CS 10:
Problem solving via Object Oriented Programming

Lists Part 1

Main goals

- Implement singly linked list
- Handle exceptions
- Use iterators

Reminder: List ADT defines required operations, not implementation

List ADT

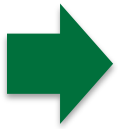
Operation	Description
`size()`	Return number of items in List
`isEmpty()`	True if no items in List, otherwise false
`add(e)`	Add item *e* to end of the list
`add(i,e)`	Insert item *e* at index *i*, moving all subsequent items one index larger
`remove(i)`	Remove and return item at index *i*, move all subsequent items one index smaller
`get(i)`	Return the item at index *i*
`set(i,e)`	Replace the item at index *i* with item *e*

SimpleList.java is an interface that specifies what operations MUST be implemented

SimpleList.java

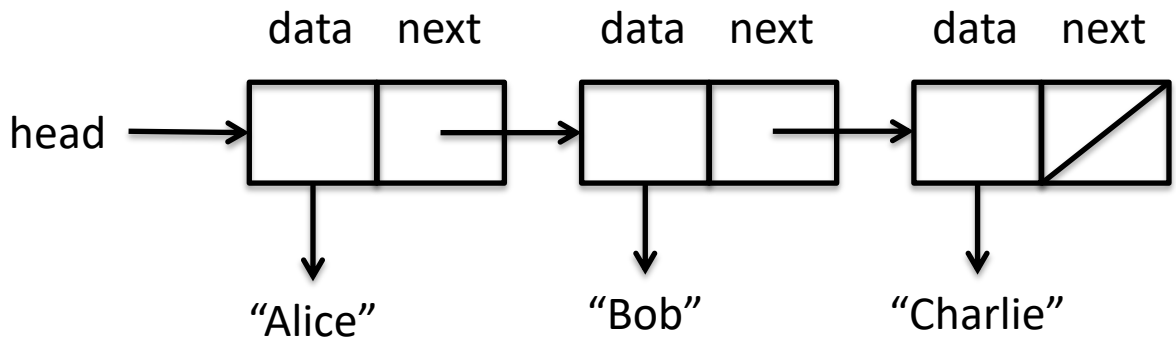
```
public interface SimpleList<T> extends Iterable<T> {
  /**
   * Returns # elements in the List (they are indexed 0..size-1)
   */
  public int size();

  /**
   * Returns true if there are no elements in the List, false otherwise
   * @return true or false
   */
  public boolean isEmpty();

  /**
   * Adds the item at the index, which must be between 0 and size
   */
  public void add(int idx, T item) throws Exception;

  /**
   * Add item at end of List
   */
  public void add(T item) throws Exception;

  /**
   * Removes and returns the item at the index, which must be between 0 and size-1
   */
  public T remove(int idx) throws Exception;

  /**
   * Returns the item at the index, which must be between 0 and size-1
   */
  public T get(int idx) throws Exception;

  /**
   * Replaces the item at the index, which must be between 0 and size-1
   */
  public void set(int idx, T item) throws Exception;
```

Agenda



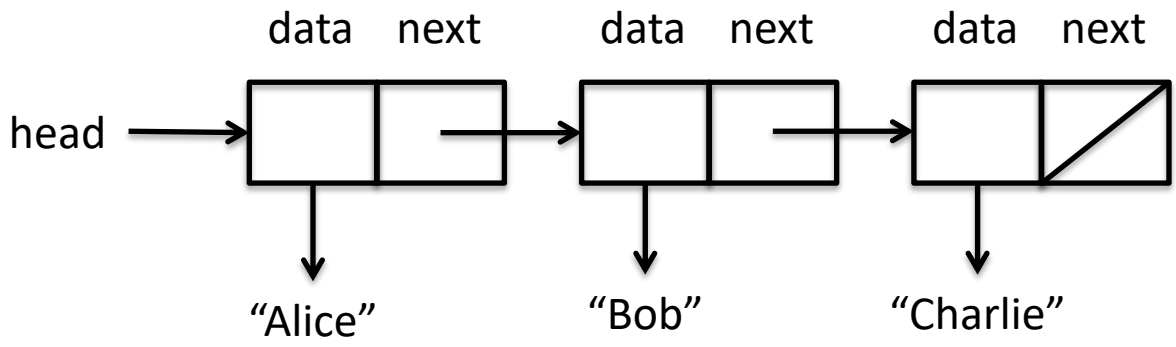
1. Singly linked list implementation
2. Exceptions
3. Iterators

Singly linked list review: elements have data and a next pointer



To get an item at index i, start at head and march down

get(i) – return item at specified index

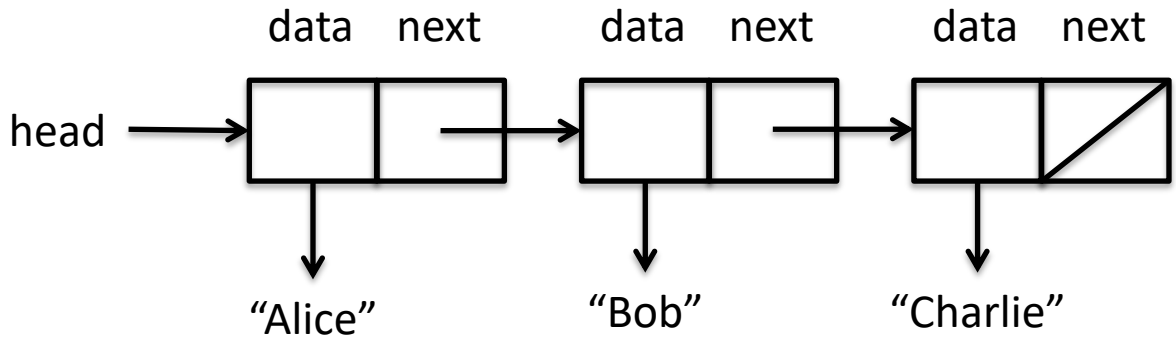


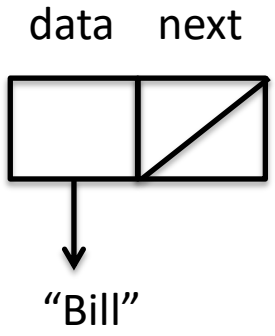
Get item at index 2

On paper example

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")





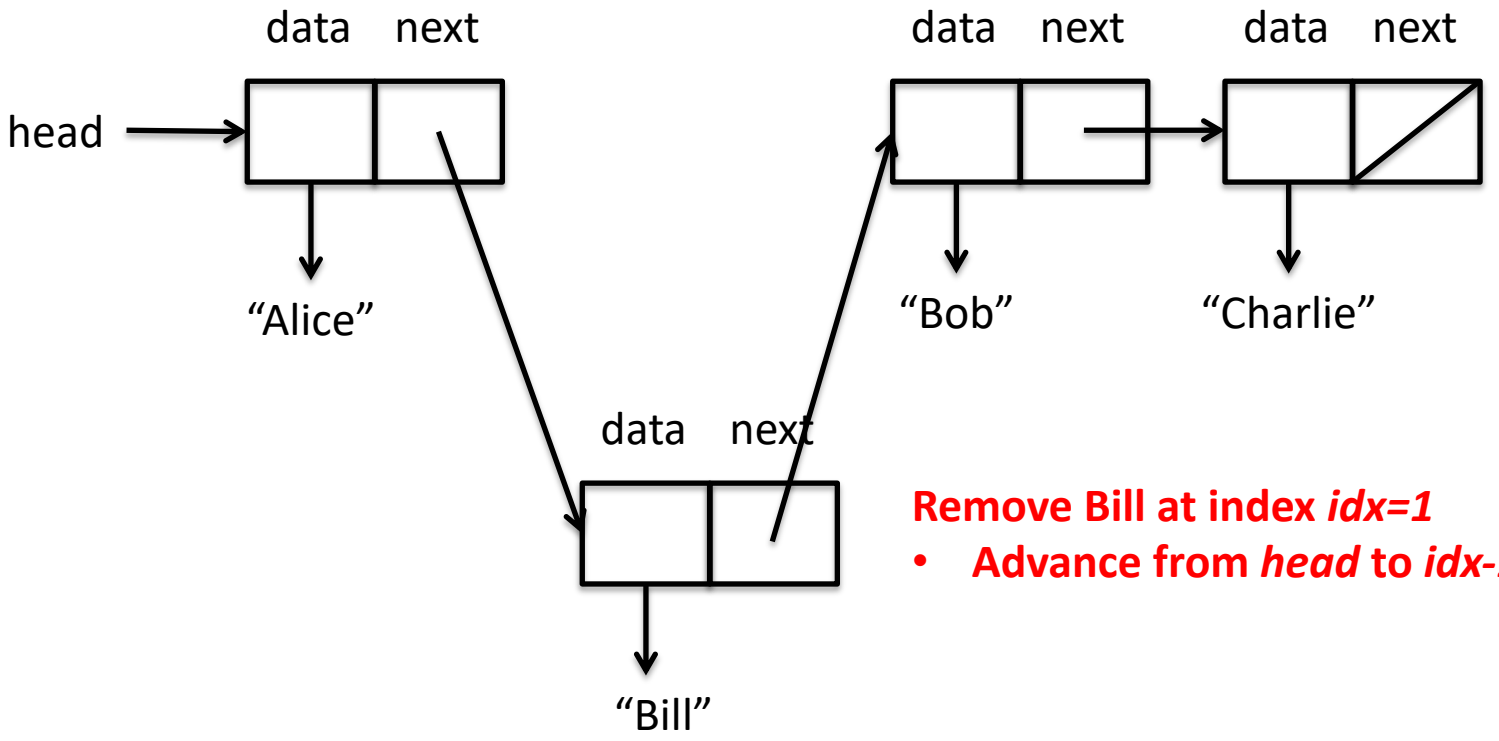
Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**

On paper example

remove() takes an item out of the list by updating next pointer

remove(1)



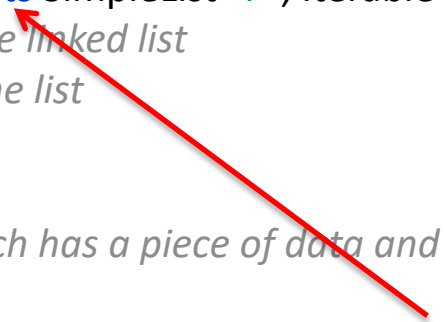
Remove Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**

On paper example

SinglyLinked.java: Implementation of List interface

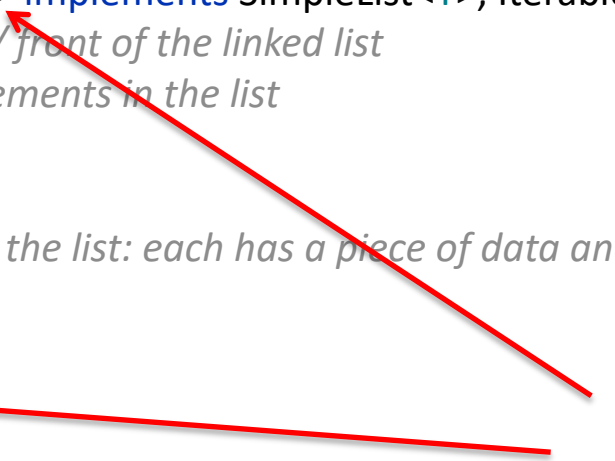
SinglyLinked.java



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

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

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

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

Lists hold items of generic type

SinglyLinked.java

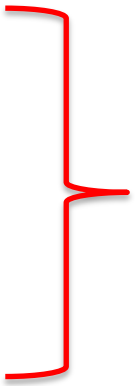


```
public class SinglyLinked<T> implements SimpleList<T>, Iterable<T> {
  private Element head;  // front of the linked list
  private int size;     // # elements in the list

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

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

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

Implement a private "nested" class to hold data and next pointer

SinglyLinked.java

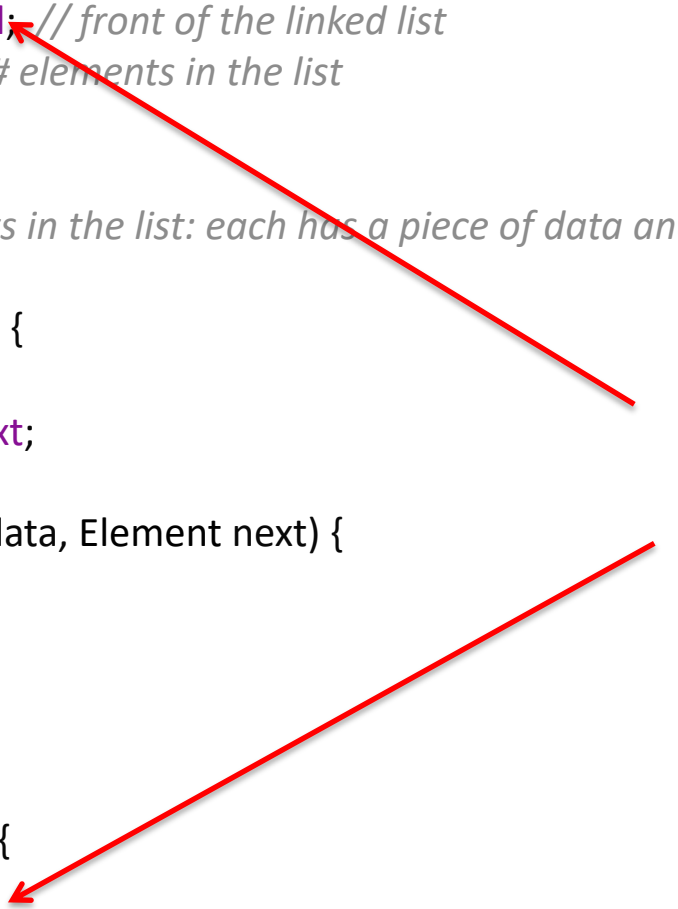
```
public class SinglyLinked<T> implements SimpleList<T>, Iterable<T> {
  private Element head;  // front of the linked list
  private int size;    // # elements in the list

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

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

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



Set head to null and size to zero in constructor

SinglyLinked.java

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

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

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

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



Increment size instance variable on add, decrement on remove operation

SinglyLinked.java

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

/**
 * Returns true if there are no elements in the List, false otherwise
 * @return true or false
 */
public boolean isEmpty() {

}
```

- **How can *isEmpty()* be easily implemented?**

Implementing *isEmpty* "isEasy" ☺

SinglyLinked.java

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

/**
 * Returns true if there are no elements in the List, false otherwise
 * @return true or false
 */
public boolean isEmpty() {
   return size == 0;
}
```

- **How can *isEmpty()* be easily implemented?**
- **Run-time complexity?**
- **O(1)**

advance is a helper method to move to the n^{th} item in the List

SinglyLinked.java

```
/**
 * Helper function, advancing to the nth Element in the list and returning it
 * (exception if not that many elements)
 */
private Element advance(int n) throws Exception {
   Element e = head;
   //safety check for valid index (don't assume caller checked!)
   if (e == null || n < 0 || n >= size) {
     throw new Exception("invalid advance");
   }

   // Just follow the next pointers n times
   for (int i = 0; i < n; i++) {
     e = e.next;
   }
   return e;
}
```

add method uses *advance*

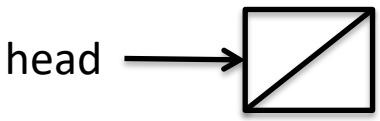
SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add method uses *advance*

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

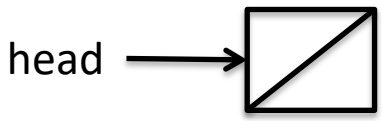


No need to advance if adding at the head

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,15)

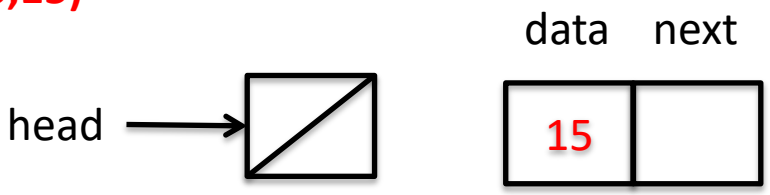


Just create a new Element and point head to it

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

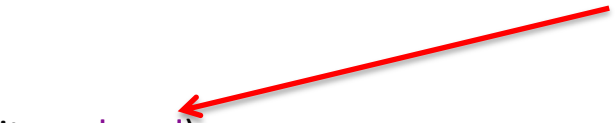
add(0,15)



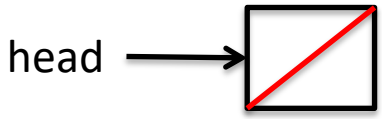
Just create a new Element and point head to it

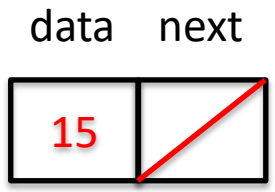
SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



add(0,15)



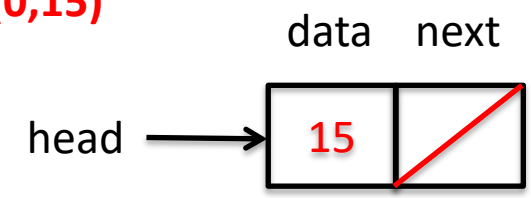


Just create a new Element and point head to it

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,15)

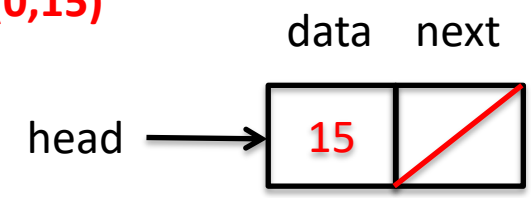


Don’t forget to increment *size*

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,15)

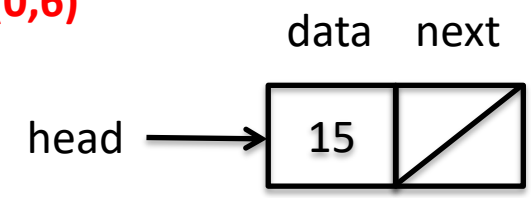


Adding at head if the List is not empty is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,6)

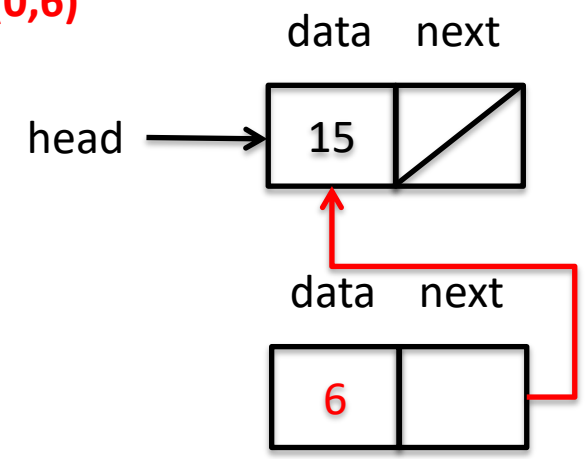


Adding at head if the List is not empty is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,6)

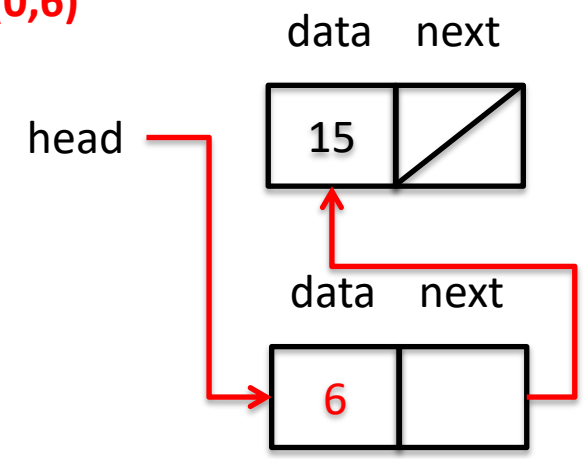


Adding at head if the List is not empty is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

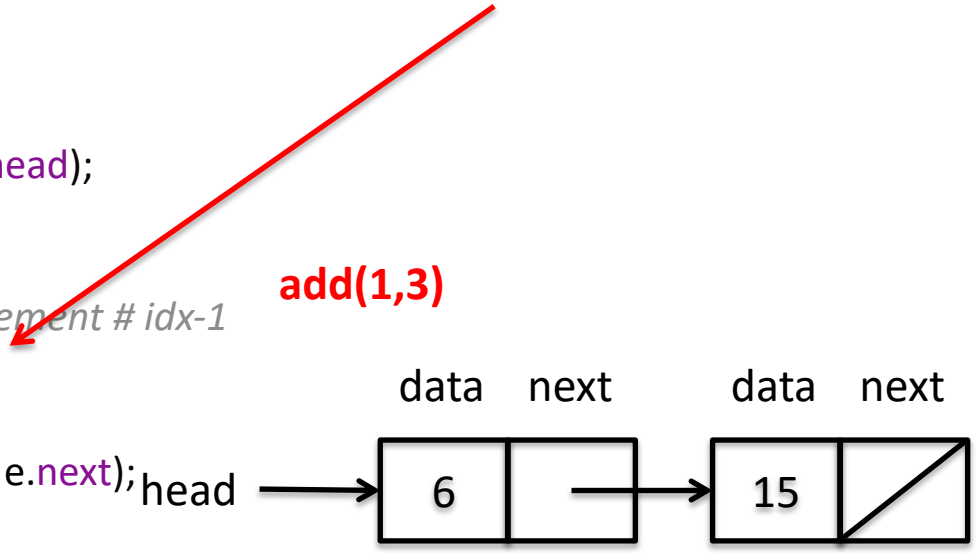
add(0,6)



If adding NOT at head, use *advance* method

SinglyLinked.java

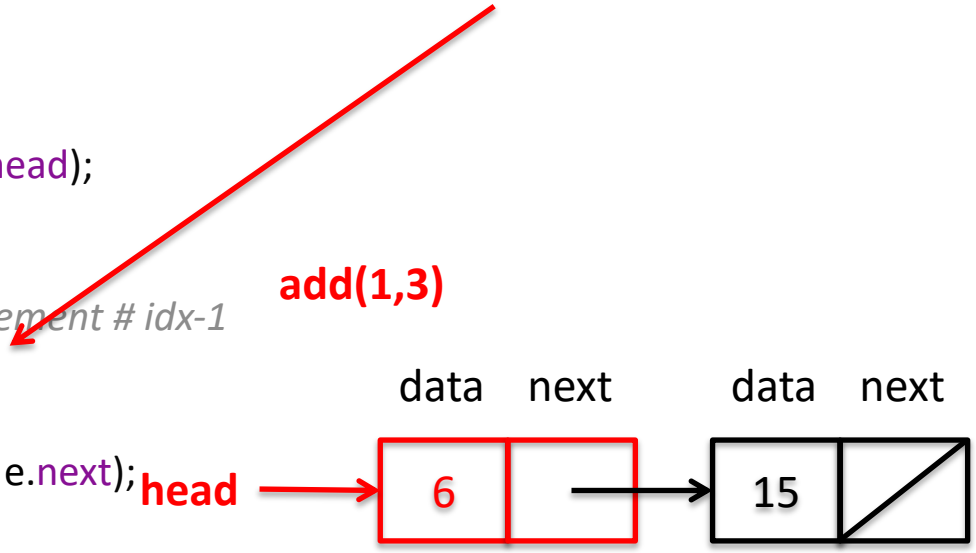
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



Move to index idx-1

SinglyLinked.java

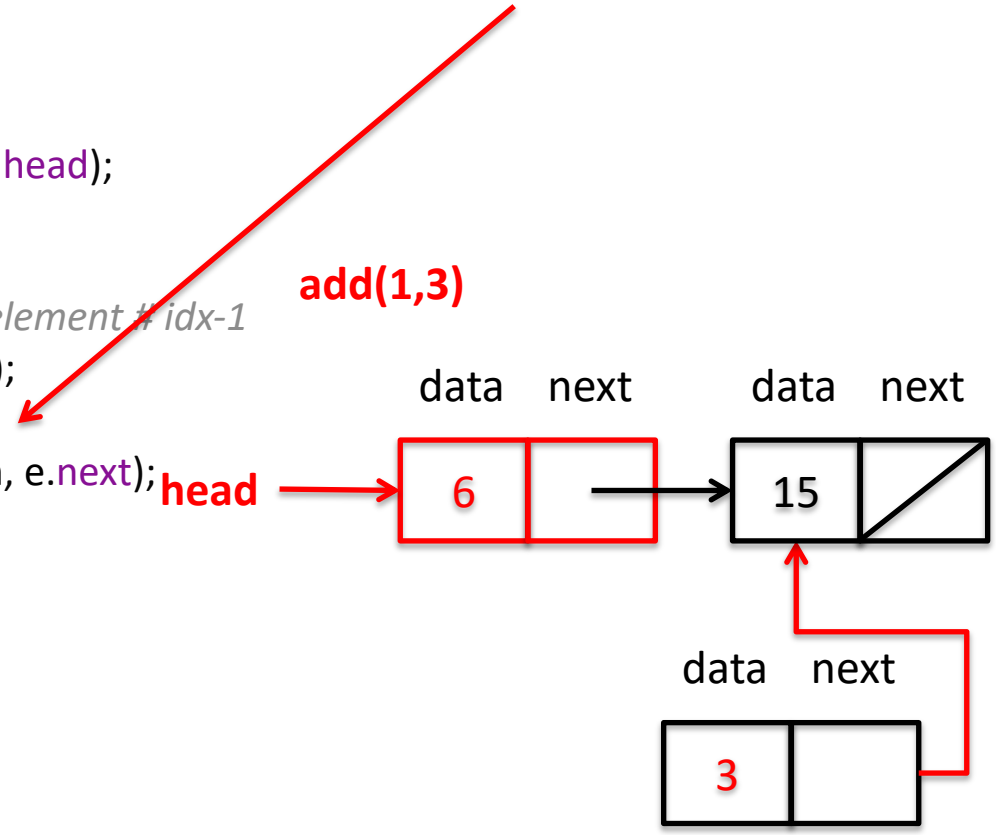
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



Splice in a new Element that points to where Element at idx-1 points

SinglyLinked.java

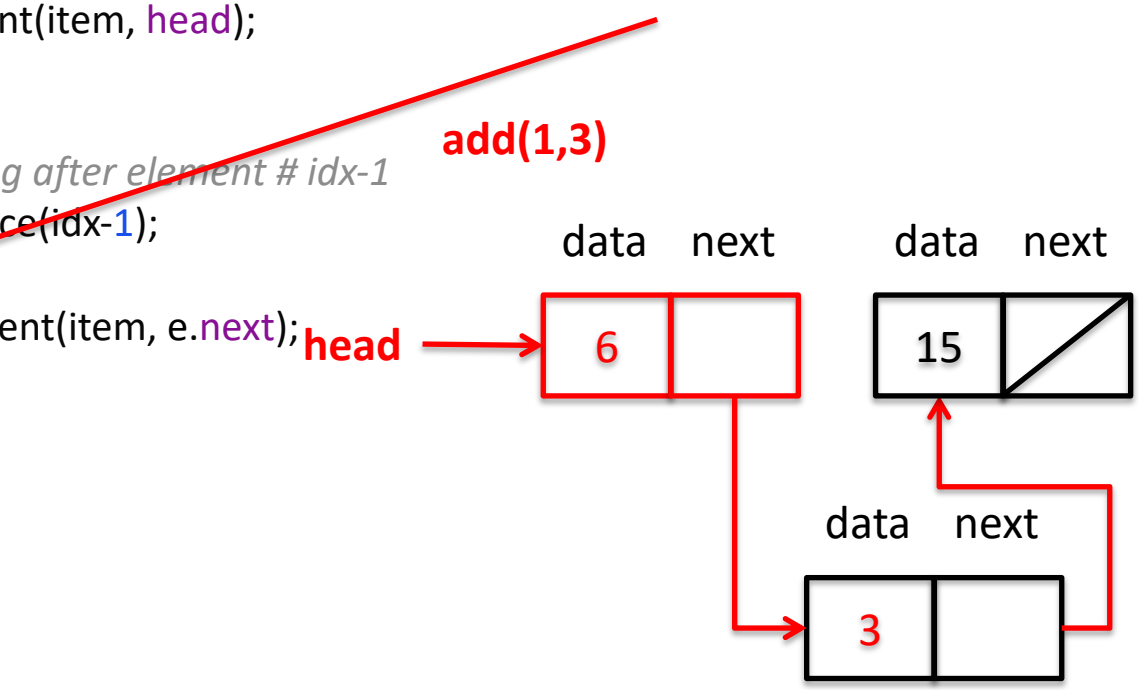
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



Set idx-1 to point to new Element

SinglyLinked.java

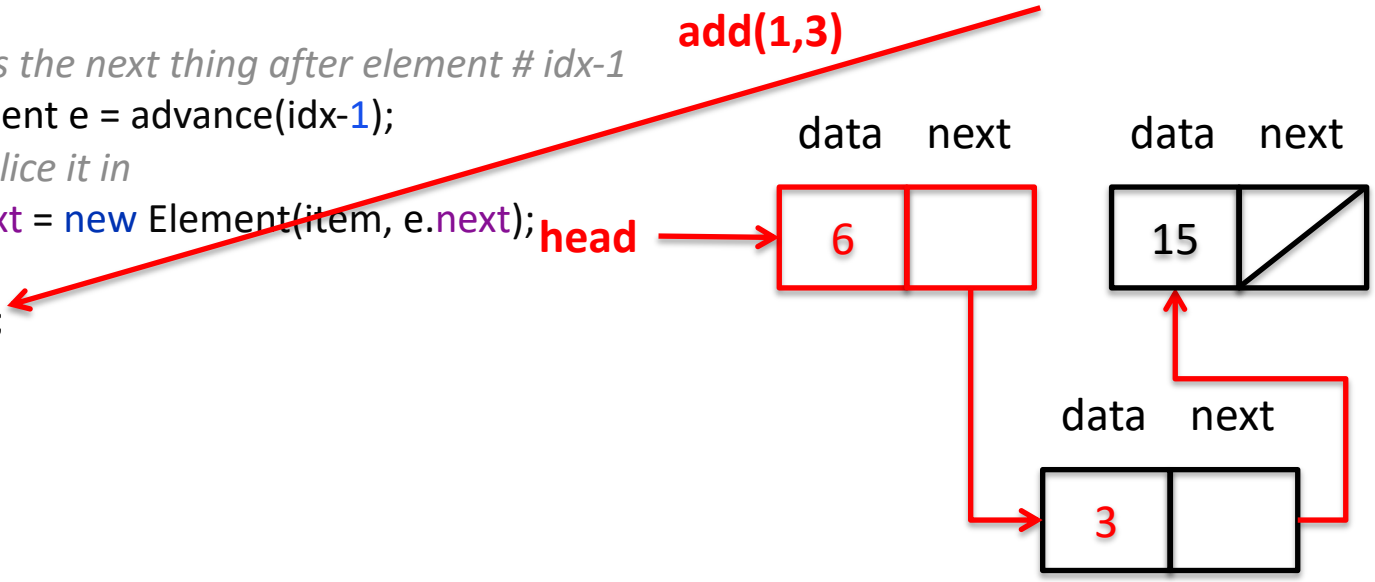
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



Don’t forget to increment *size*

SinglyLinked.java

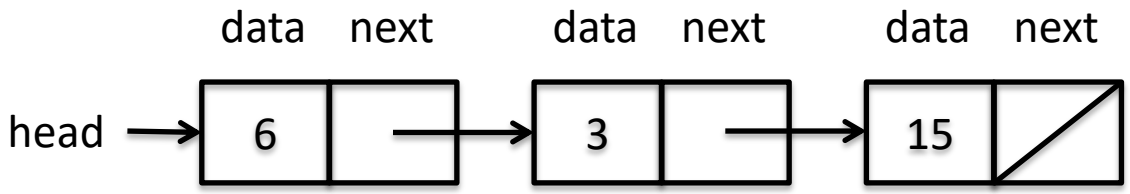
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```



Adding at the end is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
    }
    else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
    }
    else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
    }
    size++;
}

public void add(T item) throws Exception {

}
```

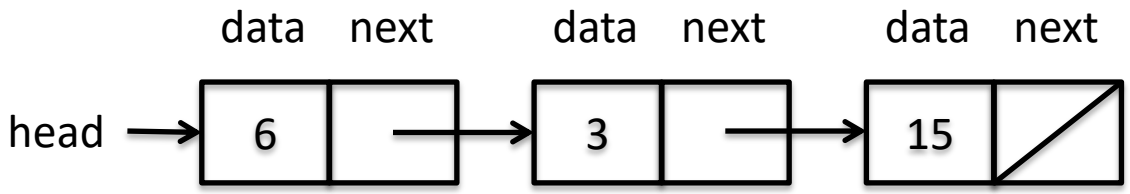


How to easily add at the end?

Adding at the end is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```




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

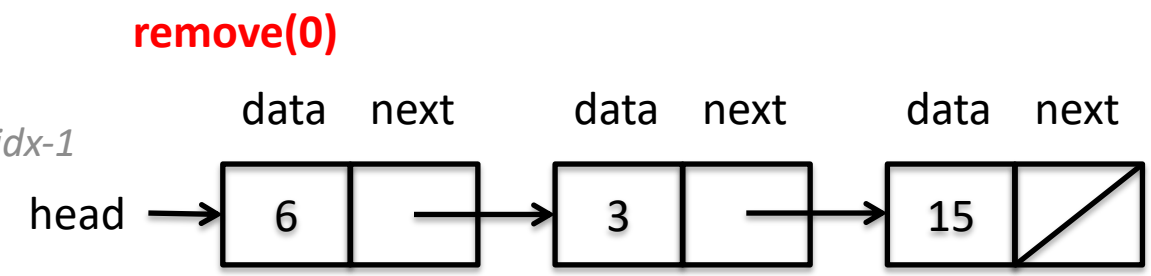
How to easily add at the end?
Just call add with size as index

On SA-4 you'll do something more efficient using a tail pointer

remove method removes and returns data at *idx*

SinglyLinked.java

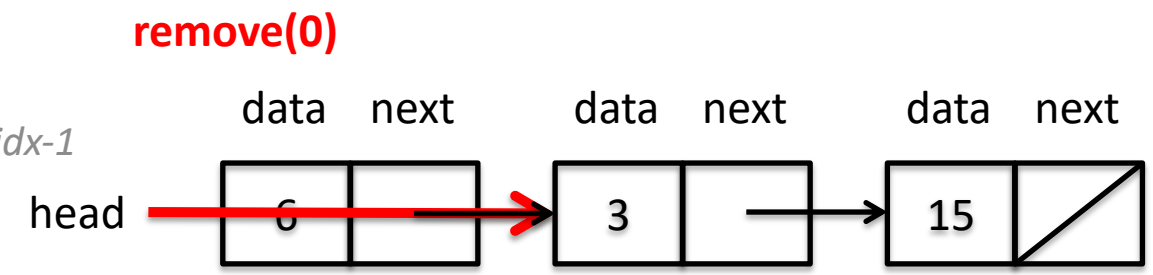
```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```



If removing at head, just set head to head.next

SinglyLinked.java

```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```

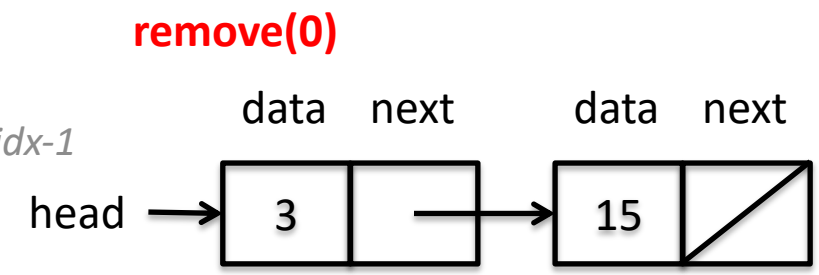


- **What happens to the old head element?**
- **Garbage collected! (memory returned to the Operating System)**

If removing at head, just set head to head.next

SinglyLinked.java

```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```

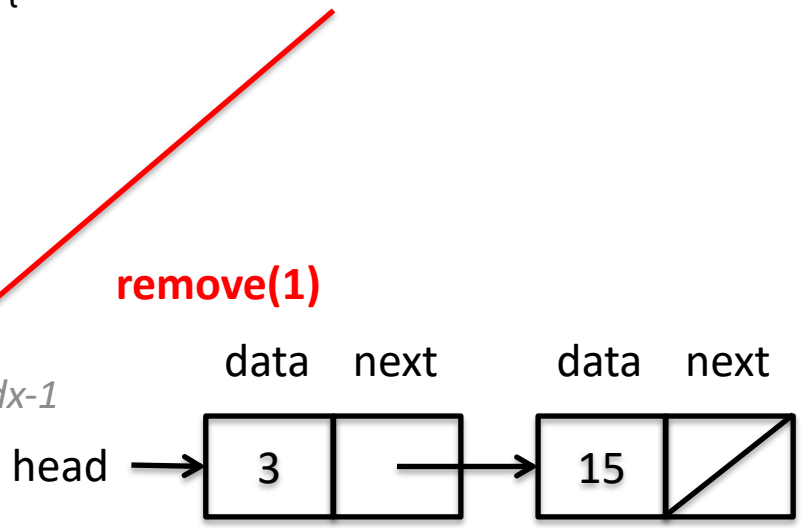


- **What happens to the old head element?**
- **Garbage collected! (memory returned to the Operating System)**

If removing NOT at head, advance to idx-1

SinglyLinked.java

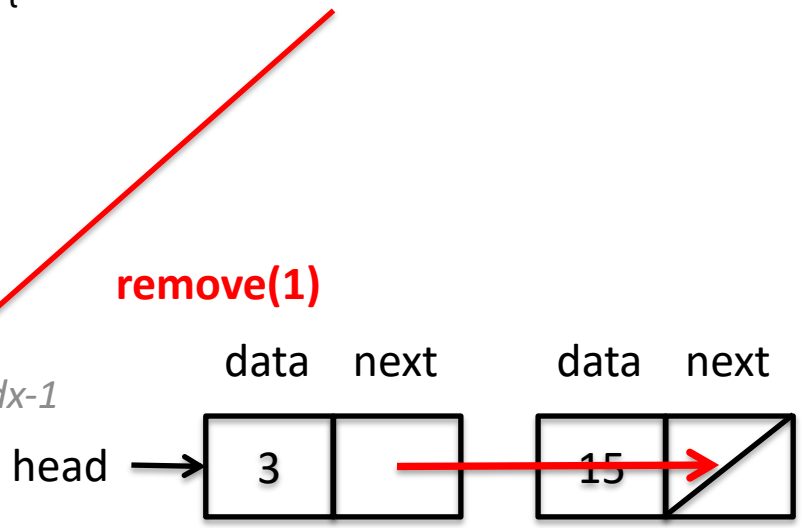
```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```



Set idx-1 Element next to point to next.next

SinglyLinked.java

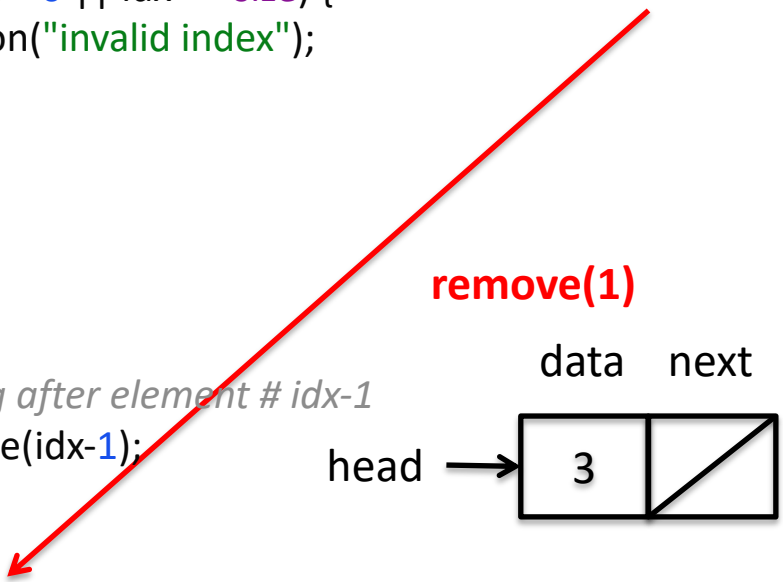
```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```



Set idx-1 Element next to point to next.next

SinglyLinked.java

```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```



get and *set* are straightforward with *advance* method

SinglyLinked.java

```
public T get(int idx) throws Exception {
   //safety check for valid index
   if (idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   Element e = advance(idx);
   return e.data;
}

public void set(int idx, T item) throws Exception {
   //safety check for valid index
   if (idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   Element e = advance(idx);
   e.data = item;
}
```

- ***Run-time complexity?***
- ***O(n)***

toString returns a String representation of the List (doesn't print!)

SinglyLinked.java

```
public String toString() {
   String result = "";
   for (Element x = head; x != null; x = x.next)
      result += x.data + "->";
   result += "[/]";

   return result;
}
```

Run-time complexity
Θ(n)

On an exam: make sure you return a String with toString(), don't print in toString()

ListTest.java: Test of List implementation

ListTest.java

```
public class ListTest {
   public static void main(String[] args) throws Exception {
     SimpleList<String> list = new SinglyLinked<String>();
     System.out.println(list);
     list.add("1"); System.out.println(list);
     list.add("2"); System.out.println(list);
     list.add(0, "a"); System.out.println(list);
     list.add(1, "c"); System.out.println(list);
     list.add(1, "b"); System.out.println(list);
     list.set(2, "e"); System.out.println(list.get(2));
     list.add(0, "z"); System.out.println(list);
     String data = list.remove(2); System.out.println(data);
     System.out.println(list);
     data = list.remove(0); System.out.println(data);
     System.out.println(list);
     data = list.remove(1); System.out.println(data);
     System.out.println(list);
     data = list.remove(list.size()-1); System.out.println(list);
```

Output

```
[/]
1->[/]
1->2->[/]
a->1->2->[/]
a->c->1->2->[/]
a->b->c->1->2->[/]
e
z->a->b->e->1->2->[/]
b
z->a->e->1->2->[/]
z
a->e->1->2->[/]
e
a->1->2->[/]
a->1->[/]
```

Summary of SinglyLinked run-time complexity

Run-time complexity

	Linked list
get(i)	?
set(i,e)	?
add(i,e)	?
remove(i)	?

Summary of SinglyLinked run-time complexity

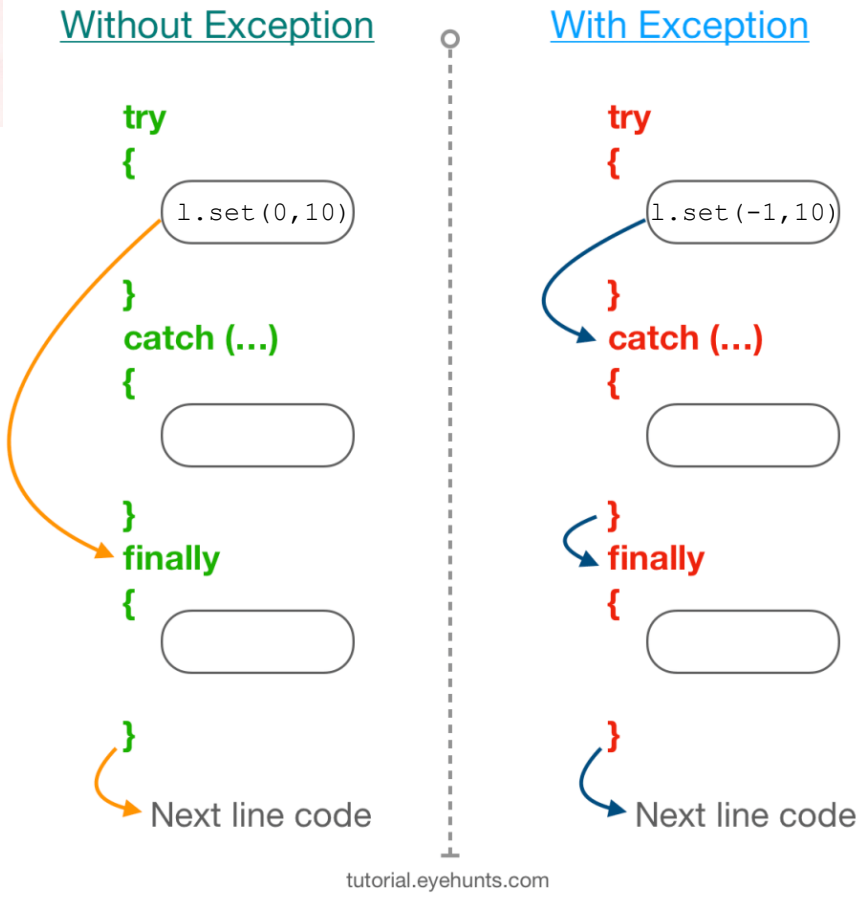
Run-time complexity

	Linked list
get(i)	O(n)
set(i,e)	O(n)
add(i,e)	O(n)
remove(i)	O(n)

Agenda

1. Singly linked list implementation
2. Exceptions
3. Iterators

An exception indicates that something unexpected happened at run-time



e.g.,

```
public void set(int idx, T item) throws Exception {
    if (idx < 0) {
        throw new Exception("invalid index");
    }
    Element e = advance(idx);
    e.data = item;
}
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```java
public class ListExceptions {
    public static void main(String[] args) { // note: no "throws exception", as every method that could is

        SimpleList<String> list = new SinglyLinked<String>();

        try {
            list.add(-1, "?");
            System.out.println("I never run!");
            System.out.println("Neither do I");
        }
        catch (Exception e) {
            System.out.println("caught it!"); // will print -- we know this is bogus
        }

        try {
            list.add(-1, "?");
            System.out.println("Do I run?");
            System.out.println("No I don't");
        }
        catch (Exception e) {
            System.out.println("caught it again!"); // will print -- we know this is bogus
            System.out.println(e);                 //will give us the error message
        }
        finally {
            System.out.println("finally 1"); // executed whether or not caught an error
        }

        try {
            list.add(0, "?");
            System.out.println(list);
        }
        catch (Exception e) {
            System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
        }
        finally {
            System.out.println("finally 2"); // executed whether or not caught an error
        }
}
}

```

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

Agenda

1. Singly linked list implementation
2. Exceptions
3. Iterators

What is wrong with this code?

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

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

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

Instantiate SinglyLinked list of Integers

Add 1,000 Integer to List

Print each item in List

Works as intended, but slow

$O(n^2)$ – sneaky inefficiency

Why?

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

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

SinglyLinked.java

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

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

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

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

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

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

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


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

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

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

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

   public ListIterator() {
     curr = head;
   }

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

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

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

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

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

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

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

Output

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

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

Summary

- Singly linked list implementation of ADT SimpleList
  - Marching down the list -> O(n)
- Exceptions for passing an error so that the caller can handle them
- Iterators for efficient iterations over elements

Next

- Array lists

Additional Resources

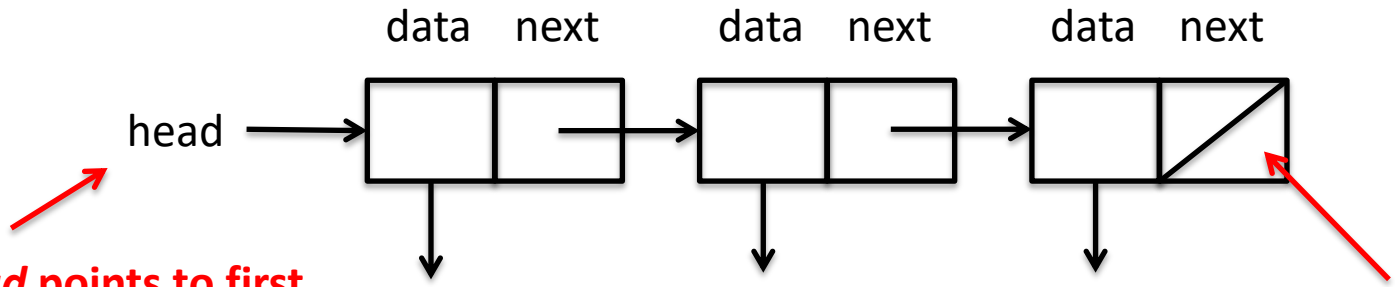
SINGLY LINKED LIST VISUALIZATION

Singly linked list review: elements have data and a next pointer

Singly linked list

"Box-and-pointer" diagram

- **Data in Box**
- **Pointer to next item in List**



***head* points to first item (index 0)**

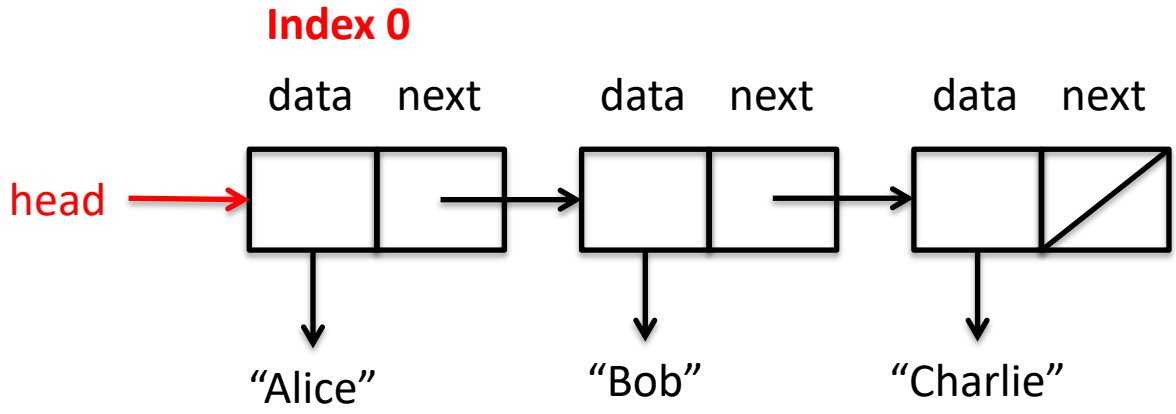
***null* if list is empty**

Slash indicates end of List

next* pointer is *null

To get an item at index i, start at head and march down

get(i) – return item at specified index

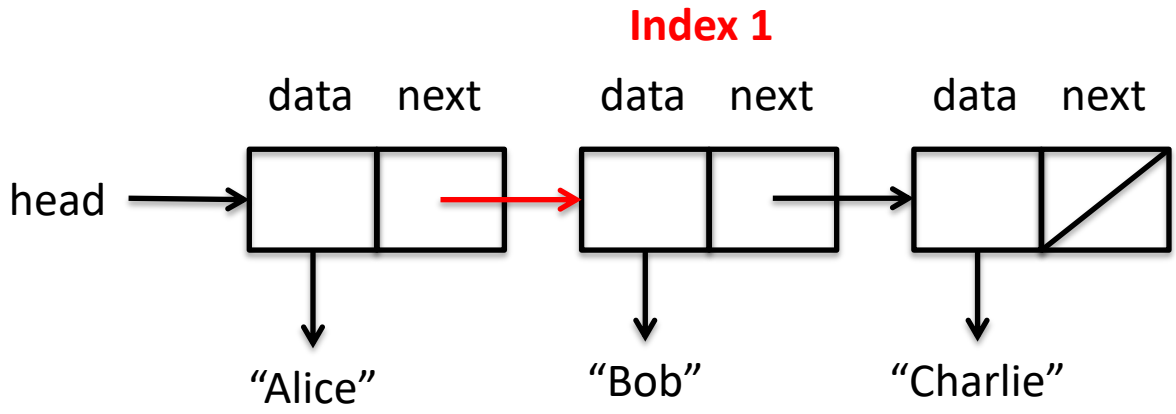


Get item at index 2

1. **Start at head (index 0)**

To get an item at index i, start at head and march down

get(i) – return item at specified index

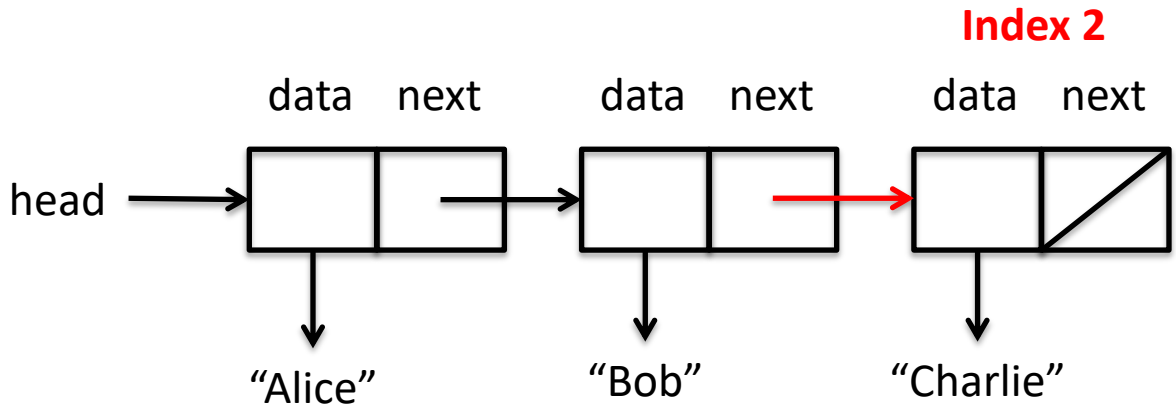


Get item at index 2

1. **Start at head (index 0)**
2. **Follow next pointer to index 1**

To get an item at index i, start at head and march down

get(i) – return item at specified index

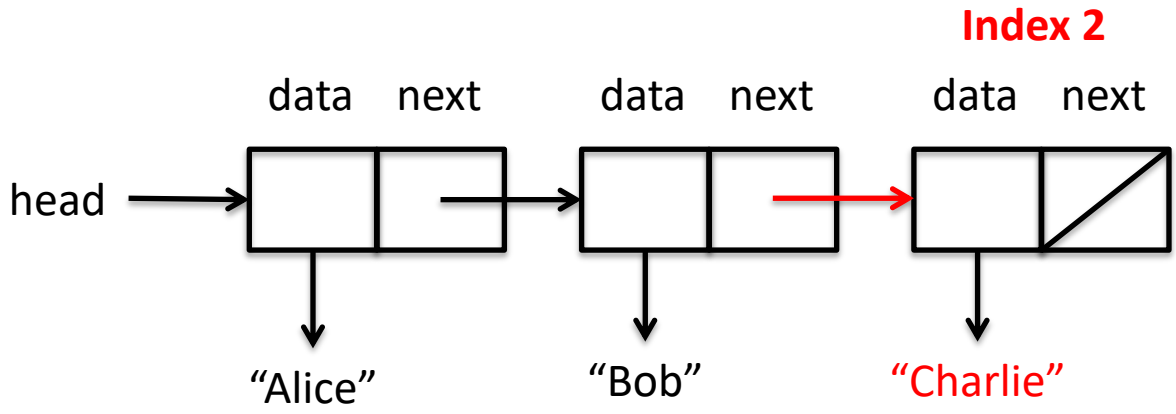


Get item at index 2

1. **Start at head (index 0)**
2. **Follow next pointer to index 1**
3. **Follow next pointer to index 2**

To get an item at index i, start at head and march down

get(i) – return item at specified index

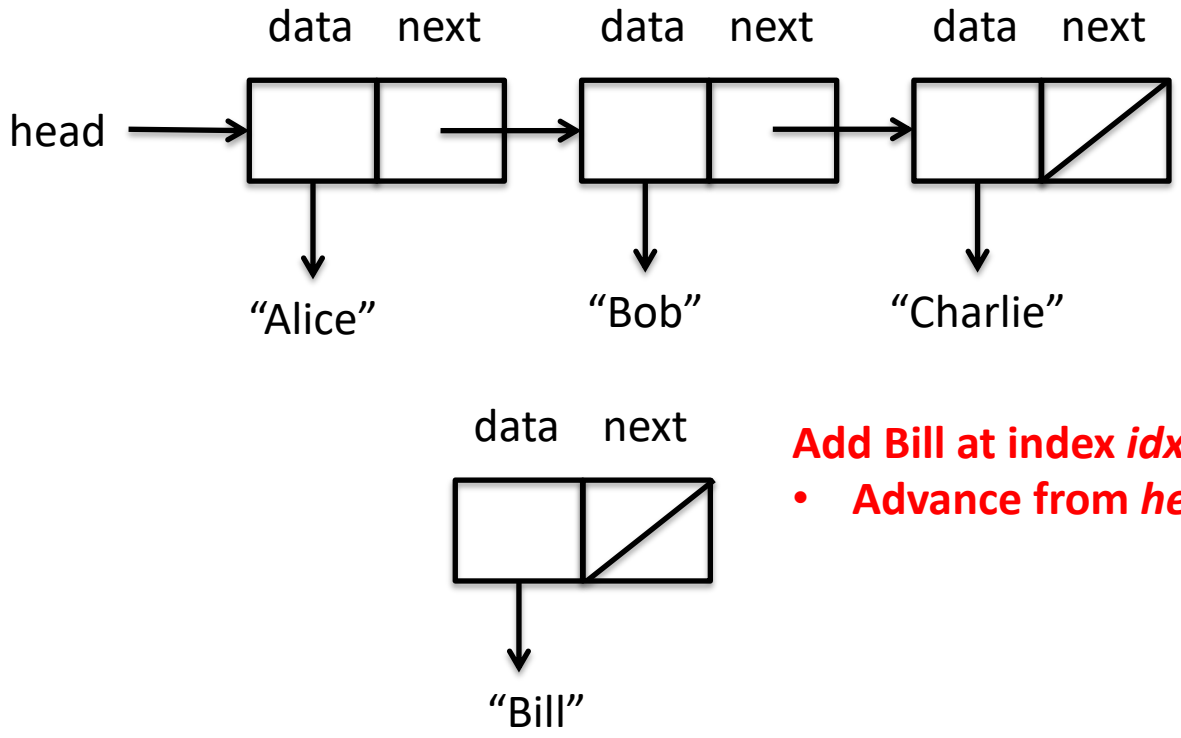


Get item at index 2

1. **Start at head (index 0)**
2. **Follow next pointer to index 1**
3. **Follow next pointer to index 2**
4. **Return "Charlie" (data of item 2)**

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")

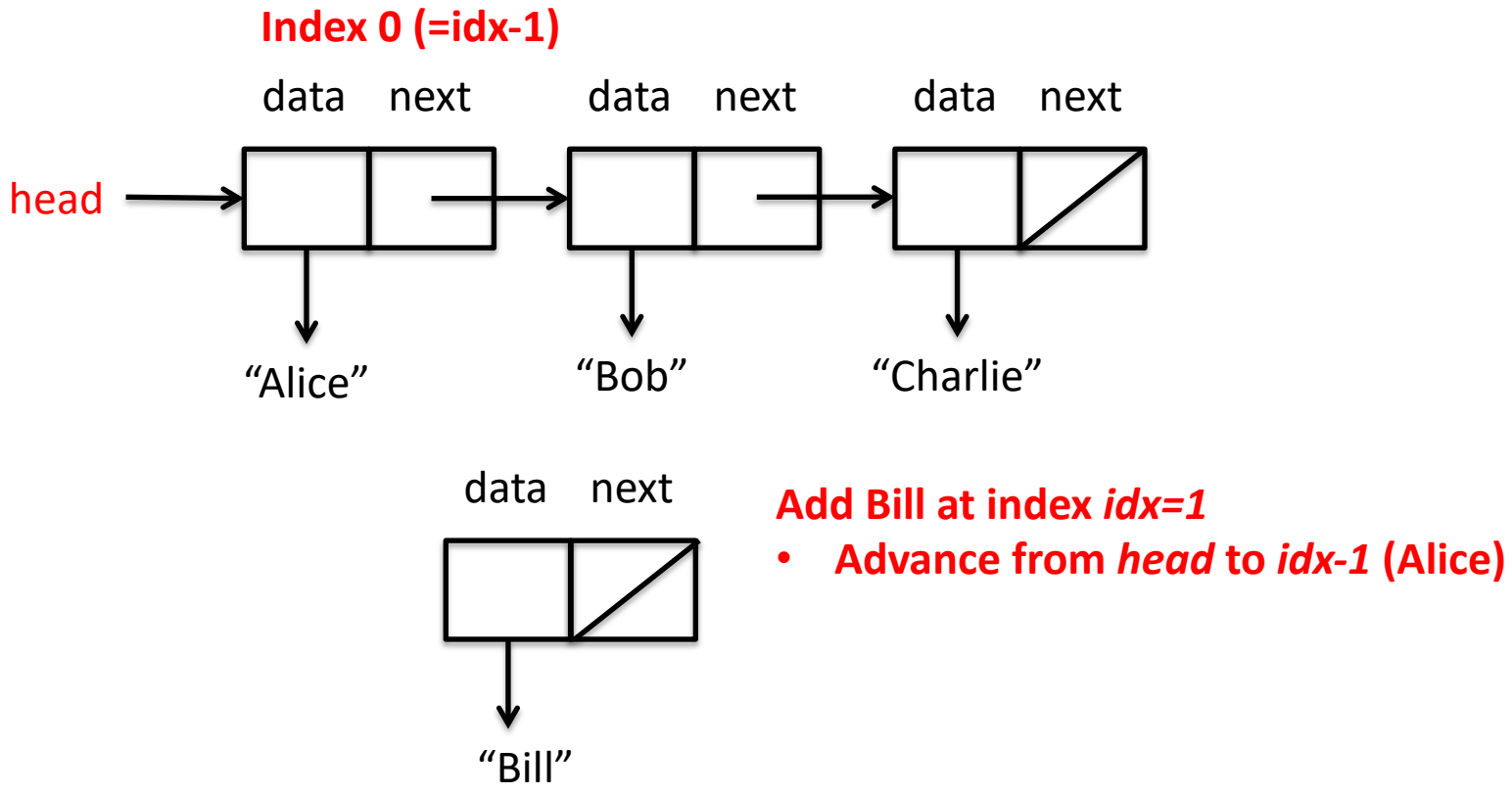


Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")

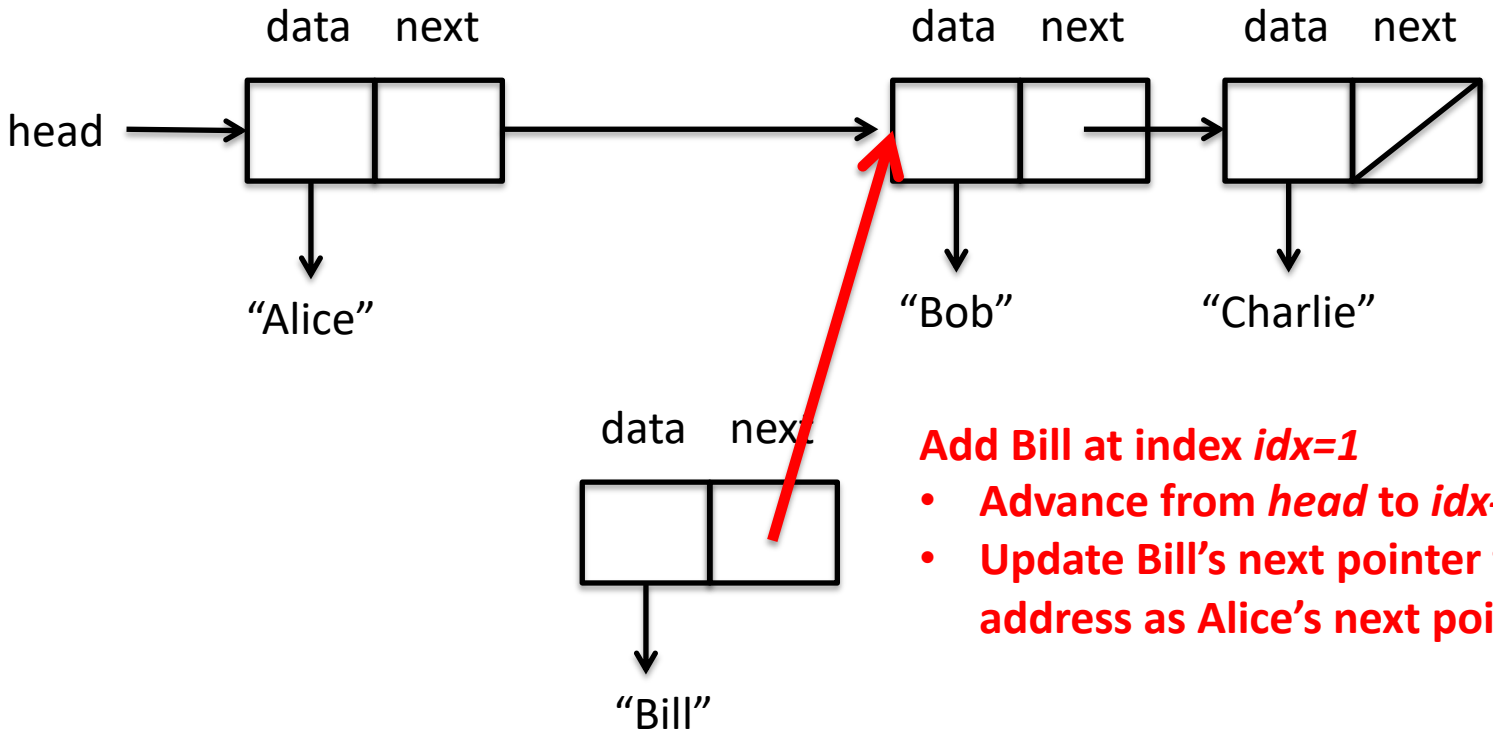


Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")

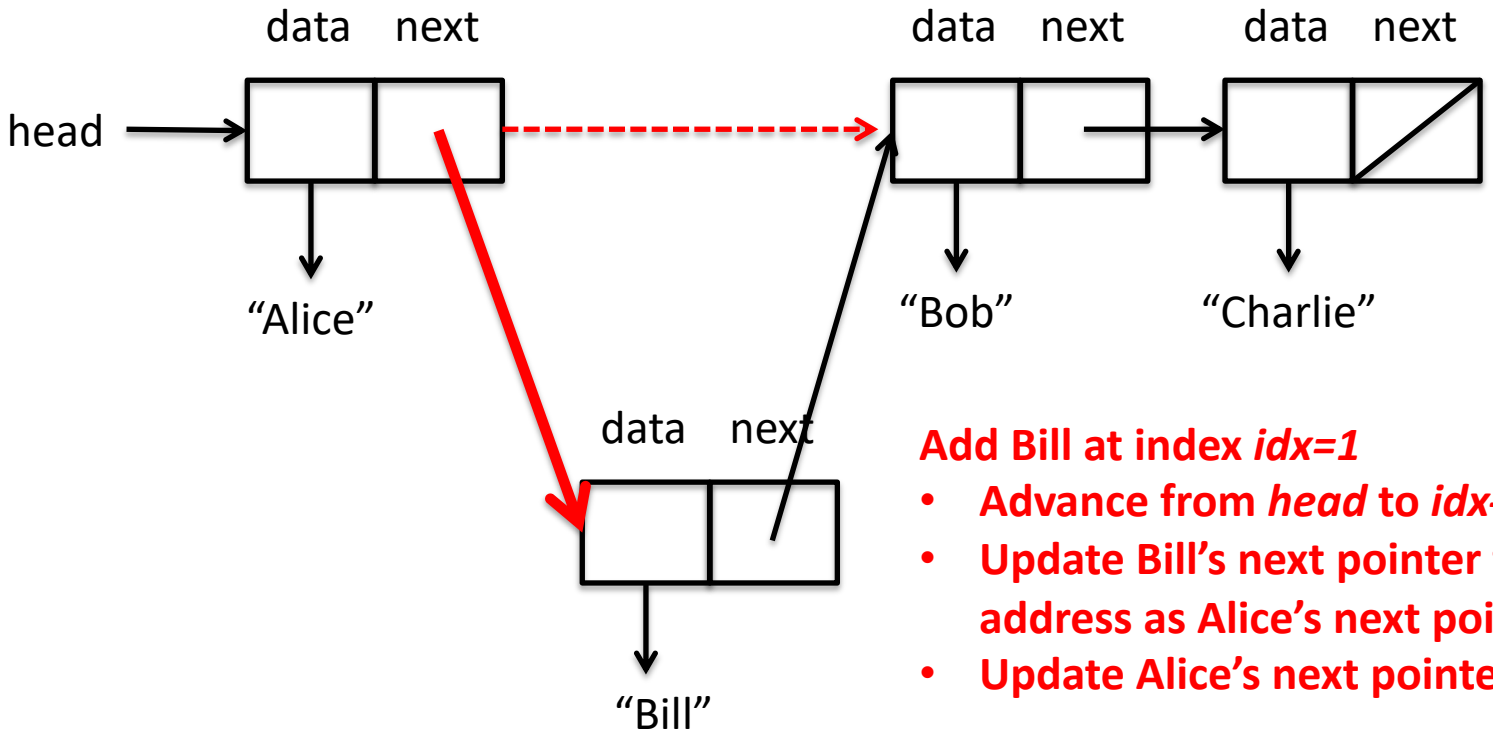


Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**
- **Update Bill's next pointer to same address as Alice's next pointer**

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")

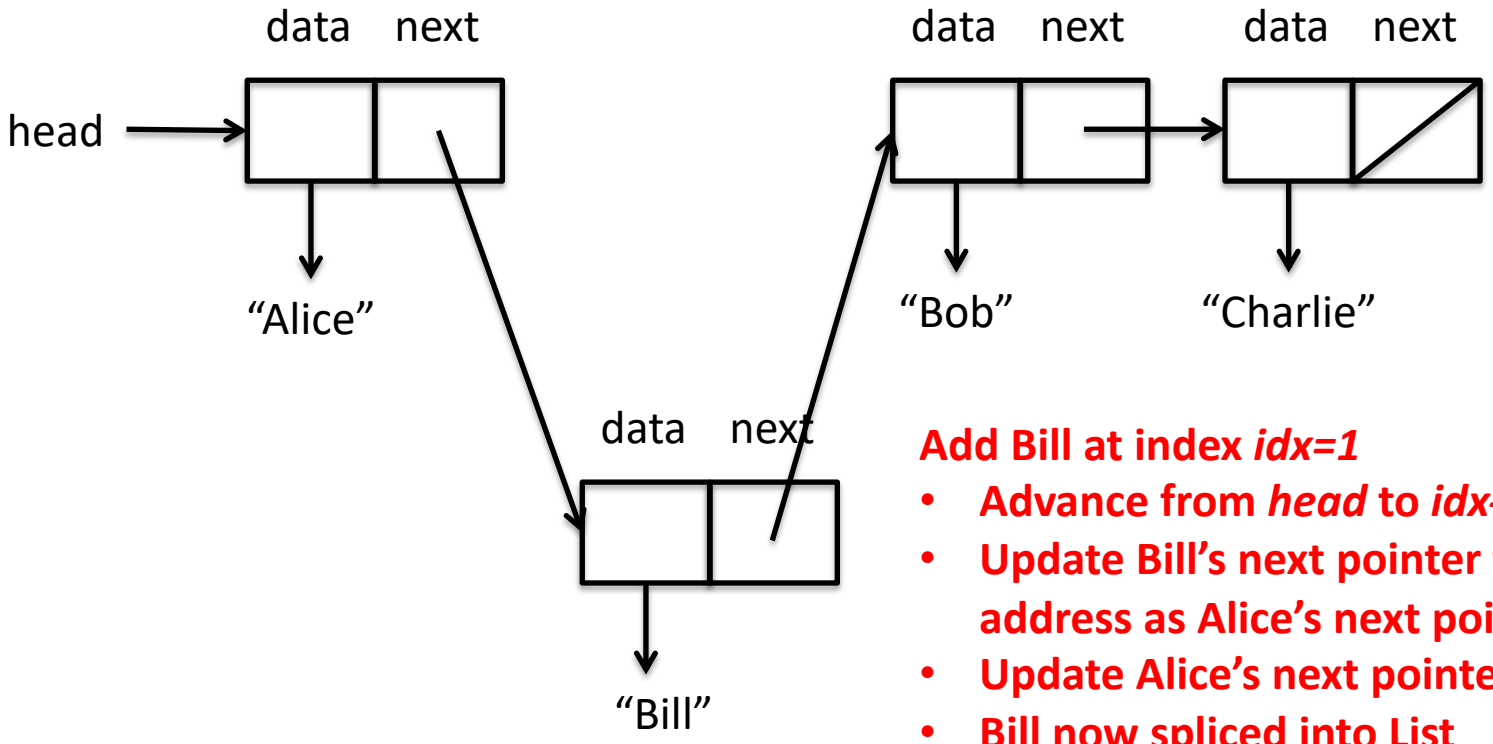


Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**
- **Update Bill's next pointer to same address as Alice's next pointer**
- **Update Alice's next pointer to Bill**

add() "splices in" a new object anywhere in the list by updating next pointers

add(1, "Bill")

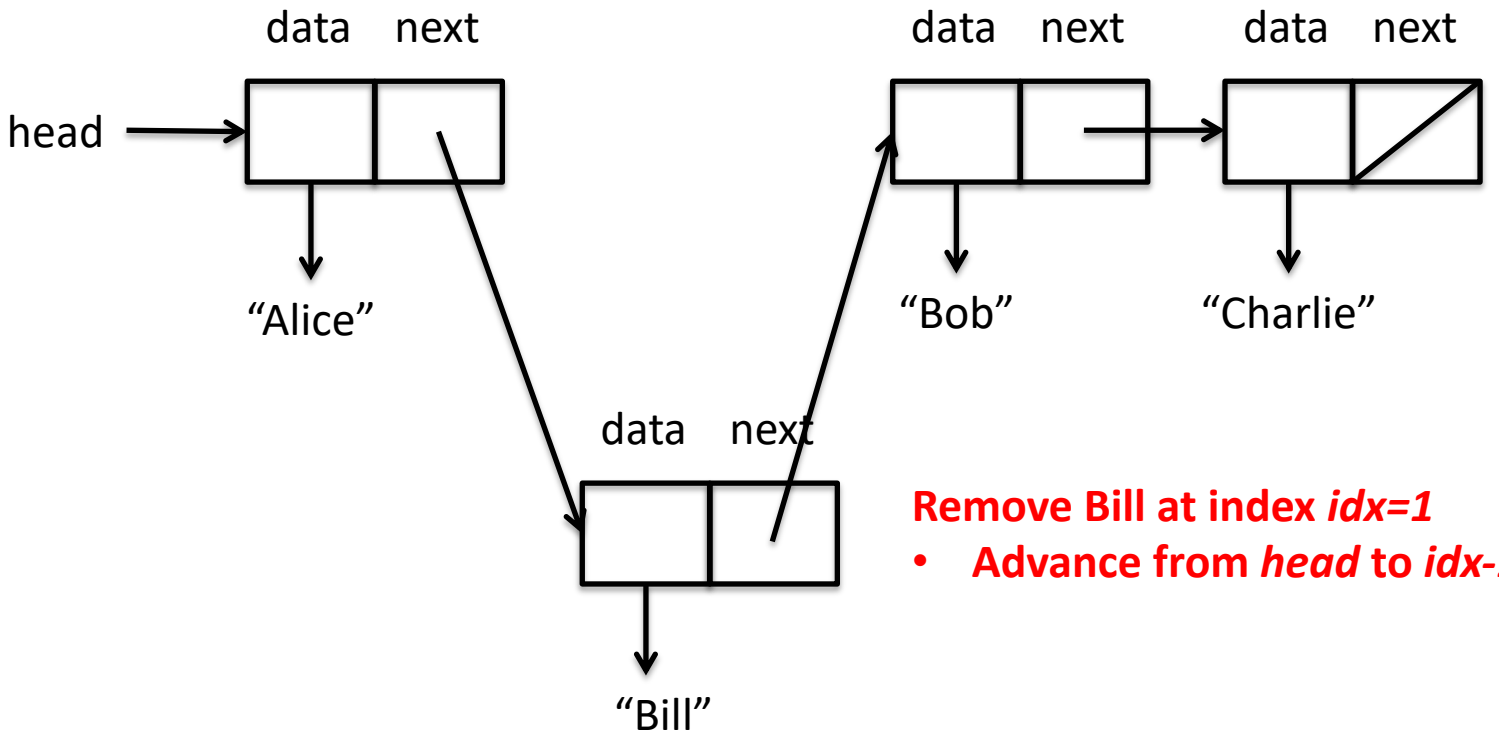


Add Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**
- **Update Bill's next pointer to same address as Alice's next pointer**
- **Update Alice's next pointer to Bill**
- **Bill now spliced into List**
- **Once find idx-1, only two pointer updates needed**

remove() takes an item out of the list by updating next pointer

remove(1)

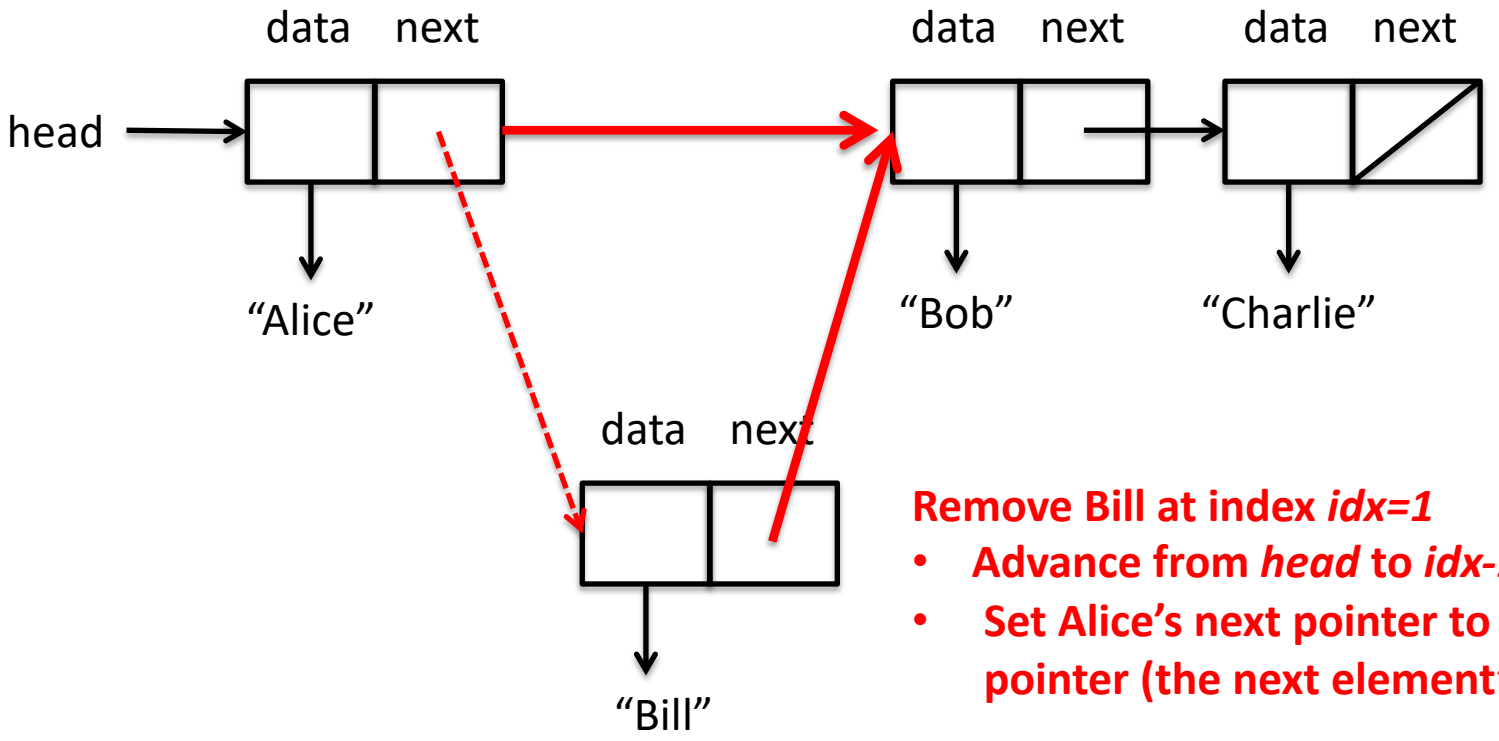


Remove Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**

remove() takes an item out of the list by updating next pointer

remove(1)

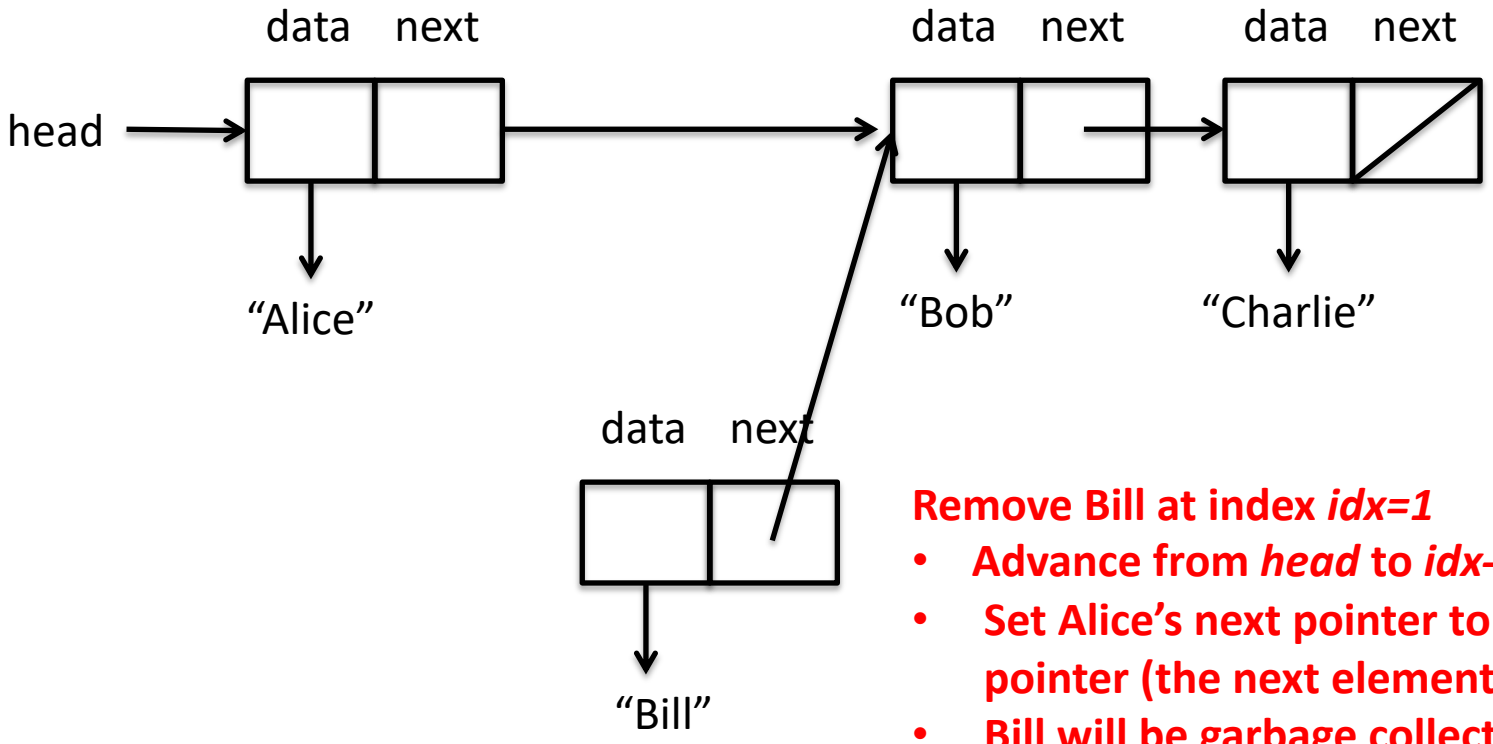


Remove Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**
- **Set Alice's next pointer to Bill's next pointer (the next element's next)**

remove() takes an item out of the list by updating next pointer

remove(1)



Remove Bill at index *idx=1*

- **Advance from *head* to *idx-1* (Alice)**
- **Set Alice's next pointer to Bill's next pointer (the next element's next)**
- **Bill will be garbage collected (in C we have to call *free()*)**

SinglyLinked.java

ANNOTATED SLIDES

SinglyLinked.java: Implementation of List interface

SinglyLinked.java

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

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

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

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

We will deal with Iterable soon, standby for more info, but can implement multiple interfaces

"implements" is a promise to implement all required methods specified by Interface SimpleList

- ***size()***
- ***isEmpty()***
- ***add()***
- ***remove()***
- ***get()***
- ***set()***

Lists hold items of generic type

SinglyLinked.java

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

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

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

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

- **Type of data is generic T**
- **Don't care what kind of data the List holds, could be Strings, Integers, Student Objects,…**
- **This way we don't have to write a separate implementation if use Strings as elements, and other implementation if use Integers, and third implementation if use …**
- **Just implement the List once and hold whatever data type needed for the application**

Implement a private "nested" class to hold data and next pointer

SinglyLinked.java

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

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

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

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

- **Define a private class within SinglyLinked called *Element* to implement *data* and *next* pointers (could be in its own file)**
- ***Element* constructor takes *data* as type T and pointer to next *Element* (could be null)**
- ***Element* is private to SinglyLinked (internal to this file, no need for others to change it)**

Set head to null and size to zero in constructor

SinglyLinked.java

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

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

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

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

- **Creates *head* Element and *size* counter**
- **Constructor initializes *head* to null and *size* to 0**
- **Notice *head* is of type Element but is never "newed"**
- ***head* will be a pointer to first *Element* in the List**

Increment size instance variable on add, decrement on remove operation

SinglyLinked.java

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

/**
 * Returns true if there are no elements in the List, false otherwise
 * @return true or false
 */
public boolean isEmpty() {

}
```

- ***size()* method just returns instance variable *size***
- ***size* will be incremented on *add()*, decremented on *remove()***
- **Run-time complexity?**
- **O(1)**

- **How can *isEmpty()* be easily implemented?**

Implementing *isEmpty* "isEasy" ☺

SinglyLinked.java

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

/**
 * Returns true if there are no elements in the List, false otherwise
 * @return true or false
 */
public boolean isEmpty() {
   return size == 0;
}
```

- **How can *isEmpty()* be easily implemented?**
- **Check if *size == 0***
- **Run-time complexity?**
- **O(1)**

advance is a helper method to move to the n^{th} item in the List

SinglyLinked.java

```
/**
 * Helper function, advancing to the nth Element in the list and returning it
 * (exception if not that many elements)
 */
private Element advance(int n) throws Exception {
  Element e = head;
  //safety check for valid index (don't assume caller checked!)
  if (e == null || n < 0 || n >= size) {
    throw new Exception("invalid advance");
  }

  // Just follow the next pointers n times
  for (int i = 0; i < n; i++) {
    e = e.next;
  }
  return e;
}
```

Key point: to get to an index in a SinglyLinked List, must always start at *head* and march down List!

- **advance() helper method**
- **Start at *head and* marches down *n* items (*e* not new'ed)**
- **Loop until hit n^{th} item**
- **Return n^{th} item (or throw exception)**
- **Note: return type from *advance()* is *Element***
- ***advance()* not specified by interface, but implementations can have more methods than required**
- **Do not assume caller checked for valid index!**
- **Run-time complexity?**
- **O(n)**

add method uses *advance*

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

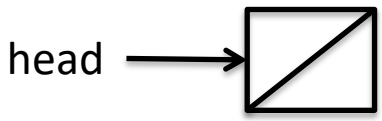
***add()/remove()* use *advance()* to march down list to item before index *idx*, then adjust pointers**

add method uses *advance*

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

Safety check for valid index



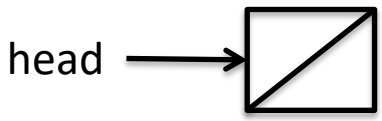
No need to advance if adding at the head

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

If adding at head (index 0)

add(0,15)



Just create a new Element and point head to it

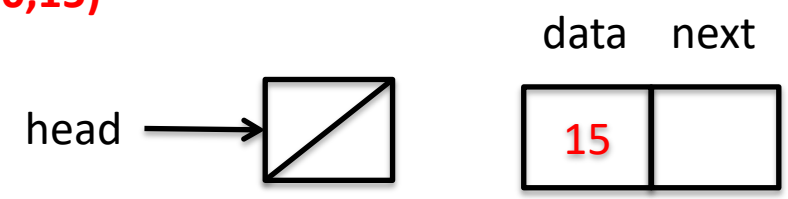
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***

add(0,15)



Just create a new Element and point head to it

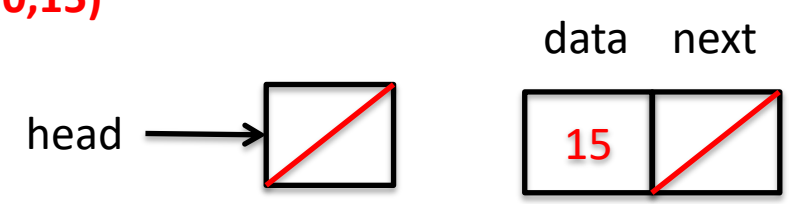
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**

add(0,15)



Just create a new Element and point head to it

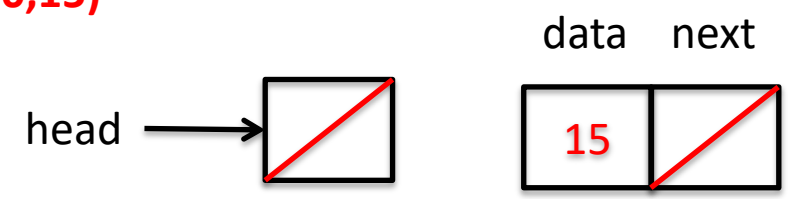
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**
- ***head* will initially point to null**

add(0,15)



Just create a new Element and point head to it

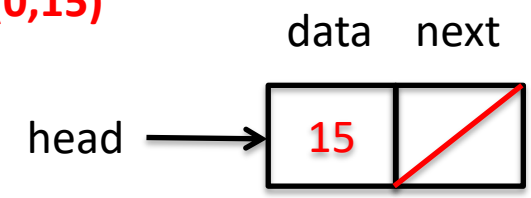
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**
- ***head* will initially point to null**
- **Set *head* to new element**

add(0,15)



Don’t forget to increment *size*

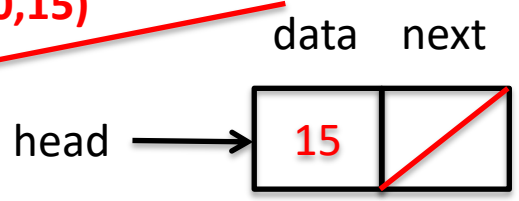
```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**
- ***head* will initially point to null**
- **Set *head* to new element**
- **Finally increment *size***

add(0,15)

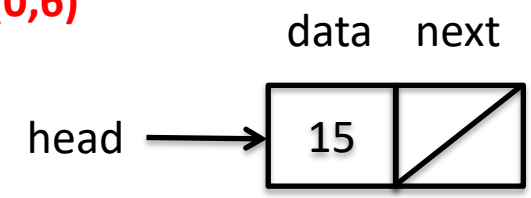


Adding at head if the List is not empty is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

add(0,6)



Adding at head if the List is not empty is easy

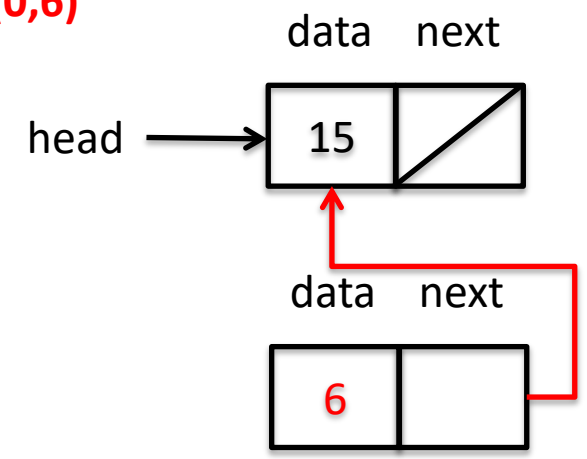
```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**

add(0,6)



Adding at head if the List is not empty is easy

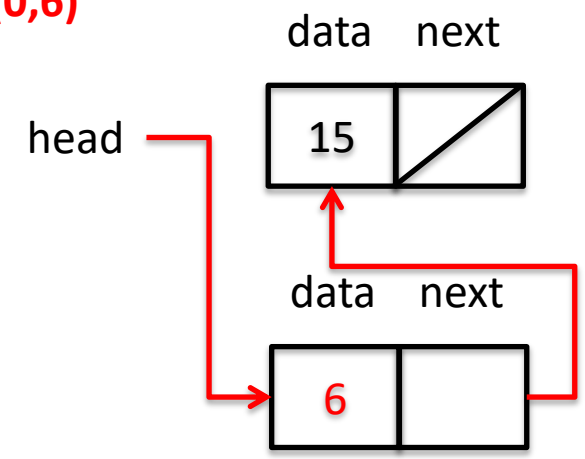
```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

SinglyLinked.java

If adding at head (index 0)

- **Create new element with data set to parameter *item***
- **Set new element next pointer to wherever *head* points**
- **Set *head* to new element**
- **Finally increment *size***

add(0,6)



If adding NOT at head, use *advance* method

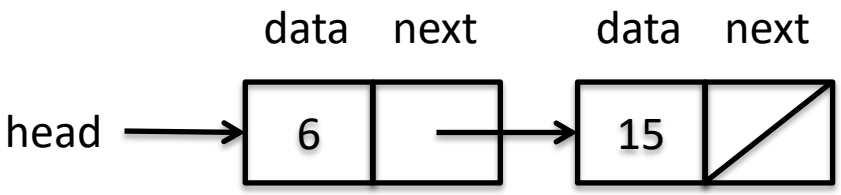
SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

If adding not at head

- ***advance*() to *e=(idx-1)th* item**

add(1,3)



Move to index idx-1

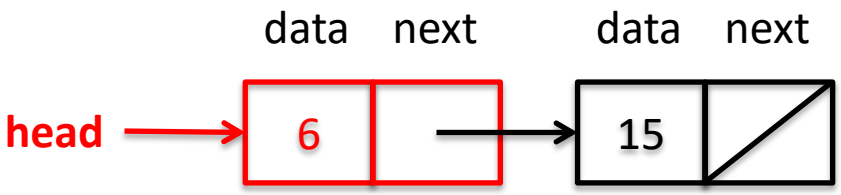
SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

If adding not at head

- ***advance*() to *e*=(*idx-1*)th item**

add(1,3)



Advance to item idx-1
Here (index 1)-1 = index 0
So *e* points to index 0 with data = 6

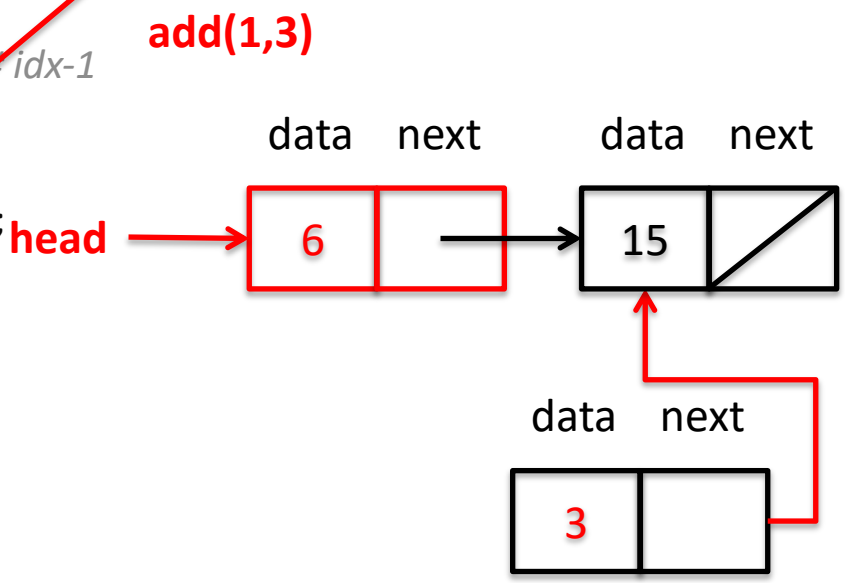
Splice in a new Element that points to where Element at idx-1 points

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

If adding not at head

- ***advance*() to *e=(idx-1)th* item**
- **Create new *Element* with *data* set to *item* and *next* to *e.next***



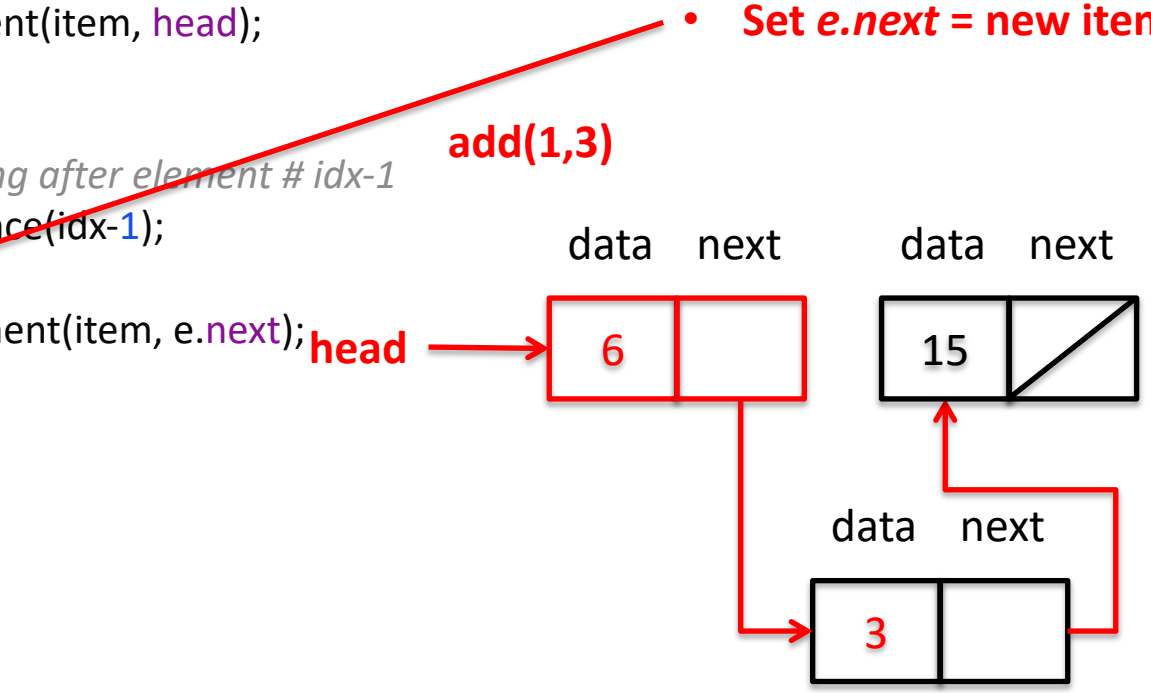
Set idx-1 to point to new Element

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```

If adding not at head

- ***advance*() to *e=(idx-1)th* item**
- **Create new *Element* with *data* set to *item* and *next* to *e.next***
- **Set *e.next* = new item**



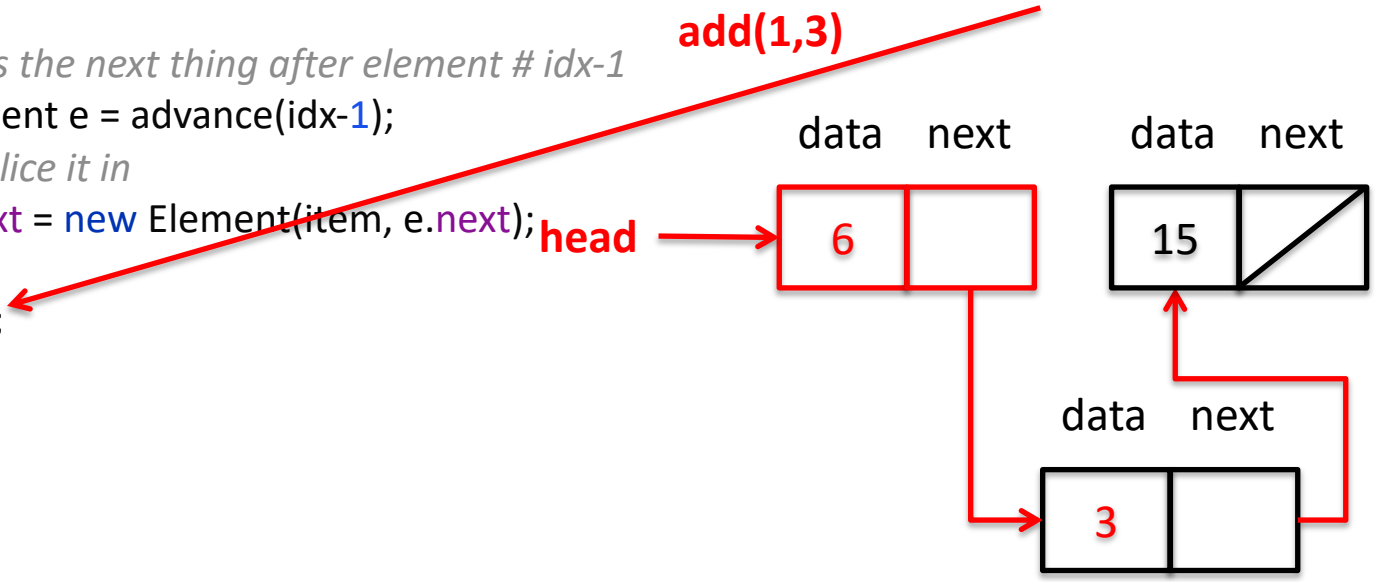
Don’t forget to increment *size*

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
  //safety check for valid index (can add at size index)
  if (idx < 0 || idx > size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    // Insert at head
    head = new Element(item, head);
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    // Splice it in
    e.next = new Element(item, e.next);
  }
  size++;
}
```

If adding not at head

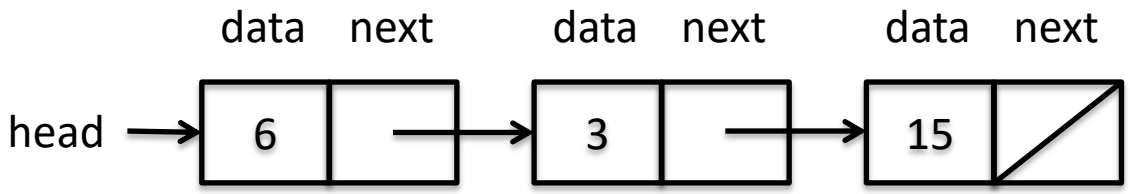
- ***advance*() to *e=(idx-1)th* item**
- **Create new *Element* with *data* set to *item* and *next* to *e.next***
- **Set *e.next* = new item**
- **Finally, increment *size***



Adding at the end is easy

SinglyLinked.java

```
public void add(int idx, T item) throws Exception {
   //safety check for valid index (can add at size index)
   if (idx < 0 || idx > size) {
      throw new Exception("invalid index");
   }
   else if (idx == 0) {
      // Insert at head
      head = new Element(item, head);
   }
   else {
      // It's the next thing after element # idx-1
      Element e = advance(idx-1);
      // Splice it in
      e.next = new Element(item, e.next);
   }
   size++;
}
```




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

How to easily add at the end?
Just call add with size as index

On SA-4 you'll do something more efficient using a tail pointer

remove method removes and returns data at *idx*

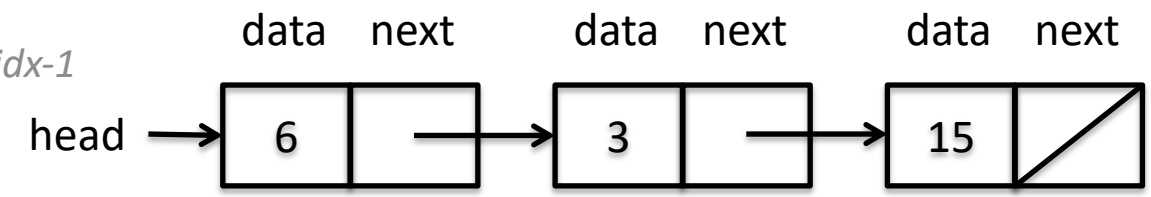
SinglyLinked.java

```
public T remove(int idx) throws Exception {
  T data = null; //data to return
  //safety check for valid index
  if (head == null || idx < 0 || idx >= size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    data = head.data;
    head = head.next;
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    data = e.next.data;
    // Splice it out
    e.next = e.next.next;  //nice!
  }
  size--;
  return data;
}
```

If removing at *head*

- **Save data at head**
- **Set *head* to next**

remove(0)



If removing at head, just set head to head.next

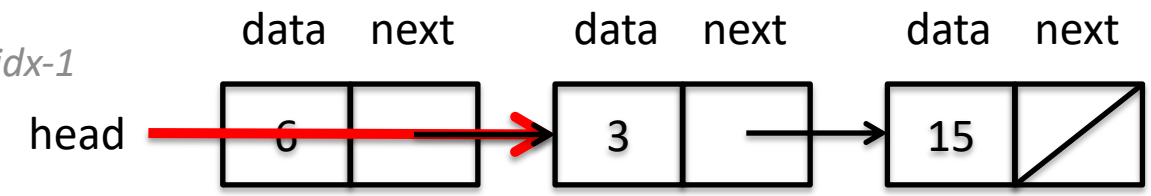
SinglyLinked.java

```
public T remove(int idx) throws Exception {
  T data = null; //data to return
  //safety check for valid index
  if (head == null || idx < 0 || idx >= size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    data = head.data;
    head = head.next;
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    data = e.next.data;
    // Splice it out
    e.next = e.next.next;  //nice!
  }
  size--;
  return data;
}
```

If removing at *head*

- **Save data at head**
- **Set *head* to next**

remove(0)



- **What happens to the old head element?**
- **Garbage collected! (memory returned to the Operating System)**

If removing at head, just set head to head.next

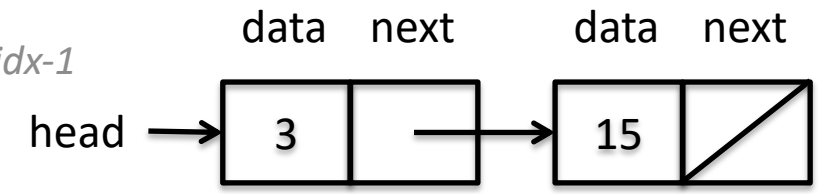
SinglyLinked.java

```
public T remove(int idx) throws Exception {
  T data = null; //data to return
  //safety check for valid index
  if (head == null || idx < 0 || idx >= size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    data = head.data;
    head = head.next;
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    data = e.next.data;
    // Splice it out
    e.next = e.next.next;  //nice!
  }
  size--;
  return data;
}
```

If removing at *head*

- **Save data at head**
- **Set *head* to next**

remove(0)



- **What happens to the old head element?**
- **Garbage collected! (memory returned to the Operating System)**

If removing NOT at head, advance to idx-1

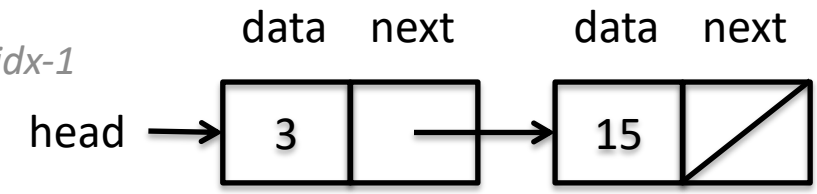
SinglyLinked.java

```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```

If removing not at *head*

- ***advance()* to *idx-1* (data 3 here)**
- **Save data at idx**
- **Set *e.next* to *e.next.next***

remove(1)



Set idx-1 Element next to point to next.next

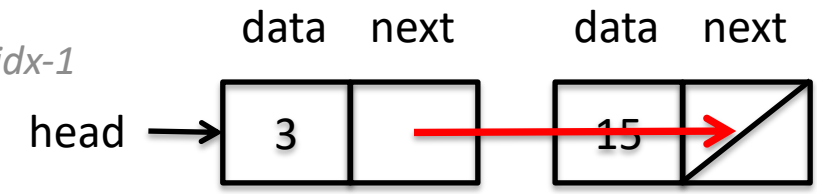
SinglyLinked.java

```
public T remove(int idx) throws Exception {
   T data = null; //data to return
   //safety check for valid index
   if (head == null || idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   else if (idx == 0) {
     data = head.data;
     head = head.next;
   }
   else {
     // It's the next thing after element # idx-1
     Element e = advance(idx-1);
     data = e.next.data;
     // Splice it out
     e.next = e.next.next;  //nice!
   }
   size--;
   return data;
}
```

If removing not at *head*

- ***advance()* to *idx-1 (data 3 here)***
- **Save data at idx**
- **Set *e.next* to *e.next.next***

remove(1)



Set idx-1 Element next to point to next.next

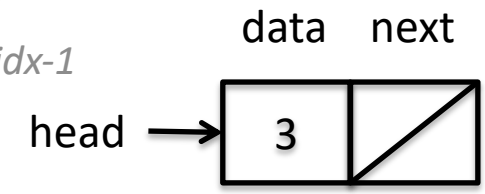
SinglyLinked.java

```
public T remove(int idx) throws Exception {
  T data = null; //data to return
  //safety check for valid index
  if (head == null || idx < 0 || idx >= size) {
    throw new Exception("invalid index");
  }
  else if (idx == 0) {
    data = head.data;
    head = head.next;
  }
  else {
    // It's the next thing after element # idx-1
    Element e = advance(idx-1);
    data = e.next.data;
    // Splice it out
    e.next = e.next.next;  //nice!
  }
  size--;
  return data;
}
```

If removing not at *head*

- ***advance()* to *idx-1* (data 3 here)**
- **Save data at idx**
- **Set *e.next* to *e.next.next***

remove(1)



get and *set* are straightforward with *advance* method

SinglyLinked.java

```
public T get(int idx) throws Exception {
   //safety check for valid index
   if (idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   Element e = advance(idx);
   return e.data;
}

public void set(int idx, T item) throws Exception {
   //safety check for valid index
   if (idx < 0 || idx >= size) {
     throw new Exception("invalid index");
   }
   Element e = advance(idx);
   e.data = item;
}
```

- ***get()/set()* also use *advance()* to march down list, this time to index *idx***
- ***Run-time complexity?***
- ***O(n)***

toString returns a String representation of the List (doesn't print!)

SinglyLinked.java

Key point: all operations start at head and march down list

```
public String toString() {
   String result = "";
   for (Element x = head; x != null; x = x.next)
      result += x.data + "->";
   result += "[/]";

   return result;
}
```

On an exam: make sure you return a String with toString(), don't print in toString()

***toString()* overrides a Java Object method and allows us to create a string representation of the object**

If *toString()* not overriden, defaults to the memory address of object

Return type is String, if used in print, doesn't actually do the printing

Run-time complexity Θ(n)

Summary of SinglyLinked run-time complexity

Run-time complexity

	Linked list
get(i)	O(n)
set(i,e)	O(n)
add(i,e)	O(n)
remove(i)	O(n)

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

ListTest.java

ANNOTATED SLIDES

ListTest.java: Test of List implementation

ListTest.java

```
public class ListTest {
  public static void main(String[] args) throws Exception {
    SimpleList<String> list = new SinglyLinked<String>();
    System.out.println(list);
    list.add("1"); System.out.println(list);
    list.add("2"); System.out.println(list);
    list.add(0, "a"); System.out.println(list);
    list.add(1, "c"); System.out.println(list);
    list.add(1, "b"); System.out.println(list);
    list.set(2, "e"); System.out.println(list.get(2));
    list.add(0, "z"); System.out.println(list);
    String data = list.remove(2); System.out.println(data);
    System.out.println(list);
    data = list.remove(0); System.out.println(data);
    System.out.println(list);
    data = list.remove(1); System.out.println(data);
    System.out.println(list);
    data = list.remove(list.size()-1); System.out.println(list);
```

Declare SinglyLinked List to hold Strings, so T = String in the implementation

Implementation is SinglyLinked which implemented SimpleList interface

Next class we'll look at an array implementation which will also be a SimpleList

Output

```
[/]
1->[/]
1->2->[/]
a->1->2->[/]
a->c->1->2->[/]
a->b->c->1->2->[/]
e
z->a->b->e->1->2->[/]
b
z->a->e->1->2->[/]
z
a->e->1->2->[/]
e
a->1->2->[/]
a->1->[/]
```

ListTest.java: Test of List implementation

ListTest.java

```
public class ListTest {
  public static void main(String[] args) throws Exception {
    SimpleList<String> list = new SinglyLinked<String>();
    System.out.println(list);
    list.add("1"); System.out.println(list);
    list.add("2"); System.out.println(list);
    list.add(0, "a"); System.out.println(list);
    list.add(1, "c"); System.out.println(list);
    list.add(1, "b"); System.out.println(list);
    list.set(2, "e"); System.out.println(list.get(2));
    list.add(0, "z"); System.out.println(list);
    String data = list.remove(2); System.out.println(data);
    System.out.println(list);
    data = list.remove(0); System.out.println(data);
    System.out.println(list);
    data = list.remove(1); System.out.println(data);
    System.out.println(list);
    data = list.remove(list.size()-1); System.out.println(list);
```

***toString()* method called in print statements**

Output

```
[/]
1->[/]
1->2->[/]
a->1->2->[/]
a->c->1->2->[/]
a->b->c->1->2->[/]
e
z->a->b->e->1->2->[/]
b
z->a->e->1->2->[/]
z
a->e->1->2->[/]
e
a->1->2->[/]
a->1->[/]
```

ListTest.java: Test of List implementation

ListTest.java

```
public class ListTest {
  public static void main(String[] args) throws Exception {
    SimpleList<String> list = new SinglyLinked<String>();
    System.out.println(list);
    list.add("1"); System.out.println(list);
    list.add("2"); System.out.println(list);
    list.add(0, "a"); System.out.println(list);
    list.add(1, "c"); System.out.println(list);
    list.add(1, "b"); System.out.println(list);
    list.set(2, "e"); System.out.println(list.get(2));
    list.add(0, "z"); System.out.println(list);
    String data = list.remove(2); System.out.println(data);
    System.out.println(list);
    data = list.remove(0); System.out.println(data);
    System.out.println(list);
    data = list.remove(1); System.out.println(data);
    System.out.println(list);
    data = list.remove(list.size()-1); System.out.println(list);
```

***toString()* method called in print statements**

Remember, *toString()* returns a String (doesn't do the printing)

Output

```
[/]
1->[/]
1->2->[/]
a->1->2->[/]
a->c->1->2->[/]
a->b->c->1->2->[/]
e
z->a->b->e->1->2->[/]
b
z->a->e->1->2->[/]
z
a->e->1->2->[/]
e
a->1->2->[/]
a->1->[/]
```

EXCEPTION

An exception indicates that something unexpected happened at run-time

- Cannot check for all errors at compile time
- What if we ask for element at an index of -1 in an array?
  - There is no clear, “always-do-this”, answer
  - Maybe we should return null or maybe we should stop execution
- Exceptions provide a way to show something is amiss, and let calling functions deal with error (or not)
- Exceptions not handled by a method are passed to calling method. If exception not handled in *main()* or before, program stops
- “Throw” error with `throw new Exception(“error description”)`
- Java provides structured error-handling via *try/catch/finally* blocks
  - *catch* executes only if there is an exception in *try* body
  - *catch* block can specify the type of error it handles
  - Can have multiple *catch* blocks for each *try*
  - *Finally* block executes regardless whether *try* succeeds or fails

ListExceptions.java

ANNOTATED SLIDES

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Create new SinglyLinked List

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Try block

Catch block
Only executes if exception in try block

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Trying to add at index -1 is an error, the catch block will execute because *add()* throws an exception for negative indices

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this
25             System.out.println(e);                  //will give us the error mess
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an
40         }
41 }
42 }
43
```

Trying to add at index -1 is an error, the catch block will execute because *add()* throws an exception for negative indices

```
49     public void add(int idx, T item) throws Exception {
50         if (idx < 0) {
51             throw new Exception("invalid index");
52         }
53         else if (idx == 0) {
54             // Insert at head
55             head = new Element(item, head); //new item ge
56         }
57         else {
58             // It's the next thing after element # idx-1
59             Element e = advance(idx-1);
60             // Splice it in
61             e.next = new Element(item, e.next);  //create
62                                                  //and pri
63         }
64         size++;
65     }
66
```

SinglyLinked.java

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

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

Trying to add at index -1 is an error, the catch block will execute because *add()* throws an exception for negative indices

Catch block on line 15 executes because exception thrown on line 10

Lines 11 and 12 never execute because exception on line 10 stops execution in try block and starts running in catch block

"I never run" and "Neither do I" are not printed

If we didn't catch exception, the program would end because *main()* wouldn't have caught the exception (but we did catch it, so *main()* doesn't end execution)

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

Trying to add at index -1 is *still* an error, the catch block will execute

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an
40         }
41 }
42 }
43
```

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

Trying to add at index -1 is *still* an error, the catch block will execute

We can see what the exception was by printing e (it is just an object)

```
49     public void add(int idx, T item) throws Exception {
50         if (idx < 0) {
51             throw new Exception("invalid index");
52         }
53         else if (idx == 0) {
54             // Insert at head
55             head = new Element(item, head); //new item ge
56         }
57         else {
58             // It's the next thing after element # idx-1
59             Element e = advance(idx-1);
60             // Splice it in
```

"invalid index" was the message we included when we threw an error in the add method of SinglyLinked.java

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

- ***finally*** **always executes, regardless of whether exception in *try* block**
- ***catch*** **only executes if exception occurs in *try* block, otherwise *catch* code does not execute**
- **If exception in *try* block, execution in the *try* block stops at the point of the exception and picks up in first line of *catch* block**
- **Code in the *try* block after the line that caused the exception is not executed**

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

This is valid, so *catch* block does not execute

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

ListExceptions.java: Exceptions can be handled with try/catch/finally blocks

```
4  public class ListExceptions {
5      public static void main(String[] args) { // note: no "throws exception", as every method that could is
6
7          SimpleList<String> list = new SinglyLinked<String>();
8
9          try {
10             list.add(-1, "?");
11             System.out.println("I never run!");
12             System.out.println("Neither do I");
13         }
14         catch (Exception e) {
15             System.out.println("caught it!"); // will print -- we know this is bogus
16         }
17
18         try {
19             list.add(-1, "?");
20             System.out.println("Do I run?");
21             System.out.println("No I don't");
22         }
23         catch (Exception e) {
24             System.out.println("caught it again!"); // will print -- we know this is bogus
25             System.out.println(e);                  //will give us the error message
26         }
27         finally {
28             System.out.println("finally 1"); // executed whether or not caught an error
29         }
30
31         try {
32             list.add(0, "?");
33             System.out.println(list);
34         }
35         catch (Exception e) {
36             System.out.println("why did I catch it again!"); // won't print -- we know this code is fine
37         }
38         finally {
39             System.out.println("finally 2"); // executed whether or not caught an error
40         }
41 }
42 }
43
```

This is valid, so *catch* block does not execute

***finally* always executes, even if no exception in try block**

Problems Javadoc Declaration Console Debug Expressions Error Log Console Call Hierarchy

<terminated> ListExceptions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Apr 6, 2018, 1:20:09 PM)

```
caught it!
caught it again!
java.lang.Exception: invalid index
finally 1
?->[/]
finally 2
```

Iterators/SinglyLinked.java

ANNOTATED SLIDES

What is wrong with this code?

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

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

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

Instantiate SinglyLinked list of Integers

Add 1,000 Integer to List

Print each item in List

Works as intended, but slow

$O(n^2)$ – sneaky inefficiency

Why?

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

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

SinglyLinked.java

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

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

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

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

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

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

Iterator<T> iterator()

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

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

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

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

Iterator interface specifies two methods:

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

Key points:

- ***next* returns the current item in the List and moves to the following item**
- **Remembers where left off so a subsequent call to *next* does not start back at the head**

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

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

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

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

   public ListIterator() {
     curr = head;
   }

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

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

iterator* method returns an object of nested class *ListIterator

Nested class *ListIterator* (private to SinglyLinked)

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

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

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

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

   public ListIterator() {
     curr = head;
   }

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

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

iterator* method returns an object of nested class *ListIterator

Nested class *ListIterator* (private to SinglyLinked)

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

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

SinglyLinked.java provides iterator method that creates an iterator

SinglyLinked.java

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

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

   public ListIterator() {
      curr = head;
   }

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

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

iterator* method returns an object of nested class *ListIterator

Nested class *ListIterator* (private to SinglyLinked)

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

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

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

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

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

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

Java converts for-each loop into

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

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

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

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

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

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

Java converts for-each loop into

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

iterator* method returns an object of nested class *ListIterator

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

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

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

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

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

Java converts for-each loop into

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

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

Notice no increment in for loop

next **will take care of moving curr**

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

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

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

Java converts for-each loop into

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

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

TimeTest.java

ANNOTATED SLIDES

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

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

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

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

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

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

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

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

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

An iterator can dramatically speed up execution time

TimeTest.java

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

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

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

Output

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

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

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