UNIVERSITY *of* VIRGINIA

CS 2100: Data Structures & Algorithms 1

Introduction to Linked Lists

Dr. Nada Basit // basit@virginia.edu

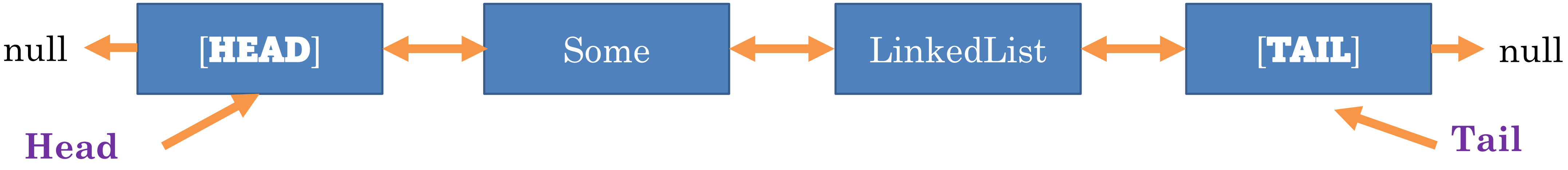
Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Linked Lists

- **Arrays** and **Vectors** use contiguous memory to store data
  - Arrays **built into Java** and have special syntax
  - Vectors an **extension of arrays**.



null

[**HEAD**]

Some

LinkedList

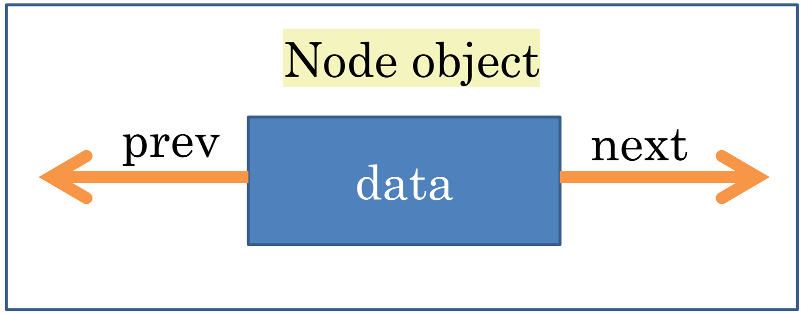
[**TAIL**]

null

Head

Tail

- A **Linked List** is a list that stores nodes connected to one another through references
  - Each element in the list is a **ListNode**
    - Stores the **data** inside that element
    - Stores references to the **next** and **previous** elements



Node object

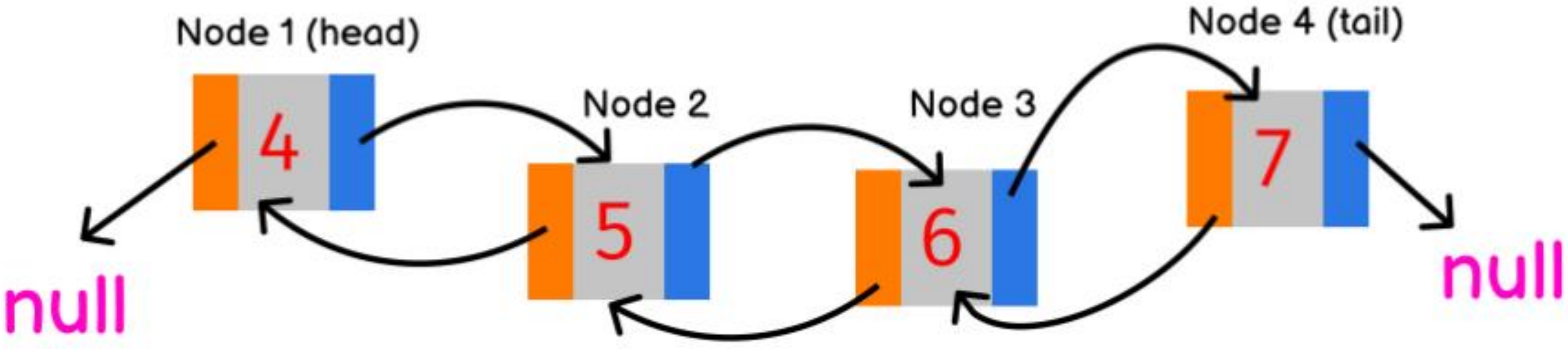
prev

data

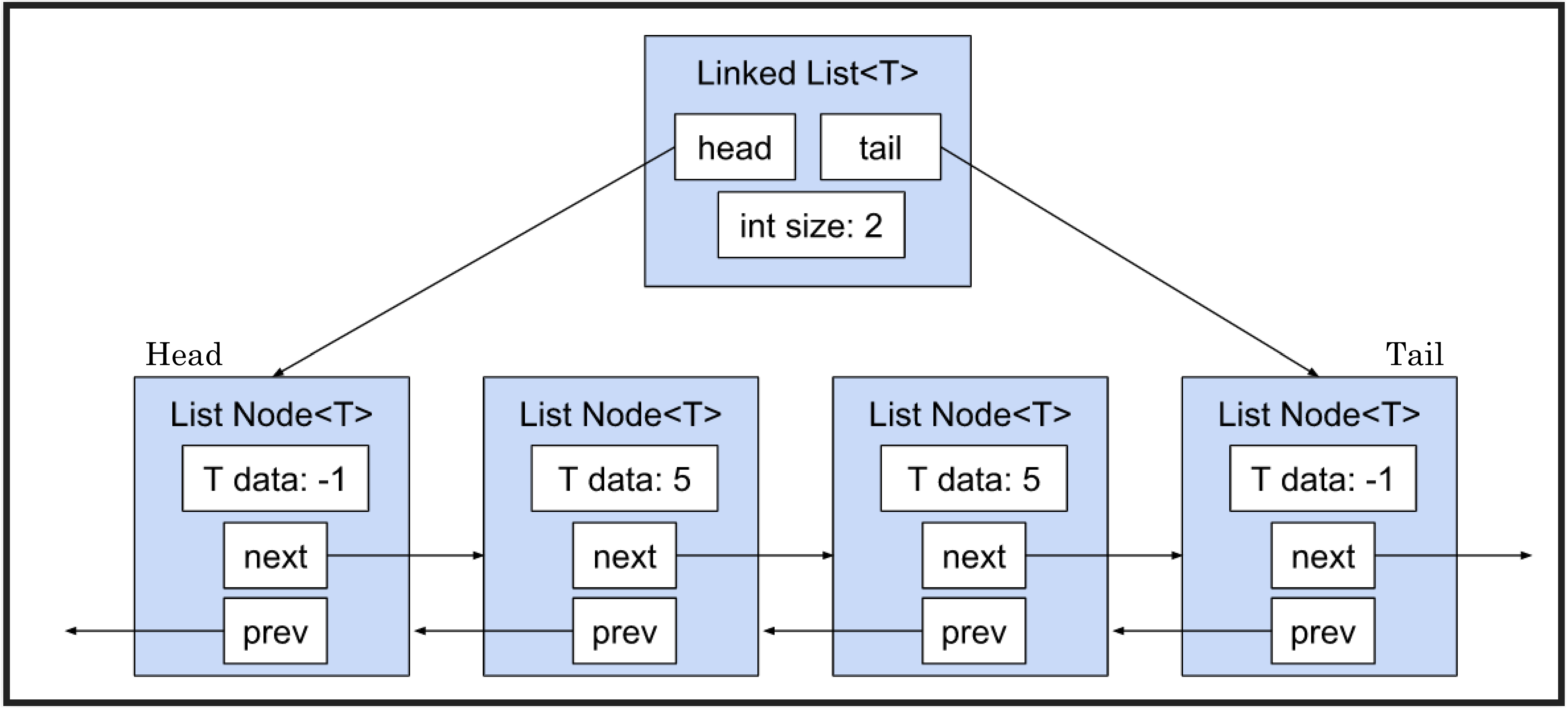
next

Linked List Example

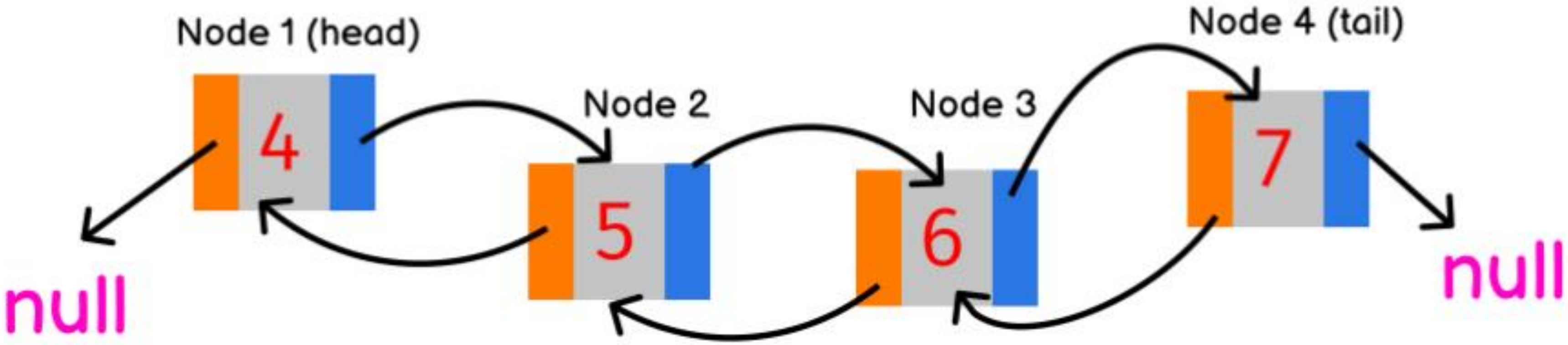
- This Linked List, specifically known as a **doubly linked list**, has nodes with **two** references
  - next and previous
- There are special **head** and **tail** references that point to the first and last node in the list respectively
- The **last** element (tail) will have the next pointer point at **null** *(end of the list!)*
- The **first** element (head) will have the prev pointer point at **null** *(front of the list!)*

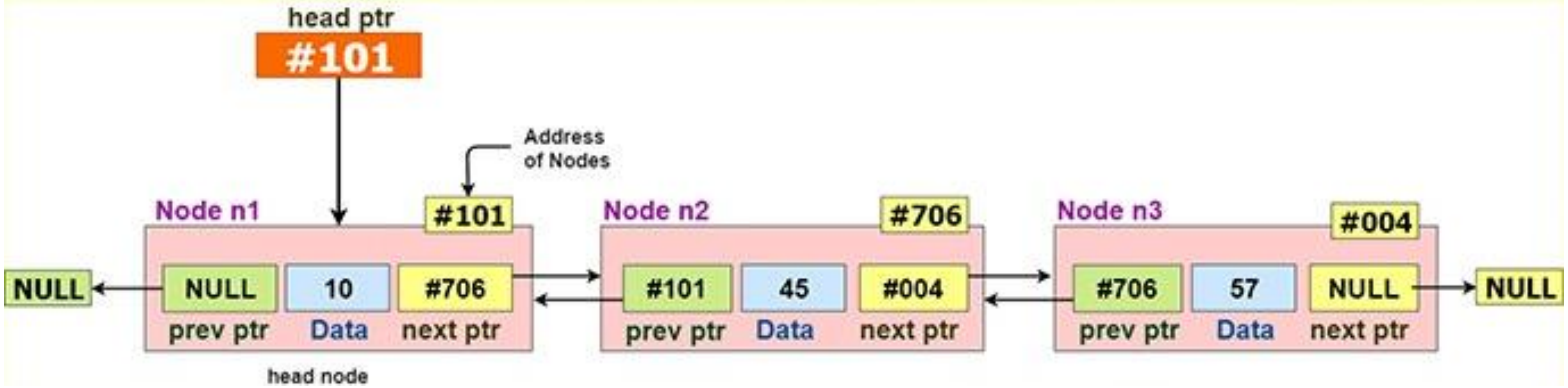


Linked List - Other Diagrams



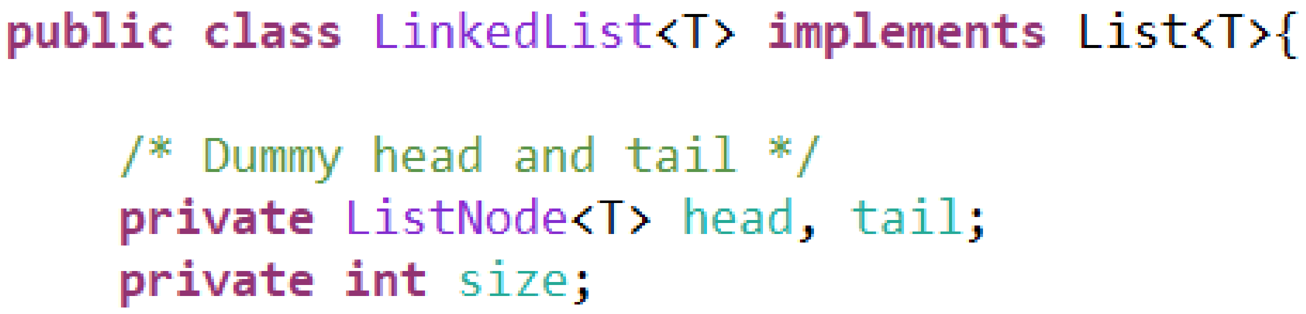
Linked List - Other Diagrams



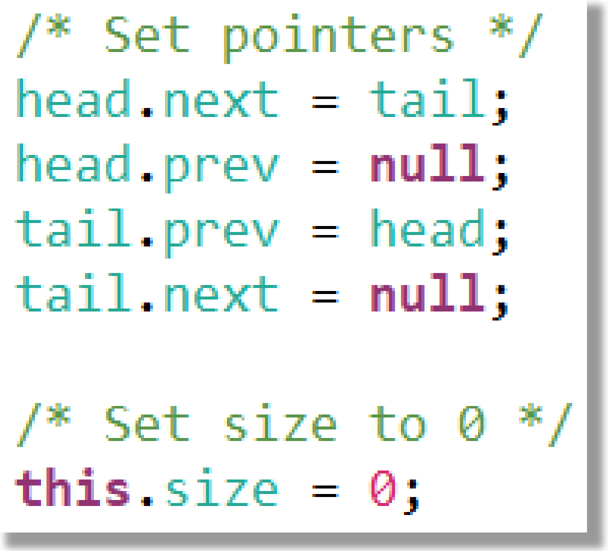


Linked List Properties [Code Example]

- **head**: reference to the first node in Linked List
  - This first node is *a dummy node* (not part of the actual list)
- **tail**: reference to the last node in Linked List
  - It is also *a dummy node*
- **size**: Number of elements in the list currently



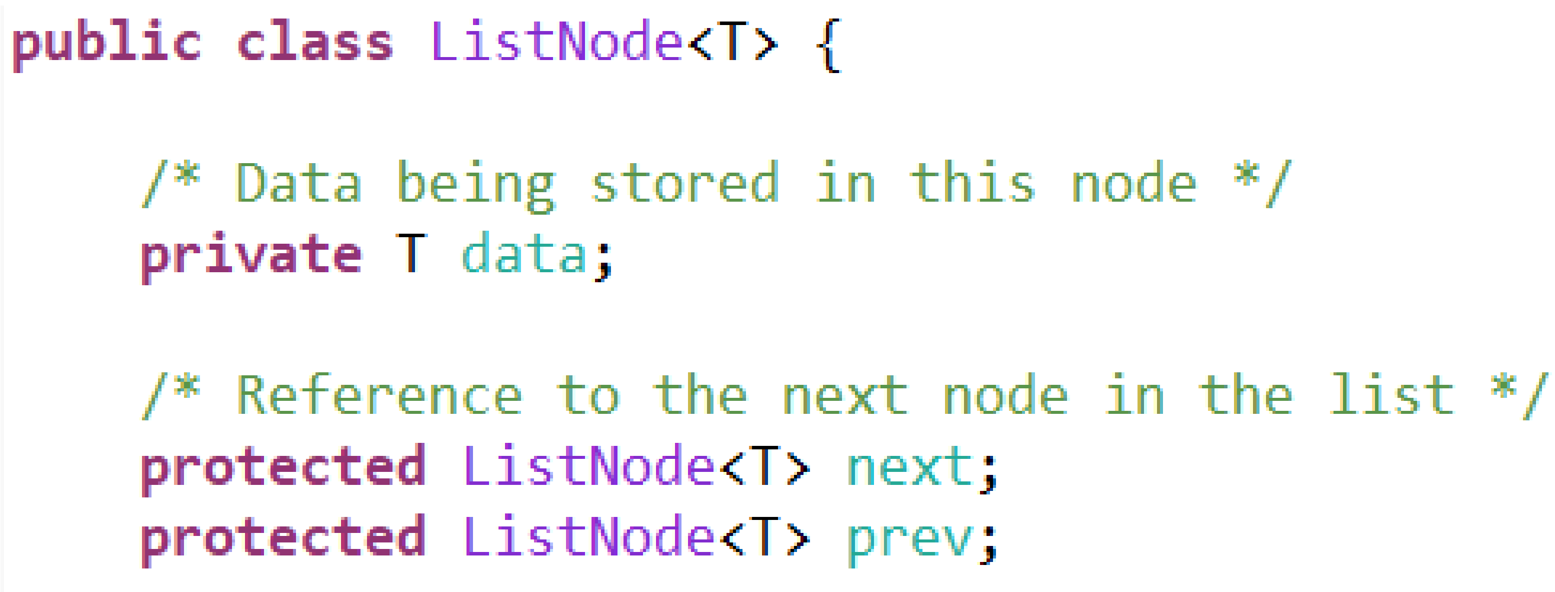
```
public class LinkedList<T> implements List<T>{

    /* Dummy head and tail */
    private ListNode<T> head, tail;
    private int size;
```



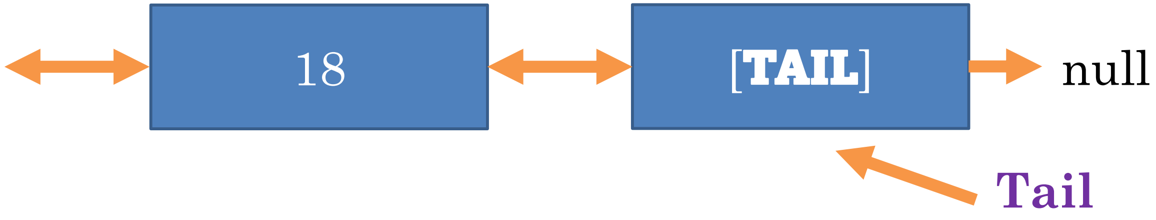
```
/* Set pointers */
head.next = tail;
head.prev = null;
tail.prev = head;
tail.next = null;

/* Set size to 0 */
this.size = 0;
```

List Node Properties [Code Example]

- **data**: the actual thing being stored in the list
- **next**: Reference to memory where the next node can be found
- **prev**: Reference to memory where the previous node can be found



```java
public class ListNode<T> {

    /* Data being stored in this node */
    private T data;

    /* Reference to the next node in the list */
    protected ListNode<T> next;
    protected ListNode<T> prev;
```

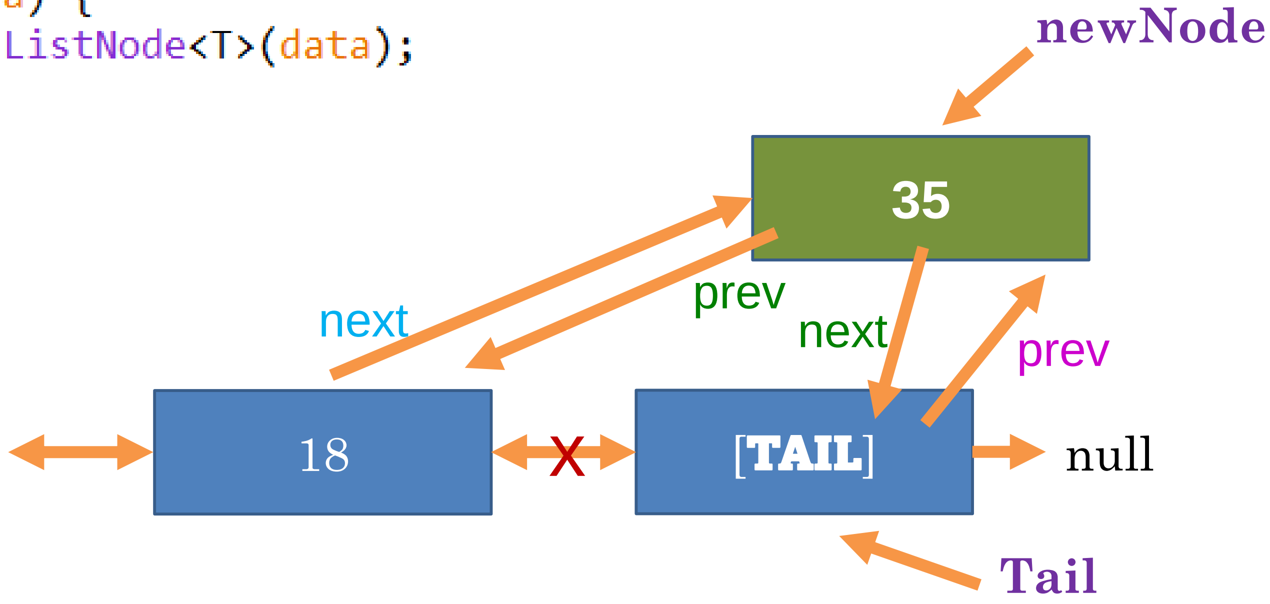


Inserting at Tail

- Here is how to insert at the tail of a Linked List
  - Notice, this is ALWAYS fast no matter how big (# elements) the list is

```java
public class LinkedList<T> {

    public void insertAtTail(T data) {
        ListNode<T> newNode = new ListNode<T>(data);
        newNode.next = tail;
        newNode.prev = tail.prev;
        tail.prev.next = newNode;
        tail.prev = newNode;

        this.size++;
    }
```

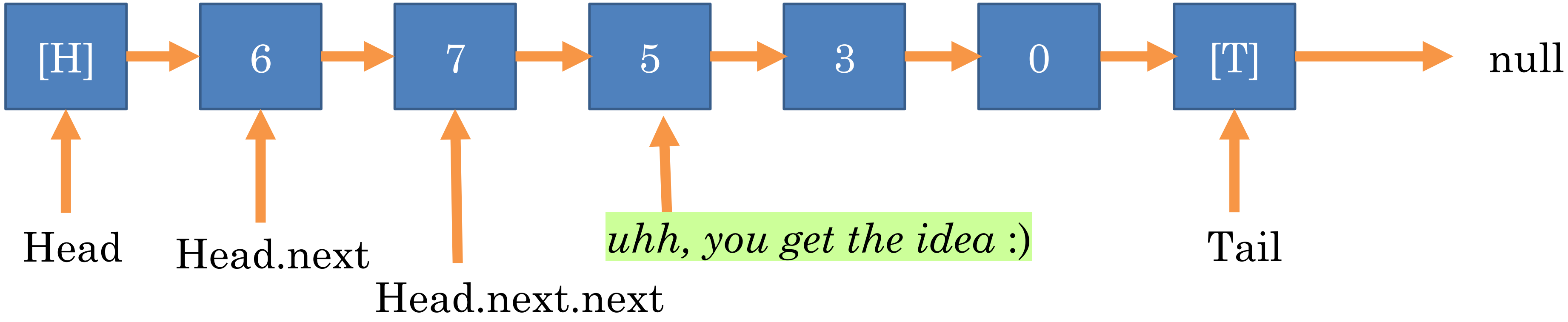
e.g. 35



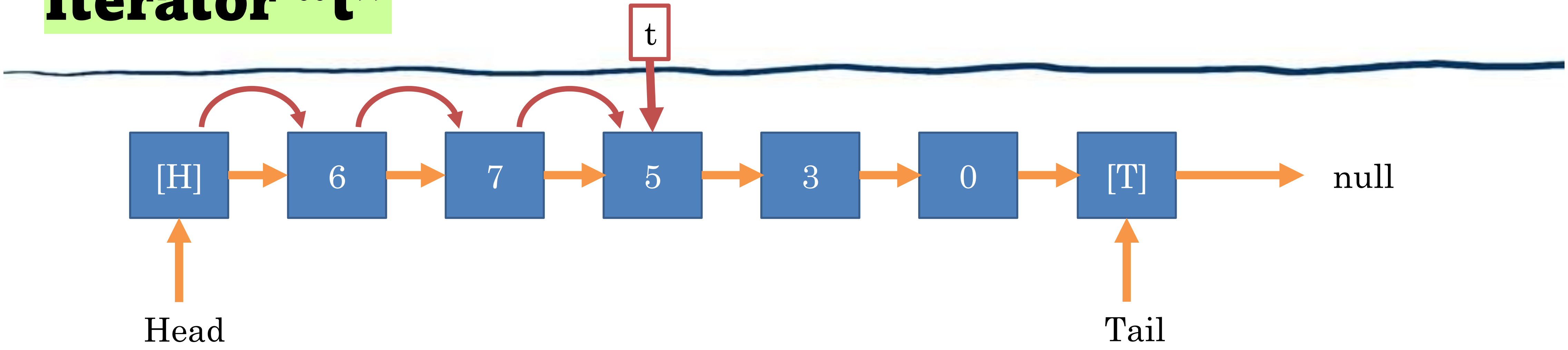
Keeping Track of Nodes

- The **LinkedList class** doesn't directly keep track of every node
  - We access every node indirectly through the `head`

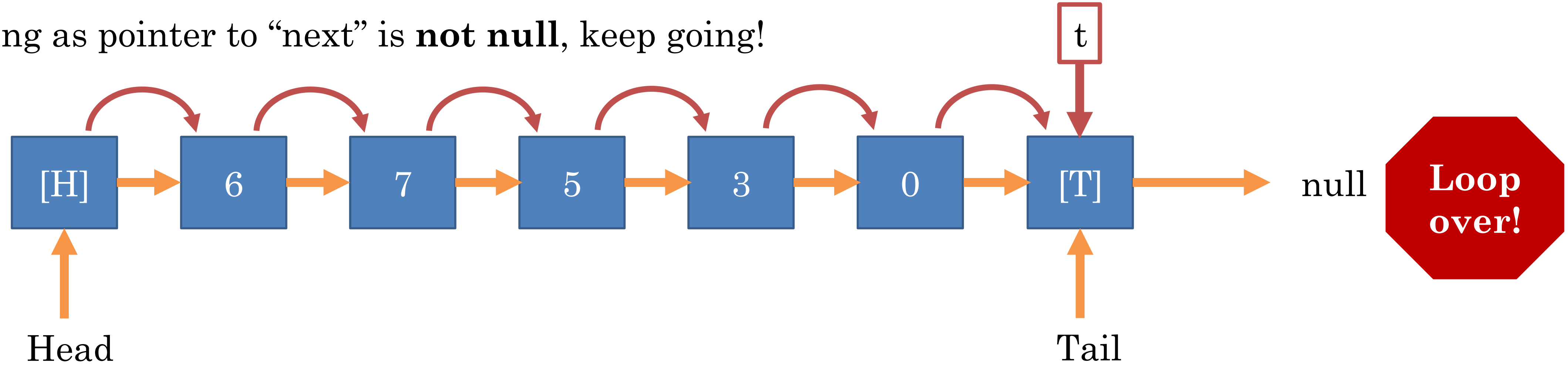
- For example, `head.next.next.next.data = 5`
  - Always remember: a node isn't a value, it's a **value** AND a **next**
    - So you need to use the **dot operator** to access the value or next separately



Iterator "t"

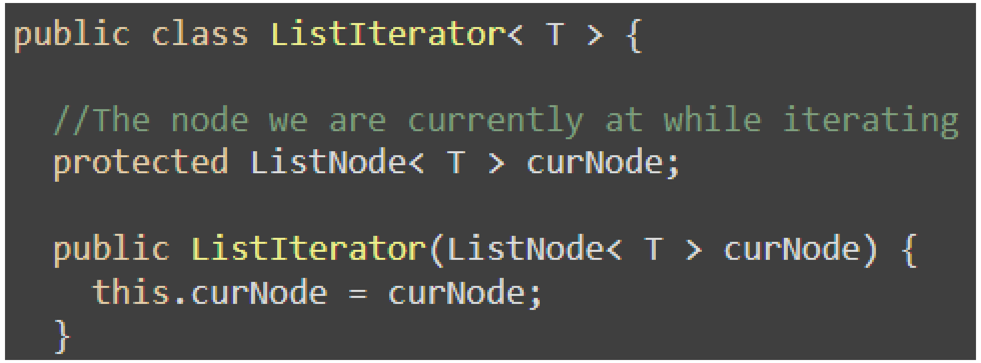


As long as pointer to "next" is **not null**, keep going!

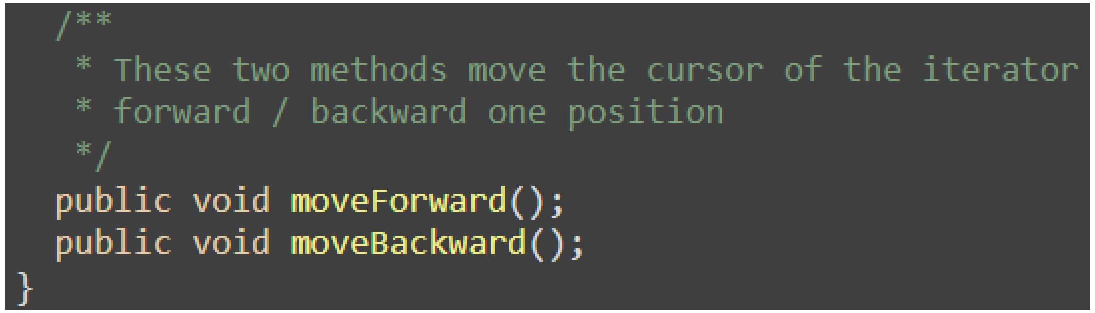


ListIterator

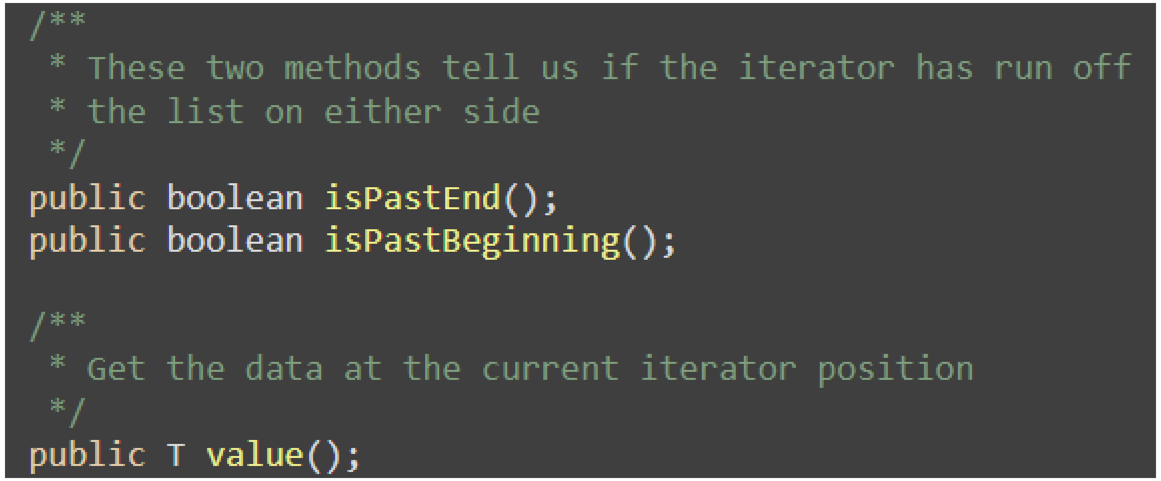
- **Problem**: head and tail fields are private! So, if I am using Linked List and need to, say, loop through it manually I can't do it.   Well, I can use **get(),** but that is VERY slow
- **Solution**: Supply a special type of object called an **iterator**
  - Provides methods for moving forward and backward through the list manually.



```
public class ListIterator< T > {

  //The node we are currently at while iterating
  protected ListNode< T > curNode;

  public ListIterator(ListNode< T > curNode) {
    this.curNode = curNode;
  }
```



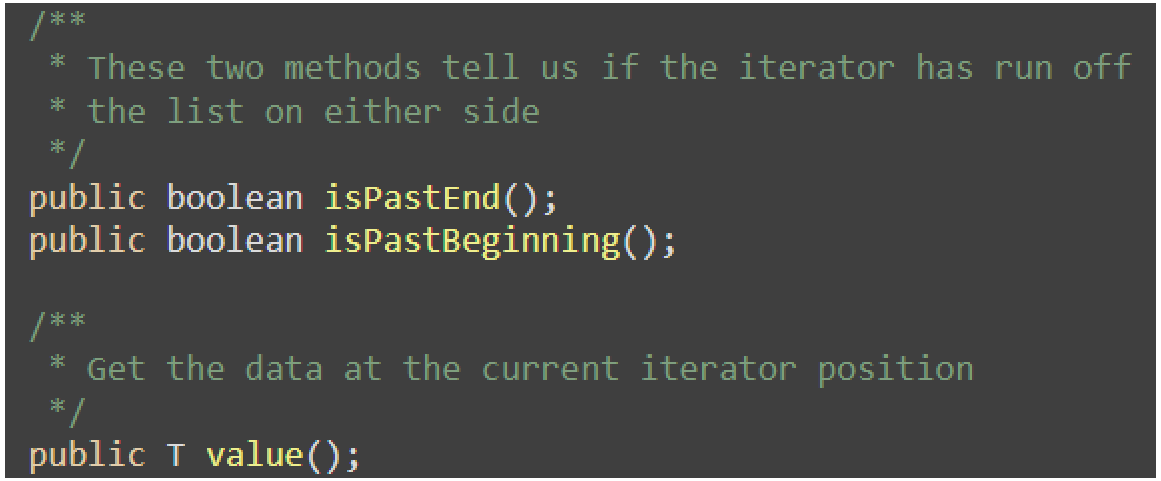
```
  /**
   * These two methods move the cursor of the iterator
   * forward / backward one position
   */
  public void moveForward();
  public void moveBackward();
}
```



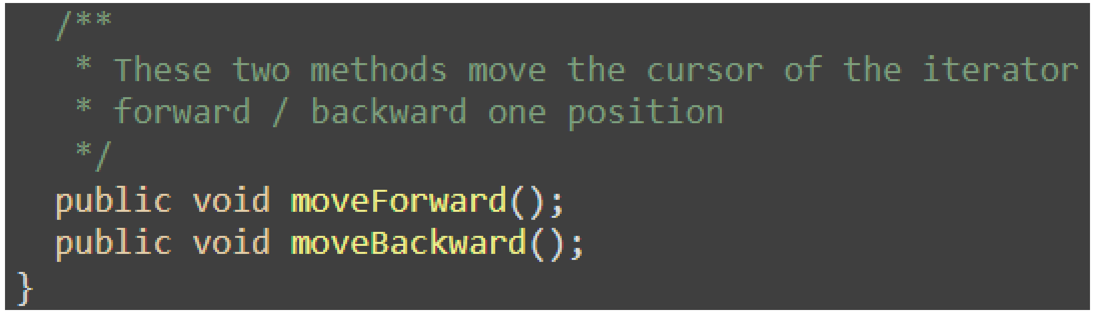
```
/**
 * These two methods tell us if the iterator has run off
 * the list on either side
 */
public boolean isPastEnd();
public boolean isPastBeginning();

/**
 * Get the data at the current iterator position
 */
public T value();
```

Using the ListIterator

```
private static <T> void printList(LinkedList< T > list) {
   //iterator points to first element
   list.ListIterator<T> it = list.front();

   while(!it.isPastEnd()) {
      System.out.print(it.value() + ", ");
      it.moveForward();
   }
}
```



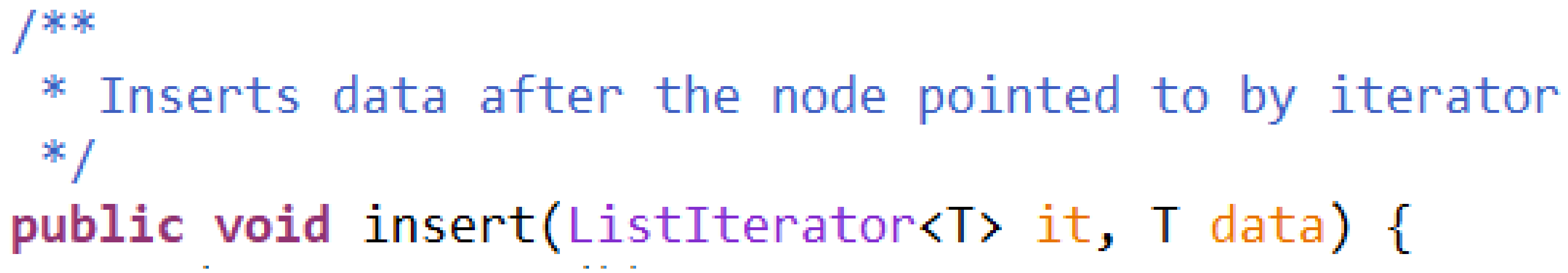
```
/**
 * These two methods tell us if the iterator has run off
 * the list on either side
 */
public boolean isPastEnd();
public boolean isPastBeginning();

/**
 * Get the data at the current iterator position
 */
public T value();
```



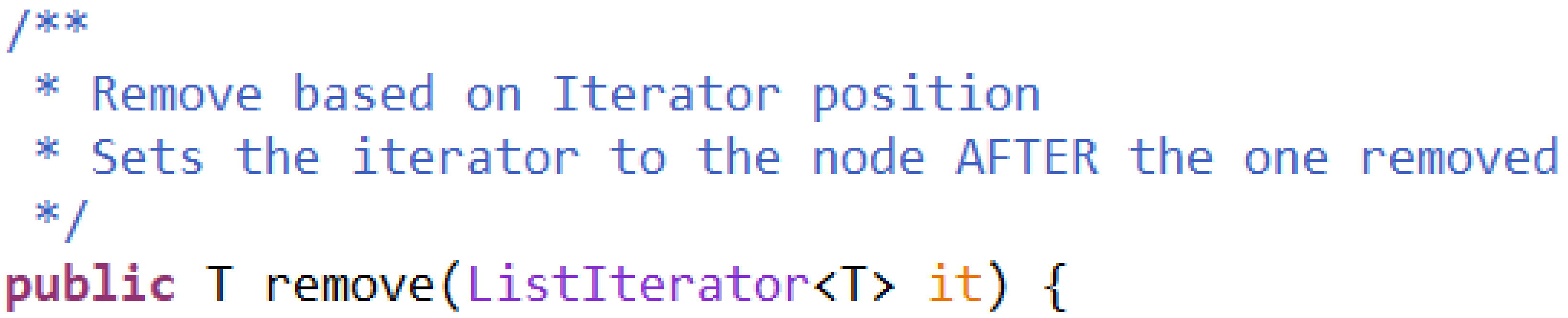
```
  /**
   * These two methods move the cursor of the iterator
   * forward / backward one position
   */
  public void moveForward();
  public void moveBackward();
}
```

Linked List: Insert and Remove at Iterator

- How might we tackle these behaviors?



```
/**
 * Inserts data after the node pointed to by iterator
 */
public void insert(ListIterator<T> it, T data) {
```



```
/**
 * Remove based on Iterator position
 * Sets the iterator to the node AFTER the one removed
 */
public T remove(ListIterator<T> it) {
```

Advantages and Disadvantages

Of Linked Lists

Linked List Advantages

- Can **insert** in **front** or **back** of list in constant time (**VERY FAST**)
  - Same for insertAt(Iterator)
- Likewise, can **remove** from front or back in **constant time**
- **List nodes** are scattered in memory, so no need for OS to find a contiguous block for the list
- Don't have unused space like a vector does
- Don't need to "grow in size" when they fill up.

Linked List Disadvantages

- **Slow** to **get** an index in middle of list because have to traverse from head or tail
  - Arrays can go directly to an index, why?
- Doesn't work well with cache, so arrays often faster in practice
  - Do you know what a cache is yet?
- All of the **next** and **prev** references use extra space.