

Queue de priorité Monceau (Tas)



CHAPITRE 6.1-6.5(WEISS)

BIG JAVA CHAPITRE 16 (C. HORSTMANN)

Queues de priorité

- **Une queue de priorité** collectionne les éléments possédants les priorités
- Exemple: Collection des requêtes de travail qui peuvent 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)*

Queues de priorité

- Opérations
- `insert`  `enqueue`
- `deleteMin`  `dequeue`
- La bibliothèque standard Java fournit la classe `PriorityQueue`
- La structure de données appelée Monceau (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

Monceau (Heaps)

- Un monceau (*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 monceau
 1. *Arbre binaire presque complet (équilibré)*
 - Tous les nœuds sont remplis sauf au dernier niveau à droite
 2. *L'arbre satisfait la propriété de monceau*
 - Tous les nœuds stockent les valeurs presque aussi grandes (petites – monceau max) que ces descendants
- La propriété de monceau garantit que le plus petit élément est toujours placé dans la racine.

Monceau (Heaps)

- Structure de données partiellement ordonnée qui permet d'accéder efficacement au plus petit (ou au plus grand) élément d'une collection d'éléments
- La propriété de monceau garantit que le plus petit élément (monceau min) est placé dans la racine
- Il n'y a pas de contraintes d'ordonnancement entre les enfants de gauche et de droite

Un arbre binaire presque complet

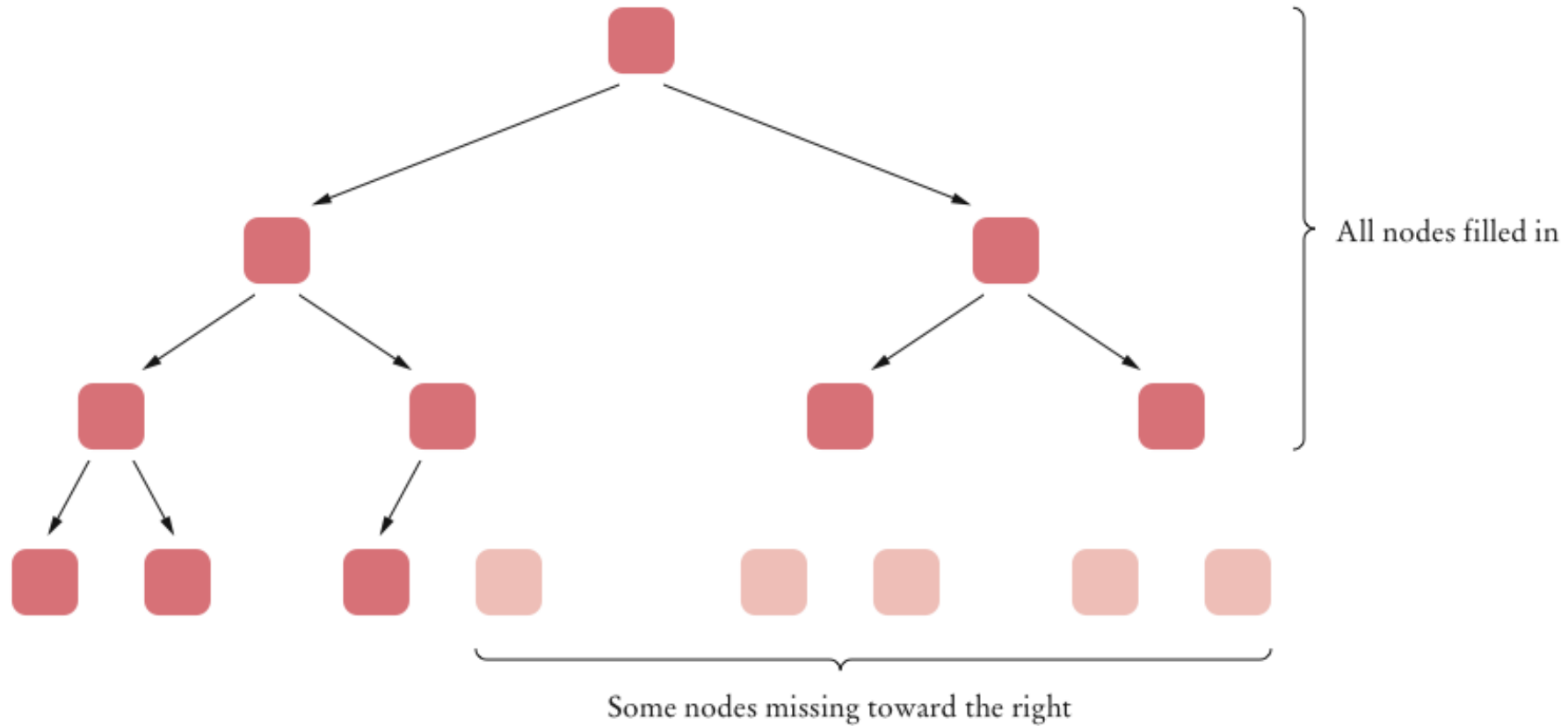
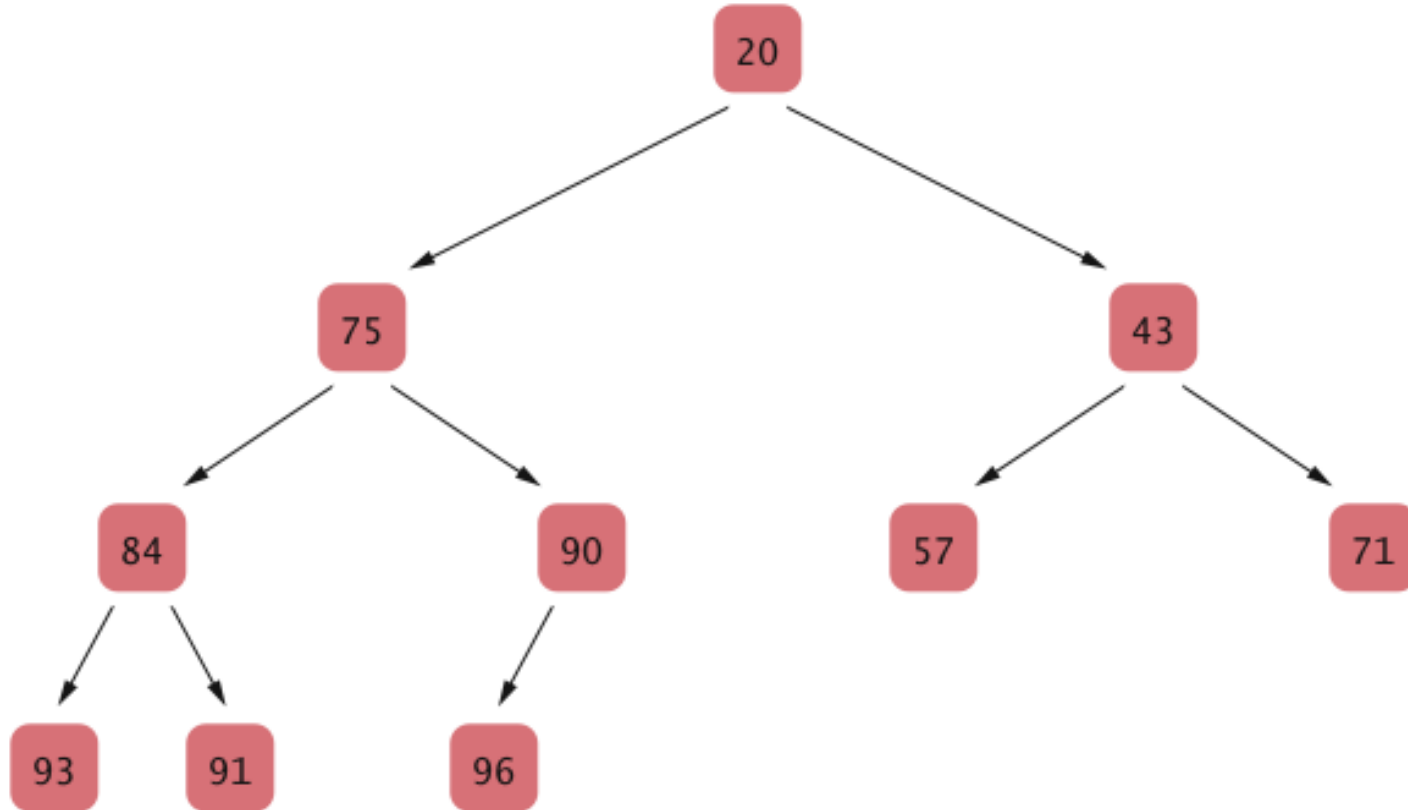


Figure 15 An Almost Completely Filled Tree

Big Java C.Horstmann

Monceau (min)

Figure 16
A Heap



Monceau et ABR

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

Insérer un nouvel élément dans un monceau

1. Ajouter une feuille à la fin de l'arbre binaire équivalent

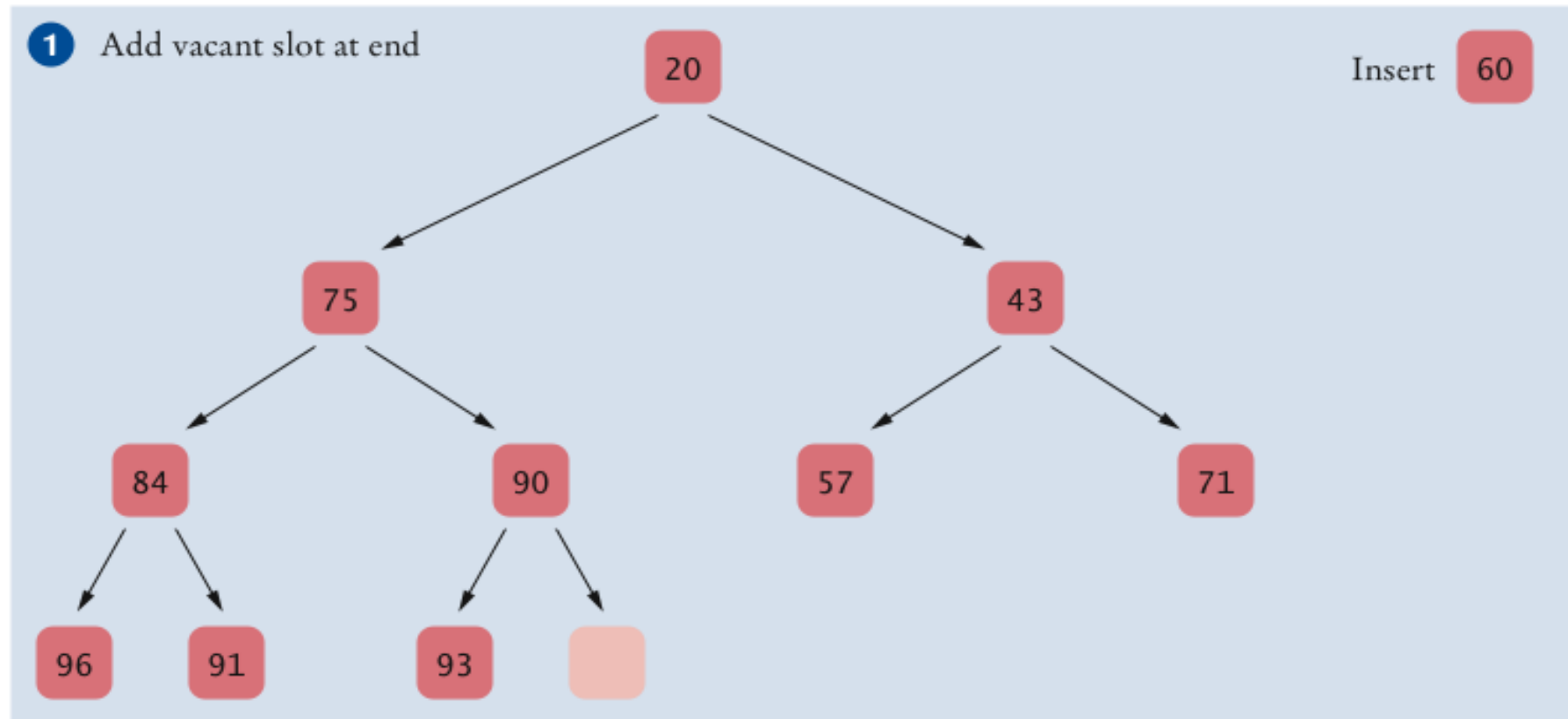
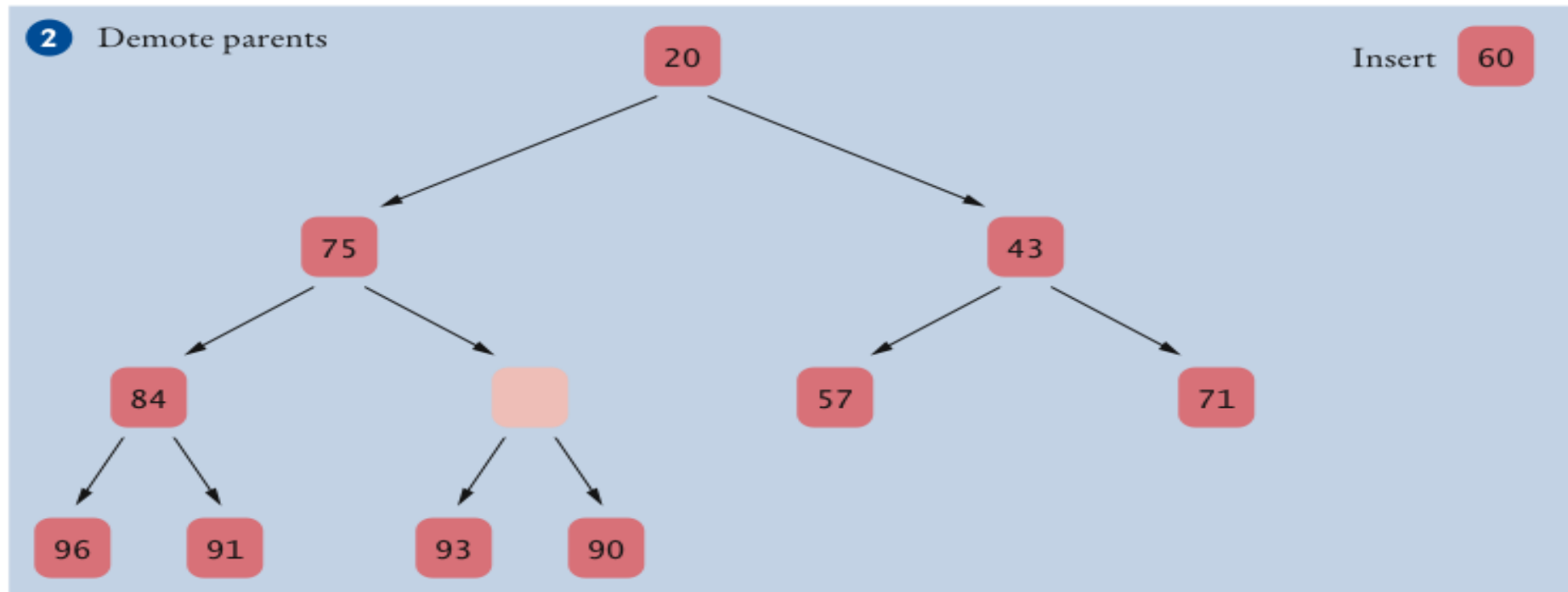


Figure 17 Inserting an Element into a Heap

Big Java C.Horstmann

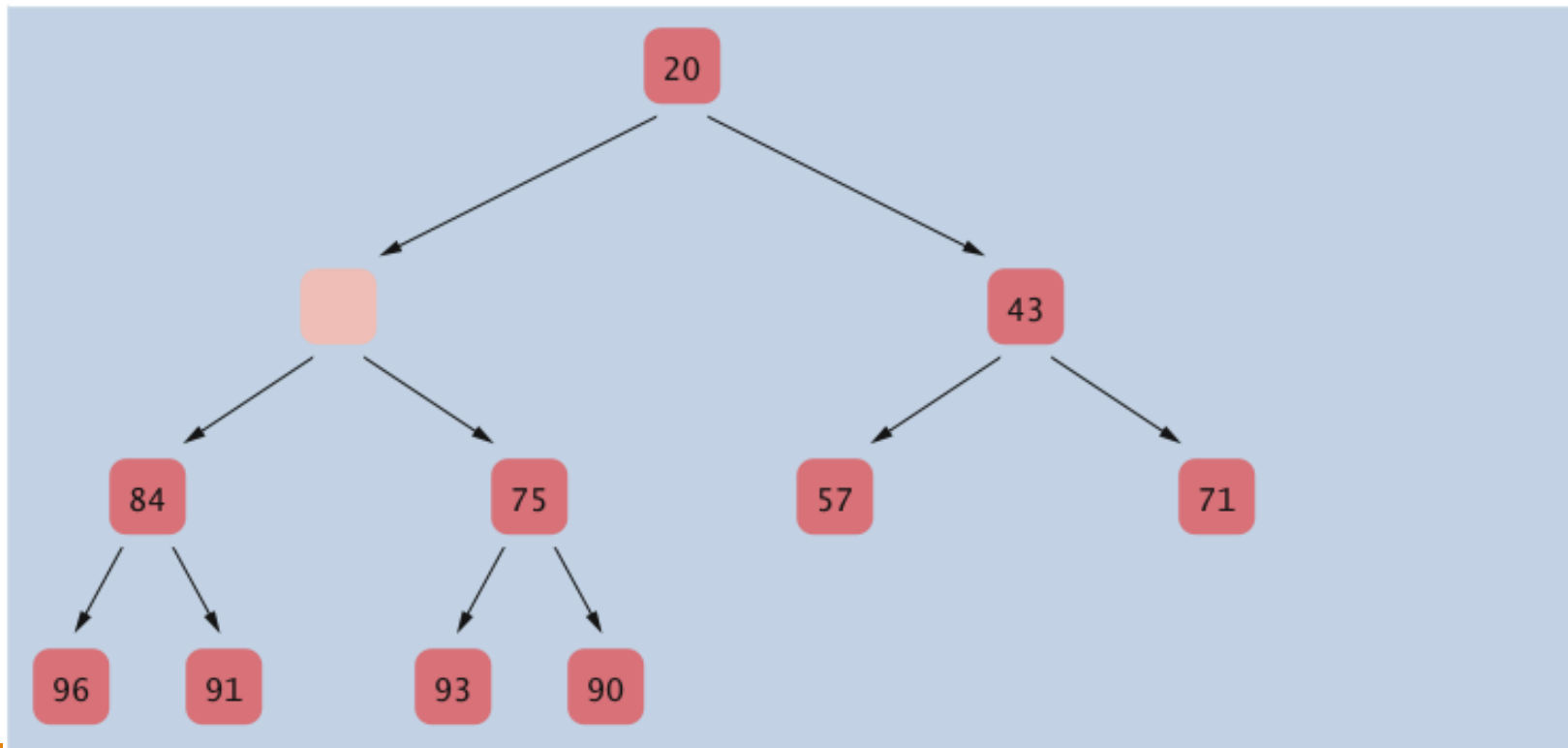
Insérer un nouvel élément dans un monceau

2. Déplacez le parent dans le slot vide et placez le slot vide à la place du parent. Répétez cette procédure jusqu'à ce que le parent du slot vide ne soit plus grand que l'élément à insérer.



Insérer un nouvel élément dans un monceau

2. Déplacez le parent dans le slot vide et placez le slot vide à la place du parent. Répétez cette procédure jusqu'à ce que le parent du slot vide ne soit plus grand que l'élément à insérer.



Insérer un nouvel élément dans un monceau

3. À ce stade, soit le slot se trouve à la position racine, soit le parent du slot est plus petit que la valeur à insérer. Insérez l'élément dans le slot.

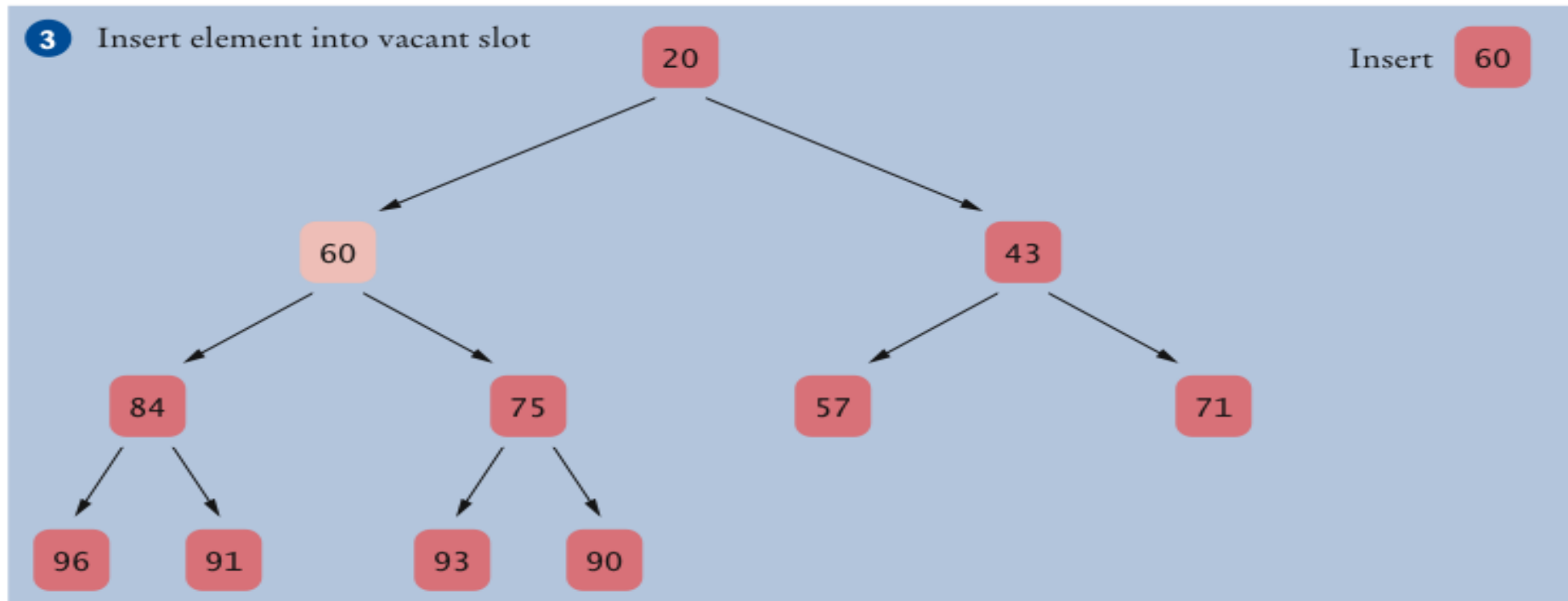
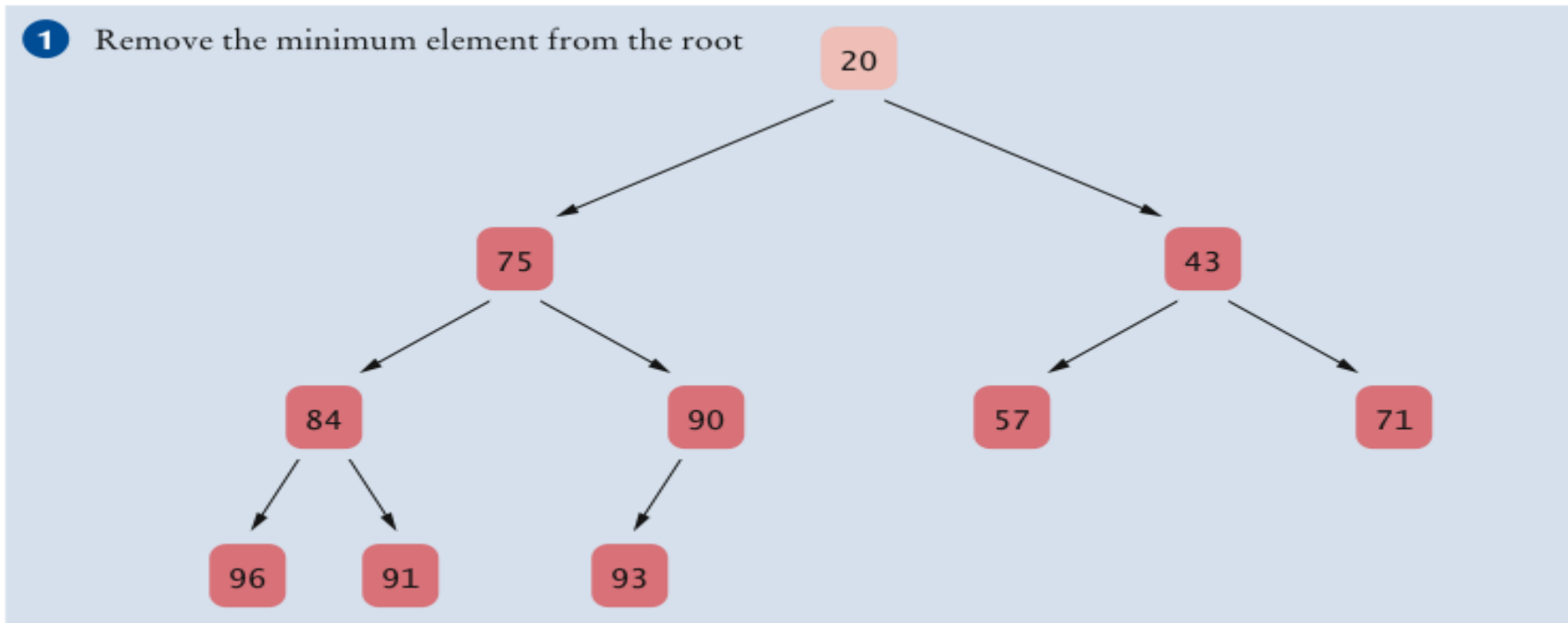


Figure 17 (continued) Inserting an Element into a Heap

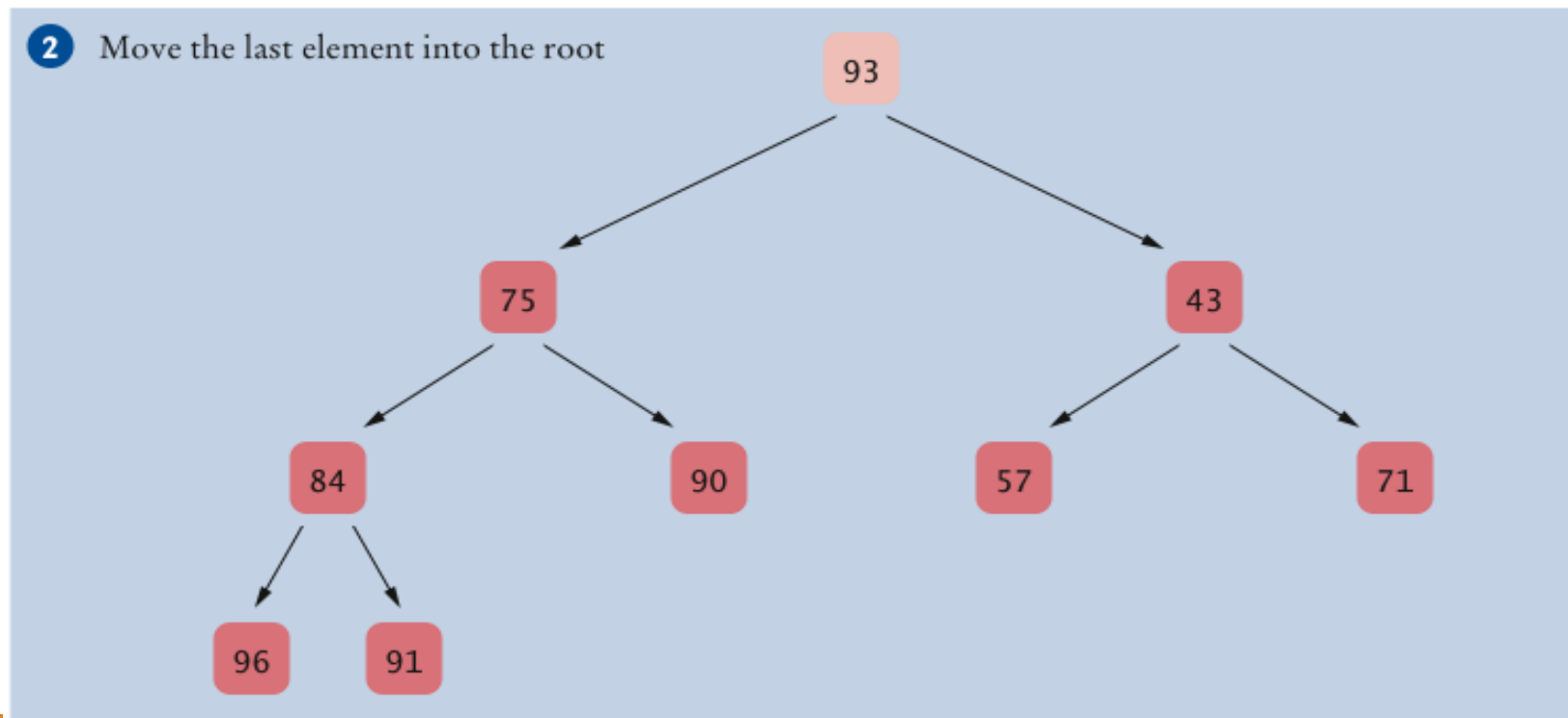
Supprimer la racine (élément Min ou Max)

1. Retirer la valeur racine – enlèvement de l'élément minimal (ou maximal)



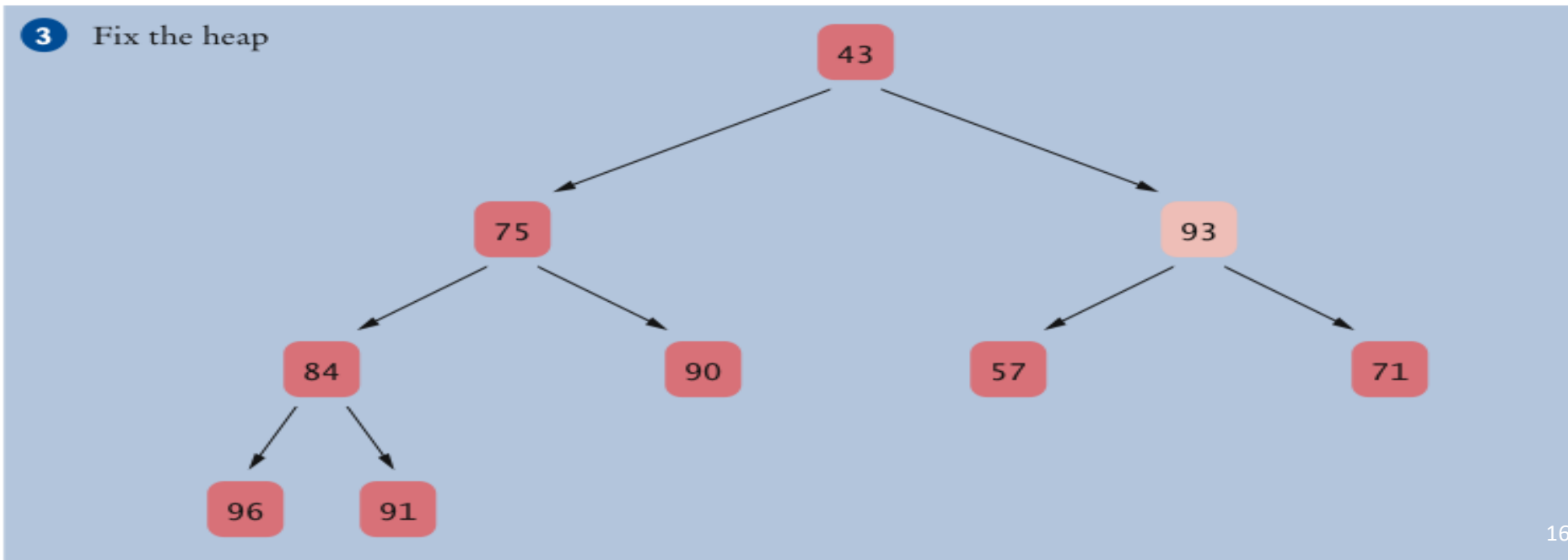
Supprimer la racine (élément Min ou Max)

2. Placer la valeur du dernier nœud dans la racine et retirer le dernier nœud. La propriété de monceau pourrait être violée pour la racine.



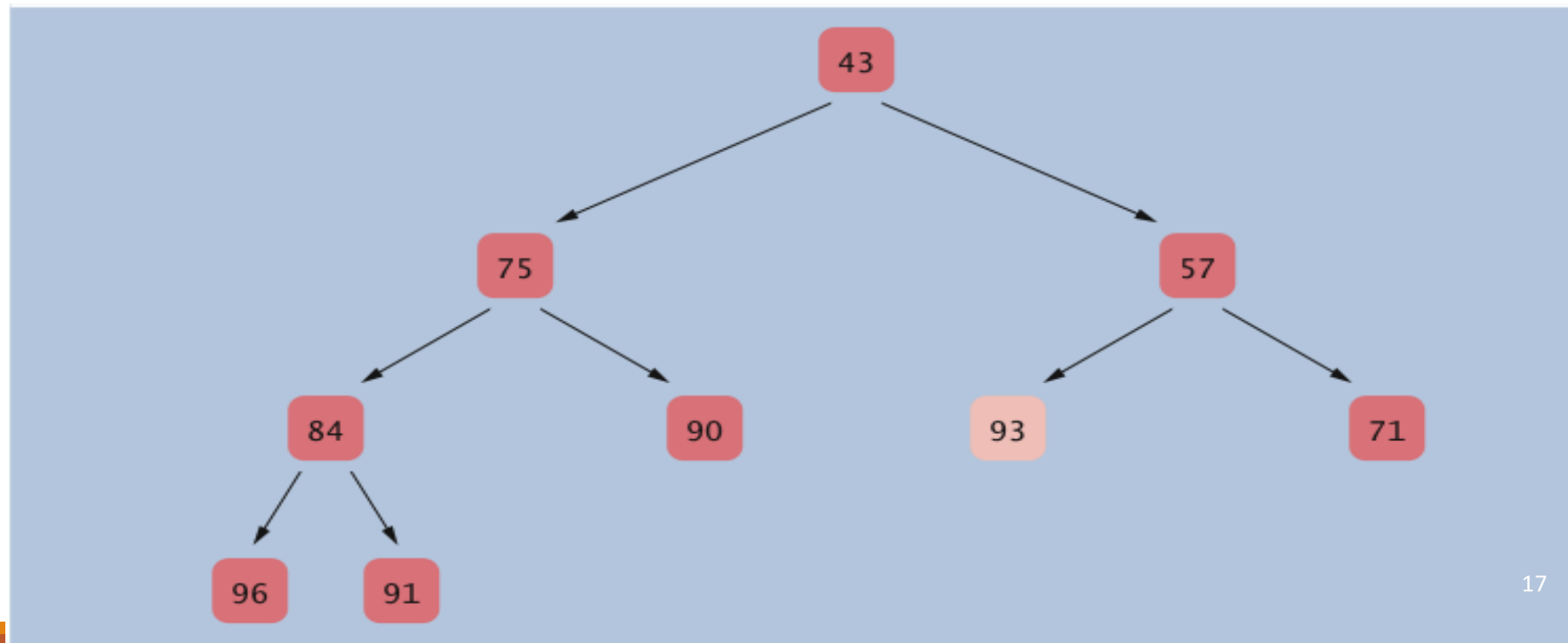
Supprimer la racine (élément Min ou Max)

3. Déplacez le plus petit fils dans la racine. La racine satisfait désormais la propriété de monceau. Répétez le processus avec le fils déplacé. Continuez jusqu'à ce que le fils déplacé n'ait pas de fils plus petits. Cela permet de réparer le monceau.



Supprimer la racine (élément Min ou Max)

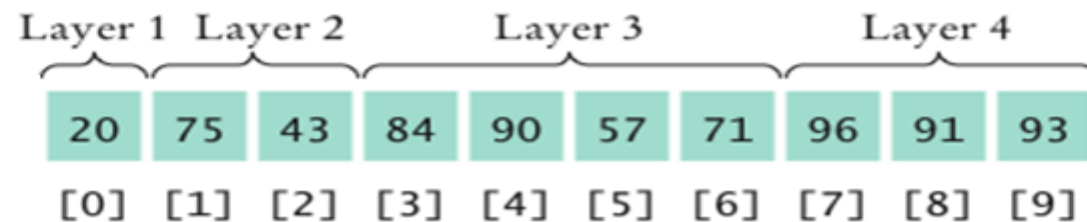
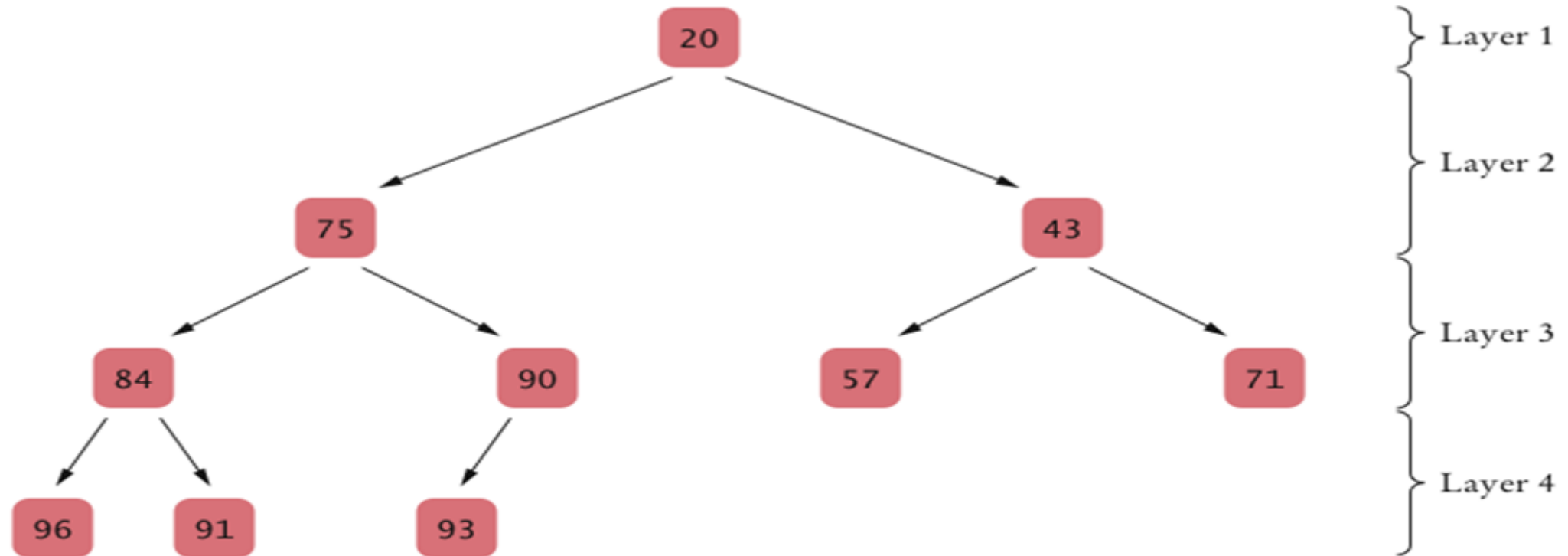
3. Déplacez le plus petit fils dans la racine. La racine satisfait désormais la propriété de monceau. Répétez le processus avec le fils déplacé. Continuez jusqu'à ce que le fils déplacé n'ait pas de fils plus petits. Cela permet de réparer le monceau.



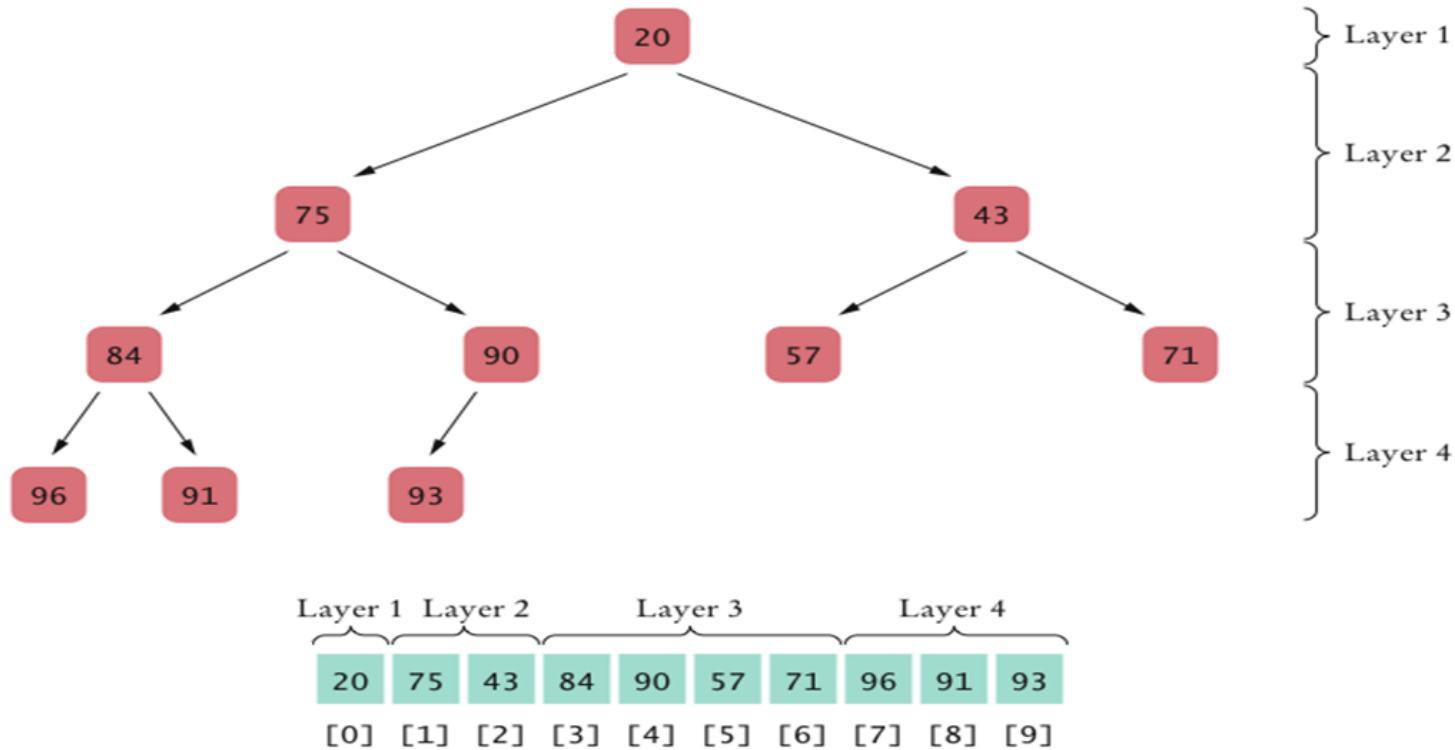
Efficacité d'un monceau

- Opérations d'insertion et de suppression visitent 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 d'un monceau permet de stocker les nœuds de manière efficace dans un tableau

Implémentation et représentation



Implémentation et représentation



- Racine se trouve à l'indice zéro (0);
- Enfants d'un nœud à l'indice i se trouvent aux indices $2i+1$ et $2i+2$ (numérotation de $i=0$)
- Parent d'un nœud à l'indice $i > 0$ se trouve à l'indice $\lfloor (i - 1) / 2 \rfloor$

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 (cont.)

```
19      /**
20         Adds a new element to this heap.
21         @param newElement the element to add
22      */
23      public void add(Comparable newElement)
24      {
25          // Add a new leaf
26          elements.add(null);
27          int index = elements.size() - 1;
28
29          // Demote parents that are larger than the new element
30          while (index > 1
31                && getParent(index).compareTo(newElement) > 0)
32          {
33              elements.set(index, getParent(index));
34              index = getParentIndex(index);
35          }
36
37          // Store the new element into the vacant slot
38          elements.set(index, newElement);
39      }
40
```

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 (cont.)

```
50      /**
51         Removes the minimum element from this heap.
52         @return the minimum element
53     */
54     public Comparable remove()
55     {
56         Comparable minimum = elements.get(1);
57
58         // Remove last element
59         int lastIndex = elements.size() - 1;
60         Comparable last = elements.remove(lastIndex);
61
62         if (lastIndex > 1)
63         {
64             elements.set(1, last);
65             fixHeap();
66         }
67
68         return minimum;
69     }
70
```

Continued

ch16/pqueue/MinHeap.java (cont.)

```
71      /**
72       * Turns the tree back into a heap, provided only the root
73       * node violates the heap condition.
74       */
75      private void fixHeap()
76      {
77          Comparable root = elements.get(1);
78
79          int lastIndex = elements.size() - 1;
80          // Promote children of removed root while they are smaller than last
81
82          int index = 1;
83          boolean more = true;
84          while (more)
85          {
86              int childIndex = getLeftChildIndex(index);
87              if (childIndex <= lastIndex)
88              {
89                  // Get smaller child
90
91                  // Get left child first
92                  Comparable child = getLeftChild(index); Continued
93
```

ch16/pqueue/MinHeap.java (cont.)

```

94         // Use right child instead if it is smaller
95         if (getRightChildIndex(index) <= lastIndex
96             && getRightChild(index).compareTo(child) < 0)
97         {
98             childIndex = getRightChildIndex(index);
99             child = getRightChild(index);
100         }
101
102         // Check if smaller child is smaller than root
103         if (child.compareTo(root) < 0)
104         {
105             // Promote child
106             elements.set(index, child);
107             index = childIndex;
108         }
109         else
110         {
111             // Root is smaller than both children
112             more = false;
113         }
114     }
```

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
```

Monceau (Heaps)

- Exemples au tableau

L'algorithme Heapsort

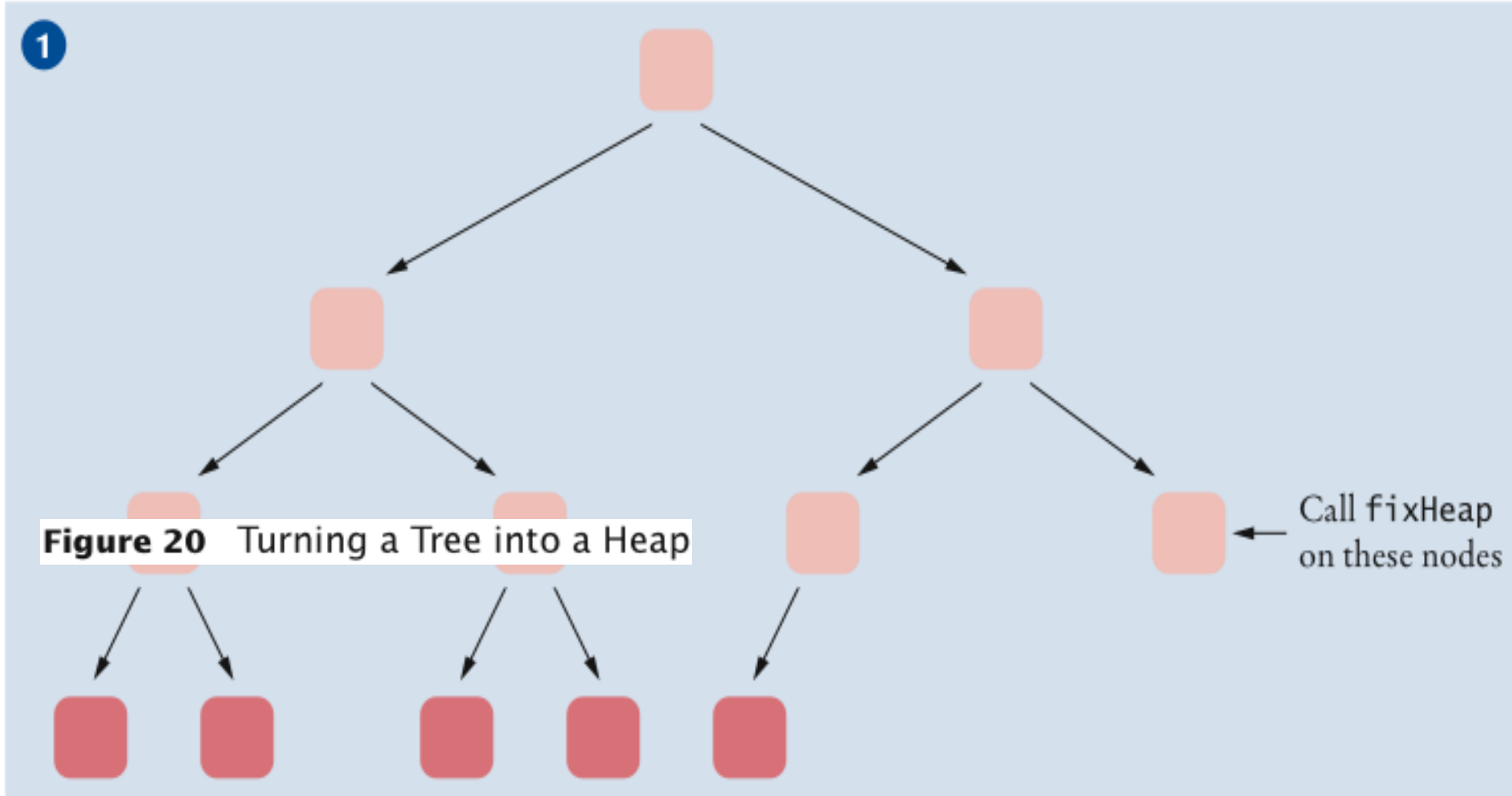
- Basé sur l'insertion des éléments dans un monceau et leurs suppressions dans l'ordre trié
- Cet algorithme est de l'ordre $O(n \log(n))$:
 - *Chaque insertion et suppression est de l'ordre $O(\log(n))$*
 - *Ces étapes sont répétées n fois, une fois pour chaque élément*

L'algorithme Heapsort

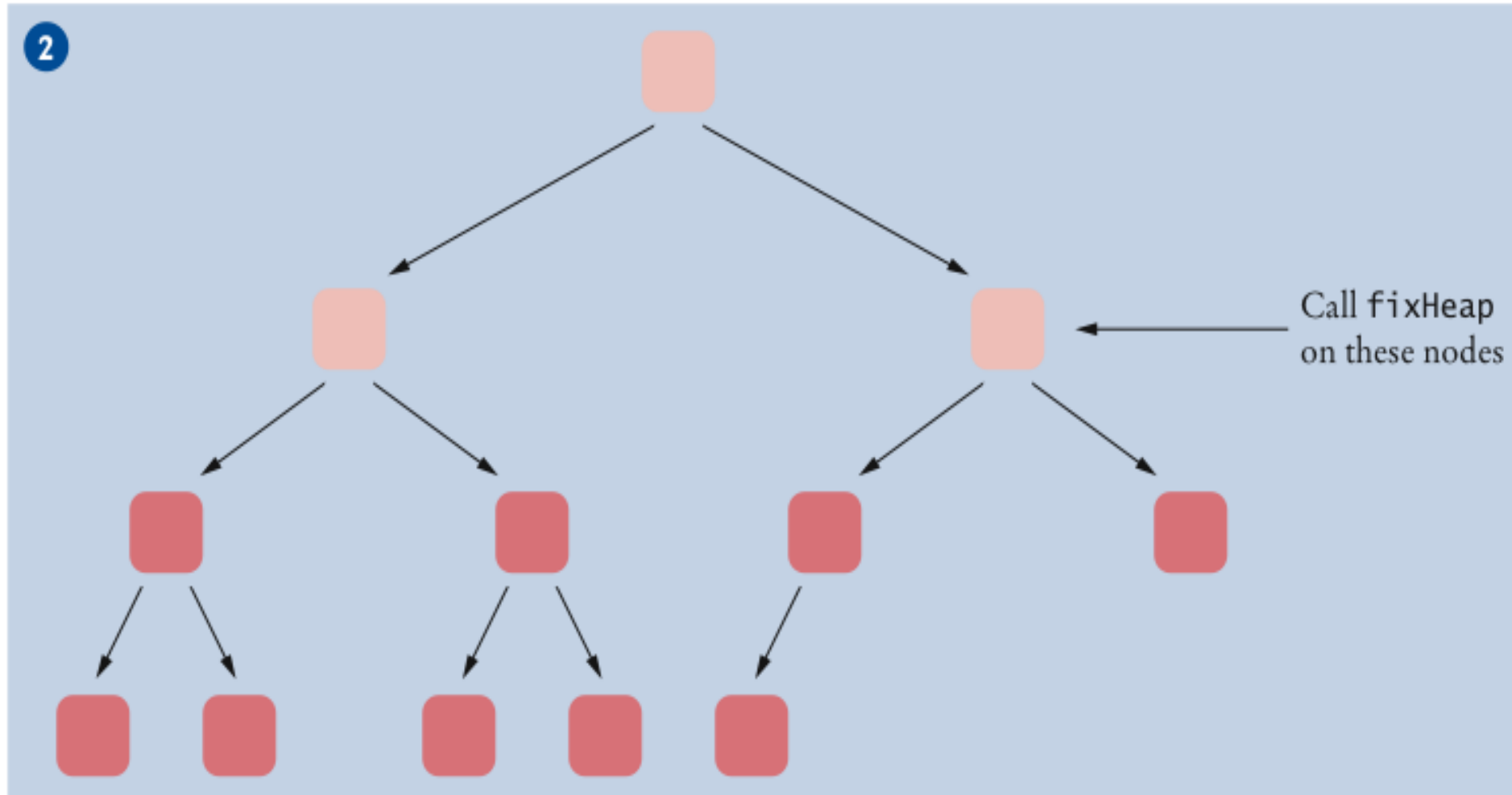
- Peut être fait plus efficacement
 - *Commencer avec une séquence des valeurs stockées dans un tableau et « réparer » la propriété de monceau de manière itérative*
- D'abord transformer les petits sous arbres en un monceau, ensuite réparer les plus grands arbres
- Les arbres de la taille 1 sont automatiquement des monceaux
- 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)
```

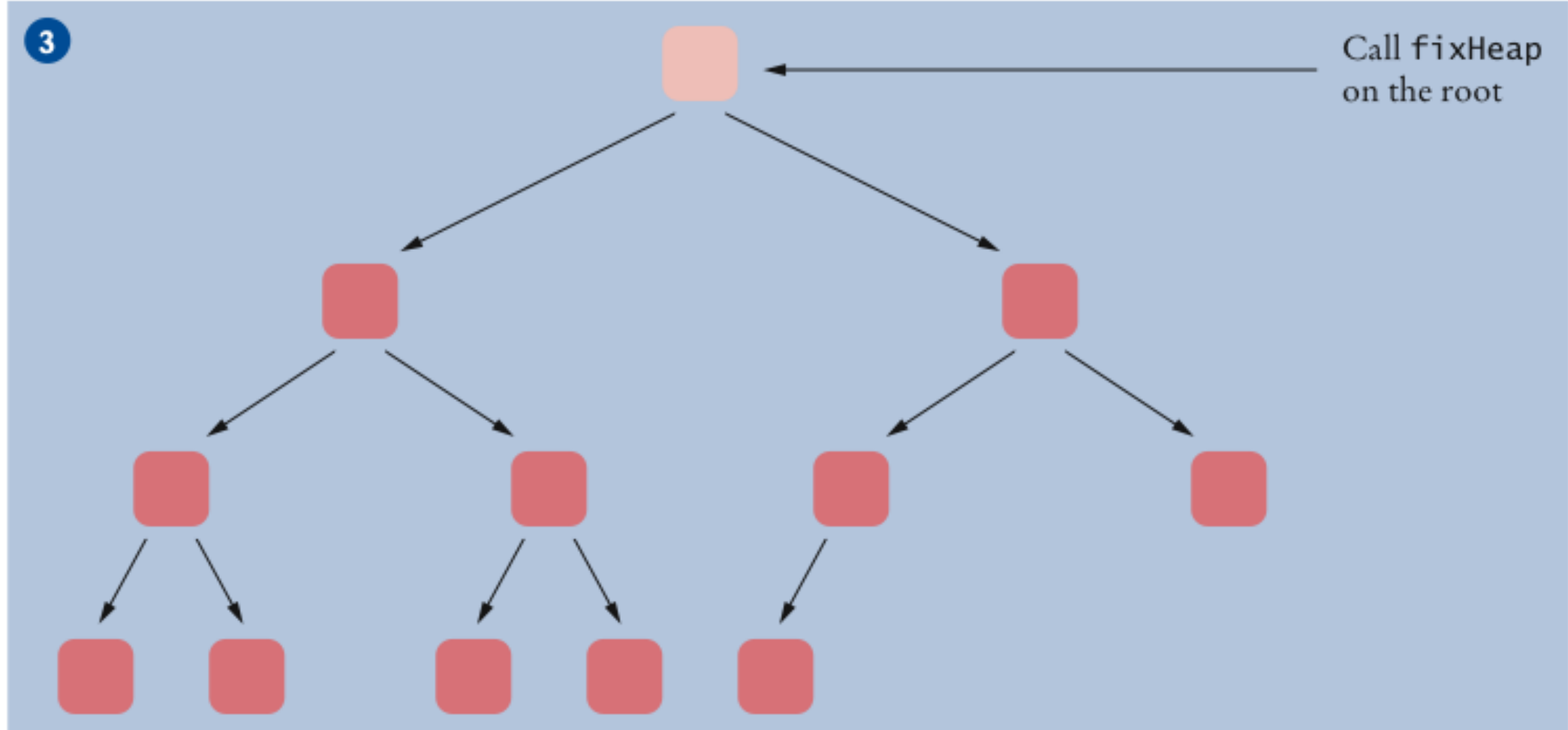
Transformation d'un arbre binaire presque complet en monceau



Transformation d'un arbre binaire presque complet en monceau



Transformation d'un arbre binaire presque complet en monceau



Algorithme HeapSort

- Après avoir transformé le tableau en monceau, répétez l'opération de suppression de l'élément racine.
 - Échangez l'élément racine avec le dernier élément du tableau et réduisez 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 monceau.
 - Vous pouvez utiliser le même tableau pour stocker à la fois le monceau et la séquence des valeurs triées.
 - Utilisez le monceau max plutôt que le monceau min pour construire la séquence des valeurs triées dans l'ordre décroissant.

Utiliser Heapsort pour trier les données

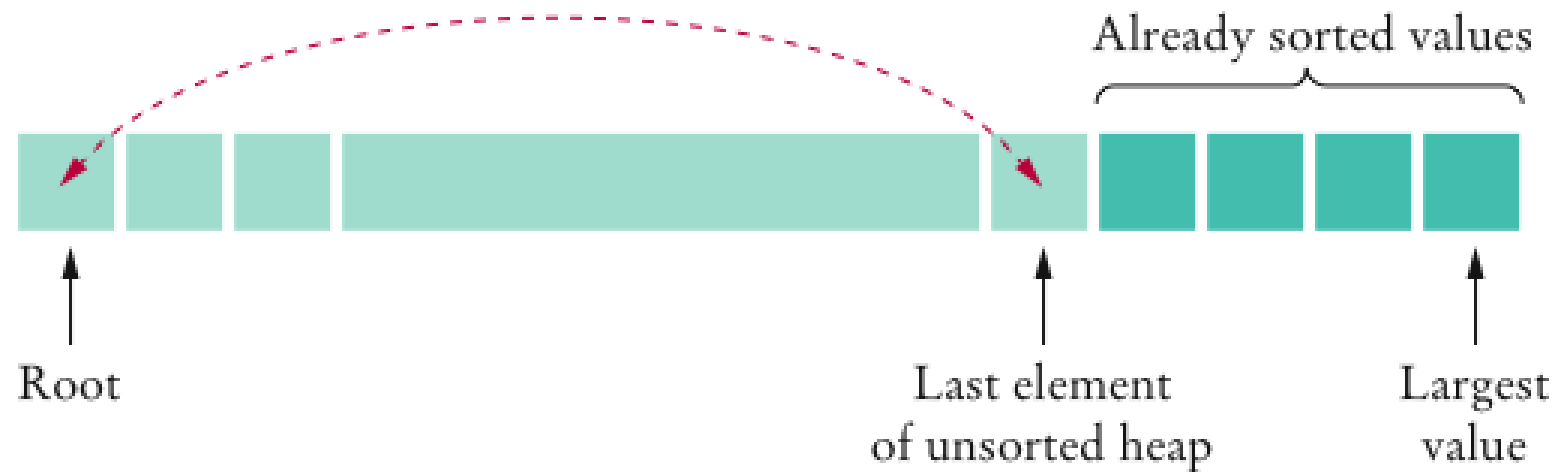


Figure 21 Using Heapsort to Sort an Array

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 }
```

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              }
```

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
```

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
```

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 }
```

Algorithme HeapSort

- Exemple au tableau