

Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
 - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

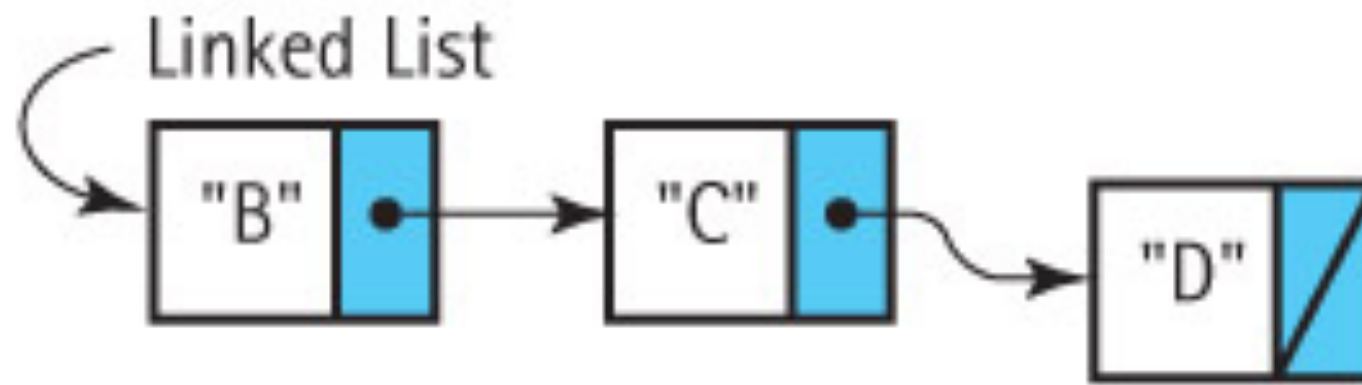
Reminders

- Read course webpage.
- Make sure you've received a Piazza invitation by email and that you've logged in.
- Get iClicker **2** and register it in Moodle.
- **Assignment 2 is due next week (this one is harder).**

Linked StringLogs

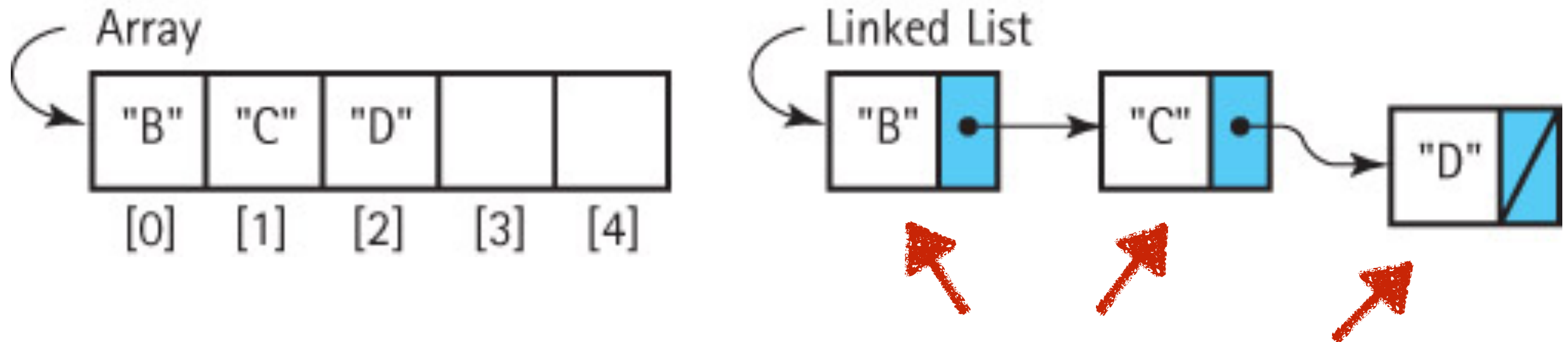
- The Linked List Data Structure
- Arrays vs. Linked Lists
- The `LLStringNode` Class
- Operations on a Linked List
- Implementing `LinkedListStringLog`

Linked List



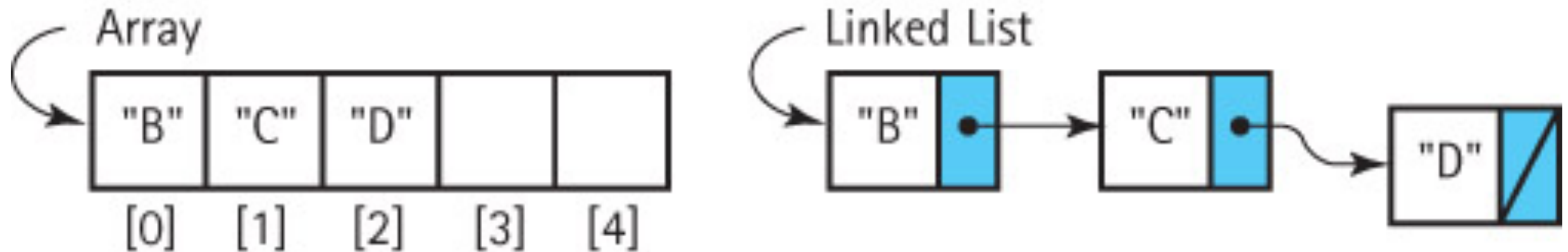
- A series of **nodes** chained together.
- Each node contains a **data element** and a **link** (i.e. reference / pointer) to the next node in the chain.
- If we know the head of the chain, we can follow successive links to reach any node on the chain.
- The last node has a **null** pointer, because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

Comparison to Arrays



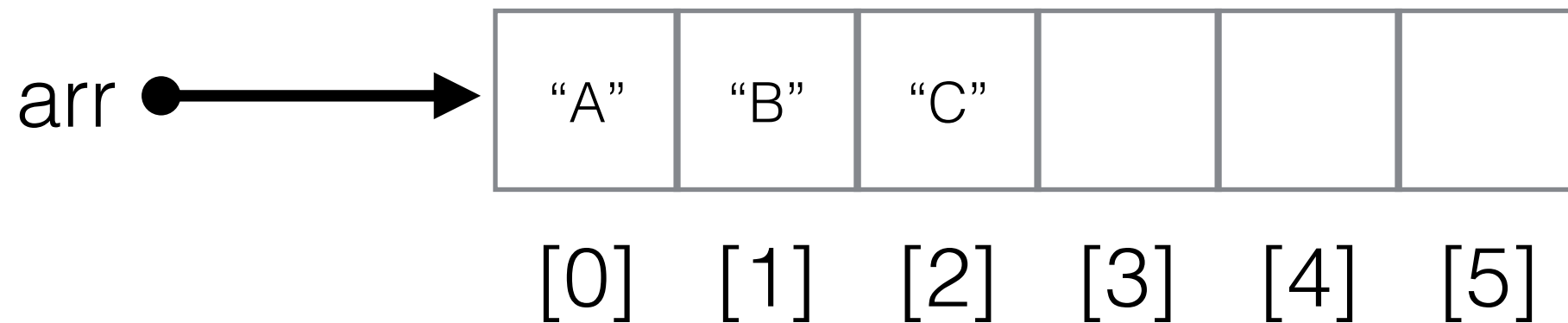
- Arrays store all data elements consecutively in memory. This makes it easy to access any element using its index.
- In a linked list, each element is separate in its own block of memory (i.e. a node). There is no easy way to directly access an element using its index. Imagine trying to find someone's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

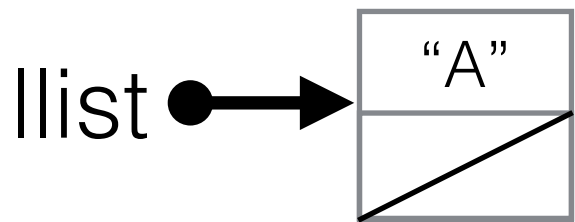
Benefits: Size/Space



- The size of an array is fixed (bounded)

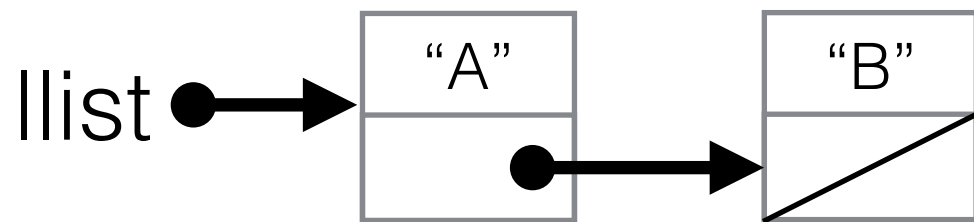
```
char[] arr = new char[6];
```

Benefits: Size/Space



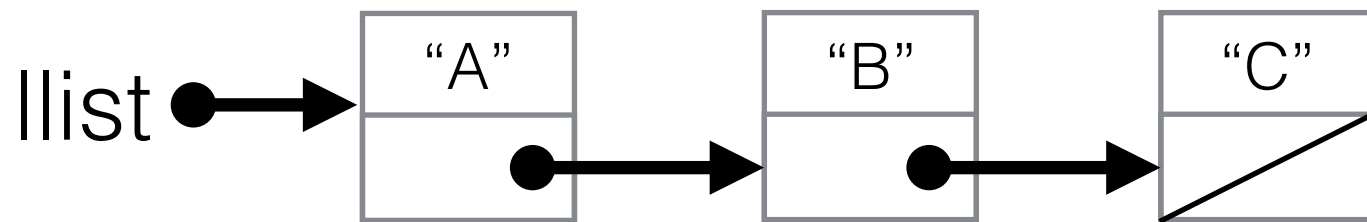
- The size of a linked list is not fixed (unbounded)
- Allocate node "A" (size 1)

Benefits: Size/Space



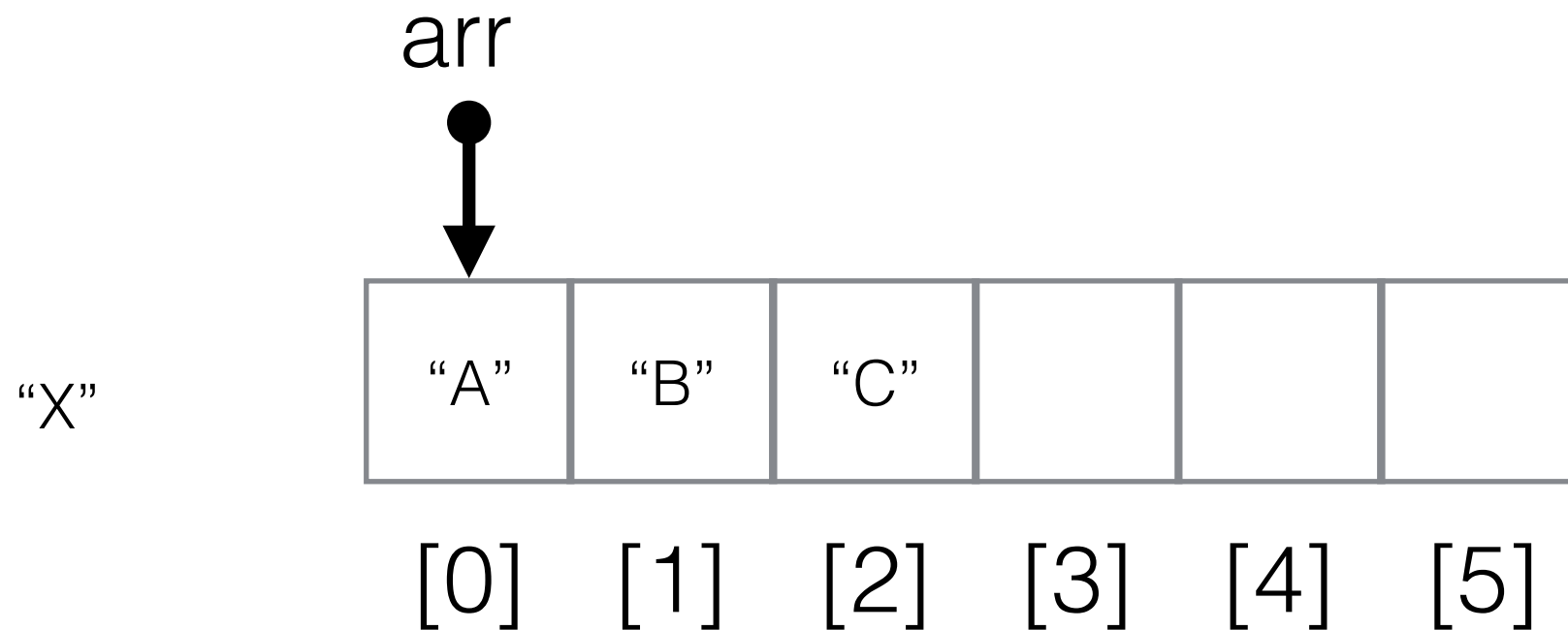
- The size of a linked list is not fixed (unbounded)
- Allocate node "B" (size 2)

Benefits: Size/Space



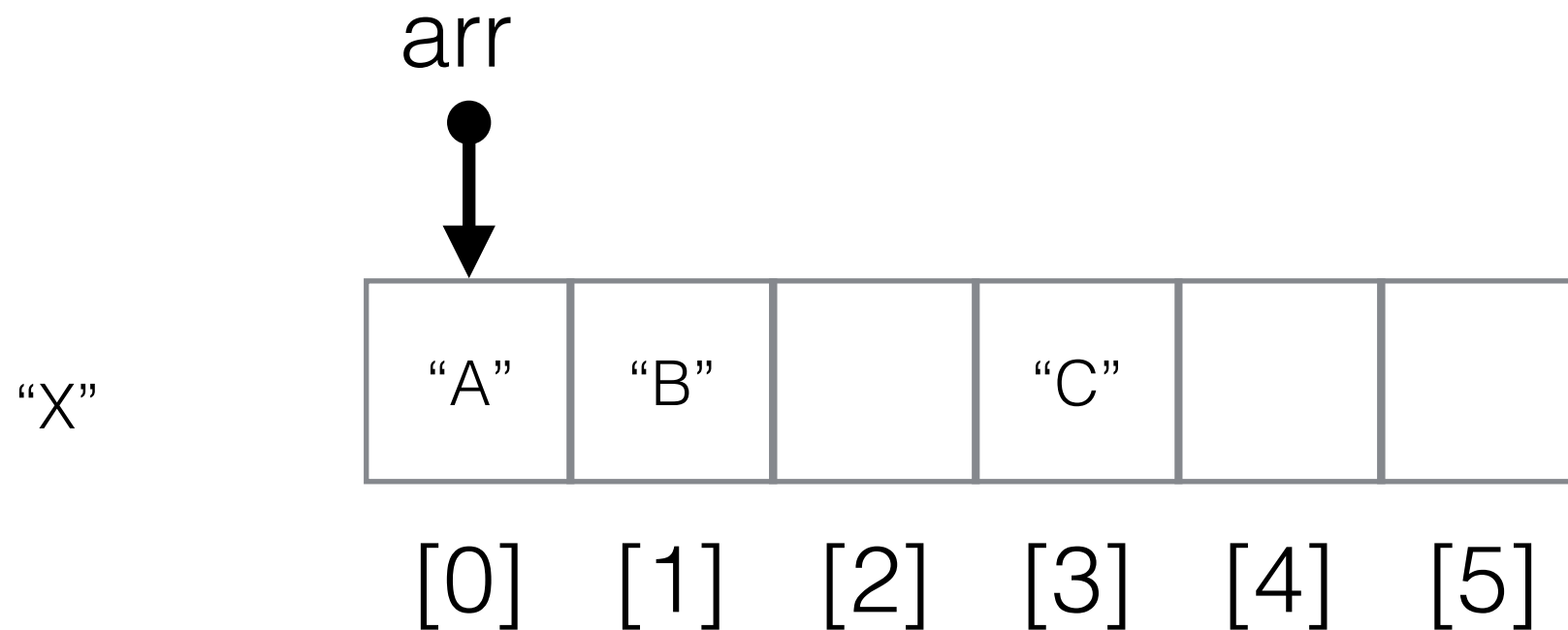
- The size of a linked list is not fixed (unbounded)
- Allocate node "C" (size 3)
- There are many applications where you can't predict, ahead of time, how many elements you will end up storing. This is where Linked List wins.

Benefits: Insertion



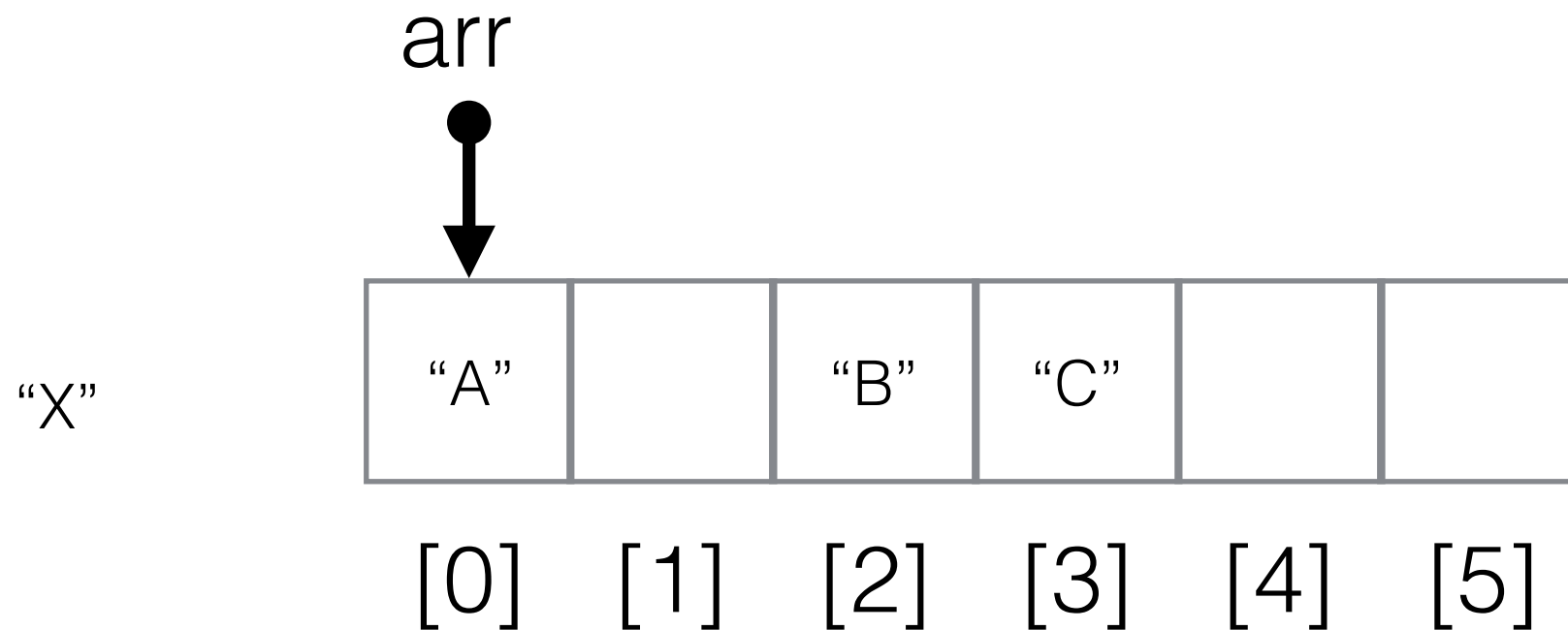
- If we need to insert a new element "X" into the front of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

Benefits: Insertion



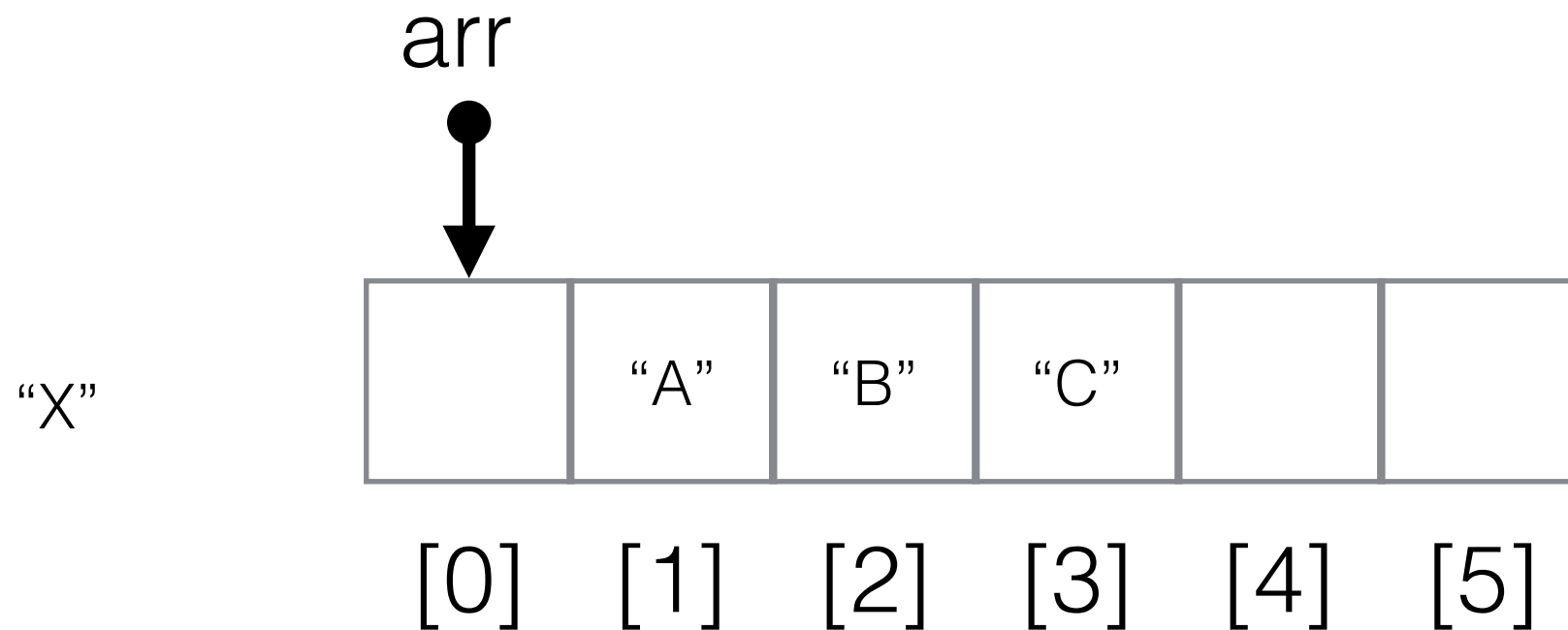
- If we need to insert a new element "X" into the front of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

Benefits: Insertion



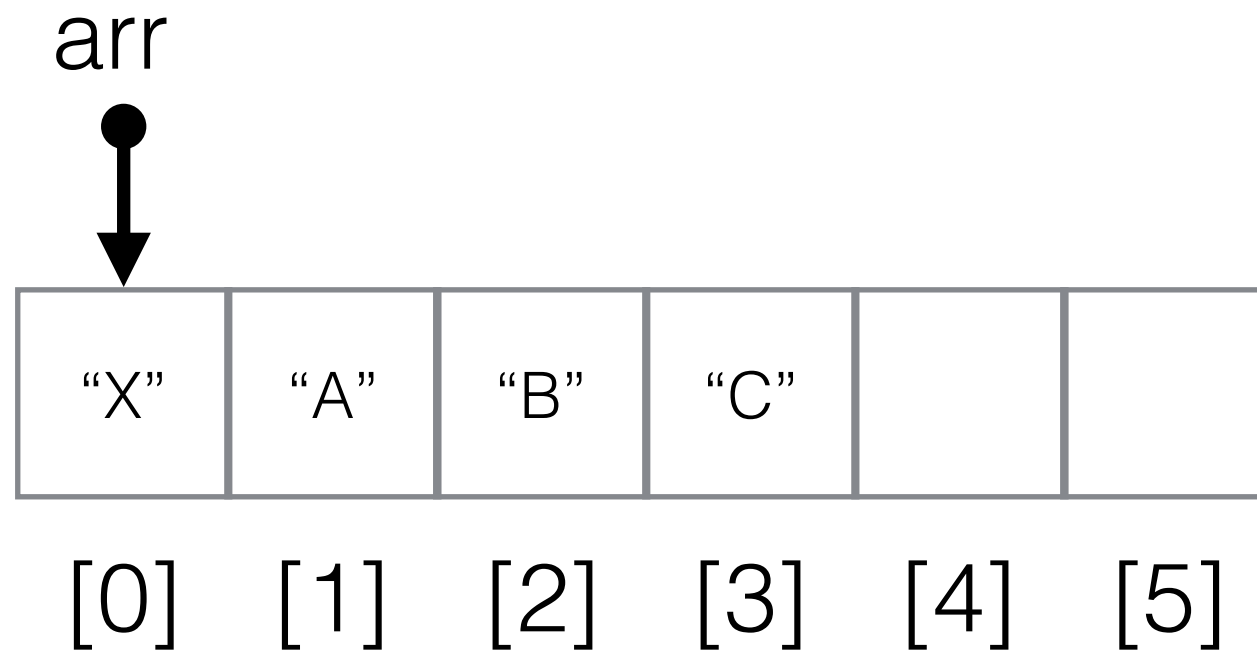
- If we need to insert a new element "X" into the front of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

Benefits: Insertion



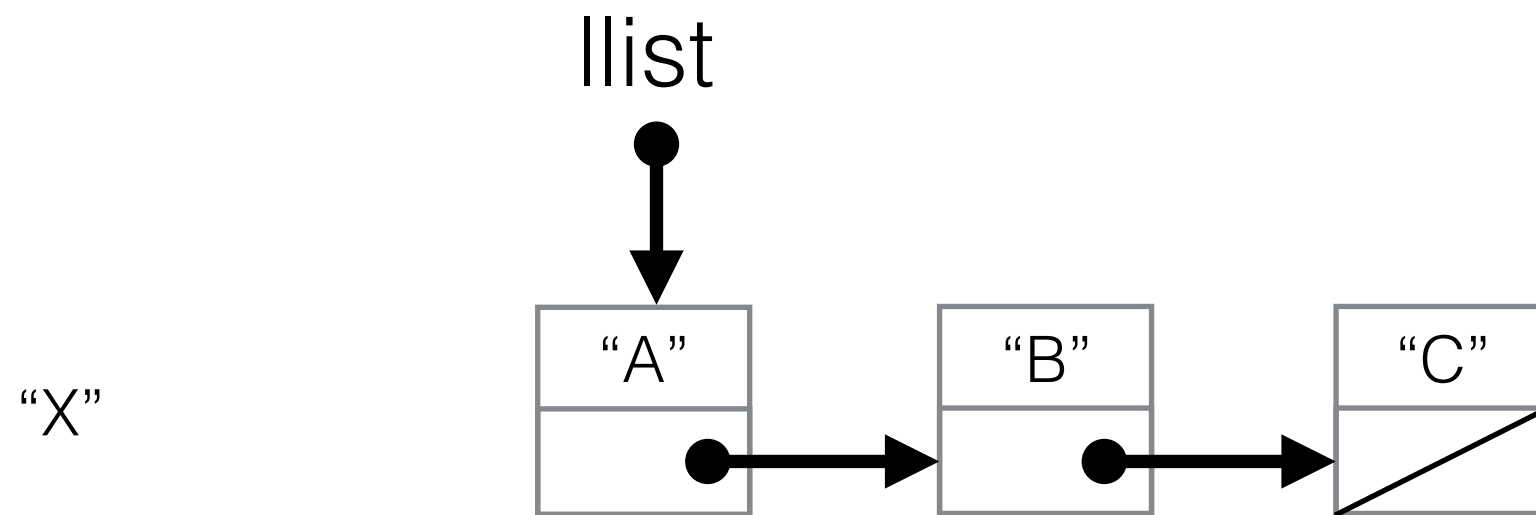
- If we need to insert a new element "X" into the front of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

Benefits: Insertion



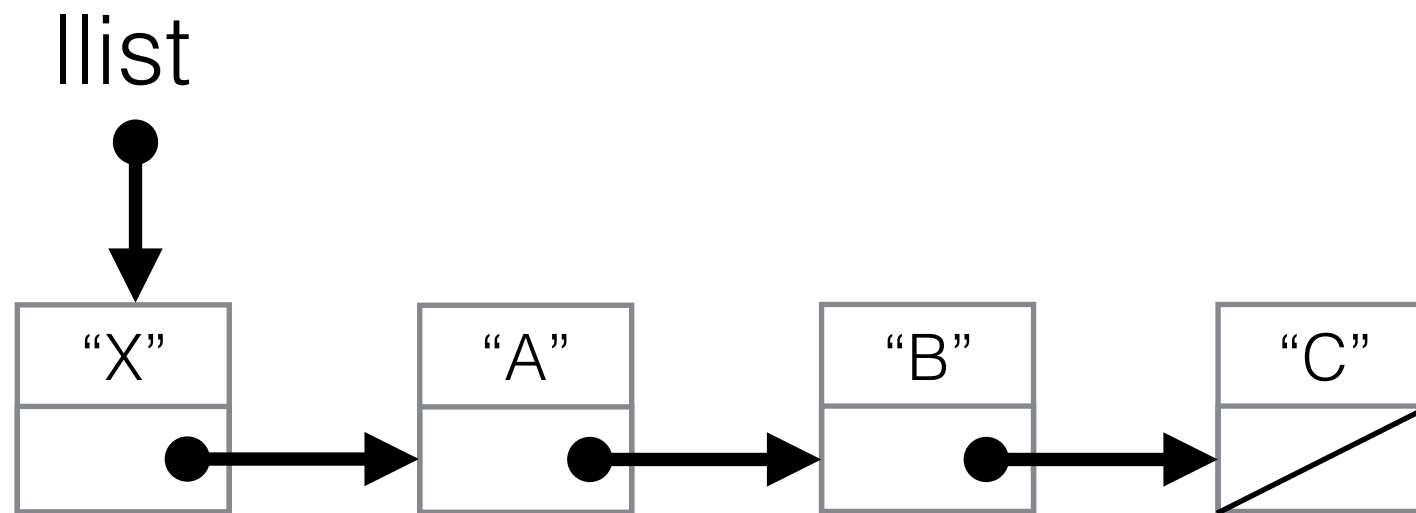
- If we need to insert a new element "X" into the front of an array, we need to shift all elements to make room at [0]. The more existing elements there are, the more steps it takes.

Benefits: 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

Benefits: 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;           → Data  
    private LLStringNode link;    → 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 data 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` ...)

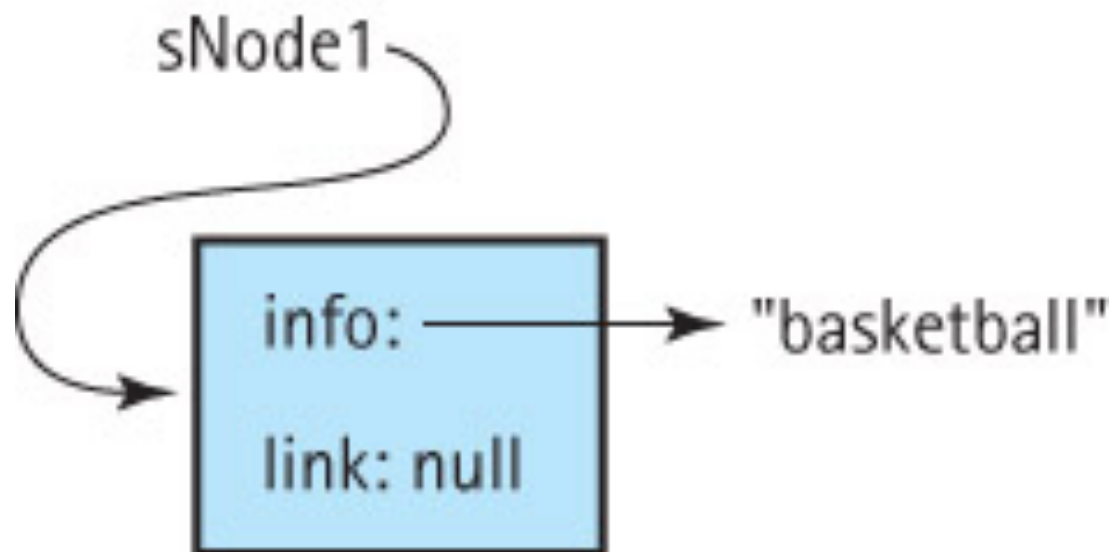
The LLStringNode Class

```
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 lk) {  
        link = lk;  
    }  
}
```

Using the LLStringNode class

Create the **first** node:

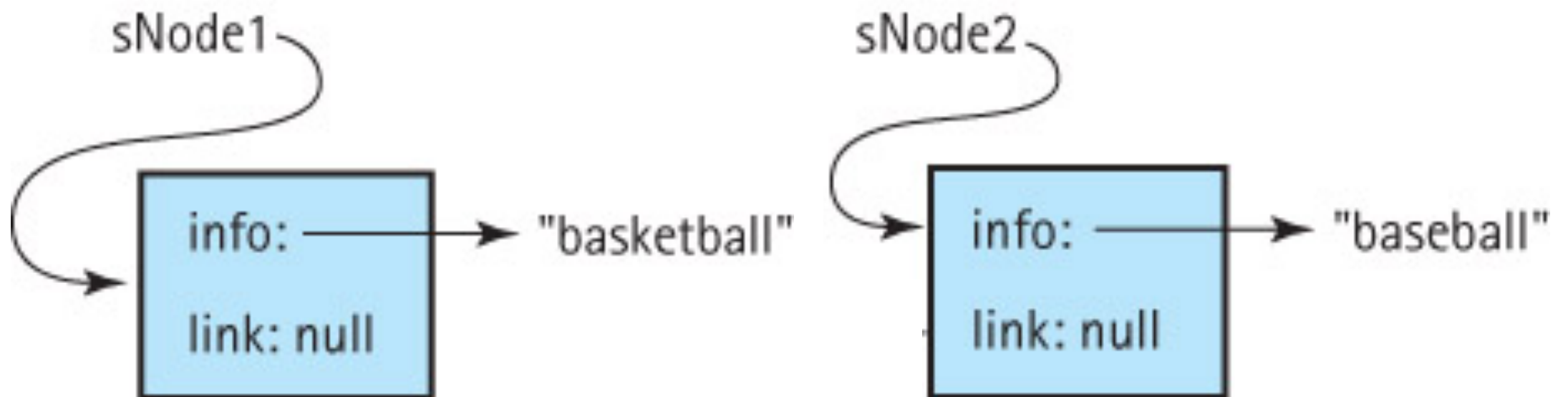
```
LLStringNode sNode1 = new LLStringNode("basketball");
```



Using the LLStringNode class

Create the **second** node:

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



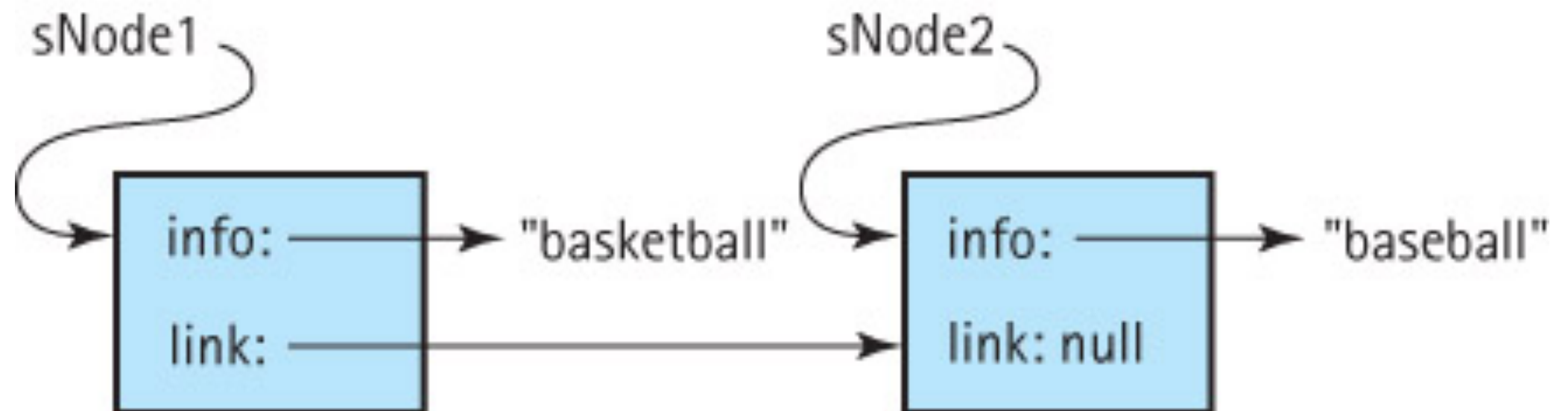
Using the LLStringNode class

Create the **second** node:

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

and add it to the chain:

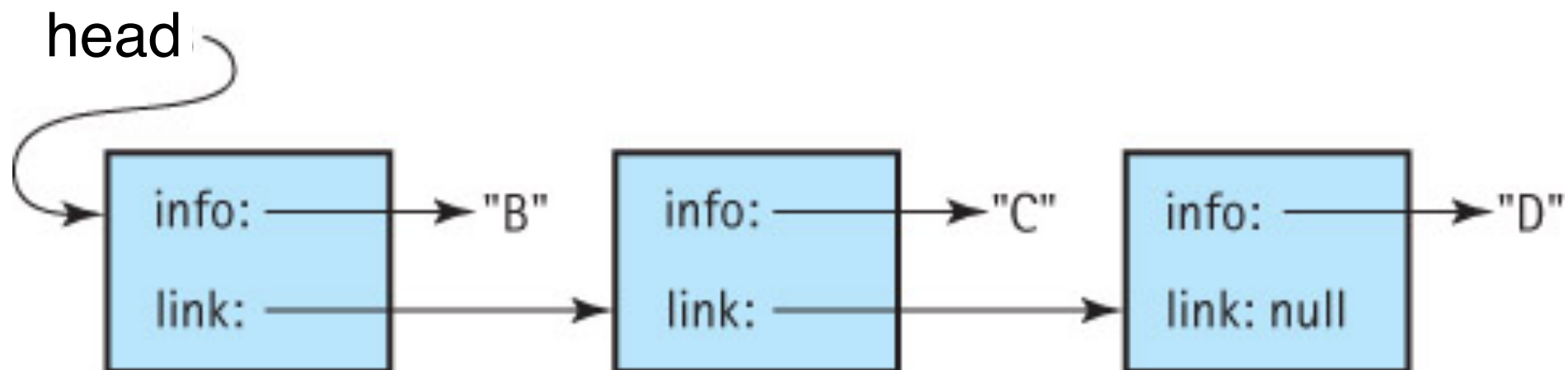
```
sNode1.setLink(sNode2);
```



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.

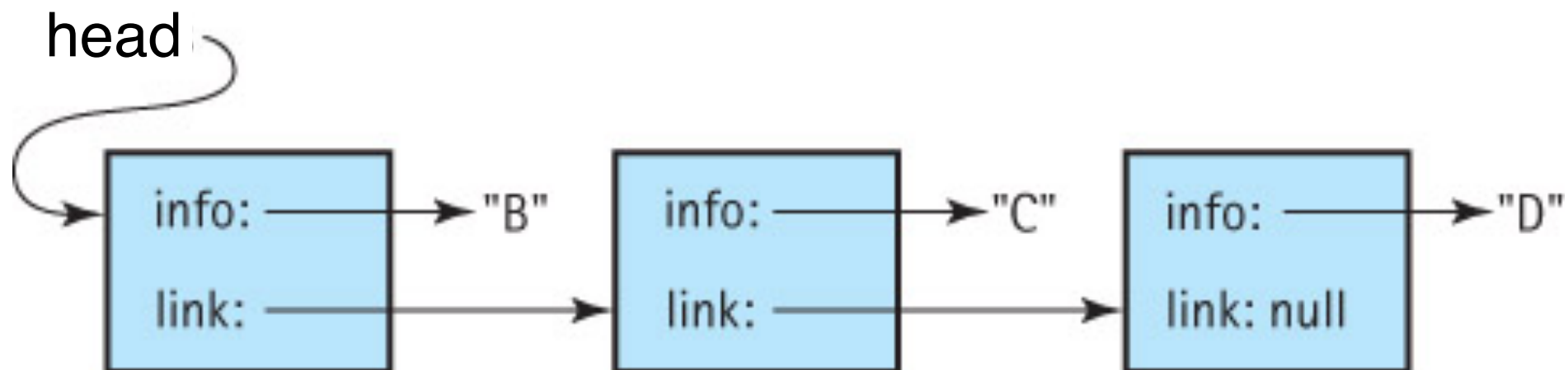
```
LLStringNode currNode = head;  
while (currNode != null) {  
    System.out.println(currNode.getInfo());  
    currNode = currNode.getLink();  
}
```



Traverse a Linked List

Would this throw out any error if head==null (i.e. an empty linked list)?

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



Arrays vs. Linked List

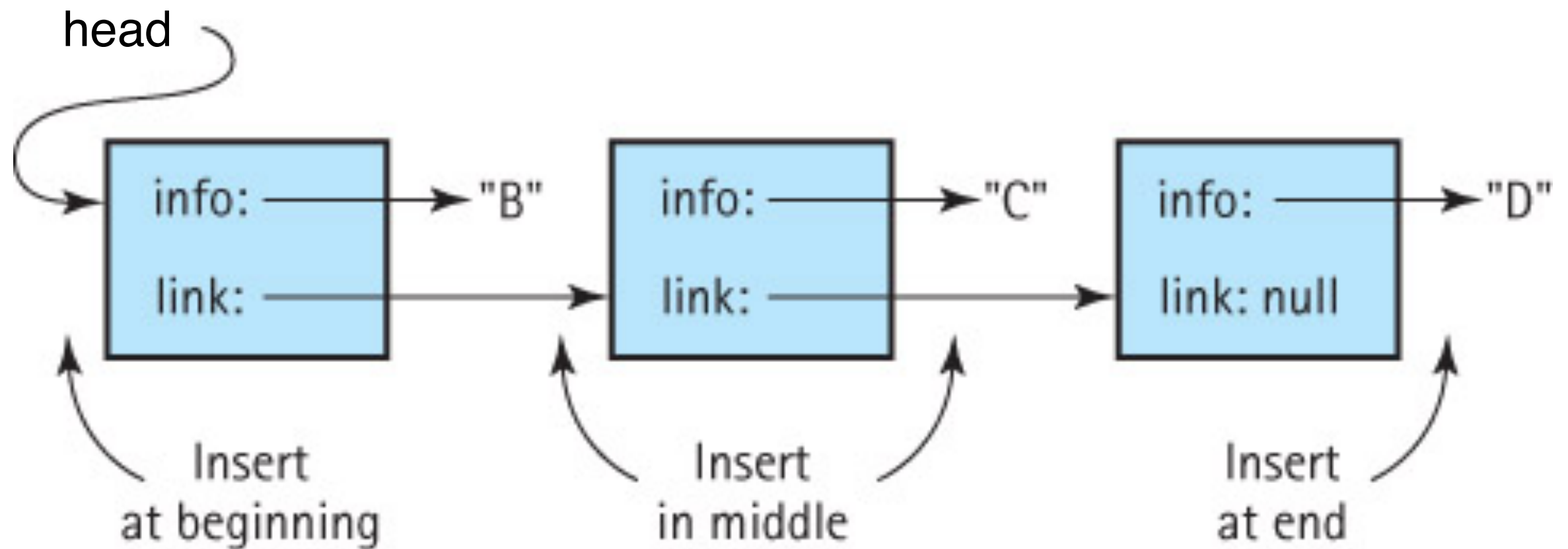
In a sense, traversing a linked list isn't all that much different from traversing an array. Compare the following:

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

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

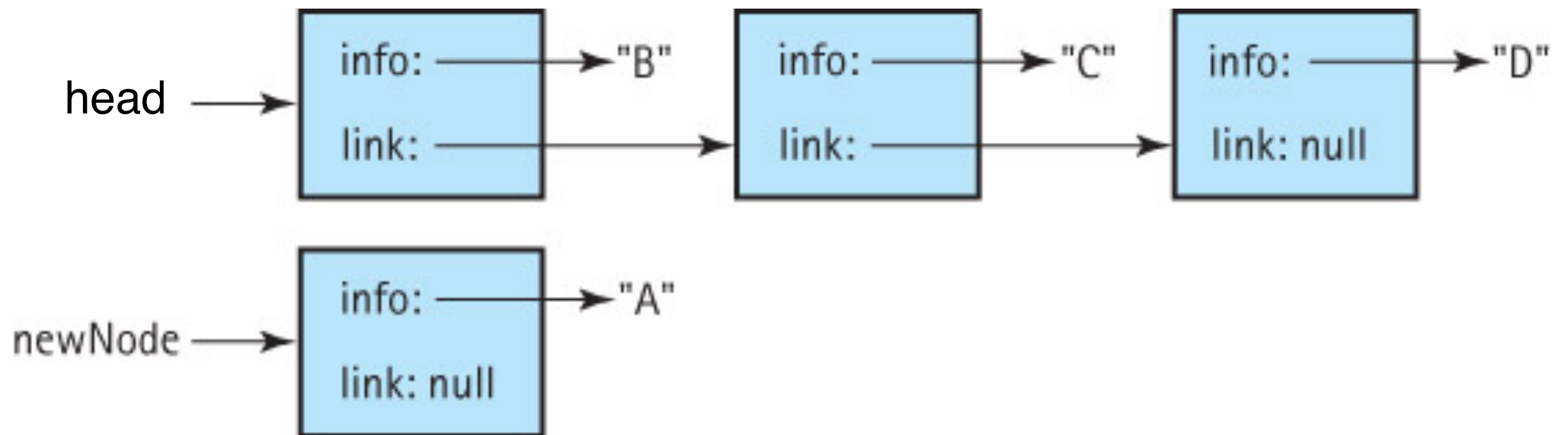
Linked List Insertion

- There are three general cases of node insertion into a linked list. For now we will focus on the first case: insertion at the beginning of the list.



Linked List Insertion

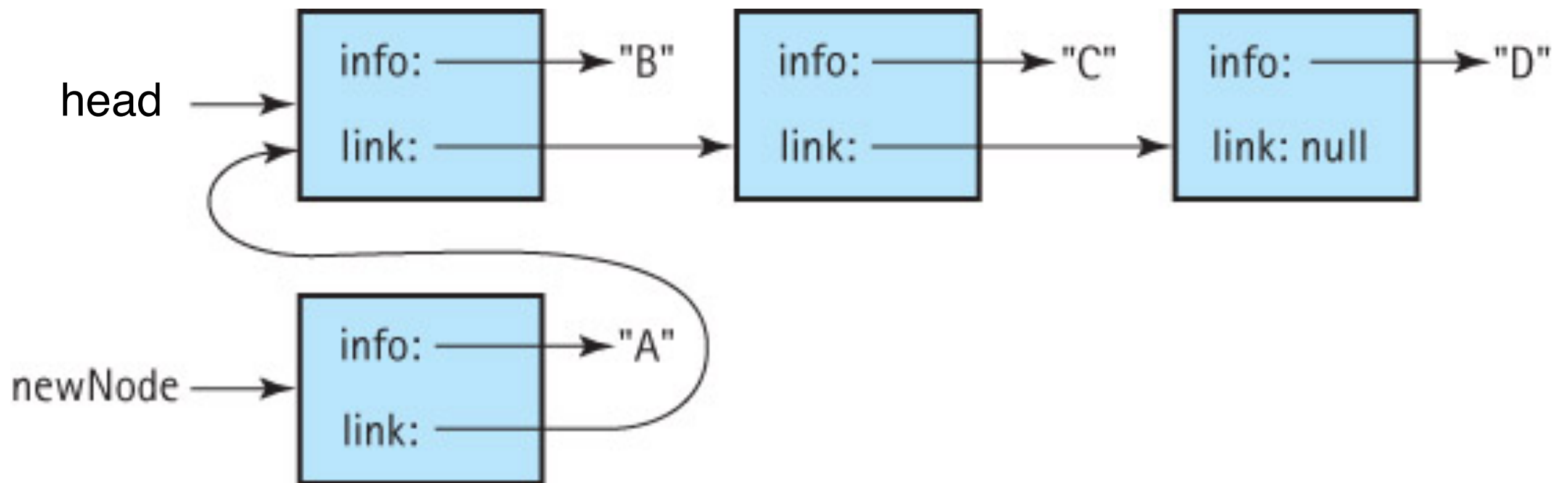
Suppose we have a `newNode` to insert at the beginning of this linked list. Basically what we want is to hook it up to the chain, and make it the first node (i.e. head).



Linked List Insertion

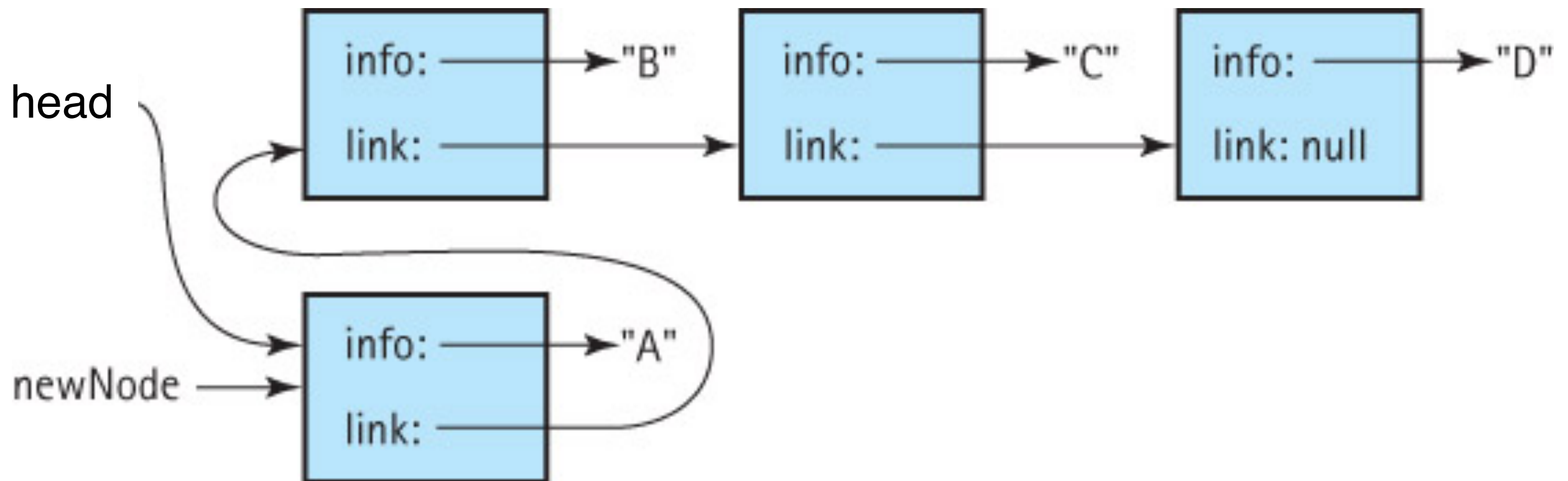
Step 1:

```
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 boundary conditions.

Implementing StringLog using Linked List

- At the last lecture, we learned how to implement the StringLog using an array. Now we will implement it using Linked List. The data members are:

```
public class LinkedStringLog implements StringLogInterface {  
    protected LLStringNode log;  
    protected String name;  
}
```

- Recall that the StringLogInterface contains 2 transformer methods and 5 observer methods, which we need to implement next.

Constructors

```
public LinkedStringLog (String name) {  
    log = null;  
    this.name = name;  
}
```

- Since LinkedList does not limit the capacity, we only need one constructor.
- Note that the first line `log=null;` is technically not necessary as Java initializes references to null automatically.

Transformers

- The clear operation simply sets the head of the list `log` to `null`.
- How about wiping the String elements (i.e. setting them to `null`) like in the Array-based implementation? Don't we need to do it anymore?

```
public void clear( ) {  
    log = null;  
}
```

Transformers

- As for `insert`, the `StringLog` description didn't specify whether the strings need to be stored in a particular order. For simplicity, we insert a new string at the beginning of the Linked List:

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

Observers

- `getName()` is the same as before.
- `isFull()` is even simpler: because it always returns `false`.

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

- As for **`size()`**, we have 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;  
}
```

- If the linked list contains N nodes, how many steps would the `size()` function take?
-

Observers

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

- If the linked list contains N nodes, how many steps would the `size()` function take? —> **O(N)**
- An alternative way is to define a integer variable to track the number of elements, hence the `size()` function simply needs to return the value of this integer.

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

- Since `insert` always inserts strings at the front of the list, `toString` will output strings in the opposite order of their insertion. This is ok as the order doesn't matter.

```
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;  
}
```

A few questions

- But what if we need to ensure the traversal must follow the same order as strings are inserted? What would you do?
- Is linked list always more memory efficient than arrays? Keep in mind that the reference (i.e. `link` variable) also takes memory space.
- What about expandable arrays (i.e. resize the array on the fly)?
- There are hybrid structures like a linked list where each node stores a small fixed-size array.