CSE 250: Linked Lists, Iterators

Lecture 9

Sept 16, 2023

Reminders

- PA1 Implementation due Sun, Sept 22 at 11:59 PM
    - Implement a Sorted Linked List

The Sequence ADT

```
public interface Sequence<E>
{
  public E get(idx: Int);
  public void set(idx: Int, E value);
  public int size();
  public Iterator<E> iterator();
}
```

Arrays

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

- `size`: 4 bytes for the number of elements.
- `bytesPerElement`: 4 bytes for `sizeof(T)`.
- `data`: `size` $\times$ `bytesPerElement` bytes.

Arrays

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

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

[1]Some languages (e.g., C) skip this, relying on the programmer to track it.

Arrays

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

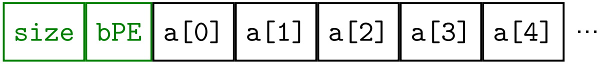
- `size`: 4 bytes for the number of elements[1].
- `bytesPerElement`: 4 bytes for `sizeof(T)`[2].
- `data`: `size` $\times$ `bytesPerElement` bytes.

[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.

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.

How do we implement...

- `public E get(int idx)`

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \textit{idx}$ to $\texttt{bPE} \times (\textit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx}+1) - 1$
  - $\theta(1)$ (if we treat bPE as a constant)
- `public void set(int idx, E value)`

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \textit{idx}$ to $\texttt{bPE} \times (\textit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat bPE as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times \textit{idx}$ to $\texttt{bPE} \times (\textit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat bPE as a constant)
- `public int size()`

How do we implement...

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times idx$ to $\texttt{bPE} \times (idx + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times idx$ to $\texttt{bPE} \times (idx + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public int size()`
  - Return `size`

How do we implement...

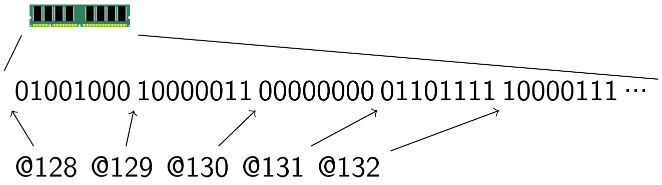
- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times \textit{idx}$ to $\texttt{bPE} \times (\textit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times \textit{idx}$ to $\texttt{bPE} \times (\textit{idx} + 1) - 1$
  - $\theta(1)$ (if we treat `bPE` as a constant)
- `public int size()`
  - Return `size`
  - $\theta(1)$

CSE 220 Crossover 2: List Harder

01001000 10000011 00000000 01101111 10000111 ···

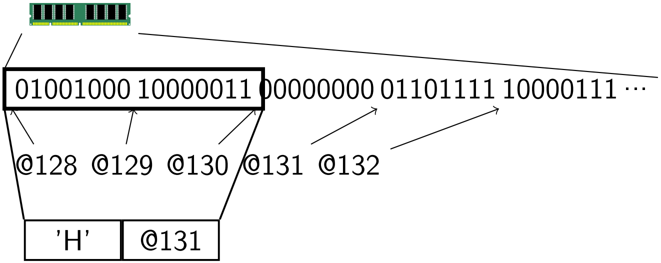
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder



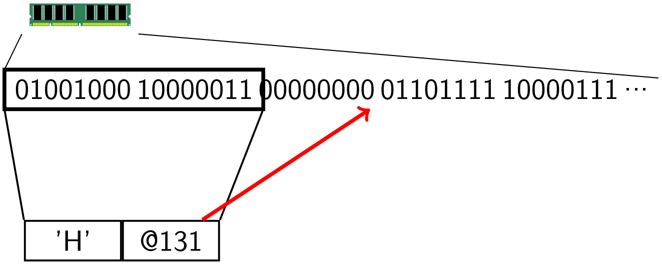
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder



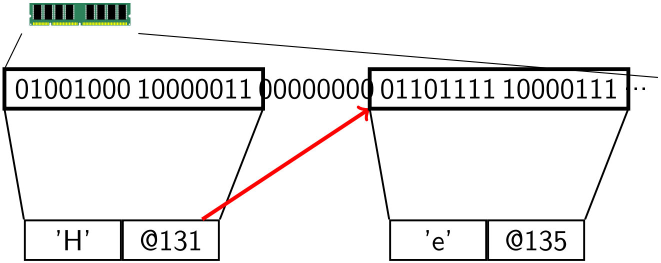
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder



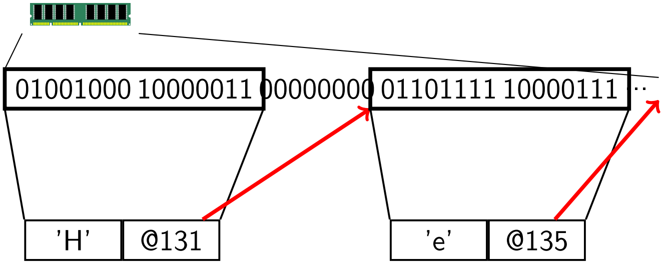
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder



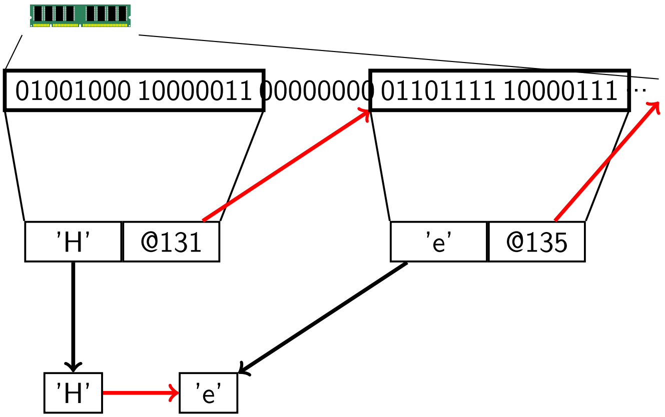
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder



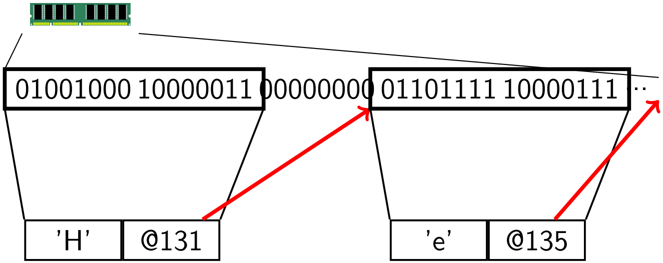
OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

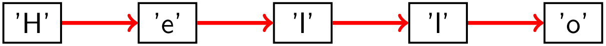
CSE 220 Crossover 2: List Harder



OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

CSE 220 Crossover 2: List Harder





OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

Linked Lists

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

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;
  /* ... */
}
```

How do we implement...

- `public E get(int idx)`

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$

How do we implement...

- `public E get(int idx)`
    - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
    - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
  - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.

How do we implement...

- `public E get(int idx)`
    - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
    - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
    - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
    - $\theta(idx)$, $O(N)$

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
  - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public int size()`

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
  - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public int size()`
  - Start at `head`, and move to the `next` element until you reach the end. Return the number of steps taken.

How do we implement...

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
  - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public int size()`
  - Start at `head`, and move to the `next` element until you reach the end. Return the number of steps taken.
  - $\theta(N)$

Linked Lists' `size`

Can we do better?

Store size

```java
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now?

Store size

```
  public class LinkedList<T> implements List<T>
  {
    LinkedListNode<T> head = null;
    int size = 0;
    /* ... */
  }
```

- How expensive is `public int size()` now? ($\theta(1)$)
- How expensive is it to maintain `size`?

Store size

```java
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now? $(\theta(1))$
- How expensive is it to maintain `size`? (Extra $\theta(1)$ work on insert/remove).

Store size

```java
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now? ($\theta(1)$)
- How expensive is it to maintain `size`? (Extra $\theta(1)$ work on insert/remove).

Storing redundant information can reduce complexity.

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.

Referential Access

What's the complexity of getting the value of the *i*'th element of a `LinkedList`?

Referential Access

What's the complexity of getting the value of the *i*'th element of a `LinkedList`?

If you have a pointer to the *i*'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting its value?

Referential Access

What's the complexity of getting the value of the i'th element of a `LinkedList`?

If you have a pointer to the i'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting its value?

If you have a pointer to the i'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting the value of the $i + 1$'th element?

Referential Access

What's the complexity of getting the value of the i'th element of a `LinkedList`?

If you have a pointer to the i'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting its value?

If you have a pointer to the i'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting the value of the $i + 1$'th element?

If you have a pointer to the i'th `LinkedListNode` of a `LinkedList`, what's the complexity of getting the value of the $i - 1$'th element?

Doubly Linked Lists

```
public class LinkedListNode<T>
{
  T value;
  Optional<LinkedListNode<T>> next = Optional.empty();
  Optional<LinkedListNode<T>> prev = Optional.empty();
}
```

```
public class LinkedList<T> implements List<T>
{
  Optional<LinkedListNode<T>> head = Optional.empty();
  Optional<LinkedListNode<T>> tail = Optional.empty();
  /* ... */
}
```

The Sequence ADT

```java
public interface Sequence<E>
{
  public E get(int idx);
  public void set(int idx, E value);
  public int size();
  public Iterator<E> iterator();
}
```

The Sequence ADT

```
public interface Sequence<E>
{
  public E get(int idx);
  public void set(int idx, E value);
  public int size();
  public Iterator<E> iterator();
}
```

What about changing the sequence's size?

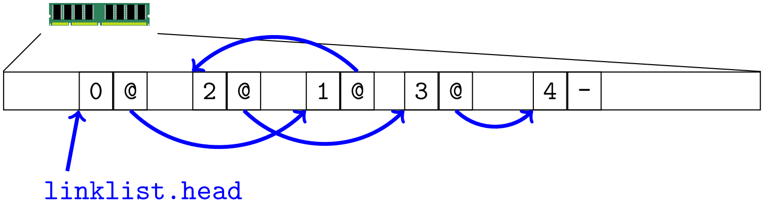
The List ADT

```
public interface List<E>
  extends Sequence<E> // Everything a sequence has, and...
{
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

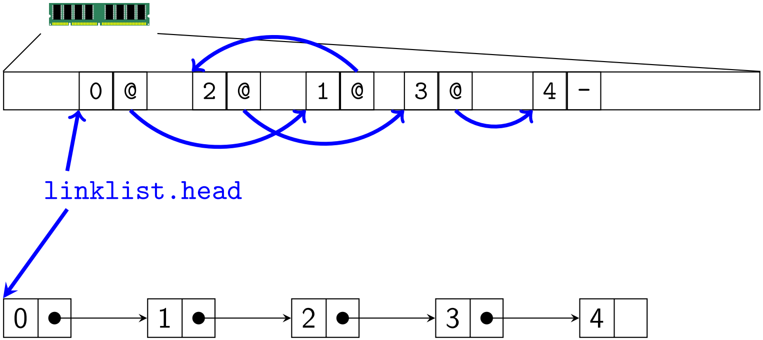
Lists in Other Languages

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

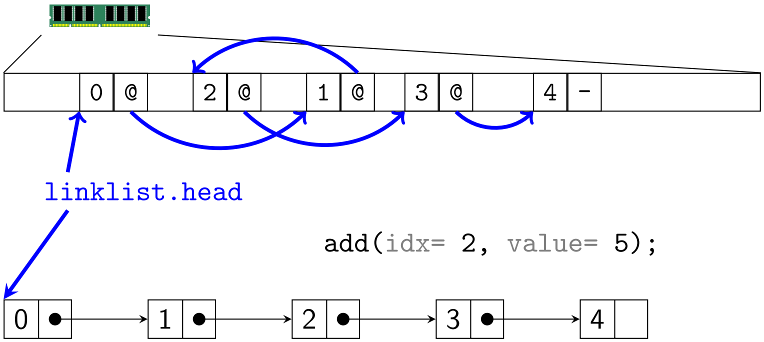
Linked Lists - `add(idx, e)`



Linked Lists - `add(idx, e)`

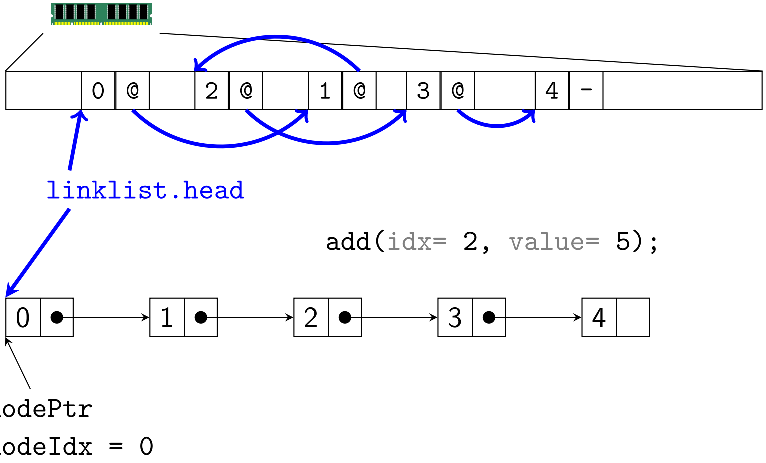


Linked Lists - `add(idx, e)`



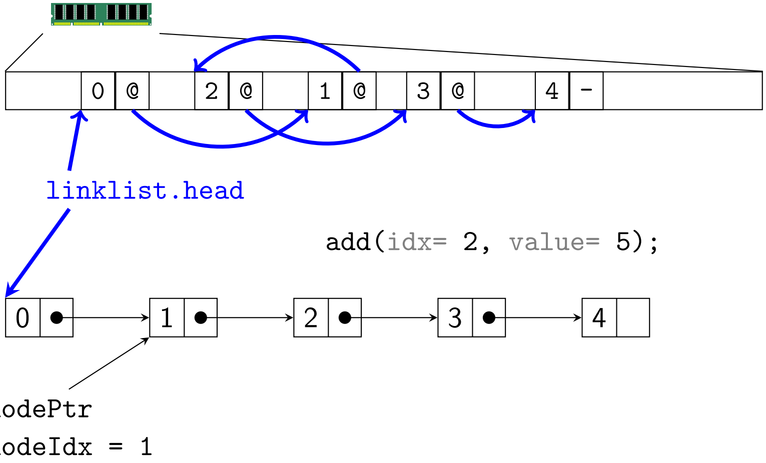
`add(idx= 2, value= 5);`

Linked Lists - add(idx, e)



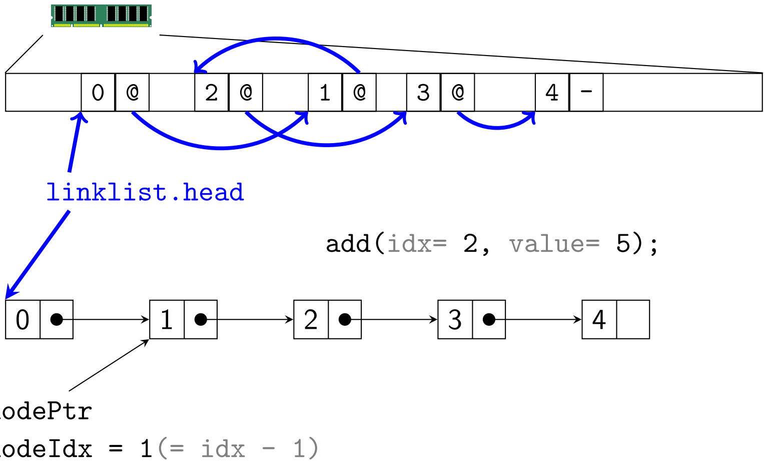
```
add(idx= 2, value= 5);
```

Linked Lists - `add(idx, e)`

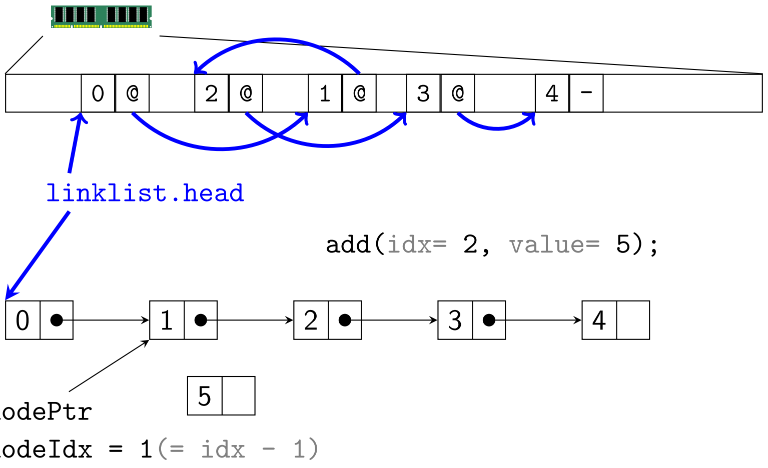


`add(idx= 2, value= 5);`

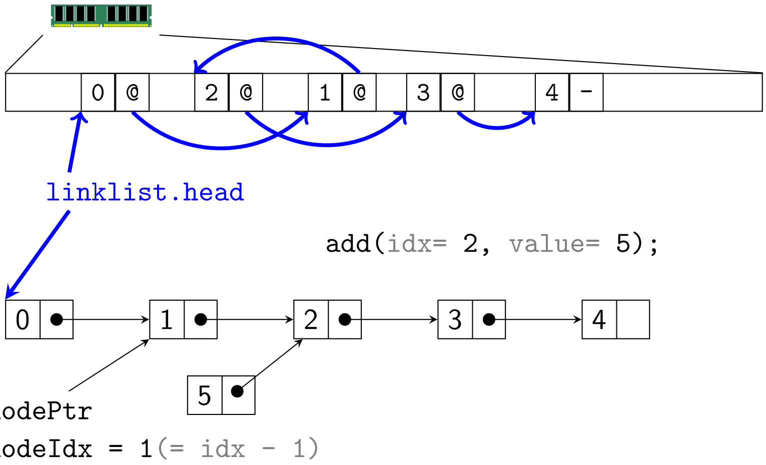
Linked Lists - `add(idx, e)`



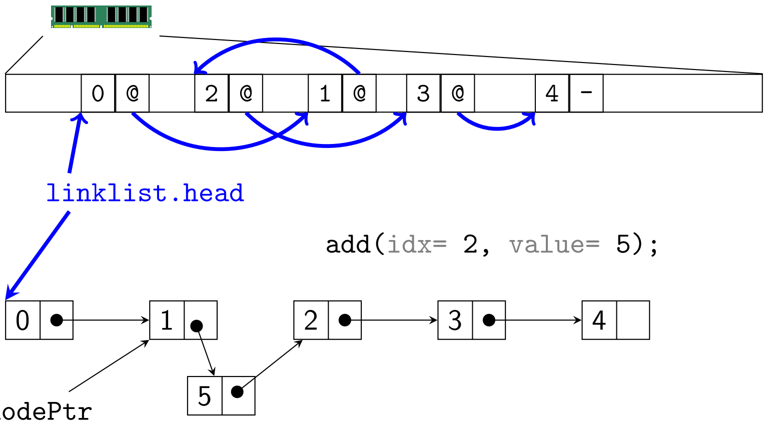
Linked Lists - add(idx, e)



Linked Lists - `add(idx, e)`



Linked Lists - `add(idx, e)`



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

`nodeIdx = 1(= idx - 1)`

Linked Lists - `add(idx, e)`

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

Linked Lists - `add(idx, e)`

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

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.

Linked Lists - `add(idx, e)`

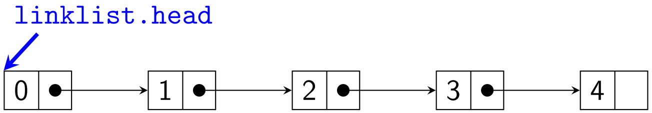
1. Find the node before `idx`.
   $O(N)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Set the new node's next pointer.
   $O(1)$
4. Update the node before `idx`'s next pointer.
   $O(1)$

Linked Lists - `add(idx, e)`

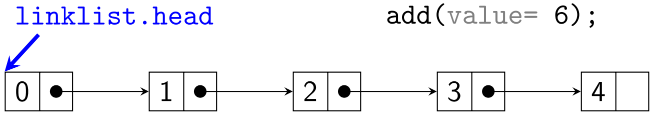
1. Find the node before `idx`.
   $O(N)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Set the new node's next pointer.
   $O(1)$
4. Update the node before `idx`'s next pointer.
   $O(1)$

Total: $O(N)$

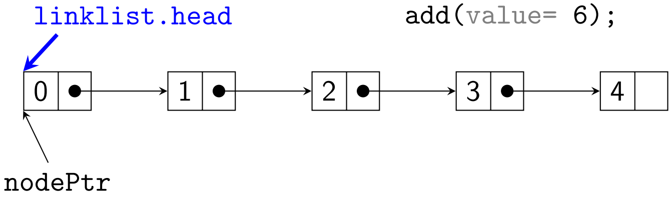
Linked Lists - `add(e)`



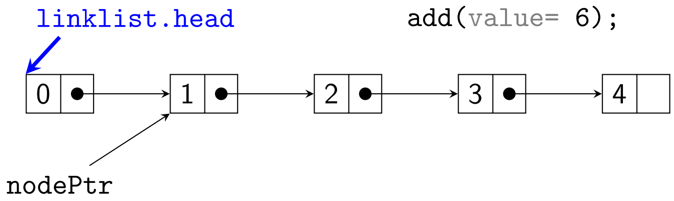
Linked Lists - `add(e)`



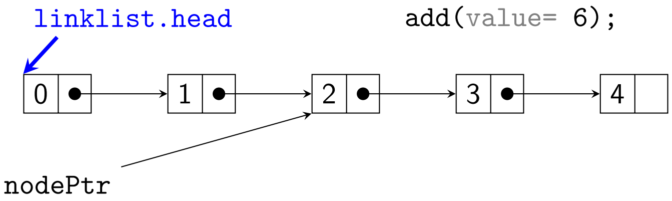
Linked Lists - add(e)



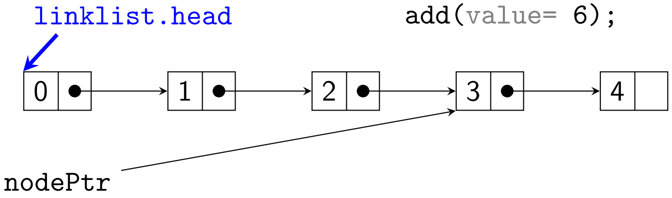
Linked Lists - `add(e)`



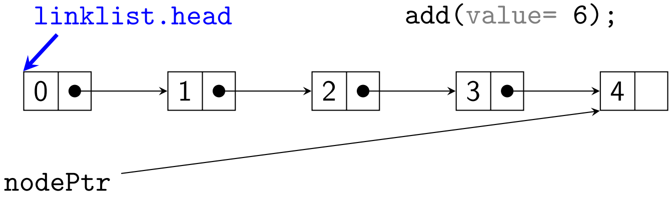
Linked Lists - add(e)



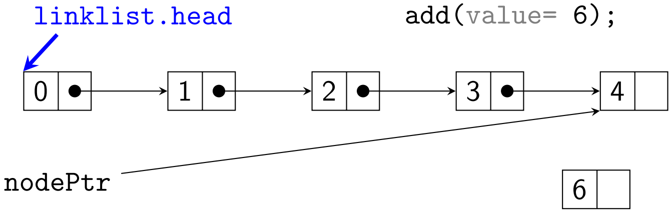
Linked Lists - add(e)



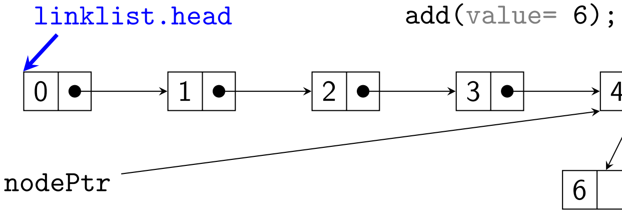
Linked Lists - add(e)



Linked Lists - `add(e)`



Linked Lists - `add(e)`



Linked Lists - `add(e)`

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

Linked Lists - `add(e)`

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

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.

Linked Lists - `add(e)`

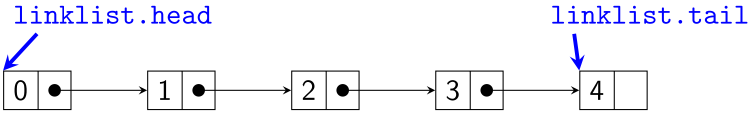
1. Find the last node.
   $O(N)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$

Linked Lists - `add(e)`

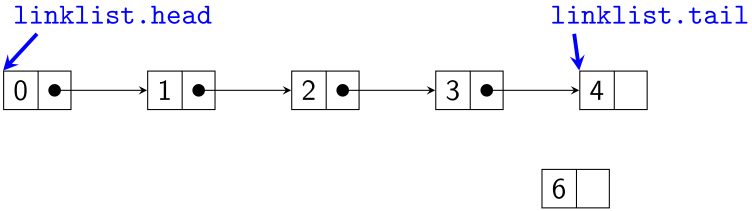
1. Find the last node.
   $O(N)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$

Total: $O(N)$

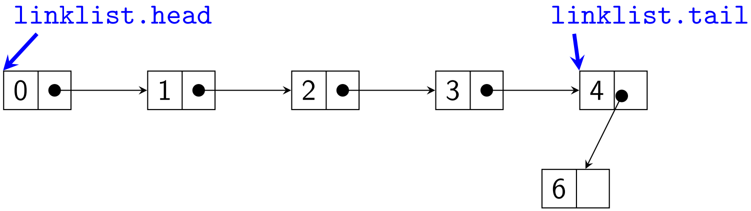
Linked Lists - add(e)



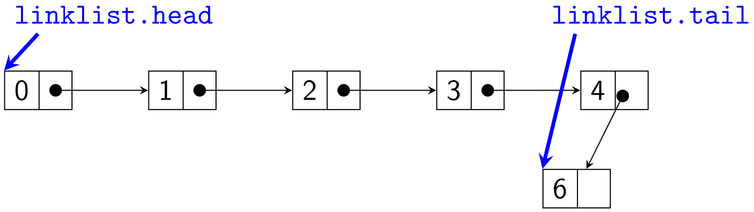
Linked Lists - add(e)



Linked Lists - `add(e)`



Linked Lists - `add(e)`



Linked Lists - `add(e)`

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

Linked Lists - `add(e)`

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

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.
4. Update the tail pointer.

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.

Linked Lists - `add(e)`

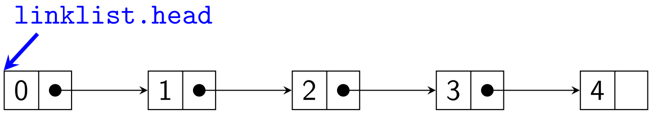
1. Find the last node.
   $O(1)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$
4. Update the tail pointer.
   $O(1)$

Linked Lists - `add(e)`

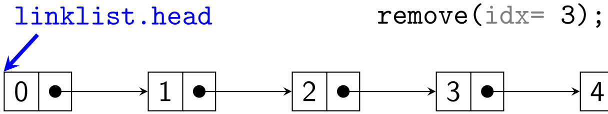
1. Find the last node.
   $O(1)$
2. Allocate a new node and assign its value.
   $O(1)$
3. Update the last node's next pointer.
   $O(1)$
4. Update the tail pointer.
   $O(1)$

Total: $O(1)$

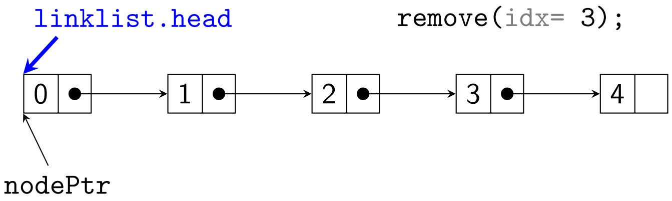
Linked Lists - `remove(idx)`



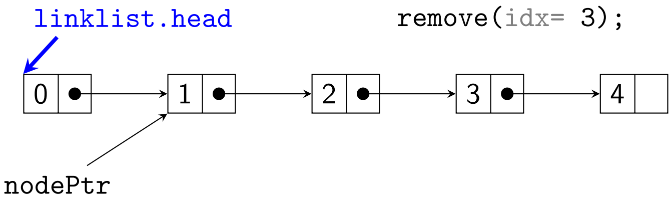
Linked Lists - `remove(idx)`



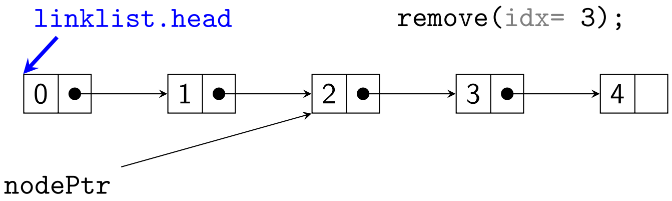
Linked Lists - `remove(idx)`



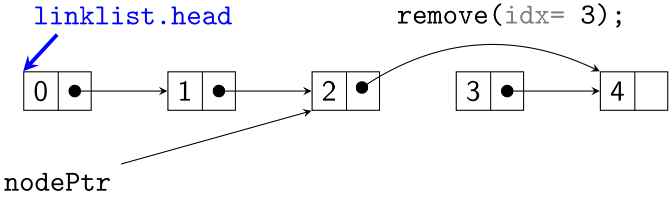
Linked Lists - `remove(idx)`



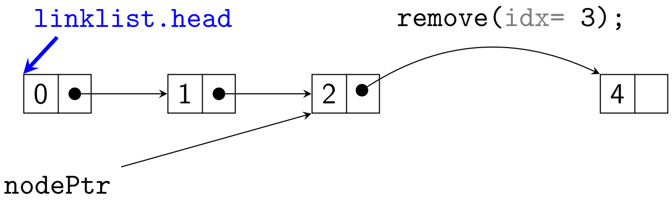
Linked Lists - `remove(idx)`



Linked Lists - `remove(idx)`



Linked Lists - `remove(idx)`



Linked Lists - `remove(idx)`

1. Find the node before `idx`.
2. Update the node before `idx`'s pointer.
3. Allow the node at `idx` to be reclaimed.

Linked Lists - `remove(idx)`

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

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.

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.
   $O(1)$

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

```
    public int sumUpList(LinkedList<Integer> list)
    {
      int total = 0;
      int N = list.size()
      for(i = 0; i < N; i++)
      {
        int value = list.get(i);
        total += value;
      }
      return total;
    }
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  for(i = 0; i < N; i++)
  {
    O(N)
    total += value;
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  for(i = 0; i < N; i++)
  {
    O(N)
    O(1)
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  for(i = 0; i < N; i++)
  {
    O(N)
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  $\sum_{i=0}^{N} O(N)$
  return total;
}
```

Enumeration

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

Enumeration

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

Enumeration

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

Enumeration

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

Enumeration

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

Why is this so expensive?

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  for(i = 0; i < N; i++)
  {
    int value = list.get(i);  ←
    total += value;
  }
  return total;
}
```

We're starting from 0 for each loop.

Can we do better?

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  while(node.isPresent())
  {
    int value = node.get().value;
    total += value;
    node = node.get().next;
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  while(node.isPresent())
  {
    O(1)
    O(1)
    O(1)
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  while(node.isPresent())
  {
    O(1)
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  $\sum_{elem : list} O(1)$
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  $O(N \cdot 1)$
  return total;
}
```

Enumeration

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

Enumeration

```
    public int sumUpList(LinkedList<Integer> list)
    {
      O(N)
    }
```

Enumeration

This code is <u>specialized</u> for `LinkedLists`

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

Enumeration

This code is <u>specialized</u> 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?

Enumeration

This code is specialized for `LinkedList`s

- 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?

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

```
public interface ListIterator<E>
{
  public boolean hasNext();
  public E next();
  public boolean hasPrevious();
  public E previous();
  public void add(E value);
  public void set(E value);
  public void remove();
}
```

ListIterator

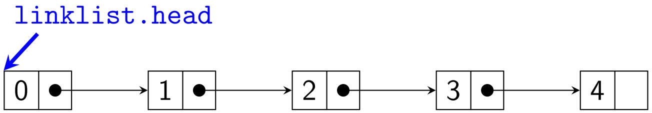
```
public interface ListIterator<E>
{
  public boolean hasNext();
  public E next();
  public boolean hasPrevious();  ← Only in ListIterator
  public E previous();           ← Only in ListIterator
  public void add(E value);      ← Only in ListIterator
  public void set(E value);      ← Only in ListIterator
  public void remove();
}
```

ListIterator adds features to Iterator

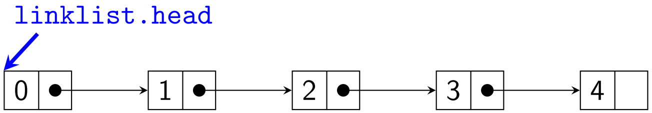
ListIterator (simplified)

```java
public interface ListIterator<E>
{
  public boolean hasNext();
  public E next();
  public void add(E value);
  public void set(E value);
  public void remove();
}
```

ListIterator

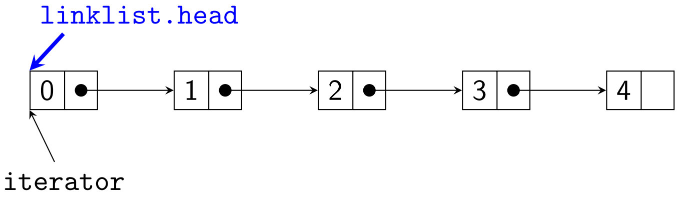


ListIterator



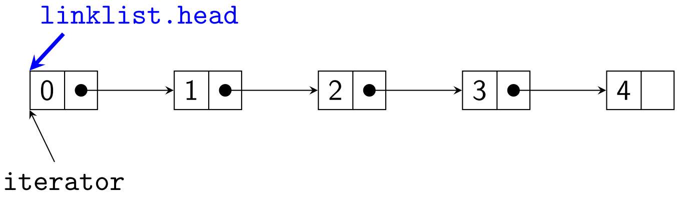
```
list.iterator();
```

ListIterator



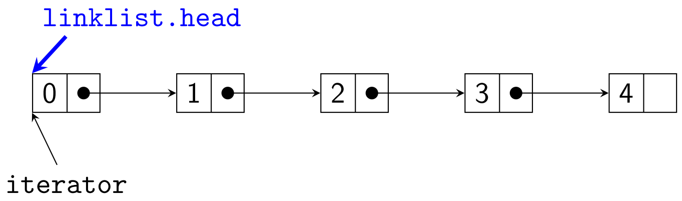
```
list.iterator();
```

ListIterator



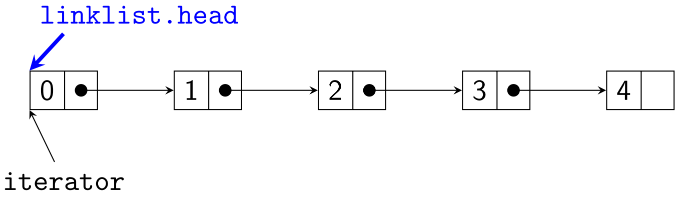
```
list.iterator();
iterator.hasNext();
```

ListIterator



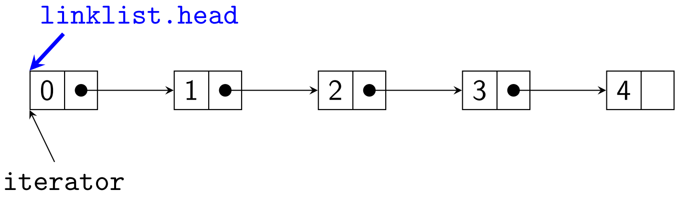
```
list.iterator();
iterator.hasNext(); → true
```

ListIterator



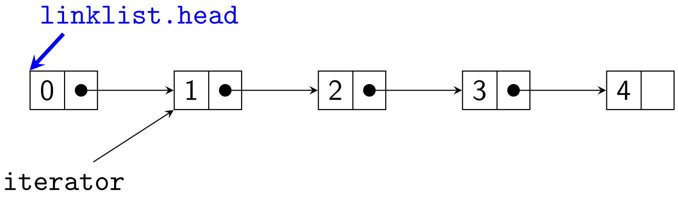
```
list.iterator();
iterator.hasNext();  → true
iterator.next();
```

ListIterator



```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
```

ListIterator

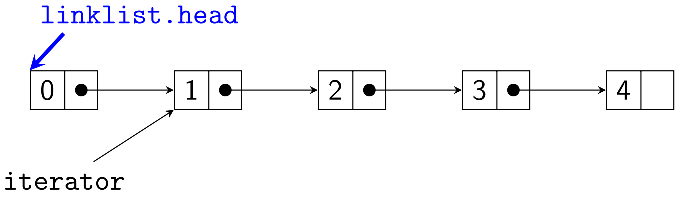


`list.iterator();`

`iterator.hasNext();` $\rightarrow$ `true`

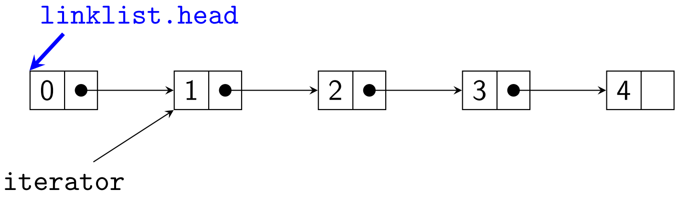
`iterator.next();` $\rightarrow$ 0

ListIterator



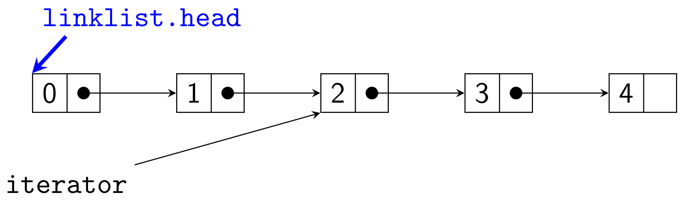
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next();
```

ListIterator



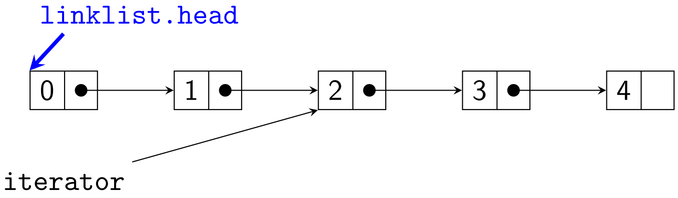
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
```

ListIterator



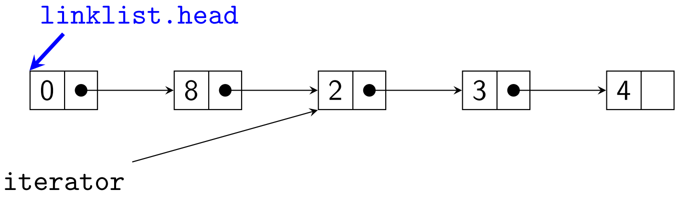
```
list.iterator();
iterator.hasNext();  → true
iterator.next();  → 0
iterator.next();  → 1
```

ListIterator



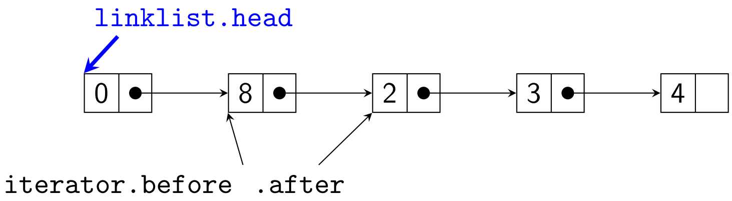
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
```

ListIterator



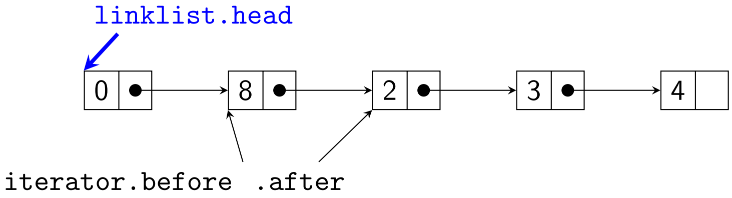
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
```

ListIterator



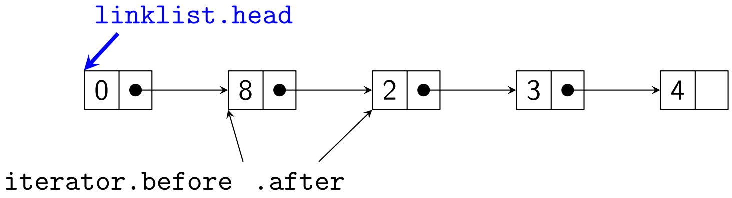
```
list.iterator();
iterator.hasNext();  → true
iterator.next();  → 0
iterator.next();  → 1
iterator.set(8);
```

ListIterator



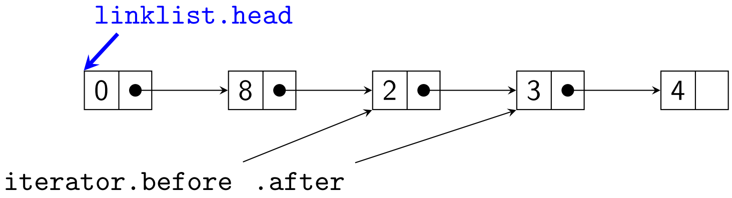
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next();
```

ListIterator



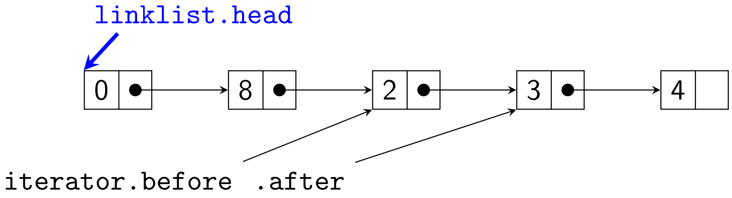
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next(); → 2
```

ListIterator



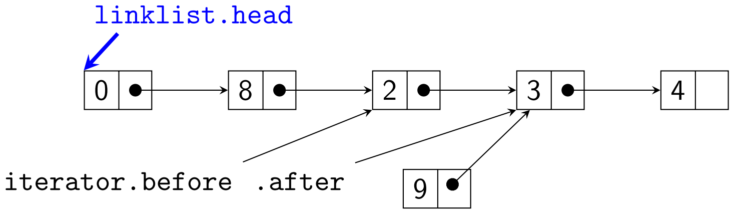
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next(); → 2
```

ListIterator



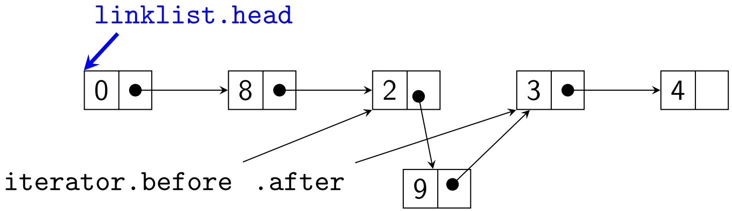
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next(); → 2
iterator.add(9);
```

ListIterator



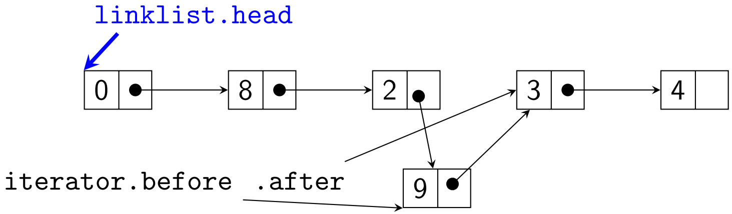
```
list.iterator();
iterator.hasNext(); → true
iterator.next(); → 0
iterator.next(); → 1
iterator.set(8);
iterator.next(); → 2
iterator.add(9);
```

ListIterator



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

ListIterator



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

Implementing LinkedListIterator

```
public class LinkedListIterator<E>
  extends ListIterator<E>
{
  LinkedList<E> list;
  Optional<LinkedListNode<E>> before = Optional.empty();
  Optional<LinkedListNode<E>> after = Optional.of(list.head);
  /* ... */
}
```

Implementing LinkedListIterator

- `public boolean hasNext();`

Implementing LinkedListIterator

- `public boolean hasNext();`
  If `after` is present, return true.

Implementing LinkedListIterator

- `public boolean hasNext();`
  If `after` is present, return true.
  $O(1)$

Implementing LinkedListIterator

- `public boolean hasNext();`
  If `after` is present, return true.
  $O(1)$
- `public E next();`

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.

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.
  $O(1)$

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);`

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`

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 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);`

Implementing LinkedListIterator

- `public void set(E value);`
  Update `before.value`.

Implementing LinkedListIterator

- `public void set(E value);`
  Update `before.value`.
  $O(1)$

Implementing LinkedListIterator

- `public void set(E value);`
  Update `before.value`.
  $O(1)$
- `public void remove();`

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`.

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

```java
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  LinkedListIterator<Integer> iterator = list.iterator();
  while(iterator.hasNext())
  {
    int value = iterator.next();
    total += value;
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  LinkedListIterator<Integer> iterator = list.iterator();
  while(iterator.hasNext())
  {
    O(1)
    O(1)
  }
  return total;
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  LinkedListIterator<Integer> iterator = list.iterator();
  $\sum_{elem:list} O(1)$
  return total;
}
```

Enumeration

```
    public int sumUpList(LinkedList<Integer> list)
    {
      int total = 0;
      int N = list.size()
      LinkedListIterator<Integer> iterator = list.iterator();
      O(N · 1)
      return total;
    }
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  $O(1)$
  $O(1)$
  $O(1)$
  $O(N \cdot 1)$
  $O(1)$
}
```

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  O(N)
}
```

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?

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 $N + 1$.

`add(e)` on an Array

- Allocate a new array of size $N + 1$.
  $O(1)$

add(e) on an Array

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

add(e) on an Array

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

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.

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.
  $O(1)$

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?

`add(e)` on an Array

Can we do better?
Next Class!