Structures de données

Structures de Données

- La structure de données que vous choisirez peut engendrer une grande différence
  - Au moment de l'implémentation des méthodes
  - En termes de performances
- On mettra en évidence la façon dont la bibliothèque Java peut aider à trouver une structure de données adaptée à une programmation sérieuse

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

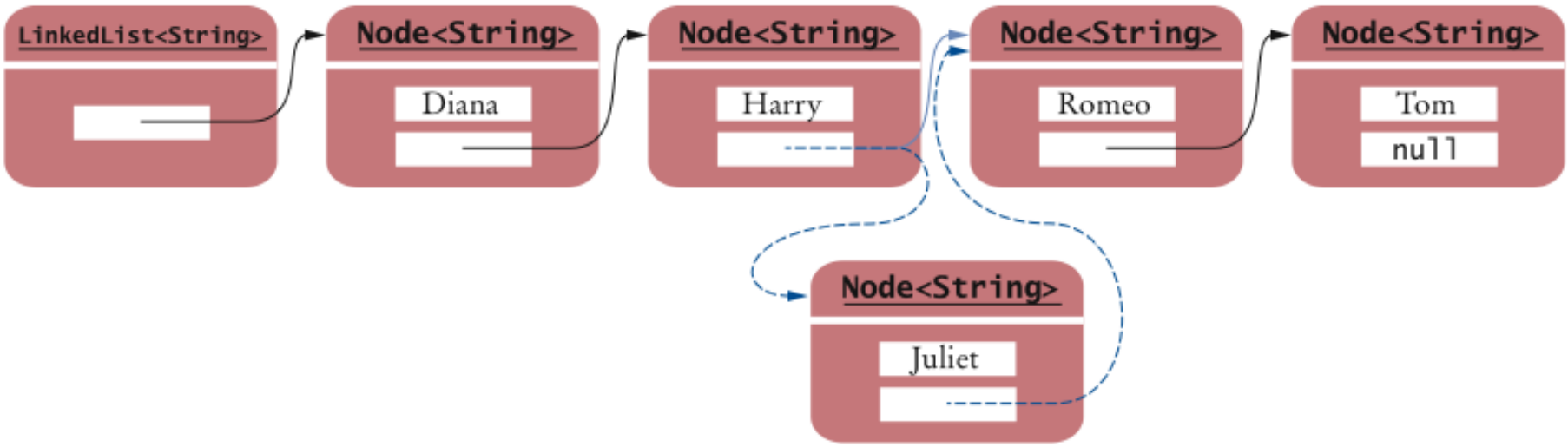


Figure 1 Inserting an Element into a Linked List

Classe `LinkedList` de Java

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

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).

List Iterator

- 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 quelleconque dans la liste
- Protège la liste lorsque l'on accède

Itérateur d'une liste

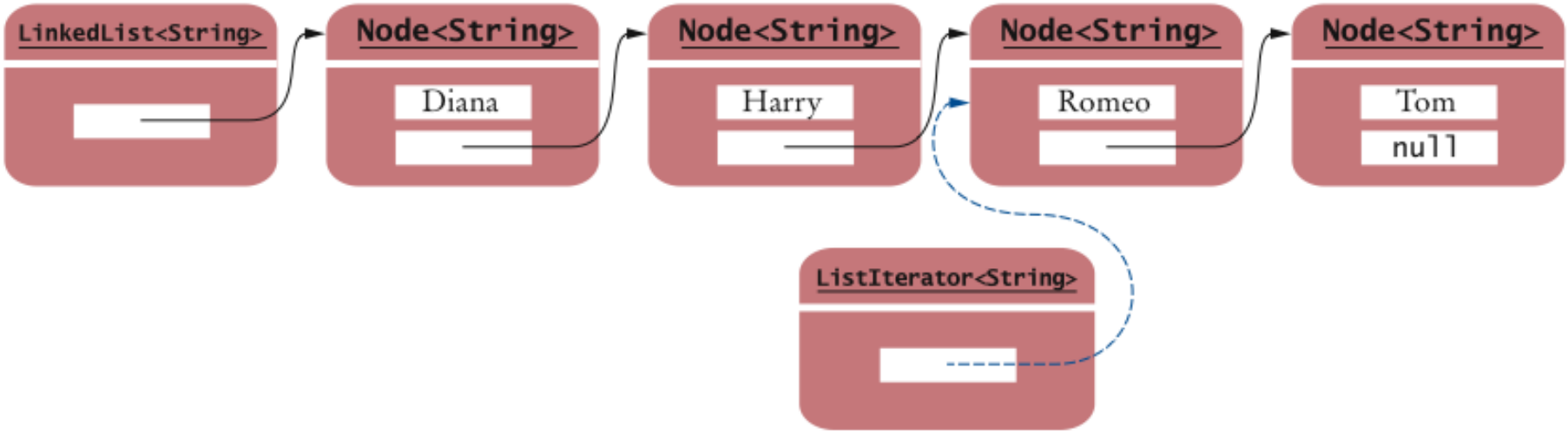


Figure 2 A List Iterator

Vue conceptuelle de ListIterator

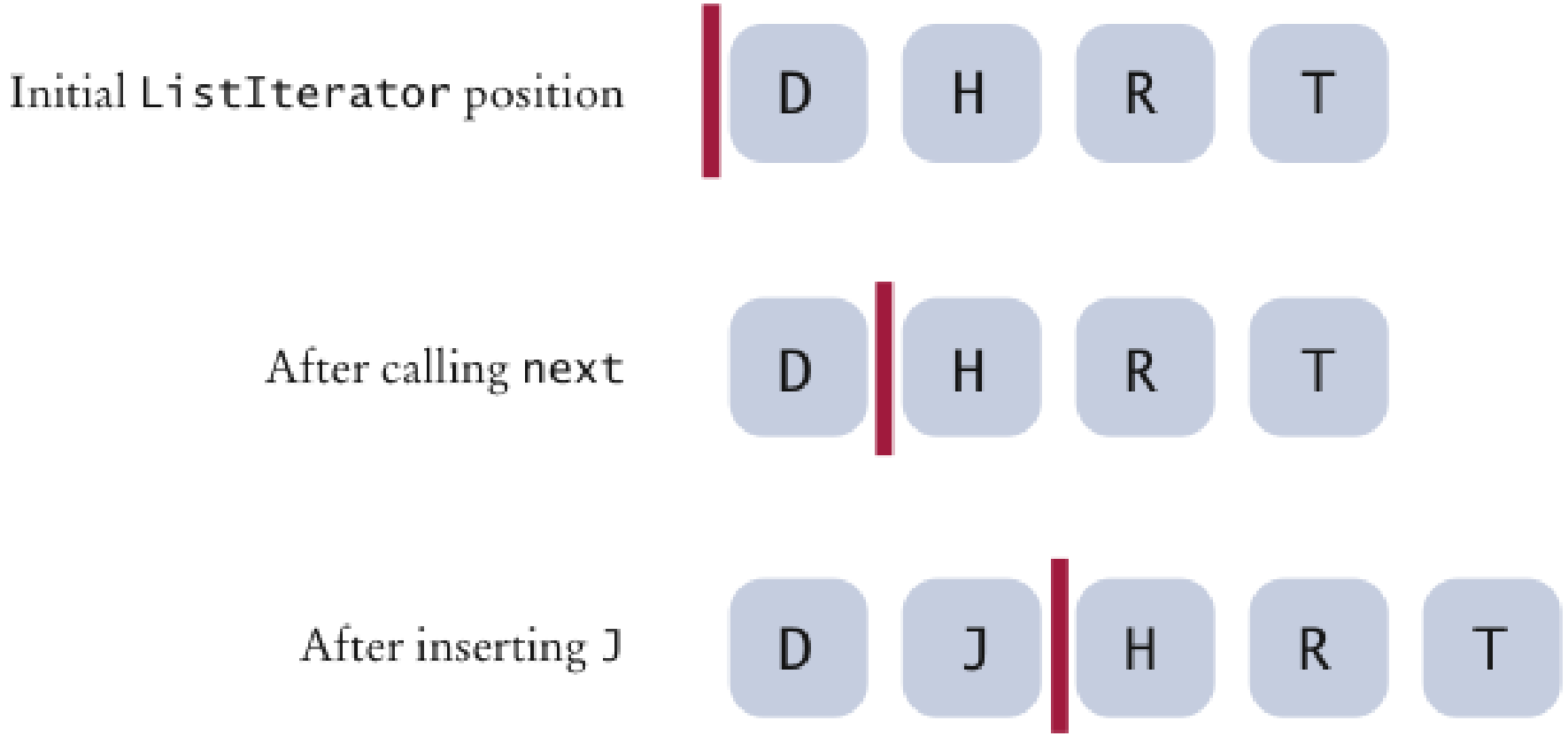


Figure 3 A Conceptual View of the List Iterator

Itérateur d'une liste

- Itérateur pointe vers la position entre deux éléments
  - *Analogie: Comme un curseur 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();
   Do something with name
}
```

- Raccourci:

```
for (String name : employeeNames)
{
   Do something with 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]`.

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 listes chaînées

- 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 listes chaînées

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

Implémentation de listes chaînées

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

Implémentation de listes chaînées

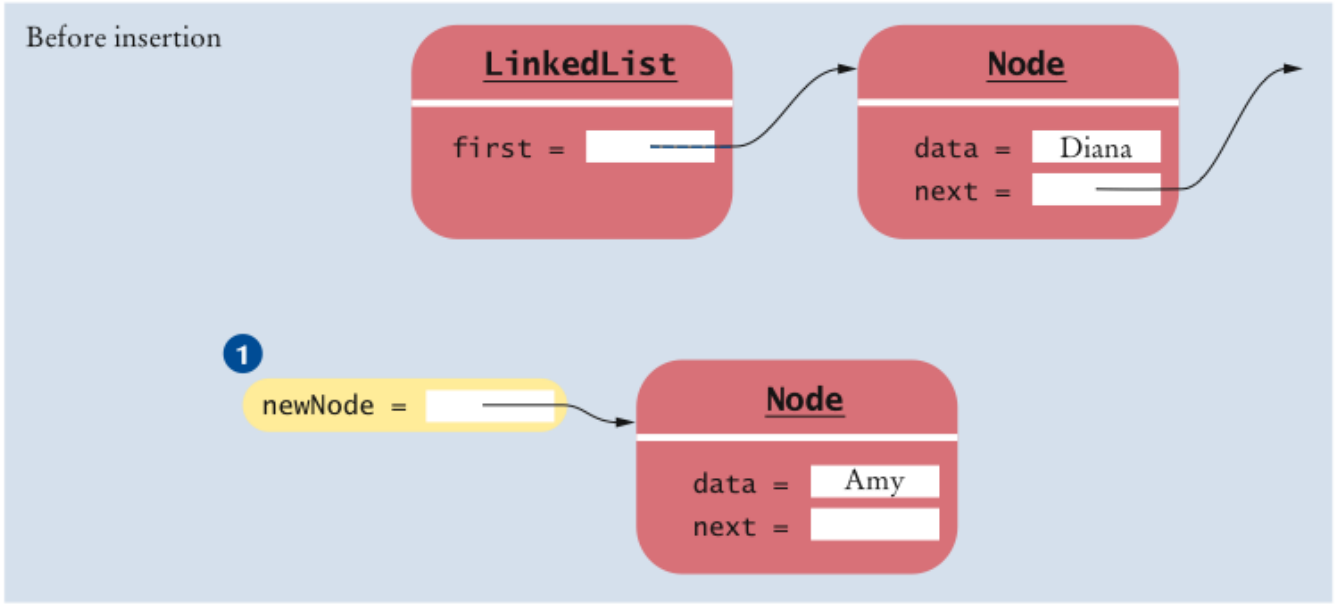
```
public class LinkedList
{
   private Node first;
   ...
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
}
```

Ajouter le nouveau premier élément

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

Ajouter le nouveau premier élément

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



Ajouter le nouveau premier élément

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

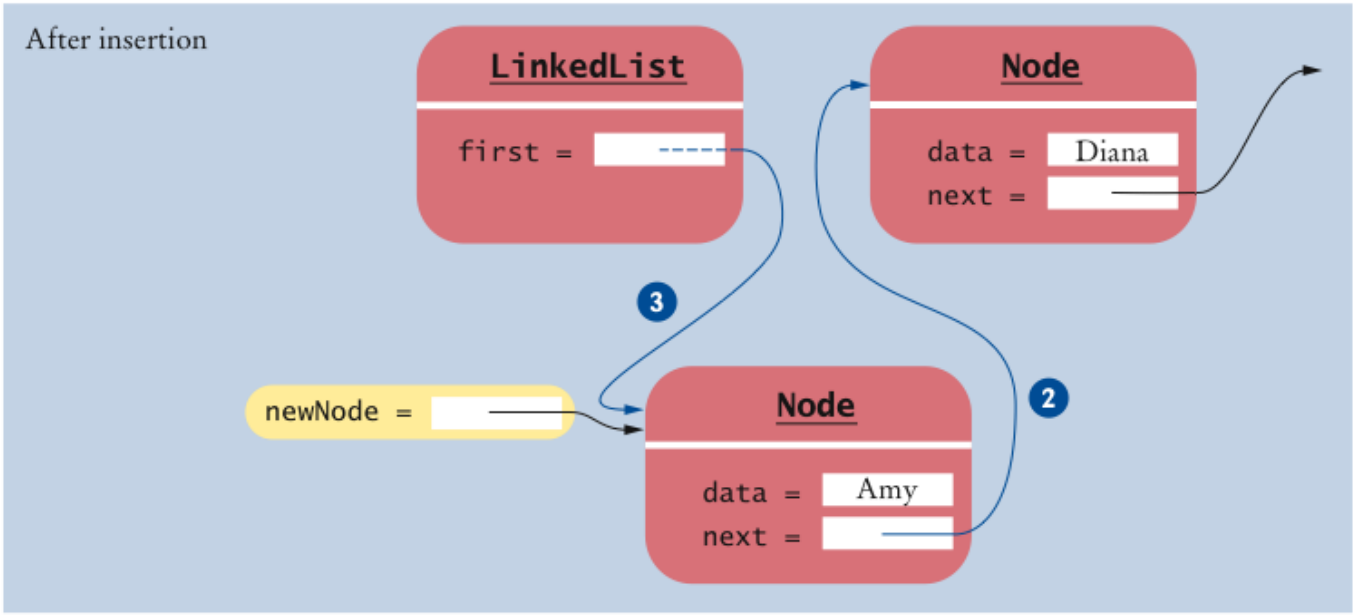


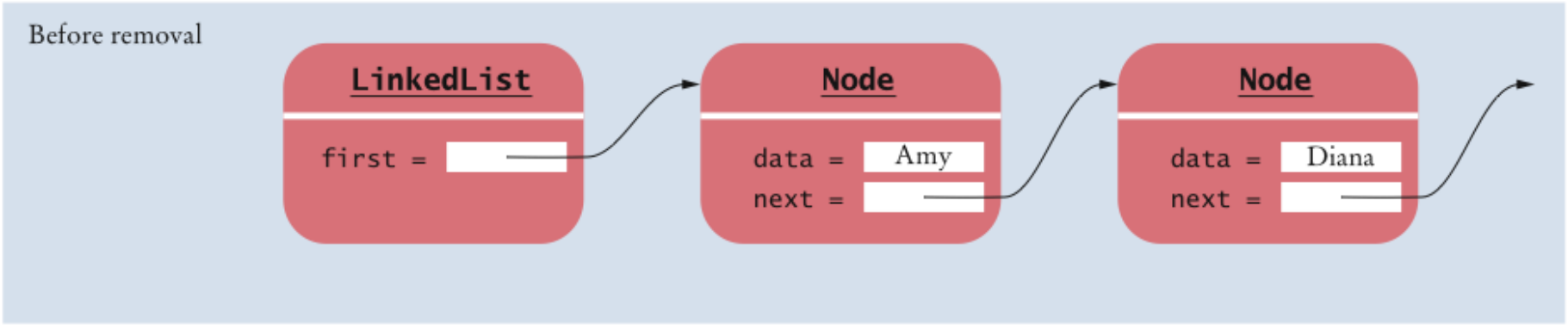
Figure 4 Adding a Node to the Head of a Linked List

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 les 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;
}
```



Retirer le premier élément

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

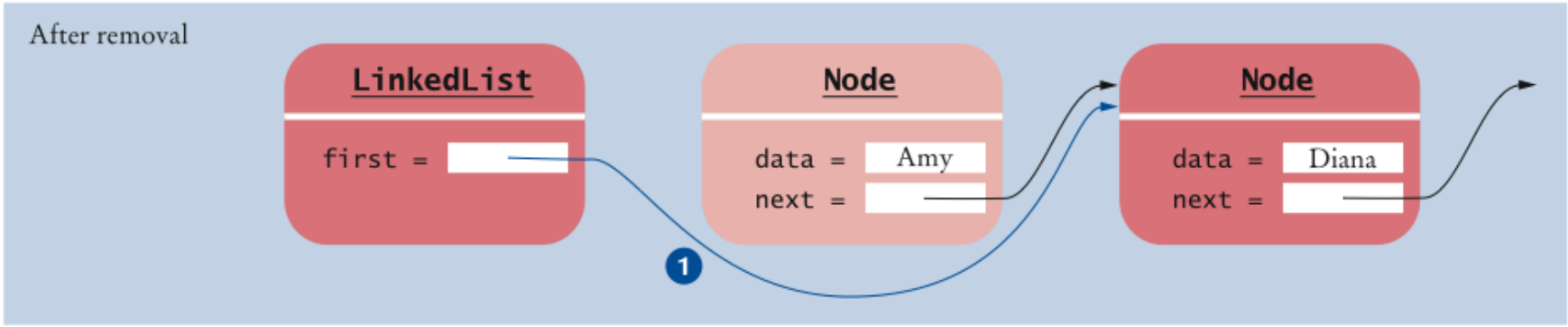


Figure 5 Removing the First Node from a Linked List

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 à la classe privée `Node`
- Clients de `LinkedList` ne savent pas le nom de la classe itérateur (`LinkedListIterator`)
  - *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
- `previous`: Référence vers le nœud visité avant le dernier nœud
- Méthode `next`:
  - référence `position` est avancée vers `position.next`
  - 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`

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

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

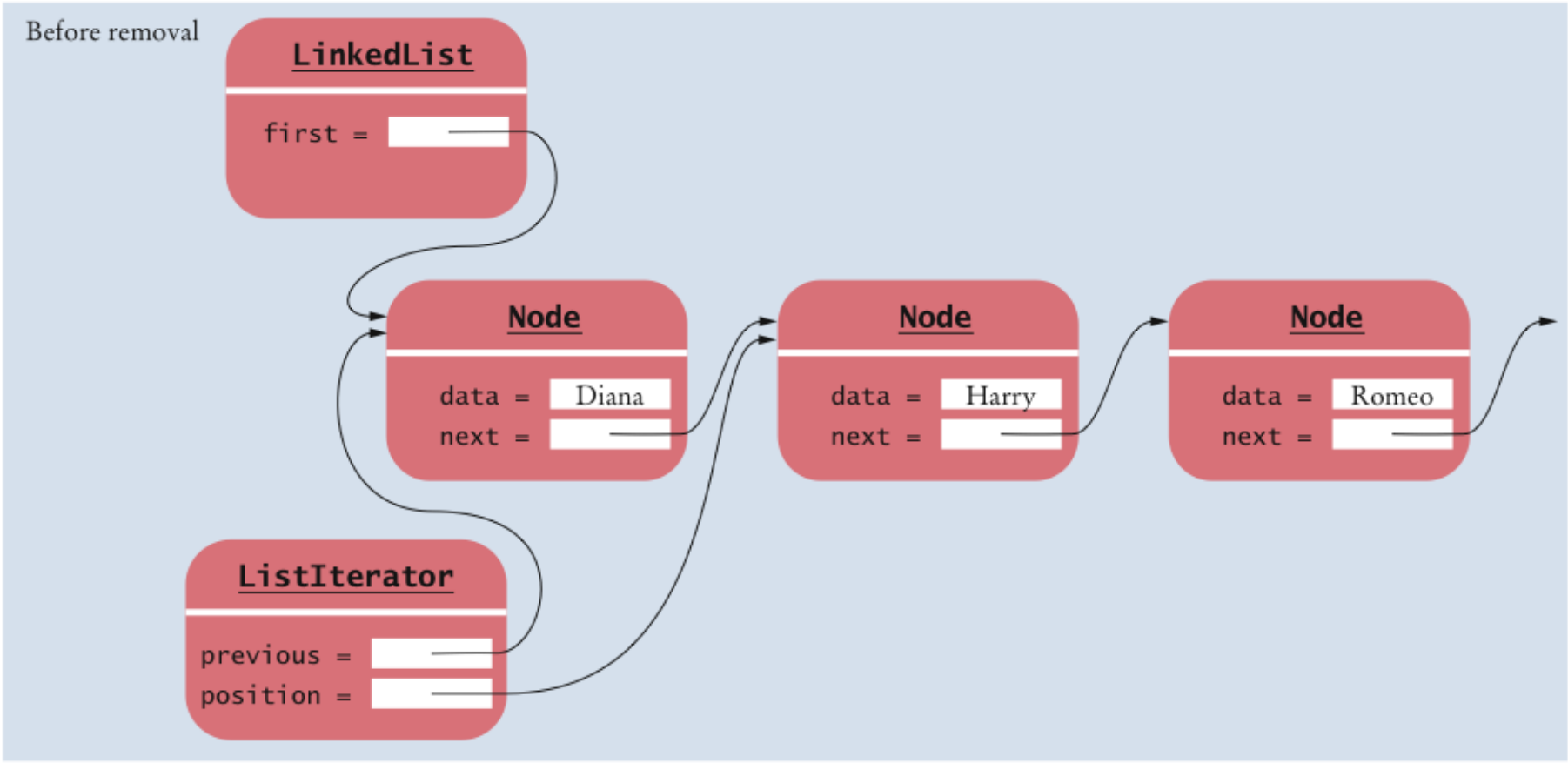
- Si un élément à supprimer est le premier élément, appelez `removeFirst`
- Si non, le nœud précédant celui à supprimer a besoin de mettre à jour sa référence `next` pour sauter l'élément à supprimer
- Si la référence `previous` est égale à `position`:
  - *Cette appel ne suis pas immédiatement l'appel* `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



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

```
else
{
   previous.next = position.next;
}
position = previous;
```

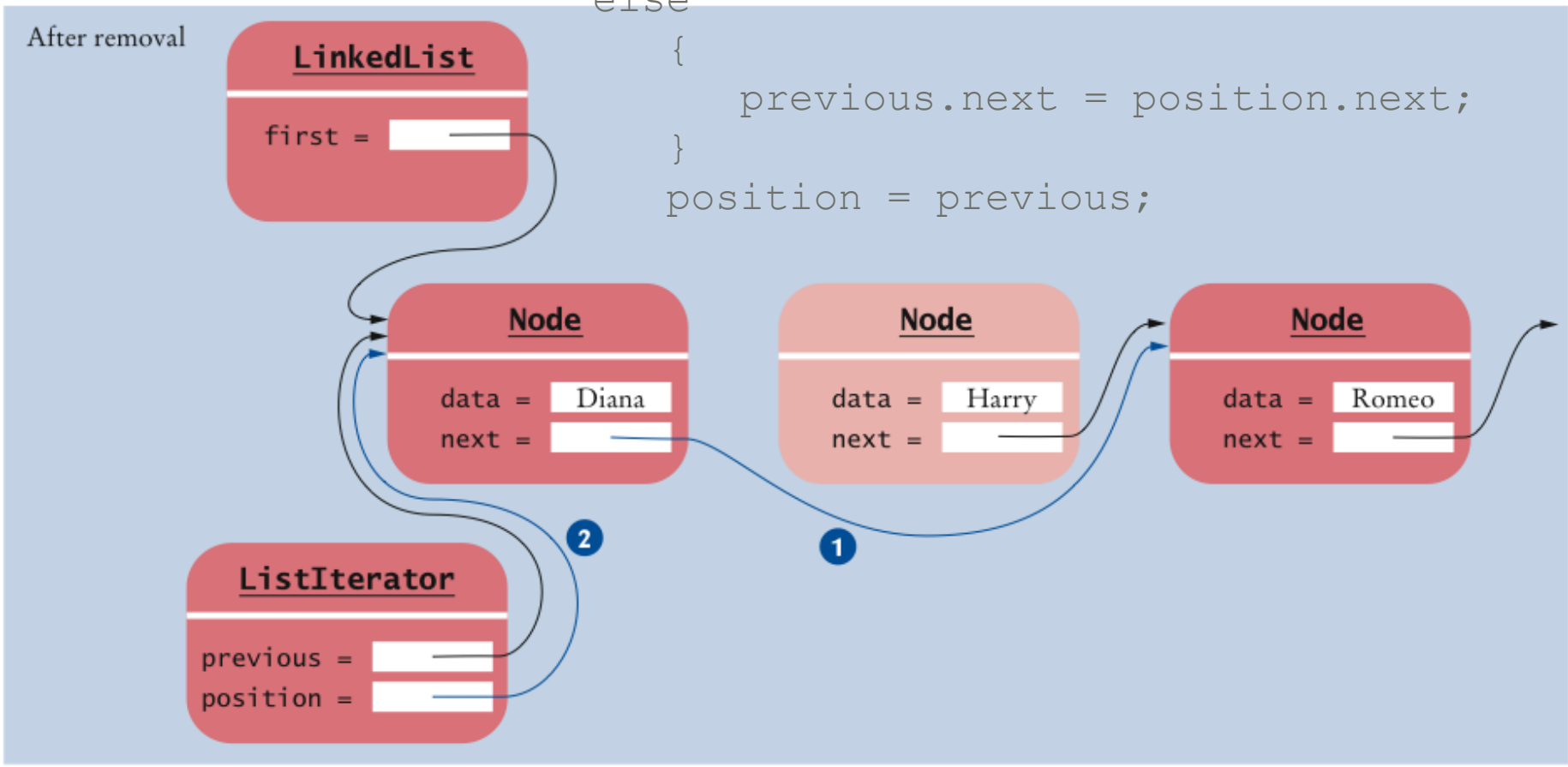


Figure 6 Removing a Node from the Middle of a Linked List

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

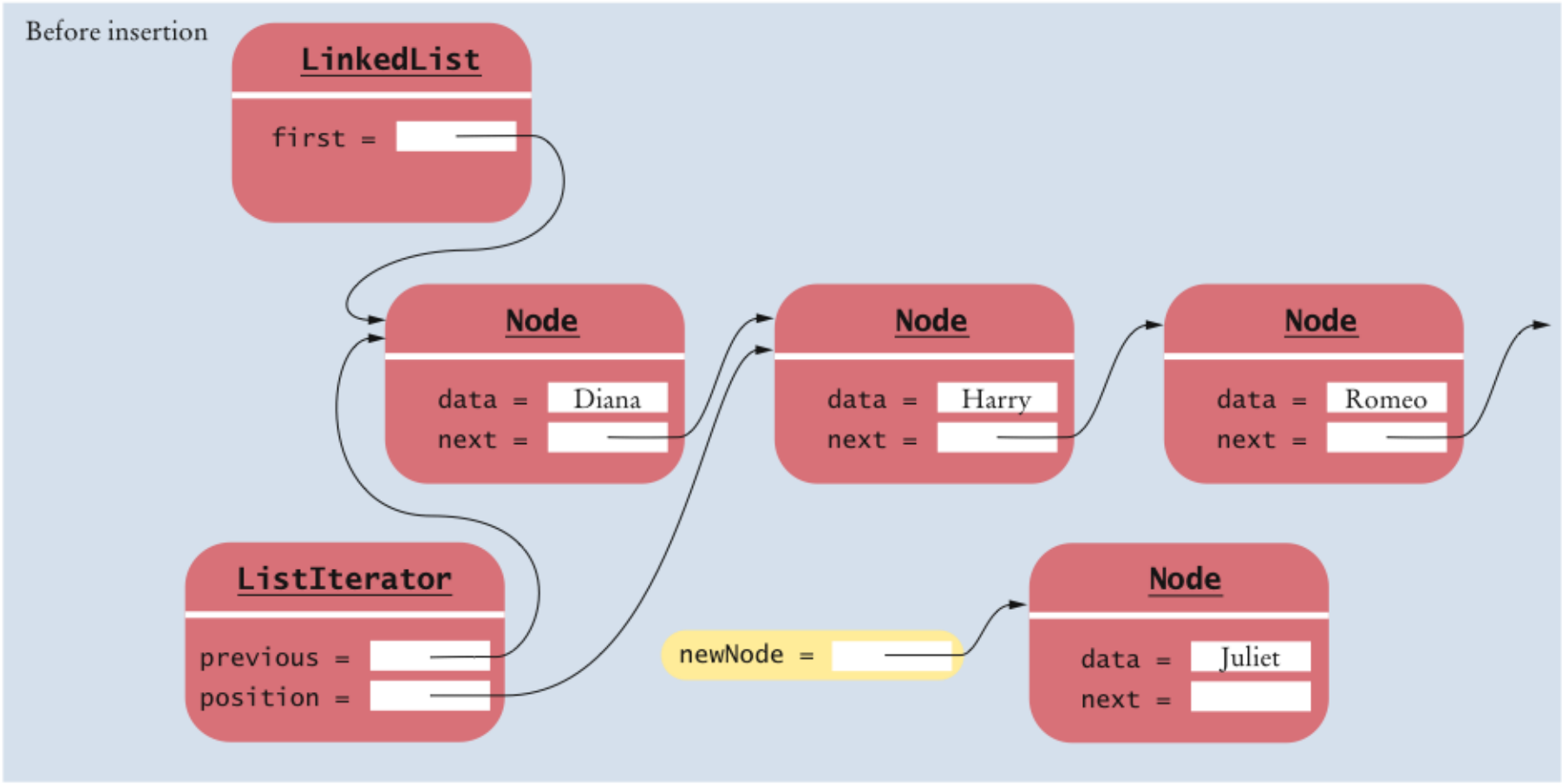
Méthode `add` de l'itérateur de la liste chaînée

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

}
```

Continued

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



Continued

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

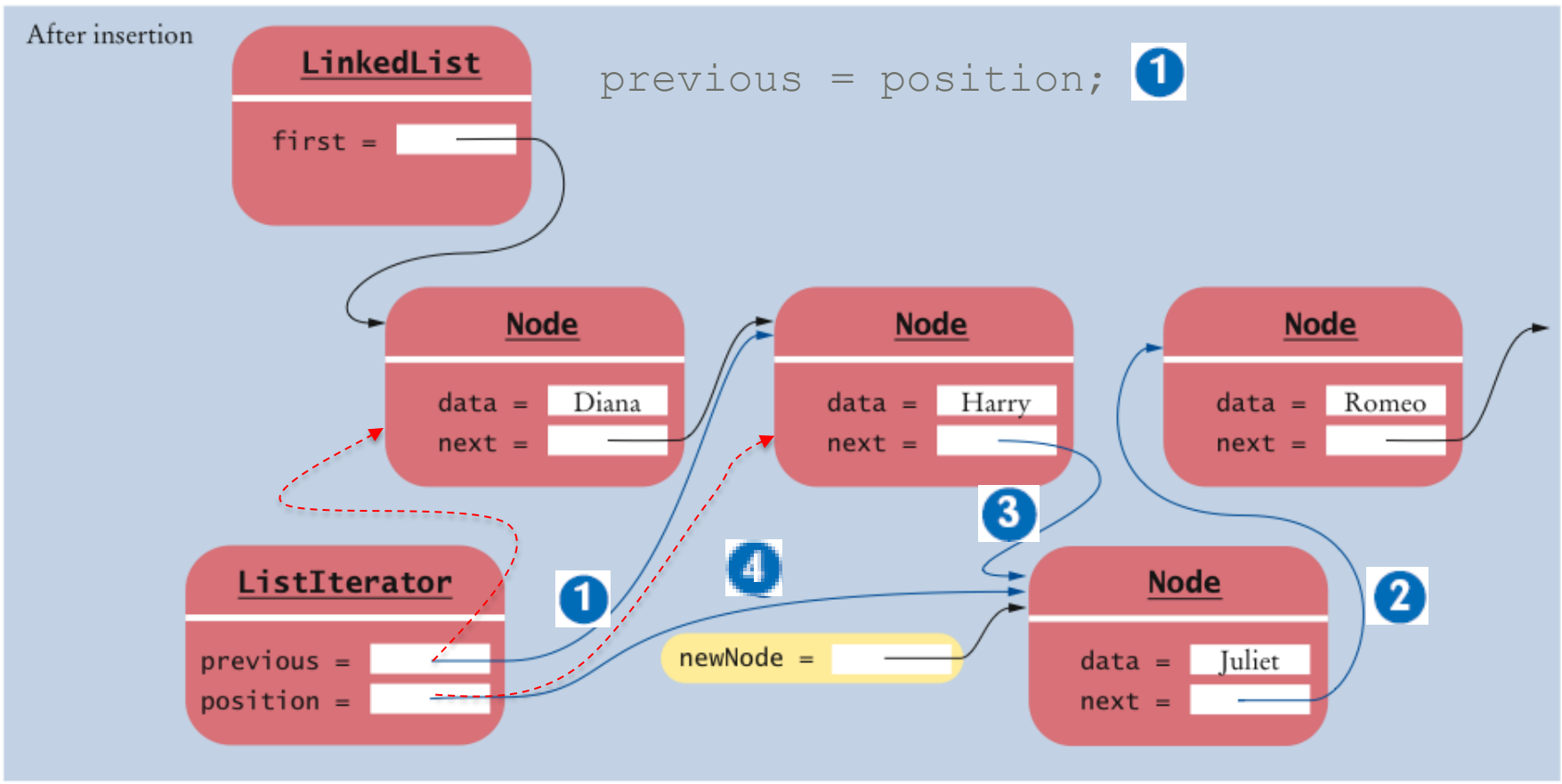


Figure 7 Adding a Node to the Middle of a Linked List

```
Node newNode = new Node();
newNode.data = obj;
newNode.next = position.next;  2
position.next = newNode;  3
position = newNode;  4
```

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.)

```
/**
   Adds an element before the iterator position
   and moves the iterator past the inserted element.
   @param element the element to add
 */
public void add(Object element)
{
        previous = position;

        if (position == null)
        {
                addFirst(element);
                position = first;
        }
        else
        {
                Node newNode = new Node();
                newNode.data = element;
                newNode.next = position.next;
                position.next = newNode;
                position = newNode;
        }
}
```

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   }
```

Structures de données abstraites

- 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

Structures de données abstraites

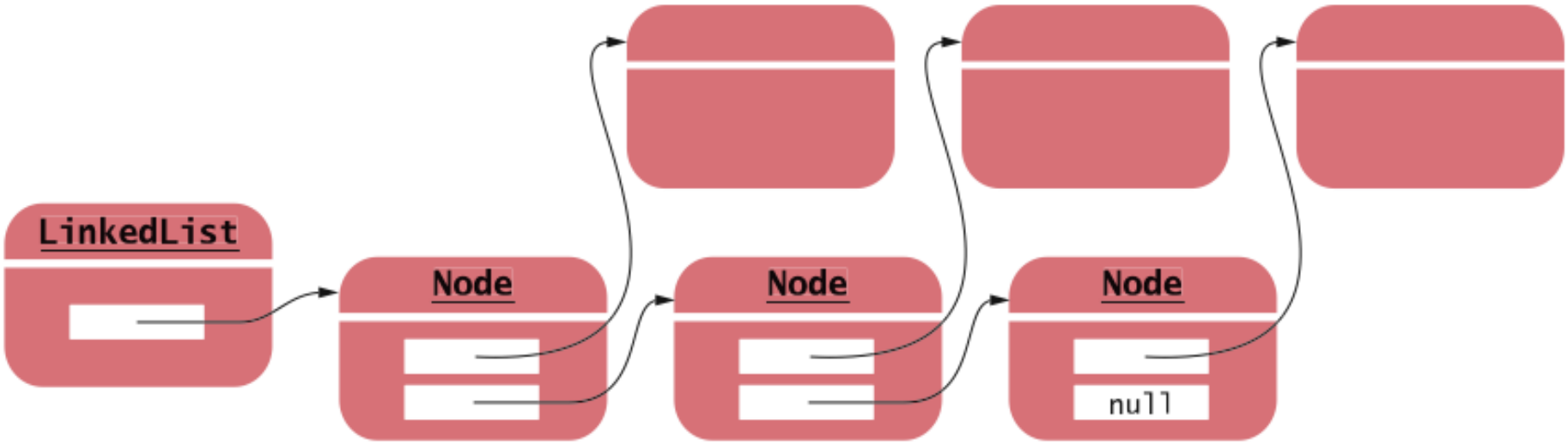


Figure 8 A Concrete View of a Linked List

Figure 9 An Abstract View of a List

Structures de données abstraites

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

Structure de données abstraite et concrète *Tableau*

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

Structure de données abstraite et concrète

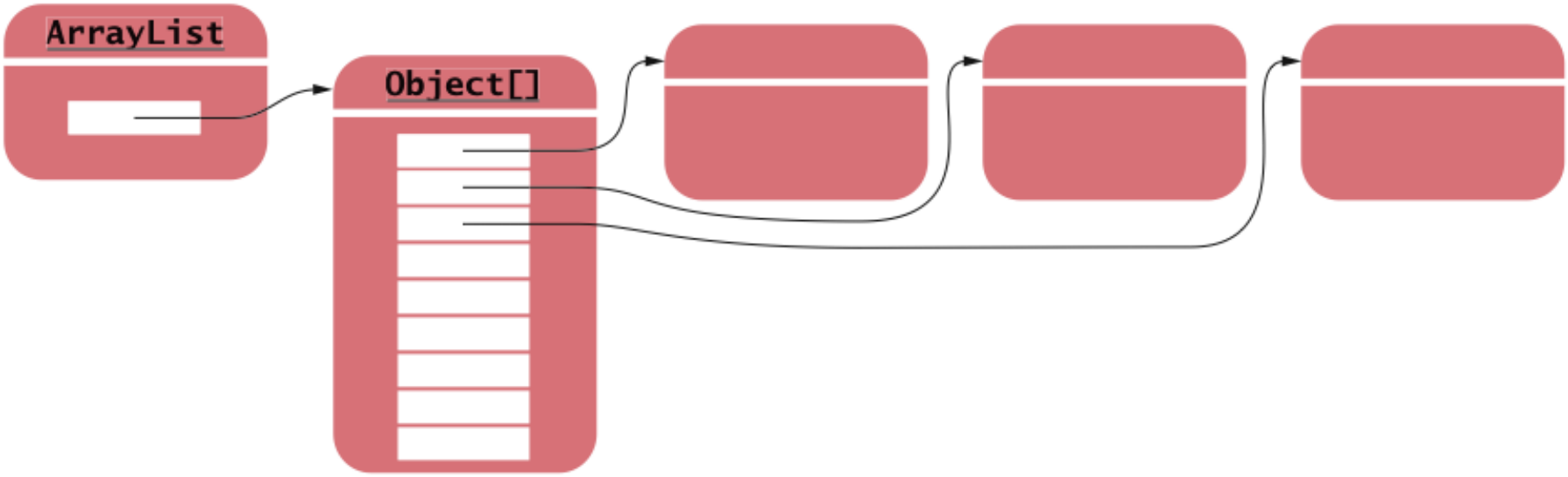


Figure 10 A Concrete View of an Array List

Figure 11 An Abstract View of an Array

Structure de données abstraite et concrète

- Les implémentations concrètes d'une liste chaînée et un tableau liste 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	$O(n)$	$O(1)$

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
- Un ordre: dernier entré, premier sorti ou LIFO
- Traditionnellement, les opérations d'ajout et de retrait sont appelées `push` et `pop`
- Pensez de la pile de livres

Pile

Figure 12
A Stack of Books

Queue

- Une queue permet d'ajouter efficacement des éléments à la fin et d'en supprimer au début, ordre FIFO
- Il est impossible d'ajouter des éléments au milieu
- Pensez de la fil d'attente des gens

Queue

Figure 13 A Queue

Piles et Queues: Utilisation en informatique

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

Piles et Queues dans la bibliothèque Java

- La classe `Stack` implémente une structure de données abstraite - pile avec les opérations `push` et `pop`
- Méthodes de l'interface `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 accèder à un 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 les queues

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.

Ensembles (Sets)

- **Ensemble**: Une collection non ordonnée des éléments distincts
- Les éléments peuvent être ajoutés, trouvés et supprimés
- Les ensembles ne possèdent les duplications

Operations fondamentales sur les ensembles

- Ajouter un élément
  - *Si un élément est déjà dans un ensemble -> cette opération n'a pas d'effet*
- Supprimer un élément
  - *Si un élément n'est pas dans un ensemble -> cette opération est ignorée*
- Tester si un élément est présent
- Lister tous les éléments

Un ensemble d'imprimantes

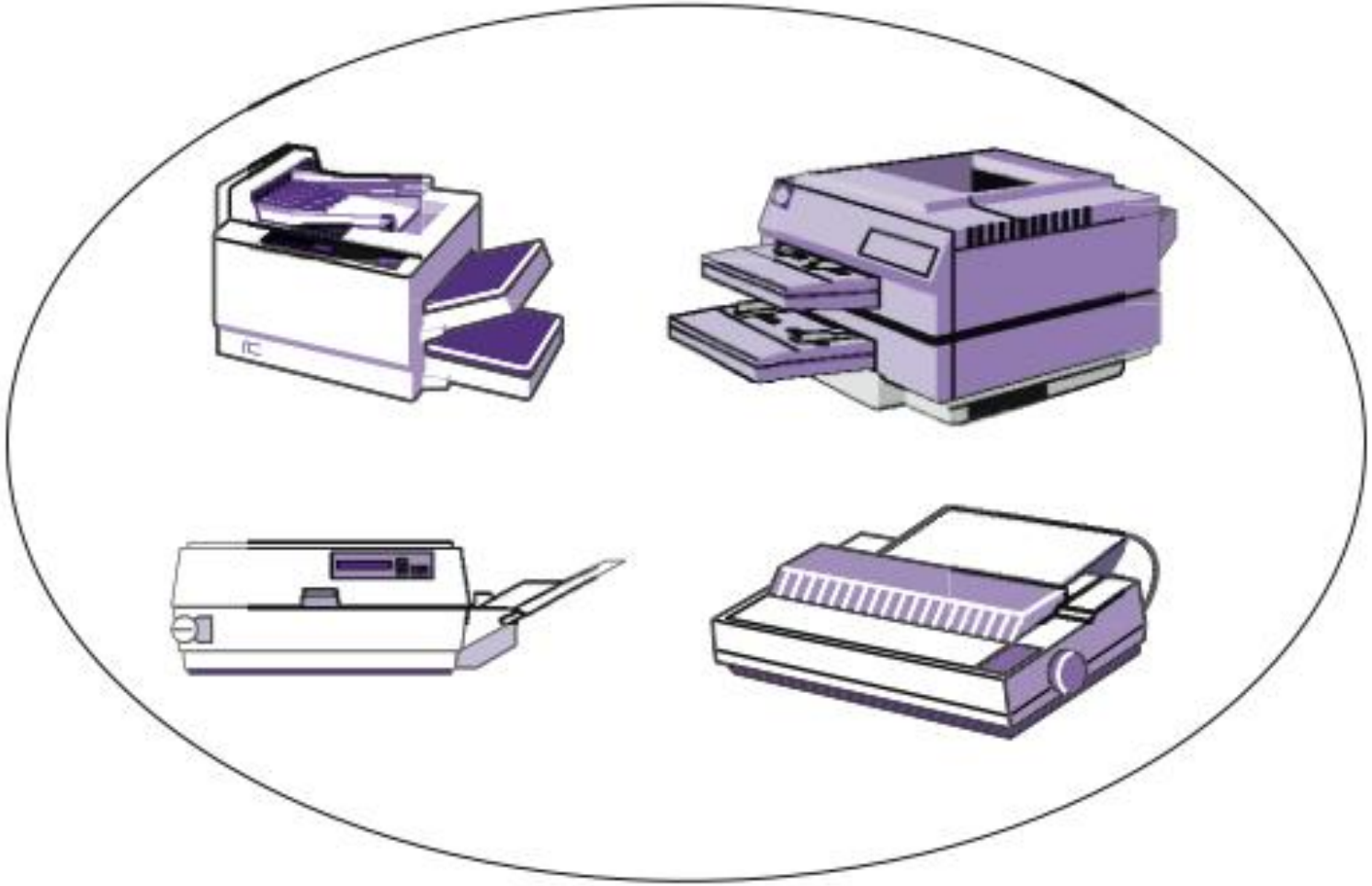


Figure 1 A Set of Printers

Ensembles (Sets)

- On pourrait utiliser la liste chaînée pour implémenter un ensemble
  - *Ajouter, supprimer et tester si un élément est présent seraient assez lents*
- Il existe des structures de données supportant ces opérations beaucoup plus efficace qu'une liste chaînée
  - *Les tables de hachage*
  - *Arbres*

Ensembles (Sets)

- La bibliothèque Java propose les deux implémentations de *set* basée sur les deux structures de données mentionnées
  - *HashSet*
  - *TreeSet*
- Ces deux implémentations implémentent l'interface `Set`
- Remarque: utiliser un *HashSet* dans les cas où vous n'avez pas besoin de visiter les éléments dans l'ordre trié

Classes et Interfaces Set dans la bibliothèque Java

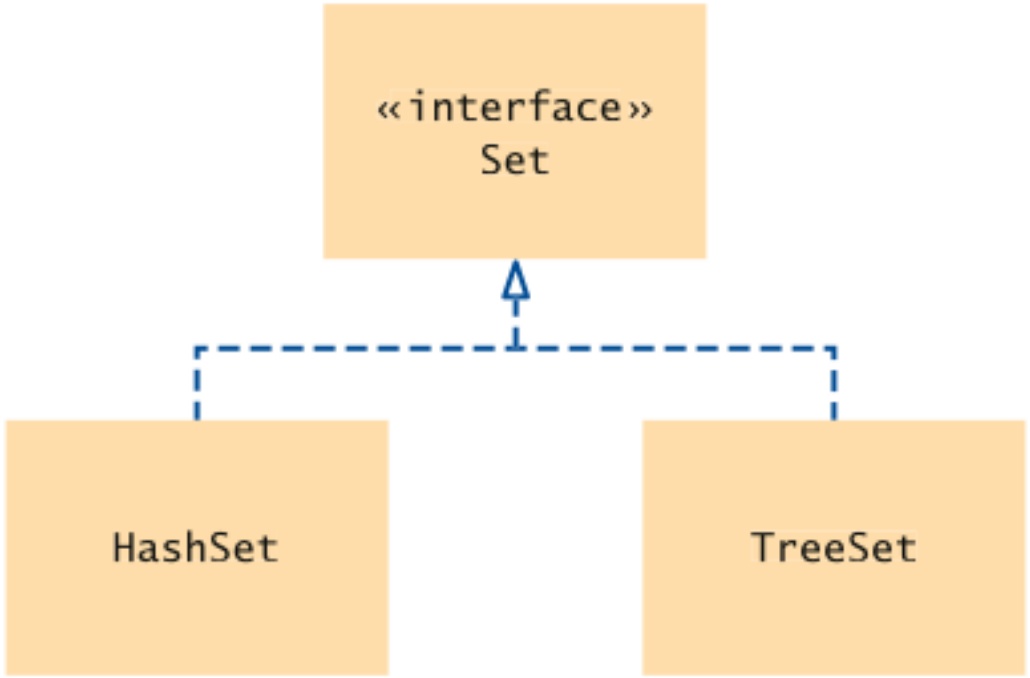


Figure 2
Set Classes and Interfaces in the Standard Library

Utiliser un `Set`

- Exemple: Utiliser un ensemble des chaînes
- Construire `Set`:

```
Set<String> names = new HashSet<String>();
```

ou

```
Set<String> names = new TreeSet<String>();
```

- Ajouter et supprimer les éléments:

```
names.add("Romeo");
names.remove("Juliet");
```

- Tester si un élément est présent dans l'ensemble:

```
if (names.contains("Juliet")) . . .
```

Itérateur

- Utiliser un itérateur pour visiter tous les éléments de l'ensemble
- Un itérateur d'ensemble ne visite pas les éléments dans l'ordre dans lequel ils ont été insérés
- Un élément d'ensemble ne peut pas être ajouté dans une position d'itérateur
- Un élément d'ensemble se trouvant dans une position d'itérateur peut être supprimé

Visiter tous les éléments avec un itérateur

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}
```

ou, utiliser la boucle “for each”:

```
for (String name : names)
{
   Do something with name
}
```

Un programme de test de Set

1. Lire tous les mots de fichier dictionnaire et les mettre dans un ensemble
2. Lire tous les mots du document concret (le livre “Alice in Wonderland”) dans un deuxième ensemble
3. Imprimer les mots du deuxième document qui ne se trouvent pas dans le premier document

ch16/spellcheck/SpellCheck.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
```

Continued

ch16/spellcheck/SpellCheck.java (cont.)

```
15          // Read the dictionary and the document
16
17          Set<String> dictionaryWords = readWords("words");
18          Set<String> documentWords = readWords("alice30.txt");
19
20          // Print all words that are in the document but not the dictionary
21
22          for (String word : documentWords)
23          {
24             if (!dictionaryWords.contains(word))
25             {
26                System.out.println(word);
27             }
28          }
29       }
30
```

Continued

ch16/spellcheck/SpellCheck.java (cont.)

```
31      /**
32         Reads all words from a file.
33         @param filename the name of the file
34         @return a set with all lowercased words in the file. Here, a
35         word is a sequence of upper- and lowercase letters.
36      */
37      public static Set<String> readWords(String filename)
38         throws FileNotFoundException
39      {
40         Set<String> words = new HashSet<String>();
41         Scanner in = new Scanner(new File(filename));
42         // Use any characters other than a-z or A-Z as delimiters
43         in.useDelimiter("[^a-zA-Z]+");
44         while (in.hasNext())
45         {
46            words.add(in.next().toLowerCase());
47         }
48         return words;
49      }
50   }
```

Continued

ch16/spellcheck/SpellCheck.java (cont.)

Program Run:

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

Cartes (Maps)

- La structure de données cartes (ou map) permet d'effectuer la recherche selon certaines informations importantes sur l'élément à rechercher
- Une carte enregistre des paires clé/valeur
- Une valeur peut être retrouvée à partir de la clé correspondante
- De point de vue mathématique, une carte est une fonction d'un ensemble, ensemble des clés, à un autre ensemble, un ensemble des valeurs
- Les clés doivent être uniques
- La même valeur peut être associée avec plusieurs clés

Cartes (Maps)

- La bibliothèque Java propose deux implémentations générales des cartes
  - `HashMap`
  - `TreeMap`
- Les deux classes implémentent l'interface `Map`
- Comment choisir entre une HashMap et une TreeMap?
  - Les cartes de hachage sont légèrement plus rapides et elles sont à privilégier si vous n'avez pas besoin de parcourir les éléments selon un ordre particulier

Exemple de la carte (map)

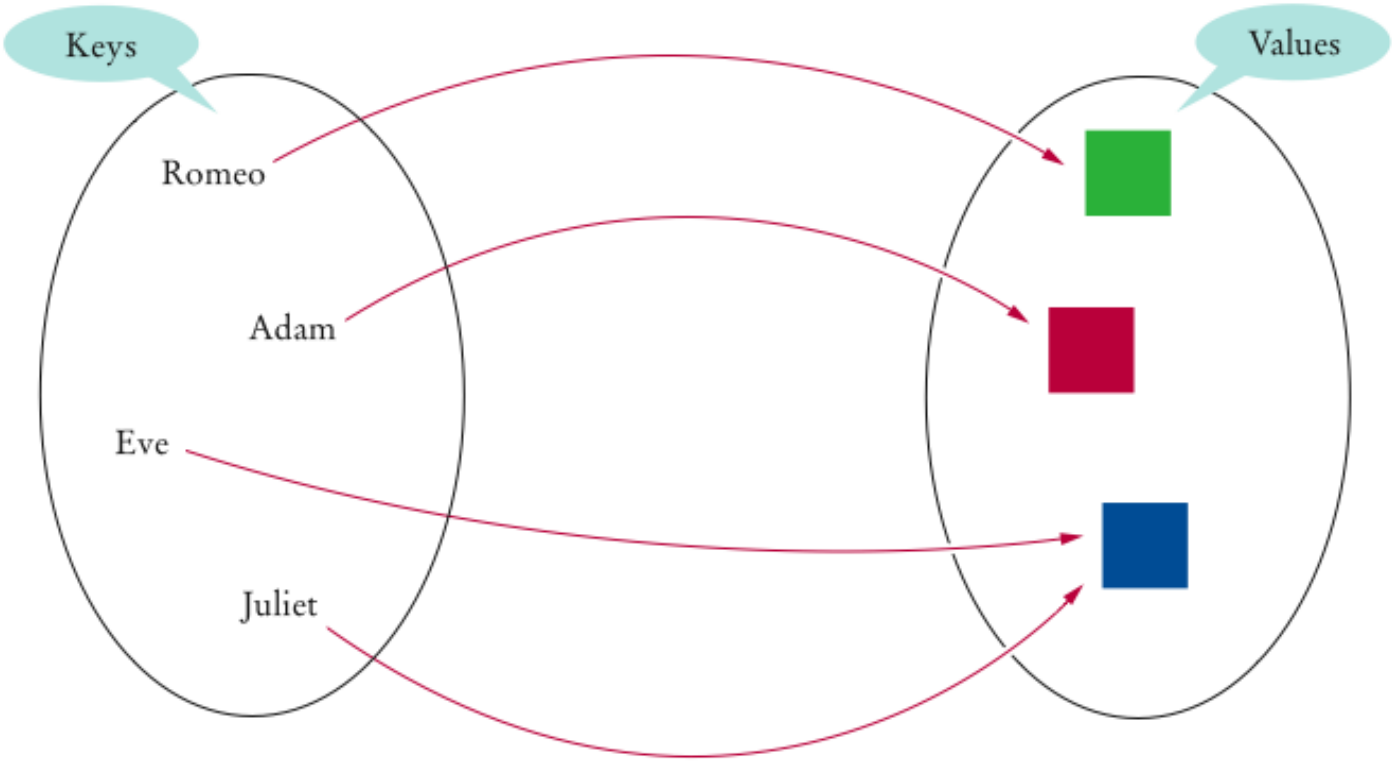


Figure 3 A Map

Carte: Classes et interfaces

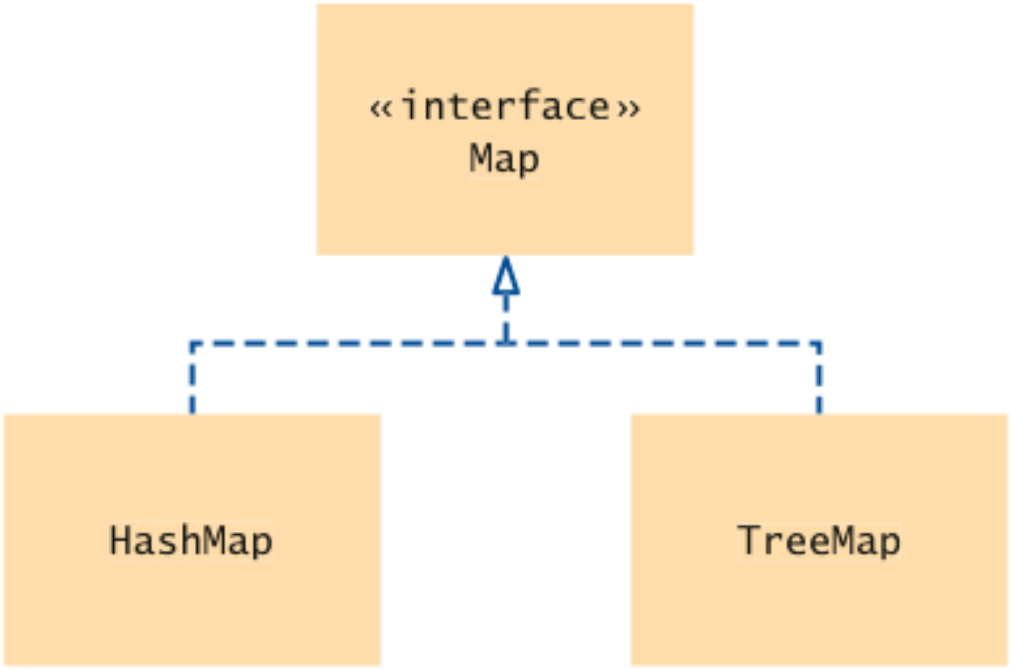


Figure 4
Map Classes and Interfaces in the Standard Library

Utiliser une carte, `Map`

- Exemple: Associer les noms avec les couleurs
- Construire une `Map`:

```
Map<String, Color> favoriteColors =
   new HashMap<String, Color>();
```

ou

```
Map<String, Color> favoriteColors =
   new TreeMap<String, Color>();
```

- Ajouter une association:

```
favoriteColors.put("Juliet", Color.RED);
```

- Changer une association existante:

```
favoriteColors.put("Juliet",Color.BLUE);
```

Utiliser une carte, `Map`

- Accéder la valeur associée avec une clé:

```
Color julietsFavoriteColor =
   favoriteColors.get("Juliet”);
```

- Supprimer la clé associée avec une certaine valeur:

```
favoriteColors.remove("Juliet");
```

Imprimer les paires clé/valeur

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + ” : " + value);
}
```

ch16/map/MapDemo.java

```
 1  import java.awt.Color;
 2  import java.util.HashMap;
 3  import java.util.Map;
 4  import java.util.Set;
 5
 6  /**
 7     This program demonstrates a map that maps names to colors.
 8  */
 9  public class MapDemo
10  {
11     public static void main(String[] args)
12     {
13        Map<String, Color> favoriteColors = new HashMap<String,
Color>();
14        favoriteColors.put("Juliet", Color.BLUE);
15        favoriteColors.put("Romeo", Color.GREEN);
16        favoriteColors.put("Adam", Color.RED);
17        favoriteColors.put("Eve", Color.BLUE);
18
```

Continued

ch16/map/MapDemo.java (cont.)

```
19          // Print all keys and values in the map
20
21          Set<String> keySet = favoriteColors.keySet();
22          for (String key : keySet)
23          {
24             Color value = favoriteColors.get(key);
25             System.out.println(key + " : " + value);
26          }
27       }
28    }
```

Program Run:

```
Romeo : java.awt.Color[r=0,g=255,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Juliet : java.awt.Color[r=0,g=0,b=255]
```

Continued

Tables de hachage

- **Hachage** peut être utilisé pour retrouver un élément dans une structure de données rapidement sans faire la recherche linéaire
- Une table de hachage peut être utilisée pour implémenter des ensembles et des cartes
- La **fonction de hachage** calcule un nombre entier, appelé code de hachage, pour chacun des éléments
- La bonne fonction d'hachage minimise les collisions — les codes de hachage identiques pour les objets différents
- Pour calculer le code de hachage de l'objet `x`:

```
int h = x.hashCode();
```

Codes de hachage résultant de la fonction `hachCode`

String	Hash Code
"Adam"	2035631
"Eve"	700068
"Harry"	69496448
"Jim"	74478
"Joe"	74656
"Juliet"	-2065036585
"Katherine"	2079199209
"Sue"	83491

Implémentation simpliste d'une table de hachage

- Pour implémenter
  - *Générer les codes de hachage des objets*
  - *Créer un tableau*
  - *Insérer chaque objet dans une position correspondante à son code de hachage*
- Pour tester si l'objet est présent dans un ensemble
  - *Calculer son code de hachage*
  - *Vérifier si la position correspondante à ce code de hachage est déjà occupée*

Implémentation simpliste d'une table de hachage

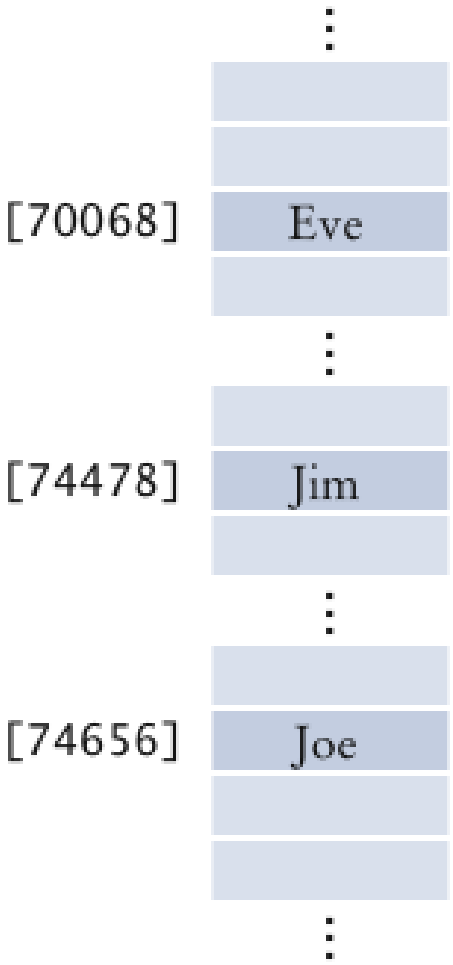


Figure 5
A Simplistic Implementation of a Hash Table

Problèmes avec l'implémentation simpliste

- Il n'est pas possible d'allouer un tableau assez grand pour contenir tous les indexes entiers possibles
- Il est possible que deux objets différents pourront avoir le même code de hachage

Solutions

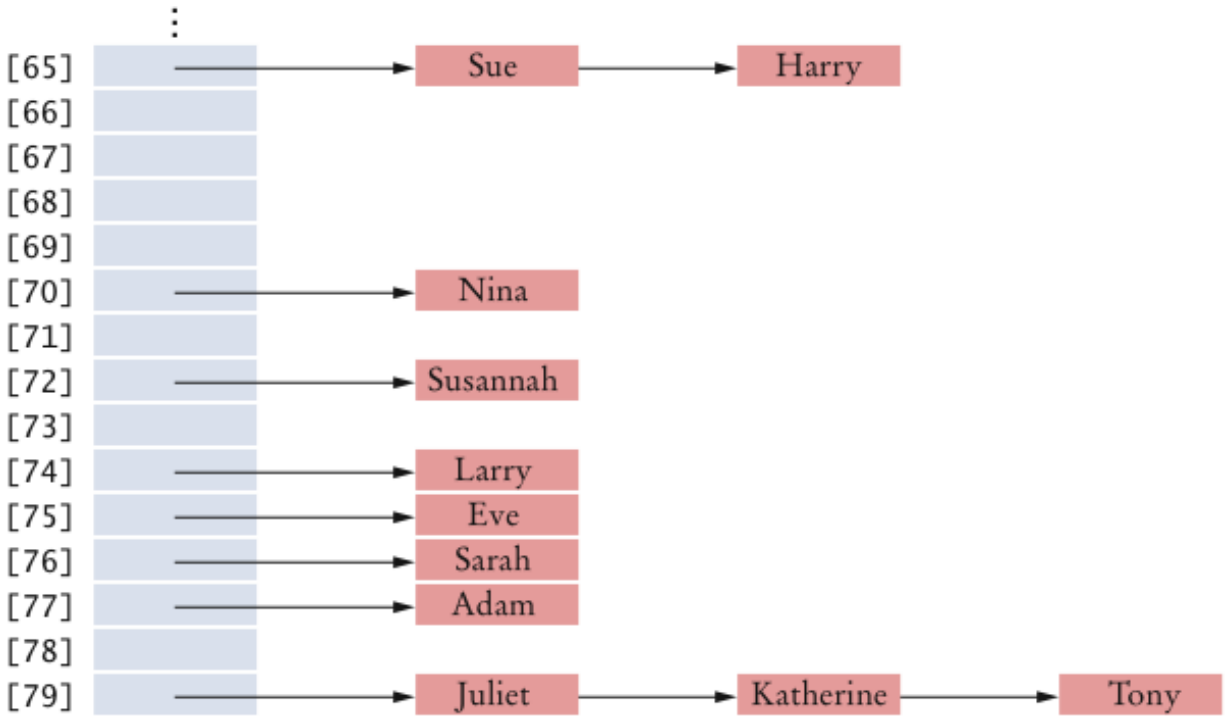
- Choisissez une taille raisonnable du tableau, réduisez le nombre de codes de hachage selon la taille du tableau

```
int h = x.hashCode();
if (h < 0) h = -h;
position = h % buckets.length;
```

- Lorsque les éléments ont le même code:
  - *Utiliser une séquence des nœuds pour stocker des nombreuses objets dans une même position du tableau*
  - *Ces séquences des nœuds sont appelées des paniers (buckets)*

Une table de hachage avec des paniers

Figure 6
A Hash Table with Buckets to Store Elements with the Same Hash Code



Algorithme pour retrouver l'objet `x` dans une table de hachage

1. Avoir un index `h` dans la table de hachage
    - *Calculer le code de hachage*
    - *Réduire ce code par un modulo du nombre total de paniers*
2. Itérer à travers des éléments du panier dans la position `h`
    - *Pour chaque élément du bucket, comparer s'il est égale à* `x`
3. Si on trouve l'égalité, donc `x` est dans l'ensemble
    - *Sinon,* `x` *n'est pas dans l'ensemble*

Tables de hachage

- Une table de hachage peut être implémentée comme un tableau des paniers
- Les paniers sont des séquences de nœuds contenants les éléments avec le même code de hachage
- S'il y a peu de collisions, l'addition suppression et recherche d'un élément prend le temps constant
  - *O(1)*
- Pour que cet algorithme soit efficace, la taille des paniers doit être petite
- La taille de la table doit être un nombre premier plus grand que le nombre d'éléments prévu
  - *Un excès de la capacité de 30% est typiquement recommandé*

Tables de hachage

- Ajouter un élément: extension simple de l'algorithme de recherche de l'objet
  - *Calculer le code de hachage pour localiser le panier où l'élément doit être inséré*
  - *Essayez de retrouver l'élément dans ce panier*
  - *Si l'élément est présent, faites rien, sinon, insérez-le*
- Supprimer un élément
  - *Calculer le code de hachage pour localiser le panier où l'élément doit être inséré*
  - *Essayez de retrouver l'élément dans ce panier*
  - *Si l'élément est présent, supprimez-le; sinon, faites rien*
- Peu de collisions, ajout et suppression prend *O*(1) temps

ch16/hashtable/HashSet.java

```
1  import java.util.AbstractSet;
2  import java.util.Iterator;
3  import java.util.NoSuchElementException;
4
5  /**
6     A hash set stores an unordered collection of objects, using
7     a hash table.
8  */
9  public class HashSet extends AbstractSet
10 {
11    private Node[] buckets;
12    private int size;
13
14    /**
15       Constructs a hash table.
16       @param bucketsLength the length of the buckets array
17    */
18    public HashSet(int bucketsLength)
19    {
20       buckets = new Node[bucketsLength];
21       size = 0;
22    }
23
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
24      /**
25         Tests for set membership.
26         @param x an object
27         @return true if x is an element of this set
28      */
29      public boolean contains(Object x)
30      {
31         int h = x.hashCode();
32         if (h < 0) h = -h;
33         h = h % buckets.length;
34
35         Node current = buckets[h];
36         while (current != null)
37         {
38            if (current.data.equals(x)) return true;
39            current = current.next;
40         }
41         return false;
42      }
43
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
44      /**
45         Adds an element to this set.
46         @param x an object
47         @return true if x is a new object, false if x was
48         already in the set
49      */
50      public boolean add(Object x)
51      {
52         int h = x.hashCode();
53         if (h < 0) h = -h;
54         h = h % buckets.length;
55
56         Node current = buckets[h];
57         while (current != null)
58         {
59            if (current.data.equals(x))
60               return false; // Already in the set
61            current = current.next;
62         }
63         Node newNode = new Node();
64         newNode.data = x;
65         newNode.next = buckets[h];
66         buckets[h] = newNode;
67         size++;
68         return true;
69      }
70
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
71      /**
72         Removes an object from this set.
73         @param x an object
74         @return true if x was removed from this set, false
75         if x was not an element of this set
76      */
77      public boolean remove(Object x)
78      {
79         int h = x.hashCode();
80         if (h < 0) h = -h;
81         h = h % buckets.length;
82
83         Node current = buckets[h];
84         Node previous = null;
85         while (current != null)
86         {
87            if (current.data.equals(x))
88            {
89               if (previous == null) buckets[h] = current.next;
90               else previous.next = current.next;
91               size--;
92               return true;
93            }
94            previous = current;
95            current = current.next;
96         }
97         return false;
98      }
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
100      /**
101         Returns an iterator that traverses the elements of this set.
102         @return a hash set iterator
103      */
104      public Iterator iterator()
105      {
106         return new HashSetIterator();
107      }
108
109      /**
110         Gets the number of elements in this set.
111         @return the number of elements
112      */
113      public int size()
114      {
115         return size;
116      }
117
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
118     class Node
119     {
120        public Object data;
121        public Node next;
122     }
123
124     class HashSetIterator implements Iterator
125     {
126        private int bucket;
127        private Node current;
128        private int previousBucket;
129        private Node previous;
130
131        /**
132           Constructs a hash set iterator that points to the
133           first element of the hash set.
134        */
135        public HashSetIterator()
136        {
137           current = null;
138           bucket = -1;
139           previous = null;
140           previousBucket = -1;
141        }
142
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
143         public boolean hasNext()
144         {
145            if (current != null && current.next != null)
146               return true;
147            for (int b = bucket + 1; b < buckets.length; b++)
148               if (buckets[b] != null) return true;
149            return false;
150         }
151
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
152         public Object next()
153         {
154            previous = current;
155            previousBucket = bucket;
156            if (current == null || current.next == null)
157            {
158               // Move to next bucket
159               bucket++;
160
161               while (bucket < buckets.length
162                     && buckets[bucket] == null)
163                  bucket++;
164               if (bucket < buckets.length)
165                  current = buckets[bucket];
166               else
167                  throw new NoSuchElementException();
168            }
169            else // Move to next element in bucket
170               current = current.next;
171            return current.data;
172         }
173
```

Continued

ch16/hashtable/HashSet.java (cont.)

```
174         public void remove()
175         {
176            if (previous != null && previous.next == current)
177               previous.next = current.next;
178            else if (previousBucket < bucket)
179               buckets[bucket] = current.next;
180            else
181               throw new IllegalStateException();
182            current = previous;
183            bucket = previousBucket;
184         }
185      }
186   }
```

Continued

ch16/hashtable/HashSetDemo.java

```
1  import java.util.Iterator;
2  import java.util.Set;
3
4  /**
5     This program demonstrates the hash set class.
6  */
7  public class HashSetDemo
8  {
9     public static void main(String[] args)
10    {
11       Set names = new HashSet(101); // 101 is a prime
12
13       names.add("Harry");
14       names.add("Sue");
15       names.add("Nina");
16       names.add("Susannah");
17       names.add("Larry");
18       names.add("Eve");
19       names.add("Sarah");
20       names.add("Adam");
21       names.add("Tony");
22       names.add("Katherine");
23       names.add("Juliet");
```

Continued

ch16/hashtable/HashSetDemo.java (cont.)

```
24          names.add("Romeo");
25          names.remove("Romeo");
26          names.remove("George");
27
28          Iterator iter = names.iterator();
29          while (iter.hasNext())
30             System.out.println(iter.next());
31       }
32    }
```

Program Run:

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

Calculer les codes de hachage

- La fonction de hachage calcule un entier à partir de l'objet
- Choisir une fonction de hachage de telle manière que les objets différents ont les codes de hachage plutôt différents
- Chaînes: Additionner les valeurs d'encodage Unicode des caractères
  - ```
    int h = 0;
    for (int i = 0; i < s.length(); i++)
          h = h + s.charAt(i);
    ```
  - Mauvaise choix pour une fonction de hachage des chaînes
    - *Permutations (`"eat«` et `"tea"`) auront le même code de hachage*
    ```

Calculer les codes de hachage

- La fonction de hachage pour une chaîne `s` de la bibliothèque standard

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = HASH_MULTIPLIER * h + s.charAt(i)
```

- Par exemple, le code de hachage de *"eat"* est

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

- Le code de *"tea"* est assez différent:

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

La méthode `hashCode` pour la classe `Coin`

- Deux variables d'instance: `String` – nom d'une monnaie et `double` - valeur
- Utilisons la méthode `hashCode` de `String` pour générer le code de hachage de la variable nom
- Pour le code de hachage d'un nombre en virgule flottante:
  - *Envelopper le nombre dans un objet* `Double`
  - *Utiliser la méthode* `hashCode` *de la classe* `Double`
- Combiner les deux codes de hachage en utilisant un nombre premier comme un multiplicateur (`HASH_MULTIPLIER`)

La méthode `hashCode` pour la classe `Coin`

```
class Coin
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(value).hashCode();
      final int HASH_MULTIPLIER = 29;
      int h = HASH_MULTIPLIER * h1 + h2;
      return h;
   }
   ...
}
```

Création des codes de hachage pour vos classes

- Utilisez le nombre premier comme `HASH_MULTIPLIER`
- Calculez les codes de hachage pour chaque champ d'instance
- Pour un champ d'instance de type entier utilisez la valeur de ce champ
- Combinez les codes de hachage:

```
int h = HASH_MULTIPLIER * h1 + h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER *h + h4;
...
return h;
```

Création des codes de hachage pour vos classes

- Votre méthode `hashCode` doit être compatible avec la méthode `equals`
  - *si* `x.equals(y)`, *donc* `x.hashCode() == y.hashCode()`
- Vous aurez de problèmes si votre classe définit la méthode `equals` mais pas la méthode `hashCode`
  - *Si on oubli de définir la méthode* `hashCode` *pour* `Coin`, *cette méthode sera héritée de la super classe* `Object`
  - *Cette méthode calcule un code de hachage à partir de l'adresse mémoire de l'objet*
  - *Effet: N'import quels deux objets auront fort probable les codes de hachage différents*

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

Création des codes de hachage pour vos classes

- En général, définissez les deux méthodes , `hashCode` et `equals` ou rien

Hash Maps

- Dans l'implémentation des cartes avec une table de hachage seulement les clés sont « hachées »
- Les clés nécessitent les méthodes `hashCode` et `equals` compatibles

ch16/hashcode/Coin.java

```
1  /**
2     A coin with a monetary value.
3  */
4  public class Coin
5  {
6     private double value;
7     private String name;
8
9     /**
10       Constructs a coin.
11       @param aValue the monetary value of the coin.
12       @param aName the name of the coin
13    */
14    public Coin(double aValue, String aName)
15    {
16       value = aValue;
17       name = aName;
18    }
19
```

Continued

ch16/hashcode/Coin.java (cont.)

```
20      /**
21         Gets the coin value.
22         @return the value
23      */
24      public double getValue()
25      {
26         return value;
27      }
28
29      /**
30         Gets the coin name.
31         @return the name
32      */
33      public String getName()
34      {
35         return name;
36      }
37
```

Continued

ch16/hashcode/Coin.java (cont.)

```
38      public boolean equals(Object otherObject)
39      {
40         if (otherObject == null) return false;
41         if (getClass() != otherObject.getClass()) return false;
42         Coin other = (Coin) otherObject;
43         return value == other.value && name.equals(other.name);
44      }
45
46      public int hashCode()
47      {
48         int h1 = name.hashCode();
49         int h2 = new Double(value).hashCode();
50         final int HASH_MULTIPLIER = 29;
51         int h = HASH_MULTIPLIER * h1 + h2;
52         return h;
53      }
54
55      public String toString()
56      {
57         return "Coin[value=" + value + ",name=" + name + "]";
58      }
59   }
```

ch16/hashcode/CoinHashCodePrinter.java

```
 1  import java.util.HashSet;
 2  import java.util.Set;
 3
 4  /**
 5     A program that prints hash codes of coins.
 6  */
 7  public class CoinHashCodePrinter
 8  {
 9     public static void main(String[] args)
10     {
11        Coin coin1 = new Coin(0.25, "quarter");
12        Coin coin2 = new Coin(0.25, "quarter");
13        Coin coin3 = new Coin(0.05, "nickel");
14
15        System.out.println("hash code of coin1=" + coin1.hashCode());
16        System.out.println("hash code of coin2=" + coin2.hashCode());
17        System.out.println("hash code of coin3=" + coin3.hashCode());
18
19        Set<Coin> coins = new HashSet<Coin>();
20        coins.add(coin1);
21        coins.add(coin2);
22        coins.add(coin3);
23
```

Continued

ch16/hashcode/CoinHashCodePrinter.java (cont.)

```
24         for (Coin c : coins)
25            System.out.println(c);
26      }
27   }
```

Program Run:

```
hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-1768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name=nickel]
```

Arbres de recherche binaire

- Les arbres de recherche binaire permettent l'insertion et suppression rapide des éléments
- Sont développés pour la recherche rapide
- **Une arbre binaire** contient deux nœuds, chacun de ces deux nœuds a deux nœuds etc.
- Tous les nœuds dans l'arbre respectent la propriété suivante:
  - *Les descendants à gauche ont les données plus petites que la donnée de son parent*
  - *Les descendants à droite ont les données plus grandes que la donnée de son parent*

Arbres binaire de recherche

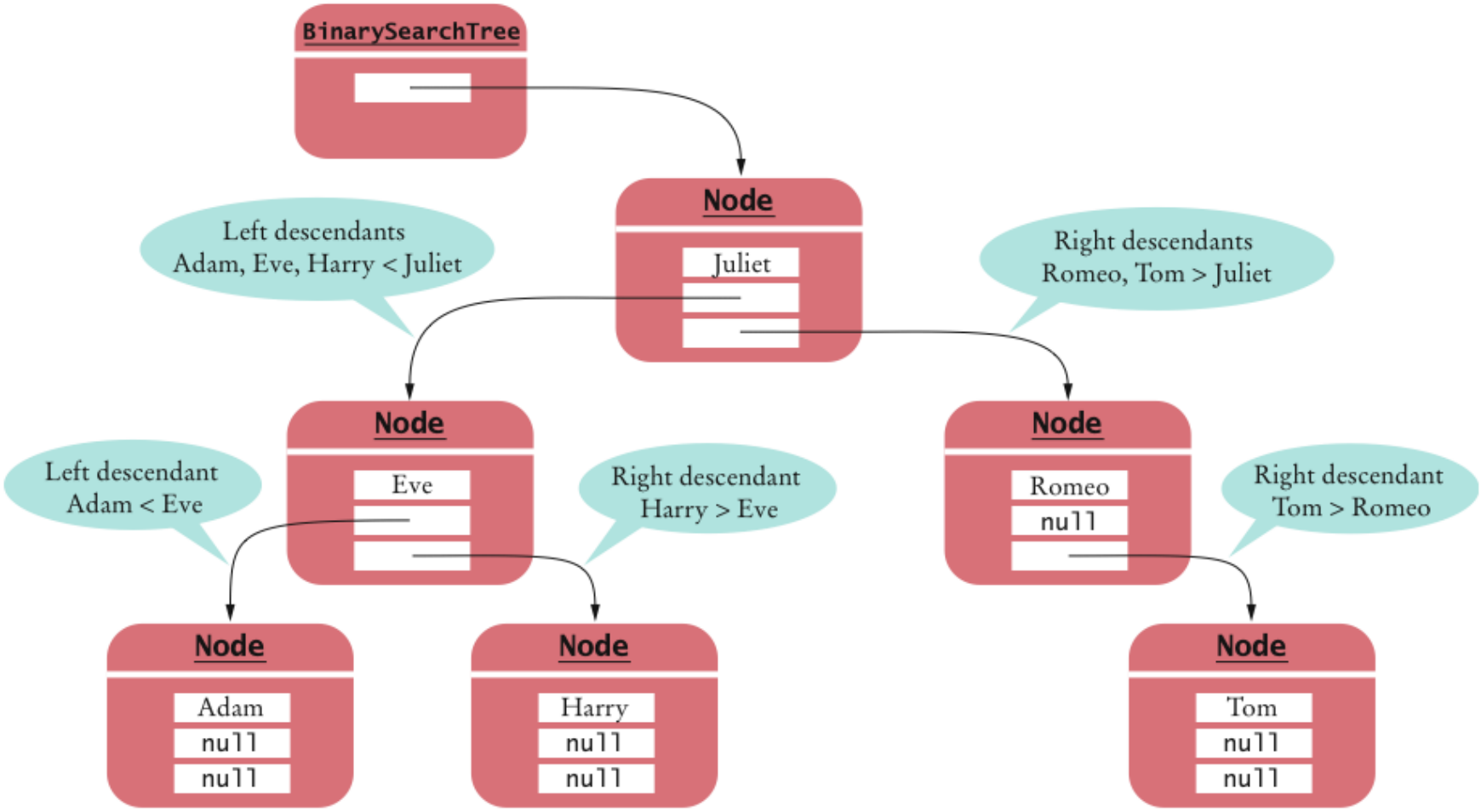


Figure 7 A Binary Search Tree

Arbre Binaire

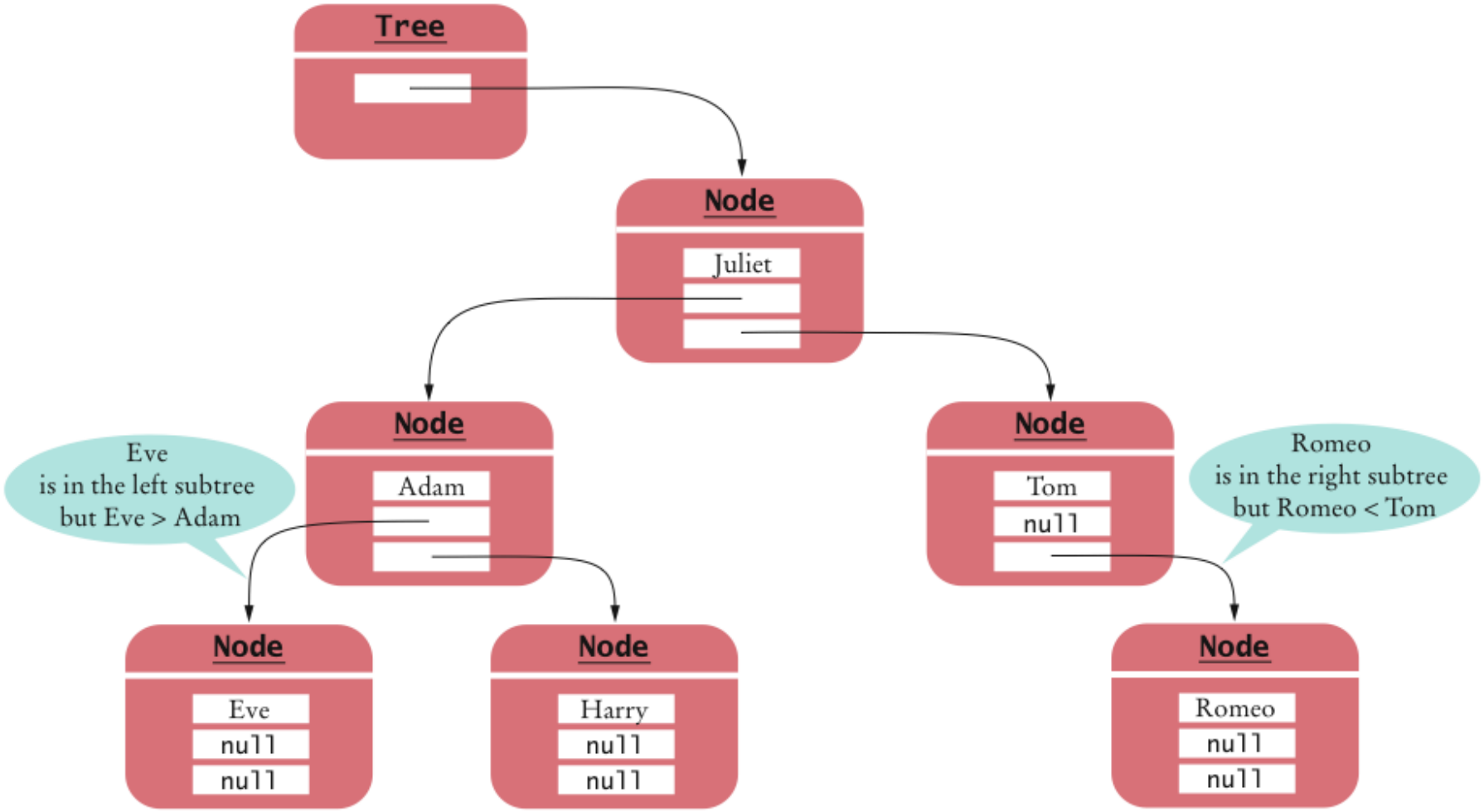


Figure 8 A Binary Tree That Is Not a Binary Search Tree

Implémentation de l'arbre binaire de recherche

- Implémentez la classe d'un arbre contenant une référence sur le nœud racine
- Implémentez la classe pour les nœuds
  - *Un nœud contient deux liens (vers les nouds enfants, gauche et droite)*
  - *Le nœud contient un champ de donnée*
  - *La donnée est de type* `Comparable`*, pour qu'on puisse comparer les valeurs pour mettre les nœuds dans les positions correctes dans l'arbre*

Implémentation de l'arbre binaire de recherche

```
public class BinarySearchTree
{
   private Node root;

   public BinarySearchTree() { ... }
   public void add(Comparable obj) { ... }
   ...
   private class Node
   {
      public Comparable data;
      public Node left;
      public Node right;

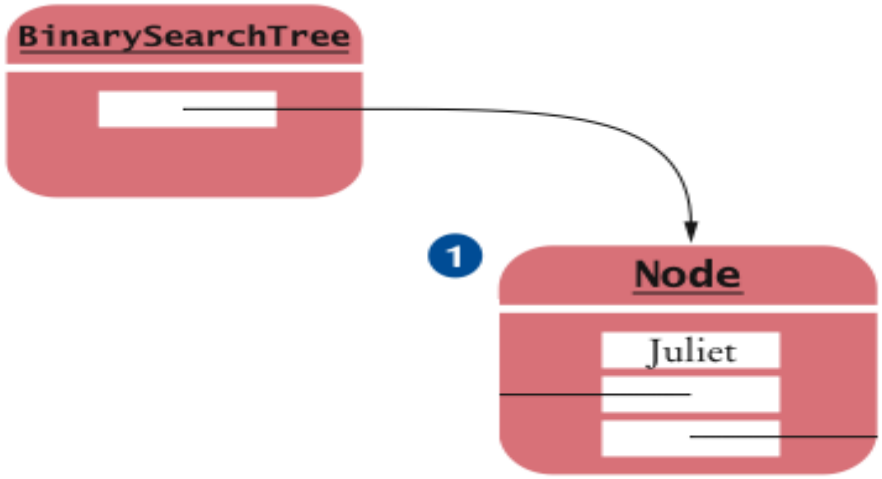
      public void addNode(Node newNode) { ... }
      ...
   }
}
```

Algorithme d'insertion

- Lorsque vous rencontrez la référence vers le nœud non nul, testez sa donnée
  - *Si* `data` *de ce nœud est plus grande que celle à insérer continuez le processus d'insertion dans le sous arbre gauche*
  - *Dans le cas contraire, continuez avec le sous arbre droite*
- Si vous rencontrez une référence vers le nœud nul, remplacez cette référence par le nouveau nœud

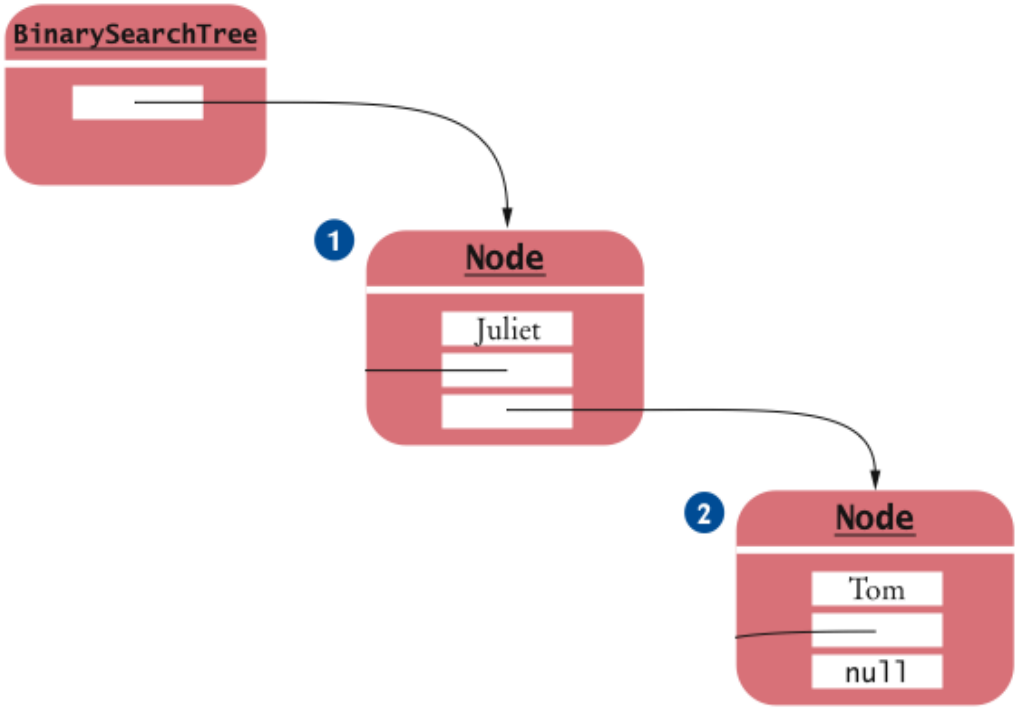
Exemple

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");  1
```



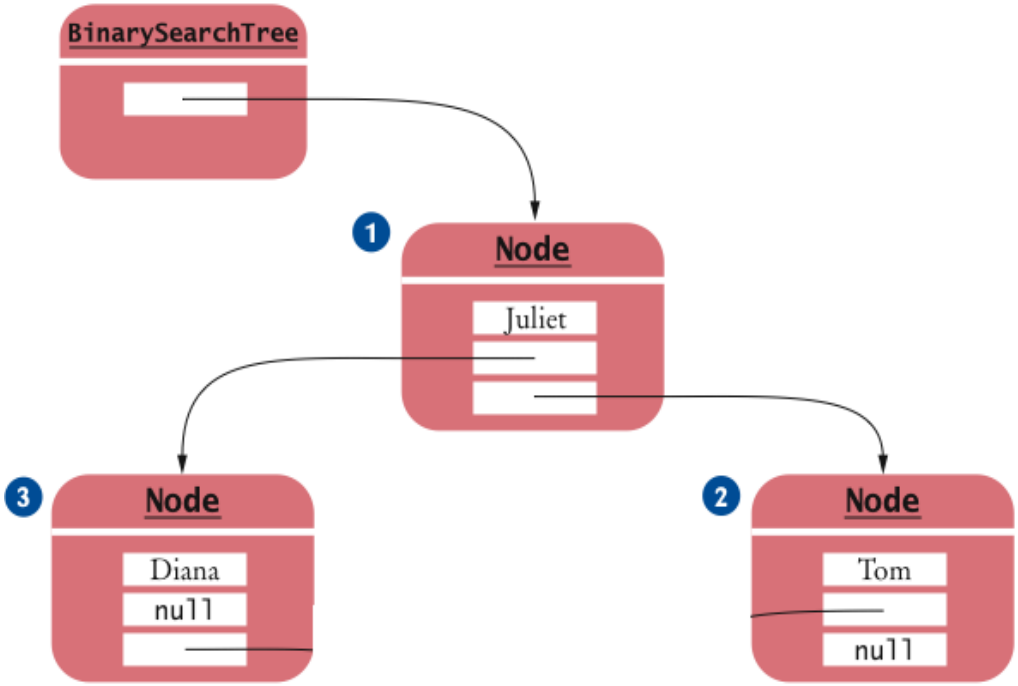
Exemple

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");
tree.add("Tom");
```



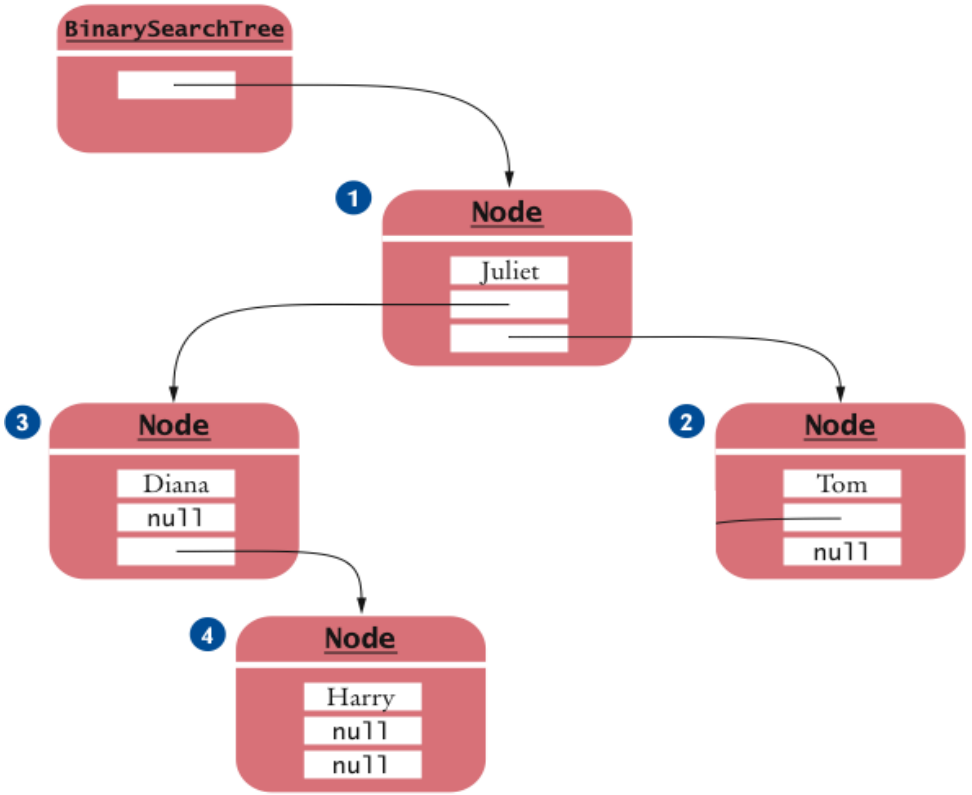
Exemple

- ```
  BinarySearchTree tree = new BinarySearchTree();
  tree.add("Juliet");
  tree.add("Tom");
  tree.add("Diana");
  ```


  ```

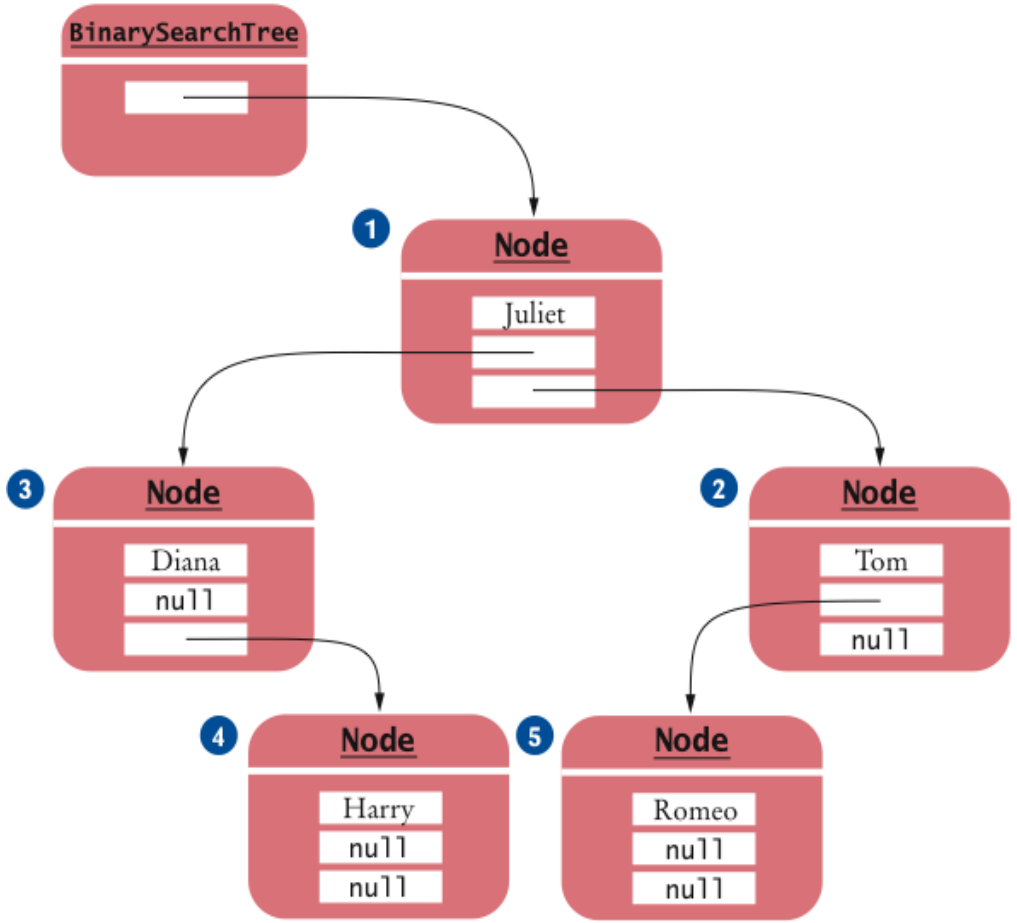
Exemple

- ```
  BinarySearchTree tree = new BinarySearchTree();
  tree.add("Juliet");
  tree.add("Tom");
  tree.add(”Diana");
  tree.add("Harry");
  ```


  ```

Exemple

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");
tree.add("Tom");
tree.add(”Diana");
tree.add("Harry");
tree.add("Romeo");
```



Méthode `add` de la classe `BinarySearchTree`

```
public void add(Comparable obj)
{
   Node newNode = new Node();
   newNode.data = obj;
   newNode.left = null;
   newNode.right = null;
   if (root == null) root = newNode;
   else root.addNode(newNode);
}
```

Méthode `addNode` de la classe `Node`

```
private class Node
{
   ...
   public void addNode(Node newNode)
   {
      int comp = newNode.data.compareTo(data);
      if (comp < 0)
      {
         if (left == null) left = newNode;
         else left.addNode(newNode);
      }
   else if (comp > 0)
   {
      if (right == null) right = newNode;
      else right.addNode(newNode);
      }
   }
   ...
}
```

Supprimer le nœud avec un seul enfant

- Lorsqu'on supprime un nœud avec un seul enfant, l'enfant remplace le nœud à supprimer

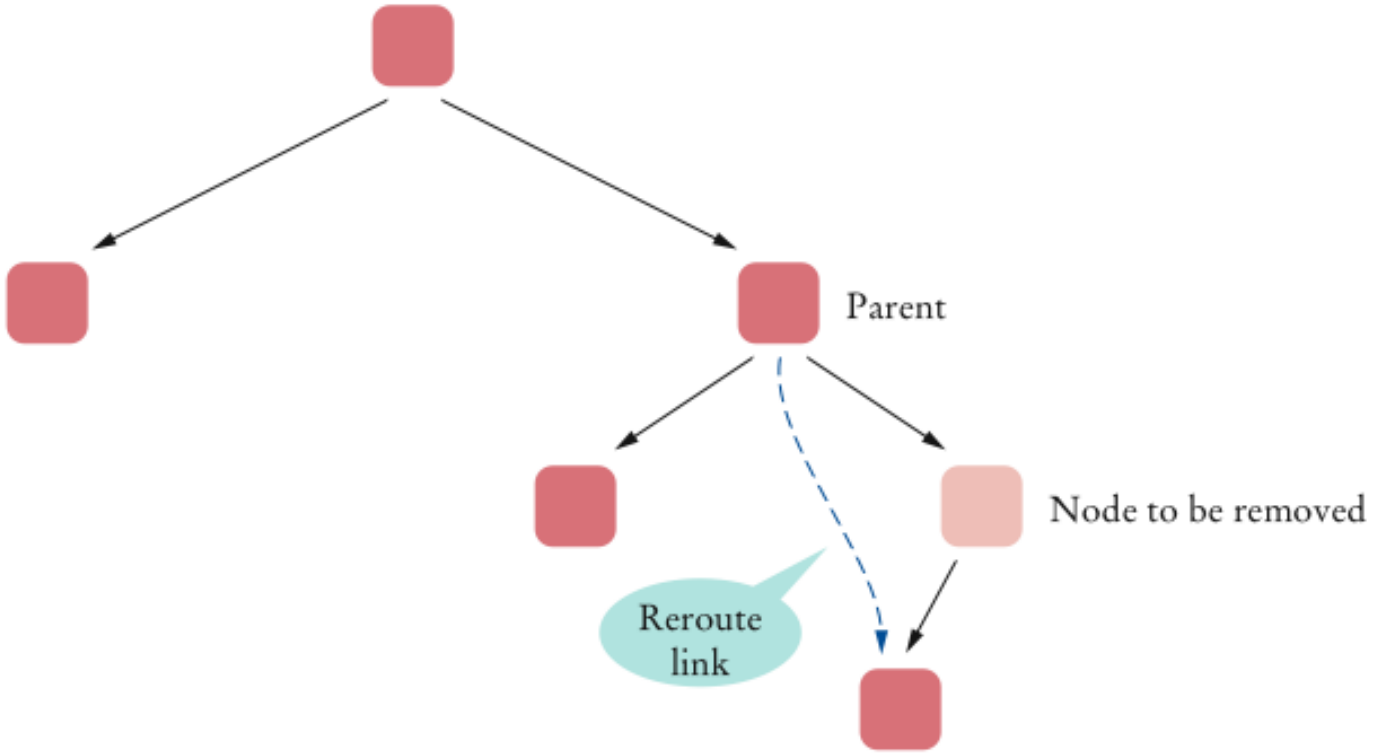


Figure 11
Removing a Node with One Child

Supprimer le nœud avec deux enfants

- Supprimer un nœud avec deux enfants – remplacer ce nœud avec le plus petit nœud de son sous arbre droit

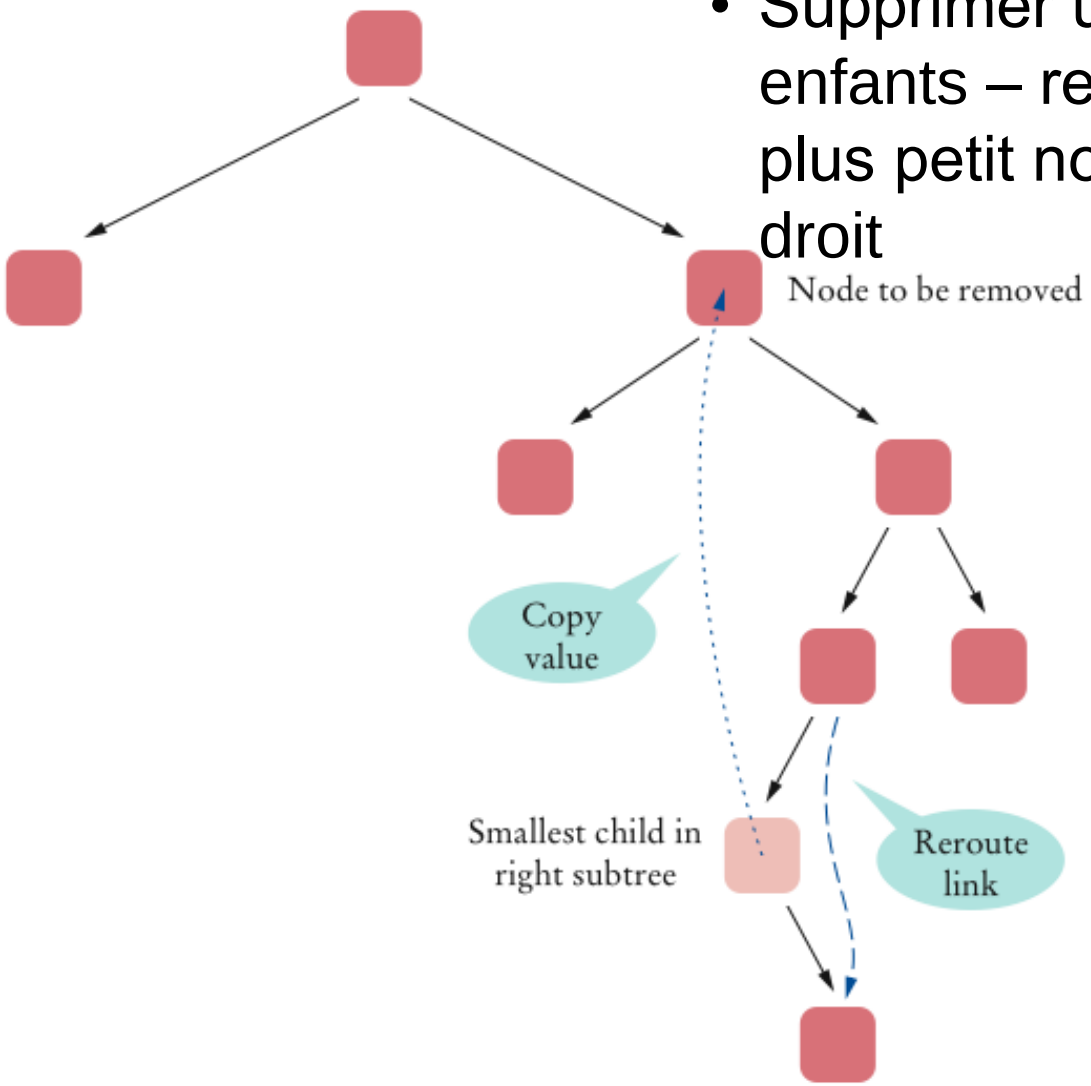


Figure 12 Removing a Node with Two Children

Arbres binaire de recherche

- **Un arbre balancé:** chaque nœud a approximativement le même nombre de descendantes à gauche qu'à droite
- Si l'arbre est balancé, ajoutez un élément nécessite le temps $O(\log(n))$
- Si l'arbre est débalancé, l'insertion pourra être lente
  - *Pire cas – aussi lente que l'insertion dans la liste chaînée*

Arbres balancés et non balancés

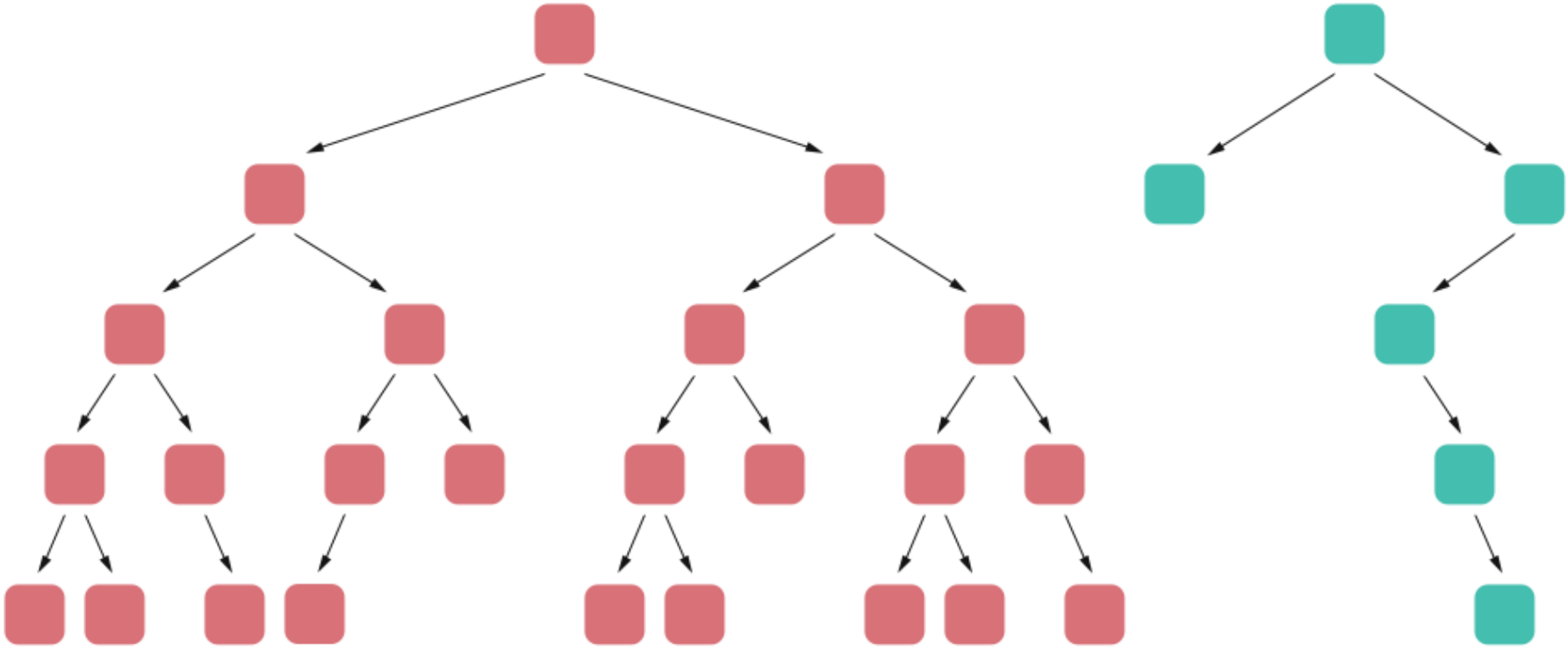


Figure 13 Balanced and Unbalanced Trees

ch16/tree/BinarySearchTree.java

```
1  /**
2     This class implements a binary search tree whose
3     nodes hold objects that implement the Comparable
4     interface.
5  */
6  public class BinarySearchTree
7  {
8     private Node root;
9
10    /**
11       Constructs an empty tree.
12    */
13    public BinarySearchTree()
14    {
15       root = null;
16    }
17
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
18      /**
19         Inserts a new node into the tree.
20         @param obj the object to insert
21      */
22      public void add(Comparable obj)
23      {
24         Node newNode = new Node();
25         newNode.data = obj;
26         newNode.left = null;
27         newNode.right = null;
28         if (root == null) root = newNode;
29         else root.addNode(newNode);
30      }
31
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
32      /**
33         Tries to find an object in the tree.
34         @param obj the object to find
35         @return true if the object is contained in the tree
36      */
37      public boolean find(Comparable obj)
38      {
39         Node current = root;
40         while (current != null)
41         {
42            int d = current.data.compareTo(obj);
43            if (d == 0) return true;
44            else if (d > 0) current = current.left;
45            else current = current.right;
46         }
47         return false;
48      }
49
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
50      /**
51         Tries to remove an object from the tree. Does nothing
52         if the object is not contained in the tree.
53         @param obj the object to remove
54      */
55      public void remove(Comparable obj)
56      {
57         // Find node to be removed
58
59         Node toBeRemoved = root;
60         Node parent = null;
61         boolean found = false;
62         while (!found && toBeRemoved != null)
63         {
64            int d = toBeRemoved.data.compareTo(obj);
65            if (d == 0) found = true;
66            else
67            {
68               parent = toBeRemoved;
69               if (d > 0) toBeRemoved = toBeRemoved.left;
70               else toBeRemoved = toBeRemoved.right;
71            }
72         }
73
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
74          if (!found) return;
75
76          // toBeRemoved contains obj
77
78          // If one of the children is empty, use the other
79
80          if (toBeRemoved.left == null || toBeRemoved.right == null)
81          {
82             Node newChild;
83             if (toBeRemoved.left == null)
84                newChild = toBeRemoved.right;
85             else
86                newChild = toBeRemoved.left;
87
88             if (parent == null) // Found in root
89                root = newChild;
90             else if (parent.left == toBeRemoved)
91                parent.left = newChild;
92             else
93                parent.right = newChild;
94             return;
95          }
96
97          // Neither subtree is empty
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
 98
 99          // Find smallest element of the right subtree
100
101          Node smallestParent = toBeRemoved;
102          Node smallest = toBeRemoved.right;
103          while (smallest.left != null)
104          {
105             smallestParent = smallest;
106             smallest = smallest.left;
107          }
108
109          // smallest contains smallest child in right subtree
110
111          // Move contents, unlink child
112
113          toBeRemoved.data = smallest.data;
114          if (smallestParent == toBeRemoved)
115             smallestParent.right = smallest.right;
116          else
117             smallestParent.left = smallest.right;
118       }
119
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
120     /**
121        Prints the contents of the tree in sorted order.
122     */
123     public void print()
124     {
125        if (root != null)
126           root.printNodes();
127        System.out.println();
128     }
129
130     /**
131        A node of a tree stores a data item and references
132        of the child nodes to the left and to the right.
133     */
134     class Node
135     {
136        public Comparable data;
137        public Node left;
138        public Node right;
139
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
140         /**
141            Inserts a new node as a descendant of this node.
142            @param newNode the node to insert
143         */
144         public void addNode(Node newNode)
145         {
146            int comp = newNode.data.compareTo(data);
147            if (comp < 0)
148            {
149               if (left == null) left = newNode;
150               else left.addNode(newNode);
151            }
152            else if (comp > 0)
153            {
154               if (right == null) right = newNode;
155               else right.addNode(newNode);
156            }
157         }
158
```

Continued

ch16/tree/BinarySearchTree.java (cont.)

```
159          /**
160             Prints this node and all of its descendants
161             in sorted order.
162          */
163          public void printNodes()
164          {
165             if (left != null)
166                left.printNodes();
167             System.out.print(data + " ");
168             if (right != null)
169                right.printNodes();
170          }
171       }
172    }
```

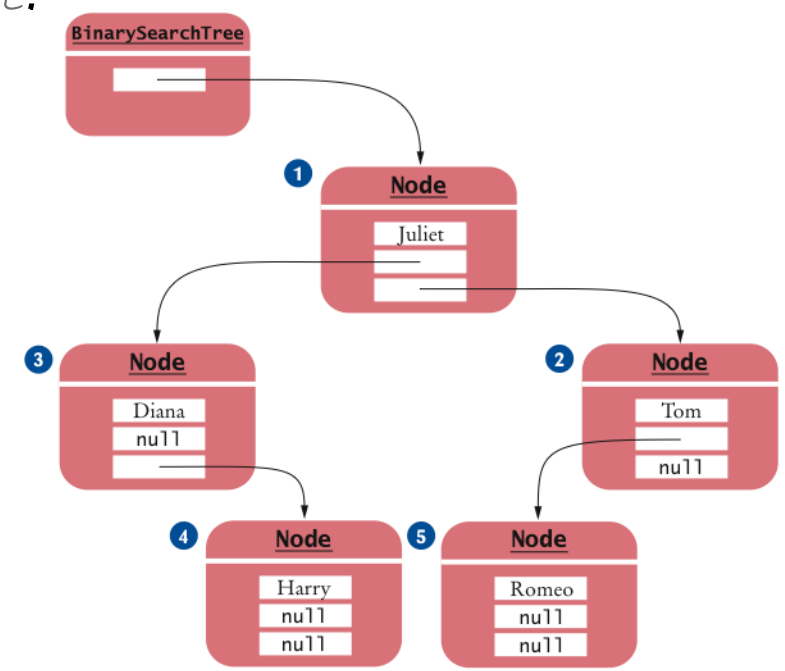
Parcours d'un arbre binaire

- Imprimer les éléments d'arbre en ordre :
  1. *Imprimer le sous arbre gauche*
  2. *Imprimer la donnée du nœud*
  3. *Imprimer le sous arbre droite*

Exemple

- Considérons l'arbre de la figure. L'algorithme dit:
  1. *Imprimer le sous arbre gauche de* `Juliet`*; c'est,* `Diana` *et descendants*
  2. *Imprimer* `Juliet`
  3. *Imprimer le sous arbre droite de* `Juliet`*:*

  c'est, `Tom` *et descendants*



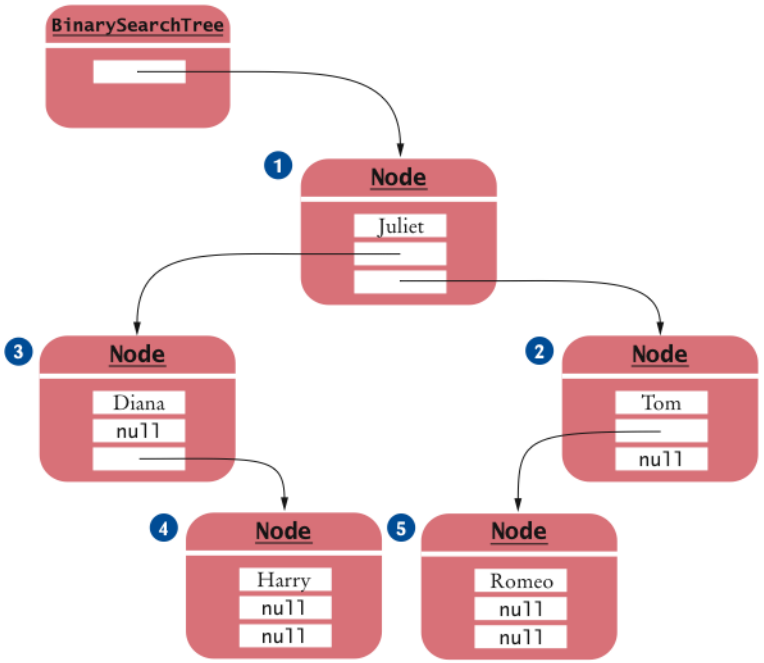
- Comment imprimer le sous arbre commençant sur `Diana`?
  1. *Imprimer le sous arbre gauche de* `Diana` *– rien à imprimer*
  1. *Imprimer* `Diana`
  2. *Imprimer le sous arbre droite de* `Diana`*, c'est,* `Harry`

Exemple

- Algorithme continue comme décrit
- **Sortie:**

```
Diana Harry Juliet Romeo Tom
```

- L'arbre est imprimé en ordre



La méthode `printNodes` de la classe Node

```
private class Node
{
   ...
   public void printNodes()
   {
      if (left != null)
         left.printNodes();
      System.out.println(data);
      if (right != null)
         right.printNodes();
   }
   ...
}
```

Méthode `print` de la classe `BinarySearchTree`

Pour imprimer l'arbre entier, commencez le processus récursif d'impression à partir de la racine :

```
public class BinarySearchTree
{
   ...
   public void print()
   {
      if (root != null)
         root.printNodes();
      System.out.println();
   }
   ...
}
```

Parcours d'arbre

- Trois schèmes de parcours
  - *Préfixe*
  - *Infixe*
  - *Postfixe*

Parcours préfixe

- Visiter la racine
- Visiter le sous arbre gauche
- Visiter le sous arbre droite

Parcours infixe

- Visiter le sous arbre gauche
- Visiter la racine
- Visiter le sous arbre droite

Parcours postfixe

- Visiter le sous arbre gauche
- Visiter le sous arbre droite
- Visiter la racine

Queues de priorité

- **Une queue de priorité** collectionne les éléments possédants les priorités
- Exemple: Collection des requêtes de travail qui pourront être de niveaux d'urgence différents
- Lorsqu'on retire un élément, c'est l'élément de la plus haute priorité est retiré
  - *D'habitude on assigne les petites valeurs aux hautes priorités (1 signifie la plus haute priorité)*
- La bibliothèque standard Java fournit la classe `PriorityQueue`
- La structure de données appelée tas (*heap) est très appropriée pour implémenter les queues de priorité*

Exemple

- Considérons le code suivant:

```
PriorityQueue<WorkOrder> q =
      new PriorityQueue<WorkOrder>;
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix overflowing sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

- En appelant `q.remove()` la première fois, le travail avec la priorité 1 est retiré
- Prochaine appel à `q.remove()` retire la requête avec la priorité 2

Tas (Heaps)

- Un tas (*min-heap*) est une arborescence binaire dans laquelle les opérations `add` et `remove` amènent l'élément le plus petit à graviter autour de la racine, sans perdre du temps à trier tous les éléments
- Les propriétés d'un tas
  1. *Presque complet*
     - Tous les nœuds sont remplis sauf au dernier niveau à droite
  2. *L'arbre satisfait la propriété du tas*
     - Tous les nœuds stockent les valeurs presque aussi grandes que ses descendants
- La propriété du tas garanti que le plus petit élément est placé dans la racine

Un arbre binaire presque complet

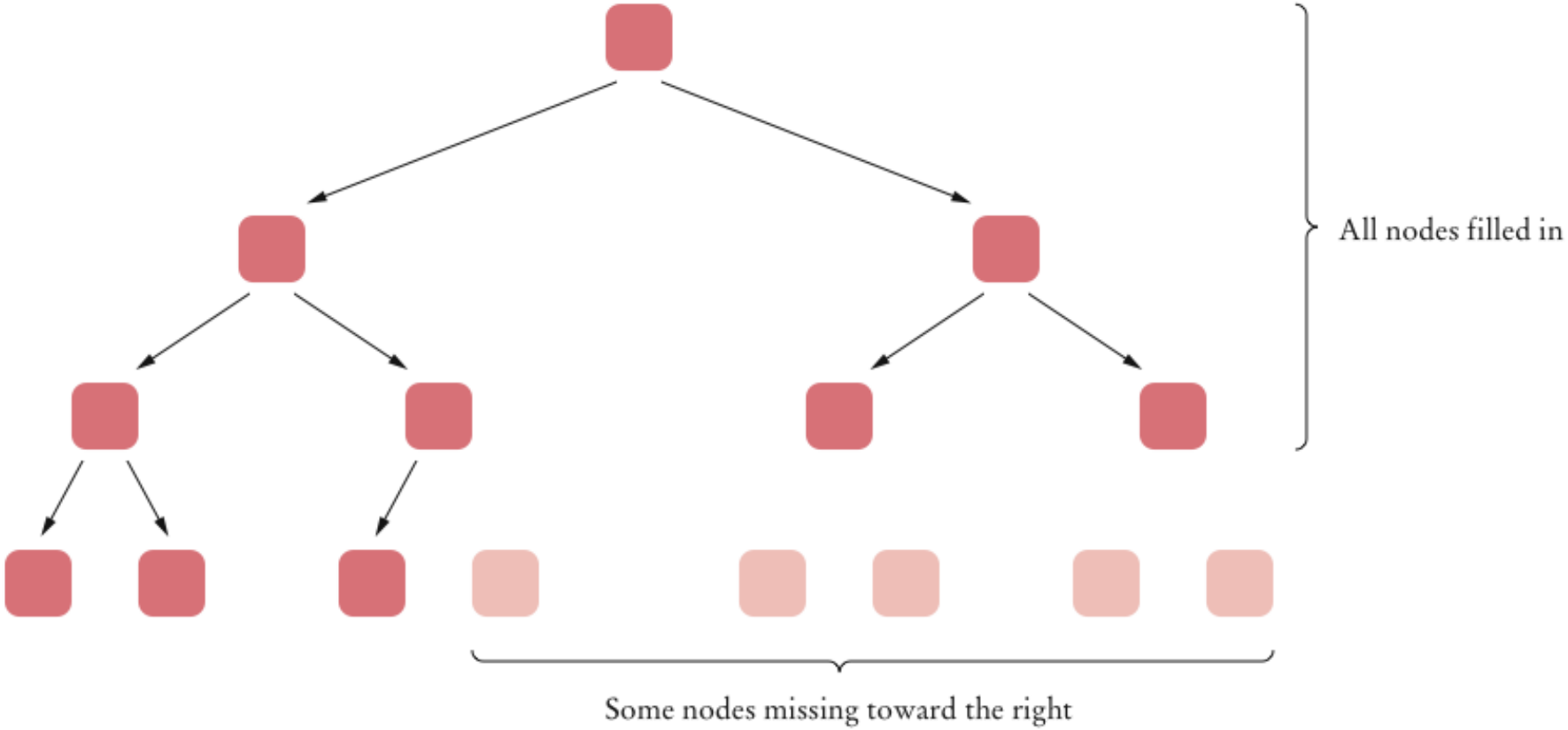
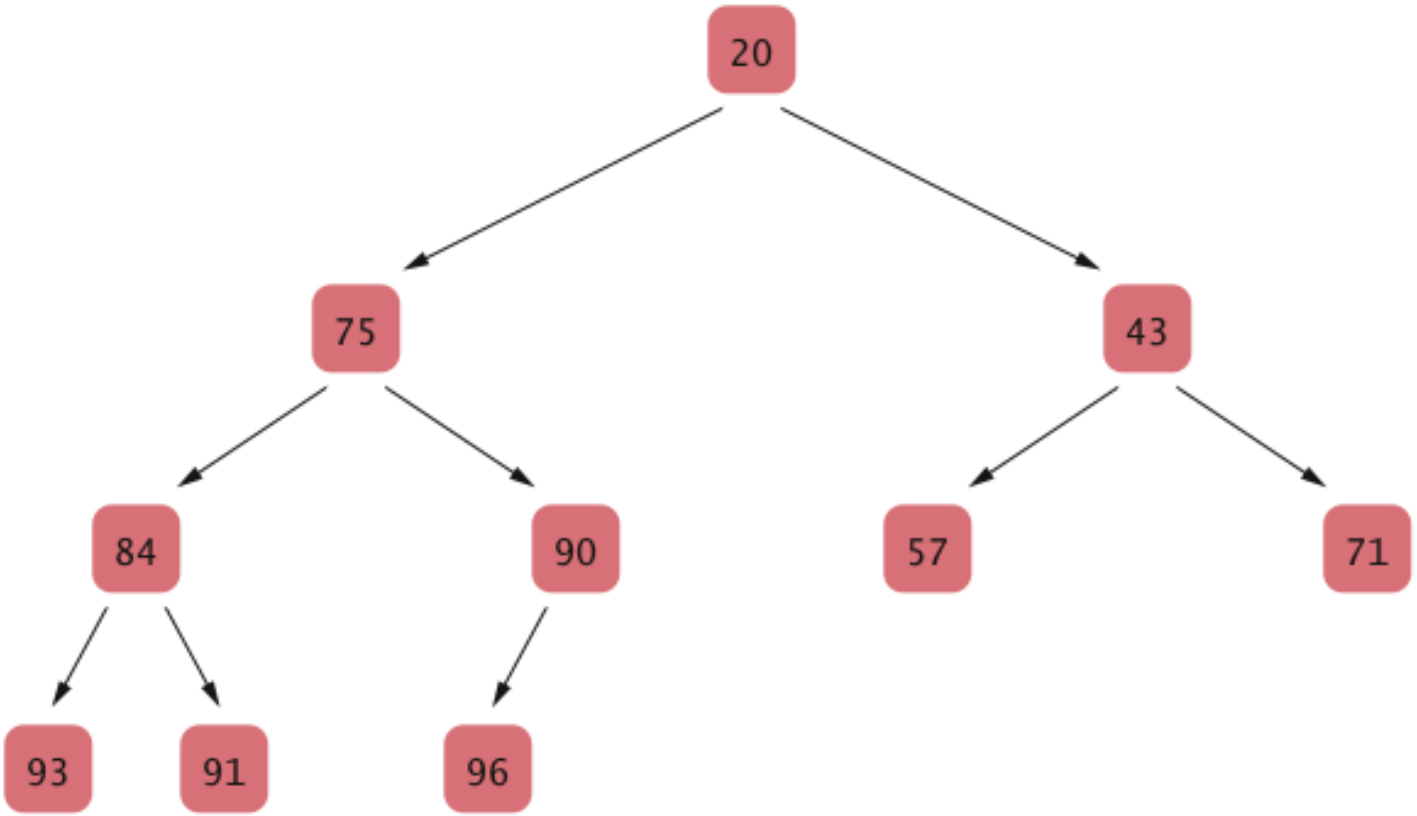


Figure 15 An Almost Completely Filled Tree

Un tas

Figure 16
A Heap



Différences entre un tas et un arbre de recherche binaire

- La forme d'un tas est très régulière
  - *Les arbres de recherche binaire peuvent avoir des formes arbitraires*
- Dans le tas le sous arbre gauche et le sous arbre droite stockent les éléments plus grands que la racine
  - *Dans l'arbre de recherche binaire les éléments les plus petits que la racine sont stockés dans le sous arbre gauche et les plus grands dans le sous arbre droit*

Insérer un nouveau élément dans le tas

1. Ajouter un slot à la fin de l'arbre

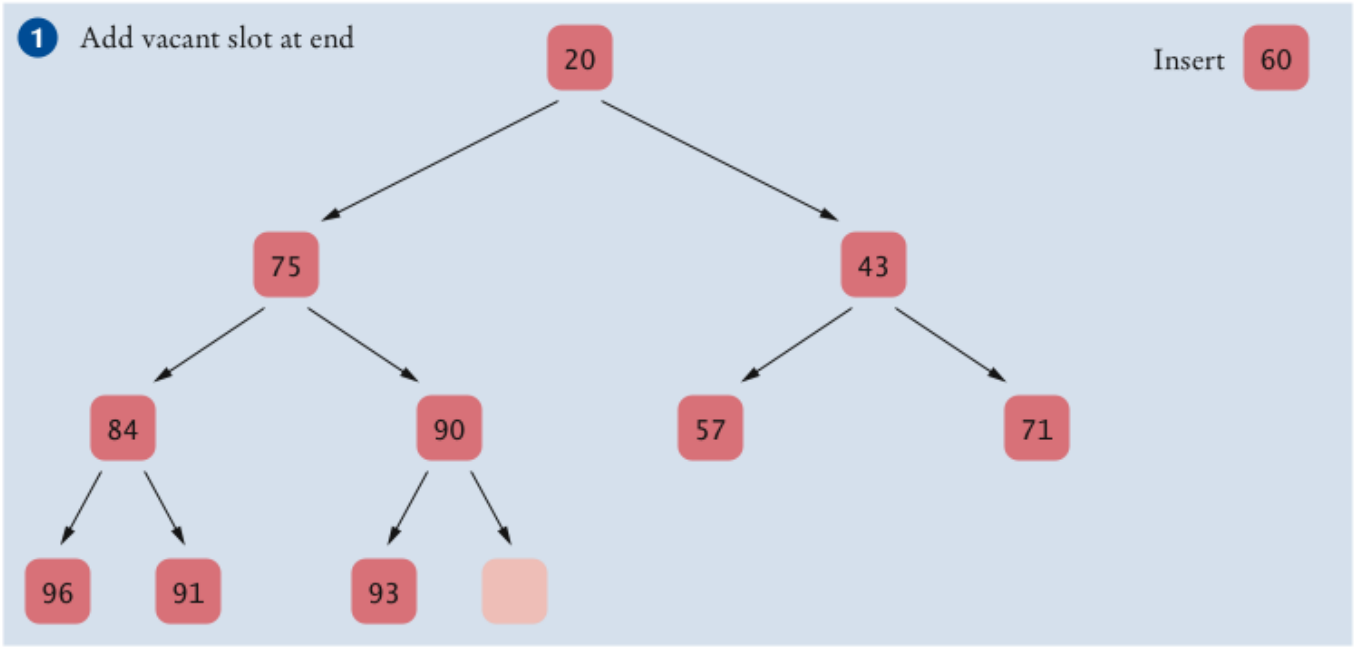


Figure 17 Inserting an Element into a Heap

Continued

Insérer un nouveau élément dans le tas

2. Descendre le parent du slot vide s'il est plus grand que la valeur à insérer
   - *Descendre le parent dans le slot vide et monter le slot vide à la place du parent*
   - *Répéter cette procédure jusqu'à ce que le parent du slot vide ne soit plus grand que l'élément à insérer*

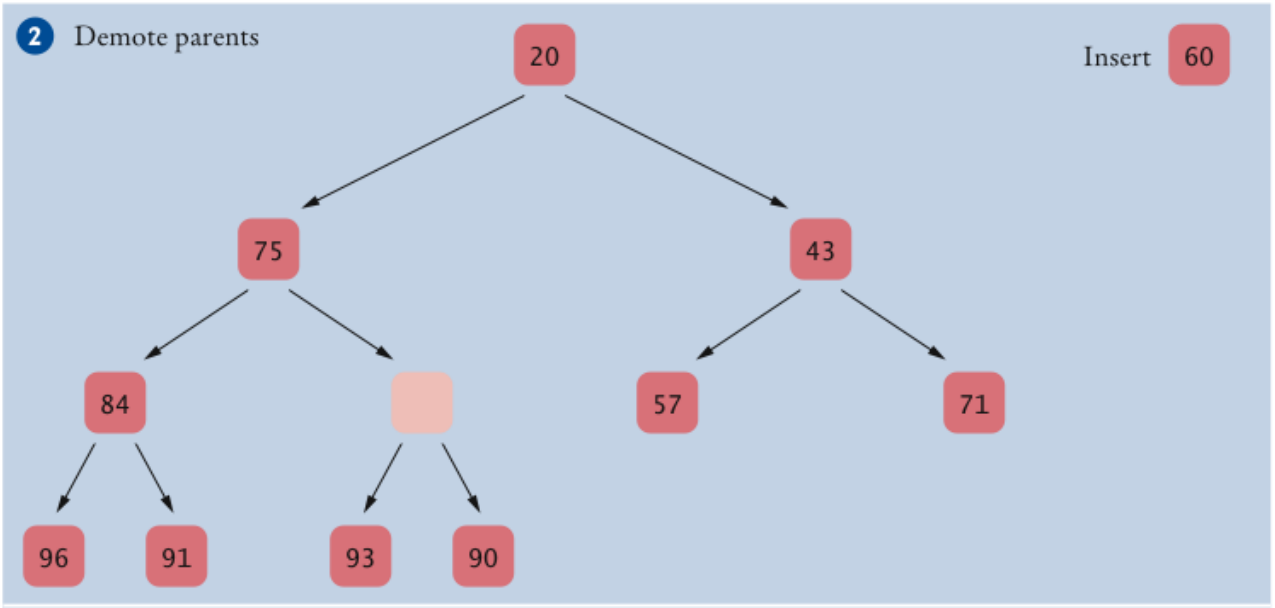


Figure 17 (continued) Inserting an Element into a Heap

Continued

Insérer un nouveau élément dans le tas

2. Descendre le parent du slot vide s'il est plus grand que la valeur à insérer
   - *Descendre le parent dans le slot vide et monter le slot vide à la place du parent*
   - *Répéter cette procédure jusqu'à ce que le parent du slot vide ne soit plus grand que l'élément à insérer*

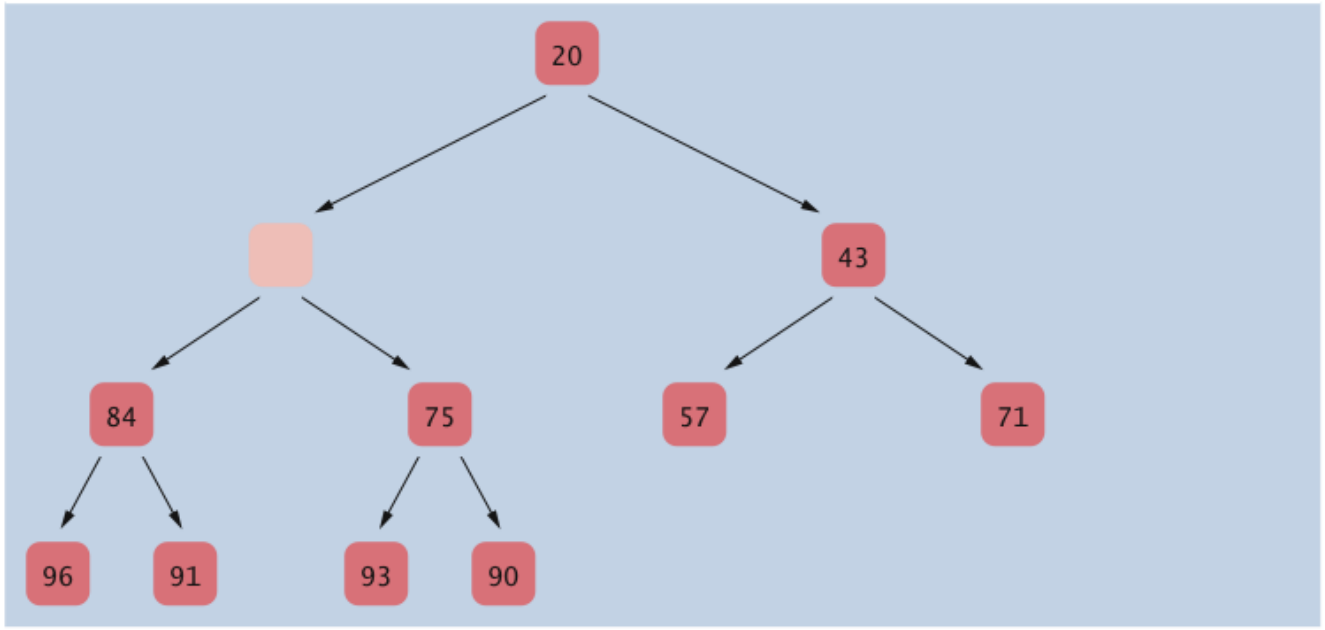


Figure 17 (continued) Inserting an Element into a Heap

Continued

Insérer un nouveau élément dans le tas

3. À ce point, soit le slot est dans la position racine ou le parent du slot est plus petit que la valeur à insérer. Insérer l'élément dans le slot

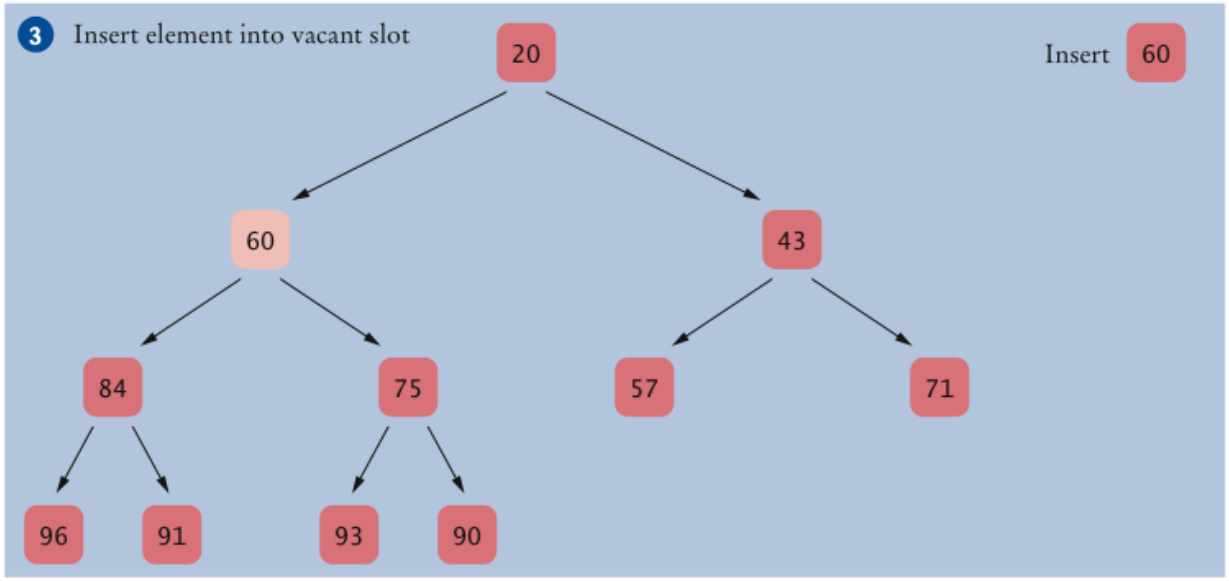


Figure 17 (continued) Inserting an Element into a Heap

Supprimer la racine du tas (Queue de priorité)

1. Retirer la valeur racine

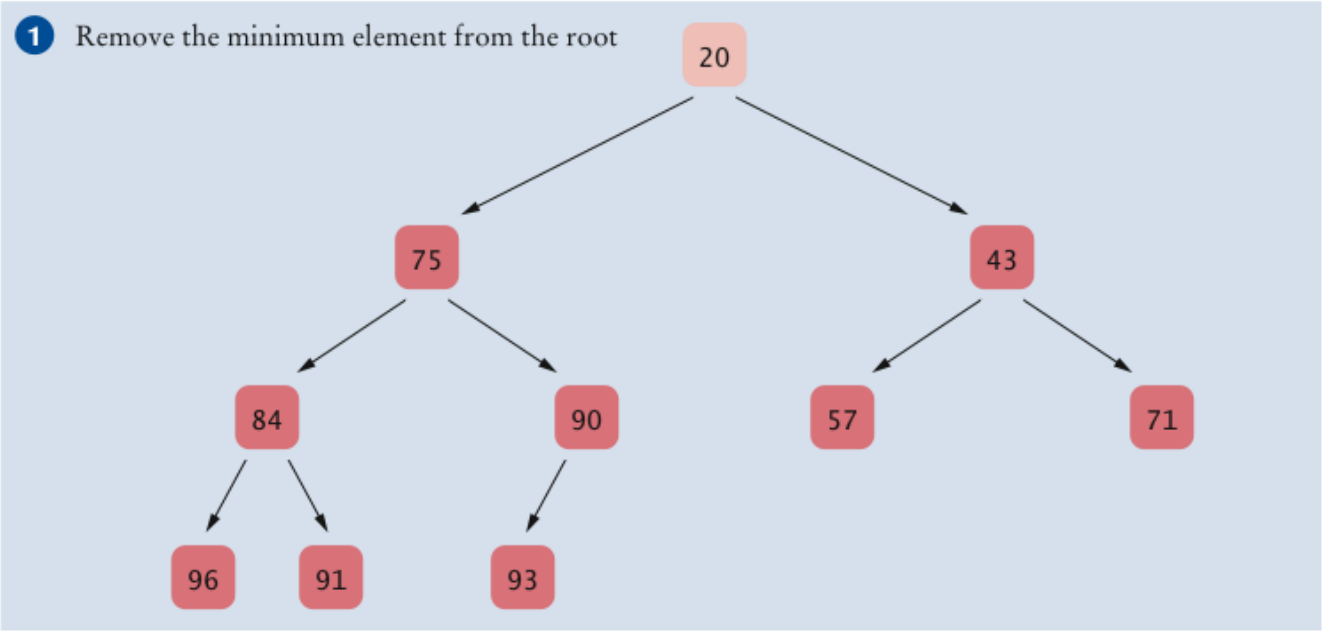


Figure 18 Removing the Minimum Value from a Heap

Continued

Supprimer la racine du tas (Queue de priorité)

2. Placer la valeur du dernier nœud dans la racine et retirer le dernier nœud. La propriété du tas pourra être violée pour la racine

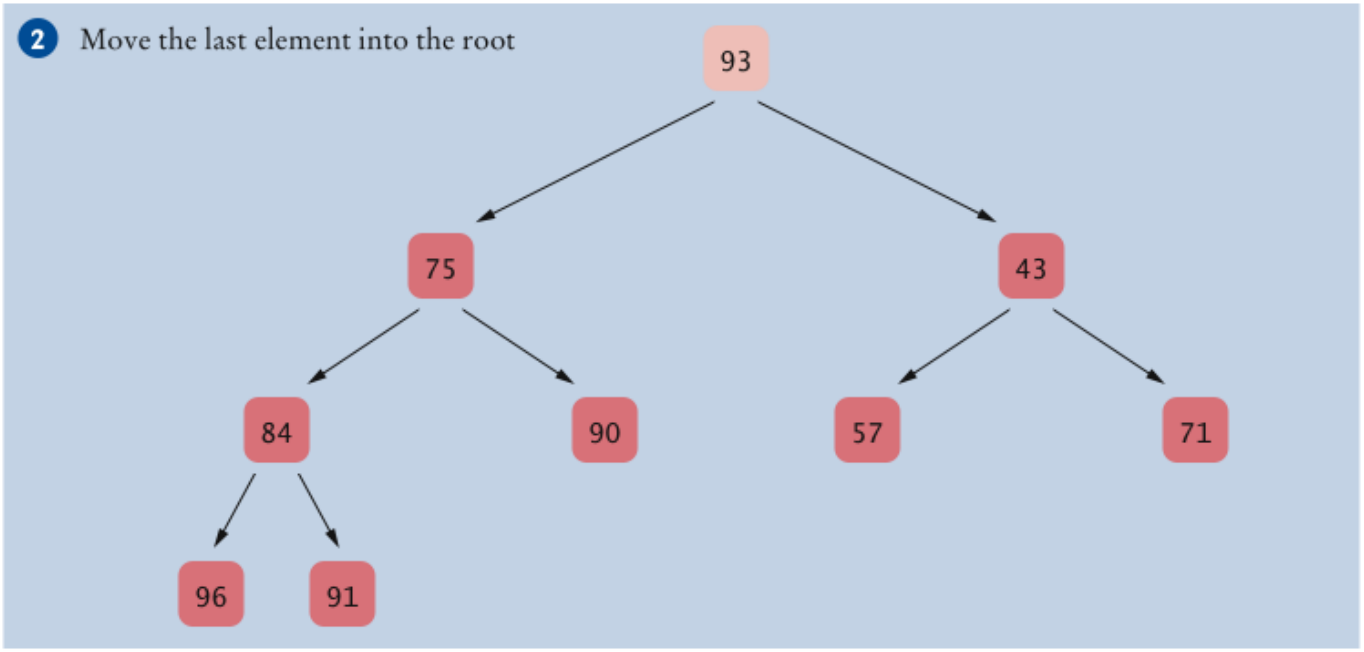


Figure 18 Removing the Minimum Value from a Heap

Continued

Supprimer la racine du tas (Queue de priorité)

3. Déplacer le plus petit fils dans la racine. La racine maintenant satisfait la propriété du tas. Répéter le processus avec le fils déplacé. Continuer jusqu'à ce que le fils déplacé n'aie pas les enfants plus petits – Réparer le tas

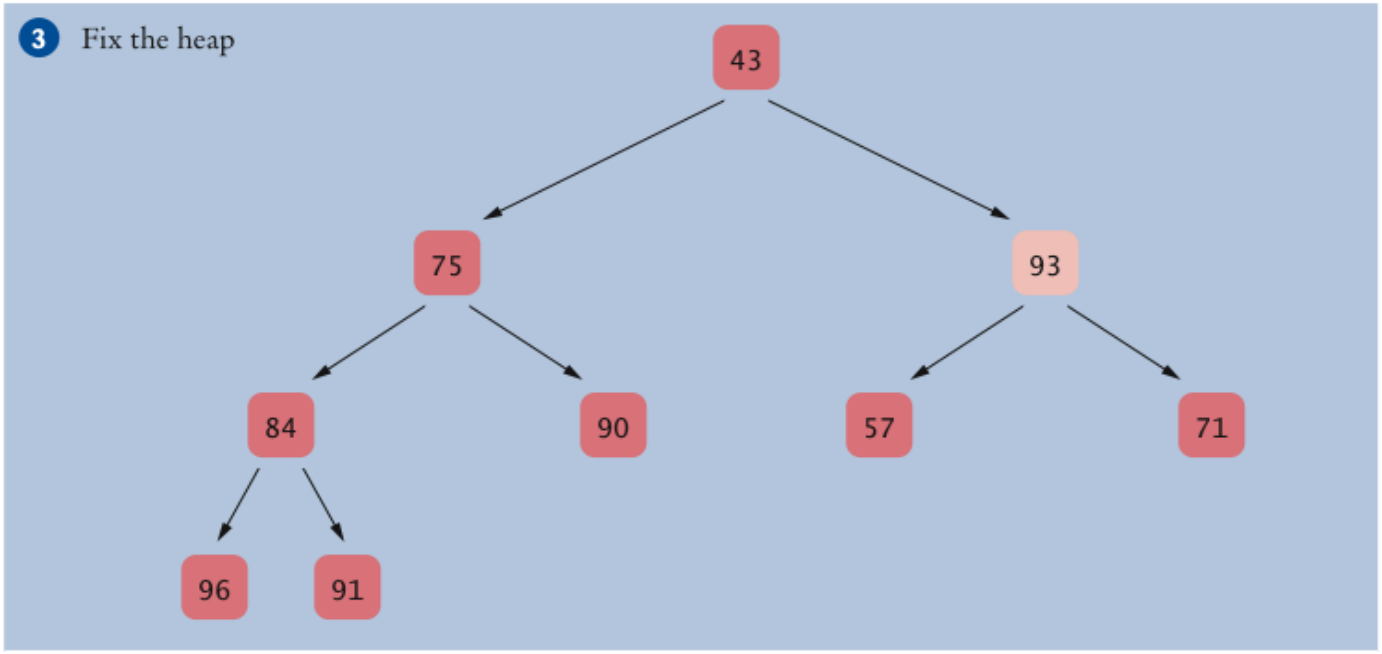


Figure 18 Removing the Minimum Value from a Heap

Continued

Supprimer un nœud arbitraire du tas

3. Déplacer le plus petit fils dans la racine. La racine maintenant satisfait la propriété du tas. Répéter le processus avec le fils déplacé. Continuer jusqu'à ce que le fils déplacé n'aie pas les enfants plus petits – Réparer le tas

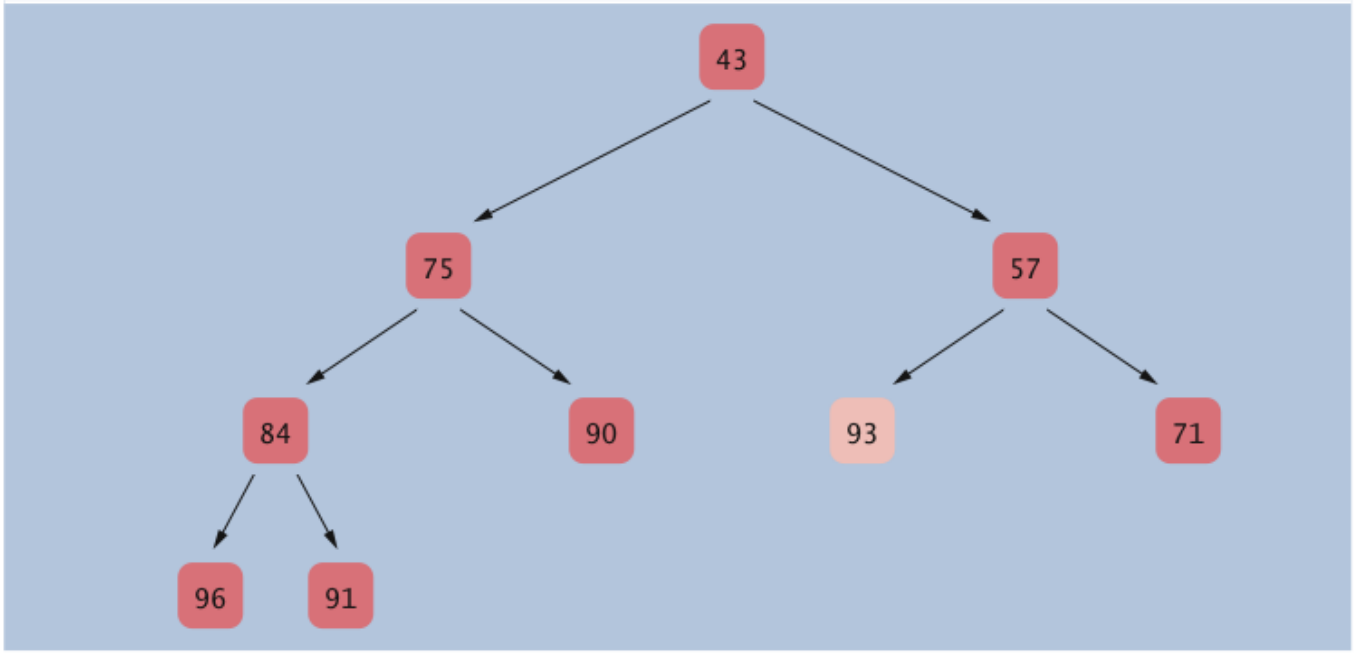


Figure 18 Removing the Minimum Value from a Heap

Efficacité d'un tas

- Opérations d'insertion et de suppression visite au maximum *h* nœuds
- *h*: Hauteur de l'arbre
- Si *n* est le nombre d'éléments, donc
  $2^{h-1} \leq n < 2^h$
  ou
  $h - 1 \leq \log_2(n) < h$
- Insertion et suppression prennent $O(\log(n))$ étapes
- La structure régulière du tas permet de stocker les nœuds du tas de manière efficace dans un tableau

Stocker un tas dans un tableau

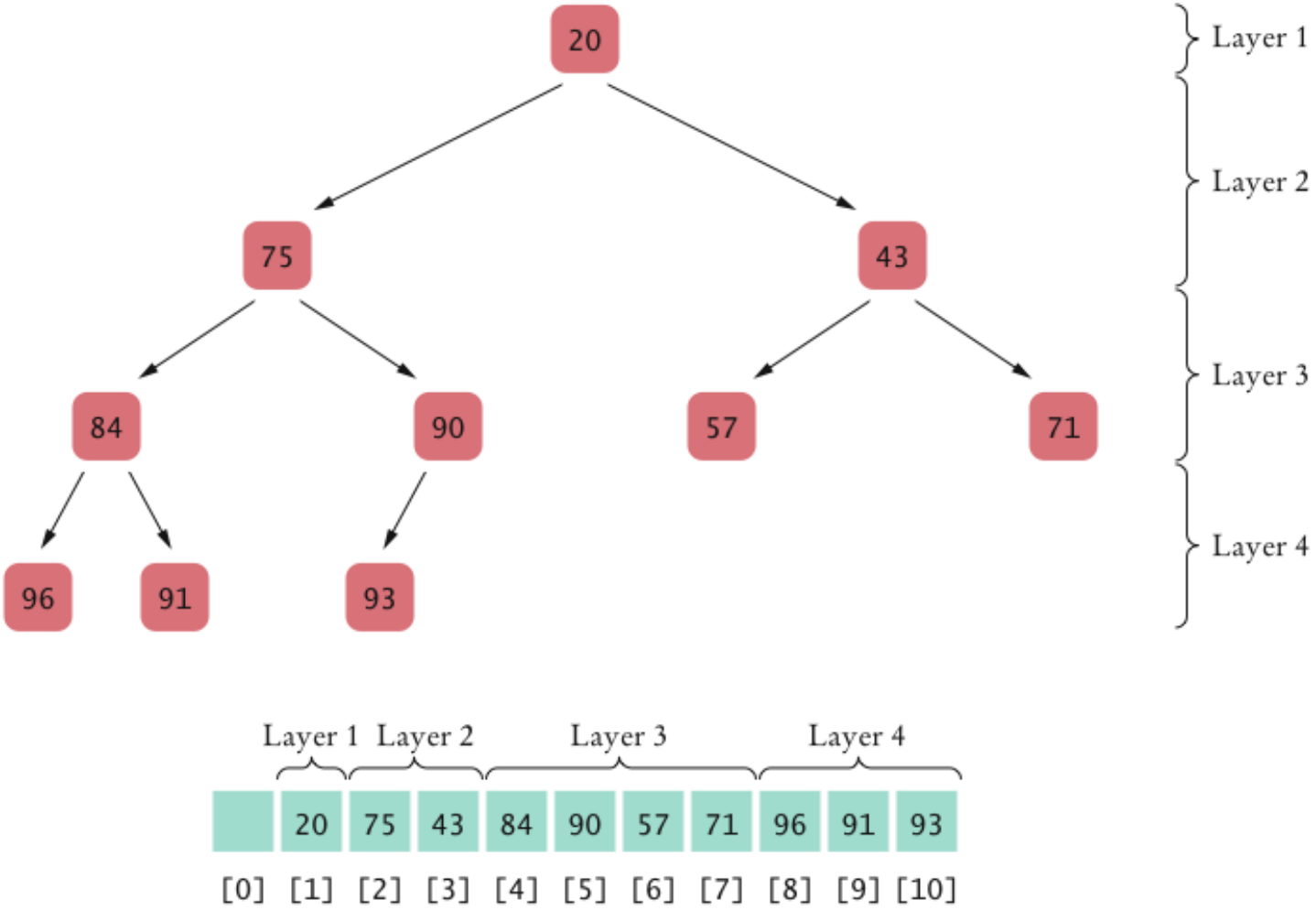


Figure 19 Storing a Heap in an Array

ch16/pqueue/MinHeap.java

```
1  import java.util.*;
2
3  /**
4     This class implements a heap.
5  */
6  public class MinHeap
7  {
8     private ArrayList<Comparable> elements;
9
10    /**
11       Constructs an empty heap.
12    */
13    public MinHeap()
14    {
15       elements = new ArrayList<Comparable>();
16       elements.add(null);
17    }
18
```

Continued

ch16/pqueue/MinHeap.java (co

```
/**
   Adds a new element to this heap.
   @param newElement the element to add
*/
public void add(Comparable newElement)
{
   // Add a new leaf
   elements.add(null);
   int index = elements.size() - 1;

   // Demote parents that are larger than the new element
   while (index > 1
         && getParent(index).compareTo(newElement) > 0)
   {
      elements.set(index, getParent(index));
      index = getParentIndex(index);
   }

   // Store the new element into the vacant slot
   elements.set(index, newElement);
}
```

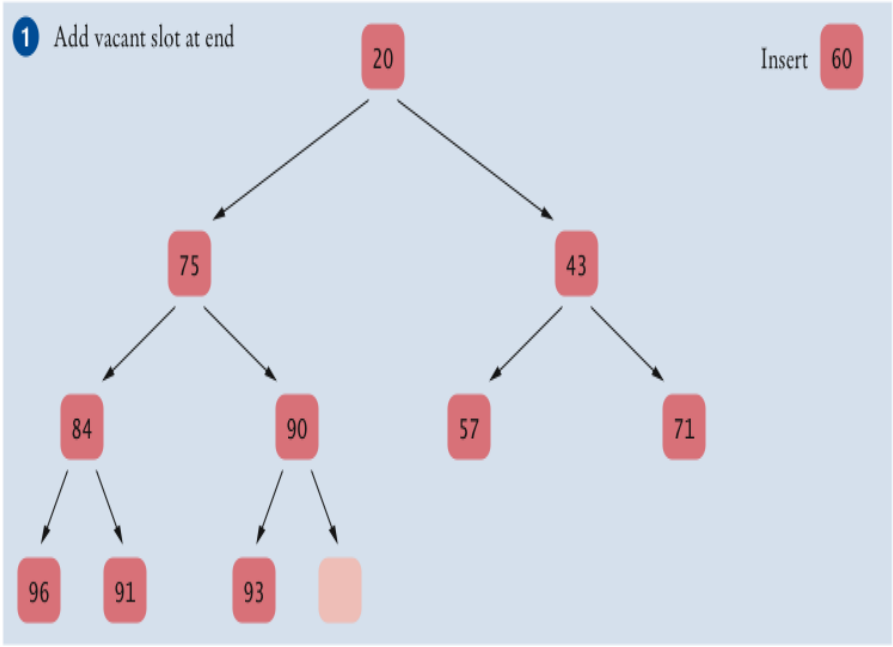


Figure 17 Inserting an Element into a Heap

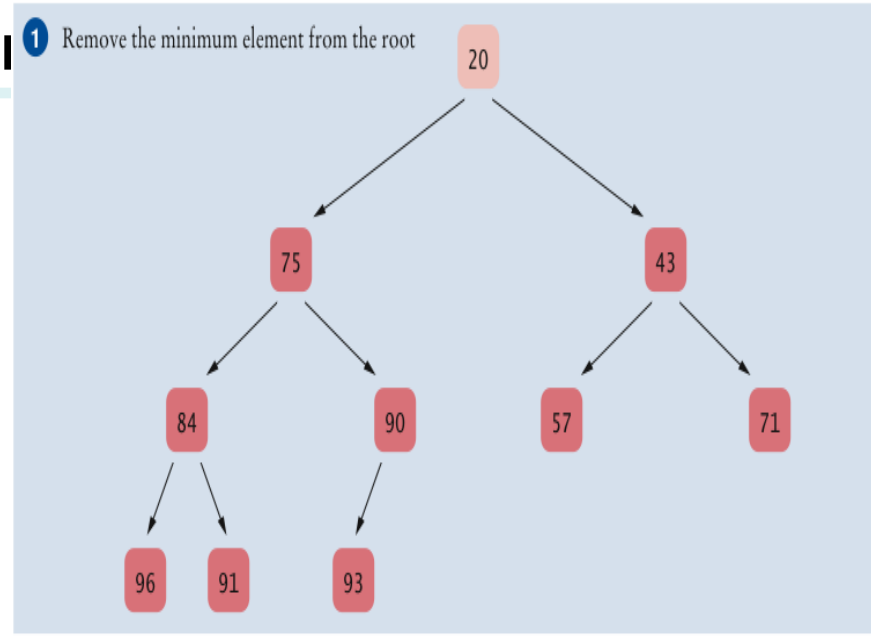
Continued

ch16/pqueue/MinHeap.java (cont.)

```
41      /**
42         Gets the minimum element stored in this heap.
43         @return the minimum element
44      */
45      public Comparable peek()
46      {
47         return elements.get(1);
48      }
49
```

Continued

ch16/pqueue/MinHeap.java (co

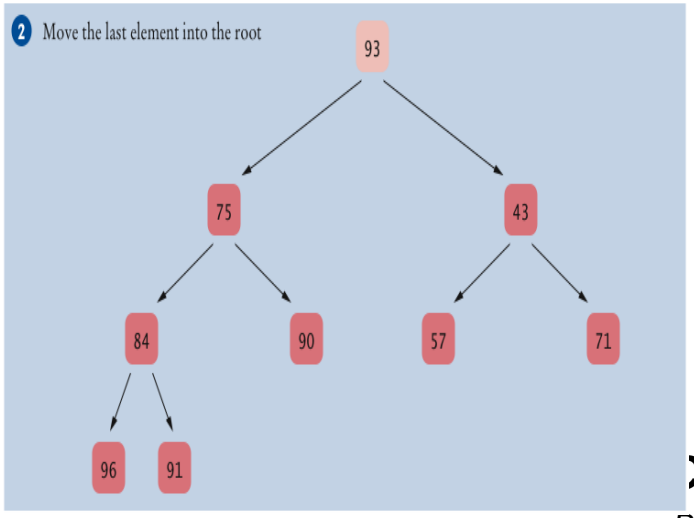



```
/**
   Removes the minimum element from this heap.
   @return the minimum element
*/
public Comparable remove()
{
   Comparable minimum = elements.get(1);

   // Remove last element
   int lastIndex = elements.size() - 1;
   Comparable last = elements.remove(lastIndex);

   if (lastIndex > 1)
   {
      elements.set(1, last);
      fixHeap();
   }

   return minimum;
}
```



Continued

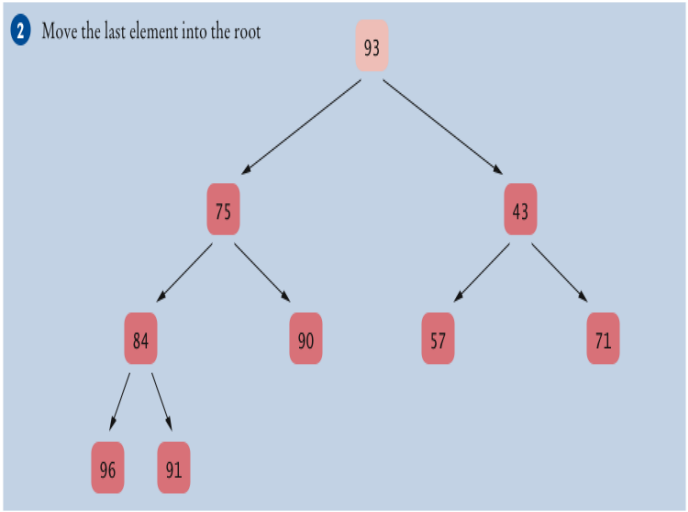
ch16/pqueue/MinHeap.java (cont.)

```
/**
   Turns the tree back into a heap, provided only the root
   node violates the heap condition.
*/
private void fixHeap()
{
   Comparable root = elements.get(1);

   int lastIndex = elements.size() - 1;
   // Promote children of removed root while they are
   //  smaller than last

   int index = 1;
   boolean more = true;
   while (more)
   {
      int childIndex = getLeftChildIndex(index);
      if (childIndex <= lastIndex)
      {
         // Get smaller child

         // Get left child first
         Comparable child = getLeftChild(index);
```



Continued

ch16/pqueue/MinHeap.java (cont.)

```
   // Use right child instead if it is smaller
   if (getRightChildIndex(index) <= lastIndex
         && getRightChild(index).compareTo(child) < 0)
   {
      childIndex = getRightChildIndex(index);
      child = getRightChild(index);
   }

   // Check if larger child is smaller than root
   if (child.compareTo(root) < 0)
   {
      // Promote child
      elements.set(index, child);
      index = childIndex;
   }
   else
   {
      // Root is smaller than both children
      more = false;
   }
}
```

Continued

ch16/pqueue/MinHeap.java (cont.)

```
115             else
116             {
117                // No children
118                more = false;
119             }
120          }
121
122          // Store root element in vacant slot
123          elements.set(index, root);
124       }
125
126       /**
127          Returns the number of elements in this heap.
128       */
129       public int size()
130       {
131          return elements.size() - 1;
132       }
133
```

Continued

ch16/pqueue/MinHeap.java (cont.)

```
134     /**
135        Returns the index of the left child.
136        @param index the index of a node in this heap
137        @return the index of the left child of the given node
138     */
139     private static int getLeftChildIndex(int index)
140     {
141        return 2 * index;
142     }
143
144     /**
145        Returns the index of the right child.
146        @param index the index of a node in this heap
147        @return the index of the right child of the given node
148     */
149     private static int getRightChildIndex(int index)
150     {
151        return 2 * index + 1;
152     }
153
```

Continued

ch16/pqueue/MinHeap.java (cont.)

```
154     /**
155        Returns the index of the parent.
156        @param index the index of a node in this heap
157        @return the index of the parent of the given node
158     */
159     private static int getParentIndex(int index)
160     {
161        return index / 2;
162     }
163
164     /**
165        Returns the value of the left child.
166        @param index the index of a node in this heap
167        @return the value of the left child of the given node
168     */
169     private Comparable getLeftChild(int index)
170     {
171        return elements.get(2 * index);
172     }
173
```

Continued

ch16/pqueue/MinHeap.java (cont.)

```
174      /**
175         Returns the value of the right child.
176         @param index the index of a node in this heap
177         @return the value of the right child of the given node
178      */
179      private Comparable getRightChild(int index)
180      {
181         return elements.get(2 * index + 1);
182      }
183
184      /**
185         Returns the value of the parent.
186         @param index the index of a node in this heap
187         @return the value of the parent of the given node
188      */
189      private Comparable getParent(int index)
190      {
191         return elements.get(index / 2);
192      }
193   }
```

ch16/pqueue/HeapDemo.java

```
1  /**
2     This program demonstrates the use of a heap as a priority queue.
3  */
4  public class HeapDemo
5  {
6     public static void main(String[] args)
7     {
8        MinHeap q = new MinHeap();
9        q.add(new WorkOrder(3, "Shampoo carpets"));
10       q.add(new WorkOrder(7, "Empty trash"));
11       q.add(new WorkOrder(8, "Water plants"));
12       q.add(new WorkOrder(10, "Remove pencil sharpener shavings"));
13       q.add(new WorkOrder(6, "Replace light bulb"));
14       q.add(new WorkOrder(1, "Fix broken sink"));
15       q.add(new WorkOrder(9, "Clean coffee maker"));
16       q.add(new WorkOrder(2, "Order cleaning supplies"));
17
18       while (q.size() > 0)
19          System.out.println(q.remove());
20    }
21 }
```

ch16/pqueue/WorkOrder.java

```
1  /**
2     This class encapsulates a work order with a priority.
3  */
4  public class WorkOrder implements Comparable
5  {
6     private int priority;
7     private String description;
8
9     /**
10       Constructs a work order with a given priority and description.
11       @param aPriority the priority of this work order
12       @param aDescription the description of this work order
13    */
14    public WorkOrder(int aPriority, String aDescription)
15    {
16       priority = aPriority;
17       description = aDescription;
18    }
19
```

Continued

ch16/pqueue/WorkOrder.java (cont.)

```
20      public String toString()
21      {
22         return "priority=" + priority + ", description=" + description;
23      }
24
25      public int compareTo(Object otherObject)
26      {
27         WorkOrder other = (WorkOrder) otherObject;
28         if (priority < other.priority) return -1;
29         if (priority > other.priority) return 1;
30         return 0;
31      }
32   }
```

Program Run:

```
priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener shavings
```

L'algorithme Heapsort

- Basé sur l'insertion des éléments dans un tas et leurs suppression dans l'ordre trié
- Cet algorithme est de l'ordre $O(n \log(n))$:
  - *Chaque insertion et suppression est de l'ordre O(log(n))*
  - *Ces étapes sont répétées n fois, une fois pour chaque élément*

L’algorithme Heapsort

- Peut être fait plus efficacement
  - *Commencer avec une séquence des valeurs stockées dans un tableau et « réparer » la propriété du tas de manière itérative*
- D’abord transformer les petits sous arbres en un tas, ensuite réparer les plus grands arbres
- Les arbres de la taille 1 sont automatiquement des tas
- Commencer la procédure de réparation à partir des sous arbres avec les racines situées au niveau avant dernier
- La méthode générique `fixHeap` répare le sous arbre avec la racine présentée par une index donnée:

```
void fixHeap(int rootIndex, int lastIndex)
```

Transformation de l'arbre en tas

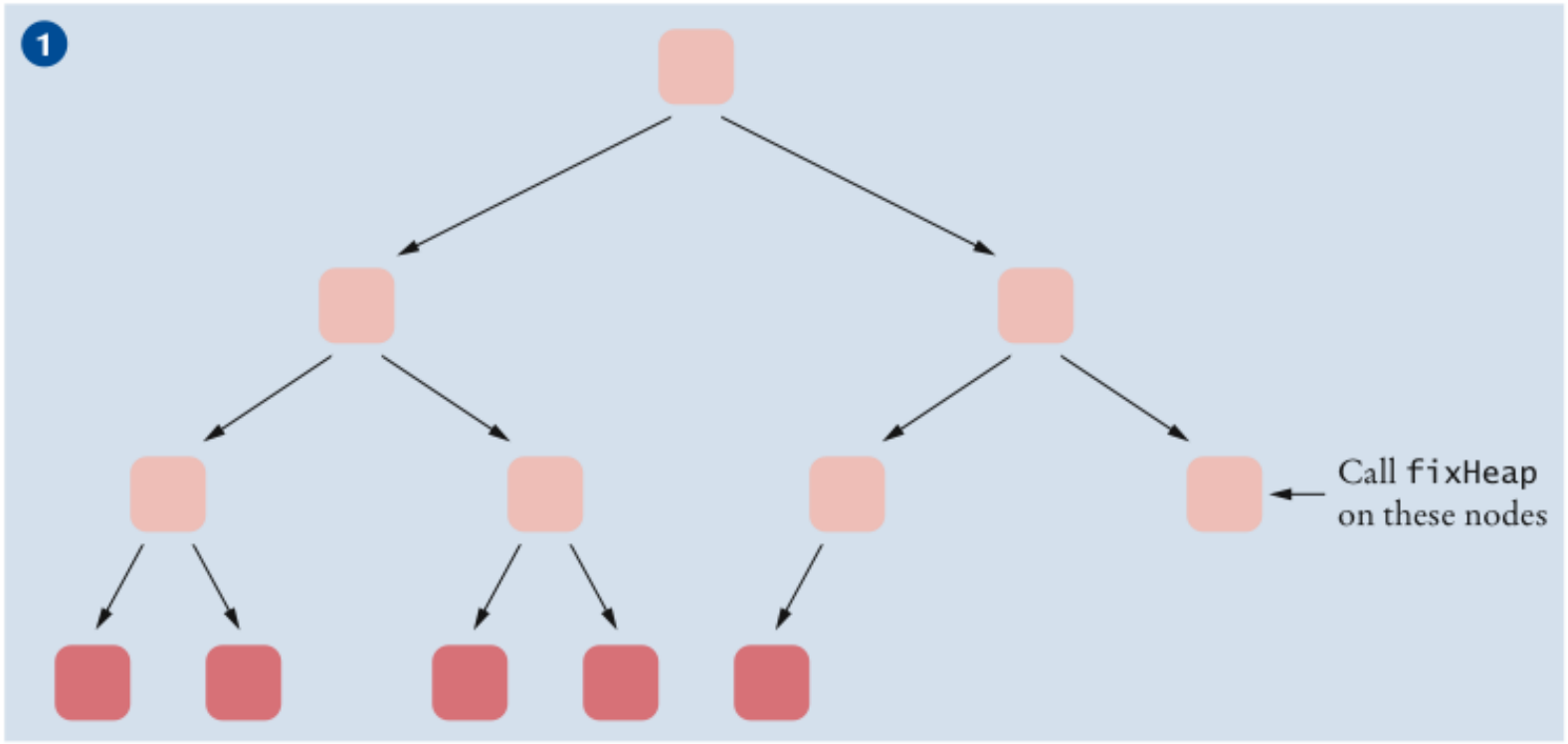


Figure 20 Turning a Tree into a Heap

Continued

Transformation de l'arbre en tas

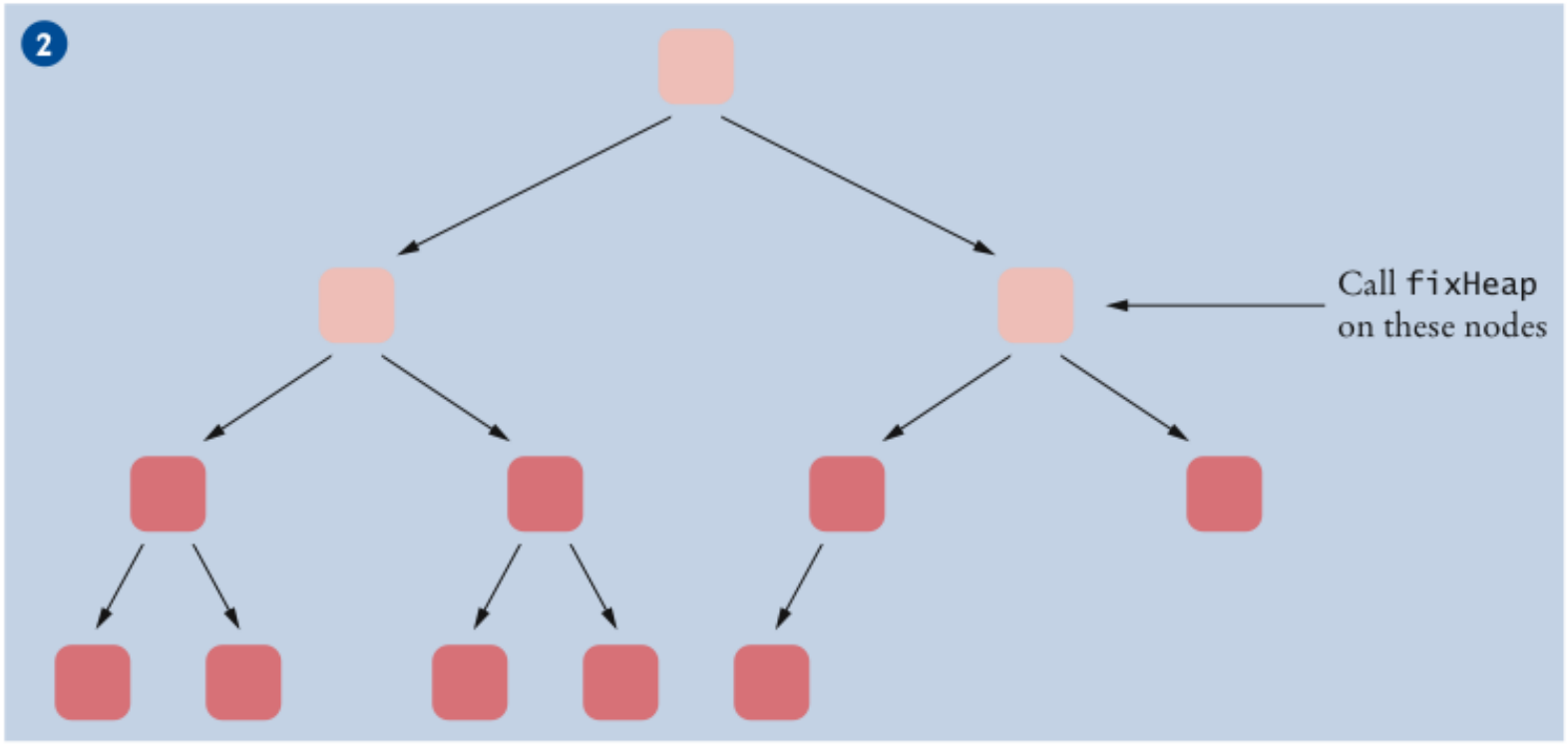


Figure 20 Turning a Tree into a Heap

Continued

Transformation de l'arbre en tas

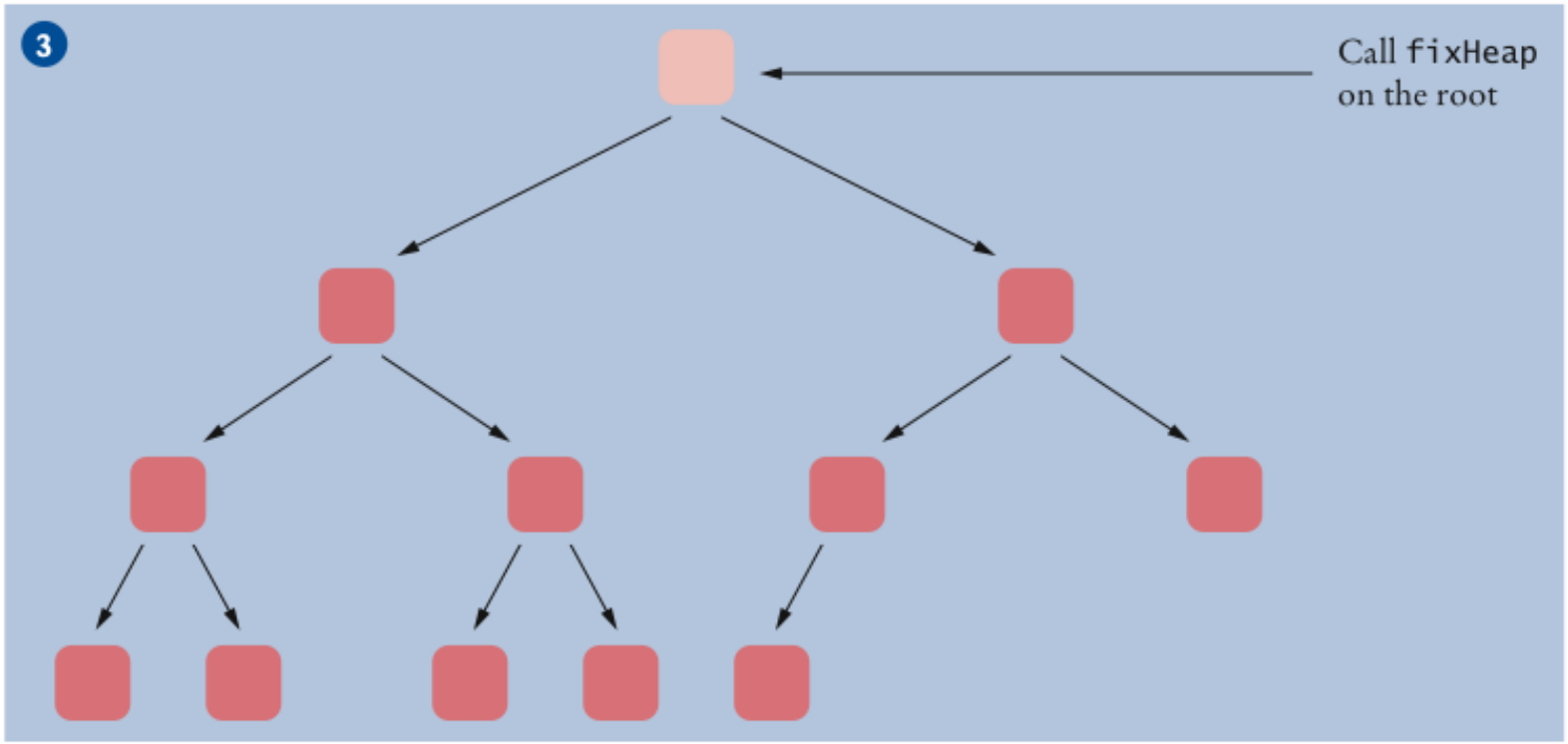


Figure 20 Turning a Tree into a Heap

Algorithme HeapSort

- Après avoir transformer le tableau en un tas, répéter l'opération de suppression de l'élément racine
  - *Échanger l'élément racine avec le dernier élément et réduire la taille de l'arbre*
- La racine supprimée sera placée dans la dernière position du tableau qui n'a pas besoin d'être utilisée par le tas
- Nous pouvons utiliser le même tableau pour stocker le tas et la séquence des valeurs triées
- Utiliser le tas max plus tôt que le tas min pour construire la séquence des valeurs triées dans l'ordre croissant

Utiliser Heapsort pour trier un tableau

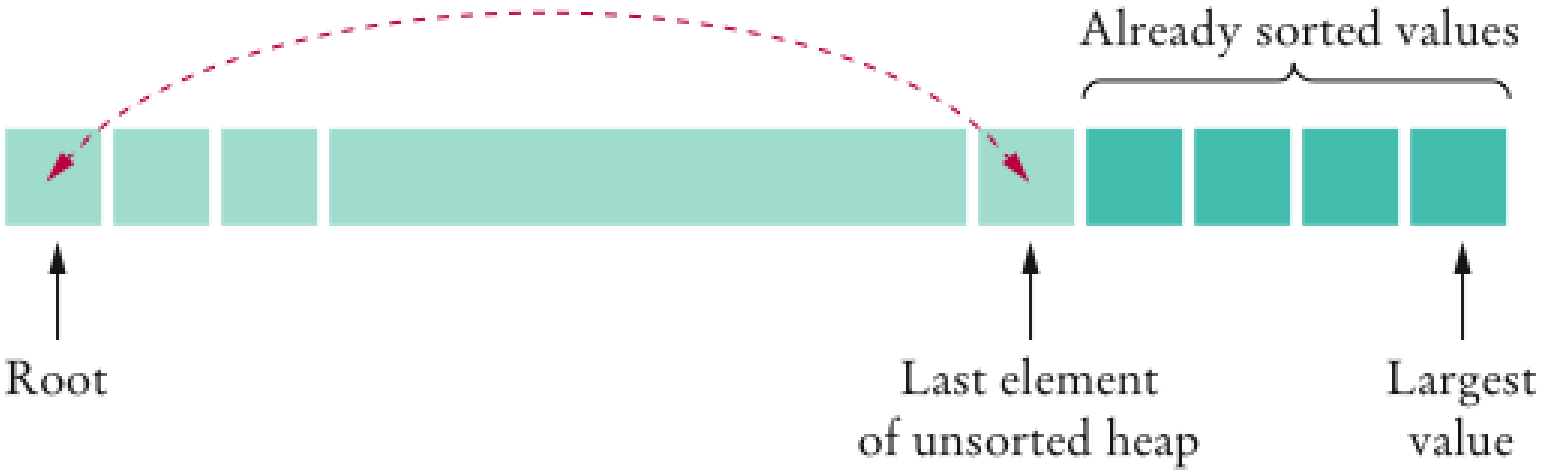


Figure 21 Using Heapsort to Sort an Array

ch16/heapsort/HeapSorter.java

```
1  /**
2     This class applies the heapsort algorithm to sort an array.
3  */
4  public class HeapSorter
5  {
6     private int[] a;
7
8     /**
9        Constructs a heap sorter that sorts a given array.
10       @param anArray an array of integers
11    */
12    public HeapSorter(int[] anArray)
13    {
14       a = anArray;
15    }
16
```

Continued

ch16/heapsort/HeapSorter.java (cont.)

```
17      /**
18         Sorts the array managed by this heap sorter.
19      */
20      public void sort()
21      {
22         int n = a.length - 1;
23         for (int i = (n - 1) / 2; i >= 0; i--)
24            fixHeap(i, n);
25         while (n > 0)
26         {
27            swap(0, n);
28            n--;
29            fixHeap(0, n);
30         }
31      }
32
```

Continued

ch16/heapsort/HeapSorter.java (cont.)

```
33      /**
34         Ensures the heap property for a subtree, provided its
35         children already fulfill the heap property.
36         @param rootIndex the index of the subtree to be fixed
37         @param lastIndex the last valid index of the tree that
38         contains the subtree to be fixed
39      */
40      private void fixHeap(int rootIndex, int lastIndex)
41      {
42         // Remove root
43         int rootValue = a[rootIndex];
44
45         // Promote children while they are larger than the root
46
47         int index = rootIndex;
48         boolean more = true;
49         while (more)
50         {
51            int childIndex = getLeftChildIndex(index);
52            if (childIndex <= lastIndex)
53            {
54               // Use right child instead if it is larger
55               int rightChildIndex = getRightChildIndex(index);
56               if (rightChildIndex <= lastIndex
57                     && a[rightChildIndex] > a[childIndex])
58               {
59                  childIndex = rightChildIndex;
60               }
```

Continued

ch16/heapsort/HeapSorter.java (cont.)

```
61
62              if (a[childIndex] > rootValue)
63              {
64                 // Promote child
65                 a[index] = a[childIndex];
66                 index = childIndex;
67              }
68              else
69              {
70                 // Root value is larger than both children
71                 more = false;
72              }
73           }
74           else
75           {
76              // No children
77              more = false;
78           }
79        }
80
81        // Store root value in vacant slot
82        a[index] = rootValue;
83     }
84
```

Continued

ch16/heapsort/HeapSorter.java (cont.)

```
85      /**
86         Swaps two entries of the array.
87         @param i the first position to swap
88         @param j the second position to swap
89      */
90      private void swap(int i, int j)
91      {
92         int temp = a[i];
93         a[i] = a[j];
94         a[j] = temp;
95      }
96
97      /**
98         Returns the index of the left child.
99         @param index the index of a node in this heap
100        @return the index of the left child of the given node
101     */
102     private static int getLeftChildIndex(int index)
103     {
104        return 2 * index + 1;
105     }
106
```

Continued

ch16/heapsort/HeapSorter.java (cont.)

```
107      /**
108         Returns the index of the right child.
109         @param index the index of a node in this heap
110         @return the index of the right child of the given node
111      */
112      private static int getRightChildIndex(int index)
113      {
114         return 2 * index + 2;
115      }
116   }
```