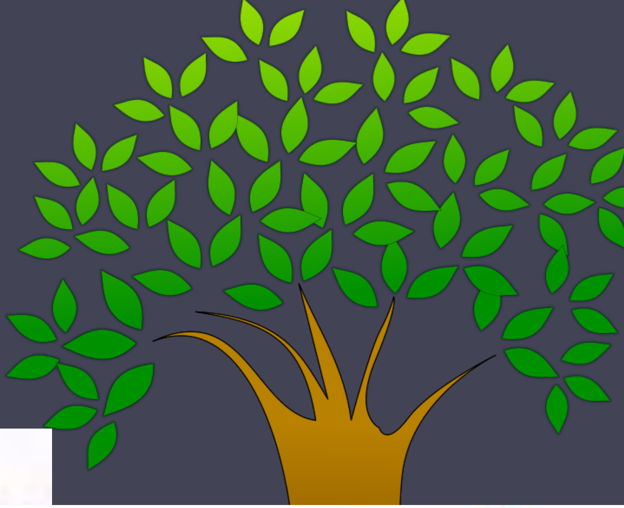
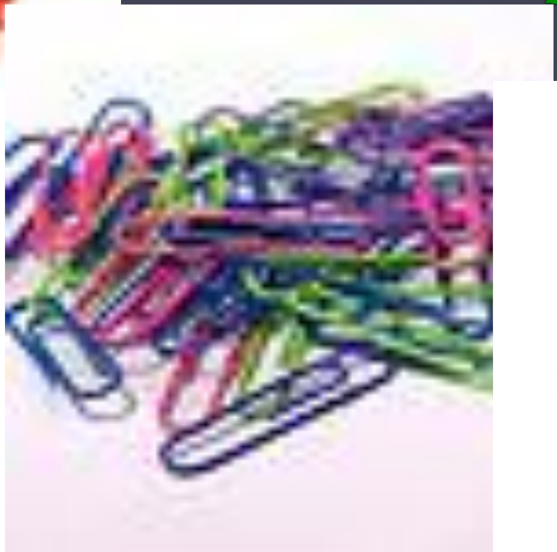




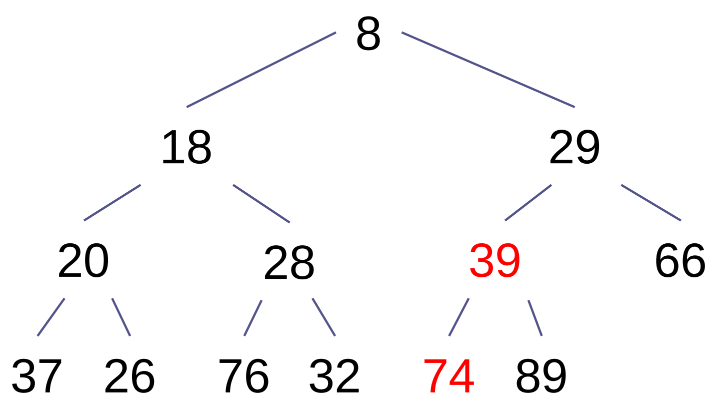

# LECTURE 24

## Heaps and Priority Queues

# Assignment

- Finish Chapter 6, and do quick-check exercises on p. 355

# Implementing a Heap (cont.)




A node at position *c* can find its parent at = $\lfloor (c-1)/2 \rfloor$ =

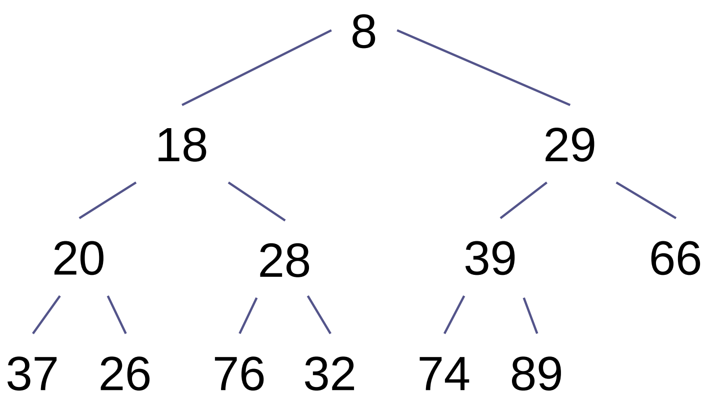
$$\begin{cases} \frac{c-1}{2}, & c \text{ is odd} \\ \frac{c-2}{2}, & c \text{ is even} \end{cases}$$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 |

Parent

Child

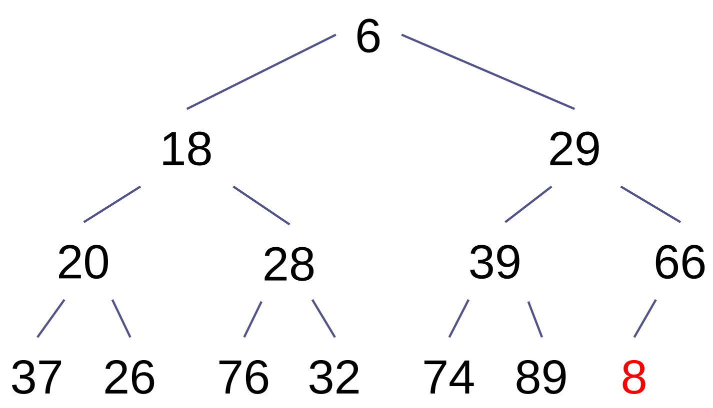
# Inserting into a Heap Implemented as an `ArrayList`




1. Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 | |

# Inserting into a Heap Implemented as an `ArrayList` (cont.)



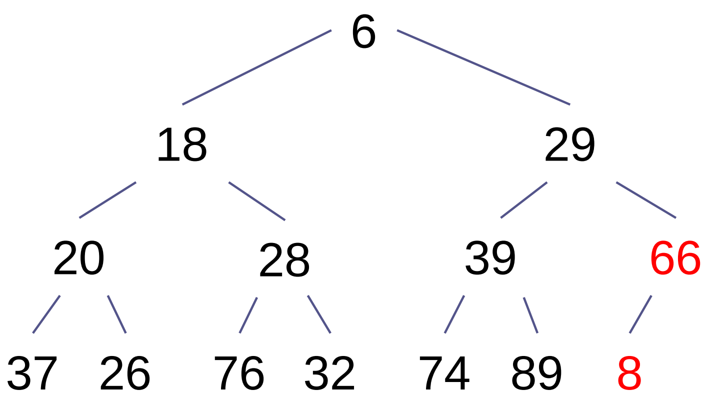

1. Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 | 8 |

**`Child`**

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

2. Set `parent` to `(child – 1)/ 2`

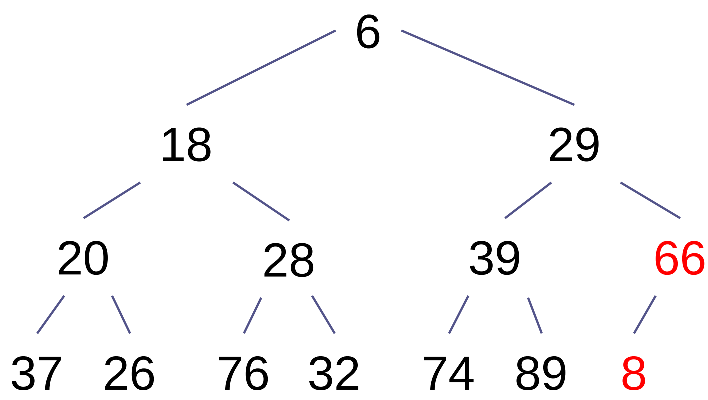



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 | 8 |

**`Parent`**

**`Child`**

# Inserting into a Heap Implemented as an `ArrayList` (cont.)



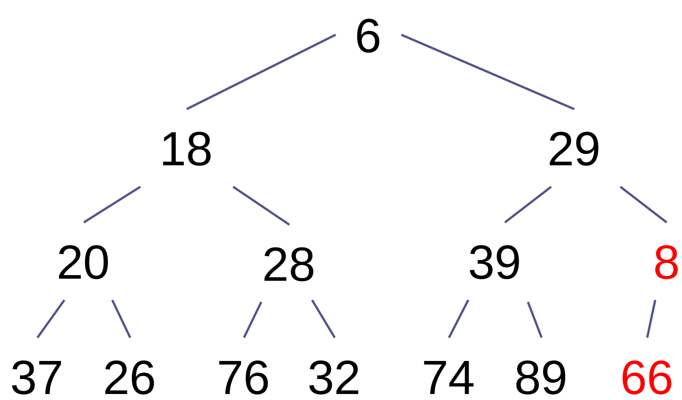

3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 | 8 |

Parent

Child

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

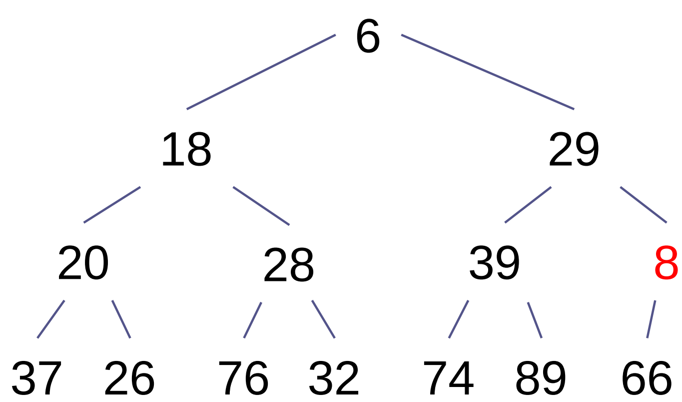



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 8 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

`Parent` `Child`

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

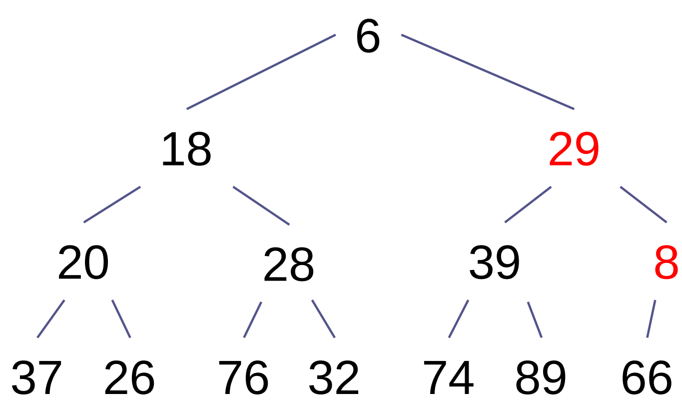



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 8 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

**Parent** **Child**

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

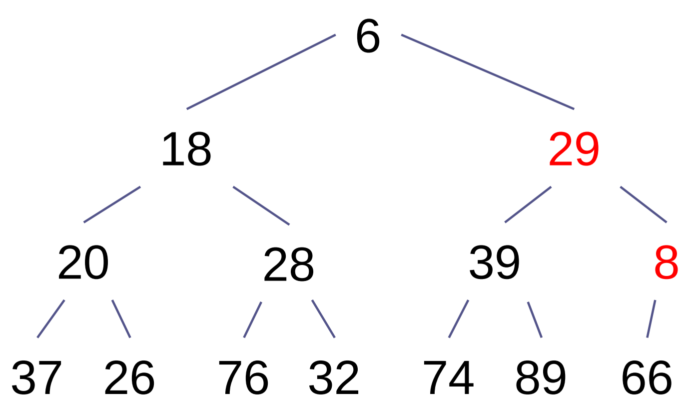



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 8 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

**`Parent`** **`Child`**

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

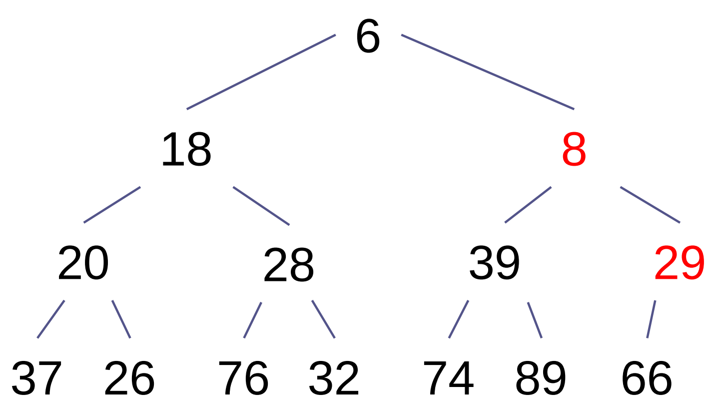



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 29 | 20 | 28 | 39 | 8 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

**`Parent`** **`Child`**

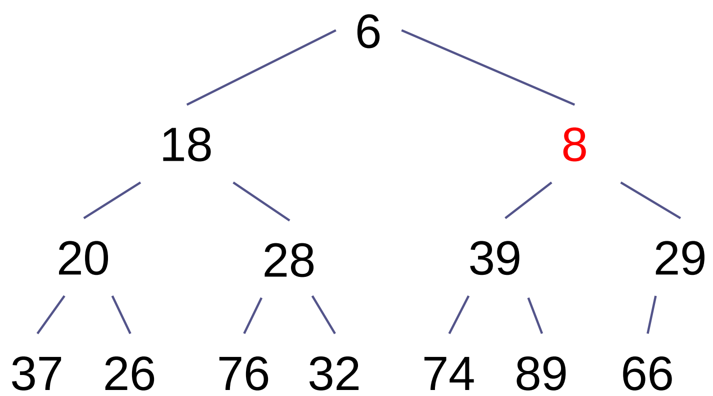
# Inserting into a Heap Implemented as an `ArrayList` (cont.)




3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

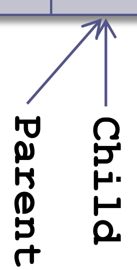
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 8 | 20 | 28 | 39 | 29 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

# Inserting into a Heap Implemented as an `ArrayList` (cont.)

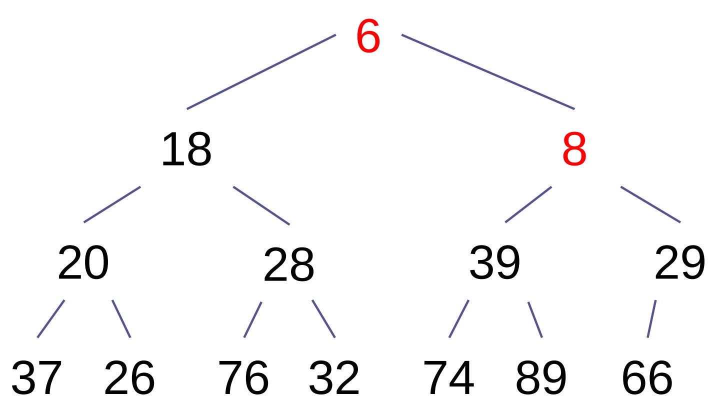



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 8 | 20 | 28 | 39 | 29 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

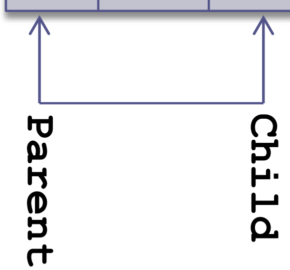


# Inserting into a Heap Implemented as an `ArrayList` (cont.)

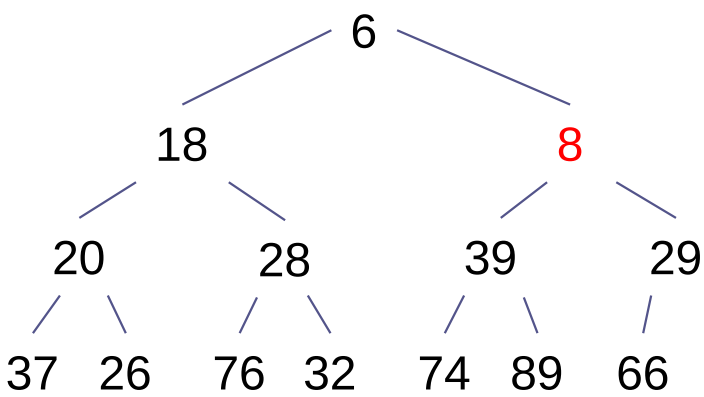



3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 8 | 20 | 28 | 39 | 29 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |



# Inserting into a Heap Implemented as an `ArrayList` (cont.)




3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 8 | 20 | 28 | 39 | 29 | 37 | 26 | 76 | 32 | 74 | 89 | 66 |

# Removal from a Heap Implemented as an `ArrayList`

Removing an Element from a Heap Implemented as an ArrayList

1. Remove the last element (i.e., the one at size() – 1) and set the item at 0 to this value.
2. Set `parent` to 0.
3. **`while (true)`**
4. Set `leftChild` to (2 * `parent`) + 1 and `rightChild` to `leftChild` + 1.
5. **`if leftChild >= table.size()`**
6. Break out of loop.
7. Assume `minChild` (the smaller child) is `leftChild`.
8. **`if`** `rightChild < table.size() and table[rightChild] < table[leftChild]`
9. Set `minChild` to `rightChild`.
10. **`if table[parent] > table[minChild]`**
11. Swap `table[parent]` and `table[minChild]`.
12. Set `parent` to `minChild`.

    **`else`**

13. Break out of loop.

# Performance of the Heap

- `remove` traces a path from the root to a leaf
- `insert` traces a path from a leaf to the root
- This requires at most *h* steps where *h* is the height of the tree
- The largest *full* tree of height *h* has $2^h$-1 nodes
- The smallest *complete* tree of height *h* has $n = 2^{(h-1)}$ nodes (and thus $h = \log_2 n + 1$)
- Both `insert` and `remove` are O(log *n*)

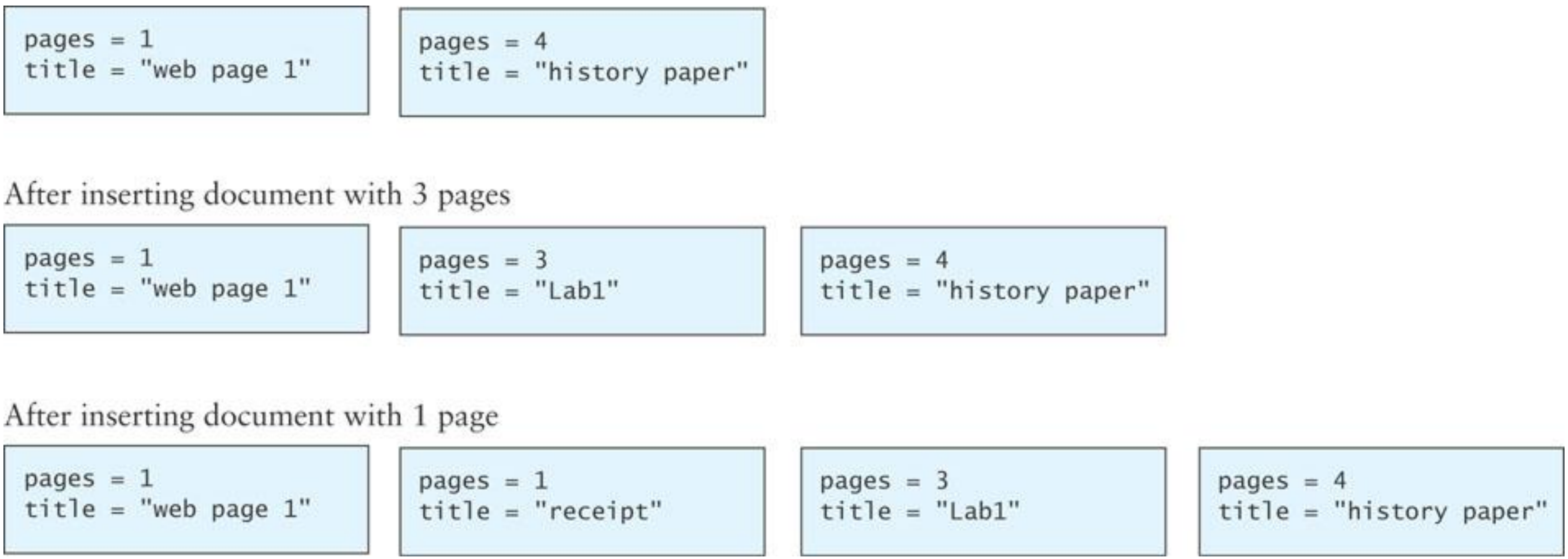
# Priority Queues

- The heap is used to implement a special kind of queue called a priority queue
- For the purposes of this class
  - We will not create a `Heap` interface or code a class that implements it
  - Instead, we will incorporate its algorithms when we implement a *priority queue* class and heapsort

# Priority Queues

- In a print queue, sometimes it is quicker to print a short document that arrived after a very long document
- A *priority queue* is a data structure in which only the highest-priority item is served first (as opposed to the first item entered)

# Insertion into a Priority Queue



# PriorityQueue Class

- Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4.

| Method | Behavior |
|---|---|
| `boolean offer(E item)` | Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted. |
| `E remove()` | Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`. |
| `E poll()` | Removes the smallest entry and returns it. If the queue is empty, returns **`null`**. |
| `E peek()` | Returns the smallest entry without removing it. If the queue is empty, returns **`null`**. |
| `E element()` | Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`. |

# Using a Heap as the Basis of a Priority Queue

- In a priority queue, just as in a heap, the smallest item is removed first
- Because heap insertion and removal are each O(log *n*), a heap can be the basis of a very efficient implementation of a priority queue
- While the `java.util.PriorityQueue` uses an `Object[]` array, the book uses an `ArrayList` for a custom priority queue, `KWPriorityQueue`

# Design of a `KWPriorityQueue` Class

| Data Field | Attribute |
|---|---|
| `ArrayList<E> theData` | An `ArrayList` to hold the data. |
| `Comparator<E> comparator` | An optional object that implements the `Comparator<E>` interface by providing a `compare` method. |
| **Method** | **Behavior** |
| `KWPriorityQueue()` | Constructs a heap-based priority queue that uses the elements' natural ordering. |
| `KWPriorityQueue (Comparator<E> comp)` | Constructs a heap-based priority queue that uses the `compare` method of `Comparator comp` to determine the ordering of the elements. |
| `private int compare(E left, E right)` | Compares two objects and returns a negative number if object `left` is less than object `right`, zero if they are equal, and a positive number if object `left` is greater than object `right`. |
| `private void swap(int i, int j)` | Exchanges the object references in `theData` at indexes `i` and `j`. |

# **Design of a** KWPriorityQueue **Class** (cont.)

```
import java.util.*;

/** The KWPriorityQueue implements the Queue interface
    by building a heap in an ArrayList. The heap is structured
    so that the "smallest" item is at the top.
*/
public class KWPriorityQueue<E> extends AbstractQueue<E>
                                implements Queue<E> {

// Data Fields
/** The ArrayList to hold the data. */
private ArrayList<E> theData;
/** An optional reference to a Comparator object. */
Comparator<E> comparator = null;

// Methods
// Constructor
public KWPriorityQueue() {
    theData = new ArrayList<E>();
}
. . .
```

# `offer` Method

```java
/** Insert an item into the priority queue.
    pre: The ArrayList theData is in heap order.
    post: The item is in the priority queue and
          theData is in heap order.
    @param item The item to be inserted
    @throws NullPointerException if the item to be inserted is null.
*/
@Override
public boolean offer(E item) {
    // Add the item to the heap.
    theData.add(item);
    // child is newly inserted item.
    int child = theData.size() - 1;
    int parent = (child - 1) / 2; // Find child's parent.
    // Reheap
    while (parent >= 0 && compare(theData.get(parent),
                                  theData.get(child)) > 0) {
        swap(parent, child);
        child = parent;
        parent = (child - 1) / 2;
    }
    return true;
}
```

A little assignment: Write a recursive method for that!

# `poll` Method

```
/** Remove an item from the priority queue
    pre: The ArrayList theData is in heap order.
    post: Removed smallest item, theData is in heap order.
    @return The item with the smallest priority value or null if empty.
*/
@Override
public E poll() {
    if (isEmpty()) {
          return null;
    }
    // Save the top of the heap.
    E result = theData.get(0);
    // If only one item then remove it.
    if (theData.size() == 1) {
        theData.remove(0);
        return result;
}
```

# `poll` Method (cont.)

```
    /* Remove the last item from the ArrayList and place it into
       the first position. */
    theData.set(0, theData.remove(theData.size() - 1));
    // The parent starts at the top.
    int parent = 0;
    while (true) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= theData.size()) {
            break; // Out of heap.
        }
        int rightChild = leftChild + 1;
        int minChild = leftChild; // Assume leftChild is smaller.
        // See whether rightChild is smaller.
        if (rightChild < theData.size()
            && compare(theData.get(leftChild),
                       theData.get(rightChild)) > 0) {
            minChild = rightChild;
        }
        // assert: minChild is the index of the smaller child.
        // Move smaller child up heap if necessary.
        if (compare(theData.get(parent),
                    theData.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        } else { // Heap property is restored.
            break;
        }
    }
    return result;
}
```

# Other Methods

- The `iterator` and `size` methods are implemented via delegation to the corresponding `ArrayList` methods
- Method `isEmpty` tests whether the result of calling method `size` is `0` and is inherited from class `AbstractCollection`
- The implementations of methods `peek` and `remove` are left as exercises

# Using a `Comparator`

- To use an ordering that is different from the natural ordering, provide a constructor that has a `Comparator<E>` parameter

```
/** Creates a heap-based priority queue with the specified initial
    capacity that orders its elements according to the specified
    comparator.
    @param cap The initial capacity for this priority queue
    @param comp The comparator used to order this priority queue
    @throws IllegalArgumentException if cap is less than 1
*/
public KWPriorityQueue(Comparator<E> comp) {
    if (cap < 1)
        throw new IllegalArgumentException();
    theData = new ArrayList<E>();
    comparator = comp;
}
```

# compare Method

- If data field comparator references a Comparator<E> object, method compare delegates the task to the objects compare method
- If comparator is null, it will delegate to method compareTo

# compare Method (cont.)

```
/** Compare two items using either a Comparator object's compare method
    or their natural ordering using method compareTo.
    pre: If comparator is null, left and right implement Comparable<E>.
    @param left One item
    @param right The other item
    @return Negative int if left less than right,
            0 if left equals right,
            positive int if left > right
    @throws ClassCastException if items are not Comparable
*/
private int compare(E left, E right) {
    if (comparator != null) { // A Comparator is defined.
        return comparator.compare(left, right);
    } else {                    // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

# `PrintDocuments` Example

- The class `PrintDocument` is used to define documents to be printed on a printer
- We want to order documents by a value that is a function of both size and time submitted
- In the client program, use

```
Queue printQueue =
    new PriorityQuene(new ComparePrintDocuments());
```

# `PrintDocuments` Example (cont.)

**LISTING 6.8**

Compare PrintDocuments.java

```
import java.util.Comparator;

/** Class to compare PrintDocuments based on both
    their size and time stamp.
*/
public class ComparePrintDocuments implements Comparator<PrintDocument> {
    /** Weight factor for size. */
    private static final double P1 = 0.8;
    /** Weight factor for time. */
    private static final double P2 = 0.2;

    /** Compare two PrintDocuments.
        @param left The left-hand side of the comparison
        @param right The right-hand side of the comparison

        @return -1 if left < right; 0 if left == right;
                and +1 if left > right
     */
     public int compare(PrintDocument left, PrintDocument right) {
         return Double.compare(orderValue(left), orderValue(right));
     }

     /** Compute the order value for a print document.
         @param pd The PrintDocument
         @return The order value based on the size and time stamp
     */
     private double orderValue(PrintDocument pd) {
         return P1 * pd.getSize() + P2 * pd.getTimeStamp();
     }
 }
```

# Huffman Trees

# Huffman Trees used for *compression*

- A Huffman tree can be implemented using a binary tree and a PriorityQueue
- A straight binary encoding of an alphabet assigns a unique binary number to each symbol in the alphabet
  - Unicode is an example of such a coding
- A string “go eagles” requires 144 bits in Unicode but only 38 bits using Huffman coding

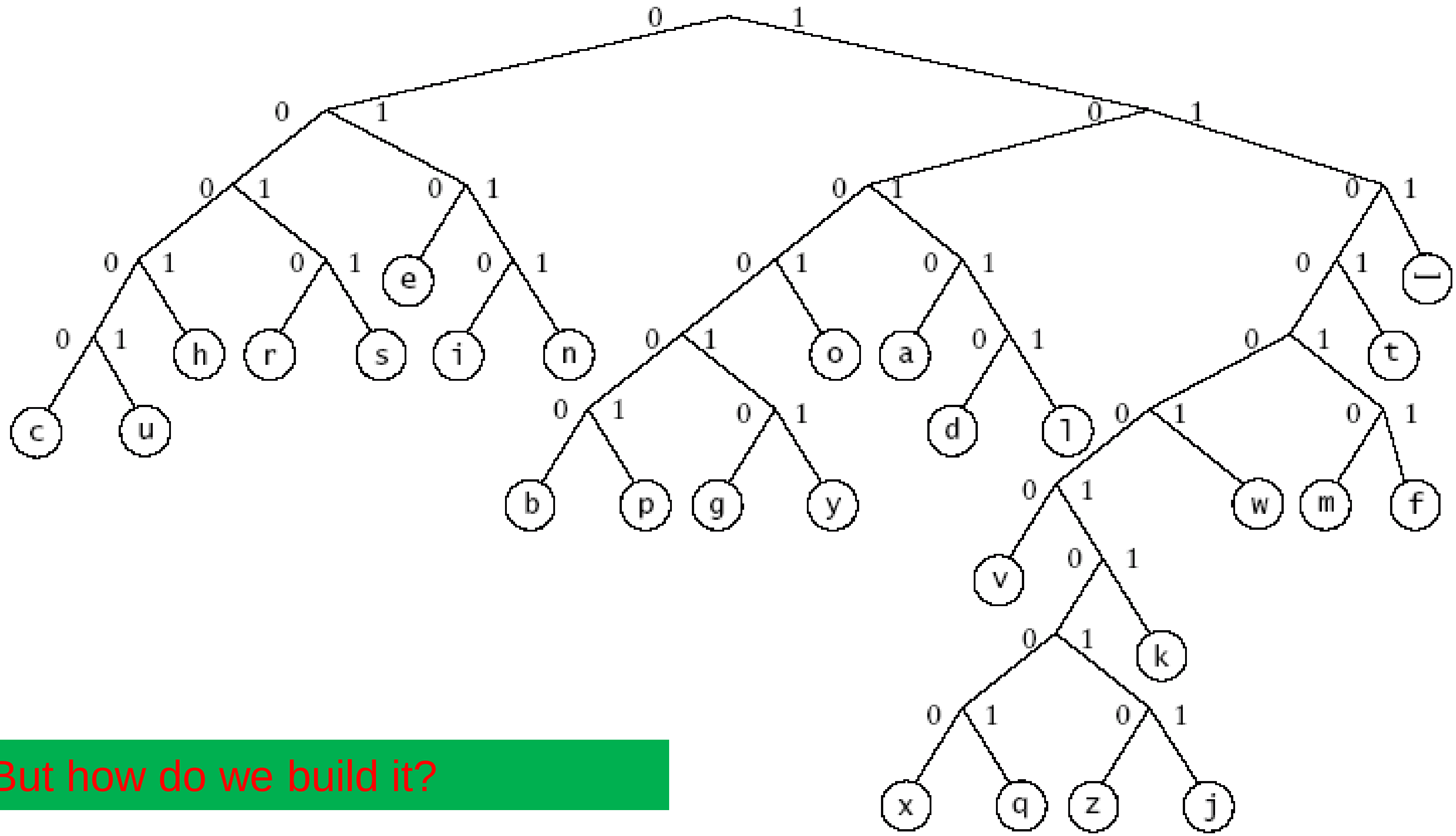
# Huffman Trees

| Symbol | Frequency | Symbol | Frequency | Symbol | Frequency |
|---|---|---|---|---|---|
| ␣ | 186 | h | 47 | g | 15 |
| e | 103 | d | 32 | p | 15 |
| t | 80 | l | 32 | b | 13 |
| a | 64 | u | 23 | v | 8 |
| o | 63 | c | 22 | k | 5 |
| i | 57 | f | 21 | j | 1 |
| n | 57 | m | 20 | q | 1 |
| s | 51 | w | 18 | x | 1 |
| r | 48 | y | 16 | z | 1 |

After D. Knuth

# Huffman Trees (cont.)

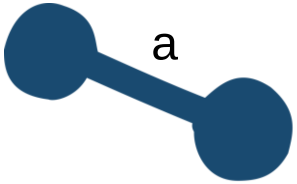
Huffman Tree Based on Frequency of Letters in English Text




But how do we build it?

# Building a Custom Huffman Tree

- Suppose we want to build a custom Huffman tree to compress a file
- **Input**: an array of objects such that each object contains a reference to a string occurring in that file and the frequency of occurrence (weight) for the string in that file


a


b

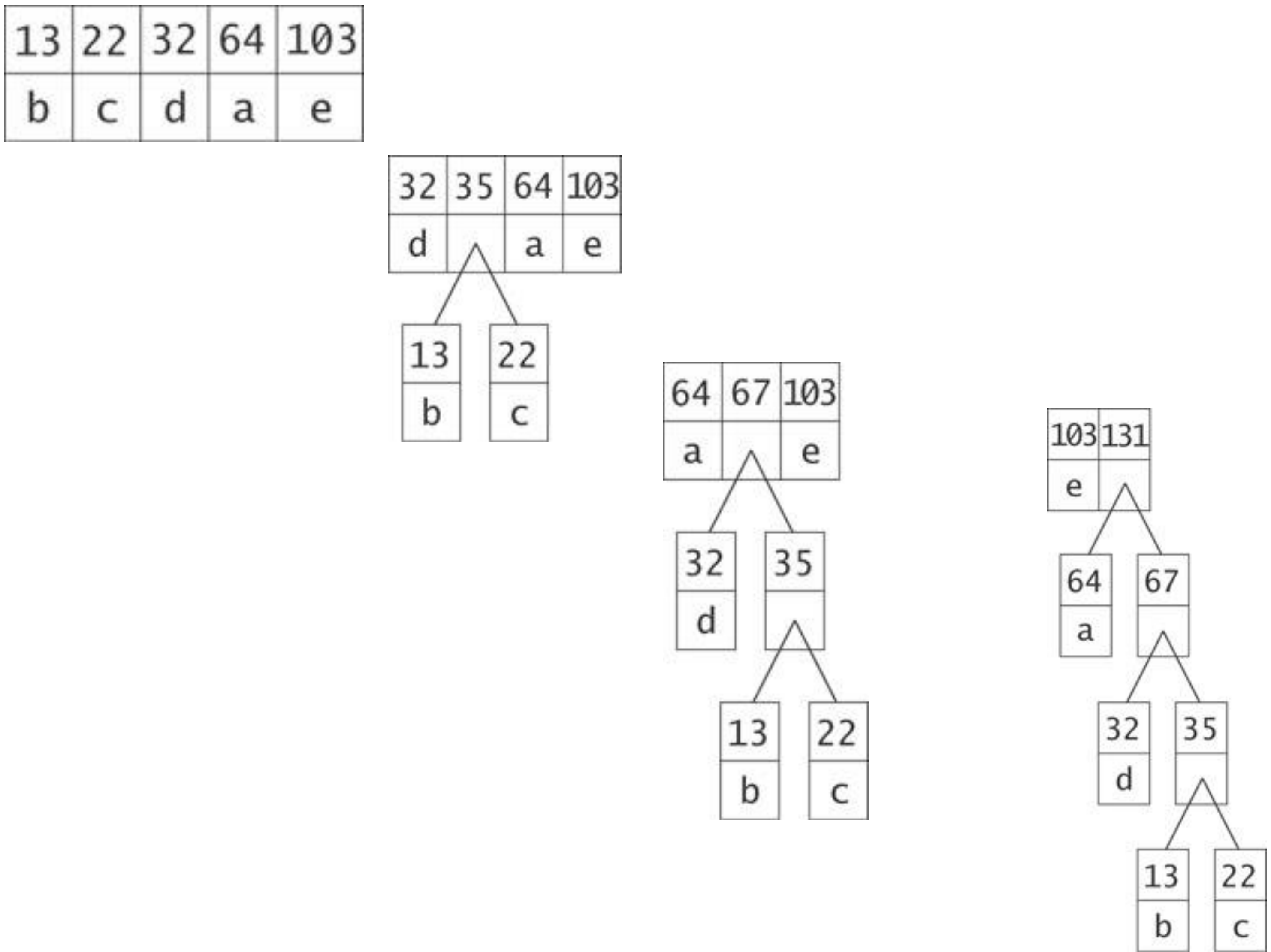
# Building a Custom Huffman Tree (cont.)

- **Analysis**:
  - Each node will have storage for two data items:
    - the weight of the node and
    - the symbol associated with the node
  - All symbols will be stored in leaf nodes
  - The weight of a leaf node will be the frequency of the symbol stored at that node
  - The weight of an interior node will be the sum of frequencies of all nodes in the subtree rooted at the interior node

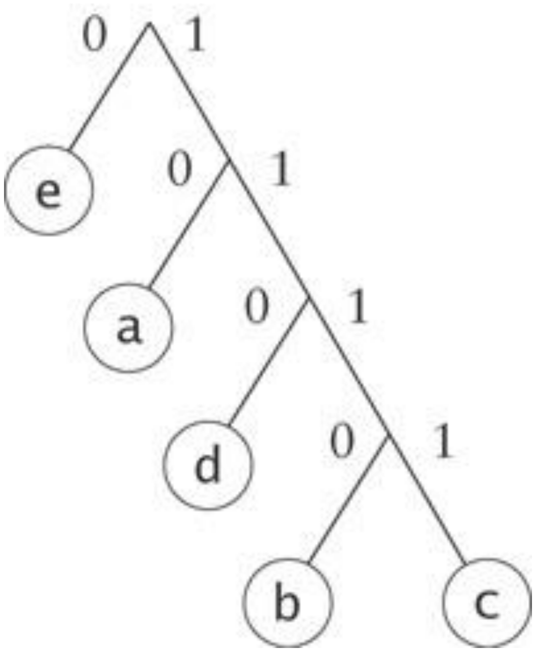
# **Building a Custom Huffman Tree** (cont.)

- **Analysis**:
  - A priority queue will be the key data structure in our Huffman tree
  - We will store individual symbols and subtrees of multiple symbols in order by their priority (frequency of occurrence)

# Building a Custom Huffman Tree (cont.)



# **Building a Custom Huffman Tree** (cont.)




| Symbol | Code |
|---|---|
| a | 10 |
| b | 1110 |
| c | 1111 |
| d | 110 |
| e | 0 |

# Design

**Algorithm for Building a Huffman Tree**

1. Construct a set of trees with root nodes that contain each of the individual symbols and their weights.
2. Place the set of trees into a priority queue.
3. `while` the priority queue has more than one item
4. Remove the two trees with the smallest weights.
5. Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.
6. Insert the newly created tree back into the priority queue.

# **Design** (cont.)

| Data Field | Attribute |
|---|---|
| `BinaryTree<HuffData> huffTree` | A reference to the Huffman tree. |
| **Method** | **Behavior** |
| `buildTree(HuffData[] input)` | Builds the Huffman tree using the given alphabet and weights. |
| `String decode(String message)` | Decodes a message using the generated Huffman tree. |
| `printCode(PrintStream out)` | Outputs the resulting code. |

# Implementation

- Listing 6.9 (Class `HuffmanTree`; page 349)
- Listing 6.10 (The `buildTree` Method (`HuffmanTree.java`); pages 350-351)
- Listing 6.11 (The `decode` Method (`HuffmanTree.java`); page 352)