LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

Restructuration 1
Structure Linéaires Chaînées + Complexité

Pierre Schaus

Quelles implémentations possibles pour une stack ?

- Avec un tableau

- Avec une liste chainée

Dans la version de Java ?

java.util.Stack est-il implémenté via un tableau (extension de Vector).

Est-ce si inefficace que cela 🙀?

	Liste chainée	Tableau
Push	O(1)	O(n) ?
Pop	O(1)	O(n) ?
n push	O(n)	O(n^2) ?
n pop	O(n)	O(n^2) ?

Stack avec tableau: on raffine un peu l'analyse

	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^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

Combien de redim.?

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

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.?

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

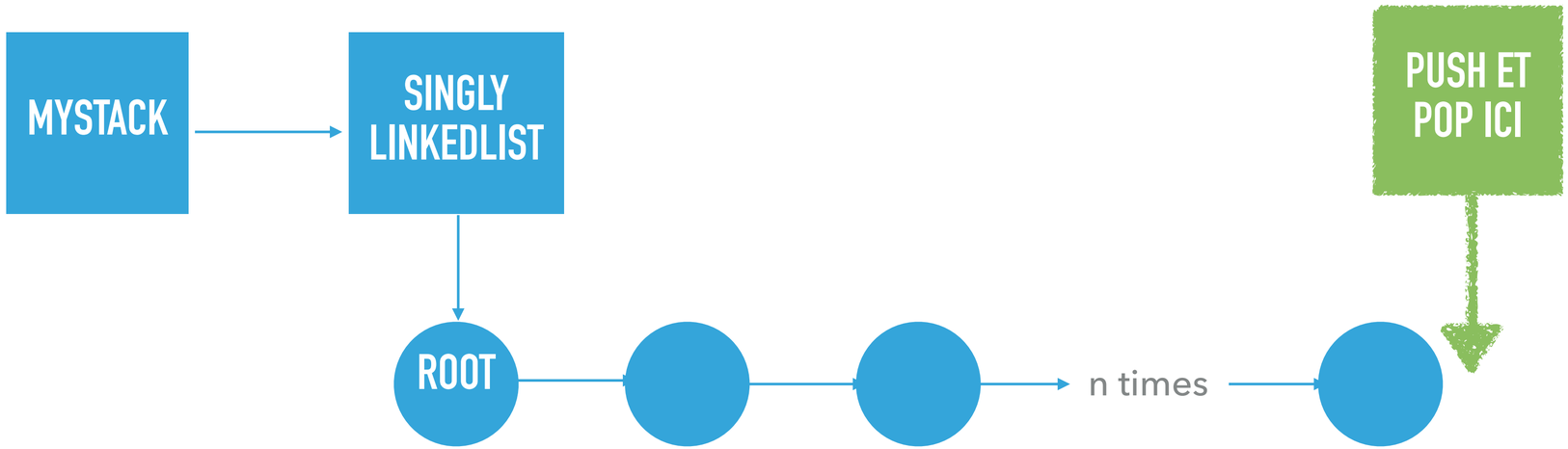
Implémentation

D’une pile (stack) avec une liste simplement chaînée, où les opérations se font en fin de liste?

Complexité d’un push/pop ?

$$\mathcal{O}(1) \quad \sim 1 \quad \Omega(1) \quad \Theta(1)$$

$$\mathcal{O}(n) \quad \sim n \quad \Omega(n) \quad \Theta(n)$$

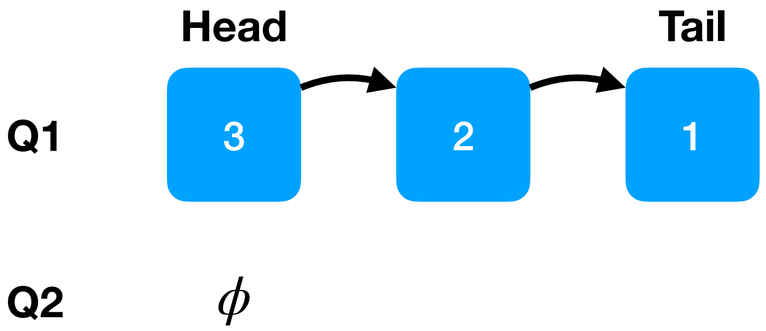


Implémentation

- D’une Stack avec deux Queue’s. Soit Q1 et Q2 les deux queues internes

- Nous allons maintenir l’invariant que Q1 contient les elements de la stack de sorte que Q1.dequeue sorte le sommet de la stack, etc.

- Supposons que nous avons fait push(1), push(2), push(3). Notre invariant doit contenir



- Si je fais push(4) ensuite … j’ajoute 4 (enqueue) dans Q2, et je vide Q1 dans Q2, puis j’échange les références.

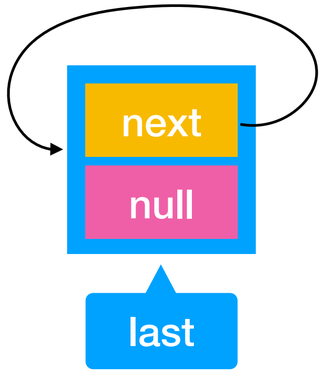
- Complexité push / pop ?

CircularLinkedList

Ceci est ma liste vide, à la construction

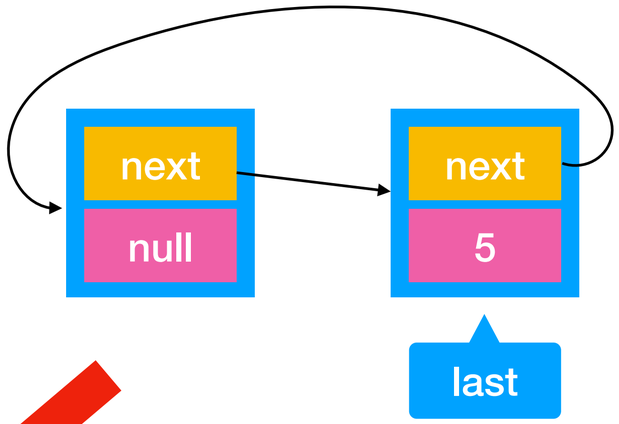
On évite la gestion des cas particulier grâce à un “dummy” element

Empty list



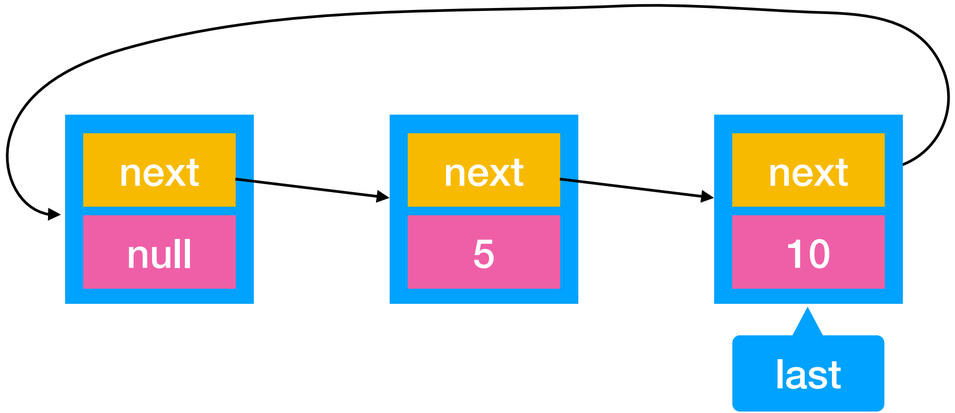
enqueue(5)

One element



enqueue(10)

Two elements



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

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

Iterable et Iterator: rappel

- *Iterable* = une interface avec a une méthode *iterator()* pour générer un Iterator

- Un *Iterator* = une interface pour itérer sur des collections avec les méthodes hasNext() et next()

Modifier and Type	Method and Description
`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration.
`void`	**`remove`**`()` Removes from the underlying collection the last element returned by this iterator (**optional operation**).

La méthode «“remove” est optionnelle et généralement pas implémentée. Si pas implémentée, il faut lancer une « NotImplementedException » pour éviter de ne rien faire silencieusement.

Iterable, iterator et boucle for

- Une collection qui implémente *Iterable* peut être utilisée dans les boucles for:

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

=

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

Concurrent Modification

- Une collection ne peut généralement pas être modifiée alors qu'un iterator est utilisé sur celle-ci.

- 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

- LEPL1402 RAPPEL: Cette stratégie est appelée *"Fail Fast"*

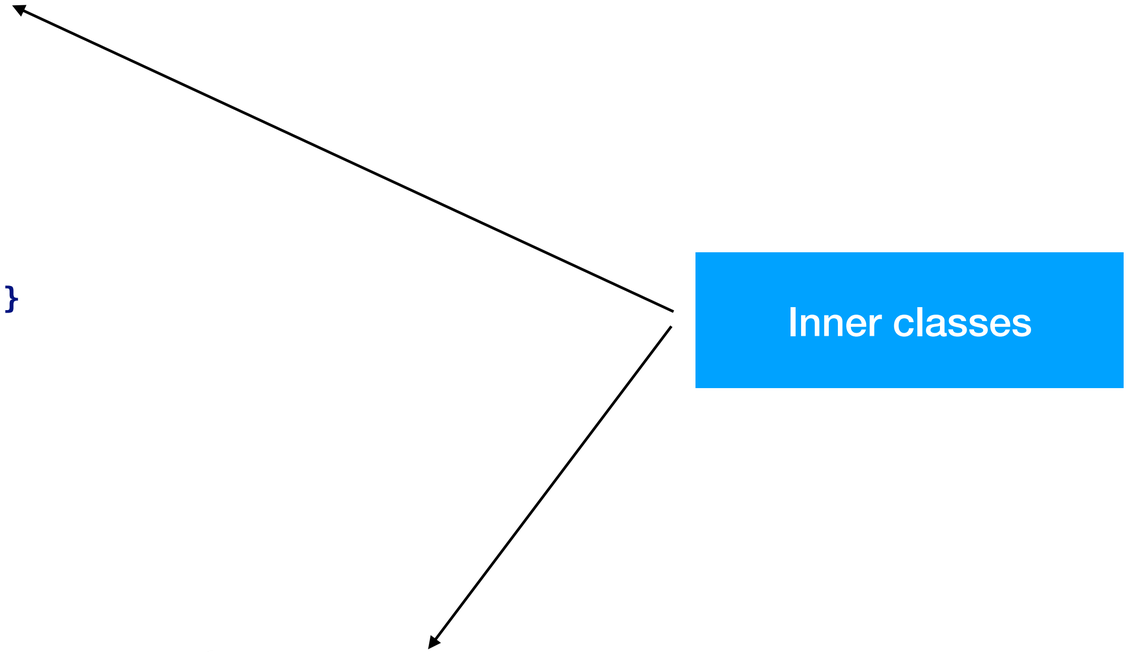
Inner vs Static Nested classe en Java

```java
public class OuterClass {

    public int a = 0;

    class InnerClass {
        public void foo() {
            a = 2;
        }
    }
    static class StaticNestedClass {
        public void foo() {
            // cannot touch a;
        }
    }
    public static void main(String[] args) {
        StaticNestedClass nested = new StaticNestedClass();
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.foo(); // will change the value a in outer
        System.out.println(outer.a);
    }
}
```

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

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() { … }
    }
}
```



Résultat si on utilise deux iterators en même temps ?

```
CircularLinkedList<Integer> list = new CircularLinkedList<>();
list.enqueue(1);
list.enqueue(2);
list.enqueue(3);

for (int i : list) {
    for (int j : list) {
        System.out.println(i+","+j);
    }
}
```

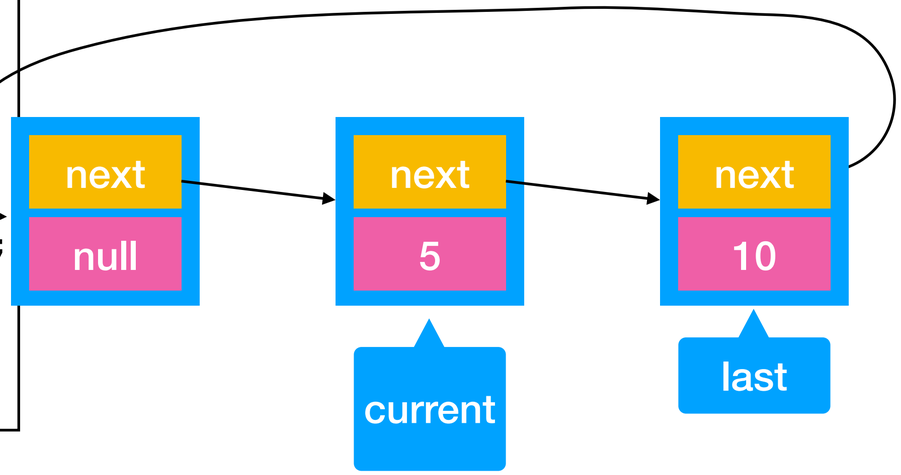
A	B	C	D
1,1	1,2	1,2	ConcurrentModificationException
1,2	1,3	1,3	
1,3	2,3		
2,1			
2,2			
2,3			
3,1			
3,2			
3,3			

ListIterator

```
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)

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

- input

4	20	+	3	5	1	*	*	-

- algo:
  - stack = new Stack()
  - i = 0
  - While (i < input.length) :
    - input[i] = a number => stack.push(input[i])
    - input[i] = an operator => stack.push(stack.pop() input[i] stack.pop())
    - i++
  - return stack.pop()

Complexité temporelle d'une éval post-fix ?

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

Functional List (FList)

- C’est une liste immutable (récursive)

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

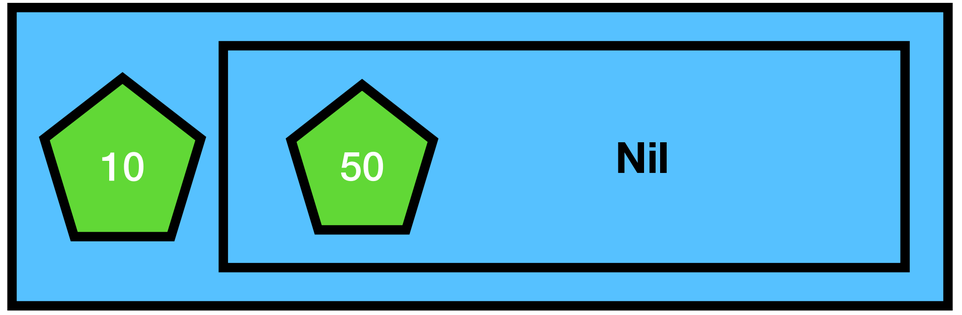
```
public static void main(String[] args) {
    FList<Integer> list = FList.nil();

    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:

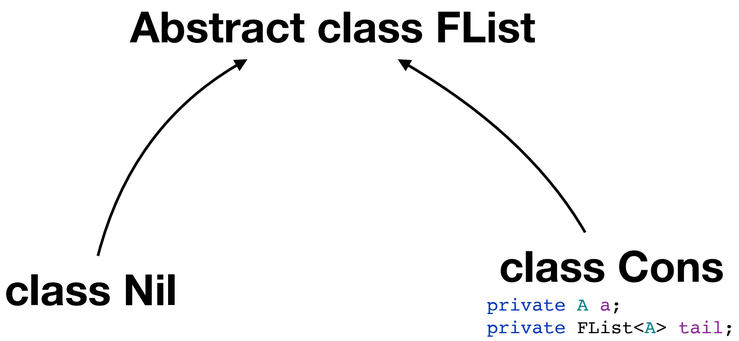
- Une FList est soit:
  - Une liste vide (Nil)
  - Un élément suivit d'une FList

- 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?

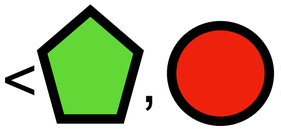
- $\Theta(n)$
- $\Theta(n^2)$
- $O(n)$
- $O(n^2)$

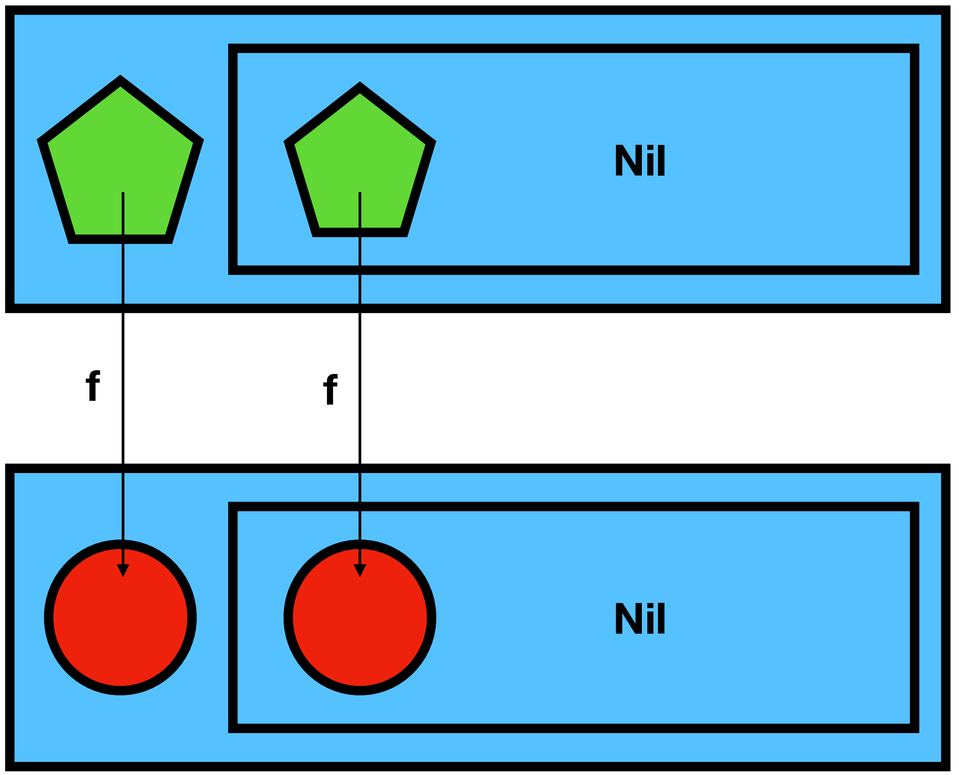
```java
/**
 * @param n the size of the list
 * @return The list [0,1,...,n-1]
 */
public static FList<Integer> rangeList(int n) {
    FList<Integer> list = FList.nil();
    for (int i = n-1; i >= 0; i--) {
        list = list.cons(i);
    }
    return list;
}
```

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

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

```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;
    }
});
```

LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

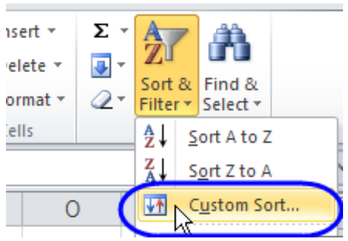
Intro 2

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 ?

Partie 2

- Reminder: Important de se procurer une copie du livre

Votre sentiment après une semaine?

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