LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

Restructuration 1
Structure Linéaires Chaînées + Complexité

Pierre Schaus

1.2.1

- Iterable vs Iterator

- Implementation of a CicularLinkedList

1.2.1: Iterable vs Iterator (True or False)

- Iterator is an interface that represents a collection of elements that can be iterated over. It provides the method iterator() that returns an Iterator object (true or false ?)

1.2.1: Iterable vs Iterator (True or False)

- ~~Iterator~~ Iterable is an interface that represents a collection of elements that can be iterated over. It provides the method iterator() that returns an Iterator object

1.2.1: Iterable vs Iterator (True or False)

- Iterator is an interface that provides methods (hasNext(), next(), and remove()) to traverse or iterate over the elements in a collection (true or false ?)

1.2.1: Iterable vs Iterator (True or False)

- Iterator is an interface that provides methods (hasNext(), next(), and remove()) to traverse or iterate over the elements in a collection. ✅

1.2.1: Iterable vs Iterator (True or False)

- The code on the left is a syntactical sugar for the code on the right (true or false ?)

```
for (Integer i: collection) {
    System.out.println(i);
}
```

```
Iterator<Integer> iterator = collection.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

- Let’s verify it by looking at the decompiled code in IntelliJ …

1.2.1: Iterable vs Iterator (True or False)

- This is the proper way of implementing an iterator for my collection ?

```
import java.util.Iterator;

class MyCollection implements Iterable<Integer>, Iterator<Integer> {
    private int[] data = {1, 2, 3, 4, 5};

    private int index = 0;


    @Override
    public Iterator<Integer> iterator() {
        return this;
    }

    @Override
    public boolean hasNext() {
        return index < data.length;
    }

    @Override
    public Integer next() {
        return data[index++];
    }
}
```

1.2.1: Iterable vs Iterator (True or False)

- No ☠️❌☠️❌☠️

```
import java.util.Iterator;

class MyCollection implements Iterable<Integer>, Iterator<Integer> {
    private int[] data = {1, 2, 3, 4, 5};

    private int index = 0;


    @Override
    public Iterator<Integer> iterator() {
        return this;
    }

    @Override
    public boolean hasNext() {
        return index < data.length;
    }

    @Override
    public Integer next() {
        return data[index++];
    }
}
```

What do you think this will print ?

```
public static void main(String[] args) {
    MyCollection collection = new MyCollection();
    for (int i : collection) {
        for (int j : collection) {
            System.out.println(i + "," + j);
        }
    }
}
```

1,2
1,3
1,4
1,5

1.2.1: Iterable vs Iterator (True or False)

- This is the proper way 👇👌✅

```java
import java.util.Iterator;

class MyCollection implements Iterable<Integer> {
    private int[] data = {1, 2, 3, 4, 5};

    @Override
    public Iterator<Integer> iterator() {
        return new MyIterator();
    }

    // Separate Iterator class
    private class MyIterator implements Iterator<Integer> {
        private int index = 0;

        @Override
        public boolean hasNext() {
            return index < data.length;
        }

        @Override
        public Integer next() {
            return data[index++];
        }
    }
}
```

With an inner-class that contains its own state (index instance variable)

```java
public static void main(String[] args) {
    MyCollection collection = new MyCollection();
    for (int i : collection) {
        for (int j : collection) {
            System.out.println(i + "," + j);
        }
    }
}
```

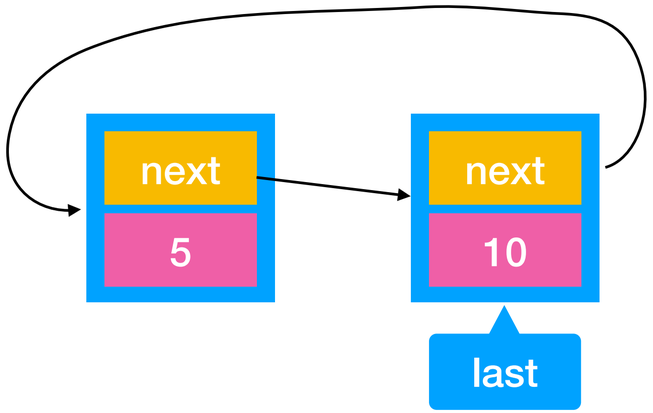
1,1
1,2
1,3
1,4
1,5
2,1
2,2
2,3
2,4
...
5,5

Les private inner classes sont très utiles pour implémenter des iterateurs car celles-ci ont accès à l'état de l'outer class

1.2.1 CircularLinkedList, ChatGPT Solution 🤔

```java
public class CircularLinkedList<Item> implements Iterable<Item> {

    private int n;          // size of the list
    private Node last;      // trailer of the list (points to the last node)

    private class Node {
        private Item item;
        private Node next;
    }
    public CircularLinkedList() {
        n = 0;
        last = null;
    }
    public boolean isEmpty() {
        return n == 0;
    }
    public int size() {
        return n;
    }
    public void enqueue(Item item) {
        Node newNode = new Node();
        newNode.item = item;
        if (isEmpty()) {
            newNode.next = newNode;   // Points to itself since it's the only node
            last = newNode;
        } else {
            newNode.next = last.next; // The new node points to the first node
            last.next = newNode;      // The current last node points to the new node
            last = newNode;           // Update the last pointer to the new node
        }
        n++;  // Increase the size of the list
    }
}
```

Two elements

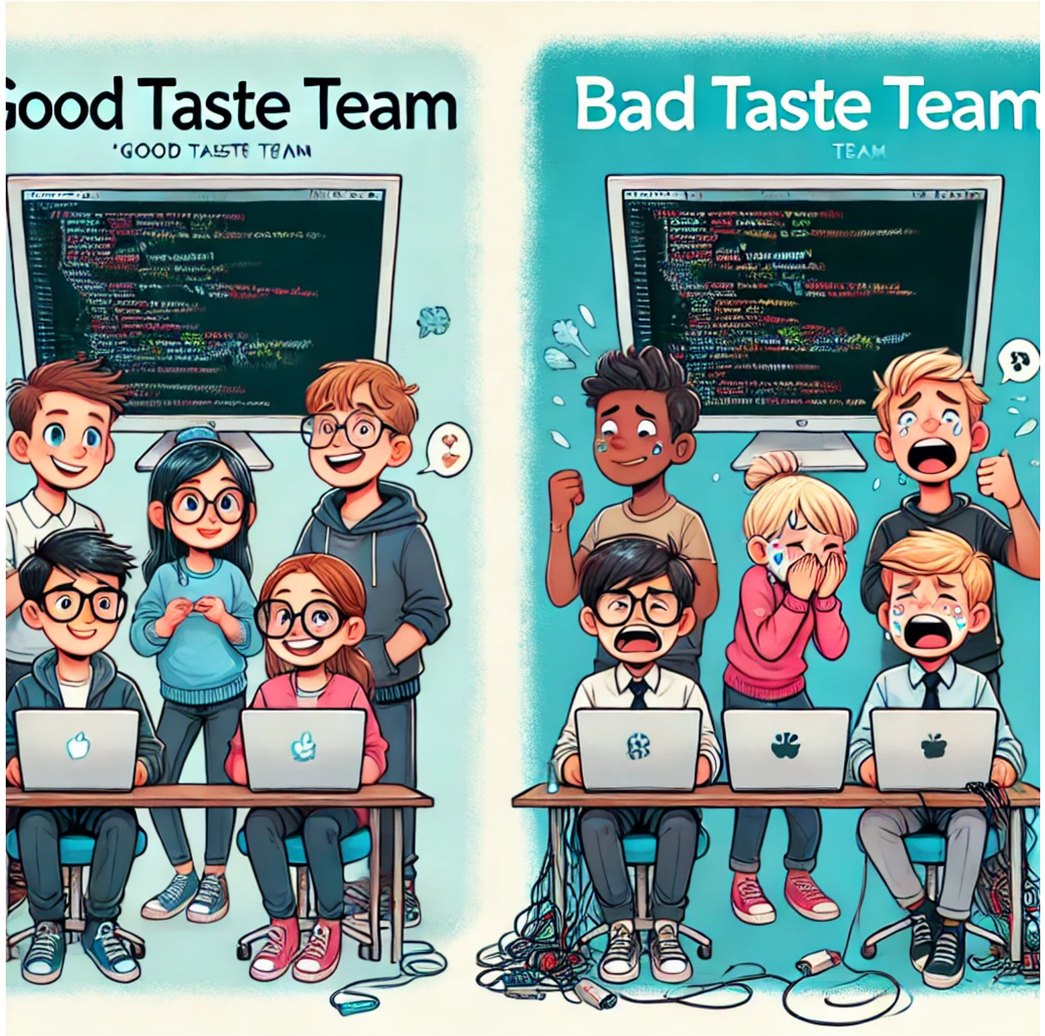



```java
public Item remove(int index) {
    if (index < 0 || index >= n) {
        throw new IndexOutOfBoundsException();
    }

    Node current = last.next;  // Start from the first node
    Item removedItem;

    if (index == 0) {  // Removing the first node
        removedItem = current.item;
        if (n == 1) {  // If there's only one node
            last = null; // The list becomes empty
        } else {
            last.next = current.next; // Bypass the first node
        }
    } else {
        Node prev = last.next;  // Start from the first node
        for (int i = 0; i < index - 1; i++) {
            prev = prev.next;
        }
        current = prev.next;  // The node to remove
        removedItem = current.item;
        prev.next = current.next; // Bypass the node to remove
        if (current == last) {    // If the last node is removed
            last = prev;
        }
    }
    n--;   // Decrease the size
    return removedItem;
}
```

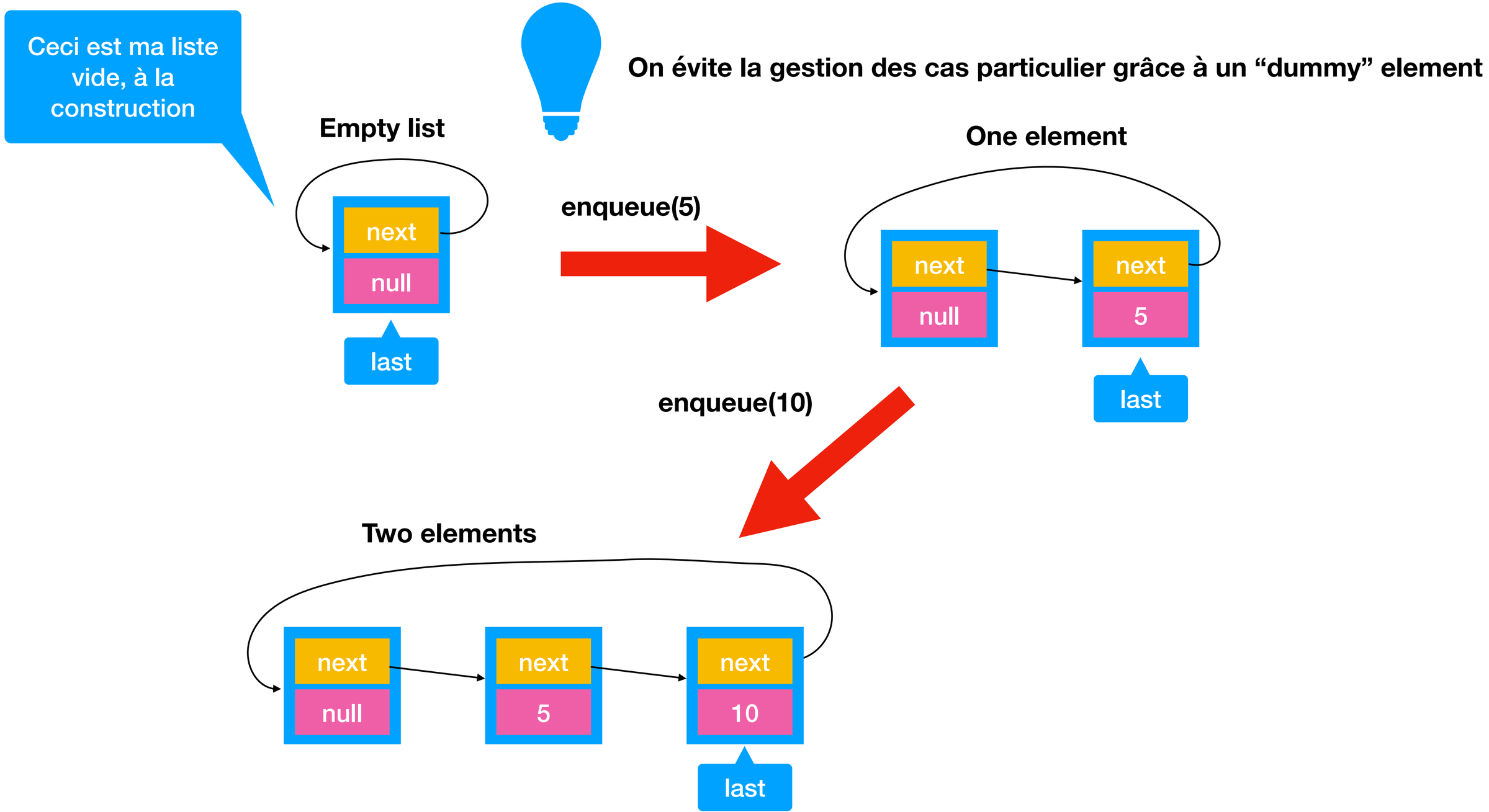
Good Taste in Coding (Linus Torvalds, creator of Linux)

What is your team ?

- Then think a bit ahead before diving in the code, use a 📝 and draw!



1.2.1 CircularLinkedList, let’s get rid of the special cases



1.2.1 CircularLinkedList, let's get rid of the special cases

Now (Good Taste Solution)

```
public class CircularLinkedList<Item> implements Iterable<Item> {

   private int n;           // size of the stack
    private Node  last;     // trailer of the list

    // helper linked list class
    private class Node {
        private Item item;
        private Node next;
    }

    public CircularLinkedList() {
        last = new Node();
        last.next = last;
        n = 1;
    }

    public boolean isEmpty() {
        return n == 1;
    }
    public int size() {
        return n-1;
    }
    private long nOp() {
        return nOp;
    }

    public void enqueue(Item item) {
        Node oldLast = last;
        last = new Node();
        last.item = item;
        last.next = oldLast.next;
        oldLast.next = last;
        n++;

    }
}
```

Before

```
public void enqueue(Item item) {
    Node newNode = new Node();
    newNode.item = item;
    if (isEmpty()) {
        newNode.next = newNode;
        last = newNode;
    } else {
        newNode.next = last.next;
        last.next = newNode;
        last = newNode;
    }
    n++;
}
```

1.2.1 CircularLinkedList, let's get rid of the special cases

Good Taste

```
public Item remove(int index) {
    if (index < 0 || index >= size()) {
        throw new IndexOutOfBoundsException();
    }
    Node prev = last.next;
    for (int i = 0; i < index; i++) {
        prev = prev.next;
    }
    Item v = prev.next.item;
    prev.next = prev.next.next;
    n--;
    return v;
}
```

Before (Bad Taste)

```
public Item remove(int index) {
    if (index < 0 || index >= n) {
        throw new IndexOutOfBoundsException();
    }

    Node current = last.next;  // Start from the first node
    Item removedItem;

    if (index == 0) {  // Removing the first node
        removedItem = current.item;
        if (n == 1) {  // If there's only one node
            last = null; // The list becomes empty
        } else {
            last.next = current.next; // Bypass the first node
        }
    } else {
        Node prev = last.next;  // Start from the first node
        for (int i = 0; i < index - 1; i++) {
            prev = prev.next;
        }
        current = prev.next;  // The node to remove
        removedItem = current.item;
        prev.next = current.next; // Bypass the node to remove
        if (current == last) {    // If the last node is removed
            last = prev;
        }
    }
    n--;   // Decrease the size
    return removedItem;
}
```

1.2.1 CircularLinkedList

- Quelle est la complexité de: `public void enqueue(Item item)` ?
  - O(1)
  - O(n) où n est le nombre d'entrées dans la liste
  - $\Theta(n)$ où n est le nombre d'entrées dans la liste
  - O(1) ammorti

1.2.1 CircularLinkedList

- Complexité de: `public Item remove(int index)` ?
  - O(1)
  - O(n) où n est le nombre d'entrées dans la liste
  - $\Theta(n)$ où n est le nombre d'entrées dans la liste
  - O(1) ammorti

1.2.1 Rappel: Concurrent Modification

- Une collection ne peut généralement pas être modifiée alors qu'un iterator est utilisé sur celle-ci. C'est la stratégie *Fail-Fast*

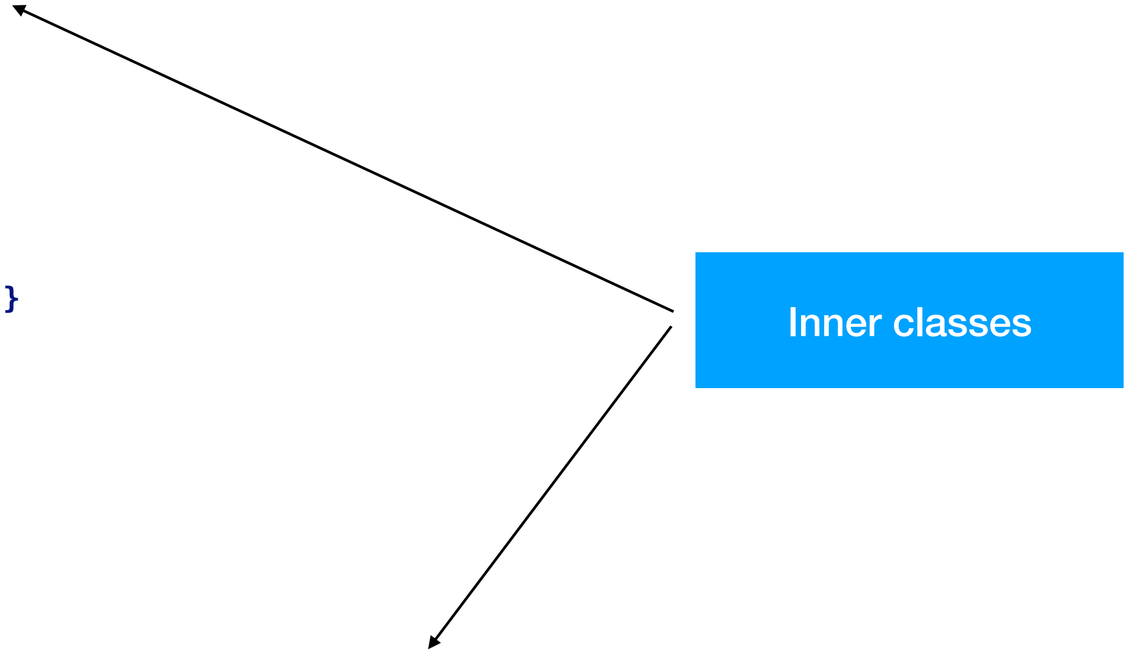
- Si entre deux « hasNext/next » la collection est modifiée, il faut lancer un « ConcurrentModificationException ».

```
List<Integer> collection = new LinkedList<>();
collection.add(1);
collection.add(2);
for (Integer i: collection) {
    collection.add(i);
}
```

Exception in thread "main"
java.util.ConcurrentModificationException

1.2.1 CircularLinkedList

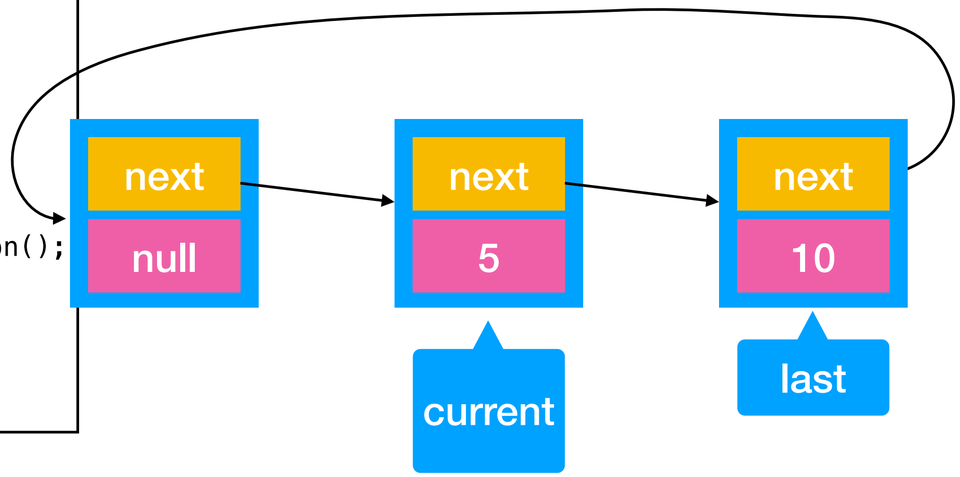
```
public class CircularLinkedList<Item> implements Iterable<Item> {

    private int n;           // size of the stack
    private Node  last;      // trailer of the list

    private class Node {
        private Item item;
        private Node next;
    }

    public CircularLinkedList() {
        last = new Node();
        last.next = last;
        n = 1;
    }
    public boolean isEmpty() { return n == 1; }
    public int size() {
        return n-1;
    }
    public void enqueue(Item item) { … }
    public Item remove(int index) { … }

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

    private class ListIterator implements Iterator<Item> {
        private ListIterator() { … }
        public boolean hasNext()  { … }
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public Item next() { … }
    }
}
```



1.2.1

```
public class CircularLinkedList<Item> implements Iterable<Item> {

    private long nOp = 0; // count the number of operations
    private int n;          // size of the stack
    private Node  last;   // trailer of the list

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

    private class ListIterator implements Iterator<Item> {

        private Node current;

        private ListIterator() {
           current = last.next.next;
        }

        public boolean hasNext()  {
           return current != last.next;
        }

        public void remove()      {
            throw new UnsupportedOperationException();
        }

        public Item next() {
            if (!hasNext()) throw new NoSuchElementException();
            Item item = current.item;
            current = current.next;
            return item;
        }
    }

}
```

L'itérateur a donc son propre état interne (variable d'instance current propre à l'instance de itérateur au sein d'une instance particulière d'une CircularLinkedList) qui est le noeuds avec le prochain item à retourner



Et le ConcurrentModificationException ?

- On va stocker un compteur interne à la liste qui compte les operations « enqueue » et « remove »

- A chaque operation « enqueue » ou « remove » on va augmenter ce compteur.

- Au moment de créer l'itérateur, on va y stocker (variable d'instance) la variable du compteur.

- Si lorsqu'on fait « hasNext() » ou « next() », on réalise que la valeur du « compteur » enregistrée dans l'itérateur est plus petite que celle du compteur de la liste, cela signifie que l'utilisateur a modifié la liste alors qu'il est en train d'utiliser l'itérateur => ConcurrentModificationException

Iterateur avec détection de modification

```
public class CircularLinkedList<Item> implements Iterable<Item> {

    private long nOp = 0;    // count the number of operations
    private int n;           // size of the stack
    private Node  last;      // trailer of the list

    …

    private long nOp() {
        return nOp;
    }

    public void enqueue(Item item) {
        nOp++;
        …
    }
    public Iterator<Item> iterator() {
        return new ListIterator();
    }
```

```
    private class ListIterator implements Iterator<Item> {

        private Node current;
        private long nOp;

        private ListIterator() {
            nOp = nOp();
            current = last.next.next;
        }

        public boolean hasNext()  {
            if (nOp() != nOp) throw new ConcurrentModificationException();
            return current != last.next;
        }
    }
```

```
}
```

Augmente le nombre d'opérations

Capture du nombre d'opérations sur la liste au moment de la création de l'itérateur

Nombre d'opérations sur la liste au temps présent

Nombre d'opérations sur la liste à la création de l'itérateur

CircularLinkedList

- Quelle est la complexité de créer un iterateur et ensuite itérer sur les k-premiers elements ?
  - O(n) où n est le nombre d'entrées dans la liste
  - Θ(n) où n est le nombre d'entrées dans la liste
  - O(k)
  - Θ(k)

1.2.2 Implementation of a Stack with an

- Array

- LinkedList

1.2.2 Implementation of a Stack with an Array

Strategy: Resize the array when reaching it's capacity (in practice: doubling it's size). How ?

```
class ArrayStack<E> implements Stack<E> {

    private E[] array;          // array storing the elements on the stack
    private int size;          // size of the stack

    public ArrayStack() {
        array = (E[]) new Object[10];
    }
    @Override
    public boolean empty() {
        return size == 0;
    }
    @Override
    public E peek() throws EmptyStackException {
        if (empty()) throw new EmptyStackException();
        else return array[size-1];
    }
    @Override
    public void push(E item) {
        if (size == array.length) {
            resize(array.length * 2); // doubling the size of the array
        }
        array[size++] = item;
    }
    // pop on next slide
    private void resize(int newCapacity) {
        E[] newArray = (E[]) new Object[newCapacity];
        System.arraycopy(array, 0, newArray, 0, size);
        array = newArray;
    }
}
```

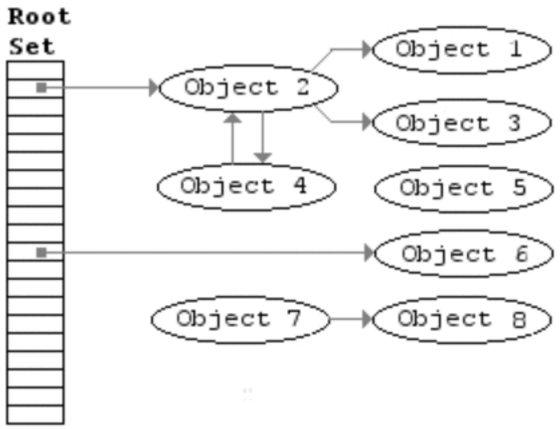
Might be more efficient that the for loop for copying

1.2.2 Implementation of a Stack with an Array

Why resizing for the pop ?

```
class ArrayStack<E> implements Stack<E> {

    @Override
    public E pop() throws EmptyStackException {
        if (empty()) throw new EmptyStackException();
        else {
            E ret = array[size-1];
            size--;
            array[size] = null; // so that the object can possibly be garbage collected
            // Resize the array if size is less than half the length
            if (size > 0 && size <= array.length / 2) {
                resize(array.length / 2);
            }
            return ret;
        }
    }
}
```

To allow the garbage collector to reclaim unused memory as soon as possible



1.2.2 Implementation of a Stack with an Array, Time Complexity

	Liste chainée	Tableau
Push	O(1)	O(1) si pas de redim O(n) si redim
Pop	O(1)	O(1) si pas de redim O(n) si redim
n push	O(n)	$O(n^2)$ 🤔 ??
n pop	O(n)	$O(n^2)$ 🤔 ??

1.2.2 Stack avec tableau: on raffine un peu l'analyse

Réfléchissons deux minutes: Quand est-ce qu'on fait un redimensionnement?

	Liste chainée	Tableau
Push	O(1)	O(1) si pas de redim O(n) si redim
Pop	O(1)	O(1) si pas de redim O(n) si redim
n push	O(n)	$O(n^2)$ 🤔 ??
n pop	O(n)	$O(n^2)$ 🤔 ??

On double la taille du tableau quand on atteint la limite.

Sur n push, on fait donc un resize à ces positions:

- 1
- 2
- 4
- 8
- 16
- ...
- n

Donc sur les n-push, nous avons fait $\log_2(n)$ redimensionnements. Chaque redimensionnement m'a couté maximum $\mathcal{O}(n)$ donc au total, les n-push sont en $\mathcal{O}(n \log(n))$.

Est-ce qu'on peut être plus précis ?

1.2.2 Stack avec tableau: on raffine un peu l'analyse

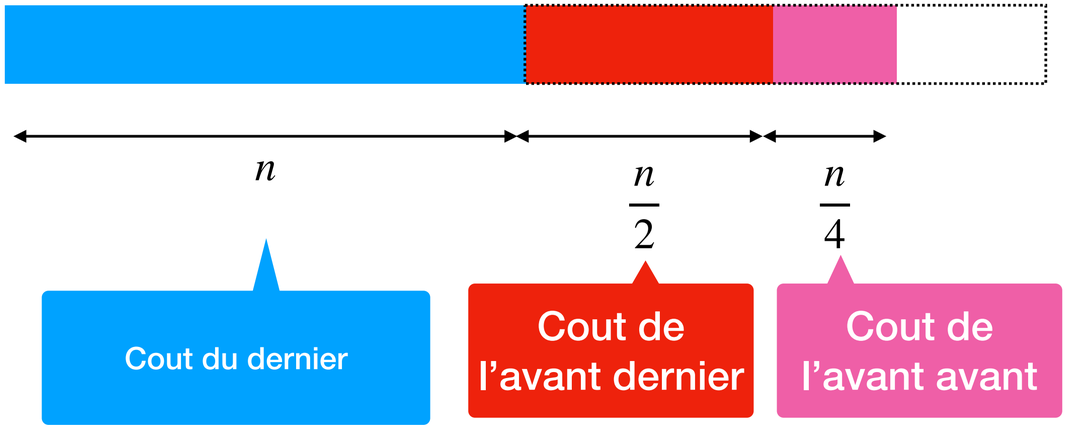
Réfléchissons deux minutes: Quand est-ce qu'on fait un redimensionnement?

	Liste chainée	Tableau
Push	O(1)	O(1) si pas de redim O(n) si redim
Pop	O(1)	O(1) si pas de redim O(n) si redim
n push	O(n)	O(n log(n)) 🤔 ??
n pop	O(n)	O(n log(n)) 🤔 ??

On double la taille du tableau quand on atteint la limite.

Sur n push, on fait donc un resize à ces positions:

- 1
- 2
- 4
- 8
- 16
- ...
- n



1.2.2 Conclusion

java.util.Stack a de très bonnes complexités amorties pour n operations

	Liste chainée	Tableau
Push	O(1)	O(1) Sauf redim. O(n)
Pop	O(1)	O(1) Sauf redim. O(n)
n push	O(n)	O(n)
n pop	O(n)	O(n)

On double la taille du tableau quand on atteint la limite.

Sur n push, on fait donc un resize à ces positions:

- 1
- 2
- 4
- 8
- 16
- ...
- n

COMPLEXITÉS AMORTIES

Combien de redim.?

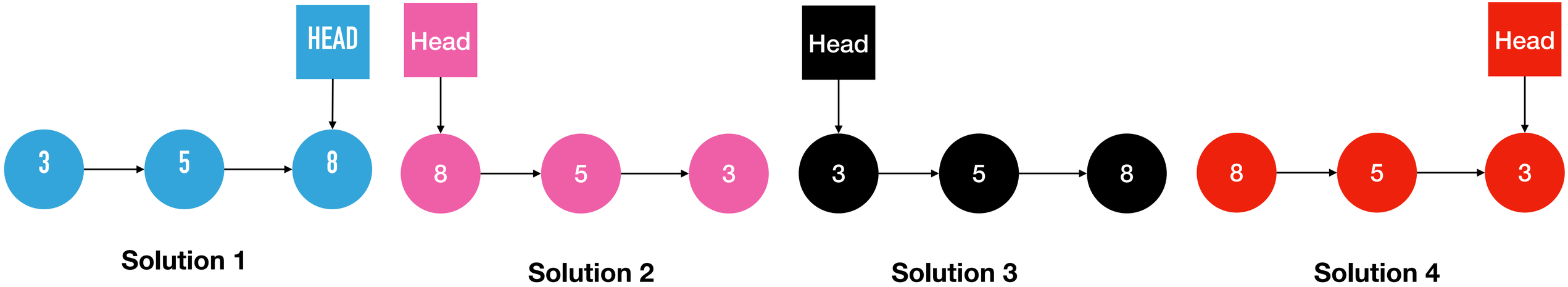
$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} < 2n \in \mathcal{O}(n)$$

1.2.2 Stack with a simply LinkedList

Pile (stack) avec une liste simplement chaînée

push(3)
push(5)
push(8)

Qu’est-ce qu’on choisit comme représentation pour faire push/pop efficacement ?



Solution 1

Solution 2

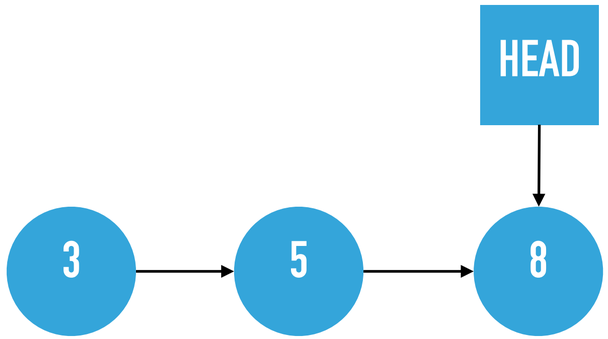
Solution 3

Solution 4

1.2.2 Stack with a simply LinkedList

Pile (stack) avec une liste simplement chaînée

push(3)
push(5)
push(8)



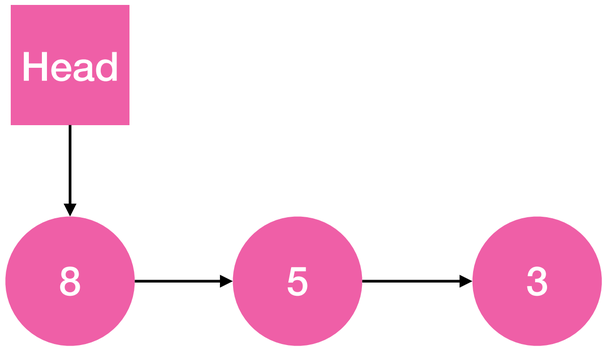
Solution 1

Ok pour les push O(1) ✅, pop impossible ❌

1.2.2 Stack with a simply LinkedList

Pile (stack) avec une liste simplement chaînée

push(3)
push(5)
push(8)



Ok pour les push O(1) ✅, ok pour pop O(1) ✅

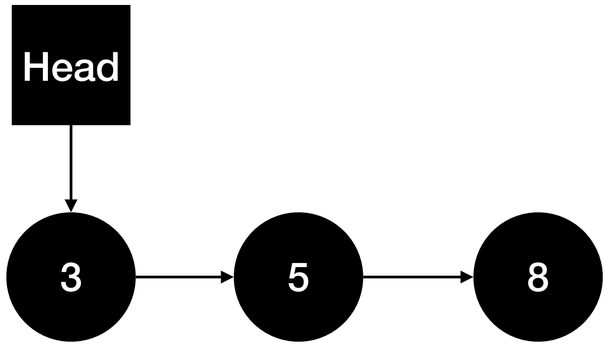
Solution 2

1.2.2 Stack with a simply LinkedList

Pile (stack) avec une liste simplement chaînée

push(3)
push(5)
push(8)

Ok pour les push O(n) ✅, ok for pop O(n) ✅



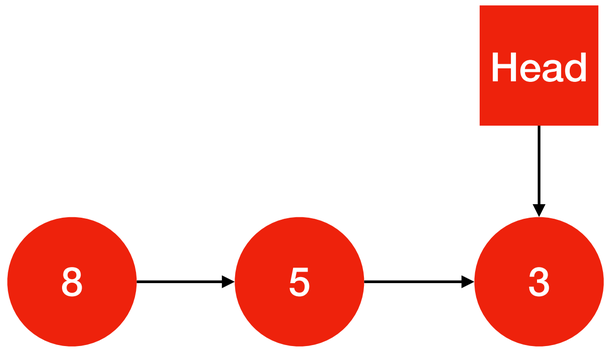
Solution 3

1.2.2 Stack with a simply LinkedList

Pile (stack) avec une liste simplement chaînée

push(3)
push(5)
push(8)

push impossible ✗, pop impossible ✗



Solution 4

1.2.3 Evaluation d’expression post-fixe

- Par exemple "2 3 1 * + 9 * »

- Ces expressions peuvent être évaluées facilement à l’aide d’une:
  - Queue (FIFO) ?
  - Stack (LIFO) ?
  - Un arbre ?

Post-fix evaluation: Algorithm

```java
public static int evaluate(char[] input) {
    Stack<Integer> stack = new Stack<>();
    int i = 0;

    while (i < input.length) {
        char current = input[i];

        // If the current character is a number
        if (Character.isDigit(current)) {
            // Convert char to int and push onto stack
            stack.push(Character.getNumericValue(current));
        }
        // If the current character is an operator
        else if (isOperator(current)) {
            // Pop two numbers from the stack
            int operand2 = stack.pop();  // Second operand
            int operand1 = stack.pop();  // First operand

            // Apply the operator and push the result back onto the stack
            int result = applyOperator(operand1, operand2, current);
            stack.push(result);
        }
        i++;  // Move to the next character
    }

    // The final result is the last value in the stack
    return stack.pop();
}
```

4	2	+	3	5	1	*	*	-

```java
// Helper method to check if a character is an operator
private static boolean isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Helper method to apply an operator to two operands
private static int applyOperator(int operand1, int operand2, char operator) {
    switch (operator) {
        case '+': return operand1 + operand2;
        case '-': return operand1 - operand2;
        case '*': return operand1 * operand2;
        case '/': return operand1 / operand2;
        default: throw new IllegalArgumentException("Invalid operator: " + operator);
    }
}
```

Complexité temporelle d’une éval post-fix ?

- En supposant un push/pop en O(1) et n la taille de l’input:
  - $\Theta(n)$
  - $\Theta(n^2)$
  - $O(n)$
  - $O(n^2)$

1.2.4 Functional List (FList)

- C’est une **liste immutable** (ne peut jamais être modifiée après sa création) récursive

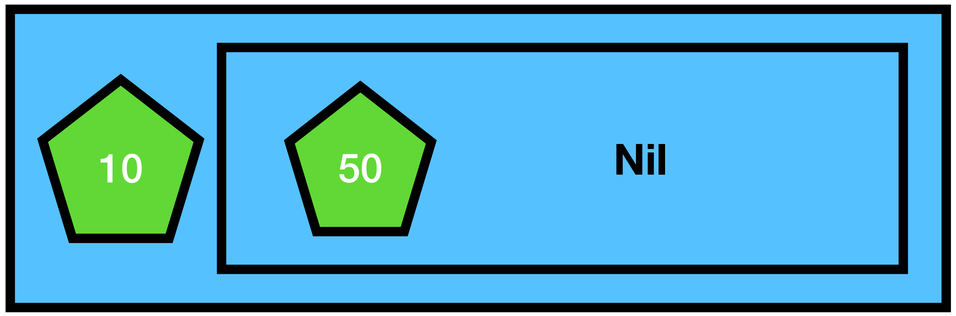
- On peu juste la créer (avec la méthode cons) et suite itérer dessus. Exemple:

```
public static void main(String[] args) {
    FList<Integer> list = FList.nil();

    for (int i = 0; i < 10; i++) {
        list = list.cons(i);
    }

    list = list.map(i -> i+1);
    // will print 10,9,...,1
    for (Integer i: list) {
        System.out.println(i);
    }

    list = list.filter(i -> i%2 == 0);
    // will print 10,...,6,4,2
    for (Integer i: list) {
        System.out.println(i);
    }
}
```

Les listes fonctionnelles: FList

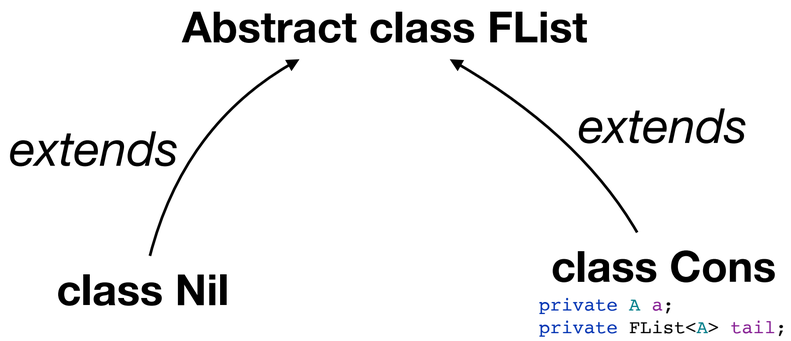
- Définition récursive, une FList est soit:
  - Une liste vide (Nil)
  - Un élément suivit d'une FList

- Exemple pour la liste [10,50]



- Notre implémentation va refléter fidèlement cette définition

Diagramme de classe



Complexité ?

```
public abstract class FList<A> implements Iterable<A> {
    // creates an empty list

    public static <A> FList<A> nil();

    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isNotEmpty();
    public final boolean isEmpty();
    public final int length();
    // return the head element of the list
    public abstract A head();
    // return the tail of the list
    public abstract FList<A> tail();
    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);
    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);
    // return an iterator on the element of the list
    public Iterator<A> iterator();

}
```

La liste vide

Ajoute un element a la liste et retourne la nouvelle liste.

Complexité ? :

- O(1)
- O(n)
- Θ(n)

Quelle est la complexité spatiale de cette méthode?

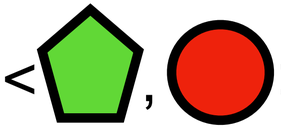
- $\Theta(n)$
- $\Theta(n^2)$
- $O(n)$
- $O(n^2)$

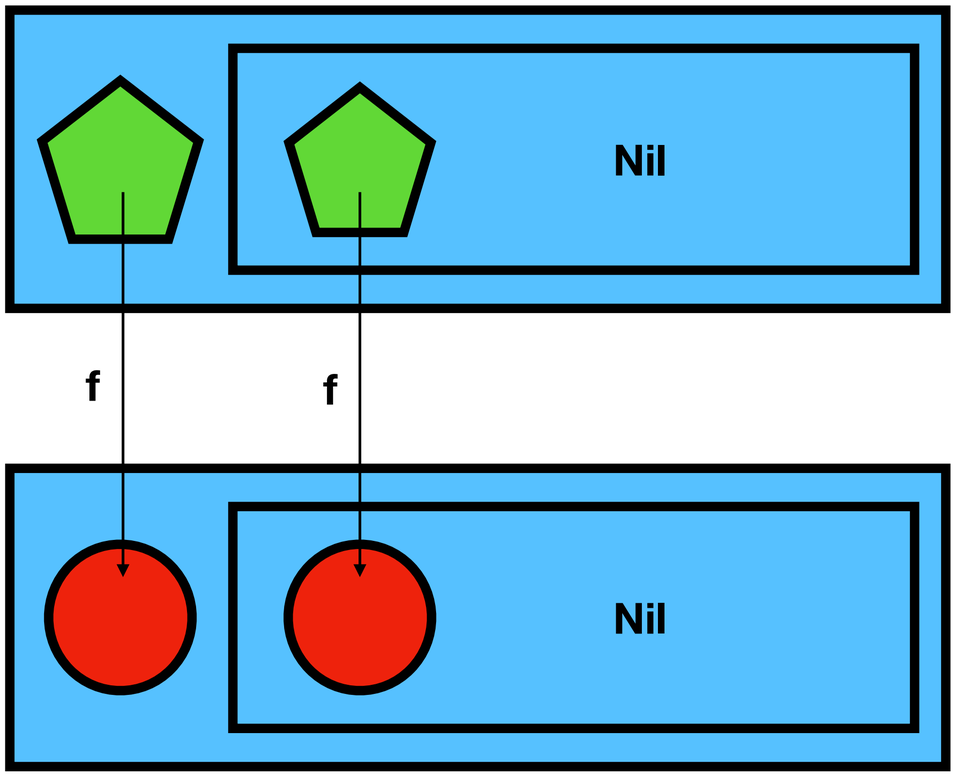
```java
/**
 * @param n the size of the list
 * @return The list [0,1,...,n-1]
 */
public static FList<Integer> rangeList(int n) {
    FList<Integer> list = FList.nil();
    for (int i = n-1; i >= 0; i--) {
        list = list.cons(i);
    }
    return list;
}
```

La méthode « map »

- Appliquée sur une FList<A>, prend en argument « une fonction » f<A,B> et reconstruit une liste de type FList<B>

```
public final <B> FList<B> map(Function<A,B> f)
```

- Exemple pour f 



La méthode « map »

- Exemple:

```
FList<Integer> list = FList.nil();

for (int i = 9; i >= 0; i--) {
    list = list.cons(i);
}

Function<Integer,String> f = new Function<Integer, String>() {
    @Override
    public String apply(Integer k) {
        String res = "";
        for (int i = 0; i < k; i++) {
            res += "*";
        }
        return res;
    }
};

FList<String> rList = list.map(f);
for (String r: rList) {
    System.out.println(r);
}
```

- TimeComplexity ?
  - $\Theta(n)$
  - $\Theta(n^2)$
  - $O(n)$
  - $O(n^2)$

Sucre syntaxique (since Java8) pour les fonctions

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

```java
FList<Integer> list = FList.nil();

for (int i = 9; i >= 0; i--) {
    list = list.cons(i);
}

FList<String> rList = list.map(k -> {
    String res = "";
    for (int i = 0; i < k; i++) {
        res += "*";
    }
    return res;
});

for (String r: rList) {
    System.out.println(r);
}
```

=

```java
FList<String> rList = list.map(
  new Function<Integer, String>() {
    @Override
    public String apply(Integer k) {
        String res = "";
        for (int i = 0; i < k; i++) {
            res += "*";
        }
        return res;
    }
});
```

Implémentation implicite de l'unique méthode de l'interface « fonctionnelle »

LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

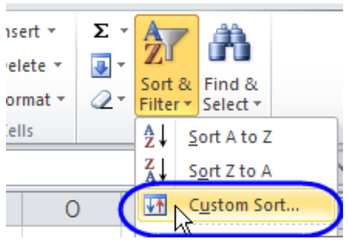
Intro 2

Algorithmes de tri et recherche dichotomique

Pierre Schaus

Trier des données

- Une des opérations les plus courantes en informatique



- Au début de l'informatique (pas si longtemps) on disait que 30% de ressources calcul étaient utilisées à trier.

- Aujourd'hui, c'est probablement moins grâce à des algorithmes de tri très efficaces.

Pourquoi étudier les algo de tri?

- D’un point de vue théorique, ils sont très intéressants et constituent un excellent exemple de comparaison d’algorithmes (mémoire, complexité, etc)
- Ils sont à la base de nombreux autres algorithmes.
- C’est un “block” de base algorithmique essentiel qu’il faut bien maitriser.
- Les idées des algorithmes de tris sont assez génériques et peuvent être réutilisées pour résoudre d’autres problèmes (divide and conquer, merging, pivoting, etc).

Les questions à se poser en lisant

- Pourquoi tant d'algorithmes de tri, est-ce qu'il existe des avantages et inconvénients pour chacun d'eux ?

- Est-il possible d'implémenter un algorithme de tri générique capable de trier n'importe quel objet ?

- Est-ce que la complexité dépend des objets à trier ?

- Est-ce que je peux trier n'importe quelle structure de données linéaire (linkedList, array, etc) ou seulement les tableaux ?
  - Quelle est l'API minimum d'une structure de données linaire pour pouvoir la trier ?

- Est-ce que je peux trier sans utiliser d'espace additionnel ?

Les questions à se poser en lisant

- O(n log(n)) vs O(n^2) est-ce que ça fait une grosse différence ?
- Est-ce qu'on parle de complexité attendue, pire-case (O, Theta, ~ ?)
- Est-ce qu'on peut un jour espérer trier encore plus rapidement que O(n log(n)) ?
  - Que représente exactement cette complexité quand on parle d'algorithme de tri ?

Les questions à se poser en lisant

- Qu’est ce qu’on entend par tri stable ? Pourquoi est-ce (parfois) important ?

- Est-ce que c’est plus couteux de faire un tri lexicographique plutôt qu’un tri sur des entiers ?

Important de se procure une copie du livre

Votre sentiment après ce premier module ?

- Livre ?
- Organisation du cours ?
- Inginious ?
- Charge de travail ?