ITI 1121. Introduction to Computing II

List: implementation

by

Marcel Turcotte

Version March 19, 2020

Preamble

Preamble

Overview

Overview

List: implementation

We focus on three implementations of the interface **List** using linked elements: the singly-linked list, the doubly-linked list, and the doubly-linked circular list starting with a dummy node.

General objective:

- This week, you will be able to design an industrial-grade implementation of the abstract data type list.

Preamble

Learning objectives

Learning objectives

- **Explain** the role of reference variables in the implementation of a linked list.
- **Modify** the implementation of a singly or doubly linked list in order to add a new method to it.
- **Justify** the purpose of the dummy node in the implementation of a doubly linked circular list.
- **Discuss** the advantages and disadvantages, particularly in terms of execution time and memory usage, for the three implementations of a list seen in this course, the singly linked list, the doubly linked list, and the doubly linked circular list starting with a dummy node.

Readings:

- Pages 84-89, 103 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Definitions

Definition

A list (**List**) is an abstract data type (ADT) to store objects, such that each element has a predecessor and a successor (thus linear, ordered), and **having no data access restrictions**; one can inspect, insert or delete anywhere in the list. A.K.A. **Sequence**.

Implementations

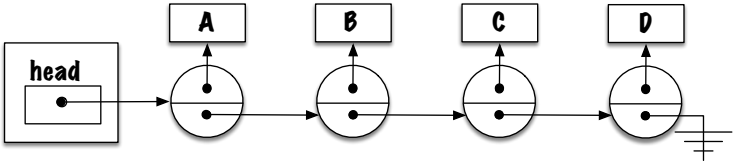
- **ArrayList**
- **LinkedList**
  - **Singly** linked list
  - **Doubly** linked list
  - List with a **dummy node**
  - **Iterative** processing (**Iterator**)
  - **Recursive** processing.

Singly linked list

- The simplest implementation is the singly linked list (**SinglyLinkedList**).
- We will use a "static" nested class to represent the nodes in the list. Each node contains a value and is connected to its next one.

```
private static class Node<T> {
    private T value;
    private Node<T> next;
    private Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

LinkedList



Definition

- We compare the efficiency of array-based (**ArrayList**) and linked-element-based (**LinkedList**) implementations.

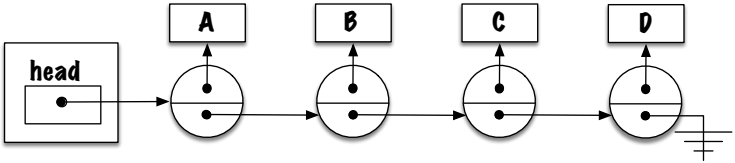
Definition

- We compare the efficiency of array-based (**ArrayList**) and linked-element-based (**LinkedList**) implementations.
  - Both can hold an unlimited number of objects, so **ArrayList** uses a dynamic array.

Definition

- We compare the efficiency of array-based (**ArrayList**) and linked-element-based (**LinkedList**) implementations.
  - Both can hold an unlimited number of objects, so **ArrayList** uses a dynamic array.
- We will say that the execution time is **variable** (slow), if the number of operations varies according to the number of elements currently saved in the data structure, and **constant** (fast) otherwise.

LinkedList



Implementations

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)		
void addLast(E elem)		
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	
void addLast(E elem)		
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)		
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)		
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()		
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()		

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	

Comparer ArrayList et LinkedList

- Can you predict which of the two implementations will be faster?

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	variable
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	variable

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;
- Looking at the table above, **when** should we use an implementation based on **arrays**?

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;
- Looking at the table above, **when** should we use an implementation based on **arrays**?
  - For **direct access** (random).

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;
- Looking at the table above, **when** should we use an implementation based on **arrays**?
  - For **direct access** (random).
- When should a linked list be used?

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;
- Looking at the table above, **when** should we use an implementation based on **arrays**?
  - For **direct access** (random).
- When should a linked list be used?
  - If all the **accesses** are at the **start** of the list;

Discussion

- For some operations, when one implementation is **fast**, the other is **slow**;
- Looking at the table above, **when** should we use an implementation based on **arrays**?
  - For **direct access** (random).
- When should a linked list be used?
  - If all the **accesses** are at the **start** of the list;
- Which implementation consumes more **memory**?

Implementations

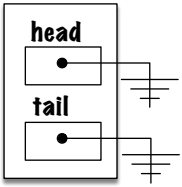
Reference to the rear node

Accelerate addLast for a singly linked list

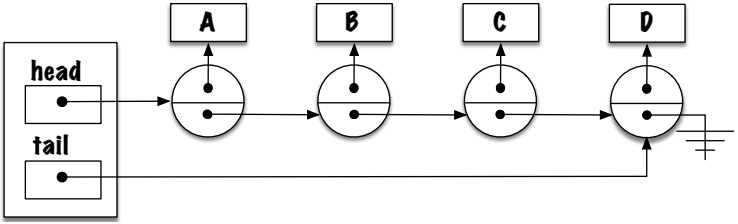
- There is a simple implementation technique for accelerating the addition of an element at the **end** of a linked structure.
- What makes the current implementation **costly**?
- Yes, you have to **traverse the list** from one end to the other in order to add the item at the very end.
- We could of course add the elements in reverse order, but that would only move the problem, the method **addFirst()** would be slow.
- For the method **size()**, we saw that the use of an additional instance variable, **count**, could save us from going through the list unnecessarily.
- What would we need in this case to **avoid traversing** the list?
- Yes, a new **variable** pointing to the **last** item on the list.

Memory diagram

- Representing the **empty list**:



- **General case**:



LinkedList

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

    private static class Node<T> {

        private T value;
        private Node<T> next;

        private Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node<E> head;
    private Node<E> tail;

    // ...
}
```

addLast

```
public void addLast(E elem) {

    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
        tail = head;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
}
```

Modify all the other methods accordingly

```
public E removeFirst() {

    E saved;
    saved = head.value;

    head = head.next;

    if (head == null) {
        tail = null;
    }

    return saved;
}
```

Compare the ArrayList and LinkedList

- Adding a **reference to the last node**.

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	**constant**
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	variable

Compare the ArrayList and LinkedList

- Adding a **reference to the last node**.

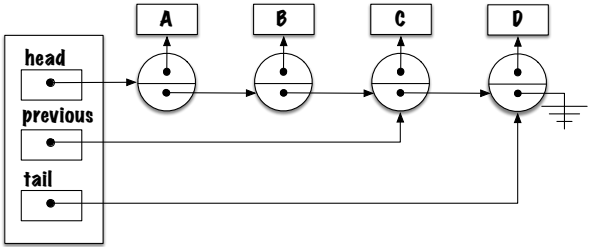
	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	**constant**
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	variable

- Is **removeLast faster now**, as well?

Implementations

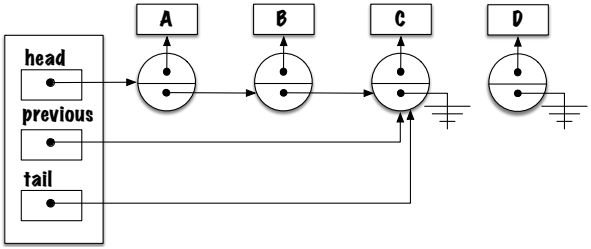
Doubly linked nodes

Accelerate removeLast



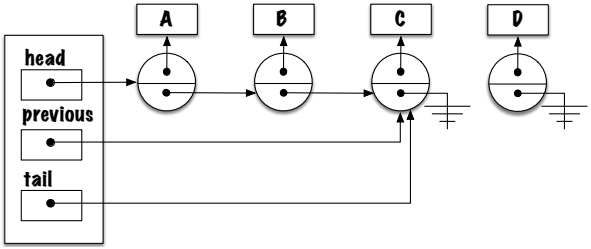
- What do you think?

Accelerate removeLast



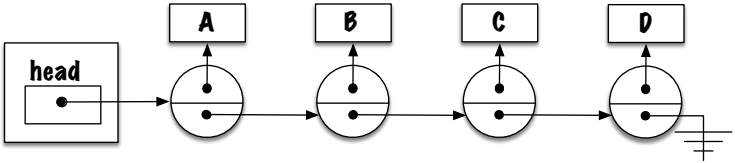
- Moving the rear reference is now easy and **fast**!

Accelerate removeLast

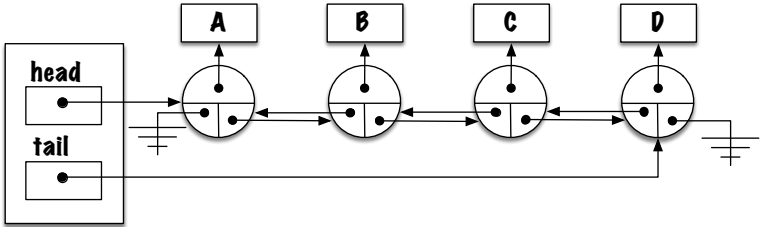


- Moving the rear reference is now easy and **fast**!
- Except moving the reference **previous** is difficult and expensive.

LinkedList



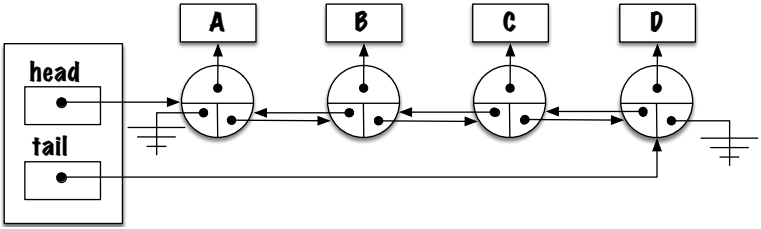
LinkedList



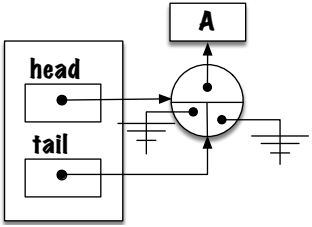
Doubly linked list

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

  private static class Node<T> {
    private T value;
    private Node<T> prev;
    private Node<T> next;
    private Node(T value, Node<T> prev, Node<T> next) {
      this.value = value;
      this.prev = prev;
      this.next = next;
    }
  }

  private Node<E> head;
  private Node<E> tail;

  // ...
}
```

removeLast: general case



removeLast: special case



```
public E removeLast() {

    E saved;
    saved = tail.value;

    if (head.next == null) {
        head = null;
        tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }

    return saved;
}
```

Compare ArrayList and LinkedList

- **Doubly** linked nodes.

	ArrayList	**LinkedList**
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	constant
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	**constant**

Discussion

- What will be the **impact** of this change?

Preconditions : add(int pos, E elem)

- What are the **prerequisites** to the method **add**?

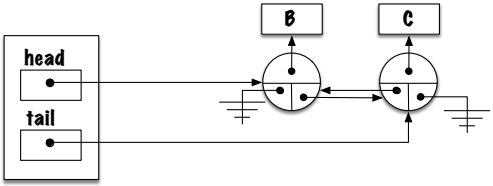
Preconditions : add(int pos, E elem)

- What are the **prerequisites** to the method **add**?

```
if (elem == null ) {
    throw new NullPointerException( "null" );
}
if (pos < 0 || pos > size) {
    throw new IndexOutOfBoundsException(pos);
}
```

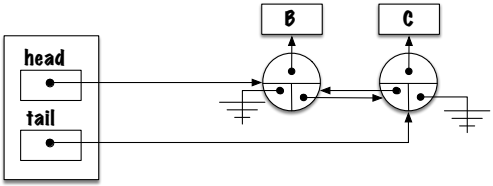
Special cases : add(int pos, E elem)

- What are the **special cases**?



Special cases : add(int pos, E elem)

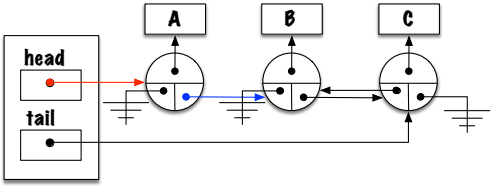
- What are the **special cases**?



- Adding at **position 0**.

Special case : add(int pos, E elem)

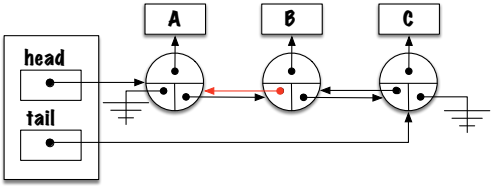
Special case: head = new Node<E>(elem, null, head)



- **What's missing?**

Special case : add(int pos, E elem)

Special case: head.next.previous = head



Special case : add(int pos, E elem)

Special case:

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    head.next.previous = head;
}
```

Special case : add(int pos, E elem)

Special case:

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    head.next.previous = head;
}
```

- Have we thought about **every possible case**?

Special case : add(int pos, E elem)

Special case:

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    head.next.previous = head;
}
```

- Have we thought about **every possible case**?
  - What if the list is **empty**?

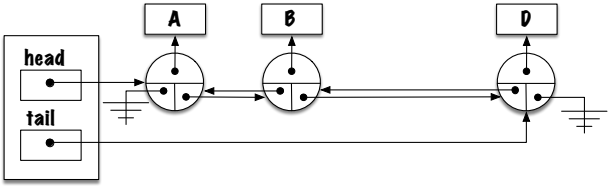
Special case : add(int pos, E elem)

Special case:

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    if (tail == null) {
        tail = head;
    } else {
        head.next.previous = head;
    }
}
```

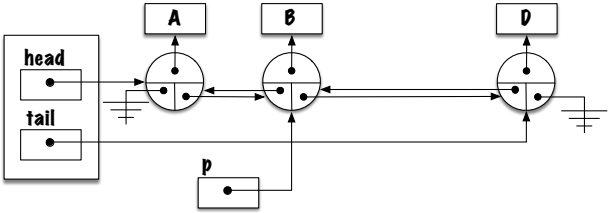
General case: add(int pos, E elem)

General case: addint at position 2.



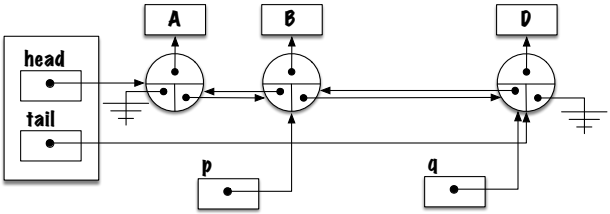
General case : add(int pos, E elem)

General case: traverse the list until **pos-1**



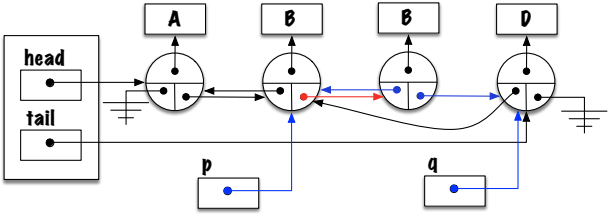
General case : add(int pos, E elem)

General case: **q = p.next**



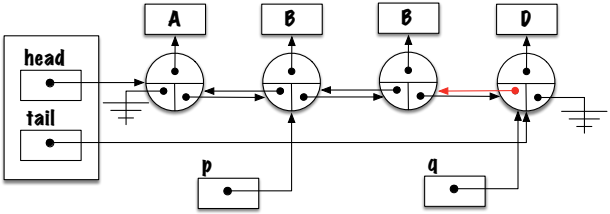
General case : add(int pos, E elem)

General case: p.next = new Node<E>(elem, p, q)



General case add(int pos, E elem)

General case: **q.previous = p.next**



General case add(int pos, E elem)

General case:

```
Node<E> before, after;
before = head;

for (int i = 0; i < (pos-1); i++) {
    before = before.next;
}

after = before.next;

before.next = new Node<E>(elem, before, after);
after.previous = before.next;
```

General case add(int pos, E elem)

General case:

```
Node<E> before, after;
before = head;

for (int i = 0; i < (pos-1); i++) {
    before = before.next;
}

after = before.next;

before.next = new Node<E>(elem, before, after);
after.previous = before.next;
```

- Have we thought of all the cases?

General case add(int pos, E elem)

General case:

```
Node<E> before, after;
before = head;

for (int i = 0; i < (pos-1); i++) {
    before = before.next;
}

after = before.next;

before.next = new Node<E>(elem, before, after);
after.previous = before.next;
```

- Have we thought of all the cases?
  - What if **before** refers to the last element?

add(int pos, E elem)

```
Node<E> before, after;

before = head;

for (int i = 0; i < (pos-1); i++) {
      before = before.next;
}

after = before.next;
before.next = new Node<E>(elem, before, after);

if (before == tail) {
  tail = before.next;
} else {
  after.previous = before.next;
}
```

```
if (elem == null ) {
    throw new NullPointerException( "null" );
}
if (pos < 0 || pos > size) {
    throw new IndexOutOfBoundsException(pos);
}
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    if (tail == null) {
        tail = head;
    } else {
        head.next.previous = head;
    }
} else {
    Node<E> before, after;
    before = head;
    for (int i = 0; i < (pos-1); i++) {
          before = before.next;
    }
    after = before.next;
    before.next = new Node<E>(elem, before, after);
    if (before == tail) {
      tail = before.next;
    } else {
      after.previous = before.next;
    }
    size++;
}
```

Implementations

Dummy node

Dummy node

- The following implementation technique allows you to **eliminate multiple special cases**.

Dummy node

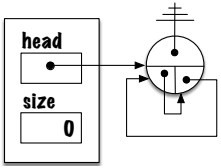
- The following implementation technique allows you to **eliminate multiple special cases**.
  - The technique uses a **dummy node** containing no element (data).

Dummy node

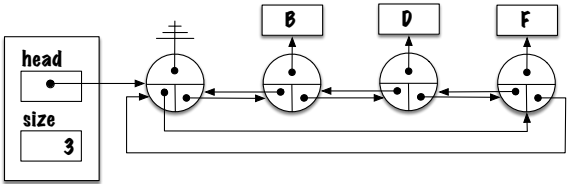
- The following implementation technique allows you to **eliminate multiple special cases**.
  - The technique uses a **dummy node** containing no element (data).
  - Plus, the list is **circular**!

Dummy node

Empty list:



General case:



```
public class LinkedList<E> implements List<E> {
   private static class Node<T> {
      private T value;
      private Node<T> prev;
      private Node<T> next;
      private Node(T value, Node<T> prev, Node<T> next) {
         this.value = value;
         this.prev = prev;
         this.next = next;
      }
   }
   private Node<E> head;
```

- Give the **implementation of the constructor**.

```
public class LinkedList<E> implements List<E> {
   private static class Node<T> {
      private T value;
      private Node<T> prev;
      private Node<T> next;
      private Node(T value, Node<T> prev, Node<T> next) {
         this.value = value;
         this.prev = prev;
         this.next = next;
      }
   }
   private Node<E> head;
```

- Give the **implementation of the constructor**.

```
    public LinkedList() {
        head = new Node<E>(null, null, null);
        head.prev = head;
        head.next = head;
    }
```

Discussion

- **What complicates** the implementation of linked-list methods without a dummy node?

Discussion

- **What complicates** the implementation of linked-list methods without a dummy node?
  - The methods usually have a **special case** for modifying in **first position**.

Discussion

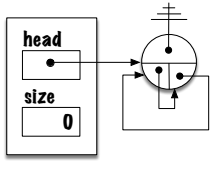
- **What complicates** the implementation of linked-list methods without a dummy node?
  - The methods usually have a **special case** for modifying in **first position**.
  - In general, one must change the variable **next** of the preceding node, unless one is processing the first node, in which case one must change the variable **head**.

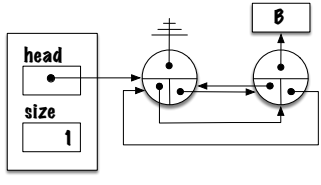
Discussion

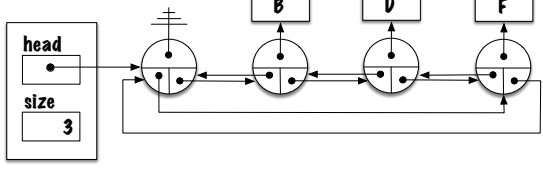
- **What complicates** the implementation of linked-list methods without a dummy node?
  - The methods usually have a **special case** for modifying in **first position**.
  - In general, one must change the variable **next** of the preceding node, unless one is processing the first node, in which case one must change the variable **head**.
  - Changes at the end of the list are also a problem since the value of **tail** must be changed.

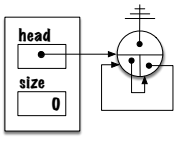
Discussion

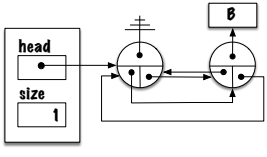
- **What complicates** the implementation of linked-list methods without a dummy node?
  - The methods usually have a **special case** for modifying in **first position**.
  - In general, one must change the variable **next** of the preceding node, unless one is processing the first node, in which case one must change the variable **head**.
  - Changes at the end of the list are also a problem since the value of **tail** must be changed.
- For the implementation having a dummy node, the treatments are uniform, we always change the variable **next** of the preceding node.

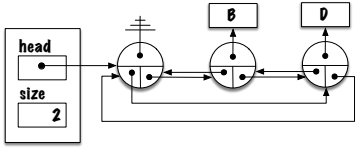

head
size
0

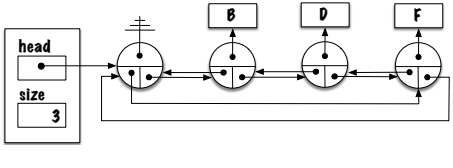

B
head
size
1


B
D
F
head
size
3


head
size
0


B
head
size
1


B
D
head
size
2


B
D
F
head
size
3

Prologue

Summary

- A reference to the **last node** makes it easy to add an element to the **end** of the list.

Summary

- A reference to the **last node** makes it easy to add an element to the **end** of the list.
- The **double linked nodes** make it easy to remove the **last** element, but also to navigate the list in reverse order.

Summary

- A reference to the **last node** makes it easy to add an element to the **end** of the list.
- The **double linked nodes** make it easy to remove the **last** element, but also to navigate the list in reverse order.
- **Circular** lists with dummy nodes have no special cases!

Next module

- **List** : iterator

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

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa