CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 10
Linked Lists and Iterators
Textbook Ch. 7

Announcements

- Starting next week, my Tuesday offices hours are moving to Thursday

Recap of Amortized Runtime

- Let's say we want to insert *n* elements into an **`ArrayBuffer`** that has an initial size of **`x`**…
- Some insertions will be very cheap ($\Theta(1)$) because we don't have to resize the underlying **`Array`**
- Some insertions will be expensive, because before we insert, we will have to create a copy of the underlying **`Array`**
- If we double the size each time, then an expensive insert that costs $\Theta(2^j \mathbf{x})$ also buys us the luxury of the next $(2^j-1)\mathbf{x}$ insertions being cheap (only costing $\Theta(1)$).

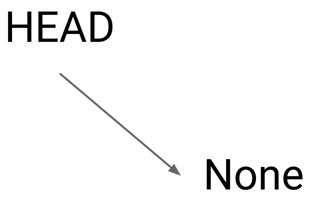
Summary of `Seq` So Far

`Array[T]`
Pros: $O(1)$ `apply`, `update`
Cons: $O(n)$ `remove`, `insert`, `append`

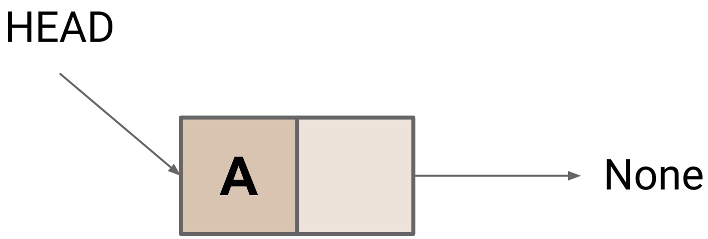
`ArrayBuffer[T]`
Pros: $O(1)$ `apply`, `update`, Amortized $O(1)$ `append`
Cons: $O(n)$ `insert`, `remove`

`List[T]` (linked list)
Pros: ???
Cons: ???

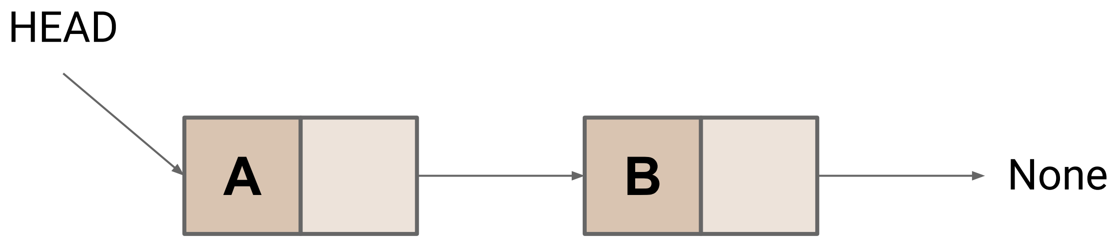
Linked Lists



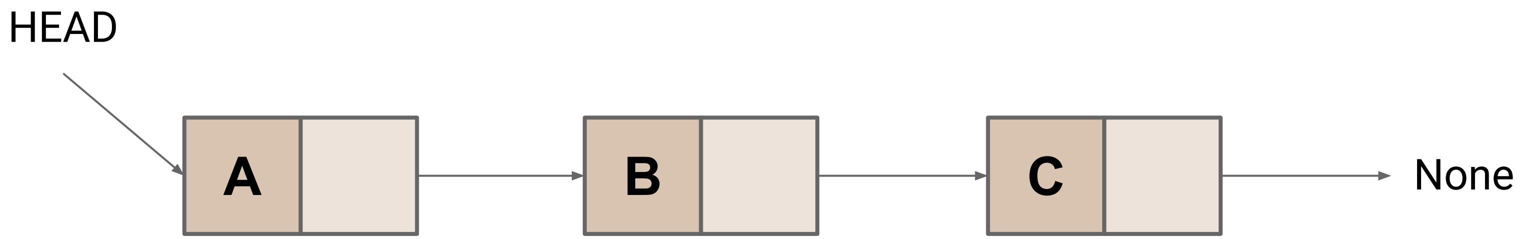
Linked Lists



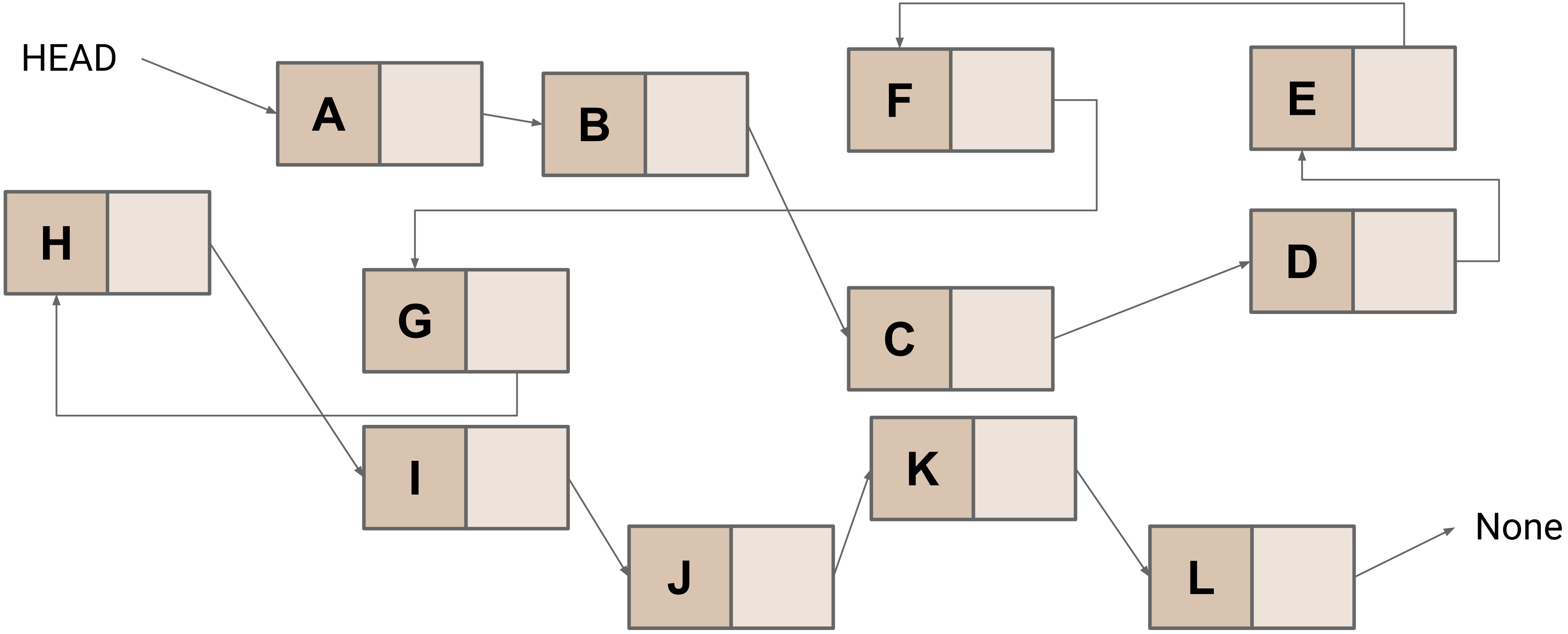
Linked Lists



Linked Lists



Linked Lists



mutable.List[T]:mutable.Seq[T]

```scala
class SinglyLinkedList[T] extends Seq[T] {
    var head: Option[SinglyLinkedListNode[T]] = None
    /* ... */
}
```

```scala
class SinglyLinkedListNode(
    var value: T,
    var next: Option[SinglyLinkedListNode[T]] = None
  )
```

mutable.List[T] : mutable.Seq[T]

```
class SinglyLinkedList[T] extends Seq[T] {
    var head: Option[SinglyLinkedListNode[T]] = None
    /* ... */
}
```

Class for our list, which right now just has a reference to **`head`** (an **`Option`**)

```
class SinglyLinkedListNode(
    var value: T,
    var next: Option[SinglyLinkedListNode[T]] = None
  )
```

mutable.List[T]:mutable.Seq[T]

```
class SinglyLinkedList[T] extends Seq[T] {
    var head: Option[SinglyLinkedListNode[T]] = None
    /* ... */
}
```

Class for our list, which right now just has a reference to **`head`** (an **`Option`**)

Class for a node in the list, which has a **`value`**, and a reference to the **`next`** node (an **`Option`**)

```
class SinglyLinkedListNode(
    var value: T,
    var next: Option[SinglyLinkedListNode[T]] = None
  )
```

The mutable.Seq ADT

`apply(idx: Int): [A]`
Get the element (of type **`A`**) at position **`idx`**

`iterator: Iterator[A]`
Get access to view all elements in the sequence, in order, once

`length: Int`
Count the number of elements in the seq

`insert(idx: Int, elem: A): Unit`
Insert an element at position **`idx`** with value **`elem`**

`remove(idx: Int): A`
Remove the element at position **`idx`**, and return the removed value

Implementing `length`

```
def length: Int = {
    var i = 0
    var curr = head
    while(curr.isDefined){ i += 1; curr = curr.get.next }
    return i
}
```

Implementing `length`

```
def length: Int = {
    var i = 0
    var curr = head
    while(curr.isDefined){ i += 1; curr = curr.get.next }
    return i
}
```

Complexity: *O(n)*

Implementing `length`

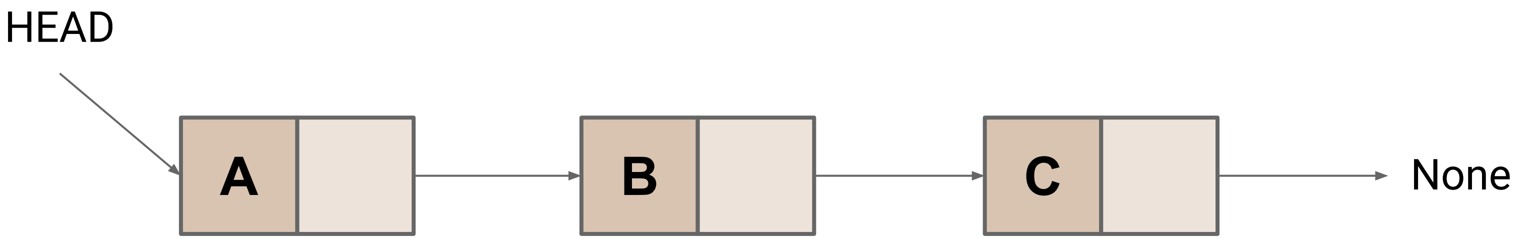
Idea: Keep track of the length

```
class SinglyLinkedList[T] extends Seq[T] {
    var head: Option[SinglyLinkedListNode[T]] = None
    var length = 0
    /* ... */
}
```

Complexity: $O(1)$

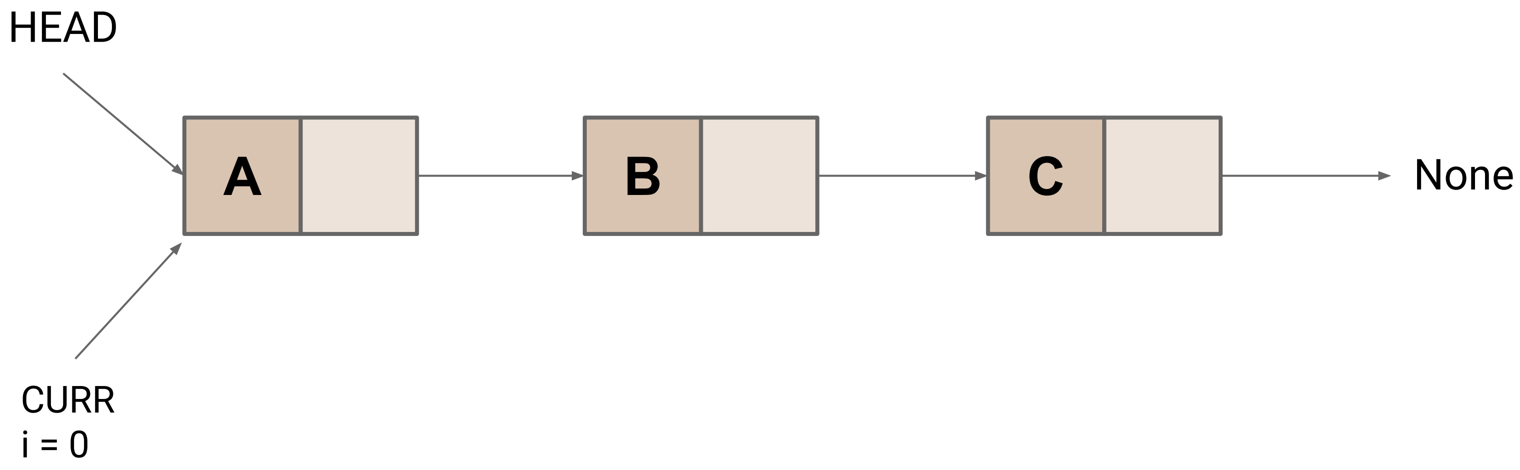
Implementing `apply`

`apply(2)`



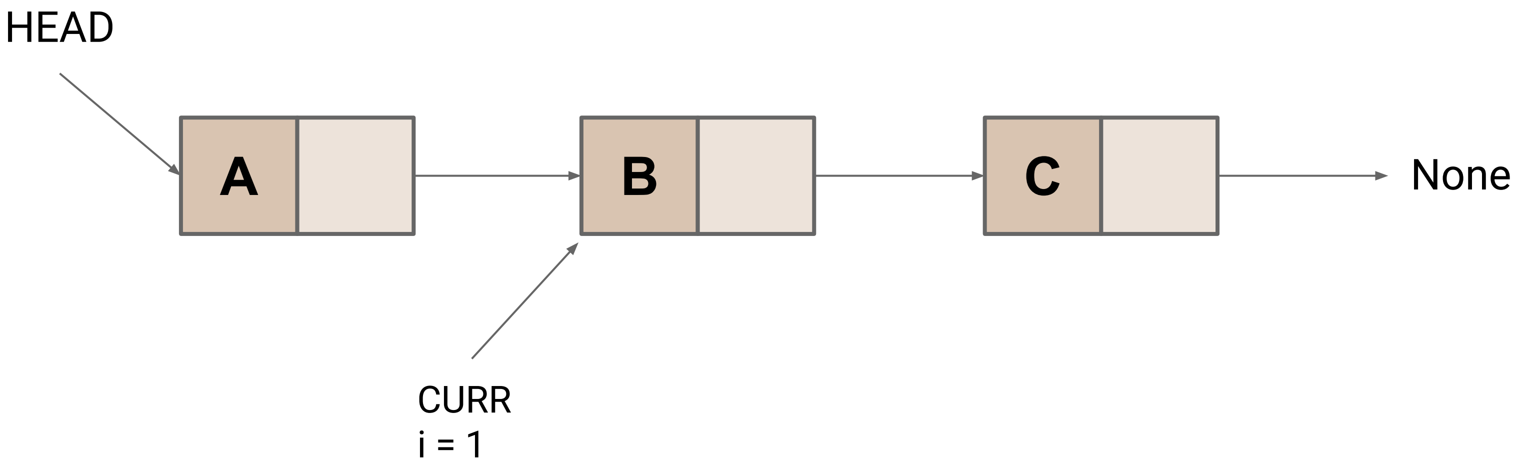
Implementing `apply`

`apply(2)`



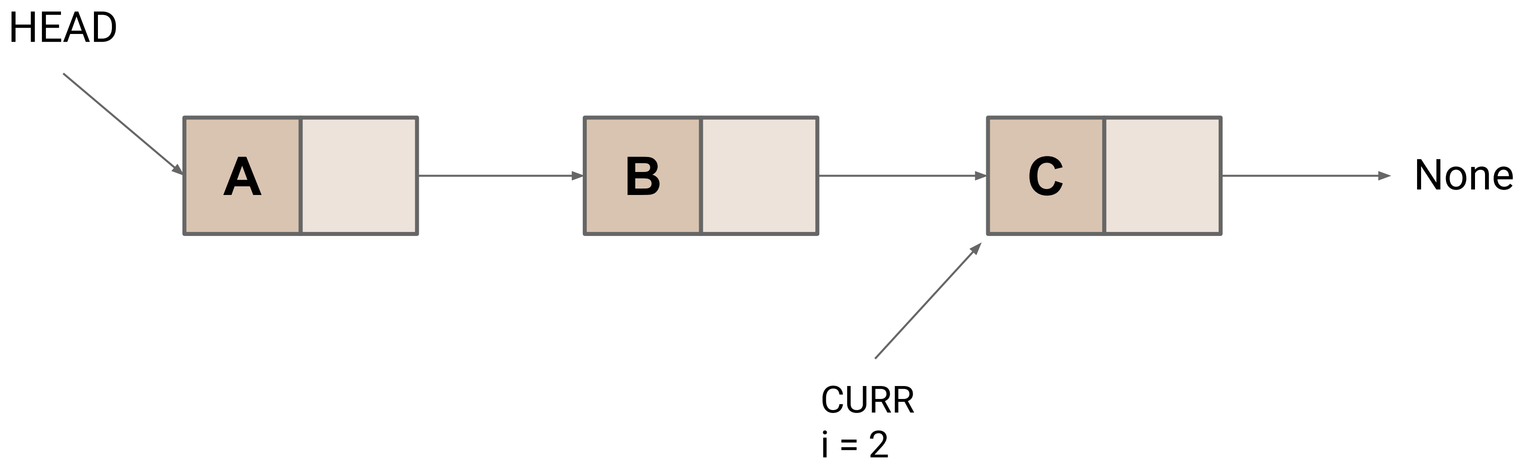
Implementing `apply`

`apply(2)`



Implementing `apply`

`apply(2)`



Implementing `apply`

```
def apply(idx: Int): T = {
    var i = 0
    var curr = head
    while(i < idx){
        if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
        i += 1; curr = curr.get.next
    }
    if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
    return curr
}
```

Implementing `apply`

```
def apply(idx: Int): T = {
    var i = 0
    var curr = head
    while(i < idx){
        if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
        i += 1; curr = curr.get.next
    }
    if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
    return curr
}
```

Iterate until we reach idx

Implementing `apply`

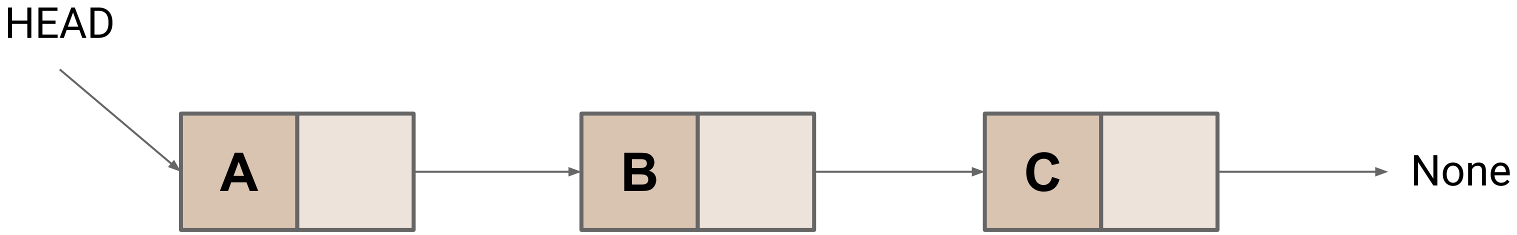
```
def apply(idx: Int): T = {
    var i = 0
    var curr = head
    while(i < idx){
        if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
        i += 1; curr = curr.get.next
    }
    if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
    return curr
}
```

Iterate until we reach idx

Complexity: $O(n)$ (or $\Theta($`idx`$)$)

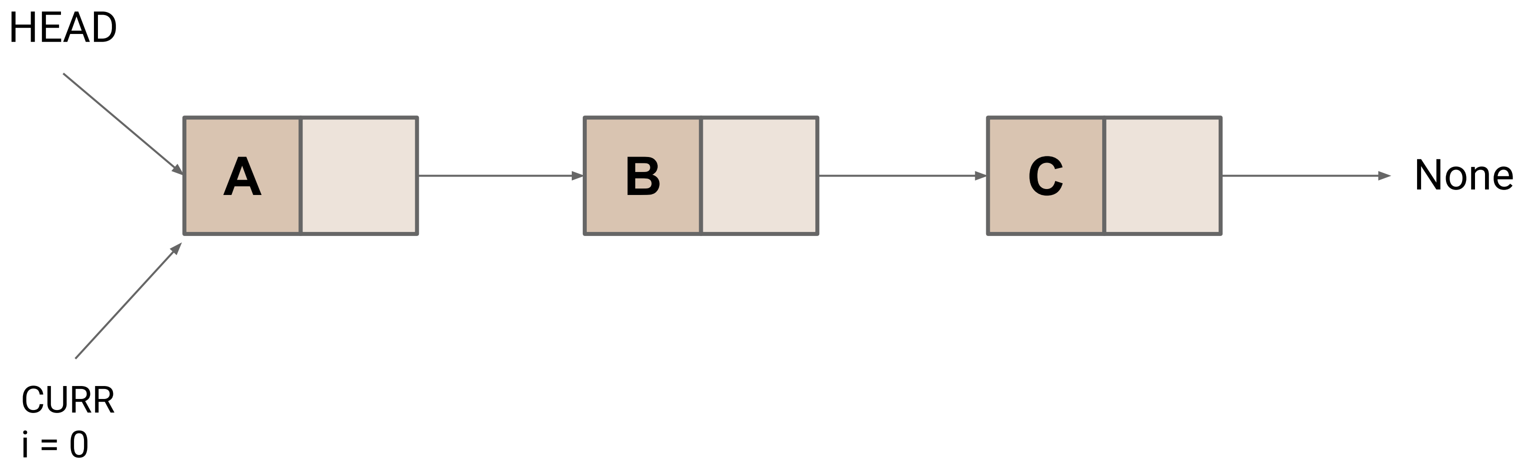
Implementing `insert`

`insert(1,"D")`



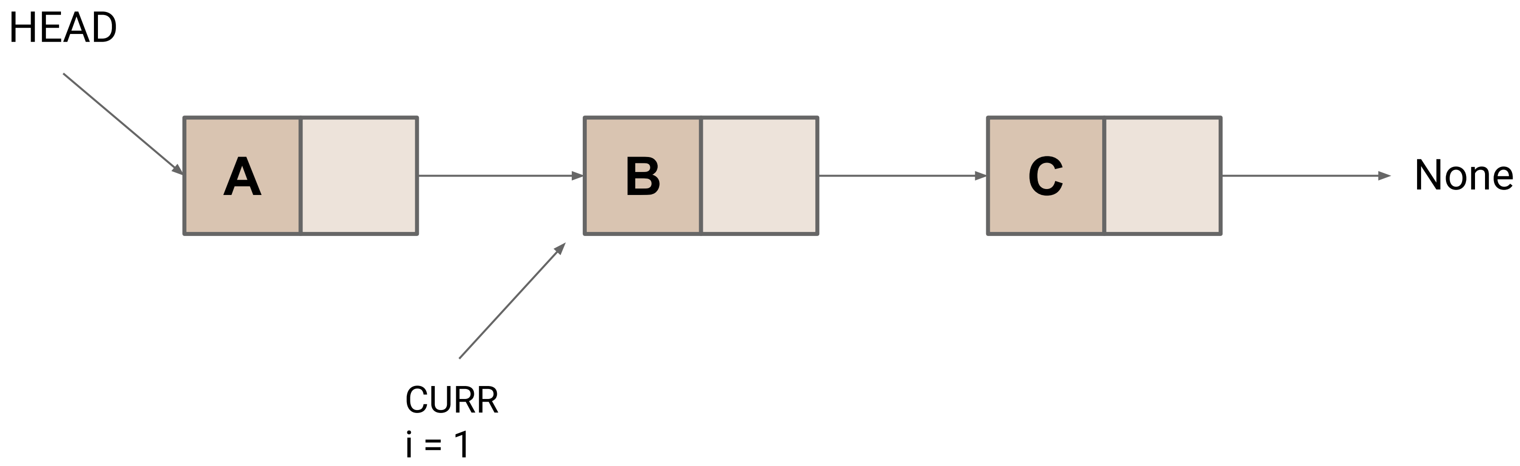
Implementing `insert`

`insert(1,"D")`



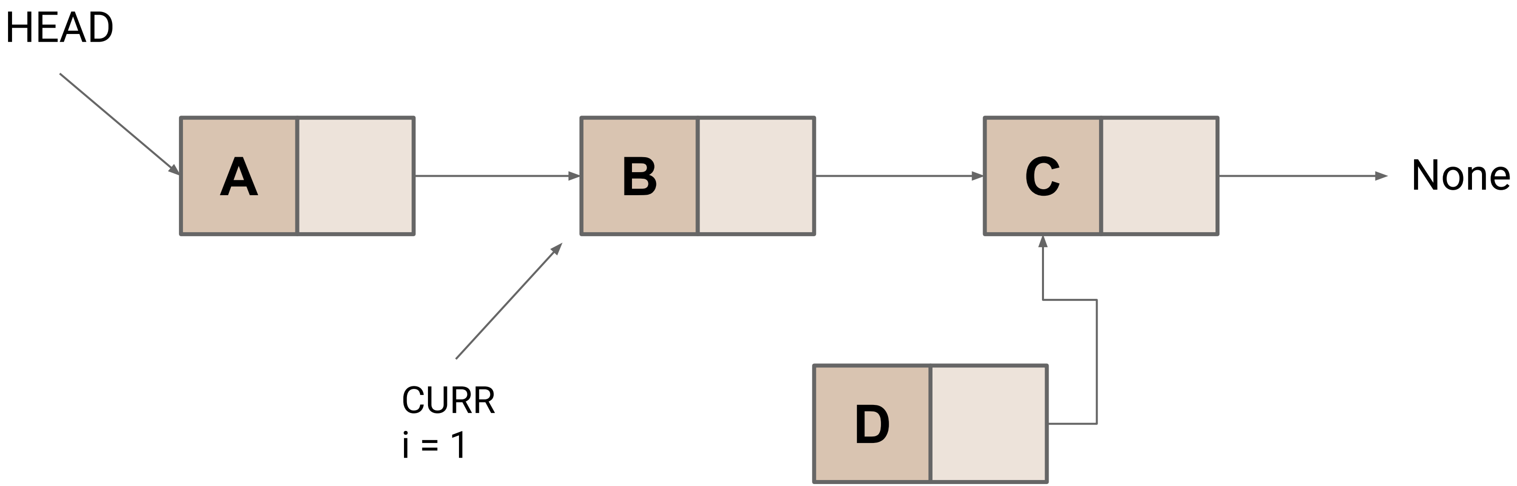
Implementing `insert`

`insert(1,"D")`



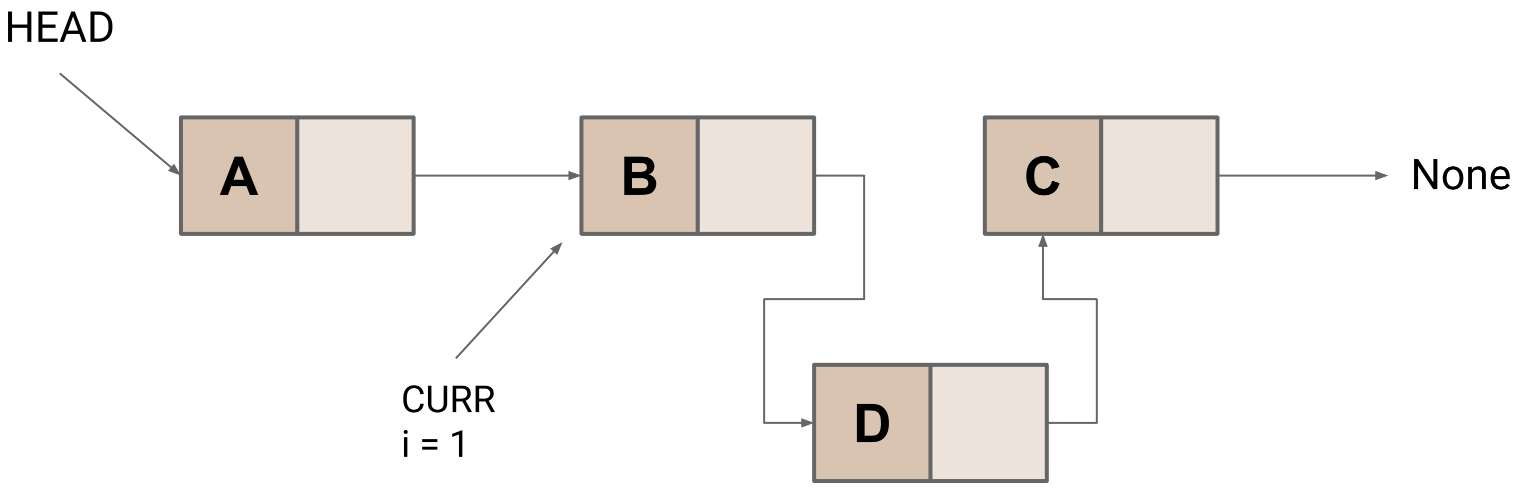
Implementing `insert`

`insert(1,"D")`



Implementing `insert`

`insert(1,"D")`



Implementing `insert`

```scala
def insert(idx: Int, value: T): Unit = {
    if(idx == 0) {
        head = Some( new SinglyLinkedListNode(value, head) )
    } else {
        var i = 1; var curr = head
        while(i < idx) {
            if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
            i += 1; curr = curr.get.next
        }
        curr.next = Some(new SinglyLinkedListNode(value, curr.next))
    }
    length += 1
}
```

Implementing `insert`

```
def insert(idx: Int, value: T): Unit = {
    if(idx == 0) {
        head = Some( new SinglyLinkedListNode(value, head) )
    } else {
        var i = 1; var curr = head
        while(i < idx) {
            if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
            i += 1; curr = curr.get.next
        }
        curr.next = Some(new SinglyLinkedListNode(value, curr.next))
    }
    length += 1
}
```

Iterate until we reach idx

Implementing `insert`

```
def insert(idx: Int, value: T): Unit = {
    if(idx == 0) {
        head = Some( new SinglyLinkedListNode(value, head) )
    } else {
        var i = 1; var curr = head
        while(i < idx) {
            if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
            i += 1; curr = curr.get.next
        }
        curr.next = Some(new SinglyLinkedListNode(value, curr.next))
    }
    length += 1
}
```

Iterate until we reach idx

Create a new node pointing to curr.next and update curr.next

Implementing `insert`

```
def insert(idx: Int, value: T): Unit = {
    if(idx == 0) {
        head = Some( new SinglyLinkedListNode(value, head) )
    } else {
        var i = 1; var curr = head
        while(i < idx) {
            if(curr.isEmpty) { throw IndexOutOfBoundsException(idx) }
            i += 1; curr = curr.get.next
        }
        curr.next = Some(new SinglyLinkedListNode(value, curr.next))
    }
    length += 1
}
```

Iterate until we reach idx

Create a new node pointing to curr.next and update curr.next

Complexity: $O(n)$ (or $\Theta(\texttt{idx})$)

Let's try actually using apply...

Using apply

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    for(i <- 0 until list.size){ total += list(i) }
    return total
}
```

What is the complexity?

Using `apply`

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    for(i <- 0 until list.size){ total += list(i) }
    return total
}
```

Iterate over the whole list: O(n)

What is the complexity?

Using `apply`

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    for(i <- 0 until list.size){ total += list(i) }
    return total
}
```

Iterate over the whole list: O(n)

What is the complexity?

Call apply: O(n)

Using `apply`

$$\sum_{i=0}^{n-1} T_{apply}(i) = \sum_{i=0}^{n-1} i$$

Using `apply`

$$\sum_{i=0}^{n-1} T_{apply}(i) = \sum_{i=0}^{n-1} i$$

$$= \sum_{i=0}^{n-1} \frac{(n-1)(n-1+1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$$

Using `apply`

$$\sum_{i=0}^{n-1} T_{apply}(i) = \sum_{i=0}^{n-1} i$$

$$= \sum_{i=0}^{n-1} \frac{(n-1)(n-1+1)}{2} = \frac{n^2 - n}{2} = \Theta(n^2)$$

Can we do better?

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head
    while(curr.isDefined){
        total += curr.get.value
        curr = curr.get.next
    }
    return total
}
```

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head
    while(curr.isDefined){
        total += curr.get.value
        curr = curr.get.next
    }
    return total
}
```

Start at the head

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head
    while(curr.isDefined){
        total += curr.get.value
        curr = curr.get.next
    }
    return total
}
```

Start at the head

Go as long as there are nodes

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head              Start at the head
    while(curr.isDefined){            Go as long as there are nodes
        total += curr.get.value       Add to the total
        curr = curr.get.next
    }
    return total
}
```

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head                Start at the head
    while(curr.isDefined){              Go as long as there are nodes
        total += curr.get.value         Add to the total
        curr = curr.get.next            Go to the next item
    }
    return total
}
```

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head                Start at the head
    while(curr.isDefined){              Go as long as there are nodes
        total += curr.get.value         Add to the total
        curr = curr.get.next            Go to the next item
    }
    return total
}
```

Now what is our complexity?

A Different Approach...

```
def sum(list: List[Int]): Unit = {
    val total: Int = 0
    val curr = list.head                 Start at the head
    while(curr.isDefined){              Go as long as there are nodes
        total += curr.get.value          Add to the total
        curr = curr.get.next             Go to the next item
    }
    return total
}
```

Now what is our complexity?

$$\sum_{i=0}^{n-1} \Theta(1) = (n - 1 + 1) \cdot \Theta(1) = \Theta(n)$$

Access-by-Reference vs -by-Index

Why does this work?

What is the expensive part of `apply`?

Access-by-Reference vs -by-Index

Index →Value: $\Theta(idx)$
(access by index)

`SinglyLinkedListNode` → Value: $\Theta(1)$
(access by reference)

`Iterator[T]`

`hasNext: Boolean`

Returns **`true`** if there are more items to retrieve

`next: T`

Returns the next item to retrieve

`Iterator[T]`

`hasNext: Boolean`

Returns **`true`** if there are more items to retrieve

`next: T`

Returns the next item to retrieve

*An iterator is a **reference** to an element of a collection*

ListIterator[T]:Iterator[T]

```
class ListIterator[T](
        var curr: Option[SinglyLinkedListNode[T]]) {
    def hasNext: Boolean = curr.isDefined
    def next: T = {
        val ret = curr.get.value
        curr = curr.get.next
        return ret
    }
}
```

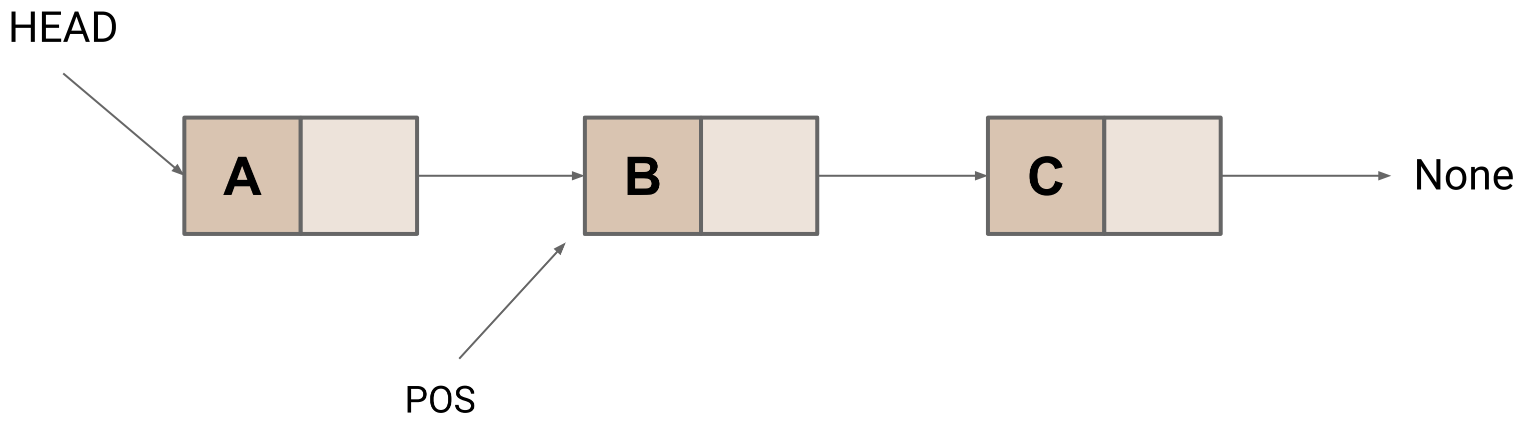
`List:` Positional Operations

What if we try something like:

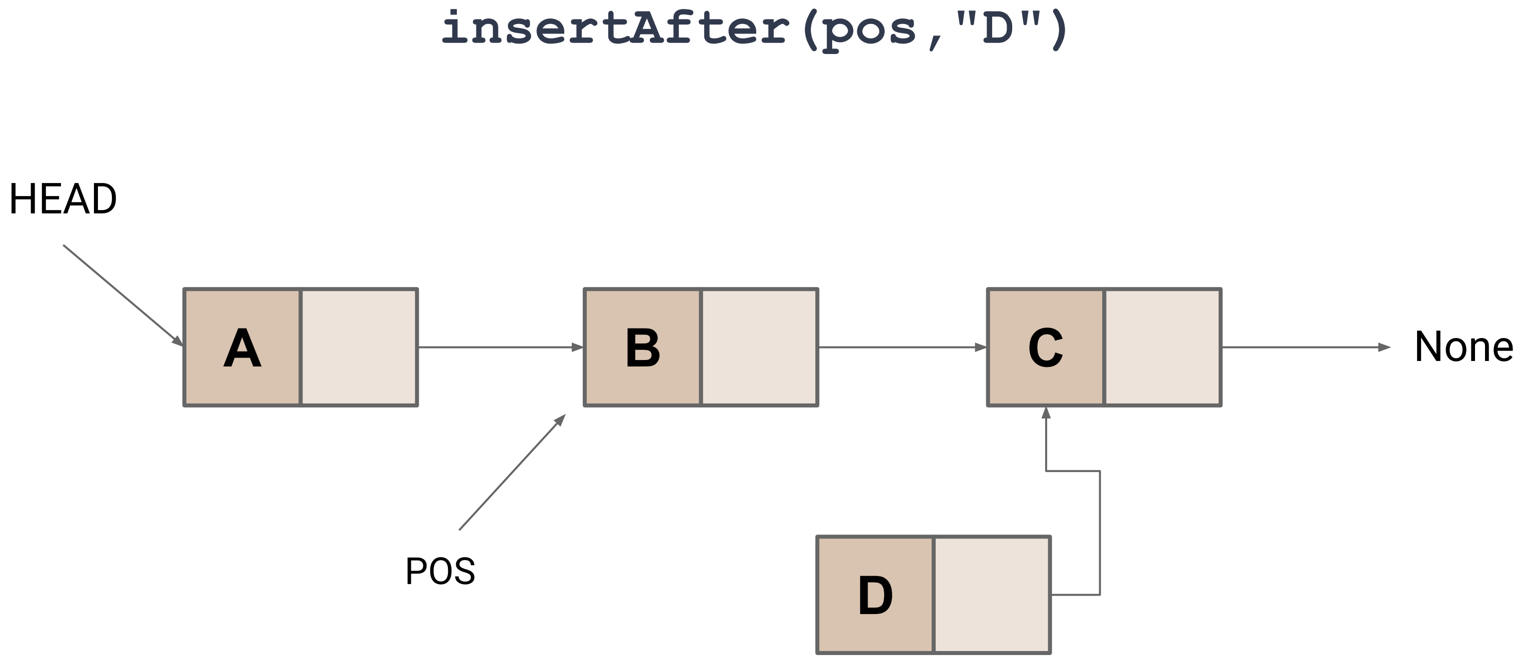
```
insertAfter(pos: SinglyLinkedListNode[T], value: T)
```

Implementing `insertAfter`

`insertAfter(pos,"D")`

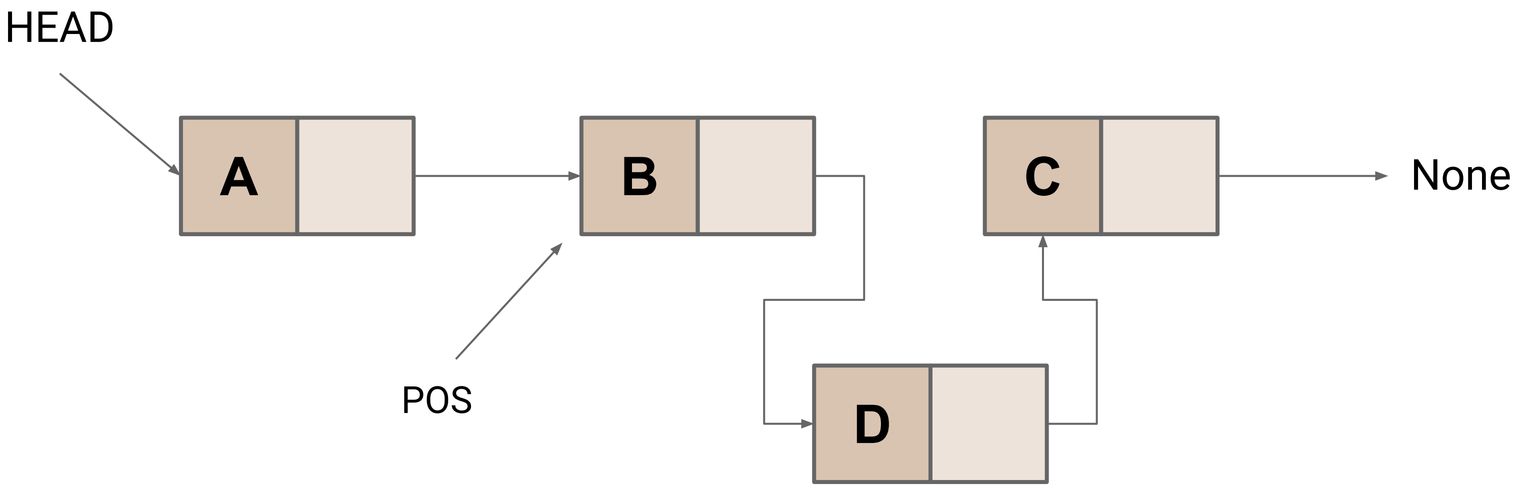


Implementing `insertAfter`



Implementing `insertAfter`

`insertAfter(pos,"D")`



Implementing `insertAfter`

```
def insertAfter(pos: SinglyLinkedListNode[T], value: T) = {
    pos.next = Some(
            new SinglyLinkedListNode(value, pos.next))
    length += 1
}
```

Implementing `insertAfter`

```
def insertAfter(pos: SinglyLinkedListNode[T], value: T) = {
    pos.next = Some(
            new SinglyLinkedListNode(value, pos.next))
    length += 1
}
```

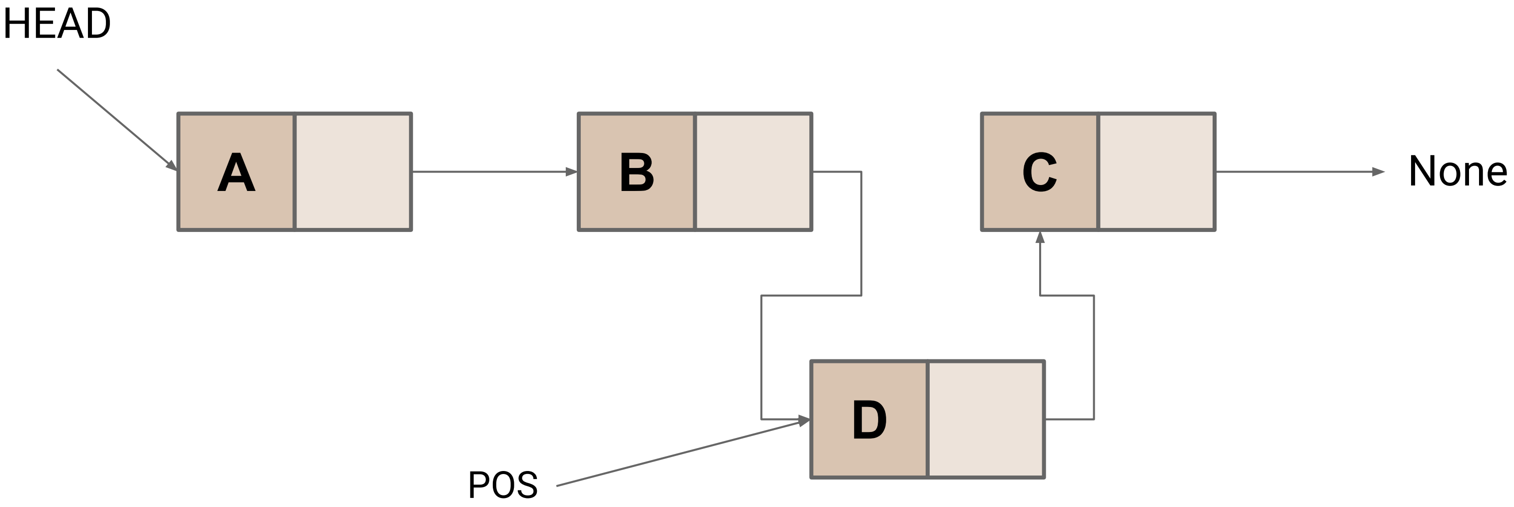
Complexity: $\Theta(1)$

Implementing positional `remove`

How would you implement a positional **`remove`**?

Implementing Positional `remove`

`remove(pos)`



Implementing positional `remove`

How would you implement a positional **`remove`**?

```
def remove(pos: SinglyLinkedListNode[T]): T = {
     val prev = ???
     prev.next = pos.next
     length -= 1
     return pos.get.value
}
```

Implementing positional `remove`

How would you implement a positional **`remove`**?

```
def remove(pos: SinglyLinkedListNode[T]): T = {
    val prev = ???
    prev.next = pos.next
    length -= 1
    return pos.get.value
}
```

Problem: ...how do we find the previous node?

Implementing positional `remove`

Idea: Use a "backward" pointer.

DoublyLinkedList

```scala
class DoublyLinkedList[T] extends Seq[T] {
    var head: Option[DoublyLinkedListNode[T]] = None
    var last: Option[DoublyLinkedListNode[T]] = None
    var length = 0

    /* ... */
}
```

```scala
class DoublyLinkedListNode[T](
    var value: T,
    var next: Option[DoublyLinkedListNode[T]] = None
    var prev: Option[DoublyLinkedListNode[T]] = None
  )
```

Re-implementing positional `insertAfter`

```scala
def insertAfter(pos: DoublyLinkedListNode[T], value: T) = {
    val newNode
        = new DoublyLinkedListNode(value, prev = Some(pos))
    if(pos.next.isDefined) {
        pos.next.prev = Some(newNode)
        newNode.next = pos.next
    } else {
        last = newNode
        newNode.next = None
    }
    pos.next = Some(newNode)
    length += 1
}
```

Re-implementing positional `insertAfter`

```
def insertAfter(pos: DoublyLinkedListNode[T], value: T) = {
    val newNode
        = new DoublyLinkedListNode(value, prev = Some(pos))
    if(pos.next.isDefined) {
        pos.next.prev = Some(newNode)
        newNode.next = pos.next
    } else {
        last = newNode
        newNode.next = None
    }
    pos.next = Some(newNode)
    length += 1
}
```

prev of the new node is the position we are inserting after

Re-implementing positional `insertAfter`

```
def insertAfter(pos: DoublyLinkedListNode[T], value: T) = {
    val newNode
        = new DoublyLinkedListNode(value, prev = Some(pos))
    if(pos.next.isDefined) {
        pos.next.prev = Some(newNode)
        newNode.next = pos.next
    } else {
        last = newNode
        newNode.next = None
    }
    pos.next = Some(newNode)
    length += 1
}
```

Insert differs depending on whether the new node is now the end or not

prev of the new node is the position we are inserting after

Implementing positional `remove`

```
def remove(pos: DoublyLinkedListNode[T]): T = {
     if(pos.prev.isDefined){ pos.prev.next = pos.next }
     else                  { head = pos.next }

     if(pos.next.isDefined){ pos.next.prev = pos.prev }
     else                  { tail = pos.prev }

     length -= 1
     return pos.get.value
}
```

`Seq` Summary So Far

Operation	`Array[T]`	`ArrayBuffer[T]`	`List[T]` *(index)*	`List[T]` *(ref)*
`apply(i)`	$\Theta(1)$	$\Theta(1)$	$\Theta(i)$, $O(n)$	$\Theta(1)$
`update(i, val)`	$\Theta(1)$	$\Theta(1)$	$\Theta(i)$, $O(n)$	$\Theta(1)$
`insert(i, val)`	$\Theta(n)$	$O(n)$	$\Theta(i)$, $O(n)$	$\Theta(1)$
`remove(i, val)`	$\Theta(n)$	$\Theta(n-i)$, $O(n)$	$\Theta(i)$, $O(n)$	$\Theta(1)$
`append(i)`	$\Theta(n)$	$O(n)$, Amortized $\Theta(1)$	$\Theta(i)$, $O(n)$	$\Theta(1)$