Data Structures

Iterators and Chain Linear List

Andres Mendez-Vazquez

August 18, 2016

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Iterator Method

Scenario

We often want to access every item in a data structure or collection in turn...

Example with the array representation

```
int listSize = SomeList.size();
for (int i = 0; i < listSize; i++)
        System.out.println(SomeList.get(i));
```

Iterator Method

Scenario

We often want to access every item in a data structure or collection in turn...

Example with the array representation

```
int listSize = SomeList.size();
for (int i = 0; i < listSize; i++)
        System.out.println(SomeList.get(i));
```

Motivation

Question

What if we have a **get** that is really complex?

For example the get in a Chain List

We will see that it has complexity $O(n)$!!!

Previous Code

What a waste of time!!! $O\left(n^2\right)$**!!!!**

Motivation

Question

What if we have a **get** that is really complex?

For example the get in a Chain List

We will see that it has complexity $O(n)$!!!

Previous Code

What a waste of time!!! $O\left(n^2\right)$**!!!!**

Motivation

Question

What if we have a **get** that is really complex?

For example the get in a Chain List

We will see that it has complexity $O(n)$!!!

Previous Code

What a waste of time!!! $O\left(n^2\right)$**!!!!**

Then

Something Notable

Iteration is such a common operation that we could include it as part of the ADT list.

However

We do not want to add another operation to the ADT each time we think of another way to use an iteration.

Then

Something Notable

Iteration is such a common operation that we could include it as part of the ADT list.

However

We do not want to add another operation to the ADT each time we think of another way to use an iteration.

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

In Java at java.util we have the interface Iterator

Interface

```
package java.util;
public interface Iterator<Item> {

public boolean hasNext();

public Item next();

// Optional method
public void remove();

} // end Iterator
```

Explanation

public boolean hasNext()

- It detects whether this iterator has completed its traversal and gone beyond the last entry in the collection of data.
- It returns true if the iterator has another entry to return.

Explanation

public boolean hasNext()

- It detects whether this iterator has completed its traversal and gone beyond the last entry in the collection of data.
- It returns true if the iterator has another entry to return.

public T next()

- It retrieves the next entry in the collection and advances this iterator by one position.
- It returns a reference to the next entry in the iteration, if one exists
- It throws NoSuchElementException if the iterator had reached the end already, that is, if hasNext() is false.

Explanation

public boolean hasNext()

- It detects whether this iterator has completed its traversal and gone beyond the last entry in the collection of data.
- It returns true if the iterator has another entry to return.

public T next()

- It retrieves the next entry in the collection and advances this iterator by one position.
- It returns a reference to the next entry in the iteration, if one exists
- It throws NoSuchElementException if the iterator had reached the end already, that is, if hasNext() is false.

Explanation

public boolean hasNext()

- It detects whether this iterator has completed its traversal and gone beyond the last entry in the collection of data.
- It returns true if the iterator has another entry to return.

public T next()

- It retrieves the next entry in the collection and advances this iterator by one position.
- It returns a reference to the next entry in the iteration, if one exists
- It throws NoSuchElementException if the iterator had reached the end already, that is, if hasNext() is false.

Explanation

public boolean hasNext()

- It detects whether this iterator has completed its traversal and gone beyond the last entry in the collection of data.
- It returns true if the iterator has another entry to return.

public T next()

- It retrieves the next entry in the collection and advances this iterator by one position.
- It returns a reference to the next entry in the iteration, if one exists
- It throws NoSuchElementException if the iterator had reached the end already, that is, if hasNext() is false.

Explanation

public void remove()

- It removes from the collection of data the last entry that next() returned.

  Precondition: next() has been called, and remove() has not been called since then.

  - It throws IllegalStateException if next() has not been called, or if remove() was called already after the last call to next().

Explanation

public void remove()

- It removes from the collection of data the last entry that next() returned.

  Precondition: next() has been called, and remove() has not been called since then.

  - It throws IllegalStateException if next() has not been called, or if remove() was called already after the last call to next().

Explanation

public void remove()

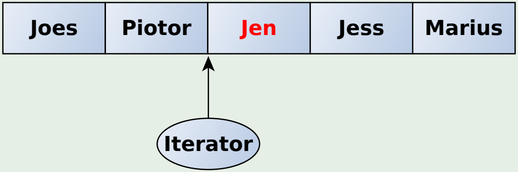
- It removes from the collection of data the last entry that next() returned.

  Precondition: next() has been called, and remove() has not been called since then.

  - It throws IllegalStateException if next() has not been called, or if remove() was called already after the last call to next().

Effect in a list

The List Before Any Operation with the Iterator

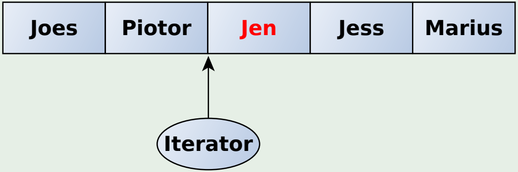


Joes	Piotor	Jen	Jess	Marius

After the next() operation

Effect in a list

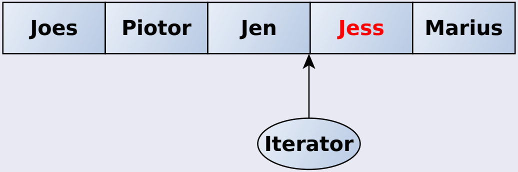
The List Before Any Operation with the Iterator



Joes	Piotor	Jen	Jess	Marius

Iterator

After the next() operation

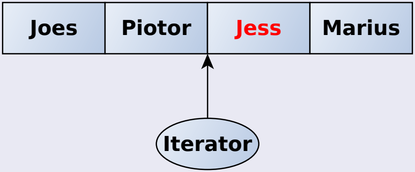


Joes	Piotor	Jen	Jess	Marius

Iterator

Effect in a list

Now, we issue a remove()



Joes

Piotor

Jess

Marius

Iterator

NO FREE LUNCH!!!

Remark

Some details of using an iterator depend on the approach used to implement the iterator methods.

NO FREE LUNCH!!!

Remark

Some details of using an iterator depend on the approach used to implement the iterator methods.

We can do the following

- A possible, but not optimal, way to provide an ADT with traversal operations is to define them as ADT operations.
  - For example, if **ListInterface extends Iterator**, a list object would have iterator methods as well as list methods.
  - PROBLEM!!! What if you want to have two or more iterators over the object list making different things at the same time!!!

NO FREE LUNCH!!!

Remark

Some details of using an iterator depend on the approach used to implement the iterator methods.

We can do the following

- A possible, but not optimal, way to provide an ADT with traversal operations is to define them as ADT operations.
  - For example, if **ListInterface extends Iterator**, a list object would have iterator methods as well as list methods.
  - PROBLEM!!! What if you want to have two or more iterators over the object list making different things at the same time!!!

NO FREE LUNCH!!!

Remark

Some details of using an iterator depend on the approach used to implement the iterator methods.

We can do the following

- A possible, but not optimal, way to provide an ADT with traversal operations is to define them as ADT operations.
  - For example, if **ListInterface extends Iterator**, a list object would have iterator methods as well as list methods.
  - PROBLEM!!! What if you want to have two or more iterators over the object list making different things at the same time!!!

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator
2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?
3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Solution

We can have

- Separate class iterator
- Inner Class Iterator

Solution

We can have

- Separate class iterator
- Inner Class Iterator

For a separate class iterator, we have stuff like this

Iterator<String> nameIterator = new SeparateIterator<String>(nameList);

Solution

We can have

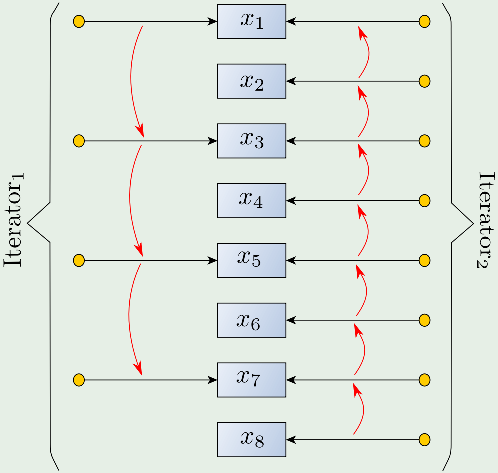
- Separate class iterator
- Inner Class Iterator

For a separate class iterator, we have stuff like this

```
Iterator<String> nameIterator = new
SeparateIterator<String>(nameList);
```

Example

This allows us to build iterators that go slower of faster over the list



Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Multiple Iterators

Thus

External Iterators provides a solution to multiple iterators

However

Huge Problem

We can only access the list through the public methods:

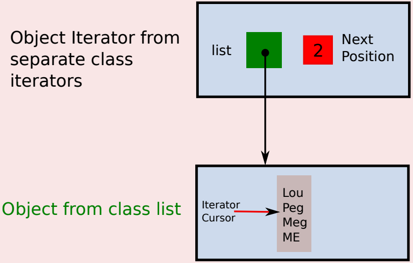
- What if they are not enough?

Multiple Iterators

Thus

External Iterators provides a solution to multiple iterators

However



Huge Problem

We can only access the list through the public methods:

- What if they are not enough?

Separate Iterator Code

Code

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SeparateIterator<Item>
                 implements Iterator<Item> {

private LinearList<Item> list;
// position of entry last returned by next()
private int nextPosition;
// needed by remove
private boolean wasNextCalled;

public SeparateIterator(ListList<Item> aList) {
// All that it implies
} // end constructor

< Implementations of the methods hasNext ,...
next , and remove go here > . . .
} // end SeparateIterator
```

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

We need a better solution!!!

Declare a inner class

So you can have direct access to the protected or private elements in the object!!!

Actually you have something like this

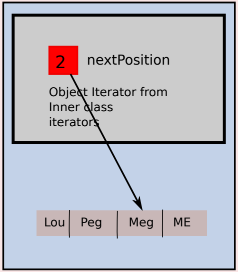
We need a better solution!!!

Declare a inner class

So you can have direct access to the protected or private elements in the object!!!

Actually you have something like this

Object from class array list



Inner Class Code

Code

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArrayListWithIterator<T>
             implements ListWithIteratorInterface<T> {

// ALL the code for array linear list

private class IteratorForArrayList
             implements Iterator<T> {
private int nextIndex;
private boolean wasNextCalled;

private IteratorForArrayList() {
        nextIndex = 0;
        wasNextCalled = false;
} // end default constructor

< Implementations of the methods in the
interface Iterator go here > . . . }
// end IteratorForArrayList

} // end  ArrayListWithIterator
```

So, we have...

Advantages

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation
- It can lead to more readable and maintainable code.

So, we have...

Advantages

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation
- It can lead to more readable and maintainable code.

Example

```
ArrayListWithIterator<Integers> SOME = new
ArrayListWithIterator<Integers>();
ArrayListWithIterator.IteratorForArrayList fl = SOME. new
IteratorForArrayList();
```

So, we have...

Advantages

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation
- It can lead to more readable and maintainable code.

Example

```
ArrayListWithIterator<Integers> SOME = new
ArrayListWithIterator<Integers>();
ArrayListWithIterator.IteratorForArrayList fl = SOME. new
IteratorForArrayList();
```

So, we have...

Advantages

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation
- It can lead to more readable and maintainable code.

Example

```
ArrayListWithIterator<Integers> SOME = new
ArrayListWithIterator<Integers>();
ArrayListWithIterator.IteratorForArrayList fl = SOME. new
IteratorForArrayList();
```

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator
2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?
3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Linked Representation

Storage

List elements are stored, in memory, in an arbitrary order.

How you move through it

Explicit information (called a link) is used to go from one element to the next.

Linked Representation

Storage

List elements are stored, in memory, in an arbitrary order.

How you move through it

Explicit information (called a link) is used to go from one element to the next.

Outline

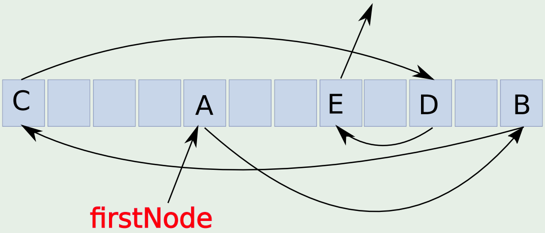
1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Memory Layout

We have the following



First

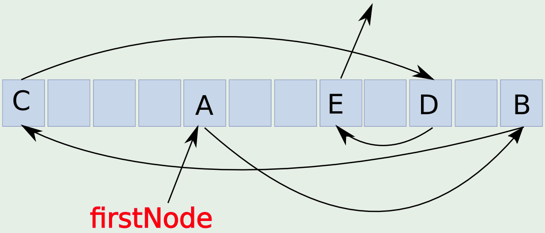
Last pointer (or link) from E is null

Thus

You can use a variable firstNode to get to the first element in the Linked List.

Memory Layout

We have the following



First

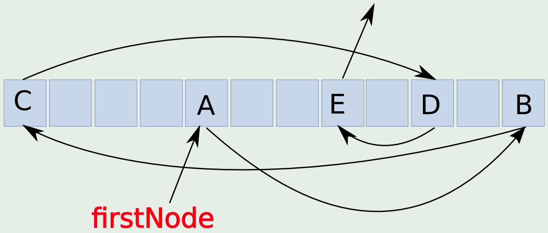
Last pointer (or link) from E is null

Thus

You can use a variable firstNode to get to the first element in the Linked List.

Memory Layout

We have the following



First

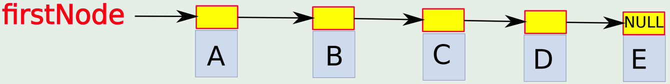
Last pointer (or link) from E is null

Thus

You can use a variable firstNode to get to the first element in the Linked List.

Normal Way To Draw A Linked List

Example



🟨 Link or pointer field of node

▭ Data field of node

Actually, we know this as a Chain

First

A chain is a linked list in which each node represents one element.

Second

There is a link or pointer from one element to the next.

Third

The last node has a null pointer.

Actually, we know this as a Chain

First

A chain is a linked list in which each node represents one element.

Second

There is a link or pointer from one element to the next.

Third

The last node has a null pointer.

Actually, we know this as a Chain

First

A chain is a linked list in which each node represents one element.

Second

There is a link or pointer from one element to the next.

Third

The last node has a null pointer.

Code for ChainNode

Code

```
package LinearList;
//Using Generic in Java
class ChainNode<Item>{
   // package data members
   protected Item element;
   protected ChainNode next;
   // constructors and methods come here
}
```

The Constructors for ChainNode

A classic one

```
package LinearList;
//Using Generic in Java
public ChainNode(){
   // Set the next node
   this.element = null;
   this.next = null;
}

public ChainNode(Item elem){
   // Set element
    this.element = elem;
    this.next = null;
}
```

What about the Class Chain?

Code

```
/** linked implementation of LinearList */
package infrastructures;
import java.util.*;  // has Iterator
public class Chain<Item> implements LinearList<Item>
{
// data members
protected ChainNode<Item> firstNode;
protected int size;

// methods of Chain come here
}
```

In addition!!! Constructors!!!

Code

```
public Chain<Item> (int initialCapacity)
{

// the default initial values of firstNode and size
// are null and 0, respectively

}
```

A simple example

```
public Chain<Item> ()
    {
        this.firstNode = new ChainNode<Item>();
        this.size = 0;
    }
```

In addition!!! Constructors!!!

Code

```
public Chain<Item> (int initialCapacity)
{

// the default initial values of firstNode and size
// are null and 0, respectively

}
```

A simple example

```
public Chain<Item> ()
    {
       this.firstNode = new ChainNode<Item>();
       this.size = 0;
    }
```

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

IsEmpty()

Really Simple Code

```
/** @return true iff list is empty */
   public boolean isEmpty()
      {return size == 0;}
```

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator
2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?
3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

size()

Really Simple Code

```
/** @return current number of elements in list */
    public int size()
        {return size;}
```

Outline

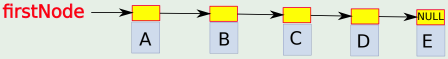
1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Operations: Get

Example



🟨 Link or pointer field of node

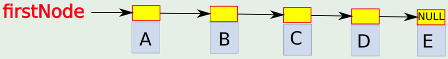
⬜ Data field of node

What to do to implement Get

1. Check Index
2. Move to the correct position
   - For example: TempNode = firstNode.next.next.next.
   - You actually use a loop.
3. return TempNode.element

Operations: Get

Example



🟨 Link or pointer field of node

▨ Data field of node

What to do to implement Get

1. Check Index
2. Move to the correct position
   - For example: TempNode = firstNode.next.next.next.
   - You actually use a loop.
3. return TempNode.element

Process

Check Index

$$0 \leq index \leq size - 1 \tag{1}$$

Negate the statement to use it

How?

Process

Check Index

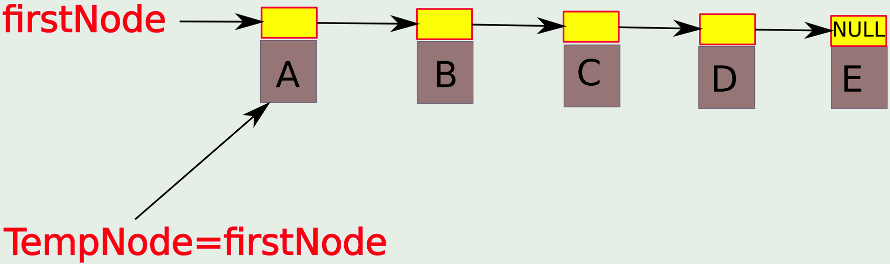
$$0 \leq index \leq size - 1 \tag{1}$$

Negate the statement to use it

How?

Move to the correct place

How?



firstNode

NULL

A

B

C

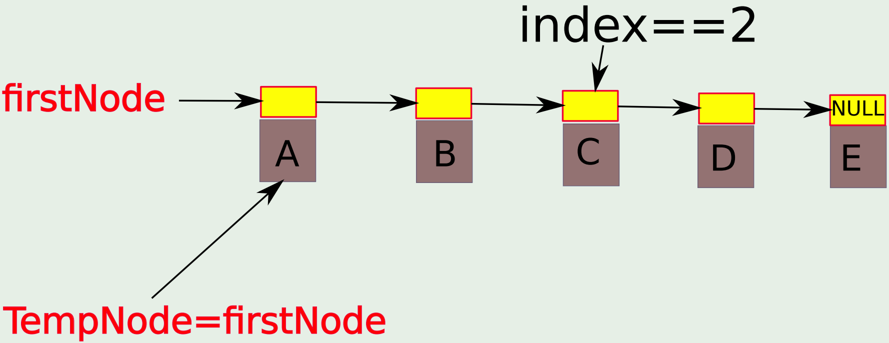
D

E

TempNode=firstNode

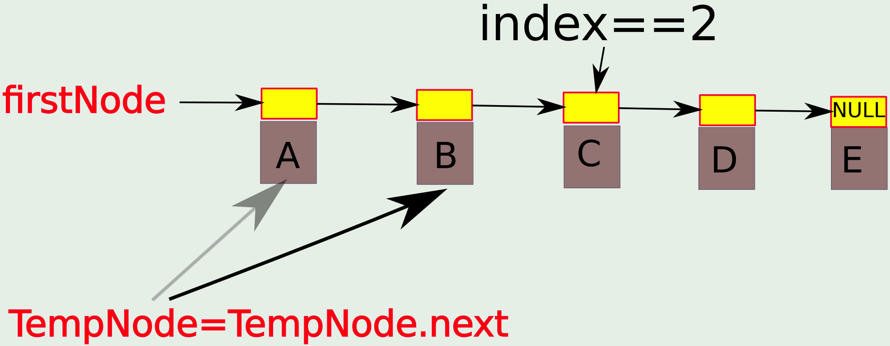
Move to the correct place

Index == 2



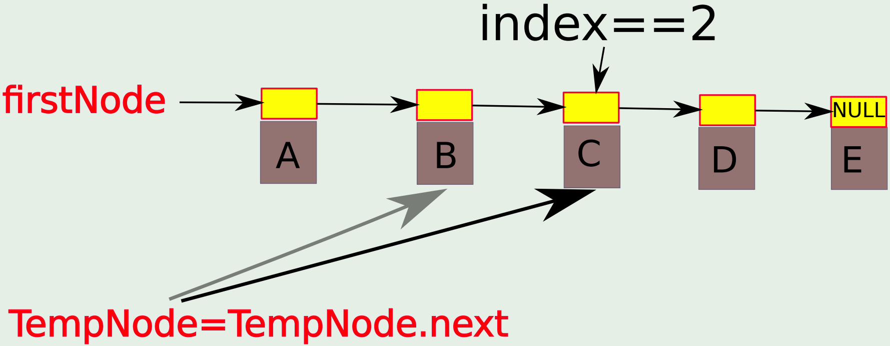
Move to the correct place

Index == 2



Move to the correct place

Index == 2



index==2

firstNode

NULL

A

B

C

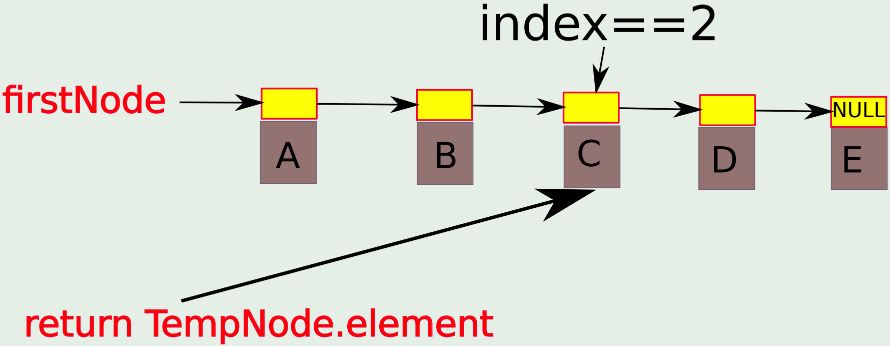
D

E

TempNode=TempNode.next

Move to the correct place

Index == 2



Code

Here is the method

```
public <Item> get(int index){
     ChainNode<Item> TempNode;
     //Check always
     if (this.size == 0) return null;
     if (index<0 || index>this.size -1){
          System.out.println("Index␣out␣of␣bound");
          System.exit(0);
     }
     //Move to the correct position
     TempNode = this.firstNode;
     for(int i=0 ; i<index ; i++){
          TempNode = TempNode.next;
     }
     return TempNode.element;
}
```

Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Now, Remove

We have two cases

Any ideas?

Case I

Removing from the beginning.

Case II

Remove from the middle.

Now, Remove

We have two cases

Any ideas?

Case I

Removing from the beginning.

Case II

Remove from the middle.

Now, Remove

We have two cases

Any ideas?

Case I

Removing from the beginning.

Case II

Remove from the middle.

Case I: Remove(0)

Process

```
1 if index == 0 do
    1 TempNode = firstNode
    2 firstNode=firstNode.next
    3 TempNode.next=NULL
    4 return TempNode.element
```

Case I: Remove(0)

Process

```
1. if index == 0 do
    1. TempNode = firstNode
    2. firstNode=firstNode.next
    3. TempNode.next=NULL
    4. return TempNode.element
```

Case I: Remove(0)

Process

```
1 if index == 0 do
    1 TempNode = firstNode
    2 firstNode=firstNode.next
    3 TempNode.next=NULL
    4 return TempNode.element
```

Case I: Remove(0)

Process

```
1 if index == 0 do
    1 TempNode = firstNode
    2 firstNode=firstNode.next
    3 TempNode.next=NULL
    4 return TempNode.element
```

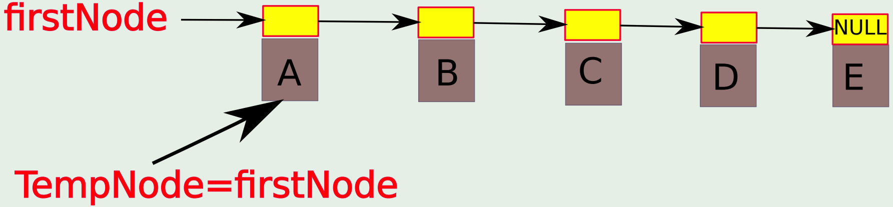
Case I: Remove(0)

Process

```
1 if index == 0 do
    1 TempNode = firstNode
    2 firstNode=firstNode.next
    3 TempNode.next=NULL
    4 return TempNode.element
```

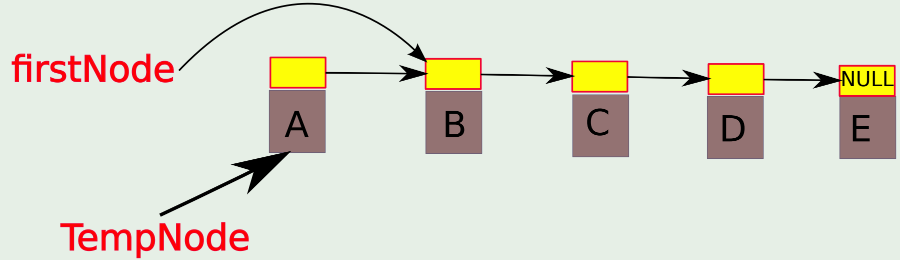
Process

TempNode=firstNode



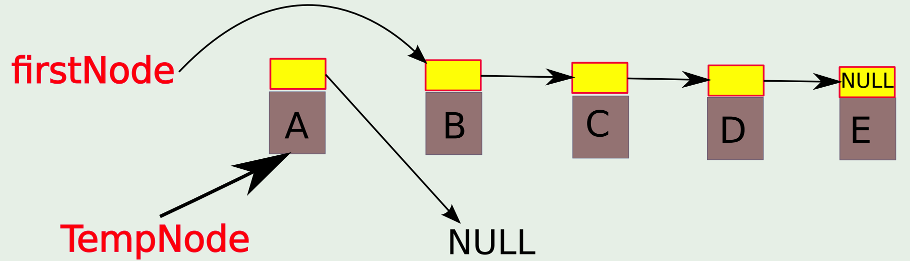
Process

firstNode=firstNode.next



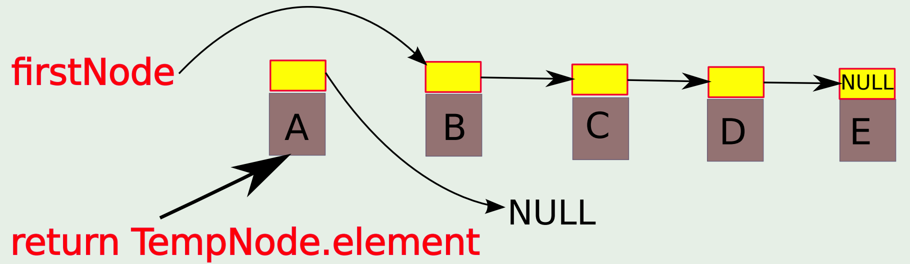
Process

TempNode.next=NULL



Process

return TempNode.element



Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

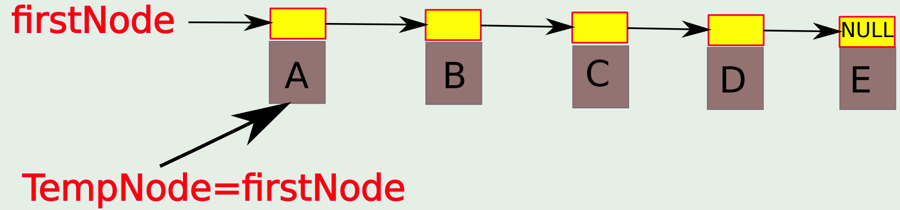
Case II: Remove(2)

Process

1. Check Index
2. Starting at the first node
3. TempNode=firstNode;
4. Loop doing
    1. beforeNode=TempNode;
    2. TempNode=TempNode.next;
5. before.next = TempNode.next
6. TempNode.next = NULL
7. return TempNode.element

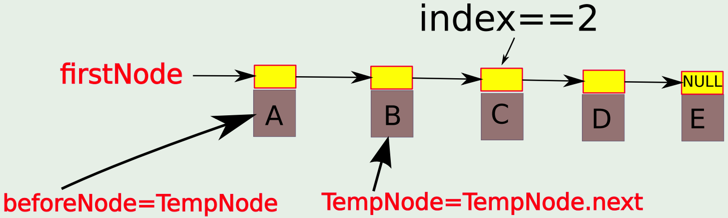
Process

TempNode=firstNode



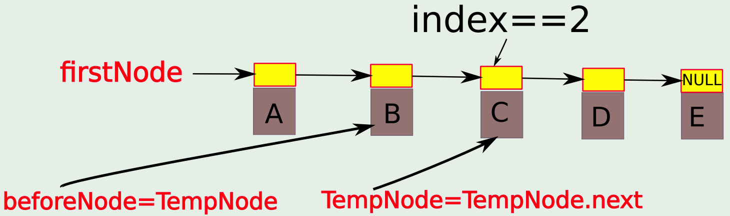
Process: index == 2

```
beforeNode=TempNode;
TempNode=TempNode.next;
```



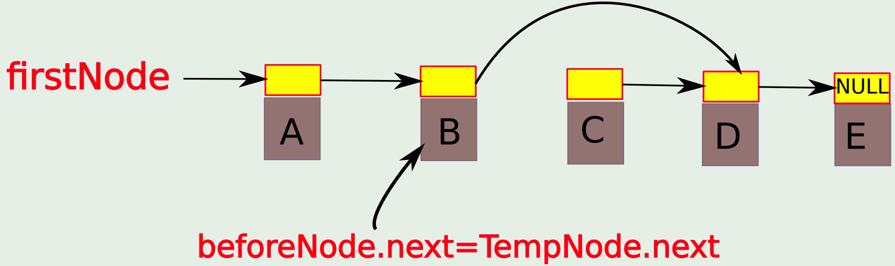
Process: index == 2

beforeNode=TempNode;
TempNode=TempNode.next;



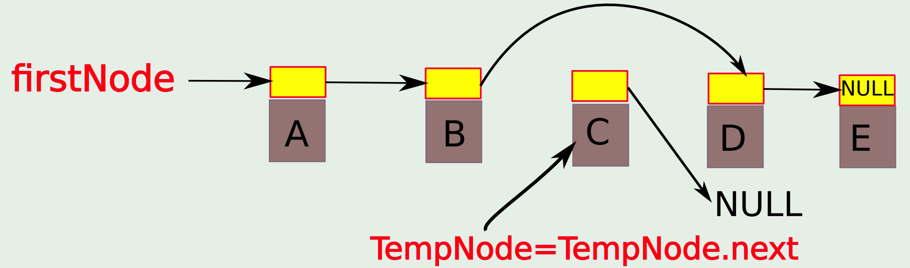
Process: index == 2

before.next = TempNode.next



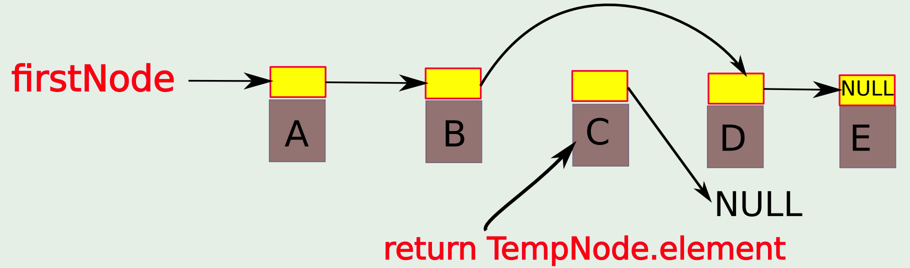
Process: index == 2

TempNode.next = NULL



Process: index == 2

TempNode.next = NULL



Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator
2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?
3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Now, Add

We have four cases

Any ideas?

Case I

The Chain is Empty

Case II

Add at the beginning.

Now, Add

We have four cases

Any ideas?

Case I

The Chain is Empty

Case II

Add at the beginning.

Now, Add

We have four cases

Any ideas?

Case I

The Chain is Empty

Case II

Add at the beginning.

Now, Add

We have four cases

Any ideas?

Case III

Add at the middle.

Case IV

Add at the end.

Now, Add

We have four cases

Any ideas?

Case III

Add at the middle.

Case IV

Add at the end.

Now, Add

We have four cases

Any ideas?

Case III

Add at the middle.

Case IV

Add at the end.

Why?

Think About it!!!

Remember you have a sequential access.

Add at an empty list

Steps

1. Create a node to store the element, NewNode
2. Then assign firstNode = NewNode

Add at an empty list

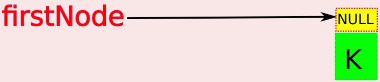
Steps

1. Create a node to store the element, NewNode
2. Then assign firstNode = NewNode

Add at an empty list

Steps

1. Create a node to store the element, NewNode
2. Then assign firstNode = NewNode



Add At the Beginning

Steps

1. Create a node to store the element, NewNode
2. Do NewNode.next = firstNode
3. Then reassign firstNode = NewNode
4. Then, size = size +1

Add At the Beginning

Steps

1. Create a node to store the element, NewNode
2. Do NewNode.next = firstNode
3. Then reassign firstNode = NewNode
4. Then, size = size +1

Add At the Beginning

Steps

1. Create a node to store the element, NewNode
2. Do NewNode.next = firstNode
3. Then reassign firstNode = NewNode
4. Then, size = size +1

Add At the Beginning

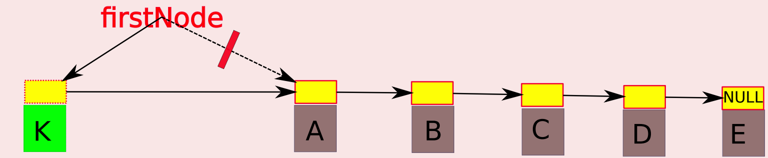
Steps

1. Create a node to store the element, NewNode
2. Do NewNode.next = firstNode
3. Then reassign firstNode = NewNode
4. Then, size = size +1

Add At the Beginning

Steps

1. Create a node to store the element, NewNode
2. Do NewNode.next = firstNode
3. Then reassign firstNode = NewNode
4. Then, size = size +1



What about the other case?

Do you have an idea?

- Add at the Middle.
- Add at the end.

What about the other methods?

Linear List

```
AbstractDataType LinearList
{
   instances
      ordered finite collections of zero or more elements
   operations
      isEmpty(): return true iff the list is empty, false otherwise
      size(): return the list size (i.e., number of elements in the list)
      get(index): return the element with the “index” index
      indexOf(x): return the index of the first occurrence of x in the list, return -1
                  if x is not in the list
      remove(index): remove and return the indexth element, elements with higher
                     index have their index reduced by 1
      add(theIndex, x): insert x as the index of th element, elements with
                        theIndex ≥ index have their index increased by 1
      output(): output the list elements from left to right
}
```

Complexity of Linked List

We have

	Dynamic Array Amortized Analysis	Linked List
Indexing	$O(1)$	$O(n)$
Search	$O(n)$	$O(n)$
Add/Remove at the beginning	$O(n)$	$O(1)$
Add/Remove at the middle	$O(n)$	Search Time $+O(1)$
Space Complexity	$O(n)$	$O(n)$

Average Performance with Each Implementation on a Slow Machine!!!

40,000 Operations each type on a 350Mhz PC

Operation	FastArrayLinearList	Chain
average get	5.6 ms	157 sec
average adds	5.8 sec	115 sec
average removes	5.8 sec	157 sec

Outline

1 Iterators
- The Method
- Interface
- Separate and Inner Class Iterator
  - Separate Class Iterator
  - Inner Class Iterator

2 Linked Representation
- Introduction
- Memory Layout
- Operations
  - IsEmpty()
  - size()
  - Get(Index)
  - Remove
  - Add
- Can we simplify the code?

3 Other data structures based in the Linked List
- Circular List
- Doubly Linked List
- Where are all these things?

Can we simplify the code?

How?

We would like to simplify the code.... How?

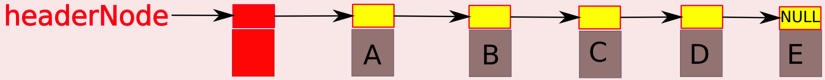
What about

Can we simplify the code?

How?

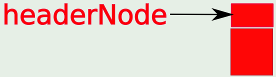
We would like to simplify the code.... How?

What about



Can we simplify the code?

Then, the empty list looks like

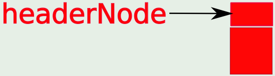


Thus

This simplify a lot the code because everything is a middle node

Can we simplify the code?

Then, the empty list looks like



Thus

This simplify a lot the code because everything is a middle node

Outline

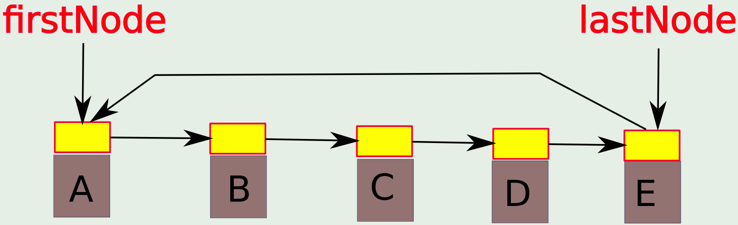
1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Circular List

Circular List



Here, we need to have the following changes

We need to add the following to the class

```
/** linked implementation of LinearList */
package infrastructures;
import java.util.*;  // has Iterator
public class Chain<Item> implements LinearList<Item>
{
// data members
protected ChainNode<Item> firstNode;
protected ChainNode<Item> lastNode;
protected int size;

// methods of Chain come here
}
```

Example of add(index) at the end

Process

- Check if (index==size-1)

Next

Make the lastNode.next = TempNode

Then

TempNode.next = firstNode

Example of add(index) at the end

Process

- Check if (index==size-1)

Next

Make the lastNode.next = TempNode

Then

TempNode.next = firstNode

Example of add(index) at the end

Process

- Check if (index==size-1)

Next

Make the lastNode.next = TempNode

Then

TempNode.next = firstNode

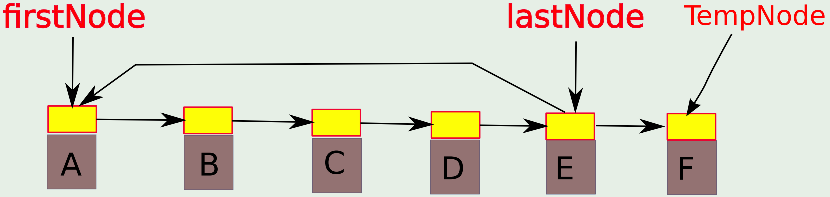
Example of add(index) at the end

Finally

lastNode = TempNode

Circular List

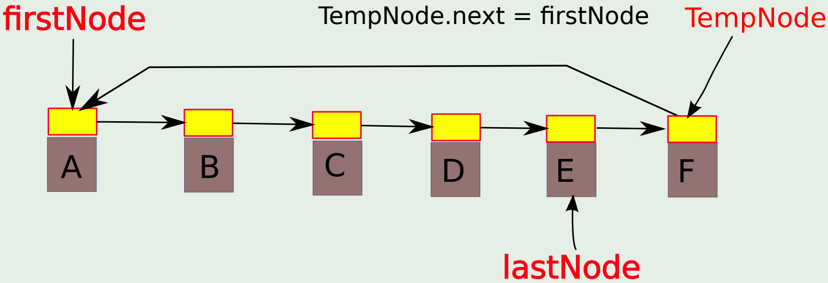
Add at the end



lastNode.next = TempNode

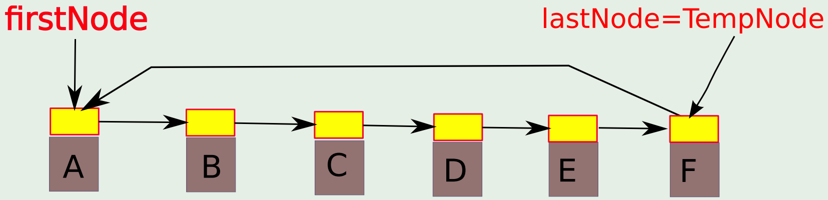
Circular List

Add at the end



Circular List

Add at the end



We have other representations for the lists

Doubly Linked List

- You have two chain list going through the nodes.
- It has a firstNode
- It has a lastNode

Doubly Linked Circular List

- You have two chain list going through the nodes.
- It has a firstNode

We have other representations for the lists

Doubly Linked List

- You have two chain list going through the nodes.
- It has a firstNode
- It has a lastNode

Doubly Linked Circular List

- You have two chain list going through the nodes.
- It has a firstNode

Outline

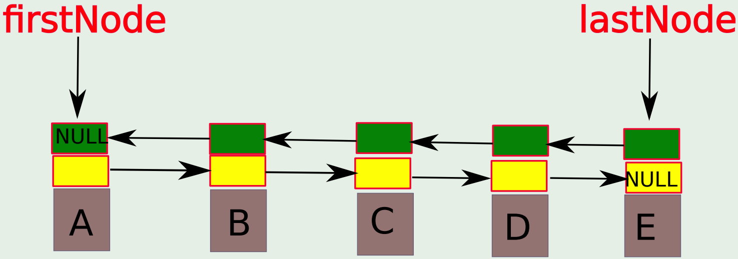
1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - **Doubly Linked List**
   - Where are all these things?

Doubly Linked List

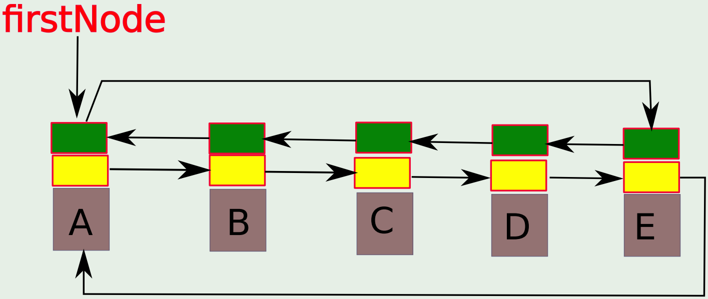
Doubly Linked List



What about the changes here

Doubly Linked Circular List

Doubly Linked Circular List



Outline

1. Iterators
   - The Method
   - Interface
   - Separate and Inner Class Iterator
     - Separate Class Iterator
     - Inner Class Iterator

2. Linked Representation
   - Introduction
   - Memory Layout
   - Operations
     - IsEmpty()
     - size()
     - Get(Index)
     - Remove
     - Add
   - Can we simplify the code?

3. Other data structures based in the Linked List
   - Circular List
   - Doubly Linked List
   - Where are all these things?

Where does Java has all these things?

Package

java.util.LinkedList

There you have

- It has the Linked implementation of a linear list.

Where does Java has all these things?

Package

java.util.LinkedList

There you have

- It has the Linked implementation of a linear list.
- It has the doubly linked circular list with header node.
- It Has all methods of LinearList plus many more.

Where does Java has all these things?

Package

java.util.LinkedList

There you have

- It has the Linked implementation of a linear list.
- It has the doubly linked circular list with header node.
- It Has all methods of LinearList plus many more.