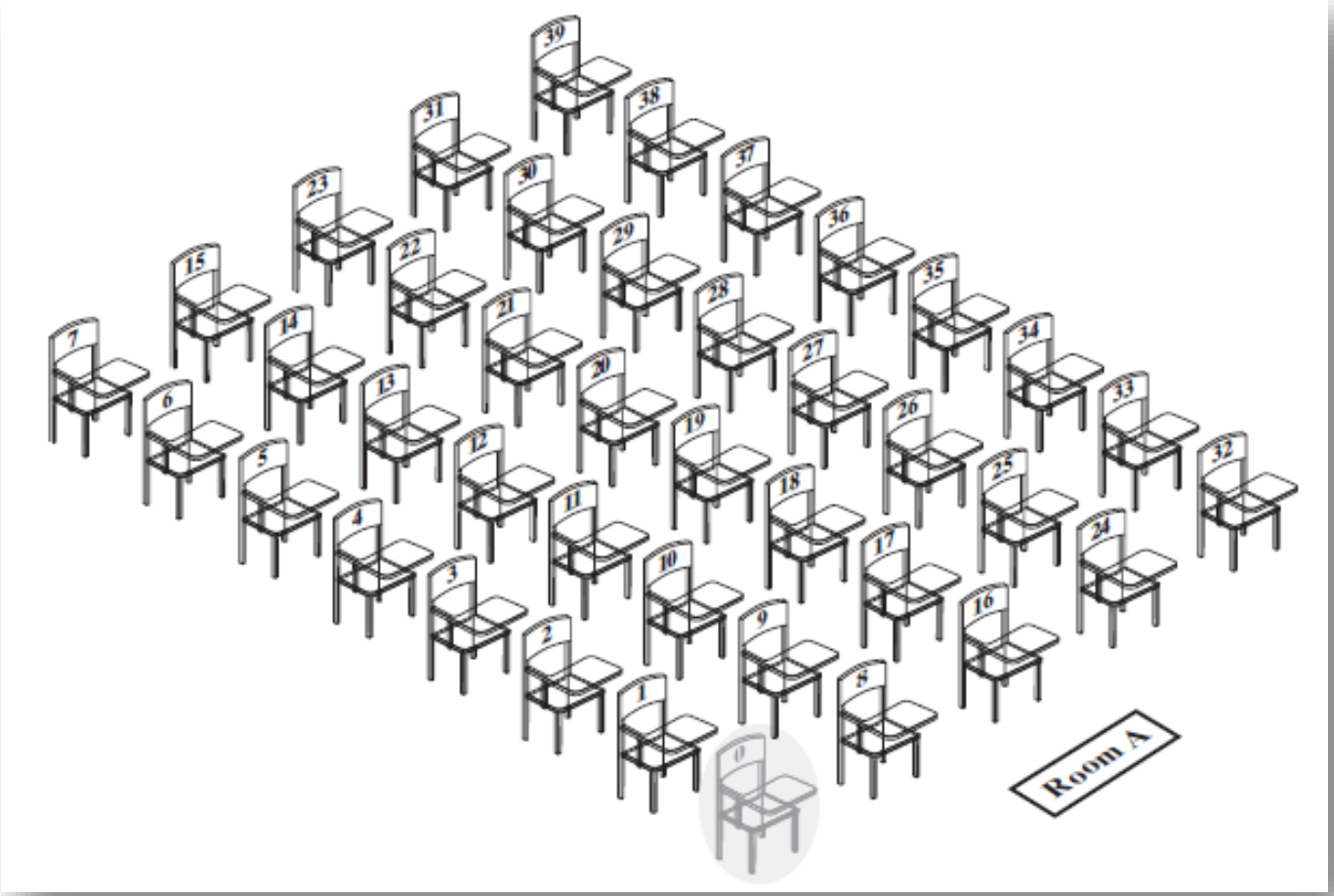
```
public boolean add(T newEntry)
public void add(int index, T newEntry)
public T remove(int index)
public void clear()
public T set(int index, T anEntry) // Like replace
public T get(int index)            // Like getEntry
public boolean contains(Object anEntry)
public int size()                  // Like getLength
public boolean isEmpty()
```

Java Class Library: The Class `ArrayList`

- Available constructors
  - `public ArrayList()`
  - `public ArrayList(int initialCapacity)`
- Similar to java.util.vector
  - Can use either `ArrayList` or `Vector` as an implementation of the interface `List`.

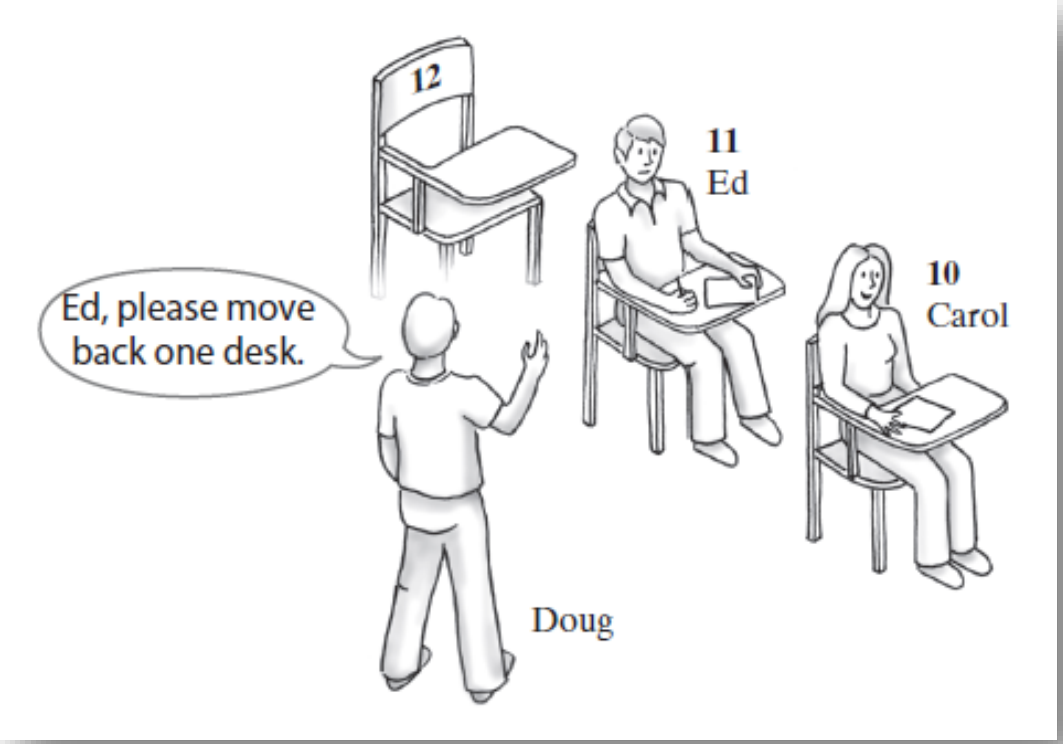
Array to Implement the ADT List

A classroom that contains desks in fixed positions



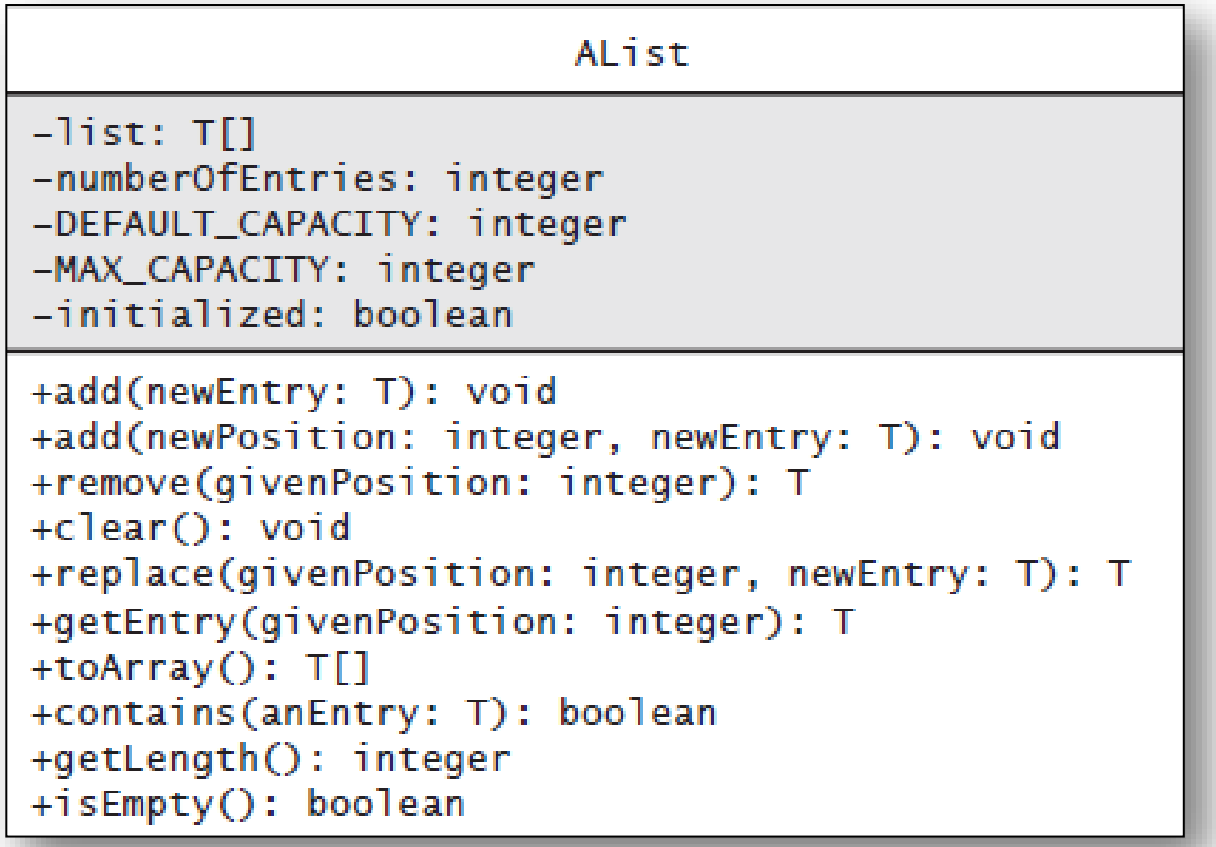
Array to Implement the ADT List

Seating a new student between two existing students:
At least one other student must move



Array to Implement the ADT List

UML notation for the class **`AList`**



Array to Implement the ADT List

The class **AList**

```
1  import java.util.Arrays;
2  /**
3      A class that implements a list of objects by using an array.
4      Entries in a list have positions that begin with 1.
5      Duplicate entries are allowed.
6      @author Frank M. Carrano
7  */
8  public class AList<T> implements ListInterface<T>
9  {
10     private T[] list;   // Array of list entries; ignore list[0]
11     private int numberOfEntries;
12     private boolean initialized = false;
13     private static final int DEFAULT_CAPACITY = 25;
14     private static final int MAX_CAPACITY = 10000;
15
16     public AList()
17     {
18        this(DEFAULT_CAPACITY); // Call next constructor
19     } // end default constructor
20
21     public AList(int initialCapacity)
```

Array to Implement the ADT List

The class **AList**

```
19      } // end default constructor
20
21      public AList(int initialCapacity)
22      {
23         // Is initialCapacity too small?
24         if (initialCapacity < DEFAULT_CAPACITY)
25            initialCapacity = DEFAULT_CAPACITY;
26         else // Is initialCapacity too big?
27            checkCapacity(initialCapacity);
28
29         // The cast is safe because the new array contains null entries
30         @SuppressWarnings("unchecked")
31         T[] tempList = (T[])new Object[initialCapacity + 1];
32         list = tempList;
33         numberOfEntries = 0;
34         initialized = true;
35      } // end constructor
36
```

Array to Implement the ADT List

The class **AList**

```
36
37      public void add(T newEntry)
38      {
39         checkInitialization();
40         list[numberOfEntries + 1] = newEntry;
41         numberOfEntries++;
42         ensureCapacity();
44      } // end add
45
46      public void add(int newPosition, T newEntry)
47      { < Implementation deferred >
59      } // end add
60
61      public T remove(int givenPosition)
62      { < Implementation deferred >
80      } // end remove
```

Array to Implement the ADT List

The class **AList**

```
81
82      public void clear()
83      { < Implementation deferred >
91      } // end clear
92
93      public T replace(int givenPosition, T newEntry)
94      { < Implementation deferred >
106     } // end replace
107
108     public T getEntry(int givenPosition)
109     { < Implementation deferred >
119     } // end getEntry
120
121     public T[] toArray()
122     {
123        checkInitialization();
124
125        // The cast is safe because the new array contains null entries
126        @SuppressWarnings("unchecked")
127        T[] result = (T[])new Object[numberOfEntries];
128        for (int index = 0; index < numberOfEntries; index++)
129        {
130           result[index] = list[index + 1];
131        } // end for
```

Array to Implement the ADT List

The class **AList**

```
130             result[index] = list[index + 1];
131          } // end for
132
133          return result;
134       } // end toArray
135
136       public boolean contains(T anEntry)
137       { < Implementation deferred >
149       } // end contains
150
151       public int getLength()
152       {
153          return numberOfEntries;
154       } // end getLength
155
156       public boolean isEmpty()
157       {
158          return numberOfEntries == 0; // Or getLength() == 0
159       } // end isEmpty
```

Array to Implement the ADT List

The class **AList**

```
159     } // end isEmpty
160
161     // Doubles the capacity of the array list if it is full.
162     // Precondition: checkInitialization has been called.
163     private void ensureCapacity()
164     {
165        int capacity = list.length - 1;
166        if (numberOfEntries >= capacity)
167        {
168           int newCapacity = 2 * capacity;
169           checkCapacity(newCapacity); // Is capacity too big?
170           list = Arrays.copyOf(list, newCapacity + 1);
171        } // end if
172     } // end ensureCapacity
        < This class will define checkCapacity, checkInitialization, and two more private
          methods that will be discussed later. >
222 } // end AList
```

Array to Implement the ADT List

- Implementation of **`add`** uses a private method **`makeRoom`** to handle the details of moving data within the array

```
// Precondition: The array list has room for another entry.
public void add(int newPosition, T newEntry)
{
   checkInitialization();
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
   {
      if (newPosition <= numberOfEntries)
         makeRoom(newPosition);
      list[newPosition] = newEntry;
      numberOfEntries++;
      ensureCapacity(); // Ensure enough room for next add
   }
   else
      throw new IndexOutOfBoundsException(
                "Given position of add's new entry is out of bounds.");
} // end add
```

Array to Implement the ADT List

Implement the private method **`makeRoom`**

```
// Makes room for a new entry at newPosition.
// Precondition: 1 <= newPosition <= numberOfEntries + 1;
//               numberOfEntries is list's length before addition;
//               checkInitialization has been called.
private void makeRoom(int newPosition)
{
   assert (newPosition >= 1) && (newPosition <= numberOfEntries + 1);

   int newIndex = newPosition;
   int lastIndex = numberOfEntries;

   // Move each entry to next higher index, starting at end of
   // list and continuing until the entry at newIndex is moved
   for (int index = lastIndex; index >= newIndex; index--)
      list[index + 1] = list[index];
} // end makeRoom
```

Array to Implement the ADT List

Making room to insert
Carla as the third entry in an array