CS 10:
Problem solving via Object Oriented Programming

Lists Part 2 (Array’s Revenge!)

Main goals

- Implement growing array list
- Characterize runtime complexity
- Compare list implementations

Agenda

1. Growing array List implementation

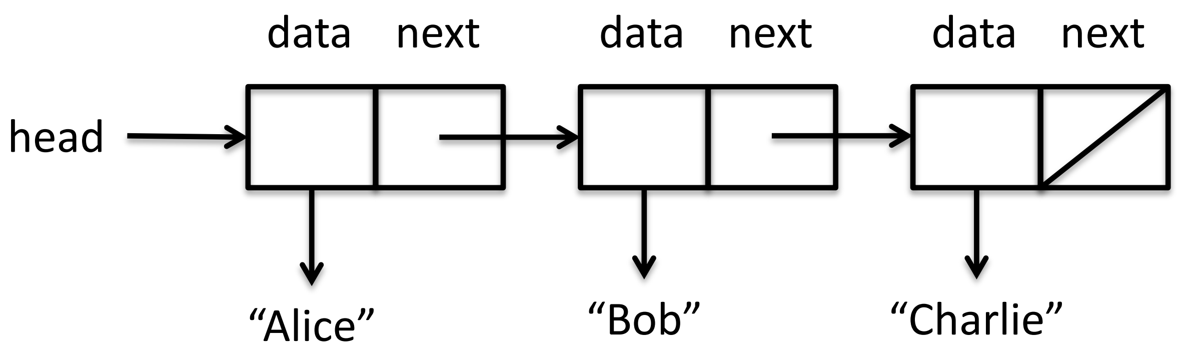
2. List analysis

3. Iteration

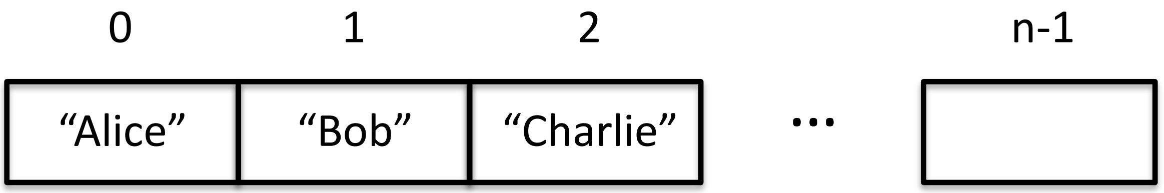
Difference between singly linked list and array

List ADT features
get()/set() element anywhere in List
add()/remove() element anywhere in List
No limit to number of elements in List

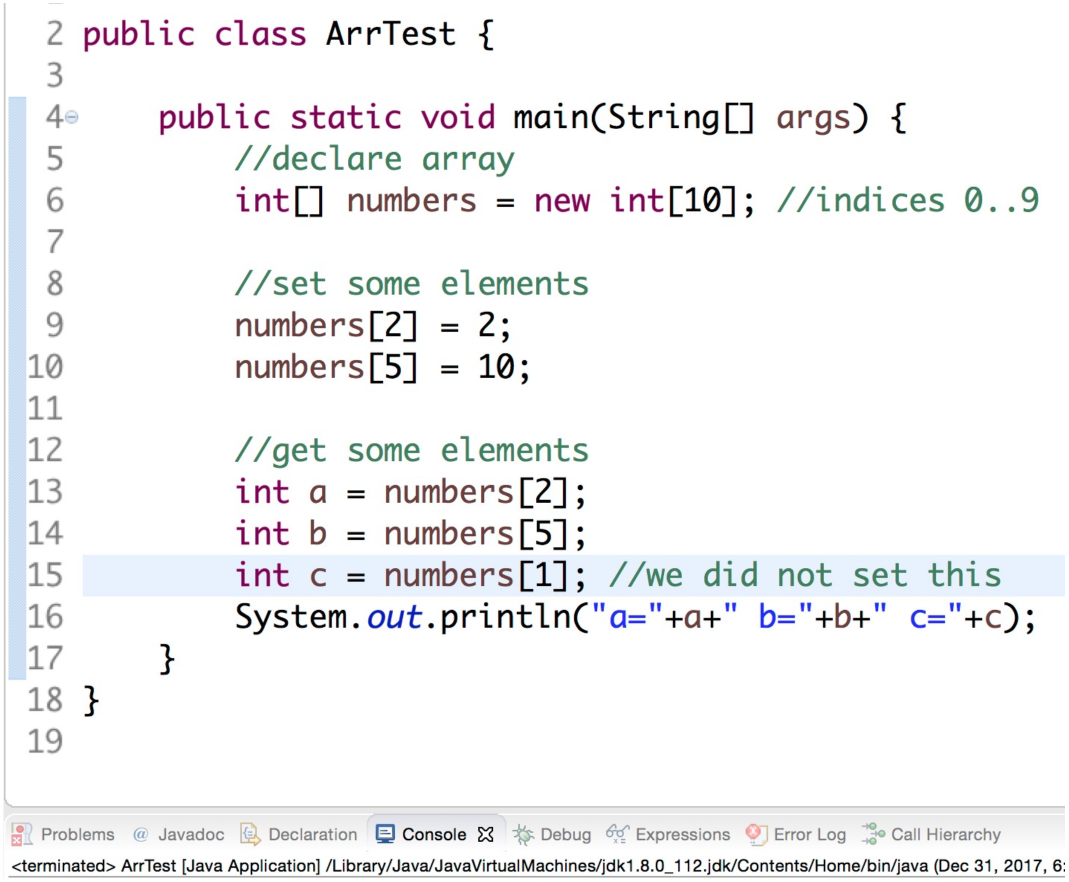
Singly linked list



Array



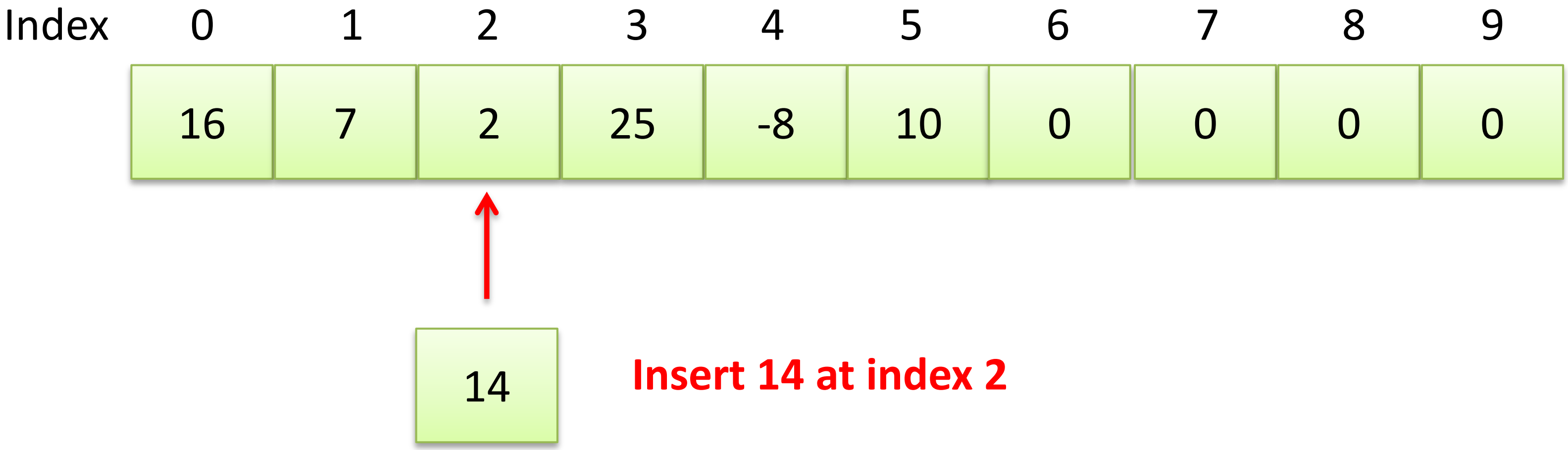
Random access aspect of arrays makes it easy to get or set any element



```
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);
    }
}

```

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

Insertion



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

On paper example

Deletion

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

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

On paper example

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;
   }
```

Run-time complexity?
O(1) for any index!

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");
}
```

Run-time complexity?
O(1) for any 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");
}
```

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++;
}
```

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);
}
```

Run-time complexity O(1)

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;
}
```

Run-time complexity?

O(n)

Agenda

1. Growing array List implementation
2. List analysis
3. Iteration

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

Worst case run-time complexity

	Linked list	Growing array
get(i)		
set(i,e)		
add(i,e)		
remove(i)		

Discussion

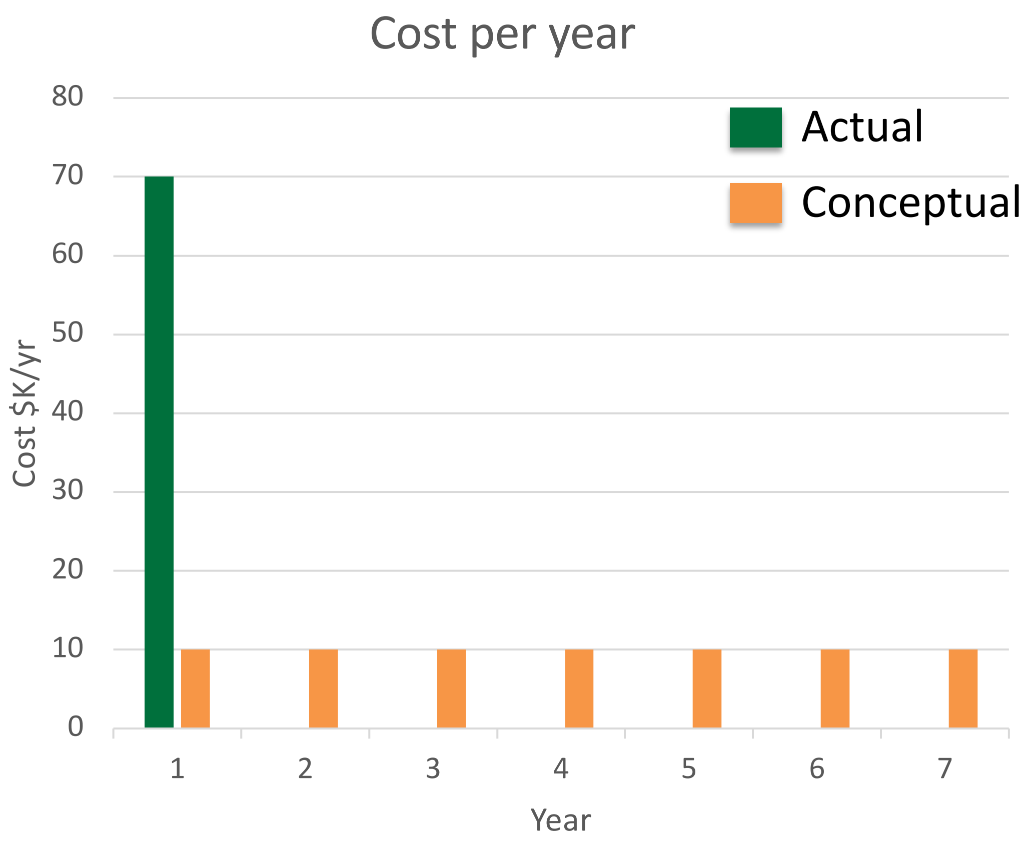
Growing array is *generally* preferable to linked list, except maybe growth operation

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)

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

Amortized analysis

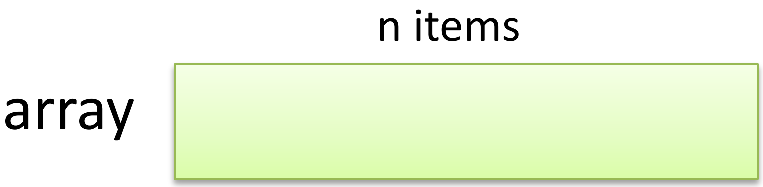


Accounting allows us to amortize costs over several years

- Buy $70K truck on year 1
- Truck is good for 7 years

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

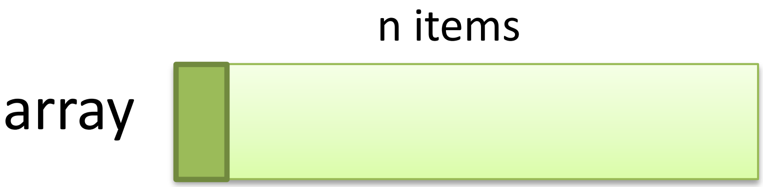
Amortized analysis



n items

array

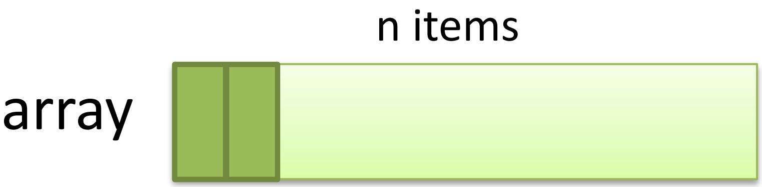
Amortized analysis shows growing array is actually only O(1)!



Bank
2

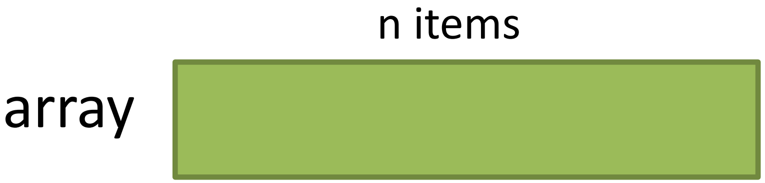
Amortized analysis shows growing array is actually only O(1)!

Bank
4

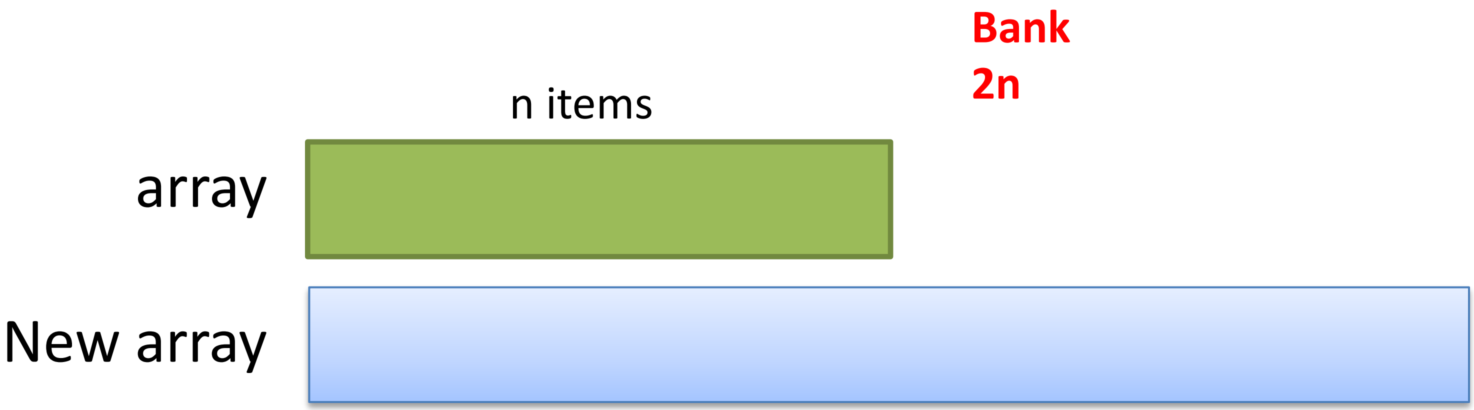


Amortized analysis shows growing array is actually only O(1)!

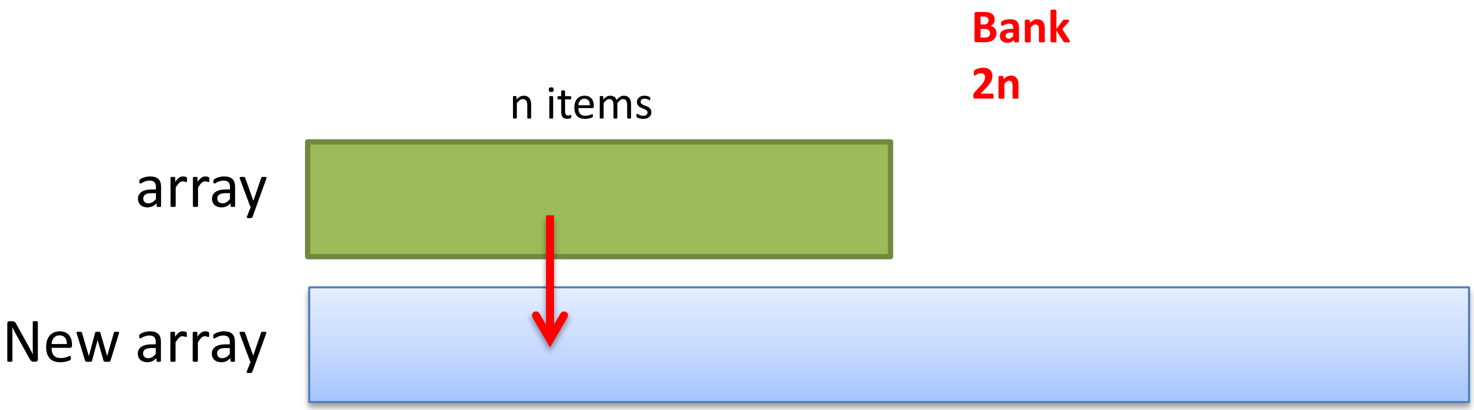
Bank
2n



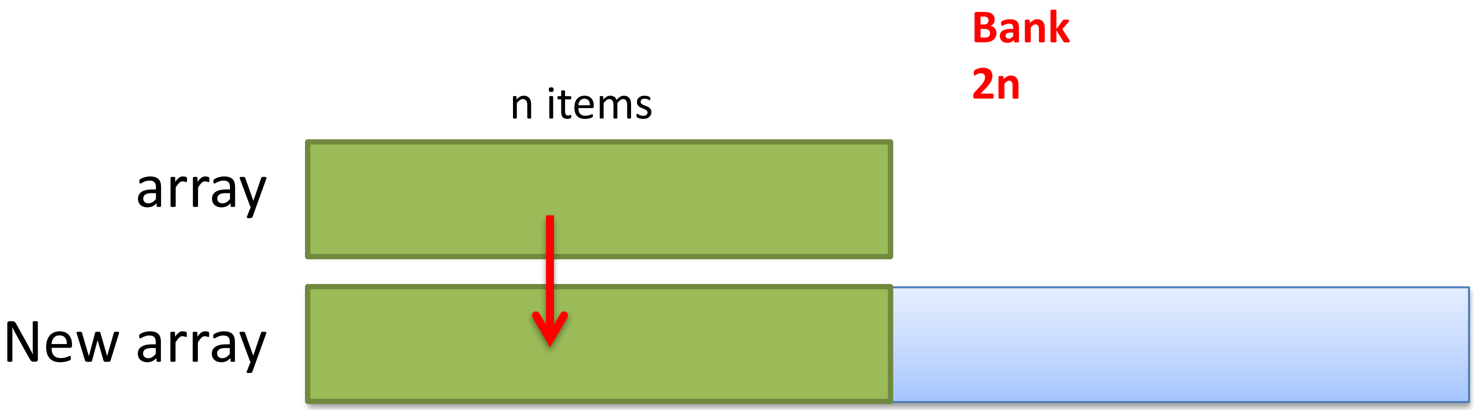
Amortized analysis shows growing array is actually only O(1)!



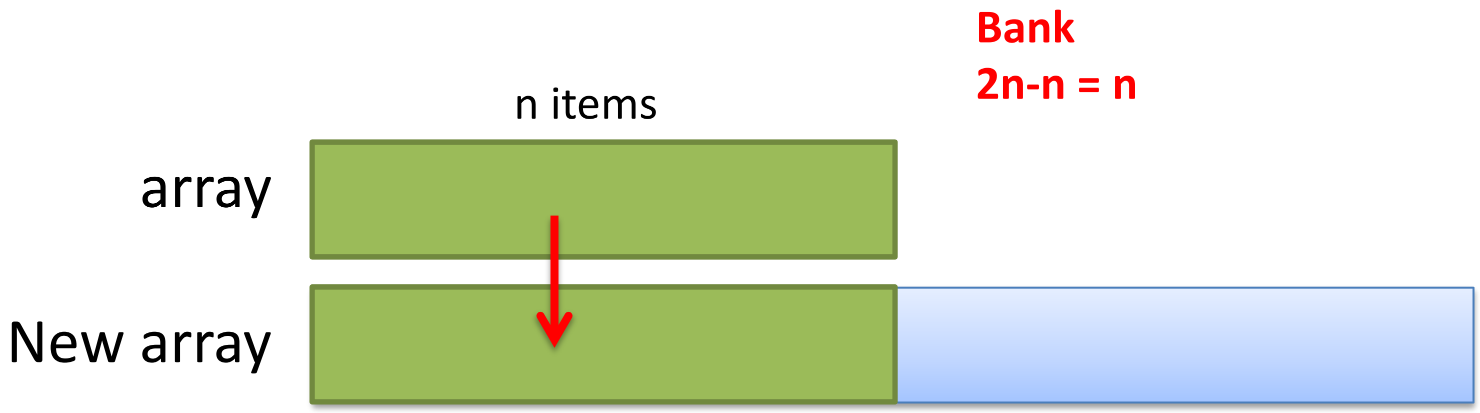
Amortized analysis shows growing array is actually only O(1)!



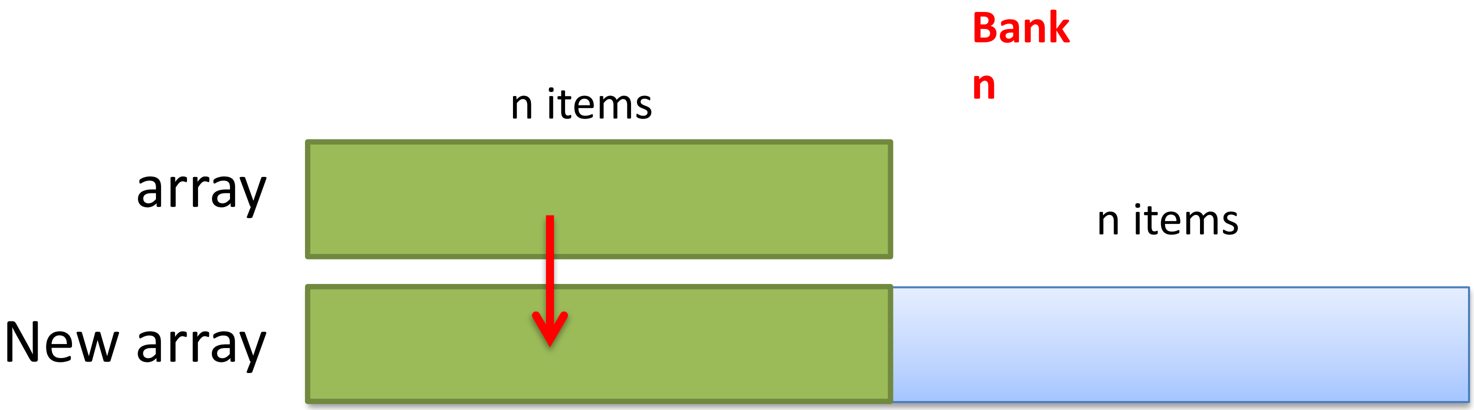
Amortized analysis shows growing array is actually only O(1)!



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



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



Growing array is *generally* preferable to linked list

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) + O(1) = O(n)**
remove(i)	O(n)	O(n)

Summary

- Growing ArrayList implementation
- Runtime complexity analysis
  - Get/set O(1)
  - Add/remove O(n)
    - Amortized analysis for growth operation
- List analysis: SinglyLinkedList vs ArrayList
  - Growing array overall more efficient, unless specific assumptions on operations

Next

- Hierarchical relationships through trees

Additional Resources

DESCRIPTION OF PROS AND CONS

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

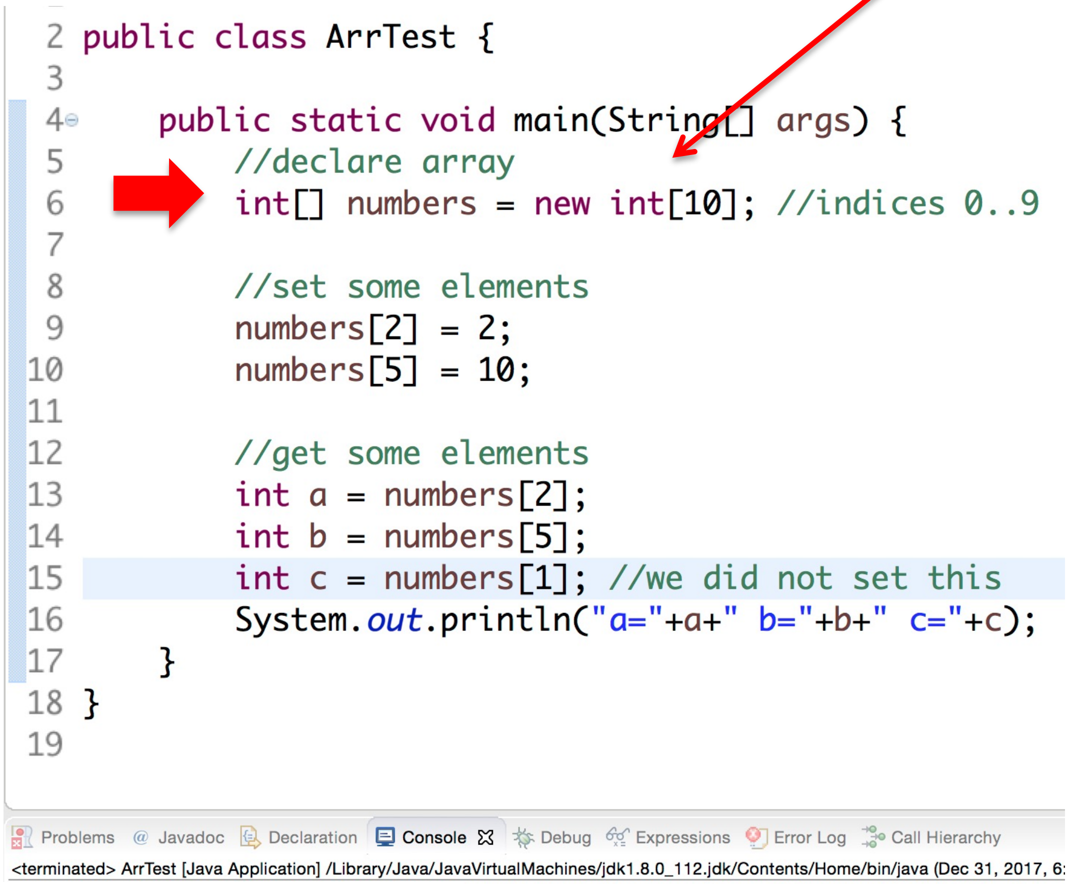
List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• 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 • Have to 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

ArrTest.java

ANNOTATED SLIDES

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

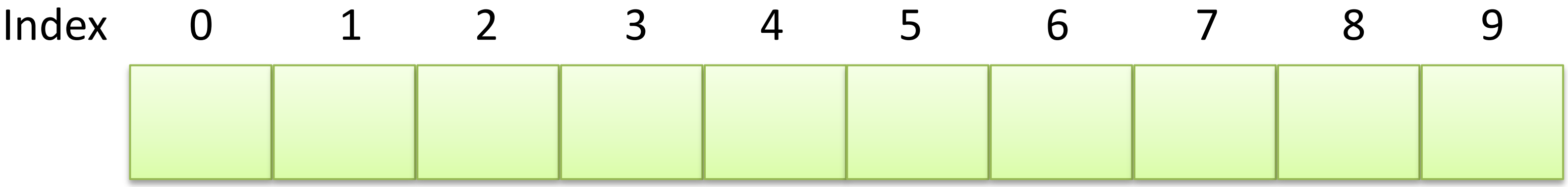
- Array reserves a contiguous block of memory
- Big enough to hold specified number of elements (10 here) times size of each element (4 bytes for integers) = 40 bytes
- Indices are 0…9

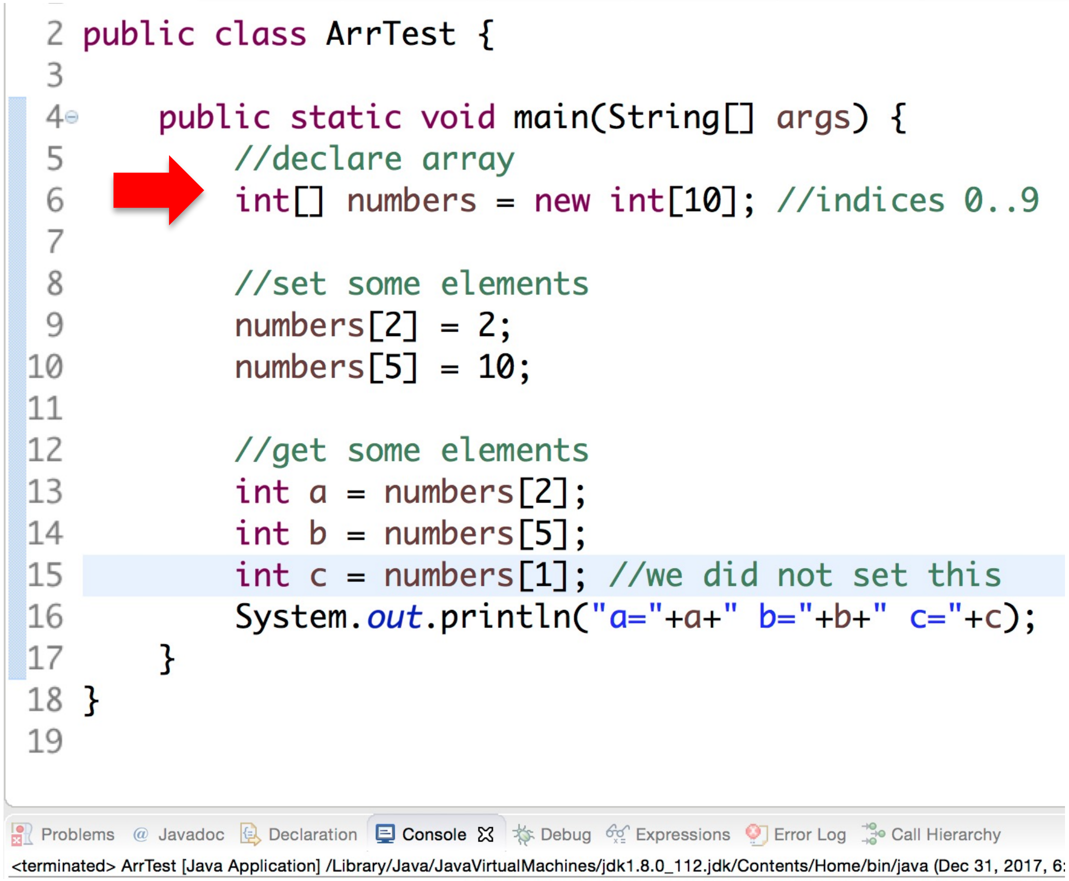


```
public class ArrTest {

    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);
    }
}
```

<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

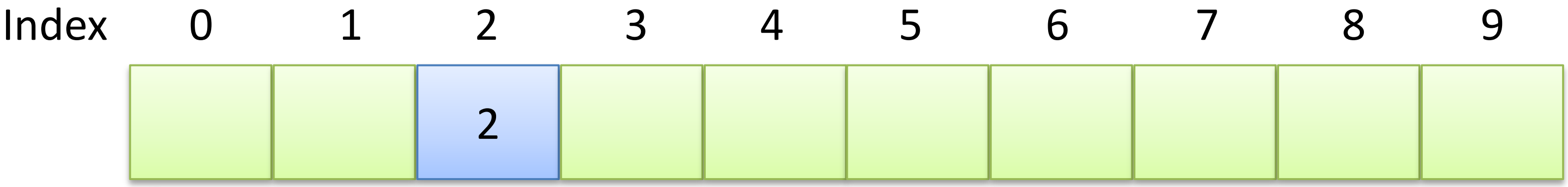


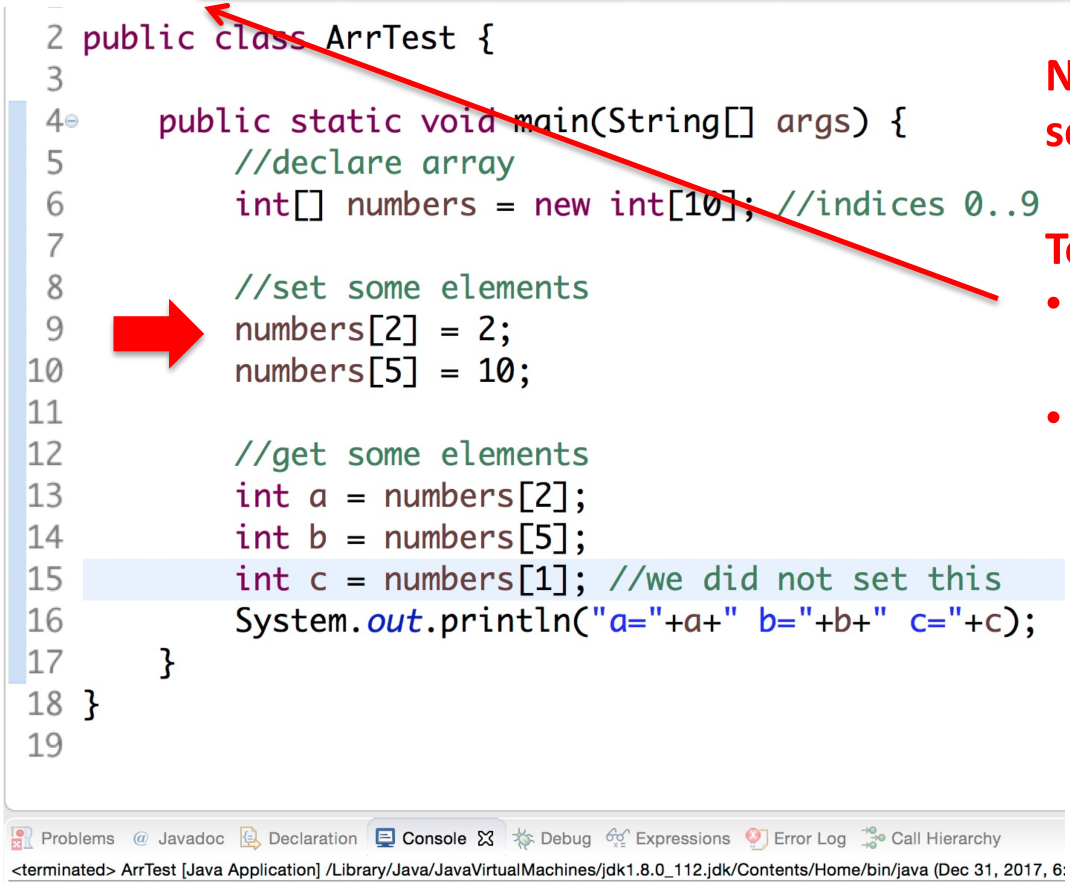


```java
public class ArrTest {

    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);
    }
}
```

<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




```
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);
    }
}
```

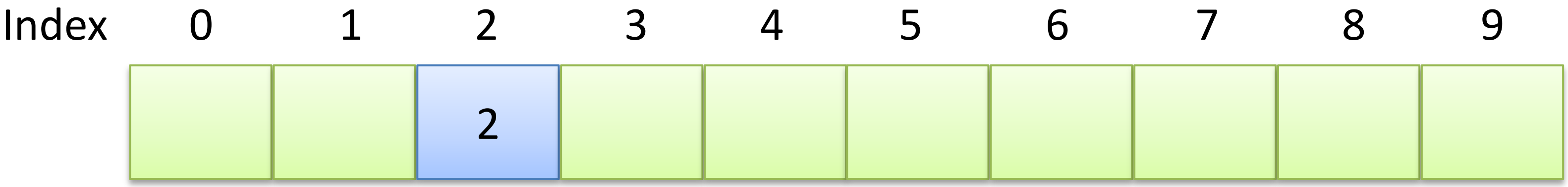
```
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:
```

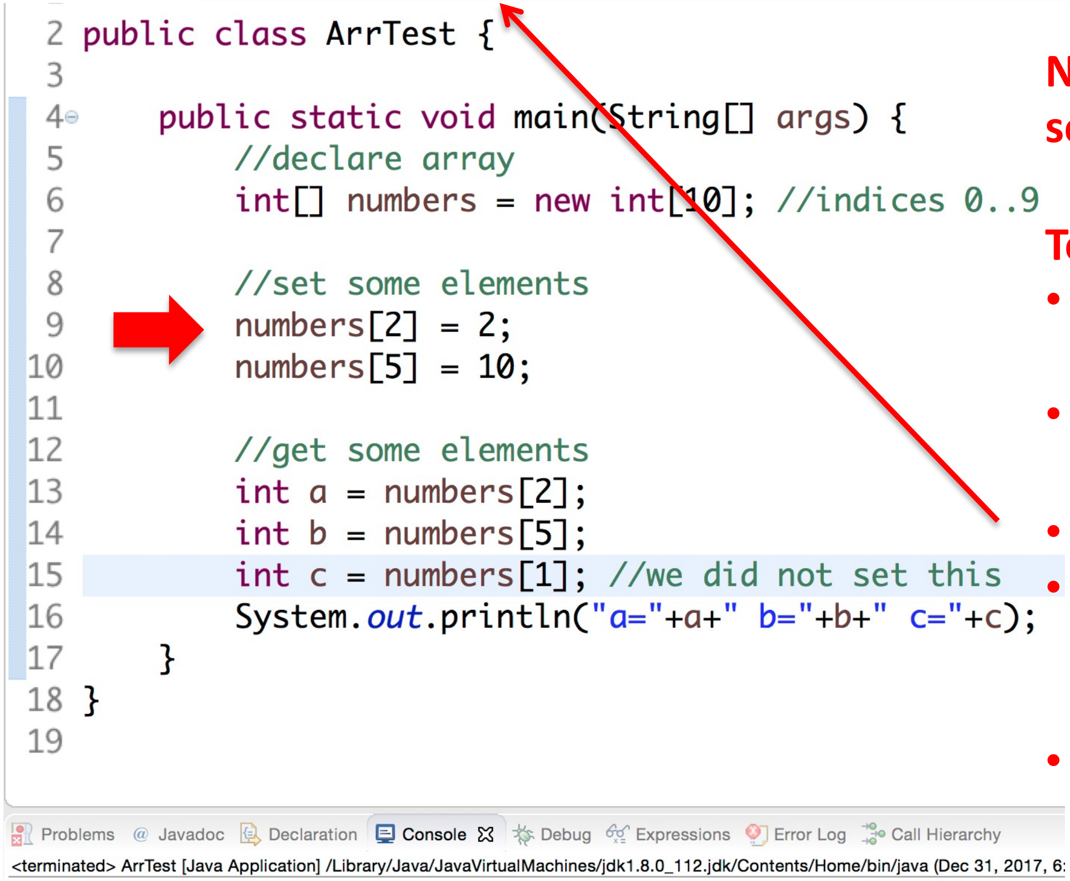
No need to march down list to get or set element

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)**

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);
    }
}

```

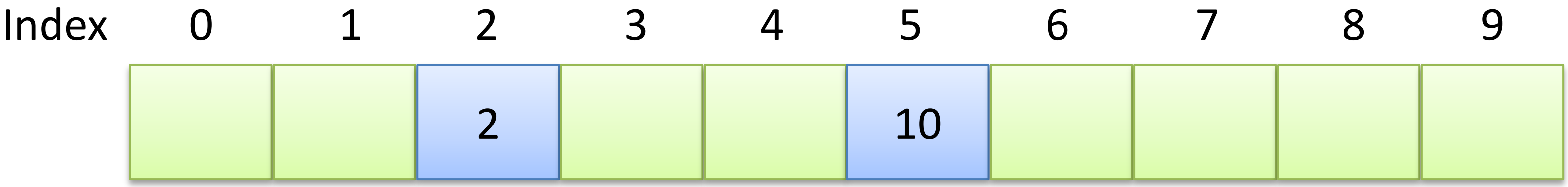
```
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:
```

No need to march down list to get or set element

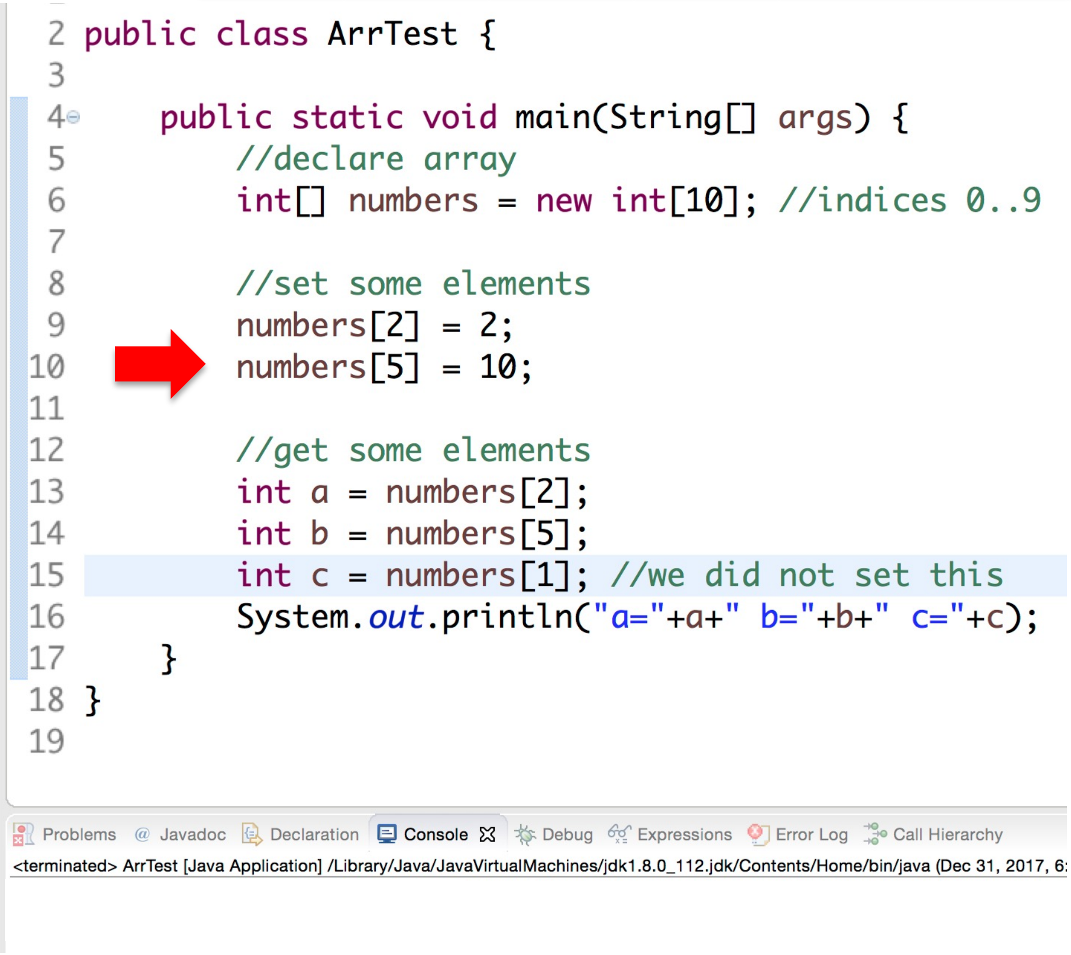
To find element:

- **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**
- **Time to access element is constant anywhere in array (just simple math operation to calculate any index)**
- **With linked list have to 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				

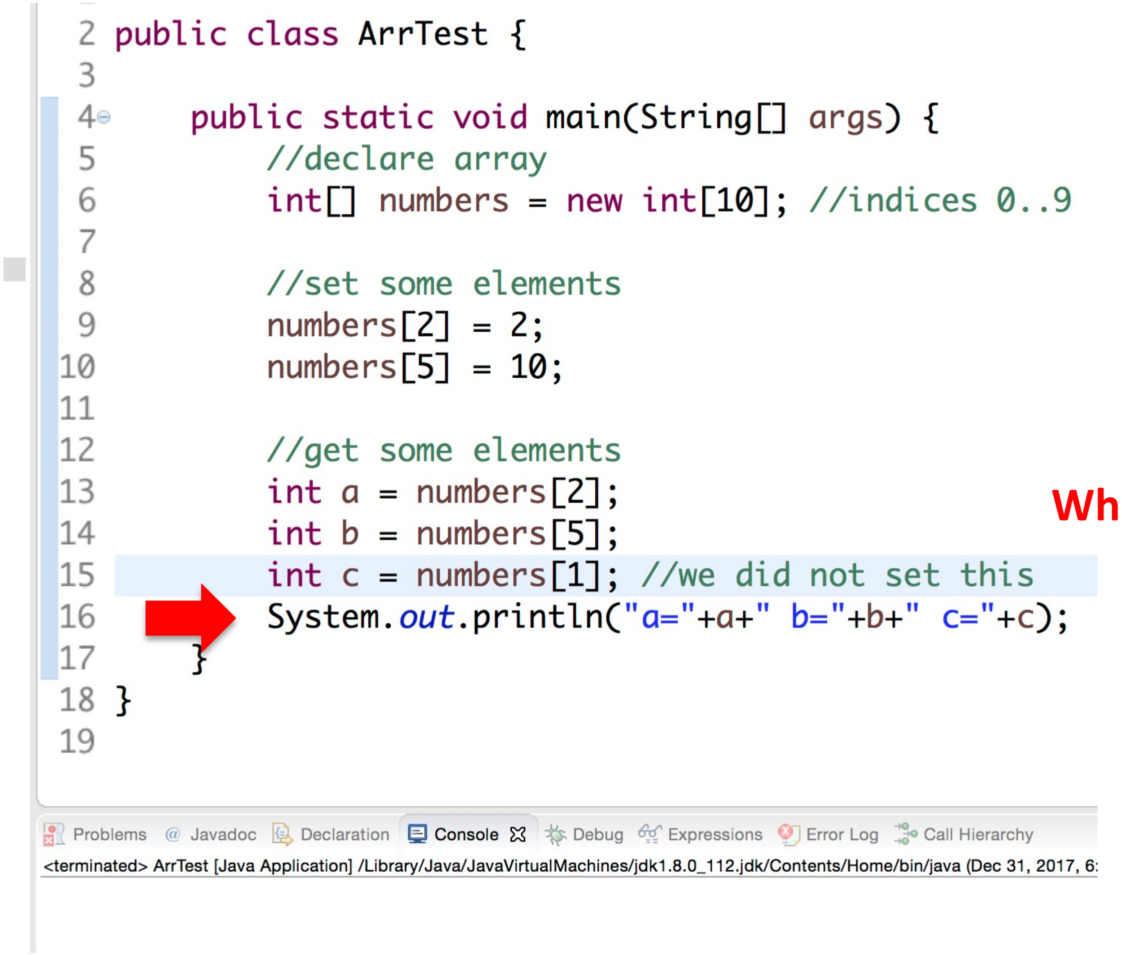


```
public class ArrTest {

    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);
    }
}

```

```
<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			10				



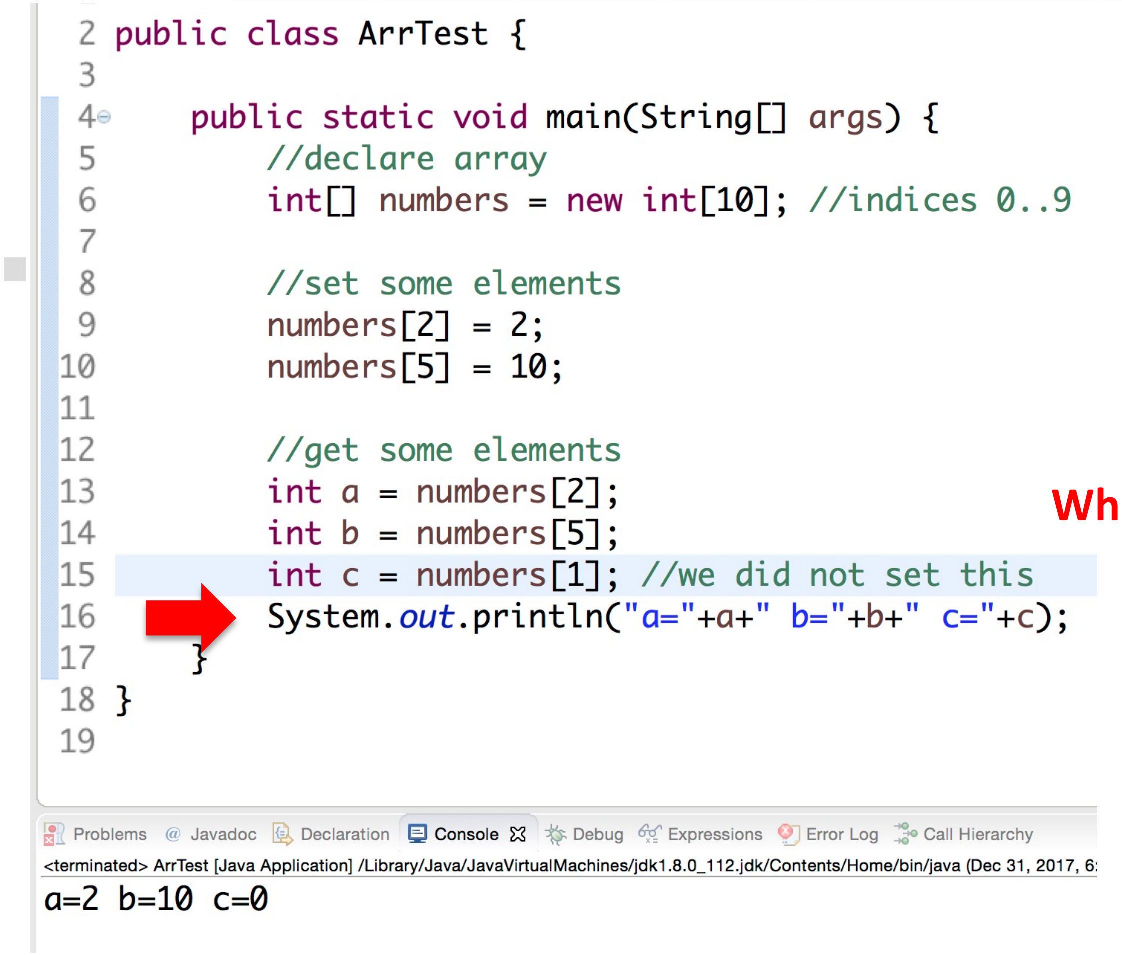
```java
public class ArrTest {

    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);
    }
}
```

```
<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				



```
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);
    }
}

```

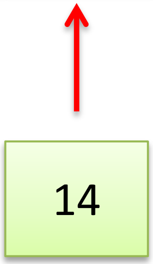
```
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:
a=2 b=10 c=0
```

What values will a, b and c have?

EXAMPLE OF INSERTION IN ARRAYLIST

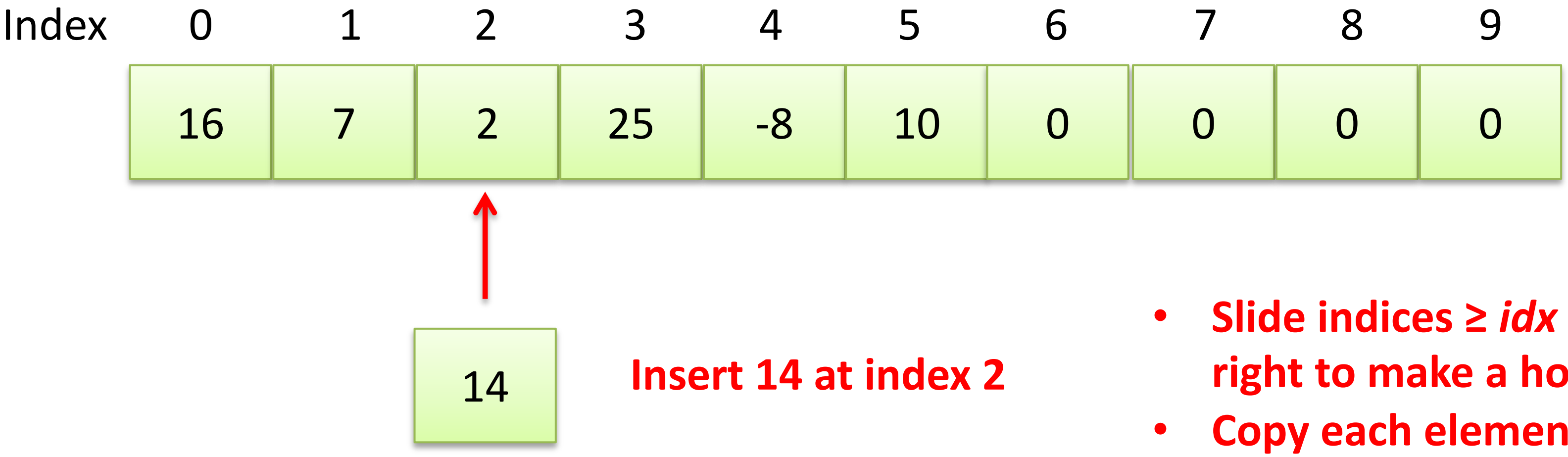
Because arrays are a contiguous block of memory, hard to insert (except at end)

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

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

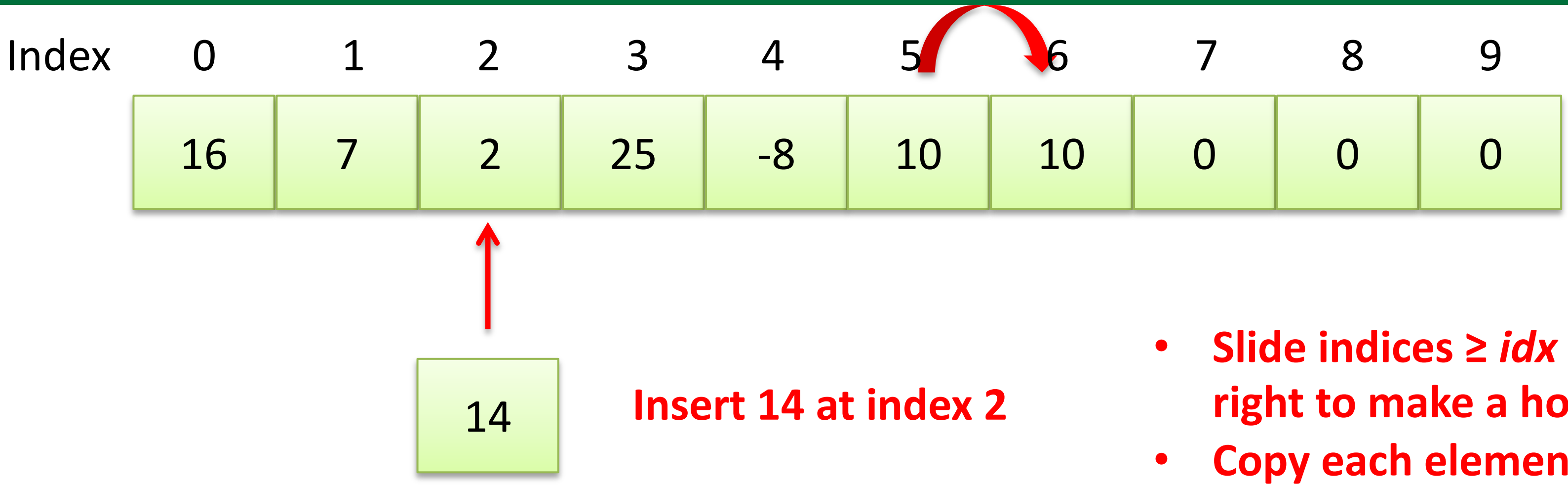


Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	0	0	0	0

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)



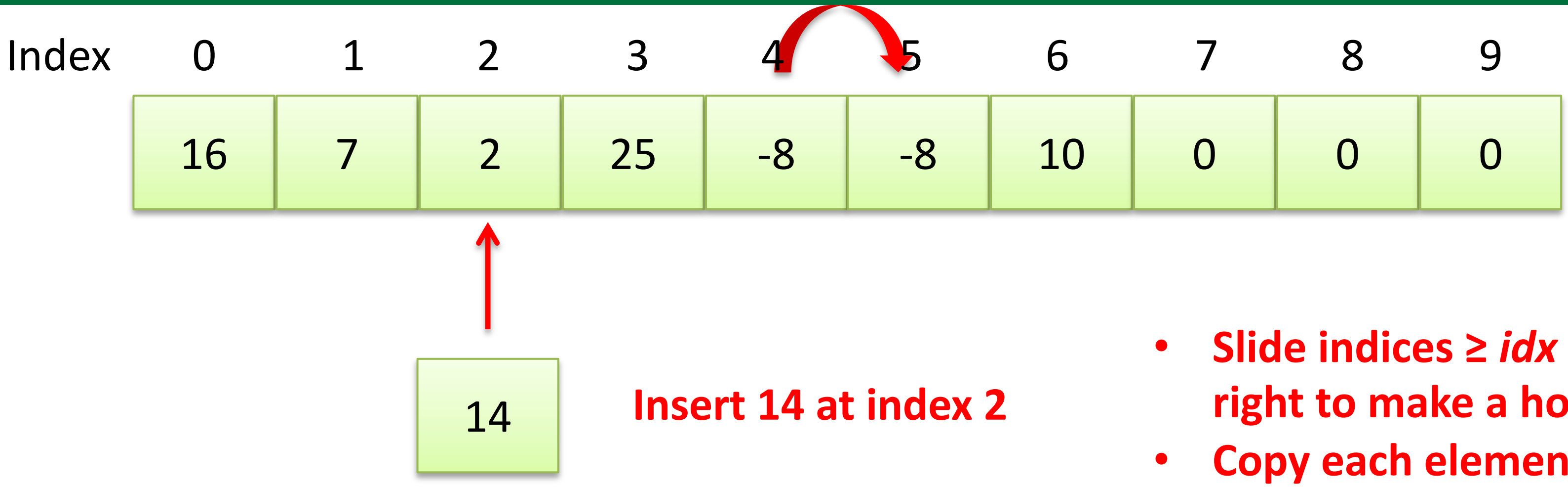
Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	10	0	0	0

14

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)

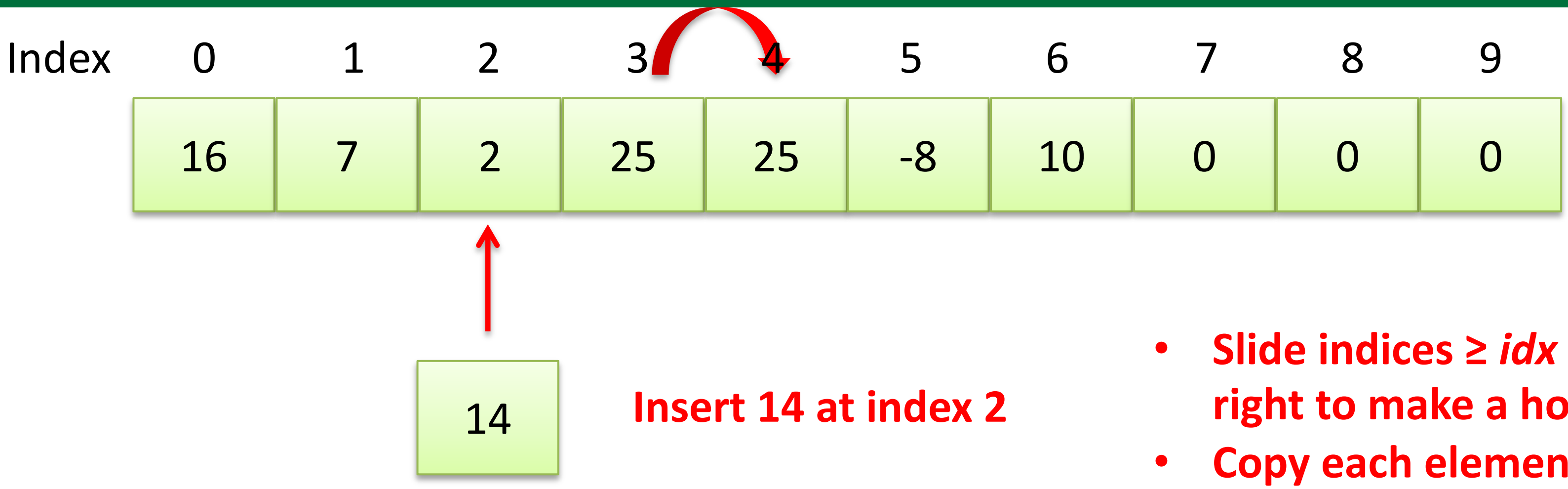


Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	-8	10	0	0	0

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)



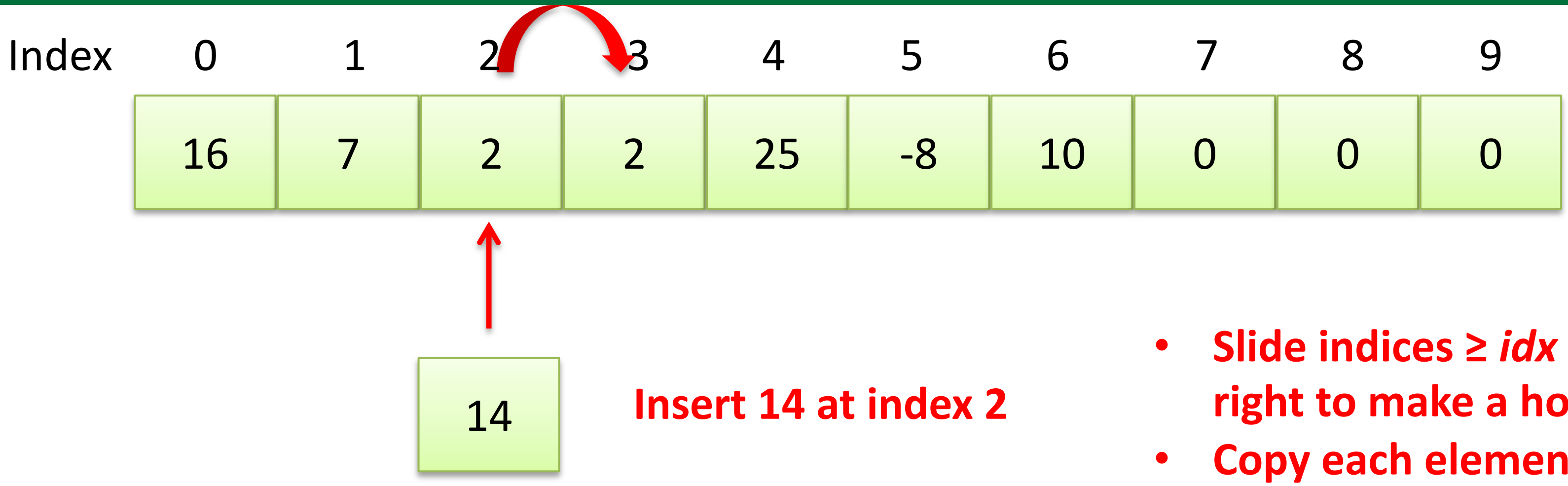
Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	25	-8	10	0	0	0

14

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)

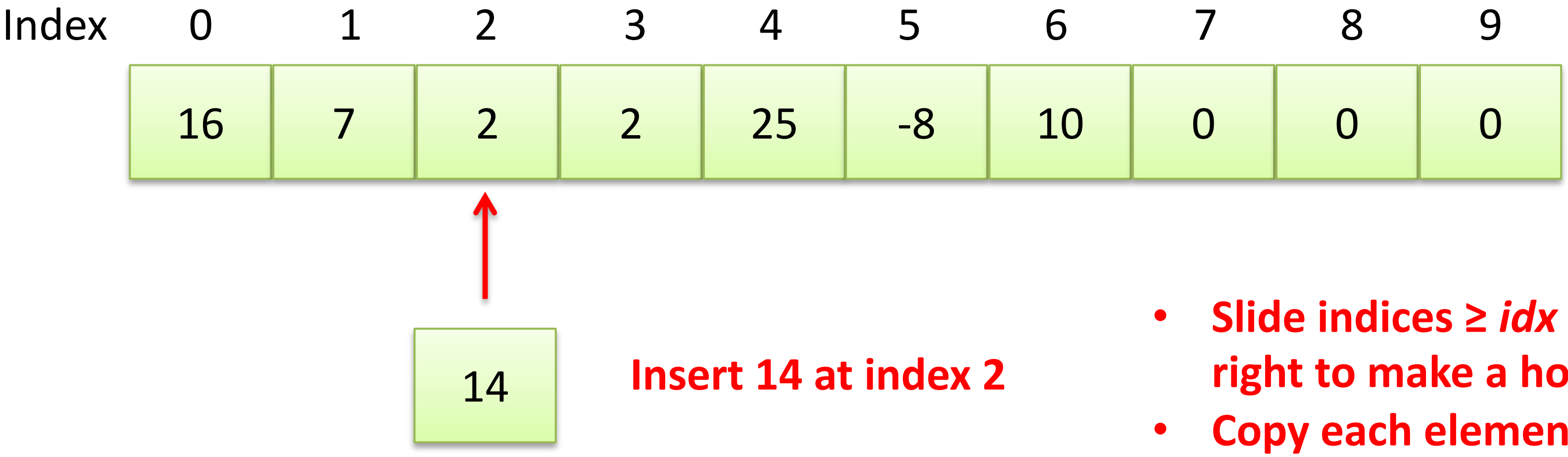


Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	2	25	-8	10	0	0	0

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)



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

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**
- **Linked list is slow to find the right place (have to 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**

EXAMPLE OF GROWING ARRAYLIST

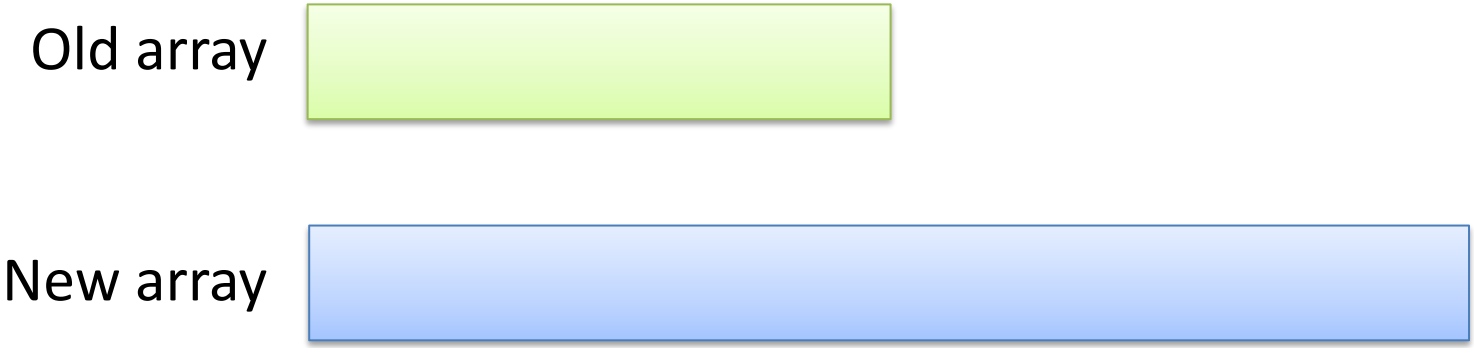
Arrays are of fixed size, but List ADT allows for growth

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



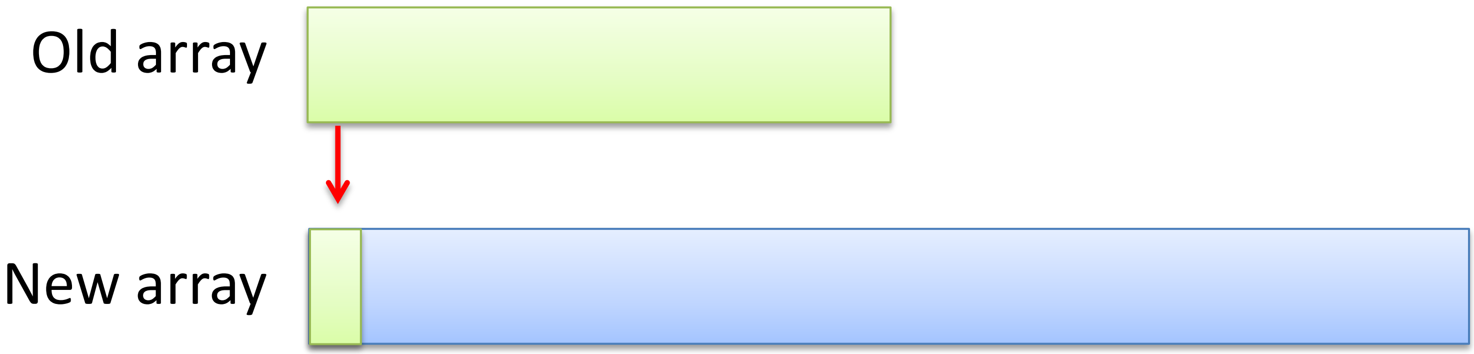
Old array

New array

Grow array

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

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



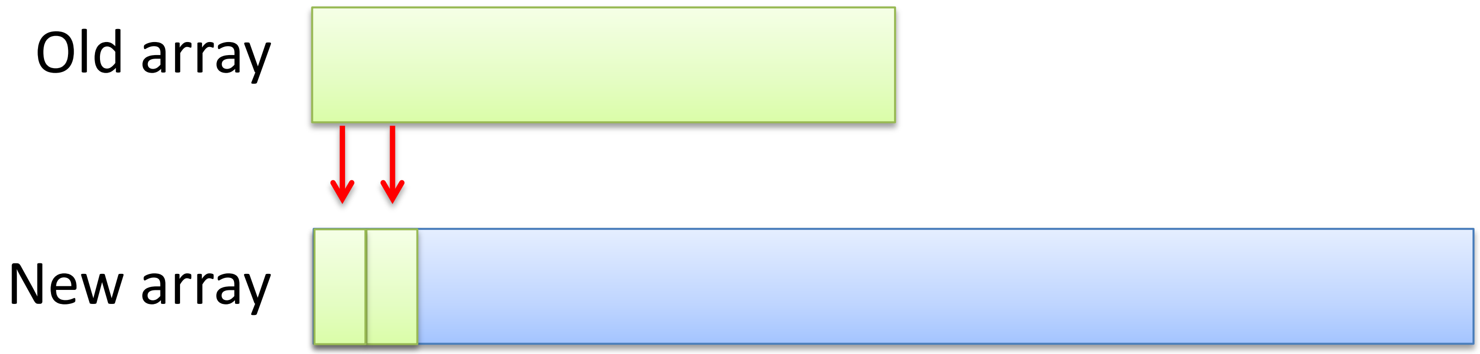
Old array

New array

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**

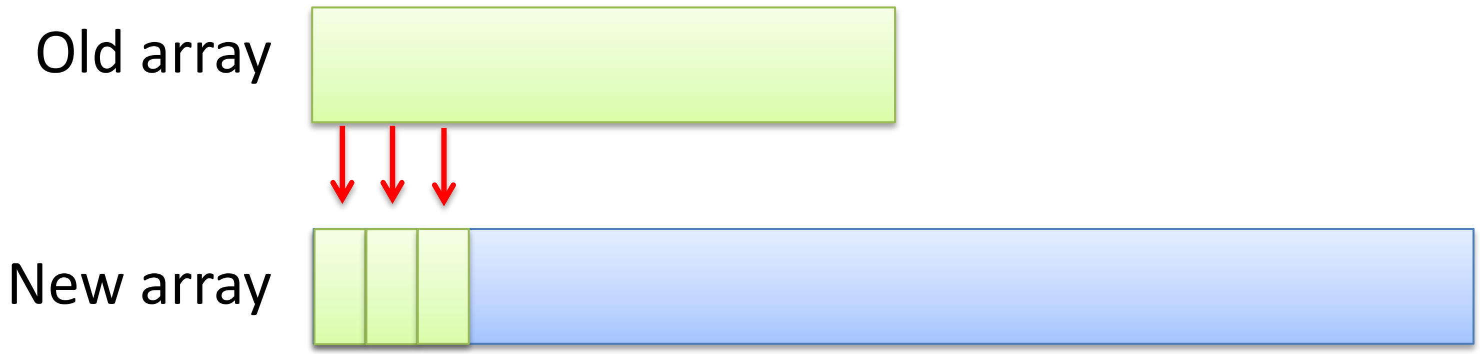
Arrays are of fixed size, but List ADT allows for growth



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**

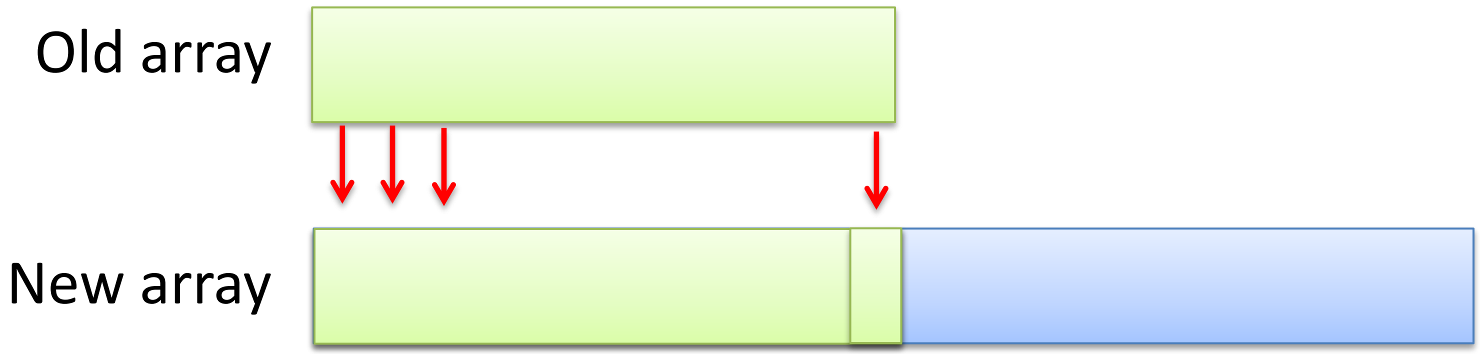
Arrays are of fixed size, but List ADT allows for growth



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**

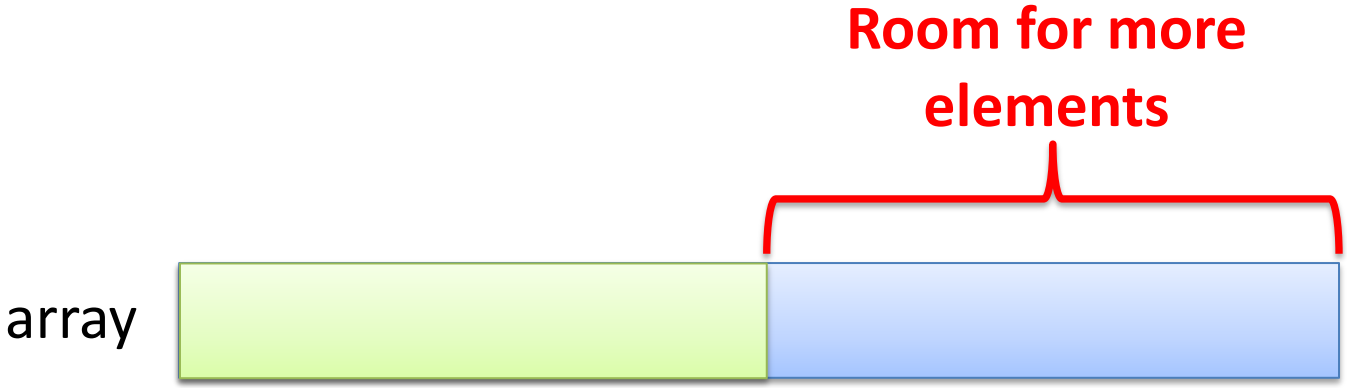
Arrays are of fixed size, but List ADT allows for growth



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

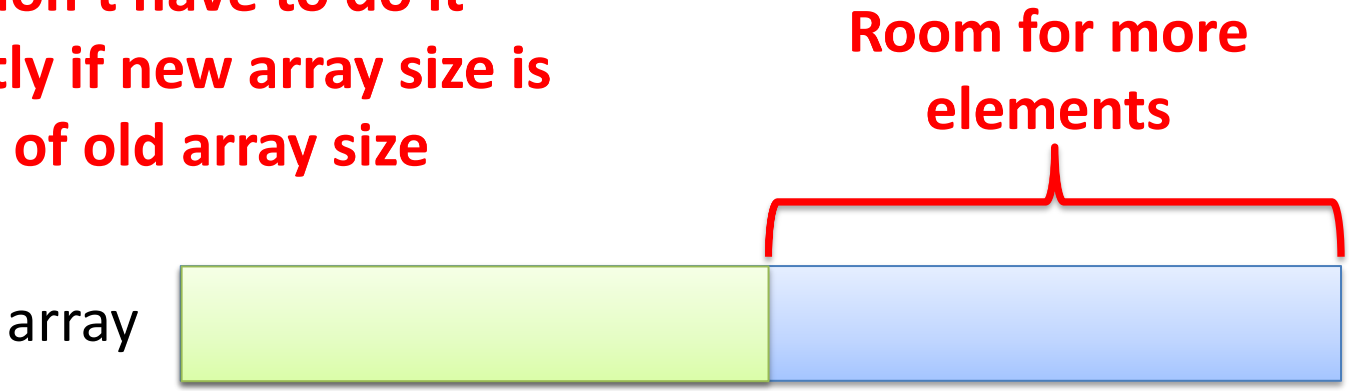


Grow array

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

ANNOTATED SLIDES

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 from last class

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?
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

```
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 can*** **hold**

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

Run-time complexity
O(1)

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

```java
/**
 * 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)

LIST ANALYSIS

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

Worst case run-time complexity

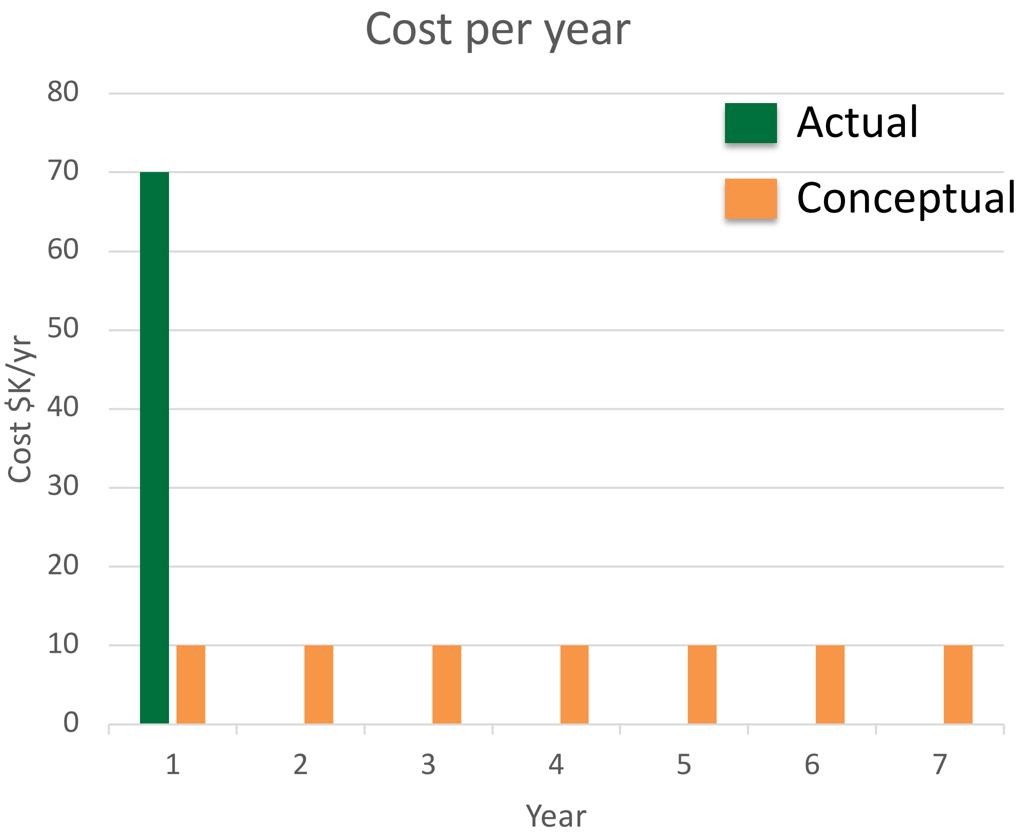
	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	O(n) + growth
remove(i)	O(n)	O(n)

- 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

- 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 be think about that?

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

Amortized analysis

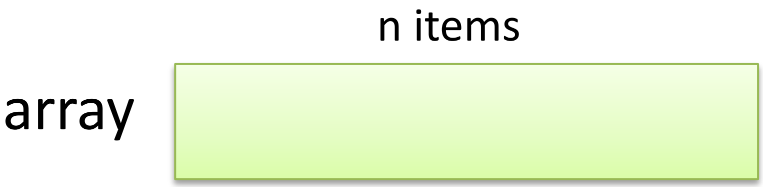


Accounting allows us to amortize costs over several years

- 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

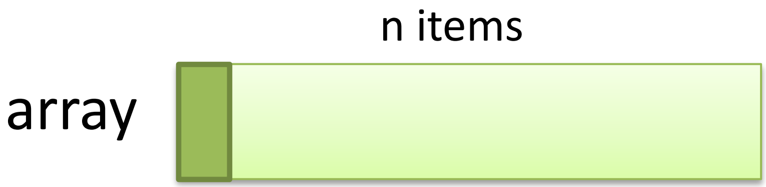


Each time add an item to array, *conceptually* charge 3 “tokens”

- **One token pays for current add()**
- **Two tokens go into “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

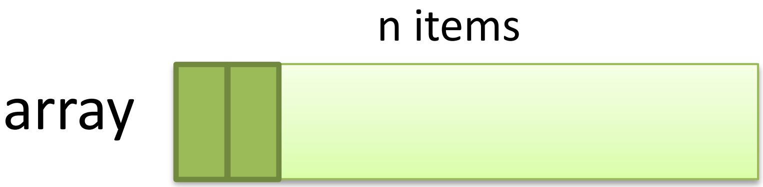


Each time add an item to array, *conceptually* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "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)!

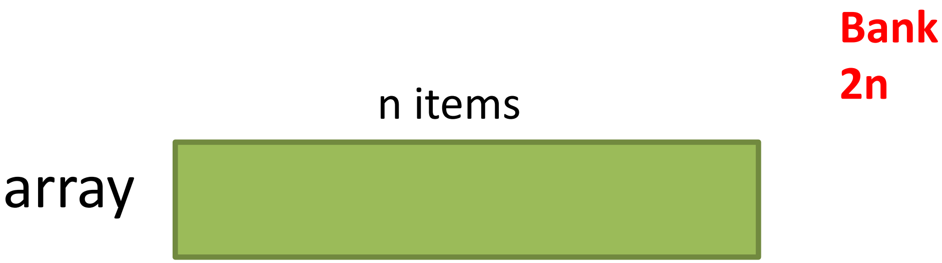
Bank
4



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

- One token pays for current add()
- Two tokens go into “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

n items

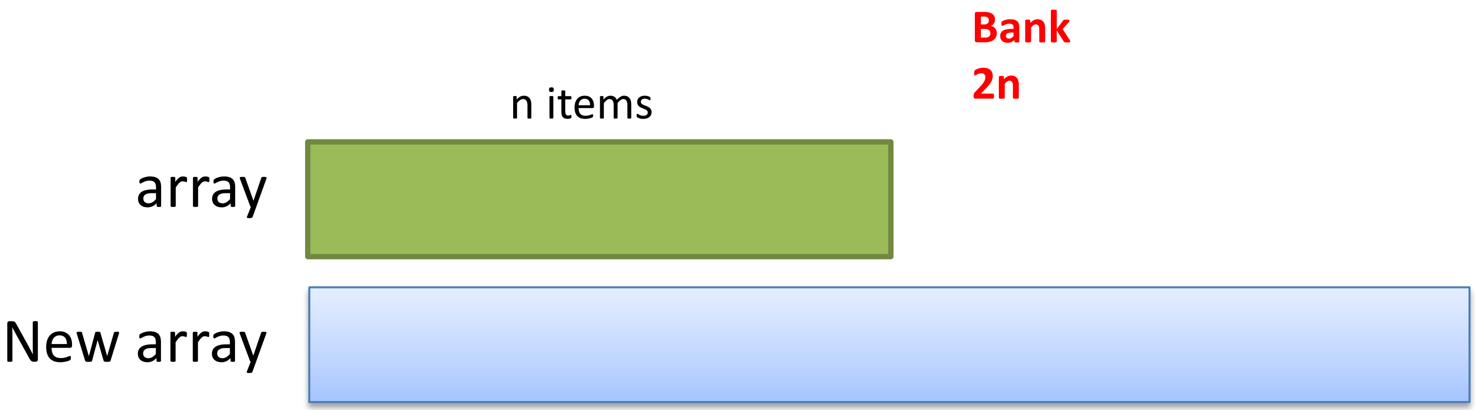
array

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

- **One token pays for current add()**
- **Two tokens go into "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)!



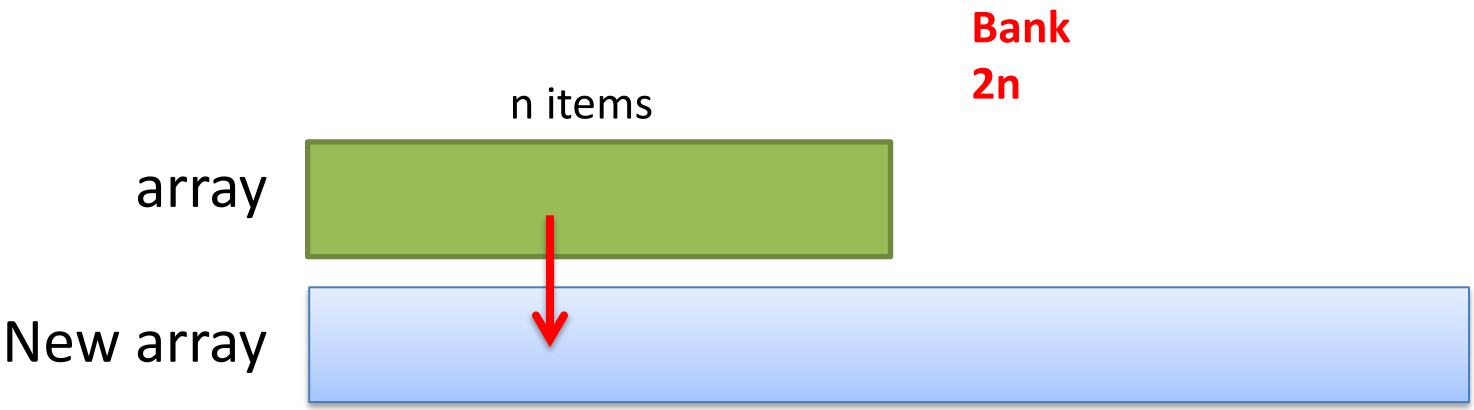
Each time add an item to array, *__conceptually__* charge 3 “tokens”

- **One token pays for current add()**
- **Two tokens go into “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, *<u>conceptually</u>* charge 3 "tokens"

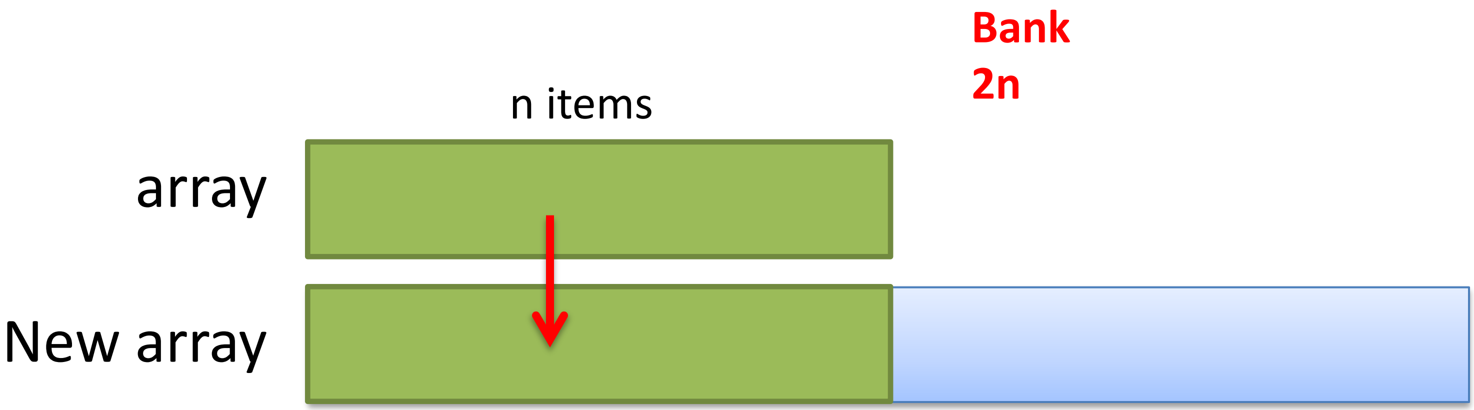
- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

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”

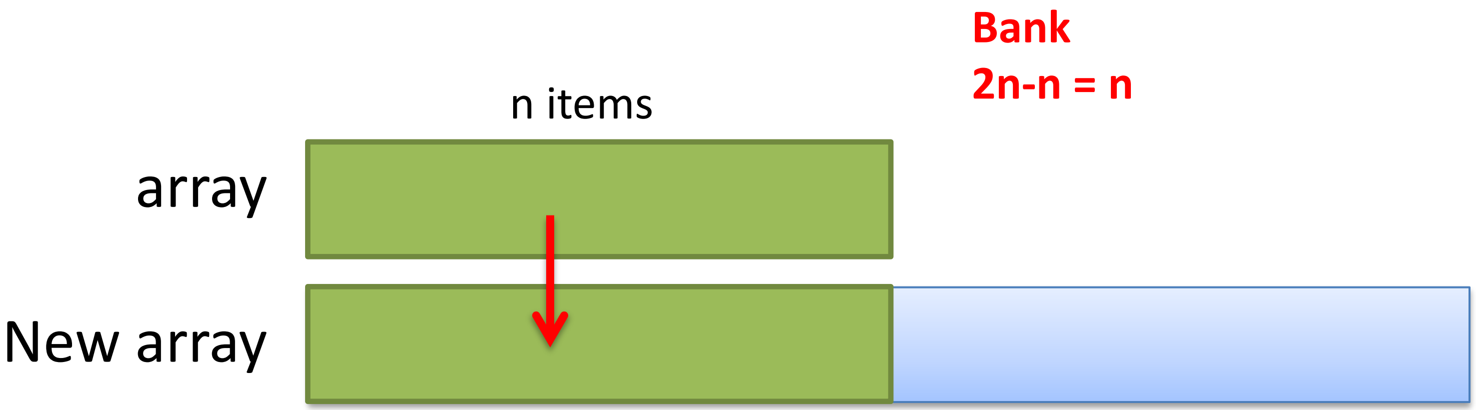
- One token pays for current add()
- Two tokens go into “Bank”
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

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 “Bank”**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

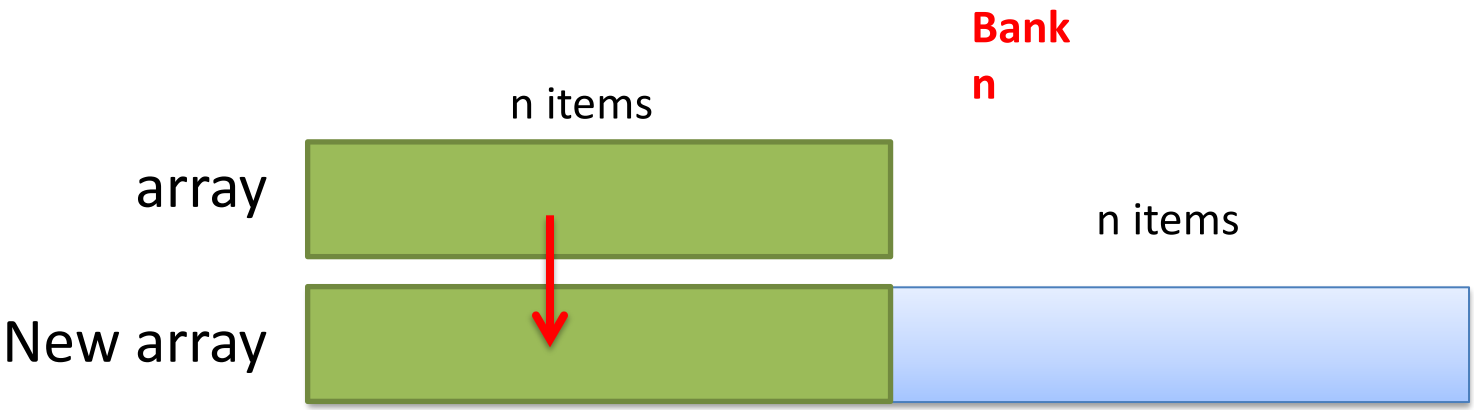
After *n add()* operations, array is full, but have *2n* tokens in bank

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)!



Bank

n

n items

array

n items

New array

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

- **One token pays for current add()**
- **Two tokens go into "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

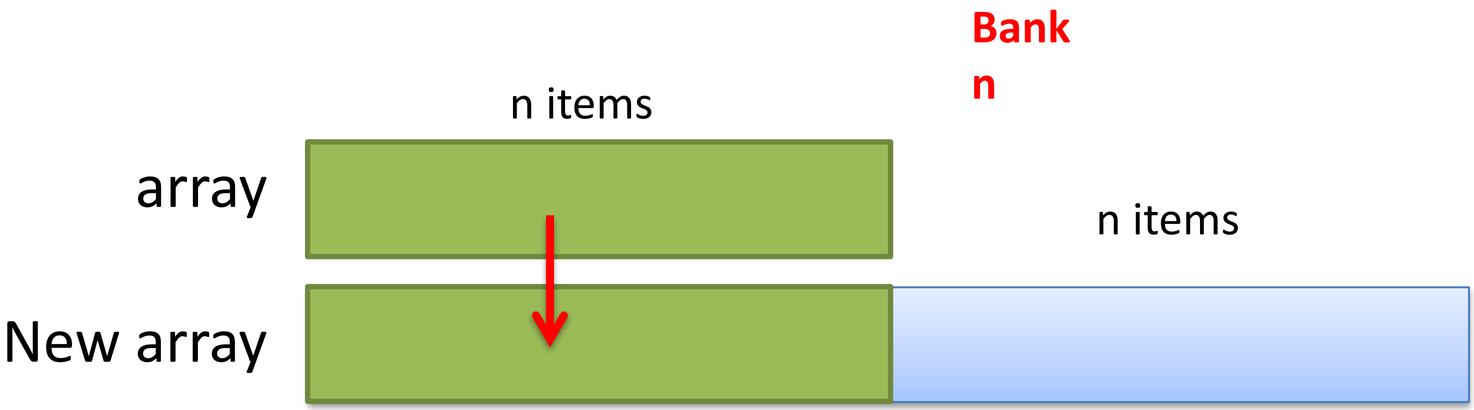
Allocate new 2X larger array

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 "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

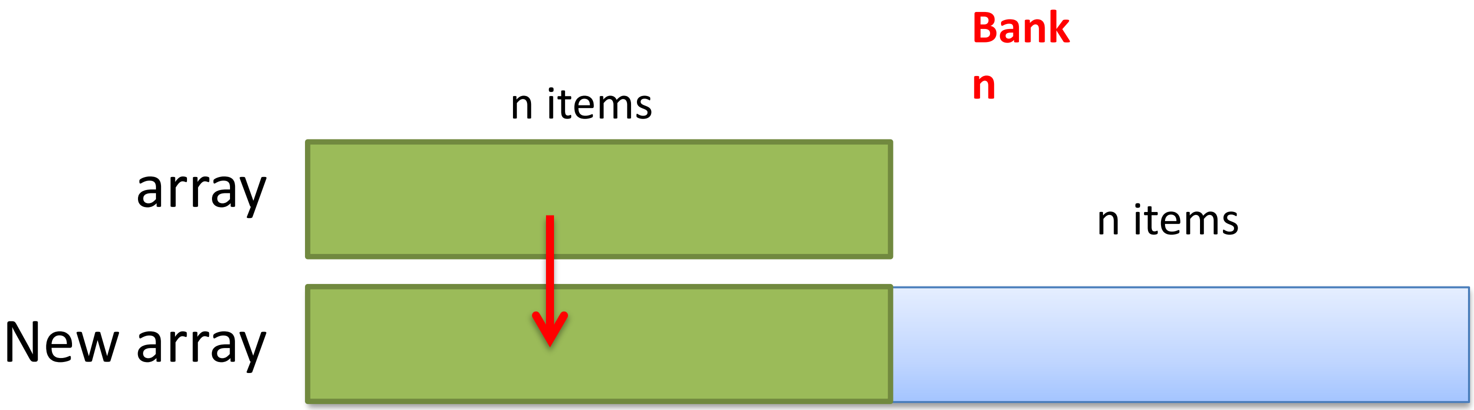
Copy elements from old array to new array

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)!



Bank

n

n items

array

n items

New array

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

- **One token pays for current add()**
- **Two tokens go into “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)

Growing array is *generally* preferable to linked list

Worst case run-time complexity

	Linked list	Growing array	
get(i)	O(n)	O(1)	**Amortized analysis shows infrequent growth operation is constant time**
set(i,e)	O(n)	O(1)	
add(i,e)	O(n)	**O(n) + O(1) = O(n)**	
remove(i)	O(n)	O(n)	**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

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