

# ExSEL Tutoring for 187

- Su, 7-8:15pm W.E.B. Du Bois Library- Room 1205
- Th, 8:30-9:45pm W.E.B. Du Bois Library- Room 1302

ExSEL sessions are 75 minutes in length and are capped at 10 students. We will hold half of the spots for appointments (students can book via the LRC page, [www.umass.edu/lrc](http://www.umass.edu/lrc)) and half of the spots for drop in visits.

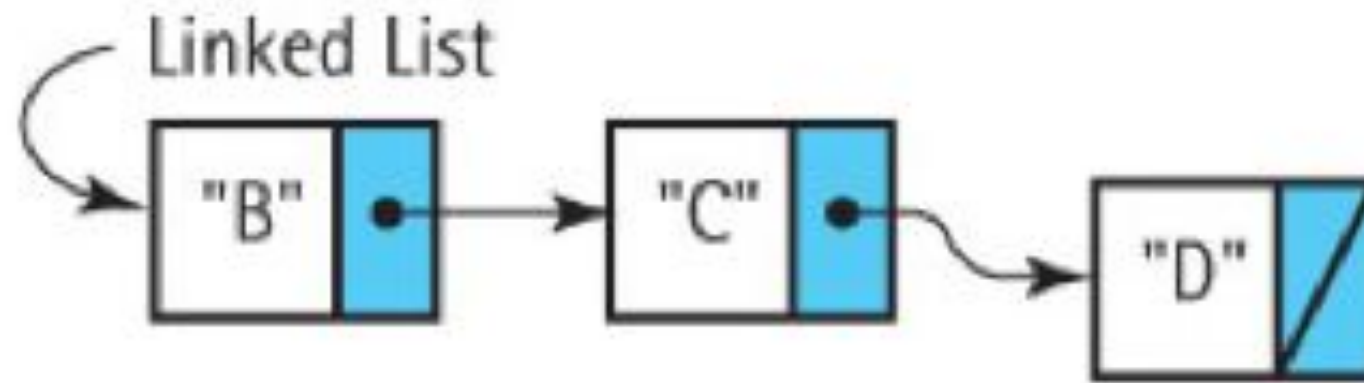
# Is there a better way?

```
if (a.equals(b)){  
    x = 10;  
} else {  
    x = 12;  
}
```

# Linked StringLogs

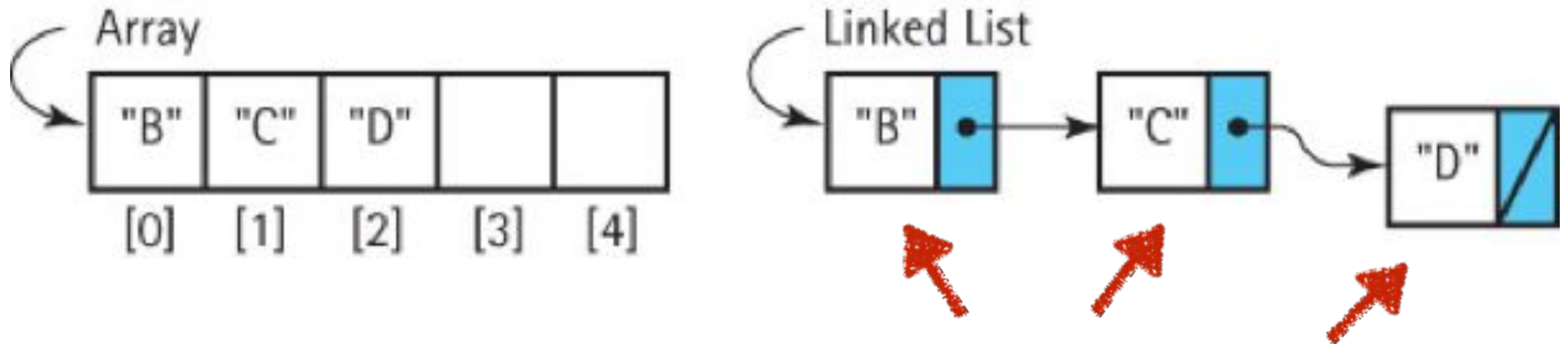
- The Linked List Data Structure
- Arrays vs. Linked Lists
- The LLStringNode Class
- Operations on a Linked List
- A linked-list implementation of StringLog

# Linked List



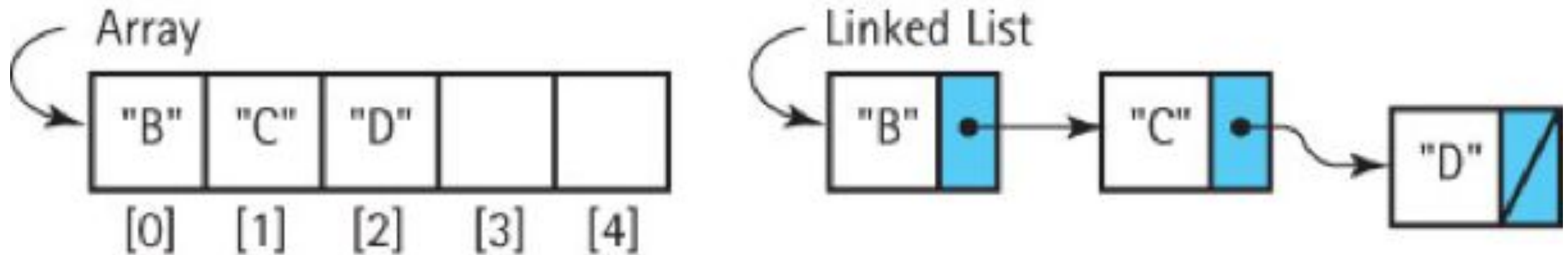
- A series of **nodes** chained together.
- Each node contains a **data element** and a **link** (i.e. pointer) to the next node in the chain.
- If we know the head of the chain, we can follow successive links to reach any node on the chain.
- The last node has a **null** pointer, because there is no node following it.

# Comparison to Arrays



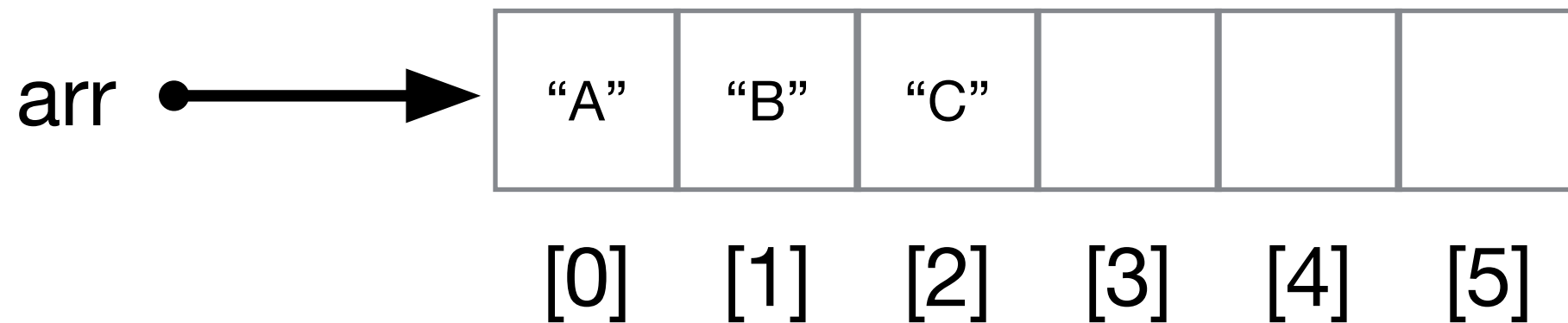
- Arrays store all data elements consecutively in memory. This makes it easy to access any element using index.
- In a linked list, each element is separate in its own block of memory (i.e. a node). There is no easy way to directly access an element using index. Imagine trying to find someone's phone number by following a chain of friends.

# Arrays vs. Linked Lists



- But the size of an array is fixed. If the array is not fully filled, you can end up wasting a lot of memory. In comparison, a linked list has truly dynamic size.
- Also, for certain operations (such as inserting or deleting an element at the front), linked list is faster.

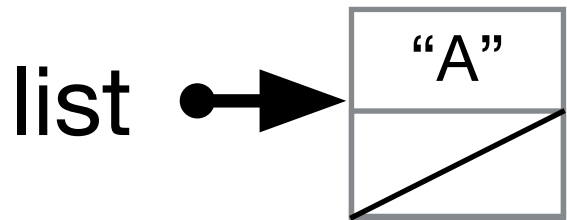
# Benefits: Size/Space



- The size of an array is fixed (bounded)

```
char[] arr = new char[6];
```

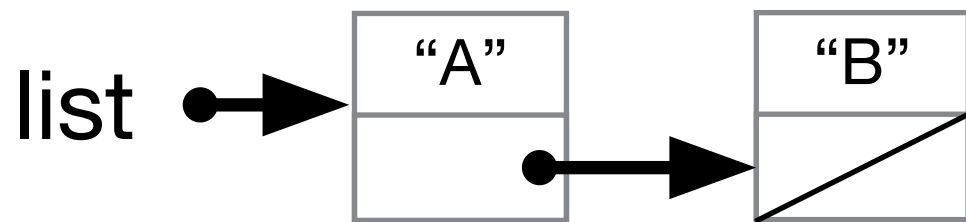
# Benefits: Size/Space



- The size of a linked list is not fixed (unbounded)
- Allocate node "A" (size 1)

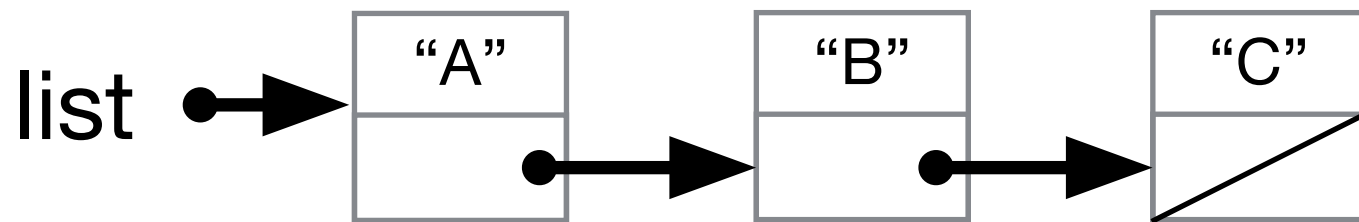


# Benefits: Size/Space



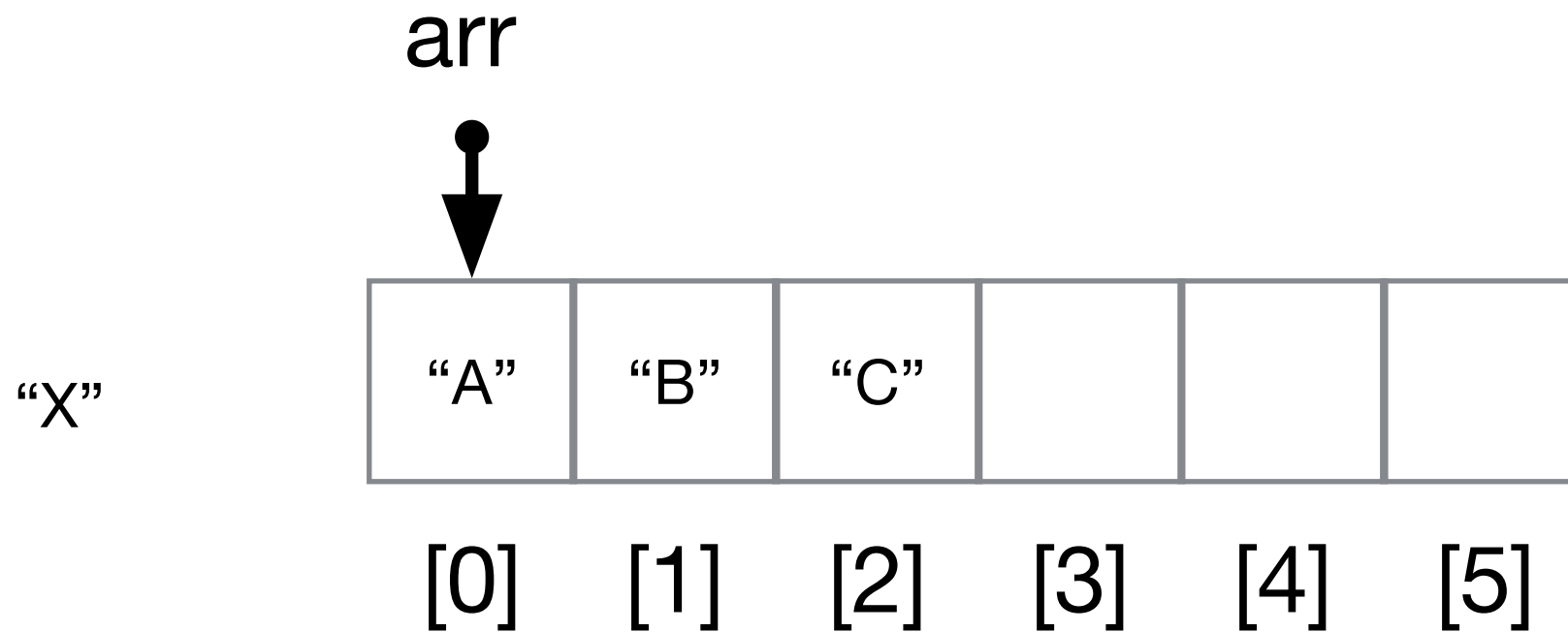
- The size of a linked list is not fixed (unbounded)
- Allocate node "B" (size 2)

# Benefits: Size/Space



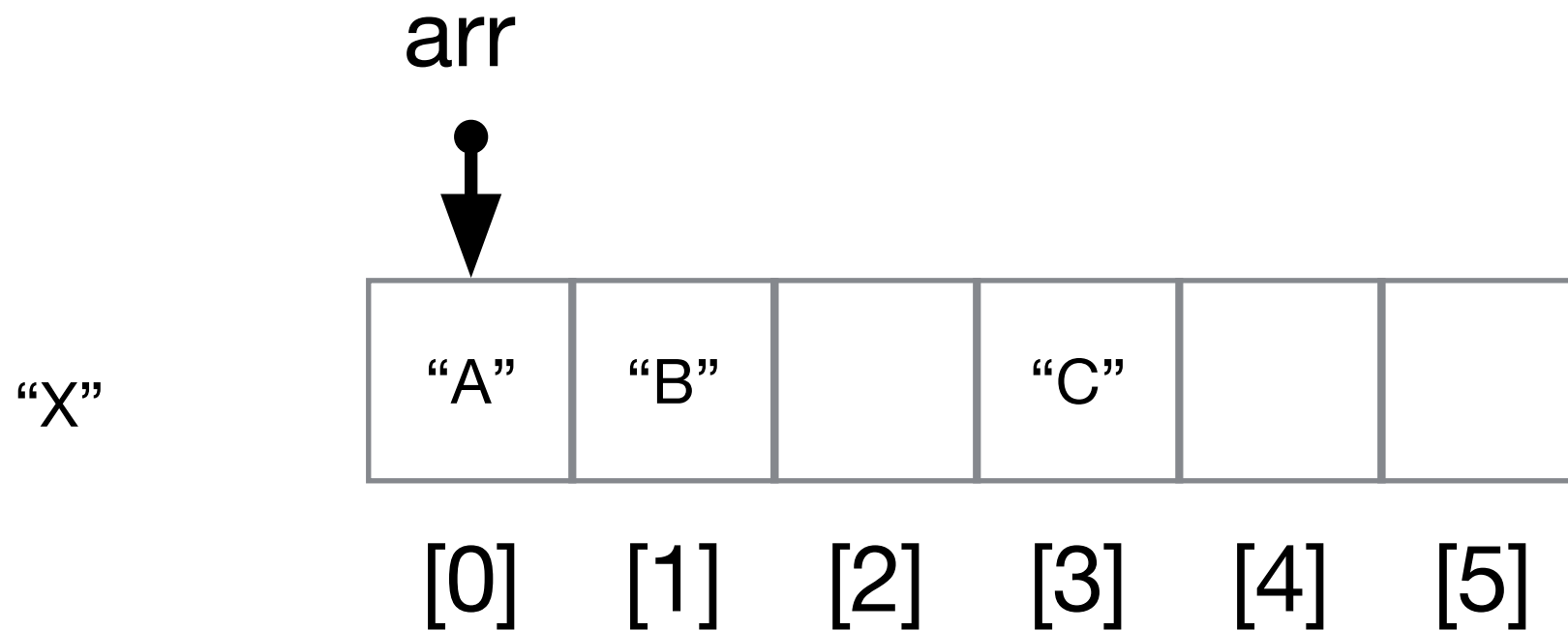
- The size of a linked list is not fixed (unbounded)
- Allocate node "C" (size 3)
- There are many applications where you can't predict, ahead of time, how many elements you will end up storing. This is where Linked List wins.

# Benefits: Front Insertion



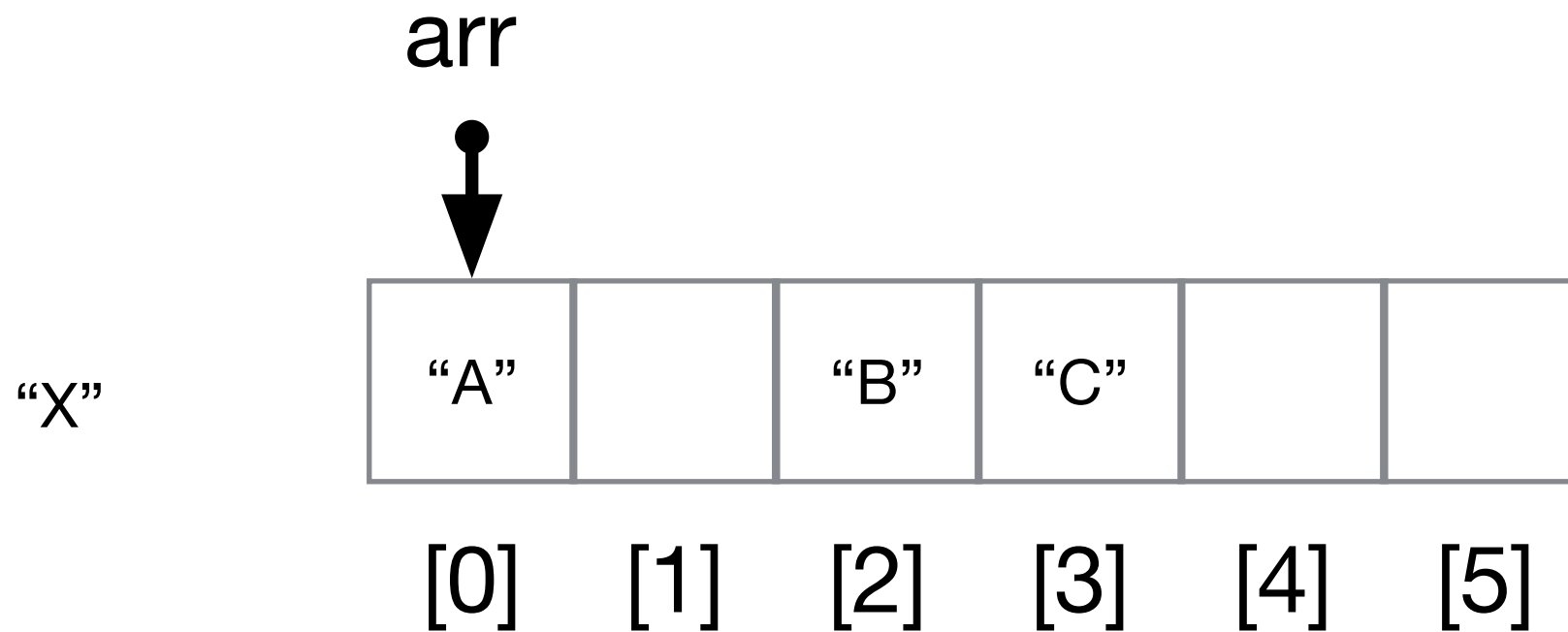
- If we need to insert a new element "X" into the **front** of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

# Benefits: Front Insertion



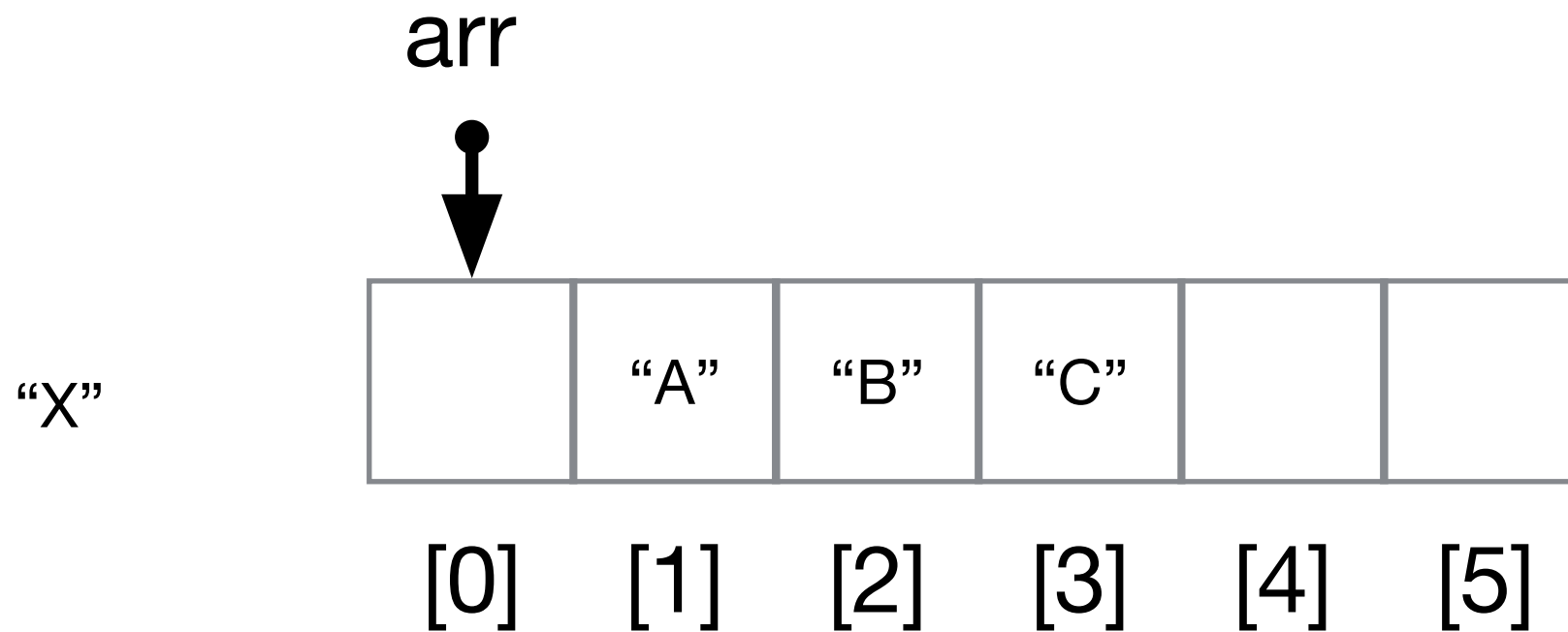
- If we need to insert a new element "X" into the **front** of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

# Benefits: Front Insertion



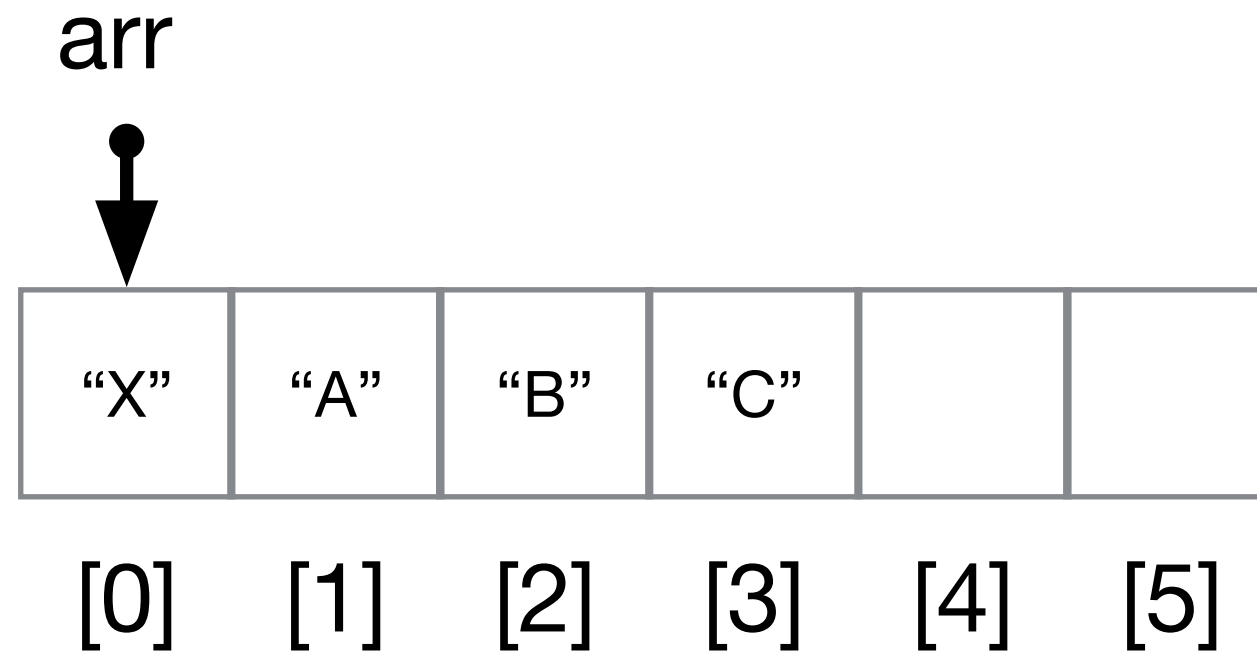
- If we need to insert a new element "X" into the **front** of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

# Benefits: Front Insertion



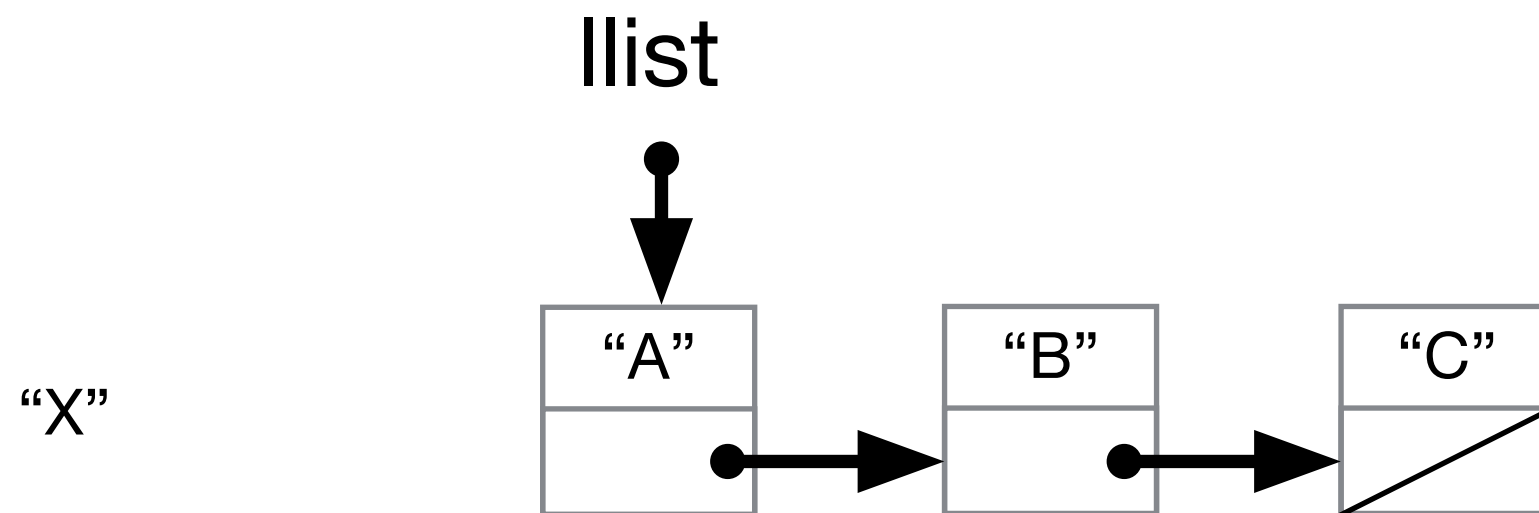
- If we need to insert a new element "X" into the **front** of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

# Benefits: Front Insertion



- If we need to insert a new element "X" into the **front** of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

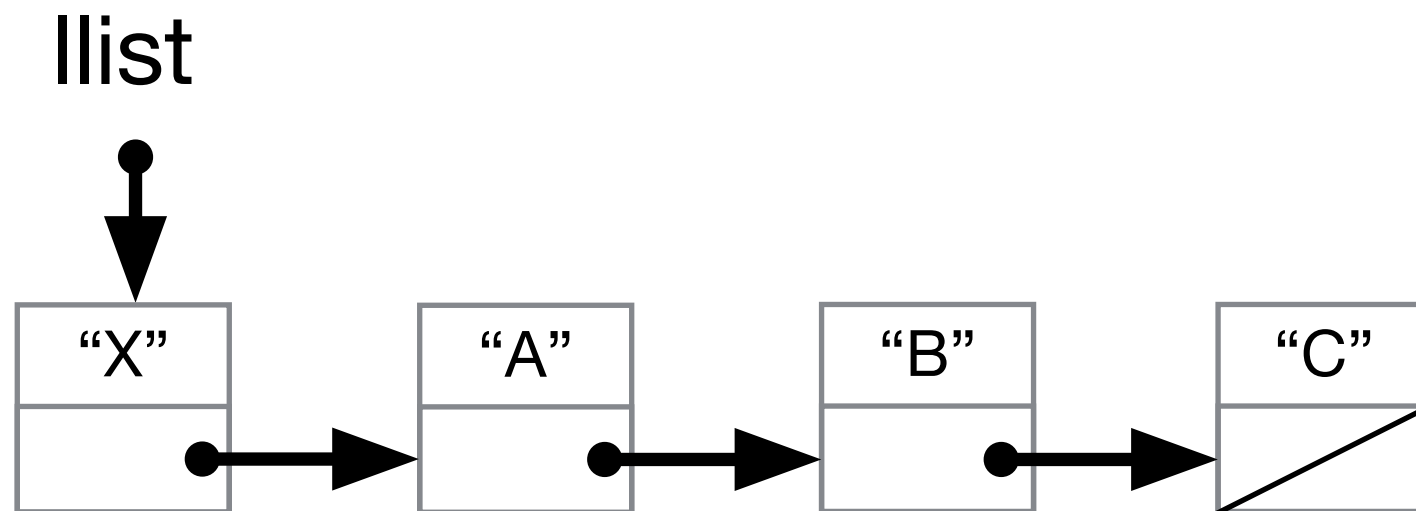
# Benefits: Front Insertion



- If we need to insert a new element “X” into the **front** of a linked list,



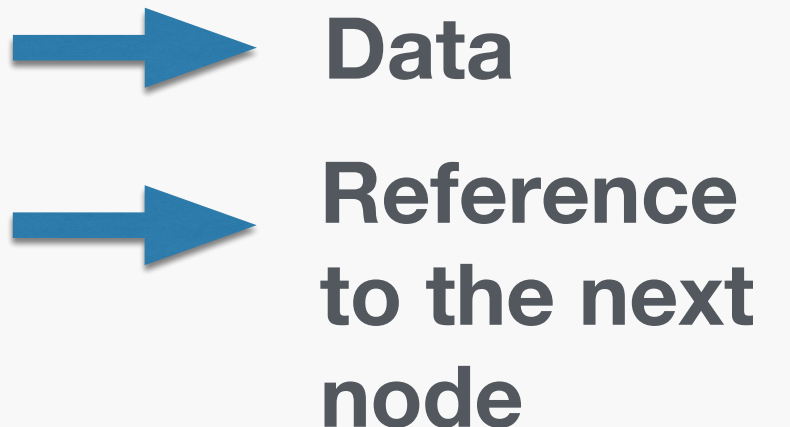
# Benefits: Front Insertion



- If we need to insert a new element "X" into the **front** of a linked list, we need only create a new node and adjust the reference. This takes just two steps, regardless of how many existing elements there are

# The LLStringNode Class

```
public class LLStringNode {  
    private String info;  
    private LLStringNode link;  
}
```



→ Data

→ Reference to the next node

- This is called **self-referential**: a class containing a reference to an object of the same class.
- What's going on here? Is this sort of recursive definition even allowed?
- How much space does the compiler allocate to the link variable?

# The LLStringNode Class

- Recall that in Java, a class-type variable is a reference (i.e. pointer) to an object, it does NOT contain the actual content of the object.
- This means the link variable merely stores a memory address. The size of the memory address is typically:
  - 4 bytes on 32-bit JVM
  - 8 bytes on 64-bit JVM

Regardless of what it's pointing to (String, Object, LLStringNode ...)

# The LLStringNode Class

```
public class LLStringNode {  
    private String info;  
    private LLStringNode link;  
  
    public LLStringNode(String info) {  
        this.info = info;  
        link = null;  
    }  
    public String getInfo() { return info; }  
    public LLStringNode getLink() { return link; }  
    public void setInfo(String i) { info = i; }  
    public void setLink(LLStringNode link) {  
        this.link = link;  
    }  
}
```

# The LLStringNode Class

```
public class LLStringNode {  
    public String info;  
    public LLStringNode link;  
  
    public LLStringNode(String info) {  
        this.info = info;  
        link = null;  
    }  
}
```

Alternatively, you can make the data members public, so you can access `info` and `link` directly without any methods.

# Clicker question #1

```
class NodeA {  
    private NodeB link;  
}  
class NodeB {  
    private NodeA link;  
}  
... ..  
NodeA node = new NodeA();
```

How many Node B objects have been instantiated?

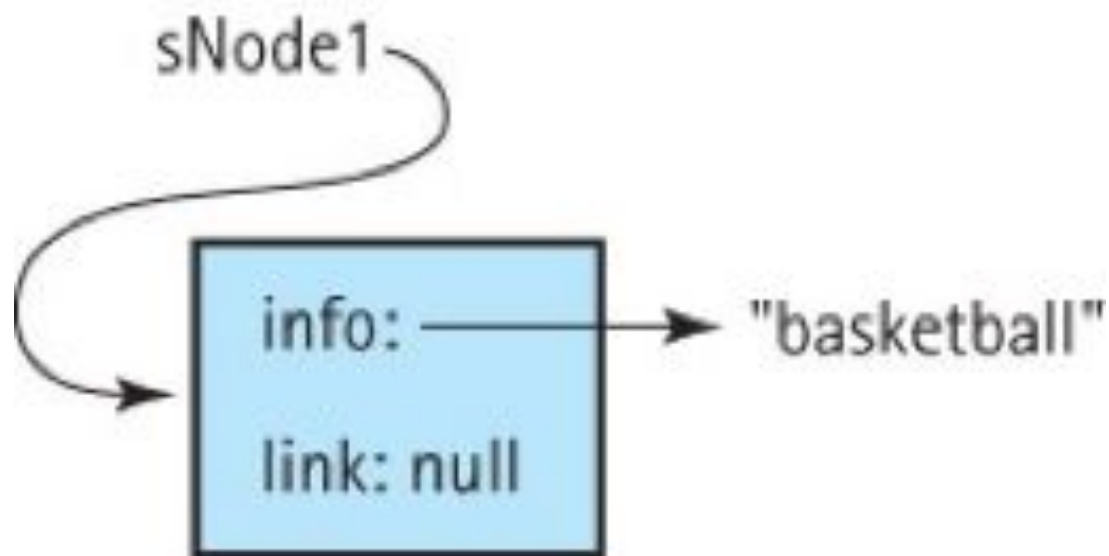
- a) 0
- b) 1
- c) infinite
- d) this won't compile

answer on next slide

# Using LLStringNode

Create the **first** node:

```
LLStringNode sNode1 = new LLStringNode("basketball");
```

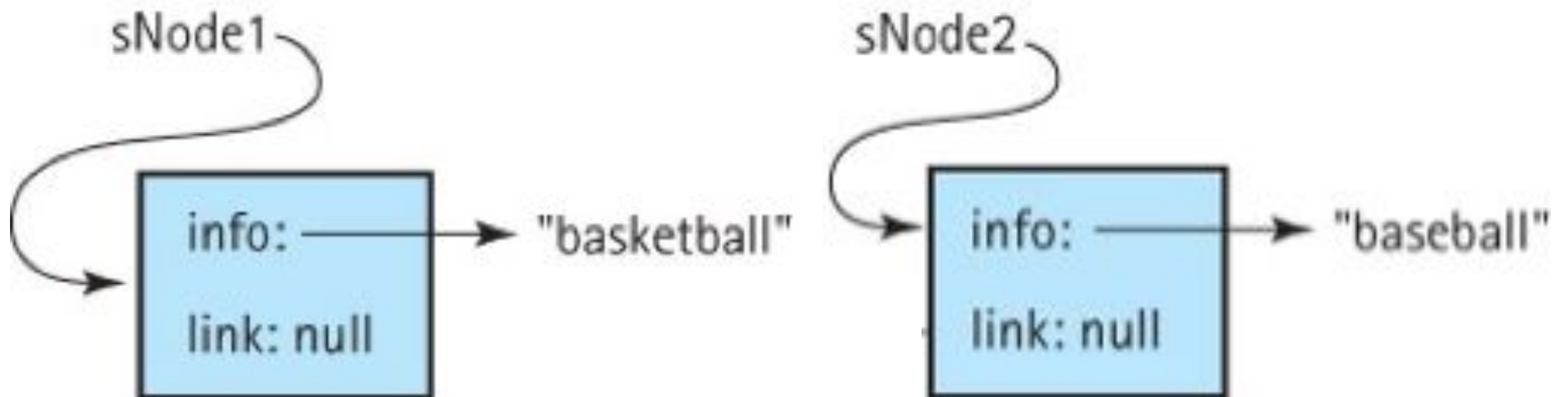




# Using LLStringNode

Create the **second** node:

```
LLStringNode sNode2 = new LLStringNode("baseball");
```



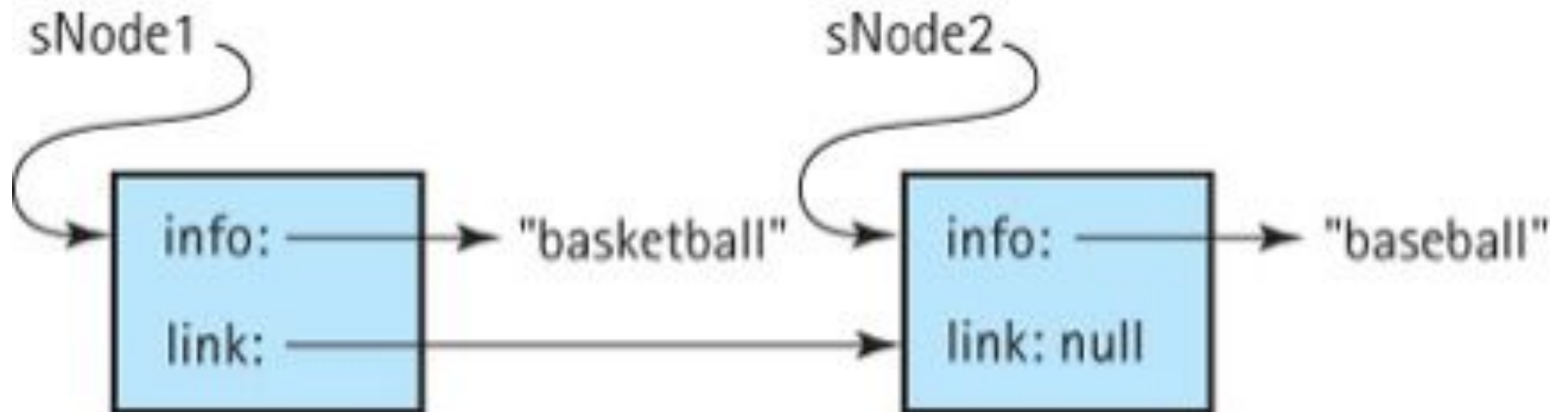
# Using LLStringNode

Create the **second** node:

```
LLStringNode sNode2 = new LLStringNode("baseball");
```

and add it to the chain:

```
sNode1.setLink(sNode2);
```



# Clicker Question #2

Suppose *x* is a node somewhere **in the middle** of a long linked list. What would the following code do?

```
x.getLink().setLink(null);
```

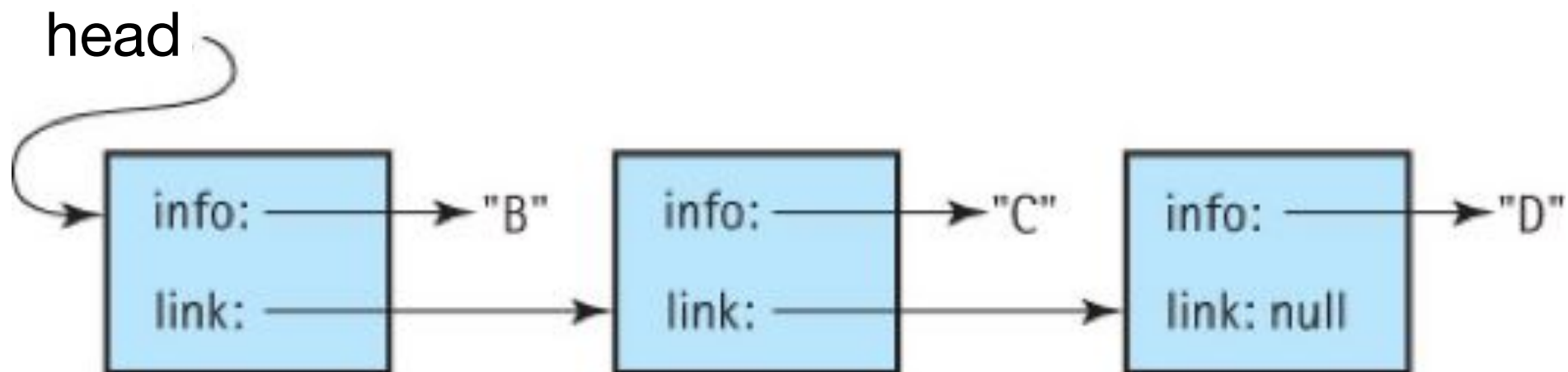
- (a) make *x* the second to last node in the list
- (b) remove the node immediately following *x* from the list, but no others
- (c) delete *x* itself from the list
- (d) make *x* the last node of the list
- (e) Michael Bay

answer on next slide

# Traverse a Linked List

Start from the first node (head), follow the chain, and make sure we don't run off the end of the list.

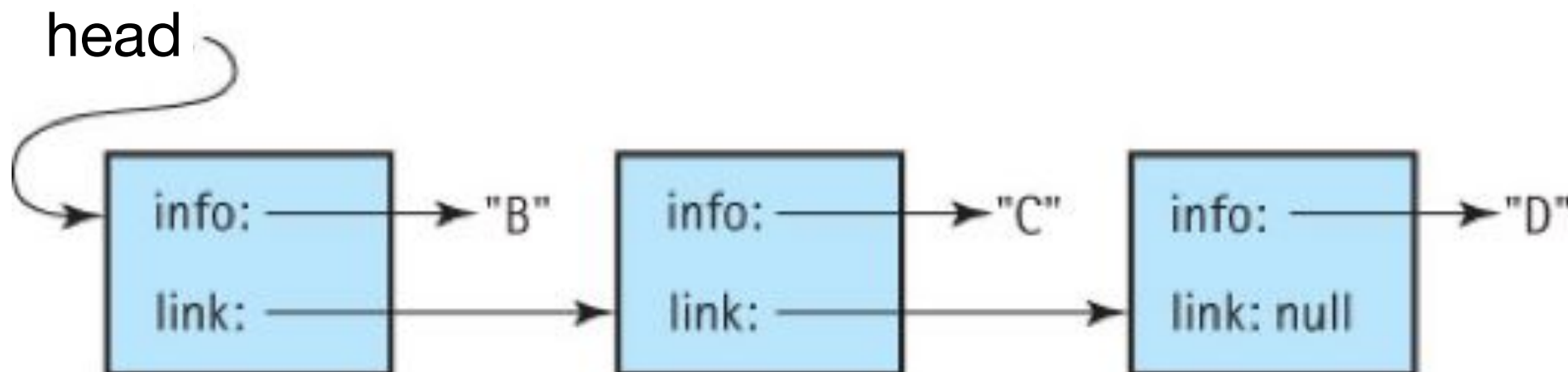
```
LLStringNode node = head;  
while (node != null) {  
    System.out.println(node.getInfo());  
    node = node.getLink();  
}
```



# Traverse a Linked List

Would this still work if the list is empty? (i.e. head is null)? This is an edge case worth considering.

```
LLStringNode node = head;  
while (node != null) {  
    System.out.println(node.getInfo());  
    node = node.getLink();  
}
```



# Arrays vs. Linked List

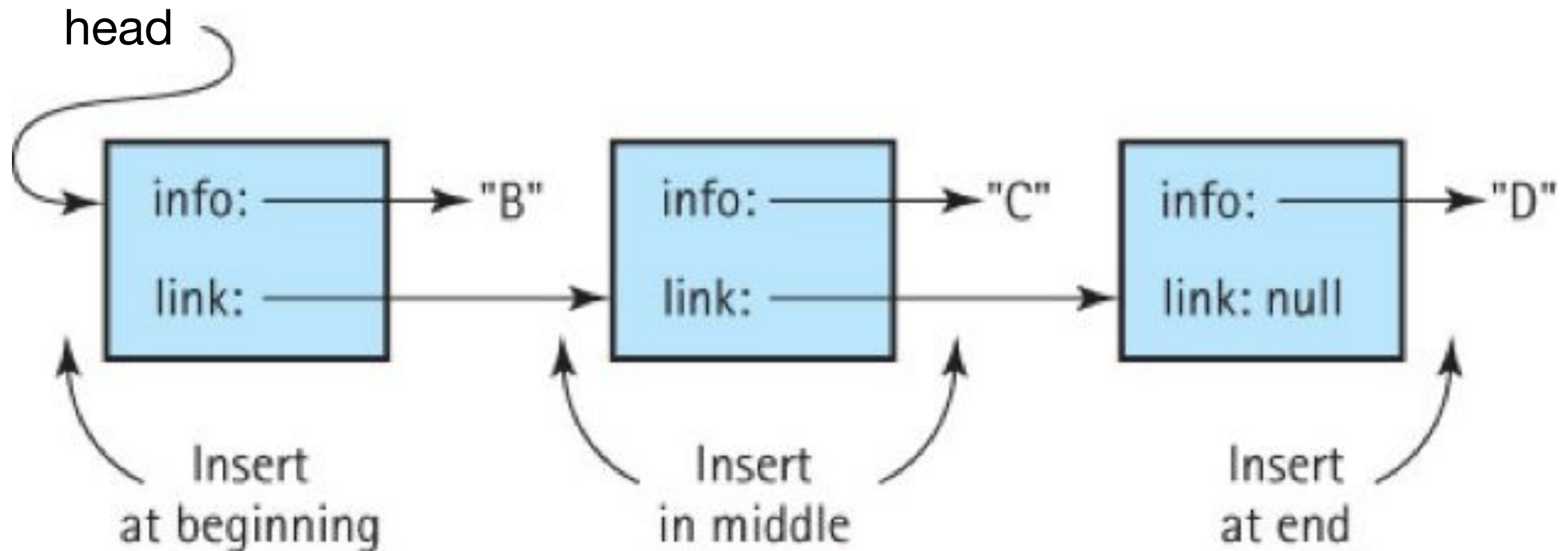
In a sense, traversing a linked list isn't all that much different from traversing an array. Compare the following:

```
LLStringNode node = head;
while (node != null) {
    System.out.println(node.getInfo());
    node = node.getLink();
}
```

```
int i = 0;
while (i != a.length) { // or < a.Length
    System.out.println(a[i]);
    i = i + 1;           // or i++;
}
```

# Linked List Insertion

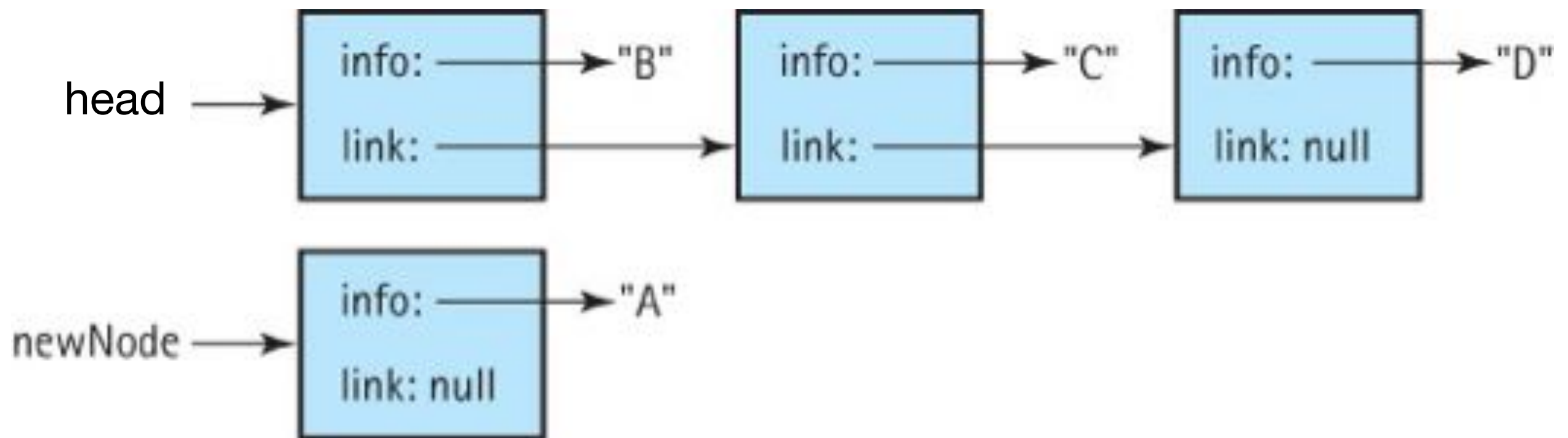
- There are three general cases of node insertion into a linked list. For now we will focus on **insertion at the front**, and **insertion at the end**.





# Linked List Insertion

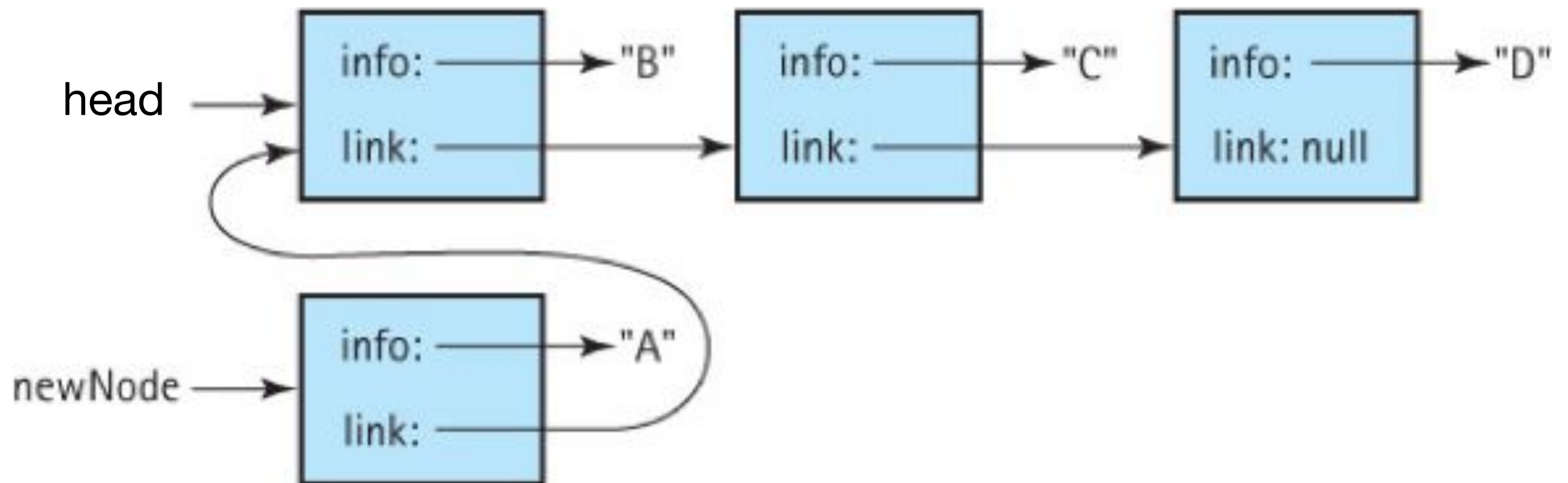
Suppose we have a newNode to insert at the front of this linked list. Basically what we want is to hook it up to the chain, and make it the first node (i.e. head).



# Linked List Insertion

**Step 1:**

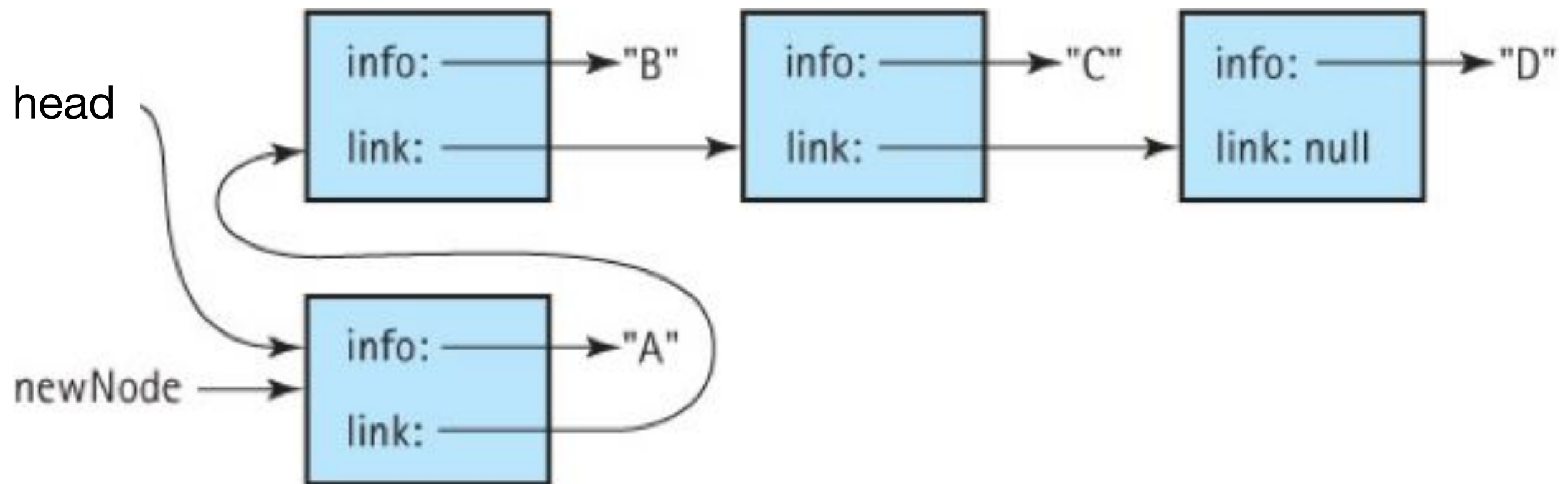
```
newNode.setLink(head);
```



# Linked List Insertion

**Step 1:** `newNode.setLink(head);`

**Step 2:** `head = newNode;`



# A few questions to ask

- What happens if insertion is called when the linked list is empty (i.e. `head==null`)? Any problem?
- What happens if we swap the two lines of code in insertion. In other words, what happens if we do:

```
head = newNode;  
newNode.setLink(head);
```

# A few questions to ask

- What happens if insertion is called when the linked list is empty (i.e. `head==null`)? Any problem?

**This is ok.**

- What happens if we swap the two lines of code in insertion. In other words, what happens if we do:

```
head = newNode;  
newNode.setLink(head);
```

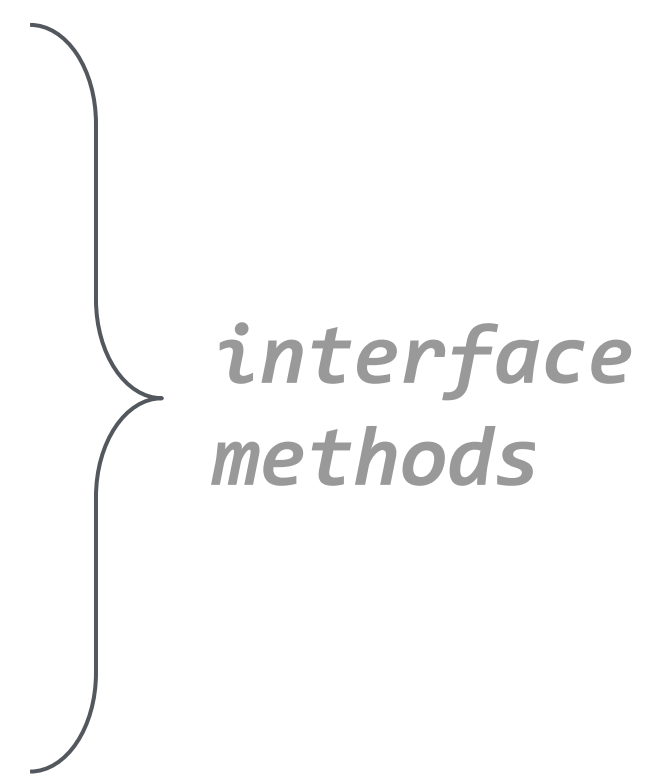


It's a good practice to do sanity checking and ensure  
Your program runs correctly under edge cases.

# Linked List Implementation of StringLog

```
public class LinkedStringLog
    implements StringLogInterface
{
    private LLStringNode log; // this is the head
    public LinkedStringLog(); // constructor

    public void insert(String e);
    public boolean isFull();
    public int size();
    public boolean contains(String e);
    public void clear();
    public String getName();
    public String toString();
}
```



*interface methods*

# Implementing the insert method

- The StringLog description didn't specify whether the strings need to be stored in a particular order. So we can use front insertion of linked list.

```
public void insert (String element) {  
    LLStringNode newNode = new LLStringNode(element);  
    newNode.setLink(log);  
    log = newNode;  
}
```

# Observers

- `getName()` is the same as `ArrayStringLog`.
- `isFull()` is even simpler: because it always returns `false` (i.e. capacity is unbounded)

```
public String getName( ) {  
    return name;  
}  
public boolean isFull( ) {  
    return false;  
}
```

- As for `size()`, one way is to traverse the linked list to count the number of nodes in the list.



# Observers

```
public int size( ) {  
    int count = 0;  
    LLStringNode node = log;  
    while (node != null) {  
        count++;  
        node = node.getLink( );  
    }  
    return count;  
}
```

# Clicker Question #3

```
public int size( ) {  
    int count = 0;  
    LLStringNode node = log;  
    while (node != null) {  
        count++;  
        node = node.getLink( );  
    }  
    return count;  
}
```

What would happen if we swap the two lines inside the while loop?

- (a) nothing would change at all
- (b) a NullPointerException
- (c) count would be too small by 1
- (d) count would be too large by 1

**answer on next slide**

# Observers

```
public int size( ) {  
    int count = 0;  
    LLStringNode node = log;  
    while (node != null) {  
        count++;  
        node = node.getLink( );  
    }  
    return count;  
}
```

- A much simpler way is to define a integer variable to track the number of elements that exist in the list. `insert()` will increment it, `clear()` will reset it, and `size()` simply returns the value of it.

# Code for contains

- For contains, we also need to traverse the list.
- This time we can terminate the loop as soon as the string to search is found.

```
public boolean contains (String element) {  
    LLStringNode node = log;  
    while (node != null) {  
        if (element.equalsIgnoreCase(node.getInfo()))  
            return true;  
        else node = node.getLink( );  
    }  
    return false;  
}
```

# Code for toString

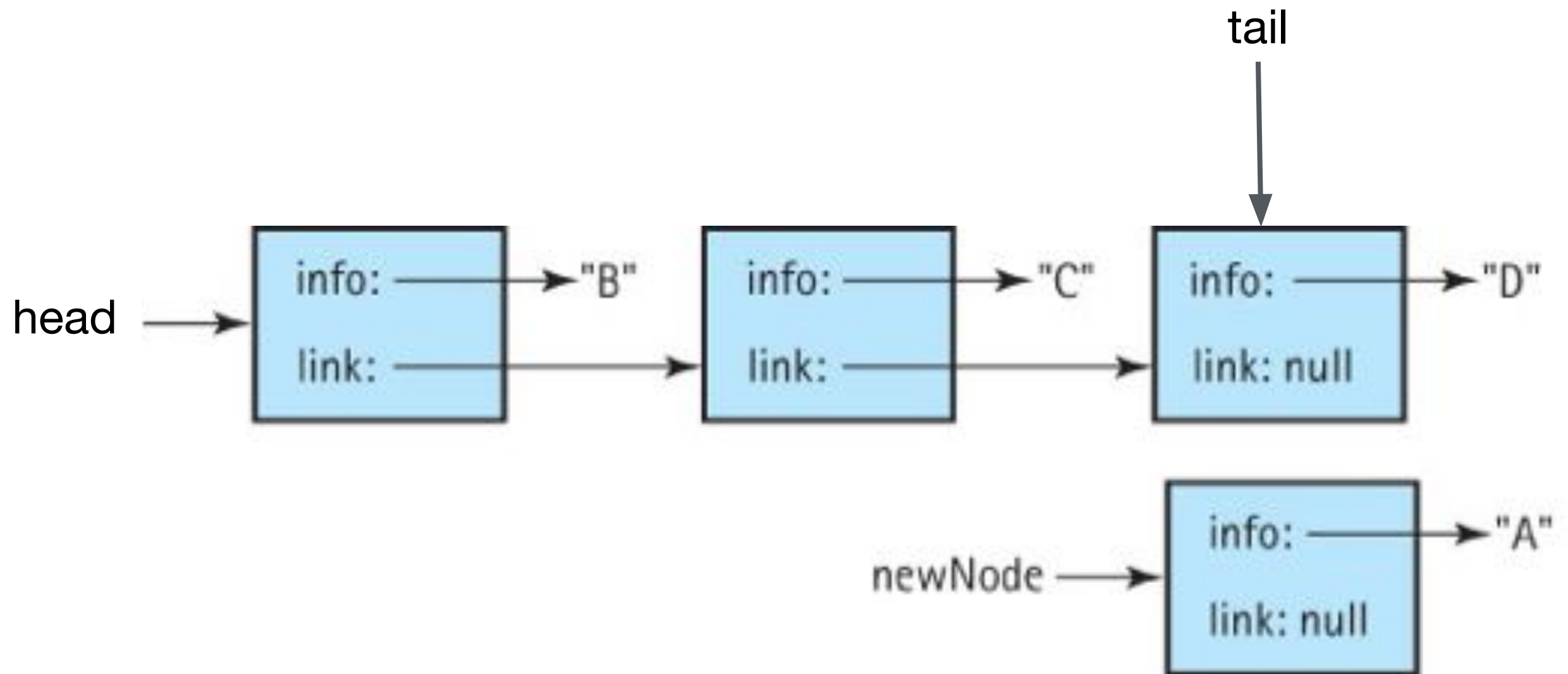
```
public String toString( ) {  
    String logString ="Log: " + name + "\n\n";  
    LLStringNode node = log;  
    int count = 0;  
    while (node != null) {  
        count++;  
        logString += count + ". " +  
            node.getInfo( ) + "\n";  
        node = node.getLink( );  
    }  
    return logString;  
}
```

# Insertion at the End

- One issue with **front insertion** is that when traversing the linked list, you will visit elements in the reverse order as they are inserted. So `toString()` will print out elements in reverse order.
- This can be solved by modifying the `insert` method to always **inserts at the end** of the list.
- How do you implement **end insertion**? What's the most efficient solution?

# Insertion at the End

- Keep a **tail** pointer that points to the last node in the list (akin to `lastIndex` in the `ArrayLog` case).
- To insert a `newNode`...



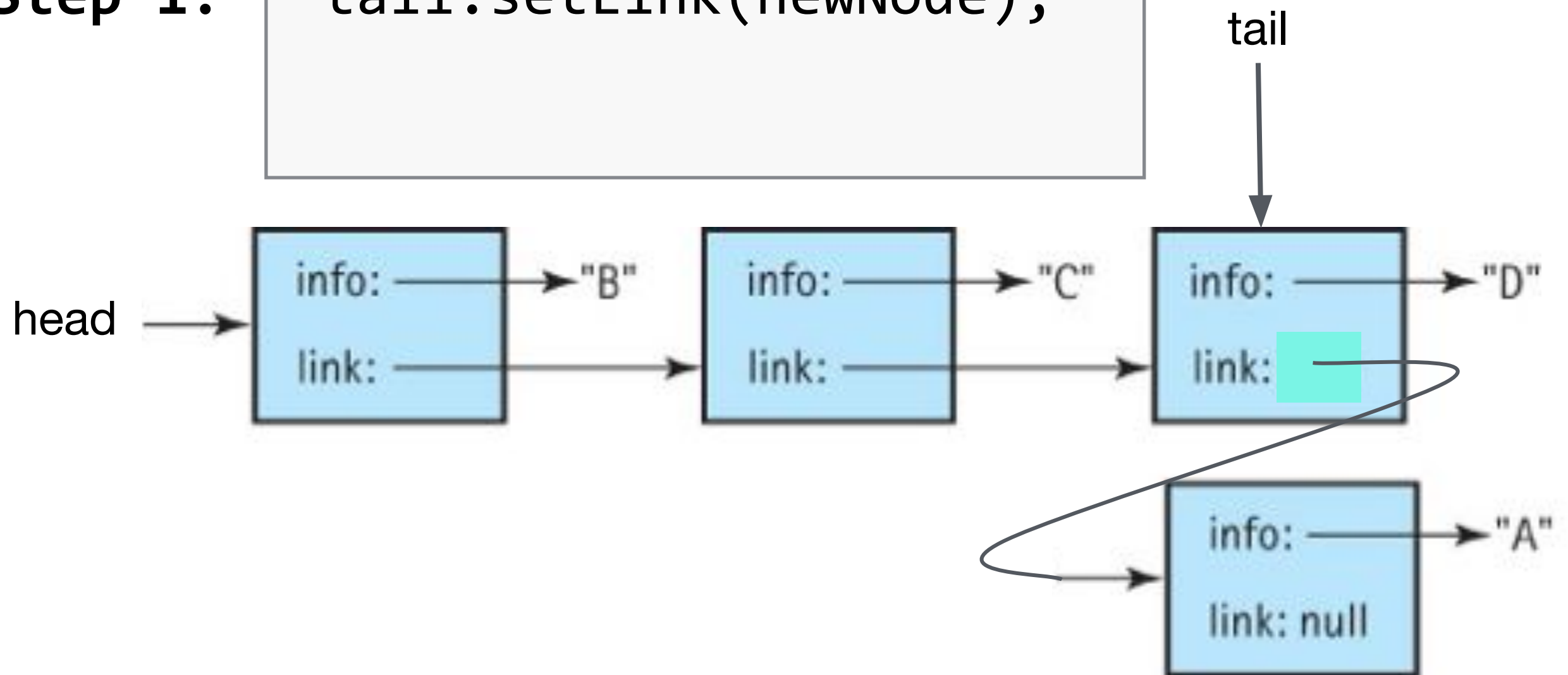


# Insertion at the End

- Step 1: chain the new node to the end of the list.

**Step 1:**

```
tail.setLink(newNode);
```

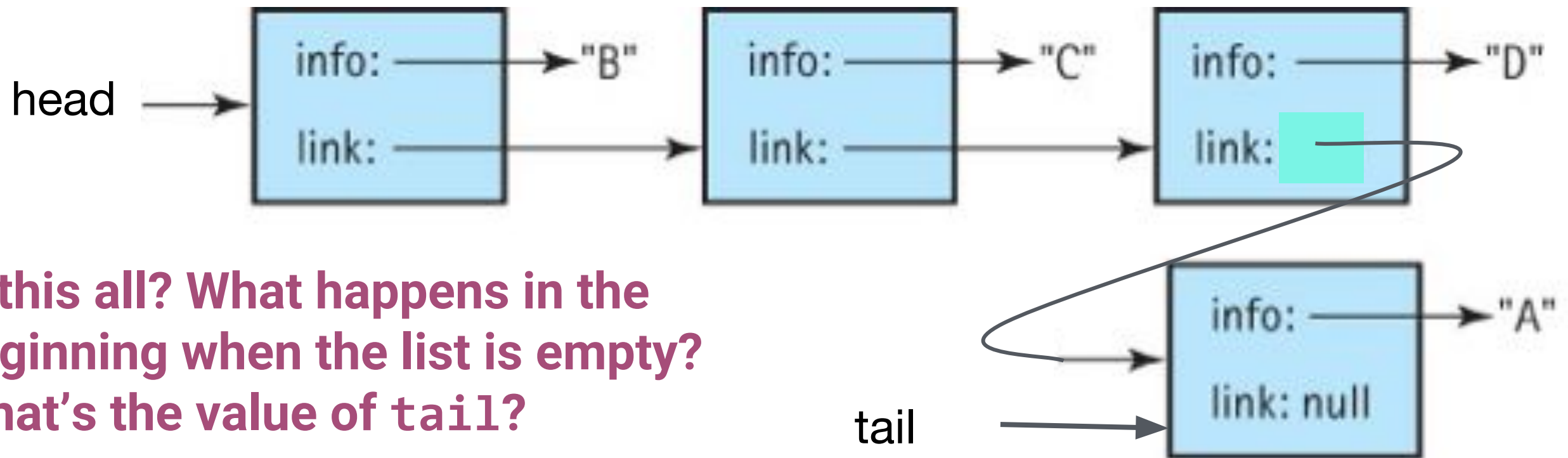


# Insertion at the End

- Step 2: update the tail pointer.

**Step 1:** `tail.setLink(newNode);`

**Step 2:** `tail = newNode;`



Is this all? What happens in the beginning when the list is empty?  
What's the value of tail?

# Insertion at the End

```
public class LinkedListLog implements StringLogInterface
{
    LLStringNode log, tail; // stores both head and tail
    public void insert(String element) { // end-insertion
        LLStringNode newNode = new LLStringNode(element);
        if (tail==null) // if list is empty
            log = newNode;
        else
            tail.setLink(newNode);
        tail = newNode; // tail always points to the new node
    }
}
```

**What's head and tail pointing to when the list has exactly one element?**