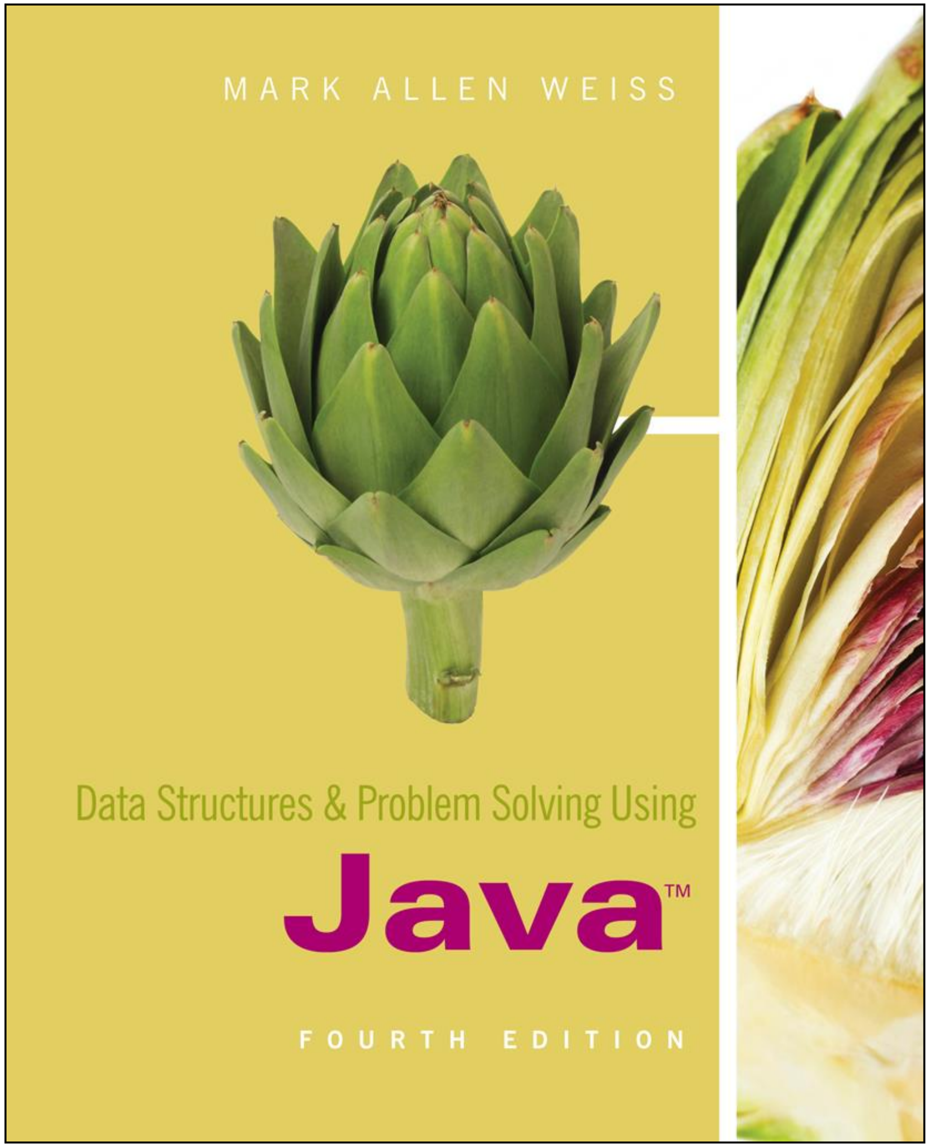
Data Structures
Lesson 9

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

Chapter 21

A Priority Queue:
The Binary Heap



Addison-Wesley
is an imprint of

PEARSON

Priority Queue

- The priority queue is a fundamental data structure that allows access only to the minimum (or maximum) item.
- We could implement it by using a simple linked list, performing insertions at the front in constant time, but then finding and/or deleting the minimum would require a linear scan of the list.
- Alternatively, we could insist that the list always be kept sorted.
  - This condition makes the access and deletion of the minimum cheap, but then insertions would be linear.

Implementation choices

- Another way of implementing priority queues is to use a binary search tree, which gives an O(log N) average running time for both operations.
- However, a binary search tree is a poor choice because the input is typically not sufficiently random.
- We could use a balanced search tree, but these structures are cumbersome to implement and lead to slow performance in practice.

Implementation choices

- On the one hand, because the priority queue supports only some of the search tree operations, it should not be more expensive to implement than a search tree.
- On the other hand, the priority queue is more powerful than a simple queue because we can use a priority queue to implement a queue as follows.
  - First, we insert each item with an indication of its insertion time.
  - Then, a deleteMin on the basis of minimum insertion time implements a dequeue.

Binary Heap

- We can expect to obtain an implementation with properties that are a compromise between a queue and a search tree. This compromise is realized by the binary heap, which:
  - Can be implemented by using a simple array (like the queue)
  - Supports insert and deleteMin in O(log N) worst-case time (a compromise between the binary search tree and the queue)
  - Supports insert in constant average time and findMin in constant worst-case time (like the queue)

Binary Heap

- The binary heap is the classic method used to implement priority queues and — like the balanced search tree structures — has two properties:
  - a structure property and an ordering property.
- As with balanced search trees, an operation on a binary heap can destroy one of the properties, so a binary heap operation must not terminate until both properties are in order.
- This outcome is simple to achieve.

Complete binary trees

- The only structure that gives dynamic logarithmic time bounds is the tree, so it seems natural to organize the heap's data as a tree.

- Because we want the logarithmic bound to be a worst-case guarantee, the tree should be balanced.

- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right and has no missing nodes.

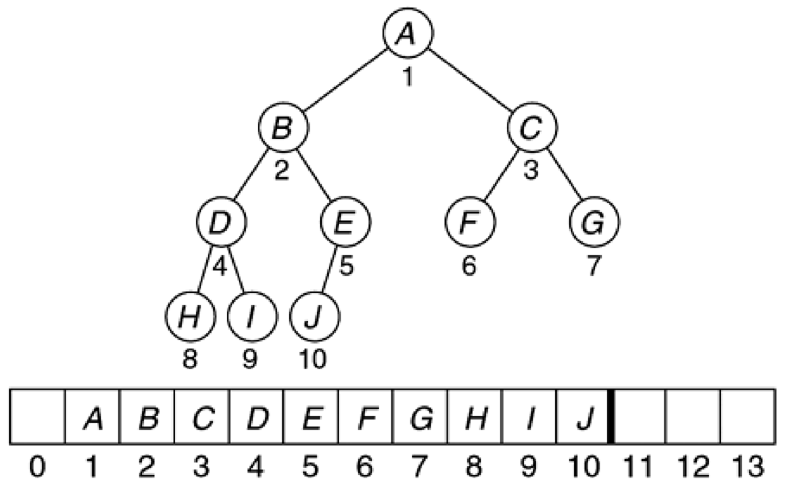


figure 21.1

A complete binary tree and its array representation

Had the node J been a right child of E, the tree would not be complete because a node would be missing.

Properties

- The complete tree has a number of useful properties.
- First, the height (longest path length) of a complete binary tree of N nodes is at most [log N].
- The reason is that a complete tree of height H has between 2^H and $2^{H+1} - 1$ nodes.
- This characteristic implies that we can expect logarithmic worst-case behavior if we restrict changes in the structure to one path from the root to a leaf.

Properties

- Second and equally important, in a complete binary tree, left and right links are not needed.

- As shown in Figure 21.1, we can represent a complete binary tree by storing its level-order traversal in an array.

- We place the root in position 1 (position 0 is often left vacant, for a reason discussed shortly).

- We also need to maintain an integer that tells us the number of nodes currently in the tree.

Properties

- Then for any element in array position i, its left child can be found in position $2i$.
- If this position extends past the number of nodes in the tree, we know that the left child does not exist.
- Similarly, the right child is located immediately after the left child; thus it resides in position $2i + 1$.
- We again test against the actual tree size to be sure that the child exists. Finally, the parent is in position [i/2].

Position 0

- Note that every node except the root has a parent.
- If the root were to have a parent, the calculation would place it in position 0.
- Thus we reserve position 0 for a dummy item that can serve as the root's parent.
- Doing so can simplify one of the operations (add).
  - If instead we choose to place the root in position 0, the locations of the children and parent of the node in position i change slightly

Implicit Representation

- Using an array to store a tree is called implicit representation.
- As a result of this representation, not only are child links not required, but also the operations required to traverse the tree are extremely simple and likely to be very fast on most computers.
- The heap entity consists of an array of objects and an integer representing the current heap size.

The heap-order property

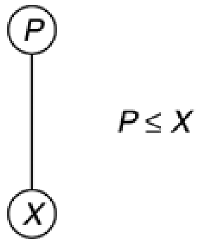
- We want to be able to find the minimum quickly, so it makes sense that the smallest element should be at the root.
- If we consider that any subtree should also (recursively) be a heap, any node should be smaller than all of its descendants. Applying this logic, we arrive at the heap-order property.

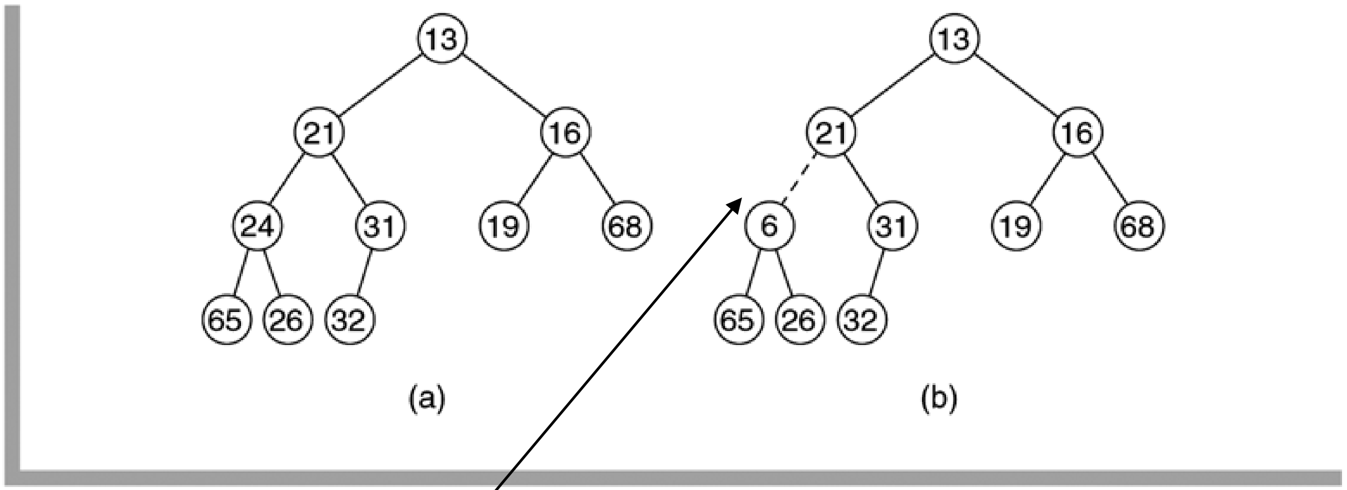
- Heap-order property

  In a heap, for every node X with parent P the key in P is smaller than or equal to the key in X.

figure 21.2

Heap-order property





The dashed line shows the violation of the heap order.

Min (or Max) heap

- Note that the root does not have a parent.
- In the implicit representation, we could place the value $-\infty$ in position 0 to remove this special case when we implement the heap.
- By the heap-order property, we see that the minimum element can always be found at the root.
- Thus findMin is a constant time operation.
- A max heap supports access of the maximum instead of the minimum. Minor changes can be used to implement max heaps.

```java
package weiss.util;

/**
 * PriorityQueue class implemented via the binary heap.
 */
public class PriorityQueue<AnyType> extends AbstractCollection<AnyType>
                                    implements Queue<AnyType>
{
    public PriorityQueue( )
      { /* Figure 21.5 */ }
    public PriorityQueue( Comparator<? super AnyType> c )
      { /* Figure 21.5 */ }
    public PriorityQueue( Collection<? extends AnyType> coll )
      { /* Figure 21.5 */ }

    public int size( )
      { /* return currentSize; */ }
    public void clear( )
      { /* currentSize = 0; */ }
    public Iterator<AnyType> iterator( )
      { /* See online code */ }

    public AnyType element( )
      { /* Figure 21.6 */ }
    public boolean add( AnyType x )
      { /* Figure 21.9 */ }
    public AnyType remove( )
      { /* Figure 21.13 */ }

    private void percolateDown( int hole )
      { /* Figure 21.14 */ }
    private void buildHeap( )
      { /* Figure 21.16 */ }

    private int currentSize;   // Number of elements in heap
    private AnyType [ ] array; // The heap array
    private Comparator<? super AnyType> cmp;

    private void doubleArray( )
      { /* See online code */ }
    private int compare( AnyType lhs, AnyType rhs )
      { /* Same code as in TreeSet; see Figure 19.70 */ }
}
```

Constructor 3 takes in input a collection!

Why? Next slide =>

We need a Comparator to compare the elements

figure 21.4

The PriorityQueue class skeleton

Constructor from collection

- In numerous applications we can add many items before the next deleteMin occurs.
- In those cases, we do not need to have heap order in effect until the deleteMin occurs.
- The buildHeap operation, declared at line 32, reinstates the heap order — no matter how messed up the heap is — and we will see that it works in linear time.

- *Thus, if we need to place N items in the heap before the first deleteMin, placing them in the array with no order and then doing one buildHeap is more efficient than doing N insertions.*

```
private static final int DEFAULT_CAPACITY = 100;

/**
 * Construct an empty PriorityQueue.
 */
public PriorityQueue( )
{
    currentSize = 0;
    cmp = null;
    array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
}

/**
 * Construct an empty PriorityQueue with a specified comparator.
 */
public PriorityQueue( Comparator<? super AnyType> c )
{
    currentSize = 0;
    cmp = c;
    array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
}


/**
 * Construct a PriorityQueue from another Collection.
 */
public PriorityQueue( Collection<? extends AnyType> coll )
{
    cmp = null;
    currentSize = coll.size( );
    array = (AnyType[]) new Object[ ( currentSize + 2 ) * 11 / 10 ];

    int i = 1;
    for( AnyType item : coll )
        array[ i++ ] = item;
    buildHeap( );
}
```

Better than N insertions

figure 21.5

Constructors for the `PriorityQueue` class

```java
/**
 * Returns the smallest item in the priority queue.
 * @return the smallest item.
 * @throws NoSuchElementException if empty.
 */
public AnyType element( )
{
    if( isEmpty( ) )
        throw new NoSuchElementException( );
    return array[ 1 ];
}
```

figure 21.6

The `element` routine

Insertion

- To insert an element X in the heap, we must first add a node to the tree.
- The only option is to create a hole in the next available location; otherwise, the tree is not complete and we would violate the structure property.
  - If X can be placed in the hole without violating heap order, we do so and are done.
  - Otherwise, we slide the element that is in the hole's parent node into the node, bubbling the hole up toward the root.
- We continue this process until X can be placed in the hole.

figure 21.7

Attempt to insert 14, creating the hole and bubbling the hole up

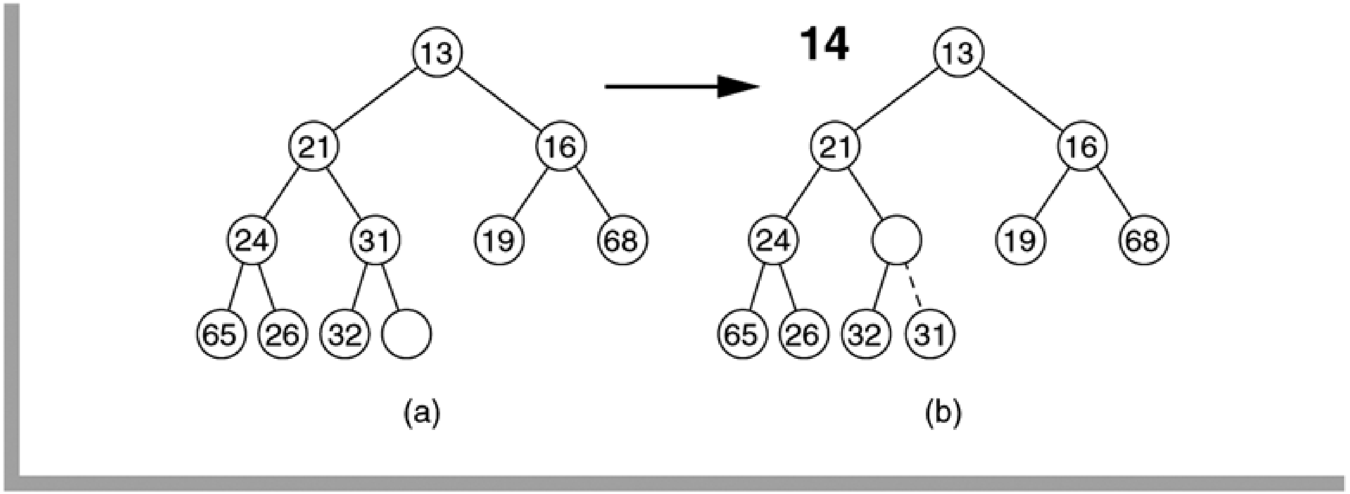


Figure 21.7 shows that to insert 14, we create a hole in the next available heap location. Inserting 14 into the hole would violate the heap-order property, so 31 is slid down into the hole.

This strategy is continued in Figure 21.8 until the correct location for 14 is found. => next slide

Percolate Up

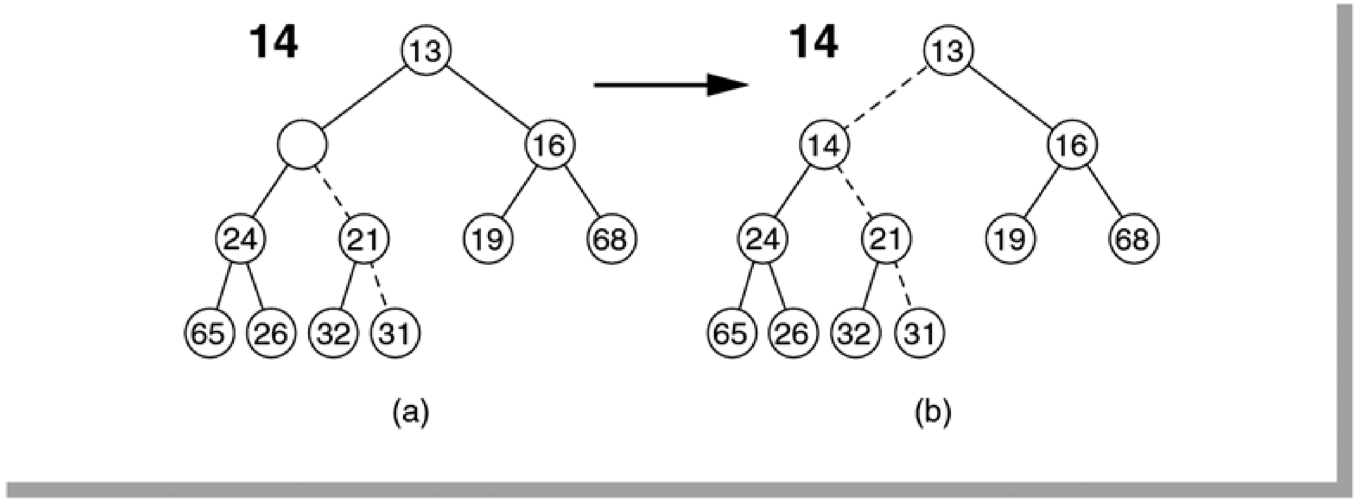


figure 21.8

The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

This general strategy is called percolate up, in which insertion is implemented by creating a hole at the next available location and bubbling it up the heap until the correct location is found.

```
 1    /**
 2     * Adds an item to this PriorityQueue.
 3     * @param x any object.
 4     * @return true.
 5     */
 6    public boolean add( AnyType x )
 7    {
 8        if( currentSize + 1 == array.length )
 9            doubleArray( );
10
11            // Percolate up
12        int hole = ++currentSize;
13        array[ 0 ] = x;
14
15        for( ; compare( x, array[ hole / 2 ] ) < 0; hole /= 2 )
16            array[ hole ] = array[ hole / 2 ];
17        array[ hole ] = x;
18
19        return true;
20    }
```

figure 21.9

The add method

Increments the current size and sets the hole to the newly added node.

Iterate the loop at line 15 as long as the item in the parent node is larger than x.

Line 16 moves the item in the parent down into the hole,

Then the third expression in the for loop moves the hole up to the parent.

When the loop terminates, line 17 places x in the hole.

Insertion: Running time

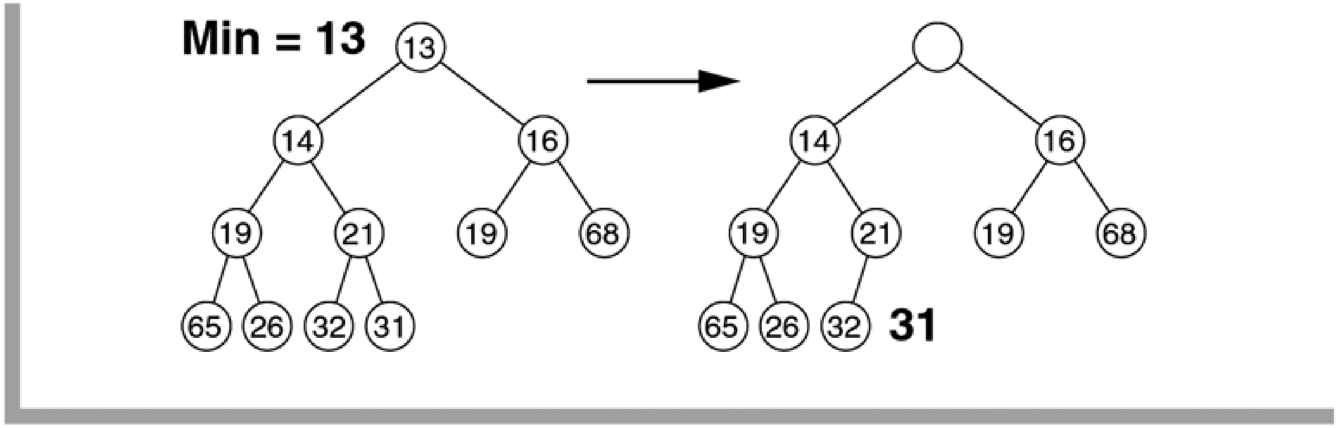
- The time required to do the insertion could be as much as O(log N) if the element to be inserted is the new minimum.
- The reason is that it will be percolated up all the way to the root.
- On average the percolation terminates early.

The deleteMin operation

- The deleteMin operation is handled in a similar manner to the insertion operation.
- As shown already, finding the minimum is easy; the hard part is removing it.
- When the minimum is removed, a hole is created at the root.
- The heap now becomes one size smaller, and the structure property tells us that the last node must be eliminated.

figure 21.10

Creation of the hole at the root



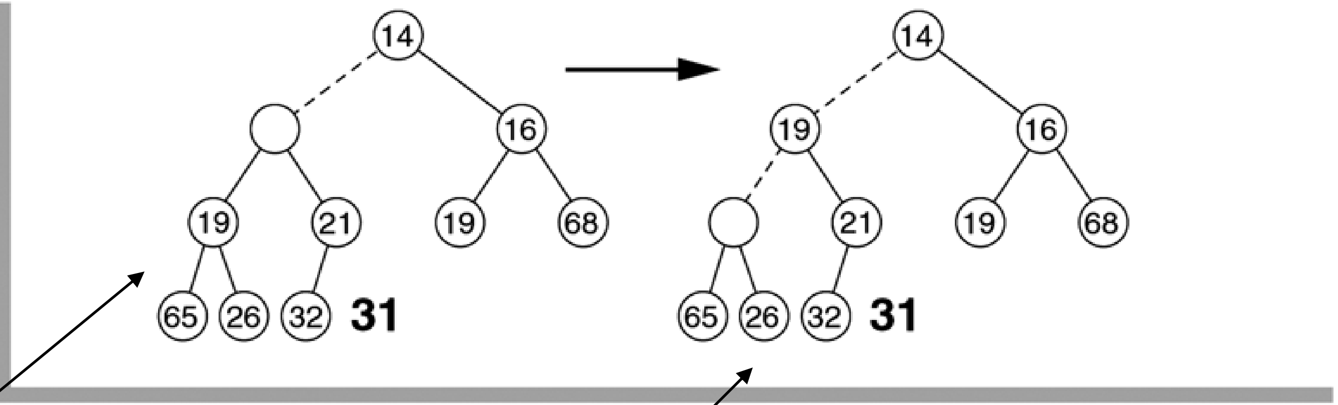
The minimum item is 13, the root has a hole, and the former last item needs to be placed in the heap somewhere.

Percolate down

- We must play the same game as for insertion: We put some item in the hole and then move the hole.
- The only difference is that for the deleteMin we move down the tree.
- To do so, we find the smaller child of the hole, and if that child is smaller than the item that we are trying to place, we move the child into the hole, pushing the hole down one level and repeating these actions until the item can be correctly placed — a process called percolate down.

figure 21.11

The next two steps in the `deleteMin` operation



We place the smaller child (14) in the hole, sliding the hole down one level.

We repeat this action, placing 19 in the hole and creating a new hole one level deeper.

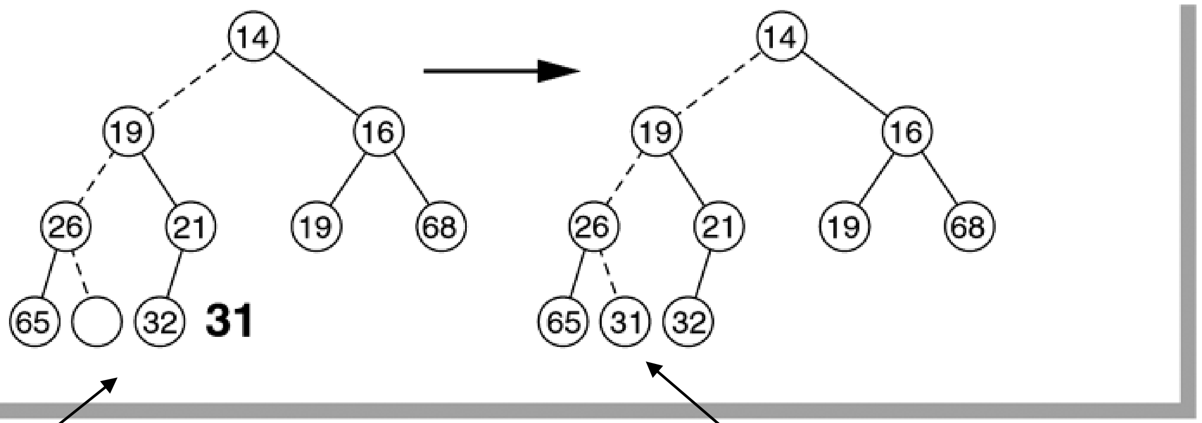


figure 21.12

The last two steps in the `deleteMin` operation

We then place 26 in the hole and create a new hole on the bottom level.

Finally, we are able to place 31 in the hole.

Running time of deleteMin

- Because the tree has logarithmic depth, deleteMin is a logarithmic operation in the worst case.
- Not surprisingly, percolation rarely terminates more than one or two levels early, so deleteMin is logarithmic on average, too.

Remove

```
/**
 * Removes the smallest item in the priority queue.
 * @return the smallest item.
 * @throws NoSuchElementException if empty.
 */
public AnyType remove( )
{
    AnyType minItem = element( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}
```

figure 21.13

The remove method

Put the bubble at the root

```
 1     /**
 2      * Internal method to percolate down in the heap.
 3      * @param hole the index at which the percolate begins.
 4      */
 5     private void percolateDown( int hole )
 6     {
 7         int child;
 8         AnyType tmp = array[ hole ];
 9
10         for( ; hole * 2 <= currentSize; hole = child )
11         {
12             child = hole * 2;
13             if( child != currentSize &&
14                     compare( array[ child + 1 ], array[ child ] ) < 0 )
15                 child++;
16             if( compare( array[ child ], tmp ) < 0 )
17                 array[ hole ] = array[ child ];
18             else
19                 break;
20         }
21         array[ hole ] = tmp;
22     }
```

figure 21.14

The percolateDown method used for remove and buildHeap

The percolateDown method takes a single parameter that indicates where the hole is placed. The item in the hole is then moved out, and the percolation begins.

The for loop at line 10 terminates when there is no left child.
The third expression moves the hole to the child.

The smaller child is found at lines 13-15. We have to be careful because the last node in an even-sized heap is an only child; we cannot always assume that there are two children, which is why we have the first test at line 13.

Line 15: right child is smaller than left child

Line 17: Move the smaller child up

The buildHeap operation

The buildHeap operation

- The buildHeap operation takes a complete tree that does not have heap order and reinstates it.
- We want it to be a linear-time operation, since N insertions could be done in O(N log N) time.
- The N successive insertions do more work than we require because they maintain heap order after every insertion and we need heap order only at one instant.

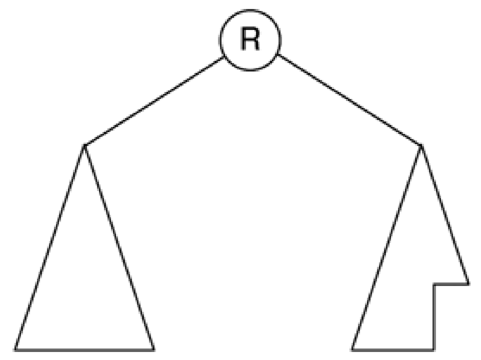


figure 21.15

Recursive view of the heap

We recursively call bui1dHeap on the left and right subheaps. At that point, we are guaranteed that heap order has been established everywhere except at the root.

We can establish heap order everywhere by calling percolateDown for the root.

buildHeap

- The recursive routine works by guaranteeing that when we apply percolateDown(i), all descendants of i have been processed recursively by their own calls to percolateDown.
- The recursion, however, is not necessary, for the following reason:
  - If we call percolateDown on nodes in reverse level order, then at the point percolateDown(i) is processed, all descendants of node i will have been processed by a prior call to percolateDown.
- This process leads to an incredibly simple algorithm for buildHeap.
- Note that percolateDown need not be performed on a leaf. Thus we start at the highest numbered nonleaf node.

```java
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

figure 21.16

Implementation of the linear-time `buildHeap` method

Start at the highest numbered non-leaf node.

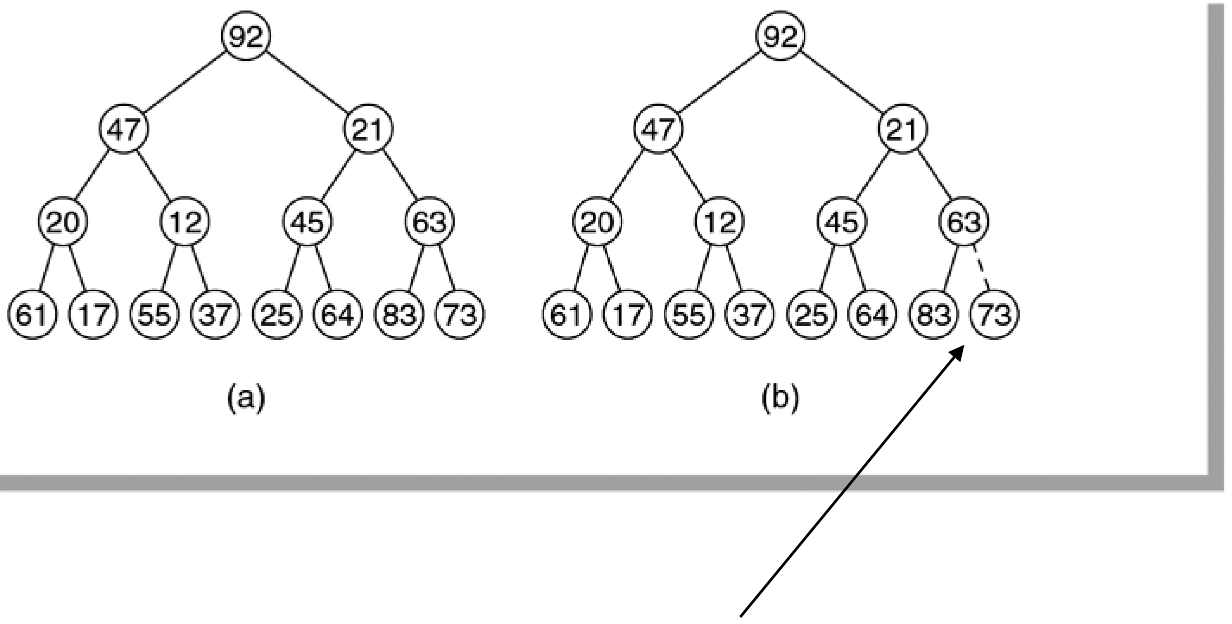


figure 21.17

(a) Initial heap; (b) after `percolateDown(7)`

Figures 21.17(b) through 21.20 show the result of each of the seven percolateDown operations. Each dashed line corresponds to two comparisons: one to find the smaller child and one to compare the smaller child with the node.

figure 21.18

(a) After `percolateDown(6);` (b) after `percolateDown(5)`

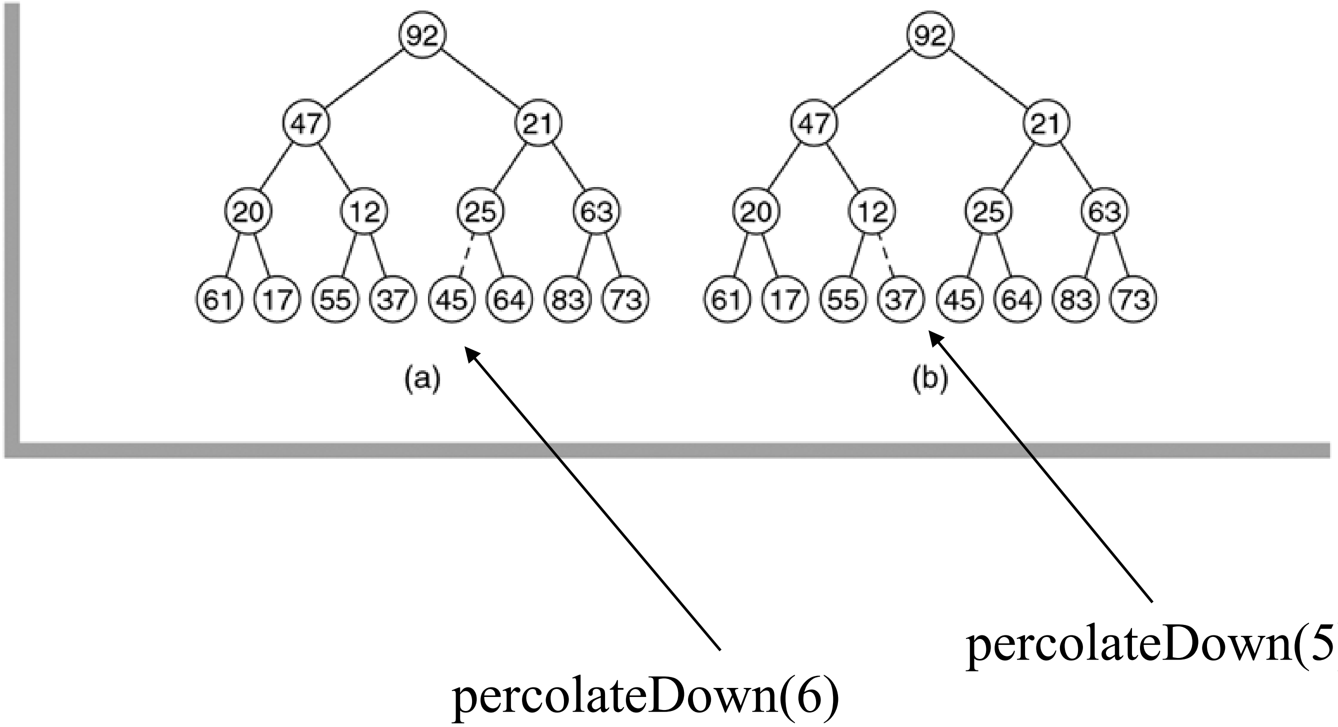


figure 21.19

(a) After `percolateDown(4);`
(b) after `percolateDown(3)`

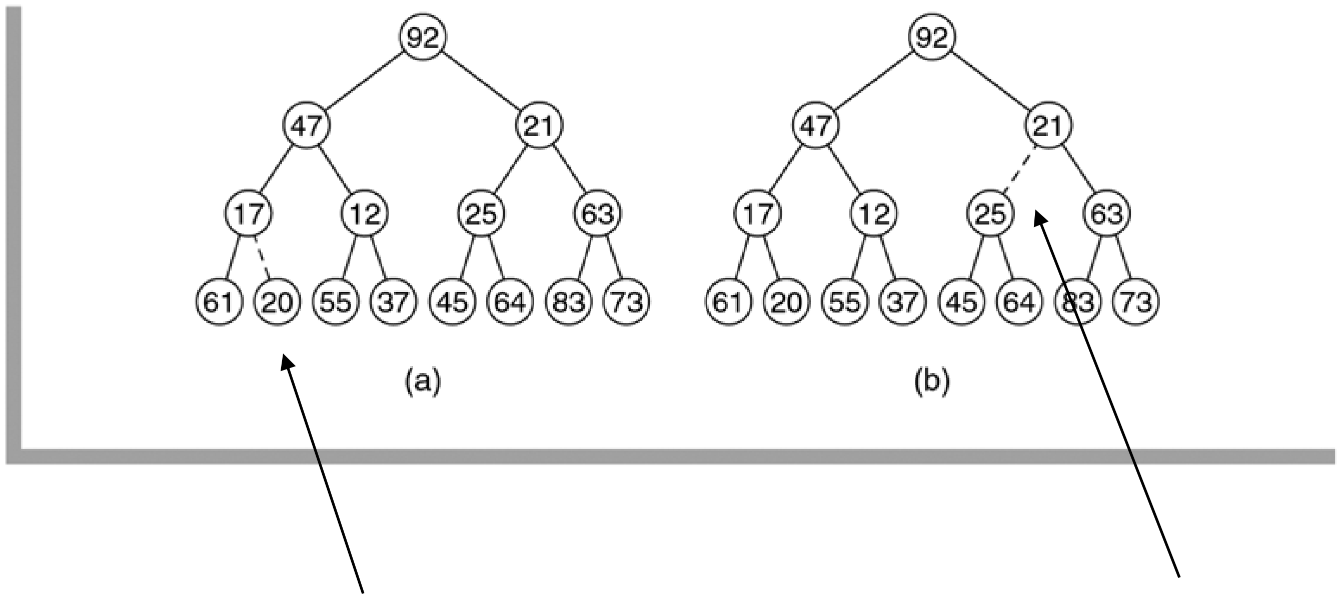
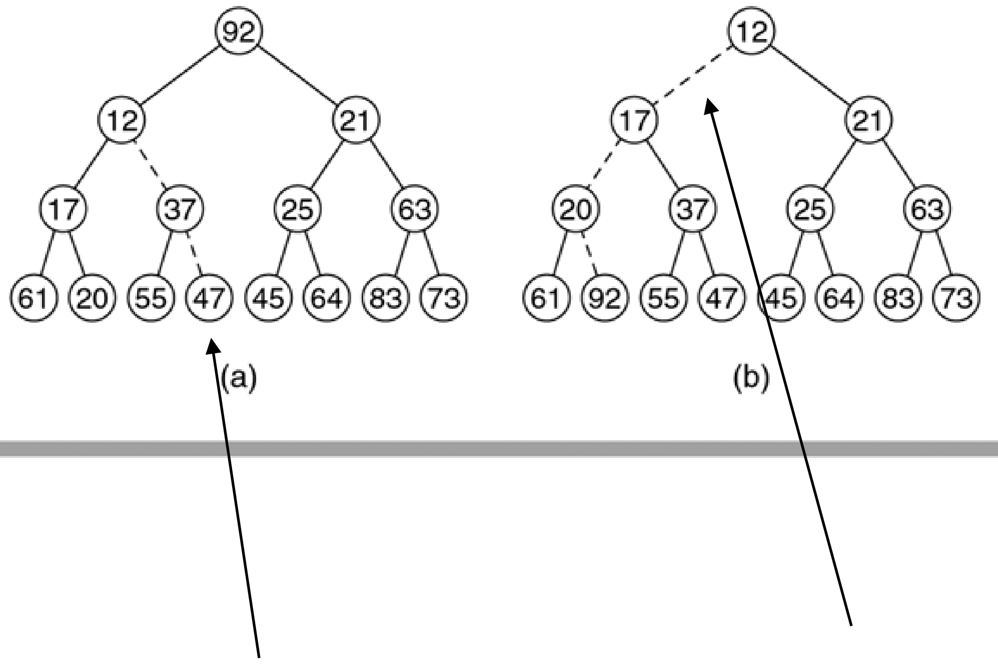


figure 21.20

(a) After `percolateDown(2);` (b) after `percolateDown(1)` and `buildHeap` terminates



Heapsort

The priority queue can be used to sort N items by the following:

1. Inserting every item into a binary heap
2. Extracting every item by calling deleteMin N times, thus sorting the result

Using the observations on buildHeap, we can more efficiently implement this procedure by

1. Tossing each item into a binary heap
2. Applying buildHeap
3. Calling deleteMin N times, with the items exiting the heap in sorted order

HeapSort

- Step 1 takes linear time total, and step 2 takes linear time.
- In step 3, each call to deleteMin takes logarithmic time, so N calls take O(N log N) time.
- Consequently, we have an O(N log N) worst-case sorting algorithm, called heapsort.

Space problem

- Even though we do not use the heap class directly, we still seem to need a second array.
- The reason is that we have to record the order in which items exit the heap equivalent in a second array and then copy that ordering back into the original array.
- The memory requirement is doubled, which could be crucial in some applications.
- Note that the extra time spent copying the second array back to the first is only O(N).
- The problem is space.

Space problem

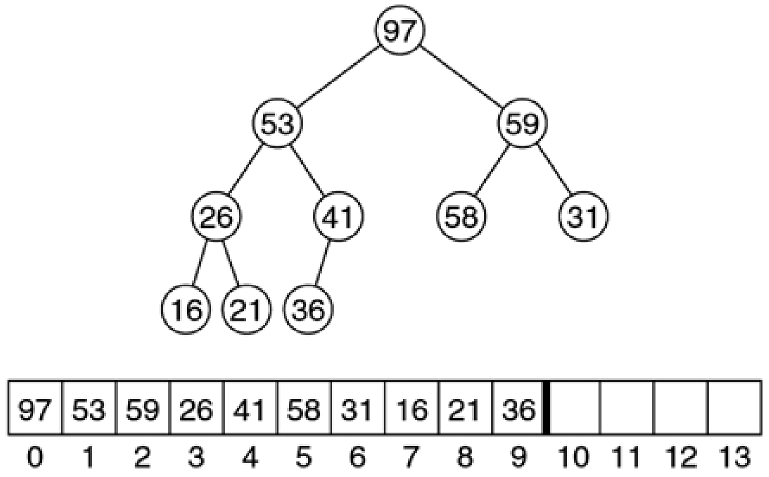
- A clever way to avoid using a second array makes use of the fact that, after each deleteMin, the heap shrinks by 1.
- Thus the cell that was last in the heap can be used to store the element just deleted.
- As an example, suppose that we have a heap with six elements.
  - The first deleteMin produces A_1. Now the heap has only five elements, so we can place A_1 in position 6.
  - The next deleteMin produces A_2. As the heap now has only four elements, we can place A_2 in position 5.

Max Heap

- After the last deleteMin the array will contain the elements in decreasing sorted order.
- If we want the array to be in the more typical increasing sorted order, we can change the ordering property so that the parent has a larger key than the child does.
- Thus we have a max heap.

figure 21.25

Max heap after the `buildHeap` phase



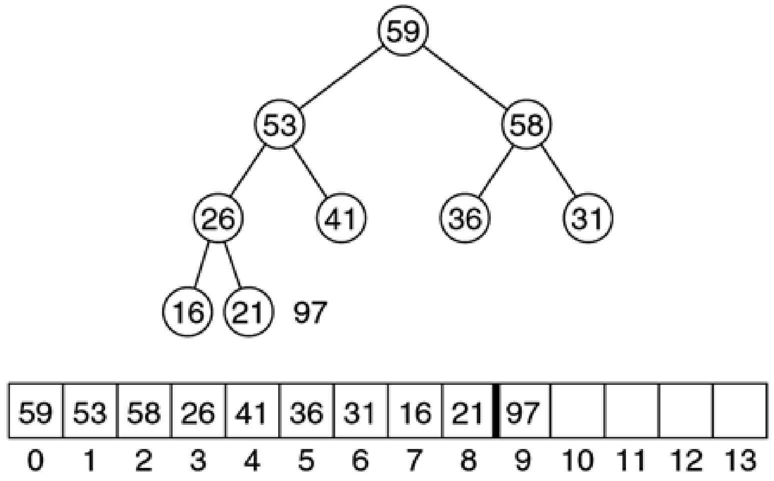


figure 21.26

Heap after the first deleteMax operation

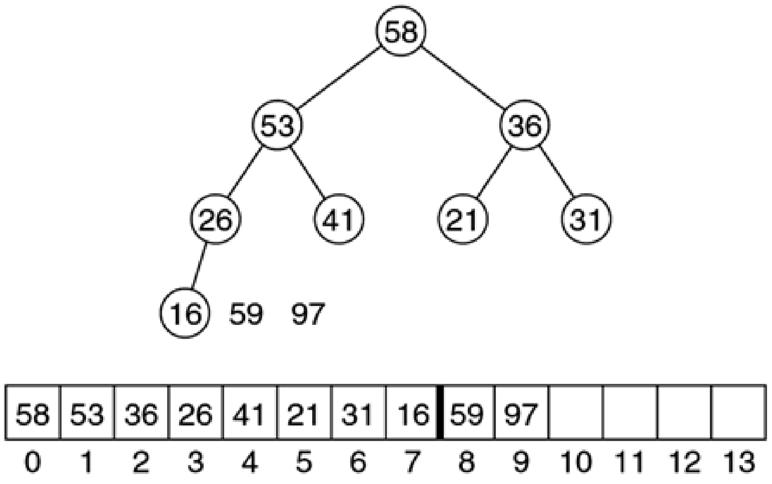


figure 21.27

Heap after the second deleteMax operation

figure 21.28

The heapSort routine

```
// Standard heapsort.
public static <AnyType extends Comparable<? super AnyType>>
void heapsort( AnyType [ ] a )
{
    for( int i = a.length / 2; i >= 0; i-- )  // Build heap
        percDown( a, i, a.length );
    for( int i = a.length - 1; i > 0; i-- )
    {
        swapReferences( a, 0, i );             // deleteMax
        percDown( a, 0, i );
    }
}
```

Swap the maximum from 0 with cell i

Readings

- Chapter 21