Linked List Operations

Testing list algorithms, testing for empty, searching, pointer pair techniques, removing by value, inserting in order, inserting by position, removing by position, generic linked lists

Testing list algorithms

List algorithms are notoriously easy to botch. You should always test your code to make sure it works:

1. On empty lists.
2. On single item lists.
3. At the beginning of the list.
4. At the end of the list.
5. In the middle of the list.

Linked list setup

This is our setup for a linked list of strings.

```
public class LinkedListOfStrings {
  Node head;  // Reference to the first element in the list

  private static class Node {
    public String data;
    public Node next;
    public Node(String data, Node next) {
       this.data = data;
       this.next = next;
    }
  }
  public LinkedListOfStrings() {
    head = null;
  }
}
```

Is the list empty?

This code is simple!

```
public  boolean isEmpty () {
   return head == null;
}
```

- isEmpty() is helpful for clients.
- You cannot count on the client always checking for empty lists when needed.
- Some list algorithms do the check for empty at the outset to avoid NullPointerExceptions.

Searching for a list item

- To see if the list contains a particular item, do a traversal.

```
public boolean contains(String s) {
   Node p = head;
   while (p != null) {
      if (p.data.equals(s))
         return true;
      p = p.next;
   }
   return false;
}
```

Pointer pairs techniques

Here's an approach to adding an item to a list where the position matters (say in alphabetical or string order):

1. Run a pointer down the list until you find the first item that is bigger than the one you want to insert.
2. Insert the item immediately *before* that first bigger item:
    1. Change the link of that previous item to point to the one to be inserted.
    2. Change the link of the inserted item to point to the bigger one.

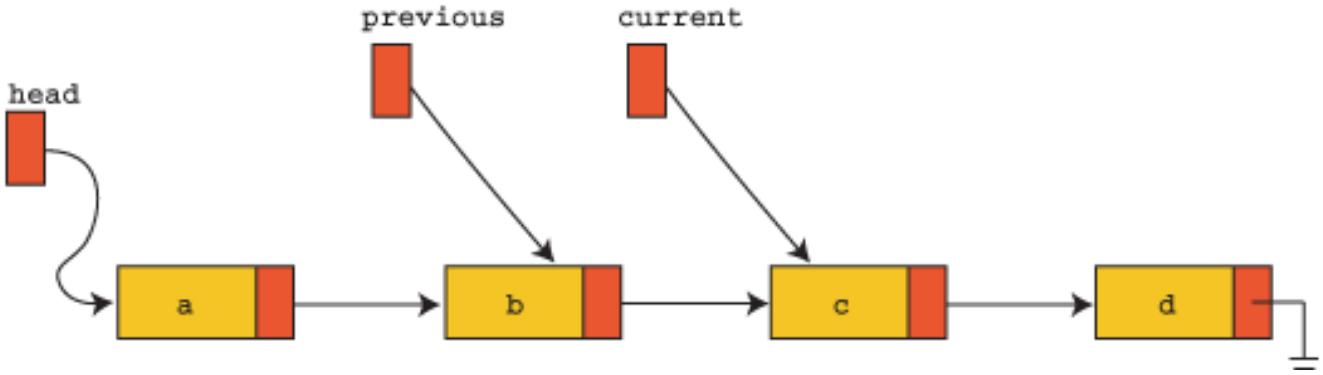
The issue: By the time you've found the bigger item, you've overrun the insertion spot.

Solution: Use TWO pointers, current and previous:

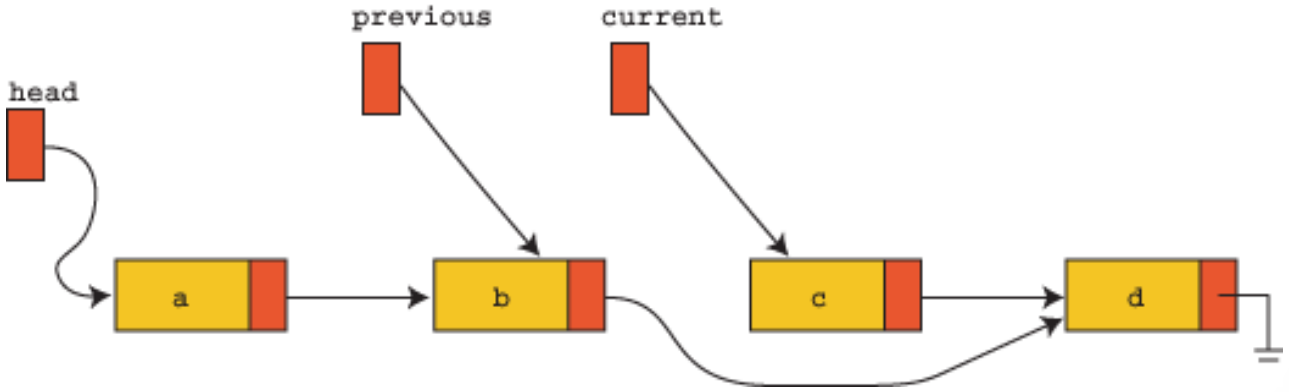
- current goes down the list
- previous follows current

Removing a list item

1. Travel down the list with 2 pointers, searching for the item to remove.



2. Change previous.next to current.next.



Removing a list item – code

```
public void remove(String x){
   Node current = head;
   Node previous = null;
   while (current != null && !current.data.equals(x)) {
      previous = current;
      current = current.next;
   }
   if (current != null) { // x is in the list
      if (current == head){ // x is the first item?
          head = head.next;
      }
      else { // x is somewhere after the first item
          previous.next = current.next;
      }
   }
}
```

Removal – code analysis

```
public void remove(String x){
   Node current = head;
   Node previous = null;
   while (current != null && !current.data.equals(x)) {
      previous = current;
      current = current.next;
   }
}
```

- `current` is initialized to the first list element.
- The while loop condition is true as long as:
  - `current` has not run off the end of the list
    and
  - `x` has not been found
- At end of loop execution, `current` points to `x` or is `null`.

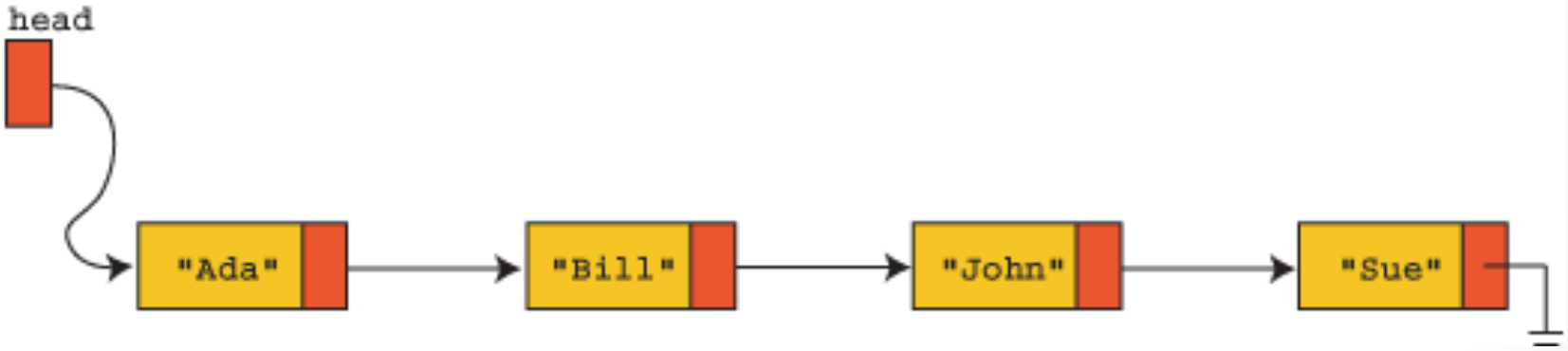
Removal – code analysis (cont)

```
if (current != null) { // 1
   if (current == head){ // 2
       head = head.next;
   }
   else { // 3
       previous.next = current.next;
   }
}
```

1. If `current` is not `null`, `current` points to `x`.
2. If `current` is `head`, then `x` is the first element in the list.
3. If `current` is not `head`, then `x` is further down the list.

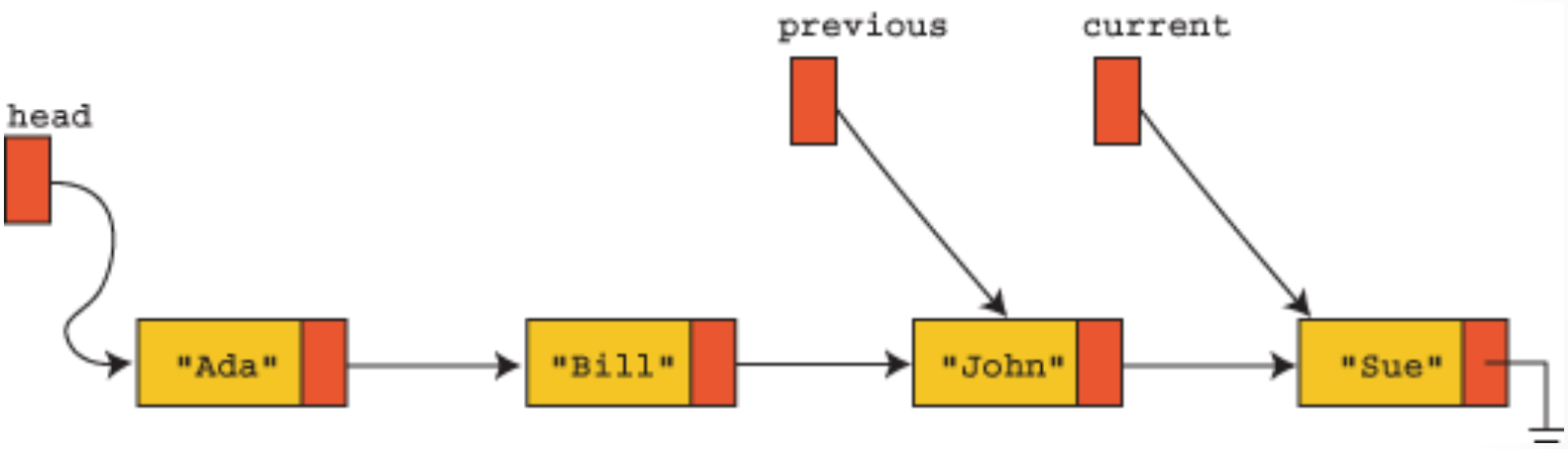
Ordered list insertion

- Ordered lists take special insertion code. (You cannot simply insert at front or arbitrary position and maintain the order.)
- Example: Insert “Mary” into the *ordered* list consisting of “Ada”, “Bill”, “John”, and “Sue”.



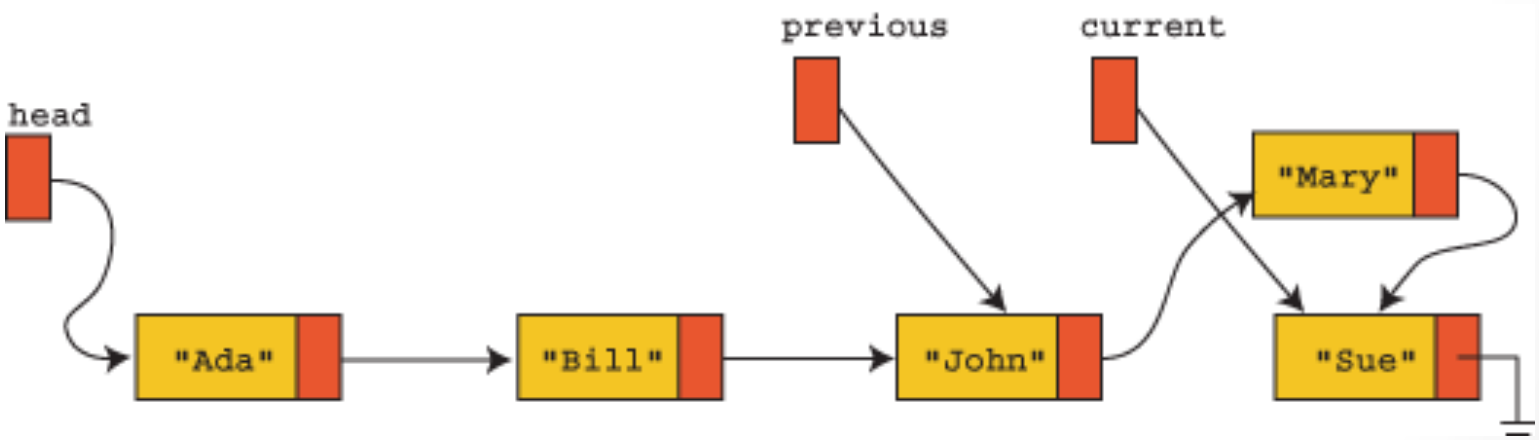
Ordered list insertion (cont)

- Traverse the list with a pair of pointers until the first points to the node immediately after the insertion position.



Ordered list insertion (cont 2)

- When **`previous`** and **`current`** are in the correct positions, create a new node with "**`Mary`**" and **`current`** as element data.
- Set **`previous.next`** to the new node.



Ordered list insertion code

```
public void insert(String x){
   Node current = head;
   Node previous = null;

   while (current != null && (current.data.compareTo(x) < 0)) {
       previous = current;
       current = current.next;
   }
   if (current == head) {
        head = new Node(x, head);
   }
   else {
        previous.next = new Node(x, current);
   }
}
```

Insertion code: analysis

```
public void insert(String x){
   Node current = head;
   Node previous = null;

   while (current != null && (current.data.compareTo(x) < 0)) {
       previous = current;
       current = current.next;
   }
```

The loop searches for the insertion position. It stops when:

- `current` runs off the end of the list. That means `x` belongs at the end of the list.
- `current` detects a list item that is greater than `x`.

Insertion code: analysis (cont)

```
// Loop to find insertion position
if (current == head) {
     head = new Node(x, head);
}
else {
     previous.next = new Node(x, current);
}
```

At the end of the loop:

- **`current`** is **`head`**

  -or-

- **`previous`** points to the item before where **`x`** should go.

Insertion code: analysis (cont 2)

- Does this code work?
  - If the list is empty?
  - If the new item goes at the front of the list?
  - If the new item goes at the end of the list?
  - If the new item goes in the middle of the list?

Operations by position

- Linked list elements are not indexed, but there is an element #0, element #1, element #2, and so on.
- To use insertions/removals by position (or index), you need to count.
- Pointer pairs are useful.
- Alternative to pointer pairs is to consider `current.next.data`, taking care not to run off the end.

Insertion by position – code

```
public void insertAt(String x, int psn) {
   if (psn == 0)
      head = new Node(x, head);
   else if (head!= null) {
      Node current = head;
      while (current != null && psn > 1) {
         current = current.next;
         psn--;
      }
      if (current != null) {
          current.next = new Node(x,current.next);
      }
   }
}
```

Position insertion code: analysis

- Does this code work?
  - If the list is empty?
  - If the new item goes at the front of the list?
  - If the new item goes at the end of the list?
  - If the new item goes in the middle of the list?

Removal by position

```
public void remove(int psn){
   Node current = head;
   Node previous = null;
   while (current != null && psn > 0)) {
      previous = current;
      current = current.next;
      psn--;
   }
   if (current != null) {
      if (current == head){ // remove item 0?
          head = head.next;
      }
      else { // not removing item 0
          previous.next = current.next;
      }
   }
}
```

Removal by position: analysis

The code for removal by position is *almost* identical to the code for removal by value.

Insertion by Value	Insertion by Position
`while (current != null && !current.data.equals(x))`	`while (current != null && psn > 0))`
`{ previous = current; current = current.next; }`	`{ previous = current; current = current.next; psn--; }`

Generic Linked Lists

- We've defined lists in which the element type is known at compile time.
- This code shows data and types for a simple generic linked list.
- The <E> parameter is needed on the outer class, but it is not needed on the inner class.

```
public class SimpleLinkedList<E> {
   private Node head;

   // Constructors and operations here

   private class Node{
      public E data;
      public Node next;
      public Node(E data, Node next) {
         this.data = data;
         this.next = next;
      }
   }
}
```

Generic Linked Lists

Except for ordered lists, all our linked list algorithms work for any type of element. This code shows removal where the element removed is also returned.

```
public E remove(int index) {
   Node current = head;
   Node previous = null;
   for (int k = 0; k < index; k++) {
      if (current == null)
         throw new IndexOutOfBoundsException();
      previous = current;
      current = current.next;
   }
   if (current == null)
      throw new IndexOutOfBoundsException();
   E value = current.data;
   if (current == head)
      head = head.next;
   else
      previous.next = current.next;
   return value;
}
```