# ITI 1121. Introduction to Computing II

**List:** recursive list processing

by

Marcel Turcotte

Version March 22, 2020

# Preamble

# Preamble

## Overview

# Overview

**List: recursive list processing**

We revisit the concept of recursivity, this time in the context of processing linked lists. We develop a general strategy, "head & tail", that can be applied to all the problems covered in this course.

**General objective:**

- This week, you will be able to design recursive methods for processing linked lists.

**«*To iterate is human, to recurse divine*»**

L. Peter Deutsch

# Preamble

## Learning objectives

# Learning objectives

- **Recognize** the problems for which recursion is appropriate.
- **Discuss** the efficiency of recursive list processing in Java, especially in relation to memory consumption.
- **Explain** the role of parameters to control the flow of recursive program execution.
- **Paraphrase** the "head & tail" for recursive processing of lists.
- **Use** the "head & tail" strategy to design a recursive method for processing a linked list.

**Readings:**

- Pages 233-238 of E. Koffman and P. Wolfgang.

# Preamble

## Plan

# Plan

# Theory

# Discussion

- What **problems** have you solved using recursivity?
- What do these problems have **in common**?

# Theory

- The **solution** to a given problem can be constructed from the **solutions** of the **subproblems**;
- These sub-problems are of the **same nature** and therefore can be **solved in the same way**;
- The sub-problems are getting **smaller and smaller** (**convergence**);
- Finally, there is a size of problems that can be solved in a **trivial** way, without recursive calls, which are the **base cases**.

# Remarks

- **Recursivity** and **iteration** are of the same nature.
- It is important that the **size of the problems to be handled decreases**, otherwise there would be **an infinite number of recursive calls** (equivalent to the infinite loop); in practice the program will terminate when all the memory reserved for method calls is exhausted.
- So there has to be **a size of problem that can be solved without recursive calls**, in order to stop recursivity.
- **These are the base cases**. There has to be at least one, but there can be more than one.
- The **base cases should be processed first** in order to stop the recursion, if necessary.

# Remarks (continued)

- The programming languages **Lisp**, **Prolog** and **Haskell**, to name but a few, have no iterative control statements, iteration is replaced by recursivity.
  - Compilers automatically transform some forms of recursivity into iteration.
- The **XSLT Transformations** technology used in particular for certain Web applications is based on the concept of recursivity.
- Some treatments of **binary search trees** will be very simple to express using recursivity, but very complex otherwise.

# Theory

## Pattern

# Pattern

```
type method(parameters) {

    type result;

    if (test) { // base case

        result = calculating the result // no recursive call

    } else { // general case

        // pre-processing: partitioning the data

        result = method(sub-problem); // recursive call

        // post-processing: combine the result
    }

    return result;
}
```

# Theory

## Factorial

# Factorielle

```java
public static int factorial(int n) {
    int s, result;
    if (n<=1) { // base case
        result = 1;
    } else { // general case
        int n1 = n-1;
        s = factorial(n1);
        result = n * s;
    }
    return result;
}
```

- The above method corresponds to the proposed model:
  - First we check the **base case**, its result is calculated without recursive call (recursivity stops here!);
  - The **general case** creates smaller and smaller subproblems.

# Factorial — a terse implementation

```
public static int factorial(int n) {

    if (n<=1) {
        return 1;
    }

    return n * factorial(n-1);
}
```

- The statement **return** returns control to the caller, it stops the execution of the method, **no other statement of this call will be executed**.

# Factorial — a terse implementation

```
public static int factorial(int n) {

    if (n<0) {
        throw new IllegalArgumentException(Integer.toString(n));
    }

    if (n<=1) {
        return 1;
    }

    return n * factorial(n-1);
}
```

# Theory : "*head & tail*"

- Let's establish a general strategy for recursive list processing.
- **Breaking the list into two parts**, the first element (**head**) and the rest of the list (**tail**)*

*Here, **head** and **tail** are not the instance variables.

# Implementation

# Implementation of the class LinkedList

- We'll use a **singly** linked list.

```
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;

    // ...
}
```

# Implementation

## size

# Calculating the size of a list

Let's first consider calculating the **size** of a list.

- Let **current**, a variable of type **Node**, designate an element of the list.
- Knowing that the size of the list starting with the element designated by **current.next** is **n**,
  - what is the size of the list starting with the element designated **current**?
    - The size of the list starting with the element designated by **current** is **n+1**.

# Calculating the size of a list

- The strategy "*head & tail*" suggests that we start by posing the **recursive call**, passing **current.next**.

```
int n = size(current.next);
```

- What's the value of **n**? What does **n** mean?
  - This is the length of the list starting with the element designated by **current.next**.
- What is the size of the list starting with the element designated **current**?
  - The length of the list starting with the element designated by **current** is **n+1**.

# Calculating the size of a list

- What's the **shortest valid list** and **how long** is it?
  - It's the **empty list** and its length is **0**.
- What is the value of **current** if the list is empty?
  - The value of **current** is **null**.

# Calculating the size of a list

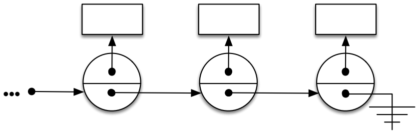
- This suggests the following partial implementation:

```
int n;
if (current == null) {
    n = 0;
} else {
    n = 1 + size(current.next);
}
```

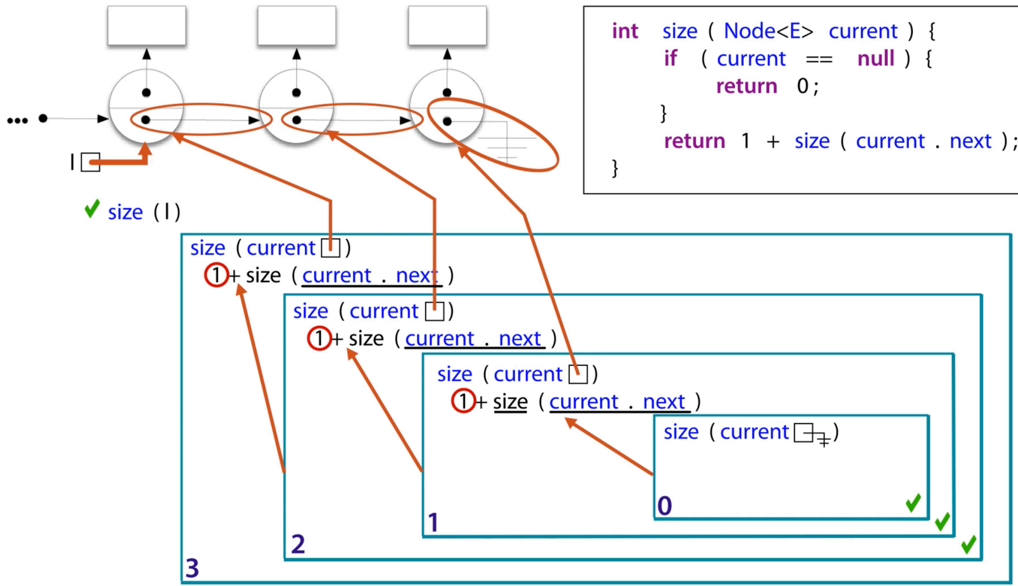
- What is the **type** of the parameter for the method **size**?
  - The **type** of the parameter is **Node**.

# Calculating the size of a list

```
int size(Node<E> current) {
    int n;
    if (current == null) {
        n = 0;
    } else {
        n = 1 + size(current.next);
    }
    return n;
}
```



```
int size(Node<E> current) {
    if (current == null) {
        return 0;
    }
    return 1 + size(current.next);
}
```




```
int  size ( Node<E>  current ) {
     if  ( current  ==  null ) {
          return  0;
     }
     return  1 + size ( current . next );
}
```

# Remarks

- Notice that the method **size** uses **no instance variables**!
  - One **controls recursivity using the parameter**.
  - Each **call** has its own **working memory** (activation block) and therefore its own copies of the local variables and parameters.

```
int size(Node<E> current) {

    if (current == null) {
        return 0;
    }

    return 1 + size(current.next);
}
```

- The **base case** is checked out first.

# Calling the method size

- **How** do we use this method to calculate the size of the list starting with the node designated by **head**?

```
int size(Node<E> current) {
    if (current == null) {
        return 0;
    }
    return 1 + size(current.next);
}
```

```
int size() {
    return size(head);
}
```

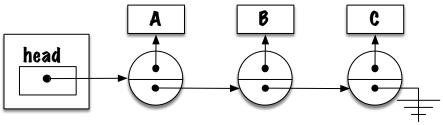
# The recursive method is a helper

- The first call is initiated by a method of visibility **public**. The value of **head** is passed as a parameter.

```
public int size() {
    return size(head);
}
```

- The **recursive** method must be of visibility **private** since its parameter is of type **Node**.

```
private int size(Node<E> current) {
    if (current == null) {
        return 0;
    }
    return 1 + size(current.next);
}
```




```java
public int size() {
    return size(head);
}

private int size(Node<E> current) {
    if (current == null) {
        return 0;
    }
    return 1 + size(current.next);
}
```

# In practice

- Each call has its own **activation block** (working memory) on the call stack, so the size of the system stack will be proportional to the size of the list.

# Implementation

## Summary

# Summary

```
type method(Node<E> current) {
    type result;
    if (current ...) {               // base case
        calculating the result       // no recursive call
    } else {                         // general case
                                     // pre-processing
        s = method(current.next);    // recursion
                                     // post-processing
    }
    return result;
}
```

# “*head & tail*”

**Steps:**

- What does **method(current.next)** mean?
  - The solution to a problem, **smaller by an element.**
- How are we going to **use this result** to construct a solution for a list beginning with the element designated by **current**?
- What are the **base cases**?
  - What’s the **shortest valid list**?
  - What’s the **result**?

# Implementation

## findMax

# LinkedList

- Let's now use a list whose elements have a method **compareTo**.

```
public class LinkedList<E extends Comparable<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;

    // ...
}
```

# findMax

- **Let's apply** the strategy as suggested.

```
result = findMax(current.next);
```

- What's the value of **result**? What does **result** mean?
  - The **largest value** for the list beginning with the element designated **current.next**
- What do we do if **result** is greater than **current.value**?

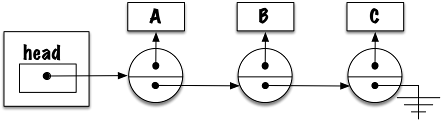
```
if (result.compareTo(current.value) > 0) {
    return result;
} else {
    return current.value;
}
```

# findMax

- This process builds smaller and smaller problems. What's the **shortest valid list**?
  - **No, not the empty list**, but the list containing **only one element**.
- What's the **returned value** going to be?

```
if (current.next == null) {
    return current.value;
}
```

```java
public E findMax() {
    if (head == null) {
        throw new NoSuchElementException();
    }
    return findMax(head);
}

private E findMax(Node<E> current) {

    if (current.next == null) {
        return current.value;
    }

    E result = findMax(current.next);

    if (result.compareTo(current.value) > 0) {
        return result;
    } else {
        return current.value;
    }
}
```




```
private E findMax(Node<E> p) {
    if (p.next == null) {
        return p.value;
    }
    E r = findMax(p.next);
    if (r.compareTo(p.value) > 0) {
        return r;‘
    } else {
        return p.value;
    }
}
```

Each of the following **examples** introduces a **new problematic**.

# Implementation

**E get(int index)**

# E get(int index)

- The method **E get(int index)** returns the element at the specified value (**index**) of the list.
- What was the **strategy** adopted for the **non-recursive** method?
  - It was necessary to count the number of visited nodes and to stop the execution of the loop **while** after having visited **index** nodes.
- For a **recursive method**, how do we **determine the number of nodes visited**?
  - We could add a **parameter** to count the number of nodes visited. Initially **0**, then **1**, **2**, **etc**.

- **Study** the following partial implementation:

```
public E get(int index) {
    return get(head, index);
}

private E get(Node<E> current, int index) {
    ...
}
```

- If **index** represents the position of the element in relation to the list starting at position **current**, what is the position of the element in relation to the list starting at position **current.next**?
  - That's right, **index-1**.
- What will the method do if the value of the **index is 0**?
  - It must return **current.value**.
  - **No** recursive call is made.
  - That's the **base case**.

# E get(int index)

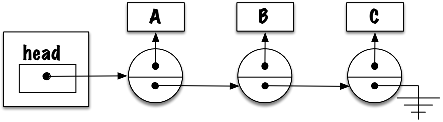
```java
private E get(Node<E> current, int index) {

    if (index == 0) {
        return current.value;
    }

    return get(current.next, index-1);

}
```

- What would happen if the initial value of the **index** was greater than the total number of items on the list?
  - **index** $>$ **0** and **current == null**

# E get(int index)

```java
private E get(Node<E> current, int index) {

    if (current == null) {
        throw new IndexOutOfBoundsException();
    }

    if (index == 0) {
        return current.value;
    }

    return get(current.next, index-1);

}
```

```java
public E get(int index) {
    if (index < 0) {
        throw new IndexOutOfBoundsException();
    }
    return get(head, index);
}

private E get(Node<E> current, int index) {

    if (current == null) {
        throw new IndexOutOfBoundsException();
    }

    if (index == 0) {
        return current.value;
    }

    return get(current.next, index-1);
}
```




```java
private E get(Node<E> p, int index) {
    if (index == 0) {
        return p.value;
    }
    return get(p.next, index -1);
}
```

# Implementation

## int indexOf(E element)

# int indexOf(E element)

- The method **indexOf** returns the position of the leftmost occurrence of the **element** in this list, and $-1$ if the value is not found there.
- The **numbering** of the elements **starts at zero**.

# int indexOf(E element)

- According to the "***head & tail***" strategy, the general case will involve a recursive call such as this:

```
s = indexOf(current.next, element);
```

- What does the value of **s** represent?
  - It's the position of the element in the list designated by **current.next**.
- Compared to the current list, the one designated by **current**, what is the position of **element**?
  - $s + 1$

# int indexOf(E element)

- If the value of **s** is **greater or equal to zero**, **s** is the **position of the element** in the **rest of list**.
- What does **s == -1** mean?
  - The element was **absent** from the rest of the list.
- Which case **hasn't** been dealt with?
  - **current.value.equals(element)**
  - What **value** should we return then?
    - **0**

# int indexOf(E element)

```
s = indexOf(current.next, element);

if (current.value.equals(element) ) {
    result = 0;
} else if (s == -1) {
    result = s;
} else {
    result = 1 + s;
}
```

# int indexOf(E element)

- What's the **base case**?
  - The shortest list is the **empty list**, it doesn't contain the element you're looking for, just return the special value **-1**.

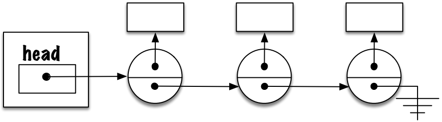
```
if (current == null) {
    return -1;
}
```

# int indexOf(E element)

```java
private int indexOf(Node<E> current, E element) {

    if (current == null) {
        return -1;
    }

    int result = indexOf(current.next, element);

    if (current.value.equals(element)) {
        return 0;
    }

    if (result == -1) {
        return result;
    }

    return result + 1;
}
```

# int indexOf(E element)

- Is it working?
  - **Yes**.
- There's still a **problem** with that implementation.
  - **What** is it?




```
int indexOf(Node<E> p, E e) {
  if (p == null) return -1;
  int r = indexOf(p.next, e);
  if (p.value.equals(e)) return 0;
  if (r == -1) return r;
  return r + 1;
}
```

# int indexOf(E element)

- How do we **stop** recursive calls as soon as the value we're looking for is found?

```
private int indexOf(Node<E> current, E element) {
    if (current == null) {
        return -1;
    }
    int result = indexOf(current.next, element);
    if (current.value.equals(element)) {
        return 0;
    }
    if (result == -1) {
        return result;
    }
    return result + 1;
}
```

# int indexOf(E element)

```
private int indexOf(Node<E> current, E element) {

    if (current == null) {
        return -1;
    }

    if (current.value.equals(element)) {
        return 0;
    }

    int result = indexOf(current.next, element);

    if (result == -1) {
        return result;
    }

    return result + 1;
}
```

# Implementation

**E indexOfLast(E element)**

# E indexOfLast(E element)

- The method **indexOfLast** returns the position of the last (rightmost) occurrence of the **element**, and -1 otherwise.
- What are the changes to be made?
- Can **current.value.equals(element)** be part of the base case?
  - **No**, recursion must go through the entire list.
- How to process the result **indexOfLast(current.next, element)**?

```java
public int indexOfLast(E element) {
    return indexOfLast(head, element);
}

private int indexOfLast(Node<E> current, E element) {

    if (current == null) {
        return -1;
    }

    int result = indexOfLast(current.next, element);

    if (result > -1) {
        return result + 1;
    } else if (element.equals(current.value)) {
        return 0;
    }

    return -1;
}
```

# Implementation

## boolean contains(E element)

# Exercise

- **boolean contains(E element)**

# Implementation

**boolean isIncreasing()**

# boolean isIncreasing()

- The methods **size**, **indexOf** and **contains** only deal with one element at a time.
- Let's consider a recursive implementation of the method **isIncreasing**.
- **Examine** each consecutive pair and return the value **false** as soon as a pair is not increasing.
- If the method **attains the end of the list** then **the list is ascending**!

# boolean isIncreasing()

```java
public boolean isIncreasing() {
    return isIncreasing(head);
}
```

# boolean isIncreasing()

- What's the **base case**?
  - What's the **shortest valid list**?
    - The **empty** list and the **singleton** are growing.

```
if ((current == null) || (current.next == null)) {
    return true;
}
```

# boolean isIncreasing()

**General case.**

- Which approach is **preferable**?
  1. Do a **recursive call**, then **process the result**.
  2. Process the **current position**, then do a recursive **recursive call**.

```
if (current.value.compareTo(current.next.value) > 0) {
    return false;
} else {
    return isIncreasing(current.next);
}
```

# boolean isIncreasing()

```java
private boolean isIncreasing(Node<E> current) {

    if ((current == null) || (current.next == null)) {
        return true;
    }

    if (current.value.compareTo(current.next.value) > 0 ) {
        return false;
    }

    return isIncreasing(current.next);
}
```

# Implementation

## Exercises

# Exercices

- **void addLast(E element)**
- **boolean equals(LinkedList<E> other)**

# Implementation

**void remove(E element)**

# void remove(E element)

- We're now considering methods that **transform the structure** of the list.
- For the methods **indexOf** and **contains**, the main consequence of additional recursive calls is the **inefficiency** of the method.
- On the other hand, recursive methods that transform lists can lead to more serious problems.
- Consider the example of the method **remove**, which removes the first occurrence of an object in the list.

# void remove(E element)

- Give the **the high-level strategy**.
  - **Traverse** the list.
  - **Find** the element.
  - **Remove** the element.

# public void remove(E element)

- What will be the **difficulties**?
  - We'll remember that when traversing a singly linked list using a **while** loop, we had to stop one position before the element to be removed, since it's the variable **next** of the preceding element that has to be modified.
  - To remove the first element, we must modify the variable **head** of the header, and not the variable **next** of the preceding node.

# public void remove(E element)

- What are the **preconditions**?
  - **element != null**
  - The list is not empty.
- What are the **special cases**?
  - The element being sought is in **first position**.

# public void remove(E element)

```
public void remove(E element) {

  if (element == null) {
      throw new NullPointerException("Illegal argument");
  }

  if (head == null) {
      throw new NoSuchElementException();
  }

  if (head.value.equals(element) ) {
      head = head.next;
  } else {
      remove(head,element);
  }

}
```

# Remarks

- For the first call to the method **remove(Node<E> current, E element)**, we know that **current.value.equals(element)** is false. **Why?**
  - It's the first call, **current == head**.
  - If **head.value.equals(element)** were true at the time of the call to the **public** method, then there would have been no call to the **private** method.
- The recursive method will preserve this property, it checks if the sought element, **element**, is at the position that follows, **current.next**, and if yes, removes this node and finishes, otherwise it continues its search.

# remove(Node<E> current, E element)

**General case:** Which **scenario** seems the most **appropriate**:

1. Do a **recursive call**, followed by a **post-processing**?
2. Do a **pre-processing**, followed by a **recursive call** (if necessary)?

Since we have to remove the **leftmost element**, should we do a **pre-processing** followed by a recursive call (if necessary)? (Strategy 2)

# remove(Node<E> current, E element)

- What's the necessary **pre-processing**?
  - If **current.next.value.equals(element)**, remove the next element.
  - Otherwise, process the rest of the list (**recursive call**).

# remove(Node<E> current, E element)

- What's the **base case**?
  - **Singleton.**
  - What do we do now?
    - Throw the exception **NoSuchElementException**.

# remove(Node<E> current, E element)

```
private void remove(Node<E> current, E element) {

    if (current.next == null ) {
        throw new NoSuchElementException();
    }

    if (current.next.value.equals(element)) {
        current.next = current.next.next; // base case
    } else {
        remove(current.next, element);     // general case
    }
}
```

```java
public void remove(E element) {
  if (element == null) {
      throw new NullPointerException("Illegal argument");
  }
  if (head == null) {
      throw new NoSuchElementException();
  }
  if (head.value.equals(element) ) {
      head = head.next; // special case
  } else {
      remove(head,element);
  }
}

private void remove(Node<E> current, E element) {
    if (current.next == null ) {
        throw new NoSuchElementException();
    }
    if (current.next.value.equals(element)) {
        current.next = current.next.next; // base case
    } else {
        remove(current.next, element);    // general case
    }
}
```

# Exercises

- void removeLast()
- void removeLast(E element)
- void removeAll(E element)
- void remove(int pos)

# Implementation

**LinkedList<E> subList(int fromIndex, int toIndex)**

# LinkedList<E> subList(int fromIndex, int toIndex)

- The method will return a **new list** containing the elements located between the positions **fromIndex** and **toIndex** of the original list, without changing it.

# Discussion

**Propose** a strategy to build the resulting list.

1. **Post-processing**
   - **Traverse** the list to the highest index;
   - **Return** a list containing only the value at that position;
   - **Add** the current element to the **start** of the list, if its position is within the range.
2. **Pre-processing**
   - **An empty list** is passed as a **parameter** to the first call;
   - **Add** the current element to the **end** of the list, if the current position is part of the interval;
   - **Recusive** call.

# Strategy 1

- Recursive calls **traverse the list from left to right**, recursion stops when the index **toIndex** is reached.
  - **Base Case:**

```
LinkedList<E> result;

if (index == toIndex) {
    result = new LinkedList<E>();
    result.addFirst(current.value);
}
```

# Strategy 1

- **General case:**

```
result = subList(current.next, index+1, fromIndex, toIndex);
```

  - What does **result** contain?
  - What's the **next step**?

```
if (index > fromIndex) {
    result.addFirst(current.value);
}
```

```java
public LinkedList<E> subList(int fromIndex, int toIndex) {
    return subList(head, 0, fromIndex, toIndex);
}

private LinkedList<E> subList(Node<E> current, int index, int fromIndex, int toIndex) {

    LinkedList<E> result;

    if (index == toIndex) {

        result = new LinkedList<E>();
        result.addFirst(current.value);

    } else {

        result = subList(current.next, index+1, fromIndex, toIndex);

        if (index >= fromIndex) {
            result.addFirst(current.value);
        }
    }

    return result;
}
```

- The handling of the **preconditions** (range of illegal values) is left as **exercise**.

# Strategy 2

- For the second strategy, the **list of results is created at the outset** and elements are **inserted while traversing** the list.

```
public LinkedList<E> subList(int fromIndex, int toIndex) {

    LinkedList result = new LinkedList<E>();

    subList(head, 0, result, fromIndex, toIndex);

    return result;
}
```

# Strategy 2

- **Base Case:**

```
if (index == toIndex) {
    result.addLast(current.value);
}
```

**result.addLast(current.value)** ou **result.addFirst(current.value)**?

# Strategy 2

- **General case:**

```
if (index >= fromIndex) {
    result.addLast(current.value);
}
subList(current.next, index+1, result, fromIndex, toIndex);
```

**result.addLast(current.value)** ou **result.addFirst(current.value)**?

```java
public LinkedList<E> subList(int fromIndex, int toIndex) {

    LinkedList result = new LinkedList<E>();

    subList(head, 0, result, fromIndex, toIndex);

    return result;
}

private void subList(Node<E> current, int index, LinkedList<E> result,
                                                  int fromIndex, int toIndex) {

    if (index == toIndex ) {

        result.addLast(current.value);

    } else {

        if (index >= fromIndex) {
           result.addLast(current.value);
        }

        subList(current.next, index+1, result, fromIndex, toIndex);
    }

}
```

# Principles

# Principles

**Parameters** play an **essential role** when writing **recursive** methods.

- A parameter of type **Node**, **current**, plays a key role in **controlling the execution** of the method.
  - Sample tests for the **base case**:
    - **current == null**
    - **current.next == null**
    - **current.value.equals(element)**
    - **current.next.value.equals(element)**
  - **General case**:
    - The value of **current.next** is passed as a parameter for the **recursive call** in order to process the rest of the list.

# Principles

There are **two mechanisms** for **passing information between calls**.

1. **Parameters**:
   - **A counter**: If each call must know its position in the list, the value of a counter is passed as a parameter, and each call passes to the value plus one.
     - We could also decrement the value with each call.
2. **Return value**:
   - The method **size** returns the size of the list from the element designated by its parameter **current**.
     - For the caller, the size of the list is one more than the returned value.

```
type method(Node<E> current) {

    type result;

    if (current ...) {            // base case

        calculating the result    // no recursive call

    } else {                      // general case

                                  // pre-processing

        s = method(current.next); // recursion

                                  // post-processing

    }

    return result;
}
```

# «head & tail»

**Steps:**

- What does **method(current.next)** mean?
  - The solution to a problem, **smaller by an element.**
- How are we going to **use this result** to construct a solution for a list beginning with the **current** element?
- What are the **base cases**?
  - What's the **shortest valid list**?
  - What's the **result**?

# Prologue

# Summary

- We proposed the “**head & tail**” strategy
  - **Base case:** usually a test involving the value of **current**.
  - **General case:** recursive call passing the value of **current.next** as a parameter.
- We control recursivity using the **parameters**.

# Next module

- **Binary Research Trees**: concept

# 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**