CSE 250: Linked Lists, Iterators

Lecture 9

Sept 18, 2023

Reminders

- PA1 Implementation due Sun, Sept 24 at 11:59 PM
  - Implement a Sorted Linked List
- WA2 due Sun, Oct 1 at 11:59 PM
  - Will be released over the next weekend
- Midterm 1 on Oct 2 in-class
  - Covers: Asymptotics, Sequences/Lists, Arrays, Linked Lists, Recursion
  - Bounds: Tight Upper/Lower, General vs Expected vs Amortized

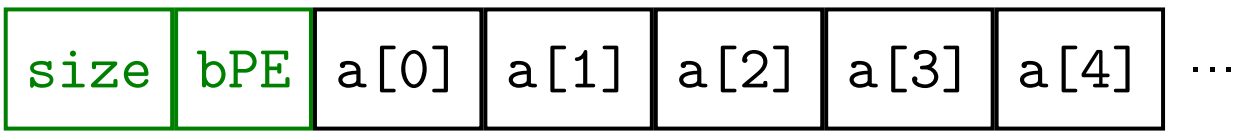
The Sequence ADT

```
1  public interface Sequence<E>
2  {
3    public E get(idx: Int);
4    public void set(idx: Int, E value);
5    public int size();
6    public Iterator<E> iterator();
7  }
```

Arrays

What information goes into an `T[]` array?

- `size`: 4 bytes for the number of elements[1].
- `bytesPerElement`: 4 bytes for `sizeof(T)`[2].
- `data`: `size` × `bytesPerElement` bytes.



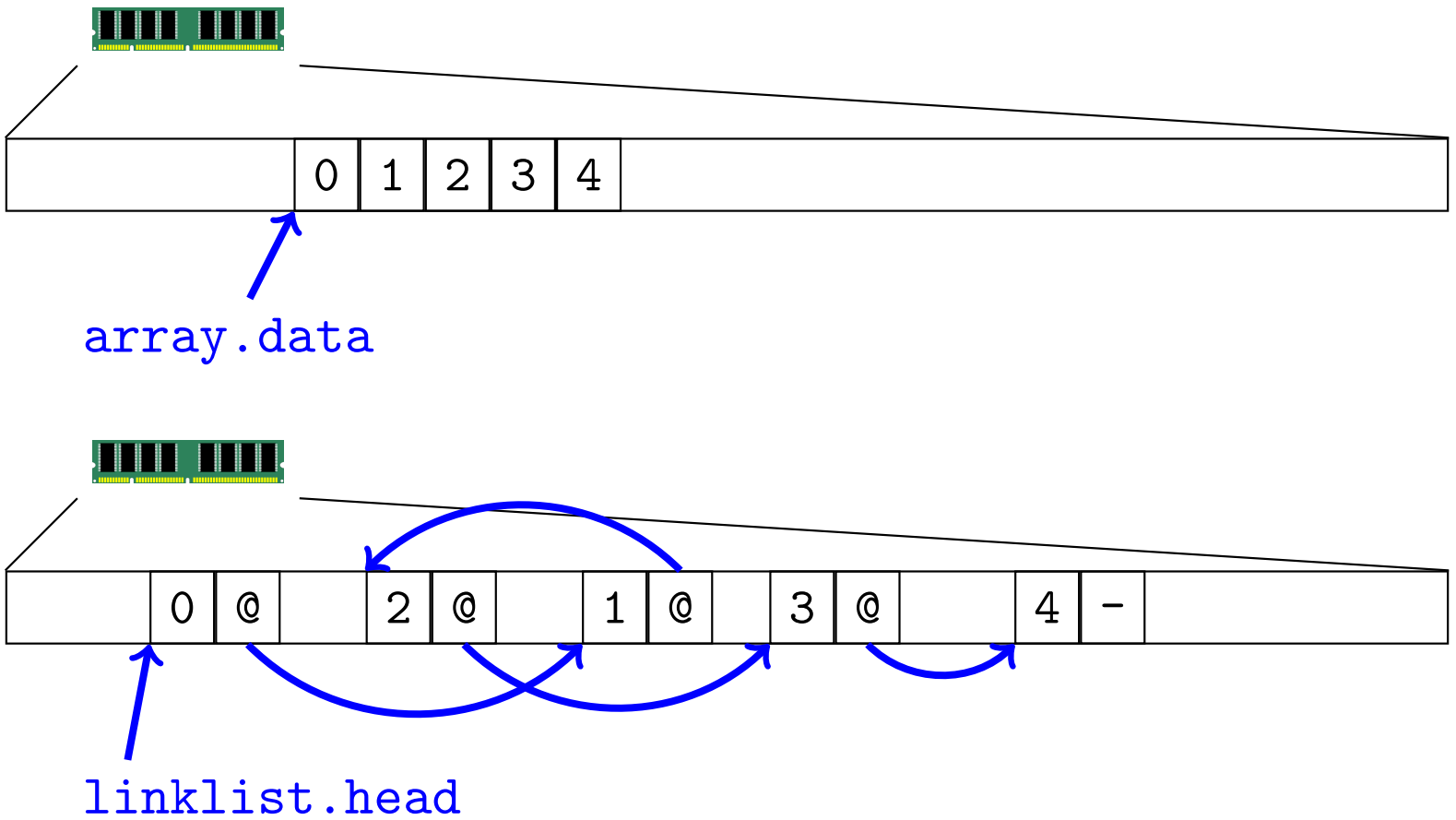
[1] Some languages (e.g., C) skip this, relying on the programmer to track it.
[2] Some languages (e.g., C, C++) skip this, since it's fixed at compile time.

Linked Lists

```java
public class LinkedListNode<T>
{
  T value;
  LinkedListNode<T> next = null;
}
```

```java
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size;
  /* ... */
}
```

Arrays vs Linked Lists



A few more reminders

- Operations on a linked list are $\theta(idx)$ (aka $\approx O(N)$) because we need to find the node at the index.
- Previous and Tail pointers make a *Doubly* Linked List, allowing us to move *backwards* through the list if needed.

The Sequence ADT

```
1  public interface Sequence<E>
2  {
3    public E get(int idx);
4    public void set(int idx, E value);
5    public int size();
6    public Iterator<E> iterator();
7  }
```

What about changing the sequence's size?

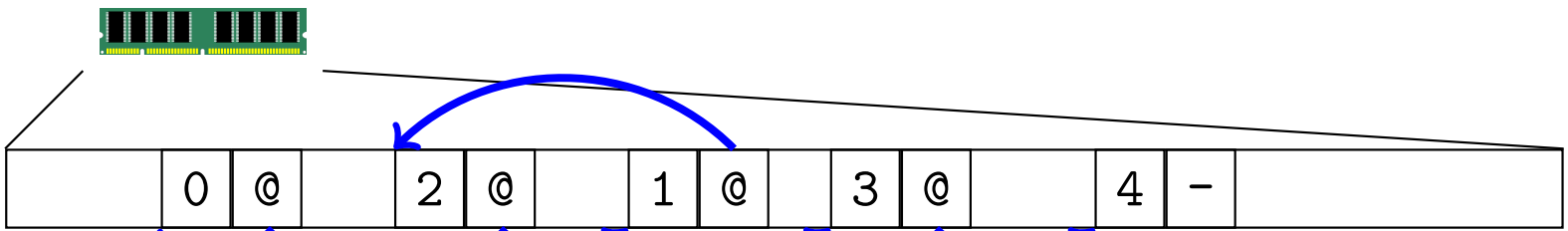
The List ADT

```
1   public interface List<E>
2     extends Sequence<E> // Everything a sequence has, and...
3   {
4     /** Extend the sequence with a new element at the end */
5     public void add(E value);
6
7     /** Extend the sequence by inserting a new element */
8     public void add(int idx, E value);
9
10    /** Remove the element at a given index */
11    public void remove(int idx);
12  }
```

Lists in Other Languages

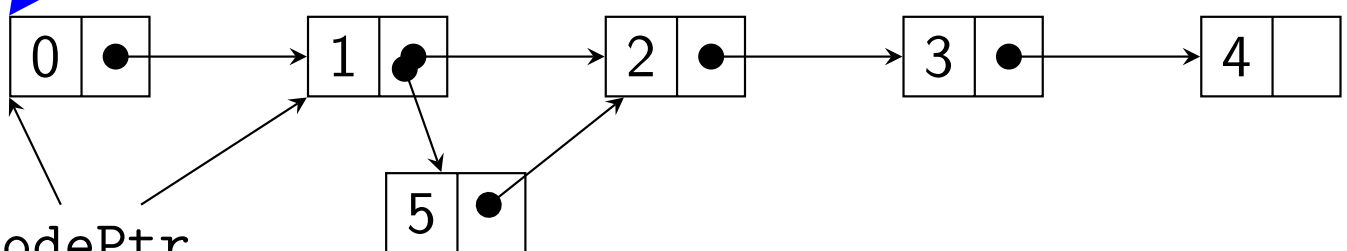
- Java, Python: `List`, `list`
- C++, Rust: `vector`, `Vec`
- Scala: `Buffer`
- Go: `Slice`

Linked Lists - add(idx, e)



linklist.head

```
add(idx= 2, value= 5);
```



nodePtr

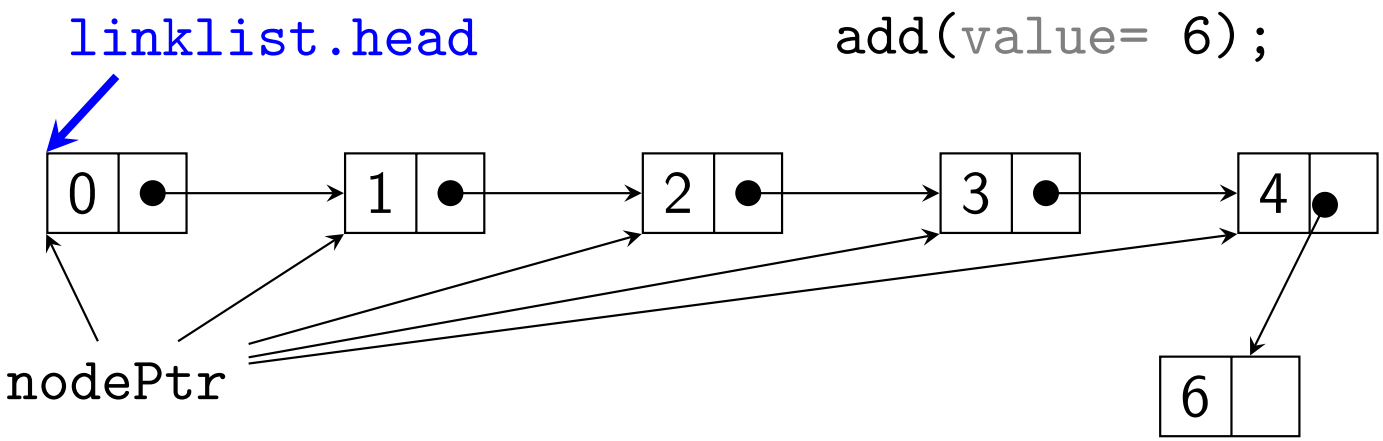
nodeIdx = 01(= idx - 1)

Linked Lists - `add(idx, e)`

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

Total: $O(N)$

Linked Lists - add(e)

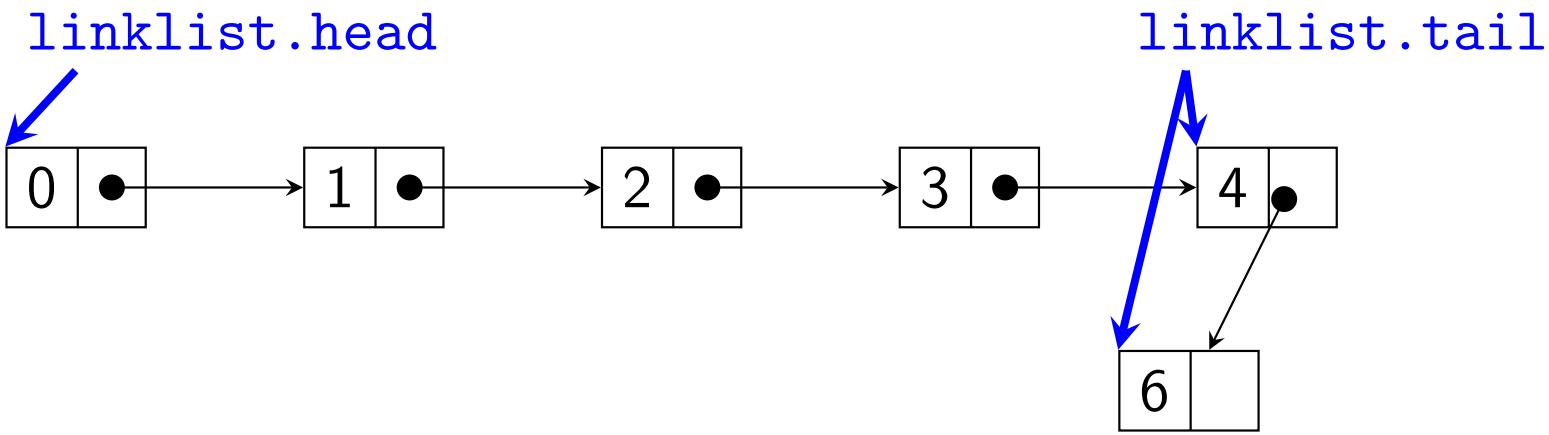


Linked Lists - `add(e)`

1. Find the last node.
   $O(N)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$

Total: $O(N)$

Linked Lists - add(e)

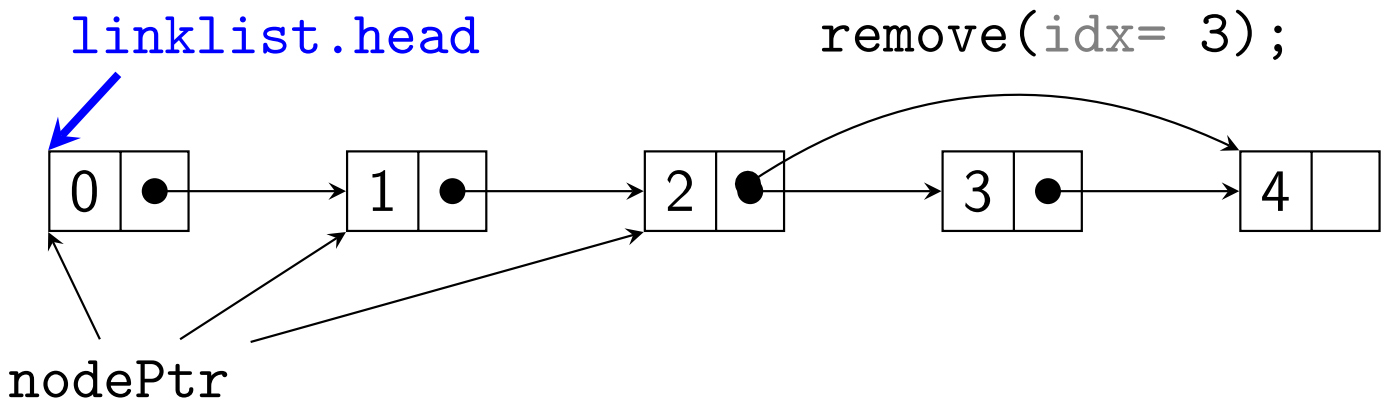


Linked Lists - `add(e)`

1. Find the last node.
   $O(1)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$
4. Update the tail pointer.
   $O(1)$

Total: $O(1)$

Linked Lists - `remove(idx)`



Linked Lists - `remove(idx)`

1. Find the node before `idx`.
   $O(N)$
2. Update the node before `idx`'s pointer.
   $O(1)$
3. Allow the node at `idx` to be reclaimed.
   $O(1)$

Total: $O(N)$

Linked Lists

The expensive operation is finding the `idx`'th node

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       for(i = 0; i < N; i++)
6       {
7         int value = list.get(i);
8         total += value;
9       }
10      return total;
11    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       for(i = 0; i < N; i++)
6       {
7         O(N)
8         total += value;
9       }
10      return total;
11    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       for(i = 0; i < N; i++)
6       {
7         O(N)
8         O(1)
9       }
10      return total;
11    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       for(i = 0; i < N; i++)
6       {
7         O(N)
8       }
9       return total;
10    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       ∑_{i=0}^{N} O(N)
6       return total;
7     }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       O(N^2)
6       return total;
7     }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       O(1);
4       int N = list.size()
5       O(N^2)
6       O(1);
7     }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       O(1);
4       O(1);
5       O(N^2)
6       O(1);
7     }
```

Enumeration

```
1    public int sumUpList(LinkedList<Integer> list)
2    {
3      O(N^2)
4    }
```

Why is this so expensive?

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       for(i = 0; i < N; i++)
6       {
7         int value = list.get(i);
8         total += value;
9       }
10      return total;
11    }
```

We're starting from 0 for each loop.

Can we do better?

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       Optional<LinkedListNode<Integer>> node = list.head;
6       while(node.isPresent())
7       {
8         int value = node.get().value;
9         total += value;
10        node = node.get().next;
11      }
12      return total;
13    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       Optional<LinkedListNode<Integer>> node = list.head;
6       while(node.isPresent())
7       {
8         O(1)
9         O(1)
10        O(1)
11      }
12      return total;
13    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       Optional<LinkedListNode<Integer>> node = list.head;
6       while(node.isPresent())
7       {
8         O(1)
9       }
10      return total;
11    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       Optional<LinkedListNode<Integer>> node = list.head;
6       ∑_{elem:list} O(1)
7       return total;
8     }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       Optional<LinkedListNode<Integer>> node = list.head;
6       O(N · 1)
7       return total;
8     }
```

Enumeration

```
1    public int sumUpList(LinkedList<Integer> list)
2    {
3      O(1)
4      O(1)
5      O(1)
6      O(N)
7      O(1)
8    }
```

Enumeration

```
1    public int sumUpList(LinkedList<Integer> list)
2    {
3      O(N)
4    }
```

Enumeration

This code is *specialized* for `LinkedLists`

- We can't re-use it for an `ArrayList`.
- If we change `LinkedList`, the code breaks.

How do we get code that is both fast and general?

- We need a way to represent a reference to the `idx`'th element of a list.

The `idx`'th element ADT

What can we do with a reference to an index?

- Get the value
- Get a reference to the next element
- Get a reference to the previous element
- Remove the element
- Insert a new element

ListIterator

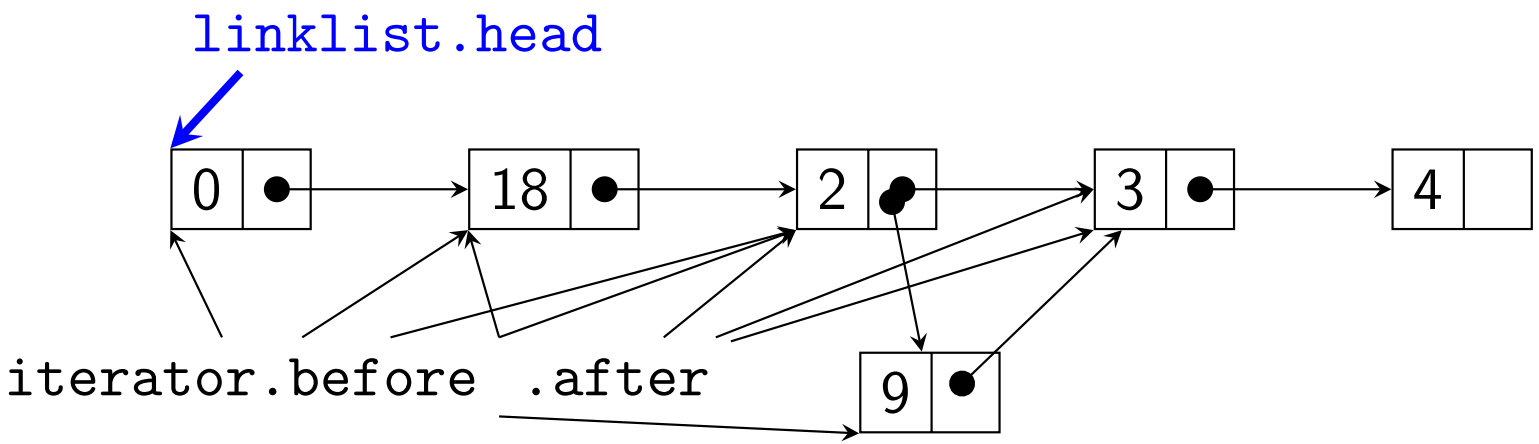
```
1   public interface ListIterator<E>
2   {
3     public boolean hasNext();
4     public E next();
5     public boolean hasPrevious();  ← Only in ListIterator
6     public E previous();           ← Only in ListIterator
7     public void add(E value);      ← Only in ListIterator
8     public void set(E value);      ← Only in ListIterator
9     public void remove();
10  }
```

ListIterator adds features to Iterator

ListIterator (simplified)

```
1    public interface ListIterator<E>
2    {
3      public boolean hasNext();
4      public E next();
5      public void add(E value);
6      public void set(E value);
7      public void remove();
8    }
```

ListIterator




```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next(); → 2
iterator.add(9);
```

Implementing LinkedListIterator

```
1  public class LinkedListIterator<E>
2    extends ListIterator<E>
3  {
4    LinkedList<E> list;
5    Optional<LinkedListNode<E>> before = Optional.empty();
6    Optional<LinkedListNode<E>> after = Optional.of(list.head);
7    /* ... */
8  }
```

Implementing LinkedListIterator

- `public boolean hasNext();`
  If `after` is present, return true.
  $O(1)$
- `public E next();`
  If `after` is present, return it's value after advancing the iterators.
  $O(1)$
- `public void add(E value);`
  Create a new node, update its' `next`; Update either `before.next` or `list.head`
  $O(1)$

Implementing LinkedListIterator

- `public void set(E value);`
  Update `before.value`.
  $O(1)$
- `public void remove();`
  Set `before.next` or `list.head` to `after.next`. Update `after`.
  $O(1)$

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       LinkedListIterator<Integer> iterator = list.iterator();
6       while(iterator.hasNext())
7       {
8         int value = iterator.next();
9         total += value;
10      }
11      return total;
12    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       LinkedListIterator<Integer> iterator = list.iterator();
6       while(iterator.hasNext())
7       {
8         O(1)
9         O(1)
10      }
11      return total;
12    }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       LinkedListIterator<Integer> iterator = list.iterator();
6       ∑_{elem:list} O(1)
7       return total;
8     }
```

Enumeration

```
1     public int sumUpList(LinkedList<Integer> list)
2     {
3       int total = 0;
4       int N = list.size()
5       LinkedListIterator<Integer> iterator = list.iterator();
6       O(N · 1)
7       return total;
8     }
```

Enumeration

```
1    public int sumUpList(LinkedList<Integer> list)
2    {
3      O(1)
4      O(1)
5      O(1)
6      O(N · 1)
7      O(1)
8    }
```

Enumeration

```
1    public int sumUpList(LinkedList<Integer> list)
2    {
3      O(N)
4    }
```

Linked Lists

Access list by index: $O(N)$

Access list by reference (iterator): $O(1)$

Thought Question

How would we implement `add(e)` on an array?

Problem: Arrays are *fixed size*.

add(e) on an Array

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

Total: $O(N)$

add(e) on an Array

Can we do better?

Next Class!