Lesson 11
Priority Queues and Heap Sort
Rapid Access on the Ground of Orderliness

Wholeness of the Lesson

Queues provide rapid insertions and removal of elements, adhering to the rule: "first in, first out." In a Priority Queue, one is allowed to remove elements in any desired order (not based on when the element was inserted). Implementing this idea using the *heap* data structure results in O(log *n*) performance of the main operations.

Science of Consciousness: Pure consciousness is the field of pure orderliness and the field of all possibilities. Experience of this field shows that, on the ground of the orderliness experienced from transcending, new possibilities arise in daily life to open new directions and solve problems.

Main Idea Behind Priority Queues

1. In an ordinary queue, elements are removed from the queue in the order in which they were originally inserted.

2. Sometimes, a different order of removal would be preferable. Example: In an operating system, a time slice is given to a requesting process according to priority rather than according to the order in which processes place requests.

3. Priority Queues support the notion that highest priority objects are removed first.

Priority Queue ADT

1. *Elements of form (k,e).* In order to specify priorities, each object to be inserted into the queue is equipped with a key – its "priority". Therefore, typical elements of the queue are pairs `(k,e)` where `k` is the key (like an integer) and `e` is the element. (Duplicate keys are allowed.)

2. *Operations.* A Priority Queue has two main operations:
   `insertItem(k,e)` – inserts the pair `(k,e)`
   `removeMin()` – removes a pair `(k,e)` for which `k` is smallest and returns `e`

   Auxiliary Operations.
   `minElement()` – returns (but does not remove) an element with smallest key
   `minKey()` – returns (but does not remove) the smallest key

Implementations of Priority Queues

1. *Background data structure is an unsorted array*.
   - `removeMin` takes Θ(n) on average
   - `insertItem` takes Θ(1) if implemented correctly

2. *Background data structure is a sorted array*
   - `removeMin` takes Θ(1) if implemented properly (could arrange to have array in reverse sorted order so the min is always last element)
   - `insertItem` takes Θ(n) on average, with many copies of array elements

3. *Background data structure is a red-black tree*
   - `removeMin` takes O(log n) – remove the min, located at the end of leftmost branch (O(log n))
   - `insertItem` takes O(log n) – this is just red-black tree insertion

4. Red-black tree performance is nearly optimal, but can be improved slightly using a different approach.

Heaps

1. A heap is a binary tree that has a structural property and an ordering property.

   - *Heap Structural Property:* Structurally, a heap is a binary tree in which every level except possibly the bottom level is filled completely, and the bottom level is filled from left to right. (Such trees are sometimes called *complete*.)

   - *Min-Heap Order Property:* A key in a node n must always be greater than or equal to the key in the parent of n (if there is a parent).

     Note: The **Min-Heap** Order Property ensures min key is at the root. A variant is the *Max-Heap Order Property* ensures the max key is at the root.

2. The structural property implies that if the heap has height *h* with *n* nodes, then
$$2^h \leq n \leq 2^{h+1} - 1$$

   Taking logs establishes:

   Theorem. If T is a heap with *n* nodes and height *h*, then *h* is O(log n) (with small constant factor).

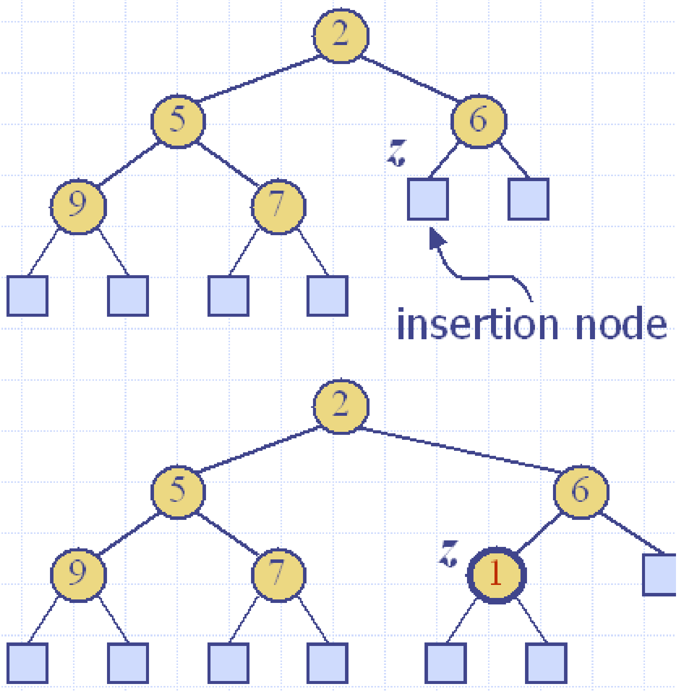
3. Consequently, when a heap is used to implement a PriorityQueue, removeMin and insertItem run in O(log n) and the constant factors are smaller than if a Red-Black tree is used.

`insertItem` Using Heaps

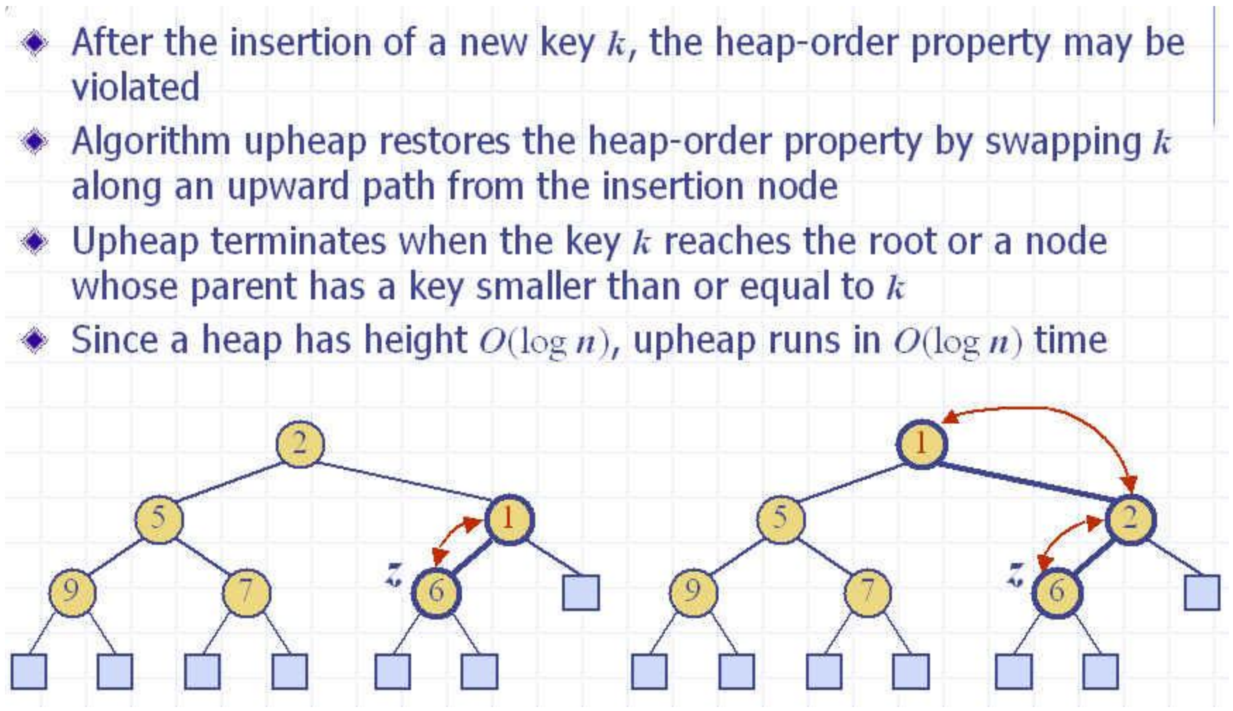
Method insertItem of the priority queue ADT corresponds to the insertion of a key *k* to the heap

The insertion algorithm consists of three steps

- Find the insertion node *z* (the new last node)
- Store *k* at *z* and expand *z* into an internal node
- Restore the heap-order property (discussed next)



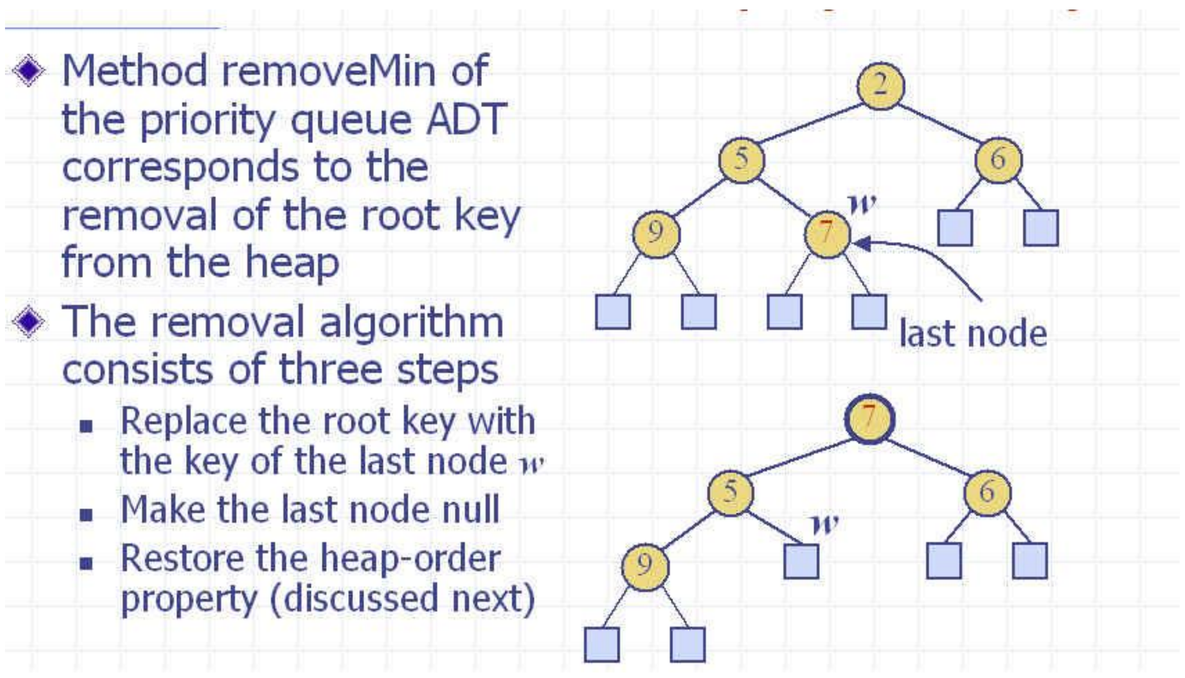
Maintaining Heap Order: *Upheap*



- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

NOTE: During upheap, a swap occurs only if the key *k* that is being moved upward is ***strictly less than*** its parent when we are maintaining a min-heap; this fact is important when heaps are used for sorting (discussed later in these slides).

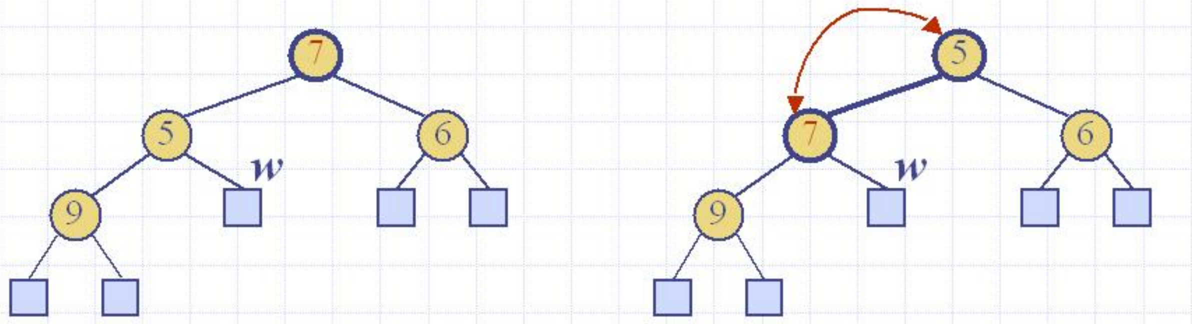
`removeMin` Using Heaps



- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Make the last node null
  - Restore the heap-order property (discussed next)

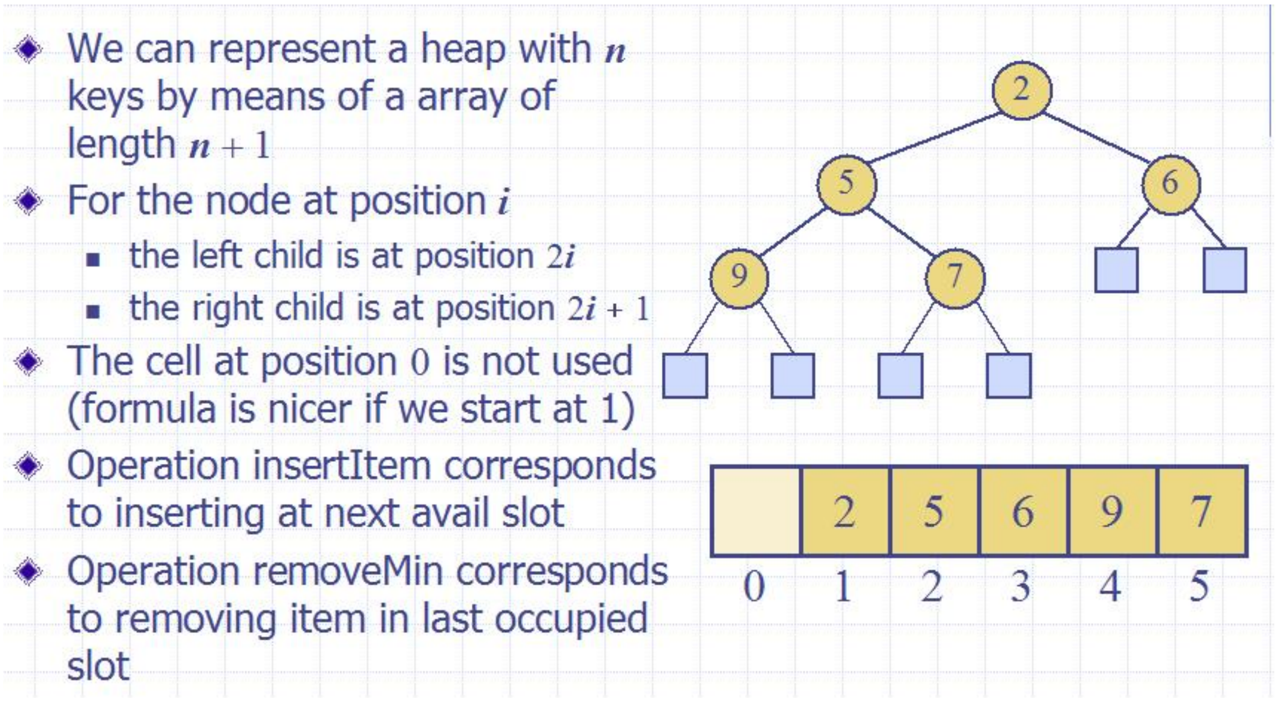
Maintaining Heap Order after Removal: *Downheap*

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



NOTE: In the downward path, whenever key *k* is **larger than a child value**, *k* is swapped with the ***smaller*** **of the two child values (when we are maintaining min-heap).**

Implementing a Heap in an Array



- We can represent a heap with n keys by means of a array of length $n + 1$
- For the node at position i
  - the left child is at position $2i$
  - the right child is at position $2i + 1$
- The cell at position 0 is not used (formula is nicer if we start at 1)
- Operation insertItem corresponds to inserting at next avail slot
- Operation removeMin corresponds to removing item in last occupied slot

NOTE: In this scheme, the parent of a key at position *j* is *j*/2 if *j* is even, (*j*-1)/2 if *j* is odd. If integer division is assumed, parent of a key at position *j* is *j*/2. **(The cell position 0 is not used.)**

Main Point

The most efficient implementation of a Priority Queue is via *heaps* which support O(log *n*) worst-case running time for both operations `removeMin` and `insertItem`, with minimal overhead. A heap is a binary tree in which every level is filled except possibly the bottom level, which is as full as possible from left to right. In addition, a binary heap satisfies the *heap order property:* For every node *X*, the key value of *X* is greater than or equal to that of its parent (if it has a parent). A Priority Queue gives a simple model of the principle of the Highest First: Putting attention on the highest value as the top priority results in fulfillment of all lower-priority values as well, in the proper time.

Using a Heap for Sorting

1. An algorithm for using a priority queue for sorting:

   Given an array of *n* items (keys) to sort:
   - Use insertItem *n* times to load the Priority Queue
   - Output the results by inserting into output array the output of removeMin, loading from left to right

2. Analysis: When a Heap is used to implement the Priority Queue

   - load the Priority Queue: O(nlog n)
   - load the output array: O(nlog n)

   Therefore *HeapSort* runs in O(nlog n), worst-case

HeapSort Optimization: *In-Place HeapSort*

Strategy:

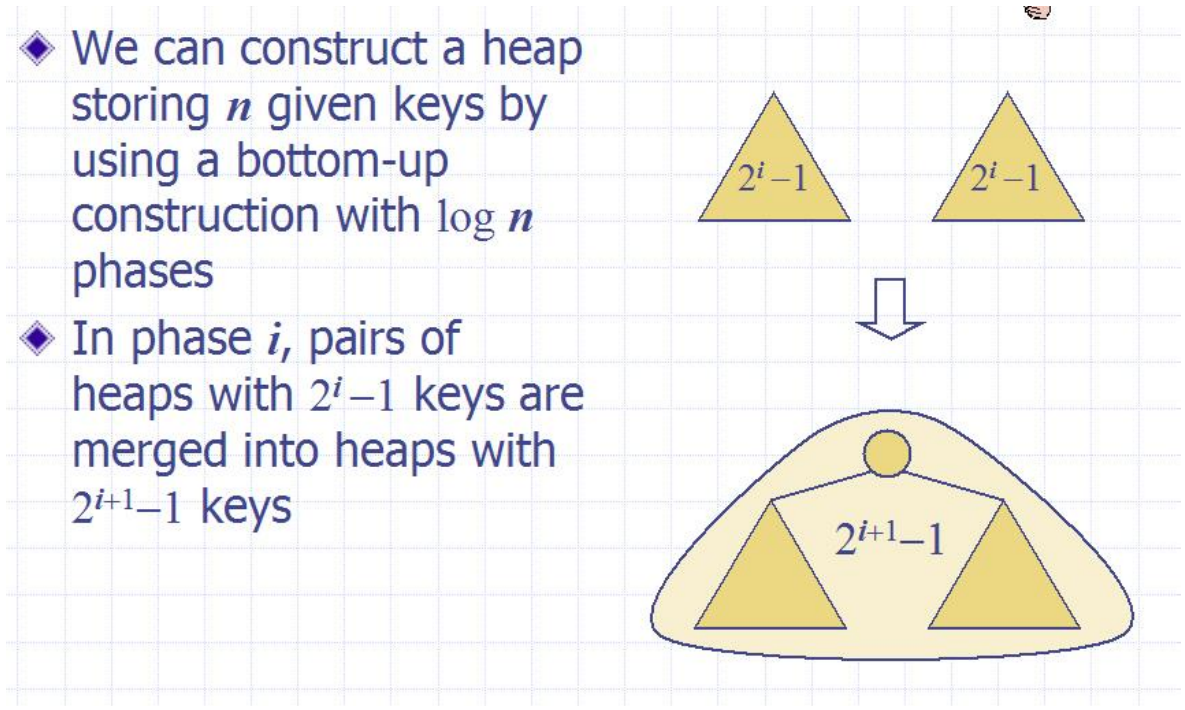
1. Use the Max-Heap Order Property, so that data in a node is always *less than or equal to* data in the parent.

2. In array implementation of Heap, we start from 0 instead of 1 – therefore, children of node at position *n* are now at positions 2*n*+1 and 2*n*+2 (so, parent of node at position j is located either in position (j-1)/2 or (j-2)/2)

3. Phase I: Given an array A, we grow an array-based heap from left to right; before stage *i*, the heap occupies positions 0..*i*-1 and the remaining “unheapified” elements are in remaining slots; at stage *i*, element at position *i* is included in heap, and array-based "upheap" is performed.

4. Phase 2: Repeatedly perform removeMax on the left part of the array that consists of the (gradually diminishing) heap. Begin by pulling off max of heap and moving value at pos n-1 into position 0, and performing "downheap"; place max into pos n-1. Then pull off next max of smaller heap, place value at pos n-2 into position 0, perform downheap, and place this next max in pos n-2. Continue.

HeapSort Optimization: *Bottom-Up Construction*



- We can construct a heap storing n given keys by using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

$2^i - 1$
$2^i - 1$
$2^{i+1} - 1$

```
Algorithm BottomUpHeap(A)
    Input: An array A storing $n = 2^h-1$ keys
    Output: A heap T storing the keys in A

    if A is empty then
        return an empty heap
    remove the first key $k$ from A
    split A into two subarrays, $A_1$ and $A_2$, of equal length
    $T_1 \leftarrow$ BottomUpHeap($A_1$)
    $T_2 \leftarrow$ BottomUpHeap($A_2$)
    create a binary tree T with root r storing k, left subtree $T_1$, right subtree $T_2$
    perform a downheap from the root r of T, if necessary
    return T
```

NOTE: The algorithm works without modification when n has the form 2^h-1; it needs to be modified slightly for other n (but the results concerning its running time still hold true).

BottomUpHeap Iteratively

- Basic idea is to build the heap from the bottom up
- Iterative strategy:
  - Make (n+1)/2 single item heap trees--(n-1)/2 items still in reserve.
  - Merge pairs of single item heaps by adding an item from the reserve. (n+1)/4 heaps now and do downheap as necessary
  - Merge pairs of three item heaps by adding item from the reserve, followed by downheap
  - Repeat log(n+1) times (= height of the tree)
- Iterative version can also be shown to run in O(n) time.

Iterative Bottom-up Heap for Any Input Size

- A BUH-number is a number of the form 2^h-1
- The BUH bounds for any positive integer n are consecutive BUH numbers $a < b$ such that $a < n \leq b$.
- Given array *A* of length *n*, compute the BUH bounds *a, b* for *n*.
- Let $m = b - n$ and $k = n - a$. (Note:
- Build lowest level of heap by using first *k* elements of *A*
- First step in building second level of heap: get next $\lceil k/2 \rceil$ elements from *A* to form size-3 heaps with bottom level values (if *k* is odd, last heap among these will be of size 2 only), then do downheaps for each to ensure order property
- Second setp in building second level of heap: get next $\lfloor m/2 \rfloor$ elements from *A* to be 1-element heaps at second level
- Next level of heap: Let *u* be num elements of 2nd level (it will be a power of 2). Take *u*/2 more from *A* to provide roots for each of the heaps so far.
- Repeat until final element of *A* is positioned at root of one large heap.

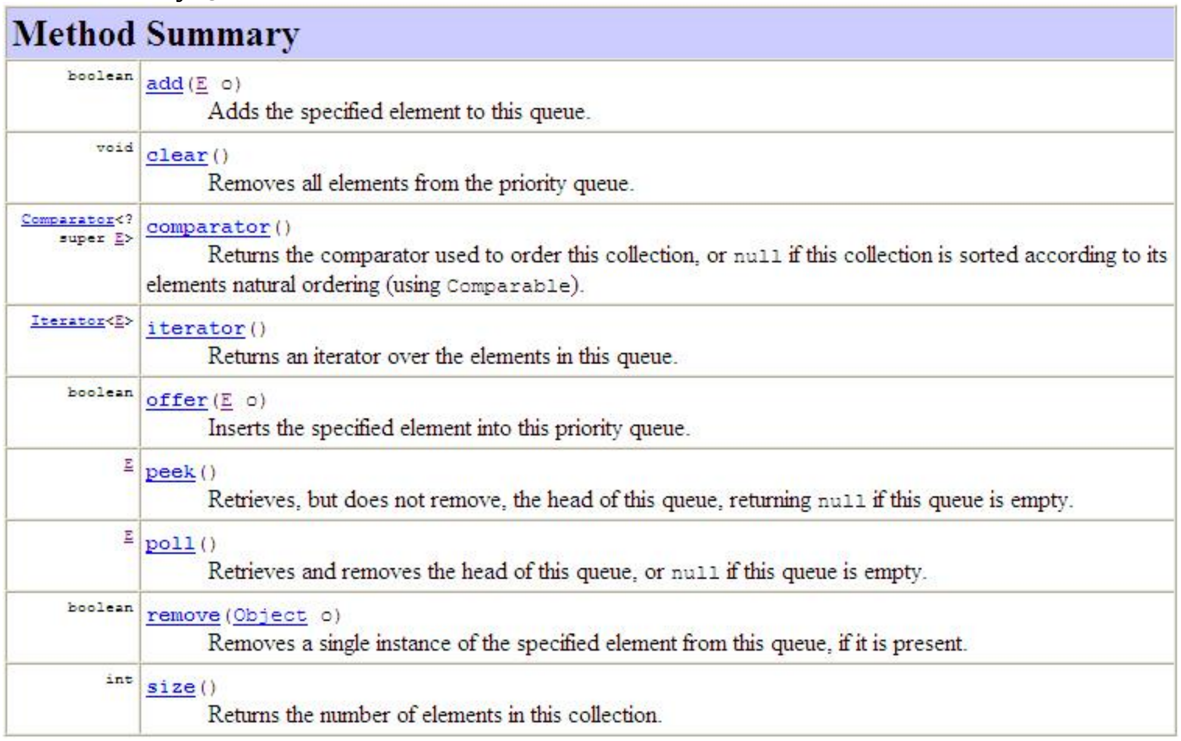
Conclusions

1. BottomUpHeap loads an array of *n* elements into a heap in O(n) time -- this is faster than the O(nlog n) time of inserting one-by-one. Therefore BottomUpHeap is often used when creating an optimized HeapSort implementation.

2. HeapSort typically runs faster than MergeSort but slower than QuickSort.

3. As of j2se5.0, Sun provides a PriorityQueue implementation in java.util, using the heap construction.

Class PriorityQueue<E>



Method Summary	
`boolean`	`add(E o)` Adds the specified element to this queue.
`void`	`clear()` Removes all elements from the priority queue.
`Comparator<? super E>`	`comparator()` Returns the comparator used to order this collection, or `null` if this collection is sorted according to its elements natural ordering (using `Comparable`).
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this queue.
`boolean`	`offer(E o)` Inserts the specified element into this priority queue.
`E`	`peek()` Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
`E`	`poll()` Retrieves and removes the head of this queue, or `null` if this queue is empty.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this queue, if it is present.
`int`	`size()` Returns the number of elements in this collection.

Main Point

The Bottom Up Heap cosntruction significantly improves the efficiency of building a heap from the usual construction in which one element is added to the heap at a time, with the resulting log n upheap operation every time. Bottom Up Heap proceeds by dividing the input list into three: a root *r* and two remaining halves. These remaining halves are recursively organized into separate heaps, and then joined together to form the final heap with root *r*. This process provides an analogy for the process of unfoldment of creation from the unmanifest. In this process, three (rishi, devata, chhandas) emerge from one and then, through self-referral dynamics (a kind of “recursion”), unfold sequentially to form the blueprint of the universe.