Lecture 4-5.

Linked Lists

Marina Langlois

Some slides were borrowed from Prof. Alvarado.

Quick check

- Did you have a chance to read through hw2?

- A: Yes. I'm speech-less.
- B: Yes. I still can talk ☺
- C: Not yet but going to read it today
- D: Not yet, I have time to start. You gave us 10 days.
- E: Not yet, a few days will be enough, I'm a good programmer.

Plan for today

- 1. Understanding a starter code
- 2. Memory model(quick review)
- 3. Linked Lists

Implementation of the List interface with a LinkedList

- `public class MySinglyLL<E> implements List<E>`

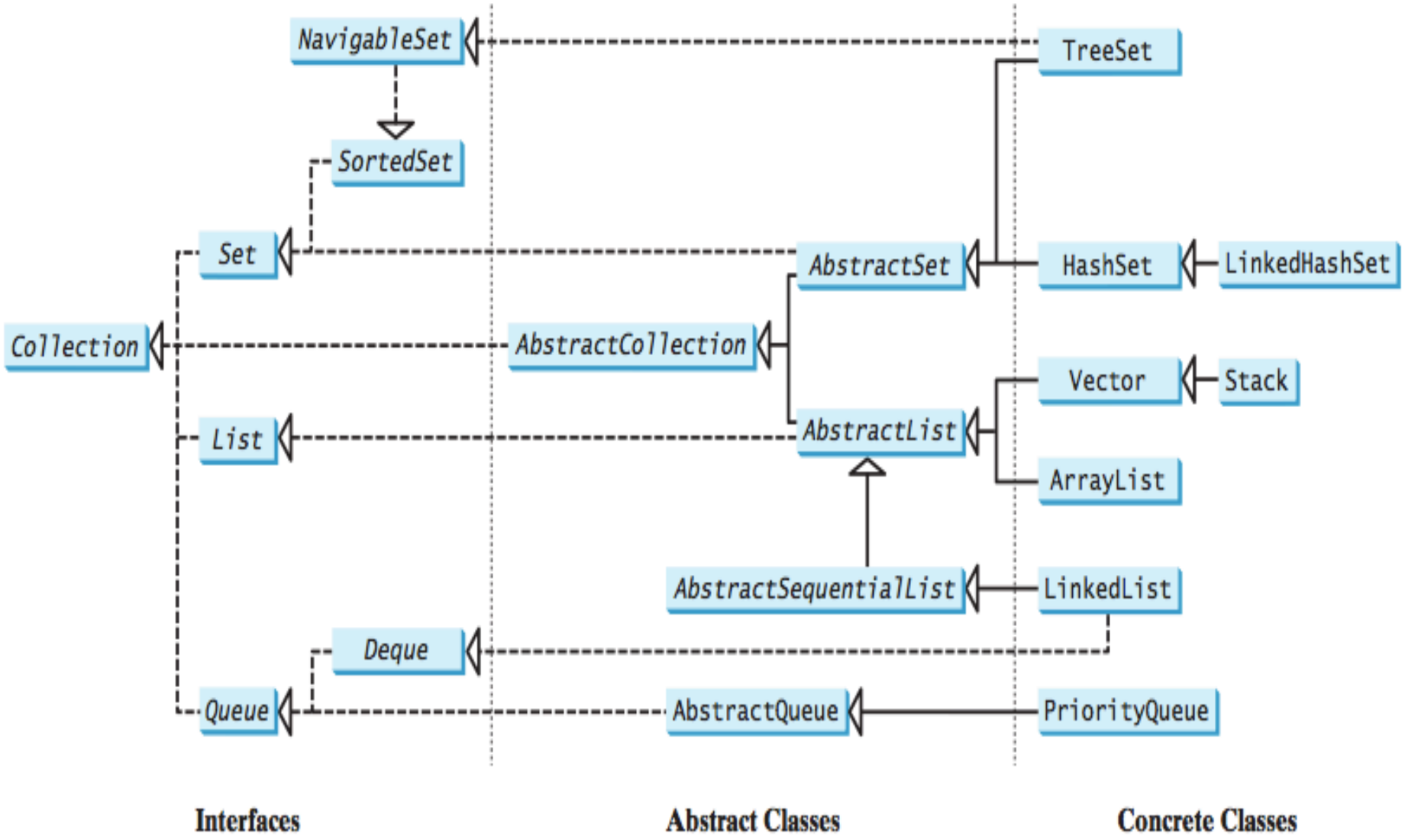
the implementation

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.

A collection is a container that stores objects.



An implementation of the List interface: a LinkedList

The *implementation*

The *ADT*

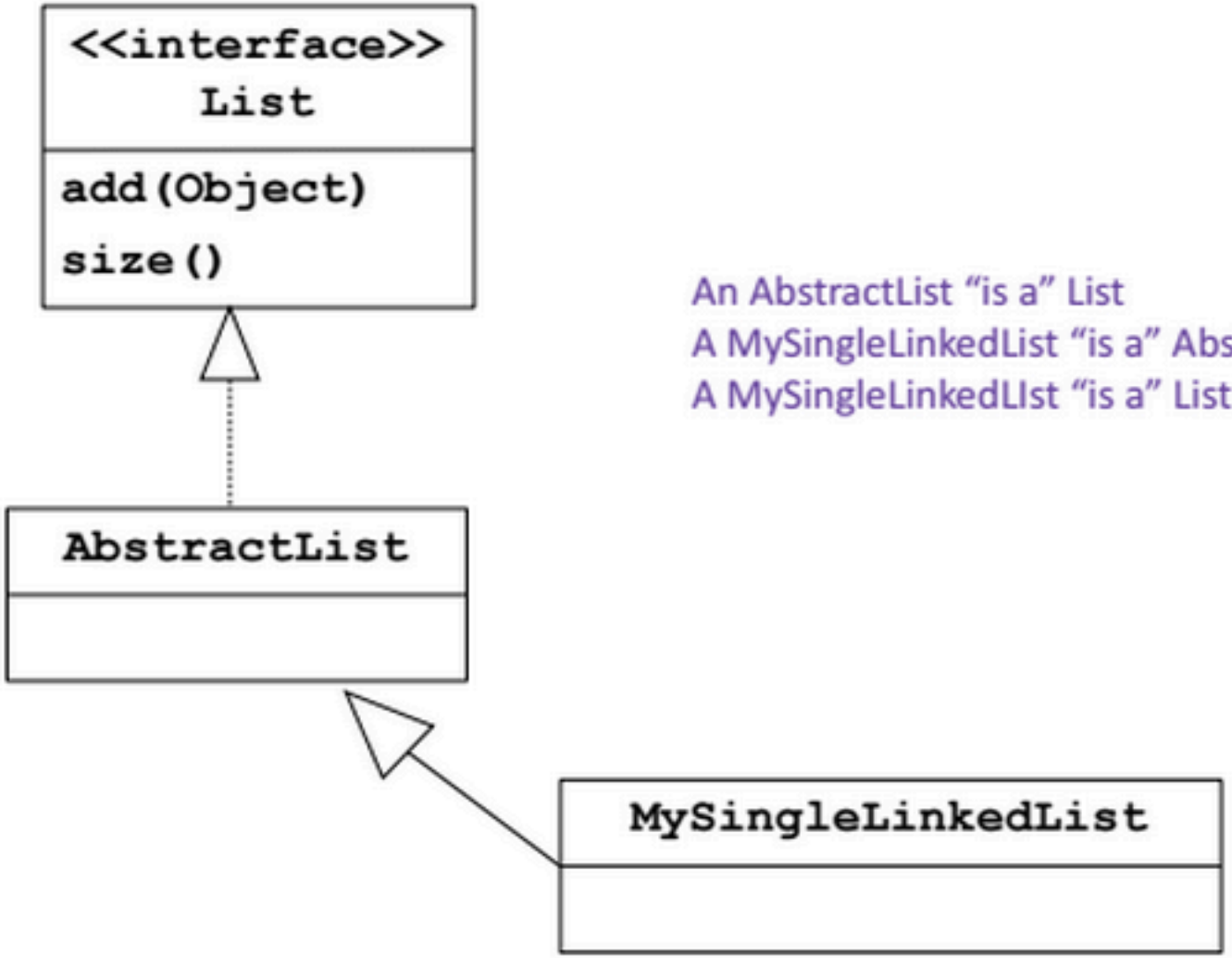
```
public class MySingleLinkedList<E> implements AbstractList<E>
```

Java provides a shortcut so that we don't have to implement ALL the List methods.

Abstract List

- public class MySingleLinkedList<E> implements List<E>
- public class MySingleLinkedList<E> extends AbstractList<E>
- AbstractList provides implementations for most methods in List interface.
- We can override its methods with our own.

UML Model (Unified Modeling Language)

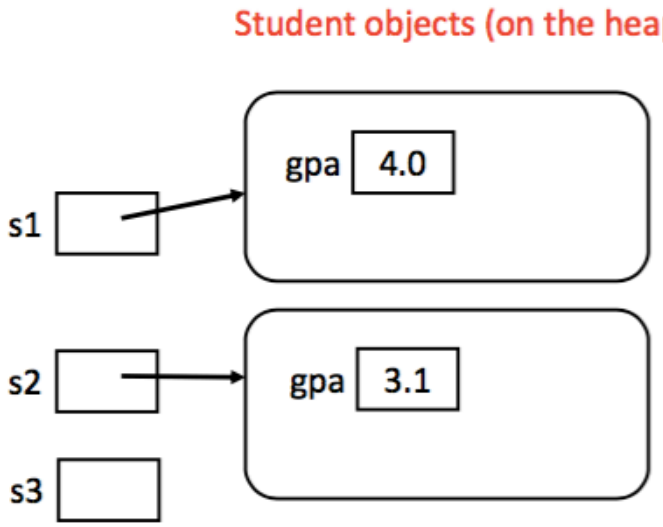


An AbstractList "is a" List
A MySingleLinkedList "is a" AbstractList
A MySingleLinkedLIst "is a" List

MEMORY MODEL

Aside: Memory Models Review

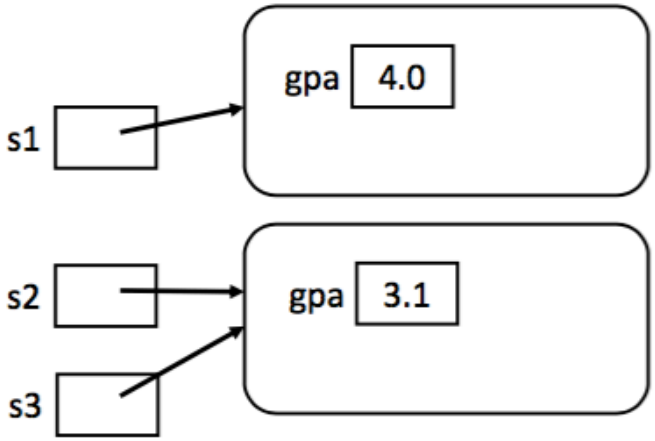
```
// Node is a class inside the class MySingleLinkedList (i.e. an inner class).
public class Student {
   public double gpa;

   public Student(double theGPA)
   {
      gpa = theGPA;
   }
}

// Somewhere else in the code…
Student s1 = new Student(4.0);
Student s2 = new Student(3.1);
Student s3 = s2;
```



Aside: Memory Models Review

```
// Node is a class inside the class MySingleLinkedList (i.e. an inner class).
public class Student {
   public double gpa;

   public Student(double theGPA)
   {
      gpa = theGPA;
   }
}

// Somewhere else in the code…
Student s1 = new Student(4.0);
Student s2 = new Student(3.1);
Student s3 = s2;
s3.gpa = 2.0;
```

Student objects (on the heap)



Today's Lecture

ArrayLists vs LinkedLists

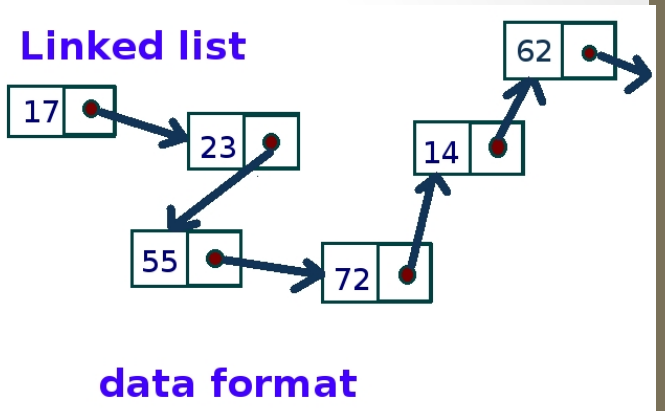
- **Problems with ArrayLists:**

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

Linked Lists solve these problems.

but introduce other problems ☺

Nodes and Lists



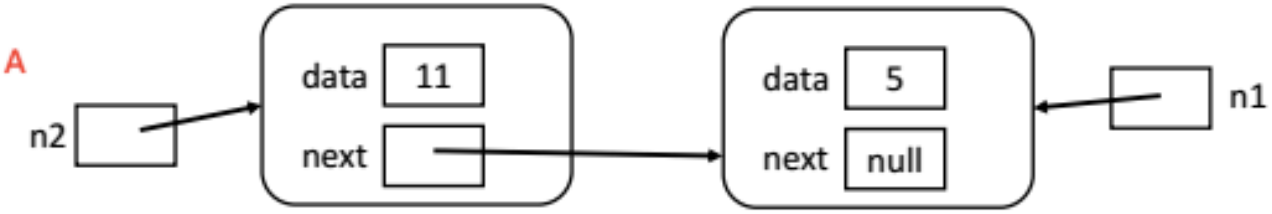
- A different way of implementing a List interface
- Each element of a Linked List is a separate Node object.
- Each Node tracks a single piece of data **plus** a reference (pointer) to the next node.
- Create a new Node every time we add something to the List
- Remove nodes when item is removed from list and allow garbage collector to reclaim that memory

Memory Model Diagrams and LinkedLists

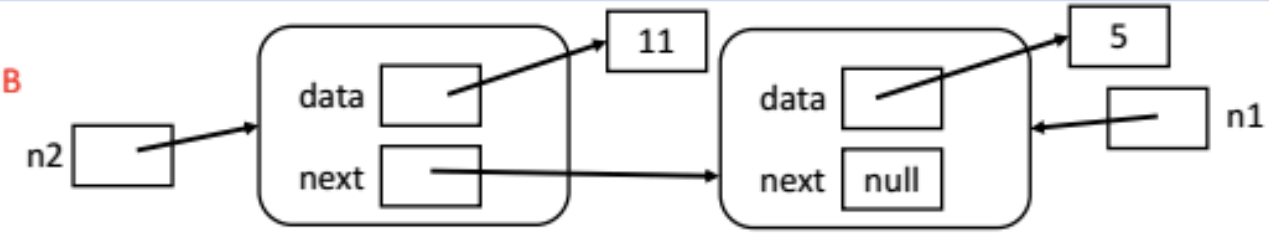
```
public class Node<E> {
    E data;
    Node next;

    /** Constructor to create singleton Node with next set */
    public Node(E element)
    {
        data = element;
        next = null;
    }
}

// Somewhere else in the code… still inside MySingleLinkedList
Node<Integer> n1 = new Node<Integer>(new Integer(5));
Node<Integer> n2 = new Node<Integer>(new Integer(11));
n2.next = n1;
```

Draw the memory model diagram for this code. Answer choices next slide.

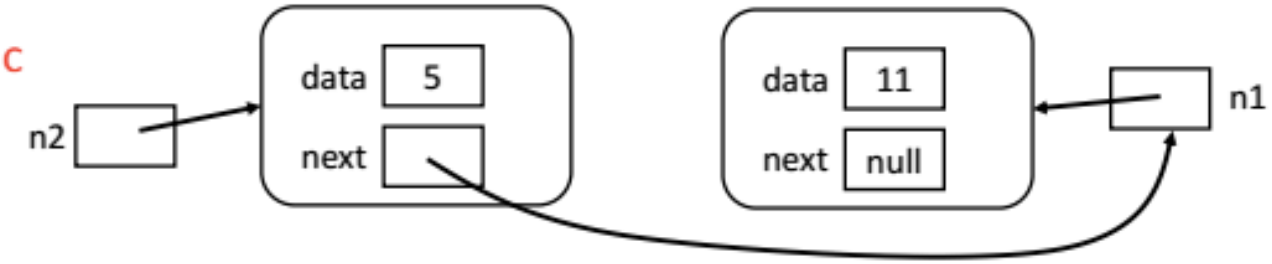
A



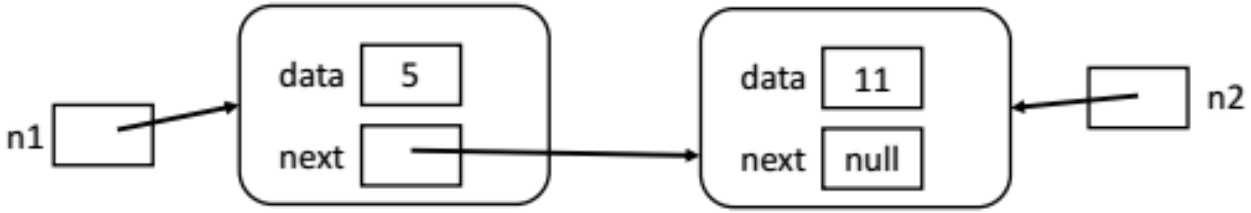
B



C



D



E. None of these

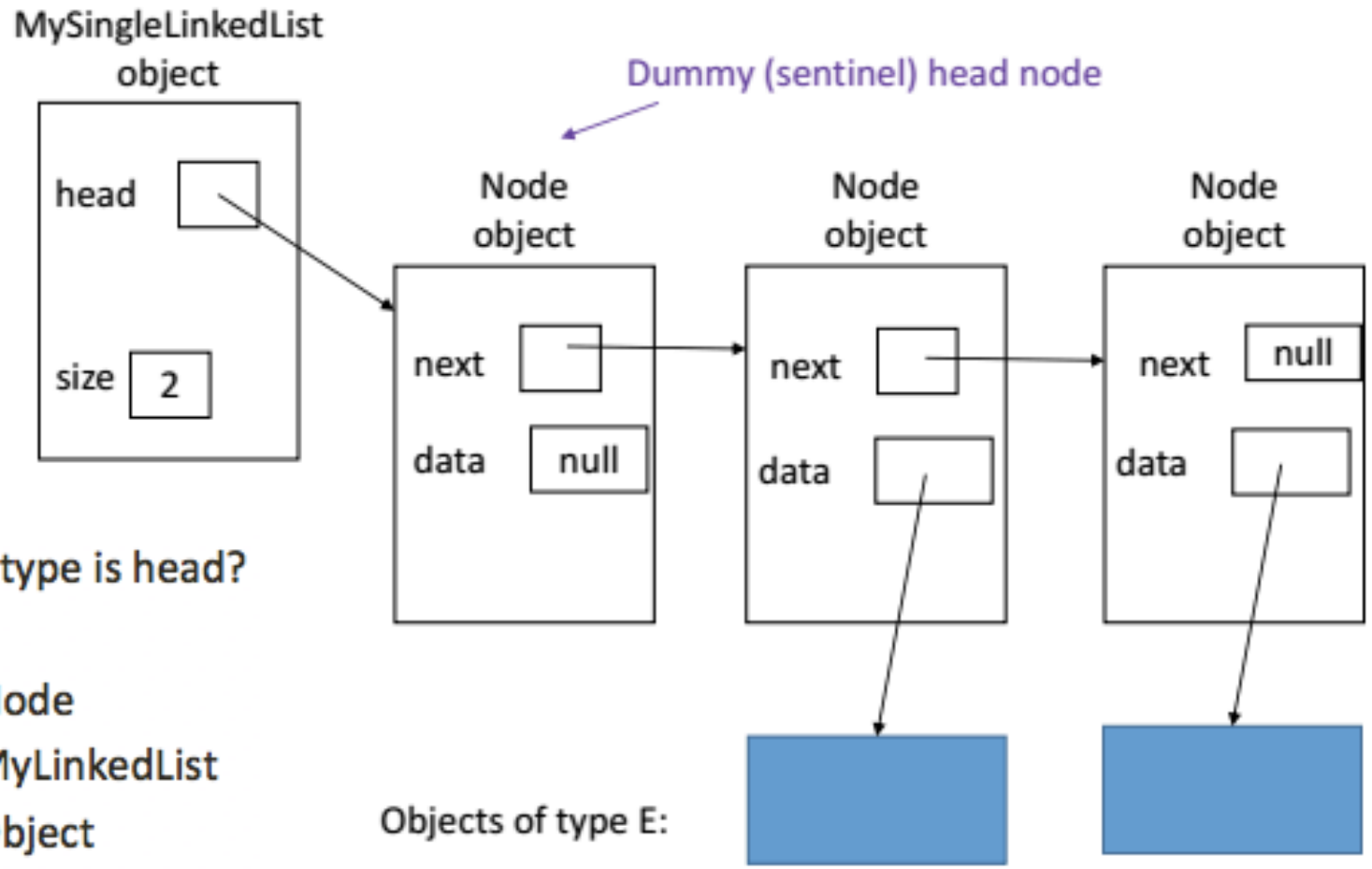
Class Node

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

- The (typical) Node contains:
  - A reference to the next node in the list
  - A reference to the data stored at that position in the list
  - For DoublyLinked List (HW2) 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).
- Sometimes it also stores a reference to the last node (tail, last).

Dummy node: Picture



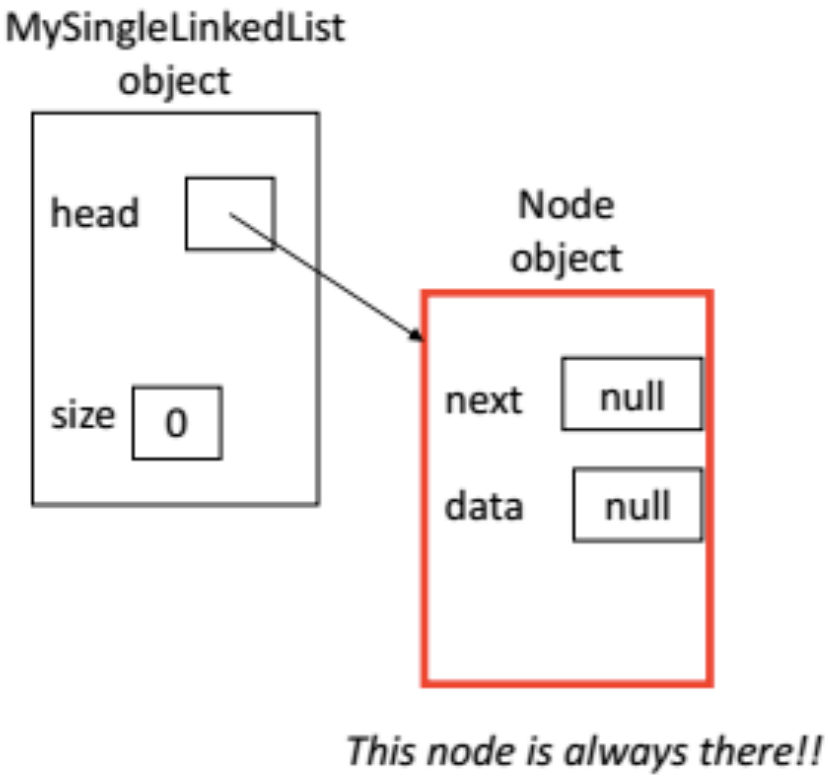
What type is head?

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

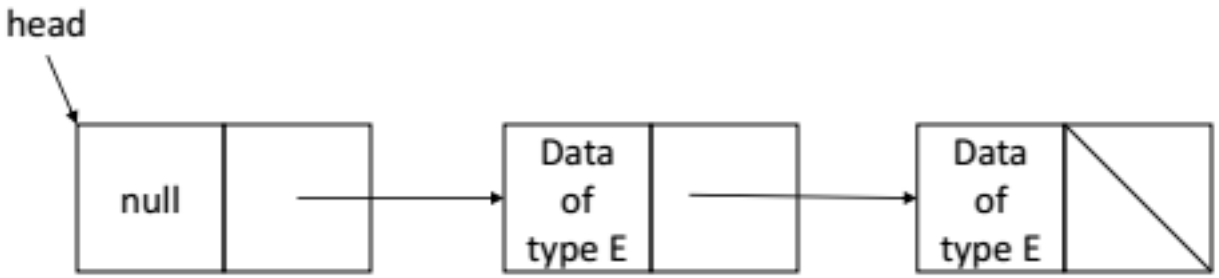
Lists with sentinel (dummy) node

- *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.

Empty list with sentinel node



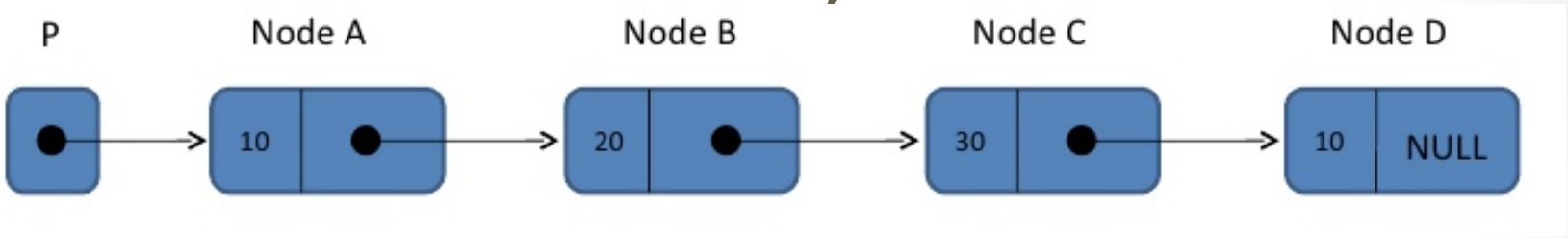
Quick check



- Does this list have a dummy (sentinel) node?

- A: Yes
- B: No.
- C: What is a dummy node?

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 = NodeE;

Single Linked List Node: Code

```
class Node<E>
{
  E data;
  Node next;

  public Node() {
   data = null;
   next = null;
  }
  public Node(E theData, Node newNodePred) {
   data = theData;
   next = newNodePred.next;
   newNodePred.next = this;
  }
}
```

```
public static void main()
{
  Node<Integer> n0 =
    new Node<Integer>();
  Node<Integer> n1=
    new Node( new Ingeter(1),n0);
}
```

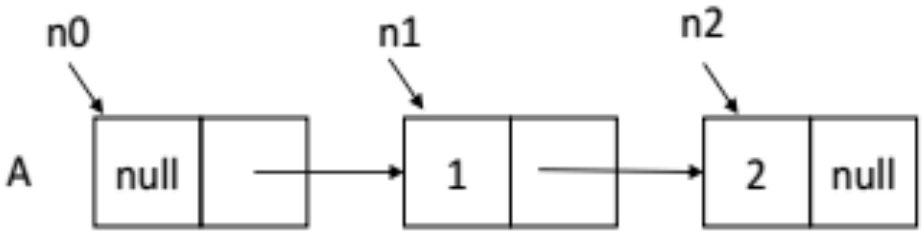
Single Linked List Node: Code

```
class Node<E>
{
  E data;
  Node next;

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

  public Node(E theData, Node newNodePred) {
   data = theData;
   next = newNodePred.next;
   newNodePred.next = this;
  }
}
```

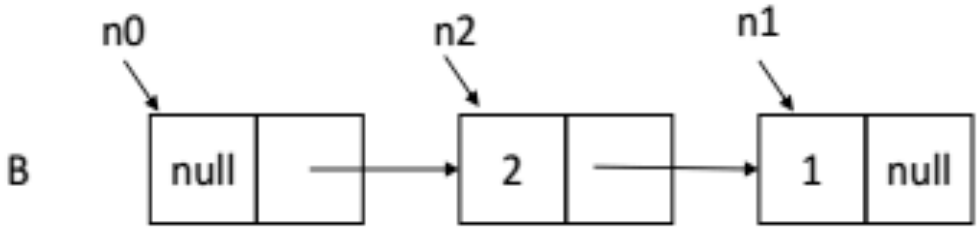
```
public static void main()
{
  Node<Integer> n0 =
    new Node<Integer>();
  Node<Integer> n1=
    new Node( new Ingeter(1),n0);
  Node<Integer> n2 =
    new Node( new Integer(2), n0);
}
```

What does the list of nodes look like after main runs? (choices next slide)

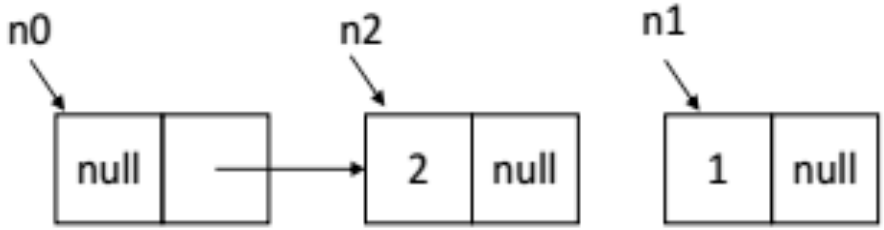
A


n0
n1
n2
null
1
2
null

B


n0
n2
n1
null
2
1
null

C

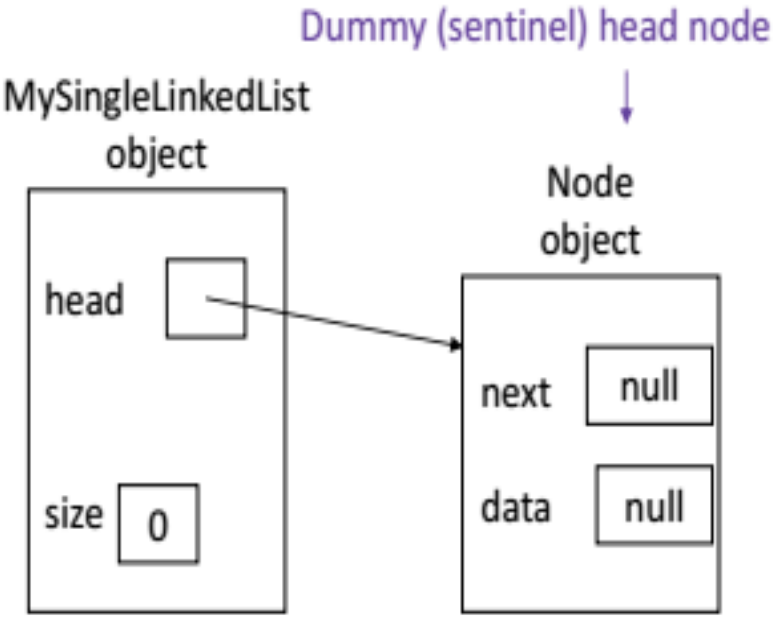

n0
n2
n1
null
2
null
1
null

D Other

Single Linked List: Code

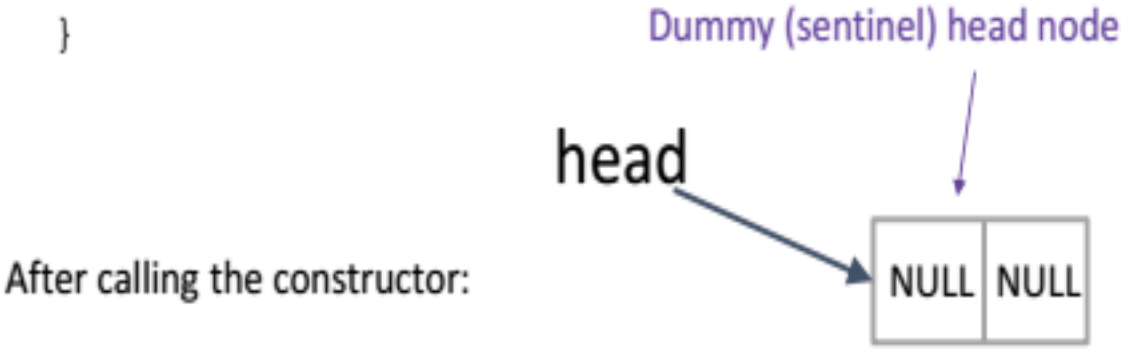
```
class MySingleLinkedList<E> extends AbstractList
{
  Node<E> head;
  int size;

  public MySingleLinkedList<E>() {
   head = new Node<E>();
   size = 0;
  }
  //… more here
}
```

After calling the constructor:



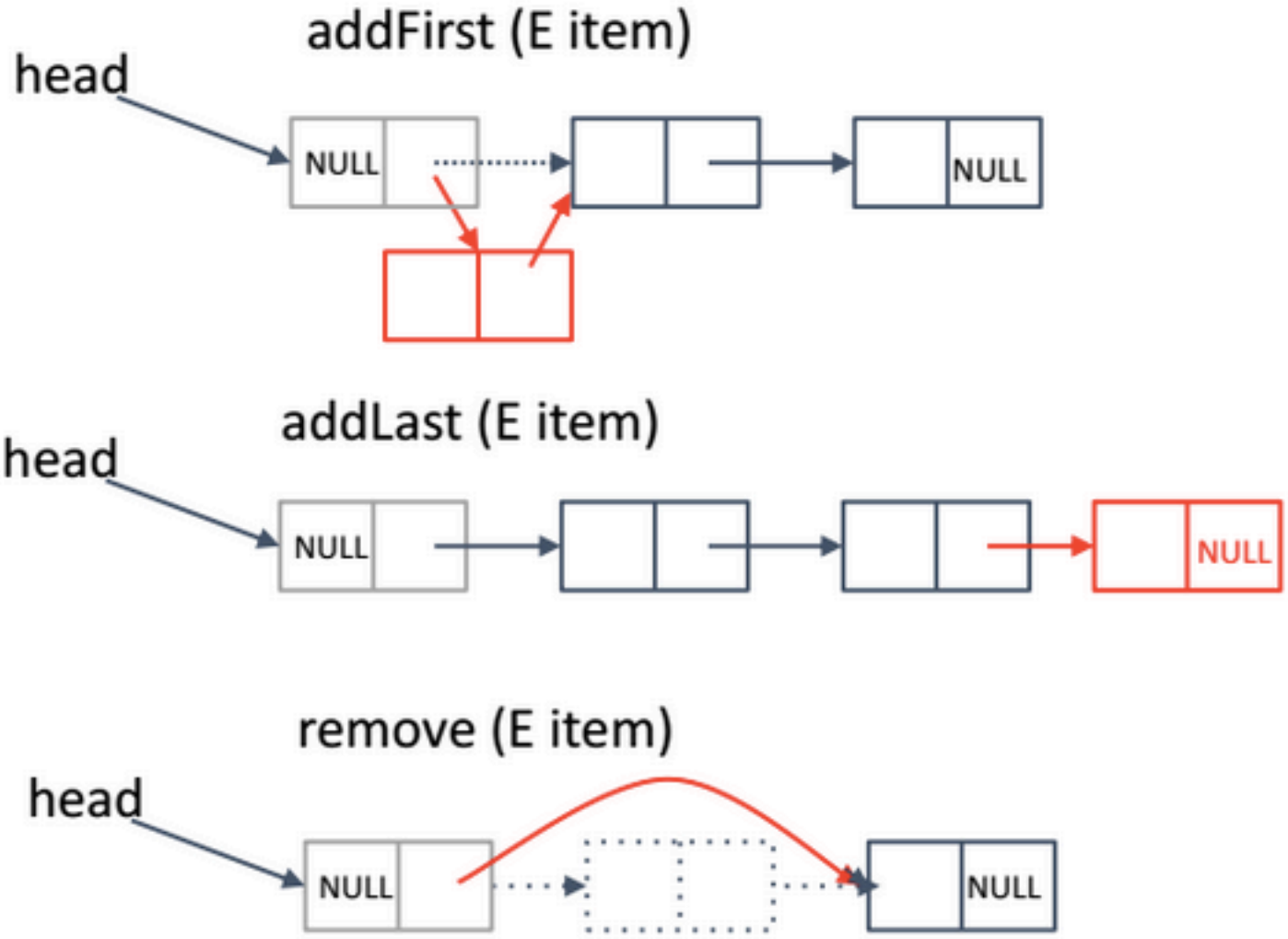
Single Linked List: Code

```
class MySingleLinkedList<E> extends AbstractList
{
  Node<E> head;
  int size;

  public MySingleLinkedList<E>() {
   head = new Node<E>();
   size = 0;
  }
  //… more here
}
```

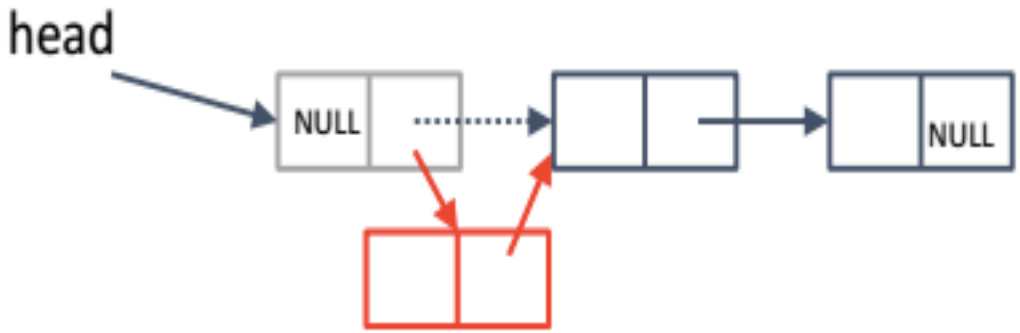


After calling the constructor:

A few methods



addFirst (E item)




```
// In MySingleLinkedList<E> class (NOT Node class)
public void addFirst (E newItem) {
      Node<E> newNode = new Node<E>(newItem, head);

      _____________________________

      size++;
}
```

What line of code will complete this method correctly (in the blank)?

A) No line is needed. The code is correct as written.

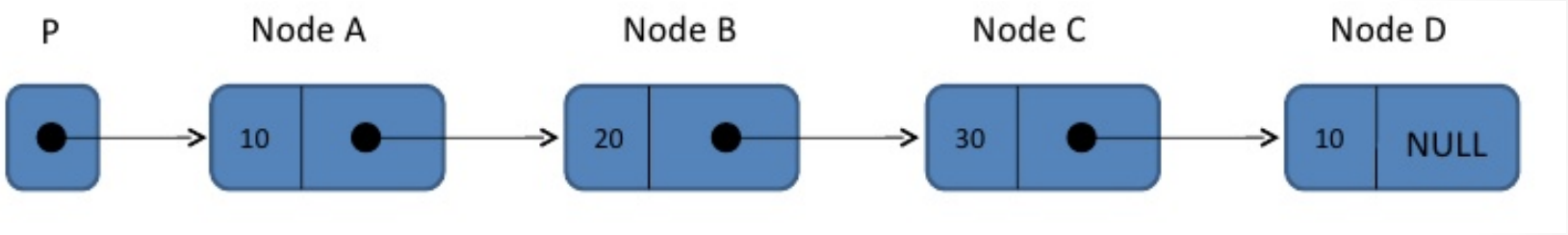
B) `head = head.next;`

C) `head = newNode;`

D) `newNode.next = head;`

```
public Node(E theData,
            Node before) {
   data = theData;
   next = before.next;
   before.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

What replaces the XXXXXX?


```
public void addLast (E newItem) {
  Node<E> current = head;
  while (XXXXXX){ current = current.next; }
  new Node<E>(newItem, current);
  size++;
}
```

A) current == head

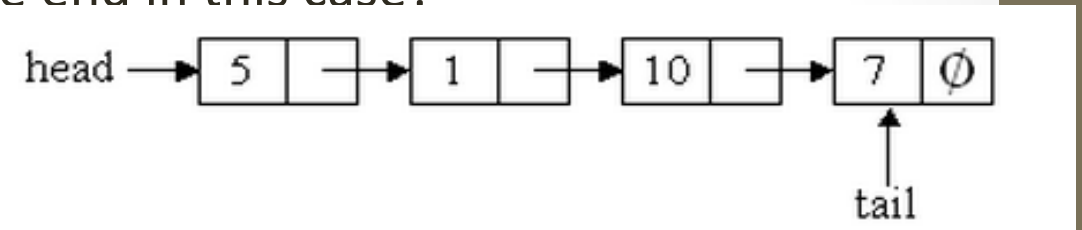
B) current != null

C) current.next != null

D) head != null

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;

Another implementation


```
public void addLast (E newItem) {
    Node<E> current = head;
    int currIndex = 0;
    while (currIndex < size){
        current = current.next;
        currIndex++; }
    new Node<E>(newItem, current);
    size++;
}
```

Change to insert at an arbitrary location?

Another implementation of addLast

```
e.g.
myL.atAtIndex( Integer(5), 1 );
```

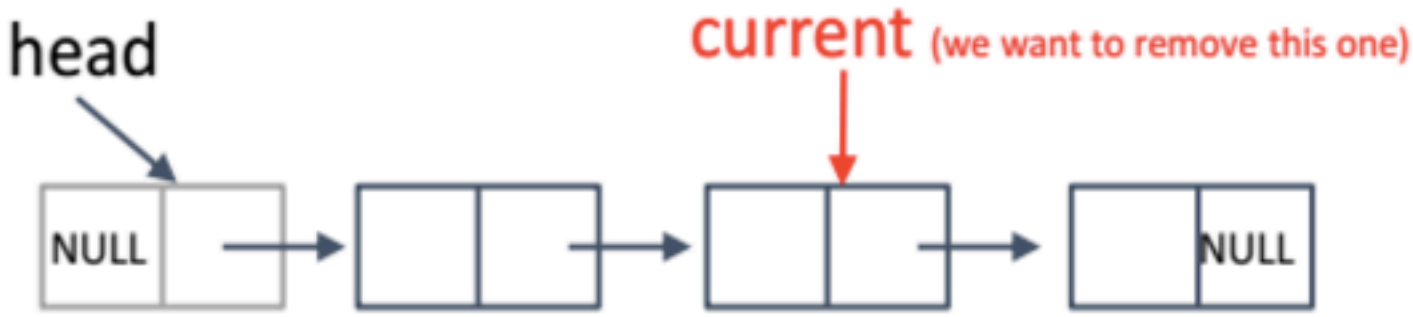

```
public void addAtIndex (E newItem, int index) {
    Node<E> current = head;
    int currIndex = 0;
    while (currIndex < index){
        current = current.next;
        currIndex++; }
    new Node<E>();




    size++;
}
```

Fill in the code to correctly insert the node,
But this time using the default constructor.

Removal from LL

```
// In MySingleLinkedList<E> class
public E remove (int position){
  // Removes the element at index position from the
  // list and returns the element.
  ...
}
```

Suppose we have a reference (current) to the node containing the item to be removed.



What additional information do we need to successfully remove the node?

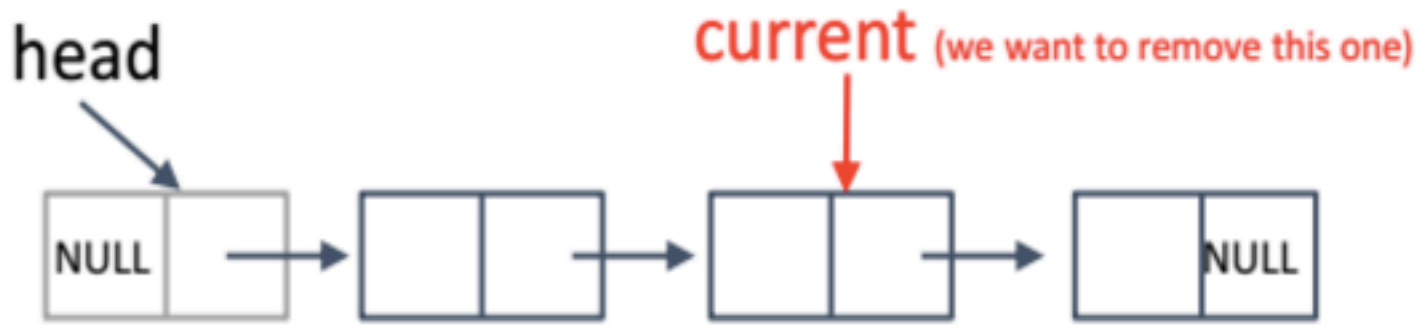
A) Nothing additional.

B) A reference to the node immediately prior to the deleted node.

C) A reference to the node immediately after the node to be deleted.

D) Both B and C.

Suppose we have a reference (current) to the node containing the item to be removed.



What additional information do we need to successfully remove the node?

A)

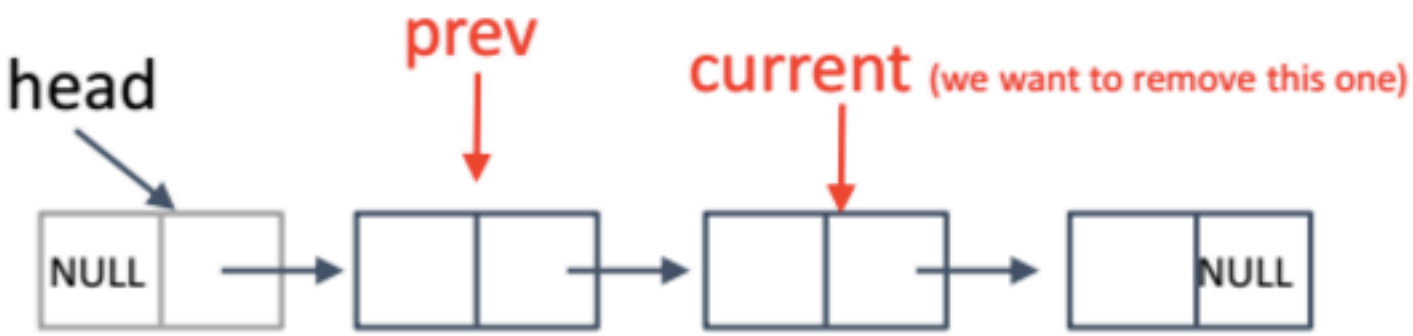
B)

C)

D)

This is different for your Double linked list implementation (where you already have a reference to the node before, in the prev pointer.

Suppose we have a reference (current) to the node containing the item to be removed.



What line of code successfully removes current?

A) current = current.next;

B) prev = current.next;

C) prev.next = current;

D) prev.next = current.next;

E) None of these

HW2: Doubly linked lists