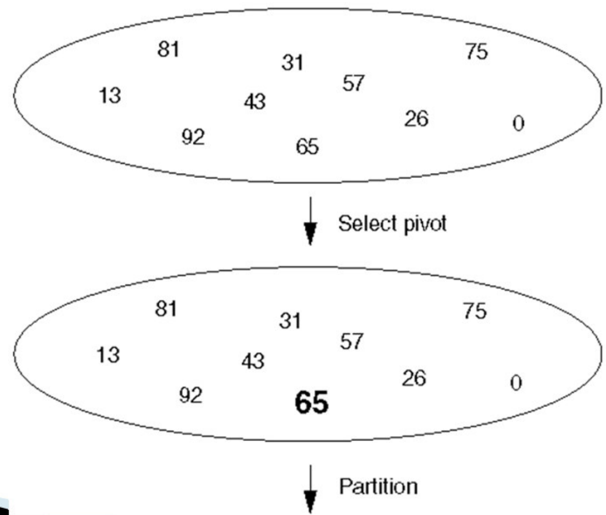
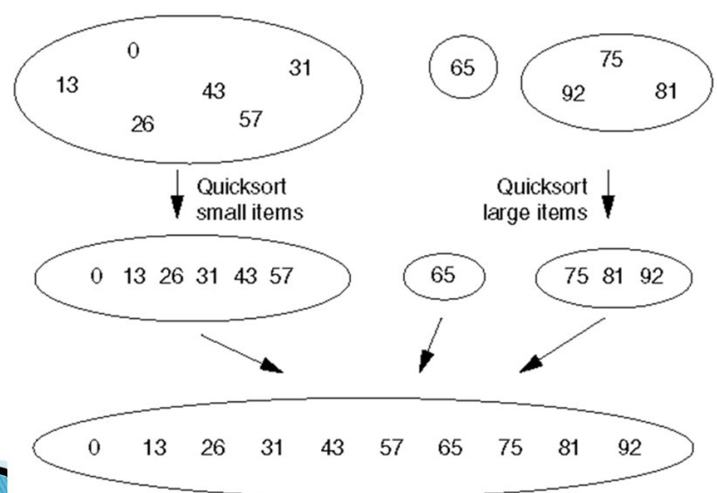
Quicksort Review:
Prepare for partition



Quicksort Review:
Partition and recursive calls



QuickSort

- Let's write some of the code
- Demo (view later)
  - http://pages.stern.nyu.edu/~panos/java/Quicksort/
- Running time for **partition of N elements**?
- Quicksort Running time:
  - call partition. Get two subarrays of sizes N_L and N_R (what is the relationship between N_L, N_R, and N?)
  - Then Quicksort the smaller parts
  - $T(N) = N + T(N_L) + T(N_R)$
- Quicksort Best case: write and solve the recurrence
- Quicksort Worst case: write and solve the recurrence
- average: a little bit trickier
  - We have to be careful how we measure

Average time for Quicksort

- Let T(N) be the average # of comparisons of array elements needed to quicksort N elements.
- What is T(0)? T(1)?
- Otherwise T(N) is the sum of
  - time for partition
  - average time to quicksort left part: $T(N_L)$
  - average time to quicksort right part: $T(N_R)$
- $T(N) = N + T(N_L) + T(N_R)$

Scenario from the Weiss book – how not to count it:

- What if we picked as the partitioning element the smallest element half of the time and the largest half of the time?
- Then on the average, $N_L = N/2$ and $N_R = N/2$,
  - but that doesn't give a true picture of this worst-case scenario.
  - In every case, either $N_L = N-1$ or $N_R = N-1$
- Instead we need to figure it out for each case, and average all of the cases

Assumption

- We always need to make some kind of "distribution" assumptions when we figure out Average case
- When we execute

  ```
  k = partition(pivot, i, j),
  ```

  all positions i..j are equally likely places for the pivot to end up
- Thus N_L is equally likely to have each of the values 0, 1, 2, … N–1
- $N_L+N_R = N-1$; thus N_R is also equally likely to have each of the values 0, 1, 2, … N–1
- Thus $T(N_L)= T(N_R) =$

Continue the calculation

- T(N) =
- Multiply both sides by N
- Rewrite, substituting N-1 for N
- Subtract the equations and forget the insignificant (in terms of big-oh) -1:
  - NT(N) = (N+1)T(N-1) + 2N
- Now we have an equation that expresses T(N) in terms of a similar formula involving T(N-1), so we can telescope

Continue continuing the calculation

- NT(N) = (N+1)T(N-1) + 2N
- Divide both sides by N(N+1)
- Write formulas for T(N), T(N-1),T(N-2) …T(2).
- Add the terms and rearrange.
- Notice the familiar series
- Multiply both sides by N+1.

Improvements to QuickSort

- Avoid the worst case
  - Select pivot from the middle
  - Randomly select pivot
  - Median of 3 pivot selection.
  - Median of k pivot selection
- "Switch over" to a simpler sorting method (insertion) when the subarray size gets small.

Weiss Quicksort Code Part 1

```
public static <AnyType extends Comparable<? super AnyType>>
       void quicksort( AnyType [ ] a ){
    quicksort( a, 0, a.length - 1 );
}

private static final int CUTOFF = 10;

public static final <AnyType>  void
  swapReferences( AnyType [ ] a, int index1, int index2 ) {
    AnyType tmp = a[ index1 ];
    a[ index1 ] = a[ index2 ];
    a[ index2 ] = tmp;
}
```

Weiss Quicksort Code Part 2

```
private static <AnyType extends Comparable<? super AnyType>> void
quicksort( AnyType [ ] a, int low, int high ) {
        if( low + CUTOFF > high )
            insertionSort( a, low, high );
        else {
                // Sort low, middle, high
            int middle = ( low + high ) / 2;
            if( a[ middle ].compareTo( a[ low ] ) < 0 )
                swapReferences( a, low, middle );
            if( a[ high ].compareTo( a[ low ] ) < 0 )
                swapReferences( a, low, high );
            if( a[ high ].compareTo( a[ middle ] ) < 0 )
                swapReferences( a, middle, high );

            // Place pivot at position high - 1
            swapReferences( a, middle, high - 1 );
            AnyType pivot = a[ high - 1 ];

                // Begin partitioning
```

Weiss Quicksort Code Part 3

```
// Begin partitioning
            int i, j;
            for( i = low, j = high - 1; ; ) {
                while( a[ ++i ].compareTo( pivot ) < 0 )
                    ;
                while( pivot.compareTo( a[ --j ] ) < 0 )
                    ;
                if( i >= j )
                    break;
                swapReferences( a, i, j );
            }

                // Restore pivot
            swapReferences( a, i, high - 1 );

            quicksort( a, low, i - 1 );     // Sort small elements
            quicksort( a, i + 1, high );    // Sort large elements
        }
    }
```

Other Sorting Demos

- http://maven.smith.edu/~thiebaut/java/sort/demo.html
- http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

Average BST node depth

- This gives the average time for finding an element in the BST.
- Do average internal path length (IPL).
- Average depth is (1/N)(Average IPL).

Average BST IPL

- Let D(N) be the average IPL of a BST with N nodes.
- If i nodes in left subtree, then N-i-1 in right subtree.
- If i nodes in left subtree, then average contribution of those nodes to IPL of whole tree is D(i) + i. Similarly right subtree contributes D(N-i-1)+ N-i-1.
- D(N) = (2/N) sum(D(i)) + N-1
- Same recurrence as average case of Quicksort, so same O(N log N) solution.
- Conclusion: Average search time in random BST is O(log N).

Priority Queue

Basic operations
Implementation options
Binary Heaps

Priority Queue operations

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

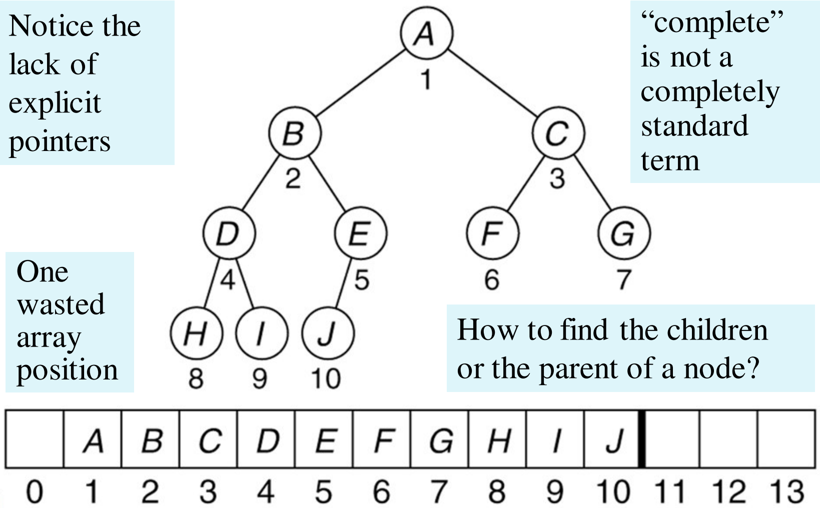
Priority queue implementation

- 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?

Figure 21.1
A complete binary tree and its array representation

Notice the lack of explicit pointers

“complete” is not a completely standard term



One wasted array position

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

Heap-order property

$P \leq X$

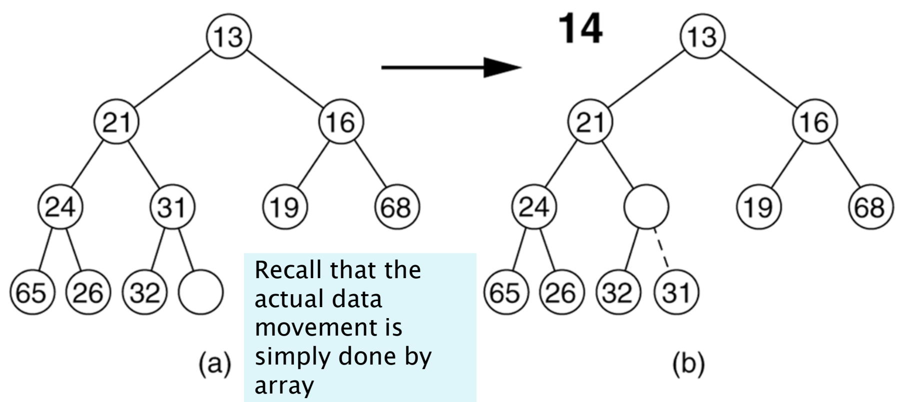
Figure 21.2
Heap-order property

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?
- 2^{nd} smallest?
- 3^{rd} smallest?

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

Insertion algorithm

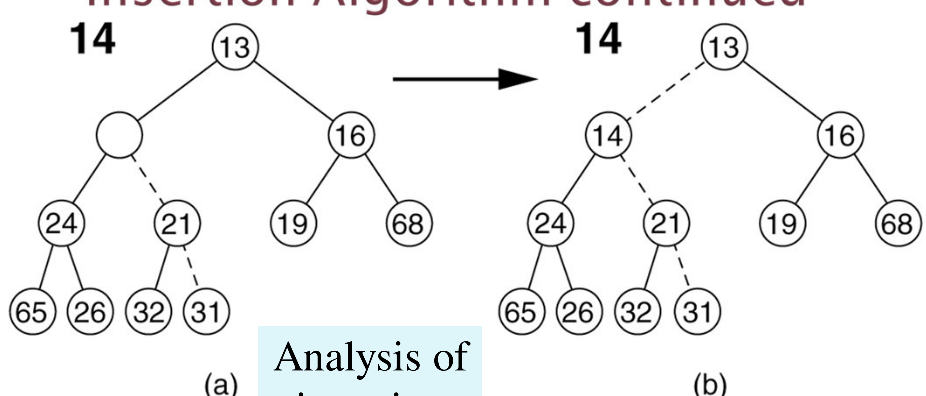


Recall that the actual data movement is simply done by array manipulation

Create a "hole" where 14 can be inserted.

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

Insertion Algorithm continued



Analysis of insertion

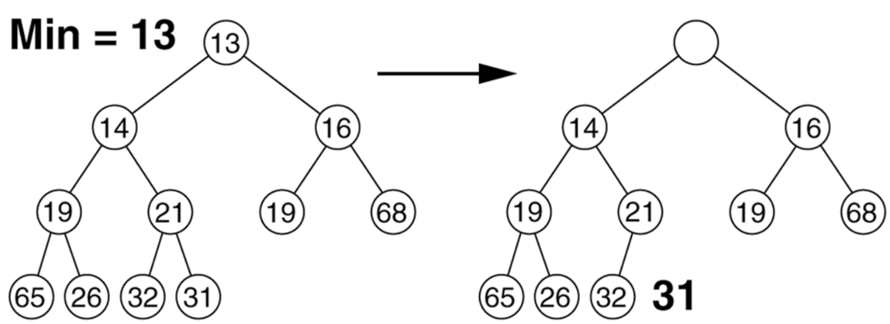
Your turn: Insert into an initially empty heap:
6 4 8 1 5 3 2 7

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

DeleteMin2

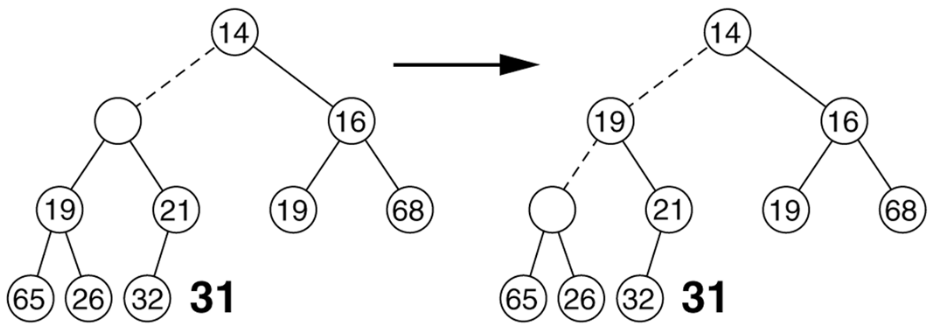


Figure 21.12
The last two steps in the deleteMin operation

DeleteMin3