Heaps: a quick summary, and some remarks on notation

Heap data structure:

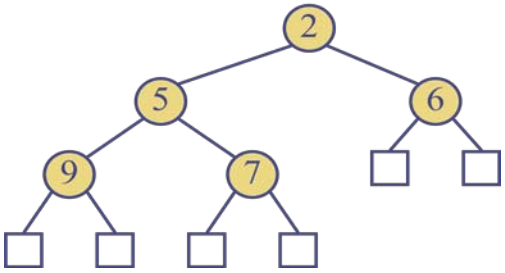
A binary tree in which nodes store *keys* (and possibly other information as well), satisfying two properties:

- The tree is *left full*: if the tree has height *h*, then the first *h* rows of the tree are full, and the last row (row *h* + 1) is partly or completely full, but is "filled in from the left": if a node is missing, so are all nodes to the right of it.
- The tree is in *heap order*: the key for any non-root node is greater than the key for its parent

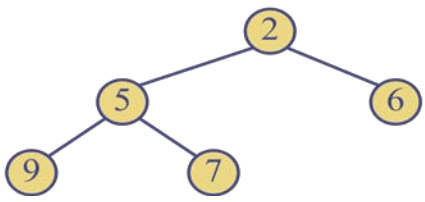
In one version of the heap implementation, all nodes have either zero or two children. Nodes with zero children are called "external" and store no key; those with two children are "internal" and store a key. Most of the diagrams in this document (taken from older CS16 slides) show this "0-2" structure; such trees are sometimes called "complete binary trees" (in contrast to the 2010 CS16 usage, in which "complete" means that the last row of the tree is completely full. Both usages are common, alas.)

The other version of the heap implementation omits the "empty leaf" nodes; not surprisingly, it has fewer nodes overall – about half as many. In big-O terms, this is of no concern; in practice, the no-external-nodes model can be very compactly represented in an array structure, and this tends to work extremely well for many purposes. For the remainder of the document, you can convert from the "0-2" tree model to the "leaves have content" model by just removing all the purple squares in any picture.

An example heap:



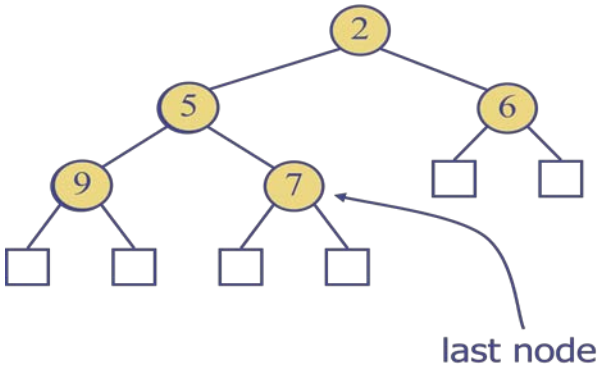
The same heap without the "external" nodes:



Continuing to follow the slides:

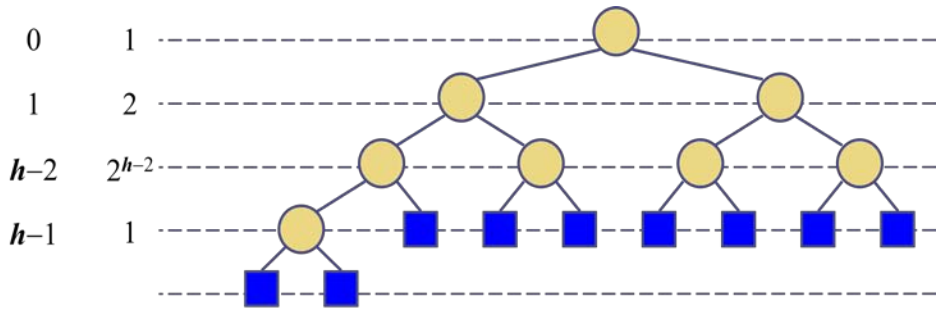
What is a heap?

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, ***key***(***v***) ≧ ***key***(***parent***(***v***)). (In other words, every internal node in the graph stores a key greater than or equal to its parent's key.)
  - Complete Binary Tree: let ***h*** be the height of the heap
    - for ***i*** = 0, … , ***h*** - 1, there are 2^i nodes of depth ***i***
    - at depth ***h*** - 1, the internal nodes are to the left of the external nodes
    - The last node of a heap is the rightmost internal node of depth ***h*** - 1



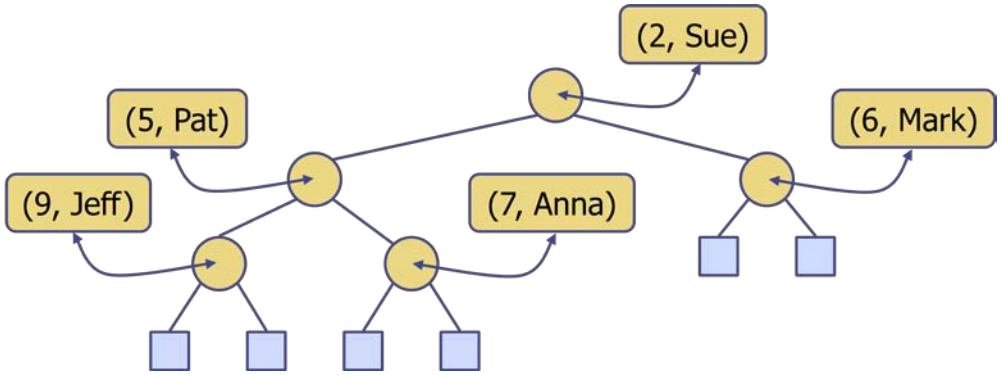
How tall is a heap?

- Theorem: A heap storing ***n*** keys has height ***O***(log ***n***)
- Proof: (we apply the (0,2)-binary tree property)
  - Let ***h*** be the height of a heap storing ***n*** keys. Since there are 2^i keys at depth ***i*** = 0, … , ***h*** - 2 and at least one at depth ***h*** - 1, we have ***n*** ≧ $1 + 2 + \ldots + 2^{h-2} + 1 = (2^{h-1} - 1) + 1 = 2^{h-1}$
  - Thus, ***n*** ≧ 2^{h-1} , i.e., log ***n*** + 1 ≧ ***h.***



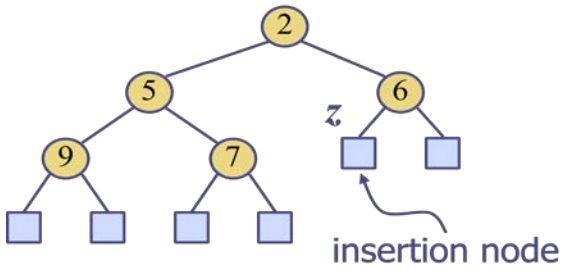
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we generally show only the keys in the pictures

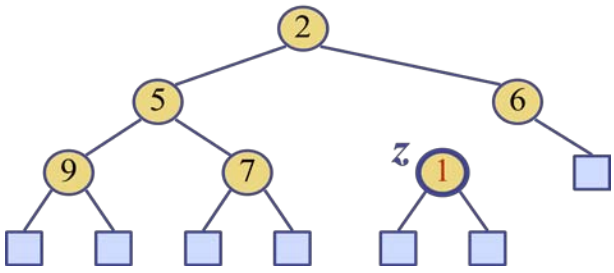


Insertion into a Heap

- 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)



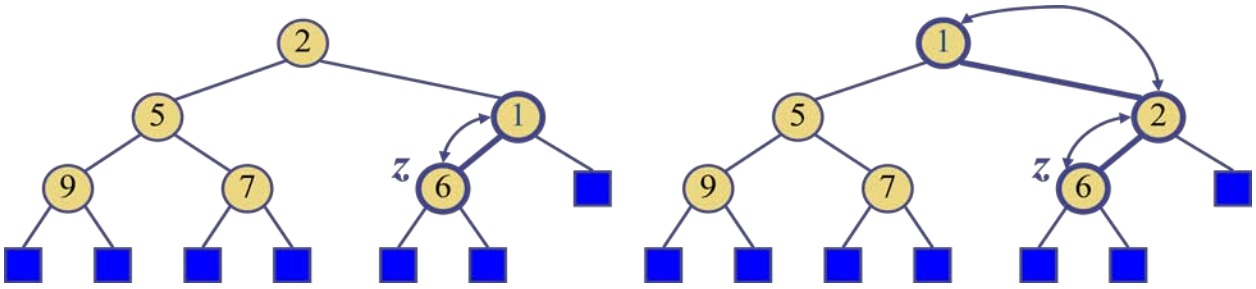
  - Store ***k*** at ***z*** and expand ***z*** into an internal node [Expansion isn't necessary for the case where it's not a (0,2)-tree) … but you have to create the insertion node – see last page!]



  - Restore the heap-order property (discussed next)

Upheap (restoring heap order after insertion)

- 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

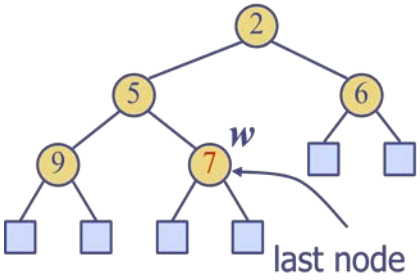


Hwo do we know upheap works?

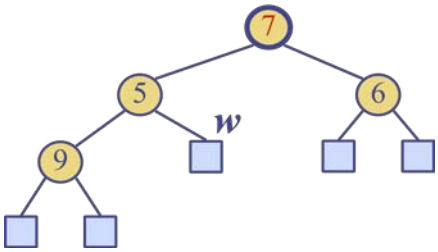
1. It seems pretty obvious
2. We can prove it by induction.
    a. After we introduce the node into the last row, the very last row consists of a collection of (very short) trees, all of them in heap order. In other words: if you covered up the rest of the heap with your hand, you'd see a bunch of one-node trees, all of which are, of course, in heap order.
    b. After we perform one iteration of upHeap, the last TWO rows consist of a collection of trees, each in heap order.
    c. In general, if k rows are in heap order before we perform upheap, k+1 rows are after.
    d. We can conclude that after *h* upheaps, the single tree starting at the root is in heap order. QED.

Removal from a Heap

- Method removeMin of the PriorityQueue ADT ⇔ removing the root key from the heap
- The removal algorithm consists of three steps
  - Swap the root key with the key of the last node ***w*** **[which destroys the heap-order property for the moment!]**



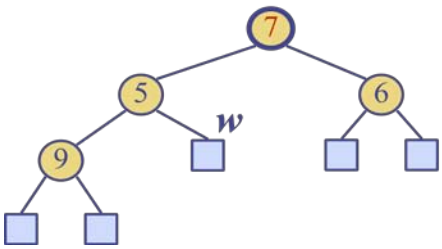
  - Compress ***w*** and its children into a single leaf node
  - Restore the heap-order property (discussed next)

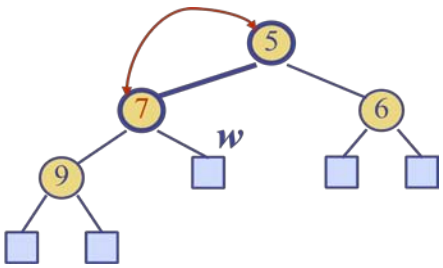


Restoring heap order (after a removeMin): downHeap!

- After replacing the root key with the key ***k*** of the last node, the heap-order property may be violated for rows below the first row.
- Algorithm downheap restores the heap-order property by swapping key ***k*** along a downward path from the root. ***Always swap with smaller child!***
- 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

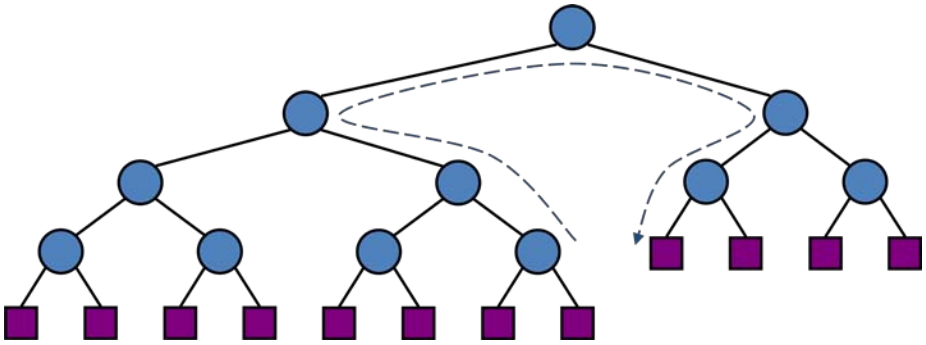
Note: the proof of correctness is the same as for upheap, except now the inductive hypothesis is P(k): “the heap order property holds for the tree consisting of all nodes whose height is less than k.”





Keeping track of the "last" node and "insertion" node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Start with the current last node
  - Go up until a left child or the root is reached
  - If a left child is reached, go to its sibling (the corresponding right child)
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



Additional Operations

If we store (key, value) pairs in a heap (which is a typical use), we can do things like replace the value at some node (pretty trivial), or replace the *key*, which is not so trivial: when you replace the key, things may end up out of order. Arguably it's best to make *two* methods: reduceKey and increaseKey. The first is fairly simple: if you reduce a key at depth h, then all rows from depth h to the last row remain in heap order; only rows at depths 0 to h-1 might *not* be in heap order. In particular, the reduced key might be less than its parent. But we know how to handle this: we just invoke upHeap on this node. So we have

```
def reduceKey(node, newKey):
   check that newKey is less than current key, and throw exception if not

   node.setKey(newKey)
   upHeap(node)
```

The case where we *increase* the key is fairly similar: after the key-increase, rows 0 to h are still in heap order, but rows h+1 to the last row might not be. So we perform downHeap on the node to fix that (possible) problem.

Heap sort

We can use a heap for sorting a bunch of items. The algorithm is called “heap sort.”

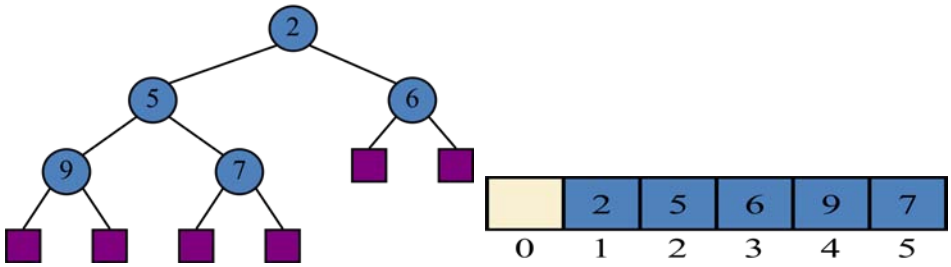
```
while (input not empty)
   insert input item into a heap H
while H is not empty
   extractMin from H and insert it into an output list
```

Runs on O(n log n) time.

Vector-based Implementation

- We can represent a heap with ***n*** keys by means of a vector of length ***n*** + 1
- For the node at rank ***i***
  - the left child is at rank 2***i***
  - the right child is at rank 2***i*** + 1
- Links between nodes are not explicitly stored
- The leaves are not represented
- The cell of at rank 0 is not used
- Operation insertItem corresponds to inserting at rank ***n*** + 1
- Operation removeMin corresponds to removing at rank ***n***
- Yields in-place heap-sort

Tree-structured heap and corresponding array implementation:



Building a heap from a set of keys

It's clear we can build a heap by inserting keys one at a time.

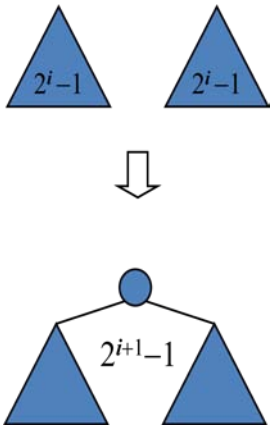
By the time we've inserted n/2 of them, the heap has height at least log(n/2) = log(n) – 1.

For the next n/2 insertions, the insert time is at least log(n) – 1, for a total insertion time of at least (n/2)[log(n) – 1], which is Omega(n log n).

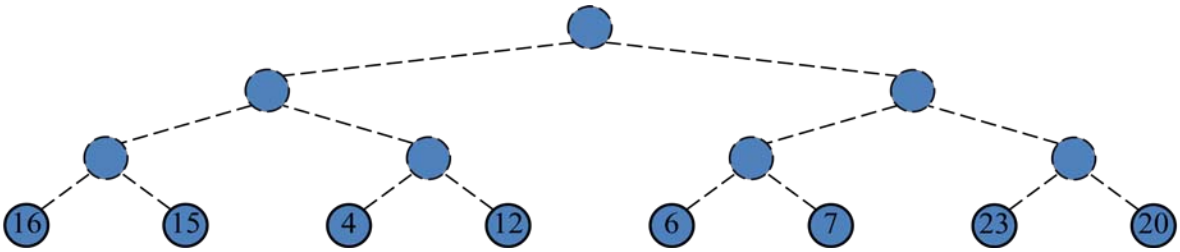
Amazingly, we can construct a heap of n items in time O(n) by being cleverer. The main idea is this: we can merge two height-h heaps and one new element into a height-(h+1) heap fairly quickly: the merge takes O(log h) time. The clever thing is that this log(h) time isn't used too often, as you'll see.

Bottom-up Heap Construction

- We can construct a heap storing ***n*** given keys in 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

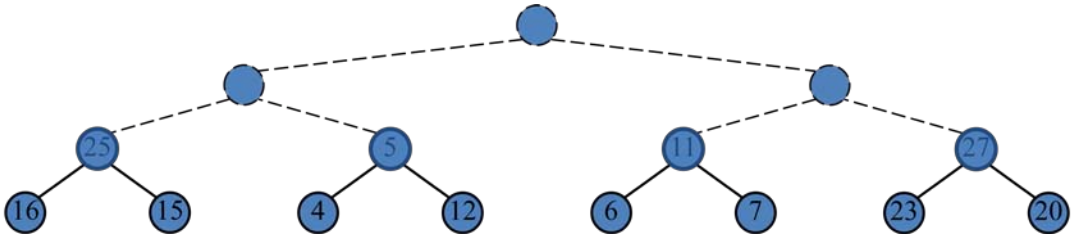


We'll form a heap from the keys 16, 15, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 7, 8, 10, in that order. There are 15 of these, so we expect a heap of 15 internal nodes. The bottom row of internal nodes will have 8 items; the next up will have 4; the next will have 2, and the last will have one, the root. We start by placing the first 8 nodes *in order* into the bottom 8 nodes of the tree:

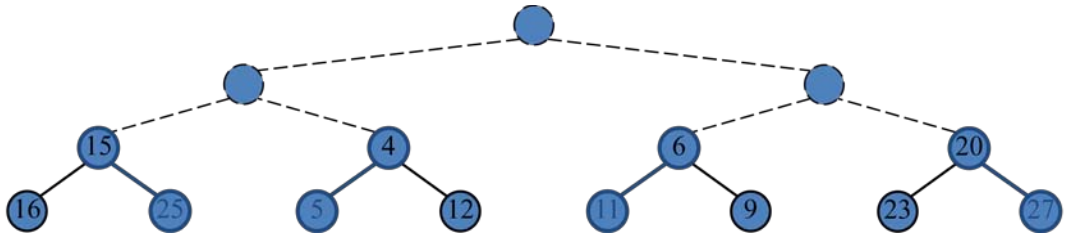


As you can see, the bottom row consists of 8 tiny trees, each in heap order.

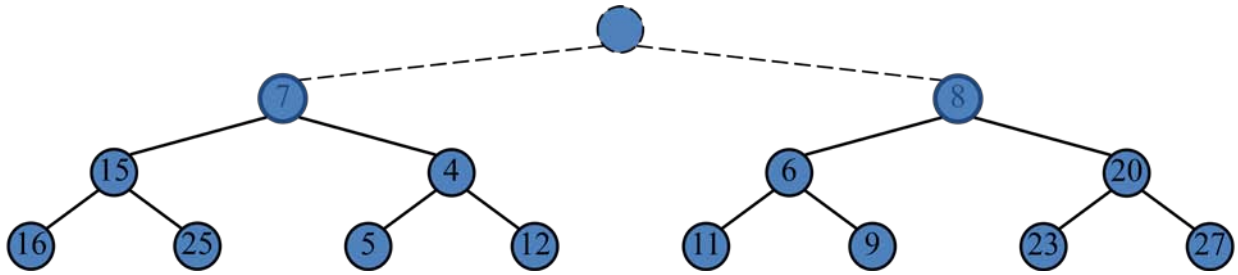
Now we insert the next 4 items into the next row up; *this destroys heap order!*\



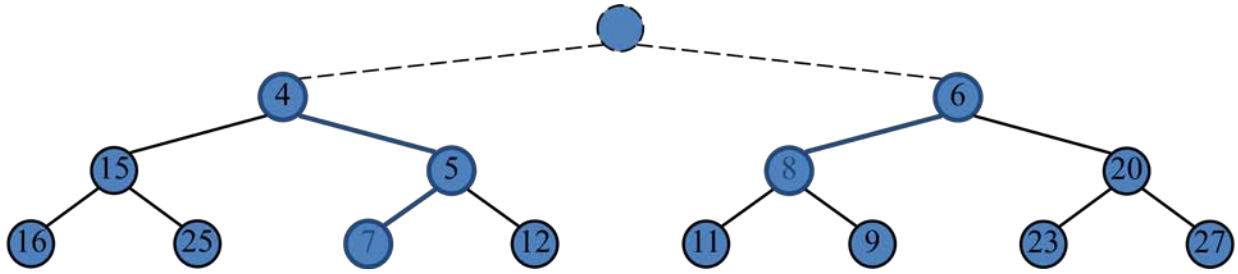
So we apply "downHeap" to the roots of each of the four trees:



Once more we insert two new nodes…which are not in heap order…

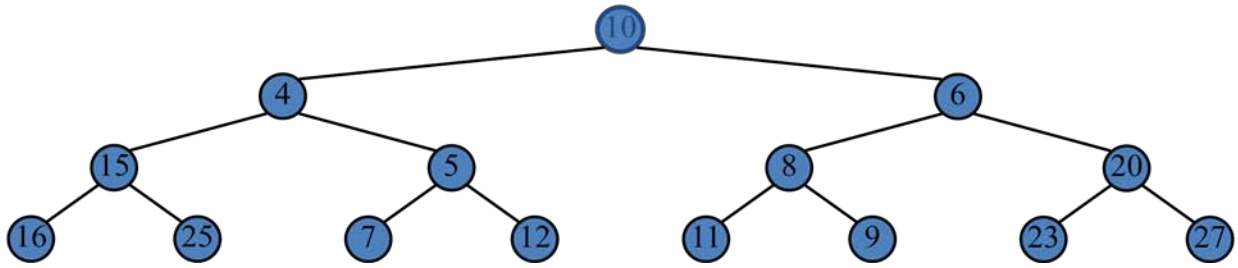


and then apply down-heap again:

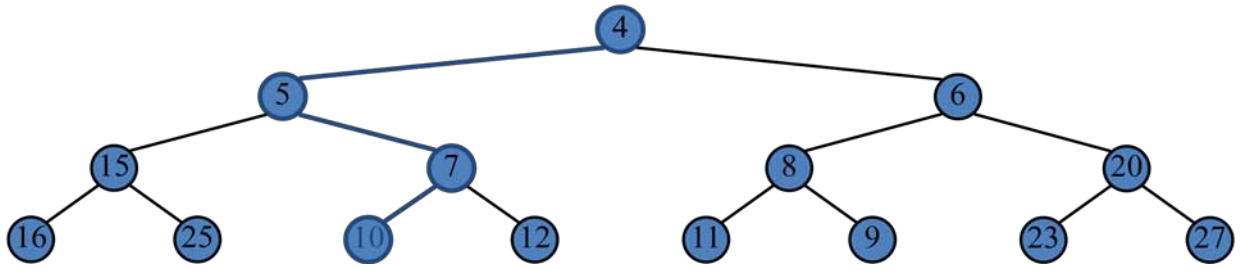


Notice that on the right-hand subtree, the down-heap operation pushed the root all the way to the bottom row.

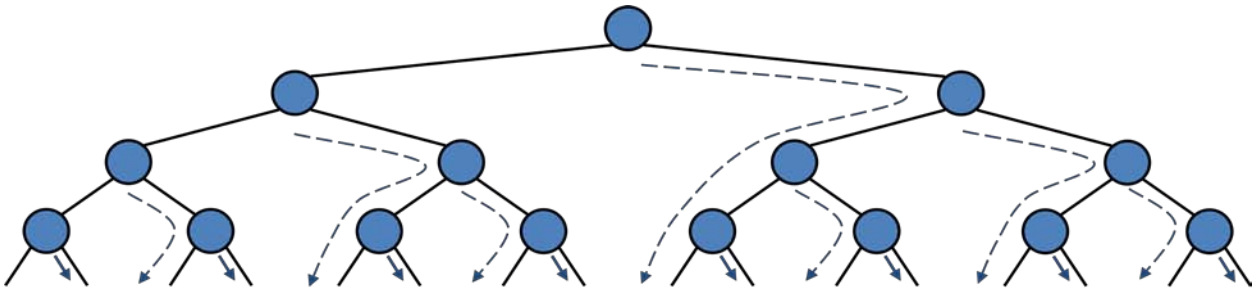
Finally, we insert the last item…



… and run downHeap on it:



That seems like a lot of down-heap operations, but let's keep track of the number of actual node-swaps we performed. When we inserted an element, it might, at worst, have been swapped all the way to the bottom row, possibly swapping with the left or right child at each stage. Let's draw an *equally long* path, but with a simpler structure: it'll start at the insertion location, go right once, and then go left thereafter. In that case, the worst-case picture of all these swaps looks like this:



Notice that each node is involved in *at most two swaps*. That means the total number of swaps is no more than twice the number *n* of nodes, so the total work is O(*n*)!

Final remarks

All our discussions have talked about heaps with numbers in them. It's a small matter to say that the heap contains key-value pairs, where the keys can be compared with some *comparator*, C – something for which C(k1, k2) returns -1, 0 or +1 according to whether k1 is less than k2, equal, or greater (in whatever ordering we want to use on keys). Similarly, it makes sense to let a user refer to a position in a heap, but not in a way that lets him/her alter the heap structure. So when we create a node, we typically create a Position as well; in Java, that looks something like this (within the Heap class definition):

```
protected class Node{
   …declarations of instance vars, etc….
   protected Node(key, value){
      _myKey = key;
      _myVal = val;
      _myPos = new Position(this)
   }
   ...
}
```

```
public class Position {
    private Node _myNode;

    protected Position(Node n){
      _myNode = n;
   }
   public getKey() {
       return _myNode.getKey();
   }
   public getVal() {
       return _myNode.getVal();
   }
   ...
}
```

Within such a structure, the heapUp operation would typically be implemented by the Node class, which would take care of swapping entries (key-value pairs) between a node and its parent or kids. (It would typically be a protected member of the Node class as well.)

Positions, meantime, remain constant: a position always refers to a particular place in the tree. For instance, if you have a position that refers to the root, and you do an extractMin, that position typically remains valid, but if you ask for its key or value after the extractMin, you'll get something different from what you got before.

One last detail

When we did insertion into the heap, we said "expand the insertion node into an internal node." What if we're not storing the leaves (i.e., we're not maintaining a (0,2) tree with "dummy" external nodes)? Then the "insertion node" isn't a node at all – it's a place where a new node is about to be put. How do we "keep track of it"?

Answer: you can instead keep track of the insertion-node-parent.

If that parent has no left child, then to insert, you create a new left-child, and put the new item in there. Because the next item inserted will be the RIGHT child of the current insertion-node-parent, you don't need to change it.

On the other hand, if the parent has a left-child already, then you create a new right-child node, and put the new item there. At this point, you need to update the insertion-node-parent using a scheme a lot like the one described above in "keeping track of the last node and insertion node".