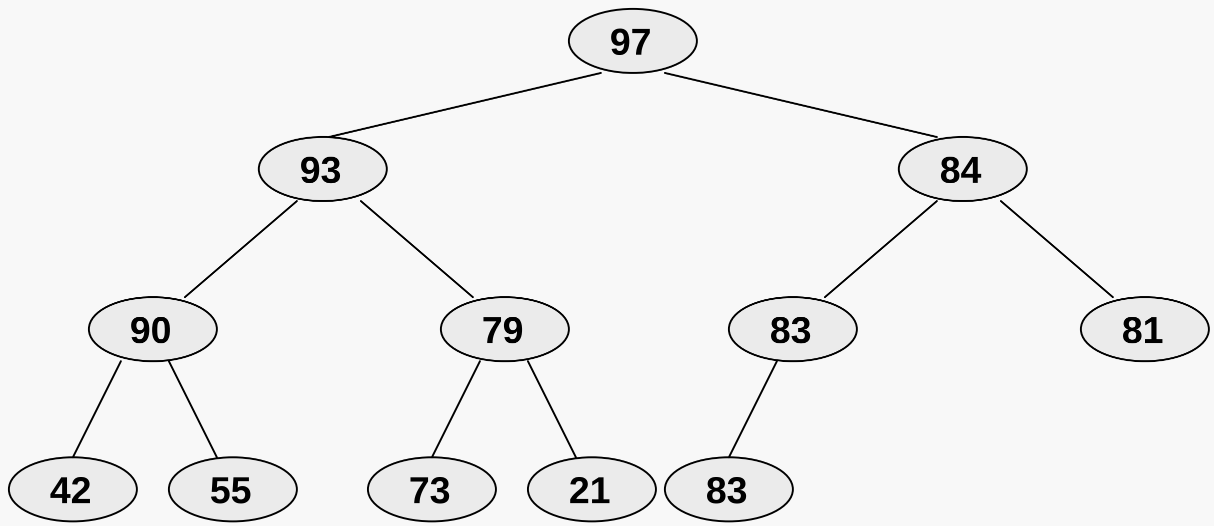
Heaps

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

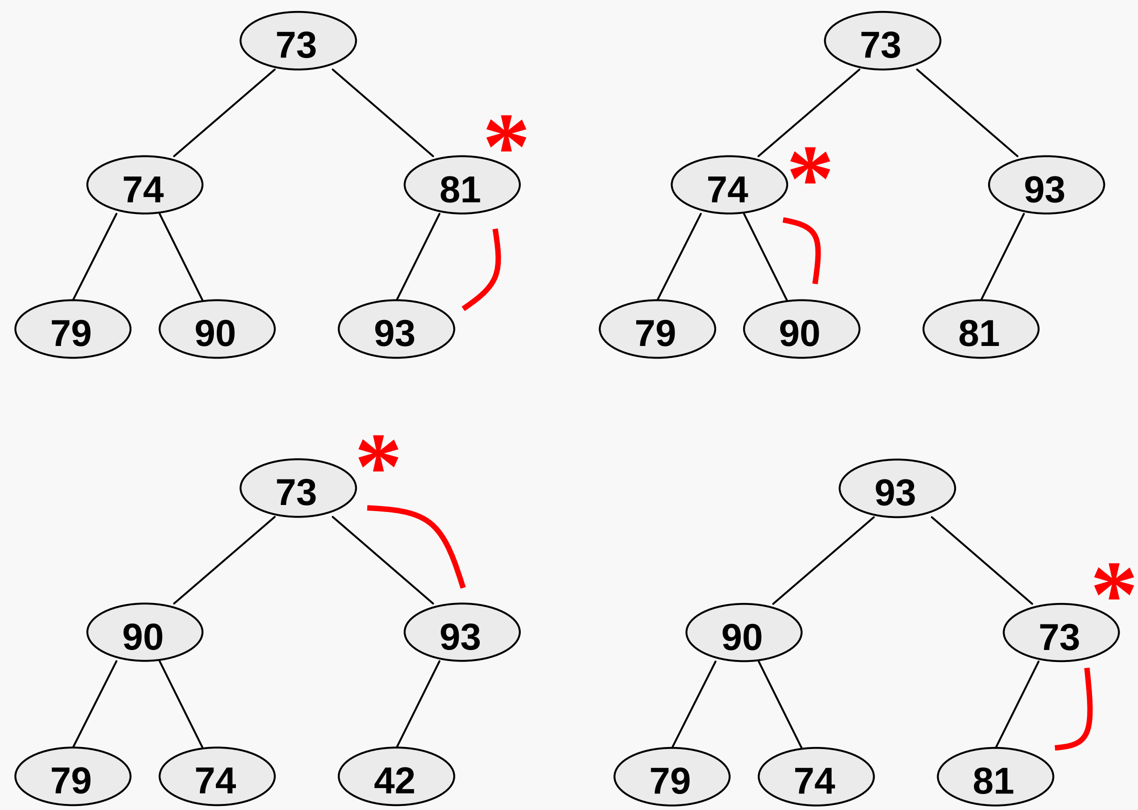
0	1	2	3	4	5	6	7	8	9	10	11
97	93	84	90	79	83	83	42	55	73	21	80

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 heap class:

```
template <class Item> class HeapT {
private:
   Item* Hp;
   int   numVertices;
   int   Size;

   void  SiftDown(int toSift);            // helper for BuildHeap, etc.

public:
   HeapT();
   HeapT(Item* List, int numItems);       // create from an array
   HeapT(const HeapT& Source);
   HeapT& operator=(const HeapT& Source);
   ~HeapT();
   bool isEmpty() const;
   bool isLeaf(int Vertex) const;
   int  leftChild(int Vertex) const;      // traversal navigation
   int  rightChild(int Vertex) const;
   int  Parent(int Vertex) const;

   Item RemoveRoot();                     // deletes root & re-heaps
   void BuildHeap();                      // heapifies the array
};
```

BuildHeap and SiftDown

As described earlier:

```
template <class Item> void HeapT<Item>::BuildHeap() {
   for (int Idx = numVertices/2 - 1; Idx >= 0; Idx--)
      SiftDown(Idx);
}
```

QTP: Why is Idx initialized this way?

```
template <class Item> void HeapT<Item>::SiftDown(int toSift) {

   while ( !isLeaf(toSift) ) {
      int MaxChild = leftChild(toSift);
      if ( (MaxChild < numVertices - 1) &&
           (Hp[MaxChild] < Hp[MaxChild + 1]) )
         MaxChild++;

      if (Hp[toSift] >= Hp[MaxChild]) return;

      Item tmpItem = Hp[toSift];
      Hp[toSift] = Hp[MaxChild];
      Hp[MaxChild] = tmpItem;

      toSift = MaxChild;
   }
}
```

Determine which child node is larger

See if node must sift down

If so, swap it with its larger child (maintaining heap property),

and continue sifting down…

Cost of SiftDown

```
template <class Item> void HeapT<Item>::SiftDown(int toSift) {

   while ( !isLeaf(toSift) ) {
      int MaxChild = leftChild(toSift);
      if ( (MaxChild < numVertices - 1) &&
           (Hp[MaxChild] < Hp[MaxChild + 1]) )
         MaxChild++;

      if (Hp[toSift] >= Hp[MaxChild]) return;

      Item tmpItem = Hp[toSift];
      Hp[toSift] = Hp[MaxChild];
      Hp[MaxChild] = tmpItem;

      toSift = MaxChild;
   }
}
```

In the worst case, we perform two element comparisons…

…and one element swap per iteration

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

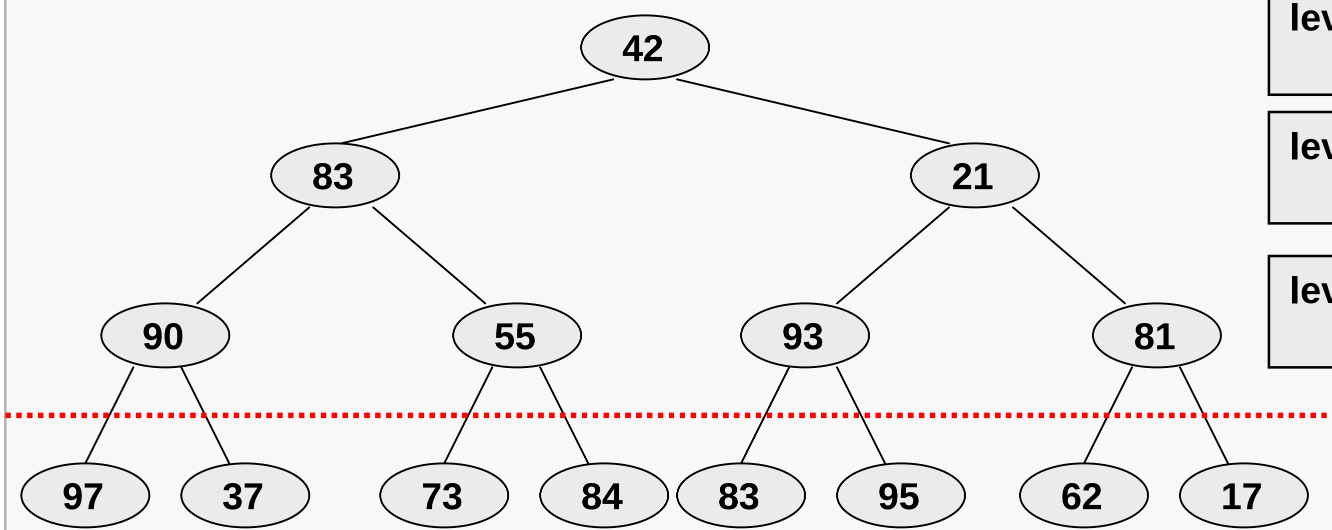
Since each non-terminating iteration of the loop moves the target value a distance of 1 level, the loop will perform no more than log(N) iterations.

Thus, the worst case cost of `SiftDown()` is O(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:

$$\text{Comparisons} = 2\sum_{k=0}^{d-1} 2^k (d-k-1) = 2\left[(d-1)\sum_{k=0}^{d-1} 2^k - 2\sum_{k=0}^{d-1} k\,2^{k-1}\right]$$

$$= 2\left[2^d - d - 1\right] = 2\left[N - \lceil \log N \rceil\right]$$

Since, at worst, there is one swap for each two comparisons, the maximum number of swaps is $N - \lceil \log N \rceil$.

Hence, building a heap of N nodes is O(N) in both comparisons and swaps.

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…

```
template <class Item> Item HeapT<Item>::RemoveRoot() {

   if (numVertices == 0) return Item();

   Item tmpItem = Hp[0];
   Hp[0] = Hp[numVertices - 1];
   Hp[numVertices - 1] = tmpItem;
   numVertices--;


   SiftDown(0);

   return tmpItem;
}
```

Check for empty heap

Swap the root with the last leaf, and

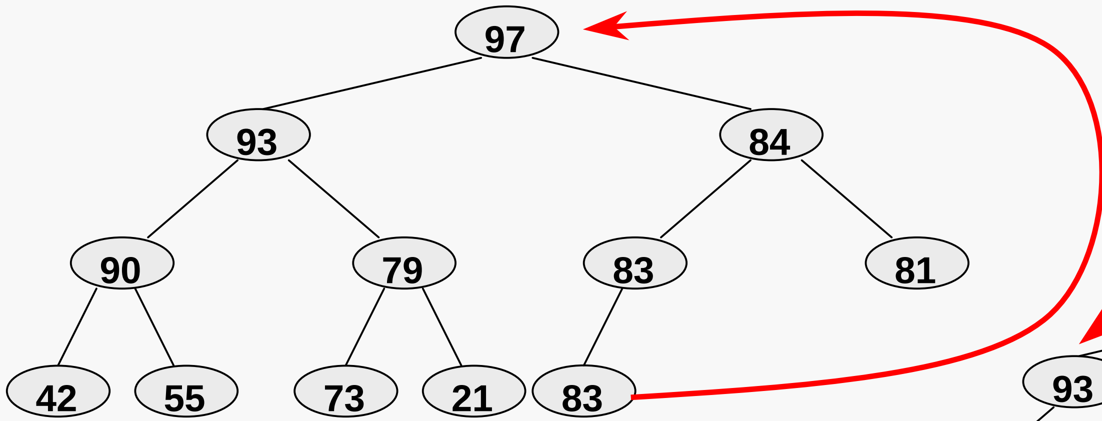
shrink the heap by one node.

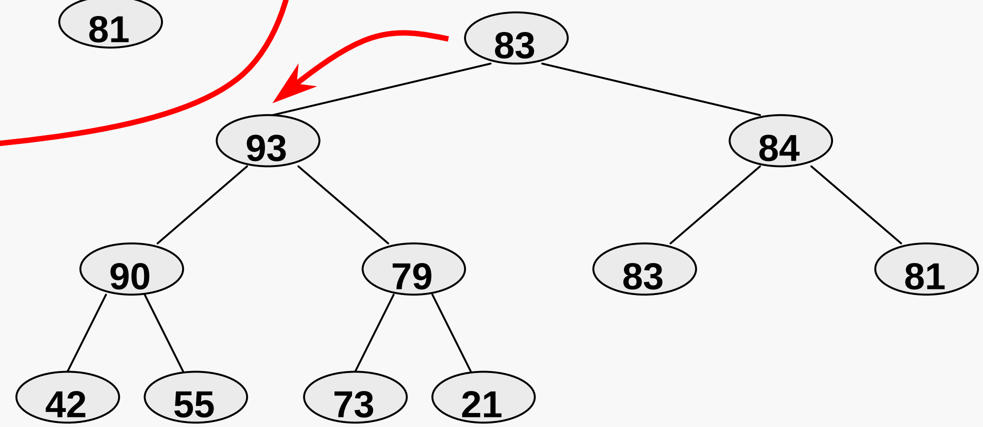
Sift the new root down to restore the heap property.

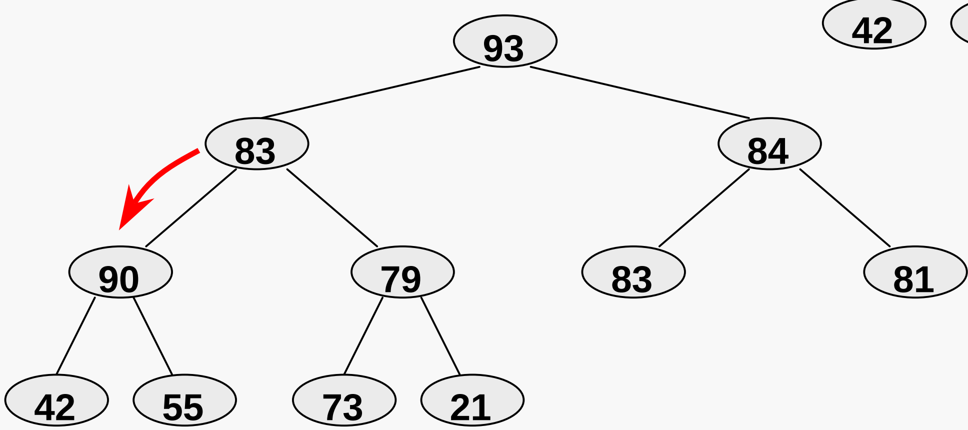
QTP: Why is the last leaf chosen as the replacement for the root?

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).

```
void HeapSort(int* List, int Size) {

   HeapT<int> toSort(List, Size);
   toSort.BuildHeap();

   int Idx = Size - 1;
   while ( !toSort.isEmpty() ) {
      List[Idx] = toSort.RemoveRoot();
      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…

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

As usual, with Heap Sort, this would entail half as many element swaps.

Cost of Heap Sort

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

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

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.