

CSCI 136

Data Structures & Advanced Programming

Singly Linked Lists

Singly Linked Lists

Linked Lists

- List: A general-purpose structure
- Implementing Lists with linked structures
 - Singly Linked Lists
 - A variant: Circularly Linked Lists
 - Doubly Linked Lists

For Maximum Value

If you haven't yet reviewed the implementations of Vector methods, do so before continuing!

Pros and Cons of Vectors

Pros

- Good general purpose list
- Dynamically Resizeable
- Fast access to elements
 - `vec.get(387425)` finds item 387425 in the same number of operations regardless of vec's size

Cons

- Slow updates to front of list (why?)
- Hard to predict time for add (depends on internal array size)
- Potentially wasted space

Today we look at another way to store data: Linked Lists

But First : List Interface

The Vector class implements many useful methods

```
size()  
isEmpty()  
contains(e)  
get(i)  
set(i, e)  
add(i, e)  
remove(i)  
addFirst(e)  
getLast()...
```

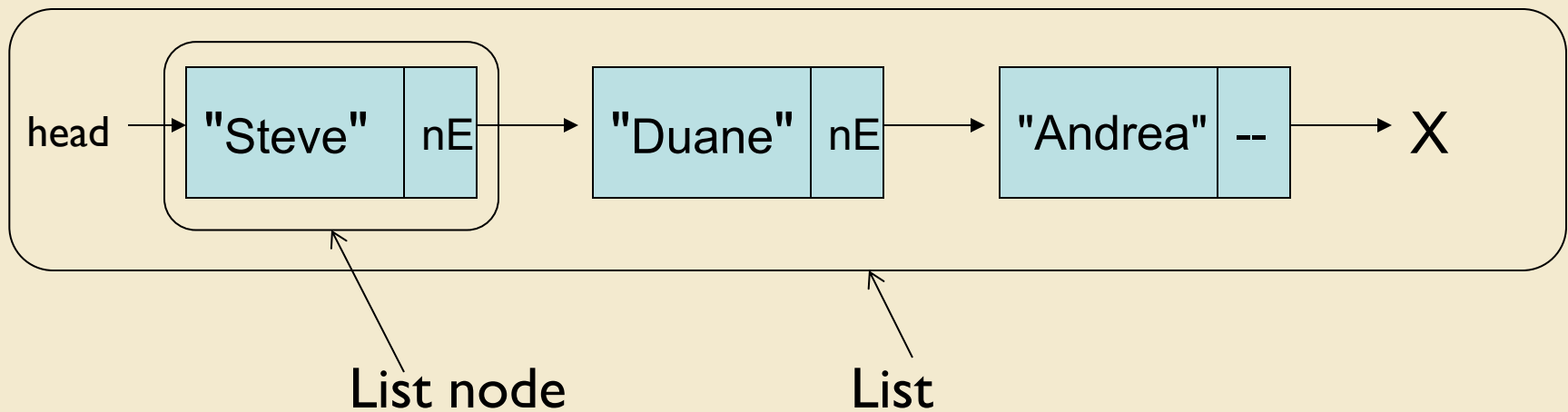
- Other data structures provide similar methods
- This set of methods is a natural candidate for forming an interface: The List Interface
- We will develop structures that implement the List interface
- They are called *linked structures*

List Interface

```
public interface List<E> extends Structure<E> {  
  
    public int size();  
    public void clear();  
    public E getFirst();  
    public E getLast();  
    public E remove(E value);  
    public E remove(int i);  
    public E remove();  
    public void add(E value);  
    public E get();  
    public E get(int i);  
    public E set(int i, E o);  
  
    public boolean isEmpty();  
    public void addFirst(E value);  
    public void addLast(E value);  
    public E removeFirst();  
    public E removeLast();  
    public boolean contains(E value);  
    public int indexOf(E value);  
    public int lastIndexOf(E value);  
    public void add(int i, E o);  
    public Iterator<E> iterator();  
}
```

Linked List Basics

- There are two key aspects of Lists
 - The *nodes* of the list
 - The list itself
- Visualizing lists



Linked List Basics

- List nodes are *recursive* data types
 - A node contains an instance variable of type node!
- Each “node” has:
 - A data value
 - A “next” value that identifies next node in the list
 - Could also have “previous” that identifies the previous node (“doubly-linked” lists)
- What state and methods does a Node need?

Public Class Node<E>

A singly-linked list node

```
protected E data; // value stored in this node
protected Node<E> nextElement; // ref to next node
public Node(E v, Node<E> next) {
    data = v;
    nextElement = next;
}
public Node(E v) { this(v,null); }

public Node<E> next() { return nextElement; }

public void setNext(Node<E> next){ nextElement = next; }

public E value() { return data; }

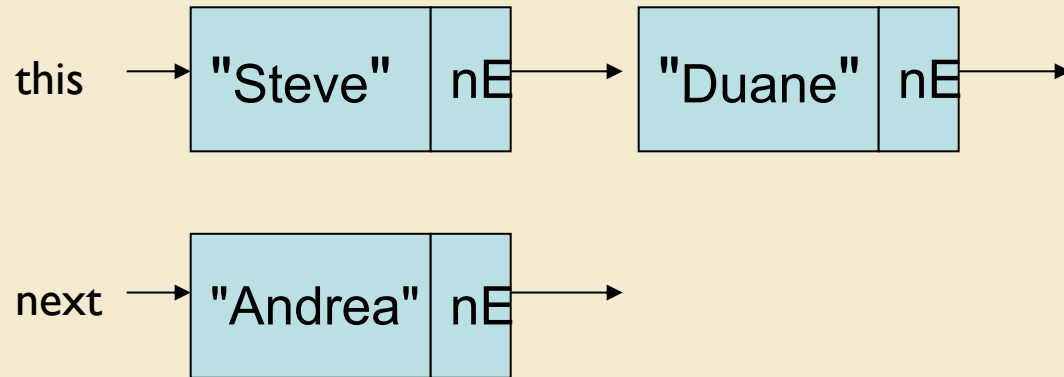
public void setValue(E value) { data = value; }

public String toString() { return "<Node: "+value()+">"; } 10
```

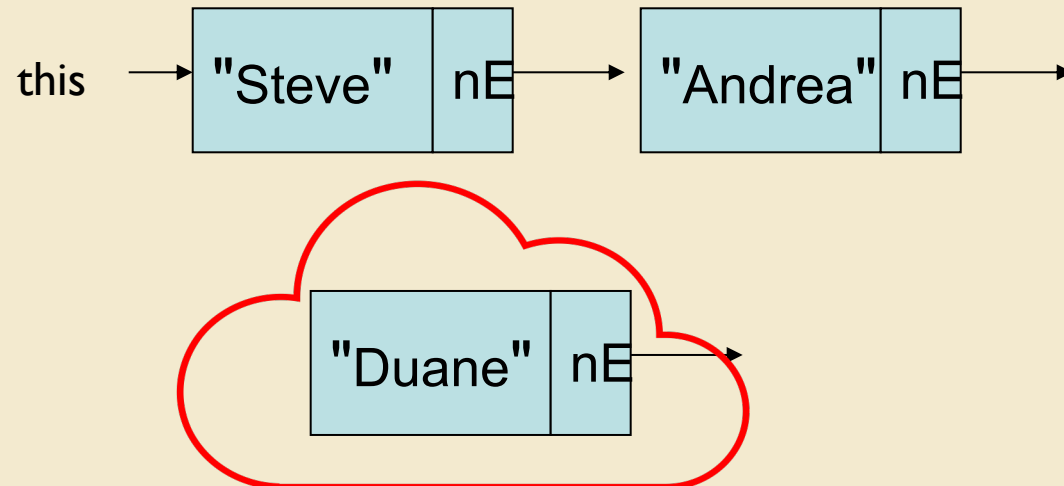
Visualizing setNext

```
public void setNext(Node<E> next){ nextElement = next; }
```

Before:

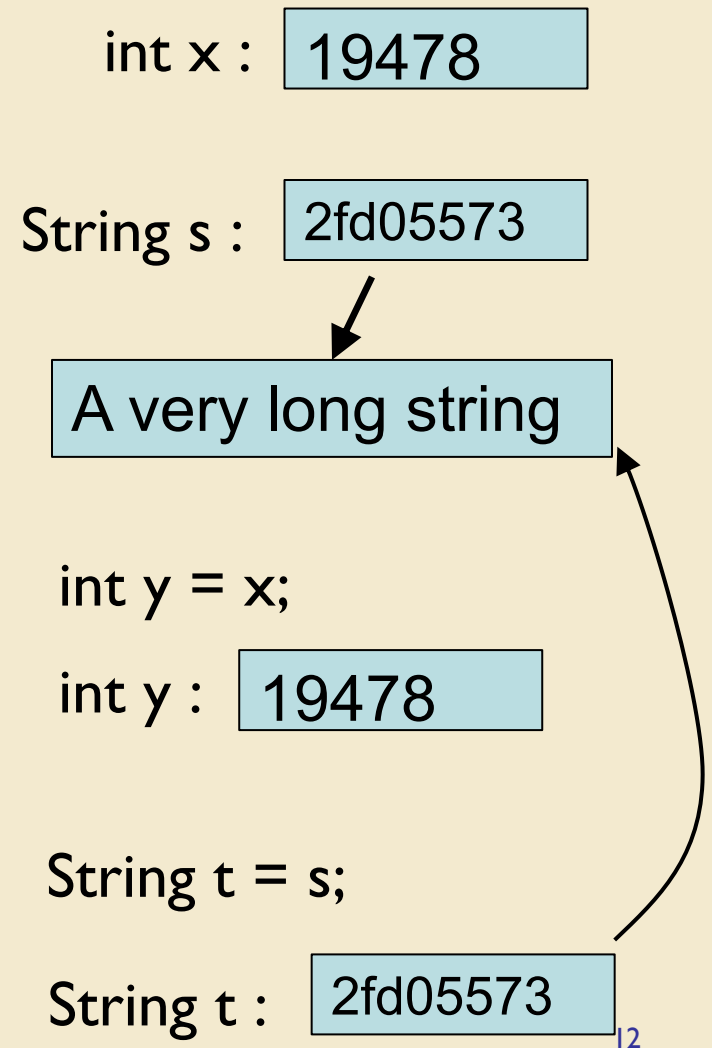


After:



Types and Memory

- Variables of primitive types
 - Hold a value of primitive type
- Variables of class types
 - Hold a *reference* to the location in memory where the corresponding object is stored
- Variable of array type
 - Holds a *reference*, like variables of class type
- Assignment statements
 - For primitive types, copies the value
 - For class/array types, copies the reference



Variables and Memory

- Instance variables
 - Upon declaration are given a default value
 - Primitive types
 - 0 for number types, false for Boolean, \u0000 for char
 - Class types and arrays: null
- Local variables
 - Are NOT given a default when declared
- Method parameters
 - Receive values from arguments in method call

Memory Management in Java

- Where do “old” objects go?

```
Track t = new Track("Hey, Jude", "The Beatles", ... );  
...  
t = new Track ("Blowin' in the Wind", "Bob Dylan", ... );
```

- What happens to Hey, Jude?
- Java has a *garbage collector*
 - Runs periodically to “clean up” memory that had been allocated but is no longer in use
 - Automatically runs in background
- Not true for many other languages!

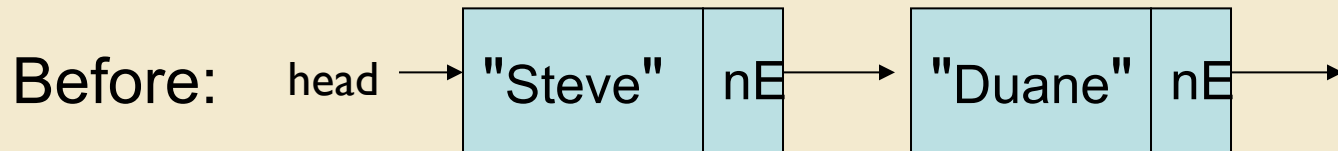
SinglyLinkedLists

- A SinglyLinkedList object stores only two values

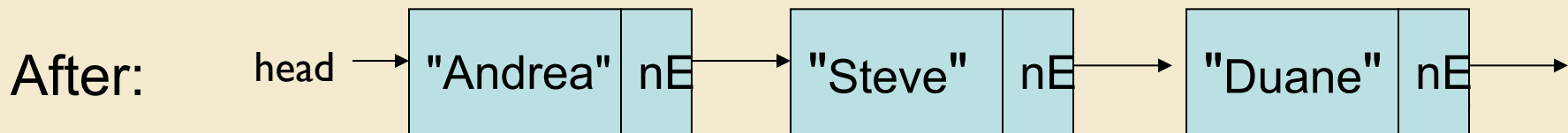
```
// list size
protected int count;
// ref. to first element
protected Node<E> head;
```

- What would addFirst(E d) look like?

```
public void addFirst(E value) {
    head = new Node<E>(value, head);
    count++;
}
```



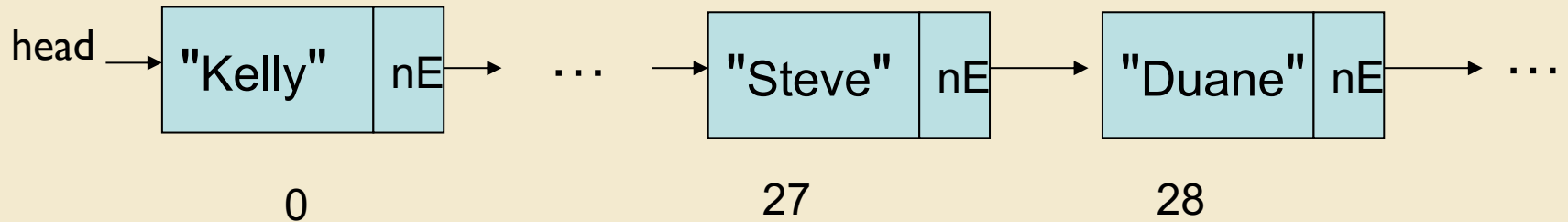
```
myList.addFirst("Andrea");
```



More SLL Methods

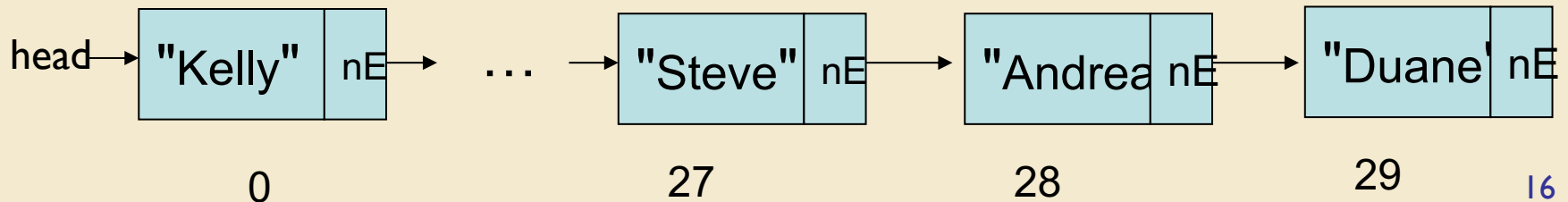
- How would we implement `add(int i, E o)`?

Before:



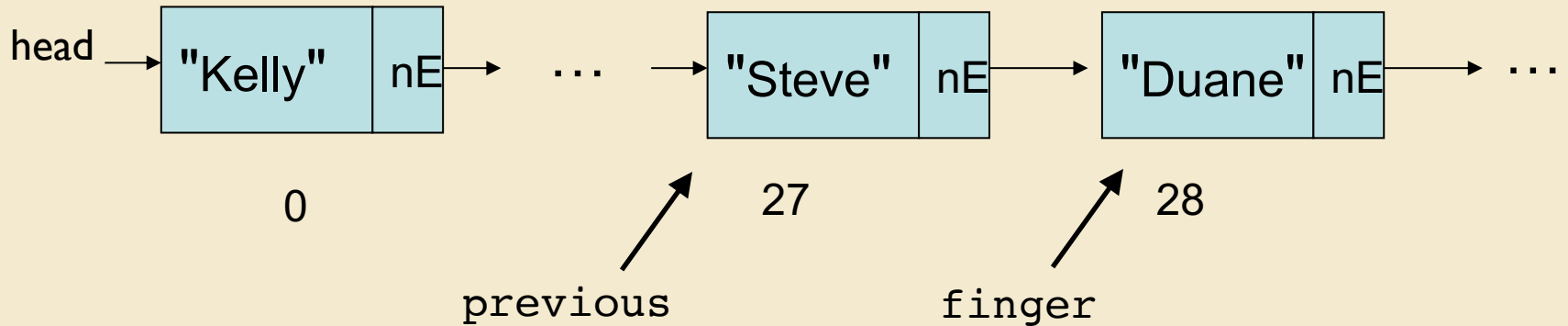
```
myList.add(28, "Andrea");
```

After:

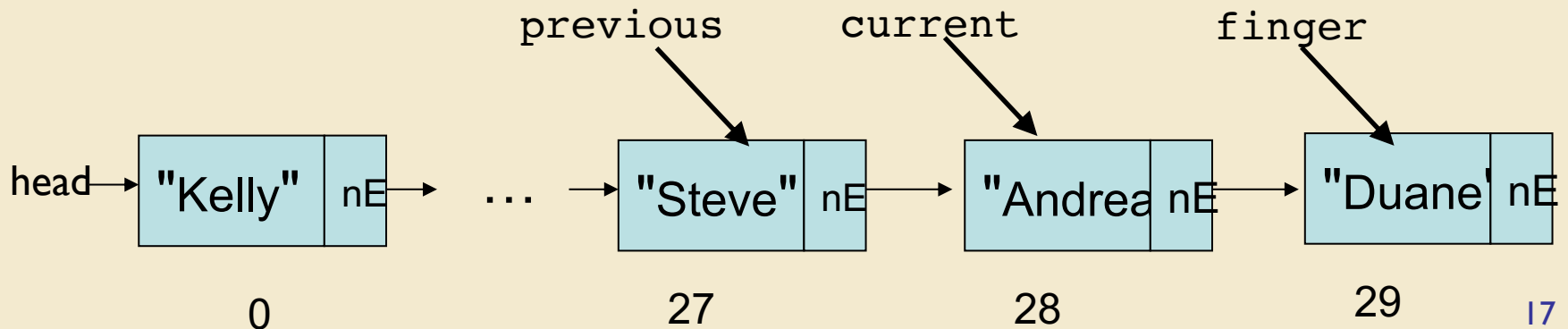


Add(int i, E o)

```
Node<E> previous = null;  
Node<E> finger = head;
```



```
Node<E> current = new Node<E>(o, finger);  
previous.setNext(current);
```



Add(int l, E o)

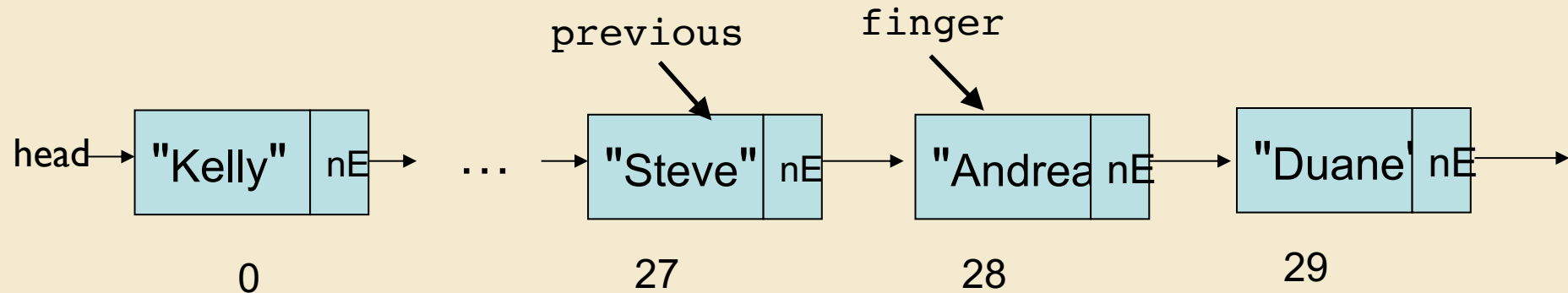
```
public void add(int i, E o) {
    Assert.pre((0 <= i) && (i <= size()), "Index in range.");
    if (i == size()) {
        addLast(o);
    } else if (i == 0) {
        addFirst(o);
    } else {
        Node<E> previous = null;
        Node<E> finger = head;
        // search for ith position, or end of list
        while (i > 0) {
            previous = finger;
            finger = finger.next();
            i--;
        }
        // create new value to insert in correct position
        Node<E> current = new Node<E>(o, finger);
        count++;
        // make previous value point to new value
        previous.setNext(current);
    }
}
```

Remove(int i)

```
myList.remove(28);
```

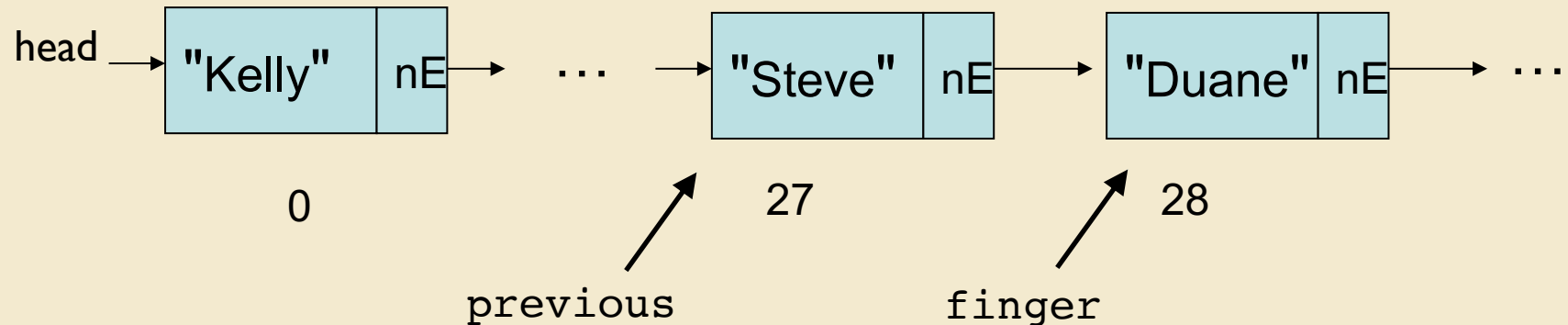
```
Node<E> previous = null;
```

```
Node<E> finger = head;
```



```
previous.setNext(finger.next());
```

```
count--;
```



Remove

```
public E remove(int i) {  
  
    Assert.pre((0 <= i) && (i < size()), "Index in range.");  
  
    if (i == 0) return removeFirst();  
    else if (i == size()-1) return removeLast();  
  
    Node<E> previous = null;  
    Node<E> finger = head;  
  
    // search for value indexed, keep track of previous  
    while (i > 0) {  
        previous = finger;  
        finger = finger.next();  
        i--;  
    }  
    // in list, somewhere in middle  
    previous.setNext(finger.next());  
    count--;  
    // finger's value is old value, return it  
    return finger.value();  
}
```

Linked Lists Summary

- Recursive data structures used for storing data
- More control over space use than Vectors
 - No hidden costs like `Vector.ensureCapacity()`
 - No "empty slots" like Vector
 - But: Keeps an extra reference for each value
- Adding objects to front of list is fast 😊
 - Adding to end of list is slow 😓
- Components of `SinglyLinkedList`
 - head, count
- Components of Node
 - data, `nextElement`