CS 10: Problem solving via Object Oriented Programming

Lists Part 2 (Array’s Revenge!)

Agenda

1. Iterators

Key points:
1. **Iterators loop over items in a List**
2. **They do not need to begin at the head at each call**

2. Growing array List ADT implementation
3. Amortized analysis of growth operation
4. Comparing List implementations

What is wrong with the code below?

```
//declare SimpleList using SinglyLinked implementation
SimpleList<Integer> list = new SinglyLinked<>();
int numberOfItems = 1000;

//add numberOfItems to list
for (int i = 0; i < numberOfItems; i++) {
    list.add(i);
}
```

```
//print each item in list
for (int i = 0; i < list.size(); i++) {
    Integer value = list.get(i);
    System.out.println(value);
}
```

Instantiate SinglyLinked list of Integers

Add 1,000 Integer to List

Print each item in List

Works as intended, but slow

$O(n^2)$ – sneaky inefficiency

Why?

- ***get(i)*** **always starts at head**
    - **Helpful if we could remember where we left off during iteration**
    - **Iterators remember**

Implementing *Iteratable* interface tells Java you promise to implement an iterator

SinglyLinked.java

```
public class SinglyLinked<T> implements SimpleList<T>, Iterable<T> {
   private Element head;  // front of the linked list
   private int size;     // # elements in the list

   /**
    * The linked elements in the list: each has a piece of data and a next pointer
    */
   private class Element {
     private T data;
     private Element next;

     private Element(T data, Element next) {
       this.data = data;
       this.next = next;
     }
   }

   public SinglyLinked() {
     head = null;
     size = 0;
   }
```

We will deal with Iterable ~~soon, standby for more info~~ now

Java's *Iterable* interface says we must provide an *iterator* method for *SinglyLinked* class that returns an iterator object

Iterator<T> iterator()

***Iterator* loops over items of type T, remembering where it left off so we don't need to start at *head* each time**

An iterator must provide a *next* and a *hasNext* method

```
public interface Iterator<T> {
   /**
    * Returns true if the iteration has more elements. (In other words,
    * returns true if next() would return an element rather than throwing an exception.)
    */
   public boolean hasNext();

   /**
    * Returns the next item and advances the iterator.
    * Throws an exception if there is no next item.
    */
   public T next() throws Exception;
}
```

Iterator interface specifies two methods:

- ***hasNext()***
- ***next()***

Key points:

- ***next*** **returns the current item in the List and moves to the following item**
- **We will implement so that the iterator remembers where left off**
- **Subsequent calls to *next* do not start back at the head**

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

```java
public Iterator<T> iterator() {  //satisfy iterator requirement in Iterable interface
   return new ListIterator();
}

/**
 * Iterator class that implements the required functionality to use this List in a for each loop
 */
private class ListIterator implements Iterator<T> {
   // Use curr to point to next item in List
   Element curr; //store current position

   public ListIterator() {
     curr = head;
   }

   public boolean hasNext() {
     return curr != null;
   }

   public T next() {
     if (curr == null) {
       throw new IndexOutOfBoundsException();
     }
     T data = curr.data;
     curr = curr.next;
     return data;
   }
}
```

iterator **method returns an object of nested class *ListIterator* to satisfy *Iterable* interface for SinglyLinked.java**

Nested class *ListIterator* (private to SinglyLinked, but doesn't have to be)

- **Implements *Iterator* interface so must implement *hasNext* and *next***
- **Uses *curr* to keep track of position in list**
- ***curr* initially set to *head***

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

```
public Iterator<T> iterator() {  //satisfy iterator requirement in Iterable interface
   return new ListIterator();
}

/**
 * Iterator class that implements the required functionality to use this List in a for each loop
 */
private class ListIterator implements Iterator<T> {
   // Use curr to point to next item in List
   Element curr; //store current position

   public ListIterator() {
     curr = head;
   }

   public boolean hasNext() {
     return curr != null;
   }

   public T next() {
     if (curr == null) {
       throw new IndexOutOfBoundsException();
     }
     T data = curr.data;
     curr = curr.next;
     return data;
   }
}
```

iterator **method returns an object of nested class** ***ListIterator*** **to satisfy** ***Iterable*** **interface for SinglyLinked.java**

Nested class ***ListIterator*** **(private to SinglyLinked, but doesn't have to be)**

- **Implements** ***Iterator*** **interface so must implement** ***hasNext*** **and** ***next***
- **Uses** ***curr*** **to keep track of position in list**
- ***curr*** **initially set to** ***head***

hasNext **returns true if curr != null (e.g., there are more items in the List), false otherwise**

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

```
public Iterator<T> iterator() {  //satisfy iterator requirement in Iterable interface
   return new ListIterator();
}

/**
 * Iterator class that implements the required functionality to use this List in a for each loop
 */
private class ListIterator implements Iterator<T> {
   // Use curr to point to next item in List
   Element curr; //store current position

   public ListIterator() {
     curr = head;
   }

   public boolean hasNext() {
     return curr != null;
   }

   public T next() {
     if (curr == null) {
       throw new IndexOutOfBoundsException();
     }
     T data = curr.data;
     curr = curr.next;
     return data;
   }
}
```

iterator **method returns an object of nested class** ***ListIterator*** **to satisfy** ***Iterable*** **interface for SinglyLinked.java**

Nested class ***ListIterator*** **(private to SinglyLinked, but doesn't have to be)**

- **Implements** ***Iterator*** **interface so must implement** ***hasNext*** **and** ***next***
- **Uses** ***curr*** **to keep track of position in list**
- ***curr*** **initially set to** ***head***

hasNext **returns true if curr != null (e.g., there are more items in the List), false otherwise**

next **throws** ***IndexOutOfBounds*** **exception if List is empty or** ***curr*** **moved past the last element**
Otherwise, gets data from ***Element*** **pointed to by** ***curr***
Moves ***curr*** **to next position in List**
Returns data
Iterator **interface now satisfied**

Now our SinglyLinked objects can be used in a for-each loop

```
SimpleList<String> list = new SinglyLinked<String>();
//add some items to list

//test for each loop works
for (String item : list) {
    System.out.print(item + "->");
}
System.out.println("[/]");
```

Java converts for-each loop into

```
for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {
    String item = iter.next();
    System.out.print(item + "->");
}
System.out.println("[/]");
```

Because *SimpleList* implements *Iterable*, Java knows *SimpleList* will have an *iterator* method that returns an iterator for the list

Java also knows the *iterator* will implement *hasNext* and *next* because the iterator implements the *Iterator* interface

Now our SinglyLinked objects can be used in a for-each loop

```
SimpleList<String> list = new SinglyLinked<String>();
//add some items to list

//test for each loop works
for (String item : list) {
    System.out.print(item + "->");
}
System.out.println("[/]");
```

Java converts for-each loop into

```
for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {
    String item = iter.next();
    System.out.print(item + "->");
}
System.out.println("[/]");
```

iterator **method returns an object of nested class** ***ListIterator***

Because SinglyLinked implements ***Iterable*** **interface, Java knows it has an** ***iterator*** **method**

```
public class SinglyLinked<T> implements SimpleList<T>, Iterable<T>
```

Now our SinglyLinked objects can be used in a for-each loop

```
SimpleList<String> list = new SinglyLinked<String>();
//add some items to list

//test for each loop works
for (String item : list) {
   System.out.print(item + "->");
}
System.out.println("[/]");
```

Java converts for-each loop into

```
for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {
   String item = iter.next();
   System.out.print(item + "->");
}
System.out.println("[/]");
```

hasNext **returns true if more elements in List, otherwise false**

Notice no increment in for loop

next **will take care of moving curr**

Now our SinglyLinked objects can be used in a for-each loop

```java
SimpleList<String> list = new SinglyLinked<String>();
//add some items to list

//test for each loop works
for (String item : list) {
   System.out.print(item + "->");
}
System.out.println("[/]");
```

Java converts for-each loop into

```java
for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {
   String item = iter.next();
   System.out.print(item + "->");
}
System.out.println("[/]");
```

***next* returns next item in List and moves to following item**

An iterator can dramatically speed up execution time

TimeTest.java

```java
public static Long loopTest1(SinglyLinked<Integer> list, Integer targetValue) throws Exception {
   //use get, start back at head each time through loop
   long startTime = System.nanoTime();
   for (int i = 0; i < list.size(); i++) {
      Integer value = list.get(i);
      if (value == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}
```

Record start time

Loop over all items using *get* (always starts at head)

looking for target value

Return elapsed time in nano seconds

```java
public static Long loopTest2(SinglyLinked<Integer> list, Integer targetValue) {
   long startTime = System.nanoTime();
   //use iterator to not start back at head each time
   Iterator<Integer> iter = list.iterator();
   while (iter.hasNext()) {
      if (iter.next() == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}

public static void main(String[] args) throws Exception {
   //add numberOfItems to list
   SinglyLinked<Integer> list = new SinglyLinked<>();
   int numberOfItems = 1000;
   for (int i = 0; i < numberOfItems; i++) {
      list.add(i);
   }
   Long time1 = loopTest1(list,numberOfItems-1);
   System.out.printf("method 1 took %,15d nanoseconds\n",time1);
   Long time2 = loopTest2(list,numberOfItems-1);
   System.out.printf("method 2 took %,15d nanoseconds\n", time2);
   System.out.println("ratio time1/time2: " + time1/(float)time2);
}
```

An iterator can dramatically speed up execution time

TimeTest.java

```java
public static Long loopTest1(SinglyLinked<Integer> list, Integer targetValue) throws Exception {
   //use get, start back at head each time through loop
   long startTime = System.nanoTime();
   for (int i = 0; i < list.size(); i++) {
      Integer value = list.get(i);
      if (value == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}
```

Record start time

Loop over all items using *get* (always starts at head) looking for target value

Return elapsed time in nano seconds

```java
public static Long loopTest2(SinglyLinked<Integer> list, Integer targetValue) {
   long startTime = System.nanoTime();
   //use iterator to not start back at head each time
   Iterator<Integer> iter = list.iterator();
   while (iter.hasNext()) {
      if (iter.next() == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}
```

Record start time

Loop over all items using iterator (remembers where it was in the list when last called) looking for a target value

Return elapsed time in nano seconds

```java
public static void main(String[] args) throws Exception {
   //add numberOfItems to list
   SinglyLinked<Integer> list = new SinglyLinked<>();
   int numberOfItems = 1000;
   for (int i = 0; i < numberOfItems; i++) {
      list.add(i);
   }
   Long time1 = loopTest1(list,numberOfItems-1);
   System.out.printf("method 1 took %,15d nanoseconds\n",time1);
   Long time2 = loopTest2(list,numberOfItems-1);
   System.out.printf("method 2 took %,15d nanoseconds\n", time2);
   System.out.println("ratio time1/time2: " + time1/(float)time2);
}
```

An iterator can dramatically speed up execution time

TimeTest.java

```
public static Long loopTest1(SinglyLinked<Integer> list, Integer targetValue) throws Exception {
   //use get, start back at head each time through loop
   long startTime = System.nanoTime();
   for (int i = 0; i < list.size(); i++) {
      Integer value = list.get(i);
      if (value == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}
```

Record start time
Loop over all items using *get* (always starts at head) looking for target value
Return elapsed time in nano seconds

```
public static Long loopTest2(SinglyLinked<Integer> list, Integer targetValue) {
   long startTime = System.nanoTime();
   //use iterator to not start back at head each time
   Iterator<Integer> iter = list.iterator();
   while (iter.hasNext()) {
      if (iter.next() == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}
```

Record start time
Loop over all items using iterator (remembers where it was in the list when last called) looking for a target value
Return elapsed time in nano seconds

```
public static void main(String[] args) throws Exception {
   //add numberOfItems to list
   SinglyLinked<Integer> list = new SinglyLinked<>();
   int numberOfItems = 1000;
   for (int i = 0; i < numberOfItems; i++) {
      list.add(i);
   }
   Long time1 = loopTest1(list,numberOfItems-1);
   System.out.printf("method 1 took %,15d nanoseconds\n",time1);
   Long time2 = loopTest2(list,numberOfItems-1);
   System.out.printf("method 2 took %,15d nanoseconds\n", time2);
   System.out.println("ratio time1/time2: " + time1/(float)time2);
}
```

Create SinglyLinked list
Add 1,000 integers (rather small amount)
Call both methods and compare execution time

An iterator can dramatically speed up execution time

TimeTest.java

```java
public static Long loopTest1(SinglyLinked<Integer> list, Integer targetValue) throws Exception {
   //use get, start back at head each time through loop
   long startTime = System.nanoTime();
   for (int i = 0; i < list.size(); i++) {
      Integer value = list.get(i);
      if (value == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}

public static Long loopTest2(SinglyLinked<Integer> list, Integer targetValue) {
   long startTime = System.nanoTime();
   //use iterator to not start back at head each time
   Iterator<Integer> iter = list.iterator();
   while (iter.hasNext()) {
      if (iter.next() == targetValue) {
         break;
      }
   }
   return = System.nanoTime() - startTime;
}

public static void main(String[] args) throws Exception {
   //add numberOfItems to list
   SinglyLinked<Integer> list = new SinglyLinked<>();
   int numberOfItems = 1000;
   for (int i = 0; i < numberOfItems; i++) {
      list.add(i);
   }
   Long time1 = loopTest1(list,numberOfItems-1);
   System.out.printf("method 1 took %,15d nanoseconds\n",time1);
   Long time2 = loopTest2(list,numberOfItems-1);
   System.out.printf("method 2 took %,15d nanoseconds\n", time2);
   System.out.println("ratio time1/time2: " + time1/(float)time2);
}
```

Output

```
method 1 took       2,944,125 nanoseconds
method 2 took          83,125 nanoseconds
ratio time1/time2: 35.418045
```

Using *get* took 35 times longer than using iterator and the list only had 1,000 items!

Results highly variable (we will see why later in the course)

Agenda

1. Iterators

Key points:

1. **Lists do not specify the number of elements they hold (unlike arrays)**
2. **We can grow an array as needed to implement a List**

2. Growing array List ADT implementation
3. Amortized analysis of growth operation
4. Comparing List implementations

Linked lists are a logical choice to implement the List ADT

List ADT features	Linked List
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there

Linked lists are a logical choice to implement the List ADT

List ADT features	Linked List
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there

Linked lists are a logical choice to implement the List ADT

List ADT features	Linked List
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast 
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size 

Or is it?

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size

Random access aspect of arrays makes it easy to get or set any element

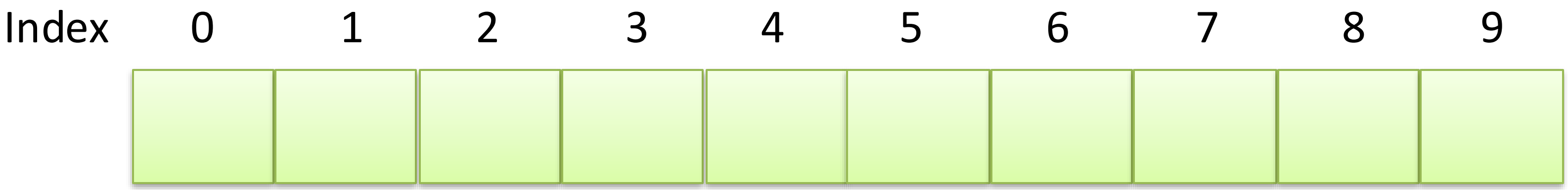
```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

- **Array reserves a contiguous block of memory**
- **Big enough to hold specified number of elements (10 here) times size of each element (4 bytes for integers) = 40 bytes**
- **Indices are 0…9**

Random access aspect of arrays makes it easy to get or set any element




```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console ☒ Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2							

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

No need to march down list to get or set element

To find element:

- **Start at base address of array (this is where "*numbers*" array points)**
- **Element at index *idx* is at address:**
  base addr* + *idxsize(element)**

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2							

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

No need to march down list to get or set element

To find element:

- **Start at base address of array (this is where "*numbers*" array points)**
- **Element at index *idx* is at address:**
  ***base addr* + *idx**size(element)**
- **Index 2 at *base addr + 2*4* bytes**
- **Java does this math for us**
- **Time to access element is constant anywhere in array (just simple math operation to calculate any index)**
- **With linked list must march down list, takes longer to find elements at end**

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = numbers[2];
14         int b = numbers[5];
15         int c = numbers[1]; //we did not set this
16         System.out.println("a="+a+" b="+b+" c="+c);
17     }
18 }
19
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

What List ADT operations are like lines 9 and 10?

set

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = numbers[2];
14         int b = numbers[5];
15         int c = numbers[1]; //we did not set this
16         System.out.println("a="+a+" b="+b+" c="+c);
17     }
18 }
19
```

Problems @ Javadoc Declaration Console ☒ Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

What List ADT operations are like lines 13-15?

get

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}

```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

What values will a, b and c have?

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

```
a=2 b=10 c=0
```

What values will a, b and c have?

a=2, b=10, c=0

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size

Because arrays are a contiguous block of memory, hard to insert (except at end)

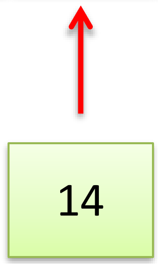
Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	0	0	0	0

14

Insert 14 at index 2

Because arrays are a contiguous block of memory, hard to insert (except at end)

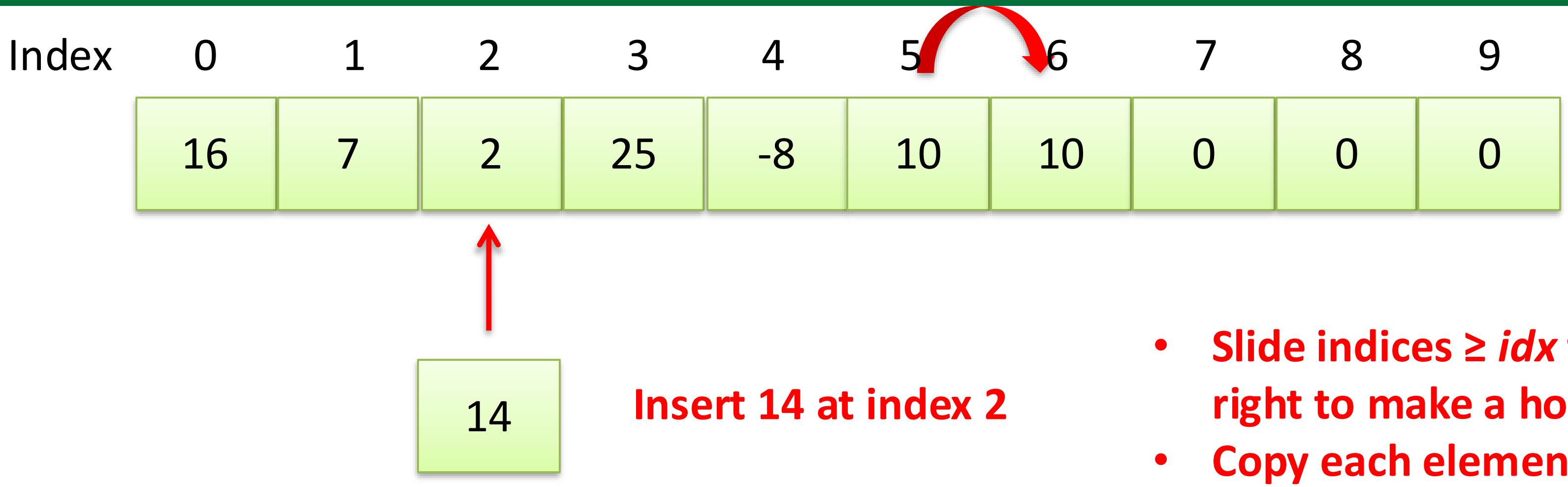
Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	0	0	0	0



Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

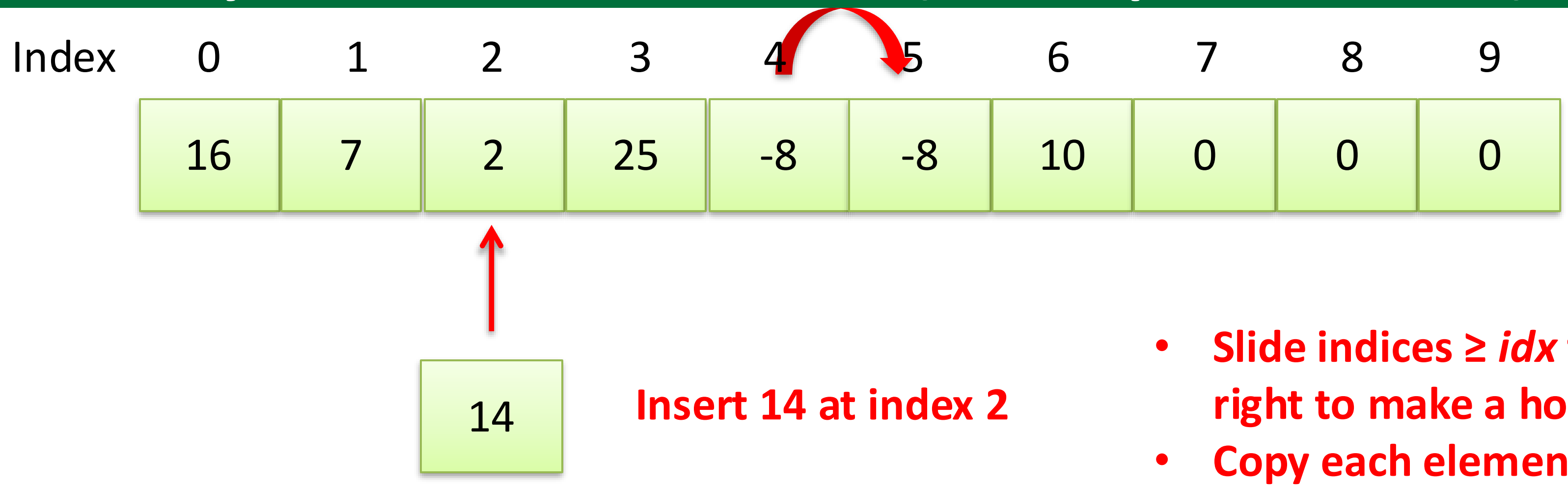
Because arrays are a contiguous block of memory, hard to insert (except at end)



Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

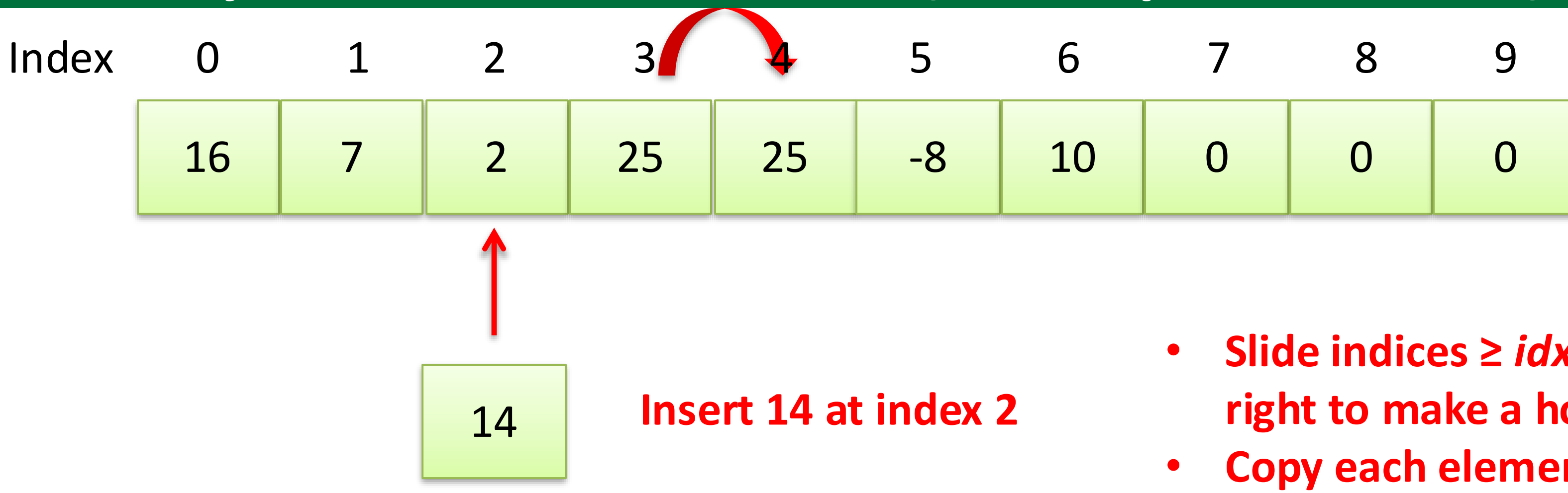
Because arrays are a contiguous block of memory, hard to insert (except at end)



Insert 14 at index 2

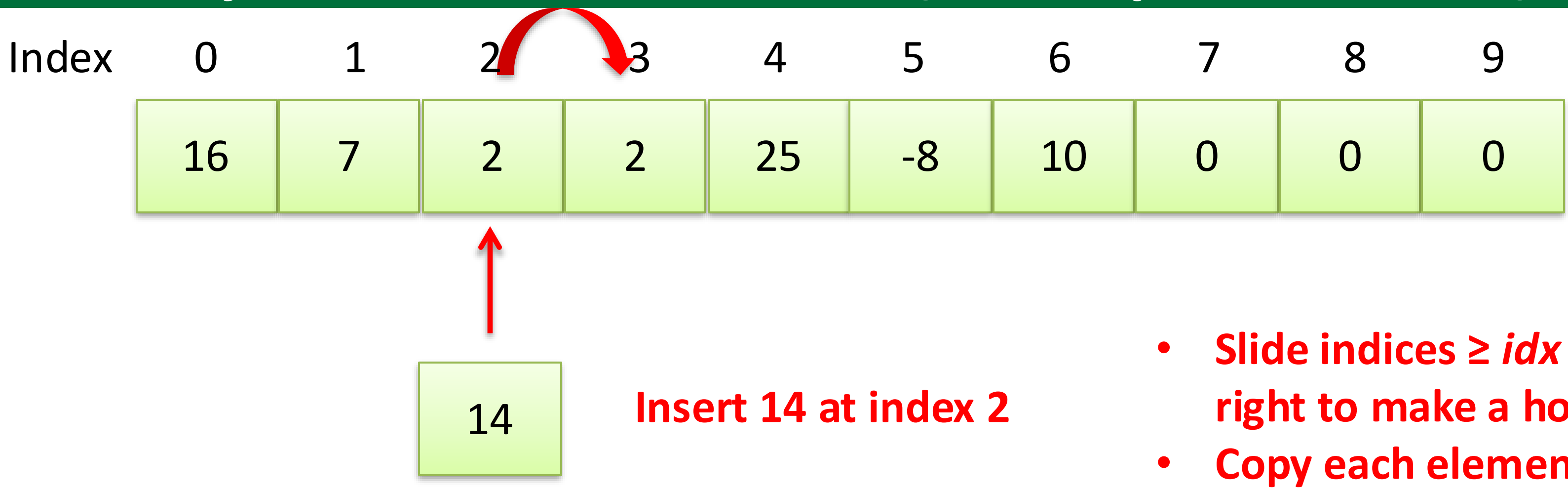
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)



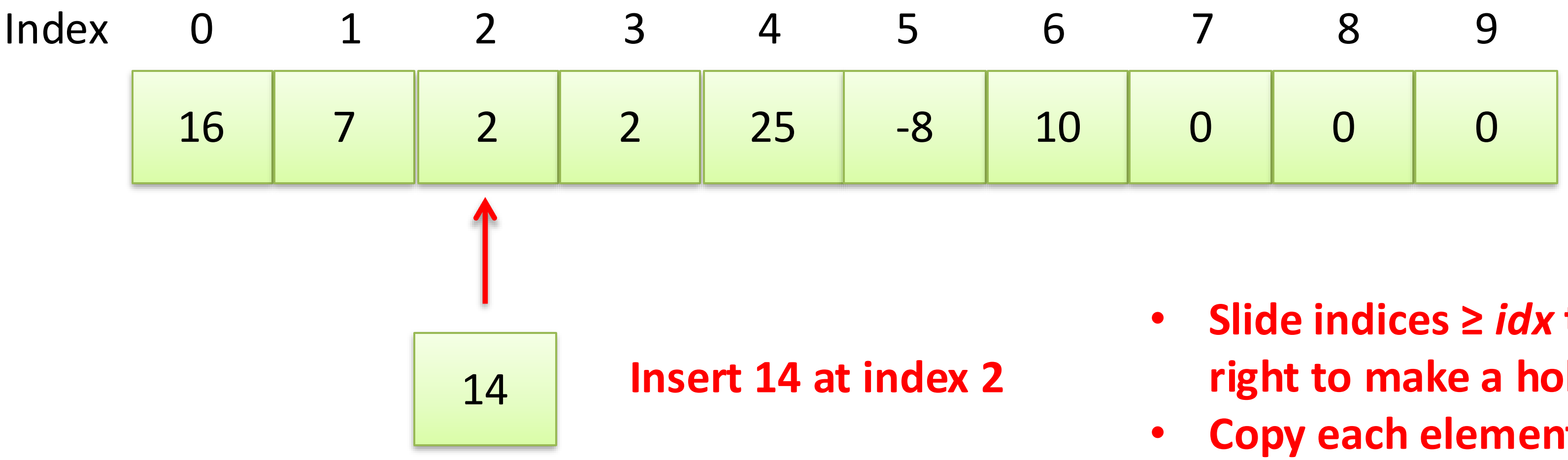
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)



- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)



Insert 14 at index 2

- Slide indices ≥ *idx* to the right to make a hole
- Copy each element to next index

Copy new element into index

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	14	2	25	-8	10	0	0	0

- Works, but takes a lot of time (said to be “expensive”)
- Especially expensive with respect to time if the array is large and we insert at the front (but fast at end!)
- Linked list is slow to find the right place (must march down list starting from head), but fast to insert, just update two pointers and you’re done
- Linked list is fast, however, if only dealing with head
- With arrays, easy to find right place, but slow afterward due to copying to make a hole

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	14	2	25	-8	10	0	0	0

Deleting an element is the same except copy elements to the left to remove the deleted element

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size 

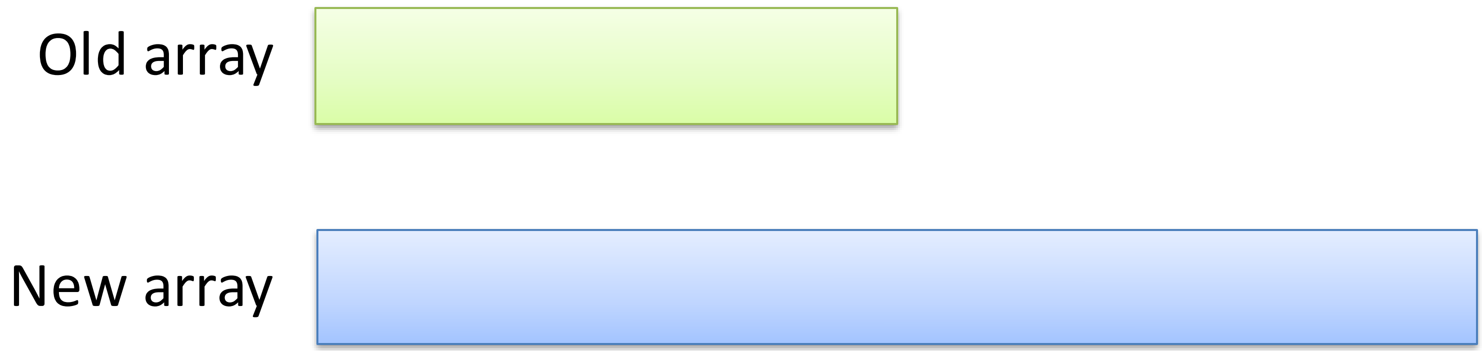
Arrays are of fixed size, but List ADT allows for growth

Index	0	1	2	3	4	5	6	7	8	9
	16	7	14	2	25	-8	10	52	-19	6

What do we do when the array is full, but we want to add more elements?

Answer: create another, larger array, and copy elements from old array into new array

Arrays are of fixed size, but List ADT allows for growth



Grow array

1. **Make new array, say 2 times larger than old array**

Arrays are of fixed size, but List ADT allows for growth

Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

Grow array

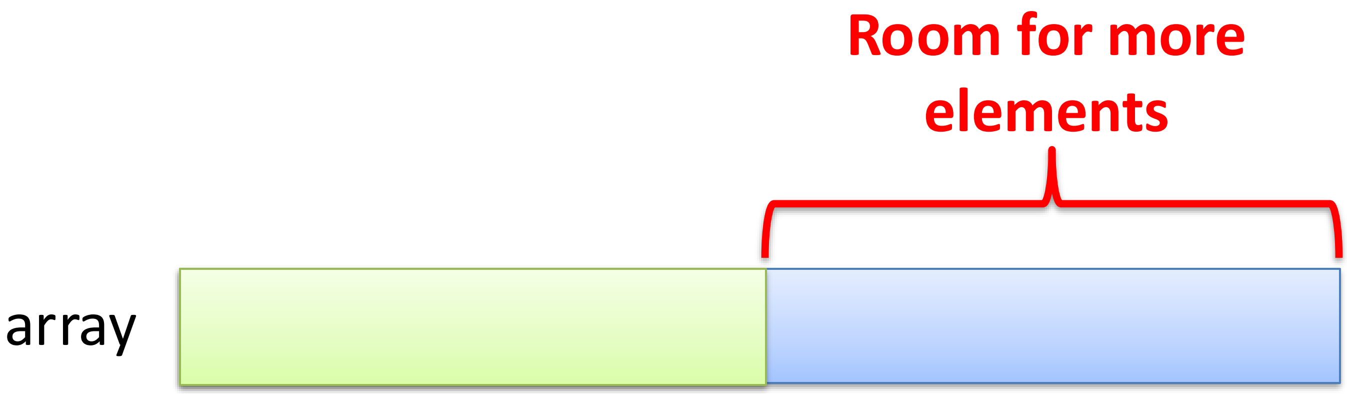
1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

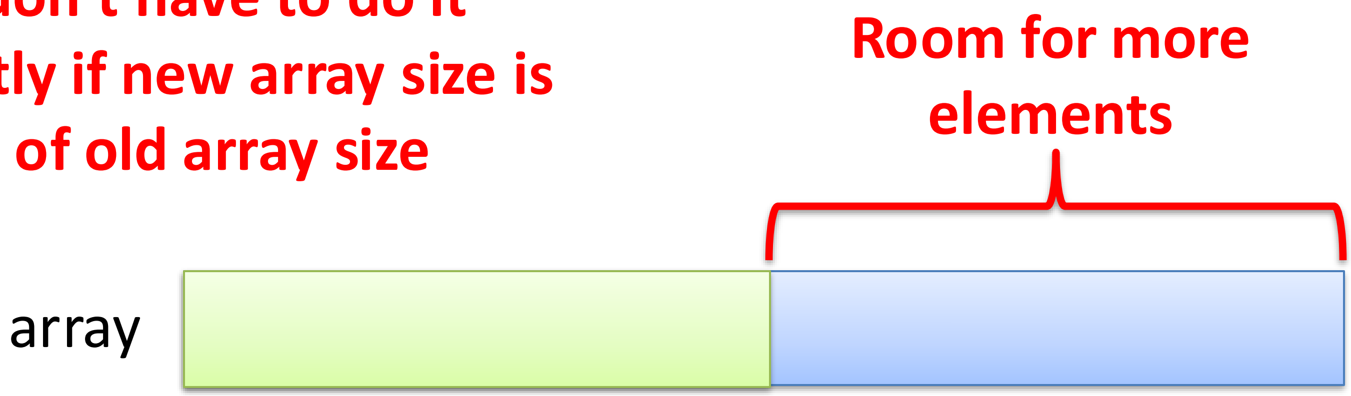


Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**
3. **Set instance variable to point at new array (old array will be garbage collected)**

Arrays are of fixed size, but List ADT allows for growth

Growing is expensive operation, but we don't have to do it frequently if new array size is multiple of old array size



Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**
3. **Set instance variable to point at new array (old array will be garbage collected)**

GrowingArray.java: implements List ADT using an array instead of a linked list

```
public class GrowingArray<T> implements SimpleList<T>, Iterable<T> {
  private T[] array;
  private int size;     // how much of the array is actually filled up so far
  private static final int initCap = 10; // how big the array should be initially

  public GrowingArray() {
    array = (T[]) new Object[initCap];  // java generics oddness – cast array of objects
    size = 0;
  }

  /**
   * Return the number of elements in the List (they are indexed 0..size-1)
   * @return number of elements
   */
  public int size() {
    return size;
  }
```

Implements SimpleList and Iterable

Array is now the data structure used to store elements in List

- **Array initially sized to 10 Objects (note the funky Java allocation syntax, must cast to array of generic type)**
- **Remember, arrays are of fixed size, but the List ADT does not specify a size**

Track size
Will increment on each *add* and decrement on each *remove*
Run-time complexity for *size* method?
O(1)

GrowingArray.java: *get()/set()* are easy and fast with an array implementation

```
/**
 * Return item at index idx
 * @param idx index of item to return
 * @return item stored at index idx
 * @throws Exception invalid index
 */
public T get(int idx) throws Exception {
   if (idx >= 0 && idx < size) return array[idx];
   else throw new Exception("invalid index");
}

/**
 * Overwrite item at index idx with item parameter
 * @param idx index of item to get
 * @param item overwrite existing item at index idx with this item
 * @throws Exception invalid index
 */
public void set(int idx, T item) throws Exception {
   if (idx >= 0 && idx < size) array[idx] = item;
   else throw new Exception("invalid index");
}
```

Get and set are easy, just make sure index is valid, then return or set item

Notice: no curly braces!

Only next line in if statement

Run-time complexity?
O(1) for any index!
Just two math operations to compute memory address

GrowingArray.java: With growing trick, can implement the List interface with an array

```java
public void add(int idx, T item) throws Exception {
    if (idx > size || idx < 0) throw new Exception("invalid index");
    if (size == array.length) {
      // Double the size of the array, to leave more space
      T[] copy = (T[]) new Object[size*2];
      // Copy it over
      for (int i=0; i<size; i++) copy[i] = array[i];
      array = copy;
    }
    // Shift right to make room
    for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
    array[idx] = item;
    size++;
}
```

array.length **is how many elements *array* *can* hold**

size **has how many elements *array* *does* hold**

add() **makes a new, larger array if needed**

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
     // Double the size of the array, to leave more space
     T[] copy = (T[]) new Object[size*2];
     // Copy it over
     for (int i=0; i<size; i++) copy[i] = array[i];
     array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}
```

array.length **is how many elements *array* <u>*can*</u> hold**

size **has how many elements *array* <u>*does*</u> hold**

add() **makes a new, larger array if needed**

Copy elements one at a time into new array

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
    if (idx > size || idx < 0) throw new Exception("invalid index");
    if (size == array.length) {
      // Double the size of the array, to leave more space
      T[] copy = (T[]) new Object[size*2];
      // Copy it over
      for (int i=0; i<size; i++) copy[i] = array[i];
      array = copy;
    }
    // Shift right to make room
    for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
    array[idx] = item;
    size++;
}
```

array.length **is how many elements** ***array*** ***can*** **hold**

size **has how many elements** ***array*** ***does*** **hold**

add() **makes a new, larger array if needed**

Update instance variable to new array

Copy elements one at a time into new array

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
     // Double the size of the array, to leave more space
     T[] copy = (T[]) new Object[size*2];
     // Copy it over
     for (int i=0; i<size; i++) copy[i] = array[i];
     array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}
```

- **Here we know we have enough room to add a new element**
- **Now do insert**
- **Start from last item and copy to one index larger**
- **Stop at index *idx***
- **Set item at *idx* to item**

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
  if (idx > size || idx < 0) throw new Exception("invalid index");
  if (size == array.length) {
    // Double the size of the array, to leave more space
    T[] copy = (T[]) new Object[size*2];
    // Copy it over
    for (int i=0; i<size; i++) copy[i] = array[i];
    array = copy;
  }
  // Shift right to make room
  for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
  array[idx] = item;
  size++;
}

public void add(T item) throws Exception {
  add(size,item);
}
```

Add an item at the end is easy
Just call *add* with *size* as index

What did we call it when two methods have the same name but different variables?
Overloading

GrowingArray.java: With growing trick, can implement the List interface with an array

```
/**
 * Remove and return the item at index idx.  Move items left to fill hole.
 * @param idx index of item to remove
 * @return the value previously at index idx
 * @throws Exception invalid index
 */
public T remove(int idx) throws Exception {
   if (idx > size-1 || idx < 0) throw new Exception("invalid index");
   T data = array[idx];
   // Shift left to cover it over
   for (int i=idx; i<size-1; i++) array[i] = array[i+1];
   size--;
   return data;
}
```

***remove()* slides elements left one slot for index > idx**

Run-time complexity?
O(n)
Where is the worst place for a remove?
Index 0

It turns out array could be a good choice to implement List ADT, if growing is fast

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Must make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size

Can get around array growth limit

Want to make sure growth is fast enough

Growing array is *generally* preferable to linked list, except maybe growth operation

Worst case run-time complexity

	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	O(n) + growth
remove(i)	O(n)	O(n)

- **Start at *head* and march down to find index *i***
- **Slow to get to index, O(n)**
- **Once there, operations are fast O(1)**
- **Best case: all operations on head**
- **If constrain to only operate at head, all operations become O(1)**

- **Faster *get()/set()* than linked list**
- **Tie with linked list on *remove()***
- **Best case: all operation at tail**
- ***add()* might cause expensive growth operation**
- **How should we think about that?**

Agenda

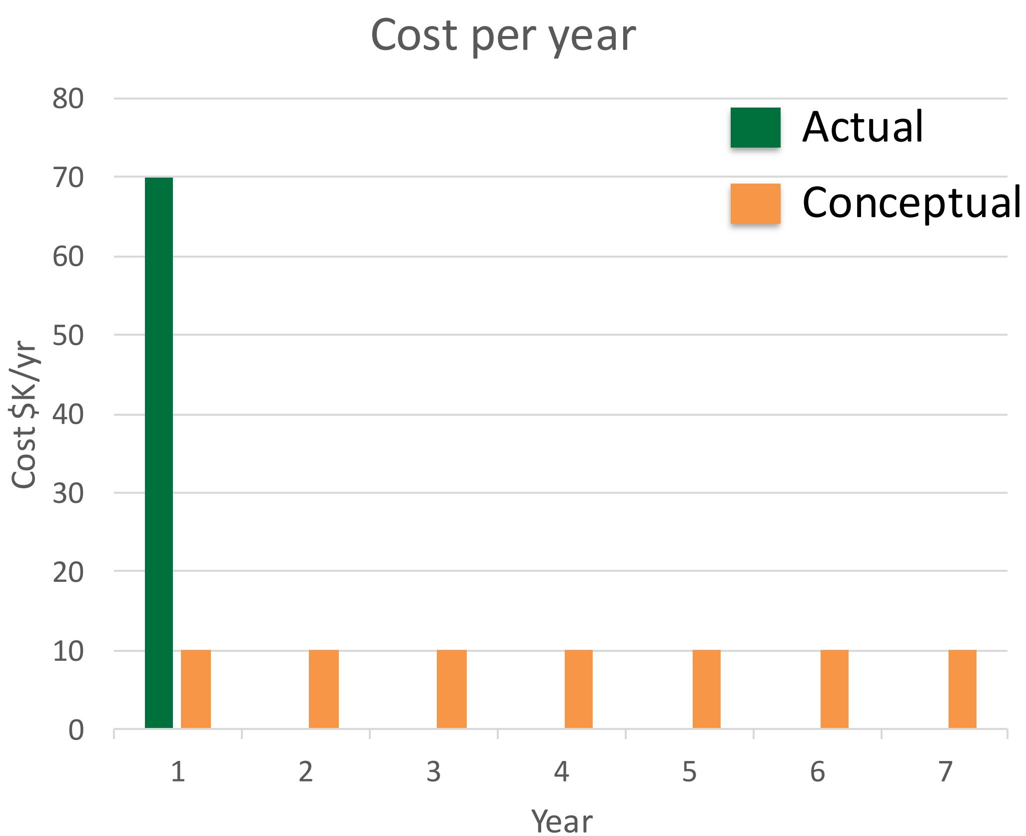
1. Iterators
2. Growing array List ADT implementation
3. Amortized analysis of growth operation
4. Comparing List implementations

Key points:

1. **Amortized analysis shows growing an array is a constant time operation!**

Amortization is a concept from accounting that allows us to spread costs over time

Amortized analysis

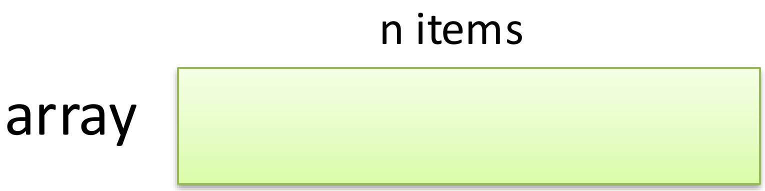


Accounting allows us to amortize costs over several years

- Buy $70K truck on year 1
- Truck is good for 7 years
- Can think of the cost as $10K/year instead of one payment of $70K on year 1
- Actually pay $70K on year 1, but this is equivalent to paying $10K/year for 7 years
- Idea is to spread the cost ("amortize" the cost) over the lifetime of the truck
- We will use this concept to "pre-pay" for expensive growth operation

Amortized analysis shows growing array is actually only O(1)!

Amortized analysis

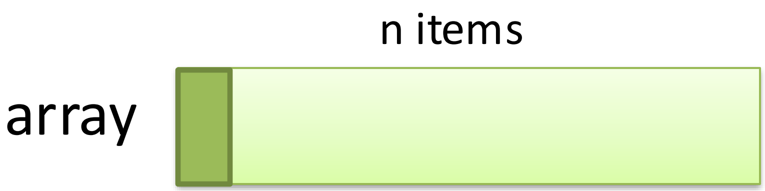


Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

Amortized analysis shows growing array is actually only O(1)!

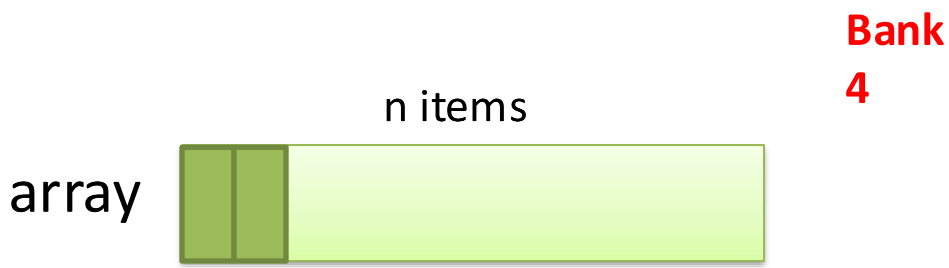
Bank
2



Each time add an item to array, *__conceptually__* charge 3 “tokens”

- One token pays for current add()
- Two tokens go into a conceptual “Bank”
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

Amortized analysis shows growing array is actually only O(1)!

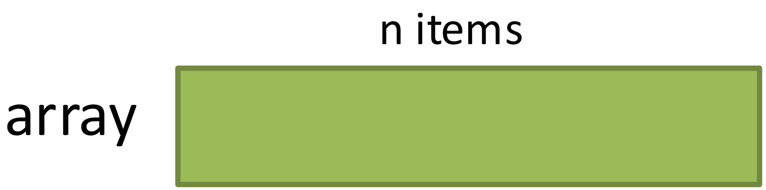


Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

Amortized analysis shows growing array is actually only O(1)!

Bank
2n

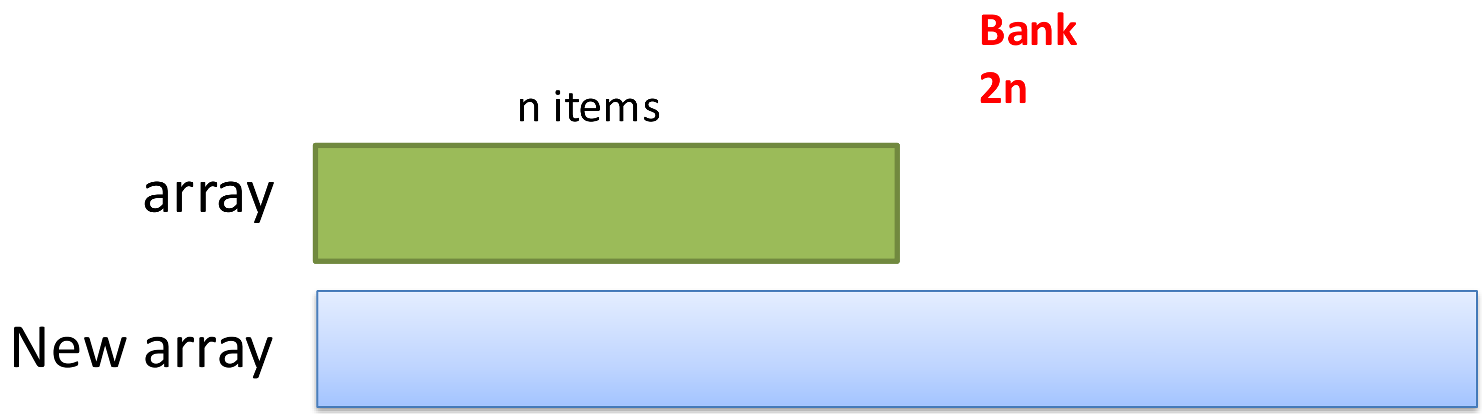


Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Amortized analysis shows growing array is actually only O(1)!



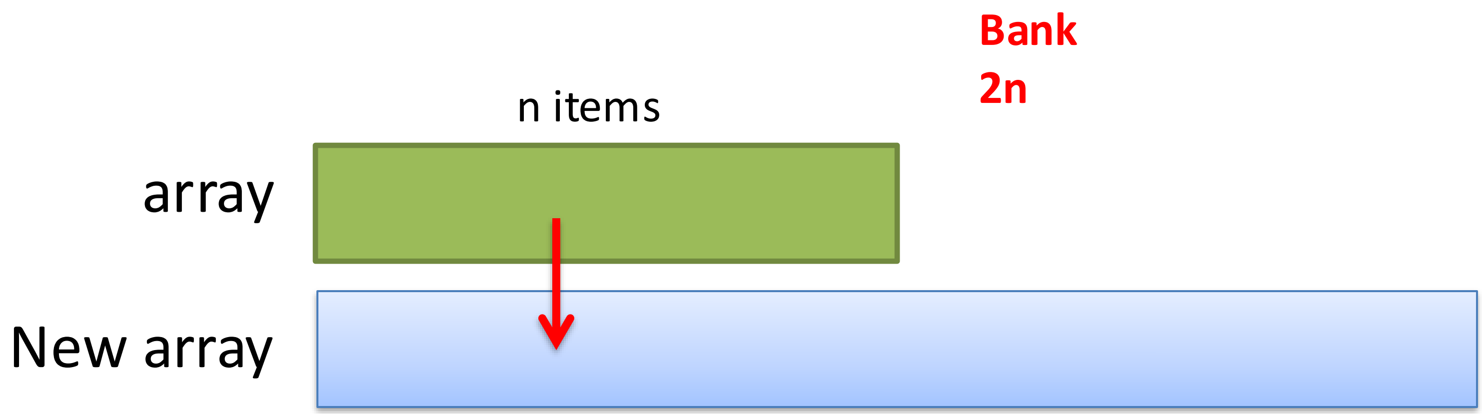
Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *conceptually* charge 3 "tokens"

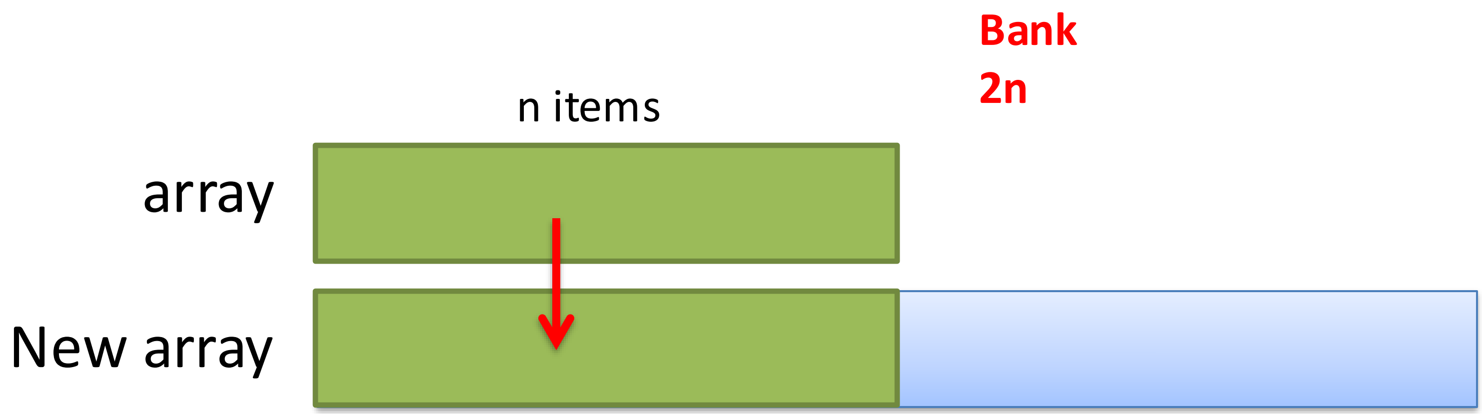
- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *__conceptually__* charge 3 "tokens"

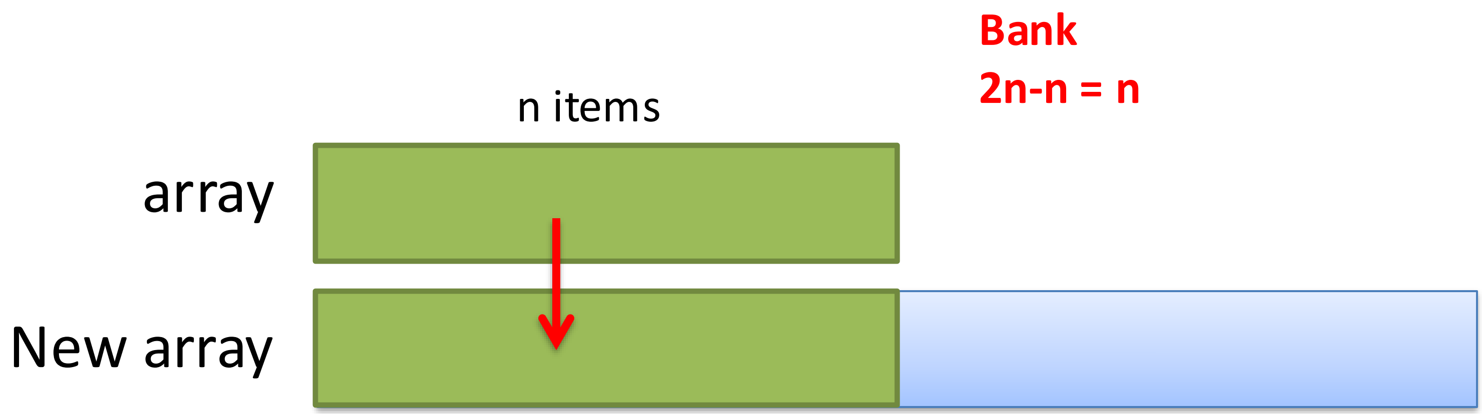
- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

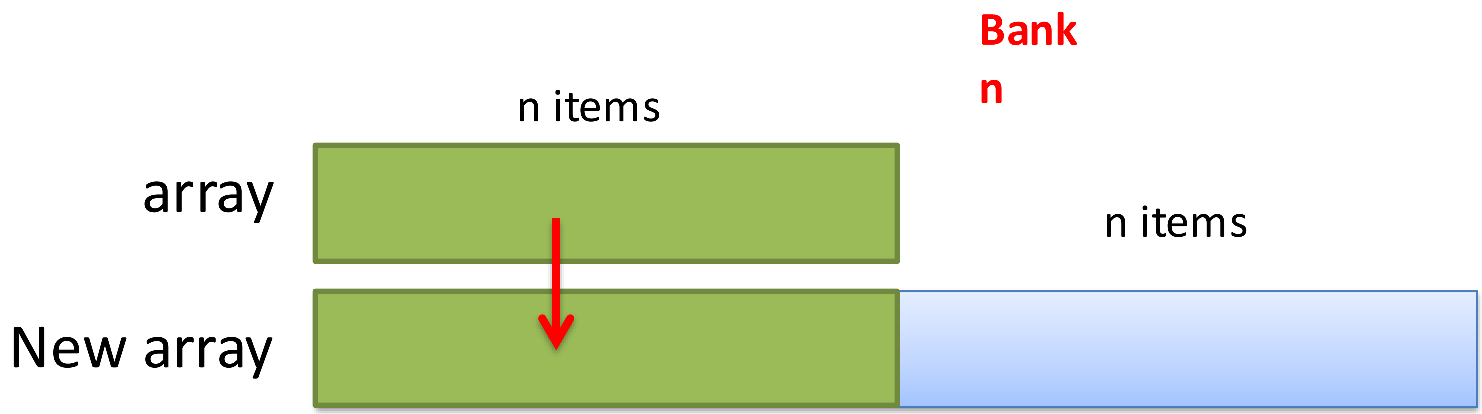
After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *<u>conceptually</u>* charge 3 “tokens”

- One token pays for current add()
- Two tokens go into a conceptual “Bank”
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

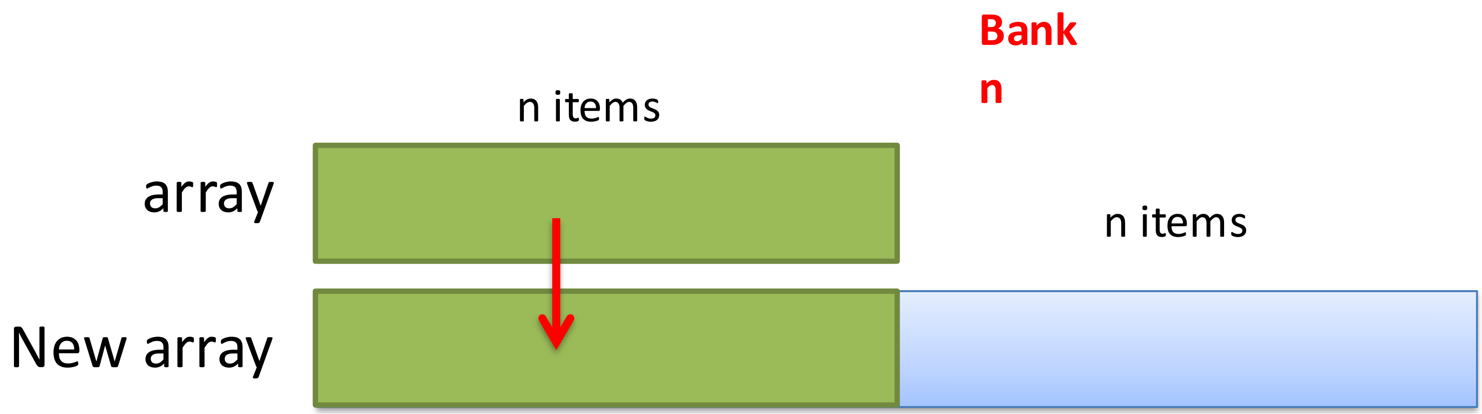
Allocate new 2X larger array

Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Remaining *n* items in bank “pay for” empty *n* spaces

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *<u>conceptually</u>* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

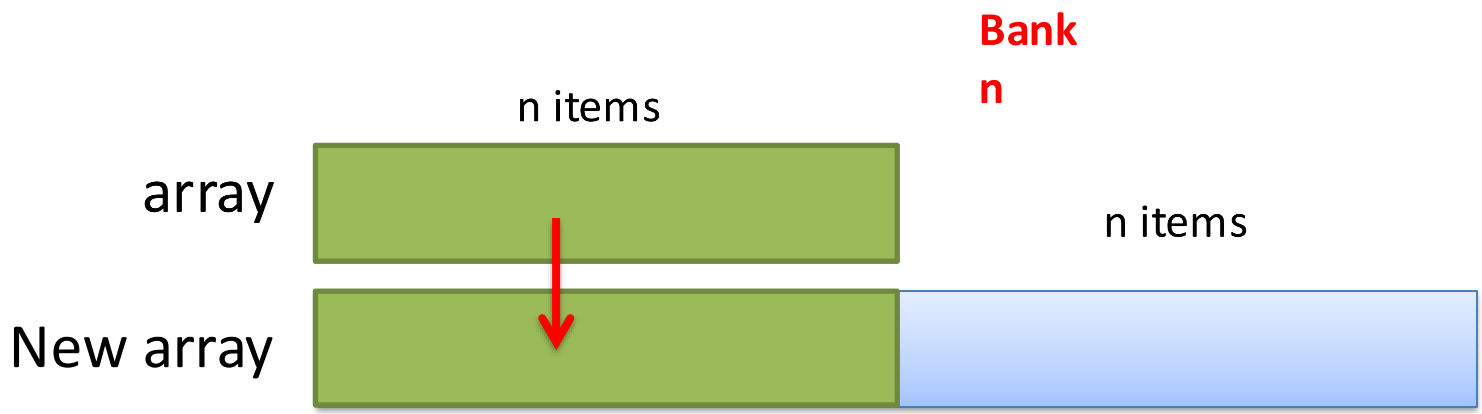
Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Remaining *n* items in bank "pay for" empty *n* spaces

Charging a little extra for each *add* spreads out cost for infrequent growth operation

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *conceptually* charge 3 "tokens"

- One token pays for current add()
- Two tokens go into a conceptual "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Remaining *n* items in bank "pay for" empty *n* spaces

Charging a little extra for each *add* spreads out cost for infrequent growth operation

The charge, however, is a constant, so O(3) = O(1)

Agenda

1. Iterators
2. Growing array List ADT implementation
3. Amortized analysis of growth operation
4. Comparing List implementations

Key points:

1. **An array implementation of the List ADT is generally preferable to a linked list implementation**

Growing array is *generally* preferable to linked list

Worst case run-time complexity

Array implementation has better *get/set* than linked list, ties on *add/remove*

	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	**O(n) + O(1) = O(n)**
remove(i)	O(n)	O(n)

Amortized analysis shows infrequent growth operation is constant time

Pay a constant amount more on each *add()* to pay for the occasional expensive growth

- Start at *head* and march down to find index *i*
- Slow to get to index, O(n)
- Once there, operations are fast O(1)
- Best case: all operations on head
- If constrain to only operate at head, all operations become O(1)

- Faster *get()/set()* than linked list
- Tie with linked list on *remove()*
- Best case: all operations on tail
- *add()* might cause expensive growth operation

Key points

1. Iterators loop over items in a List
2. They do not need to begin at the head at each call
3. Lists do not specify the number of elements they hold (unlike arrays)
4. We can grow an array as needed to implement a List
5. Amortized analysis shows growing an array is a constant time operation!
6. An array implementation of the List ADT is generally preferable to a linked list implementation