Introduction to Linked Lists

Linked list basics, references and null, linked list structure, linked lists vs array-based lists, Node and class setup, insertion/removal at the front, traversals

Quick terminology review

- A reference is a name that can serve as a name for an object.

  ```
  String s = “abc”;
  ```

- We usually think of references in terms of variables.
- A reference is null when it is not the name for any object.

  ```
  String t = null;
  ```

- Dereferencing means using the name to access the object.

  ```
   int k = s.length();
  ```

- A NullPointerException occurs when you try to dereference null.

  ```
  t.trim();
  ```

Working with null

- You can store null in a variable.

```
String s = null;
Thermometer r = null;
```

- You can print a null reference

```
System.out.println(r);      // null
```

- You can check to see if a variable is null

```
if (r == null) { ...
```

- You can pass null as a parameter to a method.

```
public void foo(Thermometer x) …
foo(r);
```

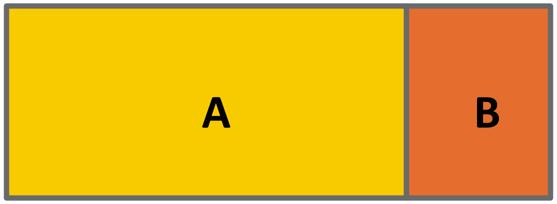
- You can return null from a method (often to indicate failure)

```
public String example() {
  if (condition) return null;
```

- null is the link for the last element in a linked list.

What is a linked list?

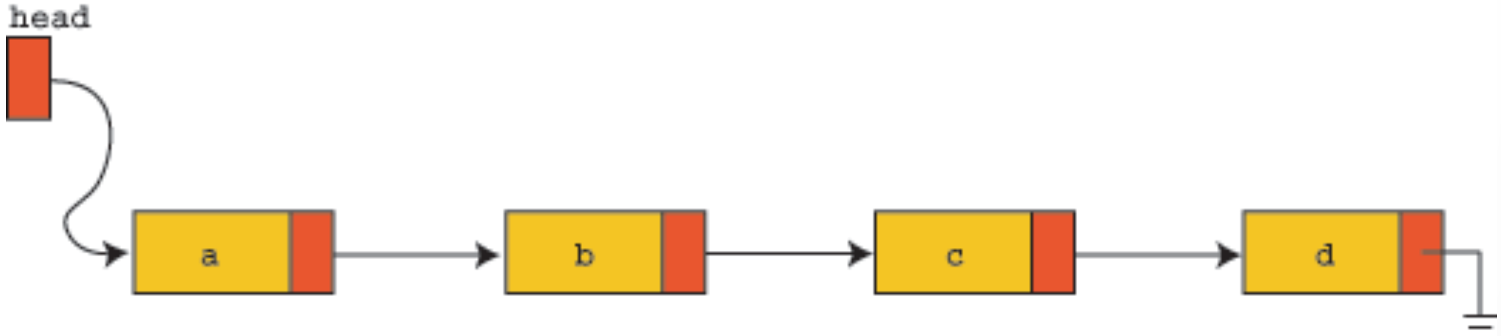
- A linked list is a list of elements consisting of two pieces of information:
  - A: the actual list data (any type)
  - B: the location of the next element



- The location, which is a reference to the next list element, is called a **link**. Links are sometimes called **pointers**.
- The elements of linked lists (data and links) are called **nodes**.

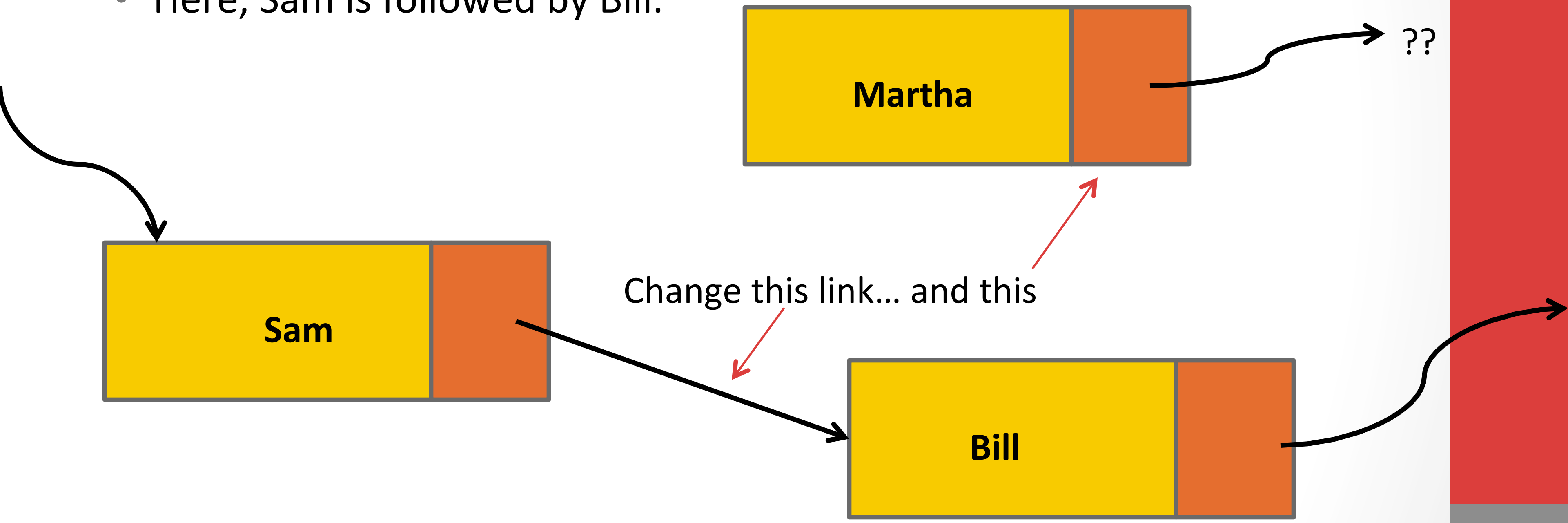
Linked list structure

- The first element of a linked list is the list **head**.
- From the head, you can get to the second element.
- From the second, you can get to the third, and so on.
- The link part of the last node is null.



Working with links

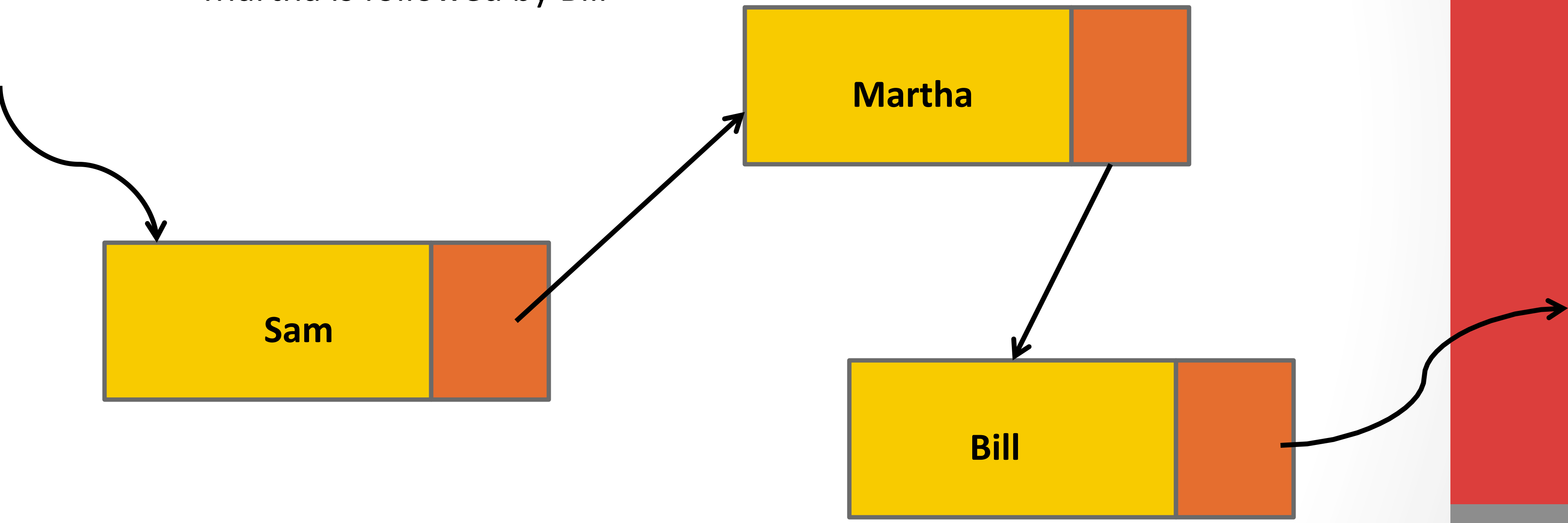
- Links are references to other nodes or null.
- When you change a link, you break the list structure.
- Here, Sam is followed by Bill.



Working with links (cont)

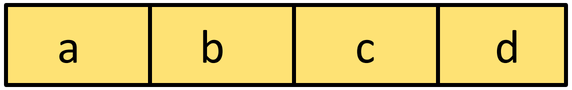
After the two link changes

- Sam is followed by Martha
- Martha is followed by Bill

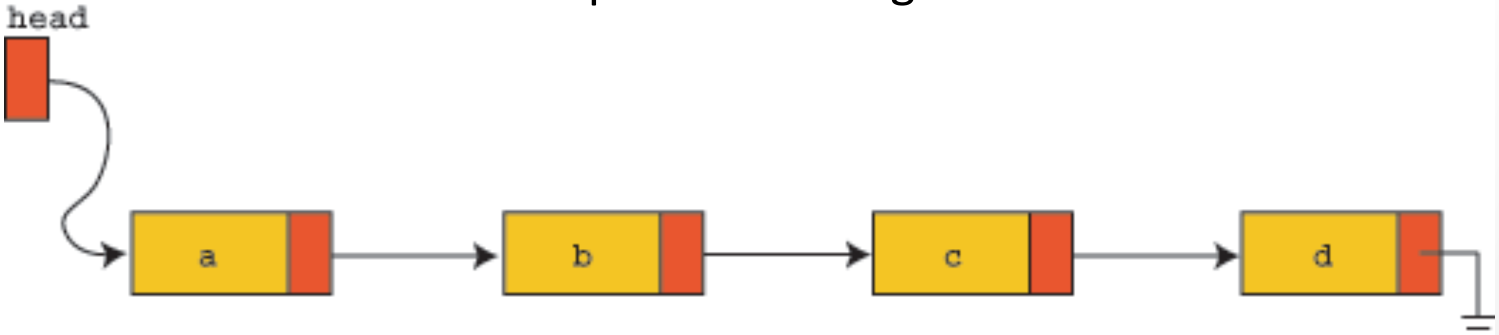


Linked lists vs array-based lists

- Indexes:
  - Each element in an array-based list has an index.
  - No indexes for linked lists.



- Capacity:
  - Array-based lists are limited by their array capacities (unless the are dynamic – expansion is expensive).
  - No theoretic capacity limitation on linked lists.
- Insertions/Removals:
  - Array-based lists require shifts.
  - Linked list insertions require link changes.



Setting up a linked list class

Here is a start on a linked list of strings.

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

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

Inner Node class

- Node is an inner class.
- Node has an element of the same type as itself.
- Node members are public (no need for getters and setters).
- Node constructor requires both data and link.

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

Code traces

Assume this code is inside LinkedListOfStrings.

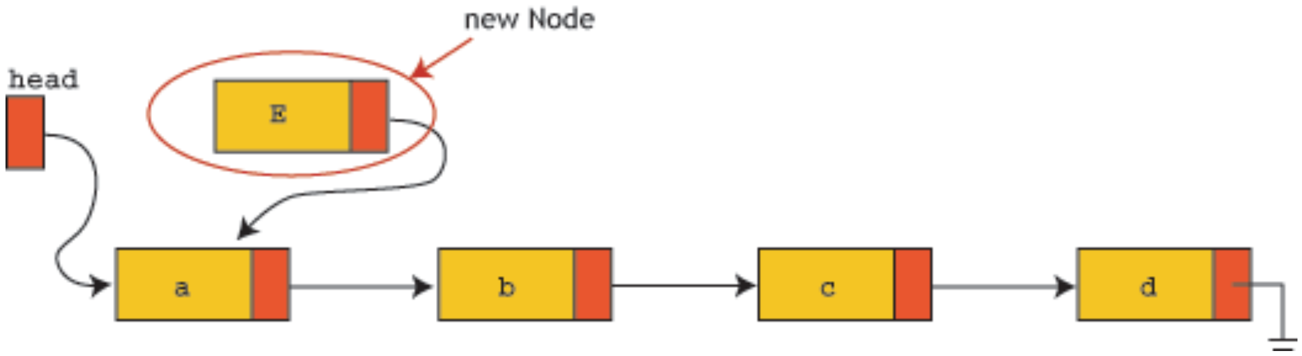
```
head = null;
head = new Node("x", null);
head.next = new Node("a", null);
head.next.next = new Node("z", null);
head = new Node("b", head);
```

What does the list look like after execution?

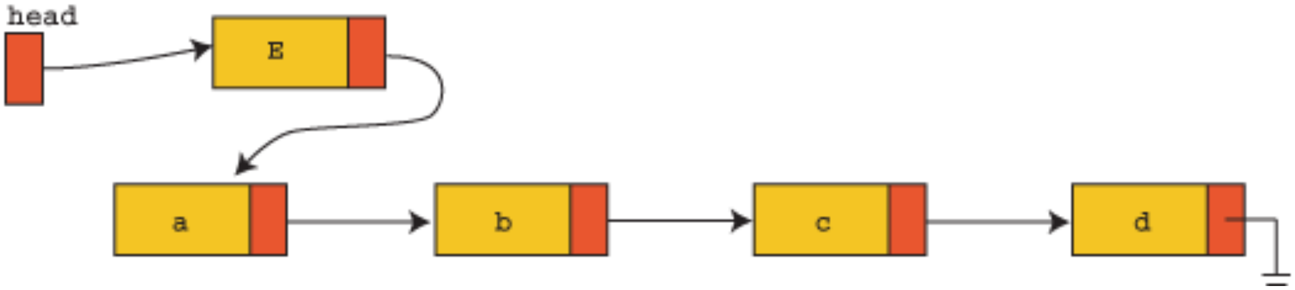
Adding to the front

Two step process:

1. Create a new Node with the desired data and a link pointing to the first list node.



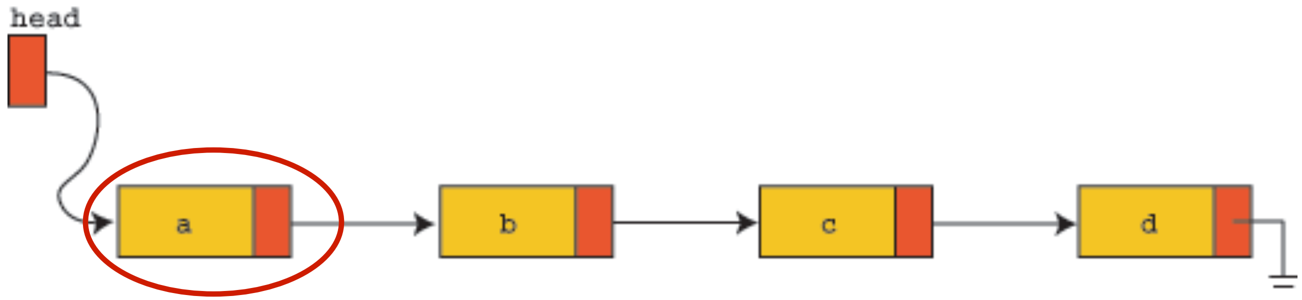
2. Change head to point to the new Node.



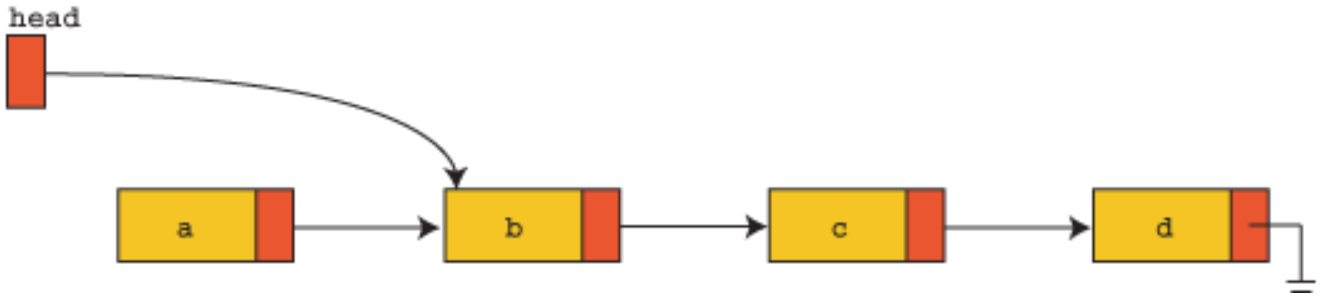
Removing from the front

Two step process:

1. Save the data to be discarded (*a* in the figure) to return it.



2. Set head to point to the next node.



Front add/remove code

```
public void addToFront(String s) {
   head = new Node(s,head);
}

public String removeFromFront() {
   if (head == null)
      return null;
   String s = head.data;
   head = head.next;
   return s;
}
```

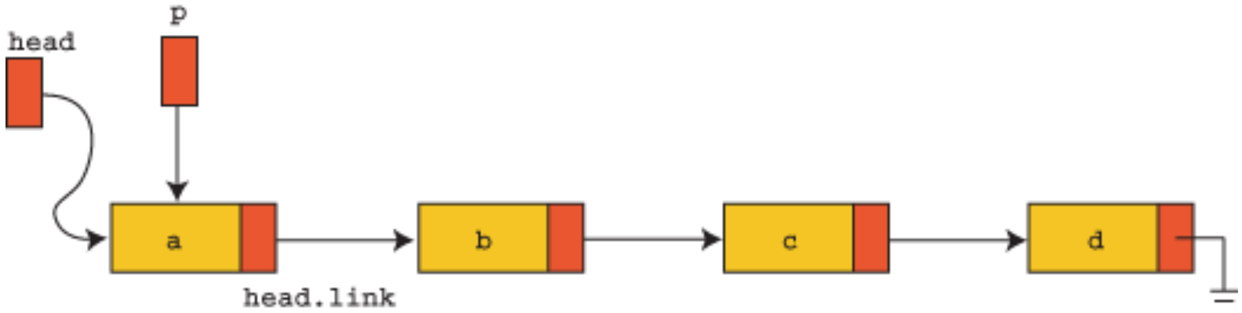
- Insertion code is a single statement.
- Check: does the code work on an empty list?
- Check: does the removal code work on a list with exactly one element?

Traversing a linked list

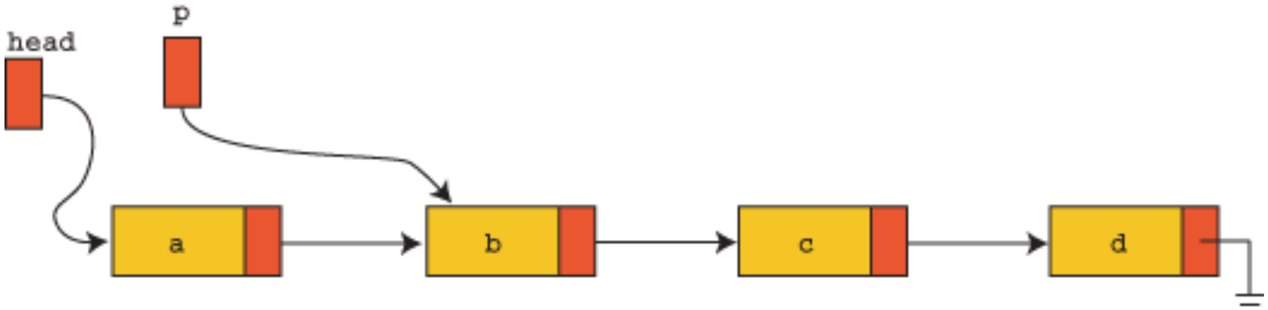
Traversing a list means walking through the list starting at the front and looking at the list items, one-at-a-time.

Use a **pointer**, which is simply a reference to a node.

```
Node p = head;
```



```
p = p.next;
```



Traversal code

```
/**
 * Find a string representation of the entire list,
 *    terminating each list item with a newline.
 * @return the string representation, or the null
 *    string if the list is empty
 */
public String toString() {
   String s = "";
   for (Node p = head; p != null; p = p.next)
      s += p.data + "\n";
   return s;
}
```

Alternative loop structure:

```
   Node p = head;
   while (p != null) {
      s += p.data + "\n";
      p = p.next; // Infinite loop without this statement!
   }
```

Traversal code variation

```
/**
 * Find a string representation of list items that start
 *    with the given string. The string representation
 *    terminates each such item with a newline.
 * @param prefix the prefix for each string represented
 * @return the string representation, or the null
 *    string if the list is empty
 */
public String toPrint(String prefix) {
   String s = "";
   Node p = head;
   while (p != null) {
      if (p.data.startsWith(prefix))
         s += p.data + "\n";
      p = p.next;
   }
   return s;
}
```

A few linked list questions

- What is the value of the last link in a list?
- What is the difference between a null list and an empty list?
- How do you insert a node at the *end* of a list?
- What is the difference?

`Node p = head;`
`p = p.next;`
	`head = head.next;`

`Node p = head;`
`p.next = new Node(“a”, null);`
	`head = new Node(“a”, null);`