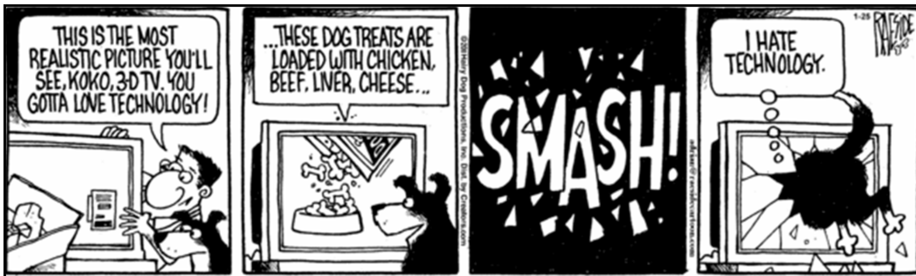


CSSE 230 Day 18

Priority Queues
Heaps
Heapsort

Today

- Student questions
  - EditorTrees
  - WA 6
  - File Compression
  - Graphs
  - Hashing
  - Anything else
- Agenda
  - Priority Queues
  - Heaps
  - Heapsort

Written Assignments 7 and 8 have been updated for this term.

Each of them is smaller than many of the written assignments have been.

No programming problems in either assignment.

Priority Queue

- Basic operations
  Implementation options

Priority Queue operations

- Each element in the PQ has an associated **priority**, which is a value from a comparable type (in our examples, an integer).
- Operations (may have other names):
  - findMin()
  - insert(item, priority)
  - deleteMin( )

Priority queue implementation

- How could we implement it using data structures that we already know about?
  - Array?
  - Queue?
  - List?
  - BinarySearchTree?
- One efficient approach uses a binary heap
  - A somewhat-sorted complete binary tree
- Questions we'll ask:
  - How can we efficiently represent a complete binary tree?
  - Can we add and remove items efficiently without destroying the "heapness" of the structure?

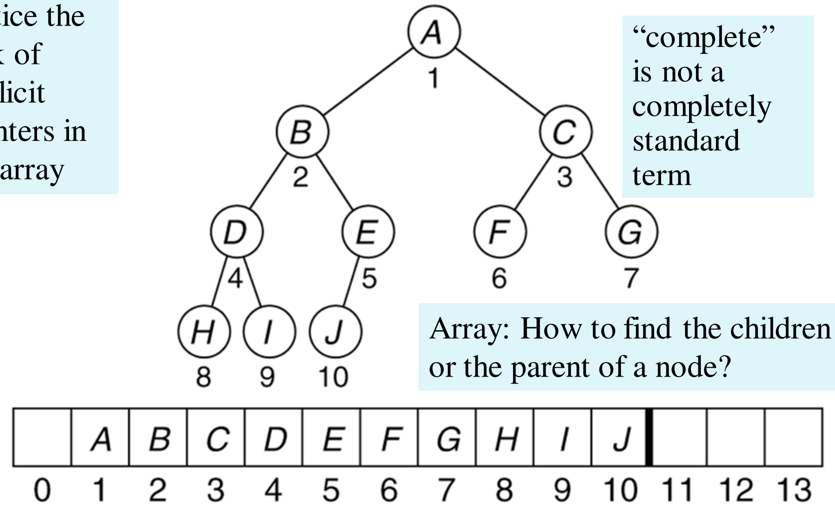
Binary Heap

- Storage (an array)
  Algorithms for insertion and deleteMin

1

Figure 21.1
A complete binary tree and its array representation

Notice the lack of explicit pointers in the array



"complete" is not a completely standard term

Array: How to find the children or the parent of a node?

One "wasted" array position (0)

2-3

The (min) heap-order property:
every node's value is ≤ its childrens' values

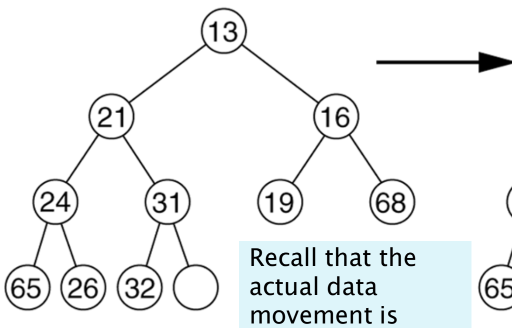
A **Binary** (min) **Heap** is a complete Binary Tree (using the array implementation, as on the previous slide) that has the heap-order property everywhere.

$$P \le X$$

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

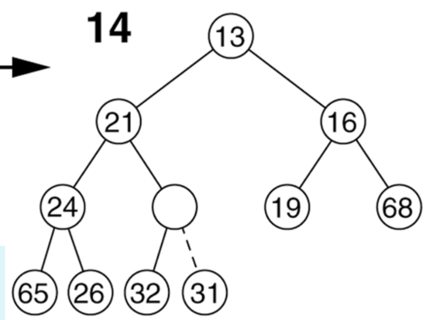
Figure 21.7
Attempt to insert 14, creating the hole and bubbling the hole up

Insertion algorithm



(a)

Recall that the actual data movement is done by array manipulation

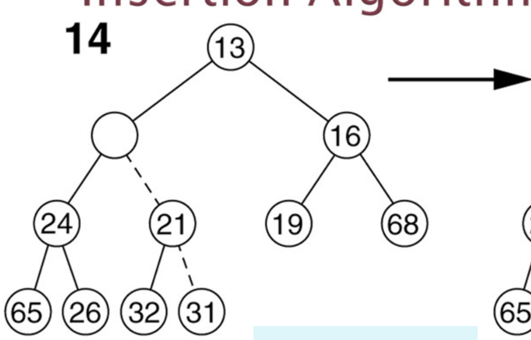


(b)

Create a "hole" where 14 can be inserted.
Percolate up!

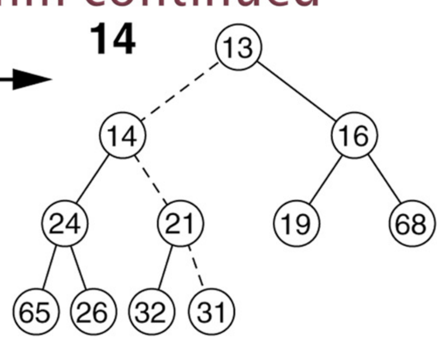
Figure 21.8
The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

Insertion Algorithm continued



(a)

Analysis of insertion …



(b)

Your turn: Insert into an initially empty heap:
6 4 8 1 5 3 2 7

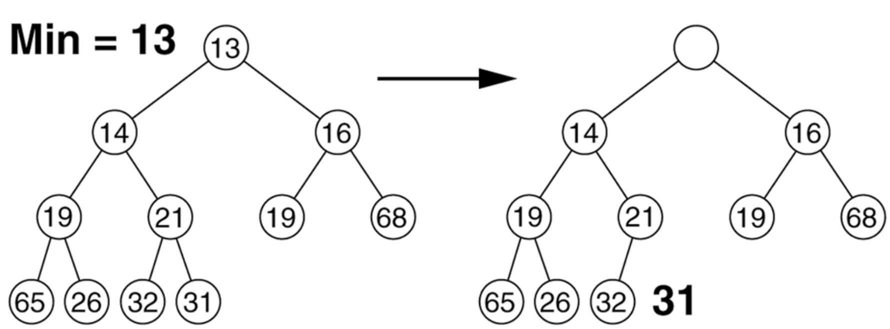
4-5

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

DeleteMin Slide 2

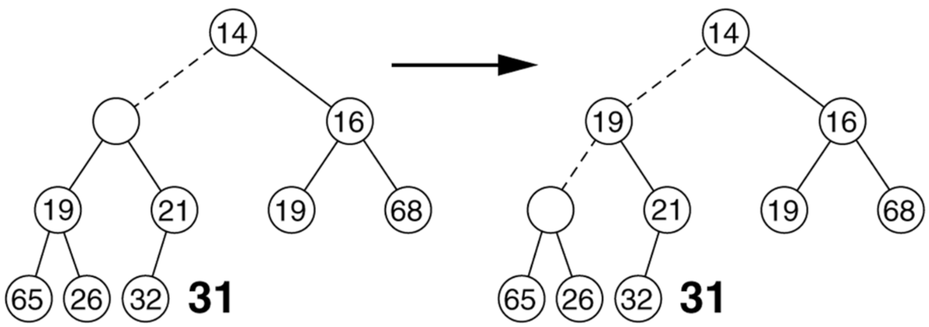
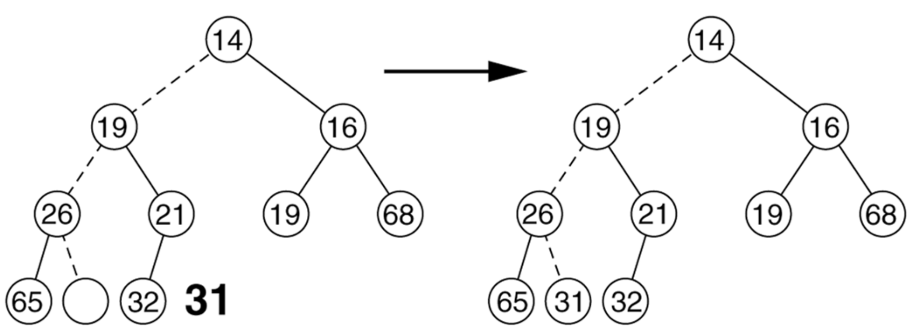


Figure 21.12
The last two steps in the deleteMin operation

DeleteMin Slide 3



6-7

```
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

8

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 in the balanced BST implementation, but ..
  - Heap operations are much simpler,
  - A heap doesn't require additional space for pointers or balance codes.

Heapsort

- Use a binary heap to sort an array.

Using a Heap for sorting

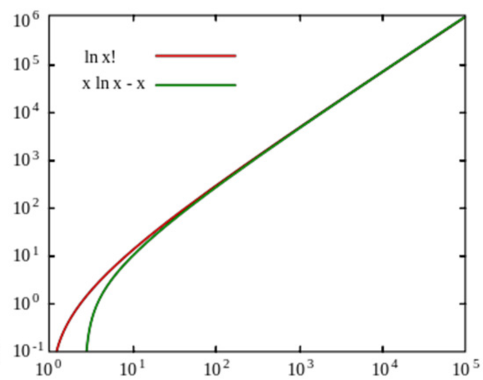
- Start with empty heap
- Insert each array element into heap
- Repeatedly do deleteMin, copying elements back into array.
- http://nova.umuc.edu/~jarc/idsv/lesson3.html
  - Can be run in demo mode or practice mode.
- We can save space by doing the whole sort in place, using a "maxHeap" (i.e. a heap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

Analysis of simple heapsort

10

- Add the elements to the heap
  - Repeatedly call insert
- Remove the elements and place into the array
  - Repeatedly call DeleteMin
- Use Stirling's approximation: $\ln n! = n \ln n - n + O(\ln(n))$

http://en.wikipedia.org/wiki/Stirling%27s_approximation



- Can we do better for the insertion part?
  - Yes, use BuildHeap (next)

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 );
}
```

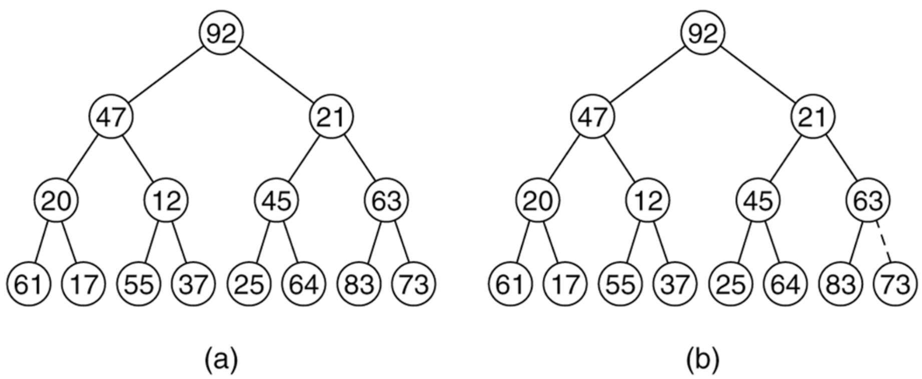


Figure 21.18
(a) After percolateDown(6);
(b) after percolateDown(5)

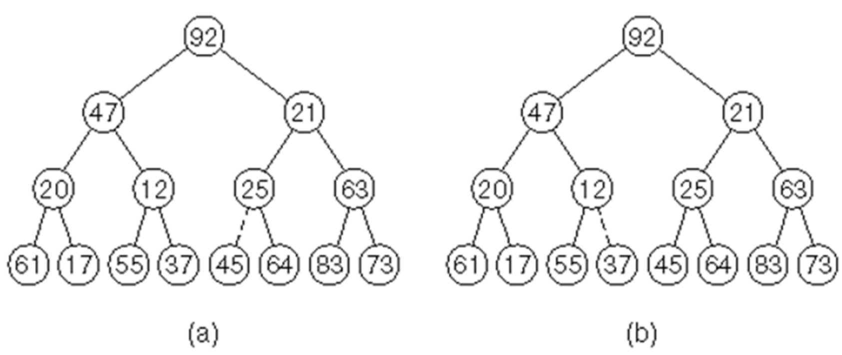


Figure 21.19
(a) After percolateDown(4);
(b) after percolateDown(3)

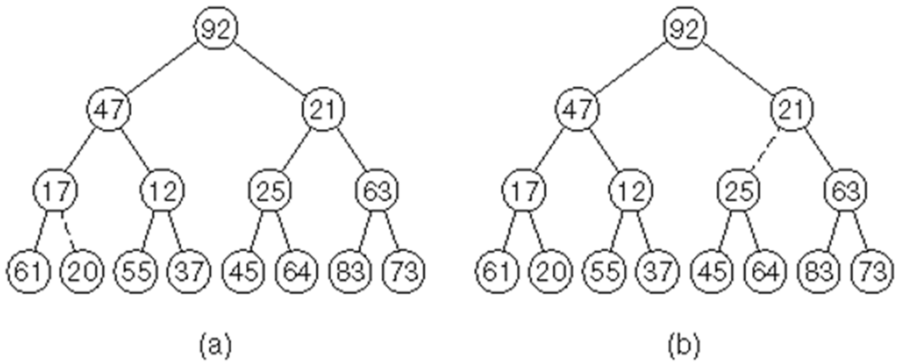
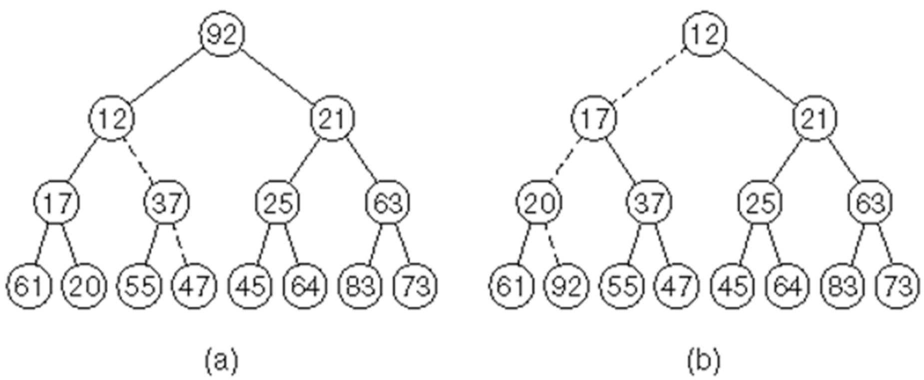


Figure 21.20
(a)After percolateDown(2);
(b) after percolateDown(1) and buildHeap terminates



Analysis of BuildHeap

- Find a summation that represents the maximum number of comparisons required to rearrange an array into a heap
- Can you find a summation and its value?

Analysis of BuildHeap

- Find a summation that represents the maximum number of comparisons required to rearrange an array of $N=2^{H+1}-1$ elements into a heap
  - The summation is $\sum_{k=0}^{H} k\,2^{H-k}$.

    and the sum is N – H – 1

- Good practice: prove this formula by induction
  - Can do it strictly by the numbers
  - Simpler: Do it based on the trees.

9-10

Analysis of better heapsort

- Add the elements to the heap
  - Use buildHeap
- Remove the elements and place into the array
  - Repeatedly call deleteMin