

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu  
208 Capen Hall

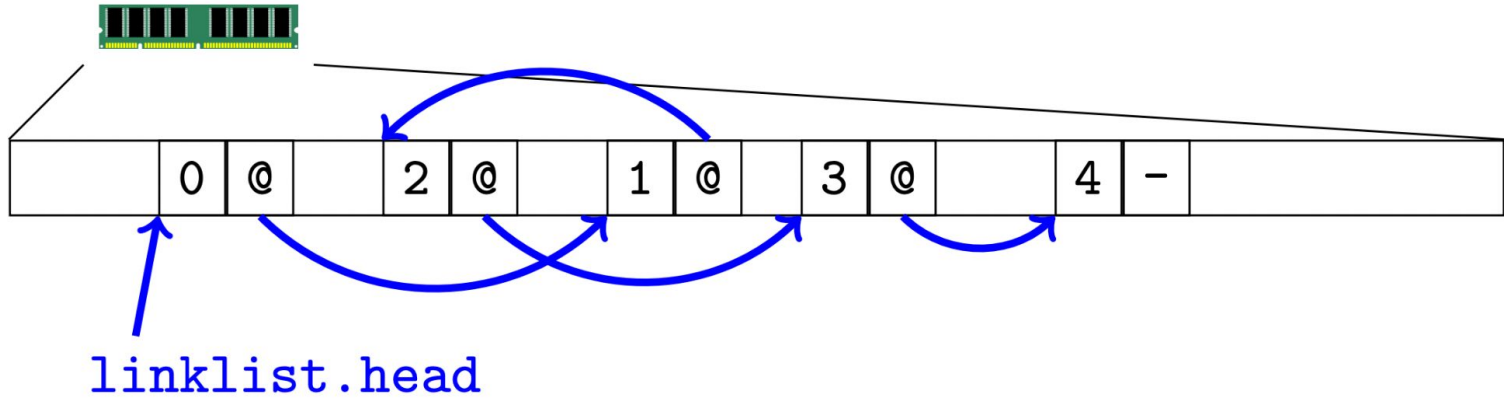
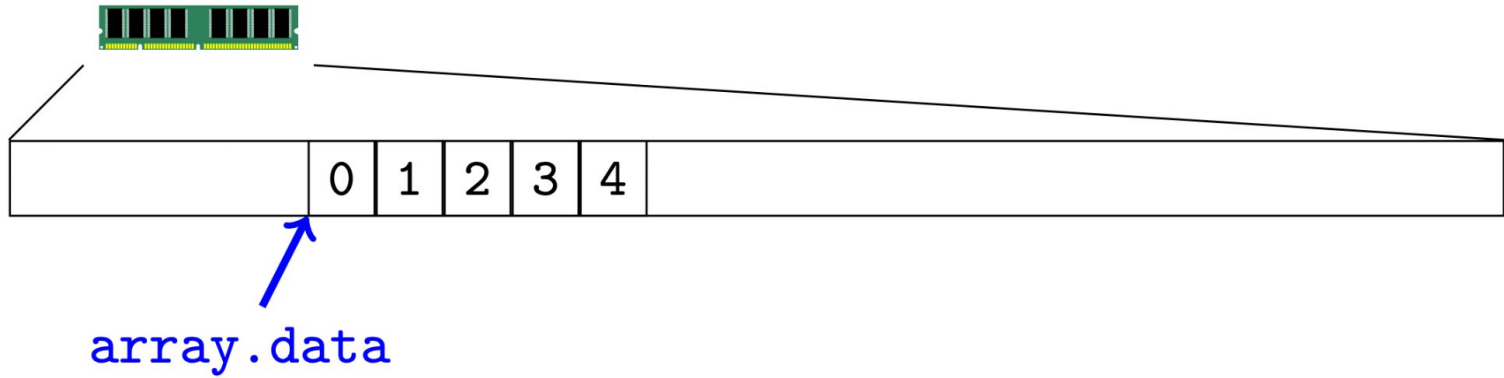
**Lec 09: List ADT and Linked Lists**

# Announcements

- PA1 Implementation due Sunday, 9/22 @ 11:59PM
  - Continue with the same repo you've been using
- WA2 will be released after the PA1 deadline, due 9/29 @ 11:59PM

# The Sequence ADT

```
1 public interface Sequence<E> {  
2     public E get(int idx);  
3     public void set(int idx, E value);  
4     public int size();  
5     public Iterator<E> iterator();  
6 }
```



## Arrays and Linked Lists in Memory

# Sequence Runtimes (so far...)

	<b>Array</b>	<b>Linked List (by index)</b>	<b>Linked List (by reference)</b>
<code>get(...)</code>	$\Theta(1)$	$\Theta(\text{idx})$ or $\mathbf{O}(n)$	$\Theta(1)$
<code>set(...)</code>	$\Theta(1)$	$\Theta(\text{idx})$ or $\mathbf{O}(n)$	$\Theta(1)$
<code>size()</code>	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

# The Sequence ADT

```
1 public interface Sequence<E> {  
2     public E get(int idx);  
3     public void set(int idx, E value);  
4     public int size();  
5     public Iterator<E> iterator();  
6 }
```

*What about adding/removing elements?*

# The List ADT

```
1 public interface List<E>
2     extends Sequence<E> { // Everything a sequence has, and...
3     /** Extend the sequence with a new element at the end */
4     public void add(E value);
5
6     /** Extend the sequence by inserting a new element */
7     public void add(int idx, E value);
8
9     /** Remove the element at a given index */
10    public void remove(int idx);
11 }
```

# Lists in Other Languages

**Java, Python:** List, list

**C++, Rust:** vector, Vec

**Scala:** Buffer

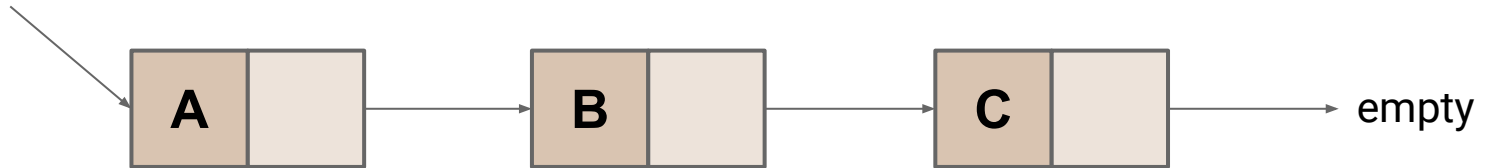
**Go:** Slice



# Linked Lists - add(idx, e)

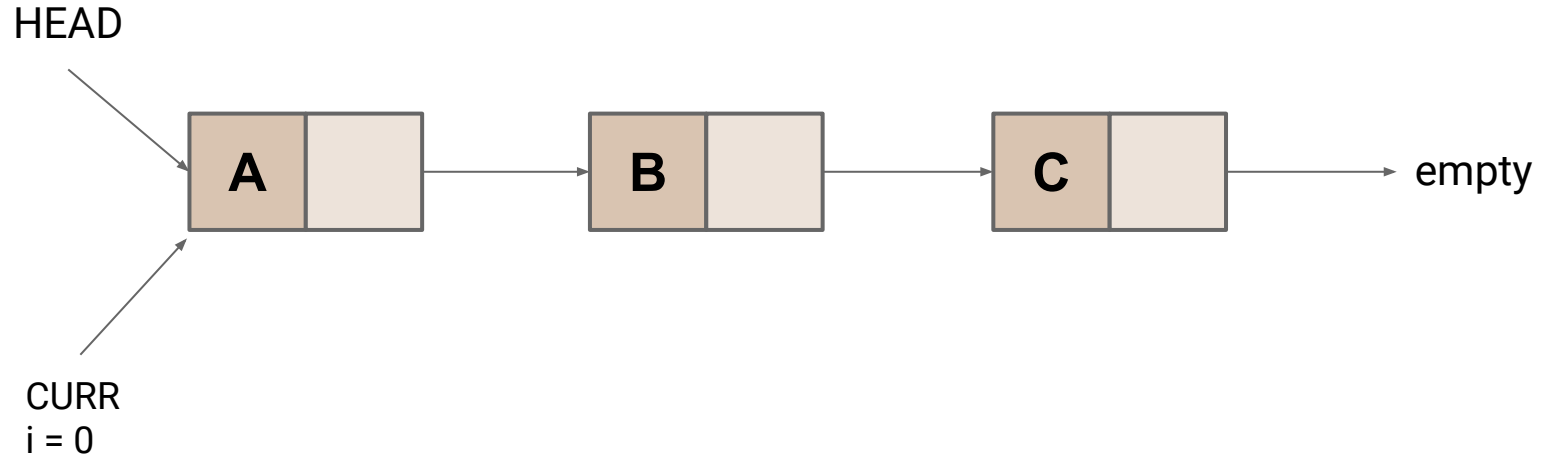
add(2, "D")

HEAD



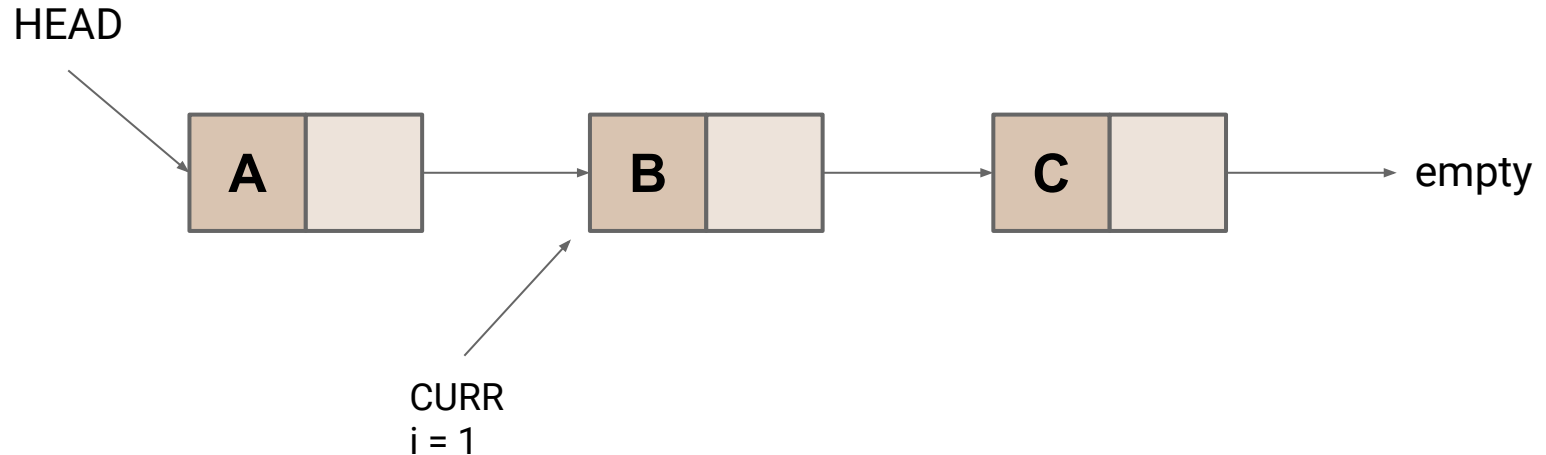
# Linked Lists - add(idx, e)

add(2, "D")



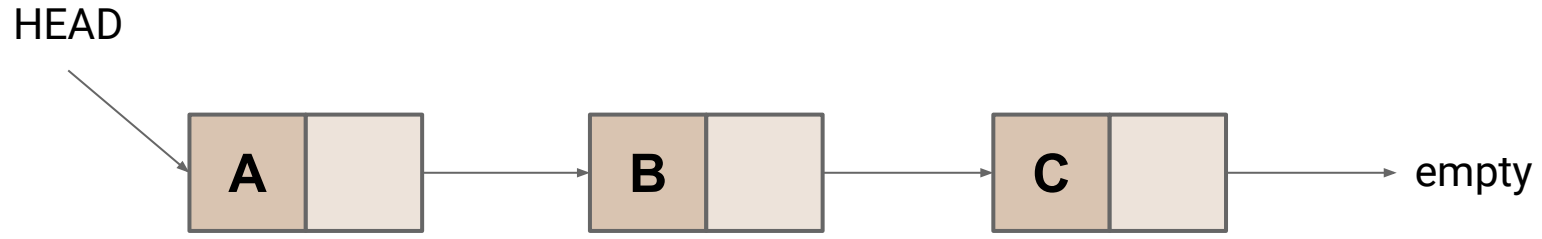
# Linked Lists - add(idx, e)

add(2, "D")



# Linked Lists - add(idx, e)

add(**2**, "D")

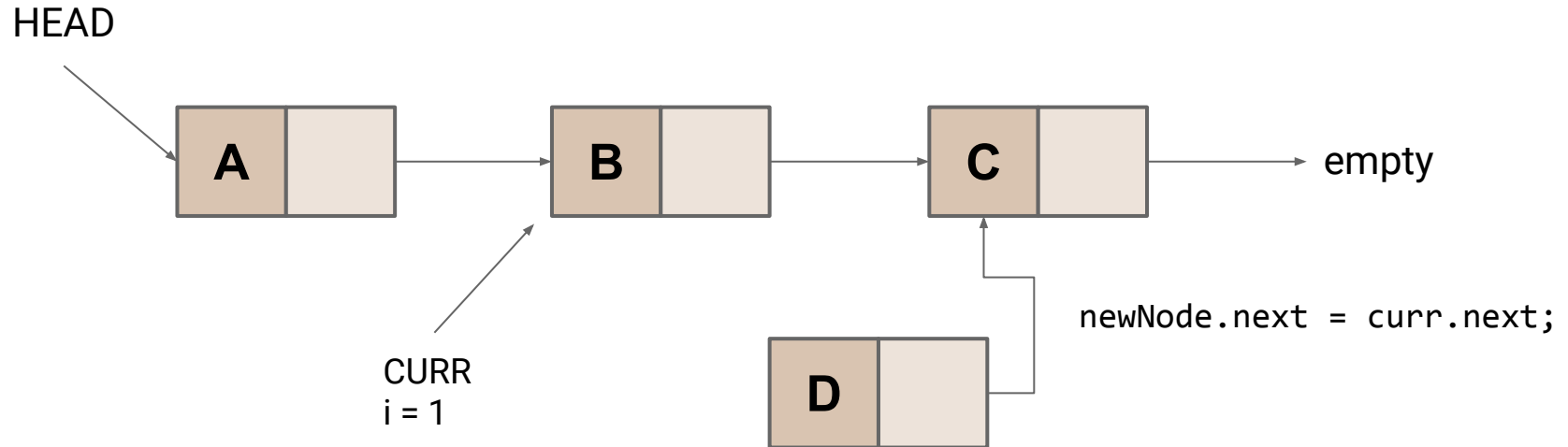


CURR  
**i = 1**

Stop when i is one less than our target

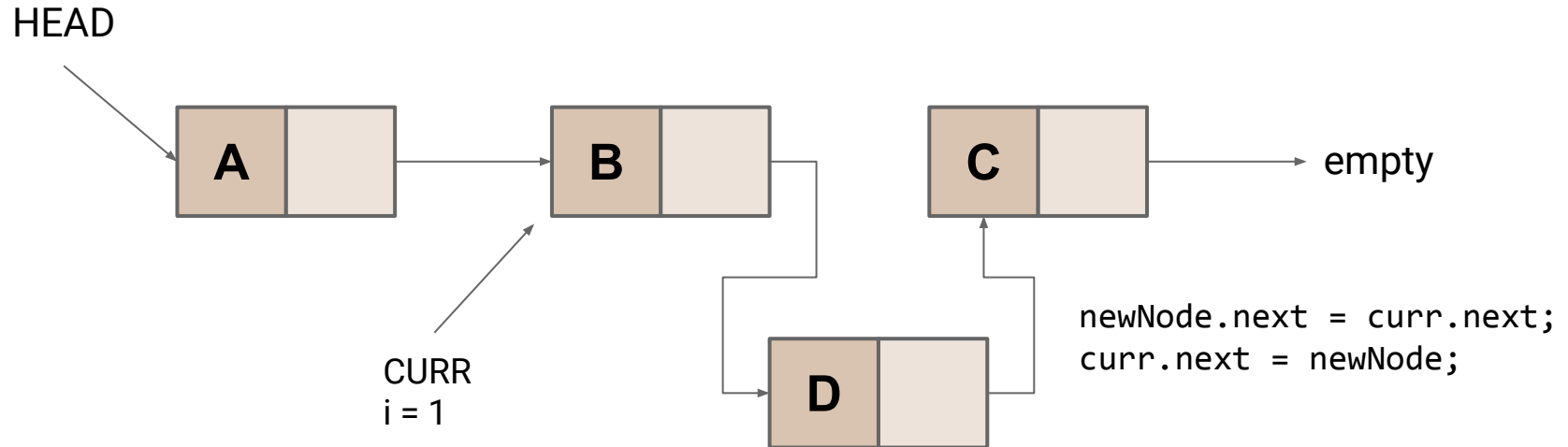
# Linked Lists - add(idx, e)

add(2, "D")



# Linked Lists - add(idx, e)

add(2, "D")



# Linked Lists - `add(idx, e)`

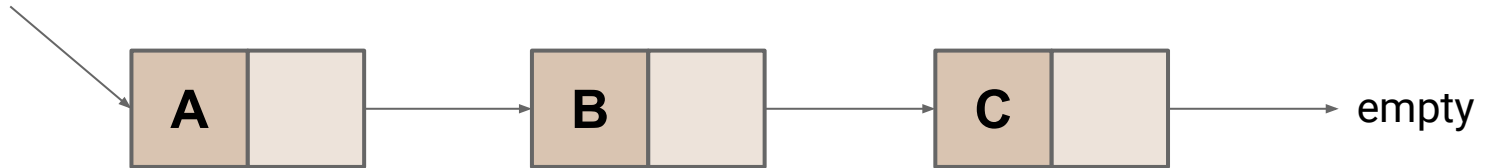
1. Find node before `idx`:  $O(n)$
2. Allocate a new node and assign its value:  $O(1)$
3. Set the new nodes `next` reference:  $O(1)$
4. Update the node before `idx`'s `next` reference:  $O(1)$

**Total:  $O(n)$**

# Linked Lists - add(e)

add("Z")

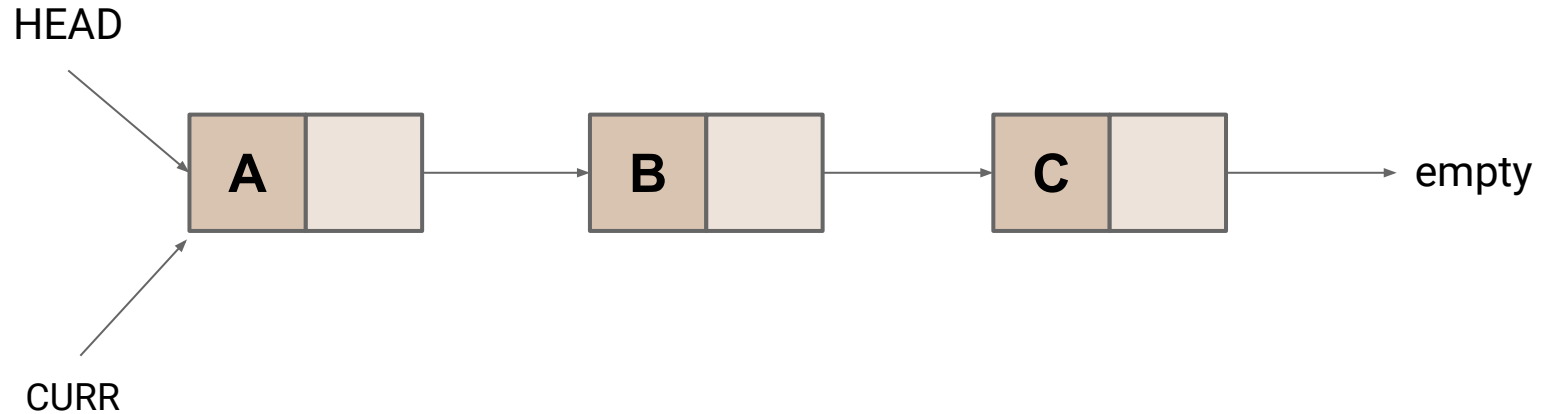
HEAD





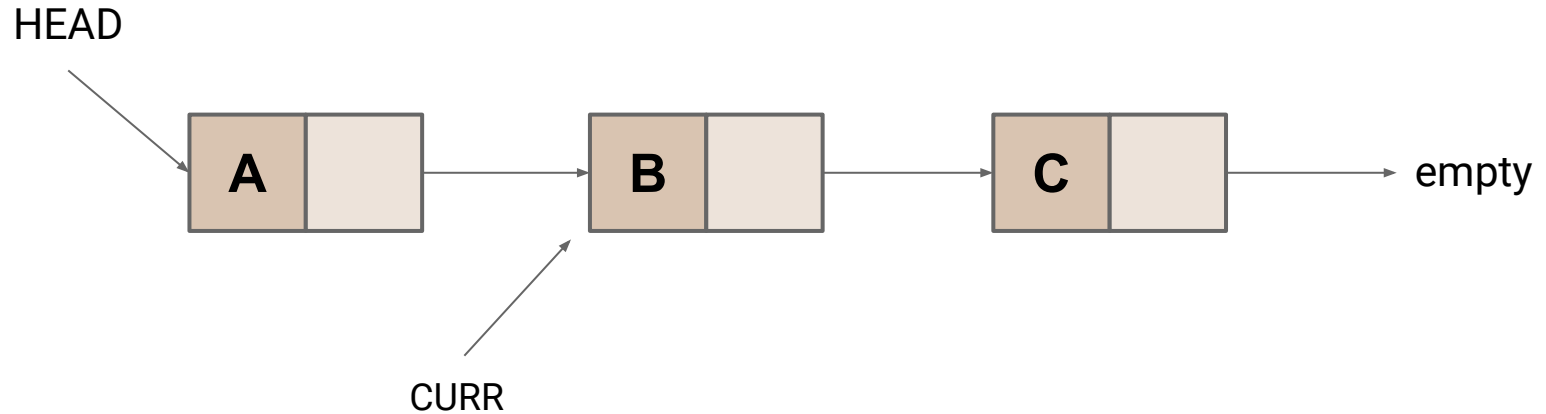
# Linked Lists - add(e)

add("Z")



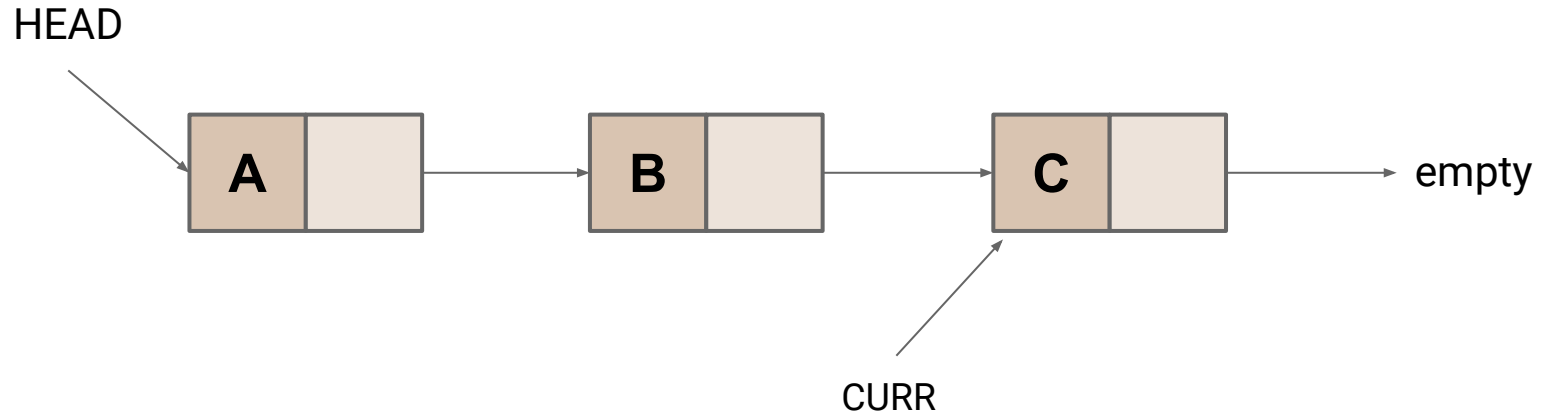
# Linked Lists - add(e)

add("Z")



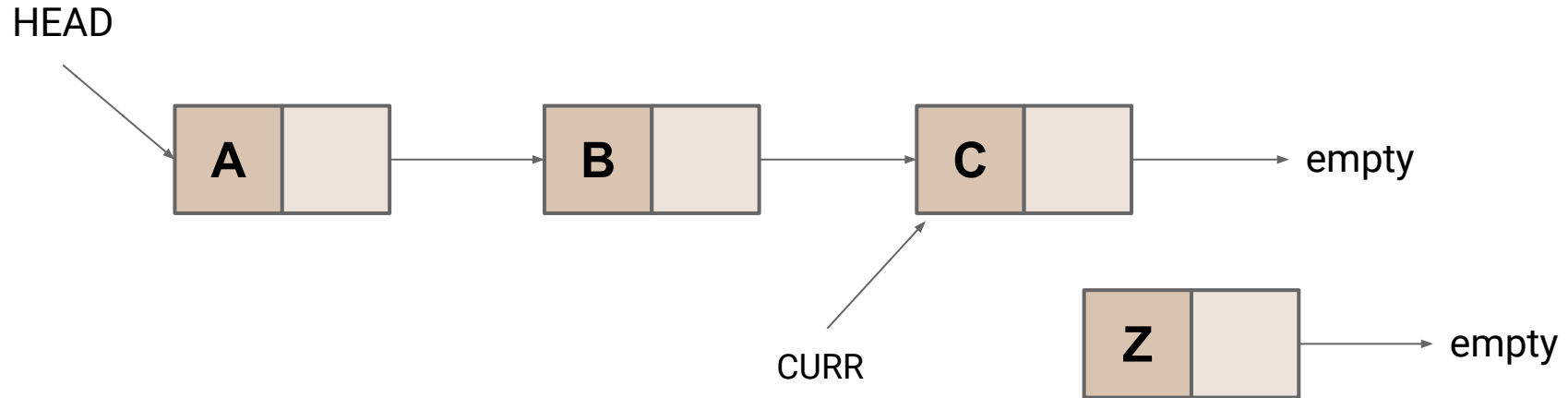
# Linked Lists - add(e)

add("Z")



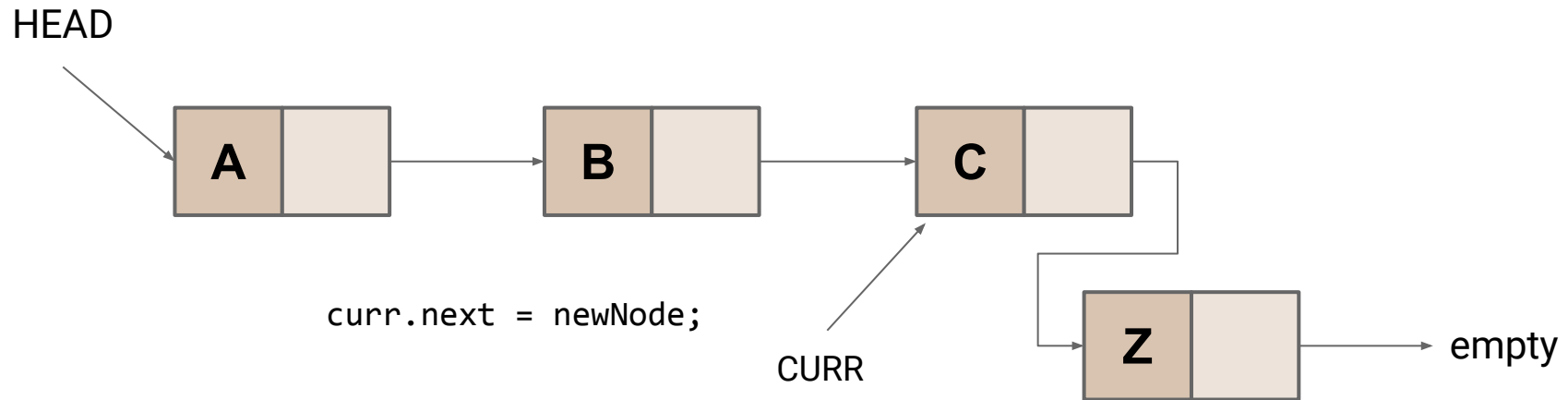
# Linked Lists - add(e)

add("Z")



# Linked Lists - add(e)

add("Z")

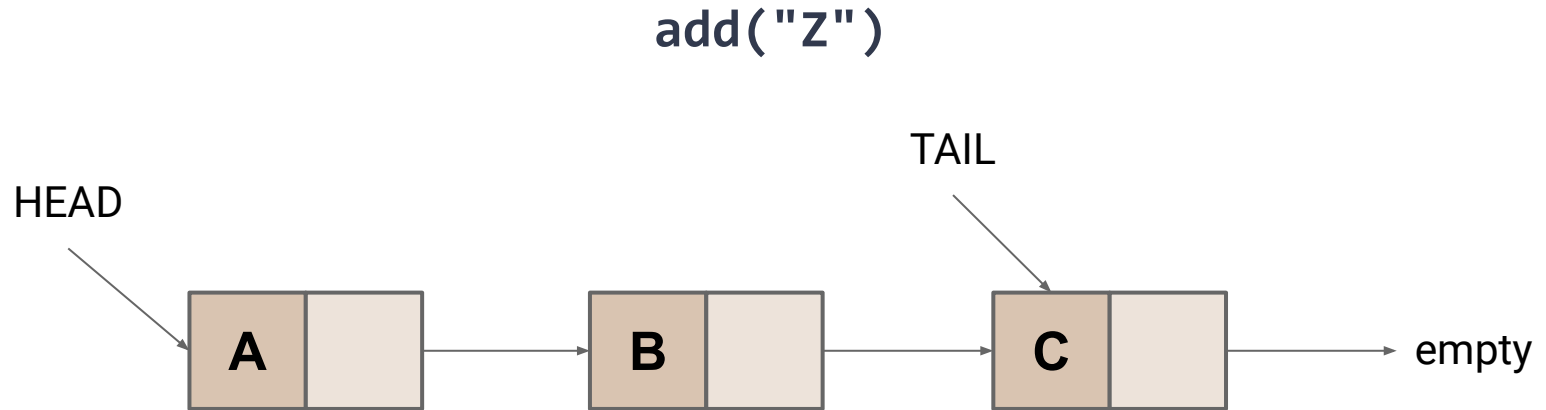


# Linked Lists - add(e)

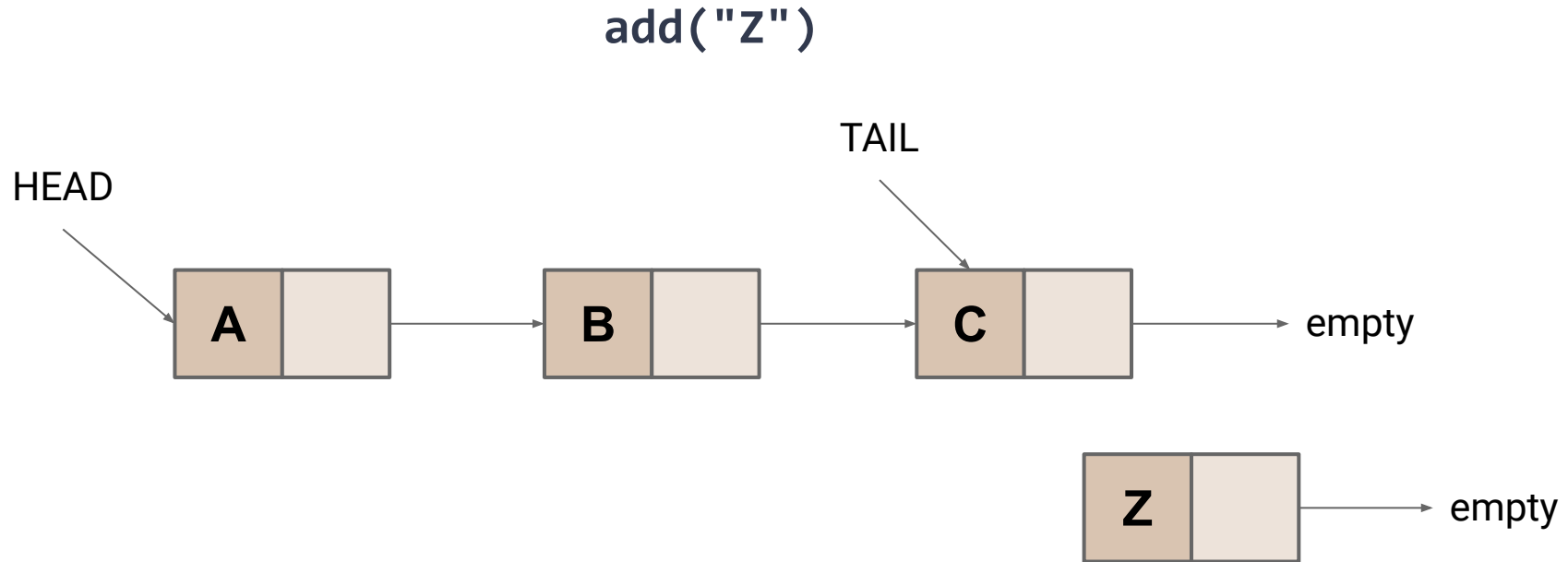
1. Find last node:  $O(n)$
2. Allocate a new node and assign its value:  $O(1)$
3. Update the last nodes `next` reference:  $O(1)$

**Total:  $O(n)$**

# Linked Lists (w/ref to tail) - add(e)

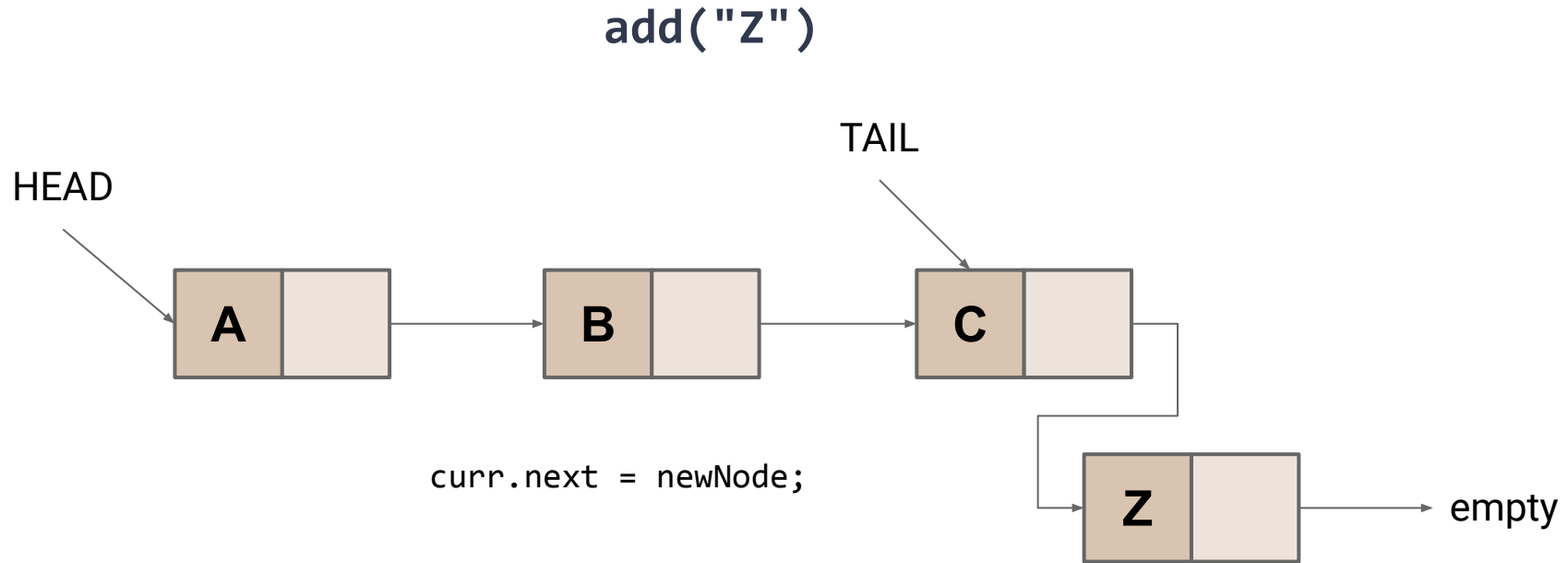


# Linked Lists (w/ref to tail) - add(e)



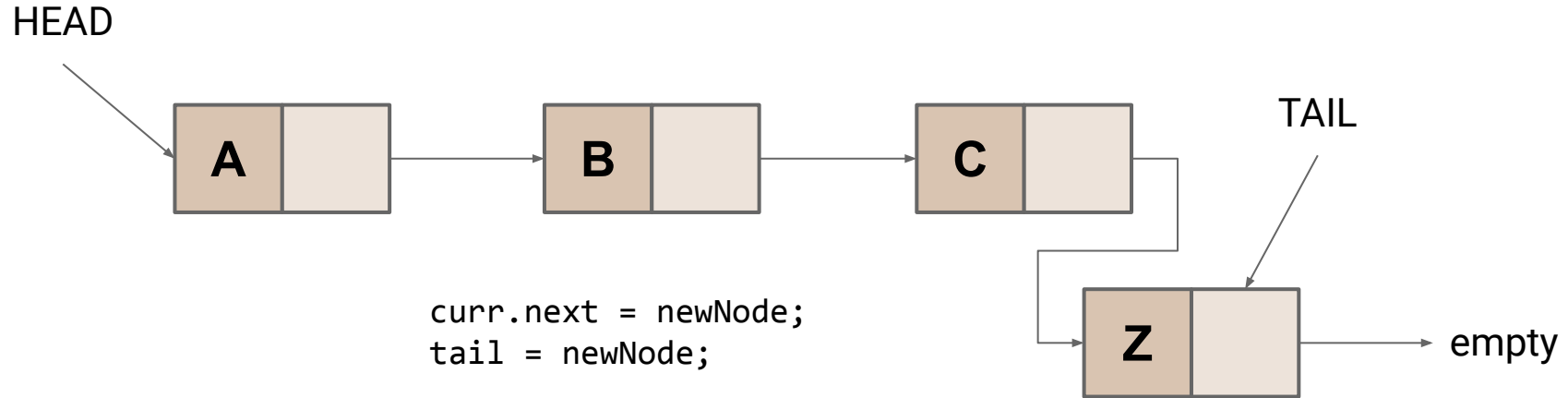


# Linked Lists (w/ref to tail) - add(e)



# Linked Lists (w/ref to tail) - add(e)

add("Z")



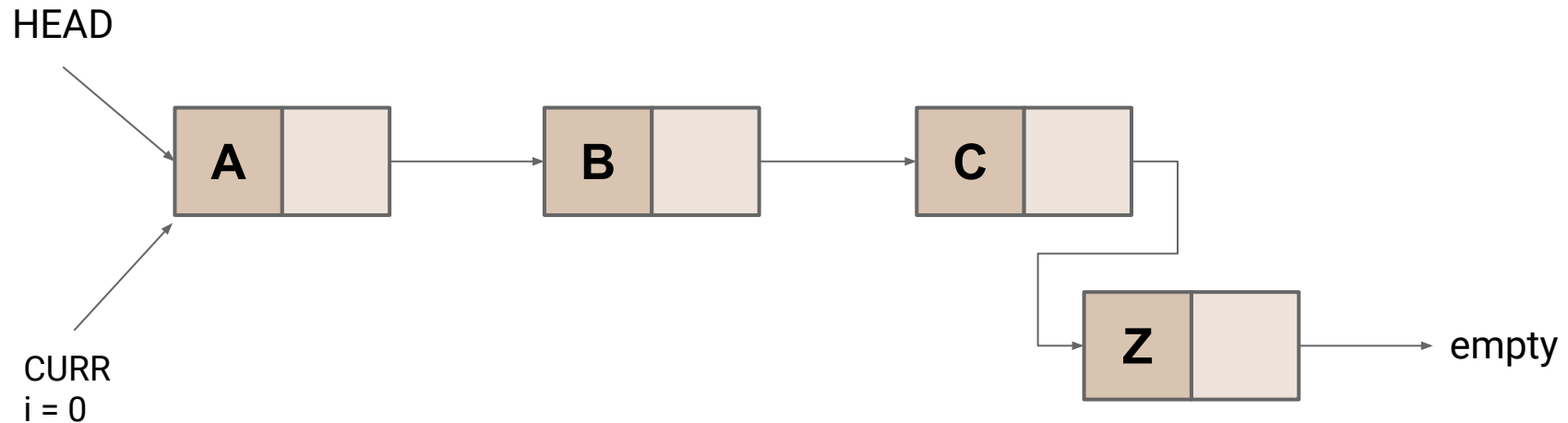
# Linked Lists (w/ref to tail) - add(e)

1. Find last node:  $O(1)$
2. Allocate a new node and assign its value:  $O(1)$
3. Update the last nodes `next` reference:  $O(1)$
4. Update the TAIL reference:  $O(1)$

**Total:  $O(1)$**

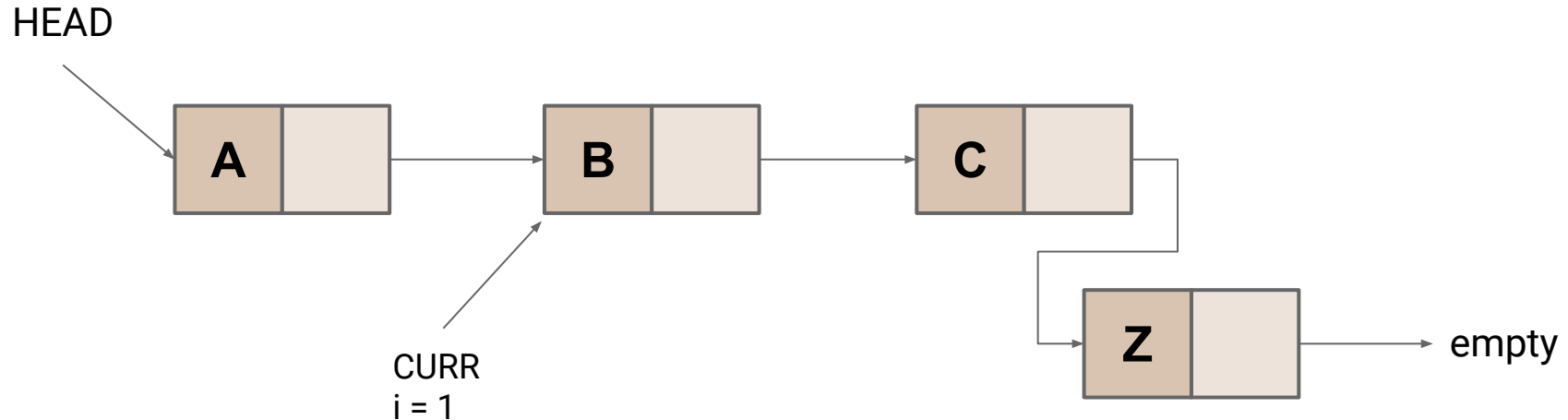
# Linked Lists - remove(idx)

remove(2)



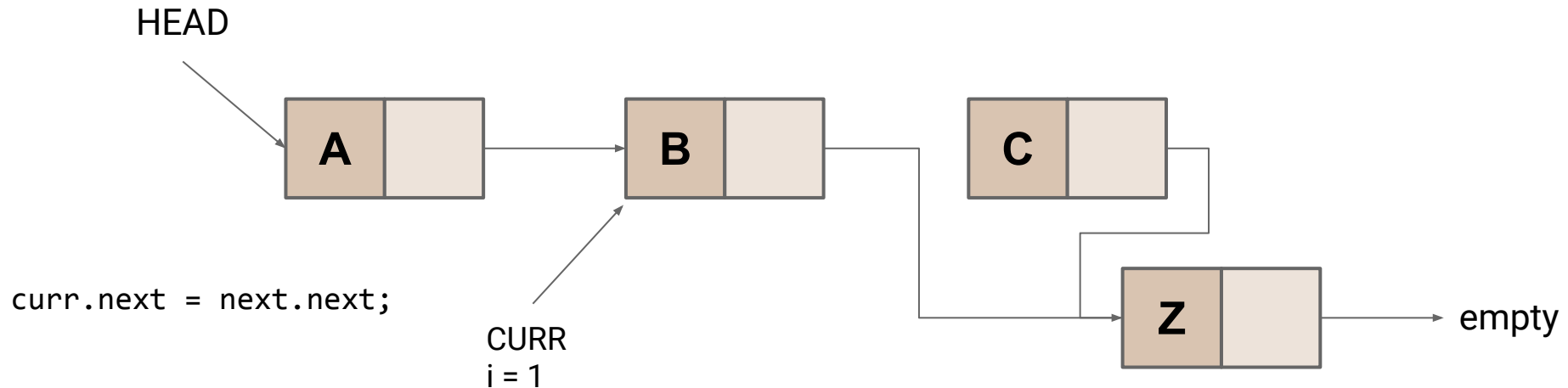
# Linked Lists - remove(idx)

remove(2)



# Linked Lists - remove(idx)

remove(2)



# Linked Lists - `remove(idx)`

1. Find node before `idx`:  $O(n)$
2. Update the node before `idx`'s `next` reference:  $O(1)$
3. Allow the old node to be reclaimed:  $O(1)$

**Total:**  $O(n)$

# Linked Lists

What is the expensive part of all of these operations?



# Linked Lists

What is the expensive part of all of these operations?

**Iterating to the correct index**

# Enumeration Example

```
1 int sumList(List<Integer> list){
2     int rslt = 0;
3     for(int i = 0; i < list.length; i++){
4         int temp = list.get(i);
5         rslt += temp;
6     }
7     return rslt;
8 }
```

What is the complexity of this code?

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2    $\Theta(1)$   
3   for(int i = 0; i < list.length; i++){  
4     int temp = list.get(i);  
5      $\Theta(1)$   
6   }  
7    $\Theta(1)$   
8 }
```

What is the complexity of this code?

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2   Θ(1)  
3   for(int i = 0; i < list.length; i++){  
4     int temp = list.get(i);  
5     Θ(1)  
6   }  
7   Θ(1)  
8 }
```

What is the complexity of this code?

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2      $\Theta(1)$   
3     for(int i = 0; i < list.length; i++){  
4          $\Theta(i)$   
5          $\Theta(1)$   
6     }  
7      $\Theta(1)$   
8 }
```

What is the complexity of this code?

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2      $\Theta(1)$   
3      $\Theta(n^2)$   
4      $\Theta(1)$   
5 }
```

What is the complexity of this code?  $\Theta(n^2)$

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2      $\Theta(1)$   
3      $\Theta(n^2)$   
4      $\Theta(1)$   
5 }
```

What is the complexity of this code?  $\Theta(n^2)$

Why is it so expensive?

# Enumeration Example

```
1 int sumList(List<Integer> list){  
2      $\Theta(1)$   
3      $\Theta(n^2)$   
4      $\Theta(1)$   
5 }
```

What is the complexity of this code?  $\Theta(n^2)$

Why is it so expensive? **We start from index 0 every time!**



# Enumeration Example #2

```
1 int sumLinkedList(LinkedList<Integer> list){
2     int rslt = 0;
3     Optional<LinkedListNode> n = list.headNode;
4     while (n.isPresent()){
5         int temp = n.get().value;
6         rslt += temp;
7         n = n.get().next;
8     }
9     return rslt;
10 }
```

# Enumeration Example #2

```
1 int sumLinkedList(LinkedList<Integer> list){
2     int rslt = 0;
3     Optional<LinkedListNode> n = list.headNode;
4     while (n.isPresent()){
5         int temp = n.get().value;
6         rslt += temp;
7         n = n.get().next;      Now this is all constant time
8     }
9     return rslt;
10 }
```

# Enumeration Example #2

```
1 int sumLinkedList(LinkedList<Integer> list){  
2      $\Theta(1)$   
3     while (n.isPresent()){  
4          $\Theta(1)$   
5     }  
6      $\Theta(1)$   
7 }
```

# Enumeration Example #2

```
1 int sumLinkedList(LinkedList<Integer> list){  
2      $\Theta(1)$   
3      $\Theta(n)$   
4      $\Theta(1)$   
5 }
```

Total complexity:  $\Theta(n)$

# Enumeration

## Problem:

- This code is specialized for `LinkedList`
- It will not work for other types of `List` (ie `ArrayList`)

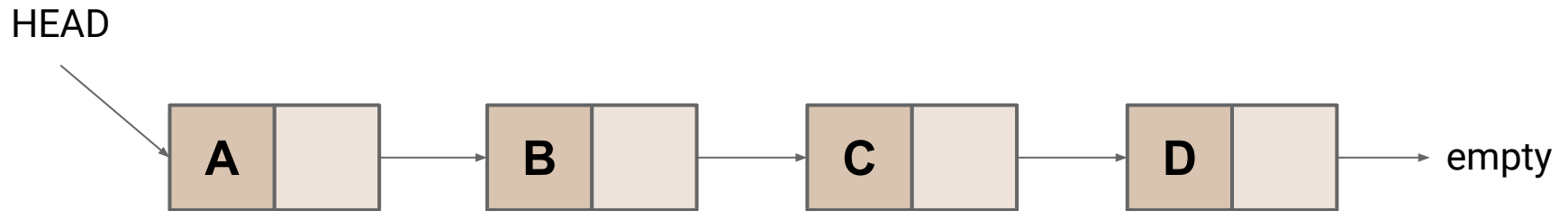
## How can we get code that is both fast and general?

- Must be able to represent a reference to the `idx`'th element of a `List`

# ListIterator

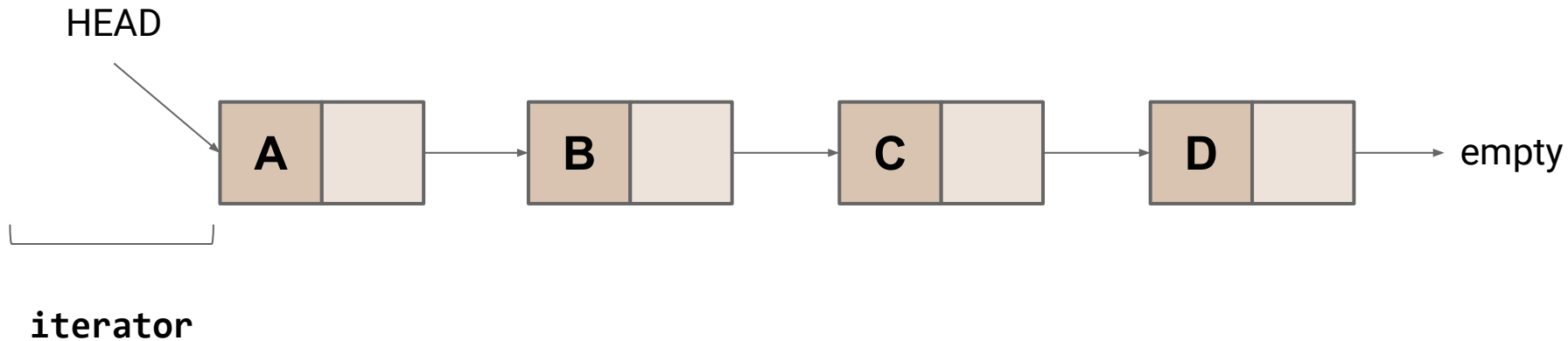
```
1 public interface ListIterator<E> {  
2     public boolean hasNext();  
3     public E next();  
4     public void add(E value);  
5     public void set(E value);  
6     public void remove();  
7 }
```

# Linked List Iterator



# Linked List Iterator

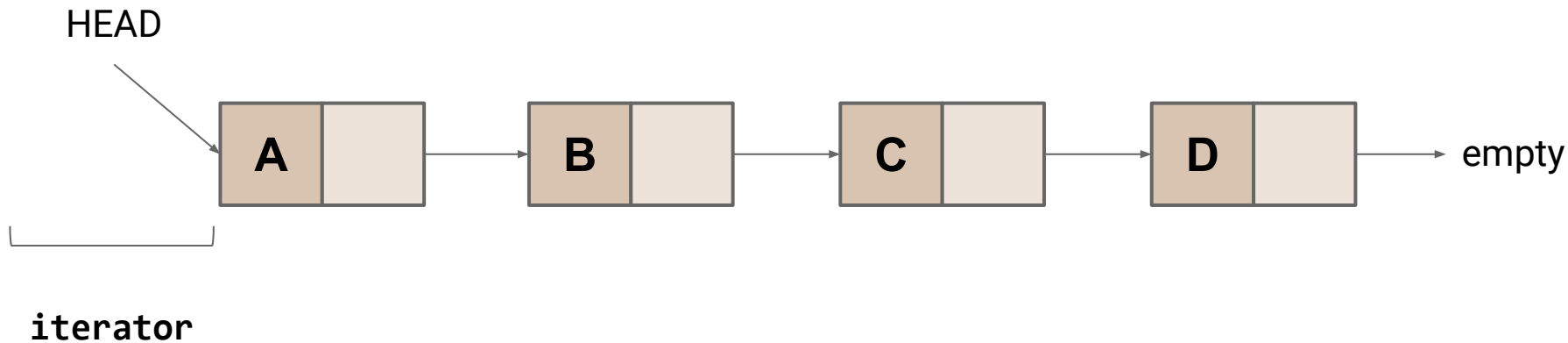
```
iterator = list.iterator()
```





# Linked List Iterator

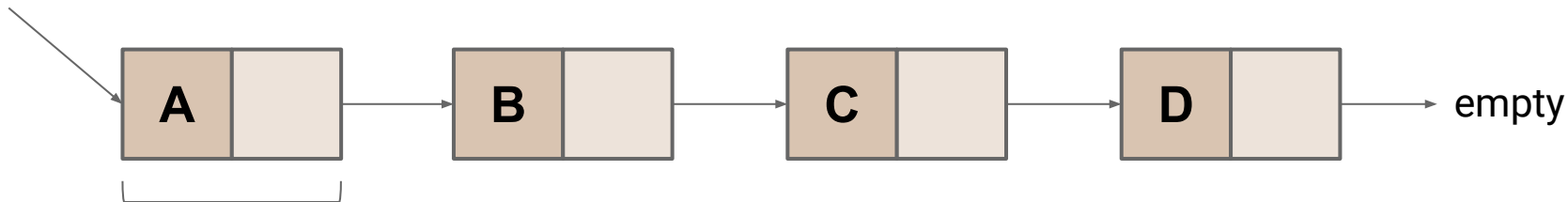
`iterator.hasNext()` → `true`



# Linked List Iterator

`iterator.next()` → A

HEAD

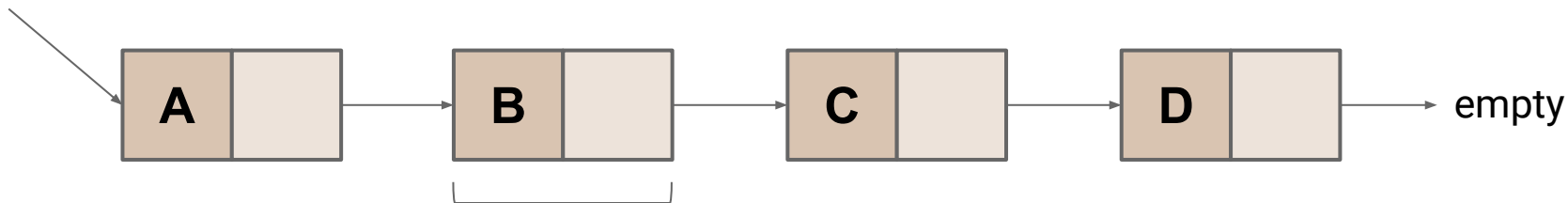


iterator

# Linked List Iterator

`iterator.next()` → B

HEAD

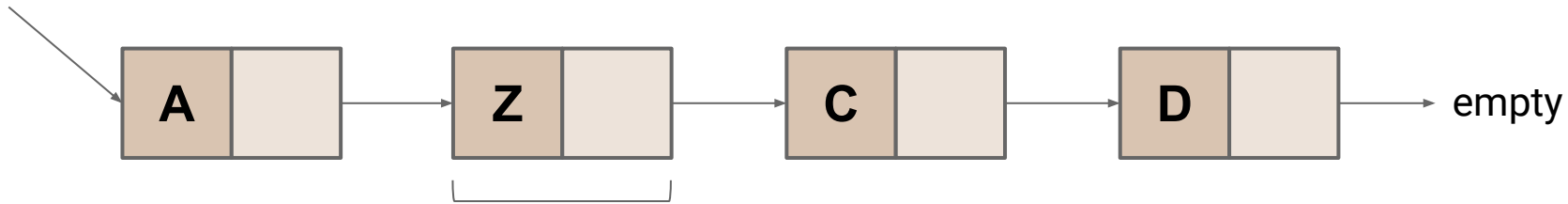


iterator

# Linked List Iterator

`iterator.set(Z)`

HEAD

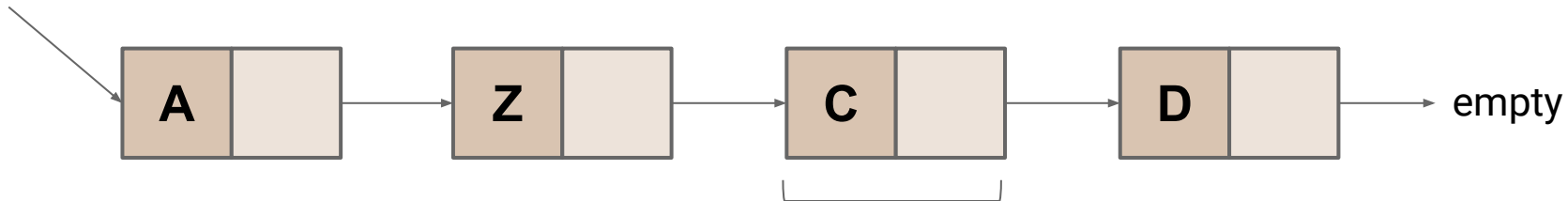


iterator

# Linked List Iterator

`iterator.next() → C`

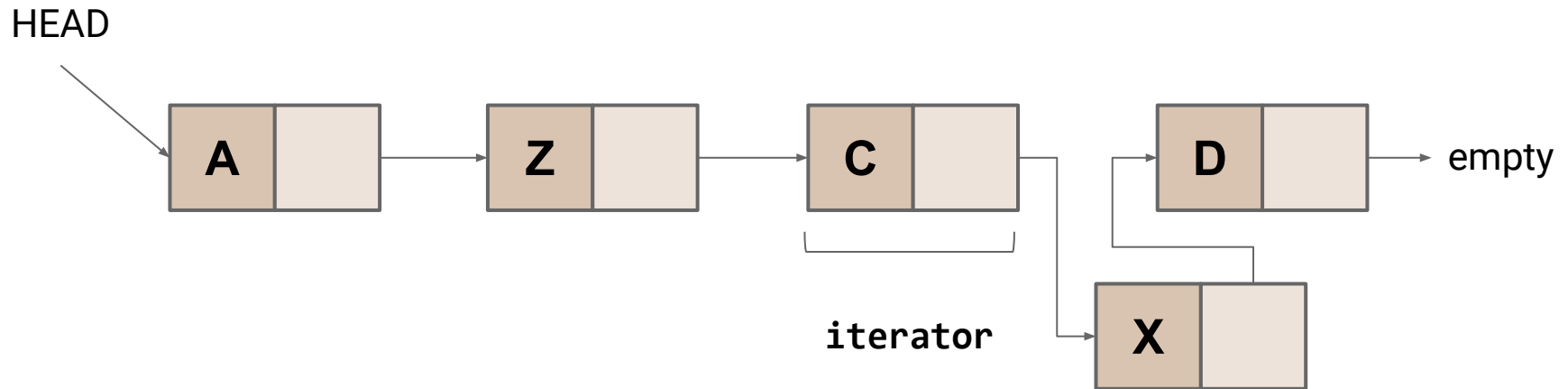
HEAD



iterator

# Linked List Iterator

`iterator.add(X)`



# Linked List Iterator

```
1 public class LinkedListIterator<E>
2     extends ListIterator<E> {
3     LinkedList<E> list;
4     Optional<LinkedListNode<E>> before = Optional.empty();
5     Optional<LinkedListNode<E>> after = Optional.of(list.head);
6     /* ... */
7 }
```

# Linked List Iterator

**boolean hasNext()**

If **after.isPresent()**, return true

**0(1)**

**T next()**

If **after.isPresent()**, advance **before** and **after**, and return the value of **before**

**0(1)**

**void set(T value)**

Update **before.value**

**0(1)**



# Linked List Iterator

**void add(T value)**

Create a new node, update its `next` and `before.next`  **$O(1)$**

**void remove(T value)**

Set `before.next` to `after.next` and update `after`  **$O(1)$**

# Enumeration using ListIterator

```
1 public void int sumUpList(List<Integer> list) {  
2     int total = 0;  
3     ListIterator<Integer> iterator = list.iterator();  
4     while(iterator.hasNext()) {  
5         int value = iterator.next();  
6         total += value;  
7     }  
8     return total;  
9 }
```

# Enumeration using ListIterator

```
1 public void int sumUpList(List<Integer> list) {  
2     int total = 0;  
3     ListIterator<Integer> iterator = list.iterator();  
4     while(iterator.hasNext()) {  
5         int value = iterator.next();  
6         total += value;  
7     }  
8     return total;  
9 }
```

Generalized to work with any kind of list!

# Enumeration using ListIterator

```
1 public void int sumUpList(List<Integer> list) {  
2     int total = 0;  
3     ListIterator<Integer> iterator = list.iterator();  
4     while(iterator.hasNext()) {  
5         int value = iterator.next();  
6         total += value;  
7     }                               Loop body only contains  $\Theta(1)$  operations  
8     return total;  
9 }
```

# Enumeration using ListIterator

```
1 public void int sumUpList(List<Integer> list) {  
2     int total = 0;  
3     ListIterator<Integer> iterator = list.iterator();  
4     while(iterator.hasNext()) {  
5         int value = iterator.next();  
6         total += value;  
7     }  
8     return total;  
9 }
```

Total Complexity:  $\Theta(n)$

# ArrayLists

**Question:** How can we implement `add(e)` on an `ArrayList`?

# ArrayLists

**Question:** How can we implement `add(e)` on an `ArrayList`?

**Problem:** Arrays have a fixed size!

# ArrayLists - Attempt #1

1. Allocate a new array of size  $n + 1$   $O(1)$
2. Copy all  $n$  elements to the new array  $O(n)$
3. Insert the new item at position  $n$   $O(1)$

**Total:  $O(n)$**



# ArrayLists - Attempt #1

1. Allocate a new array of size  $n + 1$   $O(1)$
2. Copy all  $n$  elements to the new array  $O(n)$
3. Insert the new item at position  $n$   $O(1)$

**Total:  $O(n)$**

Can we do better? next class...