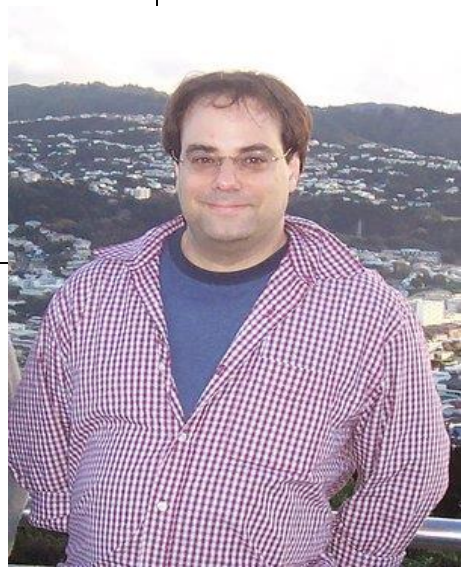
Arrays and Linked Lists

"All the kids who did great in high school writing pong games in BASIC for their Apple II would get to college, take CompSci 101, a data structures course, and when they hit the pointers business, their brains would just totally explode, and the next thing you knew, they were majoring in Political Science because law school seemed like a better idea."

-Joel Spolsky



Question 1

What is output by the following code?

```
ArrayList<Integer> a1 = new ArrayList<Integer>();
ArrayList<Integer> a2 = new ArrayList<Integer>();
a1.add(12);
a2.add(12);
System.out.println( a1 == a2 );
```

A. No output due to syntax error

B. No output due to runtime error

C. `false`

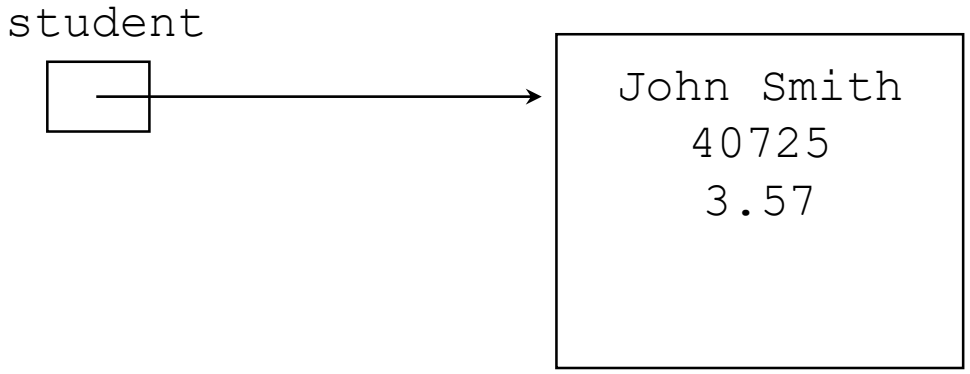
D. `true`

Dynamic Data Structures

- *Dynamic* data structures
  - They grow / shrink one element at a time
    - Normally without some of the inefficiencies of arrays
    - Unlike a static container such as an array
- Big-O of Array Manipulations
  - Access the *k*th element?
  - Add or delete an element in the middle of the array while maintaining relative order?
  - Adding element at the end of array?
    - Space available?
    - No space available?
  - Add element at beginning of an array?

Object References

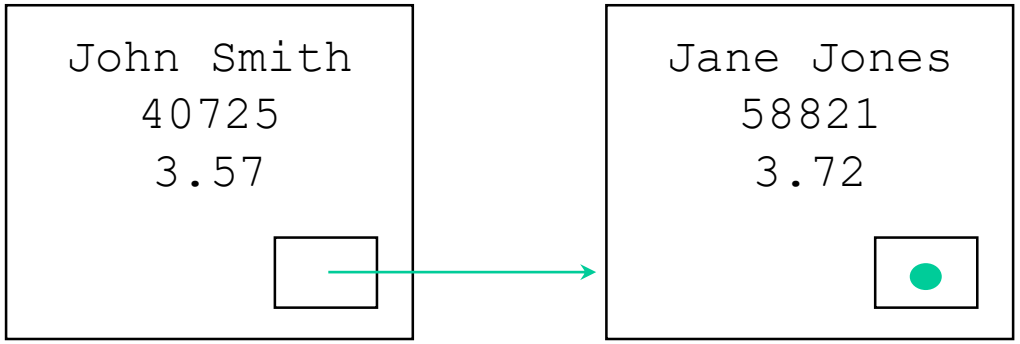
- Recall, a variable is an *object reference*, the address of an object
- A reference can also be called a *pointer*
- They are often depicted graphically:



References as Links

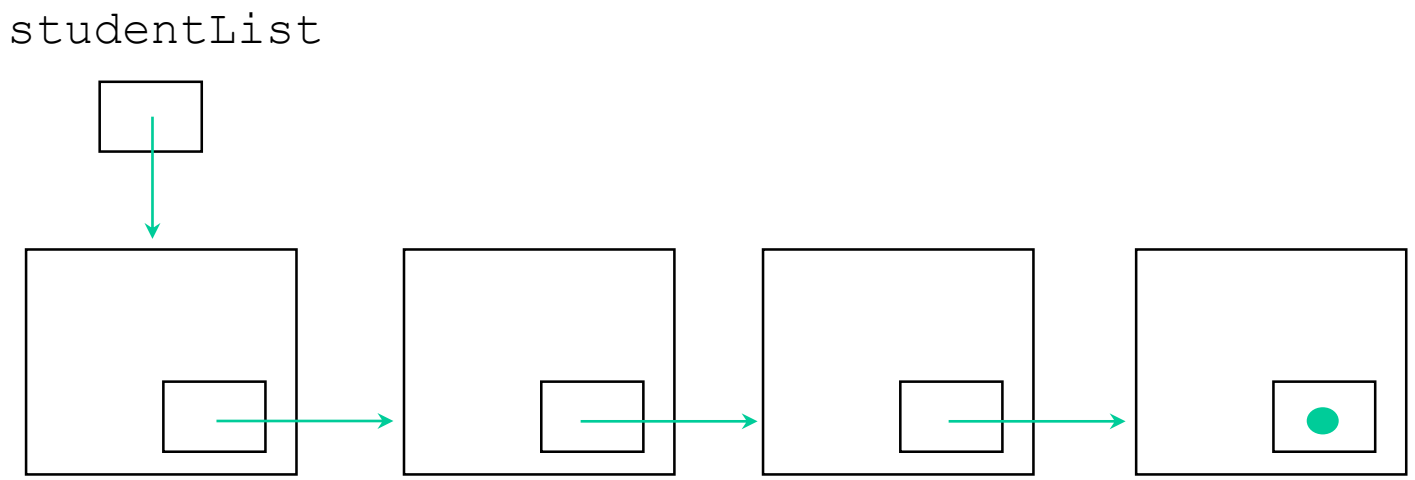
- Object references can be used to create *links* between objects

- Suppose a `Student` class contained a reference to another `Student` object



References as Links

- References can be used to create a variety of linked structures, such as a *linked list*:



Advantages of Linked Lists

- Linked lists are dynamic, so they can grow or shrink as necessary

- Linked lists are *non-contiguous;* the logical sequence of items in the structure is decoupled from any physical ordering in memory

Nodes and Lists

- A different way of implementing a list
- Each element of a linked list is a separate *Node* object.
- Each *Node* tracks a single piece of data plus a reference to the next *Node*
- Create a new *Node* when add new data to the list
- Remove a *Node* when data is removed from list and allow garbage collector to reclaim that memory

A Node Class

```
public class Node<T> {
        private T data;
        private Node<T> next;

        public Node()
        {       data = null; next = null;  }

        public Node(T data)
        {       this.data = data; next = null; }

        public T getData()
        {       return data;  }

        public Node<T> getNext()
        {       return next;  }

        public void setData(T data)
        {       this.data = data;    }

        public void setNext(Node<T> next)
        {       this.next = next;    }
}
```

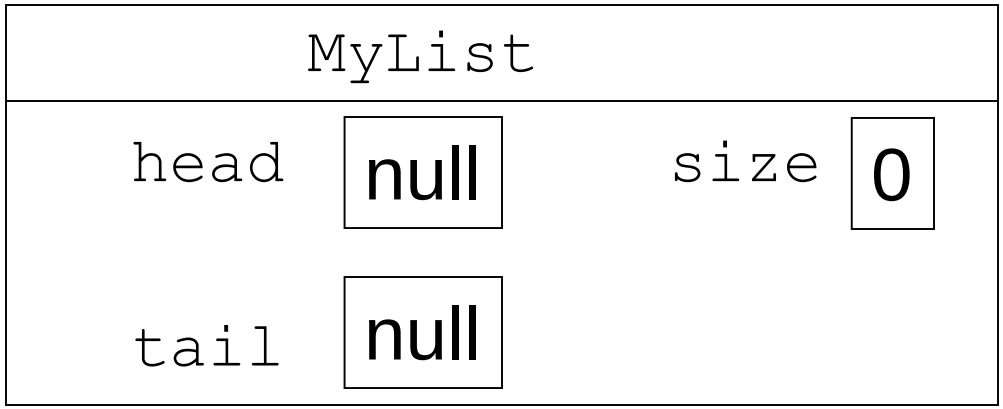
One Implementation of a Linked List

- The *Node*s shown on the previous slide are *singly linked*
  - A *Node* refers only to the next *Node* in the list
- It is also possible to have *doubly linked Node*s.
  - Each *Node* has a reference to the next *Node* in the structure and the *previous Node* in the list too
- How know when at the end of the list
  - next = *null*
  - a dummy *Node*

A Linked-List Implementation

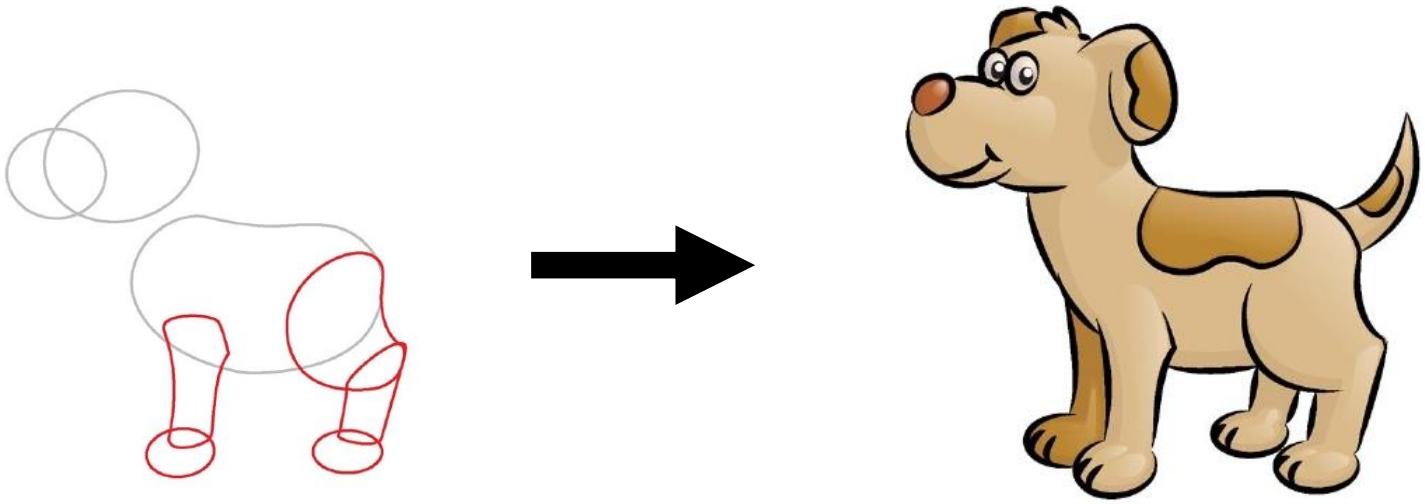
```
public class MyList<T> implements IList<T>
	private Node<T> head;
	private Node<T> tail;
	private int size;

	public MyList(){
		head = null;
		tail = null;
		size = 0;
	}
}
MyList<String> list = new MyList<String>();
```



Writing Methods

- When trying to code methods for linked lists ***draw pictures!***
  - If you don't draw pictures of what you are trying to do, it is very easy to make mistakes!

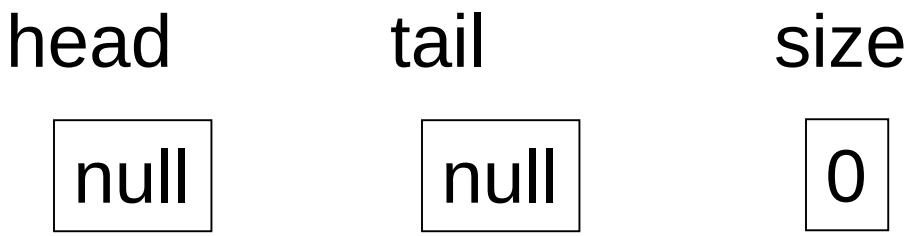


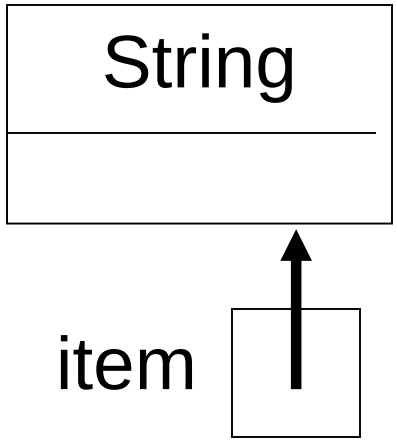
add method

```
public void add(T item)
```

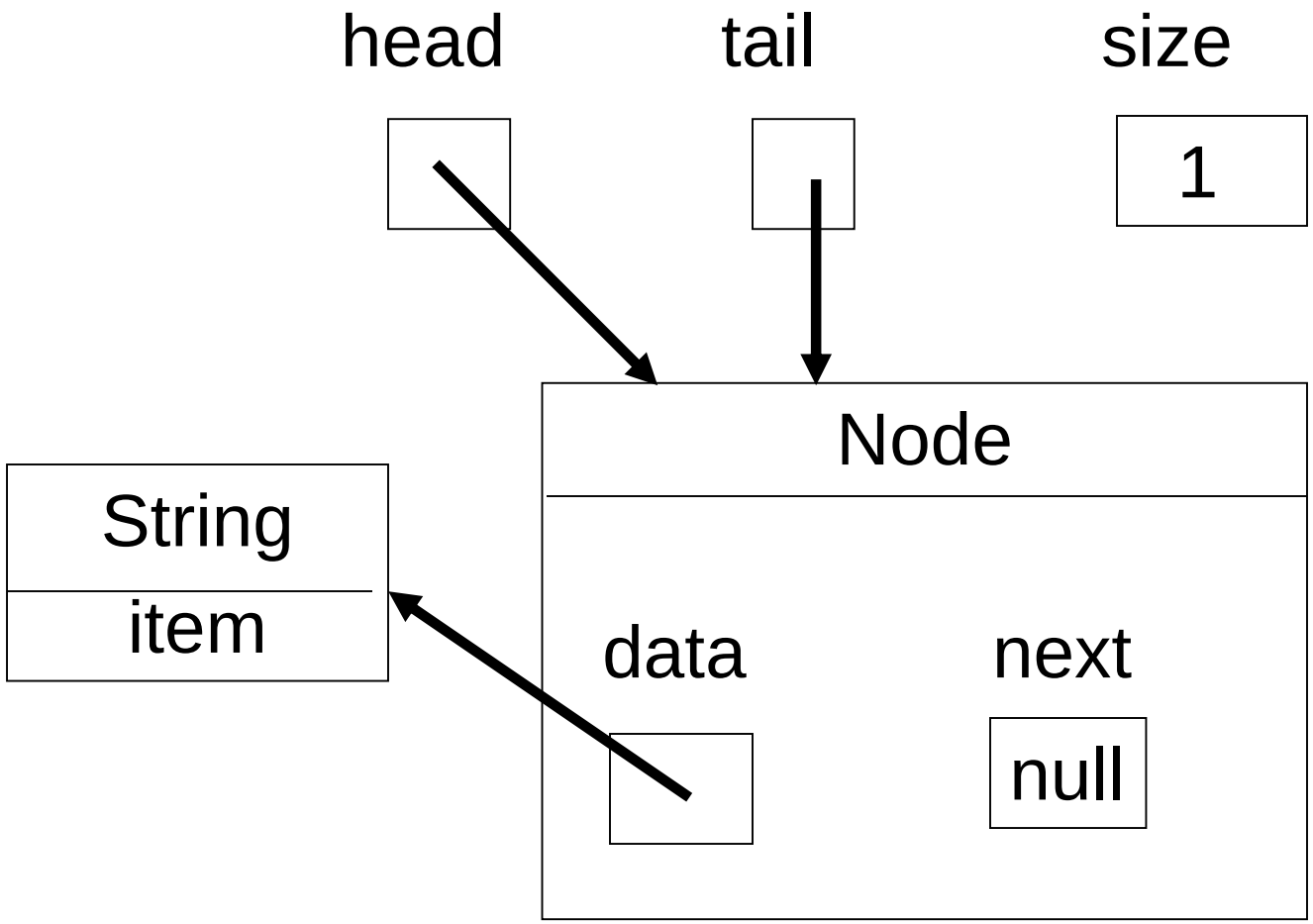
- add to the end of list
- special case if empty
- steps on following slides

Add Element - List Empty (Before)

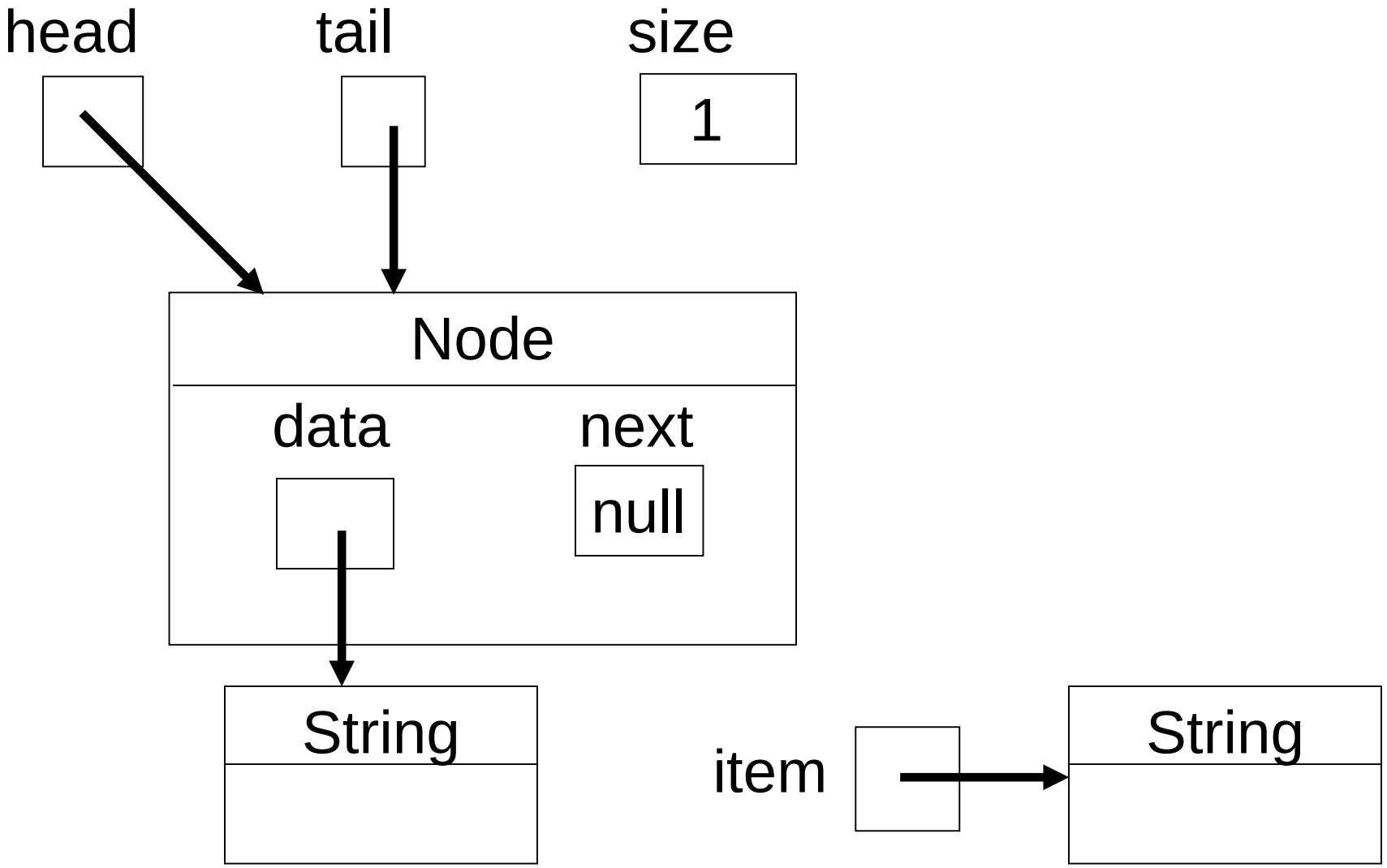




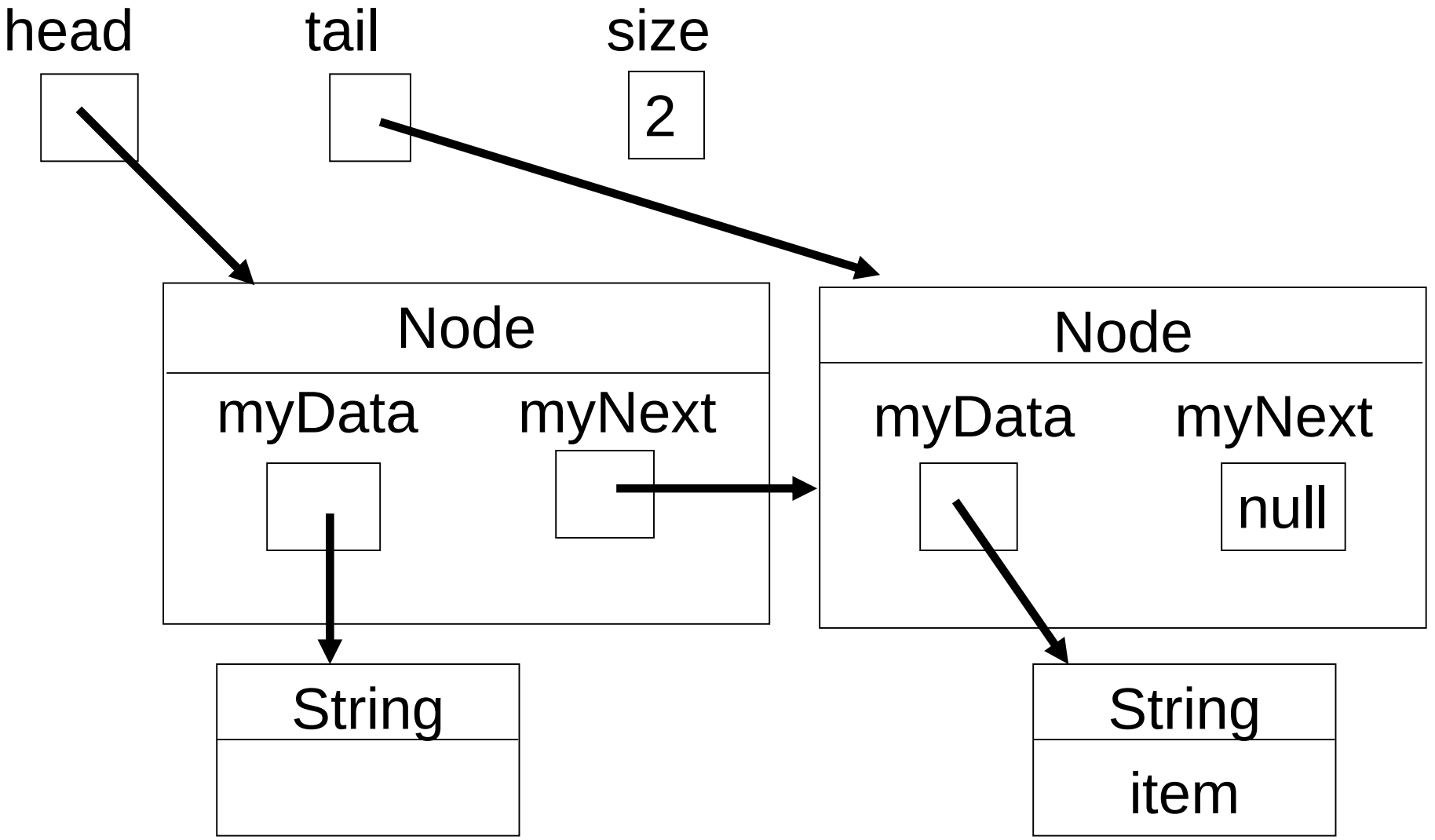
Add Element - List Empty (After)



Add Element - List Not Empty (Before)



Add Element - List Not Empty (After)



Question 2

- What is the worst case run-time *to add at the end* of an array-based list and a linked-list implementation, if the lists contain *n* items?

	Array	Linked List
A.	O(1)	O(1)
B.	O(n)	O(n)
C.	O(log n)	O(1)
D.	O(1)	O(n)
E.	O(n)	O(1)

Code for addFront

```
public void addFront(T item)
```

- add to front of list
- How does the run-time compare to adding at the front of an array-based list?

Question 3

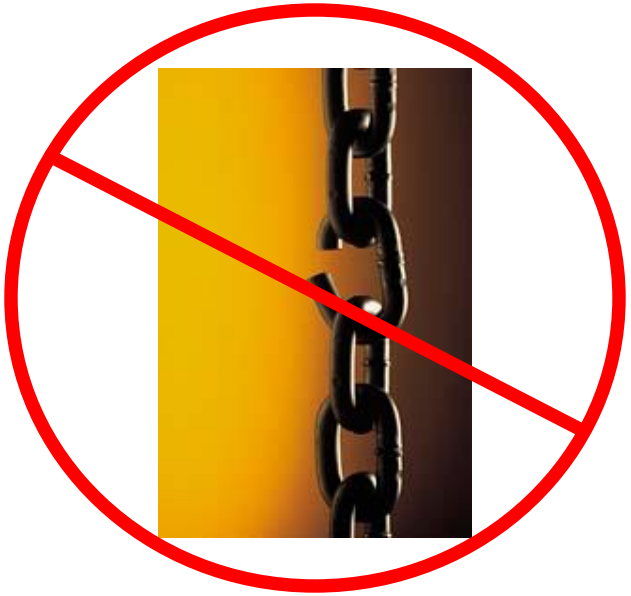
- What is the run-time *to add at the front* of an array-based list and a linked list implementation, if lists contain *n* items?

	Array	Linked List
A.	O(1)	O(1)
B.	O(n)	O(1)
C.	O(log n)	O(1)
D.	O(1)	O(n)
E.	O(n)	O(n)

Code for Insert

```
public void insert(int pos, T item)
```

- Must be careful not to break the chain!
- Where do we need to stop?
- Special cases?



Question 4

- What is the run-time *to insert an element into the middle* of an array-based list and a linked-list implementation, if lists contain *n* items.

	Array	Linked List
A.	O(n)	O(n)
B.	O(n)	O(1)
C.	O(log n)	O(1)
D.	O(log n)	O(log n)
E.	O(1)	O(n)

Question 5

- What is the run-time *to find an element based on position* from an array-based list and a linked-list implementation, if lists contain *n* items?

	Array	Linked List
A.	O(1)	O(n)
B.	O(n)	O(1)
C.	O(log n)	O(1)
D.	O(log n)	O(n)
E.	O(n)	O(n)

Code for get

```
public T get(int pos)
```

- The downside of linked lists

Code for remove

```
public T remove(int pos)
```

Why Use Linked Lists?

- What operations are faster using a linked-list implementation than an array-based implementation?
- Which are the same?

Iterators for Linked Lists

What is the Big-O of the following code?

```
MyList<Integer> list;
list = new MyList<Integer>();
// code to fill list with n elements

//Big-O of following code?
for(int i = 0; i < list.size(); i++)
   System.out.println( list.get(i) );
```

A. O(n) B. $O(2^n)$ C. O(n log n)

D. $O(n^2)$ E. $O(n^3)$

Array Efficiency

Method	Array
`add`	O(1)
`add(`**`index, value`**`)`	O(n)
`indexOf`	O(n)
`get`	O(1)
`remove`	O(n)
`set`	O(1)
`size`	O(1)

Linked List Efficiency

Method	Linked List
`add`	O(1)
`add(`**`index, value`**`)`	O(n)
`indexOf`	O(n)
`get`	O(n)
`remove`	O(n)
`set`	O(n)
`size`	O(1)

Other Linked Lists

Node class for Doubly Linked Lists

```
public class DLLNode<T> {
     private T data;
     private DLLNode<T> next;
     private DLLNode<T> previous;

}
```

Dummy Nodes

- Use of Dummy Nodes for a doubly linked list removes most special cases
- Also could make the double linked list circular

Doubly Linked List add

```
public void add(T item)
```

Insert for Doubly Linked List

```
public void insert(int pos, T item)
```