Arbres binaires de recherche

SECTION 4.3 (WEISS)

SECTIONS 16.5, 16.6, BIG JAVA 4TH EDITION BY C. HORSTMANN

Arbres binaires de recherche (ABR)

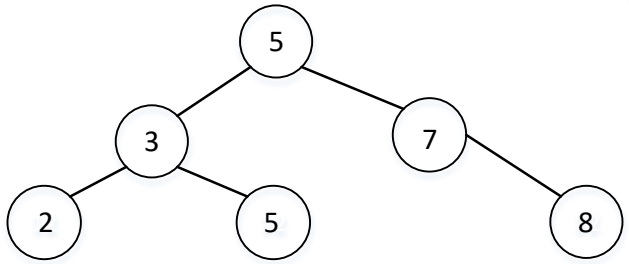
- **Une arbre binaire de recherche**
  - un arbre binaire
  - il y a une relation d'ordre entre un parent et ses enfants: gauche < parent < droite
- Cette organisation permet d’effectuer des recherches efficaces
- Dans un arbre binaire de recherche un parcours symétrique (infixe) visite les éléments dans leur ordre (trié)

Interface **abstraite** d'un ABR

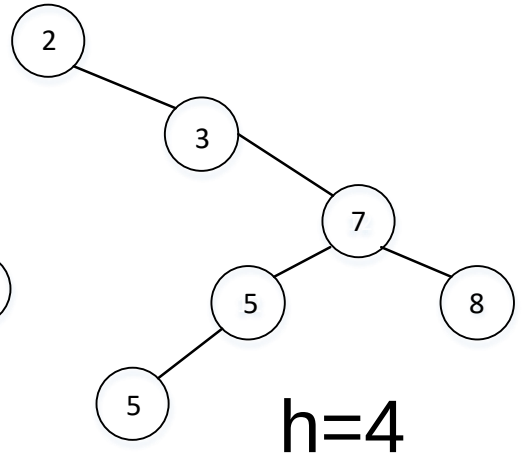
Nom de l'opération	Nom (anglais)	Action
RECHERCHER	find(e)	Rechercher un nœud ayant une clé donnée
MINIMUM	findMin()	Trouver un élément d'un ABR dont la clé est un minimum
MAXIMUM	findMax()	Trouver un élément d'un ABR dont la clé est un maximum
INSÉRER	insert(e)	Insérer une nouvelle valeur e dans un ABR
SUPPRIMER	remove(e)	Supprimer un nœud donné e d'un ABR

ABR

- Les opérations basiques sur un arbre binaire de recherche dépensent un temps proportionnel à la hauteur de l'arbre => O(h)



h=2



h=4

- ABR en Java
  - TreeSet et TreeMap

Arbres binaire de recherche

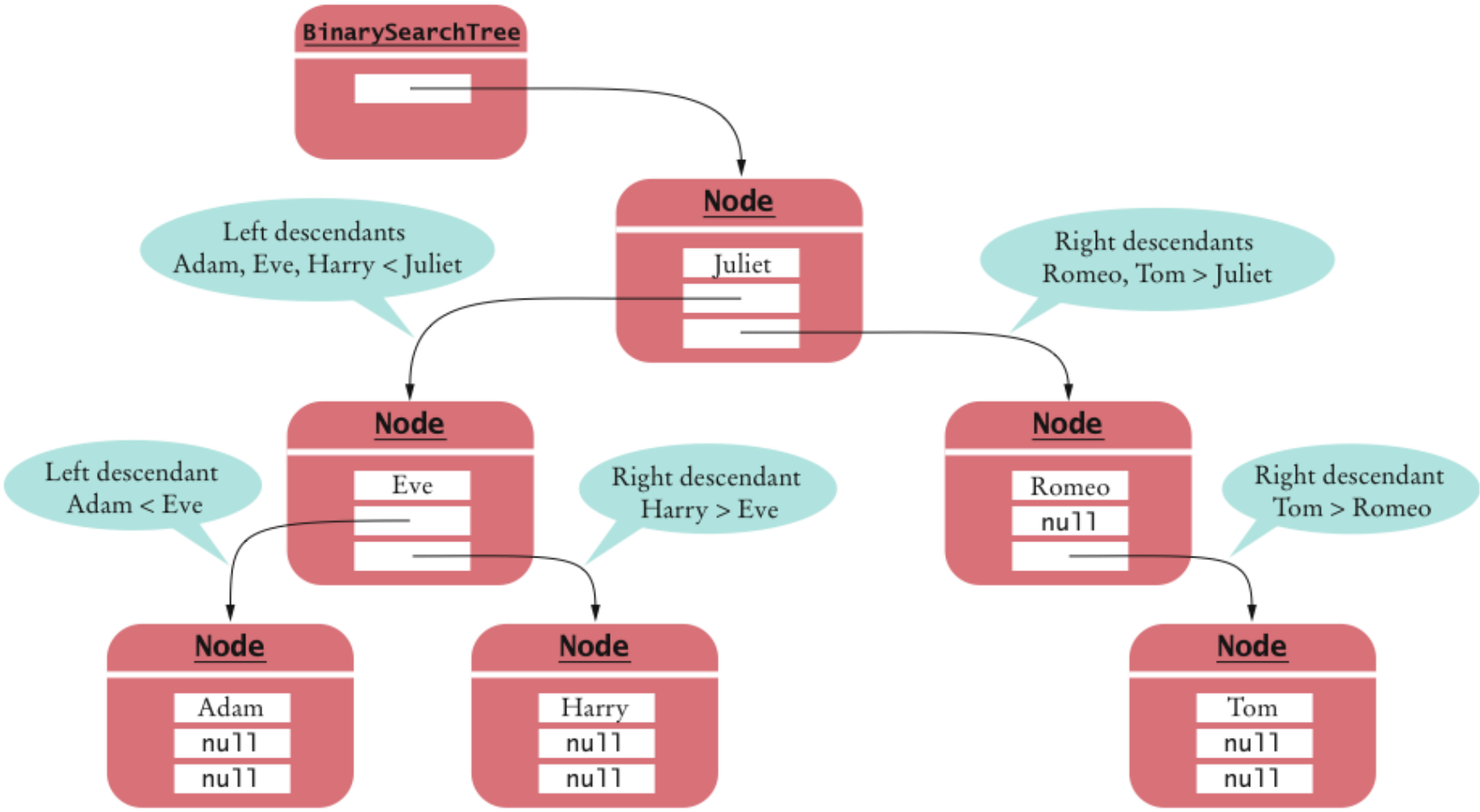


Figure 7 A Binary Search Tree

Big Java by Cay Horstmann

Arbre Binaire => non ABR

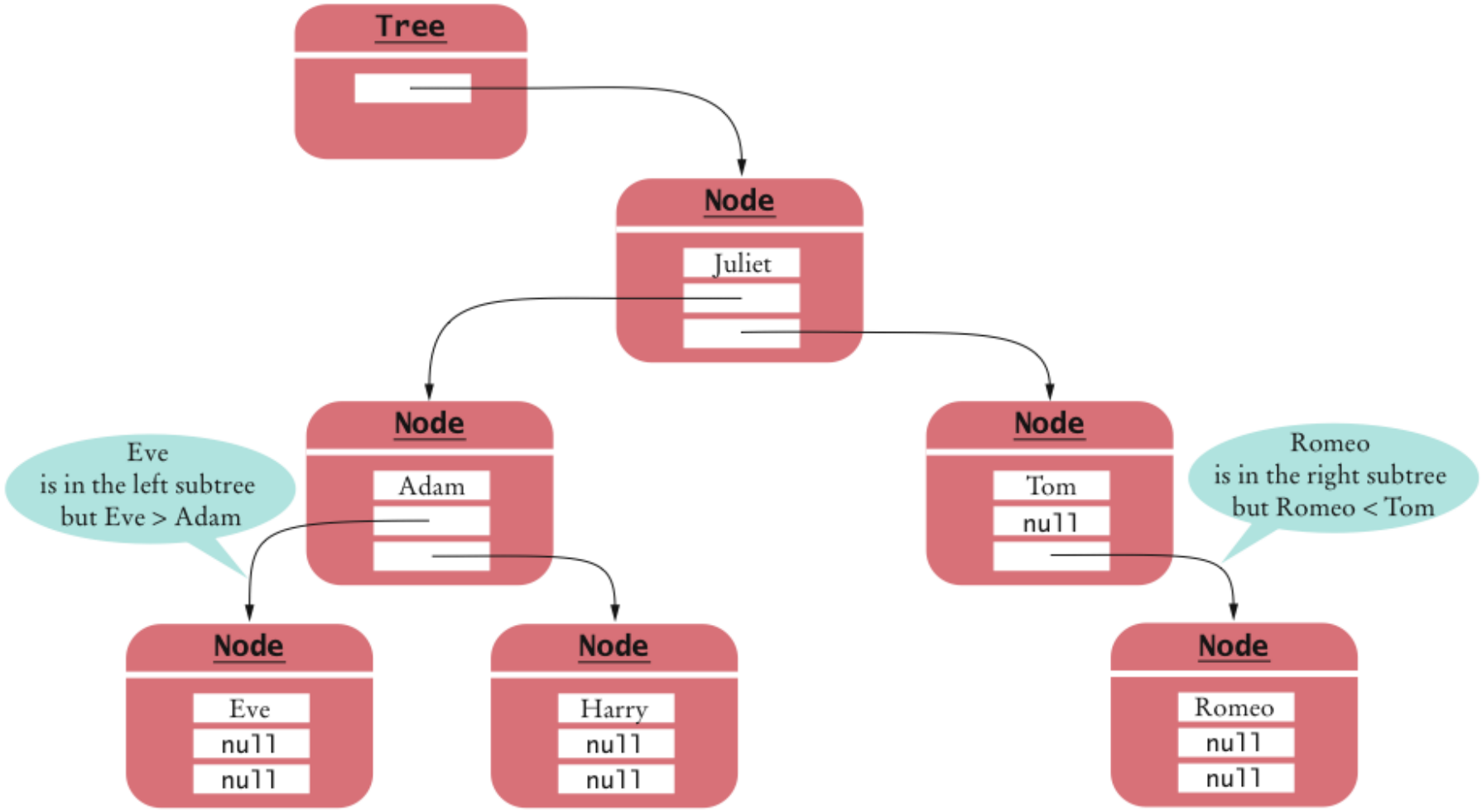


Figure 8 A Binary Tree That Is Not a Binary Search Tree

Implémentation de l'arbre binaire de recherche

- Classe ABR contient une référence sur un nœud racine
- Classe pour les nœuds
  - *Nœud contient deux liens (vers les nouds enfants, gauche et droite)*
  - *Nœud contient un champ Donnée*
  - *Donnée est de type* `Comparable`*, pour qu'on puisse comparer les valeurs pour mettre les nœuds dans les positions correctes dans l'arbre*
    - *Méthode* `compareTo`

Implémentation d'arbre binaire de recherche

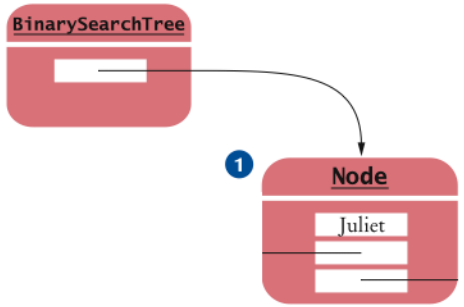
```
public class BinarySearchTree
{
   private Node root;

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

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

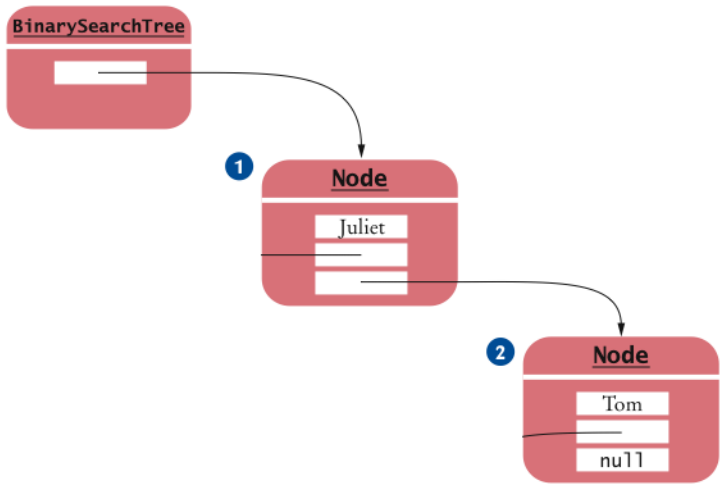
Exemple

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



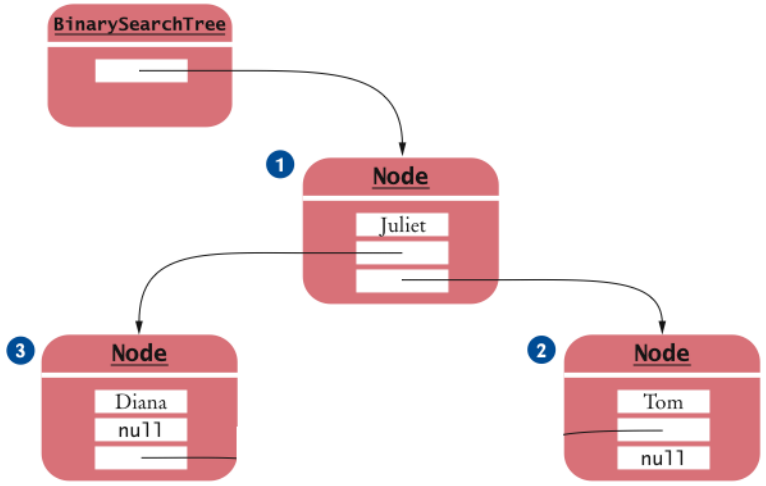
Exemple

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");  1
tree.add("Tom");     2
```



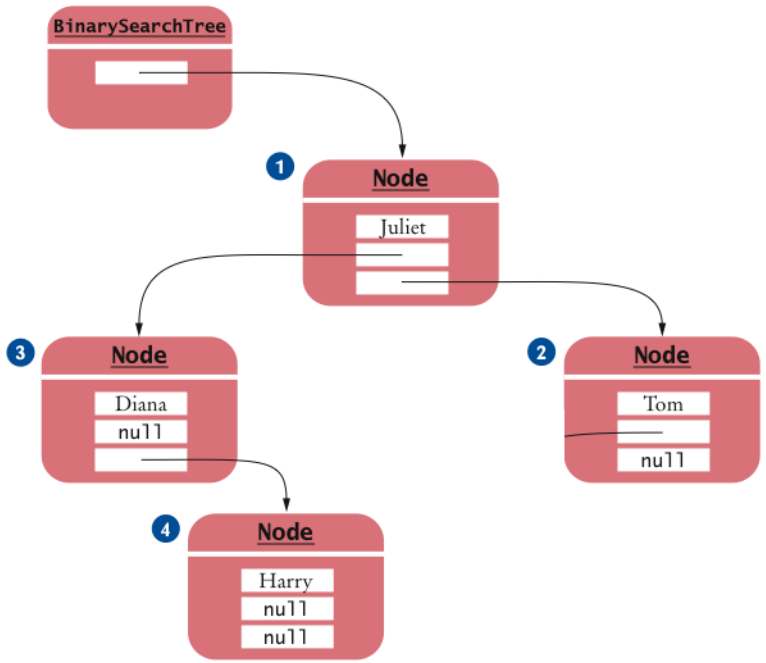
Exemple

- ```
  BinarySearchTree tree = new BinarySearchTree();
  tree.add("Juliet");1
  tree.add("Tom"); 2
  tree.add("Diana")3
  ```


  ```

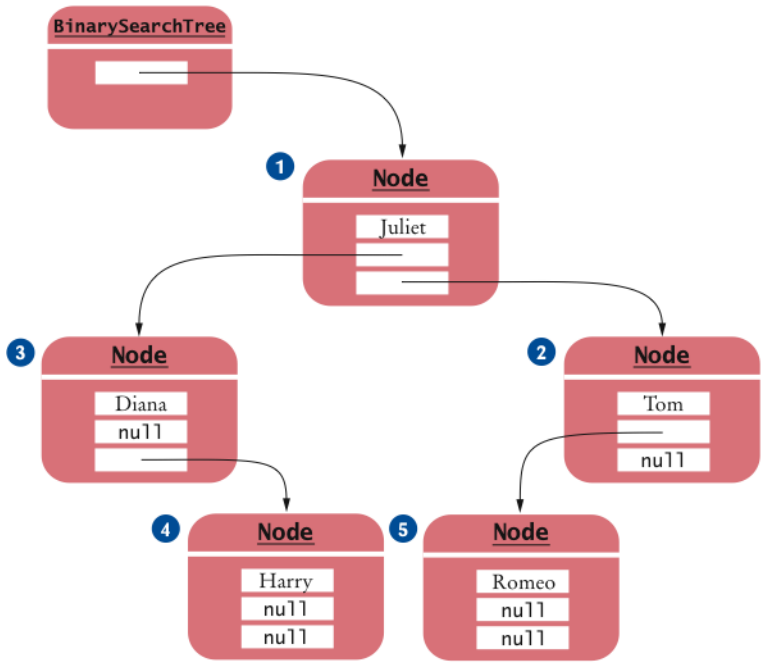
Exemple

- ```
  BinarySearchTree tree = new BinarySearchTree();
  tree.add("Juliet");1
  tree.add("Tom");  2
  tree.add(”Diana")3
  tree.add("Harry");  4
  ```


  ```

Exemple

- ```
  BinarySearchTree tree = new BinarySearchTree();
  tree.add("Juliet");1
  tree.add("Tom");  2
  tree.add(”Diana")3
  tree.add("Harry");  4
  tree.add("Romeo");  5
  ```


  ```

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);
        }
    }
    ...
}
```

Arbres binaire de recherche

- Lorsqu'on supprime un nœud avec un seul enfant, l'enfant remplace le nœud à supprimer
- Supprimer un nœud avec deux enfants
  - remplacer ce nœud avec le plus petit nœud de son sous arbre droite
  - Remplacer ce nœud avec le plus grand nœud de son sous arbre gauche

Supprimer le nœud avec un seul enfant

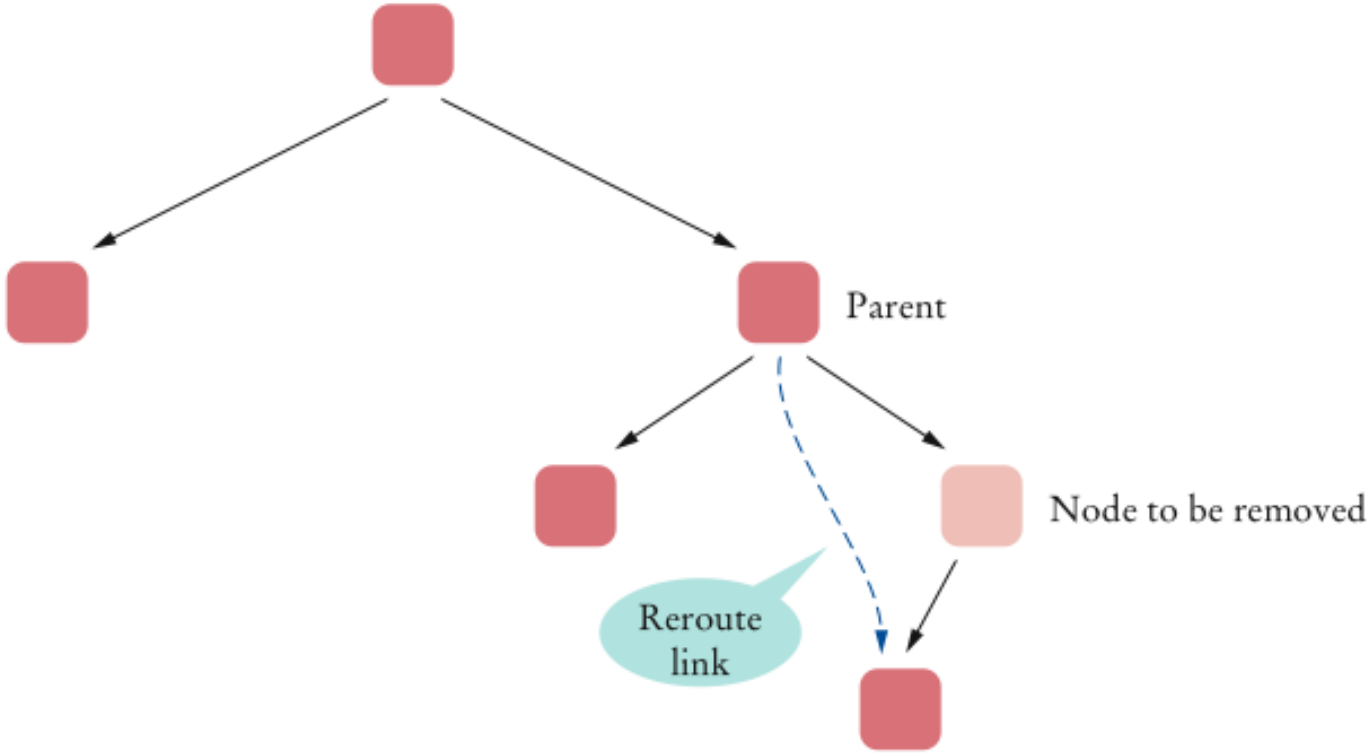


Figure 11
Removing a Node with One Child

Supprimer le nœud avec deux enfants

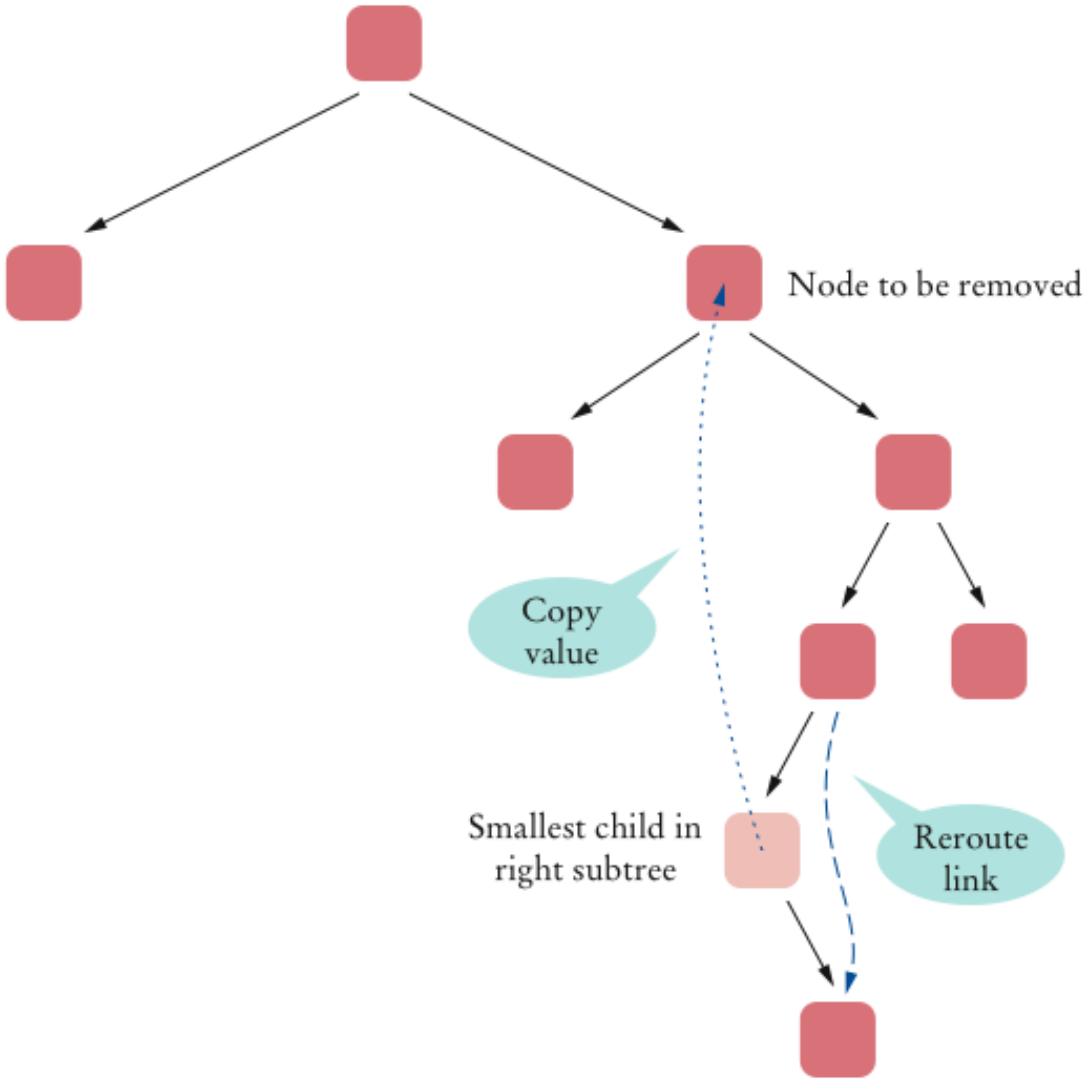
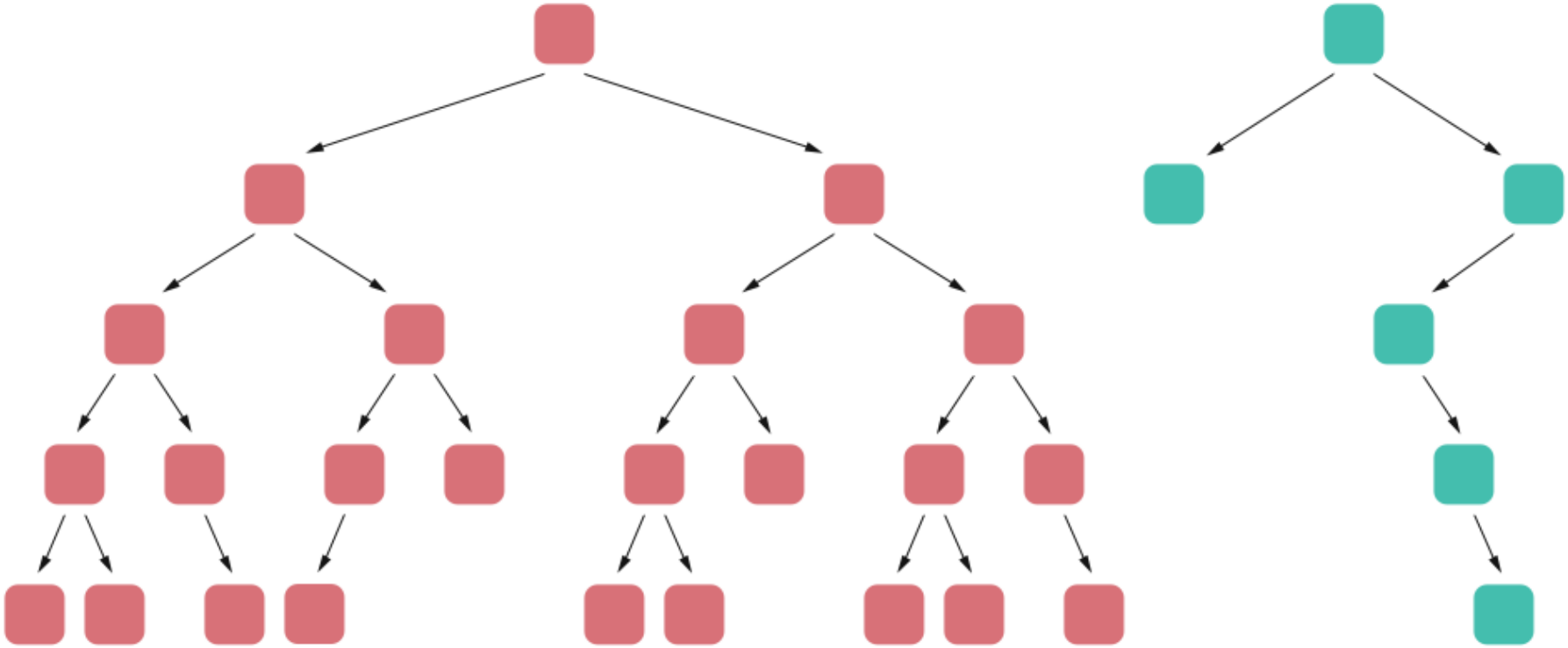


Figure 12 Removing a Node with Two Children

Arbres binaire de recherche

- **Un arbre balancé:** chaque nœud a approximativement le même nombre de descendants 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 lent
  - *Pire cas – aussi lent que l'insertion dans une liste chaînée*

Arbres binaires balancés et non balancés



Balanced

Unbalanced

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    }
```

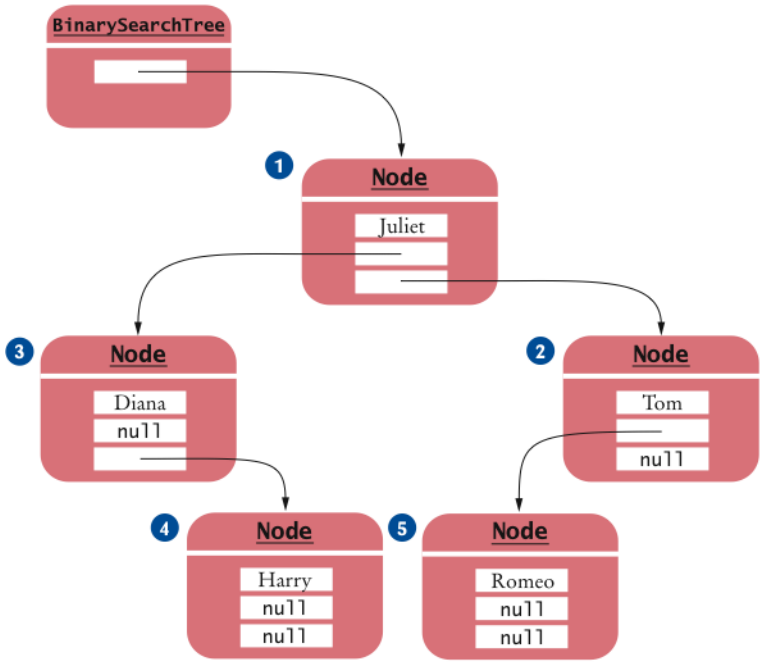
Parcours d’un arbre binaire

- Imprimer les éléments d’arbre en ordre/infixe/symmétrique :
  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*



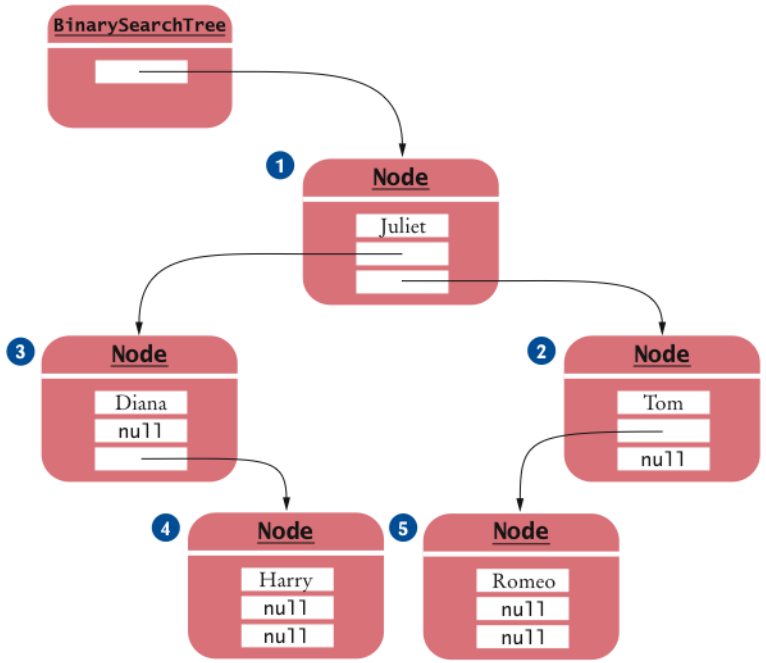
- Comment imprimer le sous arbre commençant sur `Diana`?
  1. *Imprimer le sous arbre gauche de* `Diana` *– rien à imprimer*
  1. *Imprimer* `Diana`
  2. *Imprimer le sous arbre droite de* `Diana`*, c'est* `Harry`

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();
   }
   ...
}
```