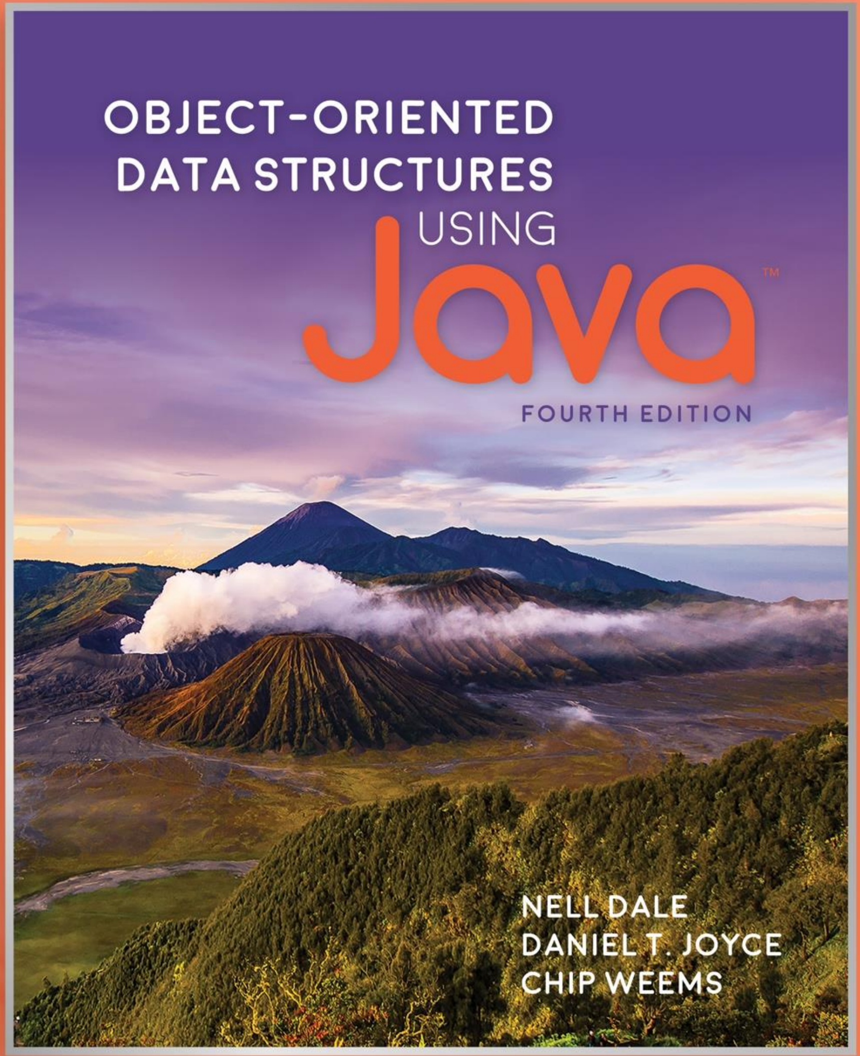
Chapter 9

The Priority Queue ADT



Chapter 9: The Priority Queue ADT

9.1 The Priority Queue Interface

- A priority queue is an abstract data type with an interesting accessing protocol - only the *highest-priority* element can be accessed
- Priority queues are useful for any application that involves processing items by priority

The Interface

```
package ch09.priorityQueues;

public interface PriQueueInterface<T>
{
  void enqueue(T element);
  // Throws PriQOverflowException if this priority queue is full;
  // otherwise, adds element to this priority queue.

  T dequeue();
  // Throws PriQUnderflowException if this priority queue is empty;
  // otherwise, removes element with highest priority from this
  // priority queue and returns it.

  boolean isEmpty();
  // Returns true if this priority queue is empty; otherwise, returns false.

  boolean isFull();
  // Returns true if this priority queue is full; otherwise, returns false.

  int size();
  // Returns the number of elements in this priority queue. }
```

9.2 Priority Queue Implementations

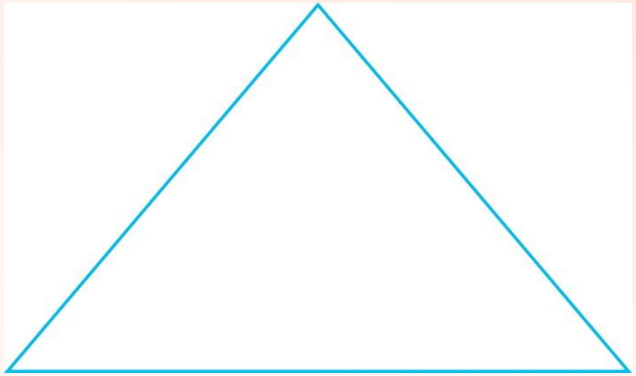
- There are many ways to implement a priority queue
  - **An Unsorted List -** dequeuing would require searching through the entire list
  - **An Array-Based Sorted List -** Enqueuing is expensive
  - **A Sorted Linked List -** Enqueuing again is 0(N)
  - **A Binary Search Tree -** On average, $0(\log_2 N)$ steps for both enqueue and dequeue
  - **A Heap -** (next section) guarantees $0(\log_2 N)$ steps, even in the worst case

9.3 The Heap

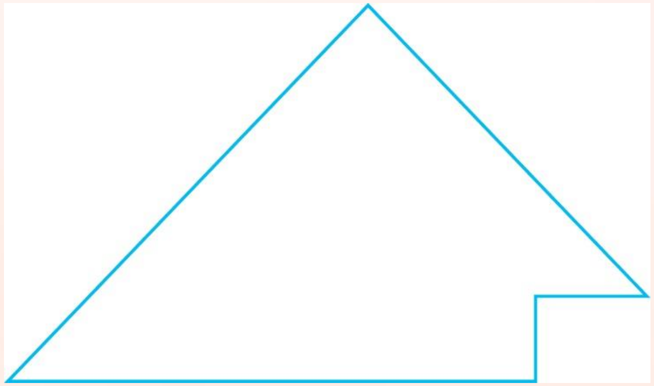
- **Heap** An implementation of a Priority Queue based on a complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children
- In other words, a heap is an implementation of a Priority Queue that uses a binary tree that satisfies two properties
  - the *shape property*: the tree must be a complete binary tree
  - the *order property:* for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children.

Tree Terminology

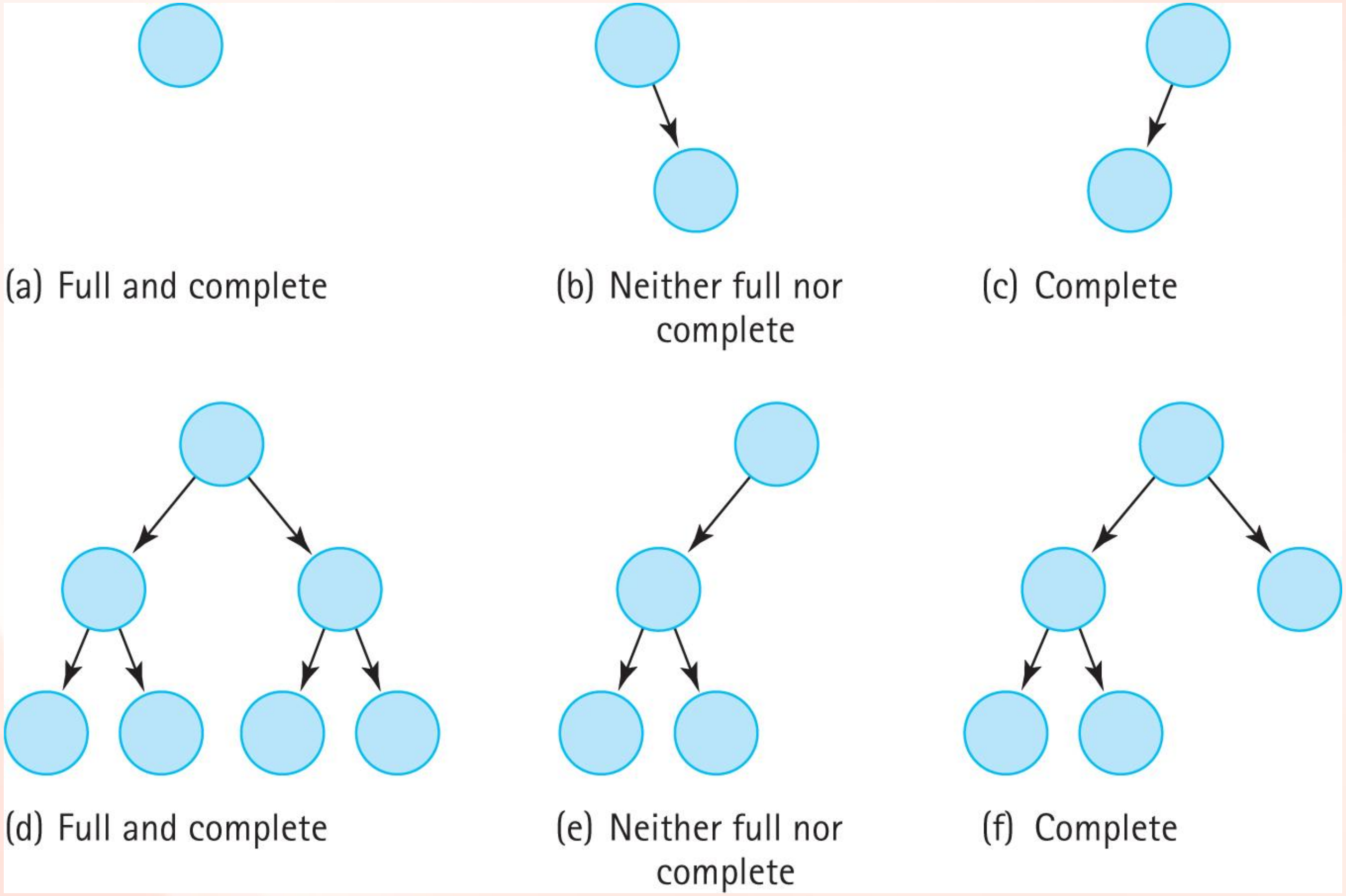
- A full binary tree



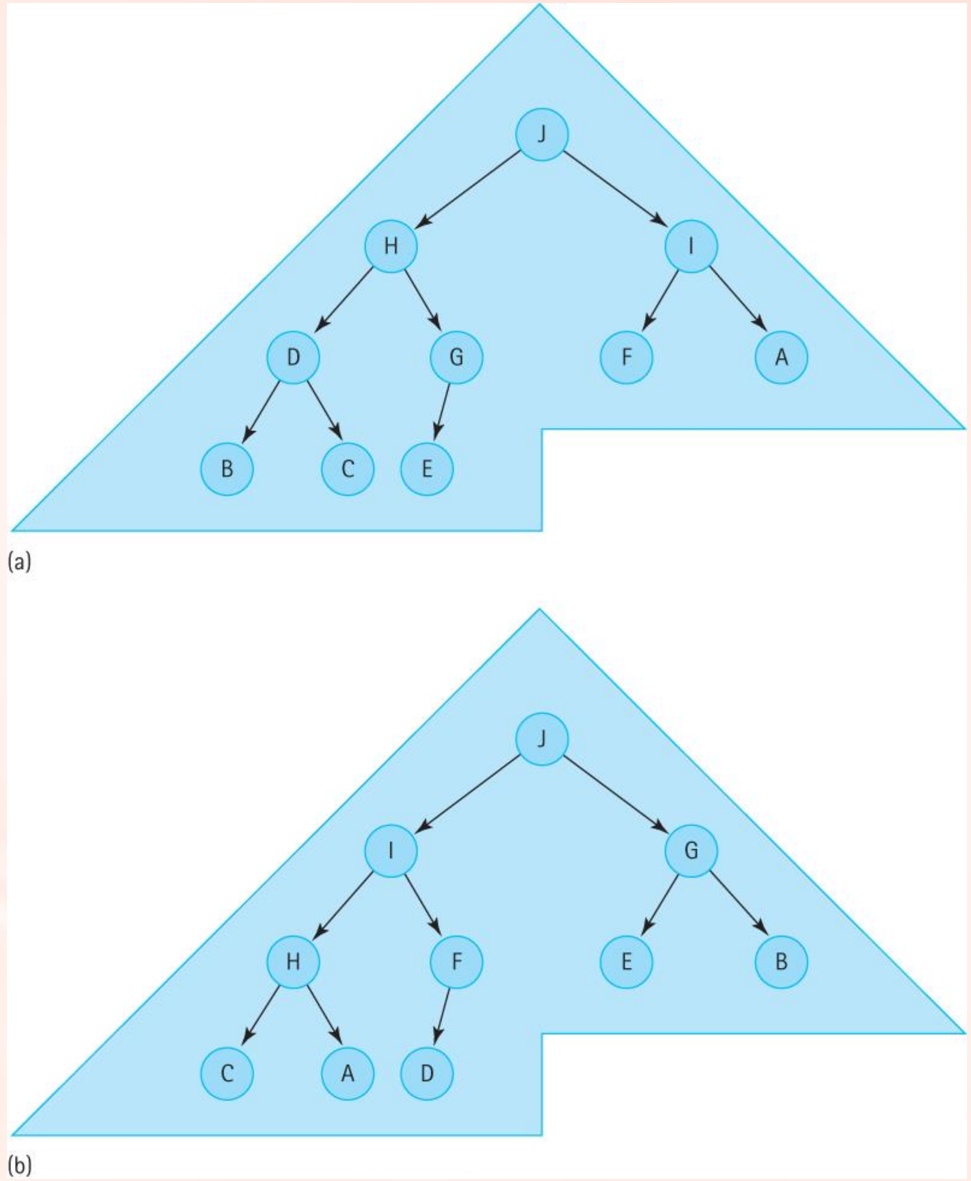
- A complete binary tree



Examples

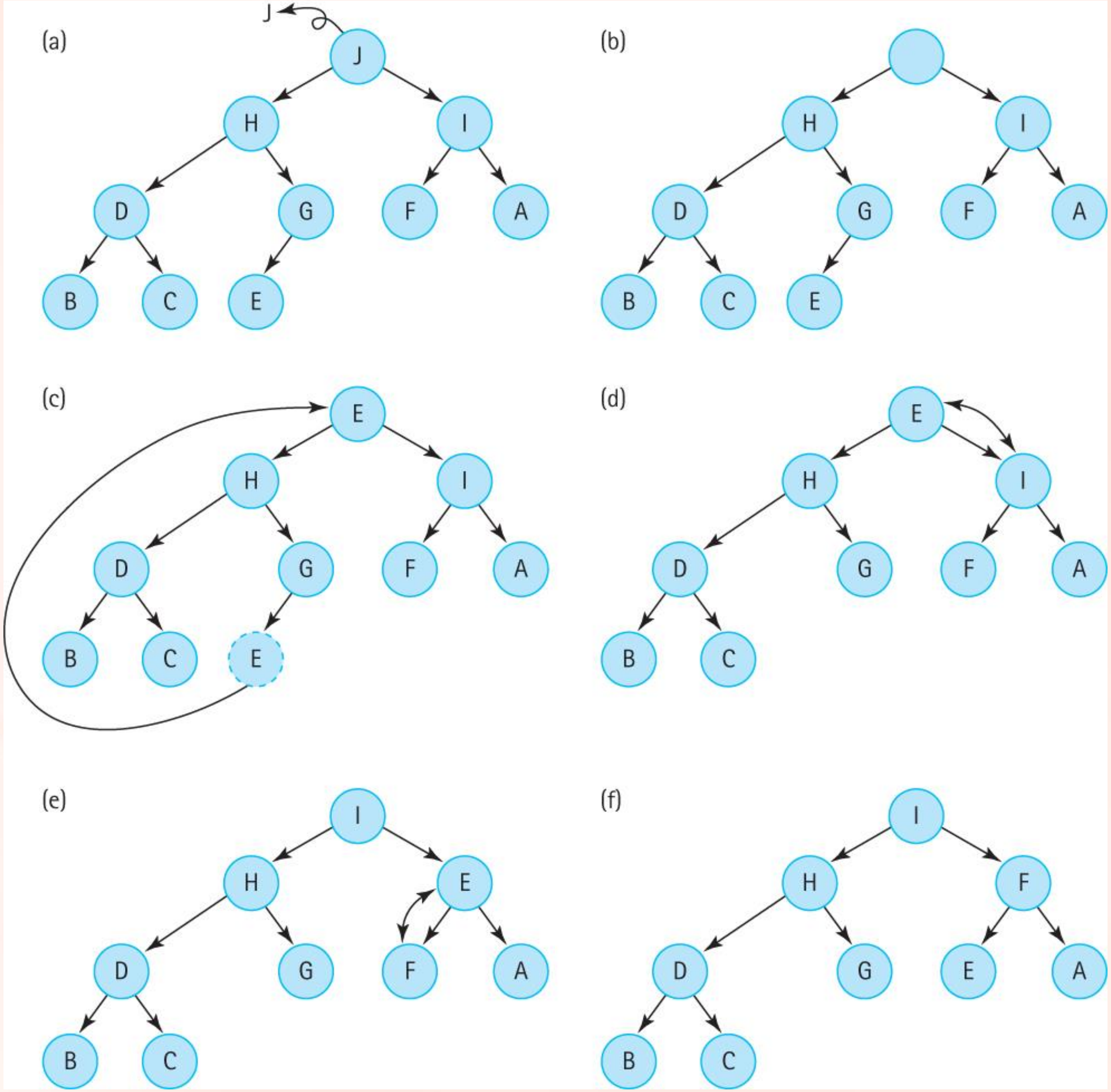

(a) Full and complete
(b) Neither full nor complete
(c) Complete
(d) Full and complete
(e) Neither full nor complete
(f) Complete

Two Heaps Containing the Letters 'A' through 'J'



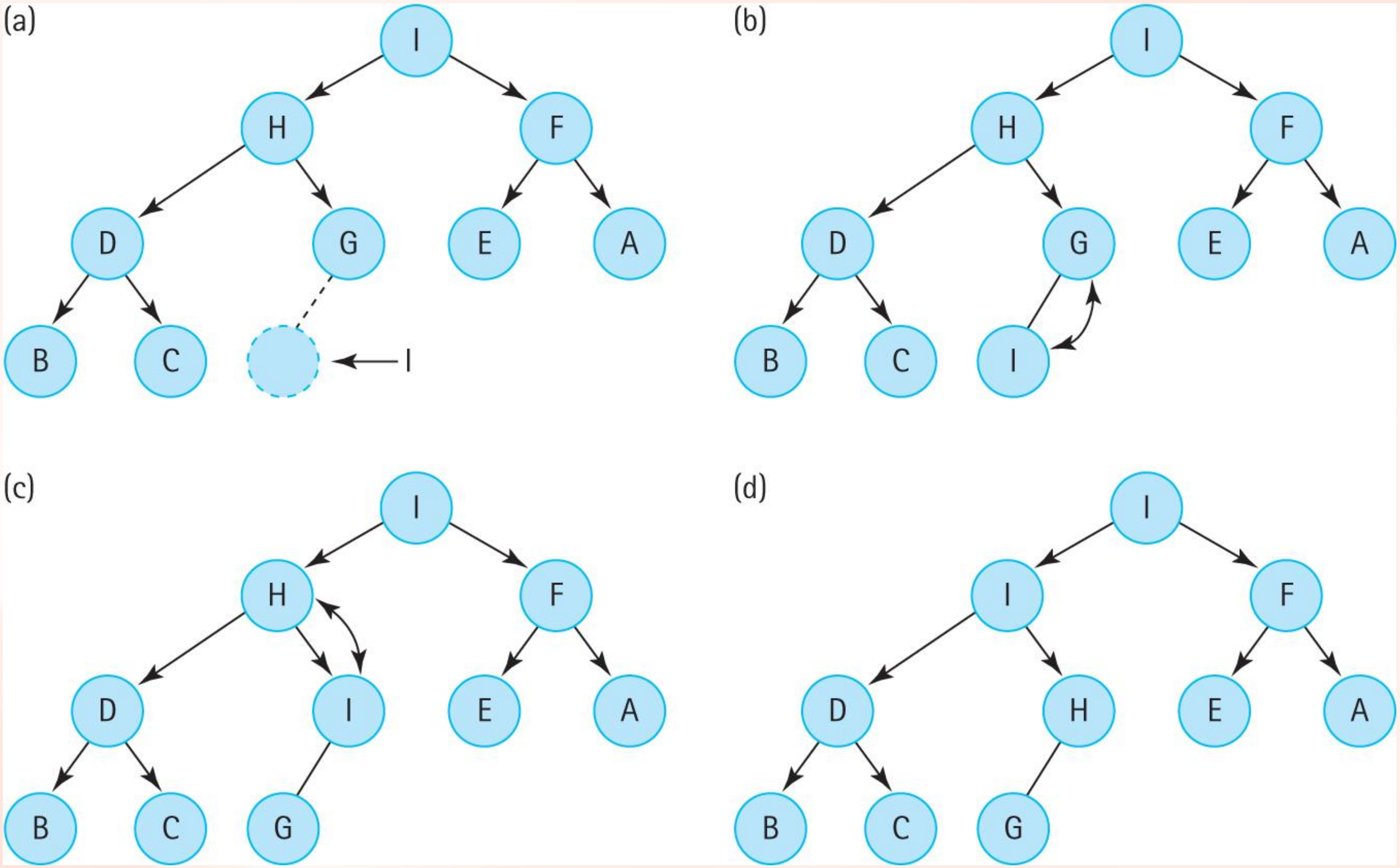
The dequeue operation

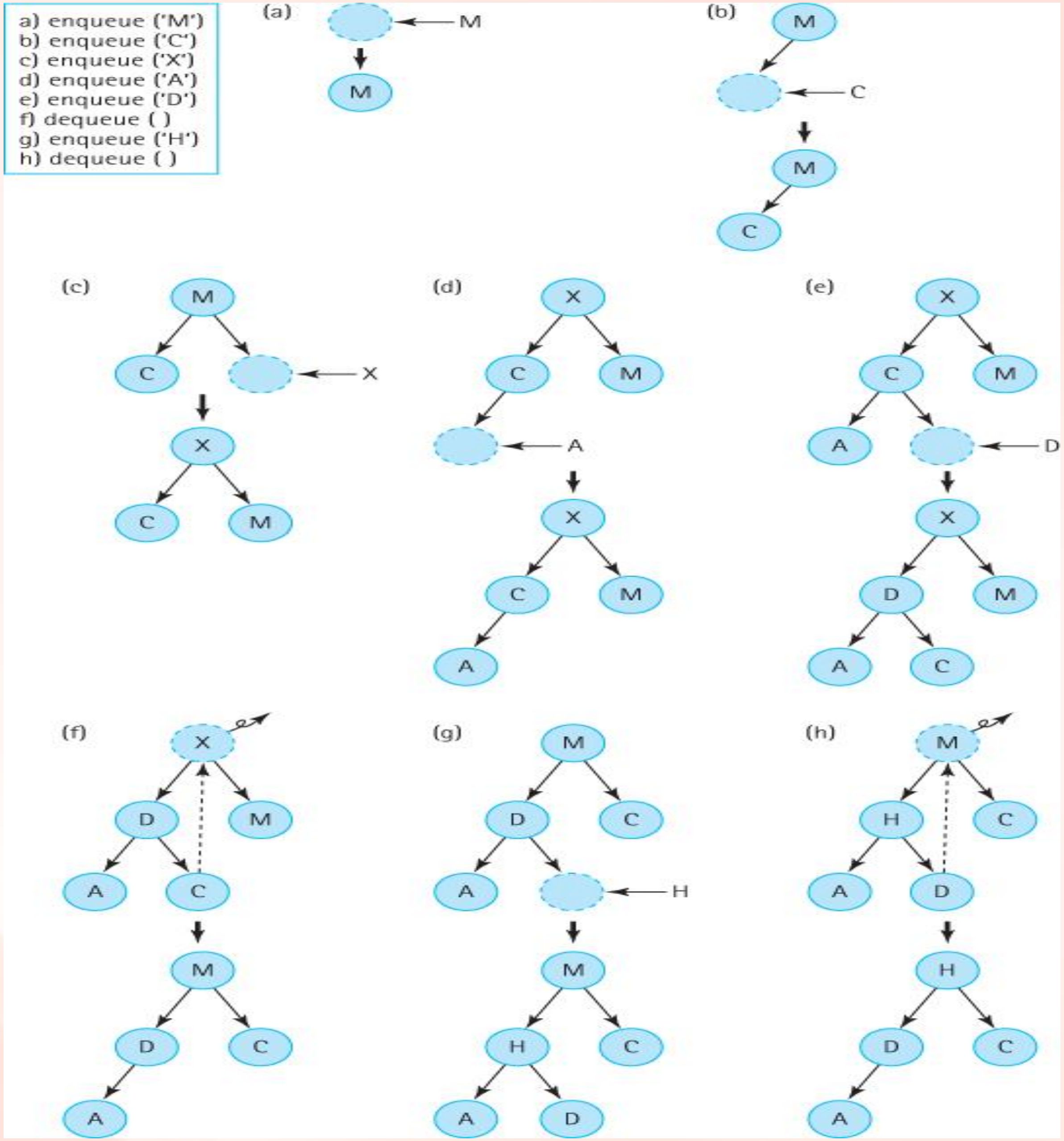
Steps d,e,f represent the "reheap down" operation



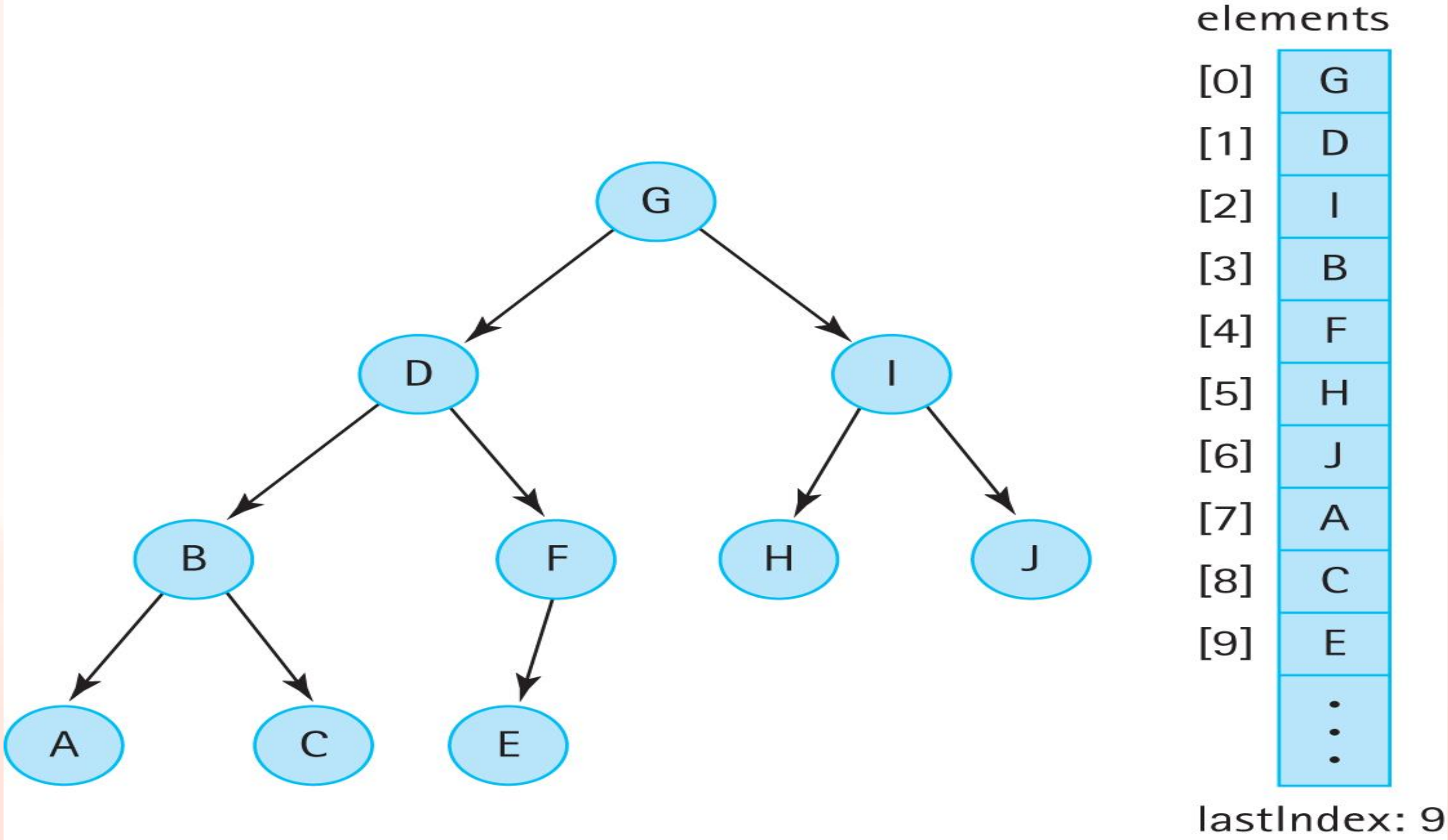
The enqueue operation

steps b, c represent the “reheap up” operation




a) enqueue ('M')
b) enqueue ('C')
c) enqueue ('X')
d) enqueue ('A')
e) enqueue ('D')
f) dequeue ( )
g) enqueue ('H')
h) dequeue ( )
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)

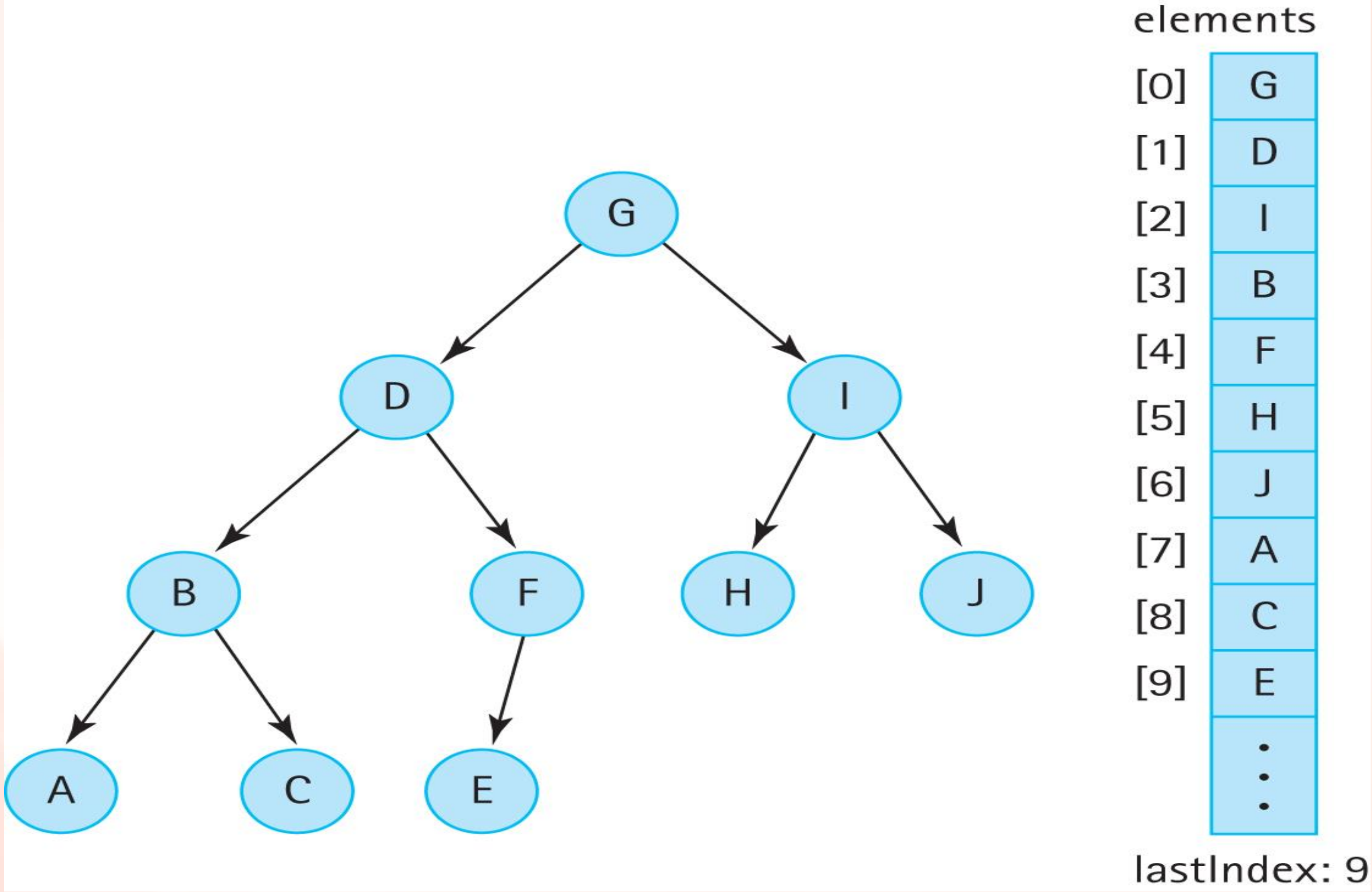
9.4 The Heap Implementation



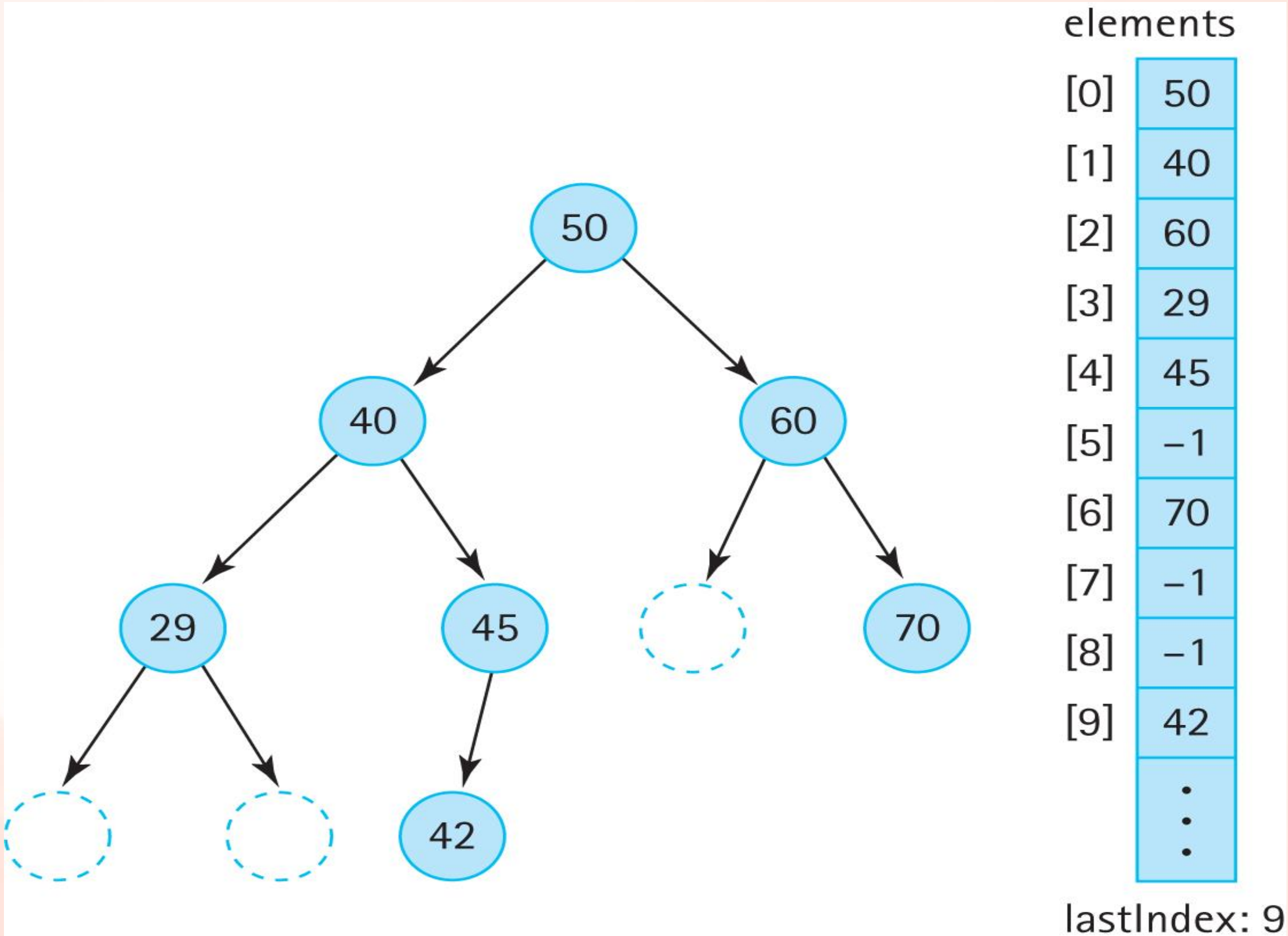
A Non-linked Representation of Binary Trees

- A binary tree can be stored in an array in such a way that the relationships in the tree are not physically represented by link members, but are *implicit* in the algorithms that manipulate the tree stored in the array.
- We store the tree elements in the array, level by level, left-to-right. We call the array `elements` and store the index of the last tree element in a variable `lastIndex`.
- The tree elements are stored with the root in `elements[0]` and the last node in `elements[lastIndex]`.

A Binary Tree and Its Array Representation



A Binary Search Tree Stored in an Array with Dummy Values



Array Representation continued

- To implement the algorithms that manipulate the tree, we must be able to find the left and right child of a node in the tree:
  - elements[index] left child is in elements[index*2 + 1]
  - elements[index] right child is in elements[index*2 + 2]
- We can also can determine the location of its parent node:
  - elements[index]'s parent is in elements[(index – 1)/2].
- This representation works best, space wise, if the tree is complete (which it is for a heap)

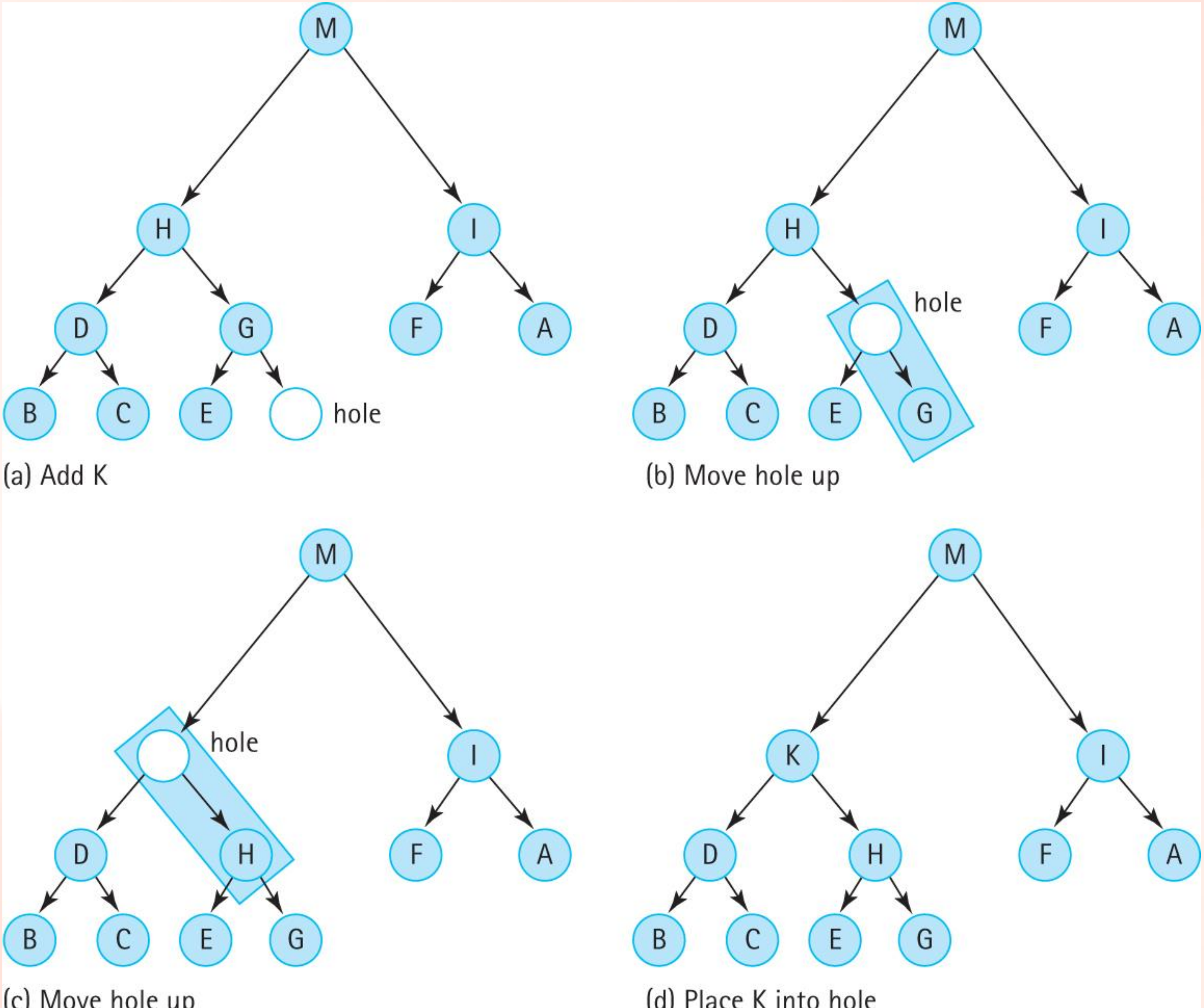
Beginning of `HeapPriQ.java`

```
//----------------------------------------------------------------------------
// HeapPriQ.java              by Dale/Joyce/Weems                   Chapter
9 // Priority Queue using Heap (implemented with an ArrayList)
//
// Two constructors are provided: one that use the natural order of the
// elements as defined by their compareTo method and one that uses an
// ordering based on a comparator argument.
//----------------------------------------------------------------------------
package ch09.priorityQueues;
import java.util.*;  // ArrayList, Comparator

public class HeapPriQ<T> implements PriQueueInterface<T>
{
  protected ArrayList<T> elements; // priority queue elements
  protected int lastIndex;         // index of last element in priority queue
  protected int maxIndex;          // index of last position in ArrayList

  protected Comparator<T> comp;

. . .
```

The enqueue method

```
public void enqueue(T element) throws PriQOverflowException
// Throws PriQOverflowException if this priority queue is full;
// otherwise, adds element to this priority queue.
{
  if (lastIndex == maxIndex)
    throw new PriQOverflowException("Priority queue is full");
  else
  {
    lastIndex++;
    elements.add(lastIndex, element);
    reheapUp(element);
  }
}
```

The `reheapUp` algorithm is pictured on the next slide


M
H
I
D
G
F
A
B
C
E
hole
(a) Add K
M
H
I
hole
D
F
A
B
C
E
G
(b) Move hole up
M
hole
I
D
H
F
A
B
C
E
G
(c) Move hole up
M
K
I
D
H
F
A
B
C
E
G
(d) Place K into hole

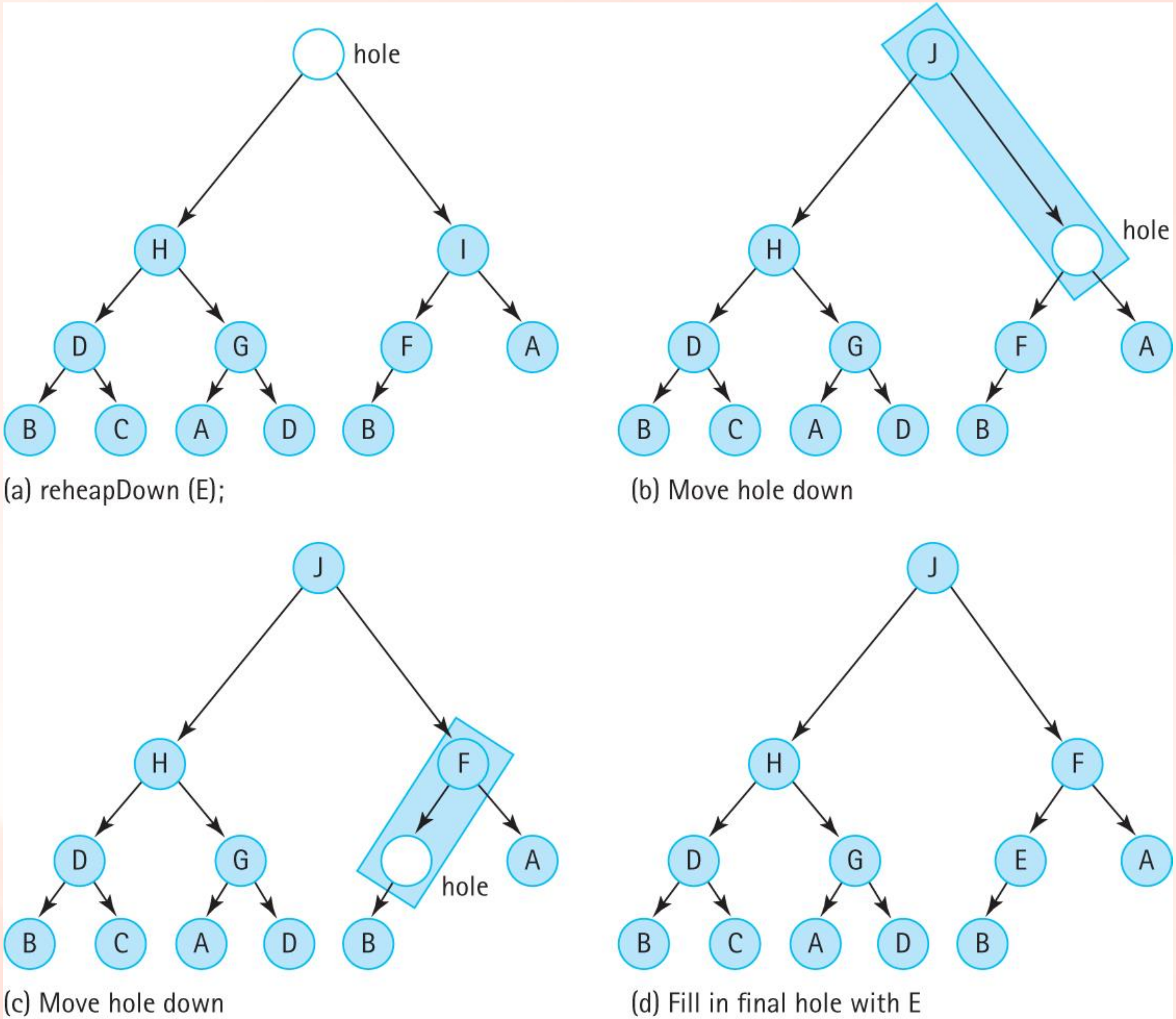
reheapUp operation

```
private void reheapUp(T element)
// Current lastIndex position is empty.
// Inserts element into the tree and ensures shape and order properties.
{
  int hole = lastIndex;
  while ((hole > 0)   // hole is not root and element > hole's parent
         &&
       (comp.compare(element, elements.get((hole - 1) / 2)) > 0))
  {
    elements.set(hole,elements.get((hole - 1) / 2));   // move hole's parent down
    hole = (hole - 1) / 2;                             // move hole up
  }
  elements.set(hole, element);                    // place element into final hole
}
```

The `dequeue` method

```
public T dequeue() throws PriQUnderflowException
// Throws PriQUnderflowException if this priority queue is empty;
// otherwise, removes element with highest priority from this
// priority queue and returns it.
{
  T hold;      // element to be dequeued and returned
  T toMove;    // element to move down heap

  if (lastIndex == -1)
    throw new PriQUnderflowException("Priority queue is empty");
  else
  {
    hold = elements.get(0);                 // remember element to be returned
    toMove = elements.remove(lastIndex); // element to reheap down
    lastIndex--;                            // decrease priority queue size
    if (lastIndex != -1)
       reheapDown(toMove);                  // restore heap properties
    return hold;                            // return largest element
  }
}
```

The `reheapDown` algorithm is pictured on the next slide



(a) reheapDown (E);

(b) Move hole down

(c) Move hole down

(d) Fill in final hole with E

reheapDown operation

```
private void reheapDown(T element)
// Current root position is "empty";
// Inserts element into the tree and ensures shape and order properties.
{
  int hole = 0;       // current index of hole
  int next;           // next index where hole should move to

  next = newHole(hole, element);   // find next hole
  while (next != hole)
  {
    elements.set(hole,elements.get(next));  // move element up
    hole = next;                             // move hole down
    next = newHole(hole, element);           // find next hole
  }
  elements.set(hole, element);               // fill in the final hole
}
```

The newHole method

```
private int newHole(int hole, T element)
// If either child of hole is larger than element return the index
// of the larger child; otherwise return the index of hole.
{
  int left = (hole * 2) + 1;
  int right = (hole * 2) + 2;
  if (left > lastIndex)
    // hole has no children
    return hole;
  else
  if (left == lastIndex)
    // hole has left child only
    if (comp.compare(element, elements.get(left)) < 0)
      // element < left child
      return left;
    else
      // element >= left child
      return hole;
  else
  // hole has two children
  if (comp.compare(elements.get(left), elements.get(right)) < 0)
    // left child < right child
    if (comp.compare(elements.get(right), element) <= 0)
      // right child <= element
      return hole;
    else
      // element < right child
      return right;
  else
  // left child >= right child
  if (comp.compare(elements.get(left), element) <= 0)
    // left child <= element
    return hole;
  else
    // element < left child
    return left;
}
```

Heaps Versus Other Representations of Priority Queues

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$