CHAPTER 10

Tree Projects

LEARNING OBJECTIVES

When you complete Chapter 10, you will be able to...

- list the rules for a heap or B-tree and determine whether a tree satisfies these rules.
- insert a new element into a heap or remove the largest element by following the insertion algorithm (with reheapification upward) and the removal algorithm (with reheapification downward)
- do a simulation by hand of the algorithms for searching, inserting, and removing an element from a B-tree.
- use the heap data structure to implement a priority queue.
- use the B-tree data structure to implement a set class.
- use Java's DefaultMutableTreeNode and JTree classes in simple programs that use trees.
- recognize which operations have logarithmic worst-case performance on balanced trees.

CHAPTER CONTENTS

The great, dark trees of the Big Woods stood all around the house, and beyond them were other trees and beyond them were more trees.

LAURA INGALLS WILDER
The Little House in the Big Woods

This chapter outlines two programming projects involving trees. The projects are improvements of classes that we've seen before: the priority queue (Section 7.4) and the set (which is like a bag but does not allow more than one copy of an element). Both projects take advantage of *balanced* trees, in which the different subtrees below a node are guaranteed to have nearly the same height. We also discuss Java support for trees and analyze the time performance of tree algorithms, concentrating on a connection between trees and logarithms and explaining the advantages obtained from balanced trees.

10.1 HEAPS

Priority queues were introduced in Section 7.4 as a variation of an ordinary queue in which entries are given priority numbers that are used to determine which entries exit the queue first. This section describes a data structure called a heap that has many applications, including an efficient implentation of a priority queue.

The Heap Storage Rules

A **heap** is a binary tree in which the elements can be compared with each other using *total order semantics*. As we've seen before (Figure 9.11 on page 488), a total order semantics means that all the elements in the class can be placed in a single line, proceeding from smaller to larger along the line. In this way, a heap is similar to a *binary search tree*, but the arrangement of the elements in a heap follows some new rules that are different from a binary search tree:

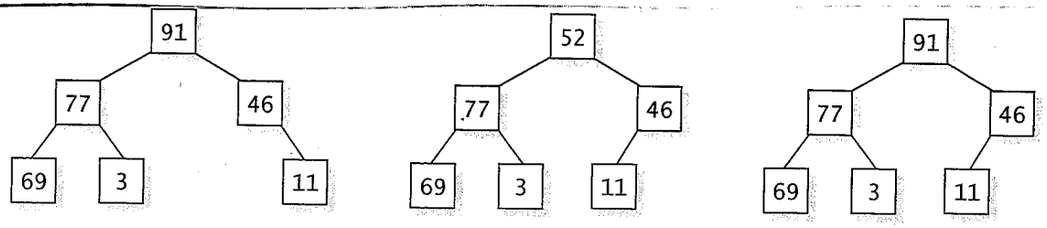
Heap Storage Rules

If the elements of a set can be compared with a total order semantics, then these elements can be stored in a heap. A **heap** is a binary tree in which these two rules are followed:

1. The element contained by each node is greater than or equal to the elements of that node's children.
2. The tree is a complete binary tree so that every level except the deepest must contain as many nodes as possible; at the deepest level, all the nodes are as far left as possible (see "Complete Binary Trees" on page 447).

implement heaps with a fixed array or with an array that grows and shrinks

As an example, suppose elements are integers. Of the three trees on the top of the next page, only one is a heap—which one?



The tree on the left is not a heap because it is not complete—a complete tree must have the nodes on the bottom level as far left as possible. The middle tree is not a heap because one of the nodes (containing 52) has a value that is smaller than its child (which contains 77). The tree on the right *is* a heap.

A heap could be implemented with the binary tree nodes from the classes of Chapter 9. But wait! A heap is a *complete* binary tree, and a complete binary tree is more easily implemented with an array than with the node class (see the section "Array Representation of Complete Binary Trees" on page 449). If we know the maximum size of a heap in advance, then the array implementation can use an array with a single fixed size. If we are uncertain about the heap's maximum size, then the array can grow and shrink as needed.

In Self-Test Exercise 2 on page 517, you'll be asked to write definitions that would support the array implementation of a heap. The rest of this section will show how heaps can be used to implement an efficient priority queue—and we won't worry much about the exact implementation.

The Priority Queue Class with Heaps

Priority queues were introduced in Section 7.4. A priority queue behaves much like an ordinary queue: Elements are placed in the queue and later taken out. But unlike an ordinary queue, each element in a priority queue has an associated number called its priority. When elements leave a priority queue, the highest priority elements always leave first. We have already seen one implementation of a priority queue using the specification from Figure 7.11 on page 390. Now we'll give an alternative implementation that uses the same specification of methods but uses a heap as the underlying data structure to store the queue's elements.

an alternative way to implement priority queues

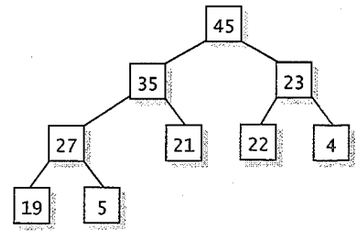
In the heap implementation of a priority queue, each node of the heap contains one element along with the element's priority, and the tree is maintained so that it follows the heap storage rules using the elements' priorities to compare nodes. Therefore:

1. The element contained by each node has a priority that is greater than or equal to the priorities of the elements of that node's children.
2. The tree is a complete binary tree.

We'll focus on two priority queue operations: adding a new element (along with its priority) and removing the element with the highest priority. In both operations, we must ensure that the structure remains a heap when the operation concludes. Also, both operations can be described without worrying about precisely how we've implemented the underlying heap.

Adding an Element to a Heap

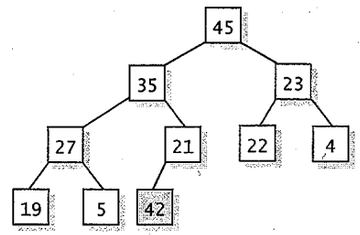
Let's start with the operation that adds a new element. As an example, suppose we already have nine elements that are arranged in a heap with the following priorities:



In this diagram, we have shown only the elements' priorities. In an actual priority queue, each element would also have some additional data in each node.

Suppose we are adding a new element with a priority of 42. The first step is to add this element in a way that keeps the binary tree complete. In this case, the new element will be the left child of the node with priority 21:

add the new element in a way that keeps the binary tree complete



But now the structure is no longer a heap since the node with priority 21 has a child with a higher priority. The algorithm fixes this problem by causing the new element (with priority 42) to rise upward until it reaches an acceptable location. This is accomplished by swapping the new element with its parent:

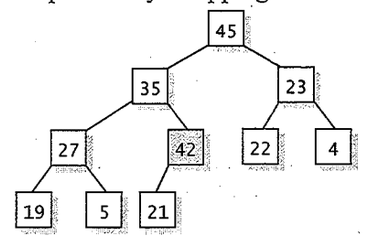


FIGURE 10.1 Adding an Element to a Priority Queue

Pseudocode for Adding an Element

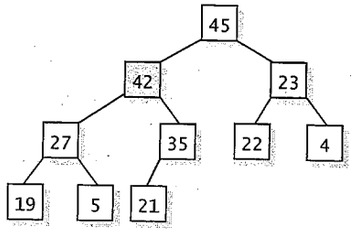
The priority queue has been implemented as a heap.

1. Place the new element in the heap in the first available location. This keeps the structure as a complete binary tree, but it might no longer be a heap since the new element might have a higher priority than its parent.

2. `while` (the new element has a priority that is higher than its parent)
   Swap the new element with its parent.

Notice that the process in Step 2 will stop when the new element reaches the root or when the new element's parent has a priority that is at least as high as the new element's priority.

The new element is *still* bigger than its parent, so a second swap is done:



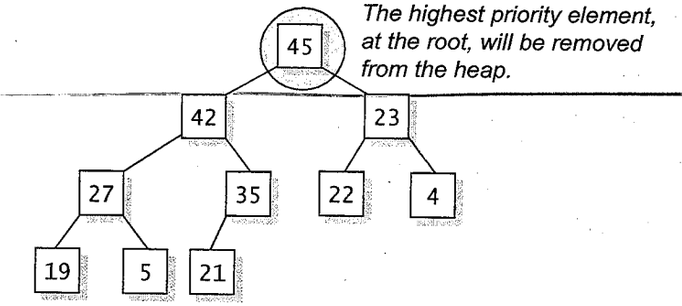
reheapification upward

Now the new element stops rising because its parent (with priority 45) has a higher priority than the new element. In general, the "new element rising" stops when the new element has a parent with a higher or equal priority or when the new element reaches the root. The rising process is called **reheapification upward**.

The steps for adding an element are outlined in Figure 10.1. Some of the details depend on how the underlying heap is implemented. For example, if the heap is implemented with a nongrowing array, then the first step must check that there is room for a new element. With an array that grows, the first step might need to increase the size of the array.

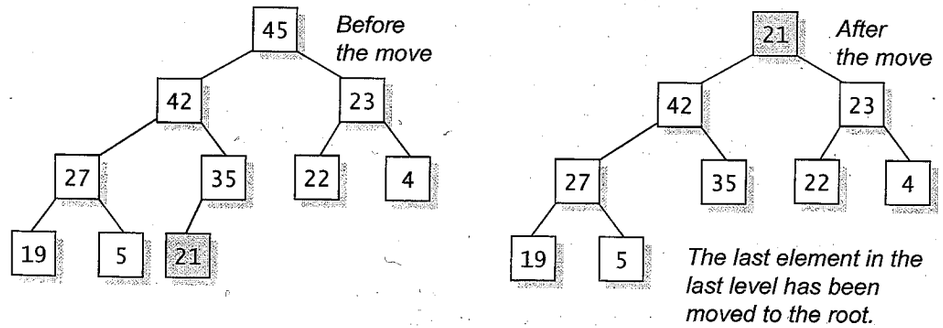
Removing an Element from a Heap

When an element is removed from a priority queue, we must always remove the element with the highest priority—the element that stands "on top of the heap." For example, consider this heap of 10 priorities:

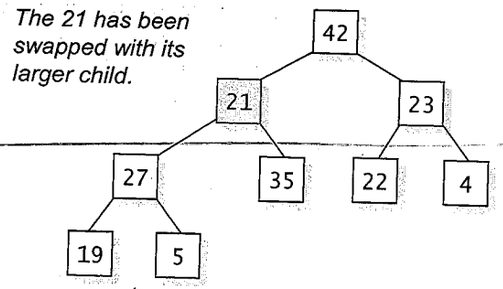


The element at the root, with priority 45, will be removed, and this is the element that is returned by the operation. But here we are ignoring a potential problem: There might be several elements with the highest priority. As described in Section 7.4, when several elements have equal priority, the elements should exit the priority queue in first-in/first-out (FIFO) order. Our adding and deleting mechanisms do not provide a way to determine which of several elements arrived first. So, in this discussion, we will suspend the requirement that elements with equal priority be treated in FIFO order. We will address this problem in Programming Project 1 on page 556.

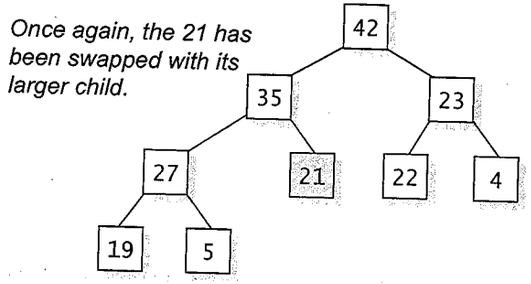
The problem of removing the root from a heap remains. During the removal, we must ensure that the resulting structure remains a heap. If the root is the only element in the heap, then there is really no more work to do except to decrement the instance variable that is keeping track of the size of the heap. But if there are other elements in the tree, then the tree must be rearranged because a heap is not allowed to run around without a root. The rearrangement begins by moving the last element in the last level up to the root, like this:



The structure is now a complete tree, but it is not a heap because the root is smaller than its children. To fix this, we can swap the root with its larger child, as shown here:



The structure is not yet a heap, so again we swap the out-of-place node with its larger child, giving us this tree:



reheapification downward

At this point, the node that has been sinking has reached a leaf, so we can stop and the structure is a heap. The process would also stop when the sinking node no longer has a child with a higher priority. This process, called **reheapification downward**, is summarized in Figure 10.2.

FIGURE 10.2 Removing an Element from a Priority Queue

Pseudocode for Removing an Element

The priority queue has been implemented as a heap.

1. Copy the element at the root of the heap to the variable used to return a value.
2. Copy the last element in the deepest level to the root and then take this last node out of the tree. This element is called the "out-of-place" element.
3. `while` (the out-of-place element has a priority that is lower than one of its children)
   Swap the out-of-place element with its highest-priority child.
4. Return the answer that was saved in Step 1.

Notice that the process in Step 3 will stop when the out-of-place element reaches a leaf or when the out-of-place element has a priority that is at least as high as its children.

Heaps have several other impressive applications, including collision detection in programs to simulate atomic particles, hidden surface removal in graphics programs, and a sorting algorithm that is discussed in Chapter 12.

Self-Test Exercises for Section 10.1

1. The most appropriate way to implement a heap is with an array rather than a linked structure. Why?
2. Define the instance variables for a Java class that would be appropriate for a priority queue implemented as a heap. Each node should have an integer priority and a piece of data that is a Java `Object`.
3. In the description of reheapification downward, we specified that the out-of-place element must be swapped with the larger of its two children. What goes wrong if we swap with the smaller child instead?
4. Start with an empty heap and enter 10 elements with priorities 1 through 10. Draw the resulting heap.
5. Remove three elements from the heap you created in the previous exercise. Draw the resulting heap.

10.2 B-TREES

Binary search trees were used in Section 9.5 to implement a bag ADT, but the efficiency of binary search trees can go awry. This section explains the potential problem and shows one way to fix it.

The Problem of Unbalanced Trees

Let's create a troublesome binary search tree of integers. The first number is 1, and then we add 2, 3, 4, and 5 in that order. The result appears in Figure 10.3(a). Perhaps you've spotted the problem. Suppose next we add 6, 7, 8, 9, and 10 in that order, ending up with Figure 10.3(b). The problem is that the levels of the tree are only sparsely filled, resulting in long, deep paths and defeating the purpose of binary trees in the first place. For example, if we are searching Figure 10.3(b) for the number 12, then we'll end up looking at every node in the tree. In effect, we are no better off

FIGURE 10.3 Two Troublesome Search Trees