Lecture 9-11

Linked Lists

Singly Linked Lists

Motivation

- Suppose I have an **array** with 5 elements: 1, 3, 5, 7, 9

```
[1, 3, 5, 7, 9]
```

- I want to insert -1 before 1. What should I do?
    - Talk to each other please

Motivation

- Suppose I have an array with 5 elements: 1, 3, 5, 7, 9

```
[1, 3, 5, 7, 9]
```

- I want to insert -1 before 1. What should I do?
    - Create a new bigger array
    - Shift *1*, *3*, *5*, *7*, *9* to the right
    - Insert -1 before 1.
    - Change reference from old array to a new one.

Motivation

```
int arr [] = {1, 3, 5, 7, 9};
```

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert -1 before 1.
  - Change reference from old array to a new one.

Motivation

```
int arr [] = {1, 3, 5, 7, 9};

int [ ] arrBigger = new int[6];
```

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1, 3, 5, 7 , 9 to the right
  - Insert -1 before 1.
  - Change reference from old array to a new one.

Motivation

- I want to insert -1 before 1.
  - Create a new bigger array
  - Shift 1,   3,   5,   7 ,   9 to the right
  - Insert -1 before 1.
  - Change reference from old array to a new one.

```
int arr [] = {1, 3, 5, 7, 9};

int [ ] arrBigger = new int[6];

for (int i = 1; i<6; i++){
        arrBigger[i] = arr[i-1];
}
```

Motivation

- I want to insert -1 before 1.
    - Create a new bigger array
    - Shift 1, 3, 5, 7 , 9 to the right
    - Insert -1 before 1.
    - Change reference from old array to a new one.

```
int arr [] = {1, 3, 5, 7, 9};

int [ ] arrBigger = new int[6];

for (int i = 1; i<6; i++){
     arrBigger[i] = arr[i-1];
}

arrBigger[0] = -1;
arr = arrBigger;
```

What is the complexity of this algorithm?

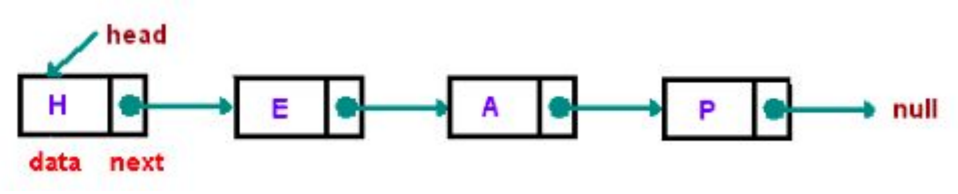
A: O (1)
B: O (log n)
C: O (n)
D: O (n^2)
E: None of the above

Disadvantage of using arrays

- arrays are static structures
    - cannot be easily extended or reduced to fit the data set
- Once you created an array, it can’t be changed anymore.
- You have to create a new one each time
- Arrays are also expensive to maintain new insertions and deletions

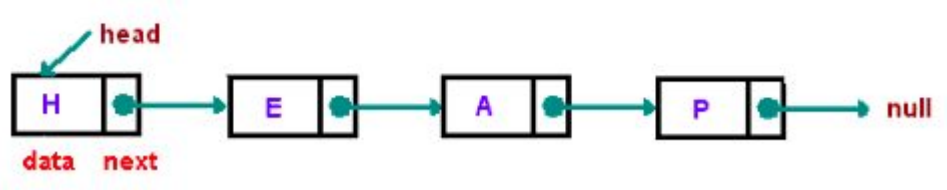
Linked Lists

- A linked list is a *linear* data structure where each element is a separate object.
- A linked list is a **dynamic** data structure.
  - The number of nodes in a list is not fixed and can grow and shrink on demand.
- Any application which has to deal with an unknown number of objects will need to use a linked list.



Disadvantage of Linked Lists

- One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements:



- If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

- Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a *reference* to the next node.

Let’s think what operations we want first

We will be creating an *interface*.

Let’s think what operations we want first

- Append element to the list (end)
- Get an element at index
- Find an element
- Insert at index
- Delete at index
- Prepend
- Size
- Sort
- Empty the list

```
interface List {

    void append (int elem);

    int get (int index);

    void insert(int index);

    int indexOf(int elem);

}
```

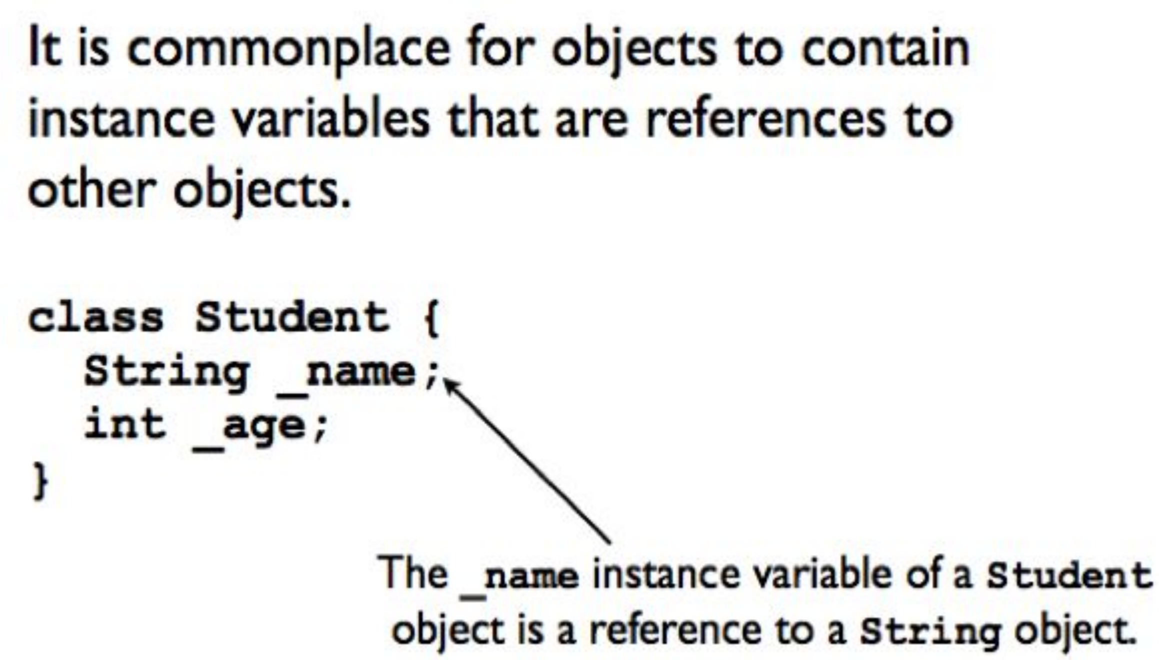
Complete `List interface`: https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Implementation

```
interface List {

    void append (int elem);

    int get (int index);

    void insert(int index);

    int indexOf(int elem);

}
```

```
class MyList implements List {










}
```

References inside objects

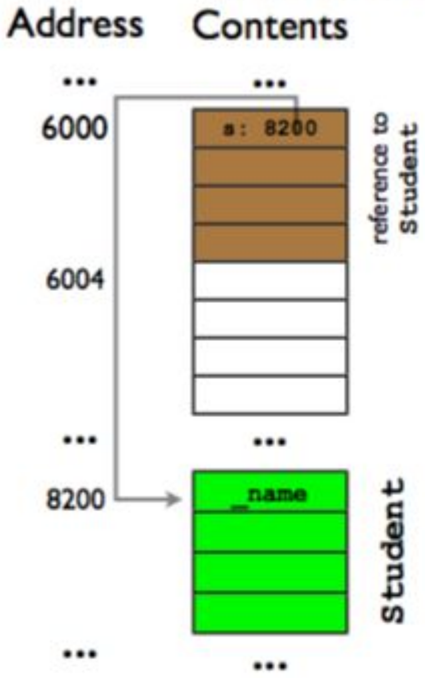


It is commonplace for objects to contain instance variables that are references to other objects.

```
class Student {
  String _name;
  int _age;
}
```

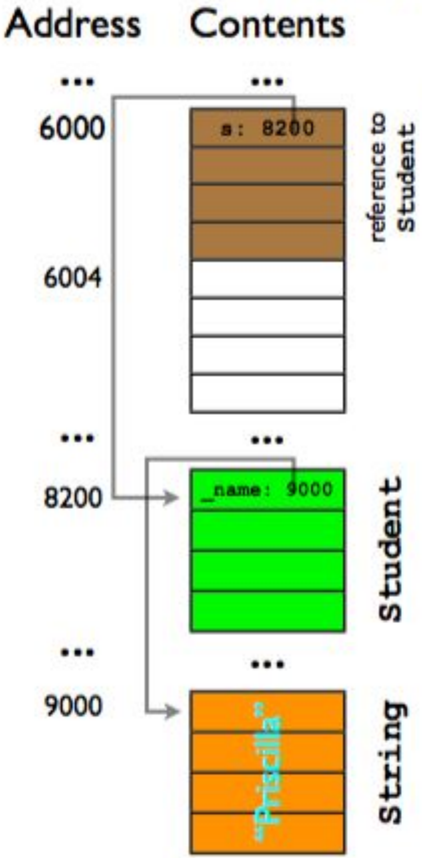
The _name instance variable of a Student object is a reference to a String object.

References inside objects



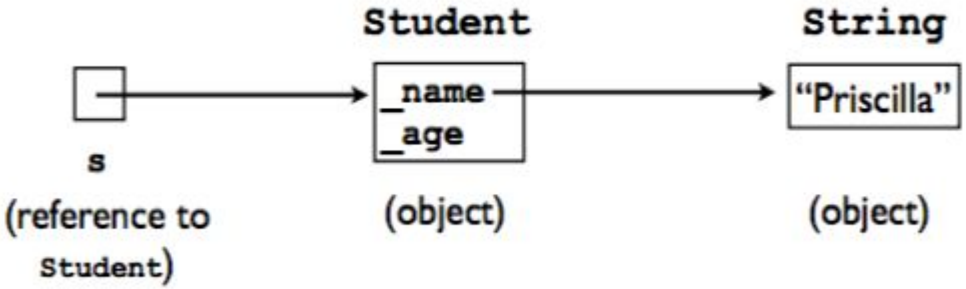
```
Student s = new Student();
```

References inside objects

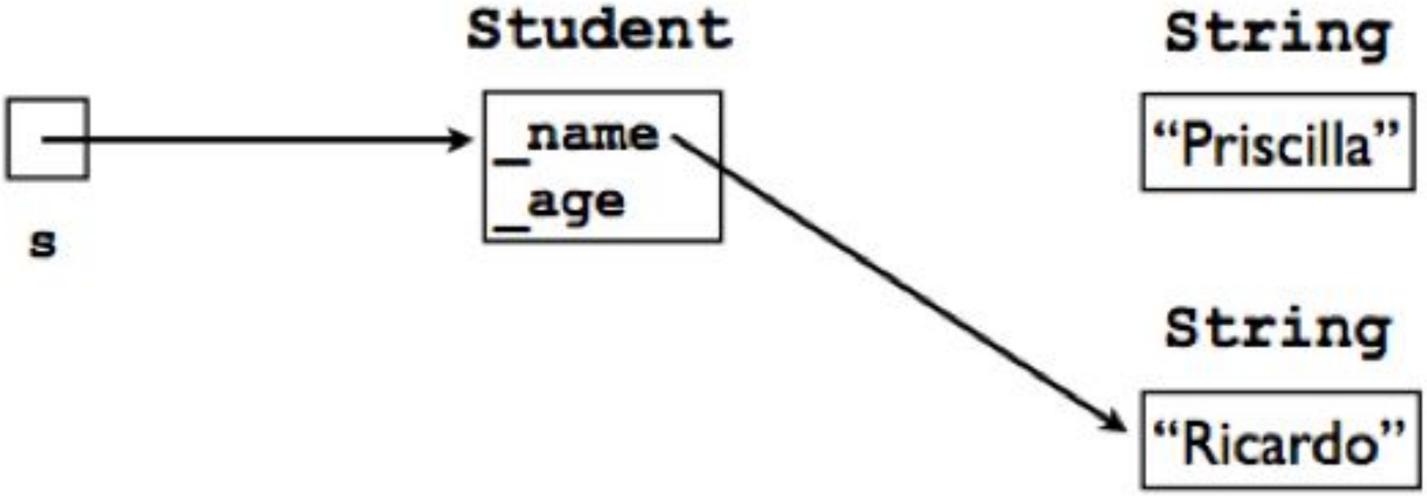


```
Student s = new Student();
s._name = "Priscilla";
```

Simplified figure for `class Student`



Inside the boxes: Sometimes I will write the *names* of instance variables and sometimes their *values*; it should be clear from the context.



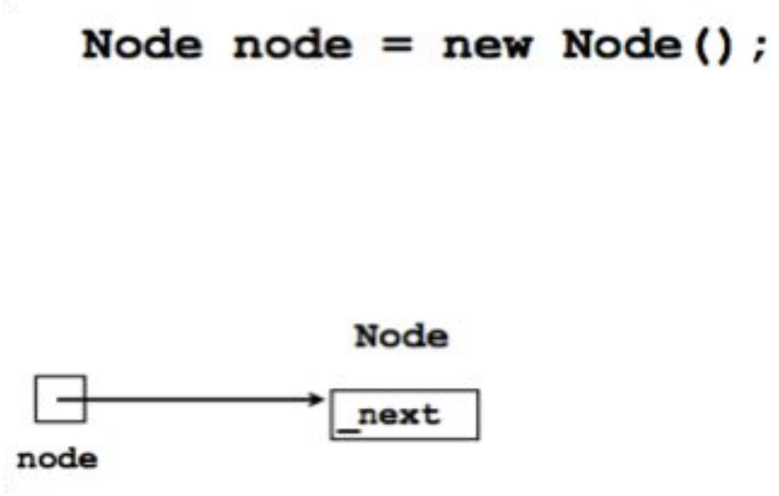
```
s._name = "Ricardo";
```

Here’s where things get fun...

It is also (sometimes) useful for an object to contain a reference to another object of the same class.

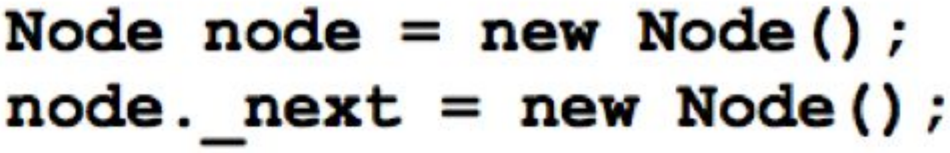
In this way, we can “chain” together multiple objects.

```
class Node {
   Node _next;
 }
```

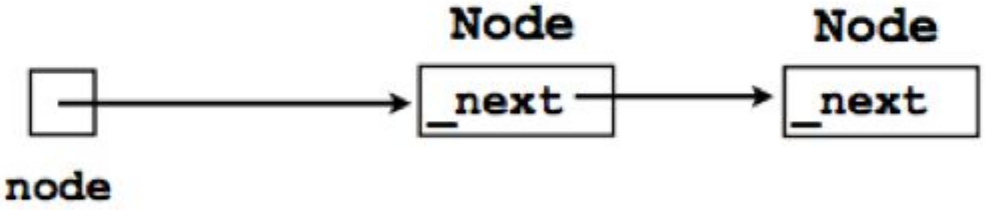


```
Node node = new Node();
```

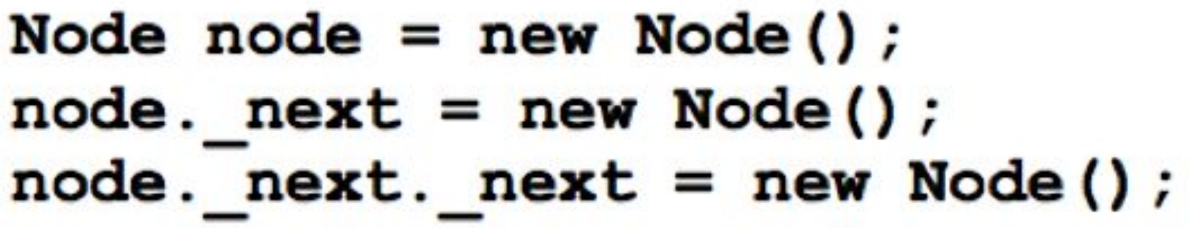
Chain of Nodes



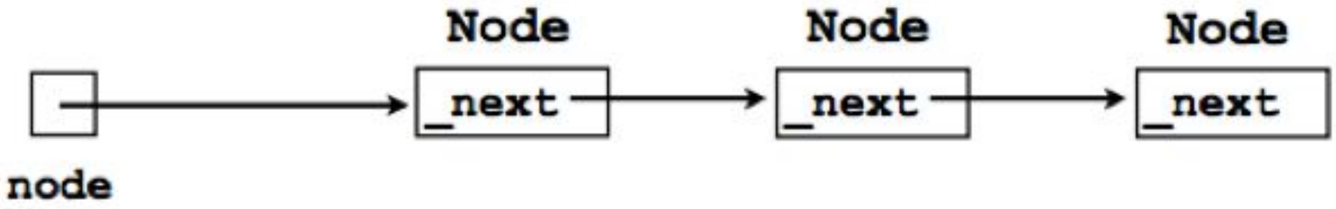
```
Node node = new Node();
node._next = new Node();
```



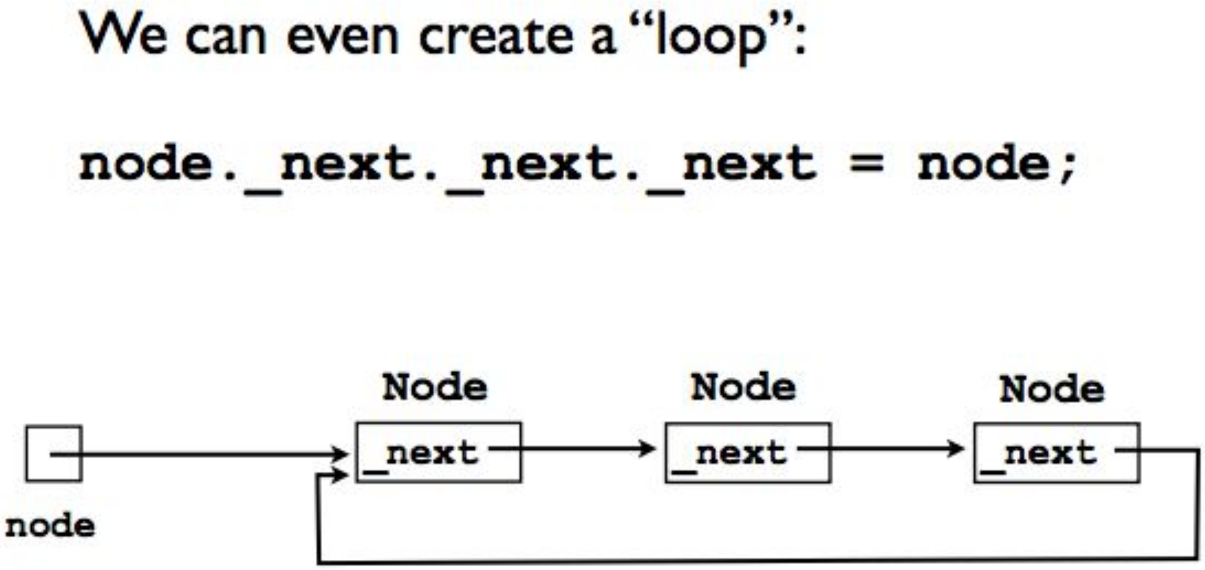
Chain of Nodes



```
Node node = new Node();
node._next = new Node();
node._next._next = new Node();
```



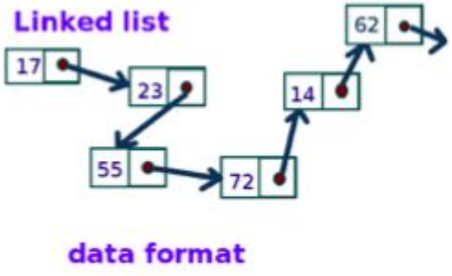
Loop of nodes



We can even create a "loop":

```
node._next._next._next = node;
```

Nodes and Lists



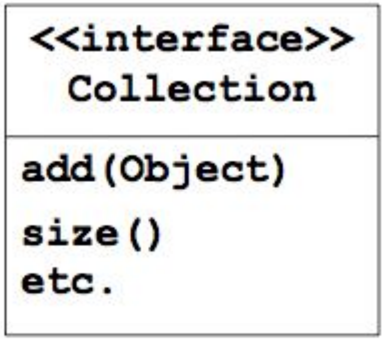
- A *different* way of implementing a List interface
  - There is another class called ArrayList that implements the same interface using arrays.
  - https://docs.oracle.com/javase/8/docs/api/java/util/List.html

- 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

Collections

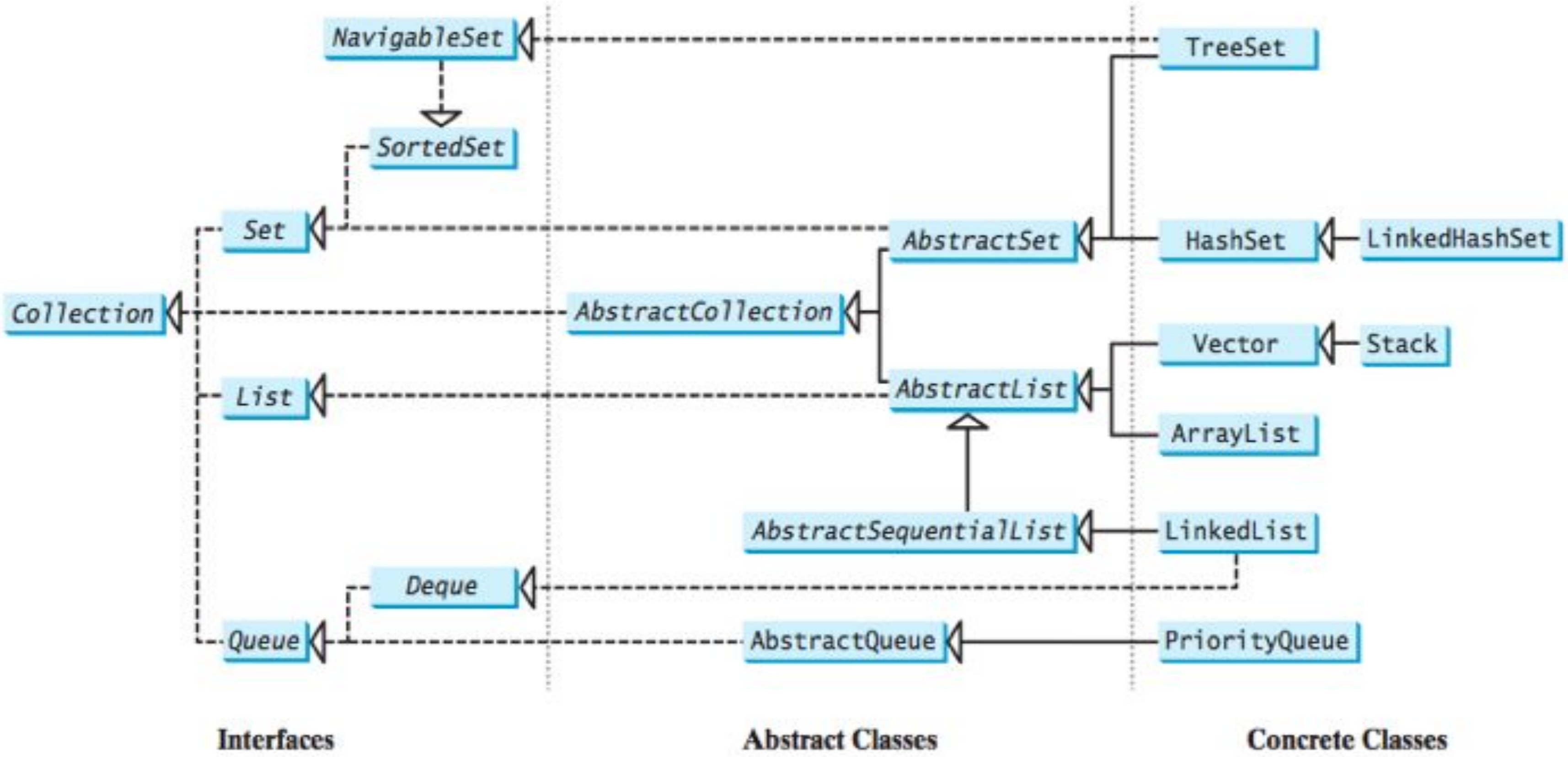
- Fundamentally, what we as programmers do with data is to store it and retrieve it and then operate on it.
- A **collection** is an ADT (Abstract Data Type) that contains data elements, and provides operations on them.
- There are different ways that elements can be collected:
  - Set, List, Sorted List…

All collections implement the interface `Collection`



```
<<interface>>
 Collection
-------------
add(Object)
size()
etc.
```

A collection is a **container** that stores objects

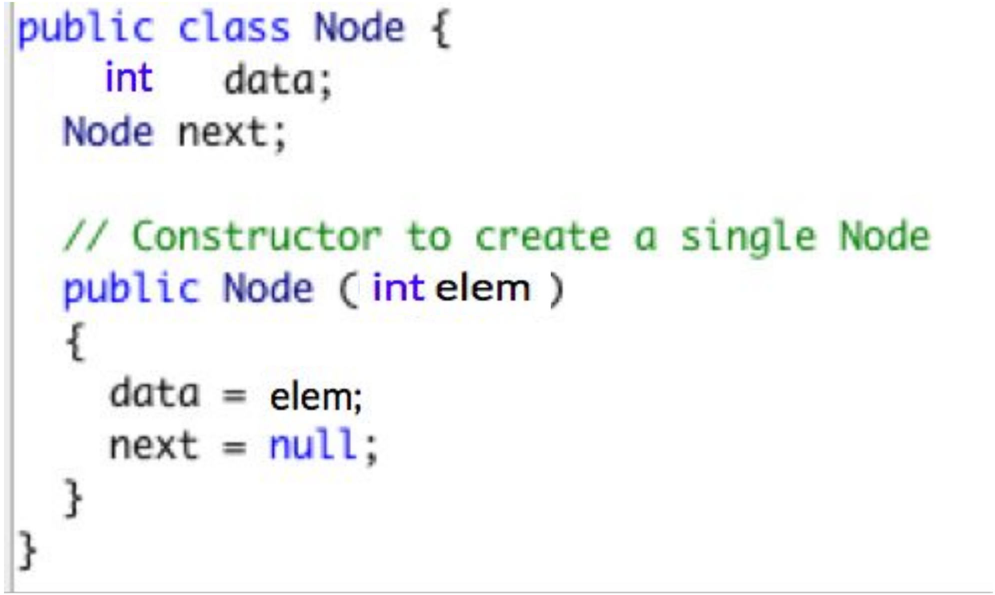


Abstract List

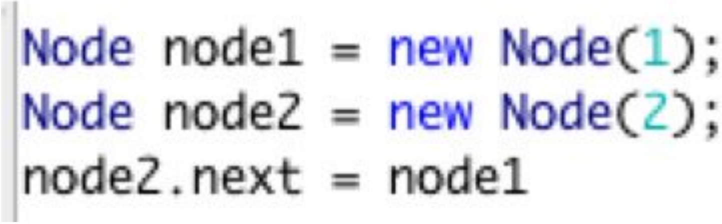
- public class DoublyLinkedList<E> implements List<E> <--- **ideal**
- public class DoublyLinkedList<E> extends `AbstractList<E>`
- `AbstractList` provides *dummy* implementations for most methods in `List` interface.
- We can override its methods with our own!!

https://docs.oracle.com/javase/9/docs/api/java/util/AbstractList.html

Draw a memory model



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

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



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

Single Linked List Node: Code

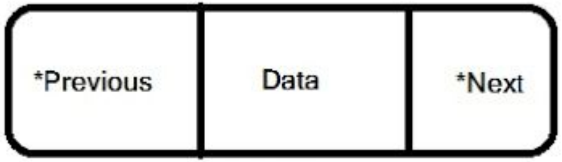
```
class Node<E>
{
  E data;
  Node next;

  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);
}
```

Class Node: PA4

- Node class is a part of MyList implementation

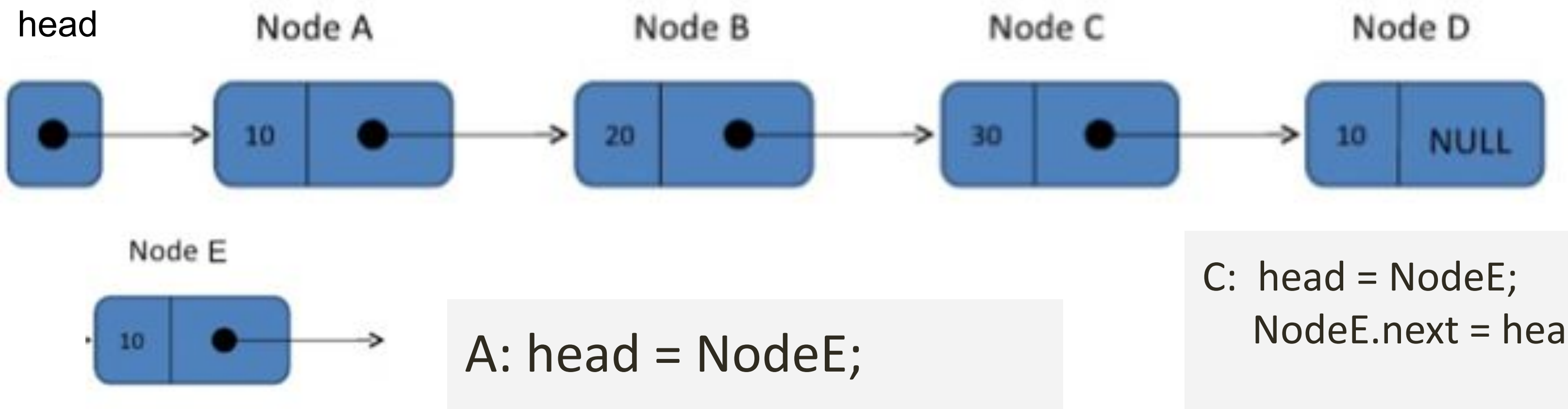
- The (typical) Node contains:
    - A reference to the ***next*** node in the list
    - A reference to the data stored at that position in the list
    - For Doubly Linked List a reference to the **previous** node



Node

- The Linked List itself contains a reference to the FIRST node in the list (*head*).
- Sometimes it might store some info about the list (like list size).
- Sometimes it also stores a reference to the last node (*tail*).

AddFront (beginning of linked list)



A: head = NodeE;

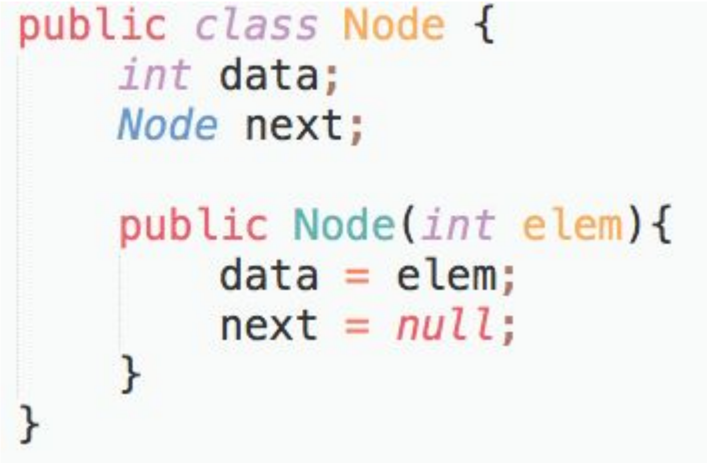
B: NodeE.next = Node A;

C: head = NodeE;
NodeE.next = head;

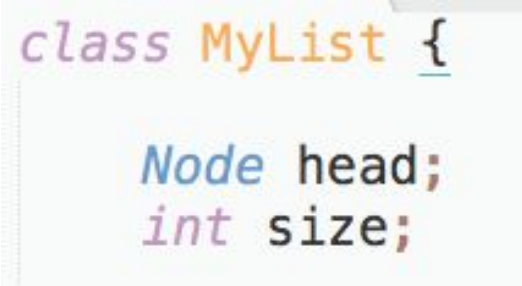
D: NodeE.next = head;

E: NodeE.next = head;
head = NodeE;

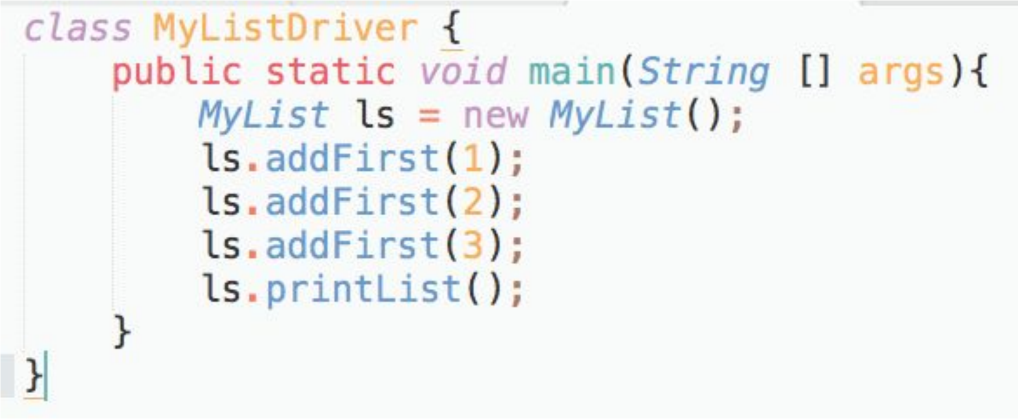
Let’s put it all together



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

    public Node(int elem){
        data = elem;
        next = null;
    }
}
```

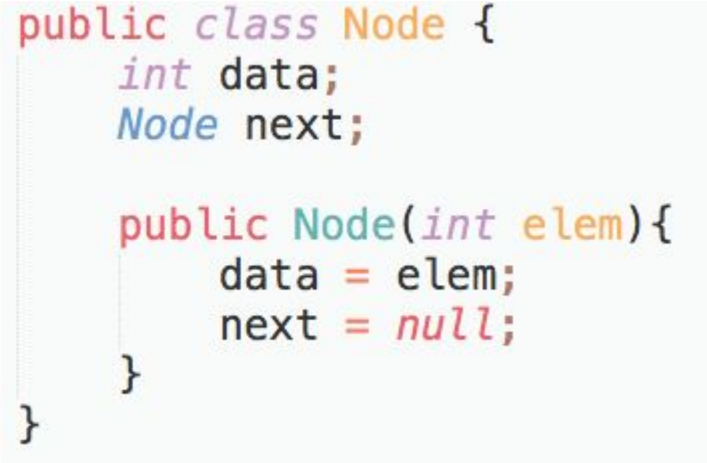


```
class MyList {

    Node head;
    int size;
```

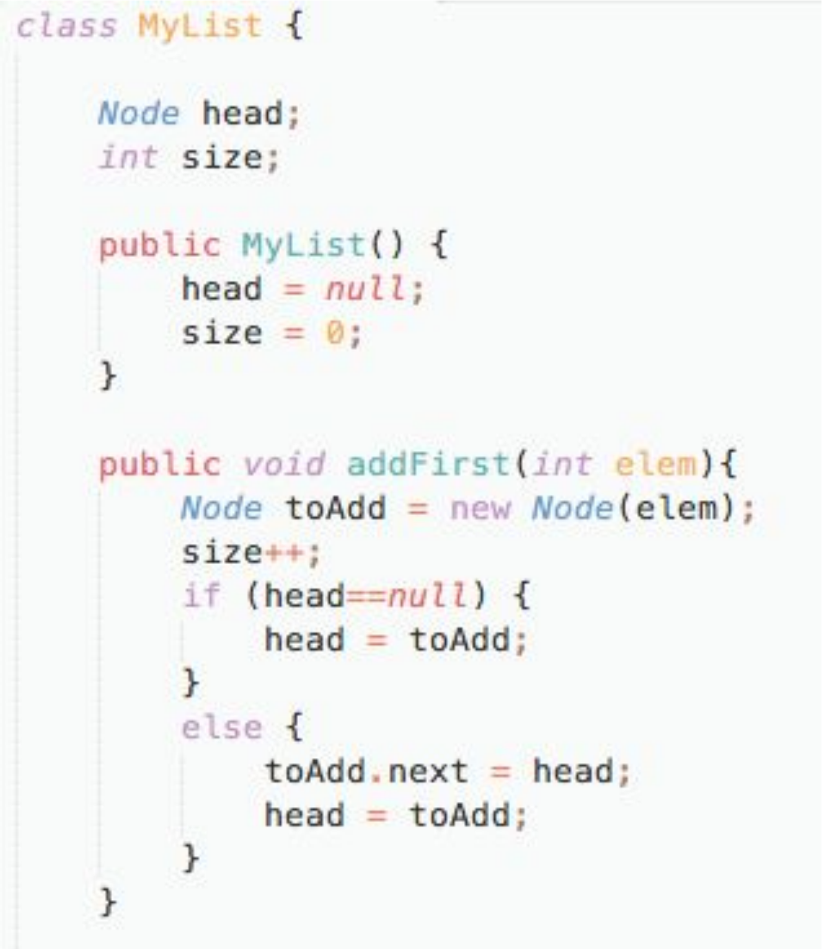


```
class MyListDriver {
    public static void main(String [] args){
        MyList ls = new MyList();
        ls.addFirst(1);
        ls.addFirst(2);
        ls.addFirst(3);
        ls.printList();
    }
}
```

Let’s put it all together

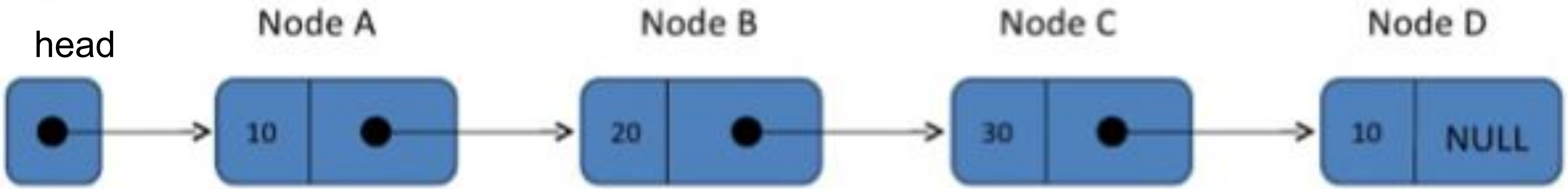


```
public class Node {
    int data;
    Node next;

    public Node(int elem){
        data = elem;
        next = null;
    }
}
```



```
class MyList {

    Node head;
    int size;

    public MyList() {
        head = null;
        size = 0;
    }

    public void addFirst(int elem){
        Node toAdd = new Node(elem);
        size++;
        if (head==null) {
            head = toAdd;
        }
        else {
            toAdd.next = head;
            head = toAdd;
        }
    }
```

Add to the Back



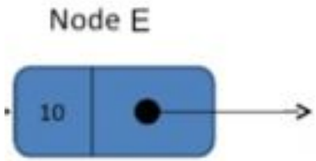
A: NodeD.next = NodeE;

B: need to loop through the list to get to nodeD.

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;

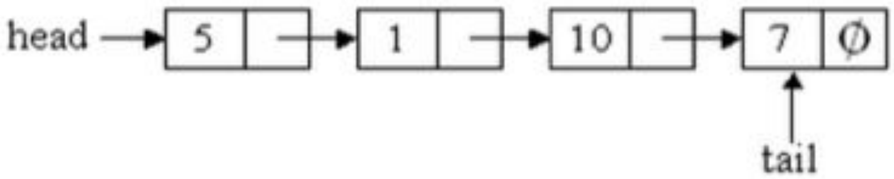
B: tail.next = Node E;

C: tail = Node E;

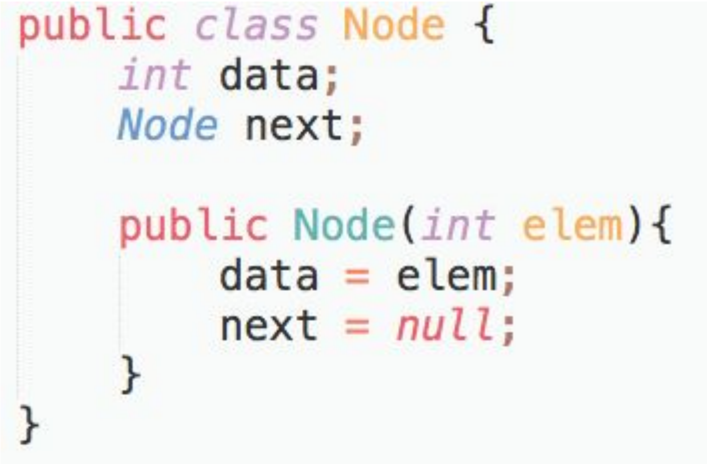
    tail.next = Node E;

D: tail.next = Node E;

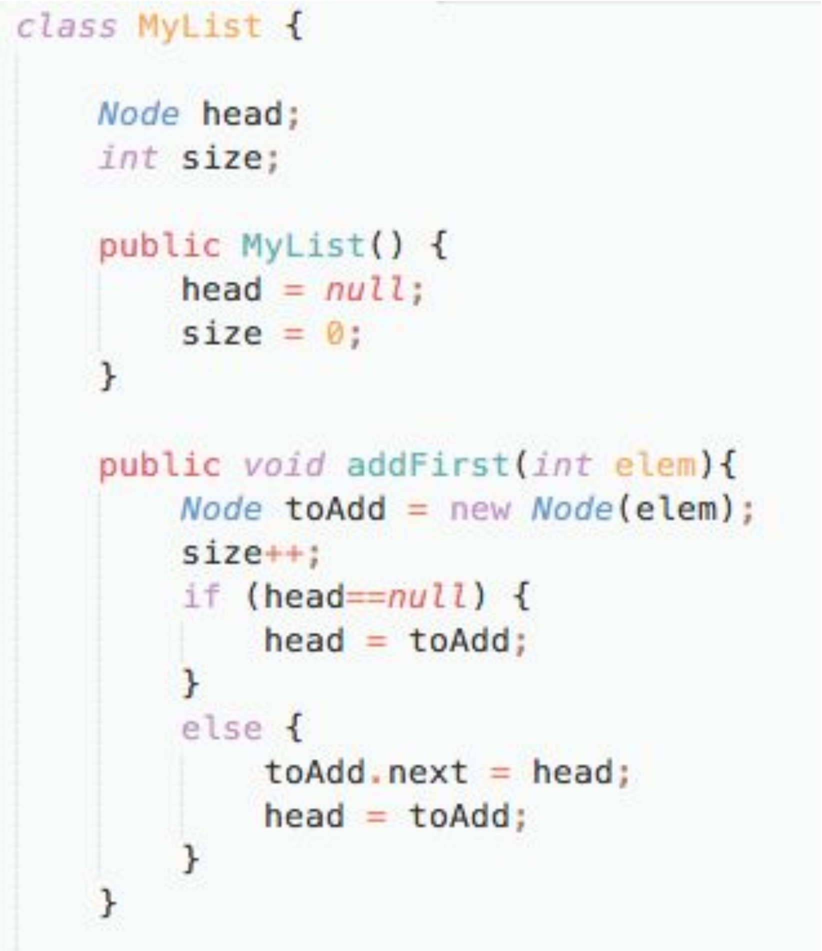
    tail = Node E;



What needs to be modified?



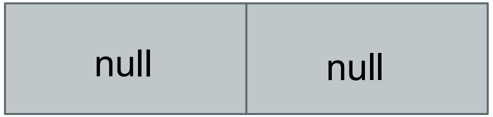
```
public class Node {
    int data;
    Node next;

    public Node(int elem){
        data = elem;
        next = null;
    }
}
```



```
class MyList {

    Node head;
    int size;

    public MyList() {
        head = null;
        size = 0;
    }

    public void addFirst(int elem){
        Node toAdd = new Node(elem);
        size++;
        if (head==null) {
            head = toAdd;
        }
        else {
            toAdd.next = head;
            head = toAdd;
        }
    }
```

Lists with sentinel (dummy) node

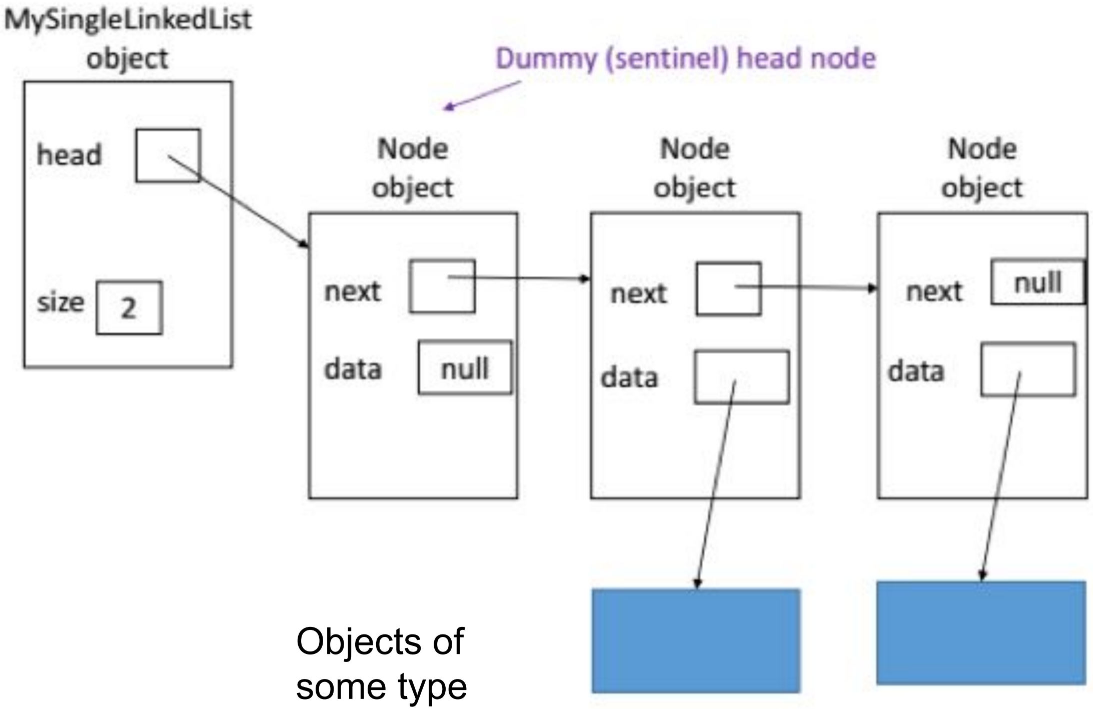
- Dummy nodes are Nodes whose data fields are always ***null*** – they contain **no** data from the “user”.



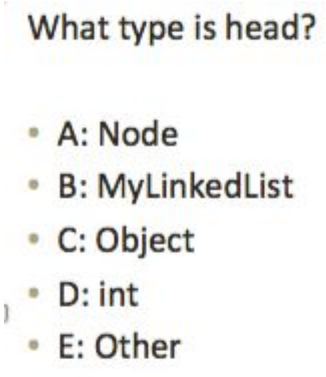
null null

- The dummy nodes **will always exist, even if the user hasn’t added any data yet.**
    - Head will never points to *null*
- These nodes will simplify the implementation for certain methods.
    - No need to check if the list is empty.

Dummy nodes

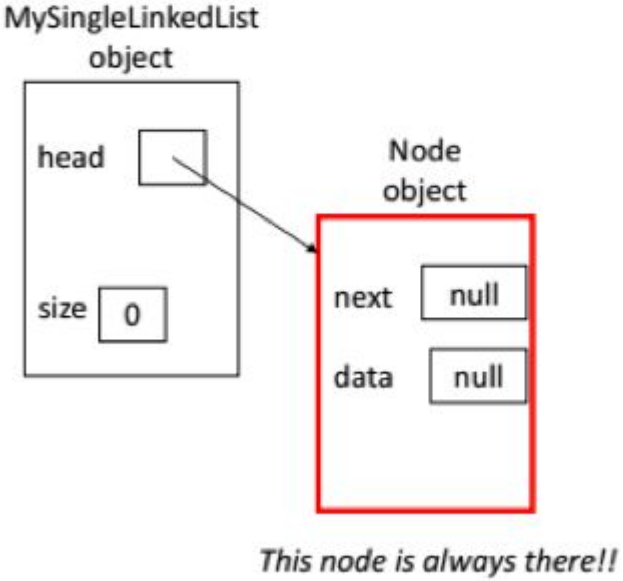


Objects of some type

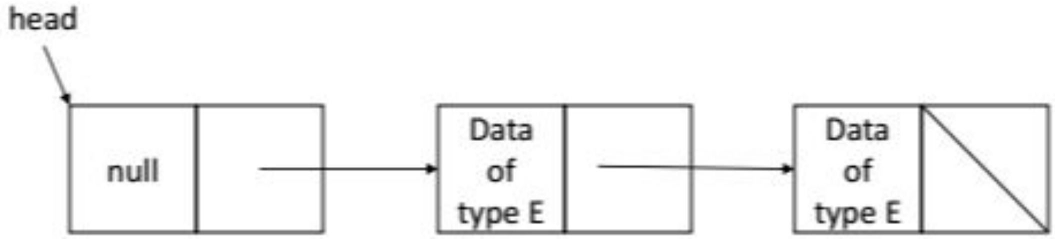

What type is head?

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

Empty list with sentinel node



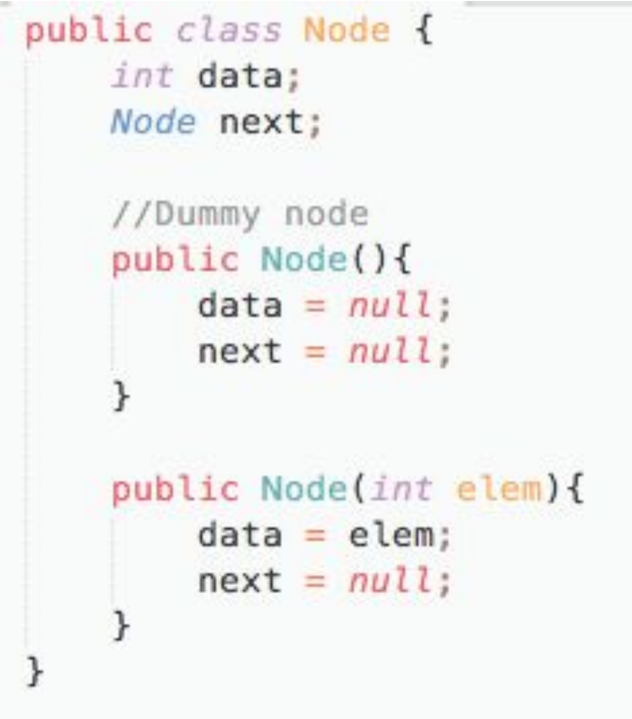
Quick check



What is the size of this linked list?

A: 0
B: 1
C: 2
D: 3

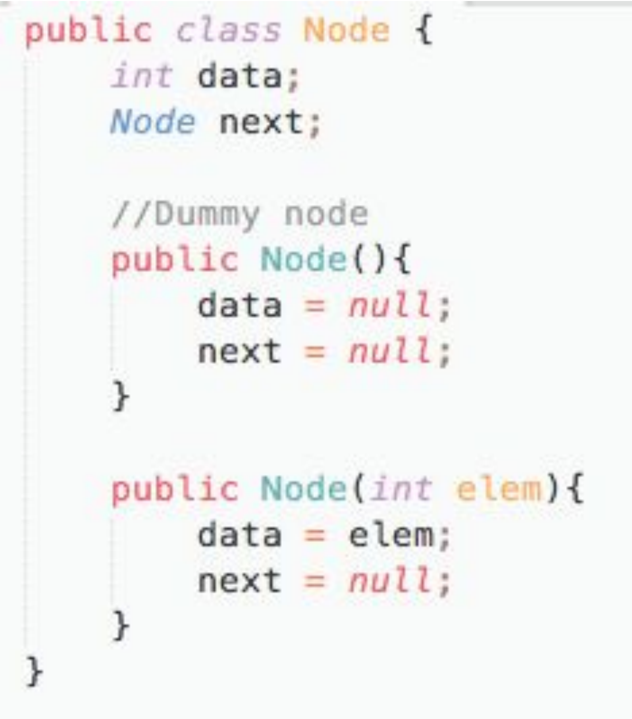
How to implement it?



```java
public class Node {
    int data;
    Node next;

    //Dummy node
    public Node(){
        data = null;
        next = null;
    }

    public Node(int elem){
        data = elem;
        next = null;
    }
}
```

```java
class MyList_dummy {  // needs modification

        Node head;
        int size;

        public MyList() {
                head = null;
                size = 0;
        }

        public void addFirst(int elem){
                Node toAdd = new Node(elem);
                size++;
                if (head==null) {
                        head = toAdd;
                }
                else {
                        toAdd.next = head;
                        head = toAdd;
                }
        }
}
```

How to implement it?



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

    //Dummy node
    public Node(){
        data = null;
        next = null;
    }

    public Node(int elem){
        data = elem;
        next = null;
    }
}
```

```
class MyList_dummy {

    Node head;
    int size;

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

    public void addFirst(int elem){
        Node toAdd = new Node(elem);
        size++;
        toAdd.next = head.next;
        head.next = toAdd;
        }
    }
}
```

Issue:

After we fix everything to work with dummy nodes, we will have a problem:

```
/Node.java:7: error: incompatible types: <null> cannot be converted to int
          data = null;
             ^
```

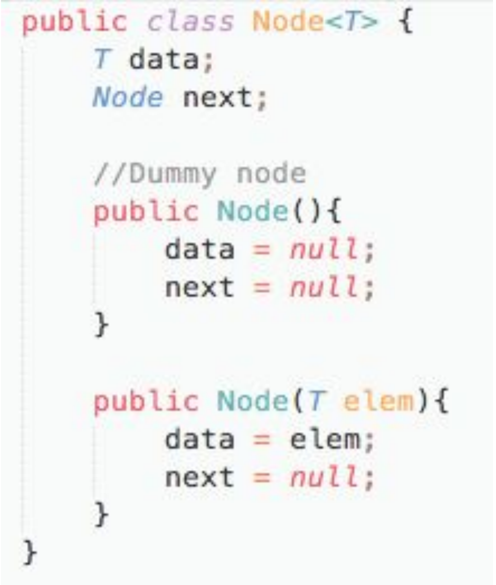
```
public Node(){
         data = null;
         next = null;
     }
```

Solution. Generics (more later)

After we fix everything to work with dummy nodes, we will have a problem:

```
/Node.java:7: error: incompatible types: <null> cannot be converted to int

          data = null;

             ^
```

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



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

    //Dummy node
    public Node(){
        data = null;
        next = null;
    }

    public Node(T elem){
        data = elem;
        next = null;
    }
}
```

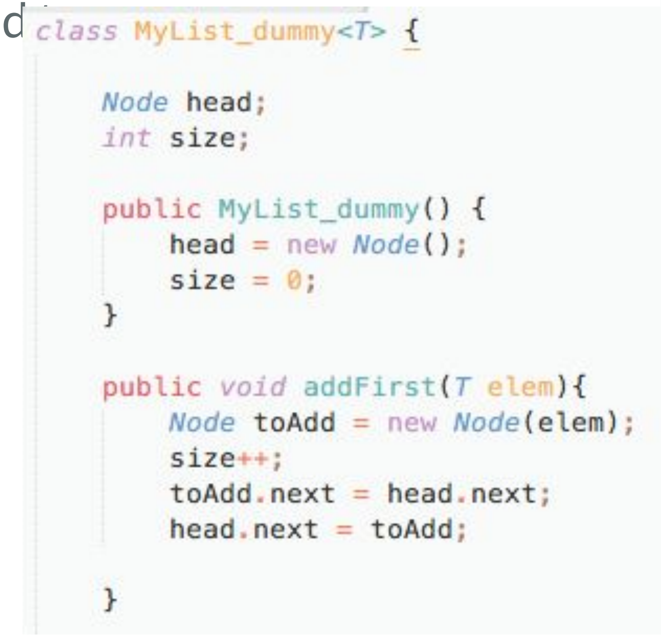
Solution. Generics (more later)

After we fix everything to work with dummy nodes, we will have a problem:

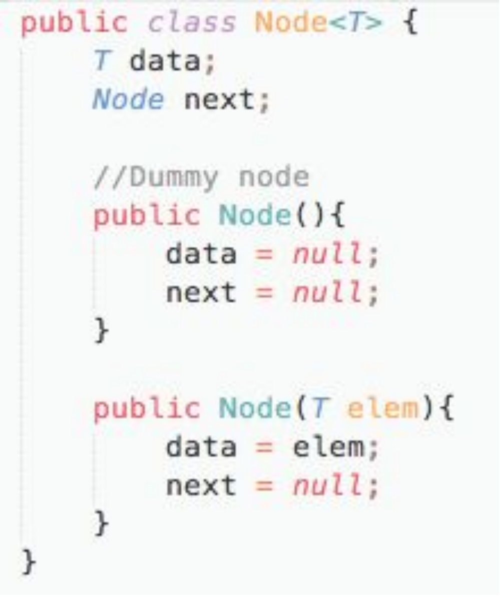
/Node.java:7: error: incompatible types: <null> cannot be converted
int

```
        data = null;
            ^
```

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

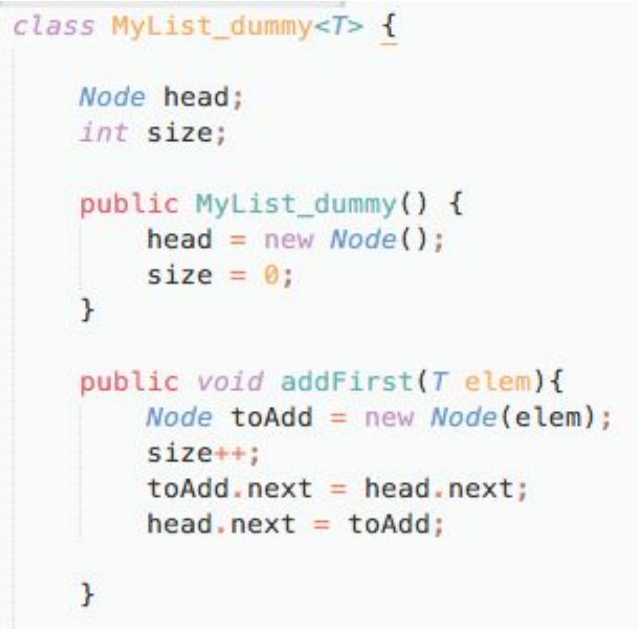


```
class MyList_dummy<T> {

    Node head;
    int size;

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

    public void addFirst(T elem){
        Node toAdd = new Node(elem);
        size++;
        toAdd.next = head.next;
        head.next = toAdd;

    }
```

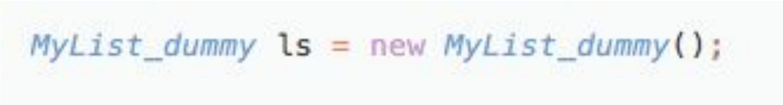


```
public class Node<T> {
    T data;
    Node next;

    //Dummy node
    public Node(){
        data = null;
        next = null;
    }

    public Node(T elem){
        data = elem;
        next = null;
    }
}
```

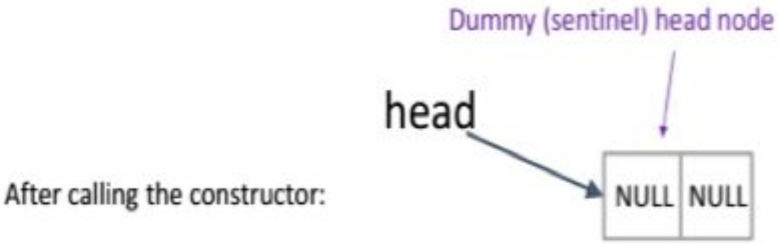
After calling constructor



```
class MyList_dummy<T> {

    Node head;
    int size;

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

    public void addFirst(T elem){
        Node toAdd = new Node(elem);
        size++;
        toAdd.next = head.next;
        head.next = toAdd;

    }
```



```
MyList_dummy ls = new MyList_dummy();
```



Generic types

- Generic types must be *reference types*. You cannot replace a generic type with a primitive type such as **int** or **char**.
- For example, the following statement is wrong:

```
ArrayList<int> intList = new ArrayList<int>();
```

- To create an **ArrayList** object for **int** values, you have to use:

```
ArrayList<Integer> intList = new ArrayList<Integer>();
```

- You can add an **int** value to **intList by creating a new object of type Integer.** For example,

```
intList.add(new Integer(5));
```

- Another way: You can add an **int** value to **intList**. For example,

```
intList.add(5);
```

Java automatically wraps 5 into new Integer(5). This is called **autoboxing**

Another example

- For example, the following statement creates a list for strings:

```
ArrayList<String> list = new ArrayList<String>();
```

- You can now add only strings into the list. For instance,

```
  list.add("Red");
```

- `list.add(new Integer(1));` // this is NOT ok

Diamond operator <>, idea

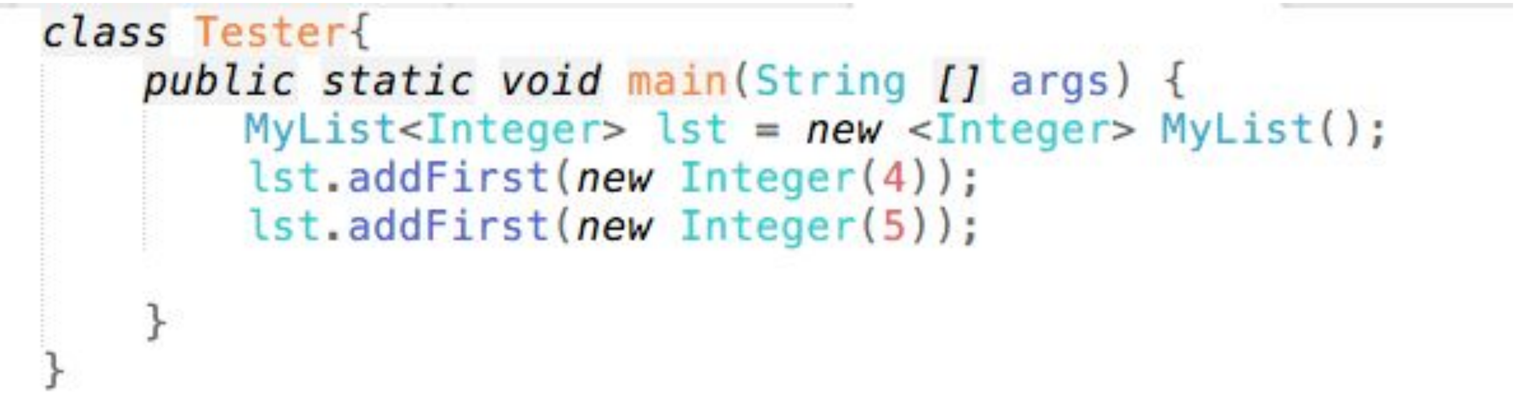
- Before JDK 7: Explicitly specifying generic class's instantiation parameter type.

```
ArrayList<String> list = new ArrayList<String>();
```

- After JDK 7:

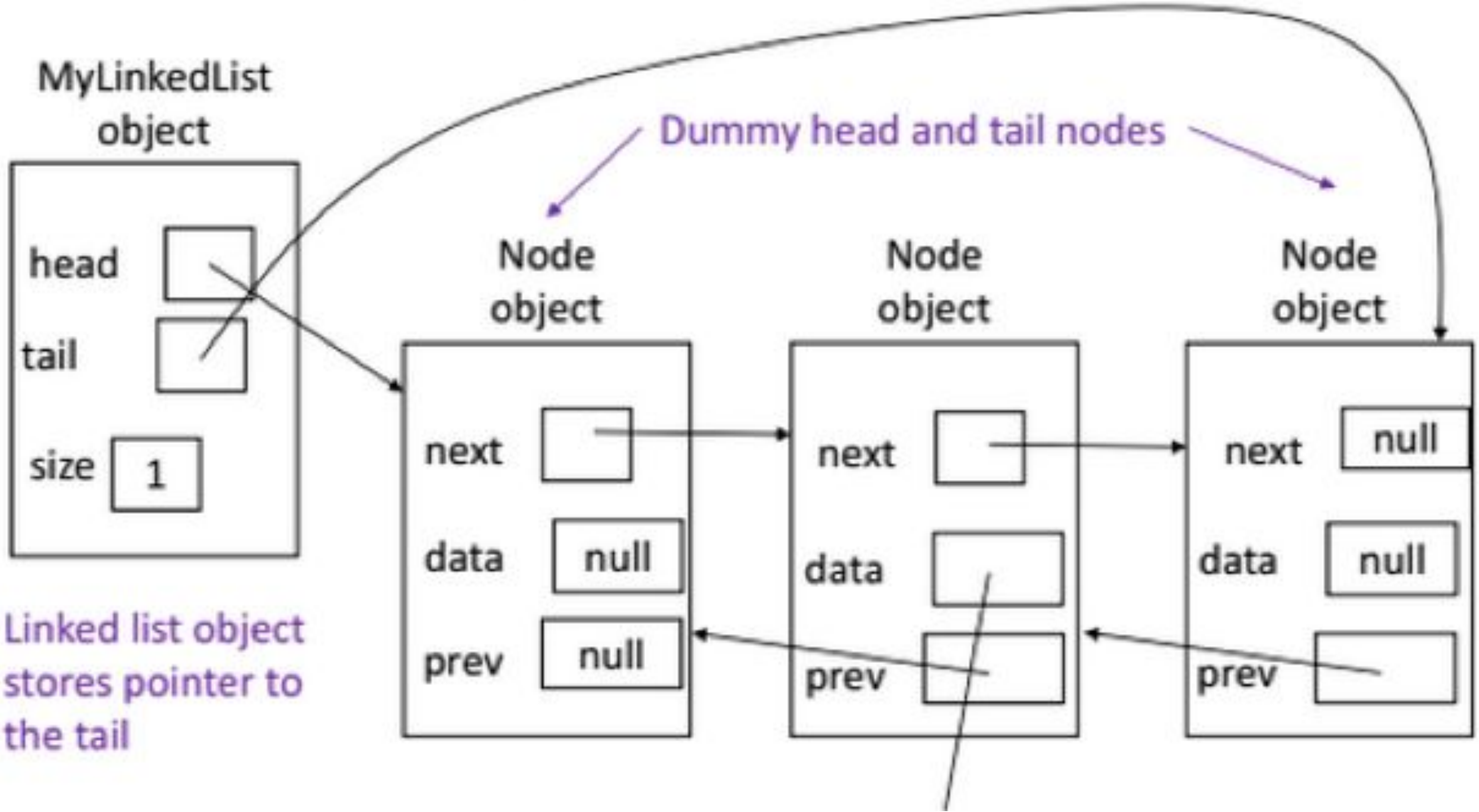
```
ArrayList<String> list = new ArrayList<>();
```

In main() or JUnit. Instantiation.



```
class Tester{
    public static void main(String [] args) {
        MyList<Integer> lst = new <Integer> MyList();
        lst.addFirst(new Integer(4));
        lst.addFirst(new Integer(5));

    }
}
```

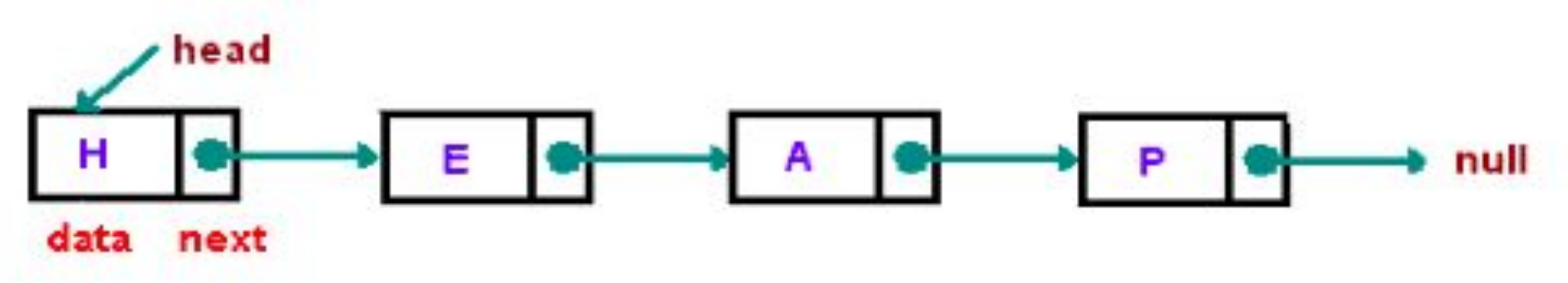
PA 4 : Doubly linked lists



Linked list object stores pointer to the tail

Let’s try to add (or remove?) in DLL

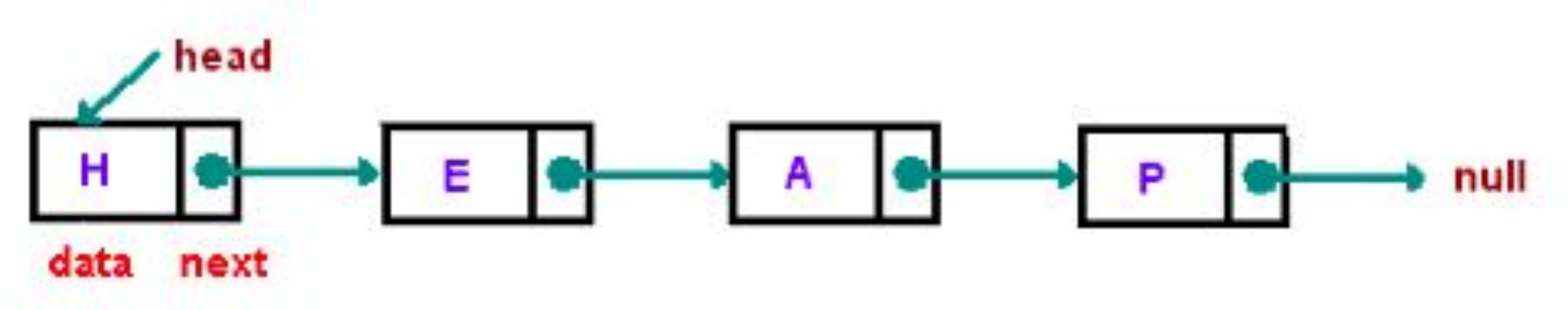
Running time for different operations



Insert new node as the first element in the list:

A: O (1)
B: O (log n)
C: O (n)
D: O (n log n)
E: O (n ^ 2)

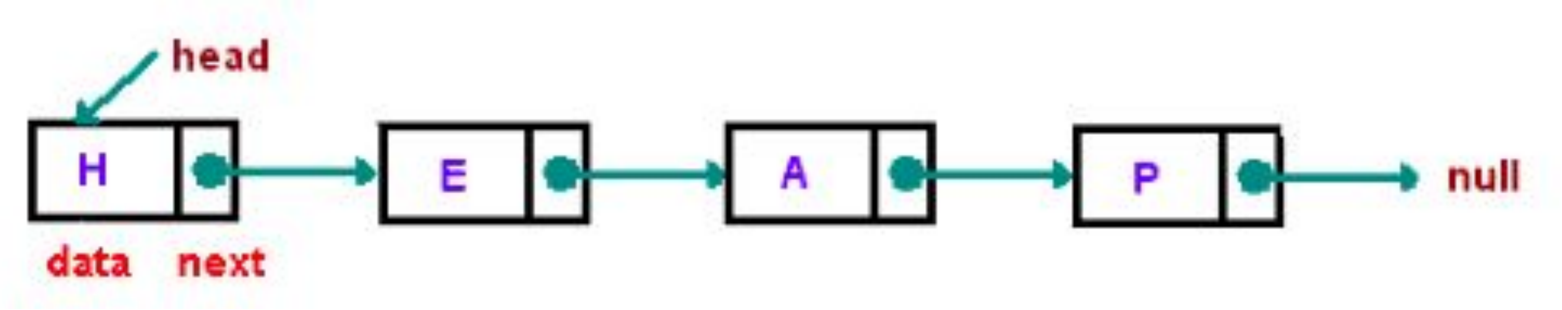
Running time for different operations



Insert new node as the last element in the list:

A: O (1)
B: O (log n)
C: O (n)
D: O (n log n)
E: O (n ^ 2)

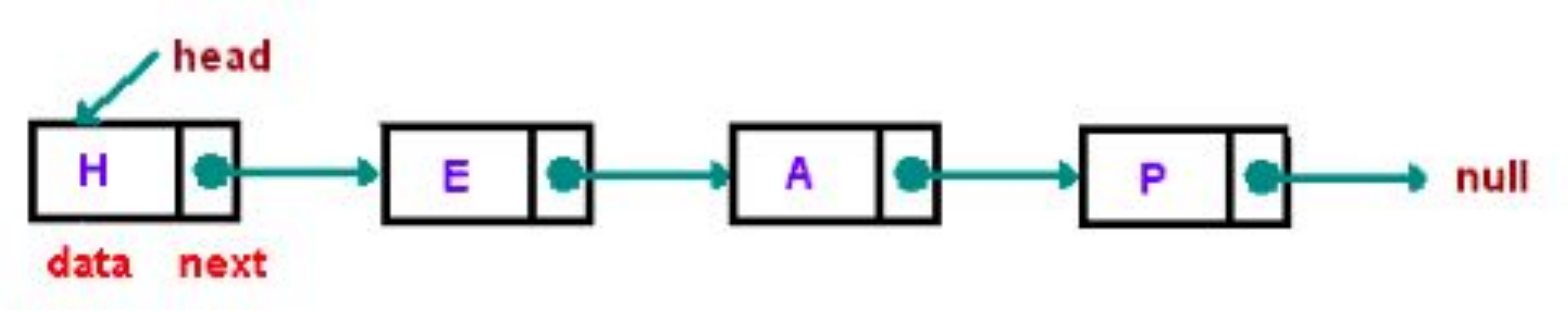
Running time for different operations



Find an element at a given index?

A: O (1)
B: O (log n)
C: O (n)
D: O (n log n)
E: O (n ^ 2)

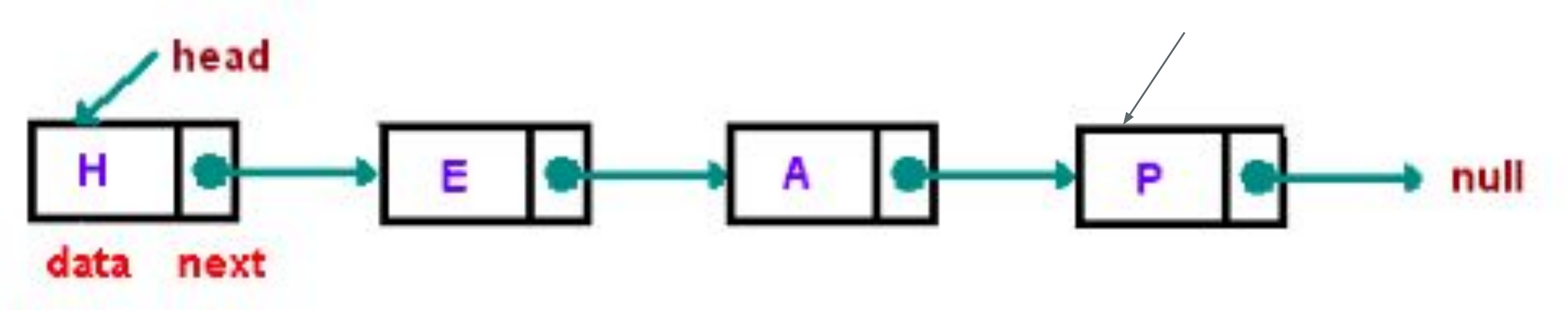
Running time for different operations



Remove the first element?

A: O (1)
B: O (log n)
C: O (n)
D: O (n log n)
E: O (n ^ 2)

Running time for different operations



Remove the last element?

A: O (1)
B: O (log n)
C: O (n)
D: O (n log n)
E: O (n ^ 2)

Doubly Linked lists

What operation is O(n), given a tail?

A: Insert front

B: Insert back

C: Search

D: Insert at the index (needs to find)

E: More than one

Question: head➳1➳2➳3➳4

What does the following function do for a given Linked List with first node as head?

```
void fun1(Node head)
{
    if(head == NULL)
        return;

    fun1(head.next);
     SOP (head.data);   # System.out.print
}
```

A: Checks if a given list is empty.

B: Print elements of the linked list.

C: Print elements in the reverse order.

D: Print the first element of the linked list, given the list is not empty.