Heaps

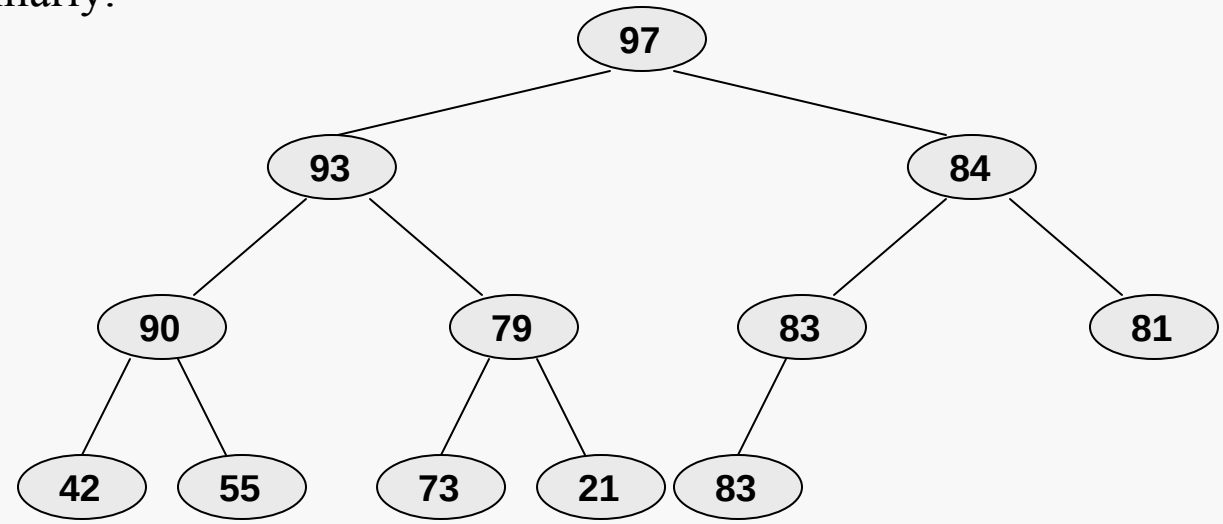
A heap is a complete binary tree.

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node.

A min-heap is defined similarly.

Mapping the elements of a heap into an array is trivial:

if a node is stored at index k, then its left child is stored at index 2k+1 and its right child at index 2k+2



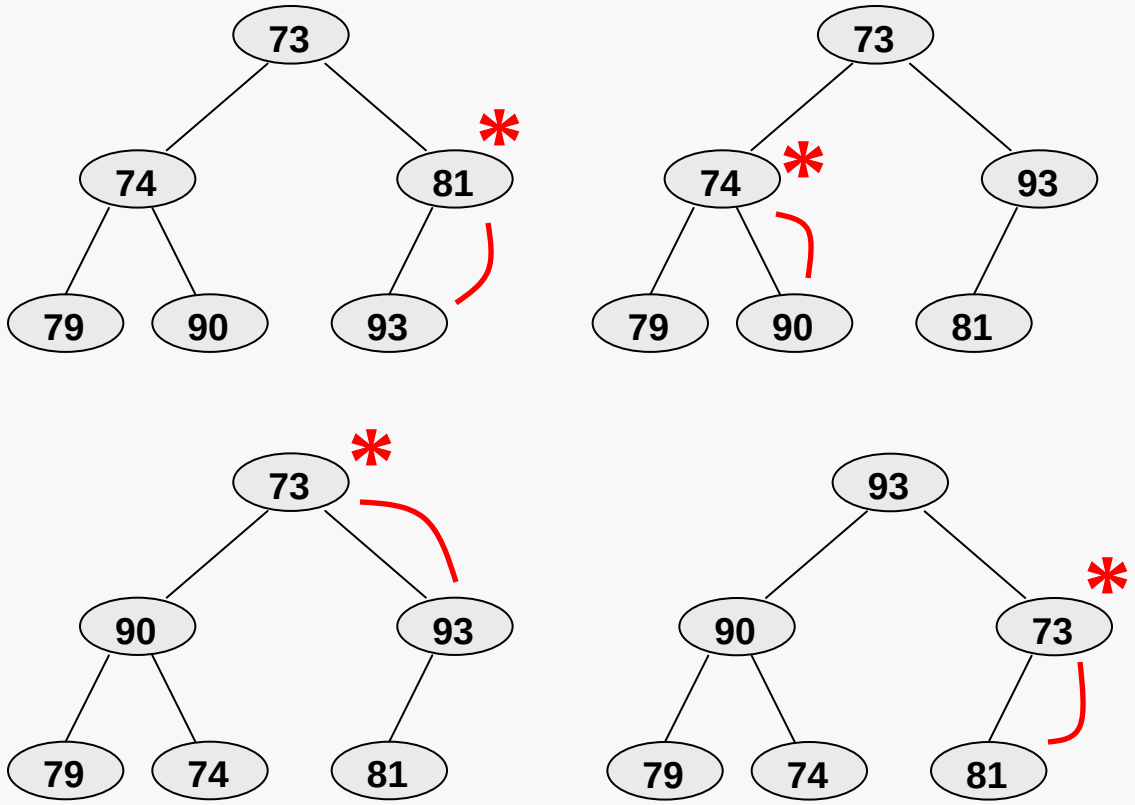
0	1	2	3	4	5	6	7	8	9	10	11
97	93	84	90	79	83	81	42	55	73	21	83

Building a Heap

The fact that a heap is a complete binary tree allows it to be efficiently represented using a simple array.

Given an array of N values, a heap containing those values can be built, *in situ*, by simply "sifting" each internal node down to its proper location:

- start with the last internal node
- swap the current internal node with its larger child, if necessary
- then follow the swapped node down
- continue until all internal nodes are done



Heap Class Interface

We will consider a somewhat minimal maxheap class:

```
public class BinaryHeap<T extends Comparable<? super T>> {

   private static final int DEFCAP = 10;          // default array size
   private int size;                              // # elems in array
   private T [] elems;                            // array of elems


   public BinaryHeap() { . . . }                  // construct heaps in
   public BinaryHeap( int capacity ) { . . . }    //    various ways
   public BinaryHeap( T [ ] items ) { . . . }

   public boolean insert( T D ) { . . . }         // insert new elem
   public T findMax() { . . . }                   // get maximum element
   public T deleteMax() { . . . }                 // remove maximum element
   public void clear() { . . . }                  // reset heap to empty
   public boolean isEmpty() { . . . }             // is heap empty?

   private void siftDown( int hole ) { . . . }    // siftdown algorithm
   private void buildHeap() { . . . }             // heapify
   public void printHeap() { . . . }              // linear display
}
```

buildHeap and siftDown

As described earlier:

```
private void buildHeap() {
   // first elem is stored at index 1, not 0
   for ( int idx = size / 2; idx > 0; idx-- )
      siftDown( idx );
}
```

QTP: Why is idx initialized this way?

```
private void siftDown(int hole) {

   int child;
   T tmp = elems[ hole ];

   for ( ; hole * 2 <= size; hole = child ) {

      child = hole * 2;
      if ( child != size &&
           elems[ child + 1 ].compareTo( elems[ child ] ) > 0 )
         child++;
      if ( elems[ child ].compareTo( tmp ) > 0 )
         elems[ hole ] = elems[ child ];
      else
         break;
   }
   elems[ hole ] = tmp;
}
```

Determine which child node is larger

If child is larger than parent, it must move up

Finally, put starting value in right place

Cost of `siftDown`

```
private void siftDown(int hole) {

   int child;
   T tmp = elems[ hole ];

   for ( ; hole * 2 <= size; hole = child ) {

      child = hole * 2;
      if ( child != size &&
           elems[ child + 1 ].compareTo( elems[ child ] ) > 0 )
         child++;
      if ( elems[ child ].compareTo( tmp ) > 0 )
         elems[ hole ] = elems[ child ];
      else
         break;
   }
   elems[ hole ] = tmp;
}
```

In the worst case, we perform two element comparisons…

…and one element copy per iteration

In a complete binary tree of N nodes, the number of levels is at most 1 + log(N).

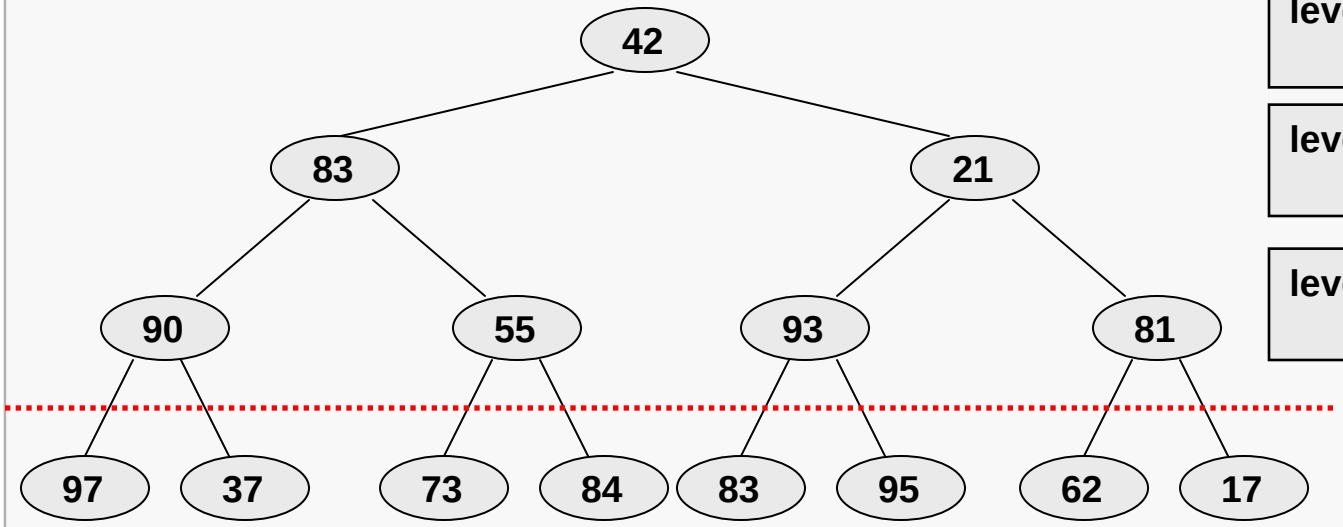
Since each non-terminating iteration of the loop moves the target value a distance of 1 level, the loop will perform no more than log(N) iterations.

Thus, the worst case cost of `SiftDown()` is Θ(log N).

That's 2log(N) comparisons and log(N) swaps.

Cost of `buildHeap`

Suppose we start with a complete binary tree with N nodes; the number of steps required for sifting values down will be maximized if the tree is also full, in which case $N = 2^d - 1$ for some integer $d = \lceil \log N \rceil$. For example:



level 0: 2^0 nodes, can sift down d - 1 levels

level 1: 2^1 nodes, can sift down d - 2 levels

level 2: 2^2 nodes, can sift down d - 3 levels

We can prove that in general, level k of a full and complete binary tree will contain 2^k nodes, and that those nodes are d – k – 1 levels above the leaves.

Thus…

Cost of `buildHeap`

In the worst case, the number of comparisons BuildHeap() will require in building a heap of N nodes is given by:

$$\text{Comparisons} = 2\sum_{k=0}^{d-1} 2^k (d-k-1) = 2\left[(d-1)\sum_{k=0}^{d-1} 2^k - 2\sum_{k=0}^{d-1} k2^{k-1}\right]$$
$$= 2\left[2^d - d - 1\right] = 2\left[N - \lceil \log N \rceil\right]$$

Since, at worst, there is one move for each two comparisons, the maximum number of element moves is $N - \lceil \log N \rceil + 1$.

Hence, building a heap of N nodes is $\Theta(N)$ in both comparisons and element moves.

Deleting the Root of a Heap

We will see that the most common operation on a heap is the deletion of the root node. The heap property is maintained by sifting down…

```
public T deleteMax() {

   if ( isEmpty( ) )
      throw new RuntimeException( );

   T maxItem = elems[ 1 ];
   elems[ 1 ] = elems[ size ];
   size--;

   siftDown( 1 );

   return maxItem;
}
```

Check for empty heap

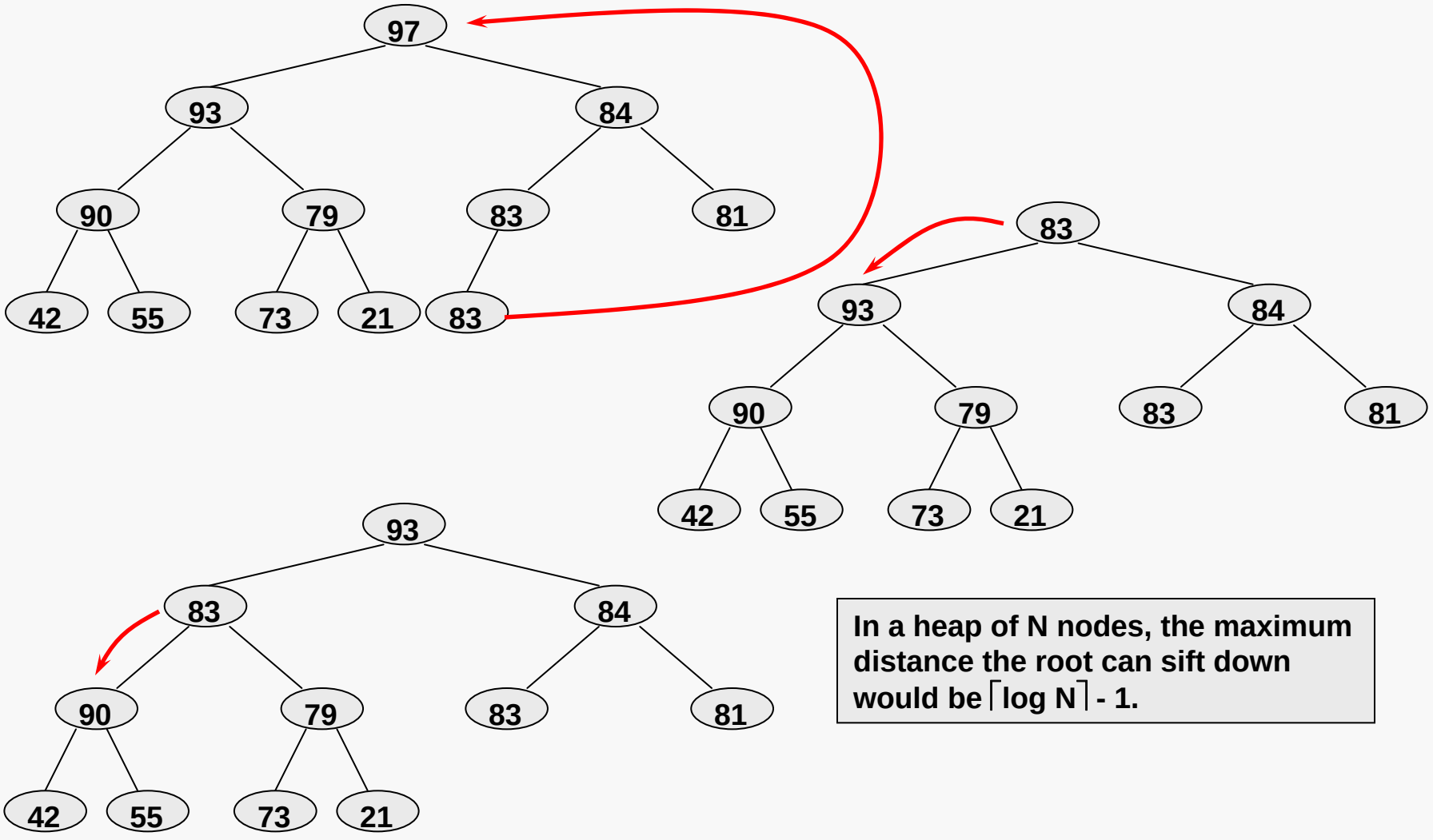
Save the root (at index 1), replace it with the last leaf, shrink the heap by one node.

Sift the new root down to restore the heap property.

QTP: Why is the last leaf chosen as the replacement for the root?

Example of Root Deletion

Given the initial heap:



In a heap of N nodes, the maximum distance the root can sift down would be ⌈log N⌉ - 1.

Heap Sort

A list can be sorted by first building it into a heap, and then iteratively deleting the root node from the heap until the heap is empty. If the deleted roots are stored in reverse order in an array they will be sorted in ascending order (if a max heap is used).

```
public static void heapSort(Integer[] List, int Sz) {

   BinaryHeap<Integer> toSort = new BinaryHeap<Integer>(List, Sz);

   int Idx = Sz - 1;
   while ( !toSort.isEmpty() ) {
      List[Idx] = toSort.deleteMax();
      Idx--;
   }
}
```

Cost of Deleting the Roots

Recalling the earlier analysis of building a heap, level k of a full and complete binary tree will contain 2^k nodes, and that those nodes are k levels below the root level.

So, when the root is deleted the maximum number of levels the swapped node can sift down is the number of the level from which that node was swapped.

Thus, in the worst case, for deleting all the roots…

$$\begin{aligned}\text{Comparisons} &= 2\sum_{k=1}^{d-1} k2^k = 4\sum_{k=1}^{d-1} k2^{k-1} = 4\left[(d-2)2^{d-1}+1\right] \\ &= 2N\lceil \log N \rceil + 2\lceil \log N \rceil - 4N\end{aligned}$$

As usual, with Heap Sort, this would entail essentially the same number of element moves.

Cost of Heap Sort

Adding in the cost of building the heap from our earlier analysis,

$$\begin{aligned}\text{Total Comparisons} &= \left(2N - 2\lceil \log N \rceil\right) + \left(2N\lceil \log N \rceil + 2\lceil \log N \rceil - 4N\right) \\ &= 2N\lceil \log N \rceil - 2N\end{aligned}$$

and...

$$\text{Total Swaps} = N\lceil \log N \rceil - N$$

So, in the worst case, Heap Sort is Θ(N log N) in both swaps and comparisons.

Priority Queues

A *priority queue* consists of entries, each of which contains a key field, called the *priority* of the entry.

Aside from the usual operations of creation, clearing, tests for full and empty, and reporting its size, a priority queue has only two operations:

- insertion of a new entry
- removal of the entry having the largest (or smallest) key

Key values need not be unique. If not, then removal may target any entry holding the largest value.

Representation

Representation of a priority queue may be achieved using:

- a sorted list, but…
- an unsorted list, but…
- a max-heap

Priority queues may be used to manage prioritized processes in a time-sharing environment, time-dependent simulations, and even numerical linear algebra.