Lecture 11

ADT List

Modeling Achievements

- We want to keep track of the world records in different sports
- The collection should be dynamic: we should be able to add/edit/remove records
- At any time we want to be able to answer the questions:
  - Who is the fifth fastest marathon runner?
  - What is the world ranking of Hikaru Nakamura?

Ath.#	Perf.#	Time (s)	Athlete	Nation	Date	Place
1	1	1:16:36	Yusuke Suzuki	Japan	15 MAR 2015	Nomi
2	2	1:16:43	Sergey Morozov	Russia	08 JUN 2008	Saransk
3	3	1:16:54	Kaihua Wang	China	20 MAR 2021	Huangshan
4	4	1:17:02	Yohann Diniz	France	08 MAR 2015	Arles
5	5	1:17:15	Toshikazu Yamanishi	Japan	17 MAR 2019	Nomi
6	6	1:17:16	Vladimir Kanaykin	Russia	29 SEP 2007	Saransk

The men's 20 km race walk. All-time top 6. Correct as of August 2023.

ADT: Sequence of values, List

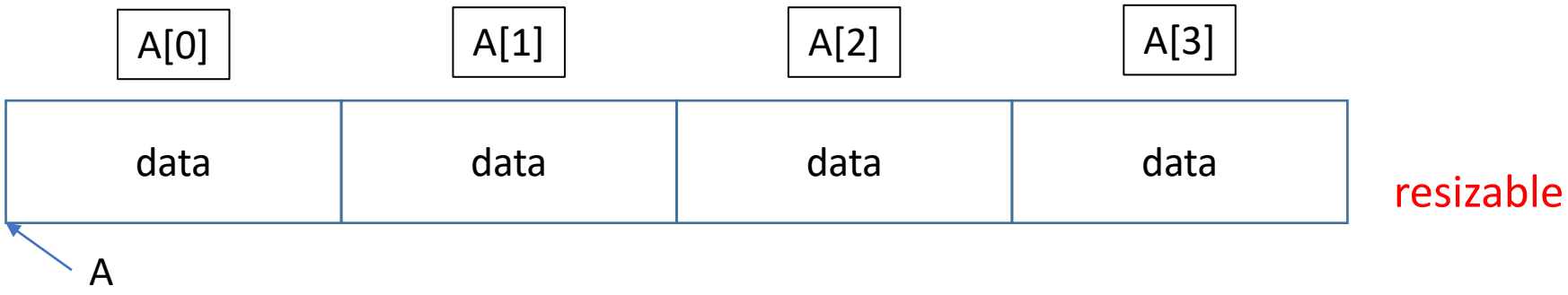
Specification for List:

- We need to store:
  - sequence of values, the order matters

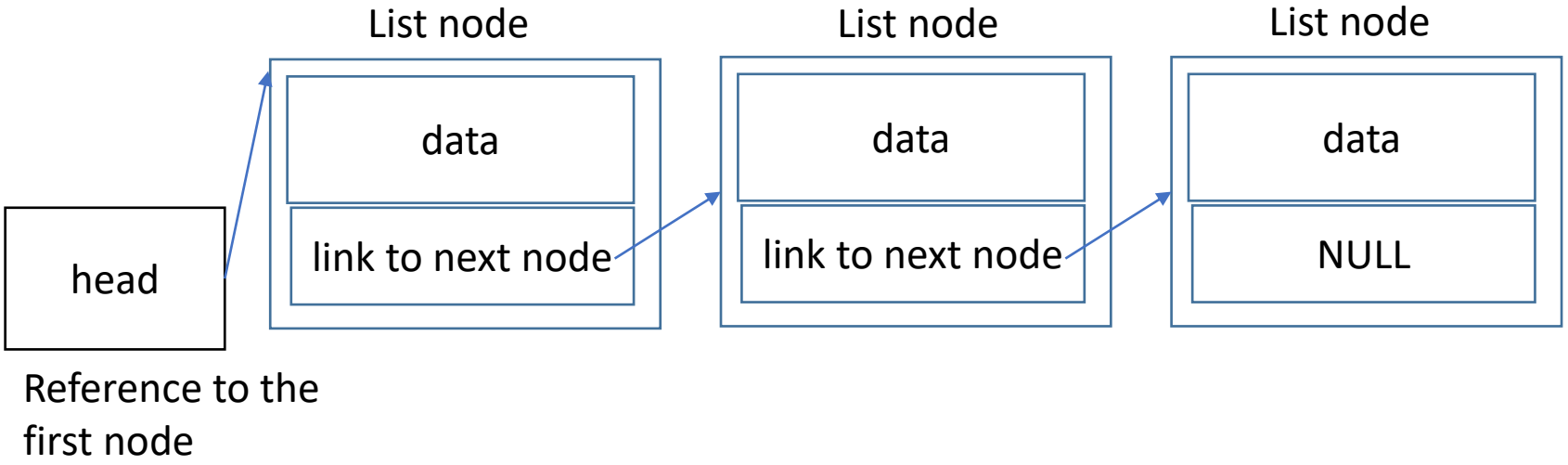
- We need to support the following operations:
  - Get element by position: `get(int index)`
  - Search for a position of a given element: `indexOf(E element)`
  - Add new element at position *i*: `add(int i, E element)`
  - Remove element by position: `remove(i)`

List ADT: possible implementations

- Using a **Dynamic Array**



- Using a **Linked List**



Implementing List ADT using a Dynamic Array: tradeoffs

+

- Get(i) in O(1)
- Removing/Adding to the end in O(1)

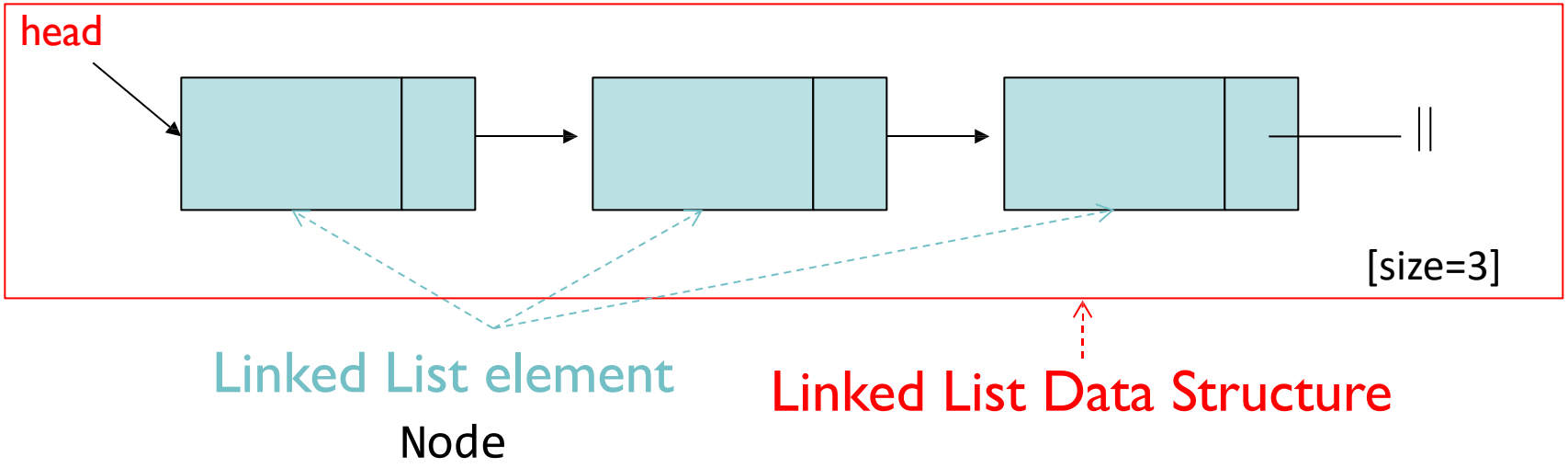
-

- Add/remove from position 0 O(n)
- Adding to the end can slow down due to doubling
- Wasted space: doubling and then removing – dynamic arrays never shrink

Implementing List ADT using Linked List

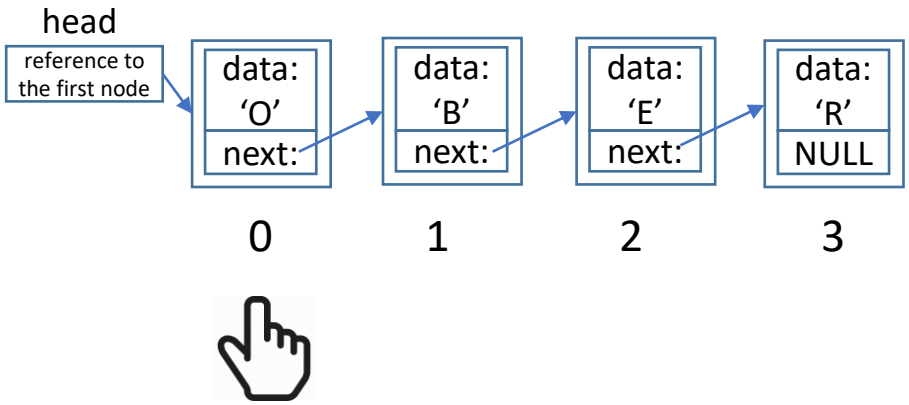
Linked List contains:

- Reference to the head of the list: `Node` *`head`*
- [Optional] The number of elements in the list: `int` *`size`*



Traversal: get node by position

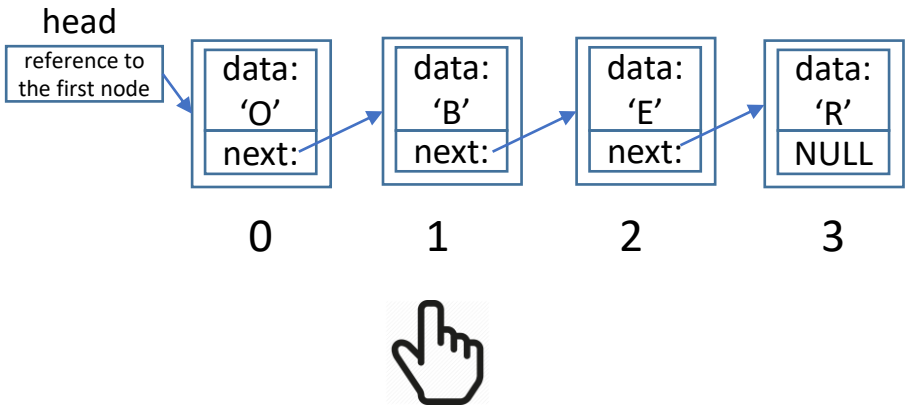
```
private Node getNth(int n) {  //Finds and returns the n-th node of the Linked List
    if (n >= size)
        Error
    Node finger = head;
    while (n > 0) {
        finger = finger.next;
        n--;
    }
    return finger;
}
```



We want the node with index 2:
`getNth(2)`
`n=2`

Traversal: get node by position

```
private Node getNth(int n) {
    if (n >= size)
        Error
    Node finger = head;
    while (n > 0) {
        finger = finger.next;
        n--;
    }
    return finger;
}
```

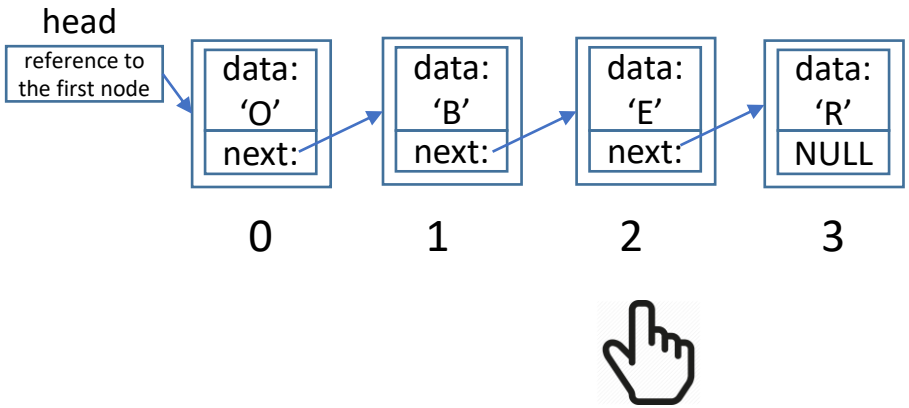


We want the node with index 2

n=1

Traversal: get node by position

```
private Node getNth(int n) {
    if (n >= size)
        Error
    Node finger = head;
    while (n > 0) {
        finger = finger.next;
        n--;
    }
    return finger;
}
```



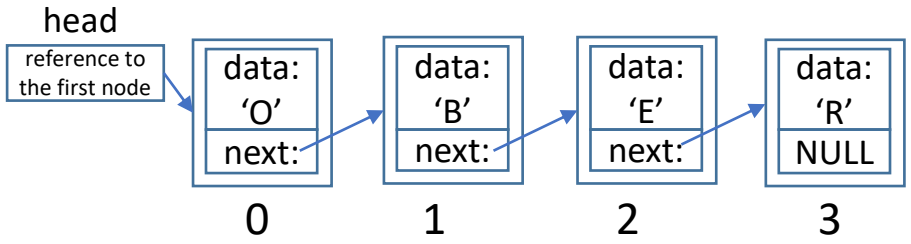
We want the node with index 2

n=0

Stop and return

General add (int index, E element)

Which of the following correctly adds a new node ‘M’ at position 1 of the Linked List below?



A.
```
Node mnode = new Node(‘M’);
Node parent = getNth(1);
mnode.next =  parent.next;
parent.next = mnode;
```

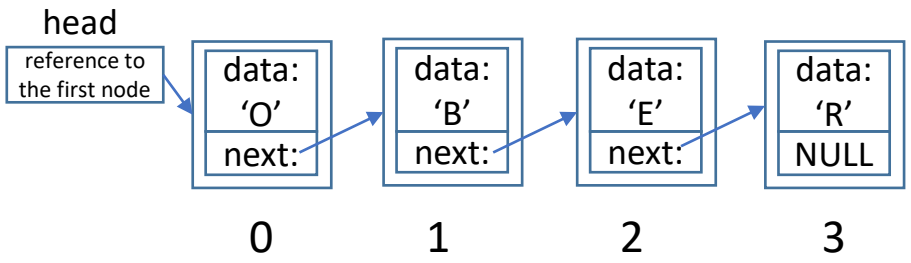
B.
```
Node mnode = new Node(‘M’);
Node parent = getNth(0);
parent.next = mnode;
mnode.next =  parent.next;
```

C.
```
Node mnode = new Node(‘M’);
Node child = getNth(1);
mnode.next = child;
```

D.
```
Node mnode = new Node(‘M’);
Node parent = getNth(0);
mnode.next =  parent.next;
parent.next = mnode;
```

E. None of the above

remove (int index, E element)



Which of the following correctly removes node at index 2?

A.
```
Node parent = getNth(1);
Node child = parent.next
parent.next =  child.next;
```

B.
```
Node parent = getNth(1);
parent.next =  parent.next.next;
```

C. Both A and B

D. Neither A nore B

Implementing List ADT using Linked List: tradeoffs

+

- No worries about running out of space – no need for doubling
- No empty slots
- Direct access to head in O(1)

-

- Space overhead to keep links (reference variables)
- Difficult to access later elements: O(n)
  - We must always start from the head
  - We can traverse only forward

Using tail pointer

- Add at the end is improved

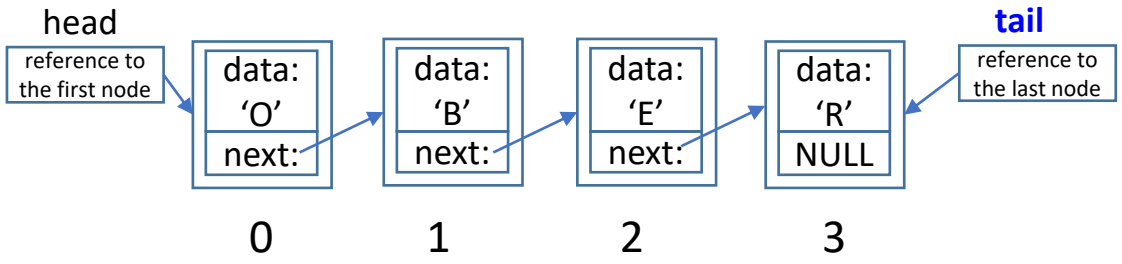
  tail.next = new Node()

- Remove from the end is not improved: we know why

  Need to update tail pointer – but we lose the tail

- Ambiguity: if head==tail – is the list empty or contains a single node?

  Ask if head==null



Doubly-linked Lists: Node

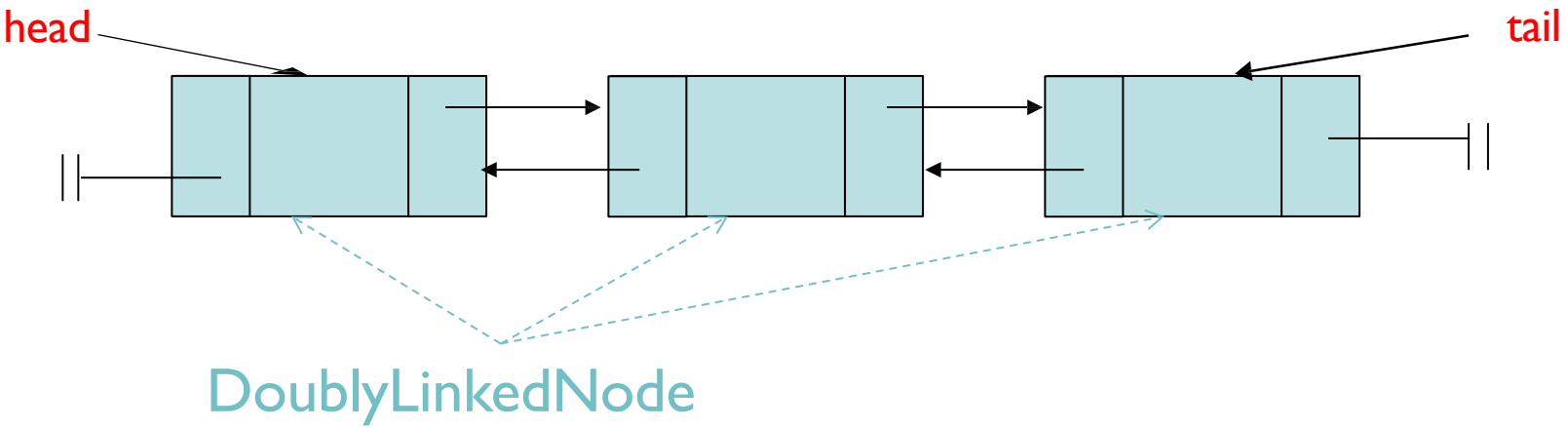
```
class Node {
    int data;
    Node next;
}
```

```
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

DoublyLinkedList:



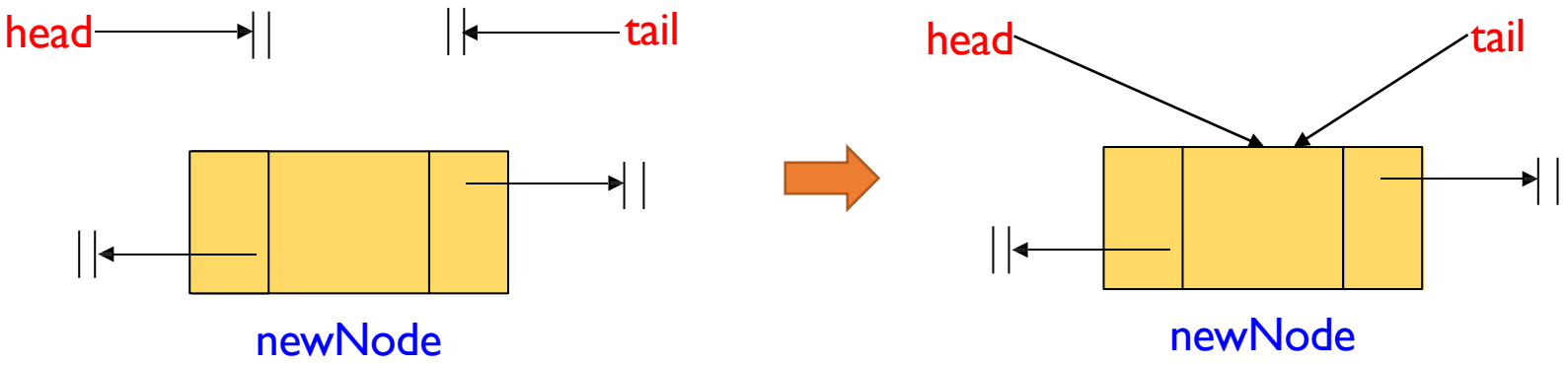
Moving heads and tails in doubly-linked lists

- 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!

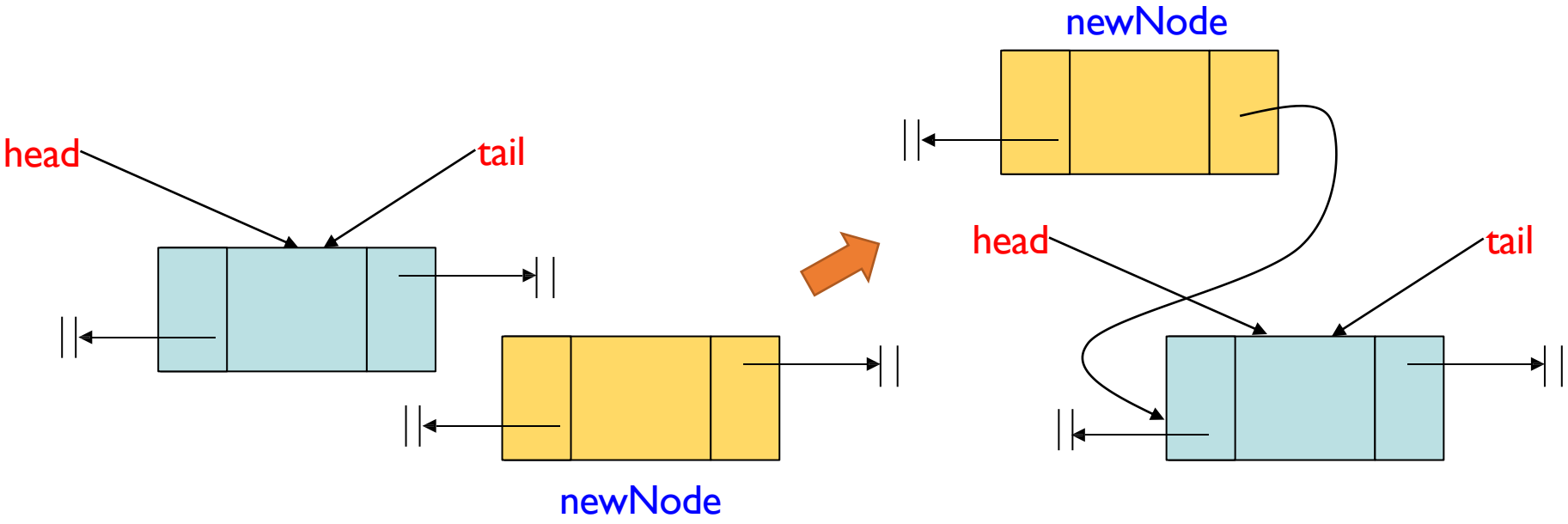
Example: add in front

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



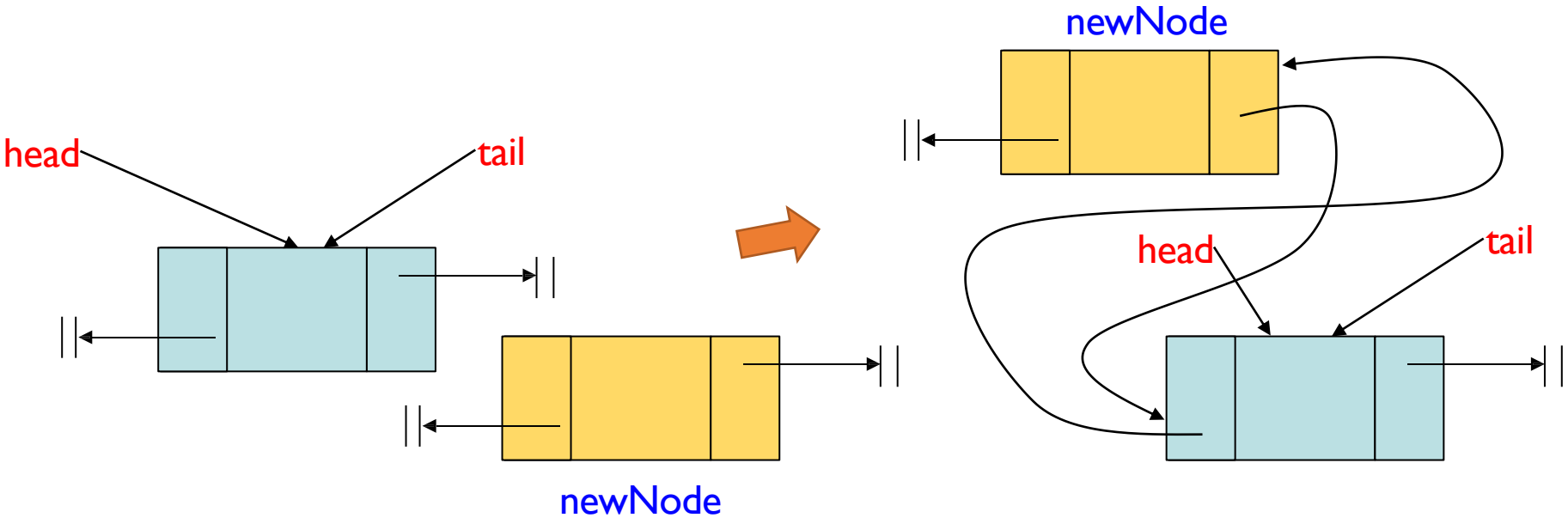
Example: add in front

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



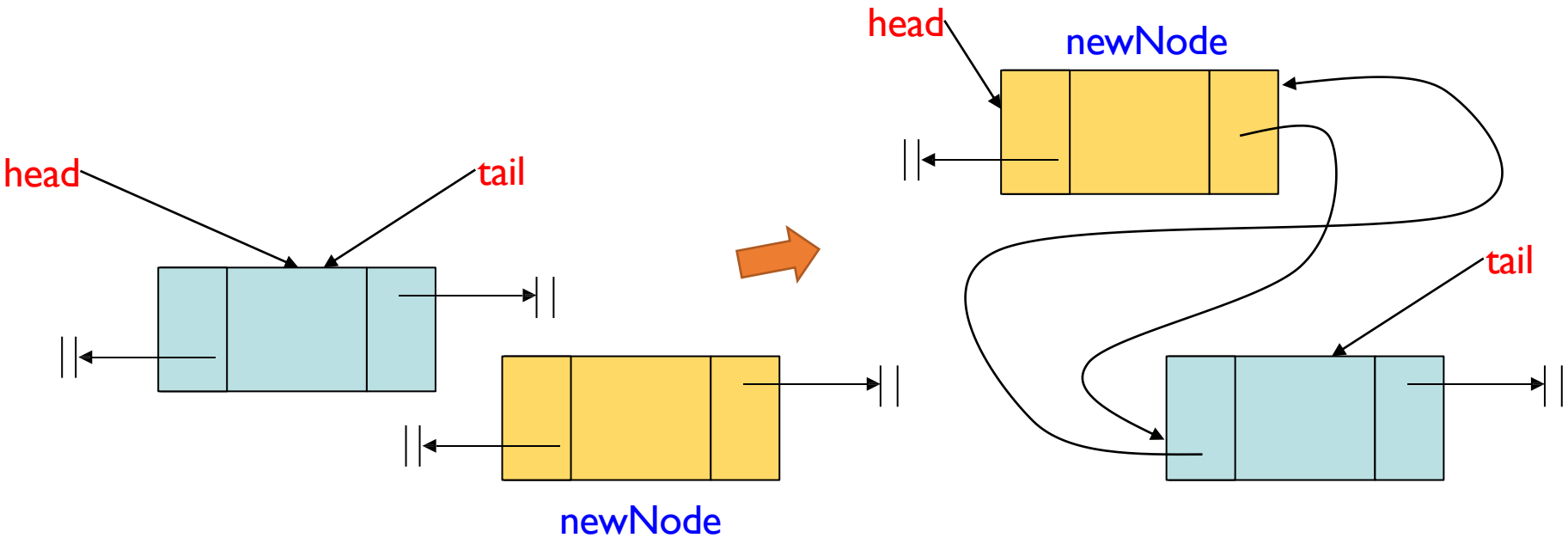
Example: add in front

```
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
```



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

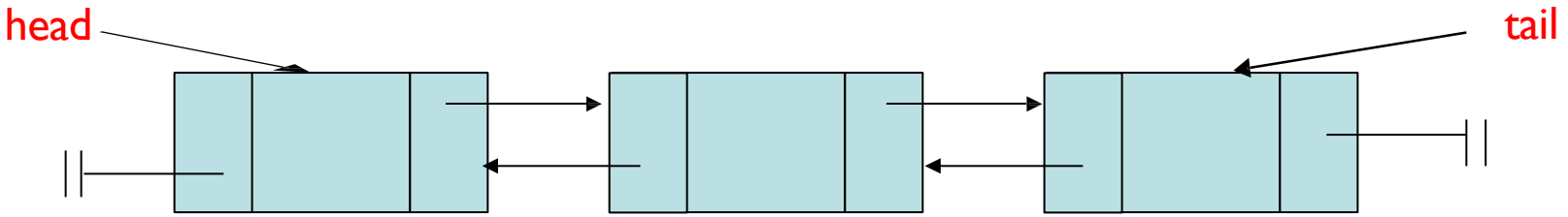
  Example: *add (int index, E element)* need to consider 4 cases:

  empty list

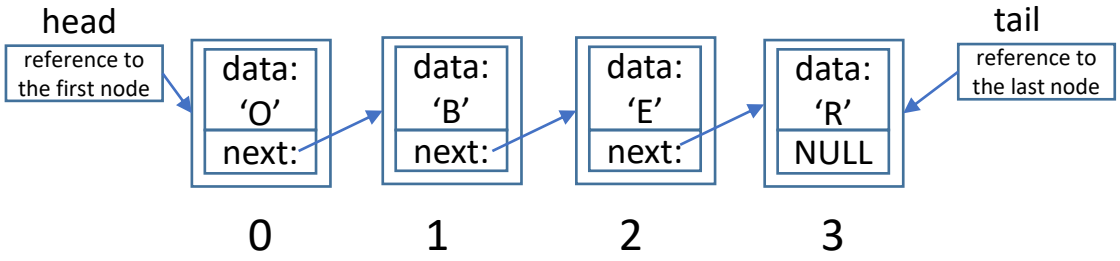
  add to front

  add to tail

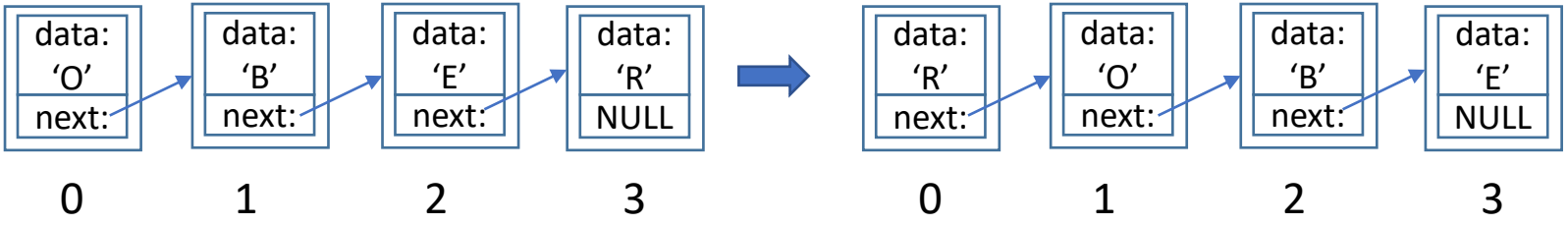
  add in middle
- × Additional space consumption (storing *previous*)



Circular lists: discussion



- Given Linked List with *tail* – how can we make a circular list?
- Do we need to keep both *head* and *tail*?
- How can we use a circular list to shift all values in the sequence by one position forward?



Java classes that implement *List* interface

- ArrayList

Resizable-array implementation of the *List* interface. Implements all optional list operations, and permits all elements, including *null*.

In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

- LinkedList

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.