Abstract Data Types (ADT)

List ADT

Lecture 14

by Marina Barsky

Abstraction

Definition

- *Abstraction* - the process of extracting only **essential property** from a real-life entity
- In CS: Problem → storage + operations

↓

Abstract Data Type (**ADT**):

result of the process of abstraction

- A specification of ***data to be stored*** together with a set of ***operations*** on that data
- ADT = Data + Operations

ADT is a mathematical concept (from *theory of concepts*)

ADT is a language-agnostic concept

- Different languages support ADT in different ways
- In C++ or Java we use *class* construct to create a new ADT

ADT includes:

- **Specification:**
  - What needs to be stored
  - What operations should be supported
- **Implementation:**
  - Data structures and algorithms used to meet the specification

ADT: Specification vs. implementation

Specification and **implementation** have to be disjoint:

- **One** specification
- **One or more** implementations
  - **Using different data structures**
  - **Using different algorithms**

Specification is expressed by defining the public variables and methods

Implementation implements these declared methods

Our First ADT: Sequence of values, **List**

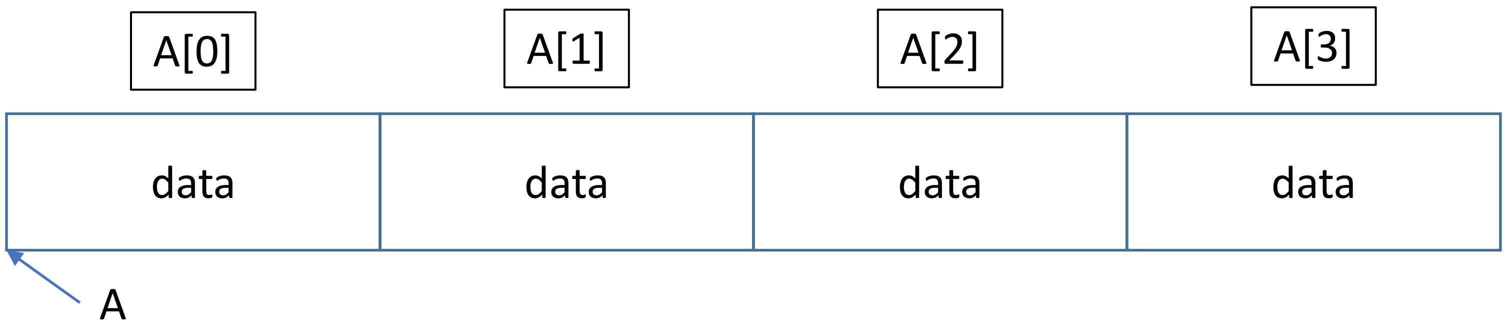
Specification for List:

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

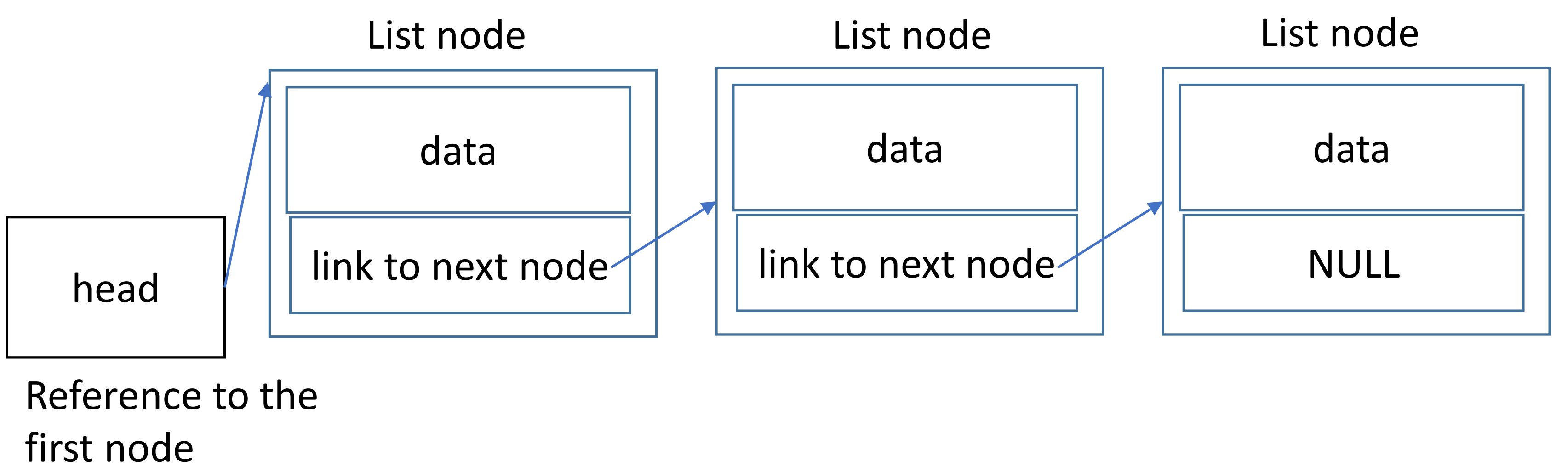
- We need to support the following operations:
  - Get element by position: `get(int index)`
  - Search element: `indexOf(E element)`
  - Add new element: `add(int index, E element)`
  - Remove element by position: `remove(i)`

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)
- Adding to the end in O(1)

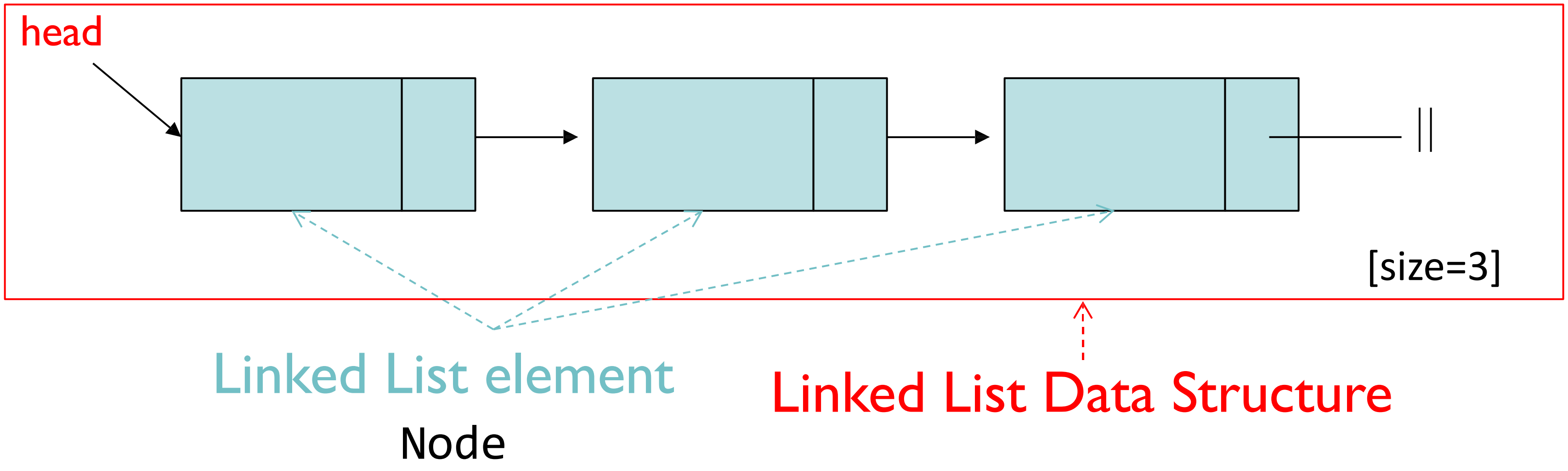
-

- 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

Alternative implementation: Linked List

Linked List contains:

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



It is easy to add in the beginning of the list

Which of the following correctly adds a new node ‘O’ to the front of the Linked List?

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

A.
```
Node rnode = new Node(‘O’);
rnode.next =  head;
```

B.
```
Node rnode = new Node(‘O’);
head.next =  rnode;
```

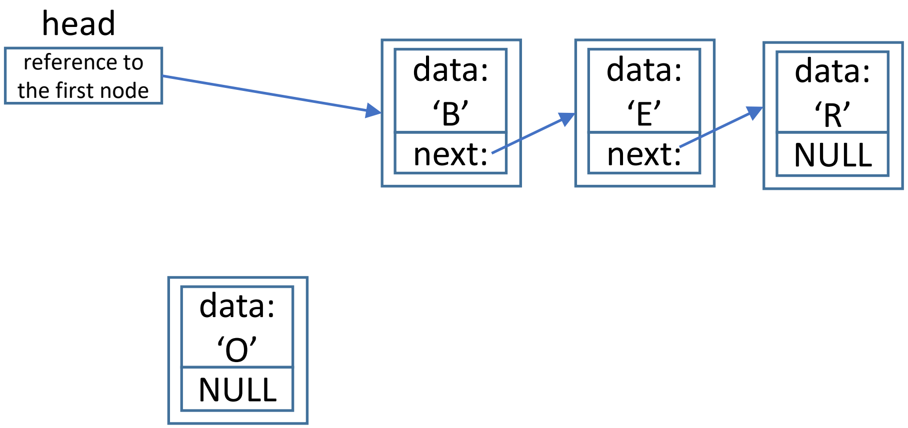
C.
```
head.data =  ‘O’;
```

D. All of the above

E. None of the above

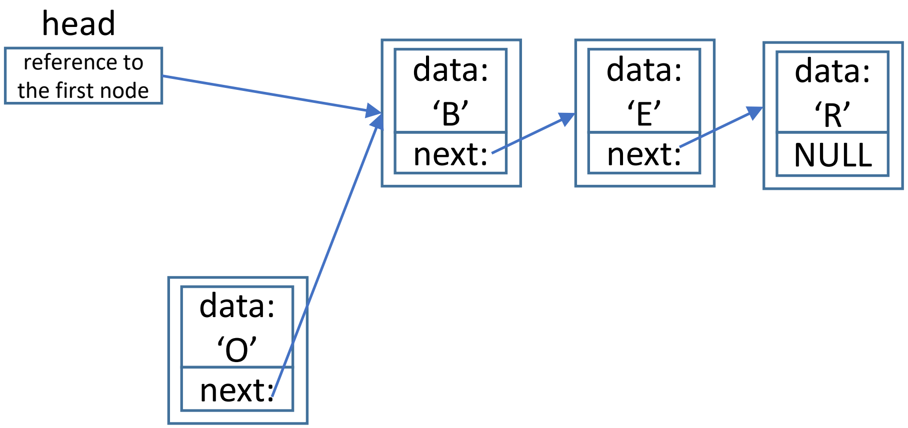
Add in front: solution 1/3

```
Node o = new Node('O');
```



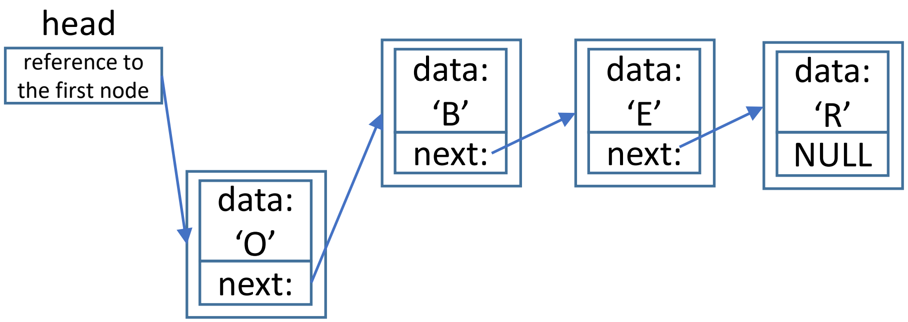
Add in front: solution 2/3

```
Node o = new Node(‘O’);
o.next = head;
```



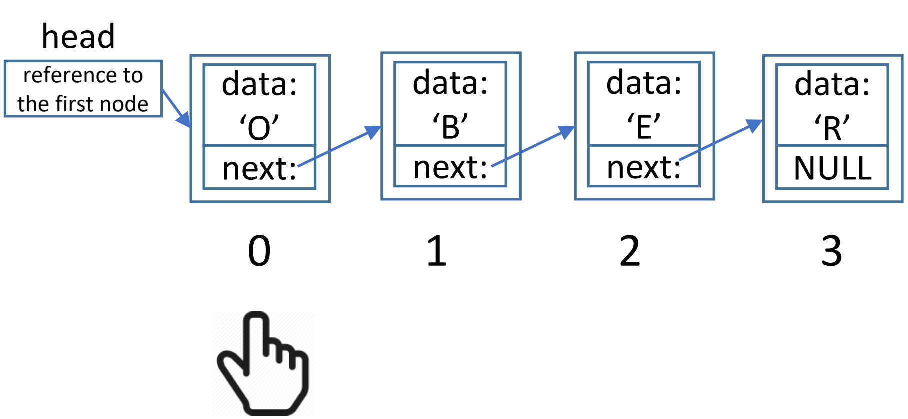
Add in front: solution 3/3

```
Node o = new Node('O');
o.next = head;
head = 0;
```



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;
}
```



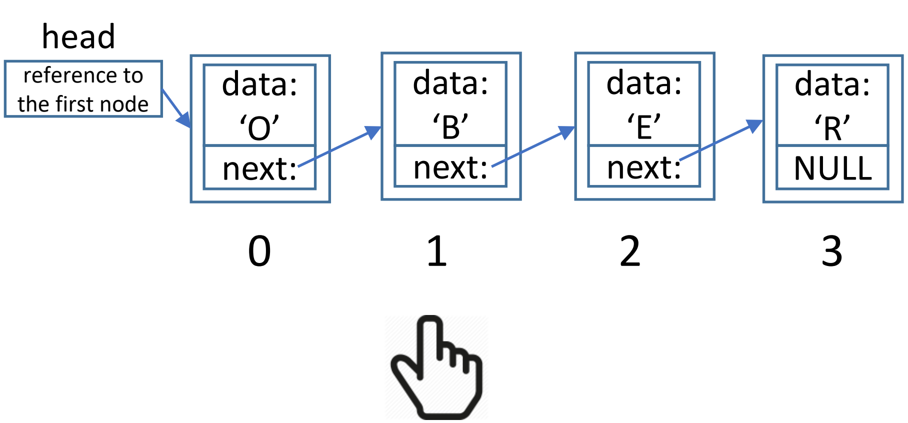
We want the node with index 2:

`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;
}
```

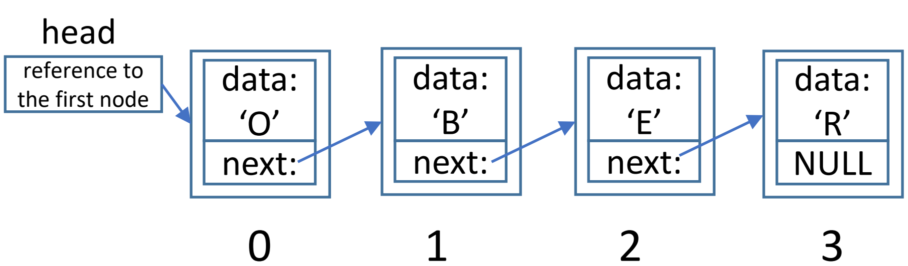


We want the node with index 2

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;
}
```



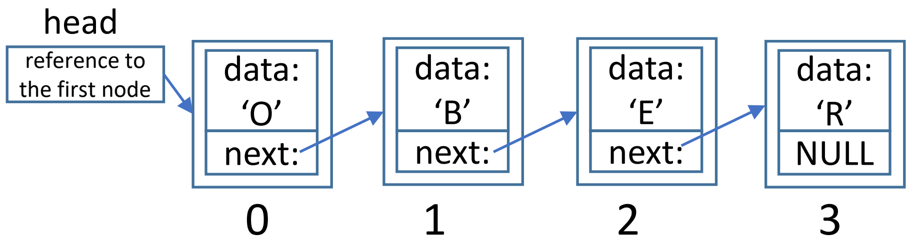
We want the node with index 2

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;
```

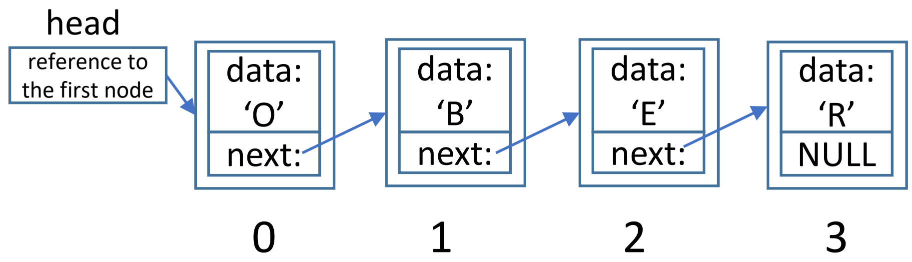
B.
```
Node mnode = new Node('M');
Node parent = getNth(0);
parent.next = mnode;
mnode.next =  parent.next;
```

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

Optimizing: tail pointer

- Add at the end is improved

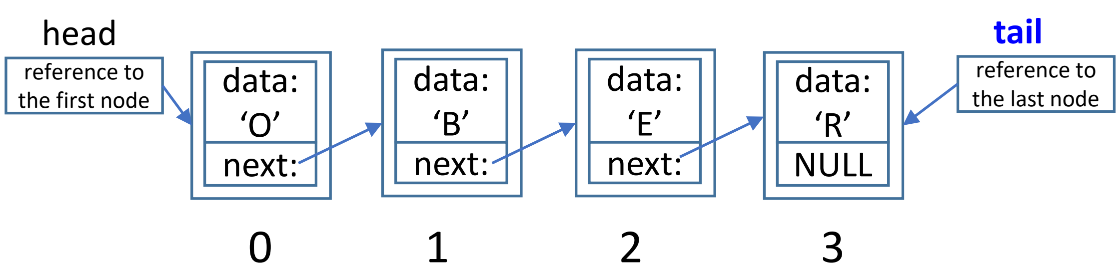
  tail.next = new Node()

- Remove from the end is not improved: why?

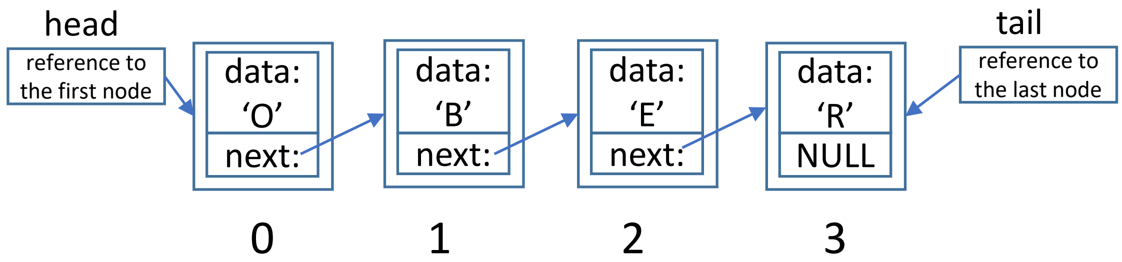
  Need to update tail pointer – but we lose the tail

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

  Ask if head==null



Circular lists



- 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?