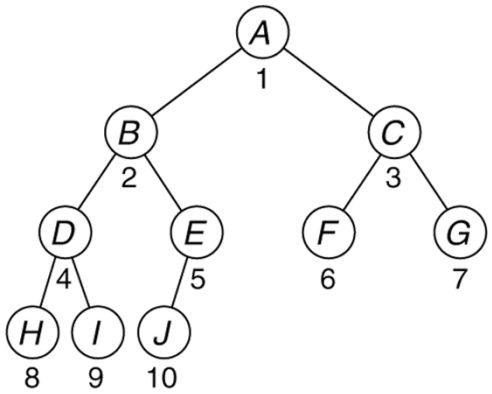
Priority Queue

Basic operations
Implementation options
Binary Heaps

Recap: Priority Queue operations

- Each element in the PQ has an associated **priority**, which is a non-negative integer.
- findMin()
- insert(item, priority)
- deleteMin( )

Recap: A complete binary tree and its array representation



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-order property

$P \leq X$

Figure 21.2
Heap-order property

A Binary Heap is a complete Binary Tree (implemented as an array) that has the heap-order property.

In a binary heap, where do we find
- The smallest element?
- 2nd smallest?
- 3rd smallest?

Figure 21.7
Attempt to insert 14, creating the hole and bubbling the hole up

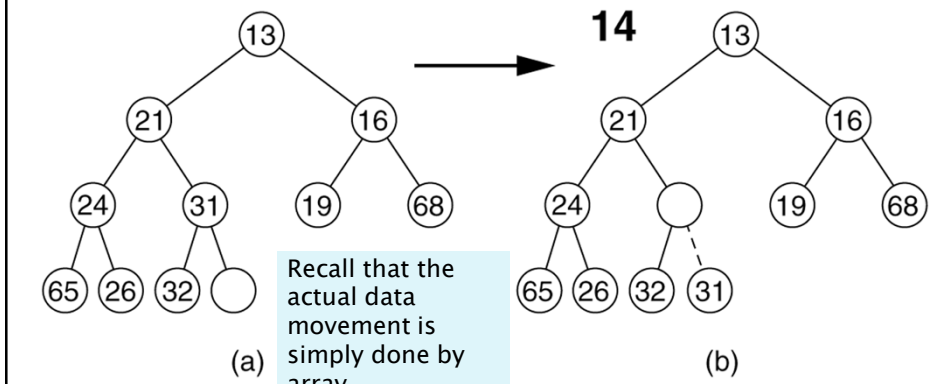
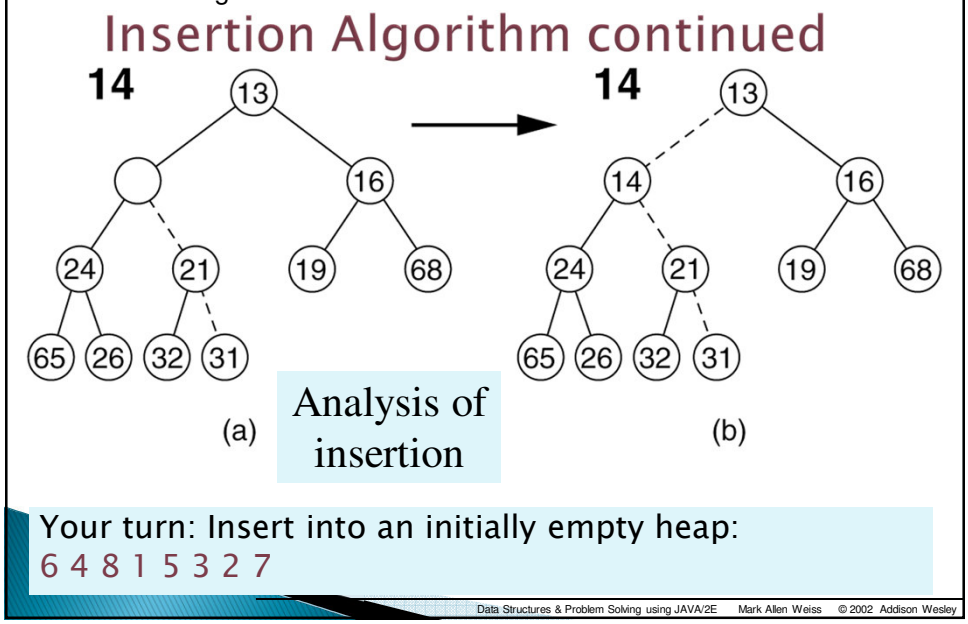


Figure 21.8
The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

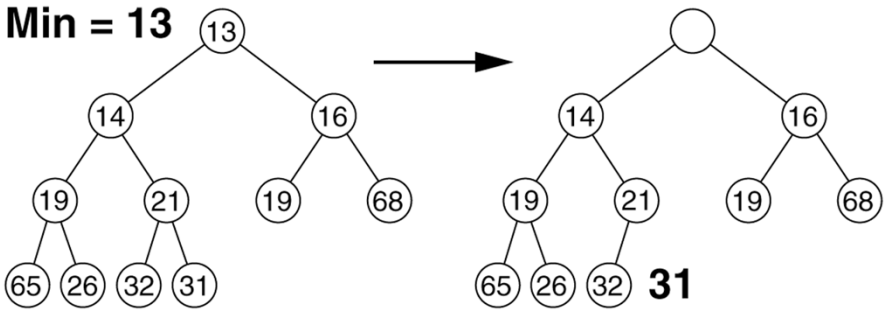


Code for Insertion

```
public PriorityQueue.Position insert( Comparable x )
{
    if( currentSize + 1 == array.length )
        doubleArray( );

        // Percolate up
    int hole = ++currentSize;
    array[ 0 ] = x;

    for( ; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;

    return null;
}
```

DeleteMin algorithm

The *min* is at the root. Delete it, then use the **percolateDown** algorithm to find the correct place for its replacement.



We must decide which child to promote, to make room for 31.

Figure 21.10 Creation of the hole at the root

Figure 21.11
The next two steps in the deleteMin operation

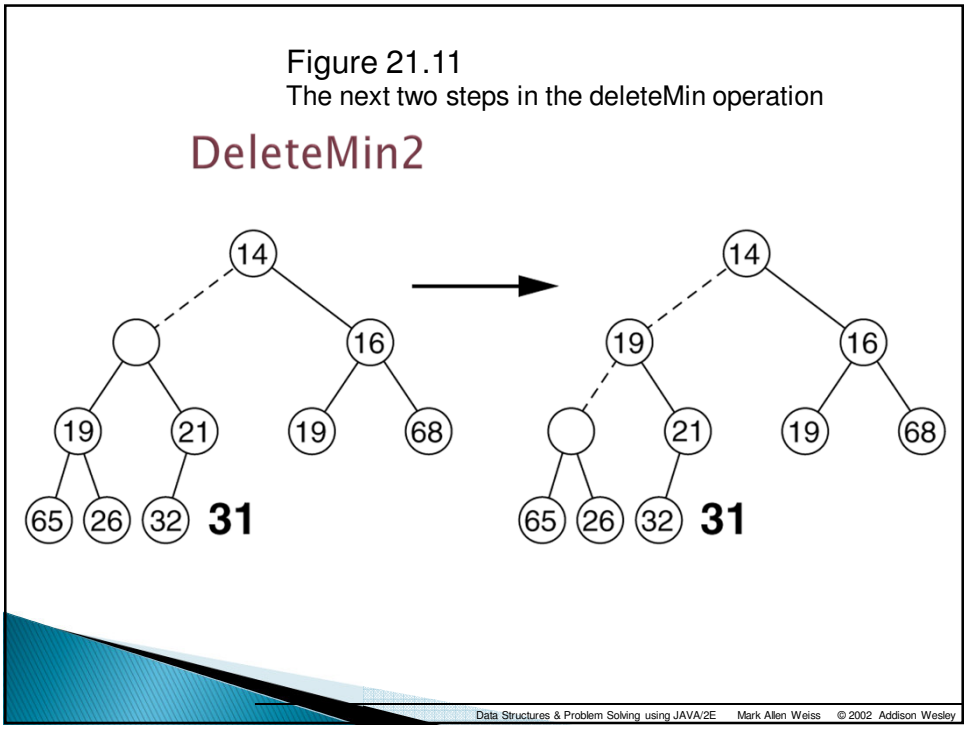
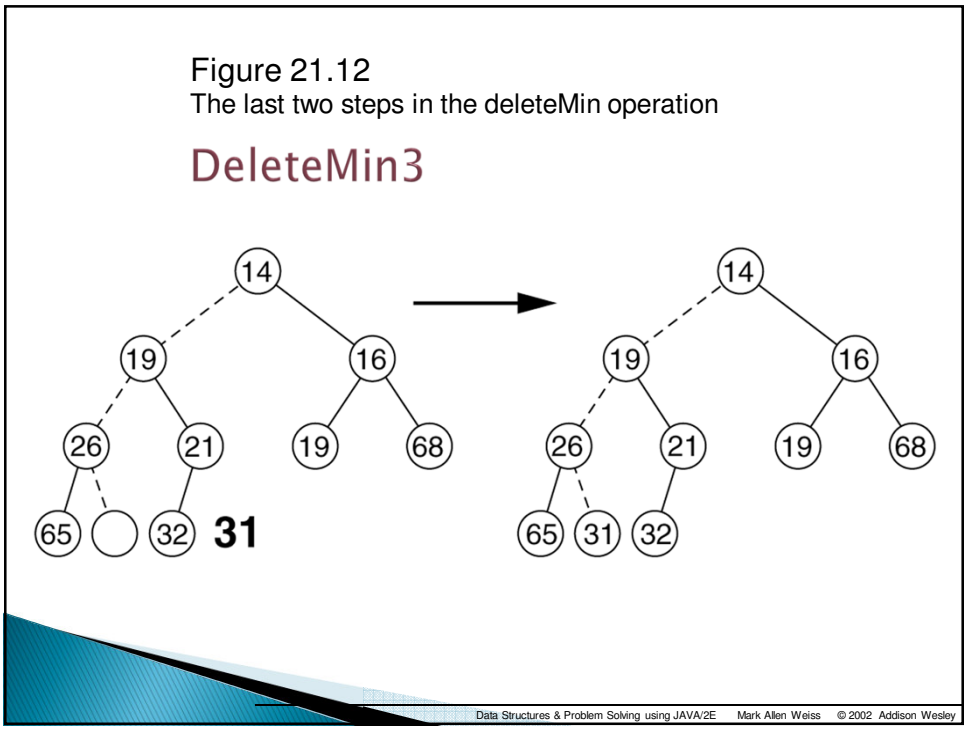


Figure 21.12
The last two steps in the deleteMin operation



```
public Comparable deleteMin( )
{
    Comparable minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}
private void percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize &&
                array[ child + 1 ].compareTo( array[ child ] ) < 0 )
            child++;
        if( array[ child ].compareTo( tmp ) < 0 )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```

Compare node to its children, moving root down and promoting the smaller child until proper place is found.

Analysis

Summary: Implementing a Priority Queue as a binary heap

- Worst case times:
  - findMin: O(1)
  - insert: O(log n)
  - deleteMin O(log n)
- big-oh times for insert/delete are the same as the balanced BST implementation,
  - but these operations are much simpler,
  - and don't require space for pointers.

Using a Heap for sorting

- Start with empty heap
- Insert each array element into heap
- Repeatedly do deleteMin, copying elements back into array.
- http://nova.umuc.edu/~jarc/idsv/lesson3.html
  - Can be run in demo mode or practice mode.
- We can save space by doing the whole sort in place, using a "maxHeap" (i.e. a heap where the maximum element is at the root instead of the minimum)
- Analysis?
  - **Next slide …**

Analysis of simple heapsort

- Add the elements to the heap
  - Repeatedly call insert
- Remove the elements and place into the array
  - Repeatedly call DeleteMin
- We can do better for the insertion part by using BuildHeap

BuildHeap takes a complete tree that is not a heap and exhanges elements to get it into heap form

At each stage it takes a root plus two heaps and "percolates down" the root to restore "heapness" to the entire subtree

```
/**
 * 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 );
}
```

Why this starting point?

Figure 21.17 Implementation of the linear-time buildHeap method

```
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

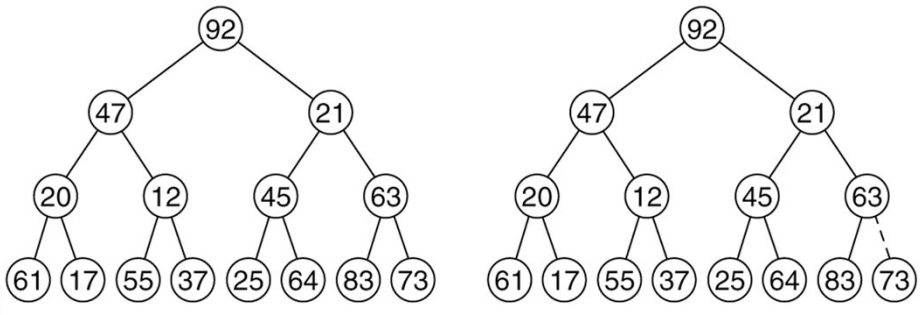


Figure 21.18
(a) After percolateDown(6);
(b) after percolateDown(5)

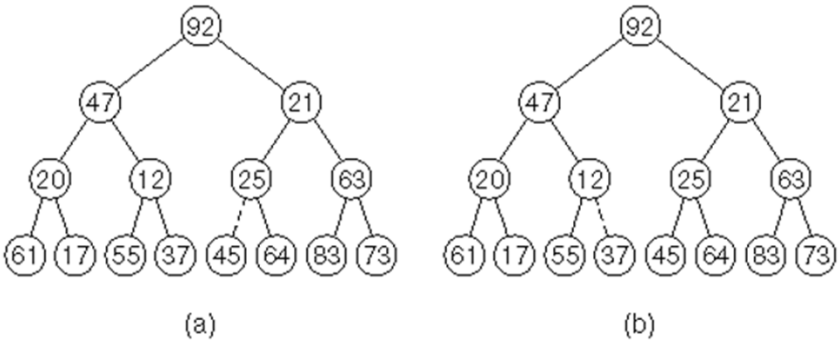


Figure 21.19
(a) After percolateDown(4);
(b) after percolateDown(3)

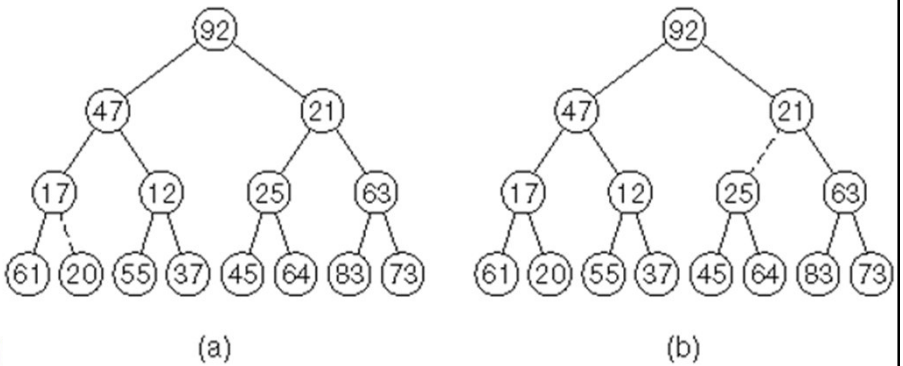
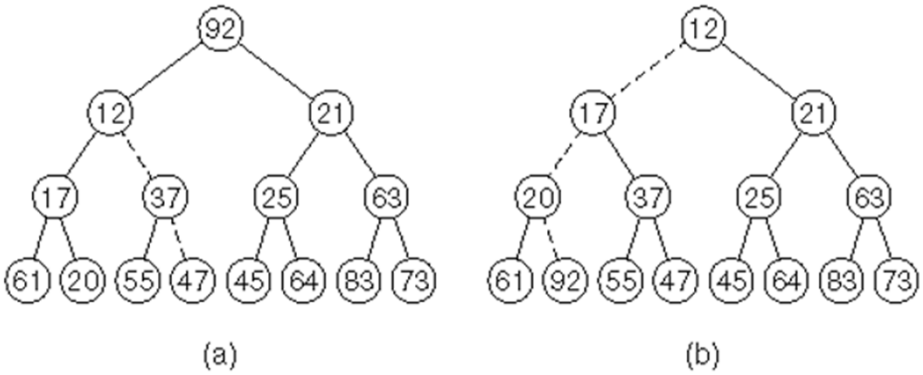


Figure 21.20
(a)After percolateDown(2);
(b) after percolateDown(1) and buildHeap terminates



Analysis of BuildHeap

- Find a summation that represents the maximum number of comparisons required to rearrange an array into a heap
- Evaluate the sum
- This was part of WA10
  - The summation is $\sum_{k=0}^{H} k\,2^{H-k}$.

  and the sum is N – H + 1

Further discussion of Hashing

Collision Resolution

Review of Hash Table ideas

- How is the data stored?
- How are the hash code and array capacity used?
- What is a collision?
- How do we attempt to avoid collisions?
- Why can we seldom guarantee that there will be no collisions?

Collision Resolution: Linear Probing

- When an item hashes to a table location occupied by a non-equal item, simply use the next available space.
- Try H+1, H+2, H+3, …
  - With wraparound at the end of the array
- Problem: Clustering (picture on next slide)
- http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html
- We'll let it keep running while we look at analysis.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 20.4
Linear probing hash table after each insertion

Analysis of linear probing

- Dependent on the **load factor**, λ, which is the ratio of the number of items in the table to the size of the table. Thus $0 \le \lambda \le 1$.
- For a given λ, what is the expected number of probes before an empty location is found?
- For simplicity, assume that all locations are equally likely to be occupied, and equally likely to be the next one we look at. Then the probability that a given cell is empty is $1 - \lambda$, and thus the expected number of probes before finding an empty cell is (write it as a summation).

```
> simplify(sum(i*(1-lambda)*lambda^(i-1), i=1..infinity));
```

$$-\frac{1}{\lambda - 1}$$

- For example, if λ is 0.75, the expected value is 4.

Analysis of linear probing (continued)

- The "equally likely" probability is not realistic, because of **clustering**
- Large blocks of consecutive occupied cells are formed. Any attempt to place a new item in any of those cells results in extending the cluster by at least one item
- Thus items collide not only because of identical hash values, but also because of hash values that happen to put them into the cluster
- Average number of probes when λ is large:
  - $0.5\ [\ 1 + 1/(1-\lambda)^2\ ]$.
    - For a proof, see Knuth, *The Art of Computer Programming*, Vol 3: Searching Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998.
  - What are the values for $\lambda = 0, 0.5, 0.75, 0.9$?
  - **When λ approaches 1, this gets bad!**
  - But if λ is close to zero, then the average is near 1.0

So why consider linear probing?

- Easy to implement
- Simple code has fast run time per probe
- Works well when load is low
  - It could be more efficient just to rehash using a bigger table once it starts to fill.
  - Typically done in practice: rehash to an array that is double in size once the load goes over 0.5
- What about other fast, easy-to-implement strategies?

Quadratic probing

- With linear probing, if there is a collision at H, we try H, H+1, H+2, H+3,... until we find an empty spot.
  - Causes (primary) clustering
- With quadratic probing, we try H, $H+1^2$. $H+2^2$, $H+3^2$,...
  - Eliminates primary clustering, but can cause secondary clustering.

Hints for quadratic probing

- **Choose a prime number for the array size**
  - Guaranteed insertion and no cell is probed twice, provided That the table is no more than half full.
  - Suppose the array size is P, a prime number greater than 3
  - Show by contradiction that if i and j are $\leq \lfloor P/2 \rfloor$, and $i \neq j$, then $H + i^2 \pmod P \not\equiv H + j^2 \pmod P$.
- **Use an algebraic trick to calculate next index**
  - Replaces mod and general multiplication with subtraction and a bit shift
  - Difference between successive probes:
    - $H + (i+1)^2 = H + i^2 + (2i+1)$ [can use bit-shift for multiplication].
    - nextProbe = nextProbe + (2i+1);
      if (nextProbe >= P) nextProbe -= P;