CS 10:
Problem solving via Object Oriented Programming

Lists Part 2 (Array's Revenge!)

Agenda

1. Growing array List implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

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 • 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	

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

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

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 • 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

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

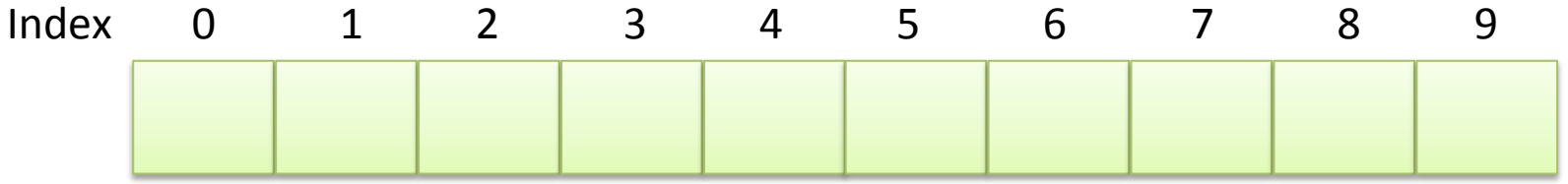
```
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = 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:

- **Array reserves a contiguous block of memory**
- **Big enough to hold specified number of elements (10 here)**
- **Indices are 0…9**

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

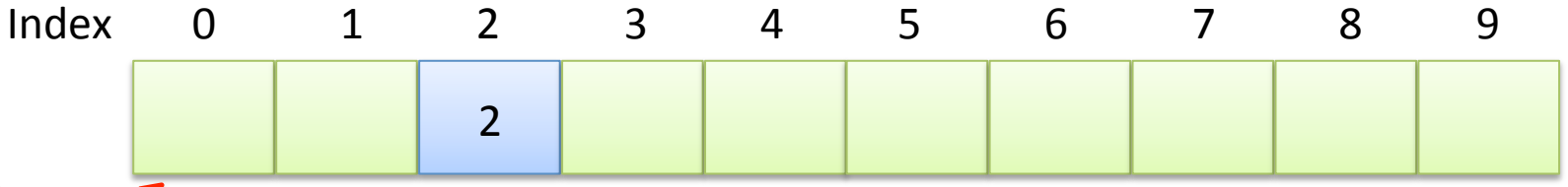



```
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = 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:

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




```
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

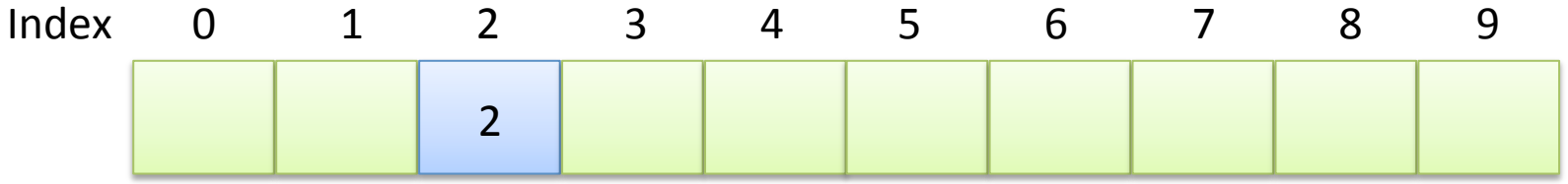
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

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




```
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

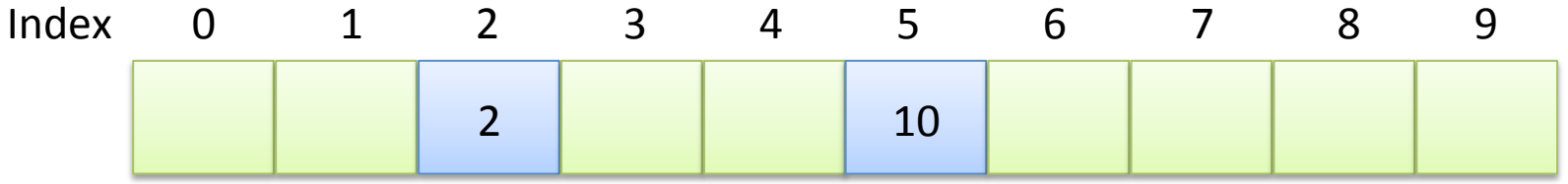
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

No need to march down list to get or set element

To find element:

- **Start at base address of array (this is where "*numbers*" 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

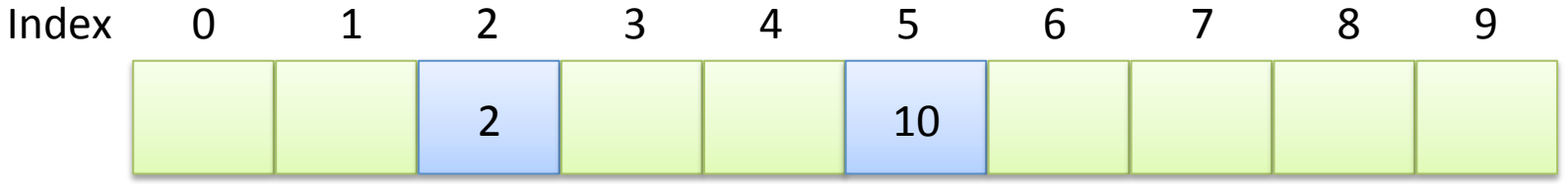



```
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = 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:

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



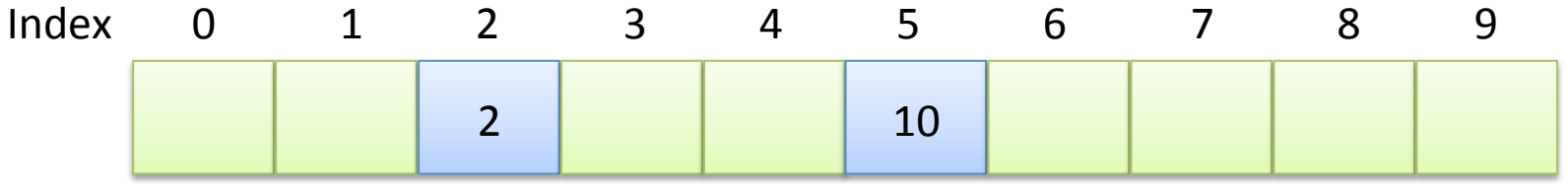

```
2  public class ArrTest {
3
4      public static void main(String[] args) {
5          //declare array
6          int[] numbers = new int[10]; //indices 0..9
7
8          //set some elements
9          numbers[2] = 2;
10         numbers[5] = 10;
11
12         //get some elements
13         int a = 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
```

What values will a, b and c have?

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




```
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
```

What values will a, b and c have?

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

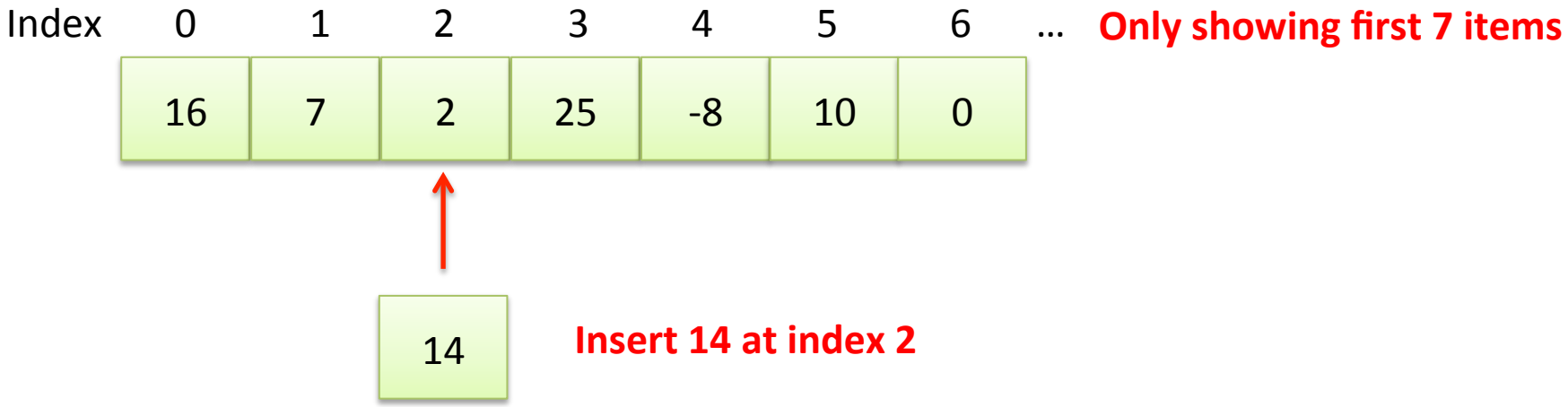
<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

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

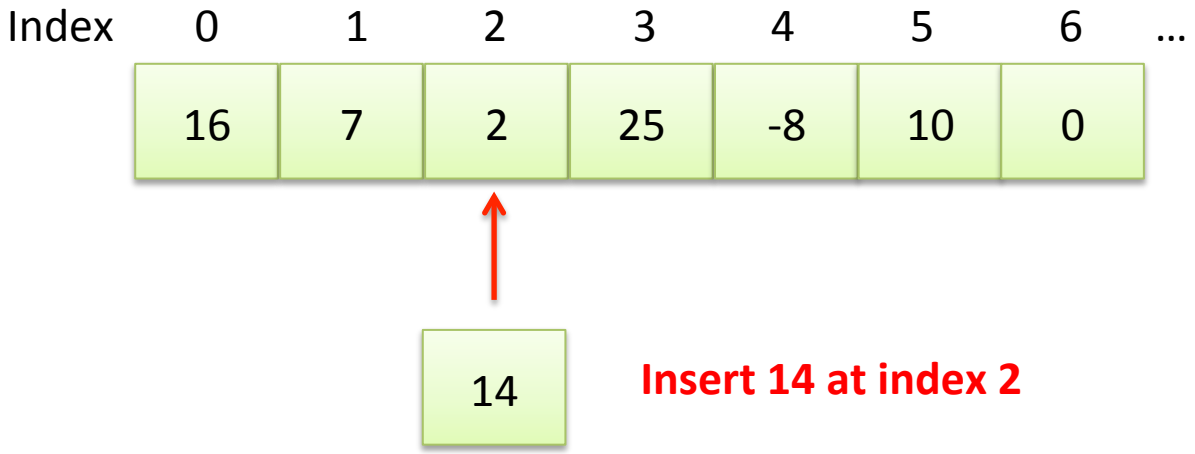
At first arrays seem to be a poor choice to implement the List ADT

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

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

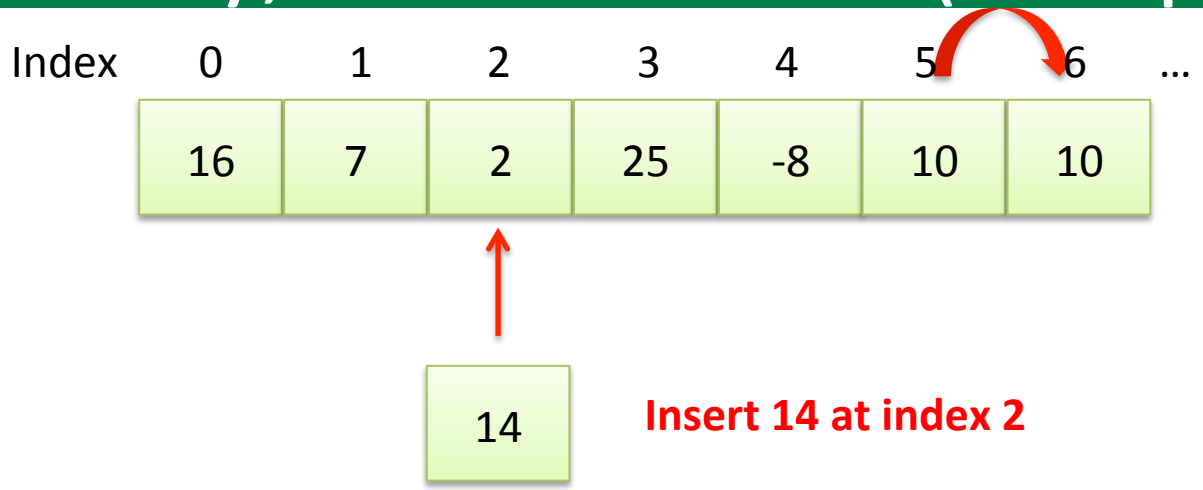


Because arrays are a contiguous block of memory, hard to insert (except at end)



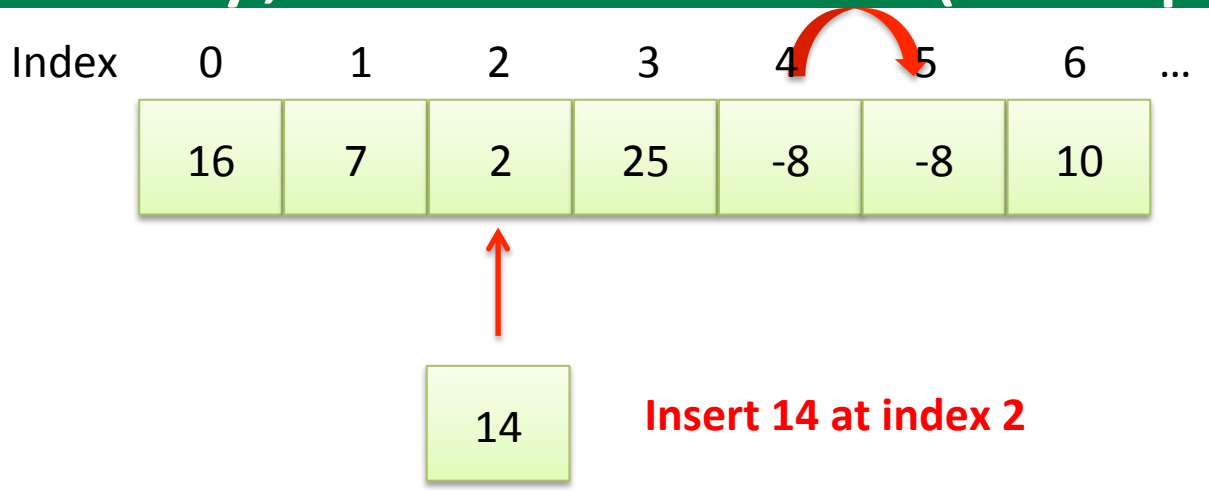
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



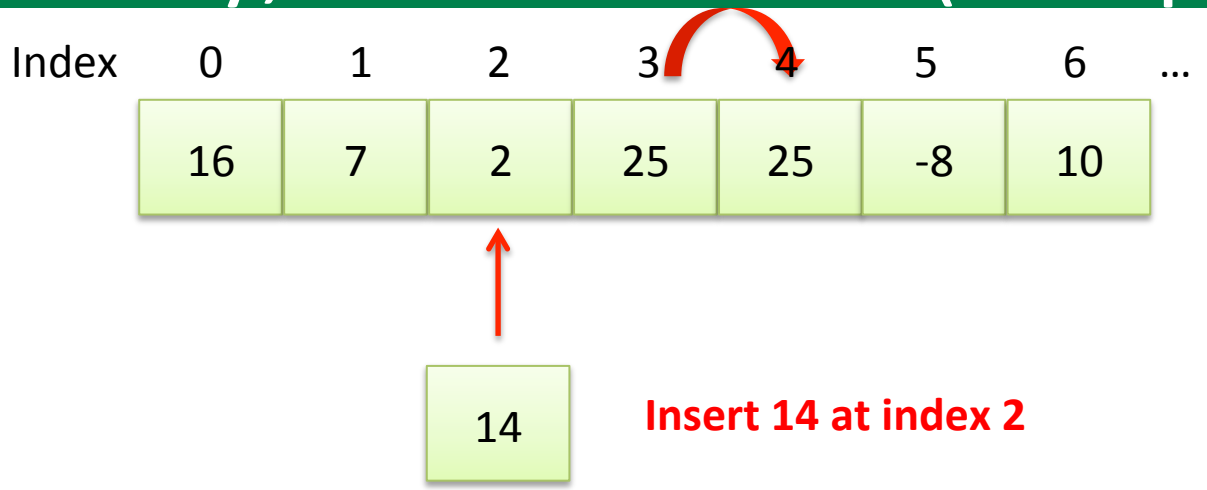
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



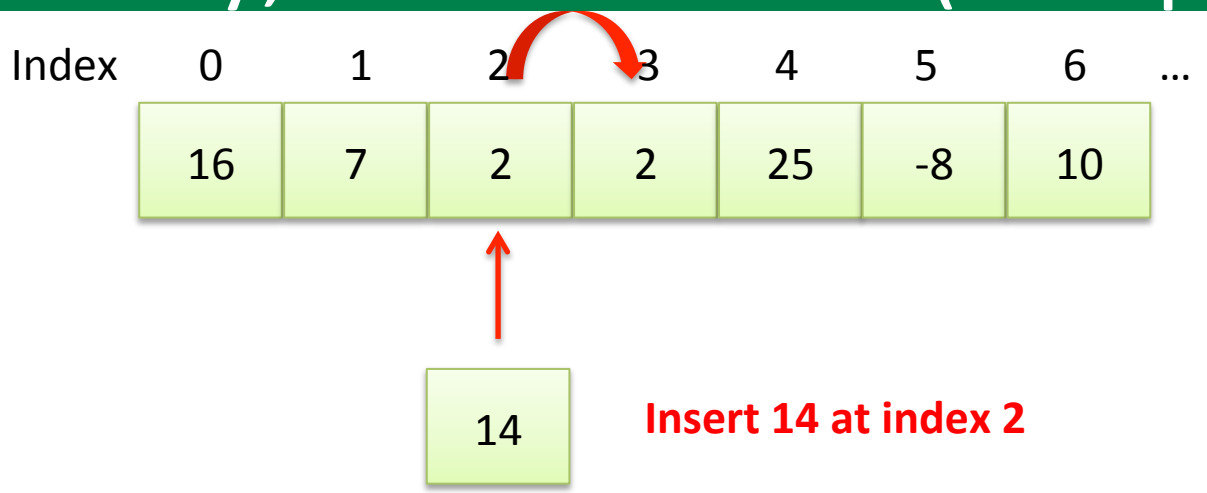
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



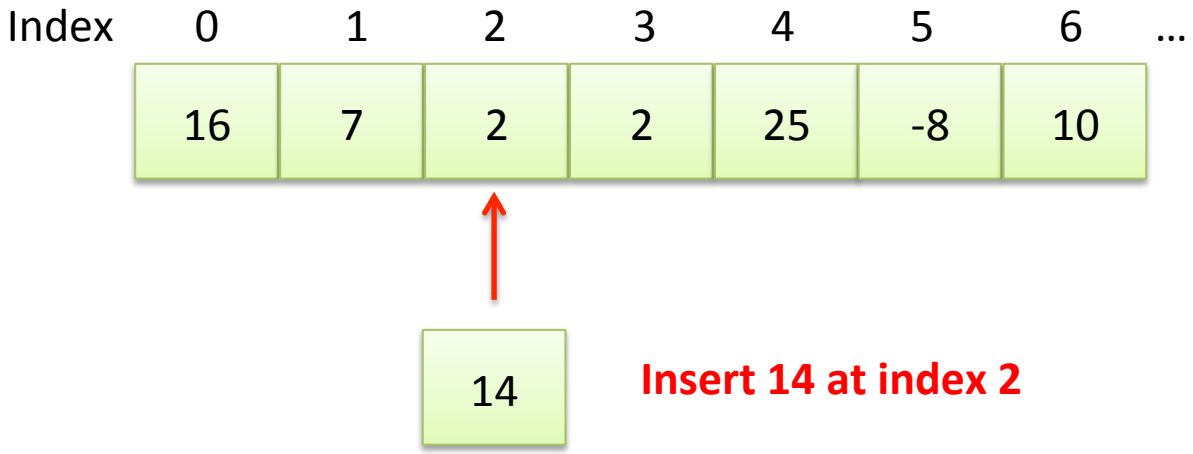
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



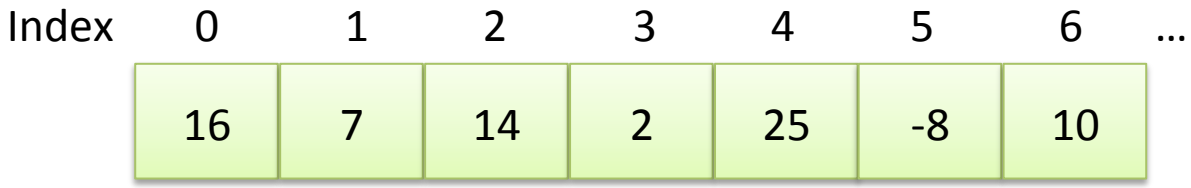
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



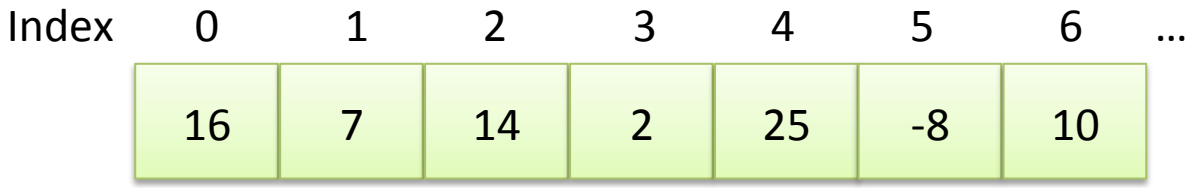
- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

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



- **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**
- **With arrays, easy to find right place, but slow afterward due to copying**

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

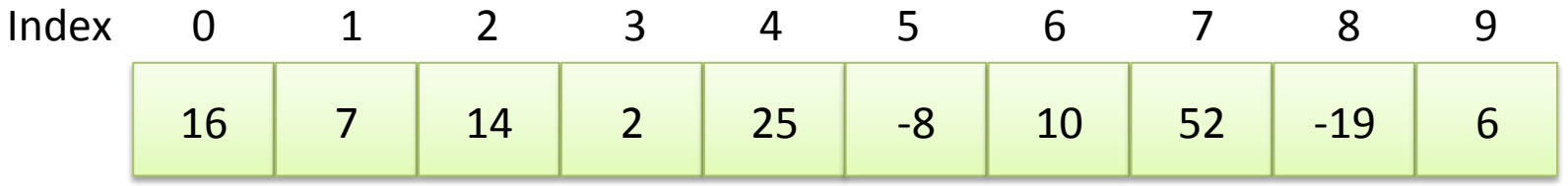


Deleting an element is the same except copy elements to the left to remove the deleted element

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 • 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 

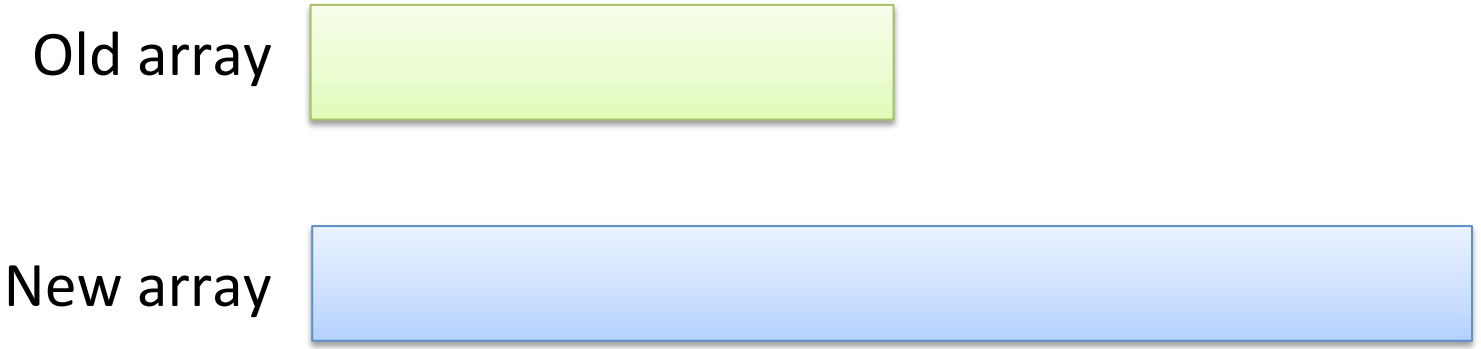
Arrays are of fixed size, but List ADT allows for growth



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

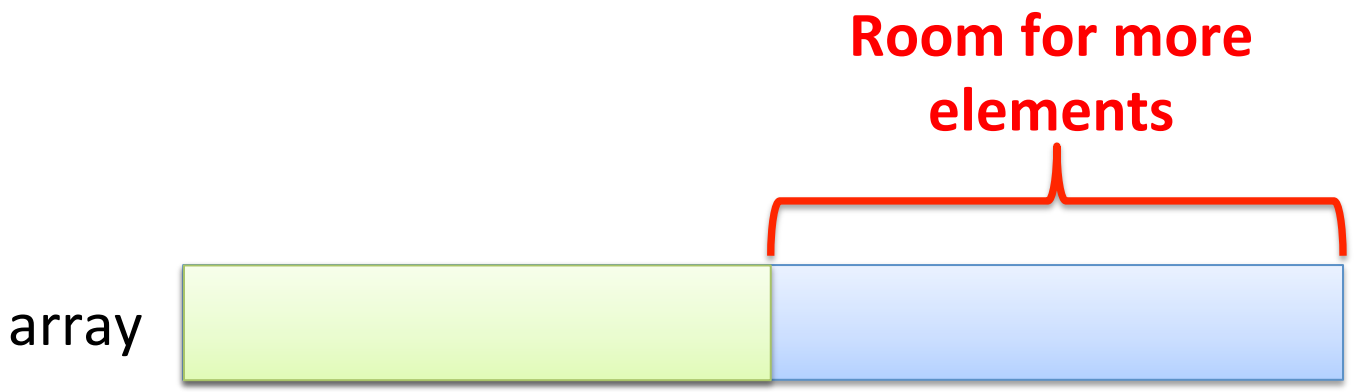
Arrays are of fixed size, but List ADT allows for growth

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

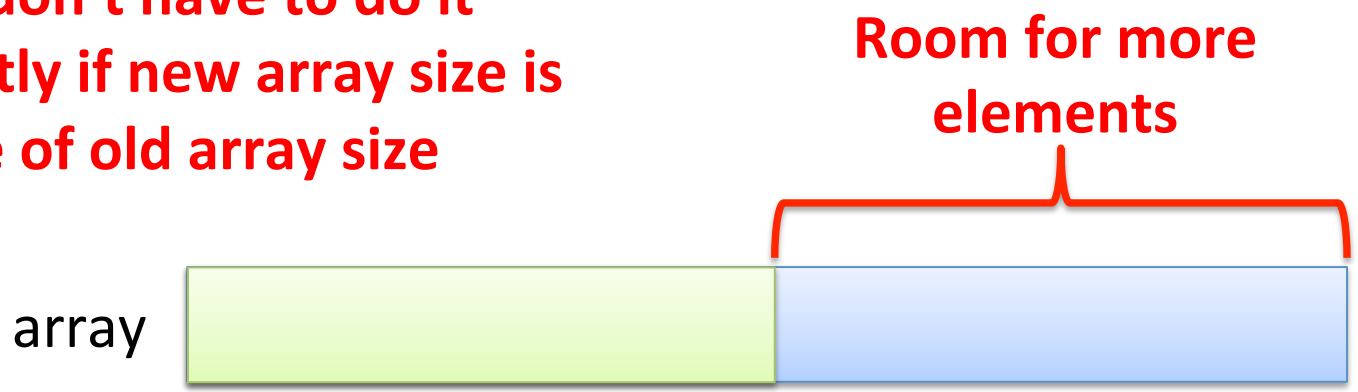


Grow array

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

```
 7 public class GrowingArray<T> implements SimpleList<T> {
 8     private T[] array;
 9     private int size;       // how much of the array is actually filled up so far
10     private static final int initCap = 10; // how big the array should be initially
11
12     public GrowingArray() {
13         array = (T[]) new Object[initCap];  // java generics oddness -- create an array of
14         size = 0;
15     }
16
17     public int size() {
18         return size;
19     }
20
21     public void add(int idx, T item) throws Exception {
22         if (idx > size || idx < 0) throw new Exception("invalid index");
23         if (size == array.length) {
24             // Double the size of the array, to leave more space
25             T[] copy = (T[]) new Object[size*2];
26             // Copy it over
27             for (int i=0; i<size; i++) copy[i] = array[i];
28             array = copy;
29         }
30         // Shift right to make room
31         for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32         array[idx] = item;
33         size++;
34     }
35
36     public void remove(int idx) throws Exception {
37         if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38         // Shift left to cover it over
39         for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40         size--;
41     }
42
43     public T get(int idx) throws Exception {
44         if (idx >= 0 && idx < size) return array[idx];
45         else throw new Exception("invalid index");
46     }
47
48     public void set(int idx, T item) throws Exception {
49         if (idx >= 0 && idx < size) array[idx] = item;
50         else throw new Exception("invalid index");
51     }
52
53     public String toString() {
54         String result = "[";
55         for (int i=0; i<size; i++) {
56             if (i>0) result += ",";
57             result += array[i];
58         }
59         result += "]";
60
61         return result;
```

Implements SimpleList from last class

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

Track size

- **Array initially sized to 10 Objects (note the funky Java allocation in line 13)**
- **Remember, arrays are of fixed size, but the List ADT does not specify a size**

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

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

```
43      public T get(int idx) throws Exception {
44          if (idx >= 0 && idx < size) return array[idx];
45          else throw new Exception("invalid index");
46      }
47
48      public void set(int idx, T item) throws Exception {
49          if (idx >= 0 && idx < size) array[idx] = item;
50          else throw new Exception("invalid index");
51      }
52
```

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

```
21    public void add(int idx, T item) throws Exception {
22        if (idx > size || idx < 0) throw new Exception("invalid index");
23        if (size == array.length) {
24            // Double the size of the array, to leave more space
25            T[] copy = (T[]) new Object[size*2];
26            // Copy it over
27            for (int i=0; i<size; i++) copy[i] = array[i];
28            array = copy;
29        }
30        // Shift right to make room
31        for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32        array[idx] = item;
33        size++;
34    }
35
36    public void remove(int idx) throws Exception {
37        if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38        // Shift left to cover it over
39        for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40        size--;
41    }
```

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

```
21    public void add(int idx, T item) throws Exception {
22        if (idx > size || idx < 0) throw new Exception("invalid index");
23        if (size == array.length) {
24            // Double the size of the array, to leave more space
25            T[] copy = (T[]) new Object[size*2];
26            // Copy it over
27            for (int i=0; i<size; i++) copy[i] = array[i];
28            array = copy;
29        }
30        // Shift right to make room
31        for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32        array[idx] = item;
33        size++;
34    }
35
36    public void remove(int idx) throws Exception {
37        if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38        // Shift left to cover it over
39        for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40        size--;
41    }
```

***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

```
21    public void add(int idx, T item) throws Exception {
22        if (idx > size || idx < 0) throw new Exception("invalid index");
23        if (size == array.length) {
24            // Double the size of the array, to leave more space
25            T[] copy = (T[]) new Object[size*2];
26            // Copy it over
27            for (int i=0; i<size; i++) copy[i] = array[i];
28            array = copy;
29        }
30        // Shift right to make room
31        for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32        array[idx] = item;
33        size++;
34    }
35
36    public void remove(int idx) throws Exception {
37        if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38        // Shift left to cover it over
39        for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40        size--;
41    }
```

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

Copy elements one at a time into new array

Update instance variable to new array

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

```
21    public void add(int idx, T item) throws Exception {
22        if (idx > size || idx < 0) throw new Exception("invalid index");
23        if (size == array.length) {
24            // Double the size of the array, to leave more space
25            T[] copy = (T[]) new Object[size*2];
26            // Copy it over
27            for (int i=0; i<size; i++) copy[i] = array[i];
28            array = copy;
29        }
30        // Shift right to make room
31        for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32        array[idx] = item;
33        size++;
34    }
35
36    public void remove(int idx) throws Exception {
37        if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38        // Shift left to cover it over
39        for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40        size--;
41    }
```

- 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

```
21    public void add(int idx, T item) throws Exception {
22        if (idx > size || idx < 0) throw new Exception("invalid index");
23        if (size == array.length) {
24            // Double the size of the array, to leave more space
25            T[] copy = (T[]) new Object[size*2];
26            // Copy it over
27            for (int i=0; i<size; i++) copy[i] = array[i];
28            array = copy;
29        }
30        // Shift right to make room
31        for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
32        array[idx] = item;
33        size++;
34    }
35
36    public void remove(int idx) throws Exception {
37        if (idx > size-1 || idx < 0) throw new Exception("invalid index");
38        // Shift left to cover it over
39        for (int i=idx; i<size-1; i++) array[i] = array[i+1];
40        size--;
41    }
```

***remove()* slides elements with index > idx left**

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 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

Can get around array growth limit

Want to make sure growth is fast enough

Agenda

1. Growing array List implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Often run-time will depend on the number of elements an algorithm must process

Constant time – does not depend on number of items

- Returning the first element of a linked list takes a constant amount of time *irrespective* of the number of elements in the list
- Just return the *head* pointer
- No need to march down list to find the first element (*head*)
- Array *get()* implementation is also constant time (array *get()* is constant time everywhere, linked list only constant at *head*)

Linear time – directly depends on number of items

- Example: searching for a particular value stored in a list
- Start at first item, compare value with value trying to find
- Keep going until find item, or end up at end of list
- Could get lucky and find item right away, might not find it at all
- Worst case is we check all *n* items

Often run-time will depend on the number of elements an algorithm must process

Polynomial time – depends on a function of number of items

- Example: nested loop in image and graphic methods
- If changing all pixels in *n* by *n* image, must do a total of n^2 operations because inner and outer loops each run *n* times
- Runs slower than a constant or linear time algorithm

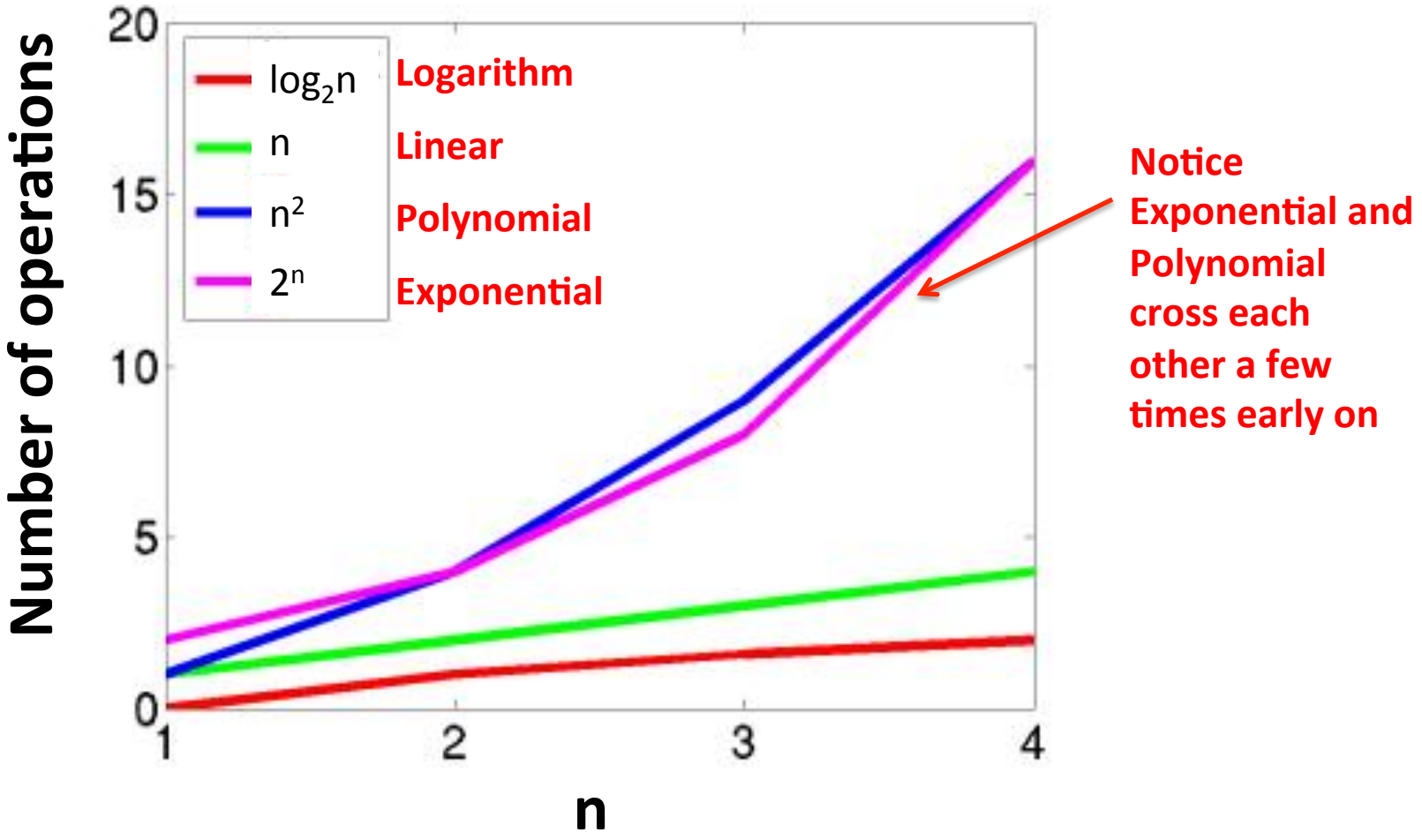
Logarithm time – avoids operations on some items

- Next class we will look at binary search
- Reduces the number of items algorithm must process (don't process all *n* items)
- Runs faster than linear or polynomial time (slower than constant)

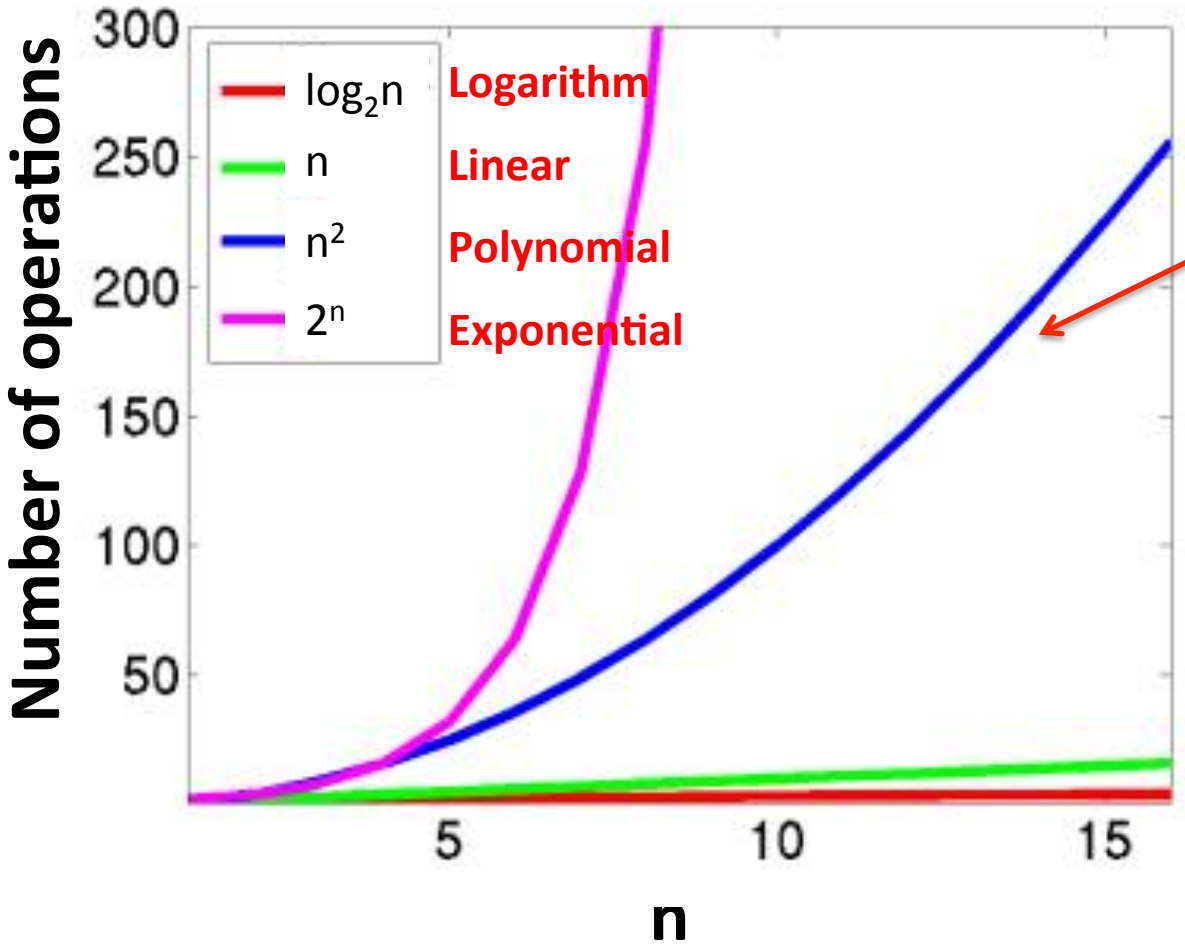
Exponential time – base raised to power

- Combination problems: all possible bit combinations in *n* bits = 2^n
- SLOW!

For small numbers of items, run time does not differ by much



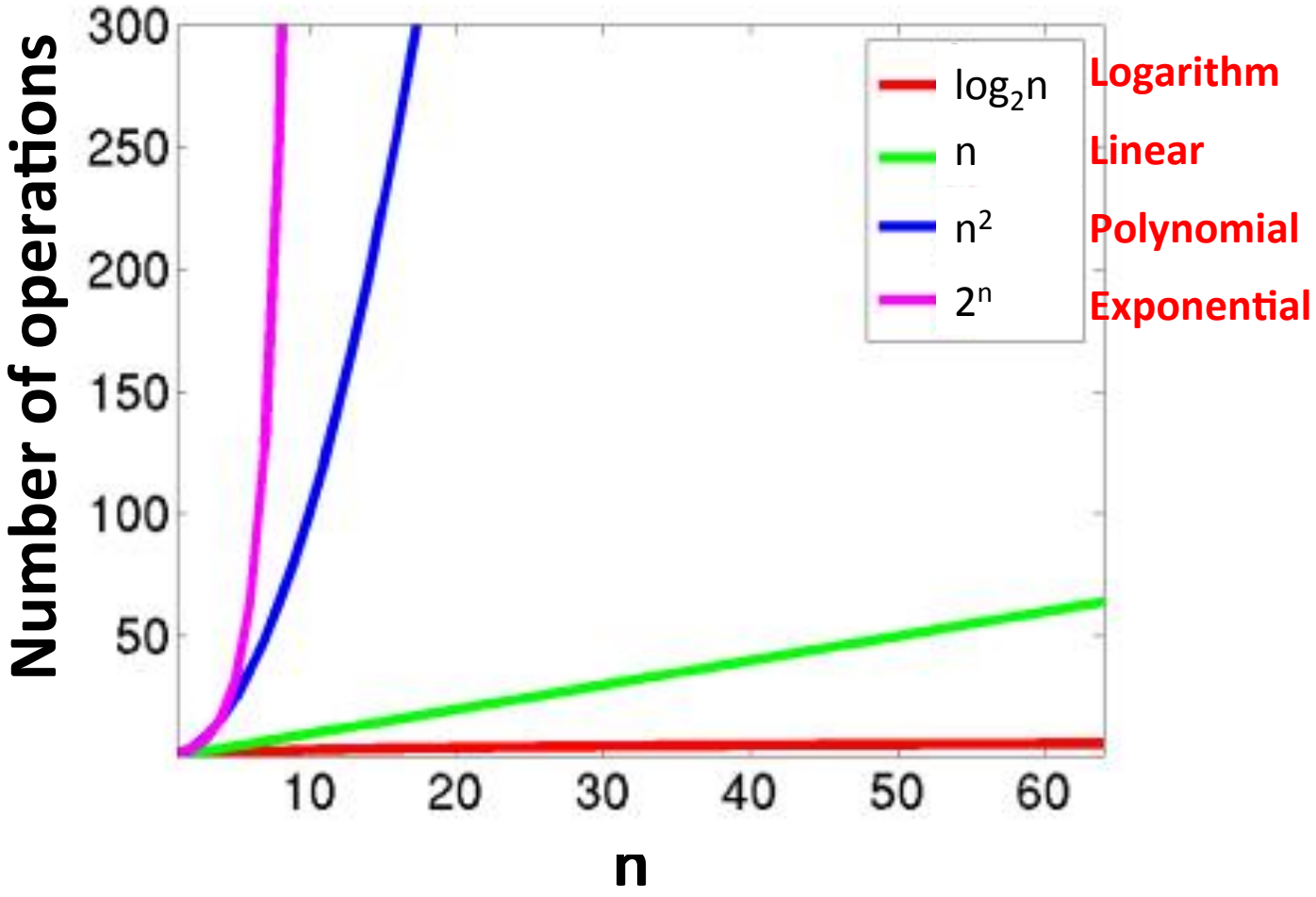
As *n* grows, number of operations between different algorithms begins to differ



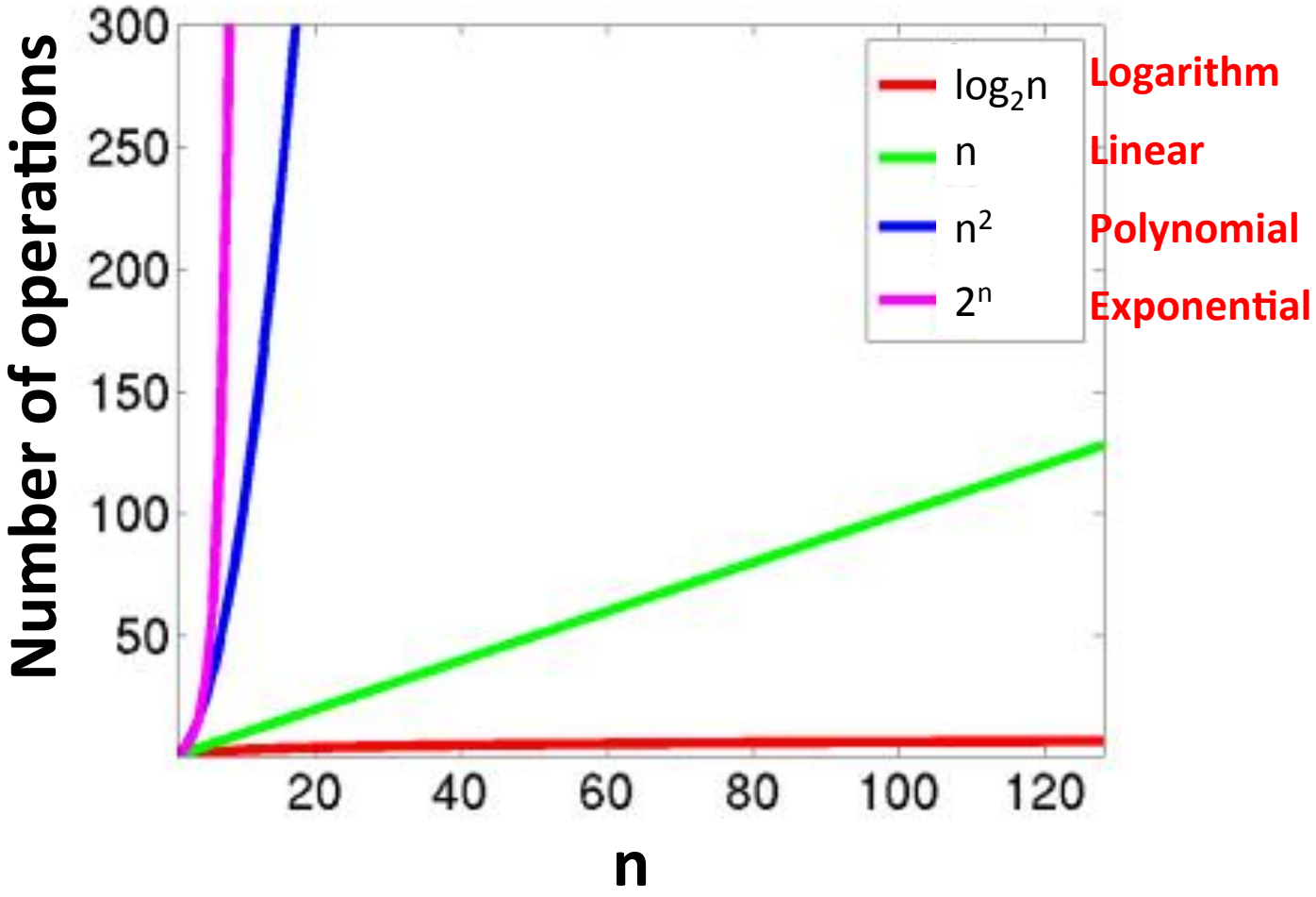
After n=4 Exponential is always greater than Polynomial

We will use that soon to define n_0 (standby for more info)

Even with only 60 items, there is a large difference in number of operations



Eventually, even with speedy computers, some algorithms become impractical



Sometimes complexity can hurt us, sometimes it can help us

Hurts us
Can't brute force chess algorithm 2^n

Helps us
Can't crack password algorithm 2^n

Images: thechessstore.com; studyoffice.org

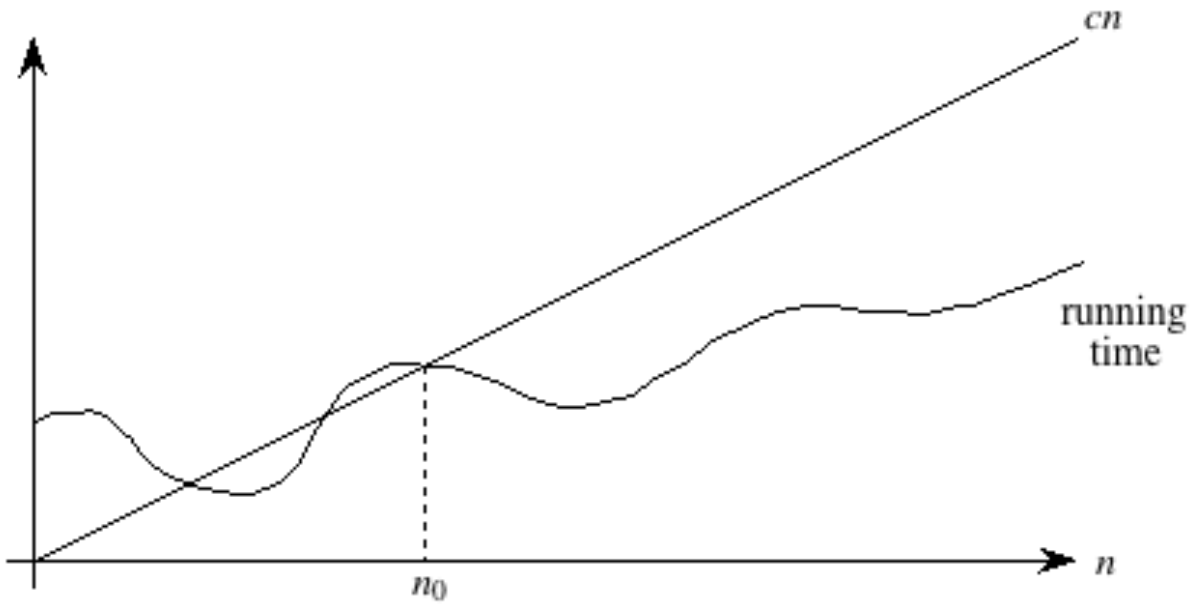
Agenda

1. Growing array List implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Computer scientists describe upper bounds on orders of growth with "Big Oh" notation

O gives an asymptotic <u>upper</u> bounds

"Big Oh of n", and "Oh of n", and "order n" all mean the same thing!



Example: find *<u>specific</u>* item in a list

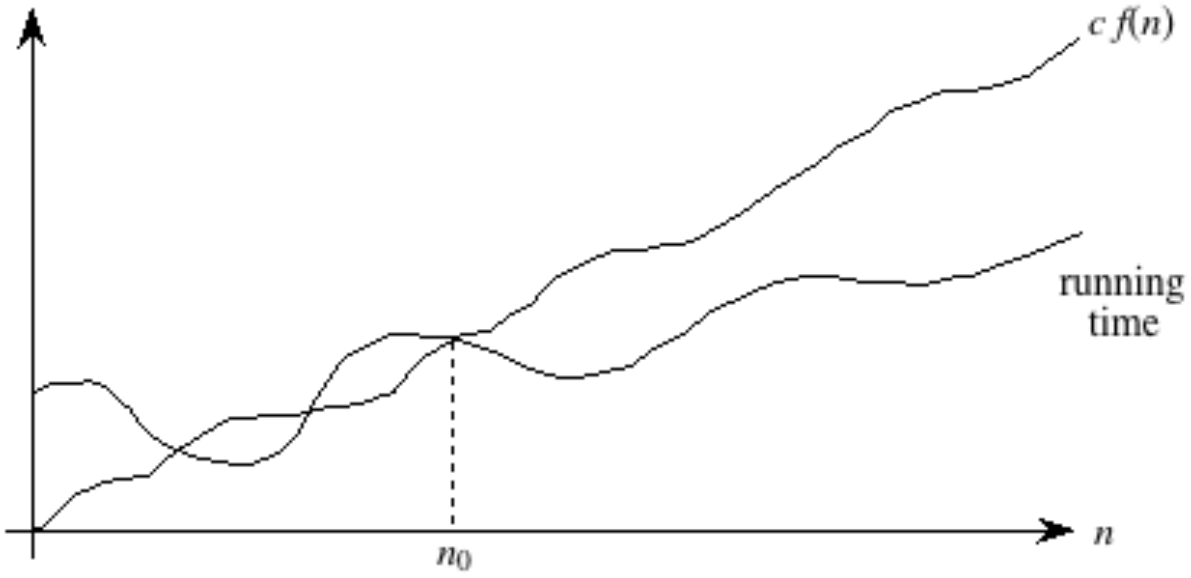
- **Might find item on first try**
- **Might not find it at all (have to check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

Run-time complexity is O(n) if there exists constants n_0 and c such that:

- $\forall\, n \geq n_0$
- run time of size n is at *<u>most</u>* cn, upper bound
- O(n) is the *<u>worst</u>* case performance for large n, but actual performance could be better
- O(n) is said to be "linear" time
- O(1) means constant time

We can extend Big Oh to any, not necessarily linear, function

O gives an asymptotic <u>upper</u> bounds



Run-time complexity is O(*f(n)*) if there exists constants n_0 and *c* such that:

- $\forall\, n \geq n_0$
- run time of size *n* is at *<u>most</u> cf(n)*, upper bound
- O(*f(n)*) is the *<u>worst</u>* case performance for large *n*, but actual performance could be better
- *f(n)* can be a non-linear function such as n^2 or *log(n)*
- In that case O(n^2) *or* O(*log n*)

We focus on upper bounds (worst case) for a number of reasons

Reasons to focus on worst case

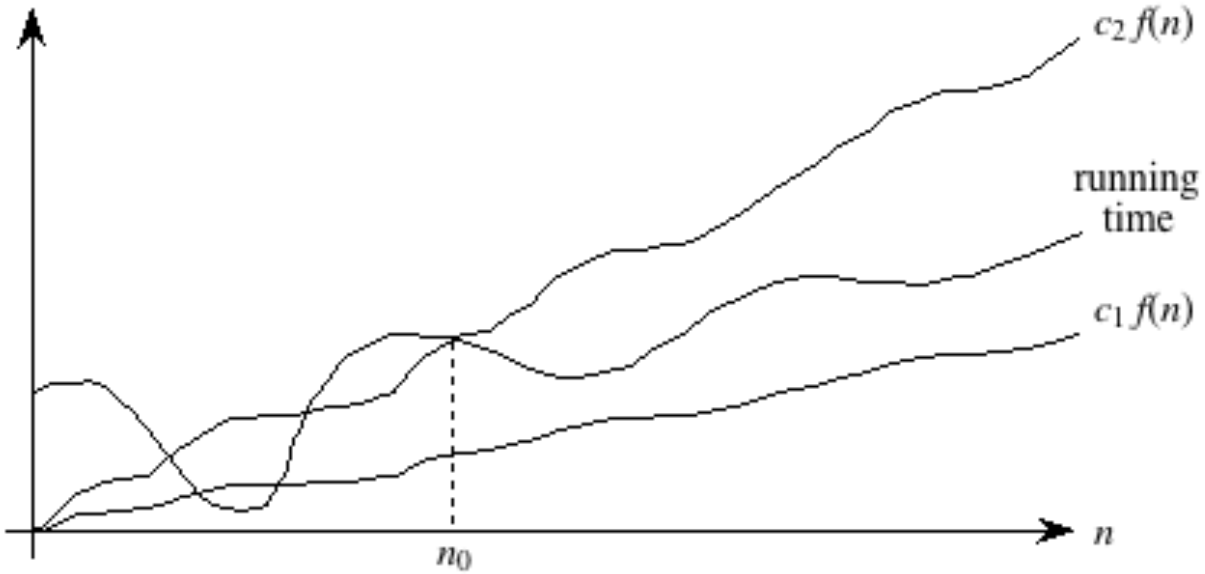
- Worst case gives upper bound on *any* input
- Gives a guarantee that algorithm never takes any longer
- We don't need to make an educated guess and hope that running time never gets much worse

Why not average case instead of worst case?

- Seems reasonable (sometimes we do)
- Need to define what *is* the average case: search example
  - Video database might return most popular items first, so might find popular items before obscure items
  - In cases like linear search, might find item half way (n/2)
  - Sometimes never find what you are looking for (n)
- Average case often about the same as worst case

Run time can also be Ω (Omega), where run time grows at least as fast

Ω gives an asymptotic <u>lower</u> bounds



Run-time complexity is $\Omega(f(n))$ if there exists constants n_0 and c_1 such that:

- $\forall\, n \geq n_0$
- run time of size n is at *<u>least</u>* $c_1 f(n)$, lower bound
- $\Omega(n)$ is the *<u>best</u>* case performance for large n, but actual performance can be worse

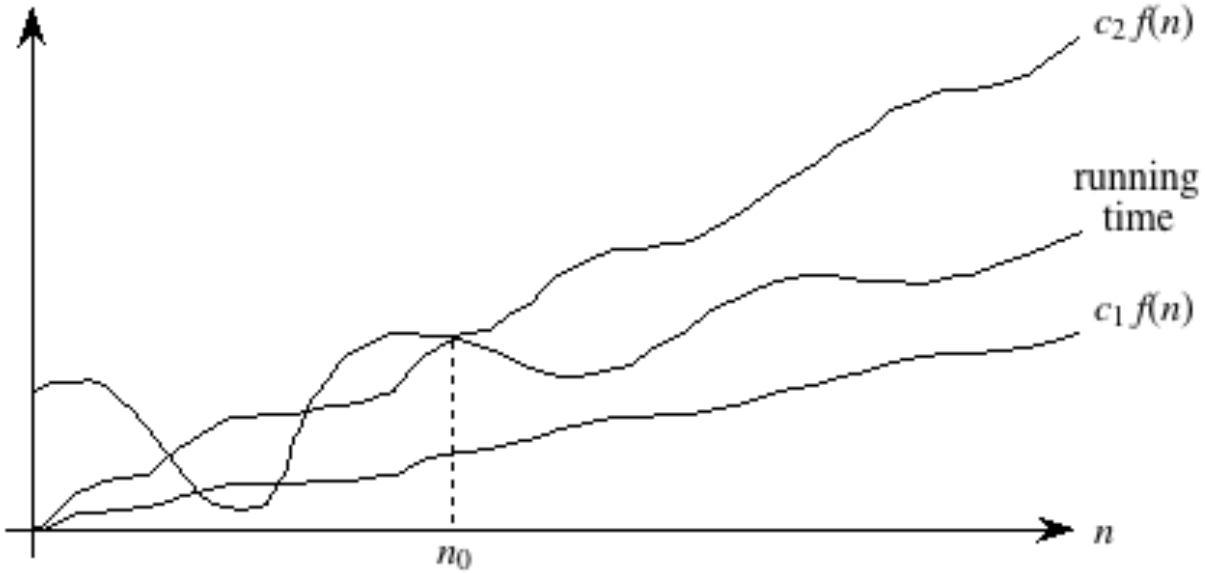
Example: find *<u>largest</u>* item in a list

- **Have to check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

We use Θ (Theta) for tight bounds when we can define O and Ω

Θ gives an asymptotic tight bounds

We can also apply these concepts to how much memory an algorithm uses (not just run-time complexity)



Example: find *largest* item in a list

- **Best case: already seen it is Ω(n)**
- **Worst case: have to check each item, so O(n)**
- **Because Ω(n) *and* O(n) we say it is Θ(n)**

Run-time complexity is Θ(*f(n)*) if there exists constants n_0 and c_1 and c_2 such that:

- $\forall\, n \geq n_0$
- run time of size *n* is at *least* $c_1 f(n)$ and at most $c_2 f(n)$
- Θ(*n*) gives a tight bound, which means run time will be within a constant factor
- Generally we will use either O or Θ
- **O, Ω, Θ called asymptotic notation**

We ignore constants and low-order terms in asymptotic notation

Constants don't matter, just adjust c_1 and c_2

- Constant multiplicative factors are absorbed into c_1 (and c_2)
- Example: $1000n^2$ is $O(n^2)$ because we can choose c_1 to be 1000 (remember bounded by $c_1 n$)
- Do care in practice – if an operation takes a constant time, O(1), but more than 24 hours to complete, can't run it everyday

Low order terms don't matter either

- If $n^2+1000n$, then choose $c_1 = 1$, so now $n^2 + 1000n \geq c_1 n^2$
- Now must find c_2 such that $n^2 + 1000n \leq c_2 n^2$
- Subtract n^2 from both sides and get $1000n \leq c_2 n^2 - n^2 = (c_2-1)n^2$
- Divide both sides by $(c_2-1)n$ gives $1000/(c_2-1) \leq n$
- Pick $c_2 = 2$ and $n_0 = 1000$, then $\forall\, n \geq n_0$, $1000 \leq n$
- So, $n^2 + 1000n \leq c_2 n^2$, try with $n=1000$ get $n^2 + 1000^2 = 2*n^2$
- **In practice, we simply ignore constants and low order terms**

Agenda

1. Growing array List implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

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

Worst case run-time complexity

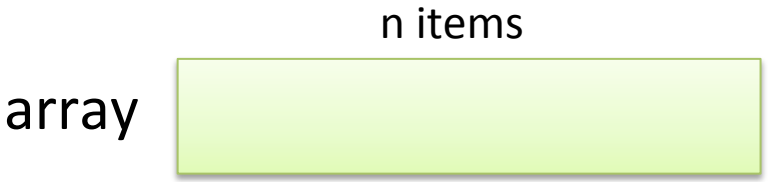
	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	O(n) + growth
remove(i)	O(n)	O(n)

- **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?**

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

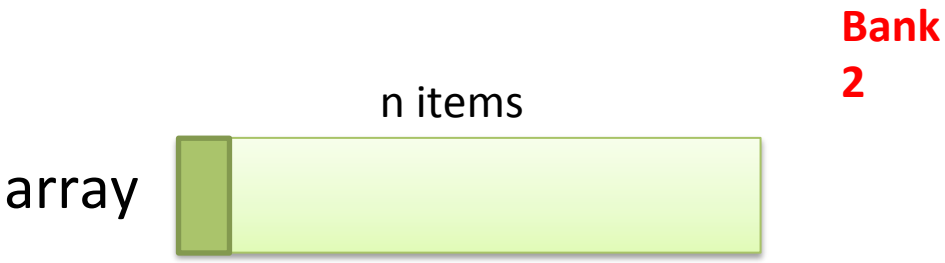
Amortized analysis



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**

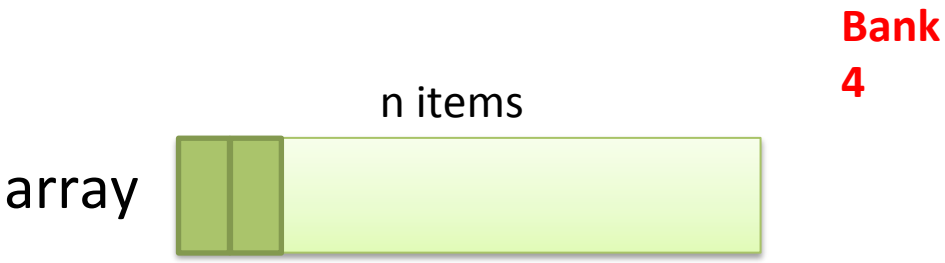
Amortized analysis shows growing array is actually only O(1)!



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**

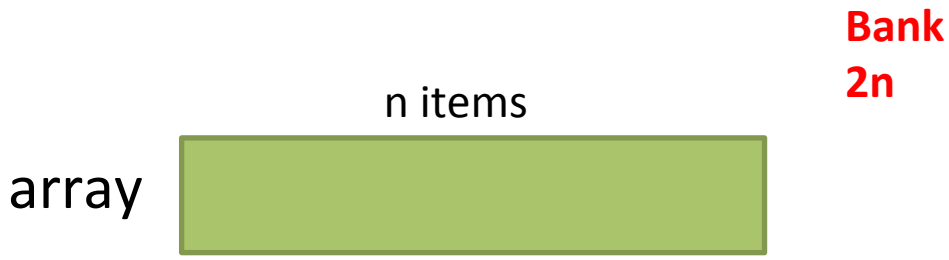
Amortized analysis shows growing array is actually only O(1)!



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)!

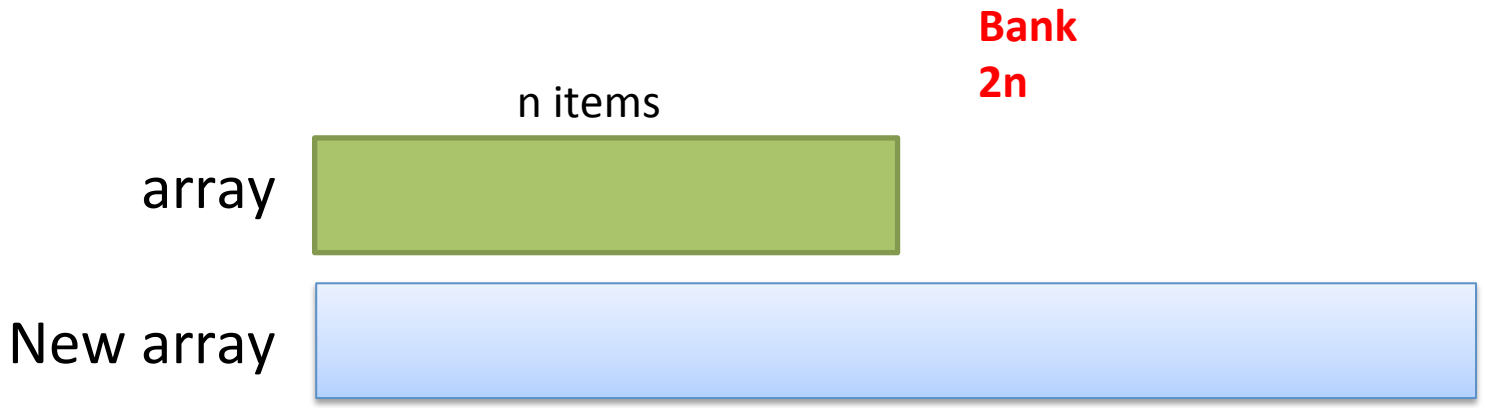


Each time add an item to array, *conceptually* charge 3 "tokens"

- **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)!



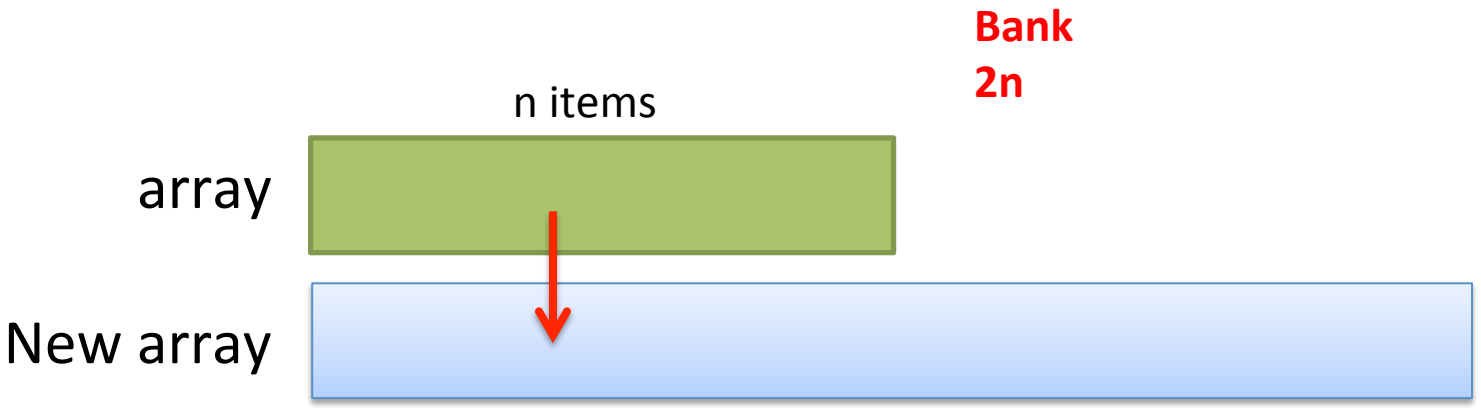
Each time add an item to array, *conceptually* charge 3 "tokens"

- **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, *conceptually* charge 3 "tokens"

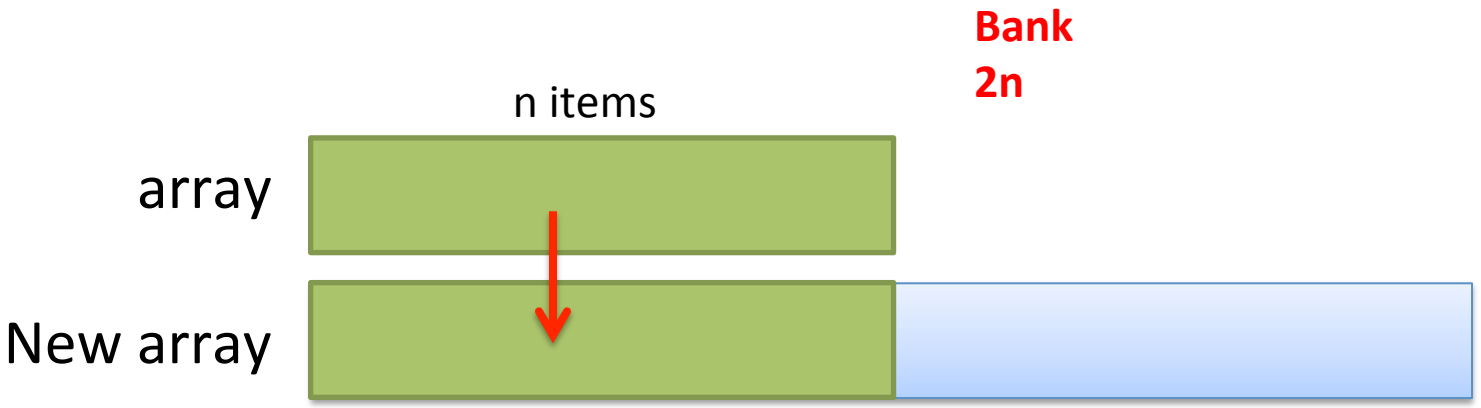
- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

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"

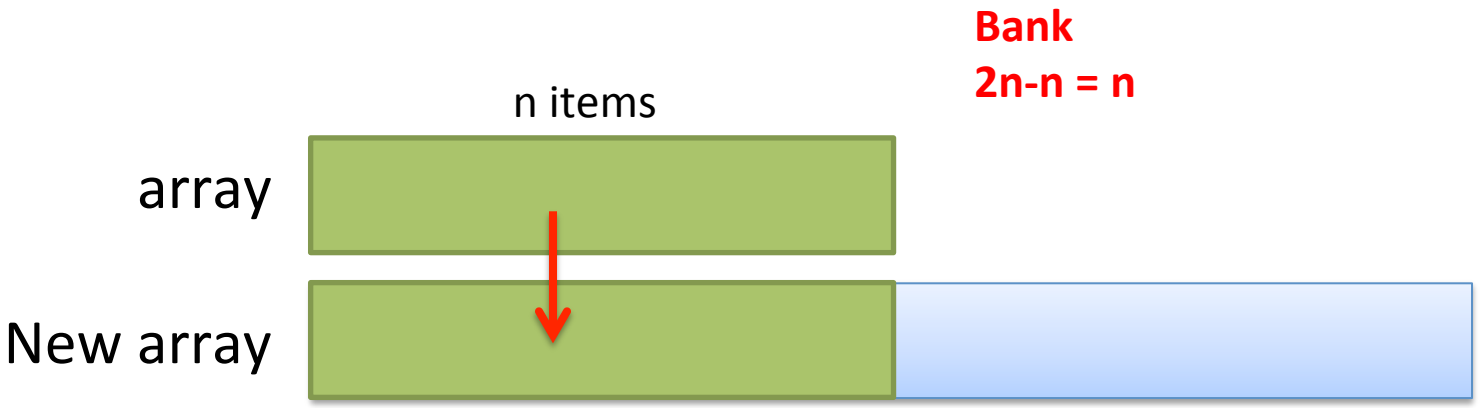
- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

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

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



Each time add an item to array, 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**

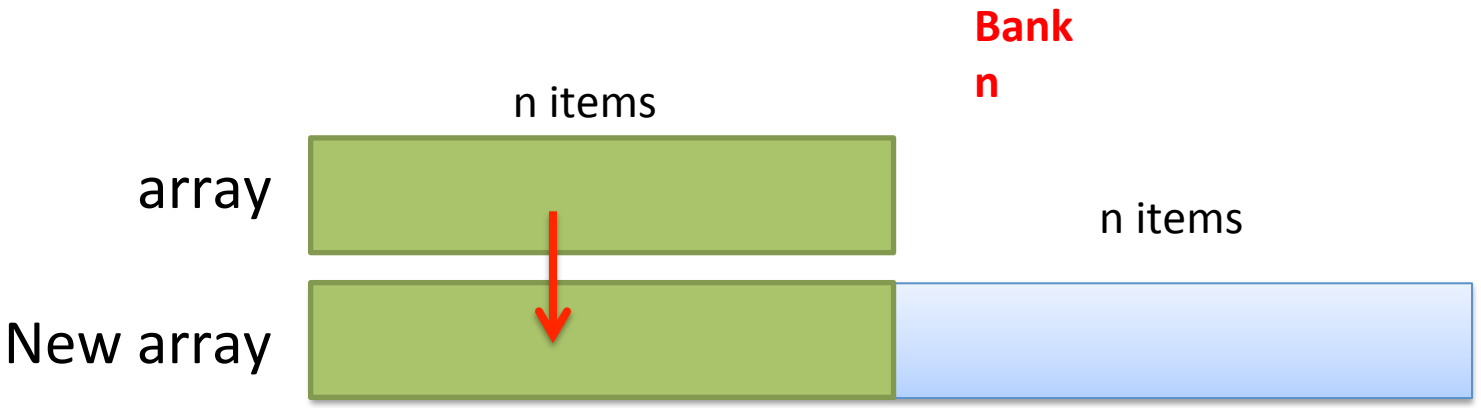
After *n add()* operations, array is full, but have *2n* tokens in bank

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)!



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

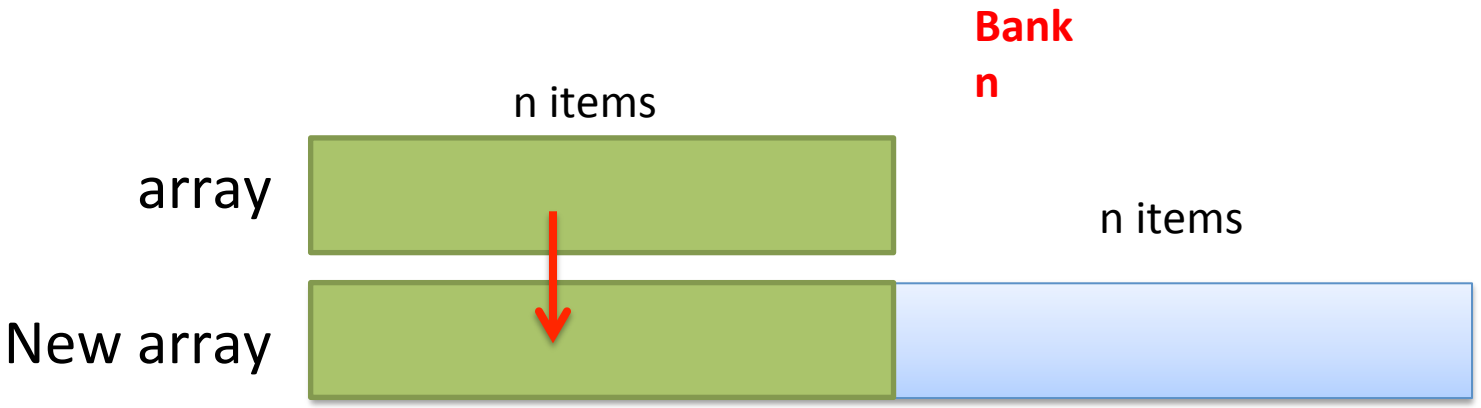
Allocate new 2X larger array

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, *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

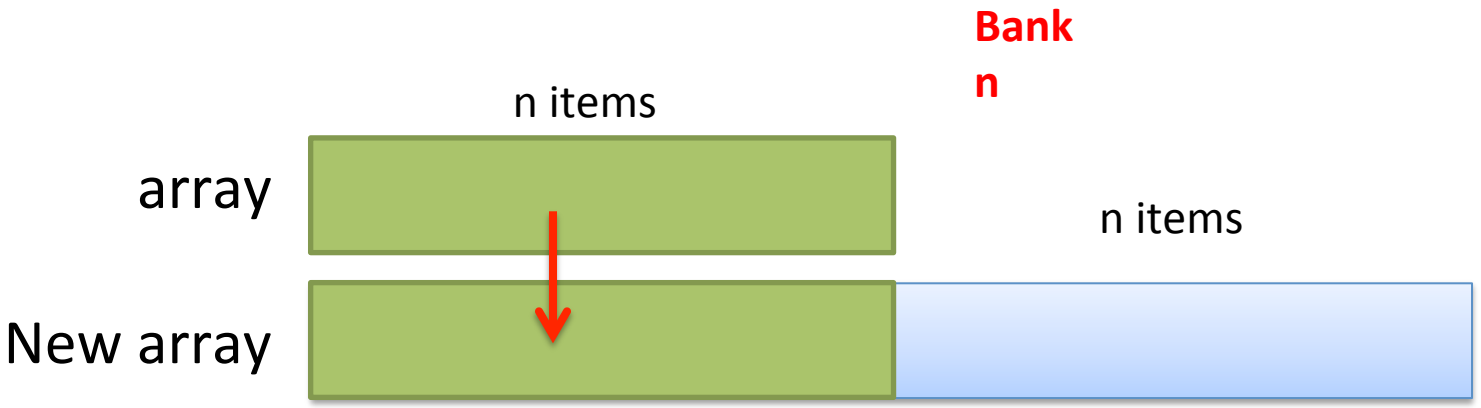
Copy elements from old array to new array

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

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

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 "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

Agenda

1. Growing array List implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Its so common to march down a list of items that Java makes it easy with iterators

Traditional for loop

```
for (int i=0;
     i<blobs.size();
     i++) {
   blobs.get(i).step();
}
```

Comments

- i serves no real purpose, don't really care what its value is at any point
- *get()* is reset every time, doesn't keep track of where it was last
- **Could lead to O(n^2) on linked lists**

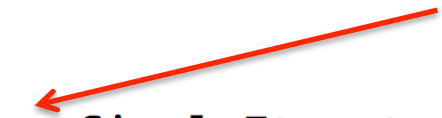
Iterator

```
For (Blob b : blobs) {
   b.step();
}
```

Comments

- Easier to read?
- Keeps track of where it left off
- Implicitly uses iterator
- Iterator has two main methods:
  - `hasNext()` can advance?
  - `next()` do advance

SimpleIterator.java: Defines an iterator interface




```
4  public interface SimpleIterator<T> {
5      /**
6       * Is there anything left to retrieve?
7       */
8      public boolean hasNext();
9
10     /**
11      * Returns the next item and advances the iterator.
12      * Throws an exception if there is no next item.
13      */
14     public T next() throws Exception;
15 }
16
```

Only two methods:

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

ISinglyLinked.java: Same linked list implementation, now with iterator

```
29
30    private class IterSinglyLinked implements SimpleIterator<T> { //private cla
31        Element curr = head;     // next element to return
32
33        public boolean hasNext() {
34            return curr != null;
35        }
36
37        public T next() throws Exception {
38            if (curr == null) throw new Exception("no more elements");
39            T data = curr.data;
40            curr = curr.next;
41            return data;
42        }
43    }
44
45    public SimpleIterator<T> newIterator() { //satisfy new iterator requirement
46        return new IterSinglyLinked();
47    }
48
49
50    //no changes from previous version from here on down
```

- **Private class within ISinglyLinked, implements SimpleIterator**
- **Keeps pointer to current element**
- **Initially set to *head***

- **Implements required interface methods:**
  - ***hasNext()* – true if more items**
  - ***next()* – return current item and move to next item**

- **NOTE: there is also an iterator version for growing arrays**
- **Here we use the linked list version, but both function identically**
- **Array version just has *int curr***

***newIterator* in ISinglyLinked creates a new iterator**

IterTest.java: Use iterator to manipulate list

```
4  public class IterTest {
5      public static void main(String[] args) throws Exception {
6          SimpleIList<String> list1 = new ISinglyLinked<String>();
7          SimpleIList<String>list2 = new ISinglyLinked<String>();
8          //SimpleIList<String> list1 = new IGrowingArray<String>();
9          //SimpleIList<String> list2 = new IGrowingArray<String>();
10         list1.add(0, "e");
11         list1.add(0, "d");
12         list1.add(0, "c");
13         list1.add(0, "b");
14         list1.add(0, "a");
15
16
17         // Print the elements, using an index
18         for (int i=0; i<list1.size(); i++)
19             System.out.println(list1.get(i));
20
21
22         // Print the elements, using an iterator
23         for (SimpleIterator<String> i = list1.newIterator(); i.hasNext(); )
24             System.out.println(i.next());
25
```

- **Creates two SimpleILists (remember I version has iterator, otherwise same as SinglyLinked from last class)**

IterTest.java: Use iterator to manipulate list

```
4  public class IterTest {
5      public static void main(String[] args) throws Exception {
6          SimpleIList<String> list1 = new ISinglyLinked<String>();
7          SimpleIList<String>list2 = new ISinglyLinked<String>();
8          //SimpleIList<String> list1 = new IGrowingArray<String>();
9          //SimpleIList<String> list2 = new IGrowingArray<String>();
10         list1.add(0, "e");
11         list1.add(0, "d");
12         list1.add(0, "c");
13         list1.add(0, "b");
14         list1.add(0, "a");
15
16
17         // Print the elements, using an index
18         for (int i=0; i<list1.size(); i++)
19             System.out.println(list1.get(i));
20
21
22         // Print the elements, using an iterator
23         for (SimpleIterator<String> i = list1.newIterator(); i.hasNext(); )
24             System.out.println(i.next());
25
```

- **Creates two SimpleILists (remember I version has iterator, otherwise same as SinglyLinked from last class)**
- **Add some elements**

IterTest.java: Use iterator to manipulate list

```
4  public class IterTest {
5      public static void main(String[] args) throws Exception {
6          SimpleIList<String> list1 = new ISinglyLinked<String>();
7          SimpleIList<String>list2 = new ISinglyLinked<String>();
8          //SimpleIList<String> list1 = new IGrowingArray<String>();
9          //SimpleIList<String> list2 = new IGrowingArray<String>();
10         list1.add(0, "e");
11         list1.add(0, "d");
12         list1.add(0, "c");
13         list1.add(0, "b");
14         list1.add(0, "a");
15
16
17         // Print the elements, using an index
18         for (int i=0; i<list1.size(); i++)
19             System.out.println(list1.get(i));
20
21
22         // Print the elements, using an iterator
23         for (SimpleIterator<String> i = list1.newIterator(); i.hasNext(); )
24             System.out.println(i.next());
25
```

- **Print the elements in the list without using the iterator**
- **This is $O(n^2)$, why?**
- **Because with the linked list the *get(i)* operation has to start at the *head* and march down *i* items each time it is called!**
- **Sneaky inefficiency!**
- **Not a problem for the array implementation, however**

IterTest.java: Use iterator to manipulate list

```
4  public class IterTest {
5      public static void main(String[] args) throws Exception {
6          SimpleIList<String> list1 = new ISinglyLinked<String>();
7          SimpleIList<String>list2 = new ISinglyLinked<String>();
8          //SimpleIList<String> list1 = new IGrowingArray<String>();
9          //SimpleIList<String> list2 = new IGrowingArray<String>();
10         list1.add(0, "e");
11         list1.add(0, "d");
12         list1.add(0, "c");
13         list1.add(0, "b");
14         list1.add(0, "a");
15
16
17         // Print the elements, using an index
18         for (int i=0; i<list1.size(); i++)
19             System.out.println(list1.get(i));
20
21
22         // Print the elements, using an iterator
23         for (SimpleIterator<String> i = list1.newIterator(); i.hasNext(); )
24             System.out.println(i.next());
25
```

- Print elements using iterator is O(n)
- Why?
- Iterator keeps track of current position in list using *curr* pointer
- Does not need to start at the *head* each time

IterTest.java: Use iterator to manipulate list

```
4  public class IterTest {
5      public static void main(String[] args) throws Exception {
6          SimpleIList<String> list1 = new ISinglyLinked<String>();
7          SimpleIList<String>list2 = new ISinglyLinked<String>();
8          //SimpleIList<String> list1 = new IGrowingArray<String>();
9          //SimpleIList<String> list2 = new IGrowingArray<String>();
10         list1.add(0, "e");
11         list1.add(0, "d");
12         list1.add(0, "c");
13         list1.add(0, "b");
14         list1.add(0, "a");
15
16
17         // Print the elements, using an index
18         for (int i=0; i<list1.size(); i++)
19             System.out.println(list1.get(i));
20
21
22         // Print the elements, using an iterator
23         for (SimpleIterator<String> i = list1.newIterator(); i.hasNext(); )
24             System.out.println(i.next());
25
```

I prefer this way:

```
SimpleIterator<String> i = list1.newIterator();
while (i.hasNext()) {
        System.out.println(i.next());
}
```