ITI 1121. Introduction to Computing II

List: iterative list processing

by

Marcel Turcotte

Version March 20, 2020

Preamble

Preamble

Overview

Overview

List: iterative list processing

We compare the computational time required to traverse a linked list when statements have access to the nodes of the list against the implementation using the methods of the interface of the list. We explore an efficient implementation without accessing the nodes of the list directly.

General objective :

- This week you will be able to explain and use an iterator.

Foreword

- The topics covered in this module will reinforce the notions of **encapsulation** and **object-oriented programming**, including the notion of the **state** of the object, as well as the **interfaces**.
- It is also an opportunity to informally introduce the **complexity of computation** (**asymptotic analysis**) that will be presented to you in the data structure course.

Preamble

Learning objectives

Learning objectives

- **Compare** the time required to traverse a linked list, discuss the case where statements have access to nodes in the list compared to the implementation having access only to the methods of its interface.
- **Compare** nested static and non-static Java classes.
- **Modify** the implementation of an iterator in order to add a method to it.

Lectures:

- Pages 89-96, 103-112 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Motivation

Motivation

Problem

Motivation

- You must devise a method to **traverse** a **linked list**.

Motivation

Details

Details

- We're working with an **singly linked** implementation of the interface **List**.

```
public interface List<E> {
    boolean add(E element);
    E get(int index);
    boolean remove(E element);
    int size();
}
```

Details

- We're working with an **singly linked** implementation of the interface **List**.

```
public interface List<E> {
    boolean add(E element);
    E get(int index);
    boolean remove(E element);
    int size();
}
```

- The difficulties would be the same if the list was **doubly linked**.

Details

- We're working with an **singly linked** implementation of the interface **List**.

```
public interface List<E> {
    boolean add(E element);
    E get(int index);
    boolean remove(E element);
    int size();
}
```

- The difficulties would be the same if the list was **doubly linked**.
- We'll call this implementation **LinkedList**.

An example of a list

```java
List<String> colors;
colors = new LinkedList<String>();

colors.add("bleu");
colors.add("blanc");
colors.add("rouge");
colors.add("jaune");
colors.add("vert");
colors.add("orange");
```

Motivation

Internal implementation

Implementation A : inside the class

- Inside the class **LinkedList**, we have access to the implementation details. In particular, we have **access to the nodes**.

Implementation A : inside the class

- Inside the class **LinkedList**, we have access to the implementation details. In particular, we have **access to the nodes**.
  - **Give an implementation:**

Implementation A : inside the class

- Inside the class **LinkedList**, we have access to the implementation details. In particular, we have **access to the nodes**.
  - **Give an implementation:**

```
Node<E> p;
p = head;

while (p != null) {
    System.out.println(p.value);
    p = p.next;
}
```

Motivation

External implementation

Implementation B : out of the class

- Outside the class **LinkedList**, we don’t have access to the implementation details. In particular, we **don’t have access to the nodes**.

Implementation B : out of the class

- Outside the class **LinkedList**, we don't have access to the implementation details. In particular, we **don't have access to the nodes**.
  - **Give an implementation:**

Implementation B : out of the class

- Outside the class **LinkedList**, we don't have access to the implementation details. In particular, we **don't have access to the nodes**.
  - **Give an implementation:**

```
for (int i=0; i < colors.size(); i++) {
    System.out.println(colors.get(i));
}
```

Remark

- From outside the class **LinkedList**, we need to use **E get(int pos)** to access the elements of the list.

Motivation

Computation time

Discussion

- **Compare** the **runtime** of the two implementations (internal and external).

Discussion

- **Compare** the **runtime** of the two implementations (internal and external).
  - Is the implementation of the **inside** the class **faster** or **slower**?

Discussion

- **Compare** the **runtime** of the two implementations (internal and external).
  - Is the implementation of the **inside** the class **faster** or **slower**?
  - Are the **differences minor** or **major**?

Computation time

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	A	B
20,000	73,214	523,248,106
40,000	138,208	2,054,870,866
80,000	277,909	8,430,799,795
160,000	671,434	36,546,381,116
320,000	1,461,222	157,744,738,581
640,000	3,428,519	655,822,468,389
1,280,000		

Computation time

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	A	B
20,000	73,214	523,248,106
40,000	138,208	2,054,870,866
80,000	277,909	8,430,799,795
160,000	671,434	36,546,381,116
320,000	1,461,222	157,744,738,581
640,000	3,428,519	655,822,468,389
1,280,000	5,922,119	

Computation time

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	A	B
20,000	73,214	523,248,106
40,000	138,208	2,054,870,866
80,000	277,909	8,430,799,795
160,000	671,434	36,546,381,116
320,000	1,461,222	157,744,738,581
640,000	3,428,519	655,822,468,389
1,280,000	5,922,119	**45 minutes!**

Computation time

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	A	B
20,000	73,214	523,248,106
40,000	138,208	2,054,870,866
80,000	277,909	8,430,799,795
160,000	671,434	36,546,381,116
320,000	1,461,222	157,744,738,581
640,000	3,428,519	655,822,468,389
1,280,000	5,922,119	**45 minutes!**

For $1,280,000$ elements, it takes about 45 minutes to go through the list with calls to **get(pos)**, whereas it only takes **5.92 milliseconds** for the approach **A**.

Motivation

Discussion

Discussion

- **How** do you explain that difference?

Discussion

- **How** do you explain that difference?
- For each implementation, what **mathematical relationship** is there between the number of elements in the list, **n**, and the **computation time**?

Discussion

- **How** do you explain that difference?
- For each implementation, what **mathematical relationship** is there between the number of elements in the list, **n**, and the **computation time**?
  - **Complete the sentence:** every time the number of elements **n** doubles, the **computation time** . . .

- **Give** the implementation of the method **E get(int pos)**.

- **Give** the implementation of the method **E get(int pos)**.

```
Node<E> p;
p = head;
for (int j=0; j<pos; j++) {
    p = p.next;
}
return p.value;
```

- **Give** the implementation of the method **E get(int pos)**.

```
Node<E> p;
p = head;
for (int j=0; j<pos; j++) {
    p = p.next;
}
return p.value;
```

- Therefore, the implementation of **B**

```
for (int i=0; i < colors.size(); i++) {
    System.out.println(colors.get(i));
}
```

- Is **equivalent to this**:

- **Give** the implementation of the method **E get(int pos)**.

```java
    Node<E> p;
    p = head;
    for (int j=0; j<pos; j++) {
        p = p.next;
    }
    return p.value;
```

- Therefore, the implementation of **B**

```java
for (int i=0; i < colors.size(); i++) {
    System.out.println(colors.get(i));
}
```

- Is **equivalent to this**:

```java
for (int i=0; i < size; i++) {
    Node<E> p;
    p = head;
    for (int j=0; j<i; j++) {
        p = p.next;
    }
    System.out.println(p.value);
}
```

Number of nodes visited

Call	# of nodes visited
get(0)	
get(1)	
get(2)	
get(3)	
...	
get(n-1)	

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	
get(2)	
get(3)	
...	
get(n-1)	

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	2
get(2)	
get(3)	
...	
get(n-1)	

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	2
get(2)	3
get(3)	
...	
get(n-1)	

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	2
get(2)	3
get(3)	4
...	
get(n-1)	

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	2
get(2)	3
get(3)	4
...	
get(n-1)	n

Number of nodes visited

Call	# of nodes visited
get(0)	1
get(1)	2
get(2)	3
get(3)	4
...	
get(n-1)	n

- Then the total number of **nodes visited** will be,

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

Motivation

Conclusion

Discussion

- Implementation **A** visits n nodes.
- Implementation **B** visits n^2 nodes!

Concept

Concept

Objective

Concept

Objective: Devise an approach to traverse the list **one and only once**.

Concept

Objective: Devise an approach to traverse the list **one and only once**.

- The user of the list will not have access to the implementation (**p.next** and others)!

Concept

Objective: Devise an approach to traverse the list **one and only once**.

- The user of the list will not have access to the implementation (**p.next** and others)!
- The proposed solution will be applicable in a very specific context, **when all nodes of the list are visited sequentially**.

Concept

Objective: Devise an approach to traverse the list **one and only once**.

- The user of the list will not have access to the implementation (**p.next** and others)!
- The proposed solution will be applicable in a very specific context, **when all nodes of the list are visited sequentially**.
- **This is not a general solution to speed up get(i)**.

Iterator

- The iterator is a **uniform** and **general** mechanism for traversing a variety of data structures, such as lists, but also trees and others (see CSI2110);

Iterator

- The iterator is a **uniform** and **general** mechanism for traversing a variety of data structures, such as lists, but also trees and others (see CSI2110);
- Provides access to the elements **one element at a time**;

Iterator

- The iterator is a **uniform** and **general** mechanism for traversing a variety of data structures, such as lists, but also trees and others (see CSI2110);
- Provides access to the elements **one element at a time**;
- Part of the Java collections.

Concept

Interface

Interface Iterator

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Discussion : implementation

- Which class will implement **Iterator**?

Discussion : implementation

- Which class will implement **Iterator**?
- How do you **create** and **initialize** an iterator?

Discussion : implementation

- Which class will implement **Iterator**?
- How do you **create** and **initialize** an iterator?
- How do you **move** the iterator?

Discussion : implementation

- Which class will implement **Iterator**?
- How do you **create** and **initialize** an iterator?
- How do you **move** the iterator?
- How do I **detect the end** of the iteration?

Implementation 1.0

- Let's develop an initial implementation that will be quite **different** from the **final implementation**.
- It will be a good **intermediate step**, though.
- The class **LinkedList** implements the interface **Iterator**.

Implementation 1.0

Implementation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;

    // ...

    public E next() { ... }

    public boolean hasNext() { ... }

}
```

Implementation 1.0

Example

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.
- When a call is made to the method **next**:

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.
- When a call is made to the method **next**:
  1. The iterator is **moving** forward;

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.
- When a call is made to the method **next**:
  1. The iterator is **moving** forward;
  2. **Returns** the value of the element visited.

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.
- When a call is made to the method **next**:
  1. The iterator is **moving** forward;
  2. **Returns** the value of the element visited.
- A call to the method **next** when the list is empty or at the end of iteration (when **hasNext** returns **false**) is an error and an exception will be thrown.

Contract

- Conceptually, the iterator is **to the left of the first element** at the beginning of the iteration.
- When a call is made to the method **next**:
  1. The iterator is **moving** forward;
  2. **Returns** the value of the element visited.
- A call to the method **next** when the list is empty or at the end of iteration (when **hasNext** returns **false**) is an error and an exception will be thrown.

Example

```
List<Integer> l;
l = new LinkedList<Integer>();

for (int i=0; i<5; i++) {
    l.add(new Integer(i));
}

int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```java
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```java
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```java
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```java
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```java
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Example


```
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Implementation 1.0

Discussion

- What are the necessary **instance variables**?

- What are the necessary **instance variables**?
  - All it takes is one, **current.**

- What are the necessary **instance variables**?
    - All it takes is one, **current.**
- What's the **type** of the variable **current**?

- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node**<**E**>

- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node<E>**
- What will be the **initial** value of **current**?

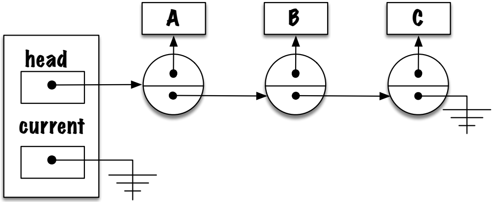
- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node<E>**
- What will be the **initial** value of **current**?
  - The initial value is **null** and conceptually, the iterator is positioned in before the first element.

- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node<E>**
- What will be the **initial** value of **current**?
  - The initial value is **null** and conceptually, the iterator is positioned in before the first element.
- On the first call,

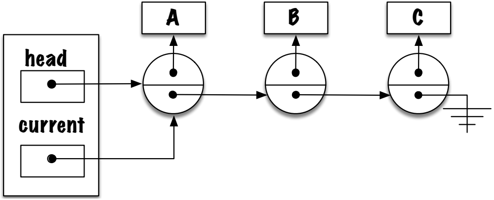
- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node<E>**
- What will be the **initial** value of **current**?
  - The initial value is **null** and conceptually, the iterator is positioned in before the first element.
- On the first call,
  - the method **next** positions the variable **current** on the first node, and returns the value found there.

- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node<E>**
- What will be the **initial** value of **current**?
  - The initial value is **null** and conceptually, the iterator is positioned in before the first element.
- On the first call,
  - the method **next** positions the variable **current** on the first node, and returns the value found there.
- For each subsequent call,

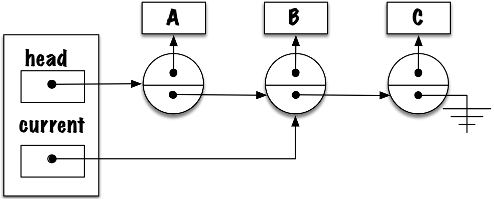
- What are the necessary **instance variables**?
  - All it takes is one, **current.**
- What's the **type** of the variable **current**?
  - **Node**<**E**>
- What will be the **initial** value of **current**?
  - The initial value is **null** and conceptually, the iterator is positioned in before the first element.
- On the first call,
  - the method **next** positions the variable **current** on the first node, and returns the value found there.
- For each subsequent call,
  - **next** moves **current** to the next element.



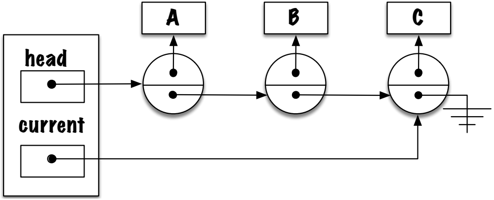
Before iteration



after the call to next()



after the call to next()



after the call to next()

Implementation 1.0

Instance variable

Implementation 1.0

```java
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    // ...

    public E next() { ... }

    public boolean hasNext() { ... }

}
```

Implementation 1.0

next

Implementation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public E next() {









    }

}
```

Implementation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public E next() {
        if (current == null) {
            current = head;
        } else {
            current = current.next;
        }
        if (current == null) {
            throw new NoSuchElementException();
        }
        return current.value;
    }

}
```

Implementation 1.0

hasNext

Implementation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public boolean hasNext() {




    }

}
```

Implementation 1.0

```java
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public boolean hasNext() {
        if (current == null && head != null) {
            return true;
        } else if (current != null && current.next != null) {
            return true;
        } else {
            return false;
        }
    }

}
```

Is it fast?

Is it fast?

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	Inside	Iterator
20,000	73,214	113,817
40,000	138,208	167,639
80,000	277,909	324,540
160,000	671,434	758,642
320,000	1,461,222	1,760,357
640,000	3,428,519	3,717,519
1,280,000	5,922,119	7,239,676

For $1,280,000$ elements, the computation time is **7.2 milliseconds**, barely **13%** slower than the implementation having access to the elements!

Implementation 1.0

Discussion

Discussion

- What's the biggest **restriction** of our implementation?

Discussion

- What's the biggest **restriction** of our implementation?
  - Only one **iteration** at a time.
- What does it take to overcome that **limitation**?

Discussion

- What's the biggest **restriction** of our implementation?
  - Only one **iteration** at a time.
- What does it take to overcome that **limitation**?

Discussion

- What's the biggest **restriction** of our implementation?
  - Only one **iteration** at a time.
- What does it take to overcome that **limitation**?
  - Several references. One reference per iterator.

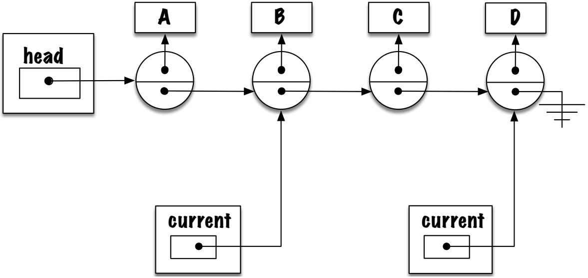
Implementation 2.0

Implementation 2.0

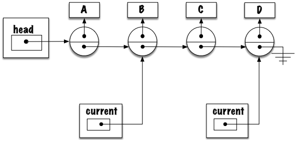
Memory diagram

Memory diagram

- **Discuss** this memory diagram.

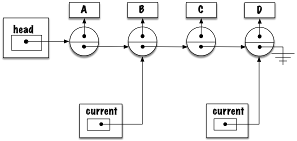


Discussion



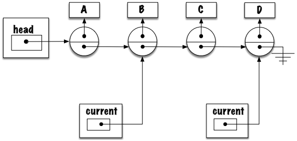
- **LinkedList** does not implement the interface **Iterator**.

Discussion



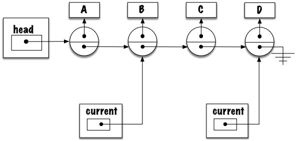
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.

Discussion



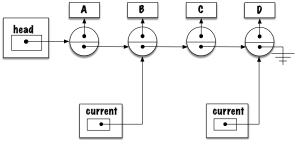
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.
- As many iterators as it takes.

Discussion



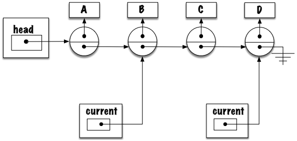
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.
- As many iterators as it takes.
- The iterator must have **access to the elements** of the list.

Discussion



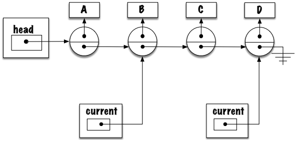
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.
- As many iterators as it takes.
- The iterator must have **access to the elements** of the list.
- A **first level** class wouldn't have access to the elements.

Discussion

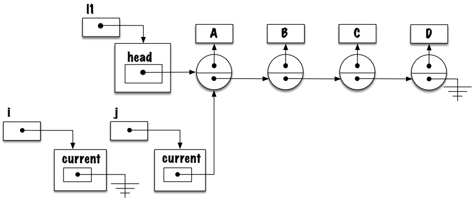


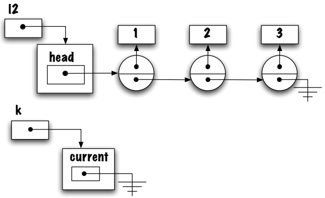
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.
- As many iterators as it takes.
- The iterator must have **access to the elements** of the list.
- A **first level** class wouldn't have access to the elements.
- **Suggestions?**

Discussion

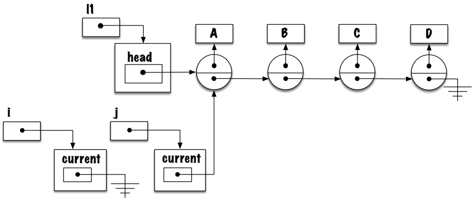


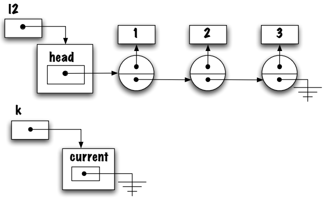
- **LinkedList** does not implement the interface **Iterator**.
- The iterator is an object that has an instance variable, **current**, of type **Node**<**E**>.
- As many iterators as it takes.
- The iterator must have **access to the elements** of the list.
- A **first level** class wouldn't have access to the elements.
- **Suggestions?**
- That's right, the iterator is a **nested class**.





- An **itrator** must belong to a given list.



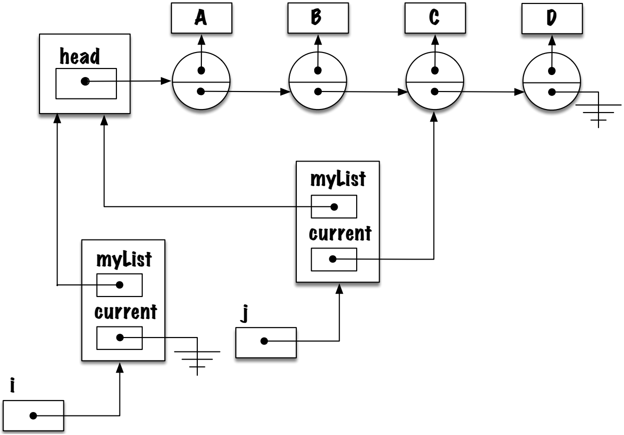


- An **itrator** must belong to a given list.
- An **itrator** must access the variable **head** from its list.

```
public E next() {
    if (current == null) {
        current = head;
    } else {
        current = current.next;
    }
    if (current == null) {
        throw new NoSuchElementException();
    }
    return current.value;
}
```

Memory diagram

- **Discuss** this memory diagram.



Implementation 2.0

Instance variables and constructor

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        private ListIterator(LinkedList<E> myList) {
            this.myList = myList;
            current = null;
        }

        public boolean hasNext() { ... }

        public E next() { ... }
    }

    private Node<E> head;

}
```

Implementation 2.0

next

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() {
            if (current == null) {
                current =                          ;
            } else {
                current = current.next;
            }
            if (current == null) {
                throw new NoSuchElementException();
            }
            return current.value;
        }

        public boolean hasNext() { ... }
    }

    private Node<E> head;
}
```

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() {
            if (current == null) {
                current = myList.head;
            } else {
                current = current.next;
            }
            if (current == null) {
                throw new NoSuchElementException();
            }
            return current.value;
        }

        public boolean hasNext() { ... }
    }

    private Node<E> head;
}
```

Implementation 2.0

hasNext

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() { ... }

        public boolean hasNext() {
            if (current == null &&                  != null) {
                return true;
            } else if (current != null && current.next != null) {
                return true;
            } else {
                return false;
            }
        }
    }

    private Node<E> head;
}
```

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() { ... }

        public boolean hasNext() {
            if (current == null && myList.head != null) {
                return true;
            } else if (current != null && current.next != null) {
                return true;
            } else {
                return false;
            }
        }
    }

    private Node<E> head;
}
```

Implementation 2.0

iterator

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        private LinkedListIterator(LinkedList<E> myList) {
            this.myList = myList;
            current = null;
        }
        public E next() { ... }
        public boolean hasNext() { ... }

    }

    public Iterator<E> iterator() {


    }

    private Node<E> head;
}
```

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        private LinkedListIterator(LinkedList<E> myList) {
            this.myList = myList;
            current = null;
        }
        public E next() { ... }
        public boolean hasNext() { ... }

    }

    public Iterator<E> iterator() {
        return new ListIterator<E>(this);
    }

    private Node<E> head;
}
```

Implementation 2.0

Example

```java
LinkedList<Integer> l;
l = new LinkedList<Integer>();

// ...

Iterator<Integer> i;
i = l.iterator();

while (i.hasNext()) {

    Integer v1 = i.next();

    Iterator<Integer> j;
    j = l.iterator();

    while (j.hasNext()) {

        Integer v2 = j.next();

        System.out.println("("+v1+","+v2+")");
    }
}
```

Implementation 2.0

Computation time

Is it fast?

Is it fast?

These are the **execution times** in **nanoseconds** for lists of increasing size.

# nodes	Inside	Iterator
20,000	73,214	113,817
40,000	138,208	167,639
80,000	277,909	324,540
160,000	671,434	758,642
320,000	1,461,222	1,760,357
640,000	3,428,519	3,717,519
1,280,000	5,922,119	7,239,676

For $1,280,000$ elements, the computation time is **7.2 milliseconds**, barely **13%** slower than the implementation having access to the elements!

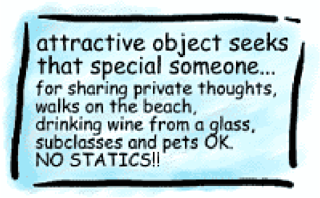
Implementation 3.0

Implementation 3.0

Inner class

«Getting in Touch with your Inner Class»

- www.javaranch.com/campfire/StoryInner.jsp

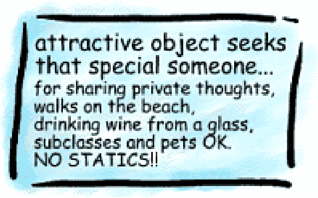


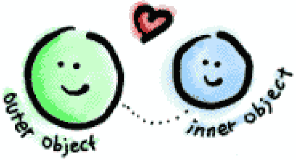
«Getting in Touch with your Inner Class»

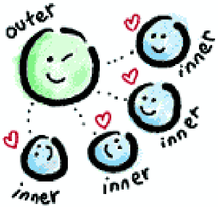
- www.javaranch.com/campfire/StoryInner.jsp

«Getting in Touch with your Inner Class»

- www.javaranch.com/campfire/StoryInner.jsp







Definition

An **inner class** is a non-static nested class.

- An object of an inner (non-static) class has **access to the variables** and **methods** of the object of the outer class from which it was created.

Implementation 3.0

next

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private class ListIterator implements Iterator<E> {

        private Node<E> current;

        public E next() {
            if (current == null) {
                current = head;
            } else {
                current = current.next;
            }
            if (current == null) {
                throw new NoSuchElementException();
            }
            return current.value;
        }

        public boolean hasNext() { ... }
    }

    private Node<E> head;
}
```

Implementation 3.0

hasNext

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private class ListIterator implements Iterator<E> {

        private Node<E> current;

        public E next() { ... }

        public boolean hasNext() {
            if (current == null && head != null) {
                return true;
            } else if (current != null && current.next != null) {
                return true;
            } else {
                return false;
            }
        }
    }

    private Node<E> head;
}
```

Implementation 3.0

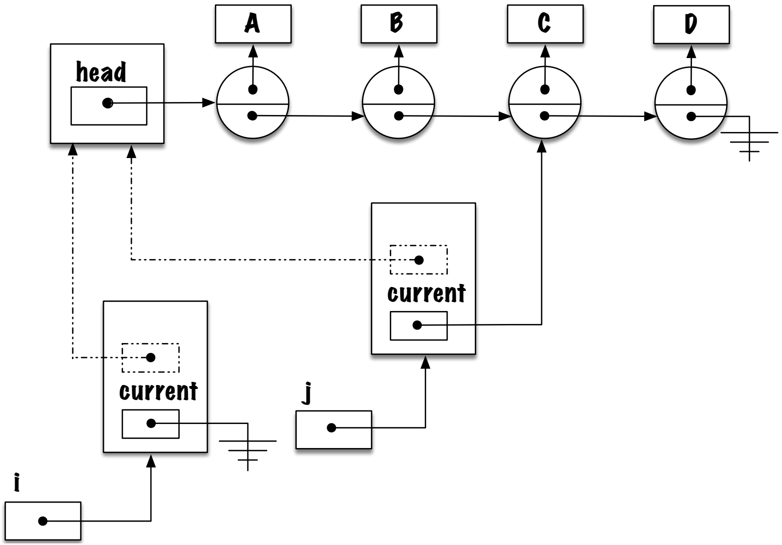
iterator

```java
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private class ListIterator implements Iterator<E> {

        private Node<E> current;

        public E next() { ... }
        public boolean hasNext() { ... }

    }

    public Iterator<E> iterator() {
        return new ListIterator<E>();
    }

    private Node<E> head;
}
```

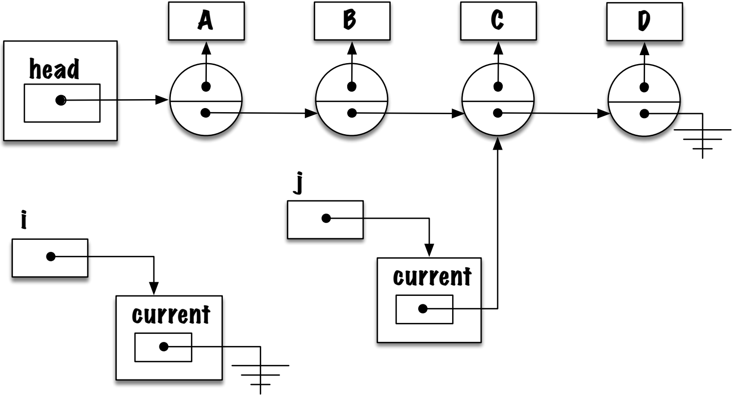
Implementation 3.0

Memory diagram

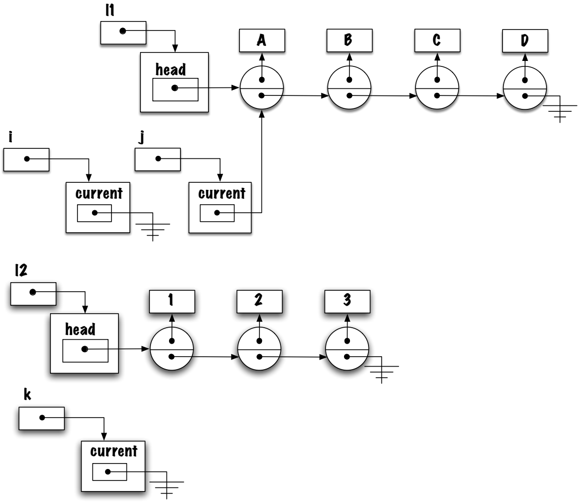
Inner class



Classe interne

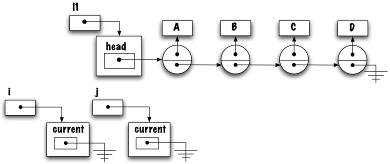


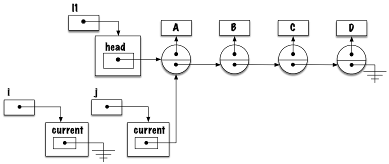
Example

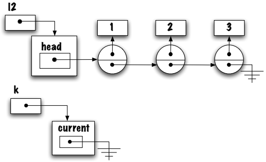


Implementation 3.0

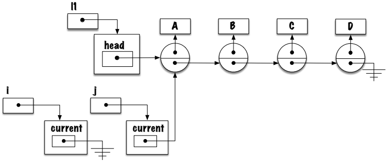
Example

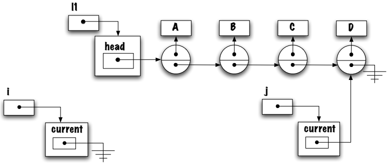


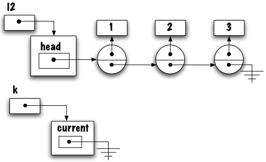




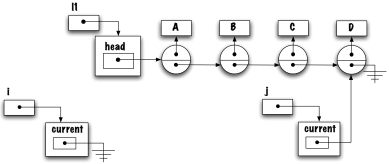

```
if (j.hasNext()) {
    String o = j.next();
}
```

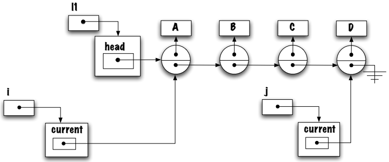


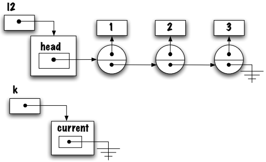




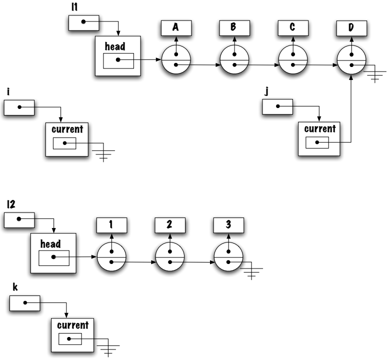

```
while (j.hasNext()) {
    String o = j.next();
}
```

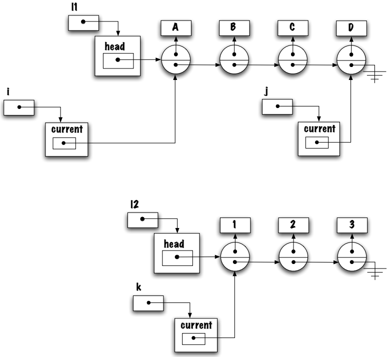




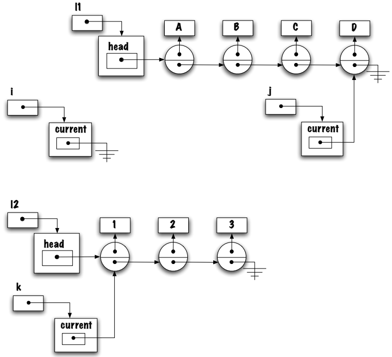


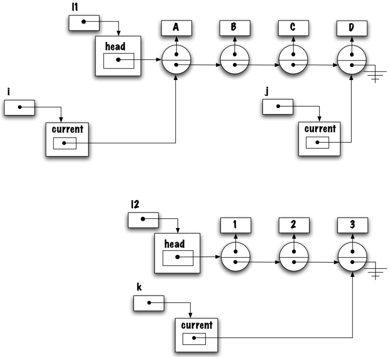

```
if (i.hasNext()) {
    String o = i.next();
}
```






```
if (k.hasNext()) {
    Integer o = k.next();
}
```






```
while (k.hasNext()) {
    Integer o = k.next();
}
```

Computation time

These are the **computation times** in **nanoseconds** for lists of increasing size.

# nodes	Inside	Iterator	Get
10,000	43,508	66,849	1.118841e+08
20,000	49,233	66,986	4.619370e+08
40,000	99,714	108,464	1.873445e+09
80,000	240,057	252,130	8.404544e+09
160,000	592,818	615,779	2.892314e+10
320,000	1,039,555	1,142,309	1.401875e+11
640,000	2,328,335	2,448,321	6.258633e+11
1,280,000	5,124,979	4,896,708	2.753671e+12
2,560,000	11,500,576	11,700,579	1.476815e+13

- For 2,560,000 elements, **get(pos)** is 1 million times slower than the iterator!
  1.48e+13 ns = 4.1 hours.

Prologue

Summary

- The iterator is a mechanism for traversing a **list one element at a time**.

Summary

- The iterator is a mechanism for traversing a **list one element at a time**.
- The method **hasNext** returns **true** if a call to the method **next** is possible.

Summary

- The iterator is a mechanism for traversing a **list one element at a time**.
- The method **hasNext** returns **true** if a call to the method **next** is possible.
- The method **next** returns the next element in the iteration.

Summary

- The iterator is a mechanism for traversing a **list one element at a time**.
- The method **hasNext** returns **true** if a call to the method **next** is possible.
- The method **next** returns the next element in the iteration.
- An **inner** class is a nested non-static class.

Summary

- The iterator is a mechanism for traversing a **list one element at a time**.
- The method **hasNext** returns **true** if a call to the method **next** is possible.
- The method **next** returns the next element in the iteration.
- An **inner** class is a nested non-static class.
- Objects of inner classes have **access** to the **variables** and **methods** of the outer class.

Next module

- **List** : recursive processing

References I

E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**
Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)
Université d'Ottawa