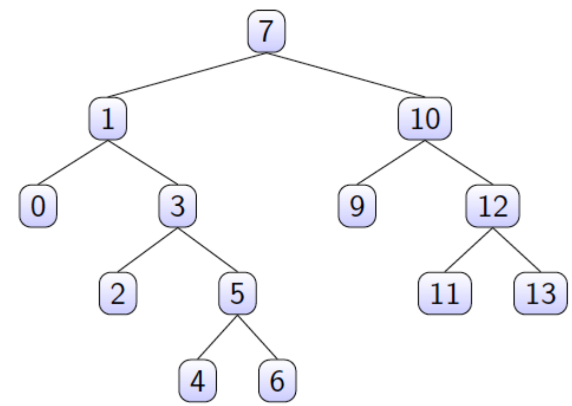
Priority Queue and Heap

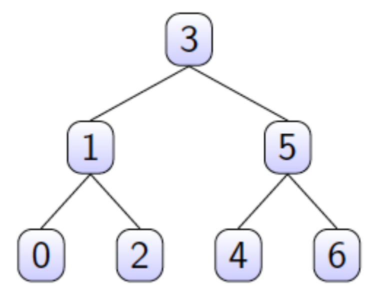
Ye Yang
Stevens Institute of Technology

Review: Binary Tree Properties

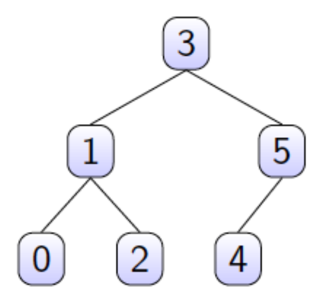
- **Full** Binary Tree
  - A binary tree is **full** if each node has 2 or 0 children



- **Perfect** Binary Tree
  - A binary tree is **perfect** if it is **full,** and
  - Each leaf is at the same depth
    - The number of nodes at depth d is **2^d**
    - Total node in a perfect tree: $\sum_{i=0}^{d} 2^i = 2^{d+1} - 1$



- **Complete** Binary Tree
  - A binary tree is **complete** if it is a **perfect** binary tree through level *$d - 1$*, with some extra leaf nodes at level *d*, all toward the left
  - A **complete** tree of height, *h*, has between *2^h* and *$2^{h+1}-1$* nodes.



General Idea

- ***Example***: Suppose a sequence of jobs sent to a printer
  - Default strategy: FIFO, placing jobs on a **queue**, {10, 8, 12, 5, 2, 3}
    - Avg. wait time = (10+18+30+5+2+3)/6 = 28.3 min
    - This might not always be the best thing to do.
  - A better strategy:
    - To make the **shortest** job goes first, even if it is the first job submitted
    - Avg. wait time = (2+5+10+18+28+40)/6 = **17** min

	findMin	**deleteMin**	**insert**
Linked List	O(N)	O(N)	O(1)
Sorted List	O(1)	O(1)	O(N)
Sorted Array	O(1)	O(N)	O(N)
Binary Search Tree	O(logN) - best case; O(N) – worst case	O(logN) - best case; O(N) – worst case	O(logN) - best case; O(N) – worst case

- This particular application requires a special kind of queue, known as a ***priority queue***.

Model

- A ***priority queue*** is a data structure that allows at least the following two operations:
- (1) *Insert*: is the equivalent of *Enqueue*
- (2) *DeleteMin*: finds, returns, and removes the minimum element in the priority queue.

Simple Implementations

- There are several obvious ways to implement a priority queue. We could use a simple linked list, performing insertions at the front and traversing the list to delete the minimum.

- Alternatively, we could insist that the list be kept always sorted.

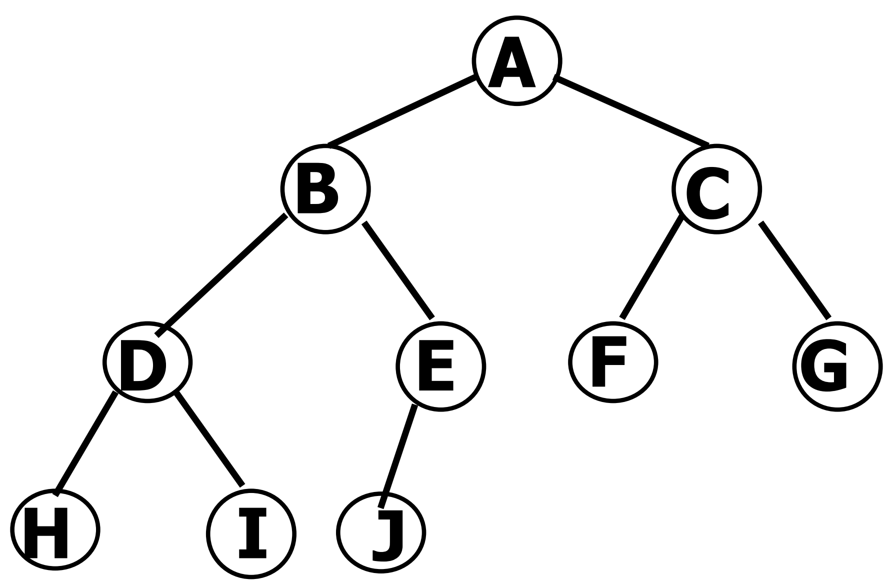
- Another way of implementing priority queues would be to use a *binary search tree*. Recall that the only element we ever delete is the minimum. Repeatedly removing a node that is in the left subtree would seem to hurt the balance of the tree by making the right subtree heavy.

Binary Heap

- The implementation we will use is known as a ***binary heap***.
- Like binary search trees, binary heaps have two properties, namely, a ***structure property*** and a ***heap order property***.

Structure Property

- **Structure property**: A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree.



Structure Property

- An important observation is that because a complete binary tree is so regular, it can be represented in an array and no pointers are necessary.

- For any element in array position i, the left child is in position $2i$, the right child is in the cell after that left child ($2i+1$), and the parent is in position $\lfloor i/2 \rfloor$.

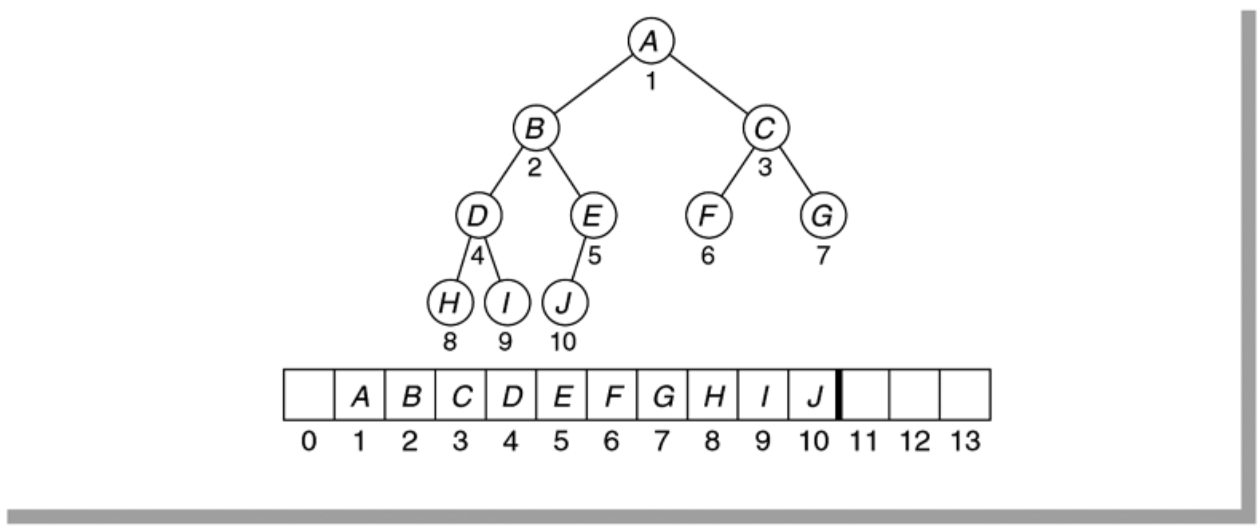


figure 21.1

A complete binary tree and its array representation

Structure Property

- Thus, not only are pointers not required, but the operations required to traverse the tree are extremely simple and likely to be very fast on most computers.
- The only problem with this implementation is that an estimate of the maximum heap size is required in advance.
- A heap data structure will then consist of an array (of whatever type the key is) and an integer representing the maximum and current heap sizes.

Heap Order Property

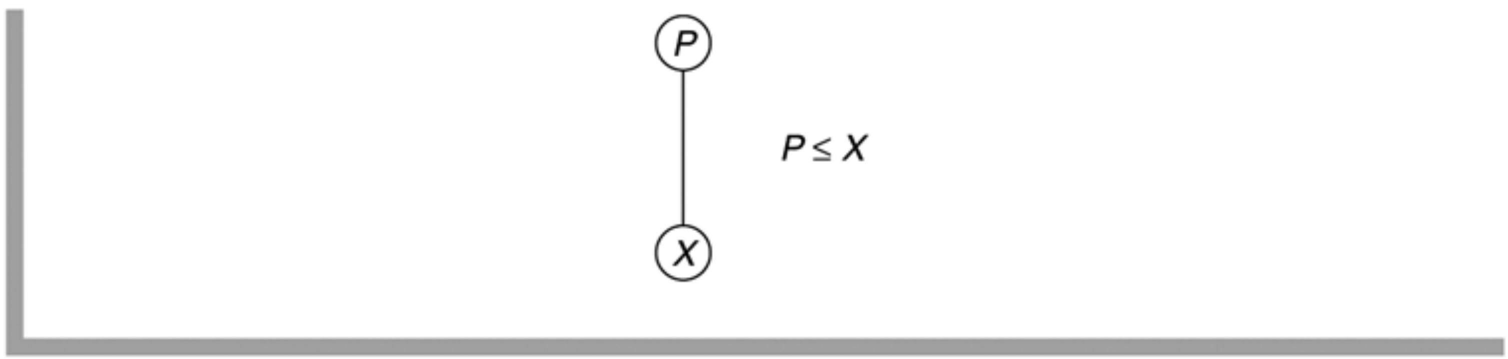
- The property that allows operations to be performed quickly is the ***heap order property***.
- Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root.
- If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

Heap Order Property

- Applying this logic, we arrive at the ***heap order property***: In a heap, for every node *X*,
  - the key in the parent of *X* is smaller than (or equal to) the key in *X*, with the exception of the root (which is has no parent).
  - A.k.a. Min-Heap

figure 21.2

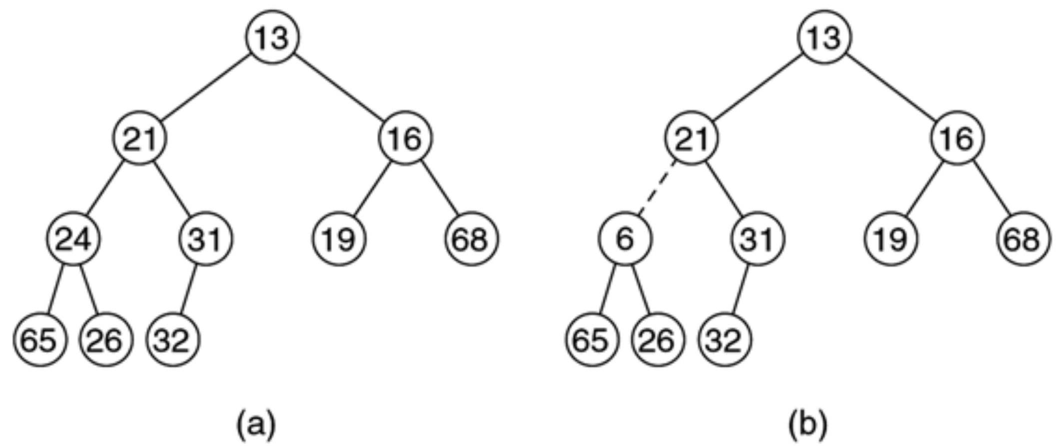
Heap-order property



- Analogously, we can declare a *Max*-Heap, which enables us to efficiently find and remove the maximum element, by changing the heap order property.

figure 21.3

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



Basic Heap Operations

- It is easy (both conceptually and practically) to perform the two required operations, namely ***Insert*** and ***DeleteMin***.
- All the work involves ensuring that the **heap order property** is maintained.

Basic Heap Operation: Insert

- ***Insert***: To insert an element *X* into the heap, we create a hole in the next available location, since otherwise the tree will not be complete.

- If *X* can be placed in the hole without violating heap order, then we do so and are done.

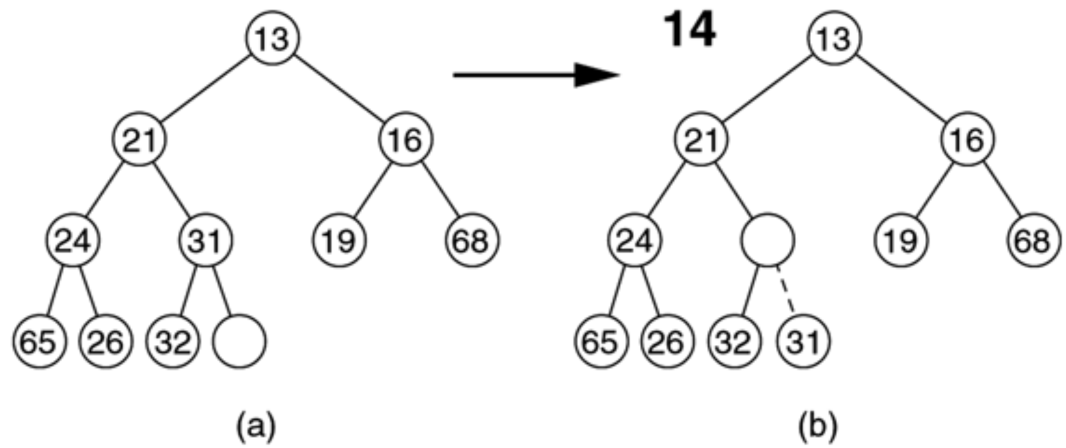
- Otherwise we slide the element that is in the hole's parent node into the hole, thus **bubbling the hole up** towards the root.

- We continue this process until *X* can be placed in the hole.

Basic Heap Operation: Insert

figure 21.7

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



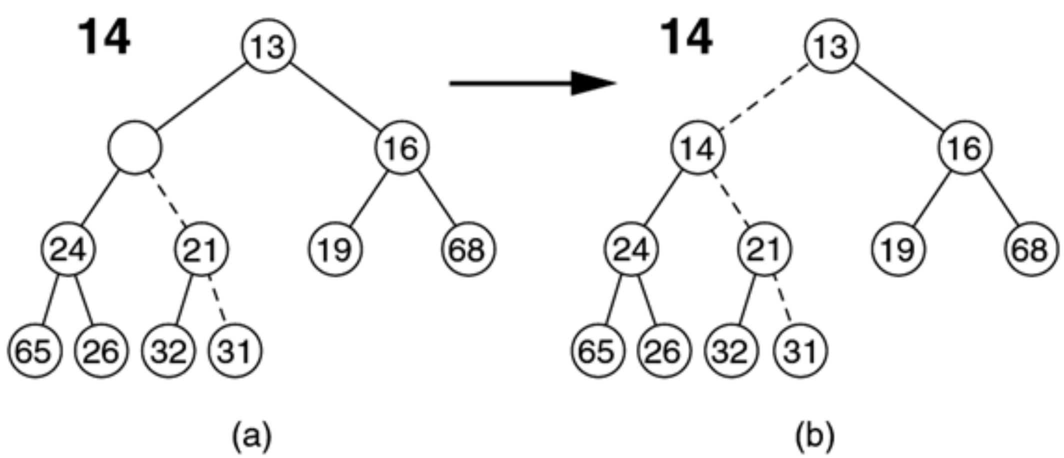


figure 21.8

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

Basic Heap Operation: Insert

```
/**
 * Adds an item to this PriorityQueue.
 * @param x any object.
 * @return true.
 */
public boolean add( AnyType x )
{
    if( currentSize + 1 == array.length )
        doubleArray( );

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

    for( ; compare( x, array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;

    return true;
}
```

figure 21.9

The `add` method

Basic Heap Operation: DeleteMin

- ***DeleteMin***: DeleteMins are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it.

- When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element *X* in the heap must move some where in the heap.

- If *X* can be placed in the hole, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus **bubbling the hole down** one level.

- We repeat this step until *X* can be placed in the hole. Thus, our action is to place *X* in its correct spot along a path from the root containing minimum children.

Basic Heap Operation: DeleteMin

figure 21.10

Creation of the hole at the root

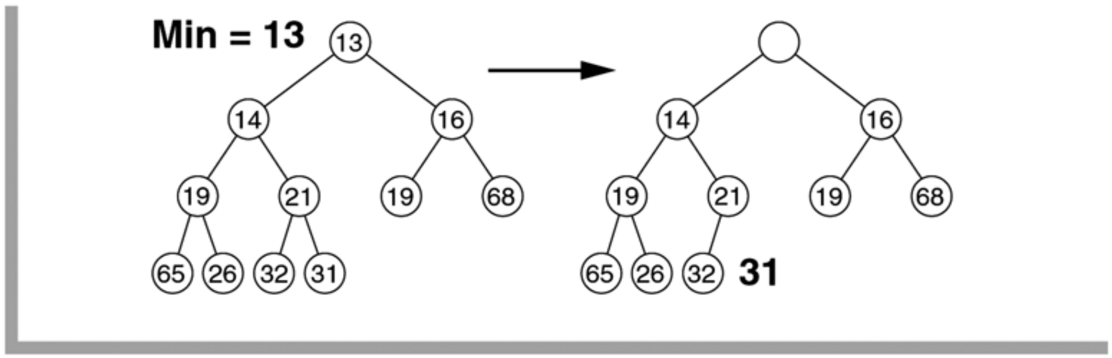
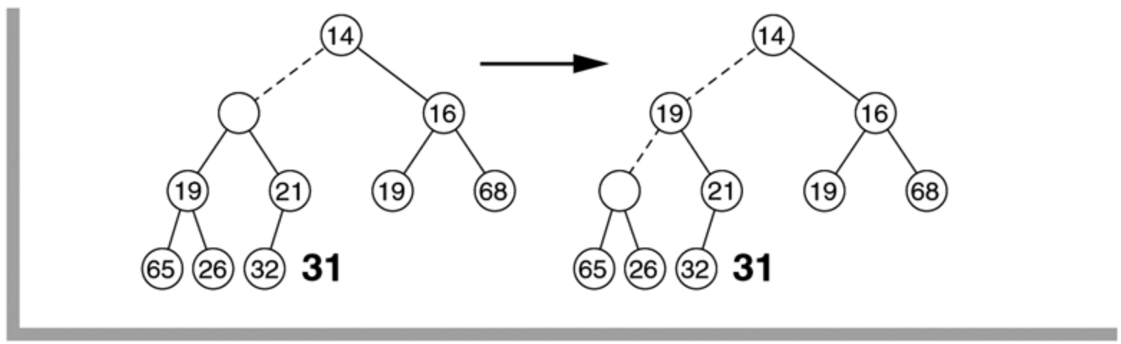


figure 21.11

The next two steps in the `deleteMin` operation



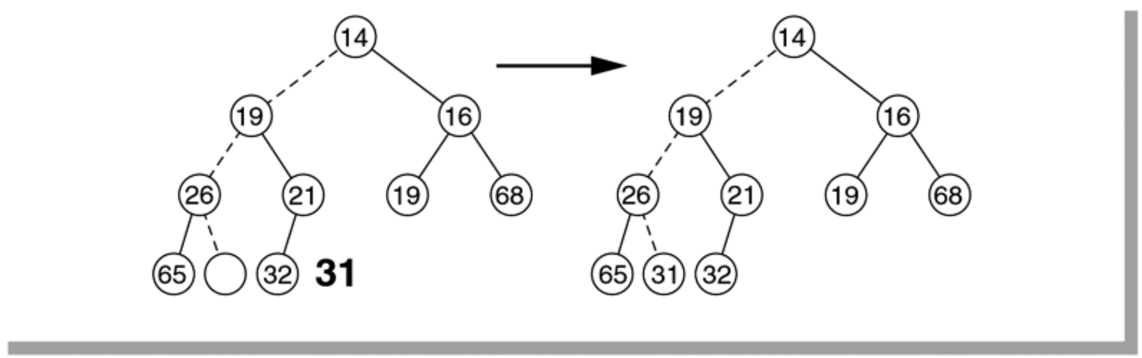


figure 21.12

The last two steps in the `deleteMin` operation

Basic Heap Operation: DeleteMin

```java
/**
 * 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

Basic Heap Operation: DeleteMin

```
/**
 * Internal method to percolate down in the heap.
 * @param hole the index at which the percolate begins.
 */
private void percolateDown( int hole )
{
    int child;
    AnyType tmp = array[ hole ];

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

figure 21.14

The `percolateDown` method used for `remove` and `buildHeap`

Exercise: Build Heap

- Draw the binary min heap that results from inserting: 10, 5, 12, 3, 2, 1, 8, 7, 9, 4 in that order into an initially empty binary min heap. Show the results after each insertion.

Linear Time BuildHeap

```
/**
 * 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

Heap PercolateDown - 1

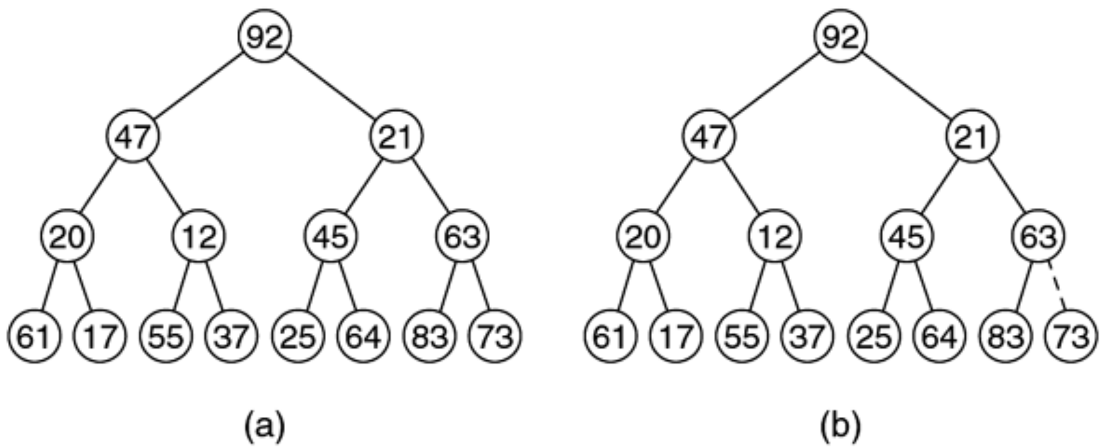


figure 21.17

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

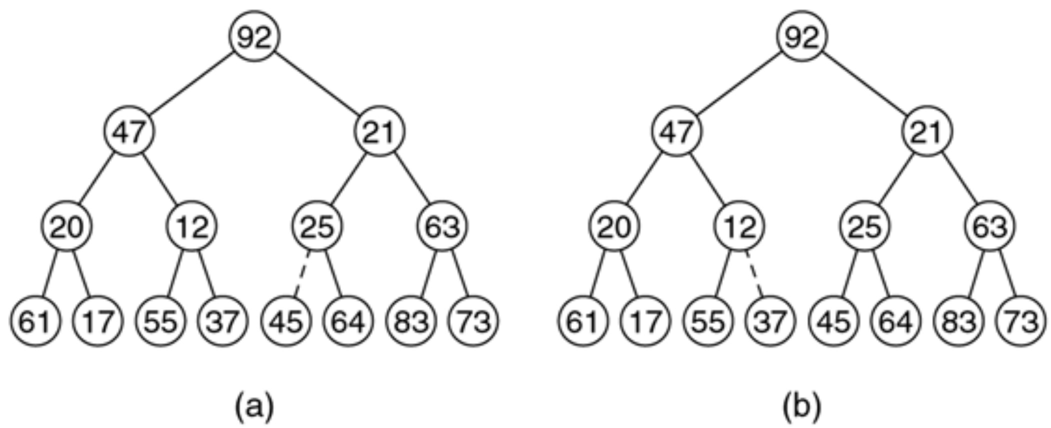


figure 21.18

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

Heap PercolateDown - 2

figure 21.19

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

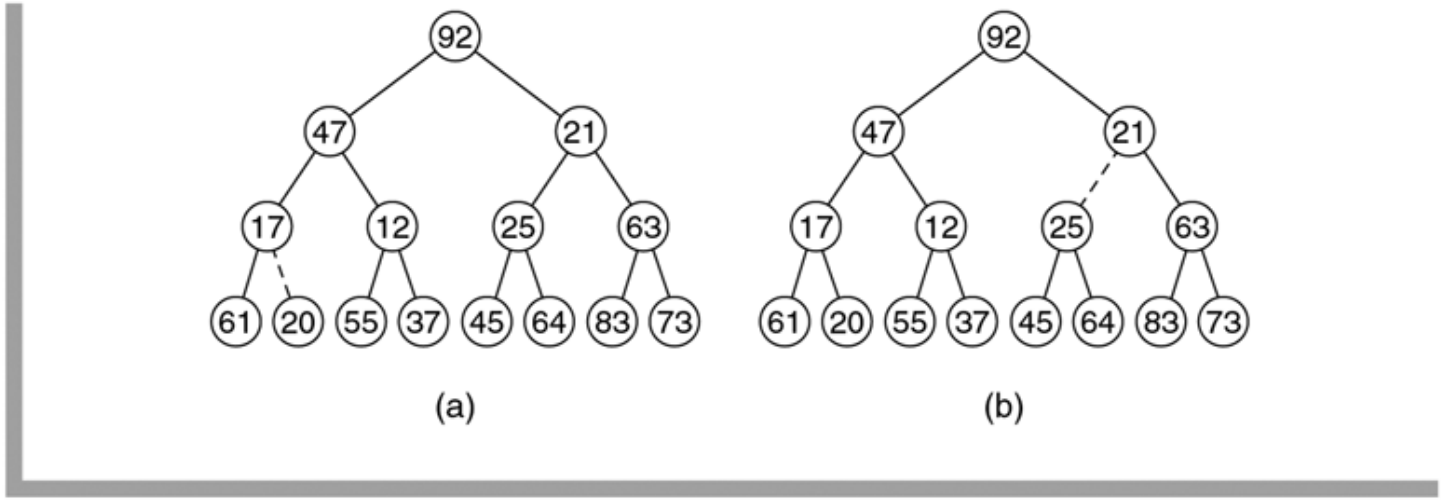
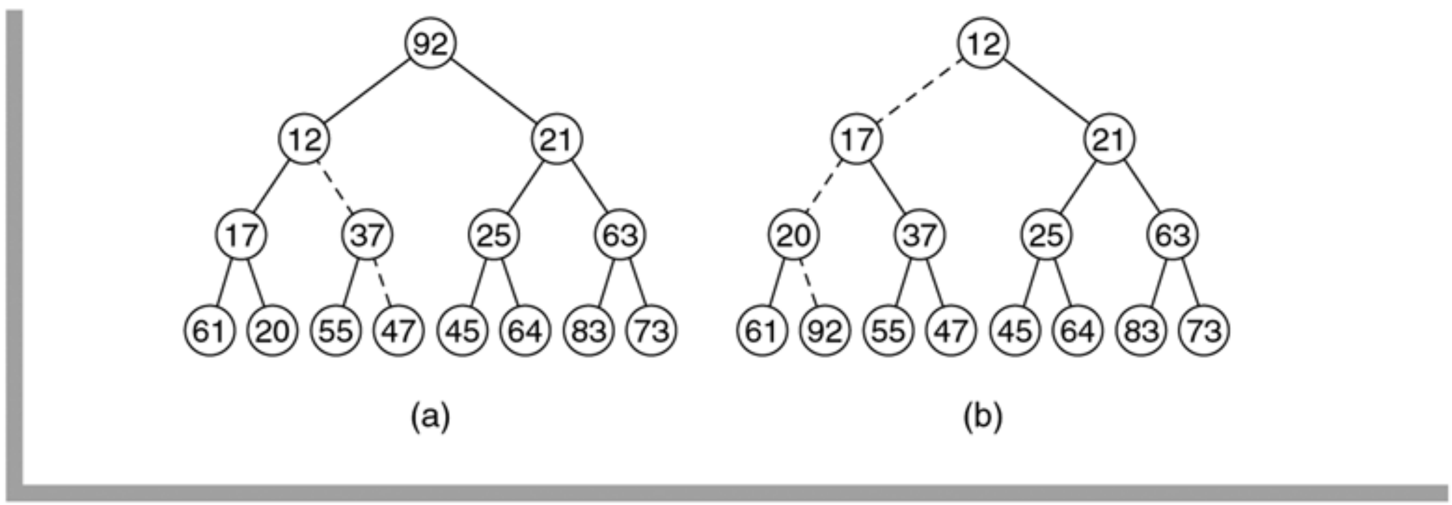


figure 21.20

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

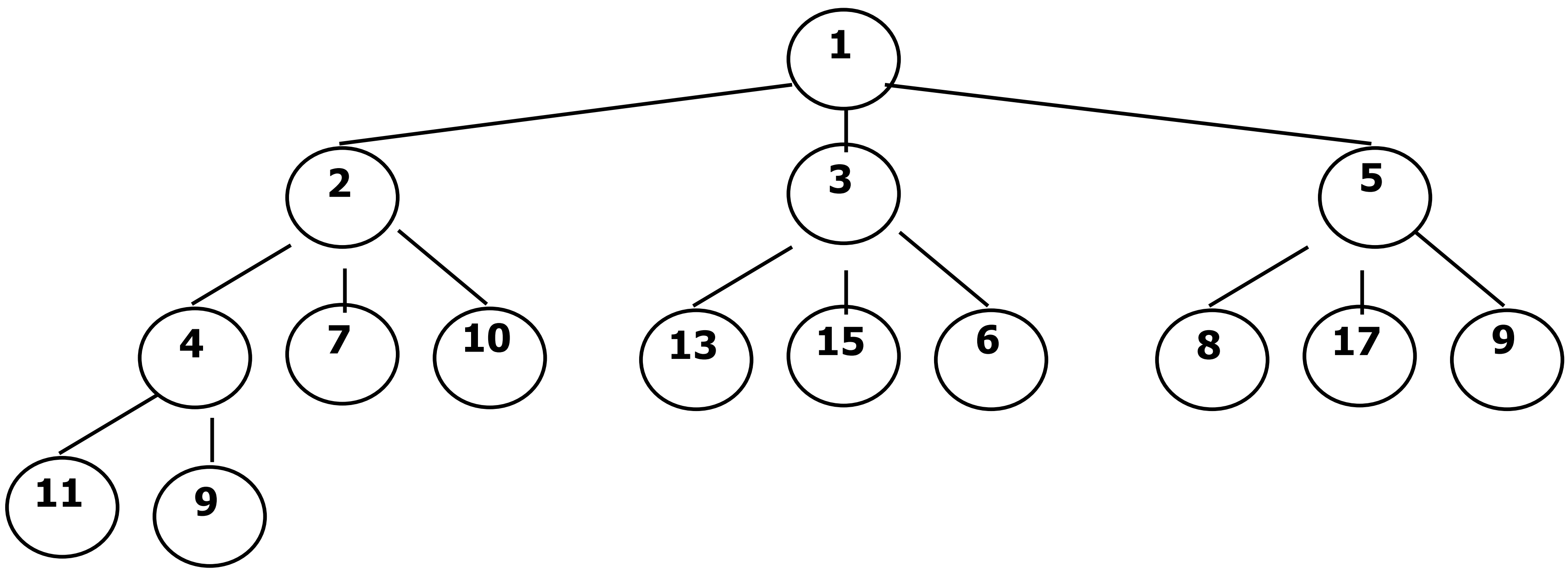


d-Heap

- Binary heaps are so simple that they are almost always used when priority queues are needed.

- A simple generalization is a *d*-heap, which is exactly like a binary heap except that all nodes have *d* children (thus, a binary heap is a 2-heap).

d-Heap



An example of 3-heap