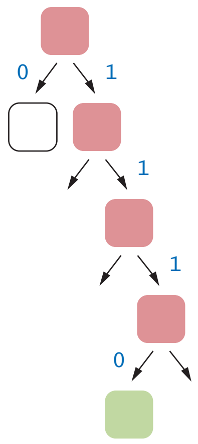
WORKED EXAMPLE 17.1 Building a Huffman Tree

A Huffman code encodes symbols into sequences of zeroes and ones, so that the most frequently occurring symbols have the shortest encodings. The symbols can be characters of the alphabet, but they can also be something else. For example, when images are compressed using a Huffman encoding, the symbols are the colors that occur in the image.

Problem Statement Encode a child's painting like the one below by building a Huffman tree with an optimal encoding. Most of the pixels are white (50%), there are lots of orange (20%) and pink (20%) pixels, and small amounts of yellow (5%), blue (3%), and green (2%).

Charlotte and Emily Horstmann.

We want a short code (perhaps 0) for white and a long one (perhaps 1110) for green. Such a variable-length encoding minimizes the overall length of the encoded data.



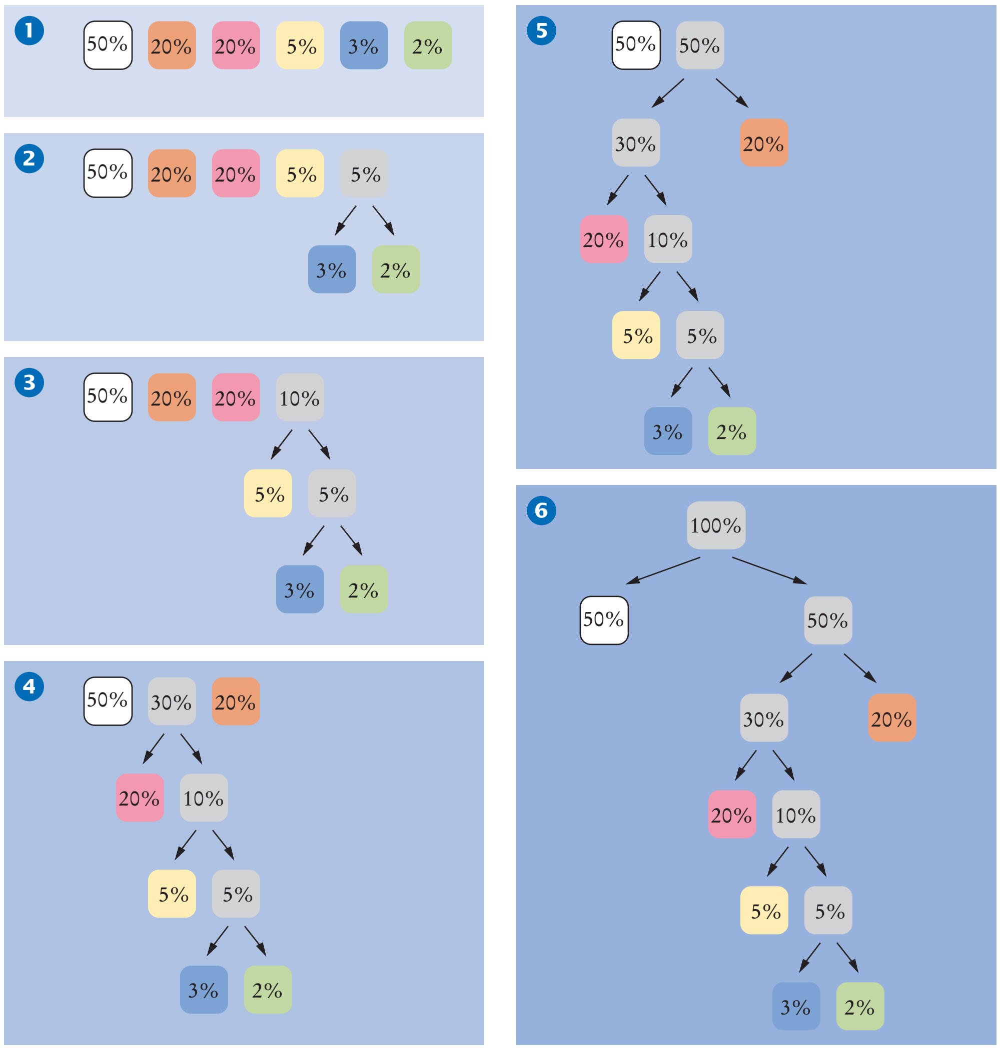
The challenge is to build a tree that yields an optimal encoding. The following algorithm, developed by David Huffman when he was a graduate student, achieves this task.

Make a tree node for each symbol to be encoded. Each node has an instance variable for the frequency.

```
Add all nodes to a priority queue.
While there are two nodes left
    Remove the two nodes with the smallest frequencies.
    Make them children of a parent whose frequency is the sum of the child frequencies.
    Add the parent to the priority queue.
```

The remaining node is the root of the Huffman tree.

The following figure shows the algorithm applied to our sample data.



After the tree has been constructed, the frequencies are no longer needed.

The resulting code is

```
White    0
Pink     100
Yellow   1010
Blue     10110
Green    10111
Orange   11
```

Note that this is not a code for encrypting information. The code is known to all; its purpose is to compress data by using the shortest codes for the most common symbols. Also note that the code has the property that no codeword is the prefix of another codeword. For example, because white is encoded as `0`, no other codeword starts with `0`, and because orange is 11, no other codeword starts with 11.

The implementation is very straightforward. The `Node` class needs instance variables for holding the symbol to be encoded (which we assume to be a character) and its frequency. It must also implement the `Comparable` interface so that we can put nodes into a priority queue:

```
class Node implements Comparable<Node>
{
   public char character;
   public int frequency;
   public Node left;
   public Node right;
   public int compareTo(Node other) { return frequency - other.frequency; }
}
```

When constructing a tree, we need the frequencies for all characters. The tree constructor receives them in a `Map<Character, Integer>`. The frequencies need not be percentages. They can be counts from a sample text.

First, we make a node for each character to be encoded, and add each node to a priority queue:

```
PriorityQueue<Node> nodes = new PriorityQueue<>();
for (char ch : frequencies.keySet())
{
   Node newNode = new Node();
   newNode.character = ch;
   newNode.frequency = frequencies.get(ch);
   nodes.add(newNode);
}
```

Then, following the algorithm, we keep combining the two nodes with the lowest frequencies:

```
while (nodes.size() > 1)
{
   Node smallest = nodes.remove();
   Node nextSmallest = nodes.remove();
   Node newNode = new Node();
   newNode.frequency = smallest.frequency + nextSmallest.frequency;
   newNode.left = smallest;
   newNode.right = nextSmallest;
   nodes.add(newNode);
}
root = nodes.remove();
```

Decoding a sequence of zeroes and ones is very simple: just follow the links to the left or right until a leaf is reached. Note that each node has either two or no children, so we only need to check whether one of the children is `null` to detect a leaf. Here we use strings of `0` or `1` characters, not actual bits, to keep the demonstration simple.

```
public String decode(String input)
{
   String result = "";
   Node n = root;
   for (int i = 0; i < input.length(); i++)
   {
      char ch = input.charAt(i);
      if (ch == '0')
      {
         n = n.left;
      }
      else
      {
         n = n.right;
         }
         if (n.left == null) // n is a leaf
         {
            result = result + n.character;
            n = root;
         }
      }
      return result;
   }
```

The tree is not useful for efficient encoding because we don't want to search through the leaves each time we encode a character. Instead, we will just compute a map that maps each character to its encoding. This can be done by recursively visiting the subtrees and remembering the current prefix, that is, the path to the root of the subtree. Follow the left or right children, adding a `0` or 1 to the end of that prefix, or, if the subtree is a leaf, simply add the character and the prefix to the map:

```
   class Node implements Comparable<Node>
   {
      . . .
      public void fillEncodingMap(Map<Character, String> map, String prefix)
      {
         if (left == null) // It's a leaf
         {
            map.put(character, prefix);
         }
         else
         {
            left.fillEncodingMap(map, prefix + "0");
            right.fillEncodingMap(map, prefix + "1");
         }
      }
   }
```

This recursive helper method is called from the `HuffmanTree` class:

```
   public class HuffmanTree
   {
      . . .
      public Map<Character, String> getEncodingMap()
      {
         Map<Character, String> map = new HashMap<>();
         if (root != null) { root.fillEncodingMap(map, ""); }
         return map;
      }
   }
```

The demonstration program (in your `ch17/worked_example_1` code folder) computes the Huffman encoding for the Hawaiian language, which was chosen because it uses fewer letters than most other languages. The frequencies were obtained from a text sample on the Internet.