Doubly-linked lists
Iterators

Lecture 15

by Marina Barsky

Doubly-linked Lists: Node

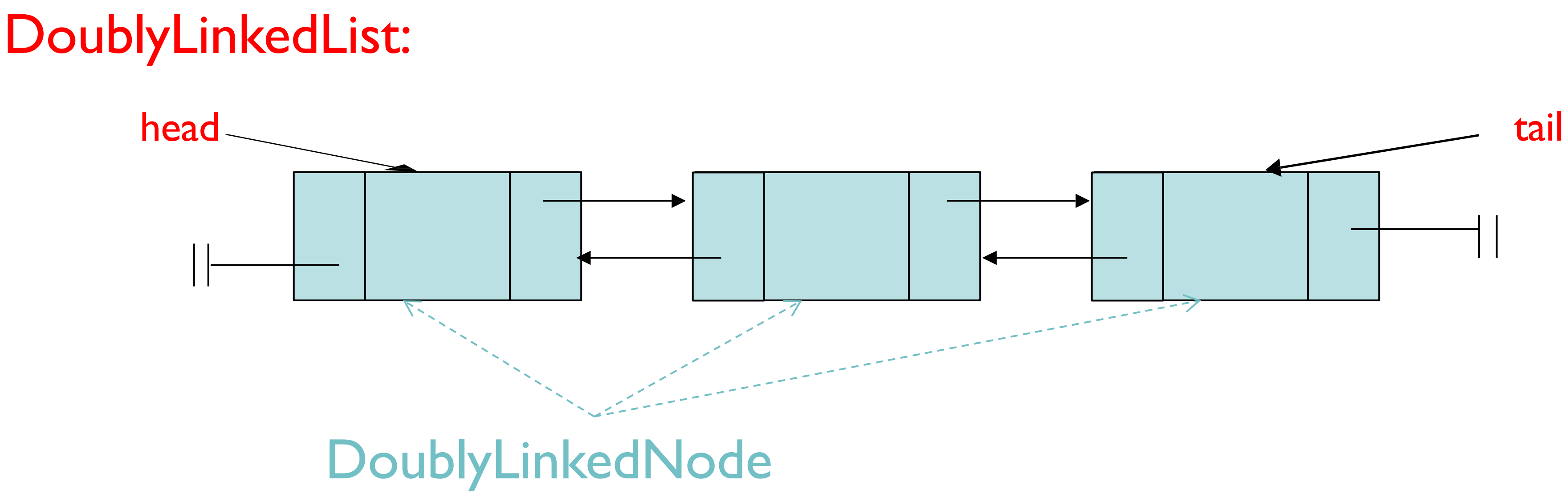
```
class Node {
    int data;
    Node next;
}
```

➡

```
class DoublyLinkedNode {
    int data;
    Node prev;
    Node next;
}
```

Doubly-Linked List with tail pointer

- Keeps reference/links in both directions
- Traversing can start from either end



In a [doubly-linked] list *head* will be equal to *tail*:

A. Always

B. Never

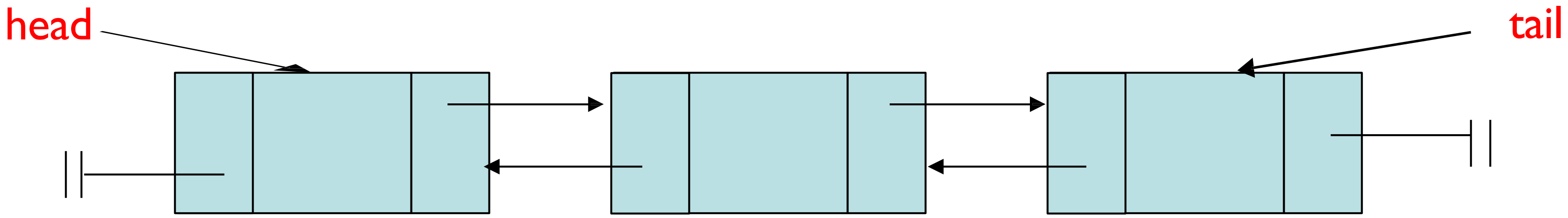
C. When the list is empty

D. When there is one element

E. More than one of the above

Doubly-Linked List: tradeoffs

- ✓ Links in both directions: → can traverse forwards and backwards!
- ✓ ALL tail operations (including *remove last*) are fast! Why?
  - We have direct access to the tail node **& its predecessor**
- × Additional code complexity in each list operation
  - Example: *add (int index, E element)* need to consider 4 cases:
    - empty list
    - add to front
    - add to tail
    - add in middle
- × Additional space consumption (storing previous)



Stitching new node between two existing nodes

The code below adds a new node with data 'X' between two nodes P (parent) and C (child) in a doubly-linked list

```
DoublyLinkedNode x = new DoublyLinkedNode('X');

if (C != null) C.previous = x;
if (P != null) P.next = x;
```

What should happen if both N and P are null?

A. Nothing should happen: the code above already covers this case

B. We need to set head = x;

C. We need to set C = x;

D. We need to set P=x;

E. Something else

Why would anyone use a singly-linked vs. a doubly-linked list?

A. A singly-linked list uses less memory.

B. It is easier to implement the insertion at position *i*.

C. It's faster to remove an element from the end.

D. None of the above.

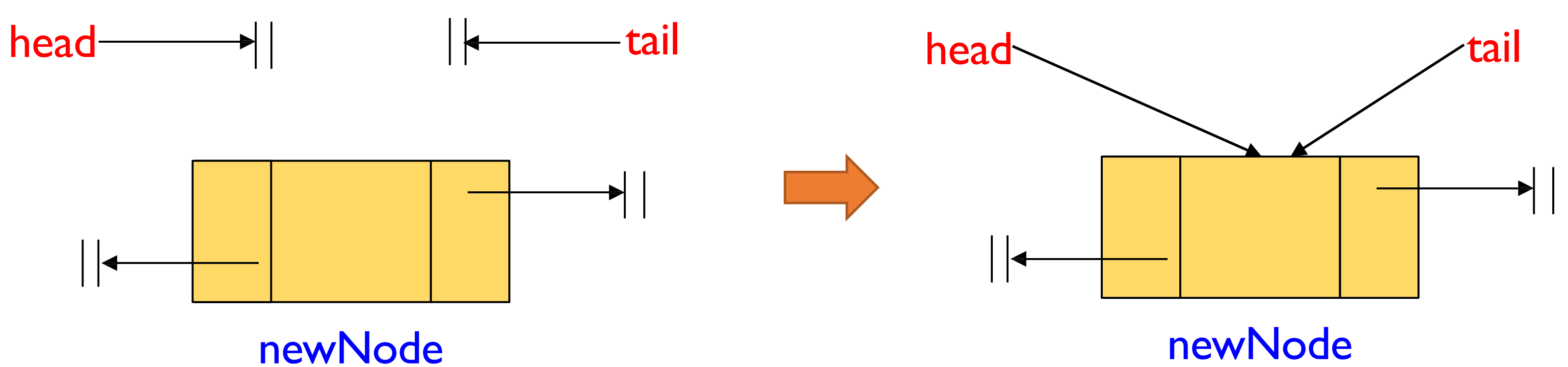
E. More than one of the above

Moving heads and tails

- When we add/remove in front – we need to update *head*
- When we add/remove at the end – we need to update *tail*
- When the linked list currently is or becomes empty: *head*=*tail*=null
- Many special cases arise!

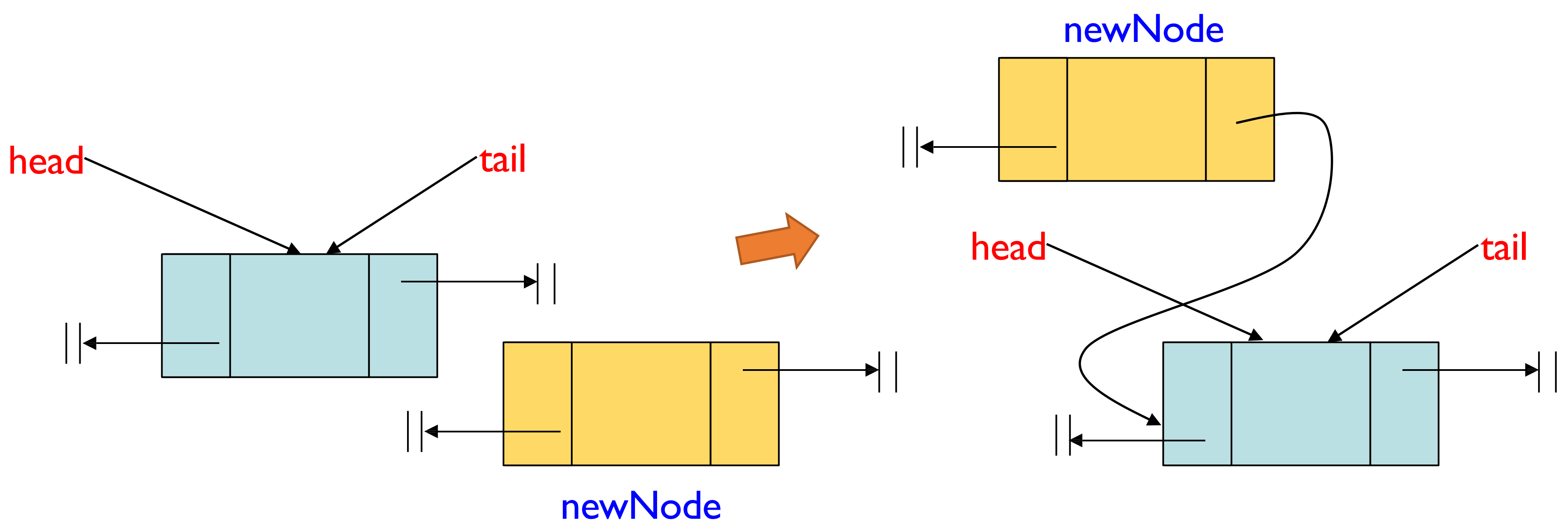
Example: add in front

```
newNode = new DoublyLinkedNode(newData, prev=null, next=null)
if head == null:  //empty list
        head = newNode
        tail = head
```



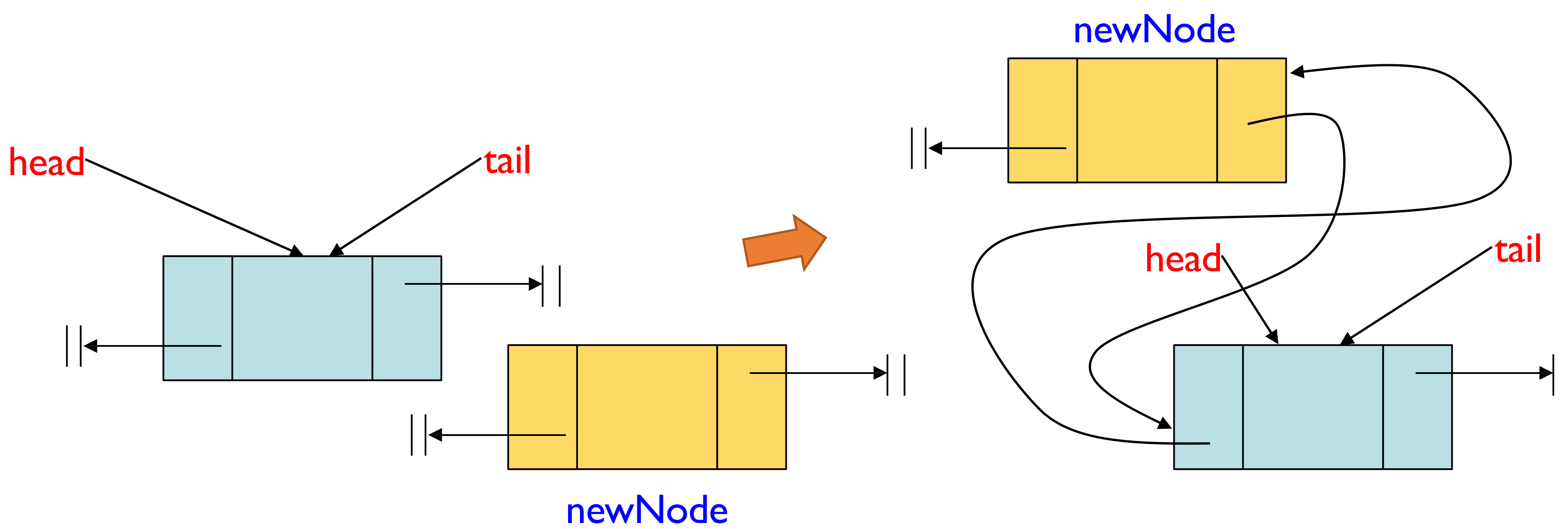
Example: add in front

```
newNode = new DoublyLinkedNode(newData, prev=null, next=null)
if head == null:  //empty list
       head = newNode
       tail = head
else: //list with at least one real node
       newNode.next = head
```



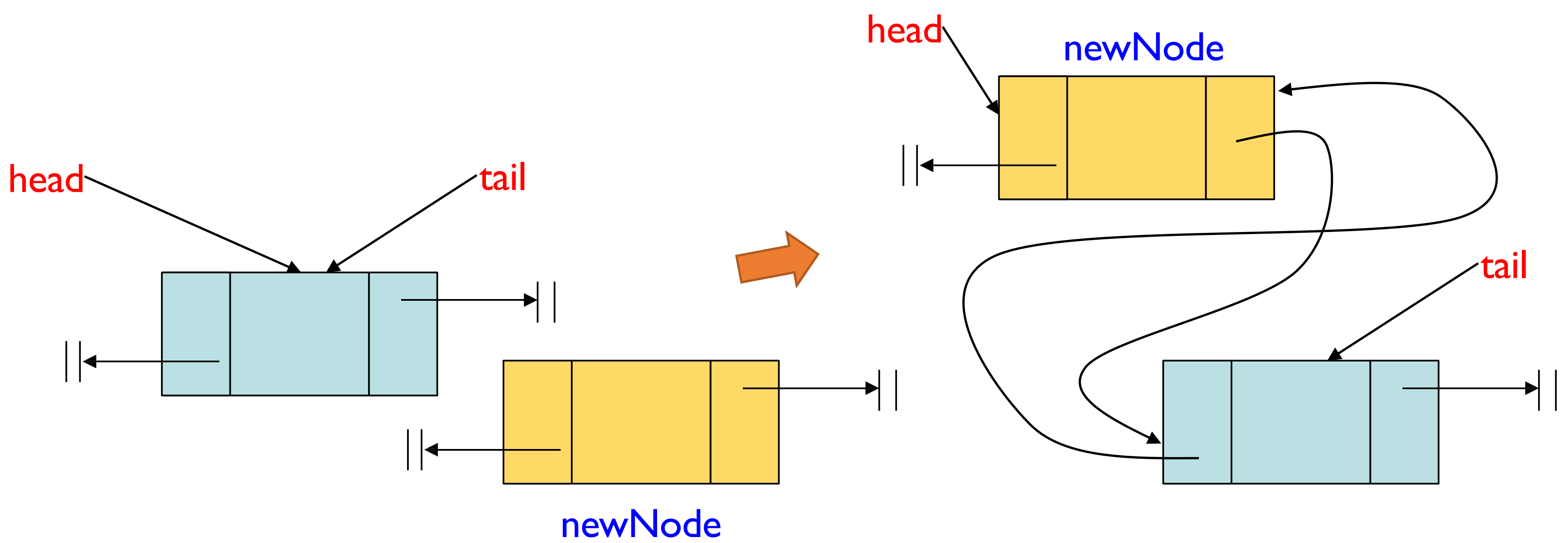
Example: add in front

```
newNode = new DoublyLinkedNode(newData, prev=null, next=null)
if head == null:  //empty list
        head = newNode
        tail = head
else: //list with at least one real node
        newNode.next = head
        head.prev = newNode
```



Example: add in front

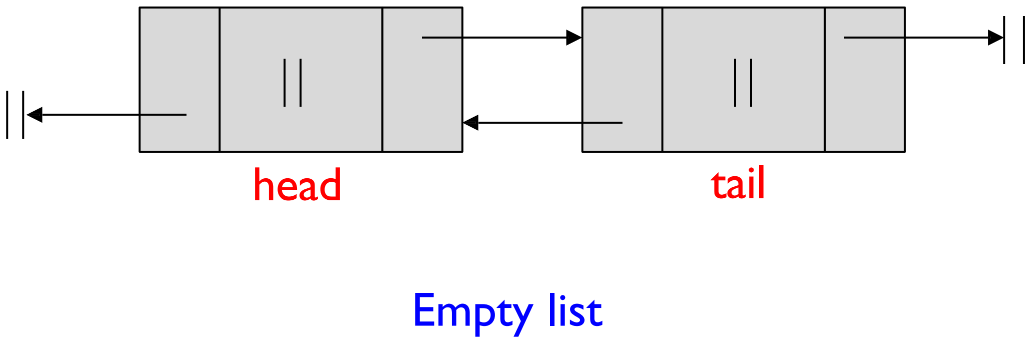
```
newNode = new DoublyLinkedNode(newData, prev=null, next=null)
if head == null:  //empty list
       head = newNode
       tail = head
else: //list with at least one real node
       newNode.next = head
       head.prev = newNode
       head = newNode
```



Sentinel Nodes (aka *Dummy* nodes)

- We can get rid of special cases if we add fake head and tail nodes
- These are called *sentinel* nodes are they are always present and contain no data
- We can have one sentinel for both head and tail, or we can have a separate node for each
- The head and tail pointers never move and the nodes are inserted between them

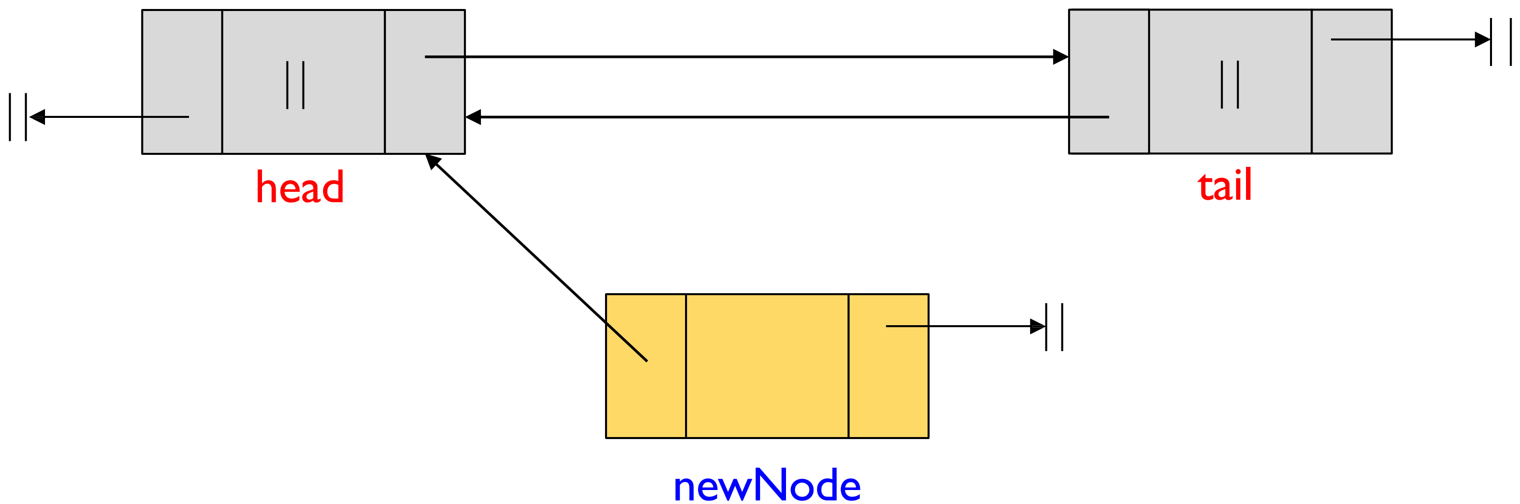
Doubly-Linked List with two sentinels: ***constructor***




```
public MyLinkedList() {
    head = new Node(null);
    tail = new Node(null);
    head.next = tail;
    tail.prev = head;
    size = 0;
}
```

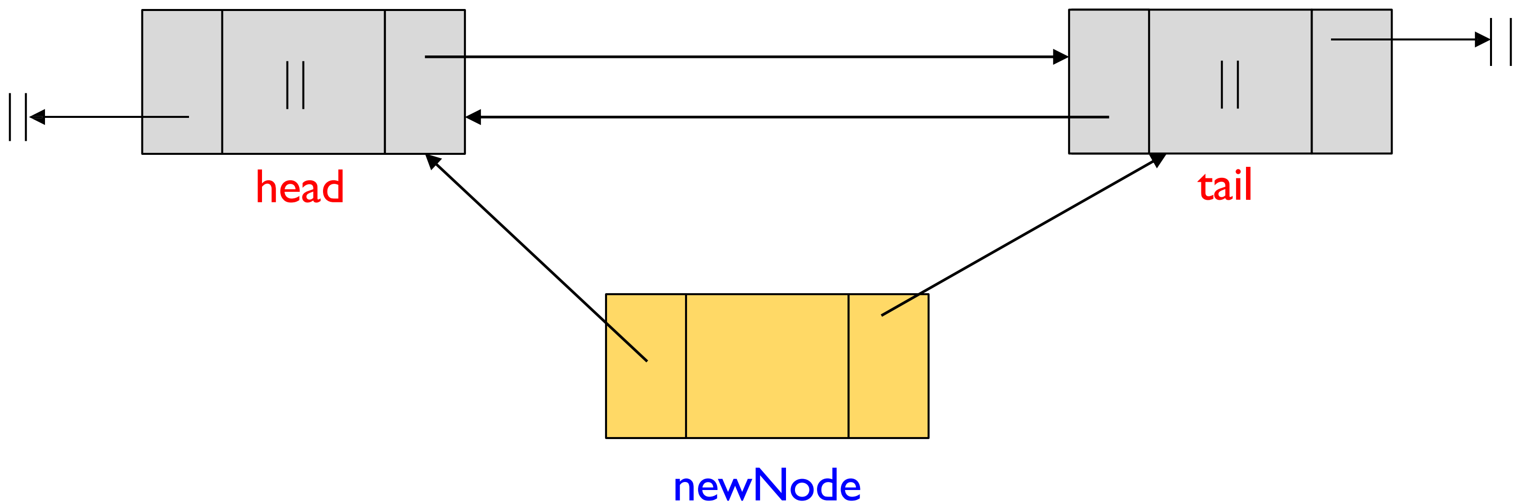
Add in front with sentinels

```
newNode = new DoublyLinkedNode (newData, prev=null, next=null)

//empty list
newNode.prev = head
```



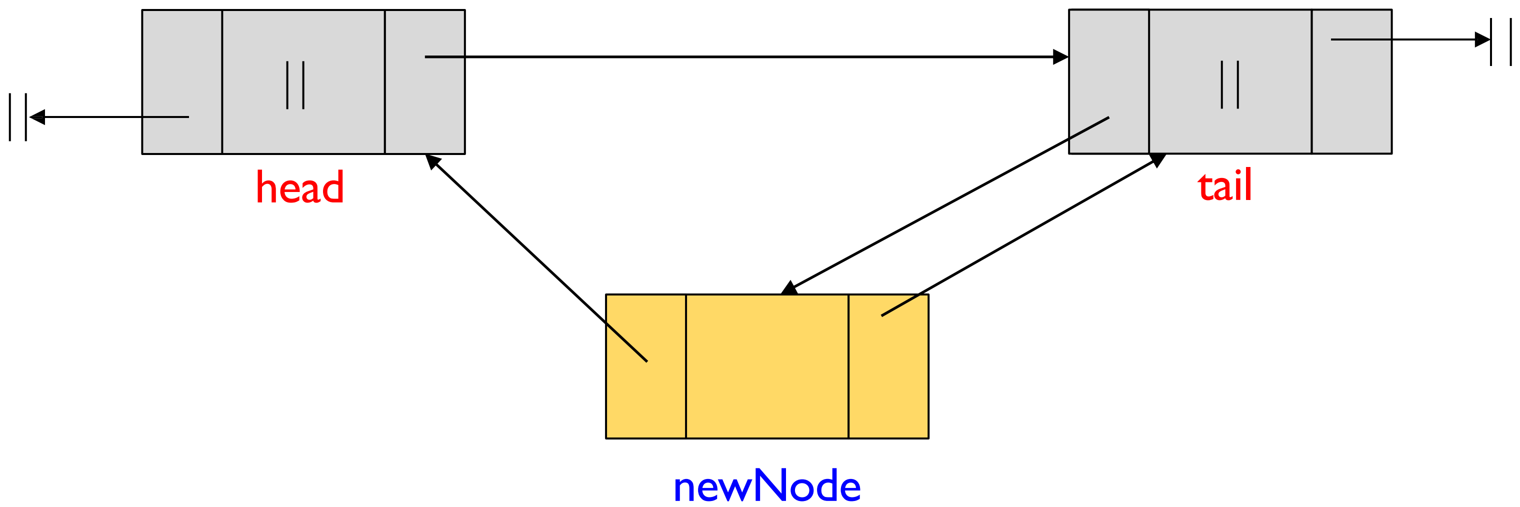
Add in front with sentinels

```
newNode = new DoublyLinkedNode (newData, prev=null, next=null)

//empty list
newNode.prev = head
newNode.next = head.next
```



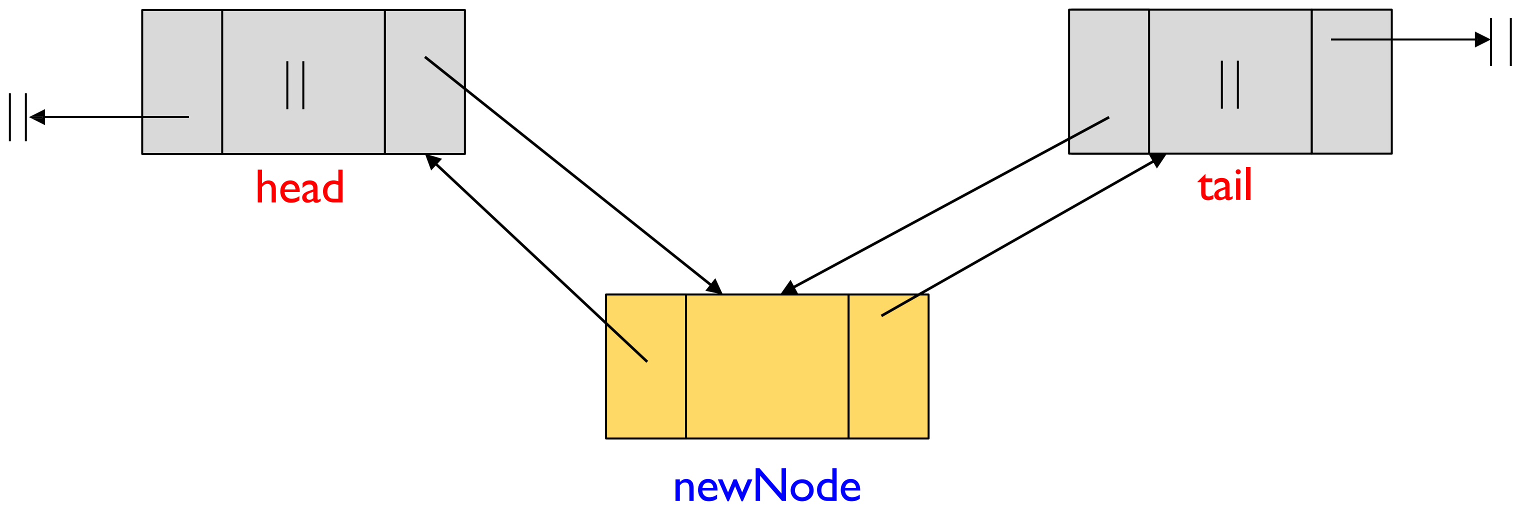
Add in front with sentinels

```
newNode = new DoublyLinkedNode (newData, prev=null, next=null)

//empty list
newNode.prev = head
newNode.next = head.next
head.next.prev = newNode
```



Add in front with sentinels

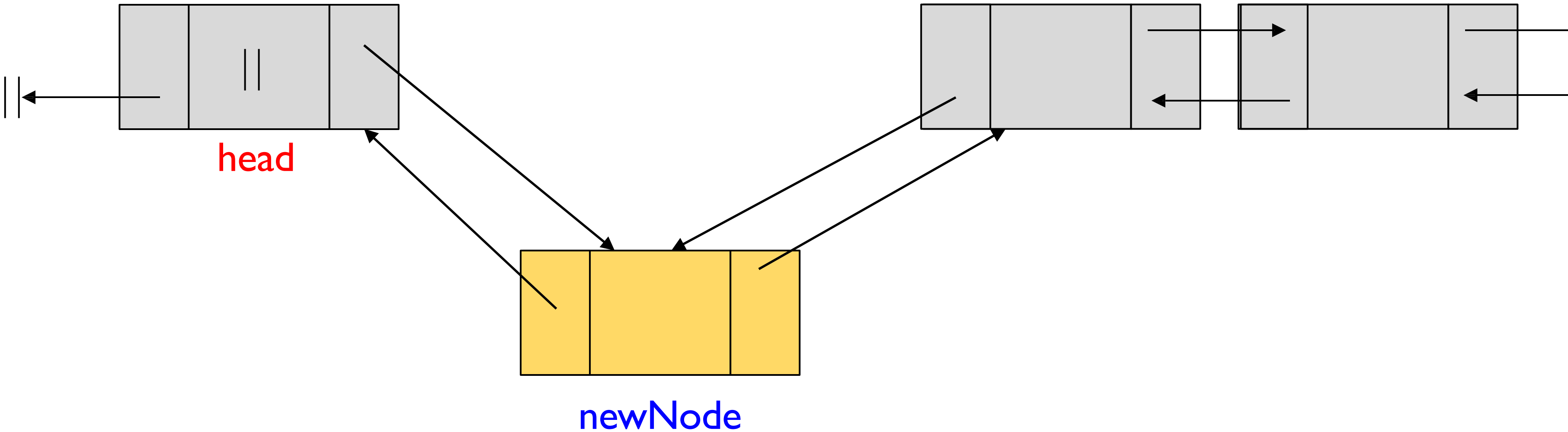
```
newNode = new DoublyLinkedNode (newData, prev=null, next=null)

//empty list
newNode.prev = head
newNode.next = head.next
head.next.prev = newNode
head.next = newNode
```



Add in front with sentinels

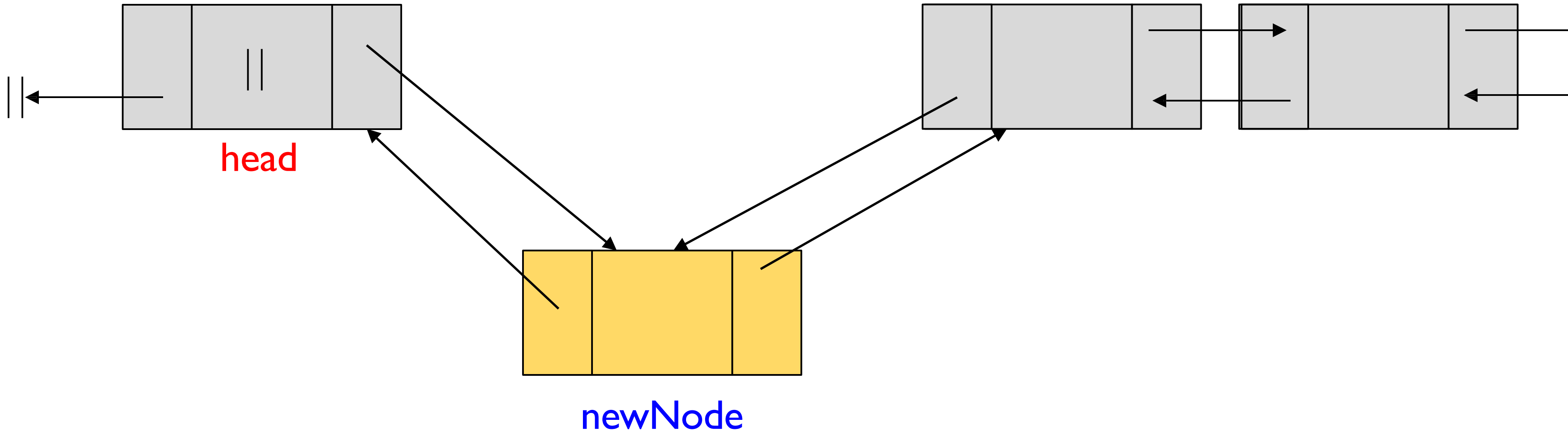
```
newNode = new DoublyLinkedNode (newData, prev=null, next=null)

newNode.prev = head
newNode.next = head.next
head.next.prev = newNode
head.next = newNode
```

This also works for non-empty lists – there are no special cases!



Lab 4.
Doubly-linked lists with two sentinels

- In Lab 4 you will implement this idea
- **Always draw the list before and after each operation** to make sure you update all the links correctly



List Iterators

Recap ADT *List*: supported operations

ADT *List* supports the following main operations

- Get element by position: `get(int index)`
- Search element: `indexOf(E element)`
- Add new element: `add(int index, E element)`
- Remove element by position: `remove(i)`

For some problems however these operations are insufficient and we **need access to the underlying implementation of the data**

Example: count occurrences

- Write a method *count* that counts the number of times a particular element *o* appears in a List:

```
public static int count(List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

- **Question**: would this work well no matter if the List is an *Array List* or a *Linked List*?

Example: count occurrences

- Write a method *count* that counts the number of times a particular element *o* appears in a List:

```
public static int count(List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

- **Answer:** No, this method is very inefficient for Linked Lists: *get(i)* always starts from the *head* and this is an O(n^2) loop

Efficient solutions are fundamentally different for:

- **Array List**

```
int count (E element){
   int counter = 0;
   for(int i=0; i<size; i++){
      if(data[i].equals(element))
           counter ++;
   }
   return counter;
}
```

Using *for* loop and indexes

- **Linked List**

```
int count (E element){
   int counter = 0;
   Node finger = head;
   while(finger != null){
    if(finger.data.equals(element))
        counter ++;
    finger = finger.next;
   }
   return counter;
}
```

Using *while* loop and *next*

- But the principle of ADT **forbids the use of underlying data structures directly**!
- We need a uniform interface to iterate over List elements efficiently

Efficient uniform iteration over List

- Problem: Efficient and uniform dispensing of values from the underlying data structures
- Solution: We create and use the common interface for iteration

Extending operations for List ADT

- get()
- indexOf()
- add()
- remove()
- size()
- isEmpty()
- clear()
- contains()

But also method for efficient data traversal

- iterator()

Iterator interface

- Iterators provide support for efficiently visiting all elements of an underlying data structure
- We customize the implementation of the iterator depending on the data structure
- We abstract away the details of how to access elements

public interface *Iterator*<E> :

boolean ***hasNext()*** – are there more elements for iteration?

E ***next()*** – return next element

Example: Iterator for *Array List*

Can be a part of the ArrayList class

```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int nextIndex;
     public ArrayListIterator (ArrayList list){
          this.list = list;
          this.nextIndex = 0;
     }

     public boolean hasNext (){
          return (this.nextIndex < list.size());
     }

     public Object next(){
          return list.data[nextIndex++];
     }
}
```

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int nextIndex;
     public ArrayListIterator (ArrayList list){
           this.list = list;
           this.nextIndex = 0;
     }

     public boolean hasNext (){
           return (this.nextIndex < list.size());
     }

     public Object next(){
           return list.data[nextIndex++];
     }
}
```

Reference to the actual Array List

We set it in the constructor

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int nextIndex;
     public ArrayListIterator (ArrayList list){
          this.list = list;
          this.nextIndex = 0;
     }

     public boolean hasNext (){
          return (this.nextIndex < list.size());
     }

     public Object next(){
          return list.data[nextIndex ++];
     }
}
```

Stores the current state of the iteration: the position in the array to be returned next

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
    ArrayList list;
    int nextIndex;
    public ArrayListIterator (ArrayList list){
        this.list = list;
        this.nextIndex = 0;
    }

    public boolean hasNext (){
        return (this.nextIndex < list.size());
    }

    public Object next(){
        return list.data[nextIndex++];
    }
}
```

As long as nextIndex is within valid bounds

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
    ArrayList list;
    int nextIndex;
    public ArrayListIterator (ArrayList list){
        this.list = list;
        this.nextIndex = 0;
    }

    public boolean hasNext (){
        return (this.nextIndex < list.size());
    }

    public Object next(){
        return list.data[nextIndex++];
    }
}
```

Return the element at position *nextIndex*, and advance *nextIndex* to the next position

ArrayList *iterator*() returns array-specific Iterator:

```
public class ArrayList {
    Object[] data;
    int size;

    public Iterator iterator (){
        return new ArrayListIterator(this);
    }

}
```

Iterator for *Linked List*

```
private class LinkedListIterator implements Iterator{
     LinkedList list;
     Node next;

     public LinkedListIterator (LinkedList list){
           this.list = list;
           this.next = list.head;
     }

     public boolean hasNext (){
     }

     public Object next(){
     }
}
```

Same as before: reference to the actual Linked List

Iterator for *Linked List*

```
private class LinkedListIterator implements Iterator{
      LinkedList list;
      Node next;

      public LinkedListIterator (LinkedList list){
            this.list = list;
             next = list.head;
      }

      public boolean hasNext (){
      }

      public Object next(){
      }
}
```

Stores the current state of the iteration: node to be read next

Linked List Iterator: *hasNext*()

Which of the following is the correct implementation of *hasNext*()?

A.
```
boolean hasNext(){
     return (this.list.size()>0)
}
```

B.
```
boolean hasNext(){
     return (next.next != null)
}
```

C.
```
boolean hasNext(){
     return (next!= null)
}
```

D. None of the above

```
public class LinkedListIterator
              implements Iterator{
      LinkedList list;
      Node next;
      public boolean hasNext (){
      }

      public Object next(){
      }
}
```

Linked List Iterator: *next*()

Which of the following is the correct implementation of *next*()?

A.
```
Object next(){
    return this.list.get(next)
}
```

B.
```
Object next(){
      next = next.next;
      return next.data;
}
```

C.
```
Object next(){
      Object result = next.data;
      next = next.next;
      return result;
}
```

D. None of the above

```
public class LinkedListIterator
                    implements Iterator{
        LinkedList list;
        Node next;
        public boolean hasNext (){
        }

        public Object next(){
        }
}
```

Iterator for *Linked List*

```
public class LinkedListIterator implements Iterator{
     LinkedList list;
     Node next;
     …
     public boolean hasNext (){
          return (next != null);
     }

     public Object next(){
          Object result = next.data;
          next = next.next;
          return next;
     }
}
```

hasNext() basically answers: can I call next()?

Linked List with its own iterator

```
public class LinkedList {
    Node head;
    int size;

    public Iterator iterator (){
        return new LinkedListIterator(this);
    }

}
```

Uniform Counting with iterator()
Works for both Array List and Linked List

```
public int count (List list, Object o) {
    int counter = 0;
    Iterator iter = list.iterator();
    while (iter.hasNext())
        if(o.equals(iter.next())) counter++;
    return counter;
}
```

Data-structure specific operations inside

Iterators: notes

- Iterator objects provide a common interface for traversing List ADT
- They have access to internal data representations
- They also store the state of traversal
- To implement an efficient iterator you need to **understand the mechanics of the underlying data structure**