Listes chaînées, Files et Piles

CHAPITRE 3 (WEISS)

CHAPITRE 15, BIG JAVA 4TH EDITION BY C. HORSTMANN

Listes chaînées

- Une liste chainée stocke chaque objet avec un lien qui y fait référence et possède également une référence vers le lien suivant de la liste
    - Avec Java, chaque élément d'une liste chainée possède en fait deux liens, chaque élément est aussi relié à l'élément précédent
- L'ajout et la suppression d'un élément au milieu d'une liste sont efficaces
- Visite séquentiel de chaque élément d'une liste – efficace
- Accès direct – pas efficace

Insertion d'un élément dans une liste chainée

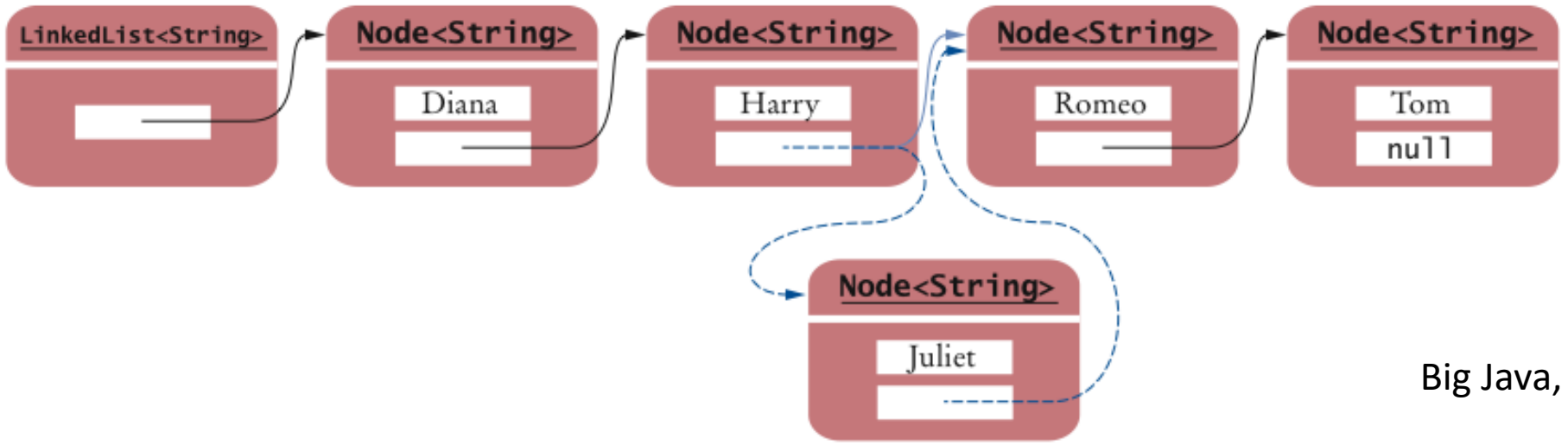


Figure 1 Inserting an Element into a Linked List

Big Java, C.Horstman

Classe `LinkedList` de Java

- Classe générique
  - *Spécifiez le type d'éléments entre les balises:* `LinkedList<Product>`
- Paquetage: `java.util`

Big Java, C.Horstman

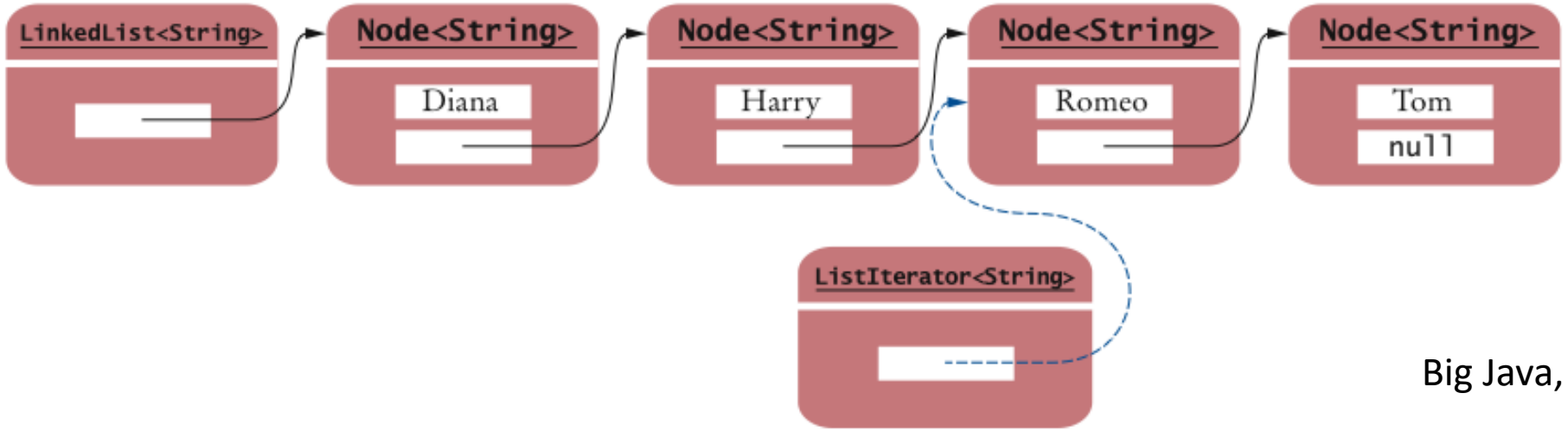
Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements (see Table 2 on page 634).

Iterateur de la liste

- Type `ListIterator`
- Vous pouvez vous servir d'un `ListIterator` pour parcourir les éléments d'une liste chainée dans n'importe quelle direction et pour ajouter ou supprimer des éléments
- Encapsule une position quelconque dans la liste
- Protège la liste lorsque l'on accède

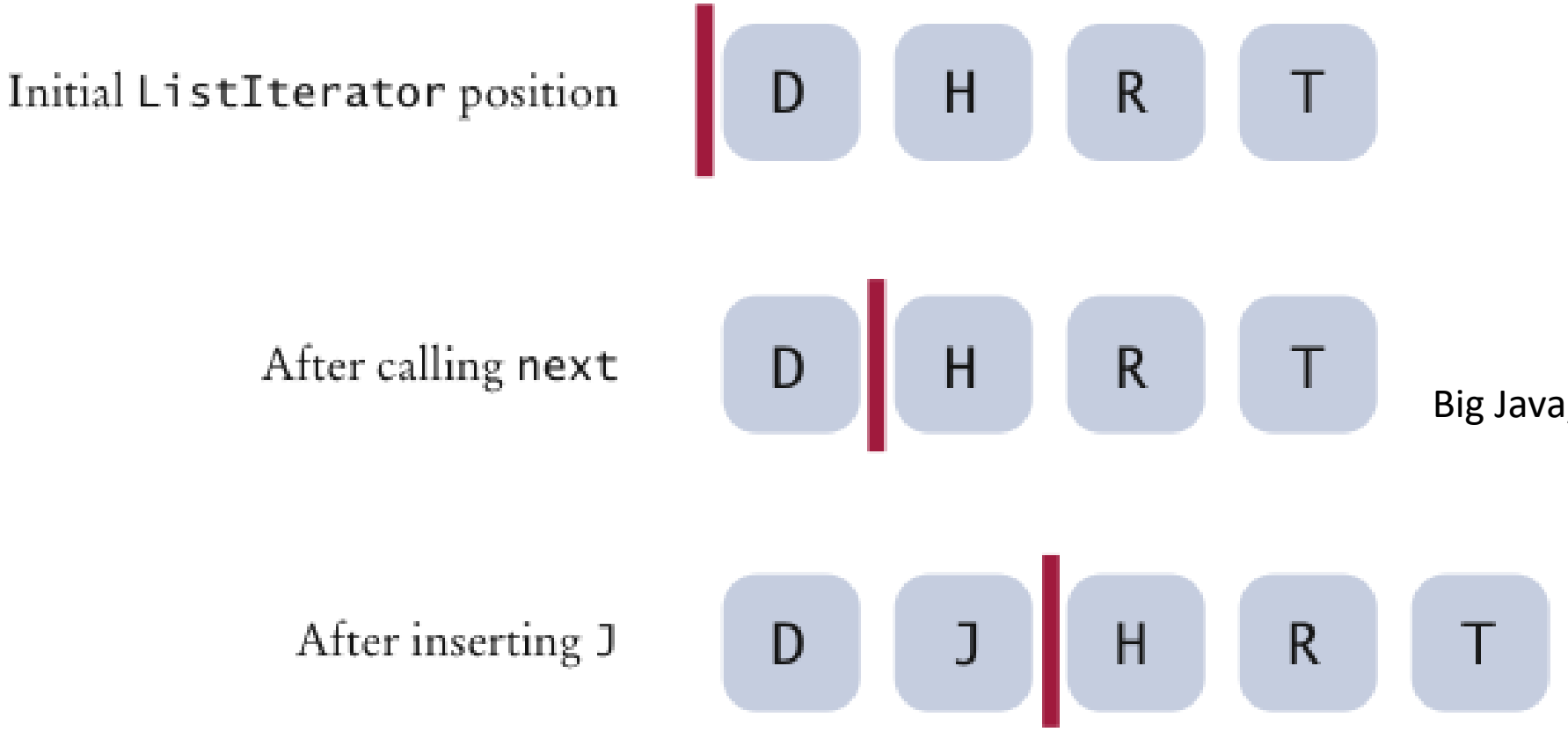
Itérateur d'une liste



Big Java, C.Horstman

Figure 2 A List Iterator

Vue conceptuelle de ListIterator



Big Java, C.Horstman

Figure 3 A Conceptual View of the List Iterator

Itérateur d'une liste

- Itérateur pointe vers la position entre deux éléments
  - *Analogie: Comme le pointeur entre les deux caractères dans le logiciel de traitement de texte*
- La méthode `listIterator` de la classe `LinkedList` reçoit un itérateur de la liste

```
LinkedList<String> employeeNames = ...;
ListIterator<String> iterator =
   employeeNames.listIterator();
```

Itérateur d'une liste

- Initialement, l'itérateur fait référence au début de la liste chaînée
- La méthode `next` déplace l'itérateur:

```
iterator.next();
```

- `next` lancera une exception `NoSuchElementException` si vous avez dépasse le dernier élément de la liste
- `hasNext` retourne vrai si l'élément suivant existe:

```
if (iterator.hasNext())
   iterator.next();
```

Itérateur d'une liste

- La méthode `next` retourne l'élément que l'itérateur vient de passer:

```
while iterator.hasNext()
{
    String name = iterator.next();
    Traitement de name
}
```

- Raccourci:

```
for (String name : employeeNames)
{
    Traitement de name
}
```

Itérateur d'une liste

- `LinkedList` est une liste doublement chaînée
  - *La classe stocke deux liens:*
    - *Une référence vers le lien suivant*
    - *Une référence vers le lien précédent*
- Pour parcours en arrière utilisez:
  - *`hasPrevious`*
  - *`previous`*

`LinkedList` : ajouter et supprimer un élément

- La méthode `add` :
  - *Ajoute le nouvel élément après la position de l'itérateur*
  - *Déplace la position de l'itérateur après le nouvel élément:*

```
iterator.add("Juliet");
```

`LinkedList` : ajouter et supprimer un élément

- La méthode `remove`
  - *Supprime et*
  - *Retourne l'objet retourné par le dernier appel de `next` ou `previous`*

```
//Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
    String name = iterator.next();
    if (name fulfills condition)
        iterator.remove();

}
```

`LinkedList` : ajouter et supprimer un élément

- Attention avec `remove`:
  - *Cette méthode pourra être appelée après un appel de `next` ou `previous`:*

```
iterator.next();
iterator.next();
iterator.remove();
iterator.remove();
// Error: You cannot call remove twice.
```

  - *Vous ne pouvez pas appeler `remove` immédiatement après `add`:*

```
iter.add("Fred");
iter.remove(); // Error: Can only call remove after
               // calling next or previous
```

  - *Si vous appelez `remove` incorrectement, l'exception `IllegalStateException`* sera lancée

Les méthodes de l'interface `ListIterator`

Table 2 Methods of the `ListIterator` Interface

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.
`if (iter.hasPrevious()) { s = iter.previous(); }`	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.
`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.
`iter.next(); iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

Big Java, C.Horstman

Programme

- `ListTester` est un programme qui
  - *Insère les chaines dans une liste*
  - *Itère à travers de la liste en insérant et supprimant les éléments*
  - *Imprime la liste*

ch15/uselist/ListTester.java

```
 1  import java.util.LinkedList;
 2  import java.util.ListIterator;
 3
 4  /**
 5     A program that tests the LinkedList class
 6  */
 7  public class ListTester
 8  {
 9     public static void main(String[] args)
10     {
11        LinkedList<String> staff = new LinkedList<String>();
12        staff.addLast("Diana");
13        staff.addLast("Harry");
14        staff.addLast("Romeo");
15        staff.addLast("Tom");
16
17        // | in the comments indicates the iterator position
18
19        ListIterator<String> iterator = staff.listIterator(); // |DHRT
20        iterator.next(); // D|HRT
21        iterator.next(); // DH|RT
22
```

Continued

ch15/uselist/ListTester.java (cont.)

```
23          // Add more elements after second element
24
25          iterator.add("Juliet"); // DHJ|RT
26          iterator.add("Nina"); // DHJN|RT
27
28          iterator.next(); // DHJNR|T
29
30          // Remove last traversed element
31
32          iterator.remove(); // DHJN|T
33
34          // Print all elements
35
36          for (String name : staff)
37             System.out.print(name + " ");
38          System.out.println();
39          System.out.println("Expected: Diana Harry Juliet Nina Tom");
40       }
41    }
```

Continued

ch15/uselist/ListTester.java (cont.)

Program Run:

```
Diana Harry Juliet Nina Tom
Expected: Diana Harry Juliet Nina Tom
```

Implémentation de liste chaînée

- La classe `LinkedList` en Java
- Considérons une implémentation simplifiée de cette classe
- Nous verrons comment les opérations de la liste manipulent les liens
- Implémentons une liste chaînée simple
  - *Classe supportera l'accès direct au premier élément (pas au dernier)*
- Notre liste ne utilisera pas un paramètre type
  - *Stocke les valeurs `Object` et insère un opérateur « cast » lors d'accès*

Implémentation de listes chaînées

- `Node`: Stocke un objet et une référence vers le nœud suivant
- Méthodes de la classe de la liste chaînée et la classe itérateur accèdent fréquemment les variables d'instance de `Node`
- Pour faciliter ces actions:
  - *Nous n'allons pas déclarer les variables d'instance private*
  - *Nous allons déclarer* `Node` *comme une classe interne privée de la classe* `LinkedList`
  - *On pourra laisser les variables public*
    - *Aucune méthode de la liste ne retournera un objet* `Node`

Implémentation de liste chaînée

```
public class LinkedList
{
    ...
    private class Node
    {
        public Object data;
        public Node next;
    }
}
```

Implémentation de liste chaînée

- Classe `LinkedList`
  - *Garde une référence vers le premier nœud* `first`
  - *Possède une méthode pour accéder le premier élément*

Implémentation de liste chaînée

```
public class LinkedList
{
   private Node first;
   ...
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
}
```

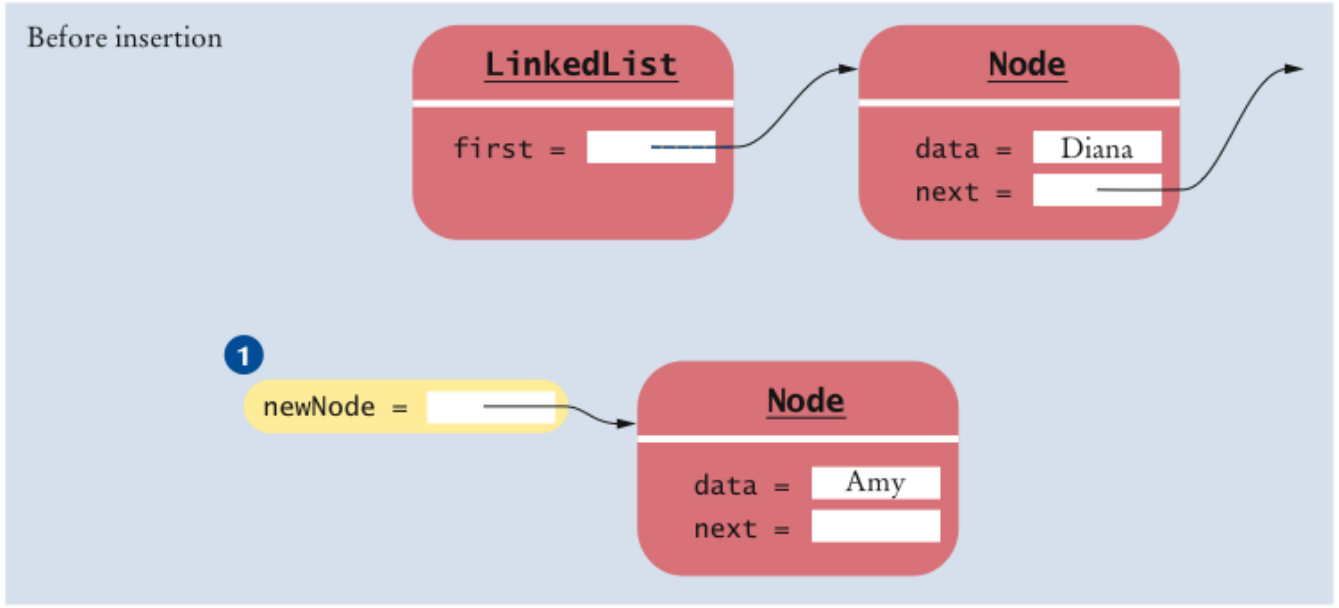
Ajouter un nouveau premier élément

- Lorsqu'on ajoute un nouveau nœud au début de la liste
  - *Il deviendra la nouvelle tête de la liste*
  - *L'ancienne tête de la liste deviendra le nœud suivant*

Ajouter un nouveau premier élément

```
public void addFirst(Object obj)
{
   Node newNode = new Node(); 1
   newNode.data = obj;
   newNode.next = first;
   first = newNode;
}
```

Big Java, C.Horstman



Ajouter un nouveau premier élément

```
public void addFirst(Object obj)
{
   Node newNode = new Node();
   newNode.data = obj;
   newNode.next = first; 2
   first = newNode; 3
}
```

Big Java, C.Horstman

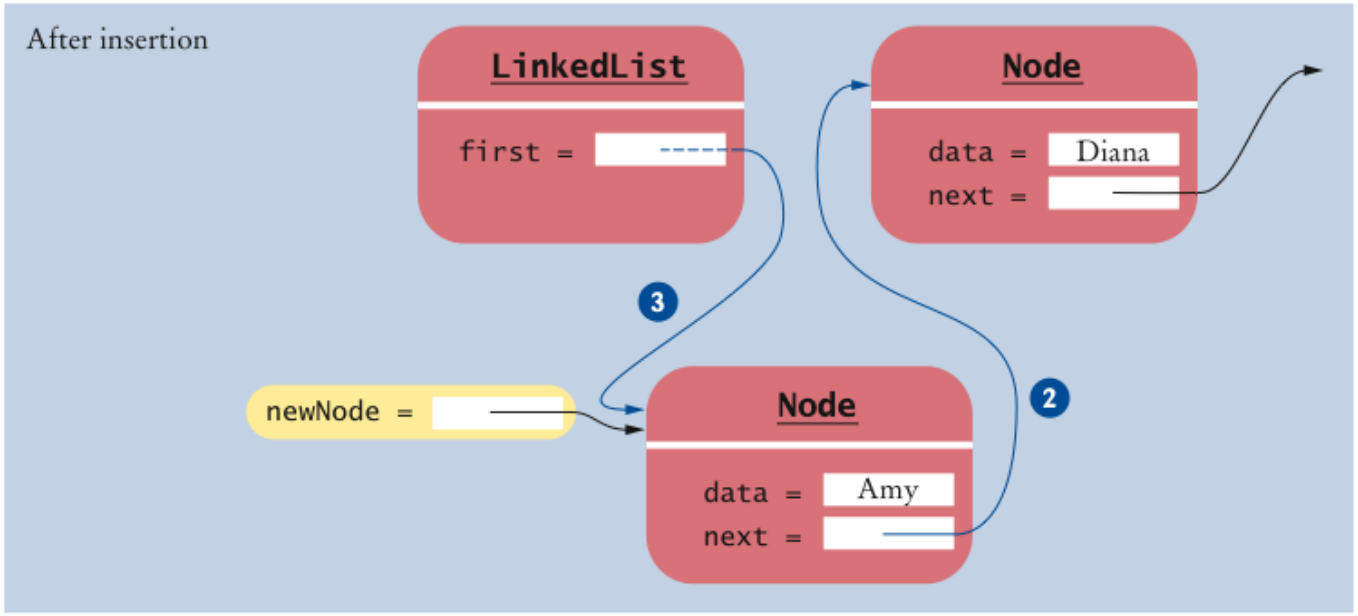


Figure 4 Adding a Node to the Head of a Linked List

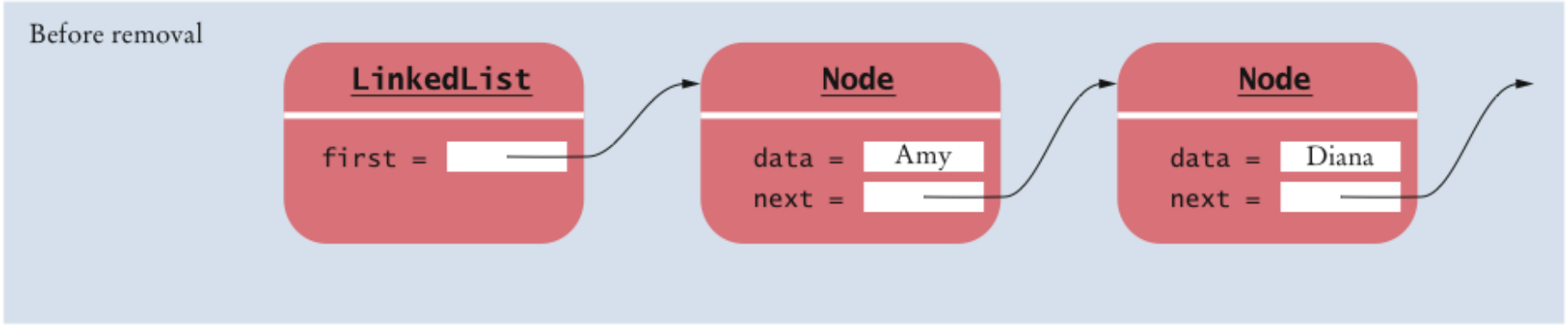
Retirer le premier élément

- Lorsque le premier élément est supprimé
  - *La donnée du premier nœud est sauvegardée et retournée comme un résultat de la méthode*
  - *Le successeur du premier nœud deviendra le premier nœud de la liste modifiée*
  - *L'ancien nœud sera retourné au système par ramasse miette lorsqu'il ne restera plus de références vers lui*

Retirer le premier élément

```
public Object removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   Object obj = first.data;
   first = first.next;
   return obj;
}
```

Big Java, C.Horstman



Retirer le premier élément

```
public Object removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   Object obj = first.data;
   first = first.next; 1
   return obj;
}
```

Big Java, C.Horstman

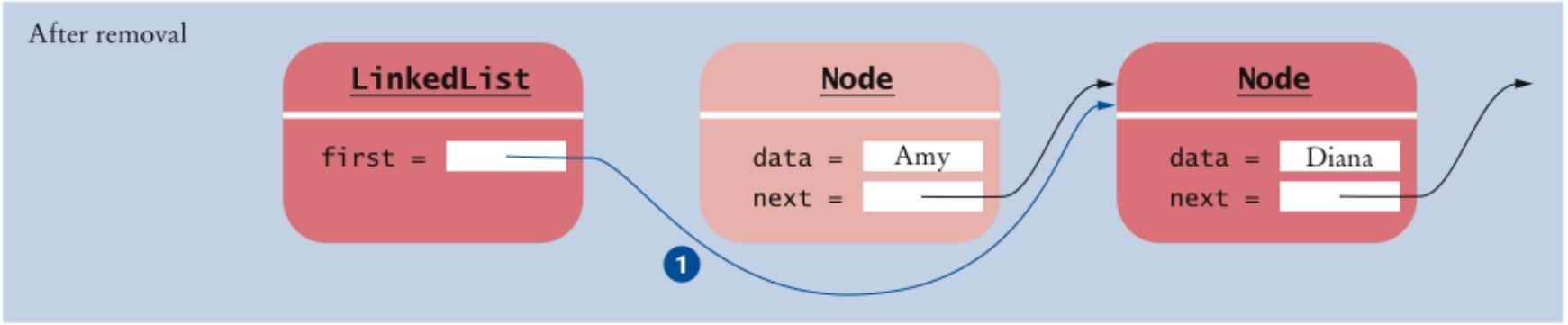


Figure 5 Removing the First Node from a Linked List

Itérateur de la liste chaînée

- On définie `LinkedListIterator`: une classe interne privée de la `LinkedList`
- Implémente l'interface simplifiée `ListIterator`
- a un accès au champ `first` et la classe private `Node`
- Clients de `LinkedList` ne savent pas le nom de la classe itérateur
  - *Ils savent que cette classe implémente l'interface `ListIterator`*

LinkedListIterator

- La classe `LinkListIterator`:

```
public class LinkedList
{
   ...
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }

   private class LinkedListIterator implements
                                        ListIterator
   {
      private Node position;
      private Node previous;

       ...
```

Continued

`LinkedListIterator` (cont.)

```
        public LinkedListIterator()
        {
           position = null;
           previous = null;
        }
    }
    ...
}
```

Méthode `next` de l'itérateur de la liste chaînée

- `position`: Référence vers le dernier nœud visité
- Aussi, on stocke la référence vers le nœud visité avant le dernier nœud
- Méthode `next`: la référence `position` est avancée vers `position.next`
- L'ancienne position est sauvegardée dans `previous`
- Si l'itérateur pointe vers la position avant le premier élément, donc l'ancienne `position` est `null` et `position` doit être `first`

Méthode `next` de l'itérateur de la liste chaînée

```
public Object next()
{
   if (!hasNext())
      throw new NoSuchElementException();
   previous = position; // Remember for remove
   if (position == null)
      position = first;
   else
      position = position.next;
   return position.data;
}
```

Méthode `hasNext` de l'itérateur de la liste chaînée

- La méthode `next` doit être appelée seulement à condition que l'itérateur n'est pas à la fin de la liste
- L'itérateur est à la fin
  - *Si la liste est vide (`first == null`)*
  - *S'il n'y a pas d'éléments après la position courante (`position.next == null`)*

Méthode `hasNext` de l'itérateur de la liste chaînée

```
public boolean hasNext()
{
   if (position == null)
      return first != null;
   else
      return position.next != null;
}
```

Méthode `remove` de l'itérateur de la liste chaînée

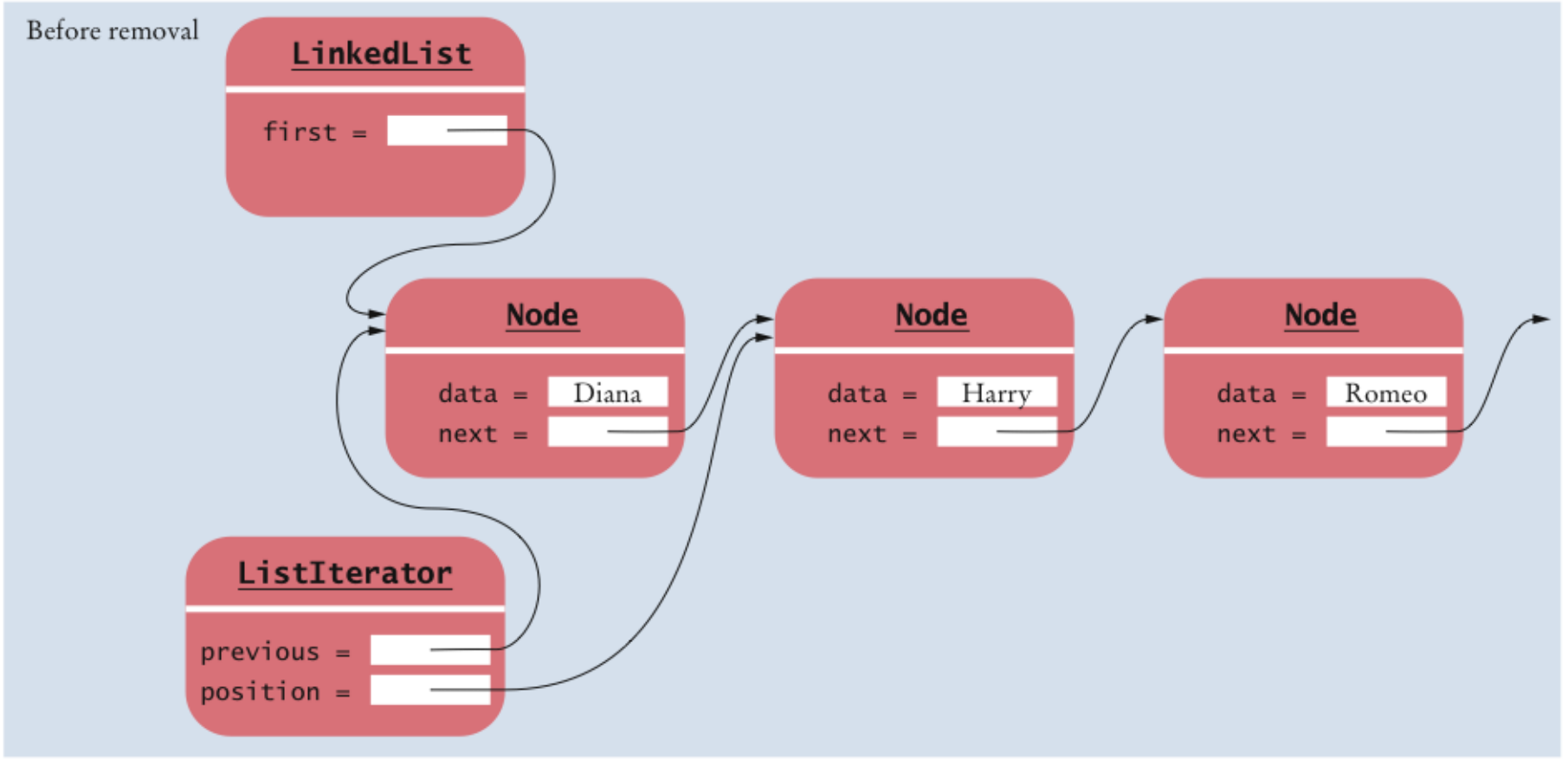
- Si un élément à supprimer est le premier élément, appelez `removeFirst`
- Si non, le nœud précédant celui à supprimer a besoin de mettre à jour sa référence `next` pour sauter l'élément à supprimer
- Si la référence `previous` est égale à `position`:
  - *Cette appel ne suis pas immédiatement l'appel* `next`
  - *Lancez une exception* `IllegalArgumentException`
- C'est illégal appeler `remove` deux fois consécutives
  - `remove` *met* `previous` *à* `position`

Méthode `remove` de l'itérateur de la liste chaînée

```
public void remove()
{
   if (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next;
   }
   position = previous;
}
```

Continued

Méthode `remove` de l'itérateur de la liste chaînée



Big Java, C.Horstman

Continued

The Linked List Iterator's `remove` Method (cont.)

```
public void remove()
{
   If (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next; (1)
   }
   position = previous; (2)
}
```

Continued

Méthode `remove` de l'itérateur de la liste chaînée

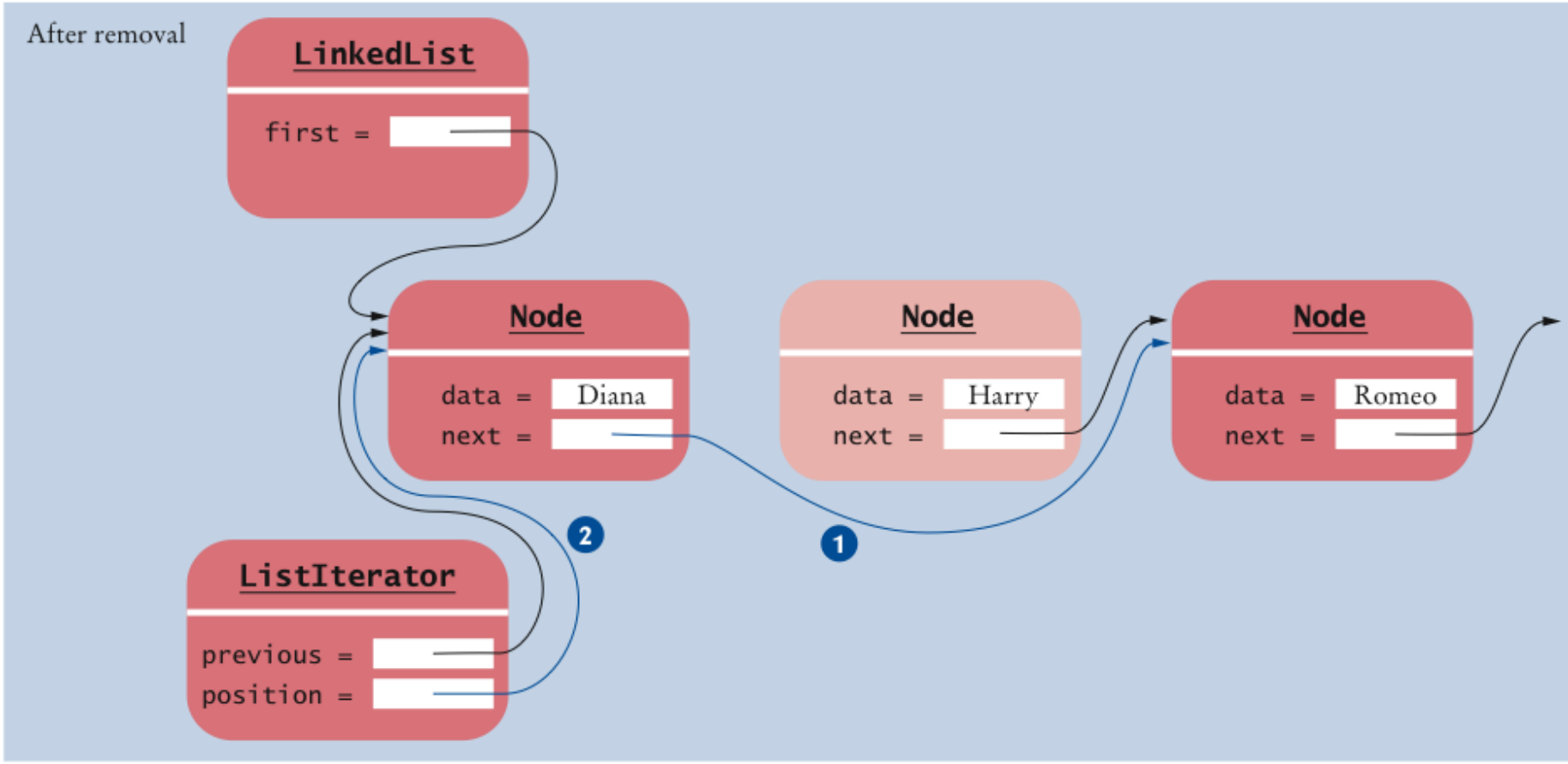


Figure 6 Removing a Node from the Middle of a Linked List

Big Java, C.Horstman

Méthode `set` de l'itérateur de la liste chaînée

- Change la donnée stockée dans l'élément visité récemment
- La méthode `set`

```
public void set(Object obj)
{
   if (position == null)
      throw new NoSuchElementException();
    position.data = obj;
}
```

Méthode `add` de l'itérateur de la liste chaînée

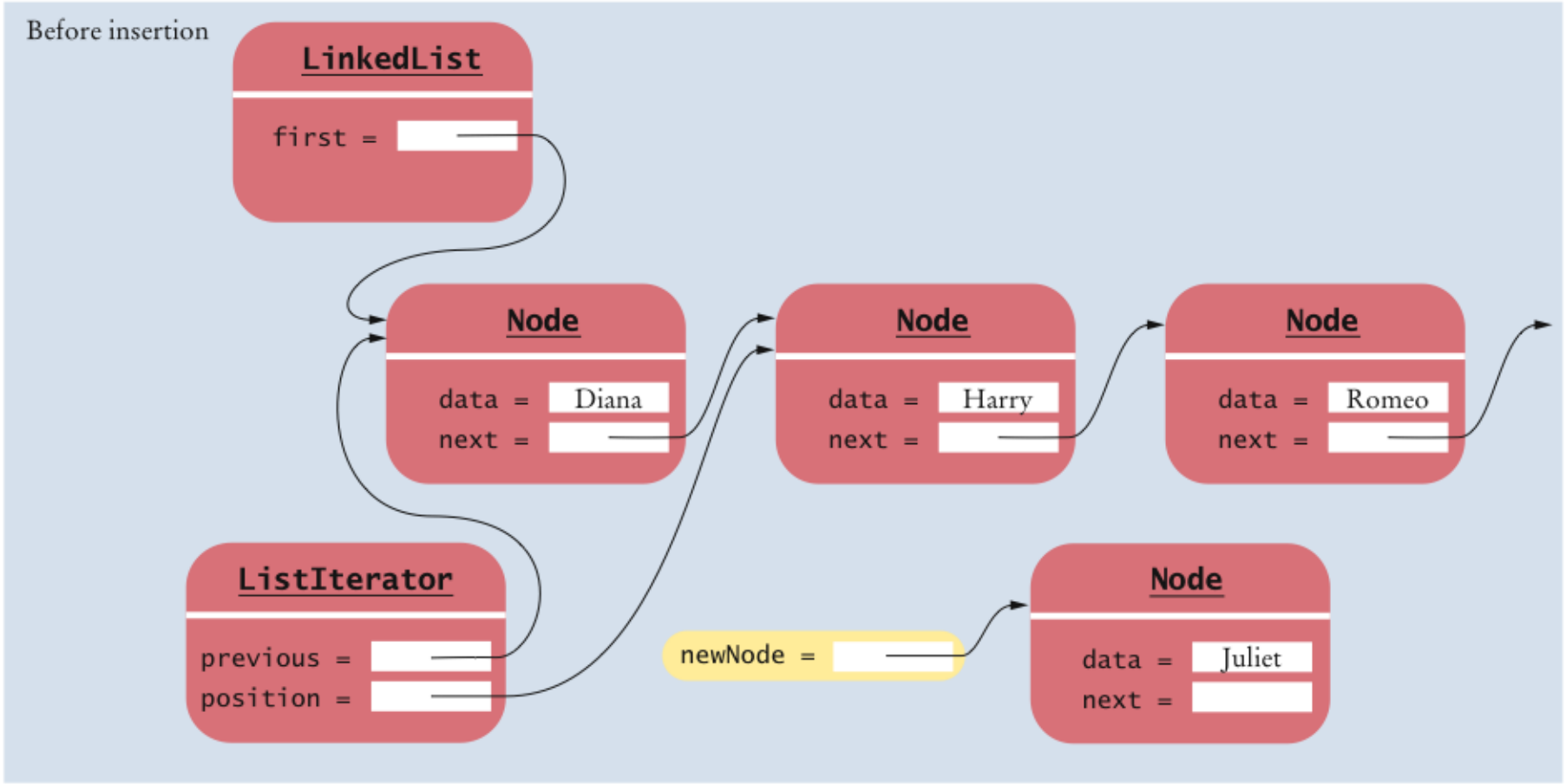
- L'opération la plus complexe
- `add` insert un nouveau nœud après la position courante
- Établit un successeur du nouveau nœud - le successeur du nœud de la position courante

Méthode `add` de l'itérateur de la liste chaînée

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next;
      position.next = newNode;
      position = newNode;
   }
   previous = position;
}
```

Continued

Méthode `add` de l'itérateur de la liste chaînée



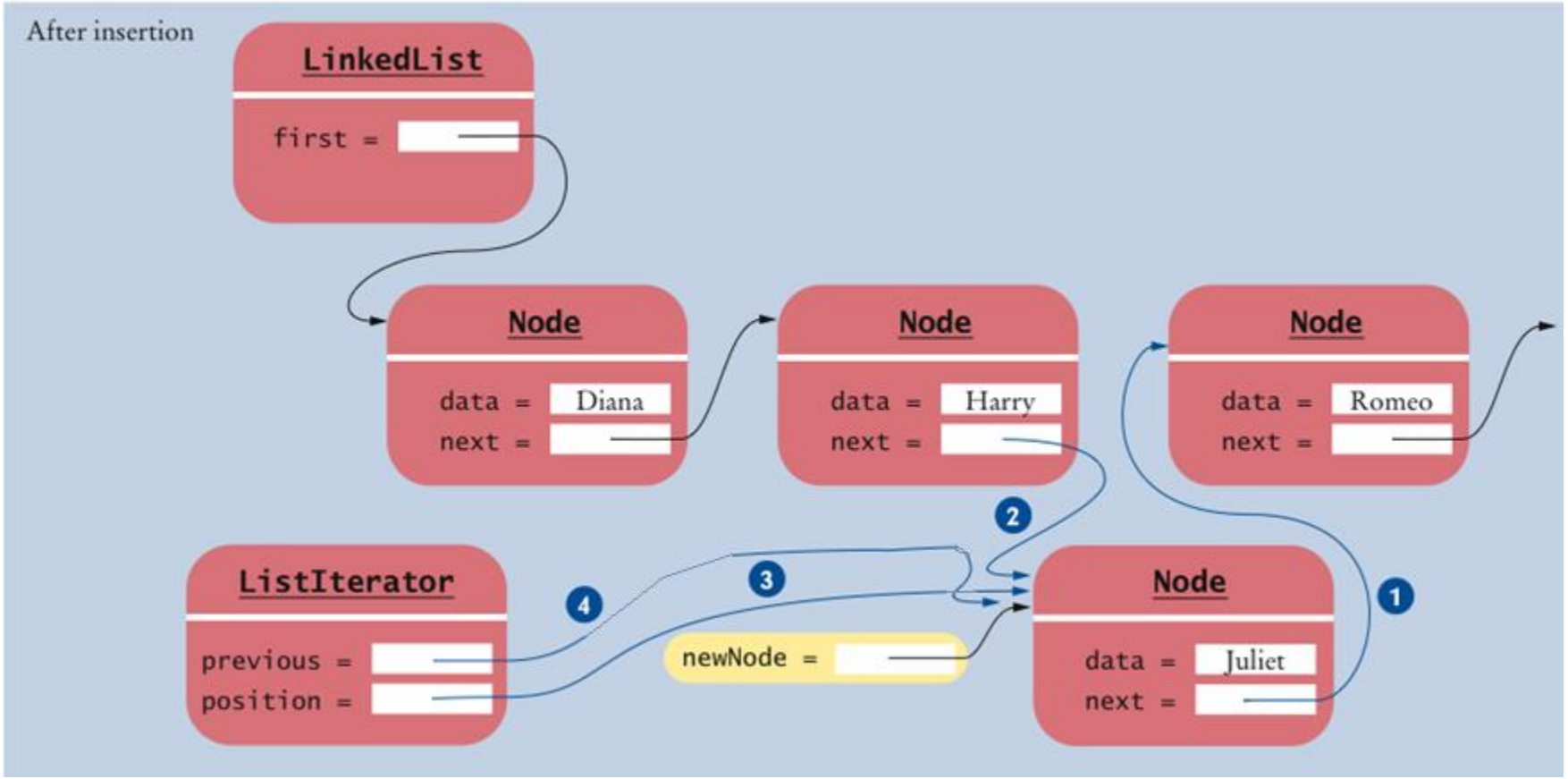
Continued

Méthode `add` de l'itérateur de la liste chaînée

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next;  (1)
      position.next = newNode;  (2)
      position = newNode;  (3)
   }
   previous = position;  (4)
}
```

Continued

Méthode `add` de l'itérateur de la liste chaînée



Big Java, C.Horstman

ch15/impllist/LinkedList.java

```
1  import java.util.NoSuchElementException;
2
3  /**
4     A linked list is a sequence of nodes with efficient
5     element insertion and removal. This class
6     contains a subset of the methods of the standard
7     java.util.LinkedList class.
8  */
9  public class LinkedList
10 {
11    private Node first;
12
13    /**
14       Constructs an empty linked list.
15    */
16    public LinkedList()
17    {
18       first = null;
19    }
20
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
21      /**
22         Returns the first element in the linked list.
23         @return the first element in the linked list
24      */
25      public Object getFirst()
26      {
27         if (first == null)
28            throw new NoSuchElementException();
29         return first.data;
30      }
31
32      /**
33         Removes the first element in the linked list.
34         @return the removed element
35      */
36      public Object removeFirst()
37      {
38         if (first == null)
39            throw new NoSuchElementException();
40         Object element = first.data;
41         first = first.next;
42         return element;
43      }
44
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
45      /**
46         Adds an element to the front of the linked list.
47         @param element the element to add
48      */
49      public void addFirst(Object element)
50      {
51         Node newNode = new Node();
52         newNode.data = element;
53         newNode.next = first;
54         first = newNode;
55      }
56
57      /**
58         Returns an iterator for iterating through this list.
59         @return an iterator for iterating through this list
60      */
61      public ListIterator listIterator()
62      {
63         return new LinkedListIterator();
64      }
65
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
66      class Node
67      {
68         public Object data;
69         public Node next;
70      }
71
72      class LinkedListIterator implements ListIterator
73      {
74         private Node position;
75         private Node previous;
76
77         /**
78            Constructs an iterator that points to the front
79            of the linked list.
80         */
81         public LinkedListIterator()
82         {
83            position = null;
84            previous = null;
85         }
86
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
87          /**
88             Moves the iterator past the next element.
89             @return the traversed element
90          */
91          public Object next()
92          {
93             if (!hasNext())
94                throw new NoSuchElementException();
95             previous = position; // Remember for remove
96
97             if (position == null)
98                position = first;
99             else
100               position = position.next;
101
102            return position.data;
103         }
104
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
105          /**
106             Tests if there is an element after the iterator position.
107             @return true if there is an element after the iterator position
108          */
109          public boolean hasNext()
110          {
111             if (position == null)
112                return first != null;
113             else
114                return position.next != null;
115          }
116
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
117          /**
118             Adds an element before the iterator position
119             and moves the iterator past the inserted element.
120             @param element the element to add
121          */
122          public void add(Object element)
123          {
124             if (position == null)
125             {
126                addFirst(element);
127                position = first;
128             }
129             else
130             {
131                Node newNode = new Node();
132                newNode.data = element;
133                newNode.next = position.next;
134                position.next = newNode;
135                position = newNode;
136             }
137             previous = position;
138          }
139
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
140          /**
141             Removes the last traversed element. This method may
142             only be called after a call to the next() method.
143          */
144          public void remove()
145          {
146             if (previous == position)
147                throw new IllegalStateException();
148
149             if (position == first)
150             {
151                removeFirst();
152             }
153             else
154             {
155                previous.next = position.next;
156             }
157             position = previous;
158          }
159
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
160          /**
161             Sets the last traversed element to a different value.
162             @param element the element to set
163          */
164          public void set(Object element)
165          {
166             if (position == null)
167                throw new NoSuchElementException();
168             position.data = element;
169          }
170       }
171    }
```

ch15/impllist/ListIterator.java

```
1  /**
2     A list iterator allows access of a position in a linked list.
3     This interface contains a subset of the methods of the
4     standard java.util.ListIterator interface. The methods for
5     backward traversal are not included.
6  */
7  public interface ListIterator
8  {
9     /**
10       Moves the iterator past the next element.
11       @return the traversed element
12    */
13    Object next();
14
15    /**
16       Tests if there is an element after the iterator position.
17       @return true if there is an element after the iterator position
18    */
19    boolean hasNext();
20
```

Continued

ch15/impllist/ListIterator.java (cont.)

```
21      /**
22         Adds an element before the iterator position
23         and moves the iterator past the inserted element.
24         @param element the element to add
25      */
26      void add(Object element);
27
28      /**
29         Removes the last traversed element. This method may
30         only be called after a call to the next() method.
31      */
32      void remove();
33
34      /**
35         Sets the last traversed element to a different value.
36         @param element the element to set
37      */
38      void set(Object element);
39   }
```

Types abstrait de données

- Il existe deux façons de voir une liste chaînée
  - *Implémentation concrète*
    - Séquence des nœuds représentants des objets avec les liens entre eaux
  - *Concept abstract d'une liste chaînée*
    - Séquence ordonnée des items qui pourra être traversée avec un itérateur

Types abstrait de données

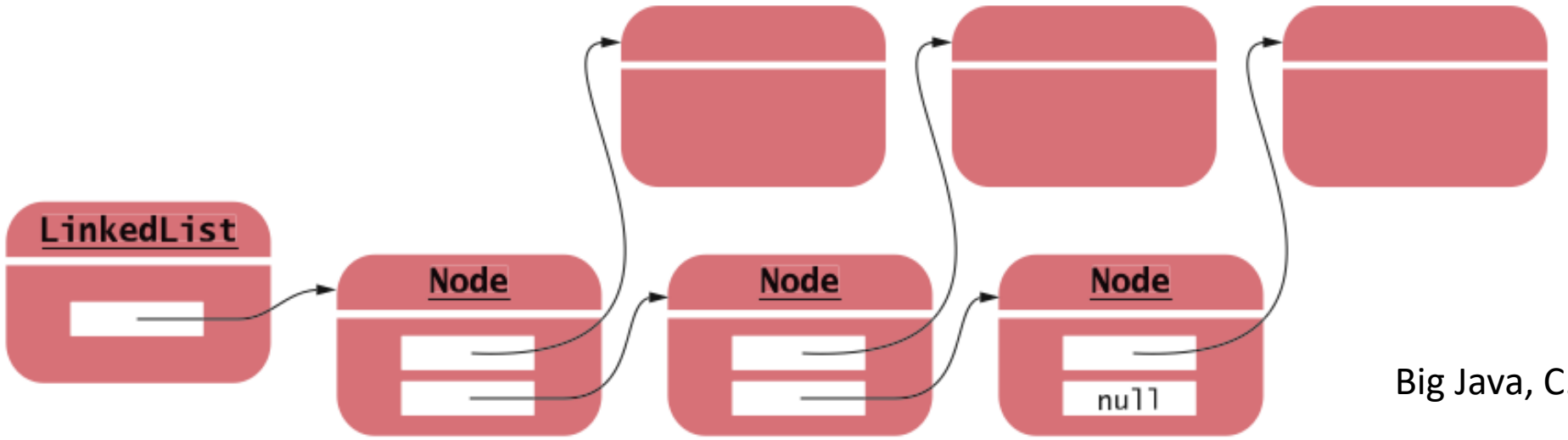


Figure 8 A Concrete View of a Linked List

Big Java, C.Horstman

Figure 9 An Abstract View of a List

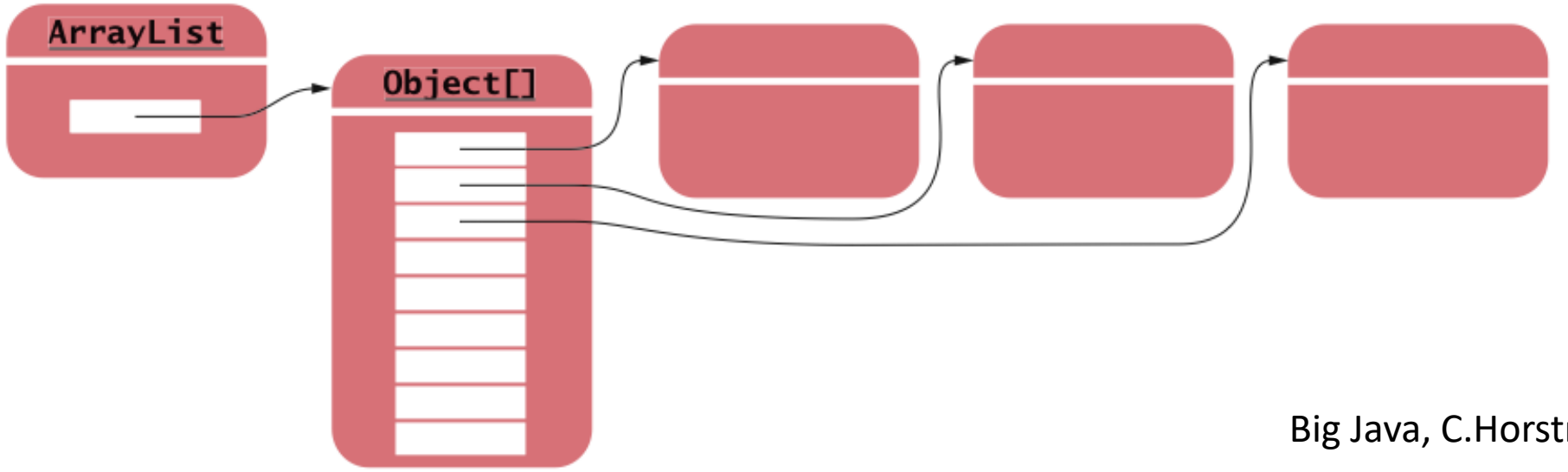
Types abstrait de données

- Définissent les opérations fondamentales sur les données
- Ne spécifient pas une implémentation

Type Tableau abstrait et concret

- Deux façon de voir un tableau dynamique (ArrayList)
- Implémentation concrète: Un tableau de références partiellement rempli
- Nous ne pensons pas à l'implémentation lorsqu'on utilise tableau liste
  - *Utilisons le point de vue abstrait*
- Vue abstraite : Séquence ordonnée de données où chaque donnée pourra être accéder par un index

Types de données abstraits et concrets



Big Java, C.Horstman

Figure 10 A Concrete View of an Array List

Figure 11 An Abstract View of an Array

Types de données abstraits et concrets

- Les implémentations concrètes d'une liste chaînée et d'un tableau dynamique (ArrayList) sont assez différentes
- Les abstractions paraissent similaires à la première vue
- Pour voir la différence, considérez les interfaces publiques (min)

Opérations fondamentales de Tableau Liste

Un tableau permet un accès direct à tous les éléments:

```
public class ArrayList
{
   public Object get(int index) {...}
   public void set(int index, Object value) {...)
   ...
}
```

Opérations fondamentales de la Liste chaînée

La liste chaînée permet un accès séquentiel à ses éléments:

```
public class LinkedList
{
   public ListIterator listIterator() {...}
   ...
}

public interface ListIterator
{
   Object next();
   boolean hasNext();
   void add(Object value);
   void remove();
   void set(Object value);
   ...
}
```

Types de données abstraits

- `ArrayList`: Combine les interfaces de array et list
- Les deux `ArrayList` et `LinkedList` implémentent une interface appelée `List`
  - `List` *définie les opérations pour l'accès direct et séquentiel*
- Terminologie n'est pas utilisée en dehors de la bibliothèque Java
- Terminologie plus traditionnelle: *array* et *list*
- Bibliothèque Java fournit les implémentations concrètes de `ArrayList` et de `LinkedList` pour ces types de données abstraits
- Tableaux de Java est une autre implémentation de type abstrait

Array

Les performances des opérations de Arrays et Lists

- Listes
  - Ajouter et supprimer un élément
    - *Un nombre fixe des références doivent être modifiées pour ajouter ou supprimer un nœud - O(1)*
- Array
  - Ajouter et supprimer un élément
    - *En moyenne, n/2 éléments doivent être déplacés - O(n)*

Les performances des opérations de Arrays et Lists

Operation	**Array**	**List**
Random access	*O*(1)	*O*(*n*)
Linear traversal step	*O*(1)	*O*(1)
Add/remove an element	*O*(*n*)	*O*(1)

Types Abstrait de Données

- Liste Abstraite
  - *Séquence ordonnée des items qui peuvent être traversés séquentiellement*
- Tableau Abstrait
  - *Séquence ordonnée des items avec l'accès direct par biais d'index entier*

Piles et Queues

- Pile: Collection des items avec “last in, first out” politique de retrait
- Queue: Collection des items avec “first in, first out” politique de retrait

Pile

- Permet une insertion et un retrait des éléments seulement d'un seul bout
  - *Traditionnellement appelé sommet de la pile*
- Nouveaux items sont ajoutés au sommet de la pile
- Items sont retirés du sommet de la pile
- Ordre d'accès: dernier entré, premier sorti ou LIFO
- Traditionnellement, les opérations d'addition et de retrait sont appelées `push(empiler)` et `pop (dépiler)`
- Pensez de la pile de livres

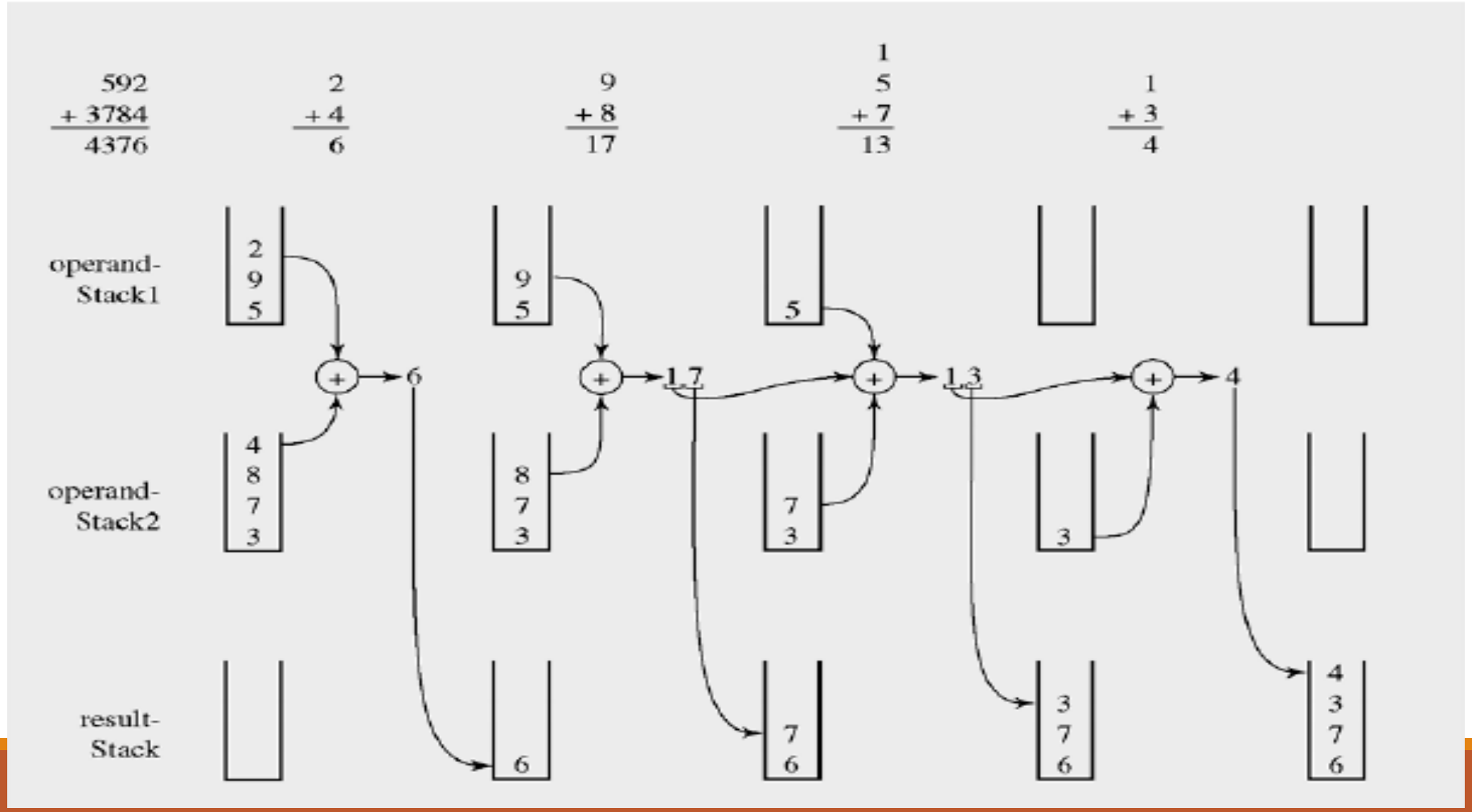
Pile

Figure 12
A Stack of Books

Exemples d'applications

- Programme récursif converti en programme non récursif
- Évaluation d'opérations arithmétiques
- Boutons précédent et suivant dans les navigateurs Internet
- Additinner de très grands nombres
- Interpréteurs de langage (pile d'appel de fonction (contexte))
- Langage Postscript dans les imprimantes.
- Etc.

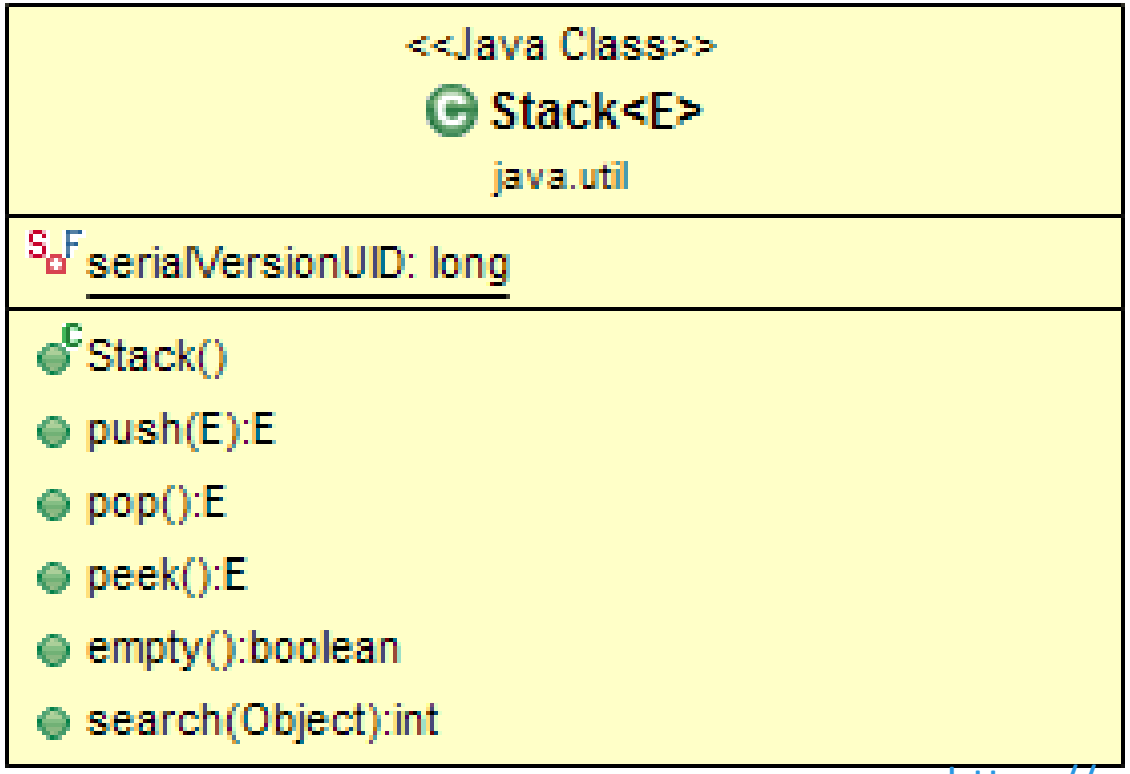
Additionner de très grands nombres



Interface **abstraite** d'une pile

Nom de l'opération	Nom (anglais)	Action
empiler(e)	push(e)	Place e au sommet de la pile
depiler()	pop()	Enlève l'élément au sommet de la pile
haut()	top()	Retourne le sommet de la pile
taille()	size()	Retourne le nombre d'éléments dans la pile
estVide()	empty()	Retourne vrai si la pile est vide, sinon faux

java.util.Stack



https://www.javaguides.net/2018/12/java-stack-class-example.html

Implémentations d’une pile

- Tableau
- Liste chaînée
- Dans le TAD pile, les opérations dépiler() et haut() ne peuvent être exécutées si la pile est vide

Implémentations d'une pile: Tableau

- La pile consiste
  - Tableau S de N-éléments
  - Variable entière t l'index du «premier» élément dans le tableau S
    - top de la pile

```
Algorithme taille():
        returner t +1
Algorithme estVide():
        returner (t < 0)
Algorithme haut():
        si estVide()
                ERREUR
        sinon returner S[t]
```

Implémentations d'une pile: Tableau

```
Algorithme empiler(obj):
        si taille = N alors
            ERREUR
        t ← t + 1
        S[t] ← obj
```

```
Algorithme dépiler():
        si estVide() alors
            ERREUR
        e ← S[t]
        S[t] ← null
        t ← t-1
        return e
```

Implémentations d'une pile: Tableau Performance et Limitations

- Performance

taille()	O(1)
estVide()	O(1)
haut()	O(1)
empiler(obj)	O(1)
dépiler()	O(1)

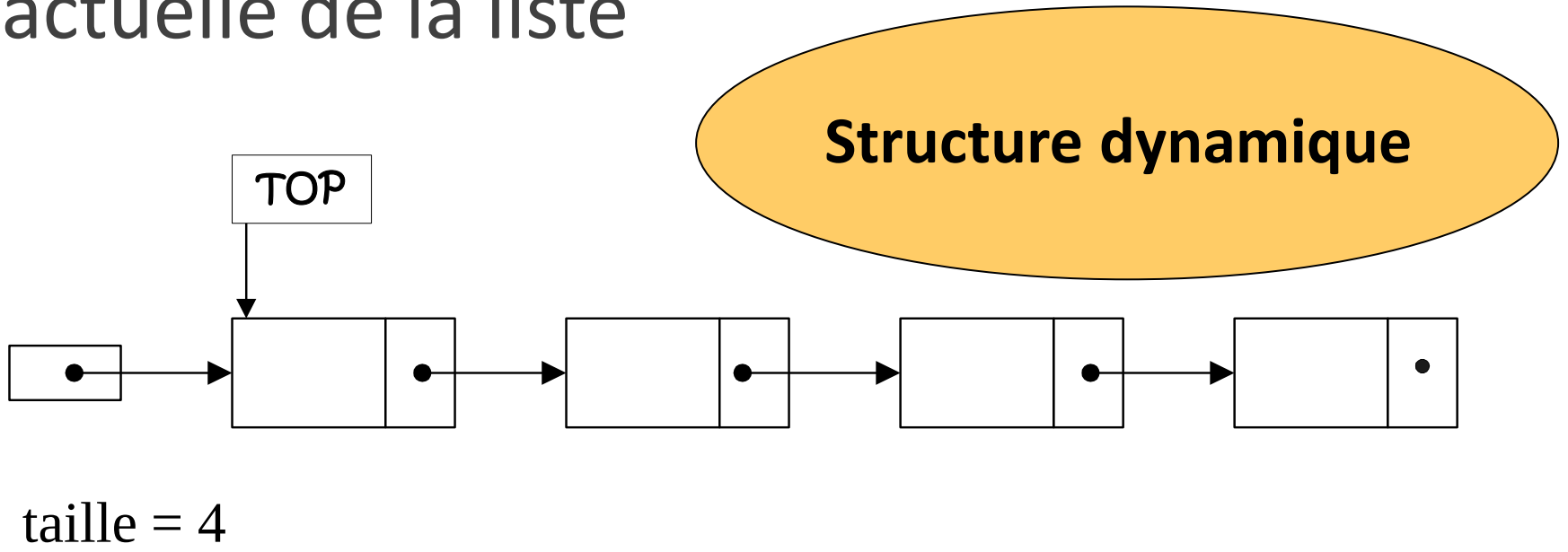
Structure statique

- Limitations

Espace: O(n), n = taille de le tableau

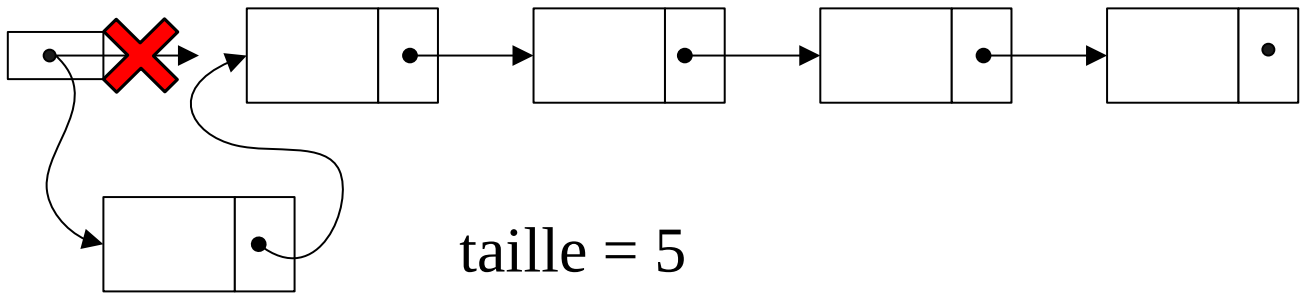
Implémentations d’une pile: Liste chaînée

- Liste simplement chaînée avec un variable contenir la taille actuelle de la liste

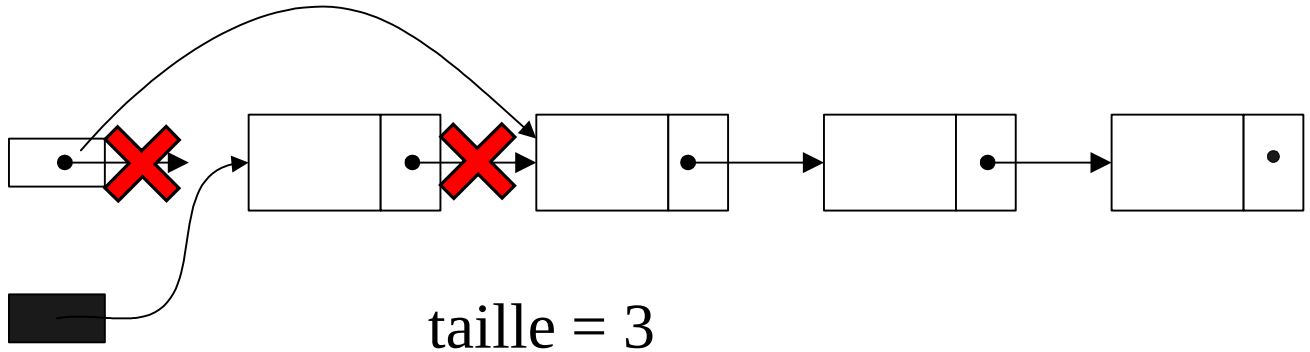


Implémentations d'une pile: Liste chaînée

- **EMPILER:** Ajouter au début



DÉPILER: Prendre le premier



Implémentations d'une pile: Liste chaînée Performance et Limitations

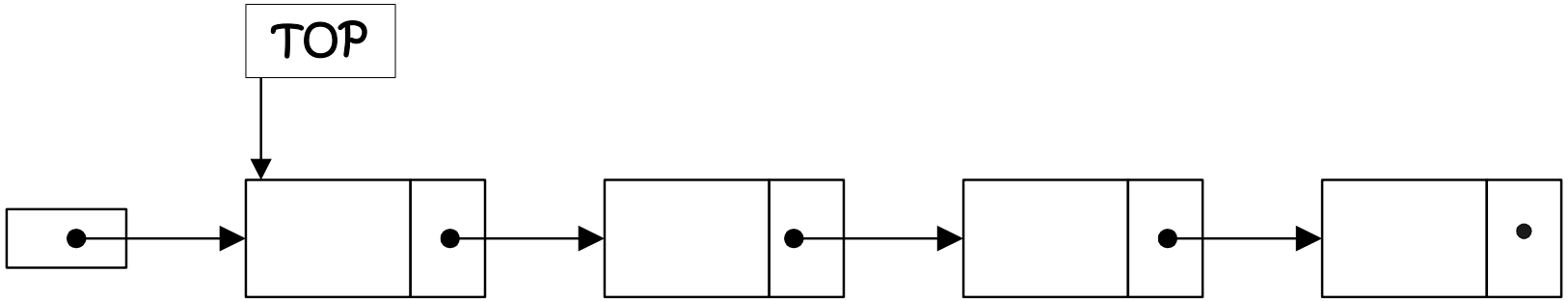
- Performance

taille()	O(1)
estVide()	O(1)
haut()	O(1)
empiler(obj)	O(1)
dépiler()	O(1)

Structure dynamique

- Limitations: ?

Espace: Variable



File (Queue)

- Une file permet d'ajouter efficacement des éléments à la fin et d'en supprimer au début
  - ordre FIFO - "premier arrivé, premier sorti (ou servi!)" first-in-first-out (FIFO)
- Il est impossible d'ajouter des éléments au milieu
- Pensez de la file d'attente des gens

Queue

Figure 13 A Queue

Interface **abstraite** d'une file

Nom de l'opération	Nom (anglais)	Action
enfiler(e)	enqueue(e)	Ajoute e à la queue de la file
defiler()	dequeue()	Enlève l'élément à la tête de la file.
tete()	front()	Retourne l'élément à la tête
taille()	size()	Retourne le nombre d'éléments dans la file
estVide()	empty()	Retourne vrai si la file est vide, sinon faux

Exemple: Palindromes

“non”

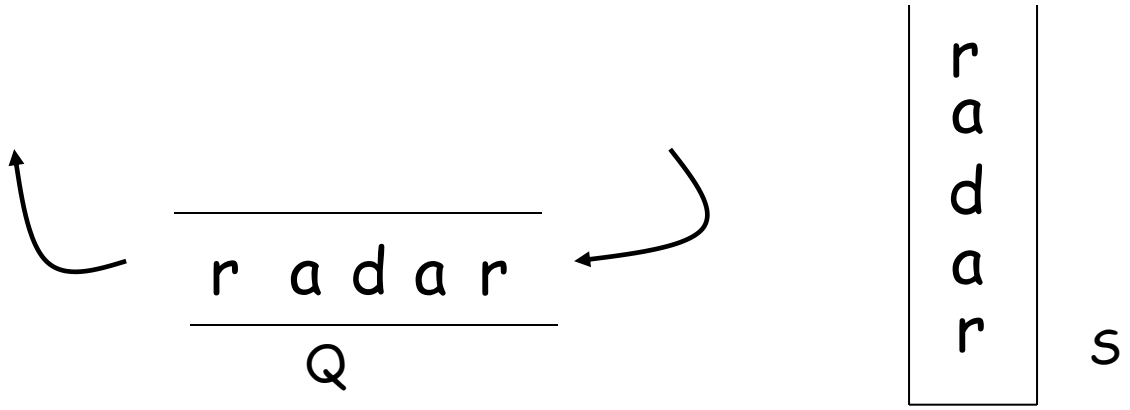
“Rions noir”

“radar”

“Engage le jeu que je le gagne”

Lire la ligne dans une pile et dans une file

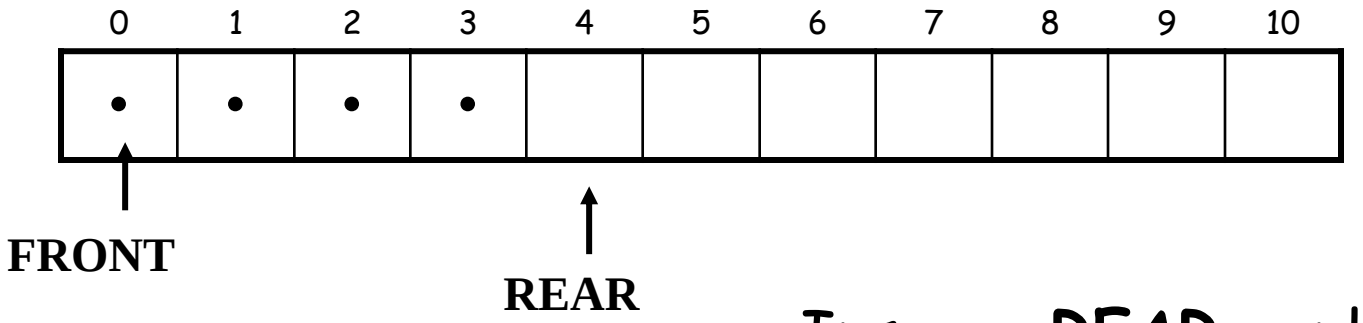
Comparer les résultants de la file et la pile



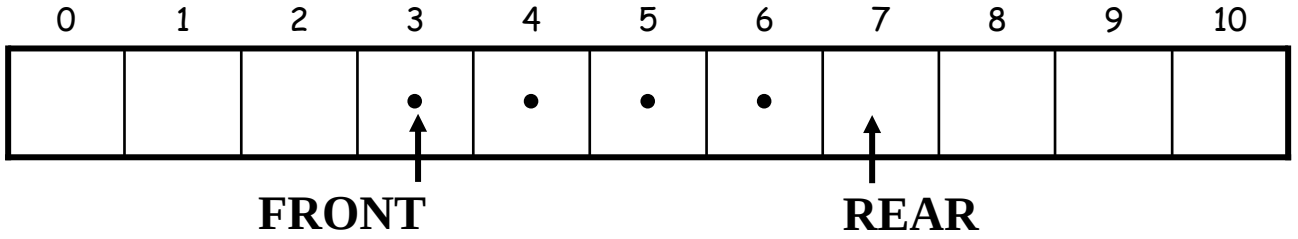
Implémentations d’une file

- Tableau
- Liste chaînée

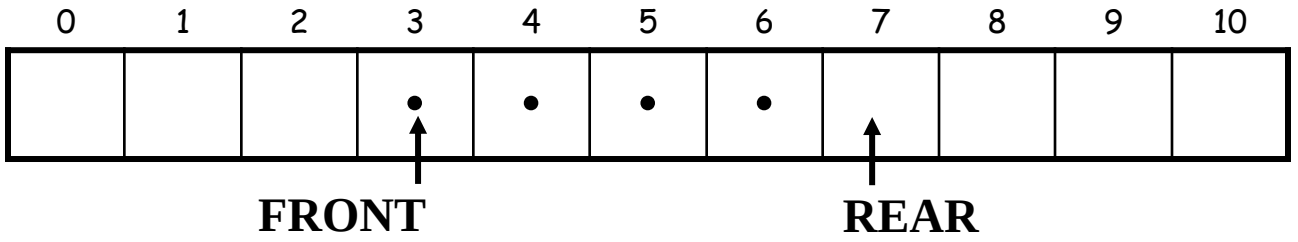
Implementation d'une file avec tableau circulaire



Inserer **REAR** enlever de: **FRONT**

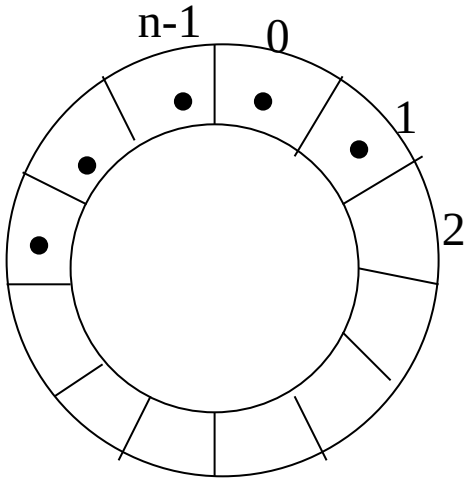


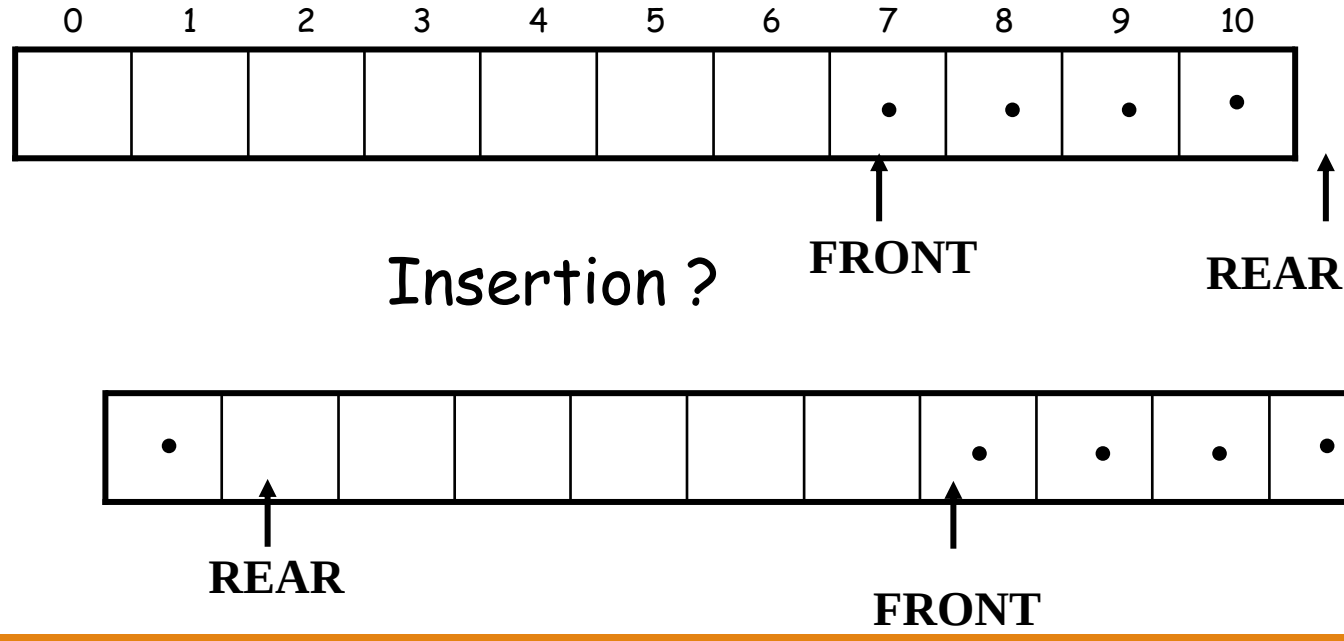
Implementation d'une file avec tableau circulaire



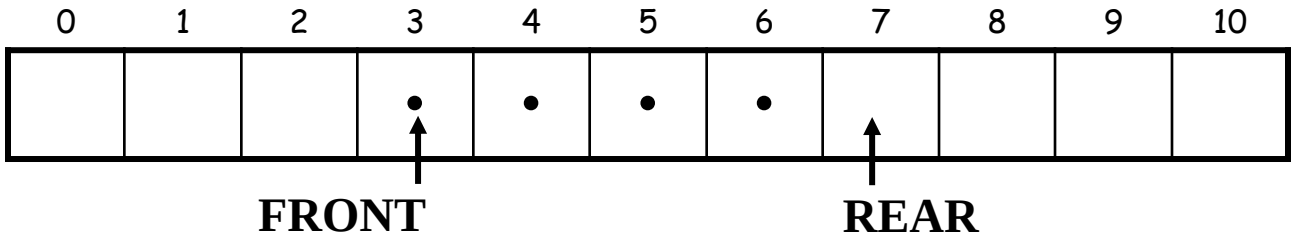
Configuration circulaire ("wrapped around")

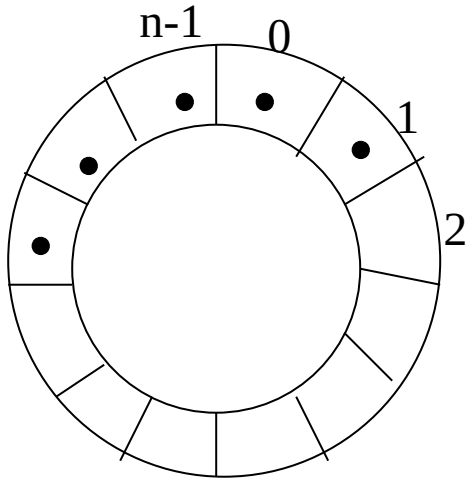
Une taille fixée au début

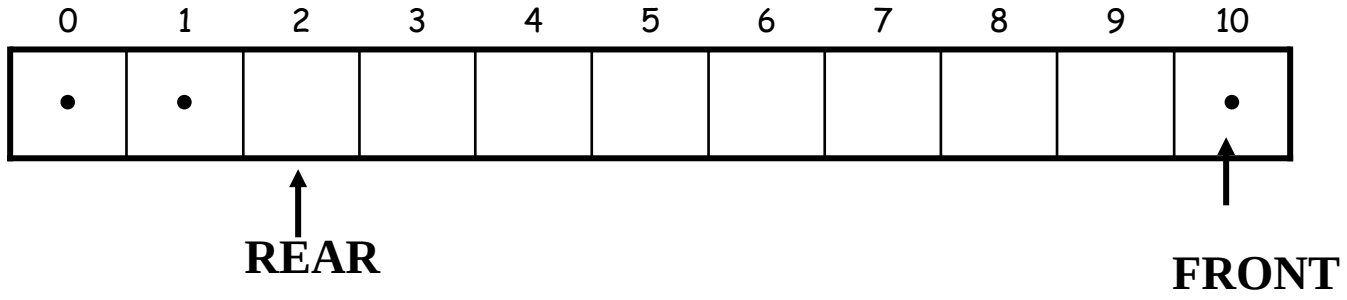




Implementation d'une file avec tableau circulaire



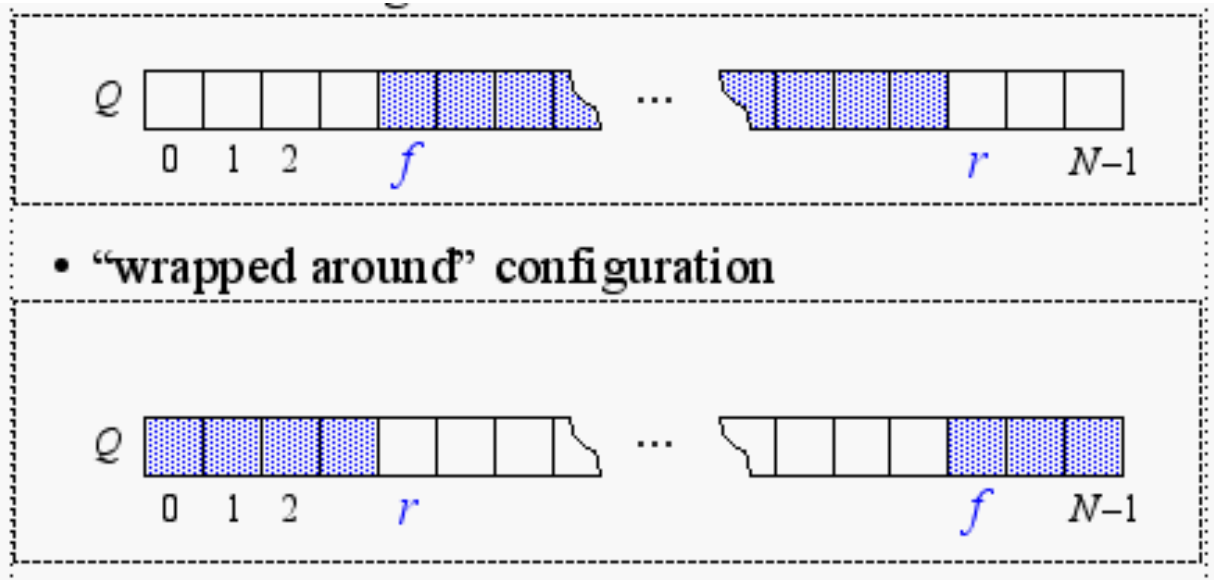


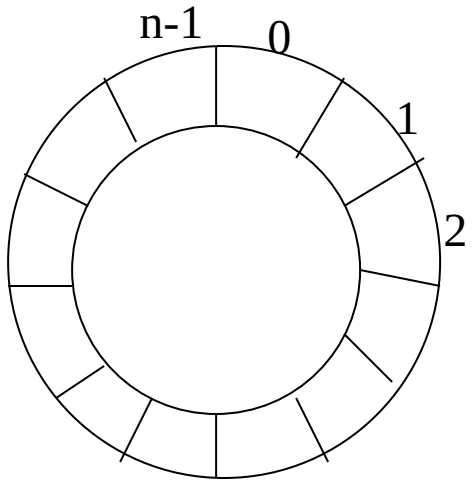


defiler: Front = (Front+1) **mod n**

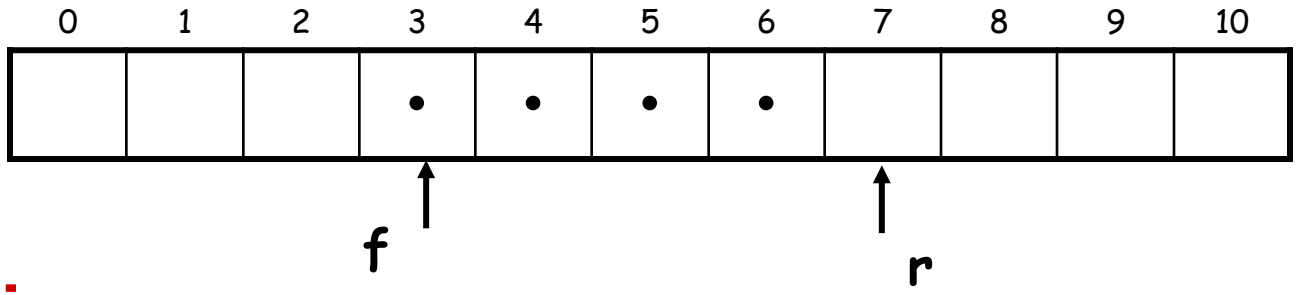
enfiler(e): Rear = (Rear+1) **mod n**

- La file est composée d'un tableau *Q* de *N* éléments et de deux variables entières:
  - f, l'index de l'élément du devant
  - r, l'index de l'élément suivant celui de l'arrière qui doit toujours pointer à une case vide
  - la file ne peut contenir que N-1 éléments





Implementation d'une file avec tableau circulaire

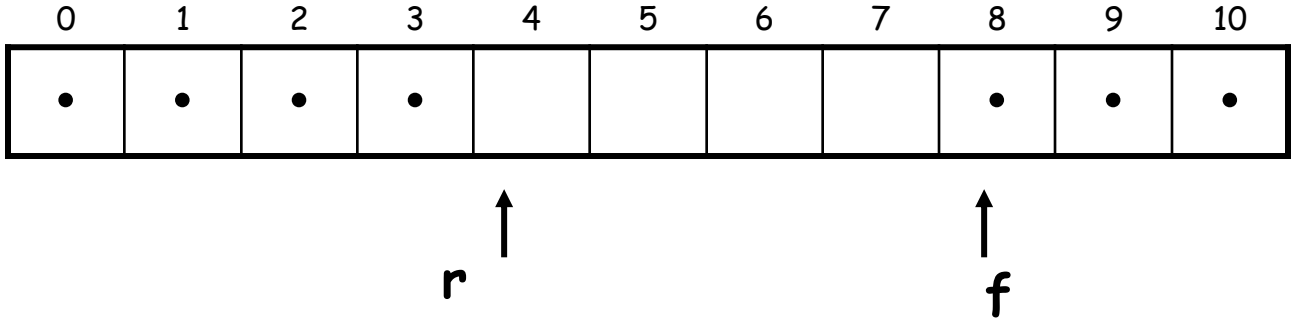


Questions:

Que veut dire *f* = *r*?

La File est vide.
Si la file est pleine, f est juste devant r.

Implementation d'une file avec tableau circulaire



Comment calculer le nombre d'éléments dans la file ?

(N - f + r) mod N

exemple:
(11 - 8 + 4) mod 11 = 7

Implementation d'une file avec tableau circulaire

Algorithme **taille**():
    returnrt (N - f + r) mod N

Algorithme **estVide**():
        returnrt (f = r)

Algorithme **tête**():
        si estVide() alors
            ERREUR
        returner Q[f]

Algorithme dequeue():
        si estVide() alors
            ERREUR
        temp ← Q[f]
        Q[f] ← null
        f ← (f + 1) mod N
        returner temp

Algorithme enqueue(o):
        si taille = N - 1 alors
            ERREUR
        Q[r] ← o
        r ← (r+1) mod N

Implémentation d'une file avec tableau circulaire

- Performance

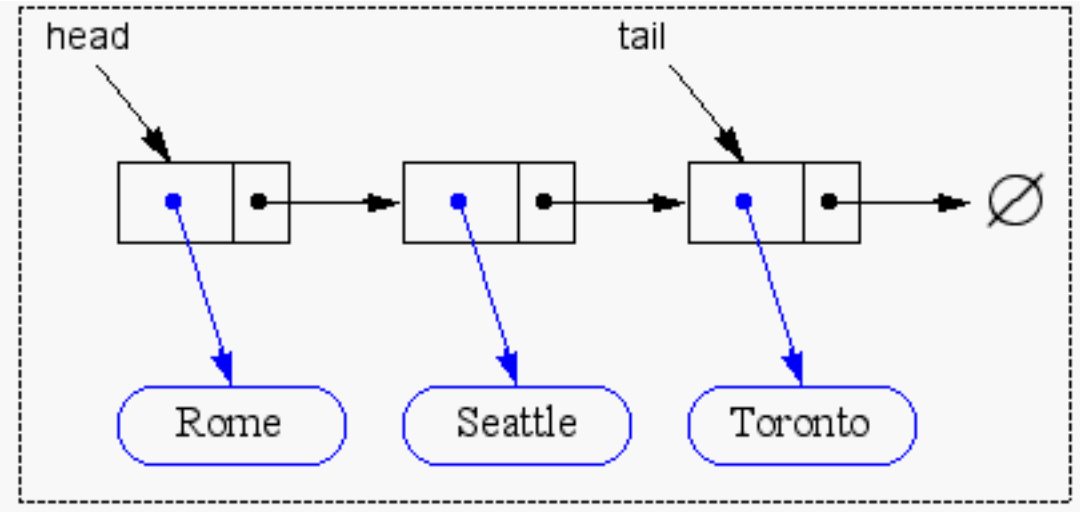
temps:

taille()	O(1)
estVide()	O(1)
tête()	O(1)
enqueue(o)	O(1)
dequeue()	O(1)

espace: O(N)

Réalisation d'une File à l'aide d'une liste simplement chaînée

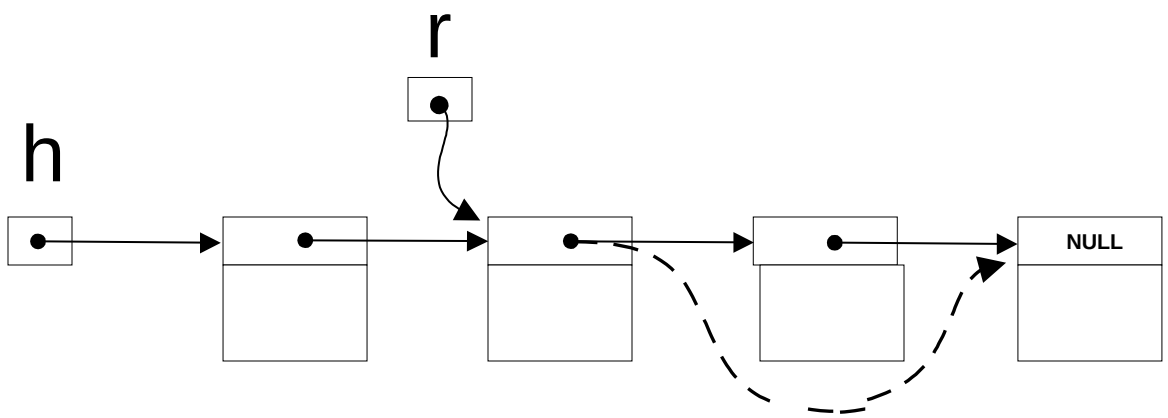
Nœuds connectés en chaîne par de liens (links)



La tête de la liste (head) est le début de la file, la queue de la liste (tail) constitue l'arrière de la file.

Pourquoi pas le contraire?

Rappel : Suppression

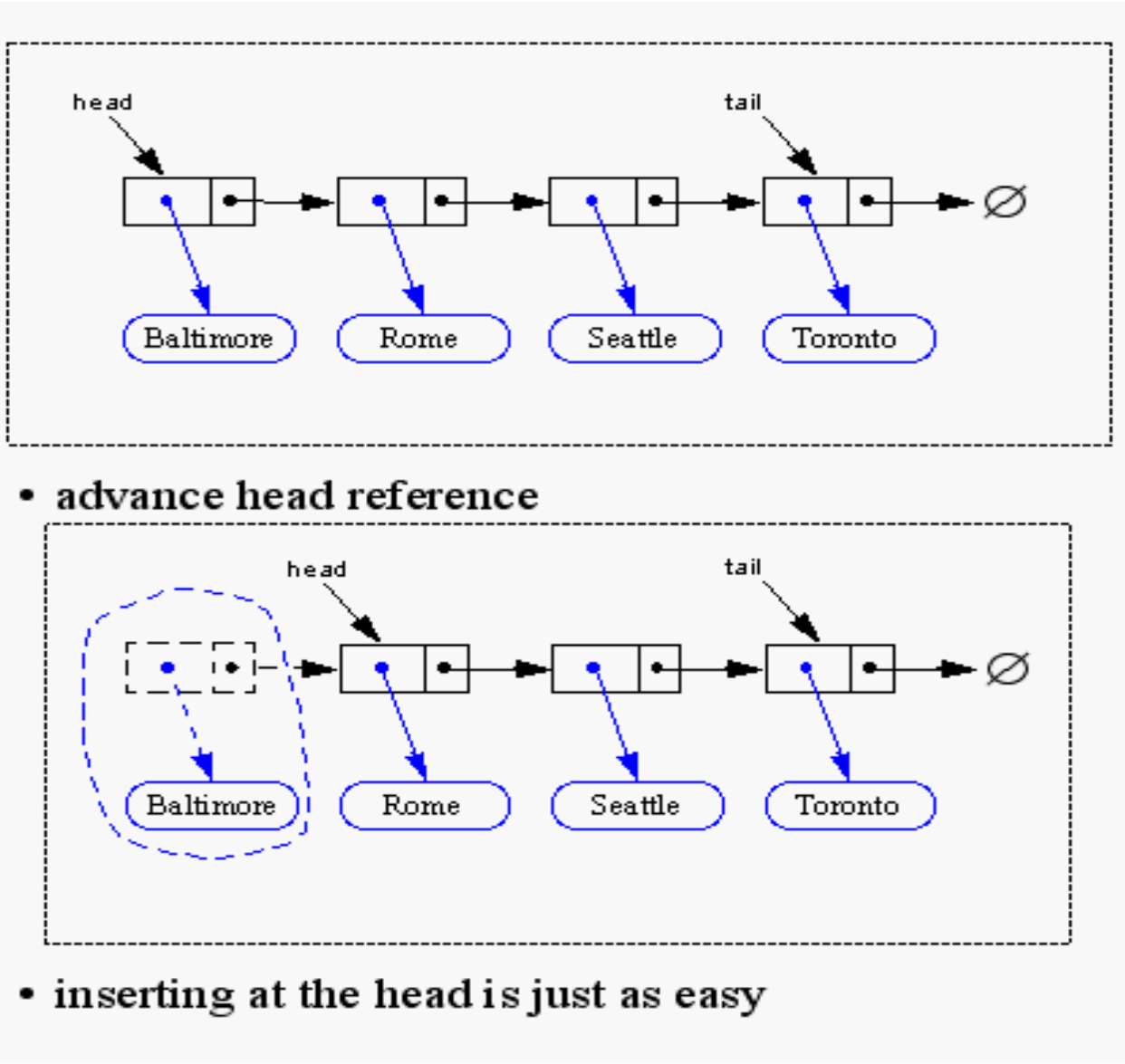


Premier élément (facile)	h ← h.getNext()
Élément après *r* (facile)	w ← r.getNext(); r.setNext(w)
Élément à r (difficile)	Utiliser un pointeur à l'élément précédant, ou Échanger les contenus de l'élément à *r* avec les contenus de l'élément suivent, et effacer l'élément après *r*. **Très difficile si *r* indique dernier élément!

Retirer l'élément de tête

Avancez la référence de la tête

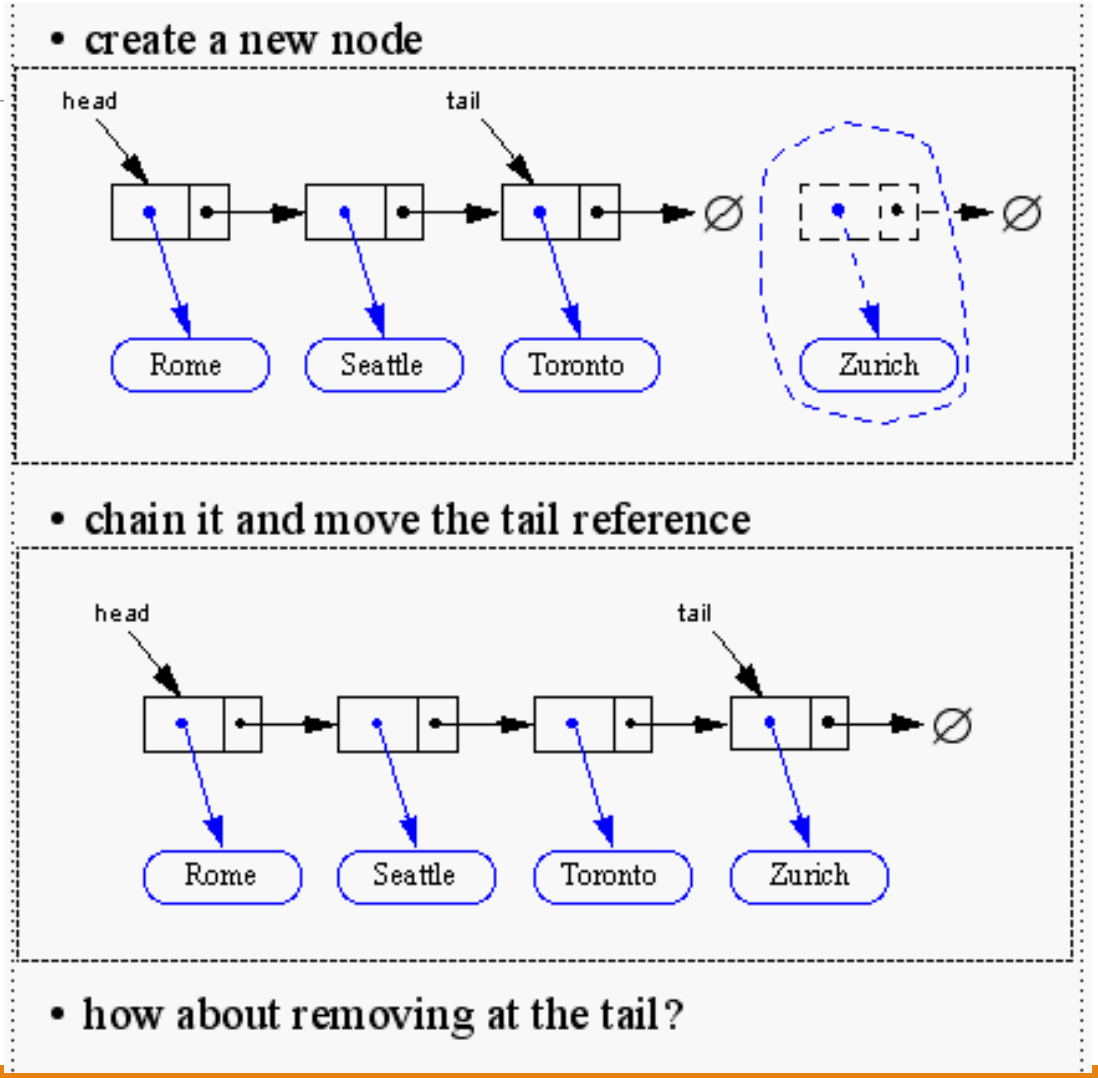
Insérer un élément à la tête est tout aussi facile



Insérer un élément à la queue

Créez un nouveau nœud

Enchaînez-le et déplacez la référence à la queue



Réalisation d'une File à l'aide d'une liste simplement chaînée

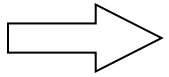
	temps:
taille()	O(1)
estVide()	O(1)
tête()	O(1)
enqueue(o)	O(1)
dequeue()	O(1)

Espace: Variable

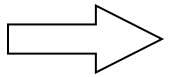
Performance

File: Tableau ou Liste chaînée ?

Si une limite supérieure raisonnable est connue à l'avance pour le nombre d'éléments dans la file, alors

 Tableau

Autrement

 Listes

Piles et Files: Utilisation en informatique

- File
  - *La file des événements stockés par le système Java GUI*
  - *File de tâches d'impression*
- Pile
  - *La pile d'exécution que le processeur ou la machine virtuelle maintient pour organiser les variables des méthodes imbriquées*

Piles et Files dans la bibliothèque Java

- La classe `Stack` implémente une structure de données abstraite - pile avec les opérations `push` et `pop`
- Méthodes de l'interface File (`Queue`) de Java inclut:
  - `add` *pour ajouter un élément à la fin de la queue*
  - `remove` *pour supprimer la tête de la queue*
  - `peek` *pour accéder au élément dans la tête de la queue sans le supprimer*
- La classe `LinkedList` implémente l'interface `Queue`, et vous pouvez utiliser lorsque vous en avez besoin:

```
Queue<String> q = new LinkedList<String>();
```

Travailler avec les piles et filles

Table 4 Working with Queues and Stacks

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.
`q.add(1); q.add(2); q.add(3);`	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.
`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.
`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.
`s.push(1); s.push(2); s.push(3);`	Adds to the top of the stack; `s` is now `[1, 2, 3]`.
`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

Big Java, C.Horstman