CS 3410 – Ch 21 – Priority Queue: Binary Heap

Sections	Pages
21.1-21.5	807-826

21

21.1 – Basic Ideas

1. The *priority queue* is a data structure that provides access to the item with minimum key. We can implement a priority in many different ways:

   a. Linked List

Operation	Average	Worst Case
findMin	$O(n)$	$O(n)$
deleteMin	$O(n)$	$O(n)$
insert	$O(1)$	$O(1)$

   b. Sorted Linked List

Operation	Average	Worst Case
findMin	$O(1)$	$O(1)$
deleteMin	$O(1)$	$O(1)$
insert	$O(n)$	$O(n)$

   c. Binary Search Tree

Operation	Average	Worst Case
findMin	$O(\log n)$	$O(n)$
deleteMin	$O(\log n)$	$O(n)$
insert	$O(\log n)$	$O(n)$

   d. Balanced Search Tree (red-black tree, aa-tree)

Operation	Average	Worst Case
findMin	$O(\log n)$	$O(\log n)$
deleteMin	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(\log n)$

   e. Splay Tree – Ch. 22 (not covered)

Operation	Amortized Worst Case
findMin	$O(\log n)$
deleteMin	$O(\log n)$
insert	$O(\log n)$

   f. Binary Heap

Operation	Average	Worst Case
findMin	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
insert	$O(1)$	$O(\log n)$

2. In this chapter, we study the binary heap, which is the classic method for implementing a priority queue. Like the binary search tree it imposes a structure property and an order property.

21.1.1 – Structure Property

1. The binary heap's data is organized as a tree.

2. 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 with no missing nodes. In the figure below, the tree is complete. If node J were a right child of E, then it would not be complete.

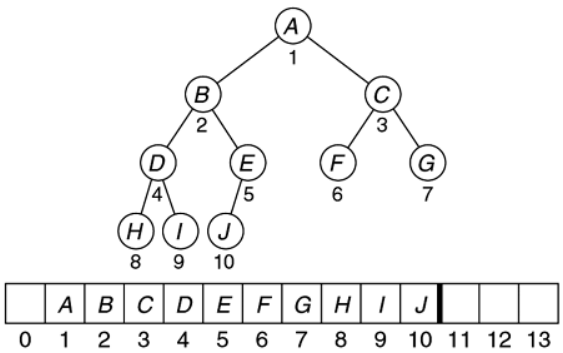


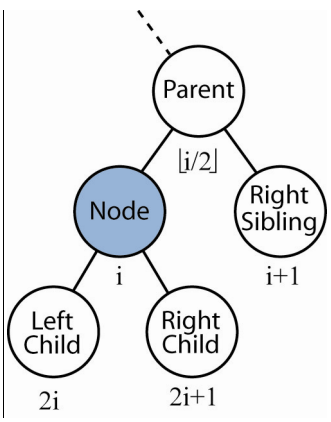
figure 21.1

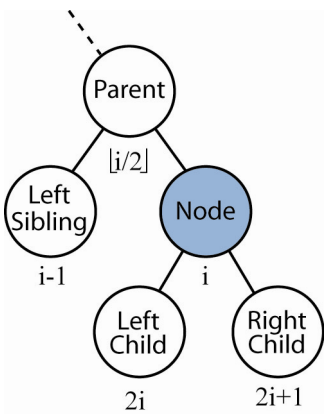
A complete binary tree and its array representation

3. The height of a complete binary tree is at most $\lfloor \log n \rfloor$.

4. As shown in the figure above, we can represent a complete binary tree by storing its *level order traversal* in an array because, by definition, there are not gaps. Thus, the left and right child links are not needed in a complete binary tree. However, inserting or removing an element will require us to modify things to maintain this structure.

5. For any node/element in position i, we can simply find it's children, parent, and sibling as shown in the figure below.





6. We will reserve position 0 in the array for a dummy node that will represent the root node's parent. This will help when we look at implementation.

7. Using an array to store a tree is called an *implicit representation* of a tree. Using the implicit representation we can see that traversing the tree is simple if we follow the rules above.

8. Thus, a heap is simply a binary tree represented as an array of objects.

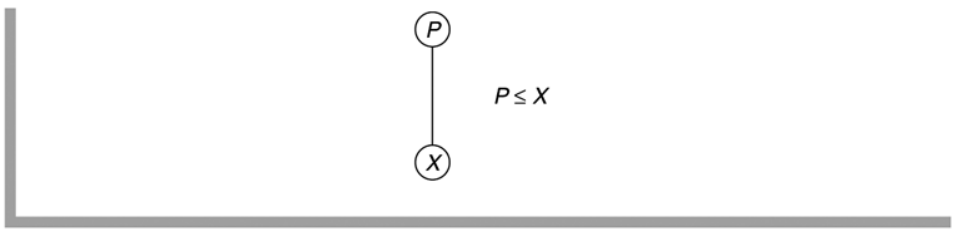
Homework 21.1

1. Consider an implicit representation of a heap of size 17. Is the 11^{th} element (a) a leaf? (b) a left child?, (c) What level is the 11^{th} element on (assume the root element is on level 1). Demonstrate that your answers are true.
2. Generalize your result from problem 1 to the case where the size is n and we are considering the i^{th} element. (a) Develop a rule that tells whether the i^{th} element is (a) a leaf, (b) a left child, (c) what level it is on.
3. Consider a heap of size n. In the implicit representation, what positions are the leaves found in? *e.g.* the range of indices.

21.1.2 – Heap-Order Property

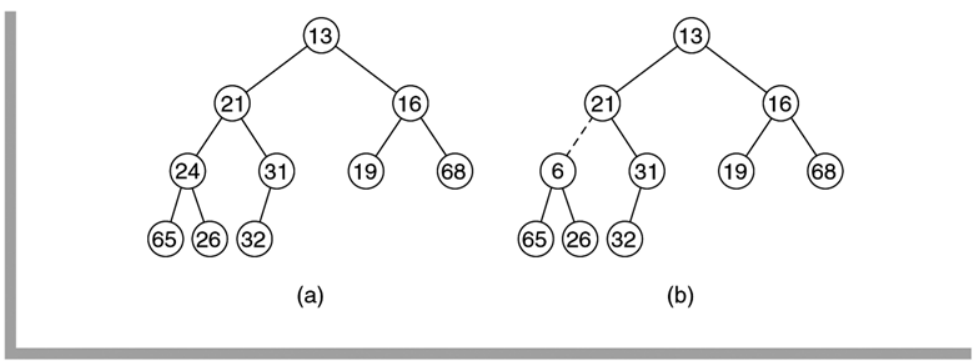
1. Heap-Order property: A heap always stores the smallest element at the root. Recursively, then any node should be smaller than any of its descendants. Thus, the *heap-order* property is that for any node *P*, its key must be less than or equal to the key for any of its descendants. Note that this is a weaker definition of order than the one we use with the binary search tree.

figure 21.2
Heap-order property



2. Figure 21.3 a shows a heap and Figure 21.3 b shows a binary tree that is not a heap because 21 is not less than or equal to 6.

figure 21.3
Two complete trees: (a) a heap; (b) not a heap



3. We use the value of $-\infty$ for the dummy node in position 0.

4. By the heap-order property, we see that the minimum is always in the root of the tree, or the item with index 1 using the implicit representation. Thus, *findMin* is a constant time operation.

Homework 21.2

1. Problem 21.1 & 21.2 from text.
2. Explain why a maximum element must always be a leaf.
3. Suppose that a heap always stores the largest element at the root. Define a heap-order property such that the *findMax* operation is $O(1)$.

21.1.3 – Data Structure – Allowed Operations

1. The historical names: *findMin*, *deleteMin*, *insert* are replaced in our implementation with *element*, *remove*, and *add,* respectively, to follow the java.util.PriorityQueue conventions.

Operation	Method[1]	Method[2]	Descriptions
Insert	add(item)	offer(item)	adds item to queue
Remove	remove()	poll()	retrieves and removes head of queue
Examine	element()	peek()	retrieves, but does not remove head of queue

1 – Throws an exception upon failure
2 – Returns a special value upon failure (*null* or *false*)

2. Below, the PriorityQueue interface is shown.

```
1  package weiss.util;
2
3  /**
4   * PriorityQueue class implemented via the binary heap.
5   */
6  public class PriorityQueue<AnyType> extends AbstractCollection<AnyType>
7                                       implements Queue<AnyType>
8  {
9      public PriorityQueue( )
10       { /* Figure 21.5 */ }
11     public PriorityQueue( Comparator<? super AnyType> c )
12       { /* Figure 21.5 */ }
13     public PriorityQueue( Collection<? extends AnyType> coll )
14       { /* Figure 21.5 */ }
15
16     public int size( )
17       { /* return currentSize; */ }
18     public void clear( )
19       { /* currentSize = 0; */ }
20     public Iterator<AnyType> iterator( )
21       { /* See online code */ }
22
23     public AnyType element( )
24       { /* Figure 21.6 */ }
25     public boolean add( AnyType x )
26       { /* Figure 21.9 */ }
27     public AnyType remove( )
28       { /* Figure 21.13 */ }
29
30     private void percolateDown( int hole )
31       { /* Figure 21.14 */ }
32     private void buildHeap( )
33       { /* Figure 21.16 */ }
34
35     private int currentSize;   // Number of elements in heap
36     private AnyType [ ] array; // The heap array
37     private Comparator<? super AnyType> cmp;
38
39     private void doubleArray( )
40       { /* See online code */ }
41     private int compare( AnyType lhs, AnyType rhs )
42       { /* Same code as in TreeSet; see Figure 19.70 */ }
43 }
```

figure 21.4

The `PriorityQueue` class skeleton

3. The implementation of the constructors is shown below. Notice that an Object array is used and then cast to an array of the appropriate (generic) type since we cannot create generic arrays in Java.

```
1     private static final int DEFAULT_CAPACITY = 100;
2
3     /**
4      * Construct an empty PriorityQueue.
5      */
6     public PriorityQueue( )
7     {
8         currentSize = 0;
9         cmp = null;
10        array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
11    }
12
13    /**
14     * Construct an empty PriorityQueue with a specified comparator.
15     */
16    public PriorityQueue( Comparator<? super AnyType> c )
17    {
18        currentSize = 0;
19        cmp = c;
20        array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
21    }
22
23
24    /**
25     * Construct a PriorityQueue from another Collection.
26     */
27    public PriorityQueue( Collection<? extends AnyType> coll )
28    {
29        cmp = null;
30        currentSize = coll.size( );
31        array = (AnyType[]) new Object[ ( currentSize + 2 ) * 11 / 10 ];
32
33        int i = 1;
34        for( AnyType item : coll )
35            array[ i++ ] = item;
36        buildHeap( );
37    }
```

A

B

figure 21.5

Constructors for the `PriorityQueue` class

4. Notice the implementation of the third constructor in Figure 21.5 above. Initially, the items in the array are inserted *sloppily* (see *A* in figure 21.5), e.g without regard to heap-order. Then, a method, *buildHeap* is called to establish heap-order (see *B* in figure 21.5). This will be examined later.

5. The *element* method implementation is shown below and simply returns the element in the first position, if it exists. Otherwise, it throws an exception.

```
1     /**
2      * Returns the smallest item in the priority queue.
3      * @return the smallest item.
4      * @throws NoSuchElementException if empty.
5      */
6     public AnyType element( )
7     {
8         if( isEmpty( ) )
9             throw new NoSuchElementException( );
10        return array[ 1 ];
11    }
```

figure 21.6

The `element` routine

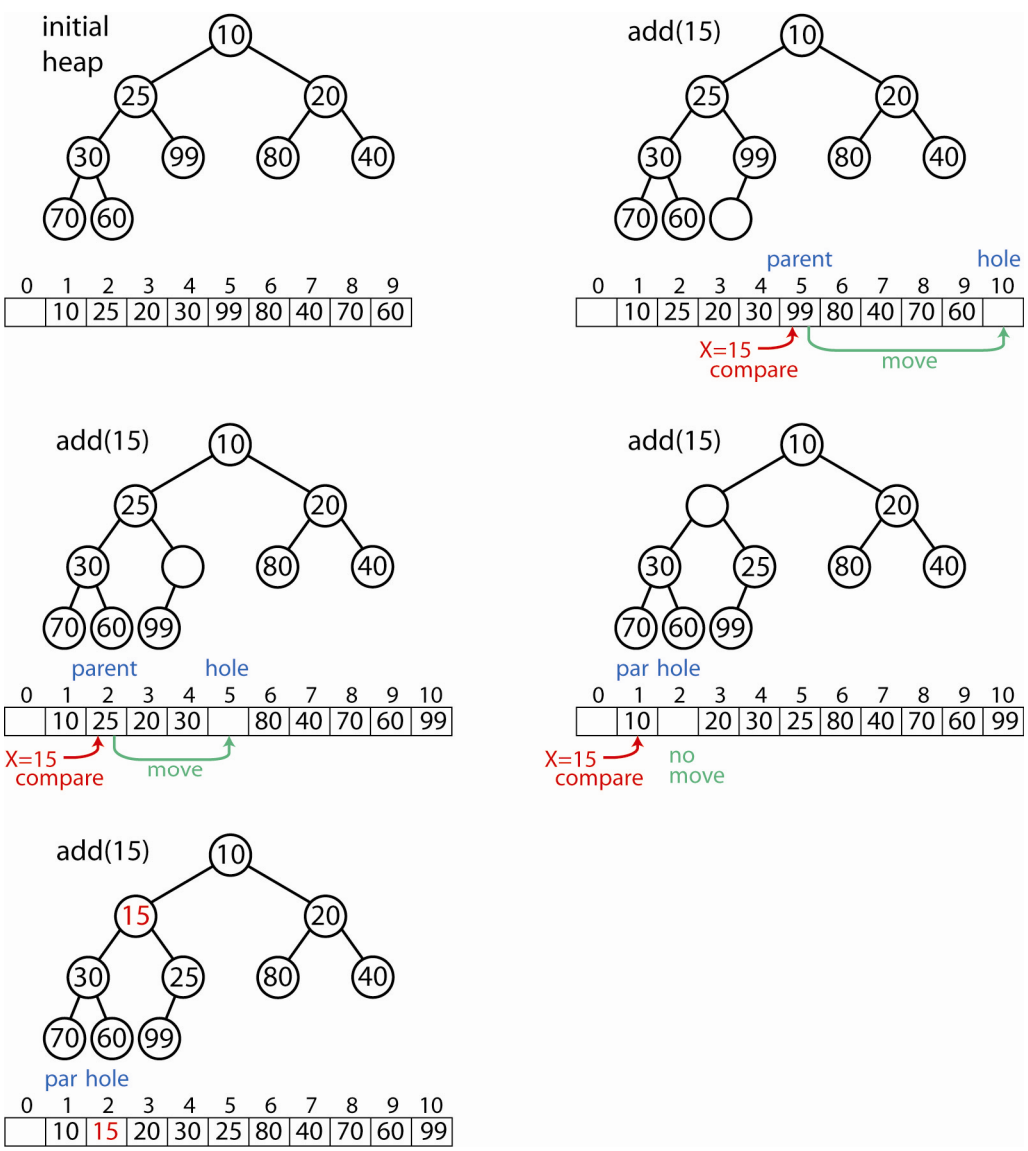
21.2 – Implementation of Basic Operations

21.2.1 – The *add* Method Implementation

1. To *add* an element, *X* in the heap, we must add a node to the tree which must be in the *next* position in order to preserve the structure property. Thus, the node we must add will necessarily occupy the location *++currentSize* in the backing array. However, the element to insert, *X*, does not necessarily belong in this node, as we must preserve the order property as well. To implement *insert,* we return to the tree representation. We refer to the newly created node as a *hole* in the tree.

2. If *X* can be placed in the hole without violating the heap-order property, then we are done. If not, simply move the hole's parent into the hole and repeat this process, asking if *X* can be placed in the new hole. This process is known as *percolate up*. Thus, the algorithm for *add* is to create a *hole* at the next available position and bubble it up until the correct location is found for the new item, *X*.

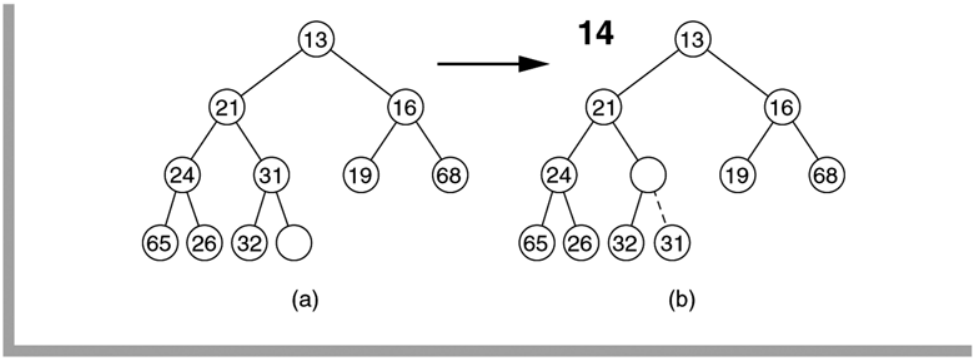
3. Example:



4. Example from text: In the figure below, we are trying to add 14. Clearly, it cannot go in the hole in figure 21.7a since 31 is larger than 14. So, we percolate the hole up and move the parent (31) down. Again, we see that 14 cannot go there as 21 is larger than 14.

figure 21.7

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



5. So, we percolate the hole up again and move the parent (21) down. Finally, we see that 14 can go in the hole now without violating the heap-order property.

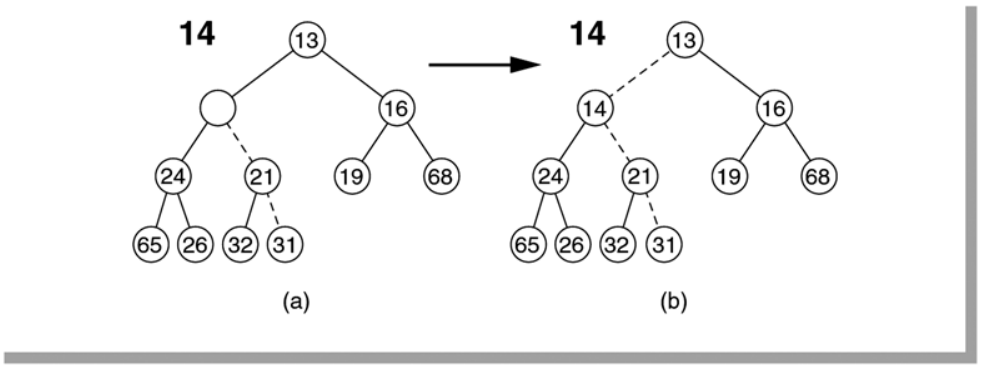


figure 21.8

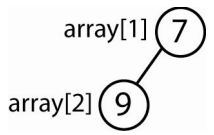
The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

6. The *add* algorithm:

```
add( x )

   // Create position for hole
   hole = ++currentSize
   // Put value in location 0 (not a part of heap)
   // Used to stop the loop below if x belongs in position 1,
   // e.g. it is the smallest.
   array[ 0 ] = x;

   // While heap-order property is not met,
   // e.g. x is less than parent
   while x.key < array[hole/2].key
      // Move parent down
      bubble hole up: array[hole] = array[ hole/2 ]
      // Move hole up
      hole /= 2

   // Put x in the array.
   array[ hole ] = x;
```

7. Example – Shows how array[0] is used to stop the percolation when the root is reached:

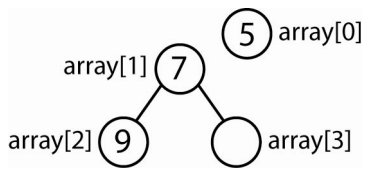
Tree



Code

add(5)

Tree

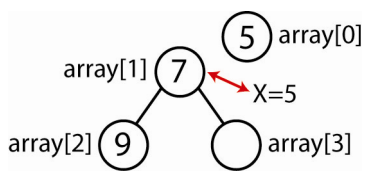


Code

hole = 3

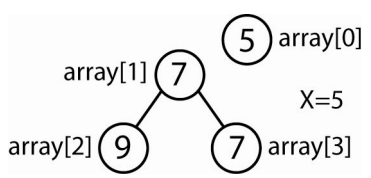
array[0] = 5

Tree



Code

while(5 < array[hole/2]=7)

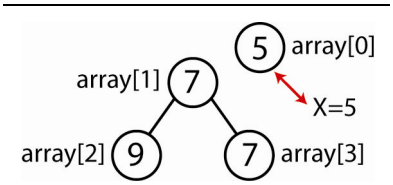


array[hole=3] = array[hole/2=1]

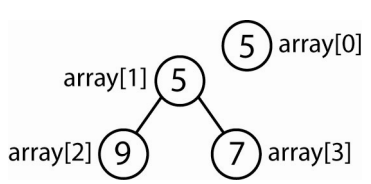
hole = hole / 2 = 1

Tree

Code



while(5 < array[hole/2]=5) --> false



array[hole=1] = 5

8. The implementation of the *add* method:

```
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

9. Thus, in the worst case, when the item being inserted is the minimum, the complexity is $O(\log n)$ because the height of the complete binary tree is $\log n$. It has been shown that on average, a random add does 2.6 comparisons so that it moves an element up an average of 1.6 levels. Thus, *add* is $O(1)$ on average.

Homework 21.3

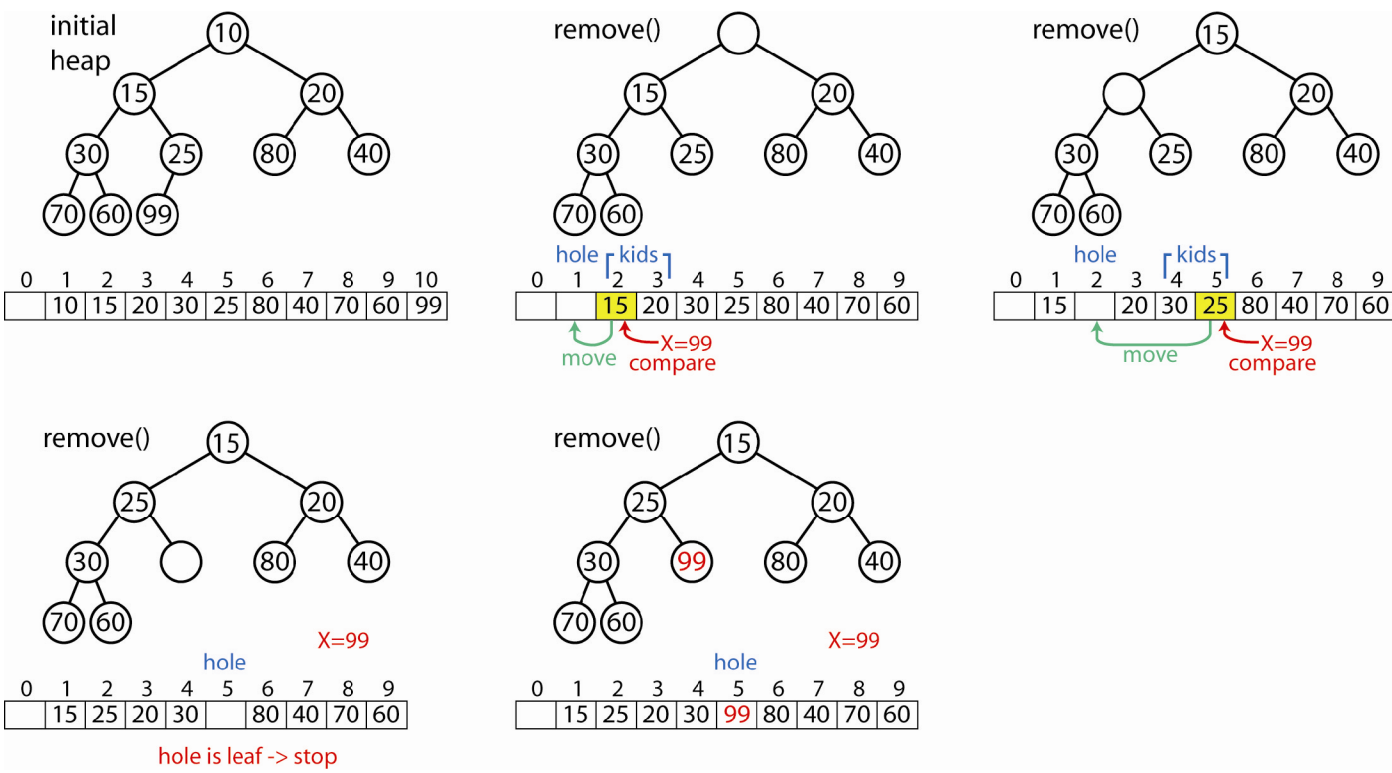
1. Add these items to a heap (in this order): 4, 10, 3, 1, 7, 6, 9, 5, 2. (a) Show a tree representation at each step. (b) Show the final implicit representation.

21.2.3 – The *remove* Method Implementation

1. The *deleteMin* (*remove*) operation will remove and return the minimum item. If we consider this as removing the root node, then this creates a *hole* in the root and reduces the size by one. Now, if we consider the array representation, this means that position 1 is empty and the last item doesn't have a valid position (because we have reduced the size by one). Let's call this last item, *X*. So, essentially, we need to *add X*, but instead of the hole being in the last position, the hole is in the first position.

2. The solution to this problem is to *percolate down* to find where X needs to be placed to preserve the heap-order property. At each step, find the smallest child of the hole, *S*. If *X*<*S*, then put *X* in the hole. Otherwise, swap the hole with *S* and repeat.

3. Example:



4. Example from text:

figure 21.10

Creation of the hole at the root

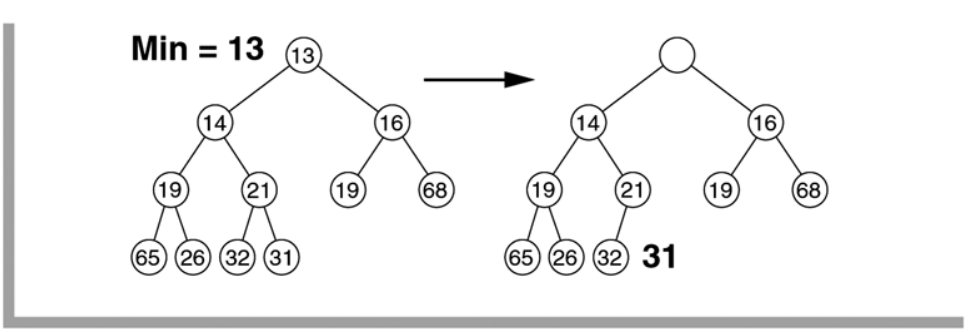
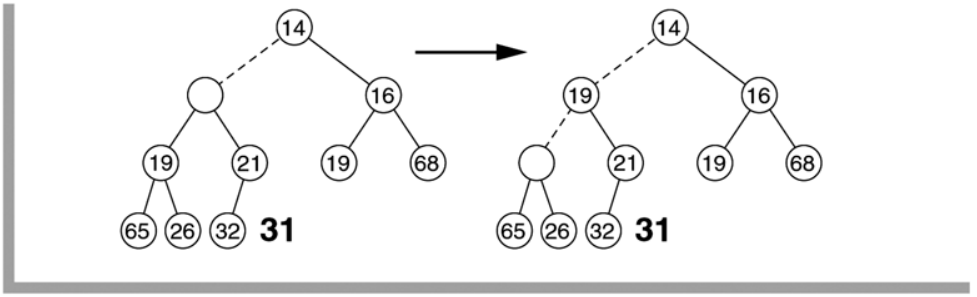


figure 21.11

The next two steps in the `deleteMin` operation



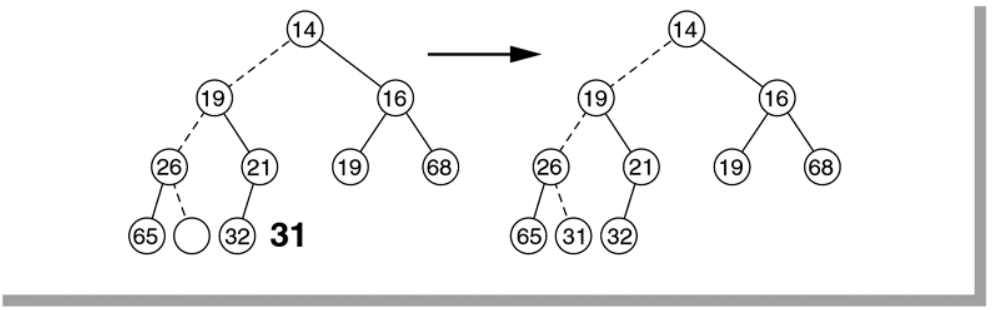


figure 21.12

The last two steps in the `deleteMin` operation

5. The *remove* algorithm:

```
remove()

   // Get minimum
   min = array[1];
   // Get last item
   X = array[ currentSize ]
   // Set hole location
   hole = 1

   // while hole not a leaf nor position found
   while hole has a left child
      // Find smallest child
      S = smallest child
      // If X is bigger than hole's children...
      if( S < X )
            // Bubble hole down...
            // Set hole to smallest child
            array[hole] = S
            // Move hole down to smallest child
            hole = position of S
      // If X is smaller than hole's children...
      else
            // X belongs in hole
            break

   // Put X in hole
   array[ hole ] = x;

   return min
```

6. The implementation of the *remove* method is shown below. Since the percolate down procedure is a bit more complex than percolate up, it is implemented as a helper method. Also, it will be used by the *buildHeap* method as we will see later.

```
1     /**
2      * Removes the smallest item in the priority queue.
3      * @return the smallest item.
4      * @throws NoSuchElementException if empty.
5      */
6     public AnyType remove( )
7     {
8         AnyType minItem = element( );
9         array[ 1 ] = array[ currentSize-- ];
10        percolateDown( 1 );
11
12        return minItem;
13    }
```

figure 21.13

The remove method

A better description of what *percolateDown* does is that it puts whatever is in *hole* initially into the correct (preserve heap-order) location which is the initial *hole,* or any of its descendants, following the path of the smallest child at each step.

```
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                                                        J
10    A   for( ; hole * 2 <= currentSize; hole = child )
11        {
12    B       child = hole * 2;
13    C       if( child != currentSize &&
14    D               compare( array[ child + 1 ], array[ child ] ) < 0 )
15    E           child++;
16    F       if( compare( array[ child ], tmp ) < 0 )
17    G           array[ hole ] = array[ child ];
18            else
19    H           break;
20        }
21        array[ hole ] = tmp;
22    }
```

figure 21.14

The percolateDown method used for remove and buildHeap

Key:

A – Check to see if hole has a left child. B – Get left child position C – Check to see if (left) child has a sibling D – Check to see if right child is less than left child E – If C and D are true, then make the right child the smallest	F – Check to see if item to insert (tmp) is larger than smallest child G – If F is true, then put the value of smallest child in hole H – If F is false, e.g. item to insert is smaller than smallest child, then item blongs in hole. J – Move the hole down to the smallest child.

7. Thus, in the worst case, *remove* has complexity is $O(\log n)$ because the height of the complete binary tree is $\log n$. Not surprisingly, the average case is also $O(\log n)$. This is so because we are taking a relatively "larger" item (the last one) and starting from the top of the tree, working our way down, to see where it belongs. Empirical evidence indicates that this percolation down rarely terminates after just a level or two.

Homework 21.4

1. Consider Homework 21.3, Problem 1. Call *remove* 3 times. (a) Show the resulting tree at each step. (b) Show the implicit representation at the final step.
2. Consider a min-heap data structure where when we call *remove* we remove the smallest item (and return it), but also remove the largest item. Describe an algorithm to do this as efficiently as possible. Hint: Use the result from Homework 21.1, Problem 3 and the results from this section.

21.3 – The *buildHeap* Operation

1. Remember the PriorityQueue has a constructor that creates a PriorityQueue from any Collection:

```
24      /**
25       * Construct a PriorityQueue from another Collection.
26       */
27      public PriorityQueue( Collection<? extends AnyType> coll )
28      {
29          cmp = null;
30          currentSize = coll.size( );
31          array = (AnyType[]) new Object[ ( currentSize + 2 ) * 11 / 10 ];
32
33          int i = 1;
34          for( AnyType item : coll )          A
35              array[ i++ ] = item;
36          buildHeap( );                       B
37      }
```

figure 21.5

Constructors for the `PriorityQueue` class

Initially, the items in the collection, *coll* are inserted *sloppily* (see *A* in figure 21.5), e.g without regard to heap-order. Then, a method, *buildHeap* is called to establish heap-order (see *B* in figure 21.5). We will show shortly that *buildHeap* is $O(n)$. The complexity of the sloppy inserts is $O(n)$. Thus, the complexity of this constructor is $O(n) + O(n) = O(n)$. Another strategy would be to simply use the PriorityQueue's *add* method to add each of the items in the collection. As shown previously, the add method is $O(\log n)$, so that n adds would have $O(n \log n)$ which is *slower* than the constructor shown above.

2. Thus, the *buildHeap* operation takes a complete tree that does not have heap order and reinstates it.

3. A simple way to achieve this is to call *percolateDown* on the nodes in reverse level-order. Thus, when we call *percolateDown(i),* then we know that the heap has the heap-order property for all of *i*'s descendants. Notice that this process does not need to call *percolateDown* on leaf nodes, so we start with the highest numbered non-leaf node, currentSize/2.

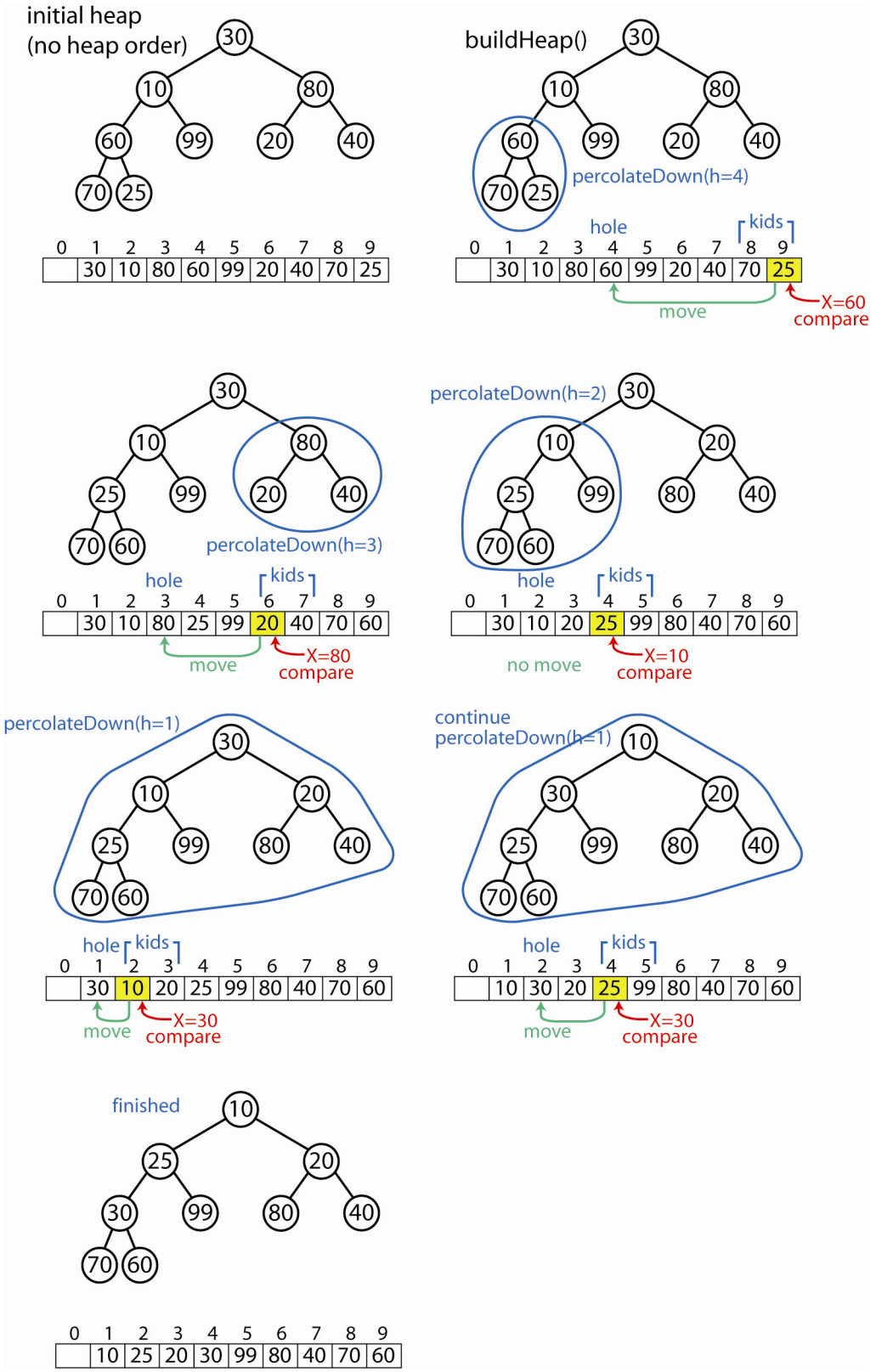
4. The Java implementation

```
1     /**
2      * Establish heap order property from an arbitrary
3      * arrangement of items. Runs in linear time.
4      */
5     private void buildHeap( )
6     {
7         for( int i = currentSize / 2; i > 0; i-- )
8             percolateDown( i );
9     }
```

figure 21.16

Implementation of the linear-time `buildHeap` method

5. Example:


initial heap
(no heap order)
buildHeap()
percolateDown(h=4)
hole
kids
move
X=60
compare
percolateDown(h=3)
X=80
percolateDown(h=2)
no move
X=10
percolateDown(h=1)
X=30
continue
percolateDown(h=1)
finished

6. Example from text:

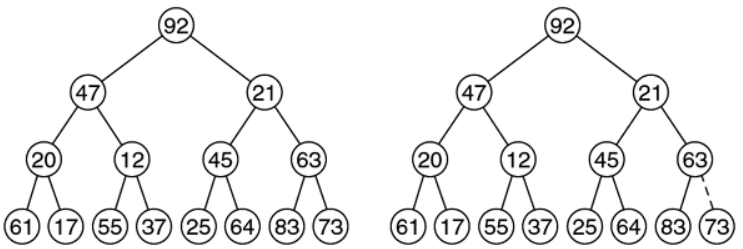


figure 21.17

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

figure 21.18

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

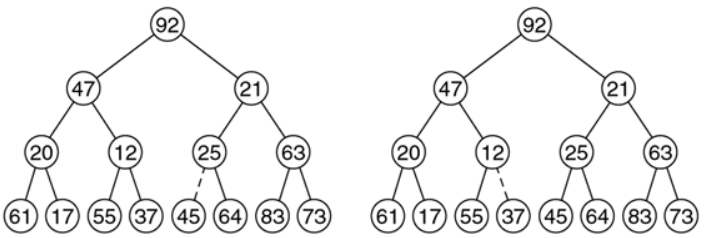


figure 21.19

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

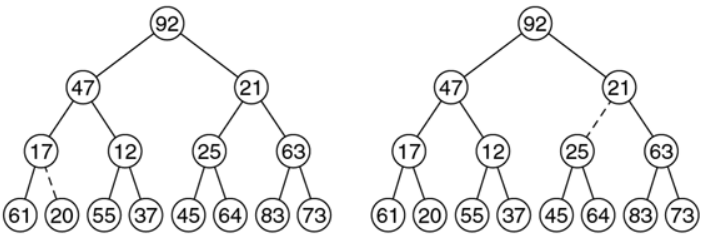
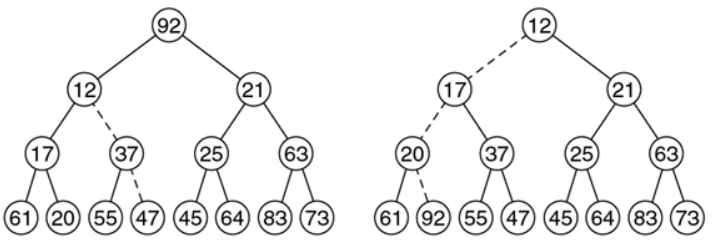


figure 21.20

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



7. It can be shown that *buildHeap* is $O(n)$.

Homework 21.5

1. Consider this input to *buildHeap*: 4, 10, 3, 1, 7, 6, 9, 5, 2. (a) Show a tree representation at each step of *buildHeap*. (b) Show the final implicit representation.

21.4 – Advanced Operations: *decreaseKey* and *merge*

1. Sometimes, a priority queue needs to support a *decreaseKey(location,value)* operation that lowers the value of an item in the priority queue, which can destroy the heap-order property. The solution is simple, keep percolating up until the order is restored.

2. Another operation that sometimes needs support is *merge* where two priority queues are merged. The most efficient solution is to copy the elements from the smaller queue to the larger and then call *buildHeap*.

21.5 – Internal Sorting: heapsort

1. The priority queue can be used to sort N items by:

   Put all items in priority queue
   Extract each item by calling *remove*

   Thus, the items will be removed in ascending order.

2. The most efficient way to do this is to use the priority queue constructor that takes a collection of items, places them sloppily into the array, and then calls *buildHeap*. Thus, a heapSort sorting algorithm is:

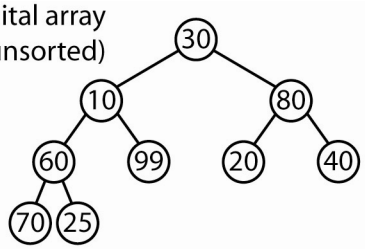
   1. Copy each item into the binary heap, $O(n)$
   2. Call buildHeap, $O(n)$
   3. Call remove, n times, $O(n \log n)$

3. Notice that if we are sorting an array of items, we can do a heap sort without a priority queue, by simply applying the heap concepts to the input array, and, introducing a second array to store the order items that are removed from the input array, and then when all items have been removed from the input array, copy the second array back to the first.

4. This can be done without a second array. Since every remove shrinks the heap by 1, we can use the last cell to store the item that was just deleted. At the completion, all the items will be in decreasing order.

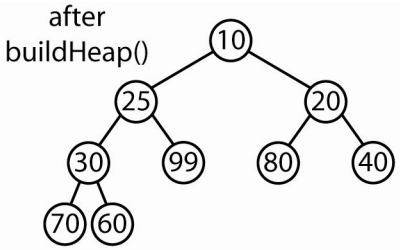
5. Example

inital array (unsorted)



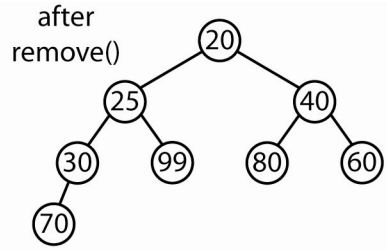
0	1	2	3	4	5	6	7	8	9
	30	10	80	60	99	20	40	70	25

after buildHeap()



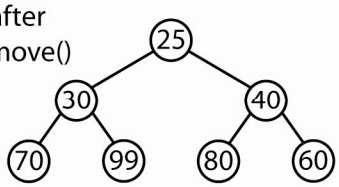
0	1	2	3	4	5	6	7	8	9
	10	25	20	30	99	80	40	70	60

after remove()



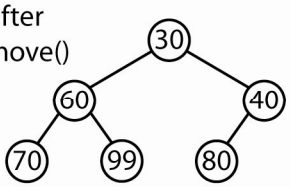
0	1	2	3	4	5	6	7	8	9
	20	25	40	30	99	80	60	70	10

after remove()



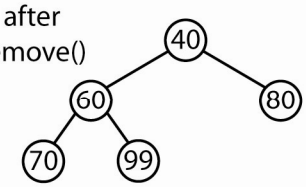
0	1	2	3	4	5	6	7	8	9
	25	30	40	70	99	80	60	20	10

after remove()



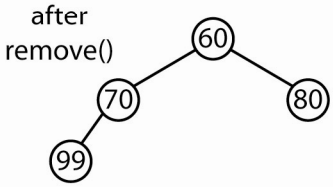
0	1	2	3	4	5	6	7	8	9
	30	60	40	70	99	80	25	20	10

after remove()



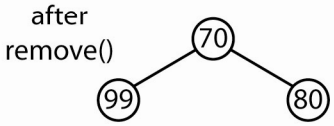
0	1	2	3	4	5	6	7	8	9
	40	60	80	70	99	30	25	20	10

after remove()



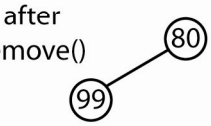
0	1	2	3	4	5	6	7	8	9
	60	70	80	99	40	30	25	20	10

after remove()



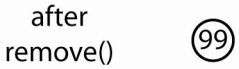
0	1	2	3	4	5	6	7	8	9
	70	99	80	60	40	30	25	20	10

after remove()



0	1	2	3	4	5	6	7	8	9
	60	70	70	60	40	30	25	20	10

after remove()



0	1	2	3	4	5	6	7	8	9
	99	80	70	60	40	30	25	20	10

after remove()

0	1	2	3	4	5	6	7	8	9
	99	80	70	60	40	30	25	20	10

6. To return the array in the more usual ascending sorted order, we can just slightly alter the logic, so that we are using the concept of a max heap, where the parent is always larger than the children.

7. The Java implementation:

figure 21.28
The heapSort routine

```
1     // Standard heapsort.
2     public static <AnyType extends Comparable<? super AnyType>>
3     void heapsort( AnyType [ ] a )
4     {
5         for( int i = a.length / 2; i >= 0; i-- )  // Build heap
6             percDown( a, i, a.length );
7         for( int i = a.length - 1; i > 0; i-- )
8         {
9             swapReferences( a, 0, i );            // deleteMax
10            percDown( a, 0, i );
11        }
12    }
```

8. The *percDown* method has 3 subtleties:

    1. Since we are using a max heap, the comparison logic must switch to > instead of <
    2. There is no sentinel item in position 0, as arrays typically use position 0. This affects the calculation of the parent of a node and it's children.
    3. It must be informed of the current heap size which is lowered by 1 every time an item is removed

9. The average and worst case complexity of heasort is $O(n \log n)$. This is so because we essentially do *buildHeap*, $O(n)$ followed by n *percDown* calls, $O(n \log n)$. Thus, $O(n) + O(n \log n) = O(n \log n)$.

Homework 21.6

1. Describe in detail a heap-sort algorithm that accepts an array as input and returns the sorted array in ascending order and does not create any additional arrays. You may use some of the methods from the heap data structure (properly modified).
2. Problem 21.7 from text.