Lecture 3.

Linked Lists

Marina Langlois

Some slides were borrowed from Prof. Alvarado.

Today's Lecture

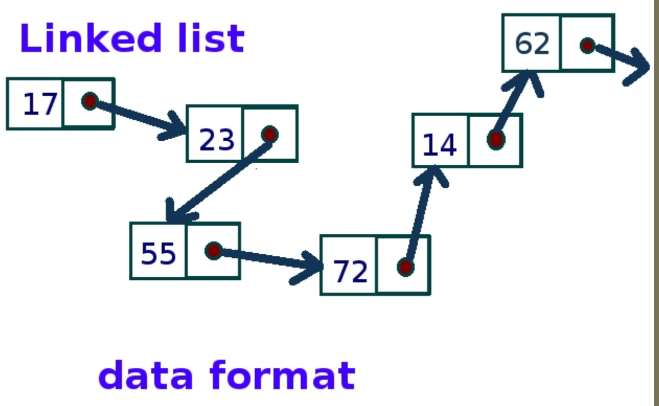
ArrayLists vs LinkedLists

- Problems:

- Wasteful in memory
- It does not solve the contiguity problem (fragmentation)
- Adding/Removing elements to the front requires shifting the whole array.

LinkedLists solve these problems.

Nodes and Lists



- A different way of implementing a list interface

- Each element of a Linked List is a separate Node object.

- Each node tracks a single piece of data **plus** a reference (pointer) to the next node.

- Create a new Node every time we add something to the List

- Remove nodes when item is removed from list and allow garbage collector to reclaim that memory

Types of Linked list

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Multilinked List

Implementation of the List interface with LinkedList

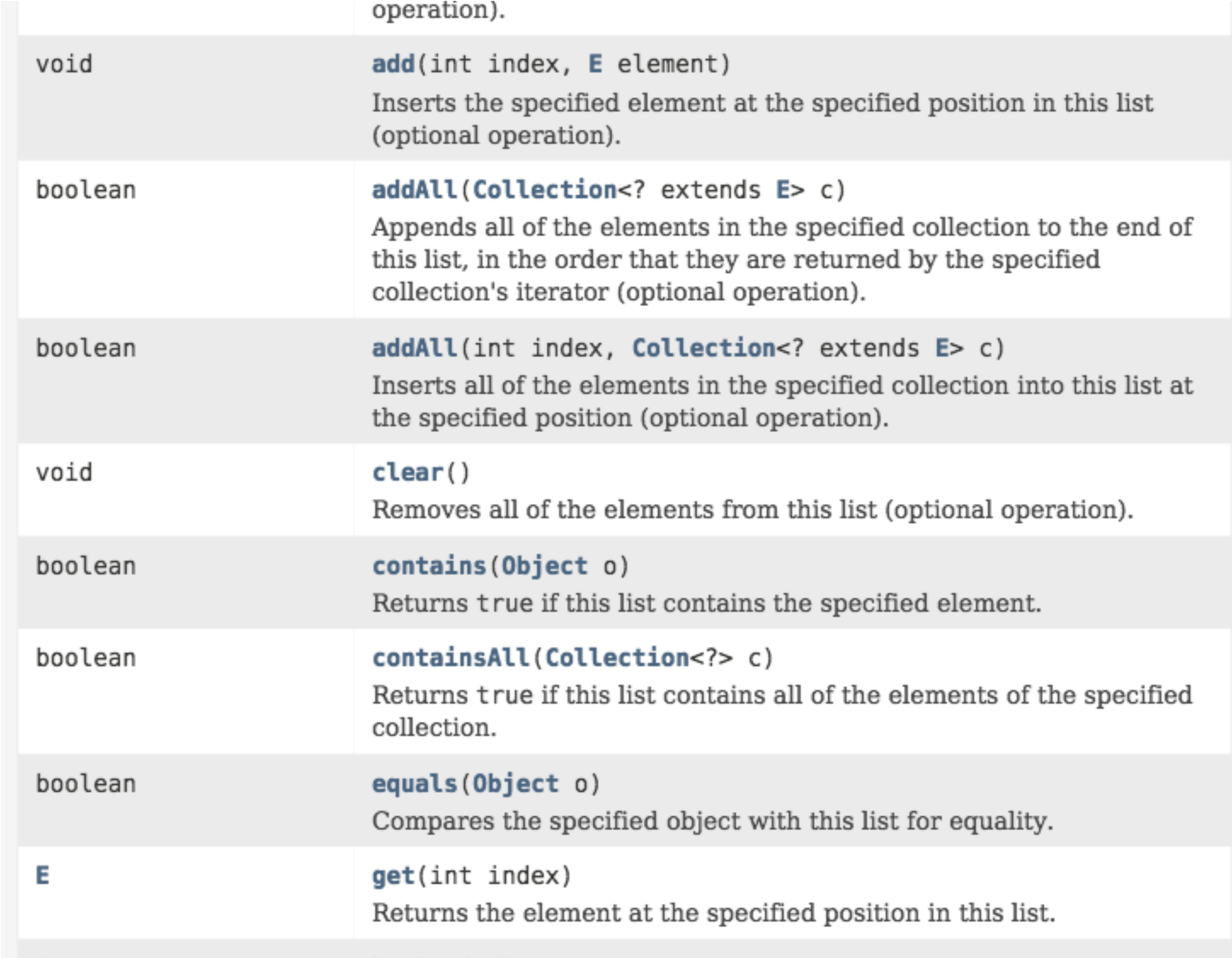
- Note: I will skip generics (stuff in < > , just ignore it until next Tuesday)

- `public class MySinglyLL implements List`

the implementation

the ADT

List interface is long

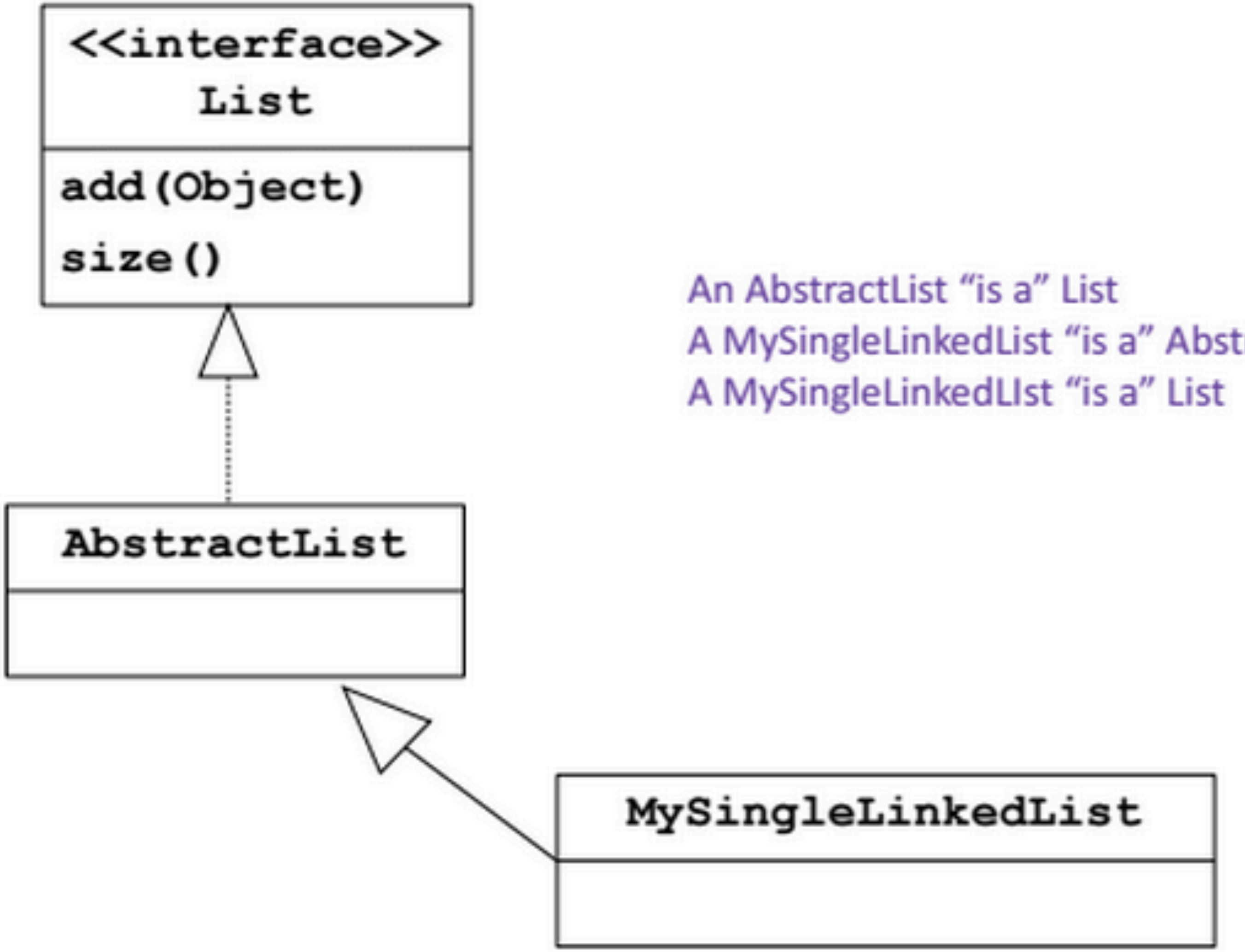


	operation).
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(**Object** o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(**Object** o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.

Abstract List

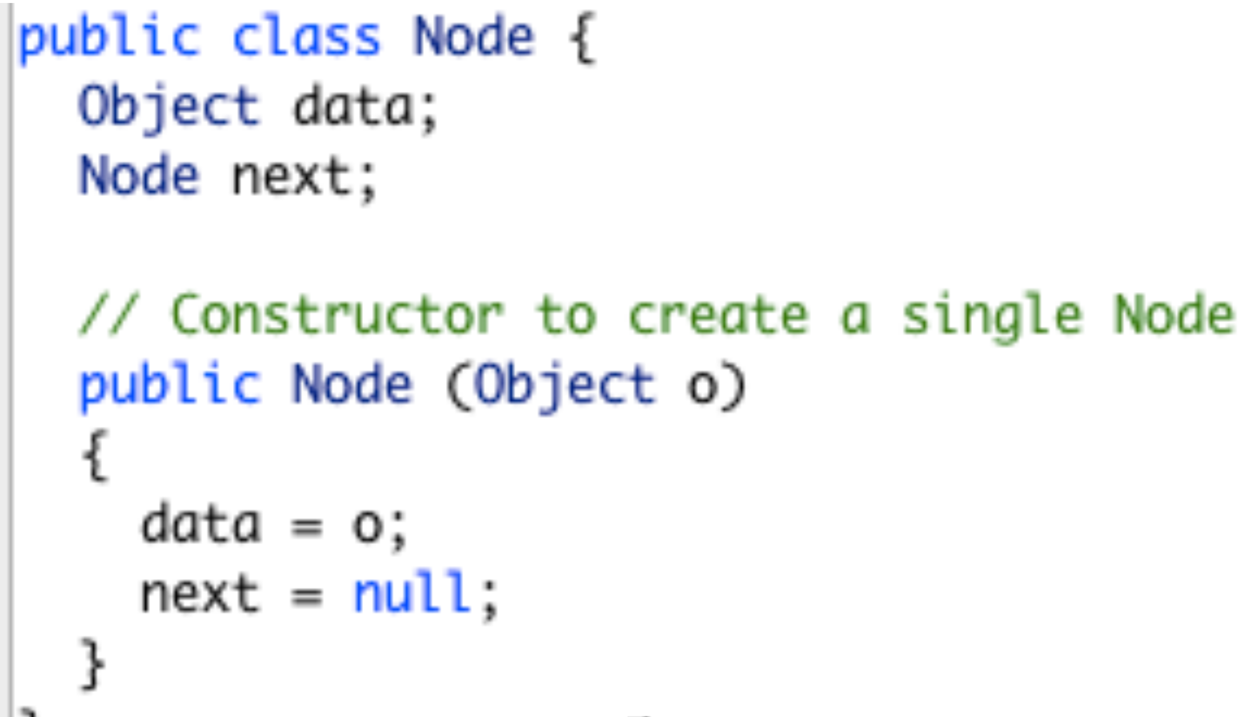
- `public class MySinglyLL extends AbstractList`
- Provides implementations for most methods in List interface.
- We can override its method with our own.

UML Model

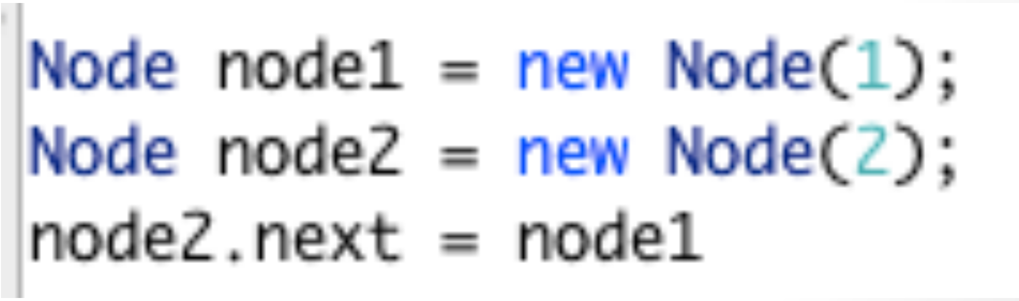


An AbstractList “is a” List
A MySingleLinkedList “is a” AbstractList
A MySingleLinkedLIst “is a” List

Draw a memory diagram



```
public class Node {
  Object data;
  Node next;

  // Constructor to create a single Node
  public Node (Object o)
  {
    data = o;
    next = null;
  }
}
```



```
Node node1 = new Node(1);
Node node2 = new Node(2);
node2.next = node1
```

Class Node

- Node class *is a part* of Linked List implementation

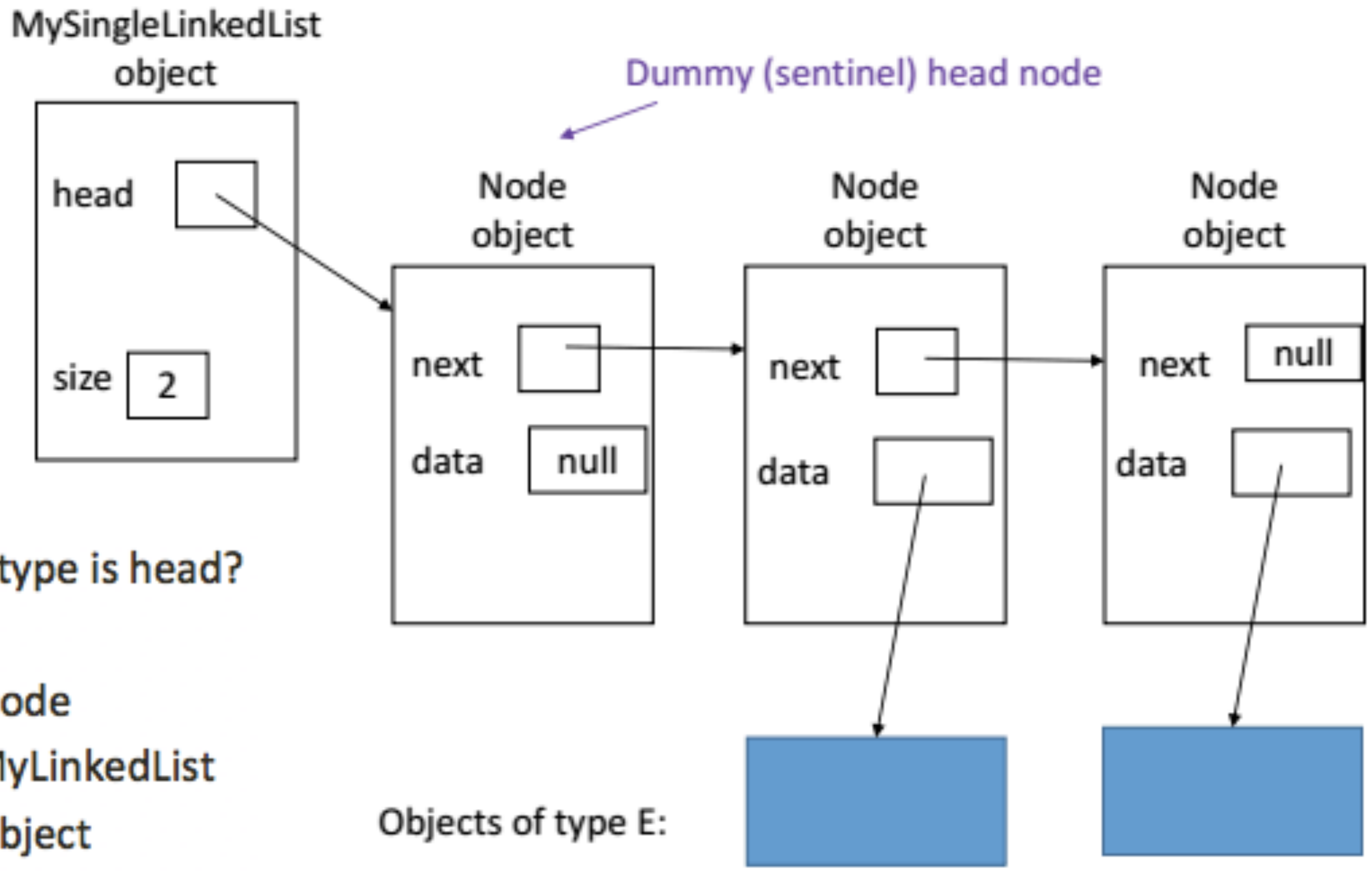
- The (typical) Node contains:
  - A reference to the next node in the lsit
  - A reference to the data stored at that position in the list
  - For Doubly Linked List a reference to the previous node

- The Linked List itself contains a reference to the FIRST node in the list (head, first). Sometimes it might store some info about the list (like list size)

Lists with sentinel (dummy) node

- *Dummy nodes* are Nodes whose data fields are always ***null*** – they contain **no** data from the “user”.
- The dummy nodes *will always exist, even if the user hasn’t added any data yet.*
- These nodes will *simplify* the implementation.

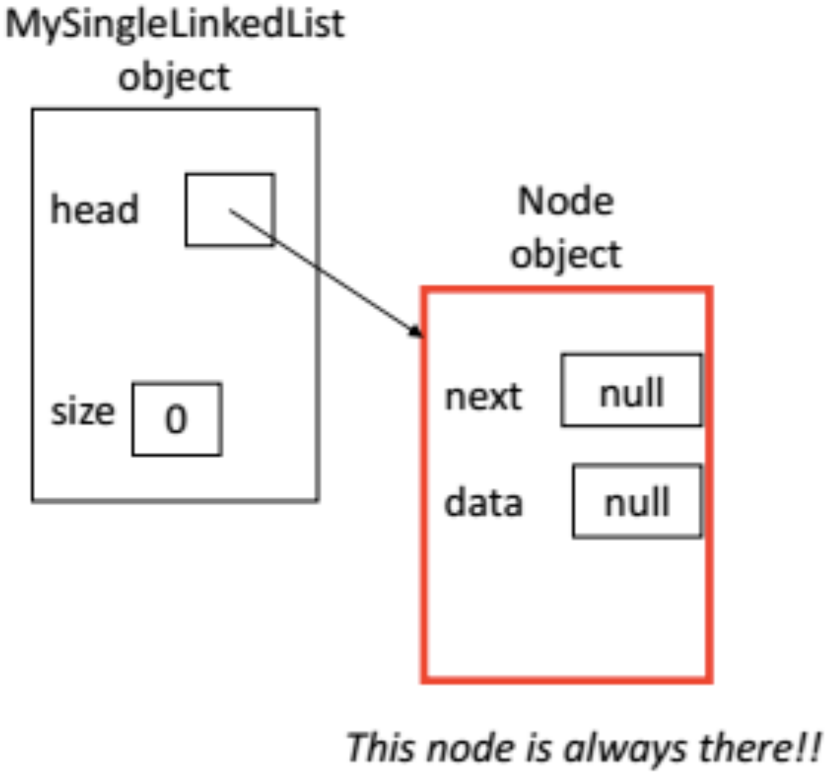
Dummy node: Picture



What type is head?

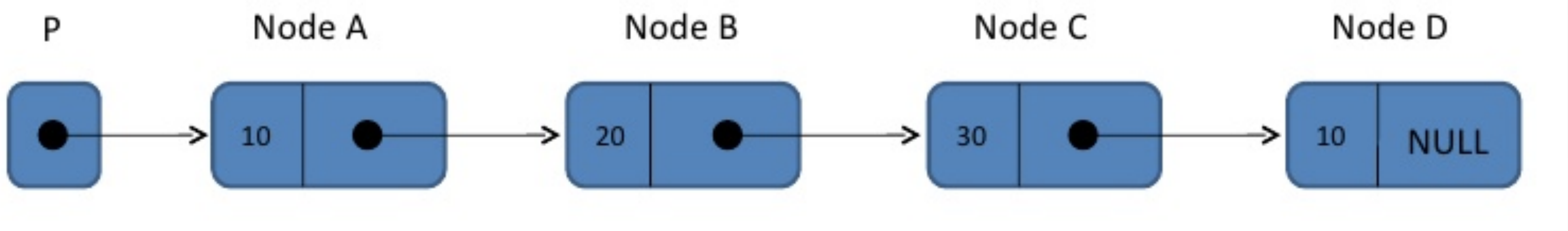
- A: Node
- B: MyLinkedList
- C: Object
- D: int
- E: Other

Empty list with sentinel node



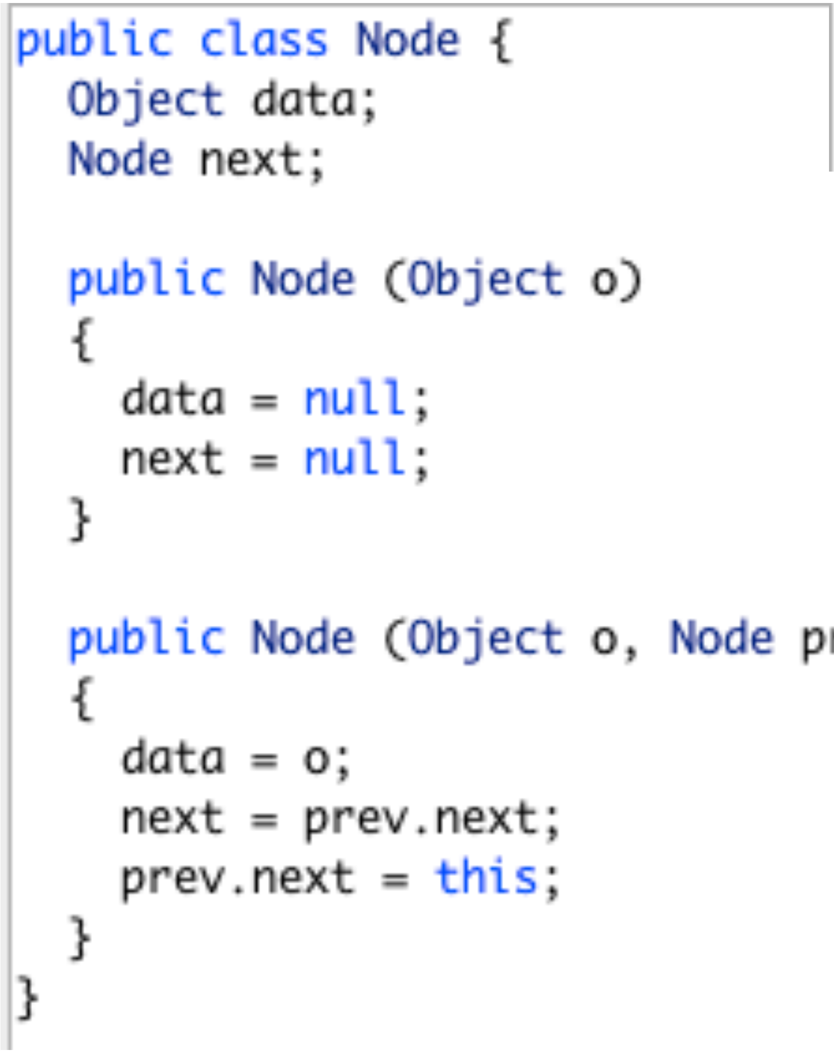
This node is always there!!

Add Front: NodeE.

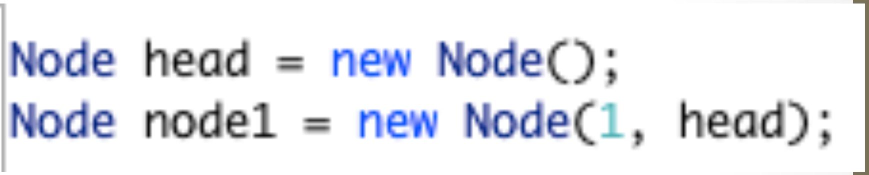


- A: P = NodeE;
- B: NodeE.next = Node A;
- C: P = Node E;
  NodeE.next = P;
- D: NodeE.next = P;
- E: NodeE.next = P;
  P = Node E;

Class Node

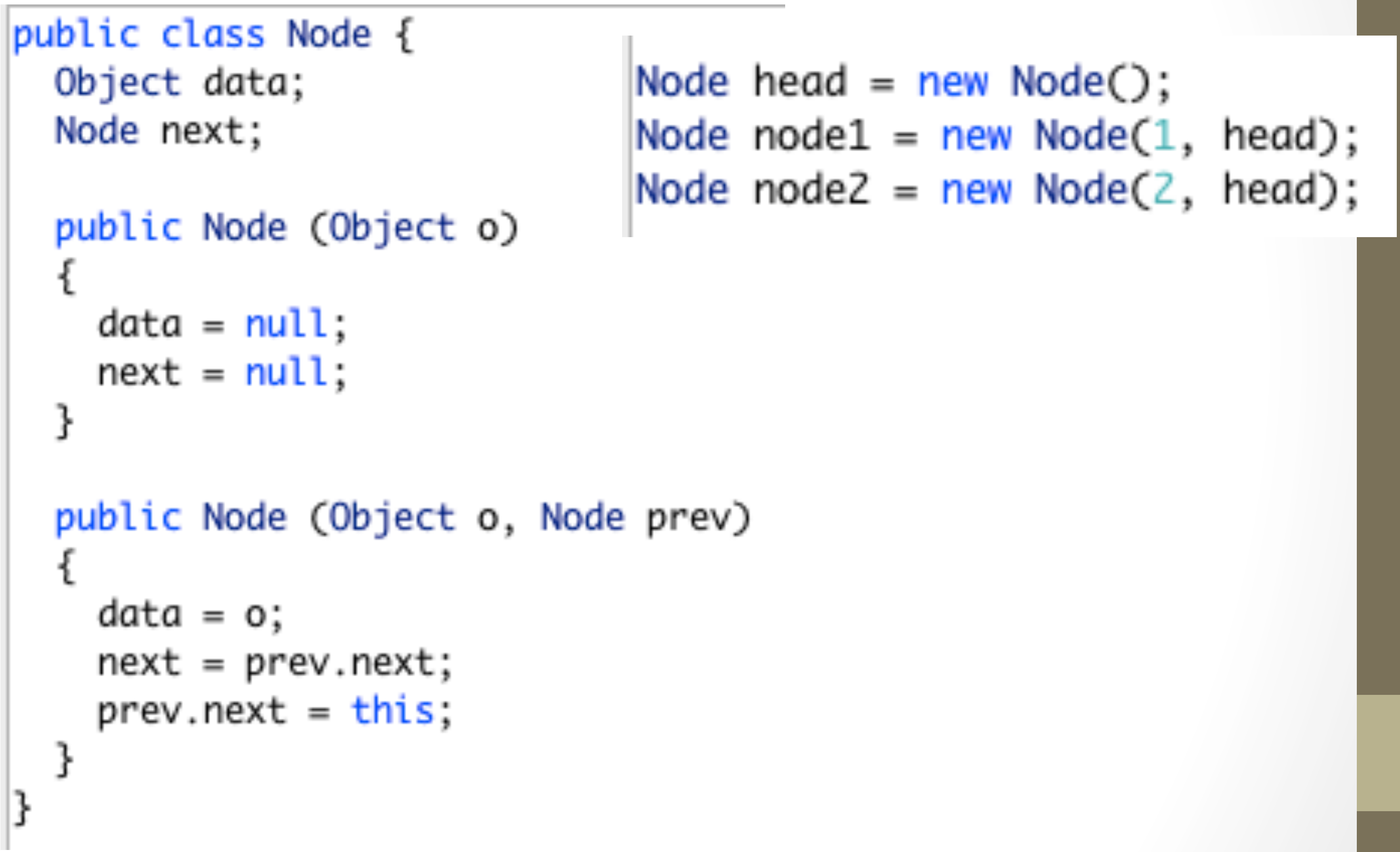


```
public class Node {
  Object data;
  Node next;

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

  public Node (Object o, Node prev)
  {
    data = o;
    next = prev.next;
    prev.next = this;
  }
}
```



```
Node head = new Node();
Node node1 = new Node(1, head);
```

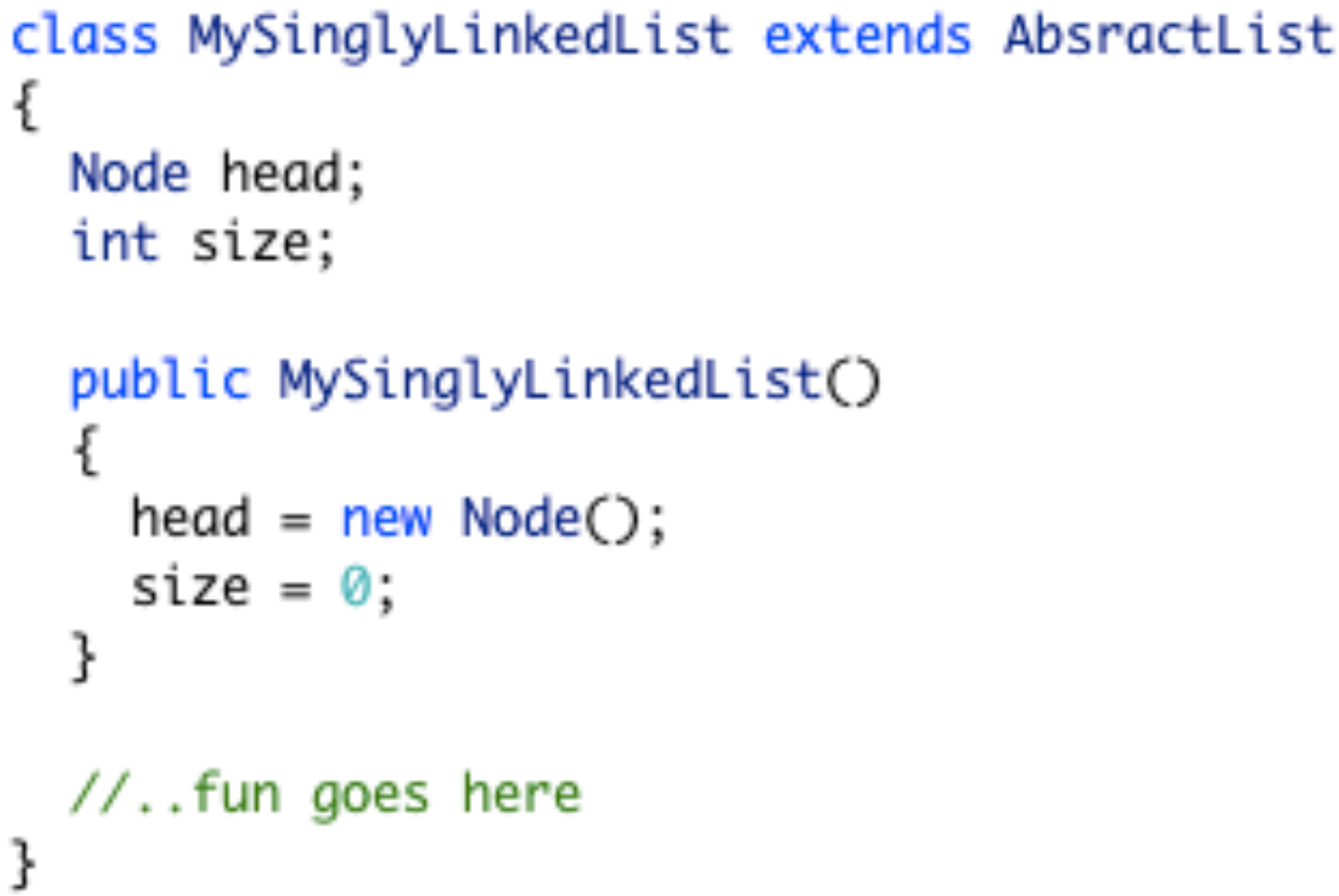
Do it yourself



```
public class Node {
  Object data;
  Node next;

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

  public Node (Object o, Node prev)
  {
    data = o;
    next = prev.next;
    prev.next = this;
  }
}
```

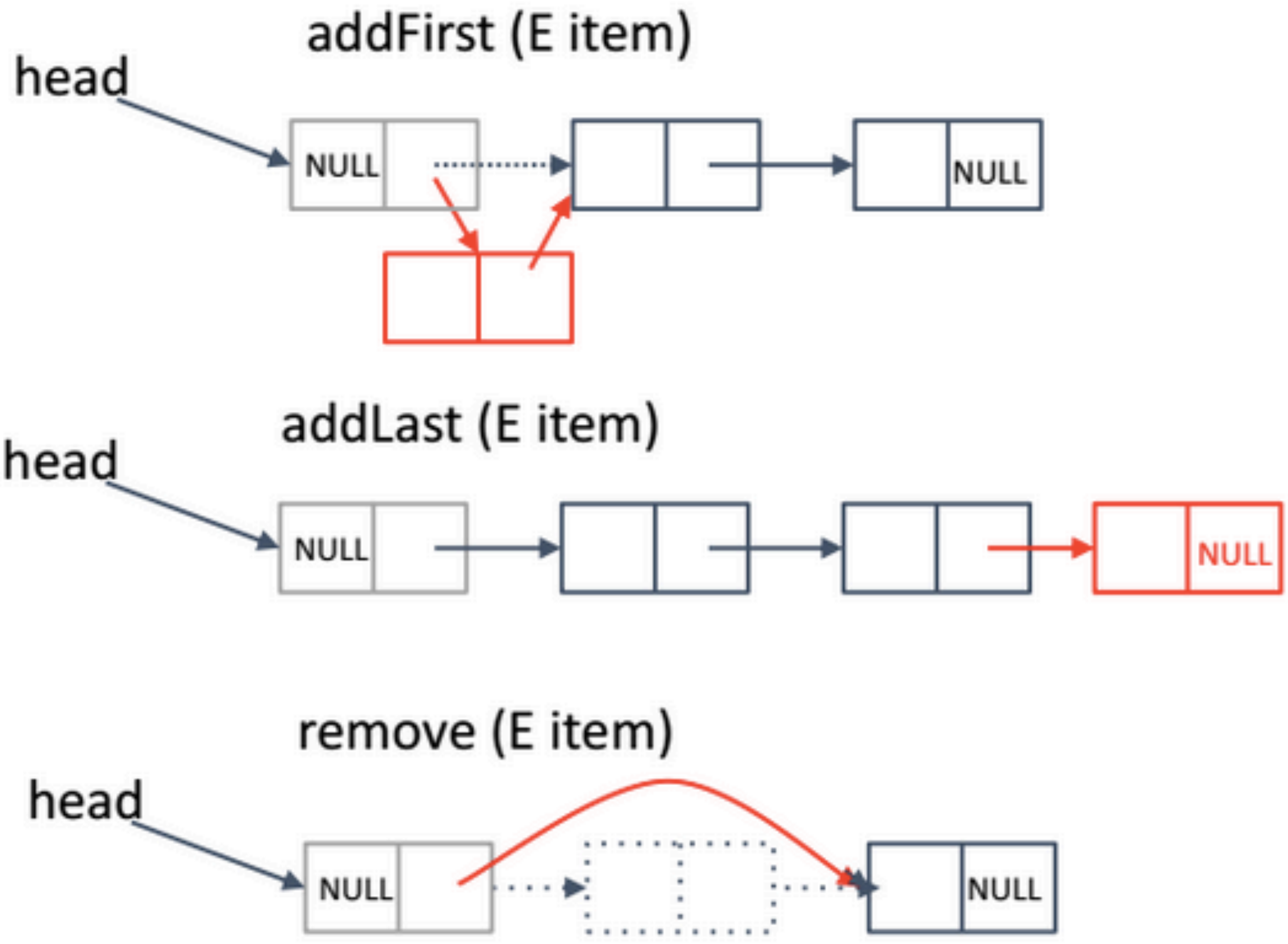
```
Node head = new Node();
Node node1 = new Node(1, head);
Node node2 = new Node(2, head);
```

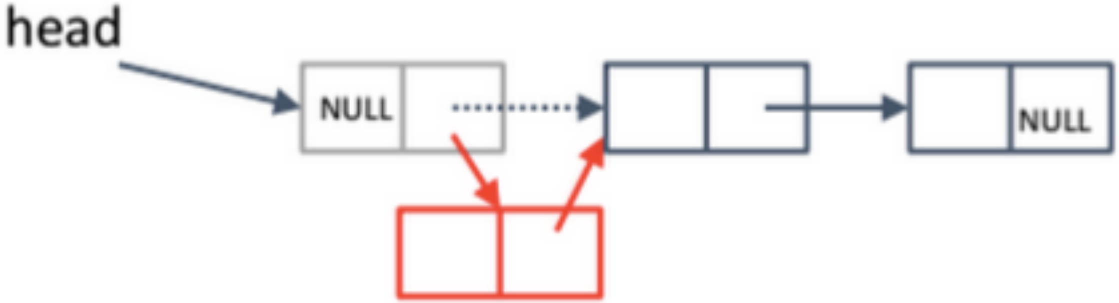
What do we have after the constructor call?

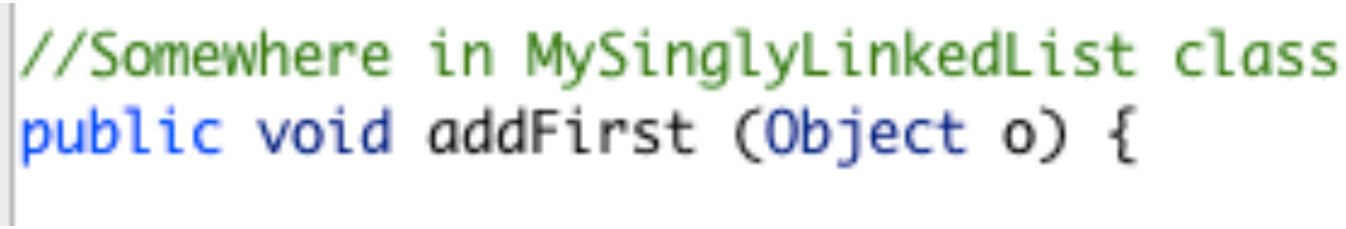


```
class MySinglyLinkedList extends AbsractList
{
  Node head;
  int size;

  public MySinglyLinkedList()
  {
    head = new Node();
    size = 0;
  }

  //..fun goes here
}
```

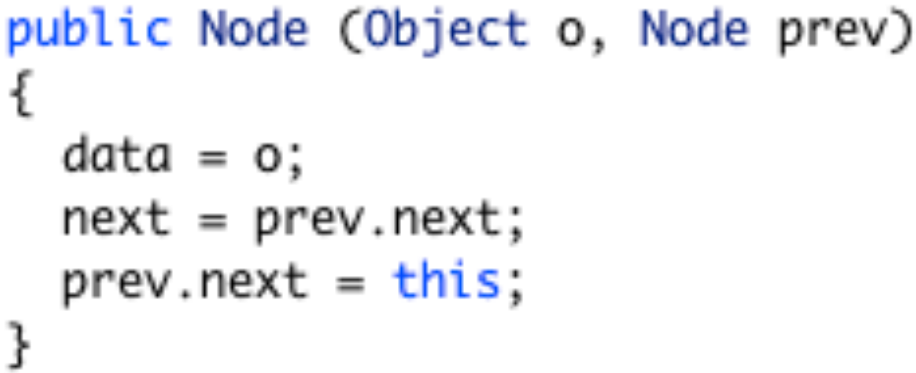
A few methods



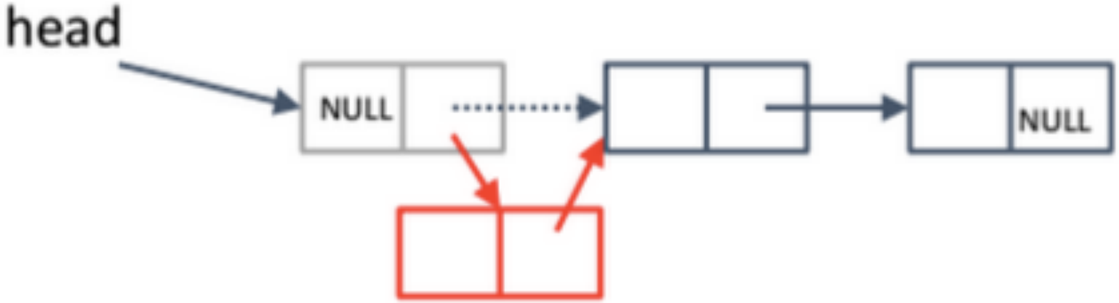


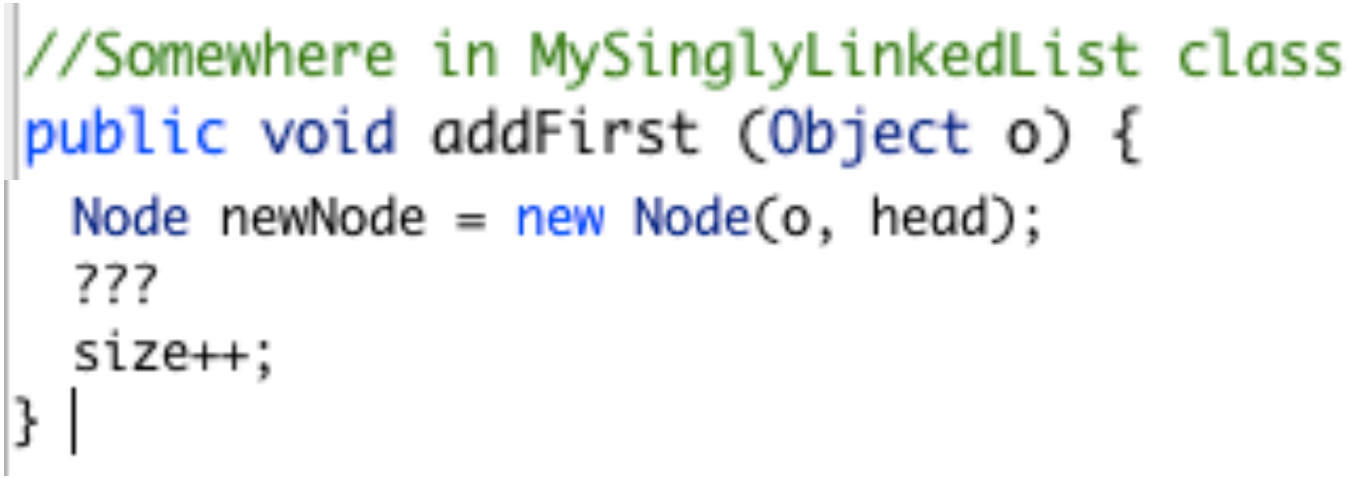


```
//Somewhere in MySinglyLinkedList class
public void addFirst (Object o) {
```



```
public Node (Object o, Node prev)
{
  data = o;
  next = prev.next;
  prev.next = this;
}
```





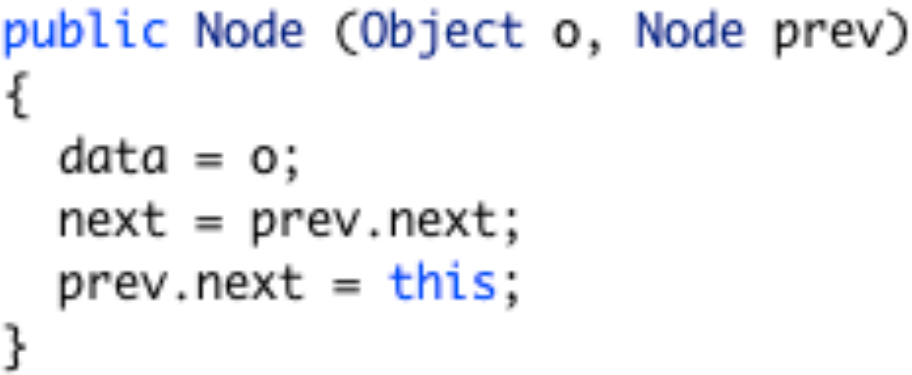
```
//Somewhere in MySinglyLinkedList class
public void addFirst (Object o) {
  Node newNode = new Node(o, head);
  ???
  size++;
}
```

A: it is complete

B: head = head.next;

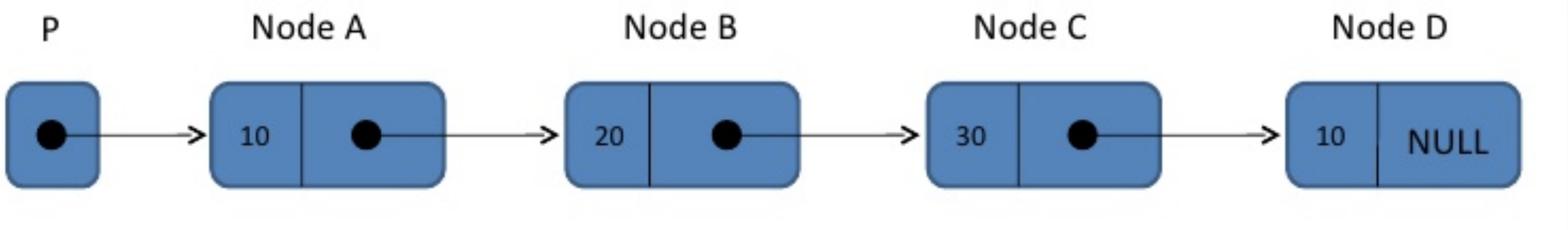
C: head = newNode;

D: newNode.next = head;



```
public Node (Object o, Node prev)
{
  data = o;
  next = prev.next;
  prev.next = this;
}
```

Add to the back: NodeE.

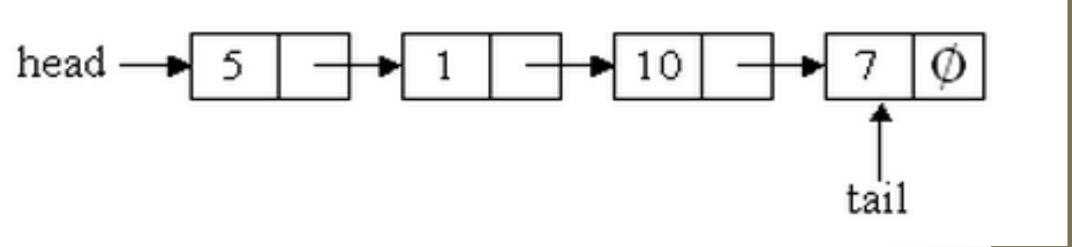


- A: NodeD.next = NodeE;
- B: need to loop through the list to get to node D.

  then NodeD.next = NodeE;
- C: NodeC.next.next = NodeE;
- D: Other

List with Head and Tail

- How to add Node E to the end in this case?

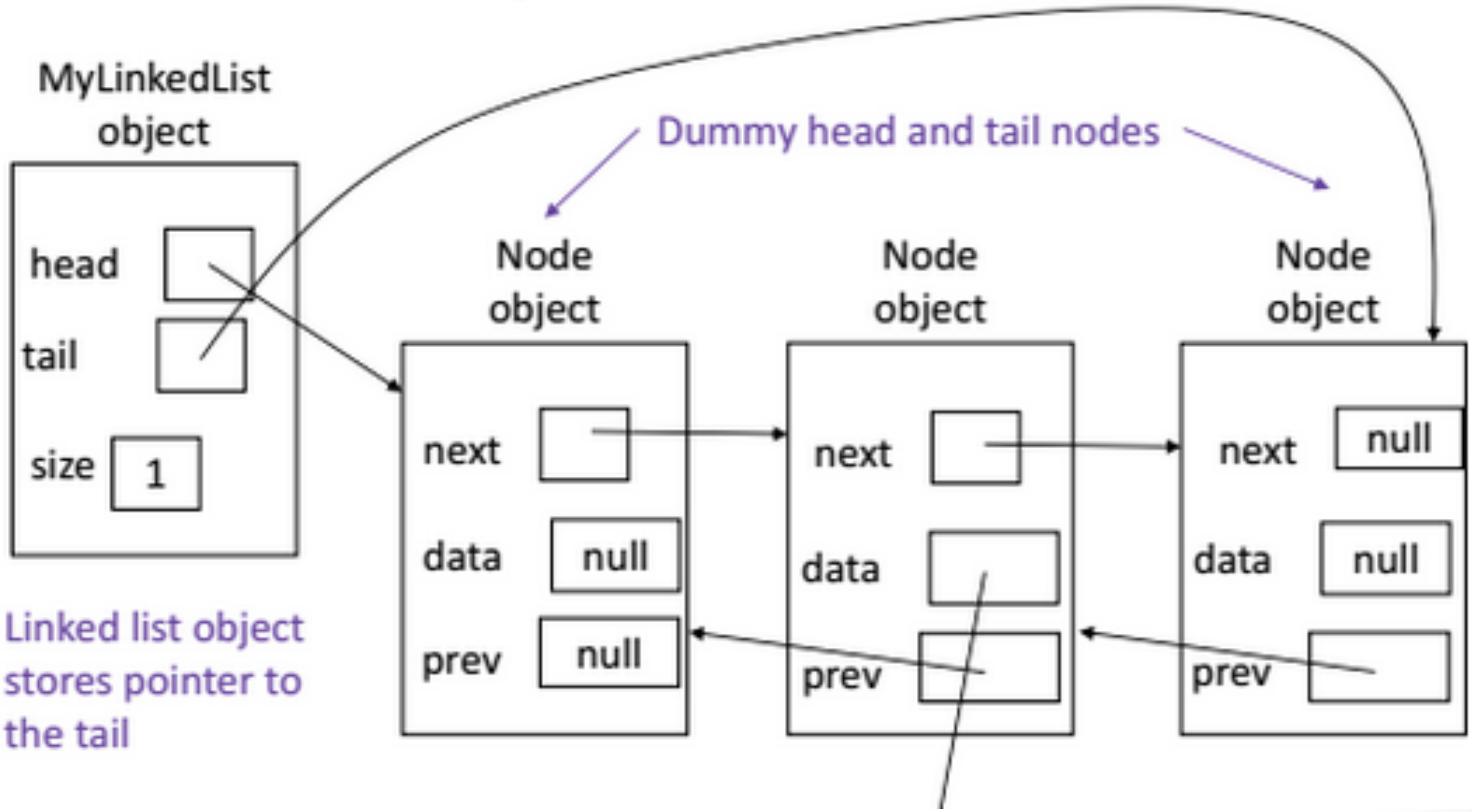


- A: tail = Node E;
- B: tail.next = Node E;
- C: tail = Node E;

  tail.next = Node E;
- D: tail.next = Node E;

  tail = Node E;

HW2: Doubly linked lists

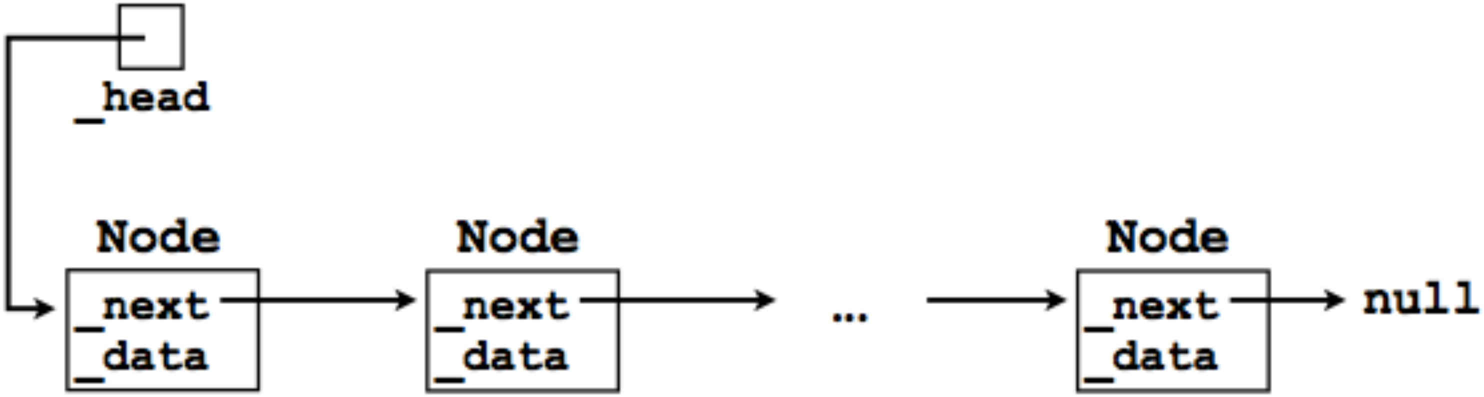


ITERATORS

Iterating through the whole list

- Suppose we wish to iterate through the *entire list* and print out the **data** in each node?

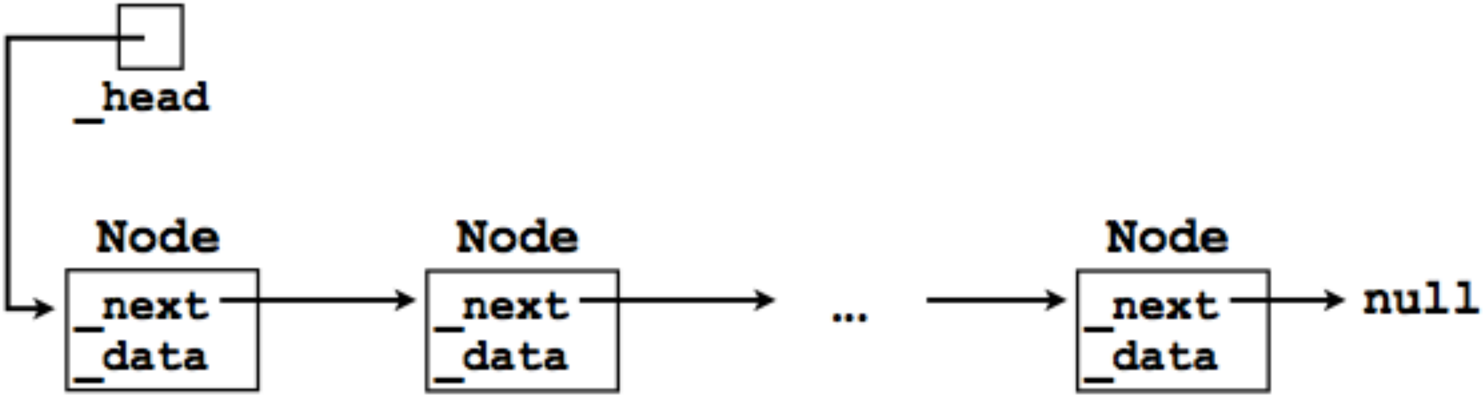
```
        Node cursor = head;
```



Iterating through the whole list

- Suppose we wish to iterate through the *entire list* and print out the **data** in each node?

```
Node cursor = head;
while (               ) {


}
```



Iterating through the whole list

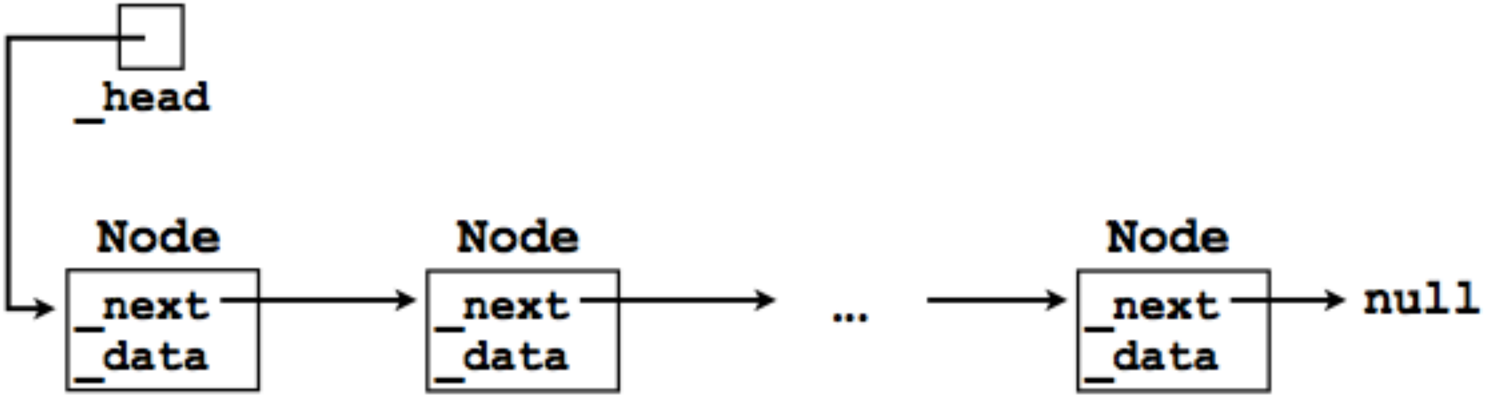
```
• Node cursor = head;
   while (  ?????  ) {
        System.out.println(cursor.data);   }
```

A: cursor=head

B: cursor!=null

C: cursor.next!=null

D: head!=null

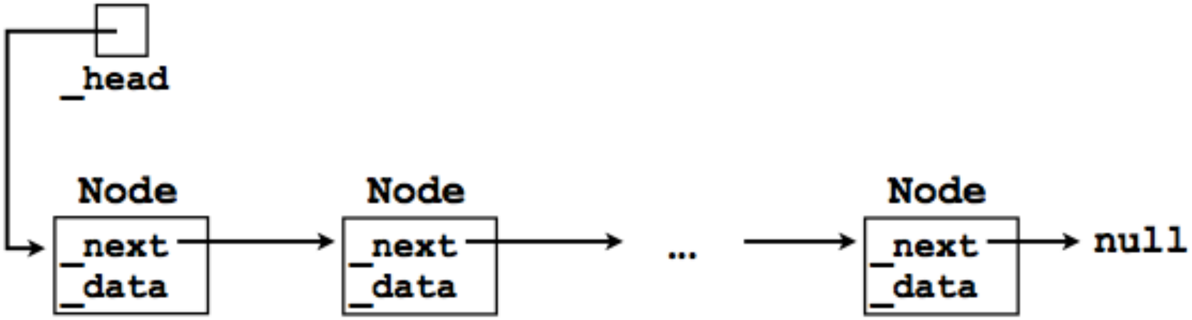


Iterating through the whole list

-
  ```
  Node cursor=_head;
  while (cursor!=null) {
    System.out.println(cursor.data);
        Done??
   }
  ```

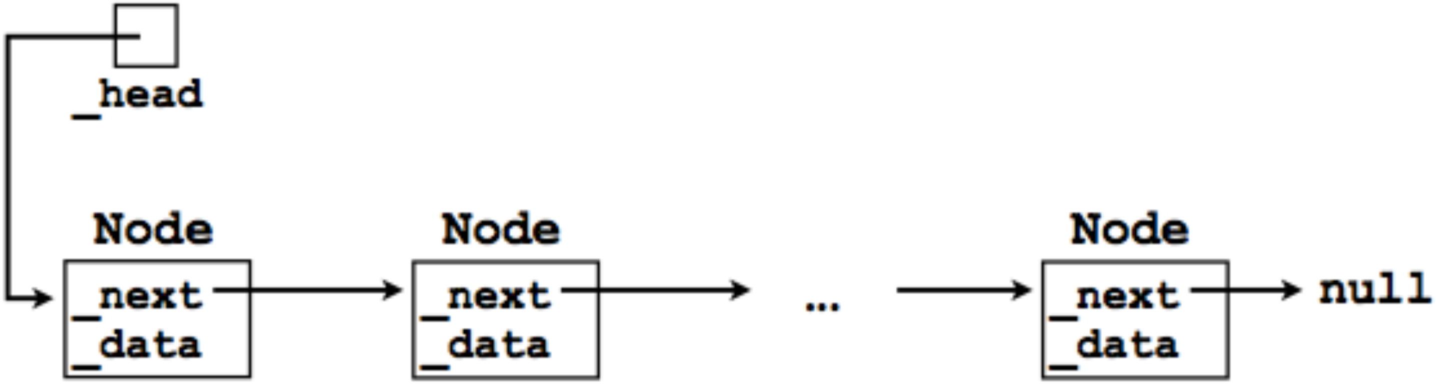
A: Yes

B: No



Iterating through the whole list

```
Node cursor=_head;
 while (cursor!=null) {
   System.out.println(cursor.data);
   cursor=cursor.next;
 }
```



Iterating through the whole list

- Could you iterate through the list using a for-loop?

- A: Yes
- B: No

Iterator: life without them

- How would you implement get method?

`Object get(int index)` **throws IndexOutOfBoundsException;**

Using either for/while loop:

```
Node cursor = head;
for (int i=0; i<index; i++)
     cursor=cursor.next;
return cursor.data
```

Iterating over elements of a data structure

- Many ADTs offer the user the ability to iterate over all of their elements in some “natural order”.

- With the simple `List` interface this is already possible using the `get (index)` methods:

```
int size = linkedList.size();
    for (int i = 0; i < size; i++) {
    System.out.println(linkedList.get(i));
}
```

VERY slow, always starts from the beginning

Iterators: performance benefits

- An “iterator” object helps us to avoid this wasted computation.
- An iterator is a “helper object” with which the user can iterate across all elements in a data structure.
- **The iterator will “remember” where it left off.**

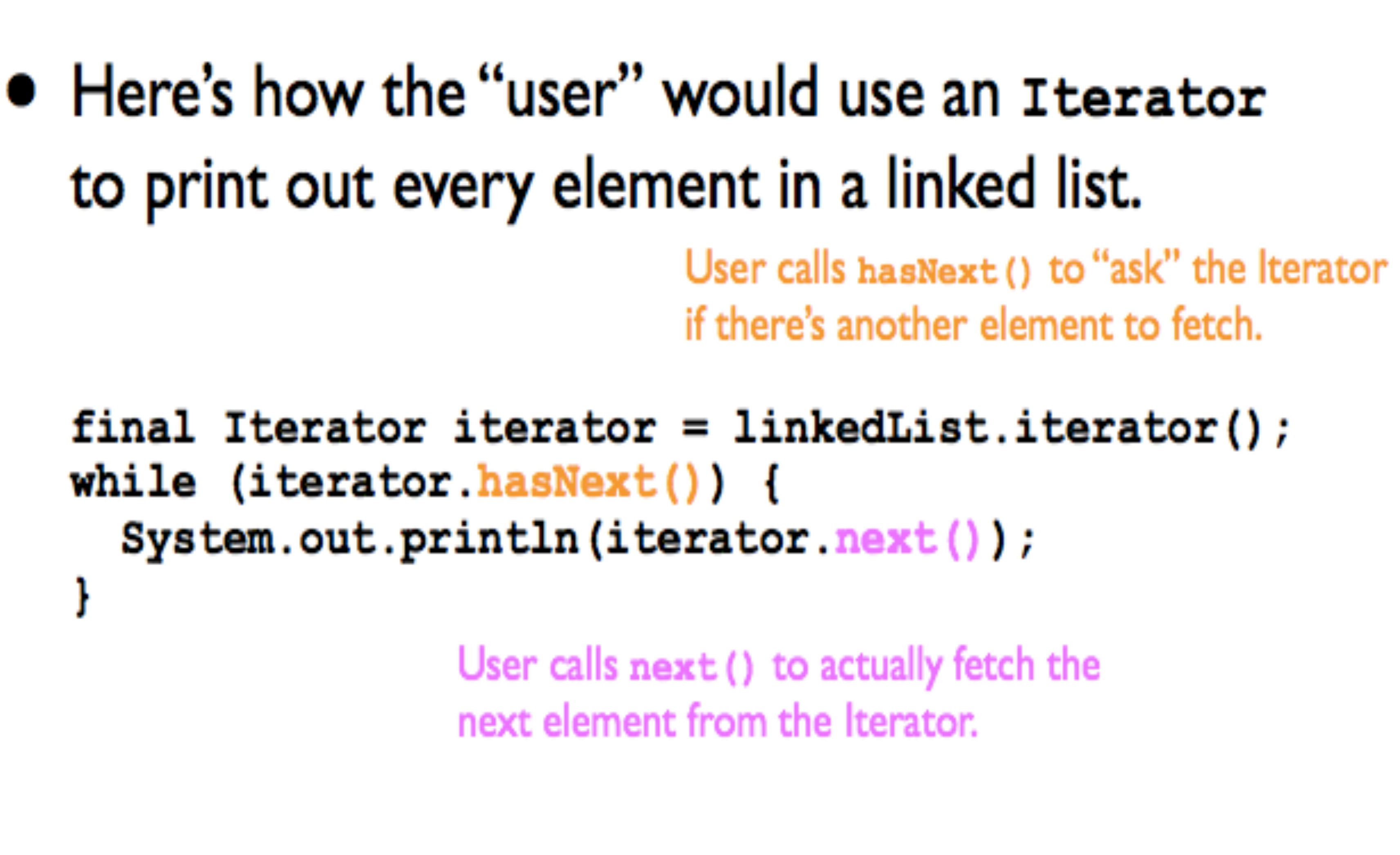
Iterators: software design gain

- Iterators are also useful because they offer a uniform way of accessing all of a data structure’s elements.

- Even very different data structures --e.g., graphs and lists -- can both support iterators.

- An “iterator” is one of the fundamental design patterns of software engineering.

How Iterators are used



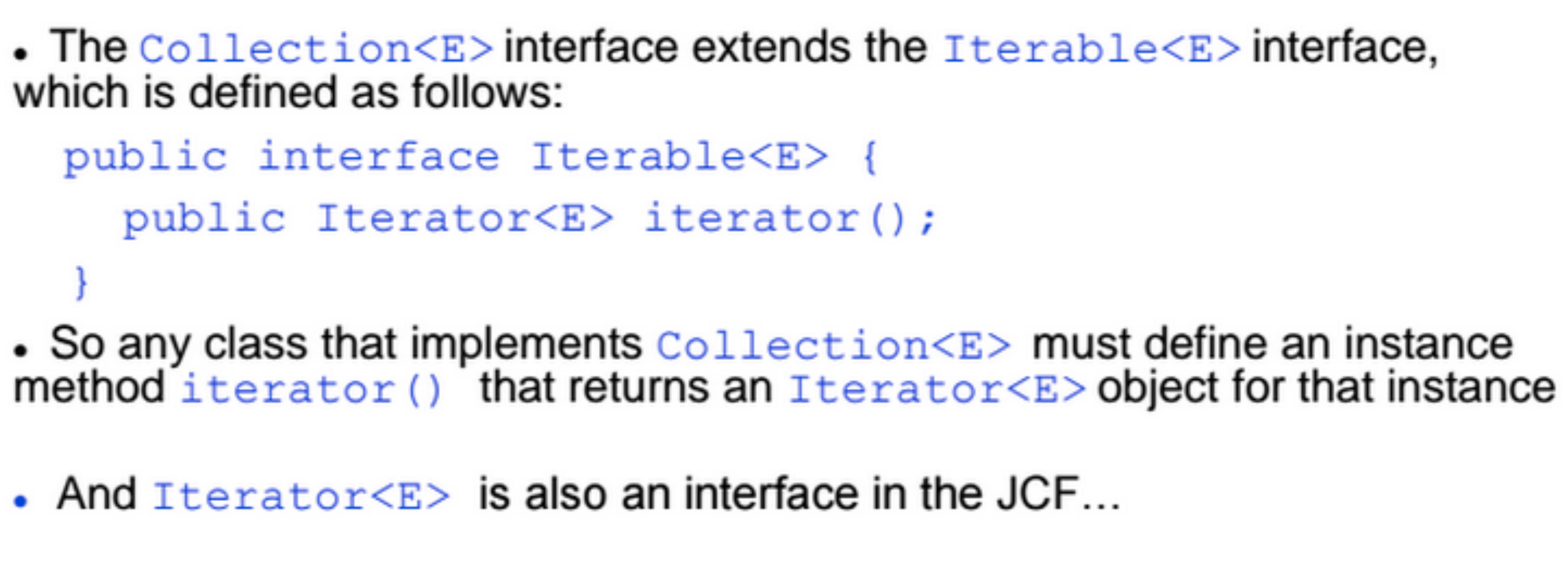
- Here's how the "user" would use an **`Iterator`** to print out every element in a linked list.

User calls **`hasNext()`** to "ask" the Iterator if there's another element to fetch.

```
final Iterator iterator = linkedList.iterator();
while (iterator.hasNext()) {
  System.out.println(iterator.next());
}
```

User calls **`next()`** to actually fetch the next element from the Iterator.

Iterable Interface



- The Collection<E> interface extends the Iterable<E> interface, which is defined as follows:

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

- So any class that implements Collection<E> must define an instance method iterator() that returns an Iterator<E> object for that instance

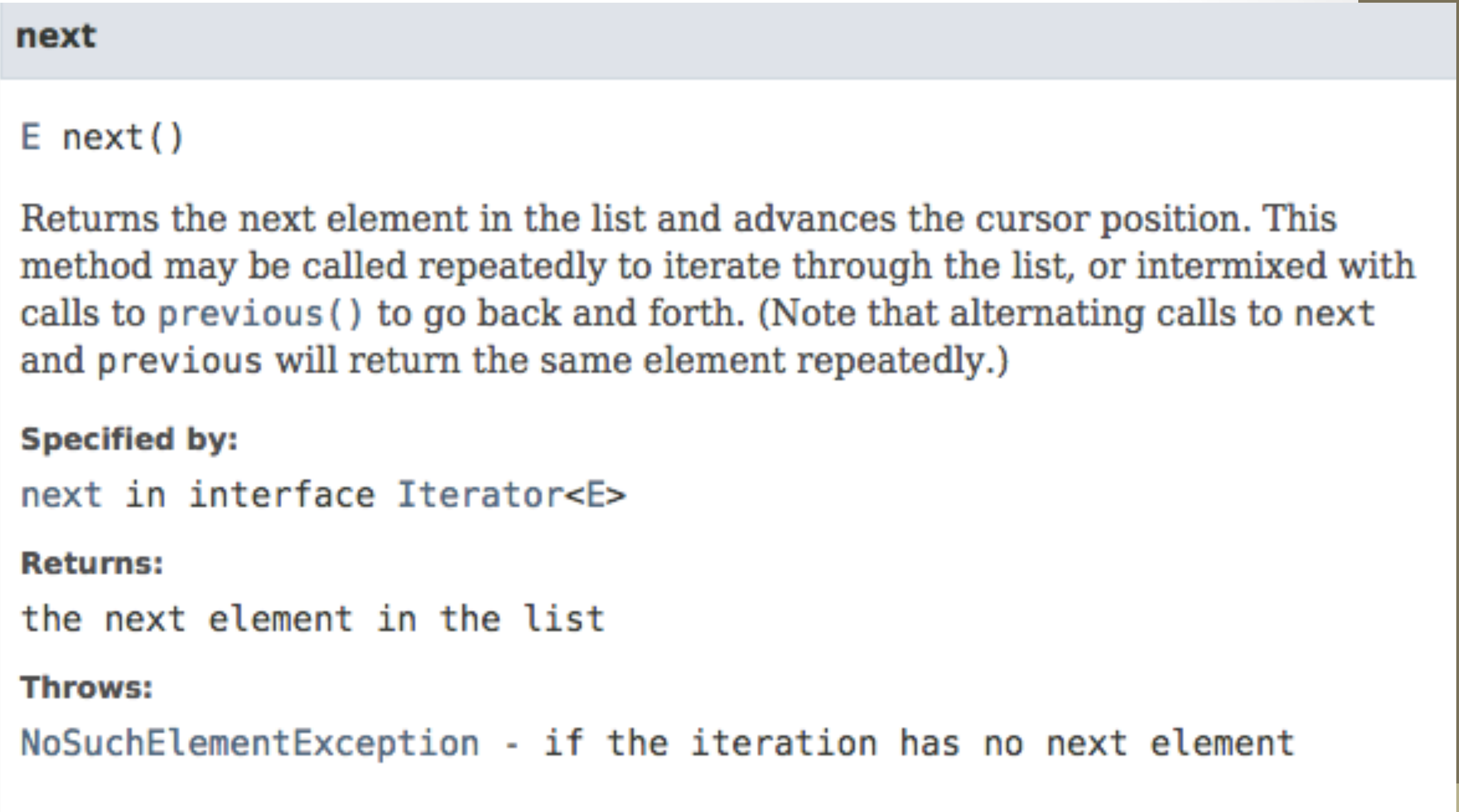
- And Iterator<E> is also an interface in the JCF…

Interface Iterator

- In Java, the `Iterator` interface contains *three* method signatures:

```
boolean hasNext();
Object next();
void remove();
```

- The `ListIterator` interface adds a few more methods.
  - Boolean hasPrevious
  - Object Previous
  - …



next

`E next()`

Returns the next element in the list and advances the cursor position. This method may be called repeatedly to iterate through the list, or intermixed with calls to `previous()` to go back and forth. (Note that alternating calls to `next` and `previous` will return the same element repeatedly.)

Specified by:

`next` in interface `Iterator<E>`

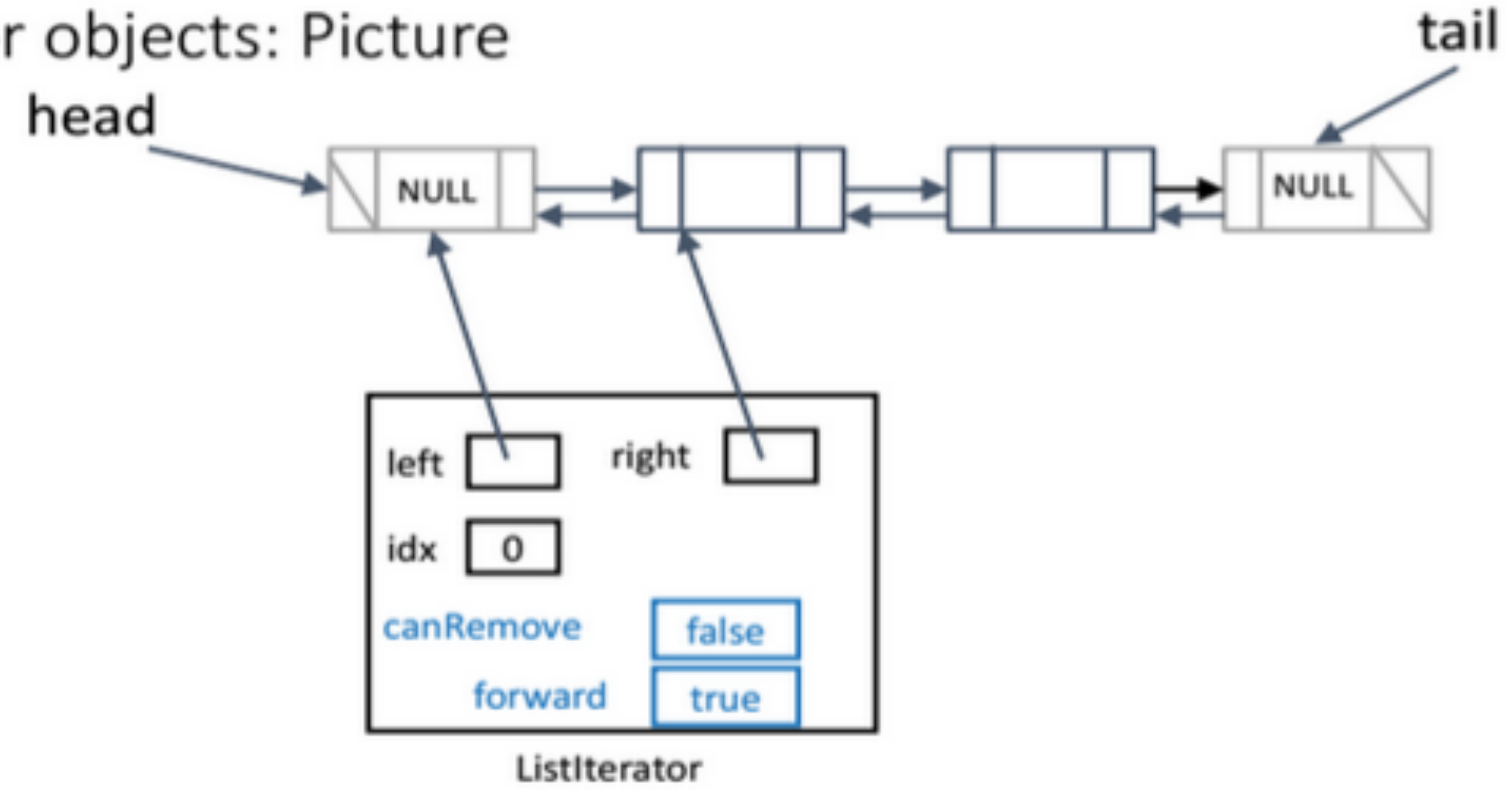
Returns:

the next element in the list

Throws:

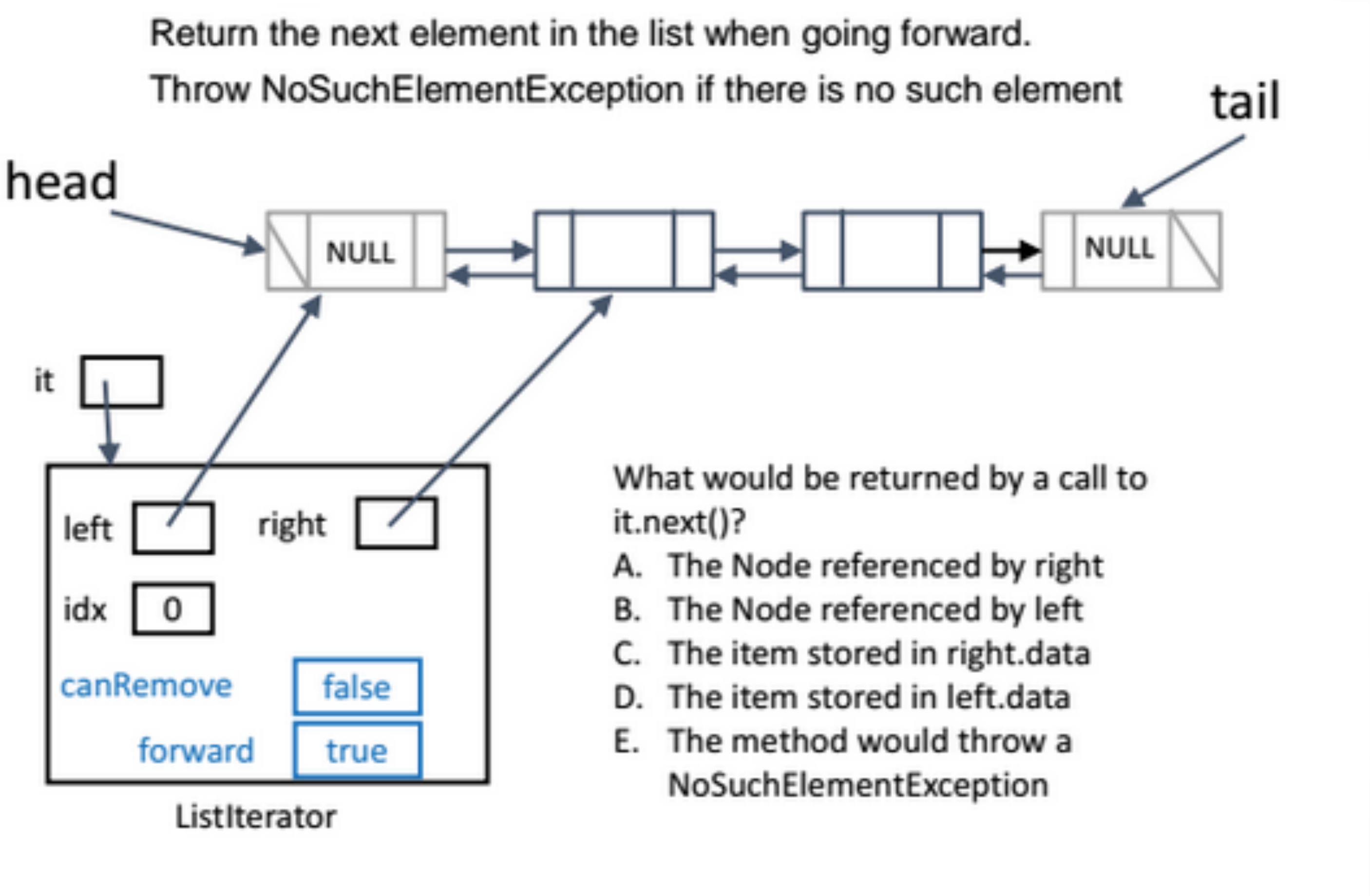
`NoSuchElementException` - if the iteration has no next element

Iterator objects: Picture



- **Forward**: direction of the iterator
- **canRemove**: only true if last call of iterator was next() or prev()

Object next()



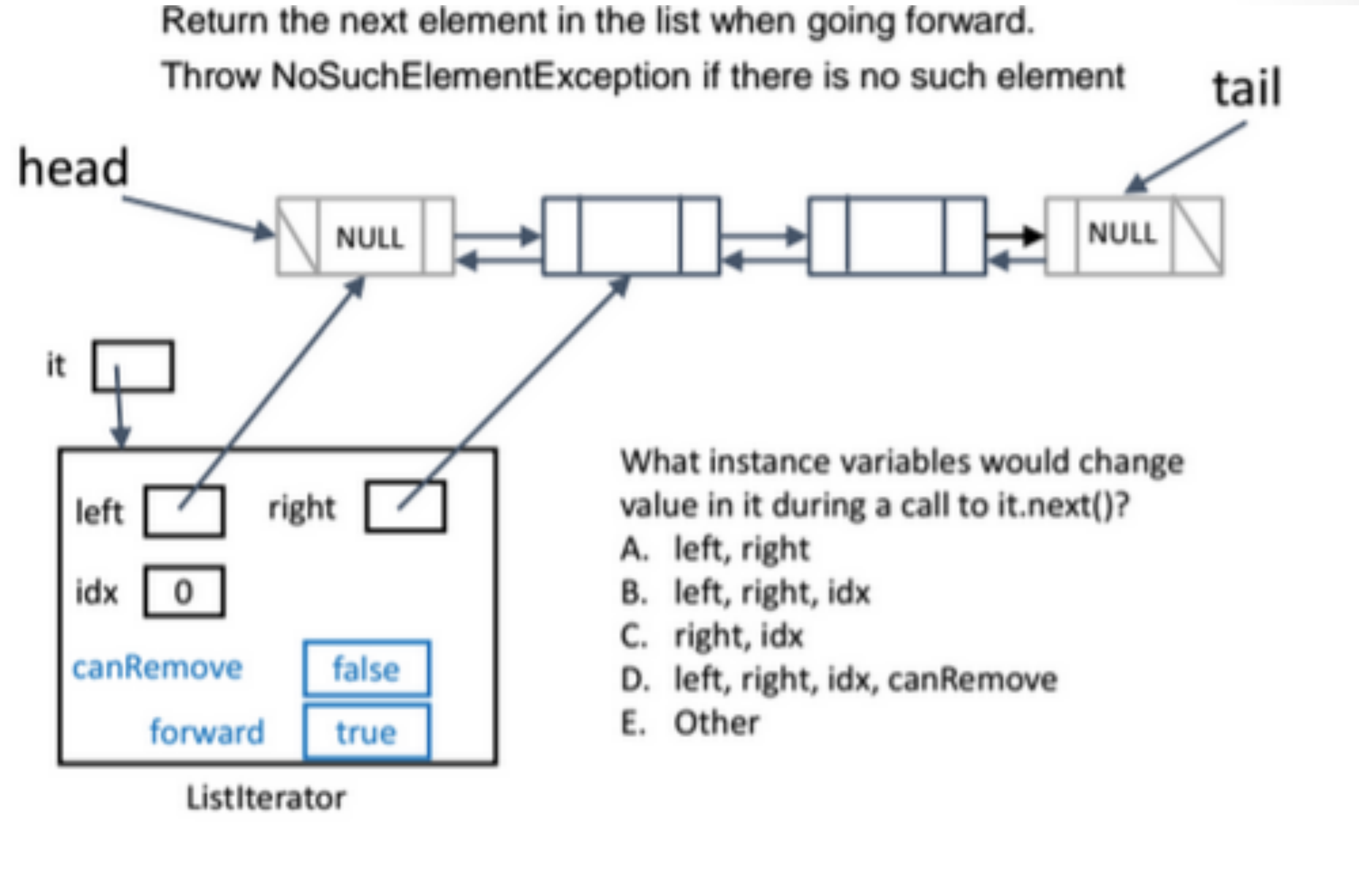
Return the next element in the list when going forward.

Throw NoSuchElementException if there is no such element

What would be returned by a call to it.next()?

A. The Node referenced by right
B. The Node referenced by left
C. The item stored in right.data
D. The item stored in left.data
E. The method would throw a NoSuchElementException

Object next()

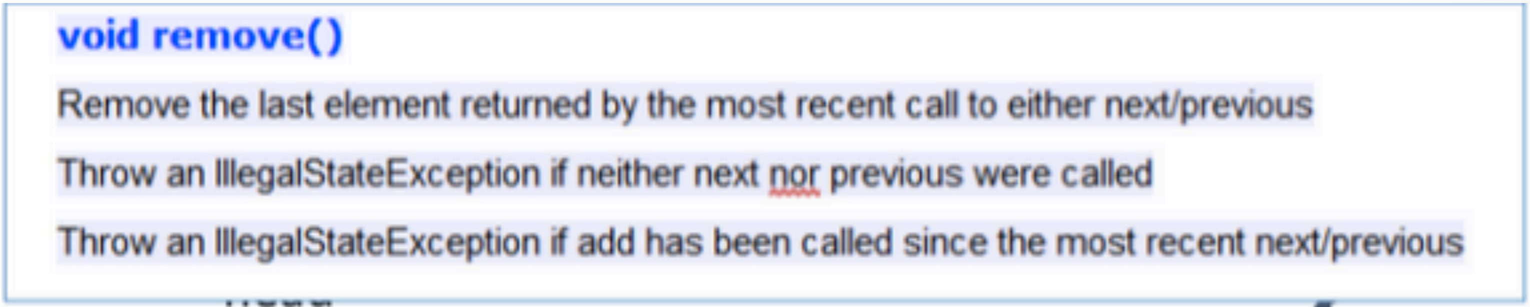


Return the next element in the list when going forward.

Throw NoSuchElementException if there is no such element

What instance variables would change value in it during a call to it.next()?

A. left, right
B. left, right, idx
C. right, idx
D. left, right, idx, canRemove
E. Other

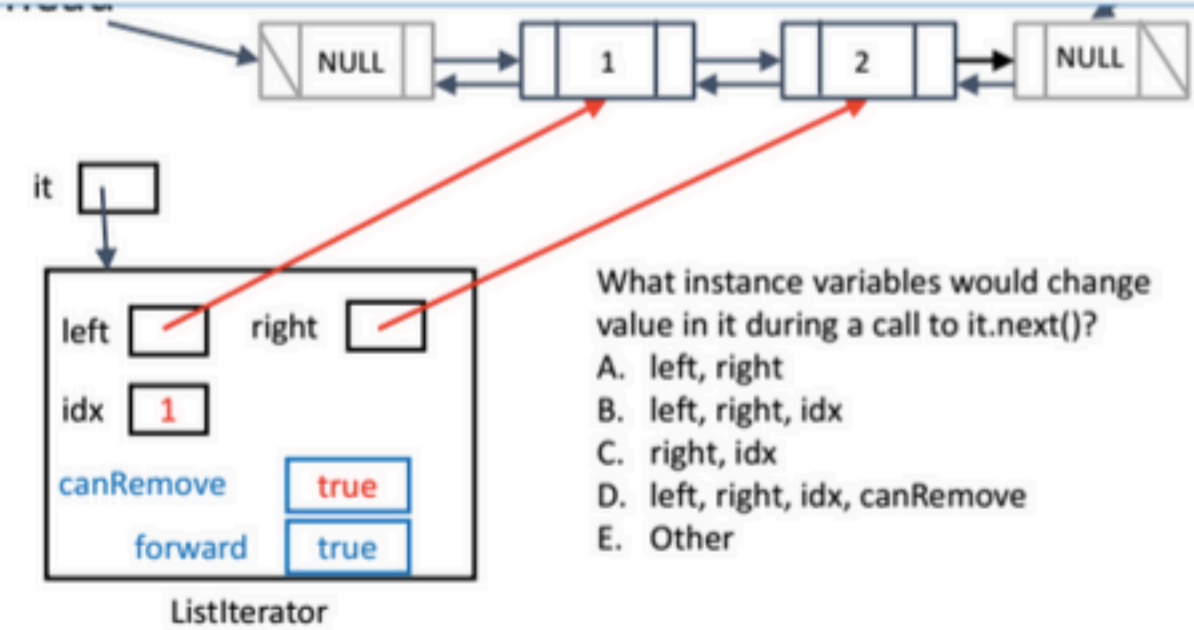


void remove()

Remove the last element returned by the most recent call to either next/previous

Throw an IllegalStateException if neither next nor previous were called

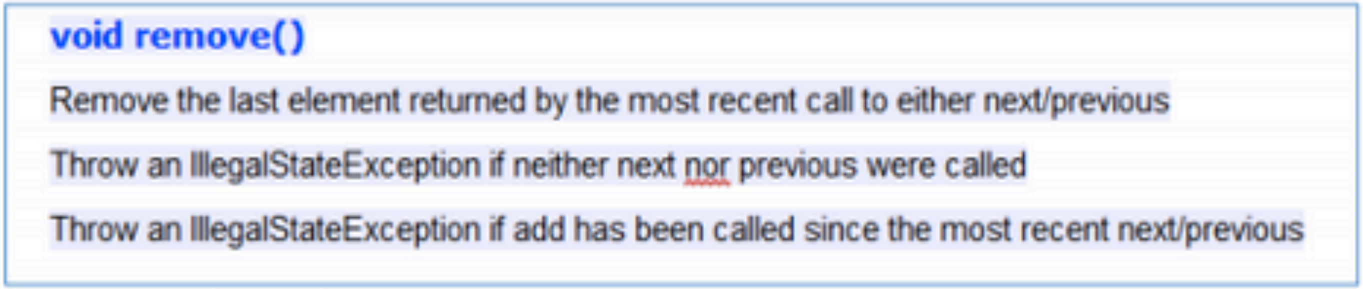
Throw an IllegalStateException if add has been called since the most recent next/previous



ListIterator

What instance variables would change value in it during a call to it.next()?

A. left, right
B. left, right, idx
C. right, idx
D. left, right, idx, canRemove
E. Other

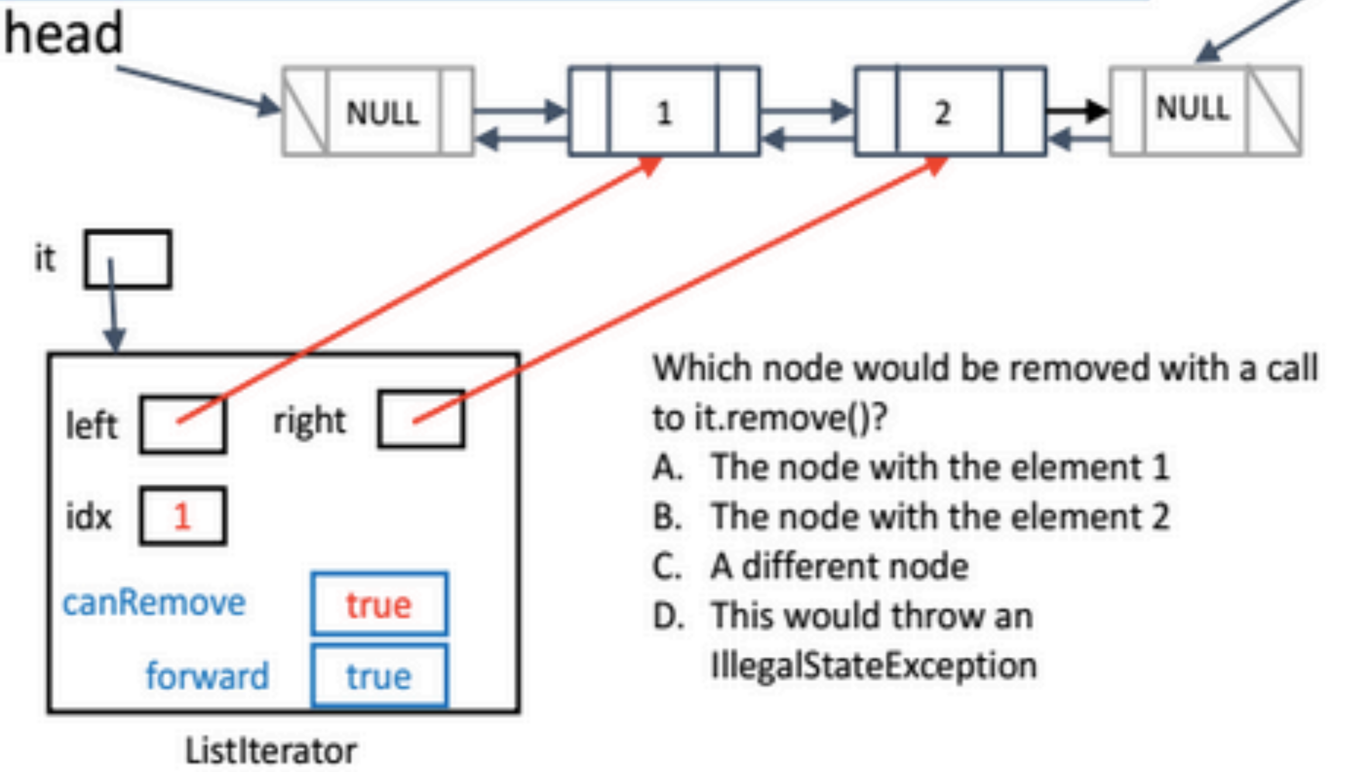


void remove()

Remove the last element returned by the most recent call to either next/previous

Throw an IllegalStateException if neither next nor previous were called

Throw an IllegalStateException if add has been called since the most recent next/previous



Which node would be removed with a call to it.remove()?

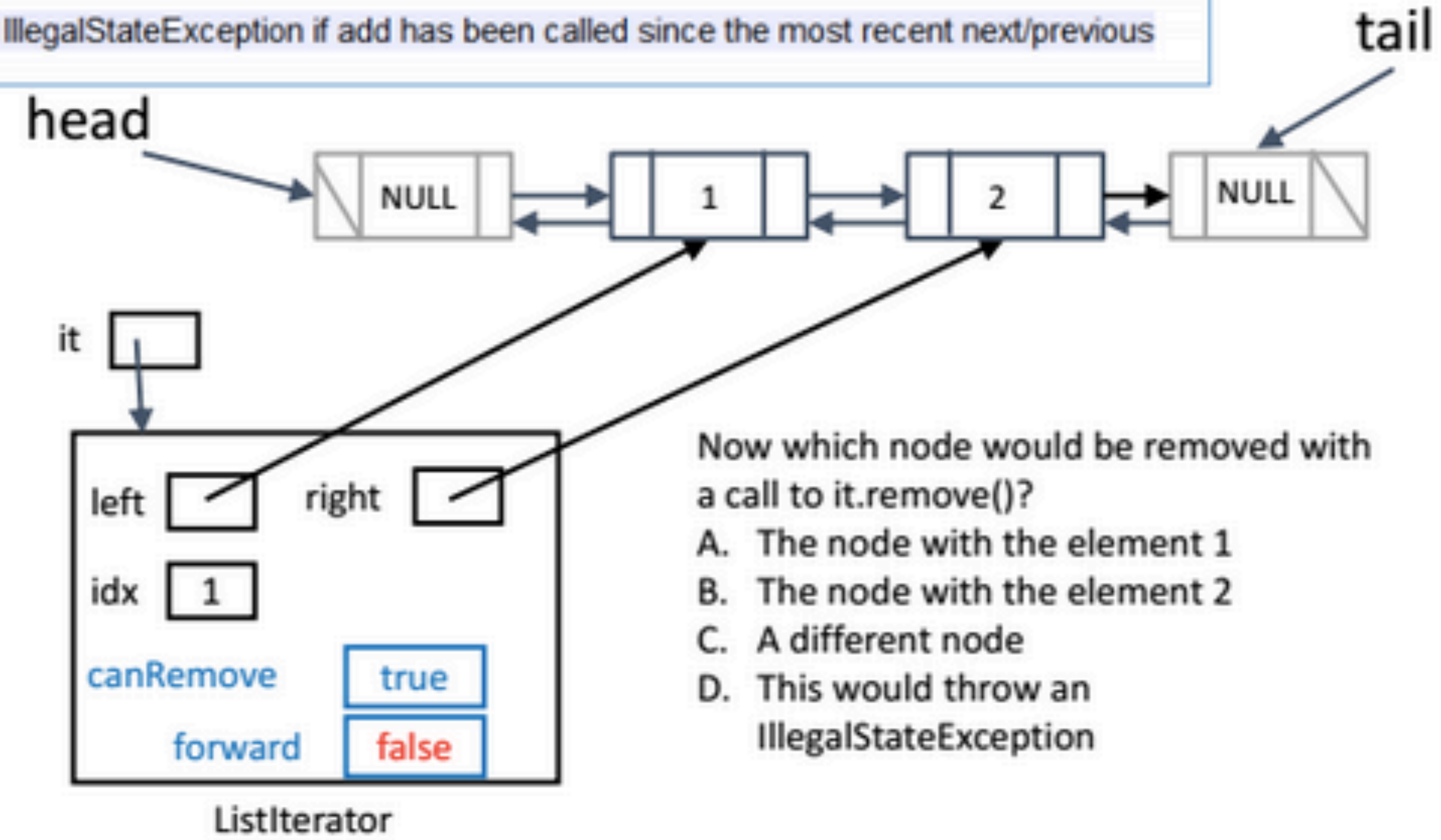
A. The node with the element 1
B. The node with the element 2
C. A different node
D. This would throw an IllegalStateException

void remove()

Remove the last element returned by the most recent call to either next/previous

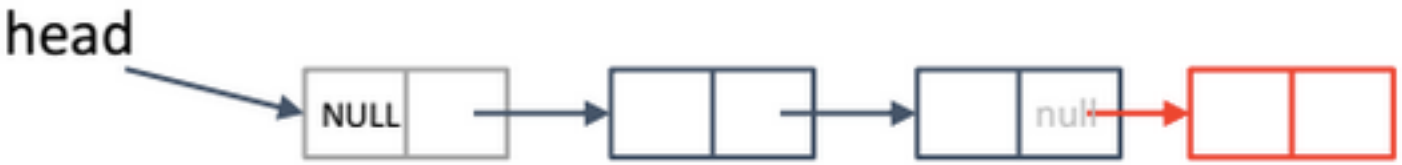
Throw an IllegalStateException if neither next nor previous were called

Throw an IllegalStateException if add has been called since the most recent next/previous



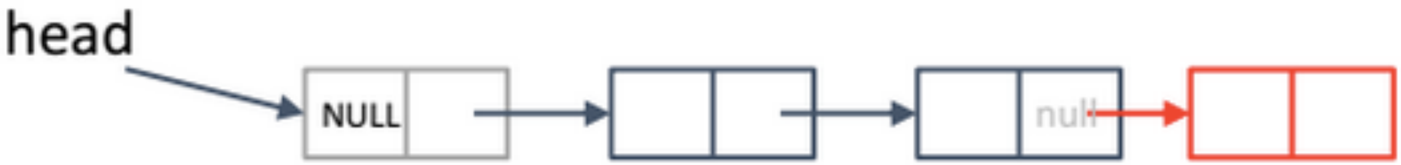
Now which node would be removed with a call to it.remove()?

A. The node with the element 1
B. The node with the element 2
C. A different node
D. This would throw an IllegalStateException

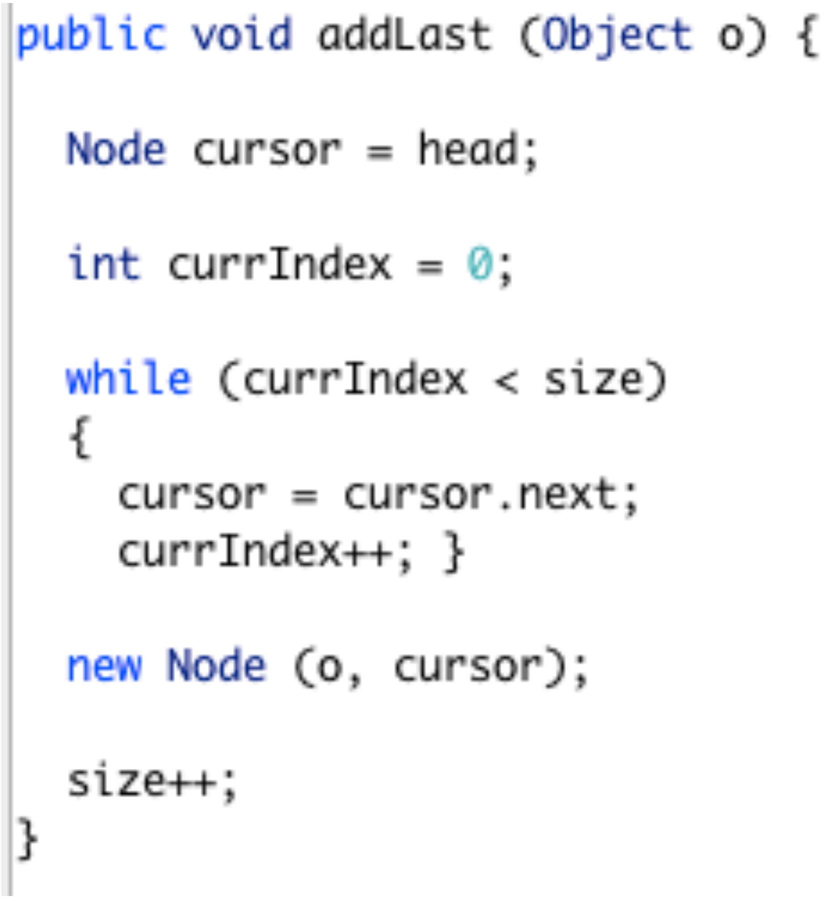


```
public void addLast (Object o) {

  Node cursor = head;

  while (???) {cursor = cursor.next;}

  new Node (o, cursor)·

  size++;
}
```

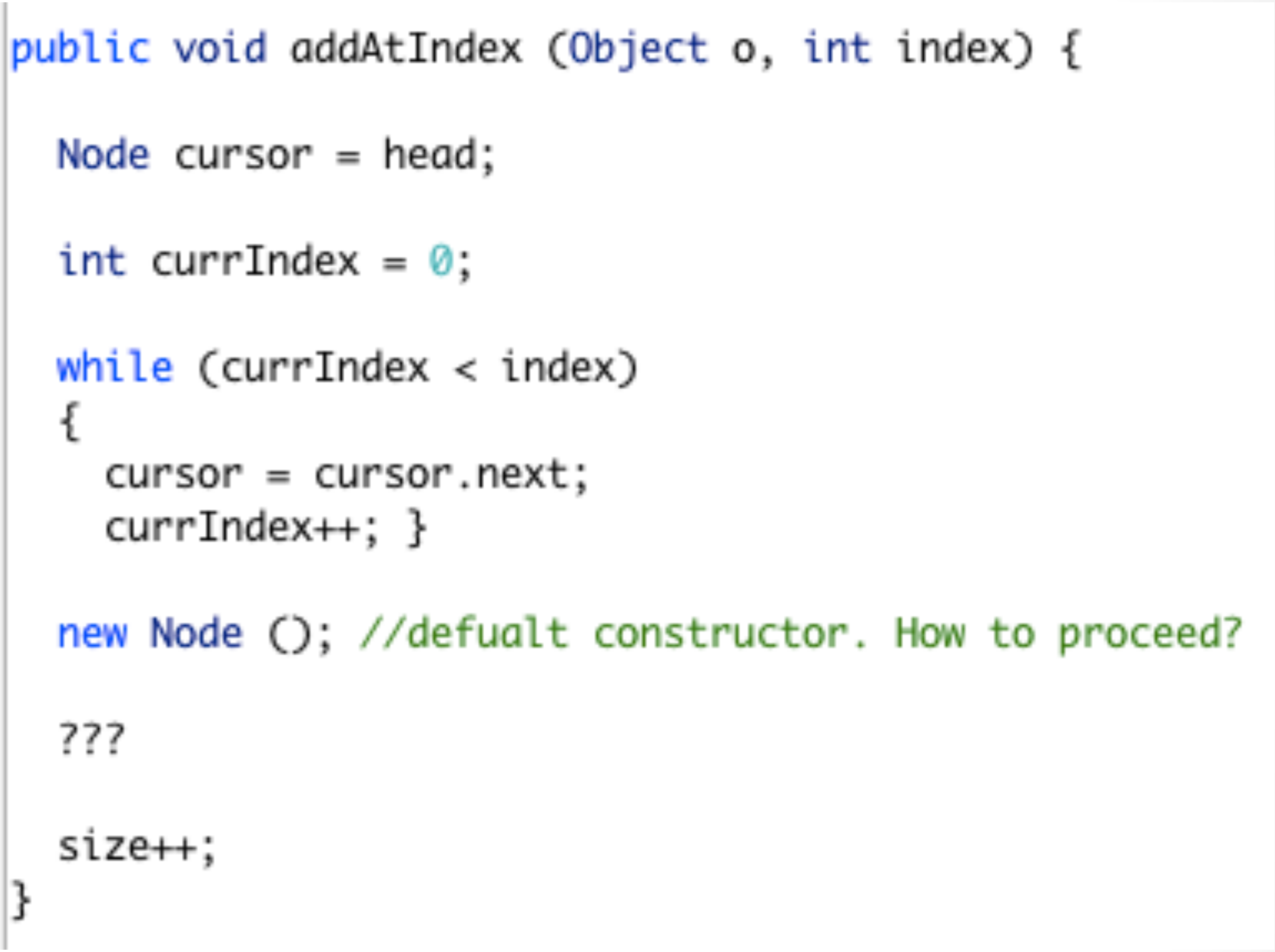
- A: cursor == head
- B: cursor.next !=null
- C: cursor != null
- D: head !=null



- Change to insert at an arbitrary location?



```
public void addLast (Object o) {

  Node cursor = head;

  int currIndex = 0;

  while (currIndex < size)
  {
    cursor = cursor.next;
    currIndex++; }

  new Node (o, cursor);

  size++;
}
```



```
public void addAtIndex (Object o, int index) {

  Node cursor = head;

  int currIndex = 0;

  while (currIndex < index)
  {
    cursor = cursor.next;
    currIndex++; }

  new Node (); //defualt constructor. How to proceed?

  ???

  size++;
}
```

Removal from Linked List

- public Object remove (int position) {

Ideas?

- }

Reading assignment

- Java documentation for your project if needed.
- No reading quiz on Tuesday.
- There is going to be in class quiz.