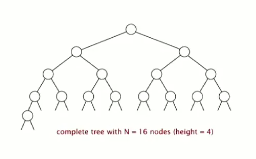
Binary Heaps

Empty or a node with links to the left and right binary trees.

A complete binary tree is perfectly balanced, except possibly for the bottom level



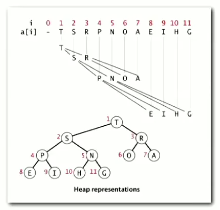
Height of a complete tree with N nodes is [log N]

Height only increases when N is a power of 2

A **binary heap** is an array representation of a heap-ordered complete binary tree

Use complete binary trees for priority queues:

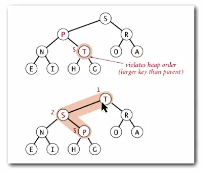
* Associate information with each node by putting keys in each node
* Represent with an array:
  + Indices start at 1
  + Take nodes in level order (root, then two nodes, then four, etc.)
* **Heap ordering:** parent’s key is no smaller than children’s keys



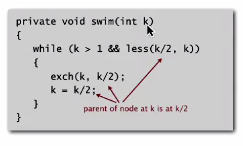
Move around tree by doing arithmetic on indices

* Largest key is a[1], which is root of binary tree
* Can use array indices to move through the tree
  + Parent of node k is at k/2
  + Children of node at k are at 2k and 2k+1

If a child is larger with a parent, exchange child with parent until it does not violate the heap condition

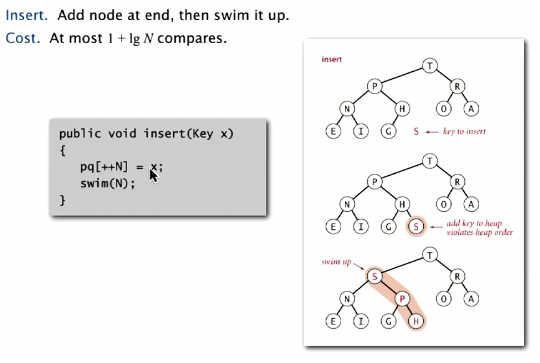


This is called the swim operation. *Implementation*:

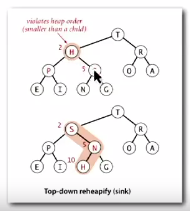


**Insertion in a heap:**

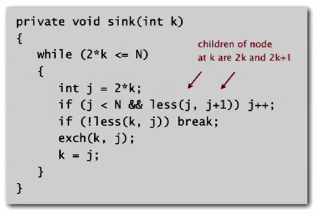
Add node at the end and then swim it up. *Implementation*



If parent is smaller than one or both of its children, parent is exchanged with child that is larger until heap condition is met. This is called the sink operation

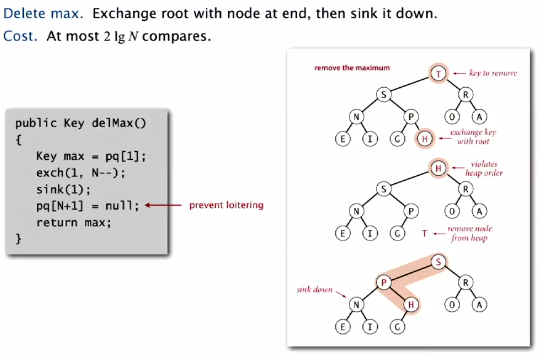


Implementation:



DELETE THE MAXIMUM:

Use the sink operation to exchange root with a node at the end, then delete that node

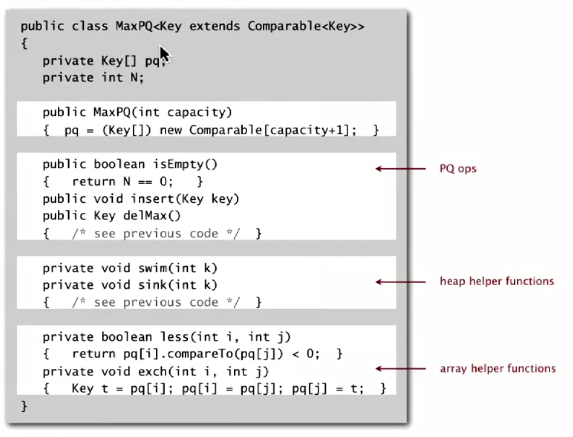


Size of the heap reduced by 1

Save the root (to return to client)

Place to remove the element is at the bottom of the tree, so exchange (top) max key with root, then sink until heap order is not violated

Full implementation of **priority queue using binary tree**:

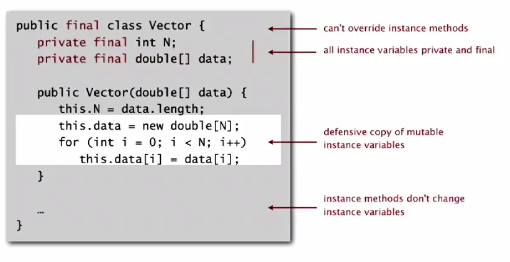


Binary heap considerations:

* Immutability
  + Use IMMUTABLE keys: Client **cannot** change keys while on the PQ.
* Overflow/underflow
  + Underflow: throw exception if deleting from empty PQ
  + Overflow: add no-arg constructor and use resizing array
* Min/max orientation
  + Replace less() with greater()
  + Implement greater()
* Other operations
  + Remove an arbitrary item
  + Change priority of an item

Immutability in Java (can’t change data type once created):

* Such as ‘final’ classes
* Such as ‘private final’ instance variables
* Example:



**Immutable** in Java: String, Integer, Double, Color, etc.

**Mutable**: StringBuilder, Stack, Counter, Java array

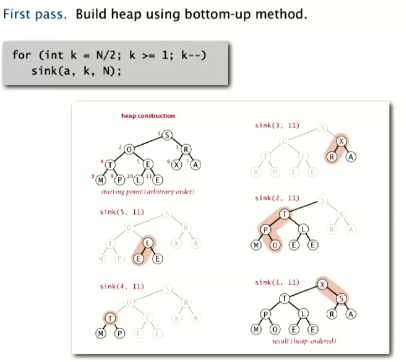
Advantages of immutability:

* Simplifies debugging
* Safer in presence of hostile code
* Simplifies concurrent programming

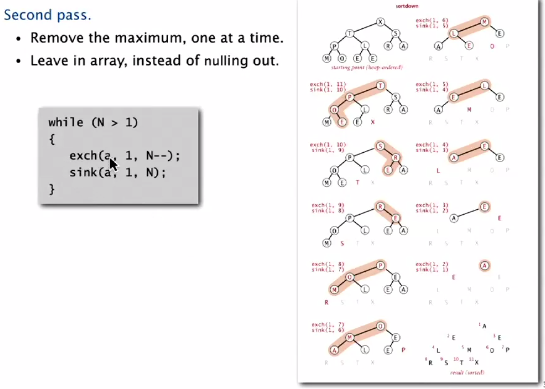
Disadvantage: must create new object for each data type value

**Heapsort**

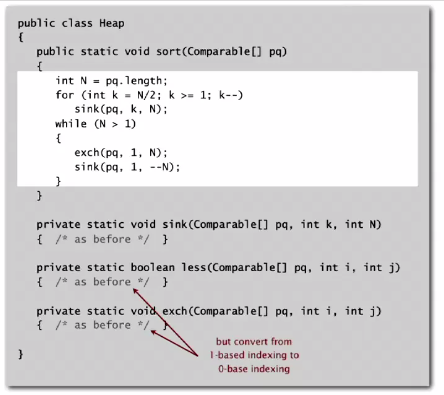
How to in-place sort:

1. Start with array of keys in arbitrary order
2. Create max-heap with all N-keys (heap order)
   1. Build max-heap bottom-up (compare children to parents) 
3. Take heap ordered array and change to sorted result in place

Next, repeatedly delete the largest remaining item, then sink to heap order



Full heapsort implementation:



Heapsort trace



Time complexity:

* Heap construction: <= 2 N compares and exchanges
* Heapsort operation <= 2 N log N compares and exchanges

**In-place sorting algorithm that is in-place AND is N log N time worst case**

Heapsort is time and space optimal, BUT

**Disadvantages:**

* Inner loop is longer than quicksort’s (more compares, array index arithmetic, etc.)
* References to memory are all over the place, so makes poor use of cache memory
* Not stable (long distance exchanges)

