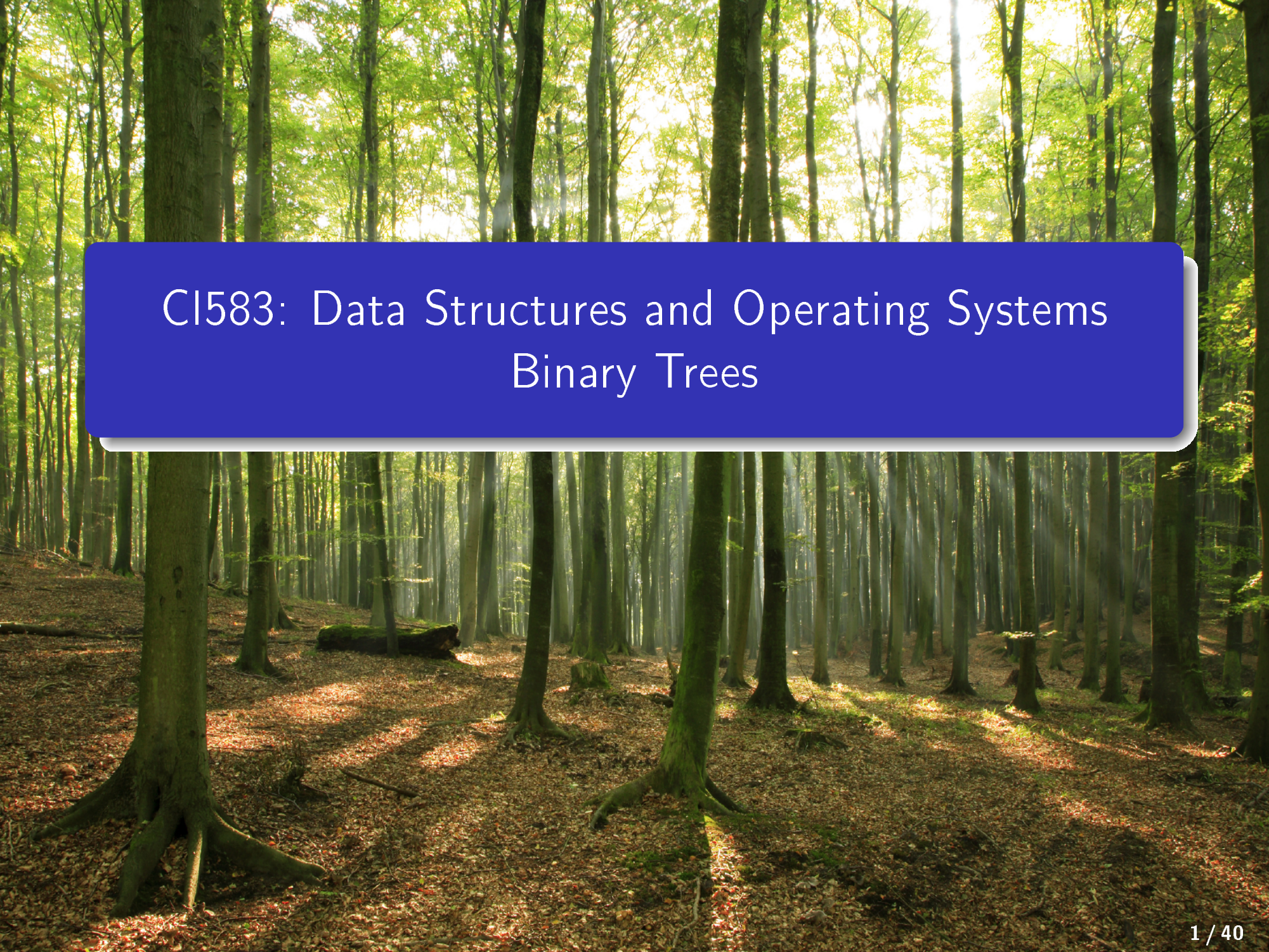


CI583: Data Structures and Operating Systems
Binary Trees

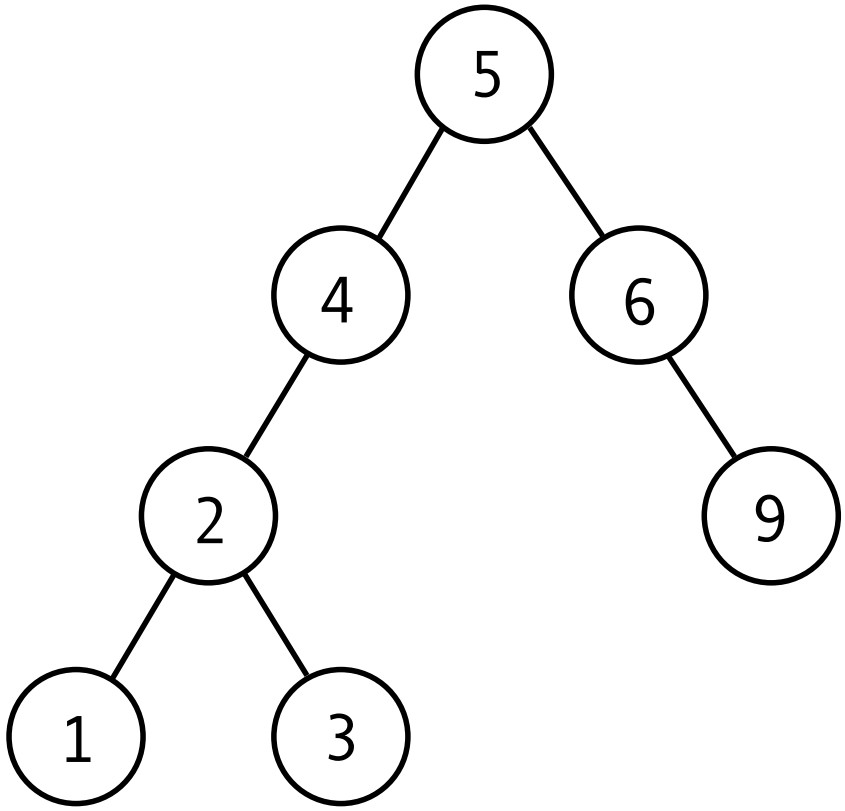
Binary search trees

Trees start to get interesting when we place some constraints on their structure. One constraint is that their labels are ordered. If we do this with a binary tree we get a binary search tree (BST), defined as follows:

1. for each non-leaf node, n, the key of the left child is less than the key of n and the key of the right child is greater than the key of n,
2. keys are unique, and
3. the left and right children of n are binary search trees.

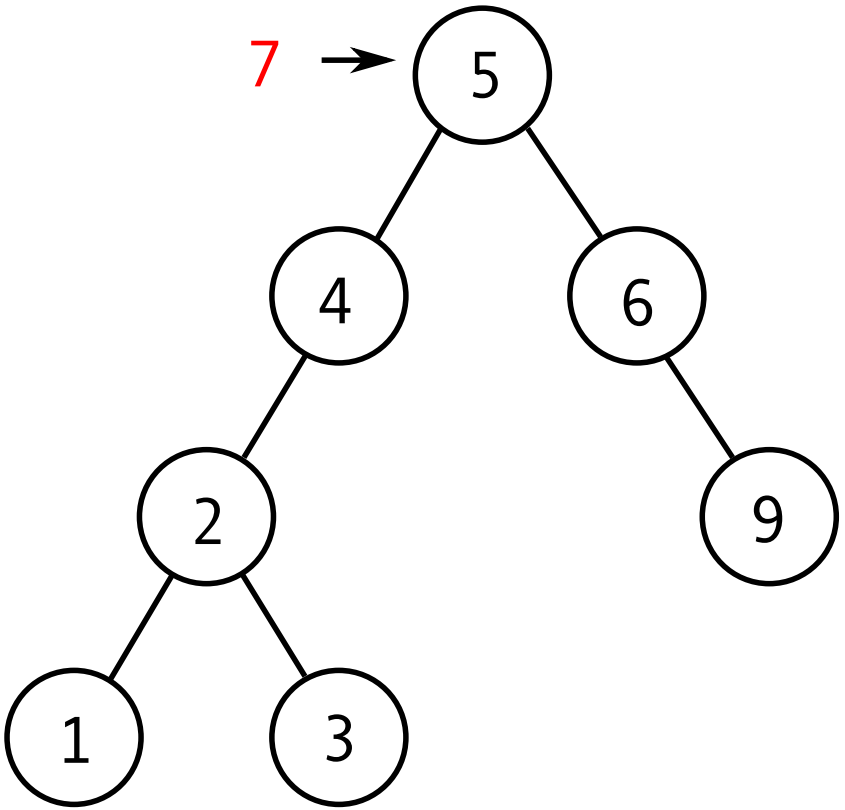
Binary search trees

To find a key, k, we start at the root, r. If the key of r is less than k, we take the right sub-tree, if it exists, otherwise we take the left. If the sub-tree we need does not exist, then k was not found. Otherwise, we keep following branches until we reach a leaf node, which either has a key equal to k or k was not found.

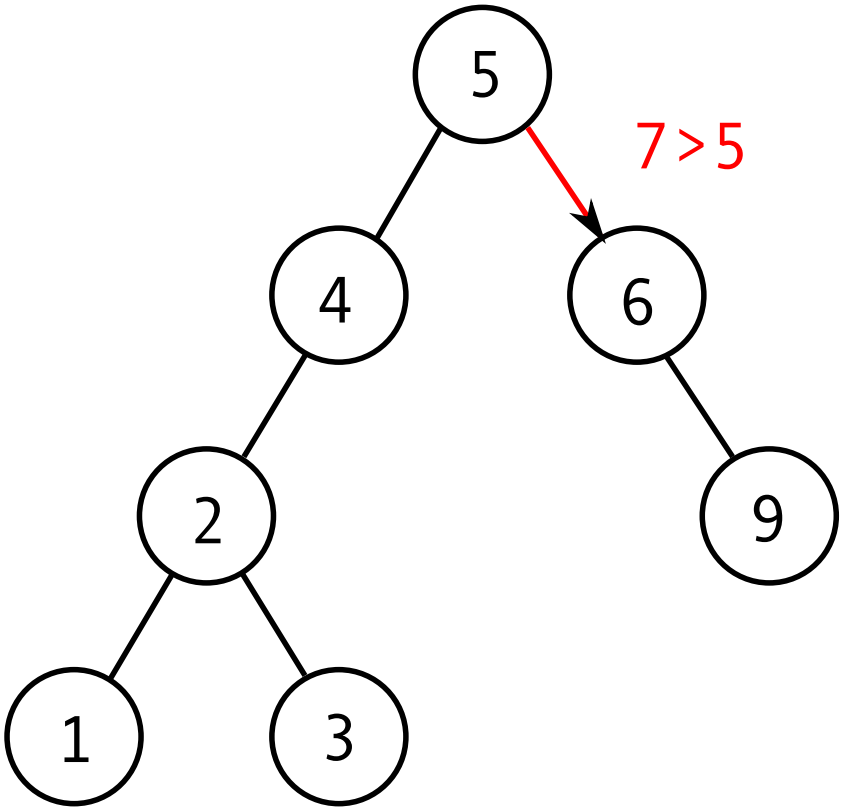


Inserting to a BST

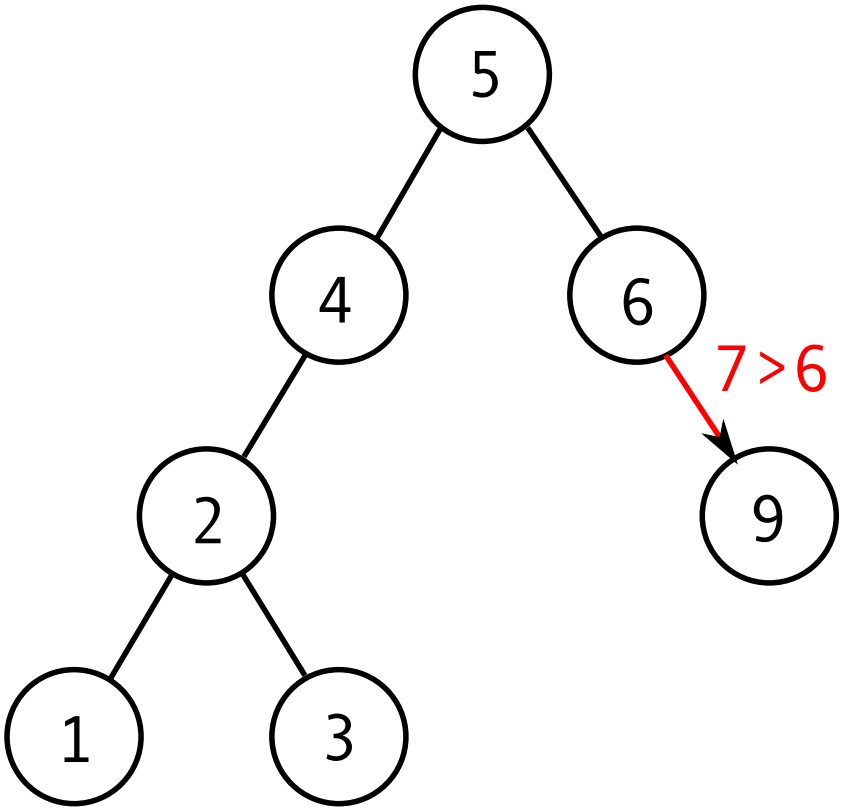
Inserting a new key, k, works similarly. We find the right place to put k by following branches until the sub-tree to follow does not exist, then we attach a new node with k as the label.



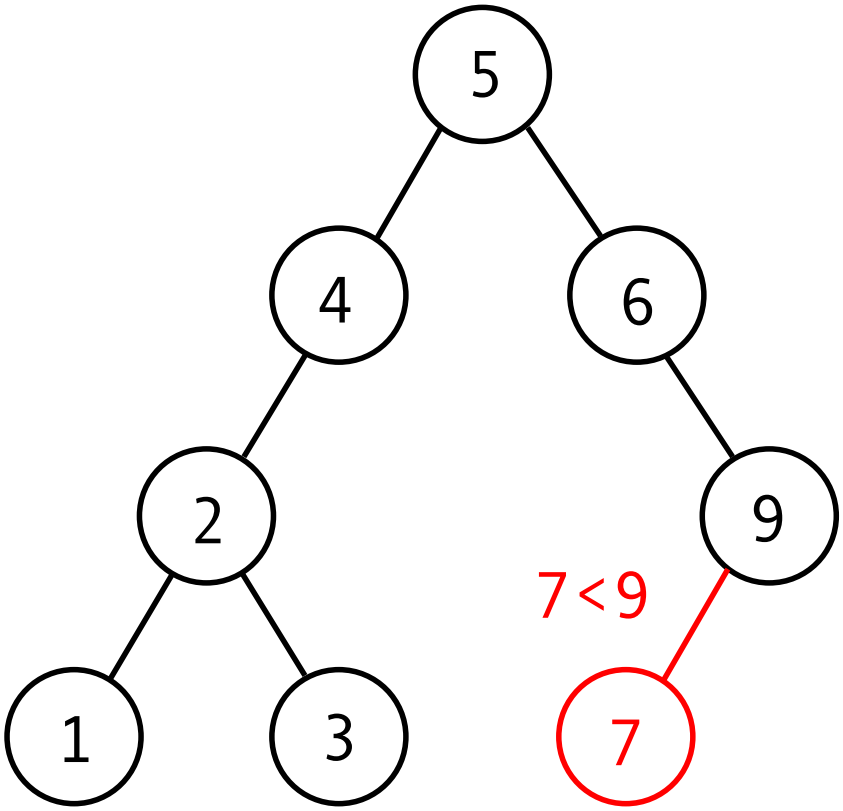
Inserting to a BST



Inserting to a BST

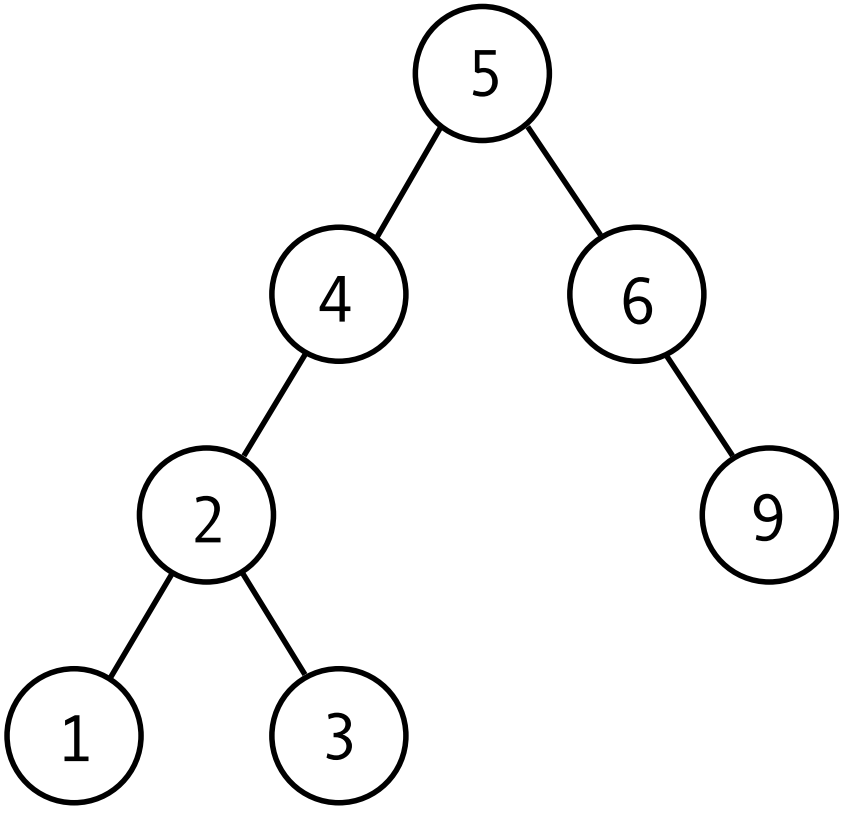


Inserting to a BST



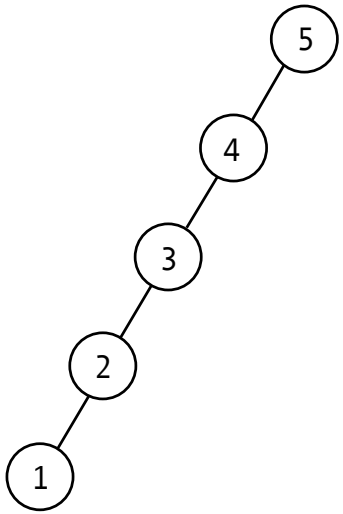
Binary search trees

Deleting a node is more tricky. Deleting a node with one or zero branches is easy enough, but to delete a node with two branches we need to merge the branches to produce a new node. (Details in a lab session coming soon!)



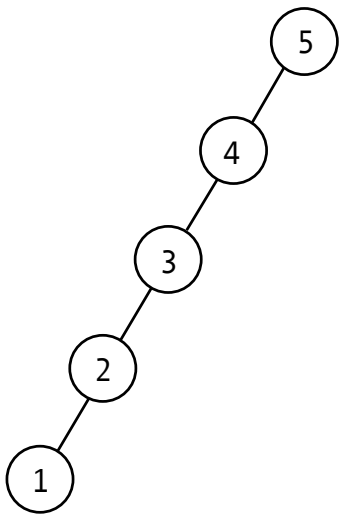
Balanced trees

If we insert random data into our trees, they will remain fairly well balanced. That is, the tree is as full as possible, or has the minimum number of missing branches. Then, each pair of left and right sub-trees will contain (approximately) the same number of nodes and the distance from the root to any leaf will be similar. If the input is not random, e.g. is in descending order, the tree will become unbalanced.



Balanced trees

In this case the tree has poor performance characteristics: search, insertion and deletion are all $O(n)$. The family of balanced trees are those where all operations on the tree maintain its balanced structure, requiring quite a lot of rearranging of nodes etc.



Balanced trees

If we can maintain the balance, search trees can be extremely efficient. If the tree is full then about half of all nodes are leaf nodes. On average, half of all searches will result in the need to traverse the tree all the way to a leaf.

In searching, we need to visit one node at each level. So, we can see how many steps a search will take by working out how many levels there are.

Balanced trees

Numbers of nodes and levels in a balanced tree:

Nodes	**Levels**
15	4
1023	10
32,767	15
1,048,575	20
33,554,432	25
1,073,741,824	30

Thus, we can find one of a million unique elements in about 20 steps (sound familiar?). A balanced tree with n nodes has $\lg(n+1)$ levels.

An imperative implementation

Implementing trees functionally gives a representation which seems "natural", but we can also implement them *imperatively*.

We can store the labels in an array, without managing links between them. Every possible node in the tree is represented by a position in the array, whether or not the node exists. The array positions that map to non-existent nodes contain `null`, or some special value.

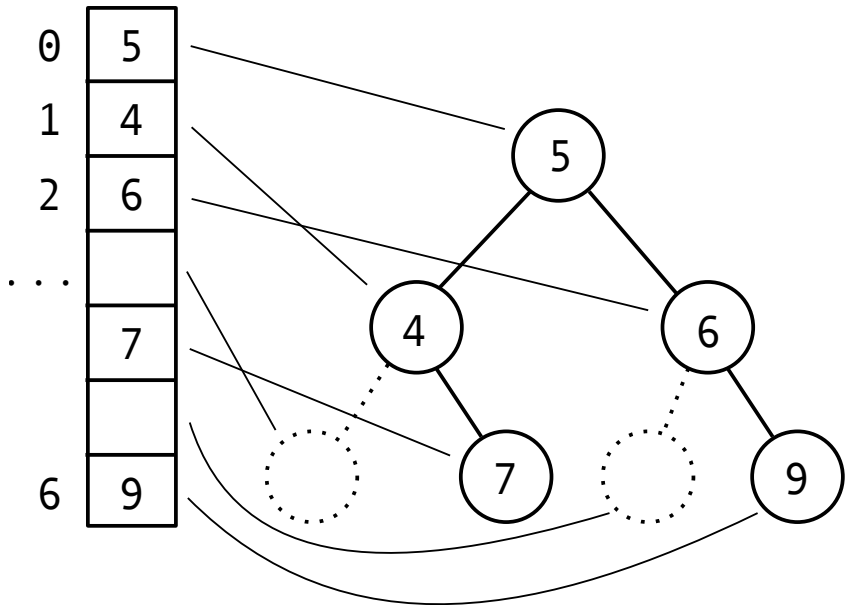
An imperative implementation

Using this scheme, we can find the child and parent nodes of an index, i, using arithmetic:

1. The left child of i is located at $2i + 1$.
2. The right child of i is located at $2i + 2$.
3. The parent of i is located at $\lfloor (i - 1)/2 \rfloor$.

An imperative implementation

Check for yourself that the formulae on the previous slide work.



Heaps

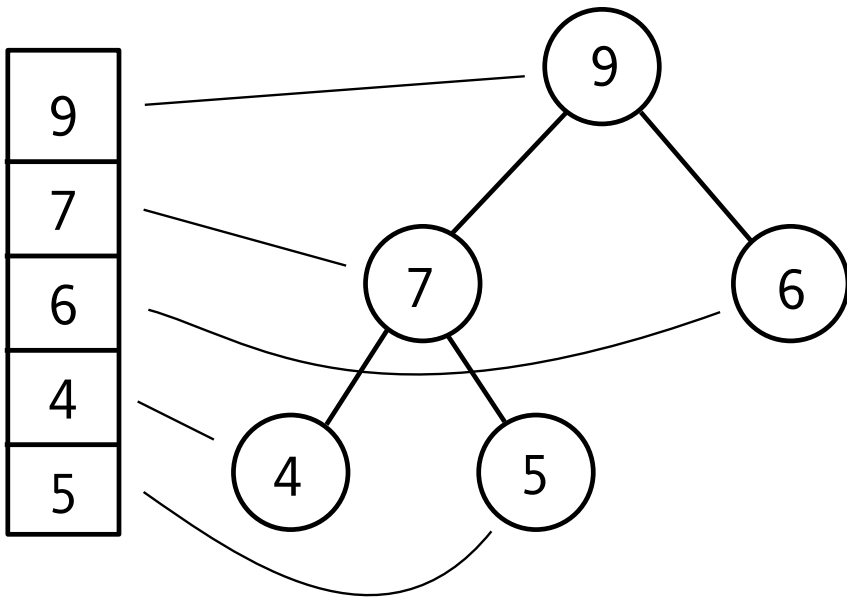
Implementing a tree as an array wastes space and deletion requires us to move every element, so it isn't normally the best choice. It does lend itself to one important application though: heaps. A heap is a binary tree with the following characteristics:

1. It is complete: every level except the last one is full and the last row has no gaps reading from left to right.
2. Each node satisfies the heap condition: its label is greater than or equal to the keys of its children.

Note that the invariants of the heap are weaker than that of the BST, but just strong enough to guarantee efficient insertion and removal.

Heaps

Any path through a heap gives an ordered list – descending in our case, but we could have arranged it the other way round. Because a heap is complete, no space is wasted in the array.



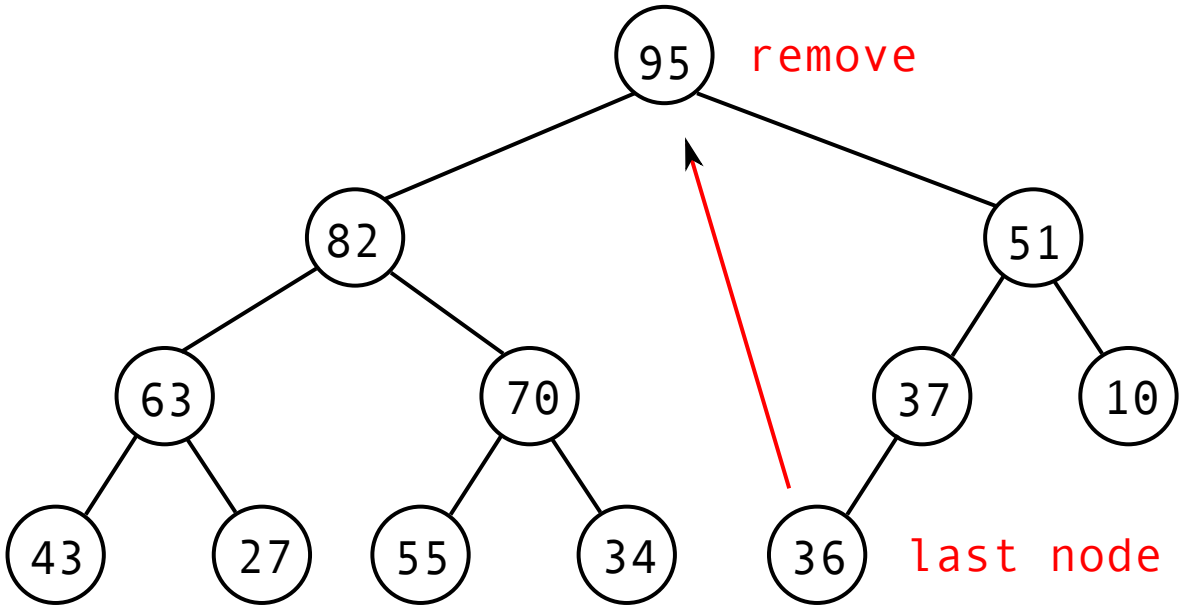
Heaps as priority queues

We can use the heap to model a priority queue, where the root is the front of the queue (or has the highest priority). When we remove the element at the front of the queue we need to restore the heap, making sure it is complete and satisfies the heap condition:

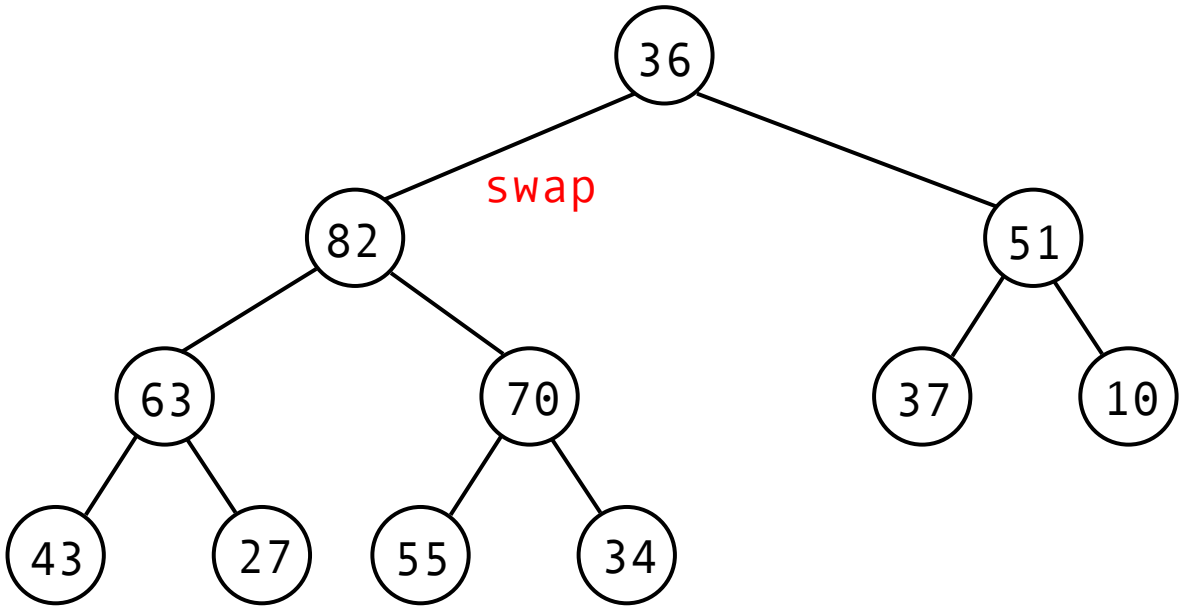
1. Remove the root node.
2. Move the *last* node to the root. The last node is the rightmost node on the lowest level.
3. Trickle down the new root until it's below a node larger than it and above a node less than it, if one exists.

Deleting from a heap

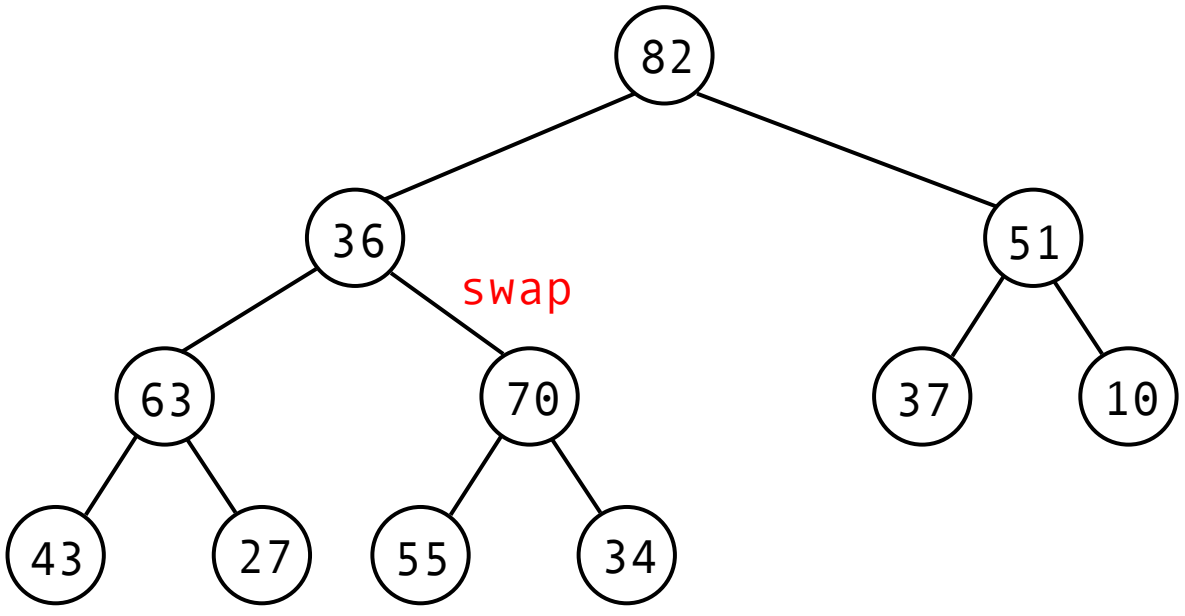
When trickling down, at each node we swap places with the *largest* child.



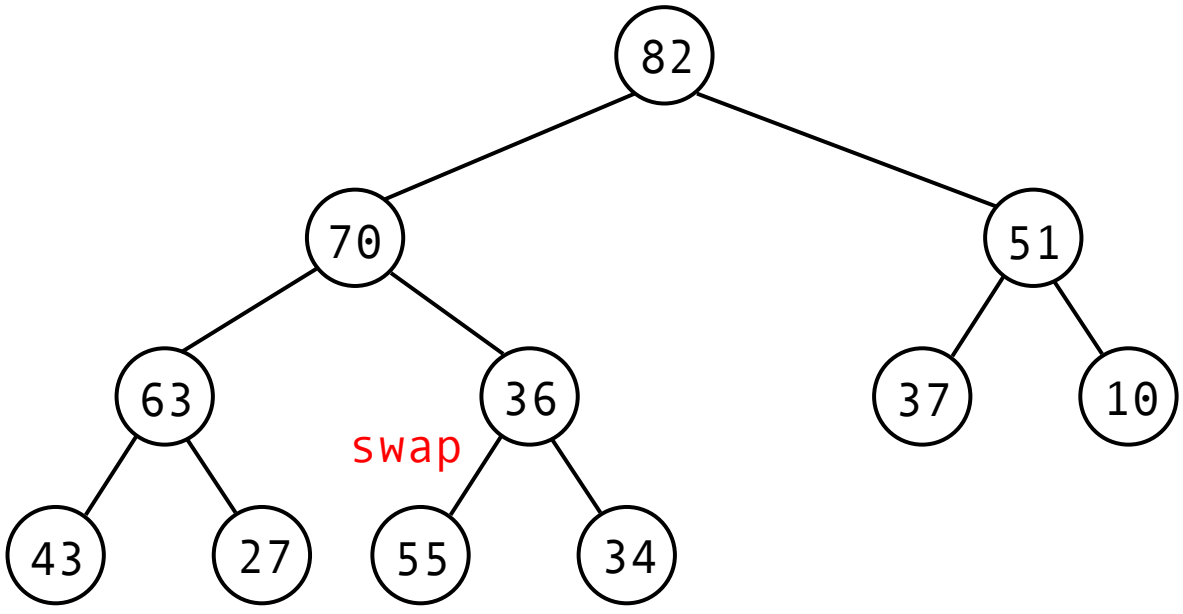
Deleting from a heap



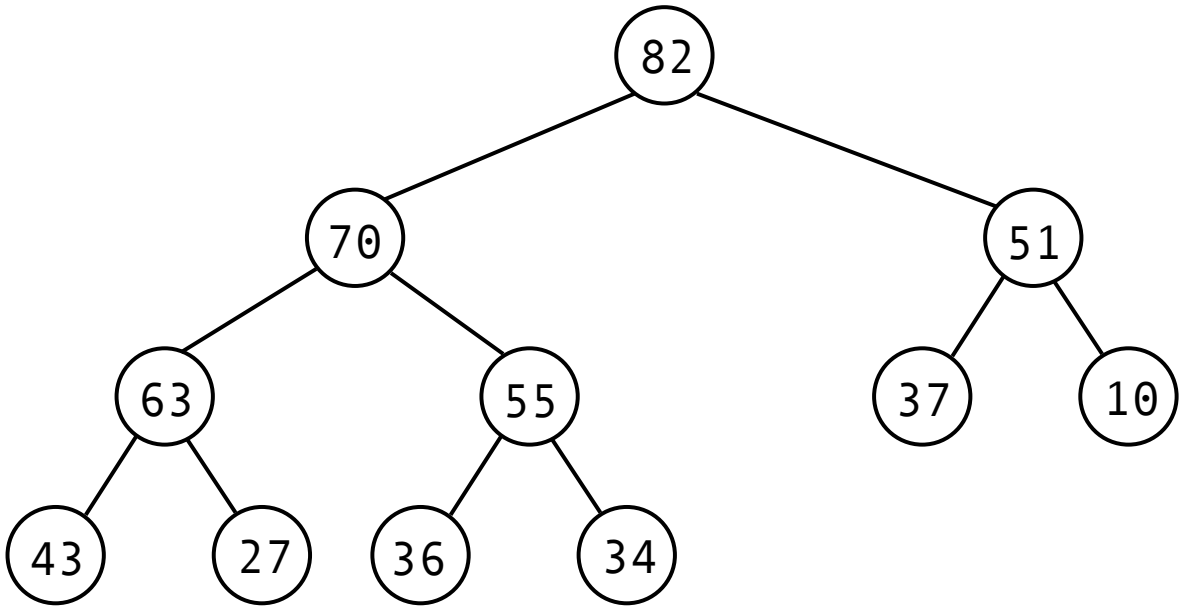
Deleting from a heap



Deleting from a heap



Deleting from a heap



Inserting to a heap

Inserting a new value to a heap is even easier: we put the new value in the first free position (starting a new level if necessary) and trickle up, swapping places with the parent, until the node is smaller than its parent.

Our heap has $\lg(n+1)$ levels, where n is the number of nodes. Insertion and deletion require visiting one node on every level (at worst), so both operations are $O(\log n)$.

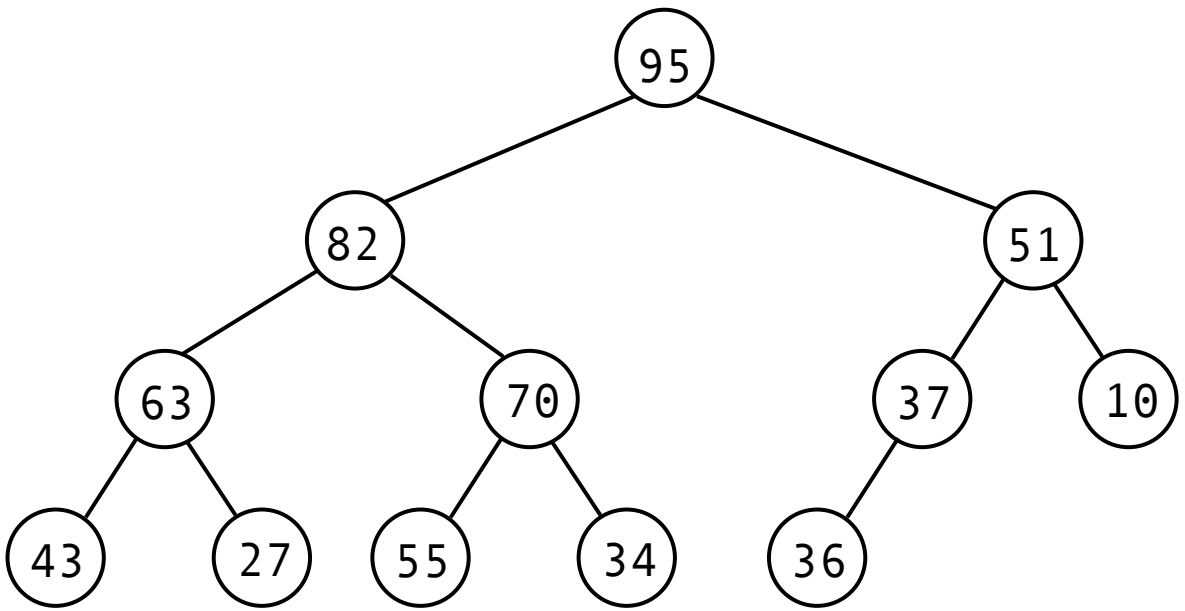
Heapsort

We can use heaps as the basis of an elegant and efficient sorting algorithm called heapsort. The idea is that we insert the unsorted values into a heap, then repeated applications of remove will give us a sorted collection.

```
1  for(int i=0;i<n;i++) {
2    theHeap.insert(theArray[i]);
3  }
4  for(int i=0;i<n;i++) {
5    array[i] = theHeap.remove();
6  }
```

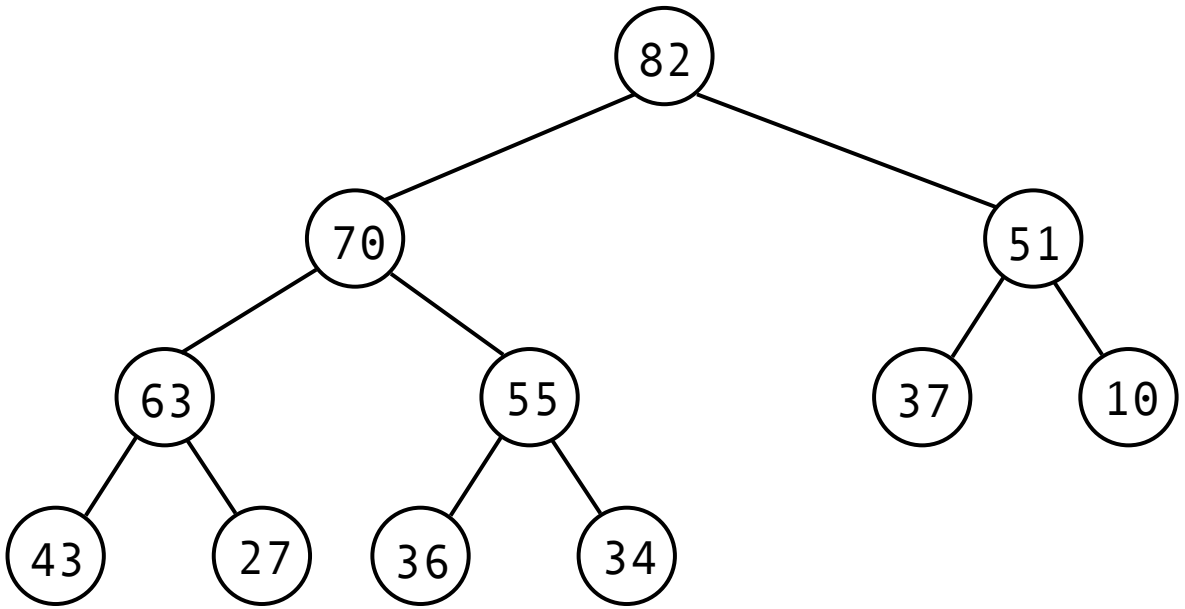
Heapsort

After inserting some unsorted data into a heap:



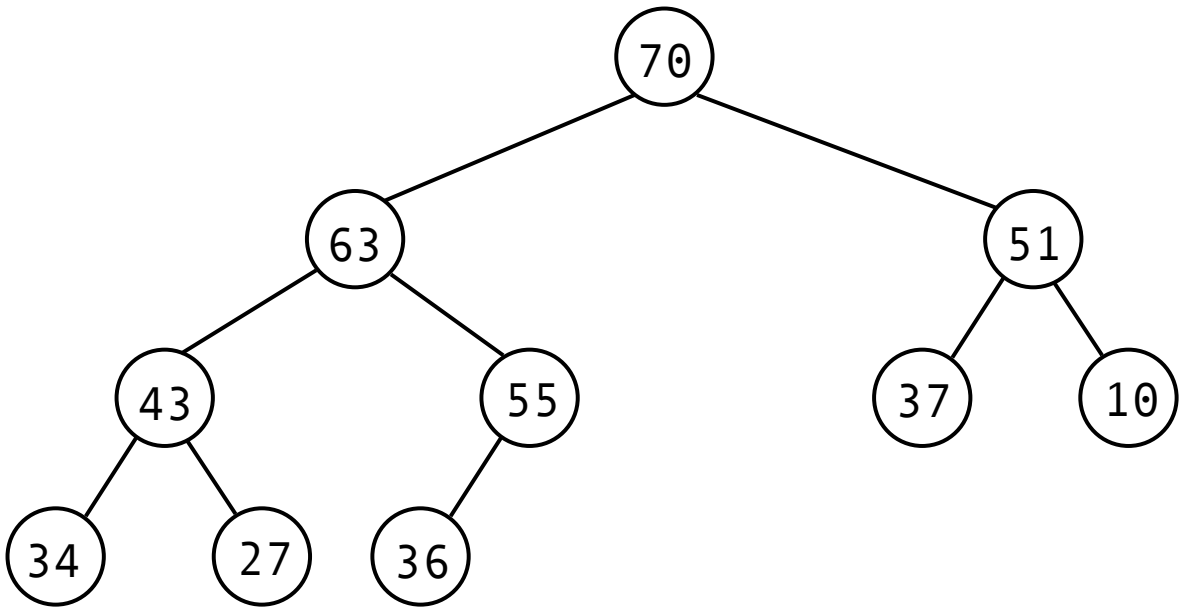
Heapsort

We remove the root repeatedly, restoring the heap condition each time.



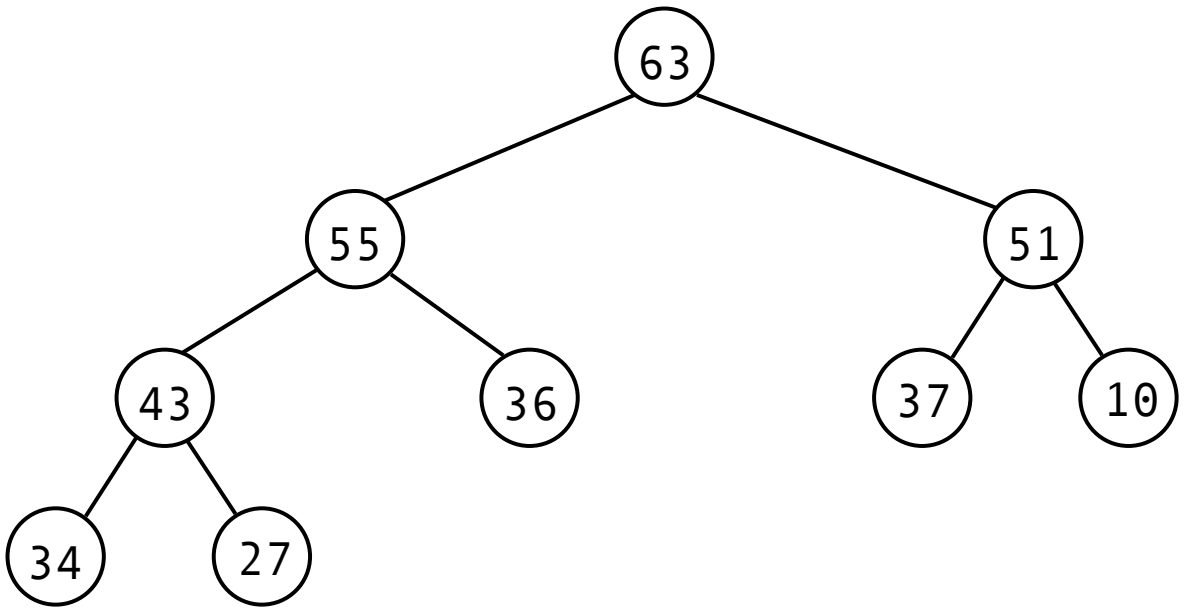
95

Heapsort



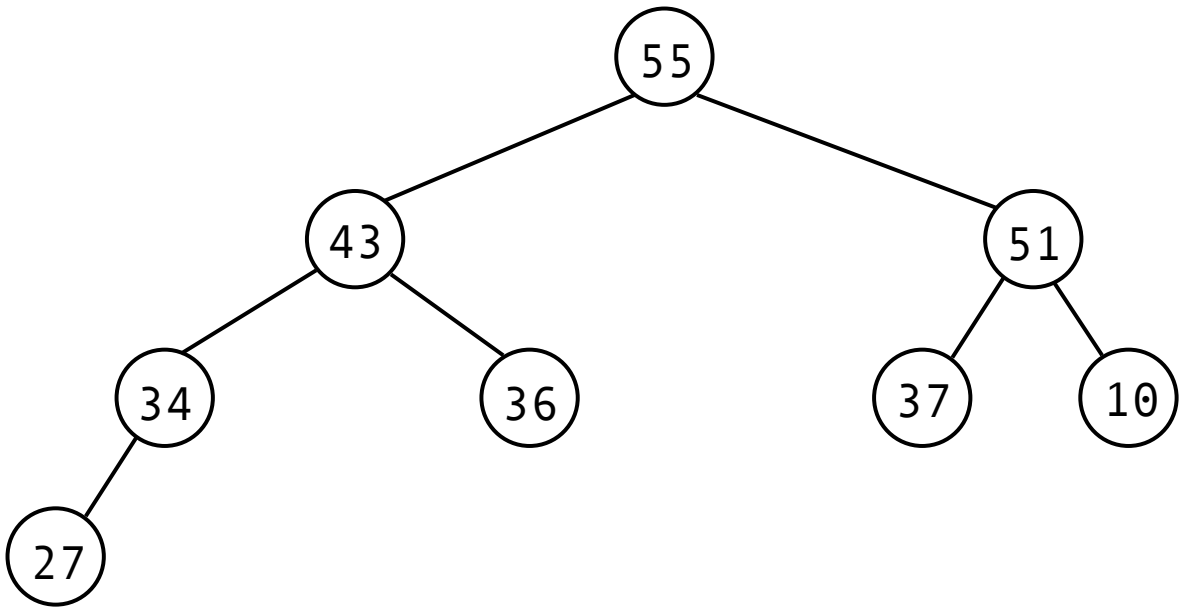
95 82

Heapsort



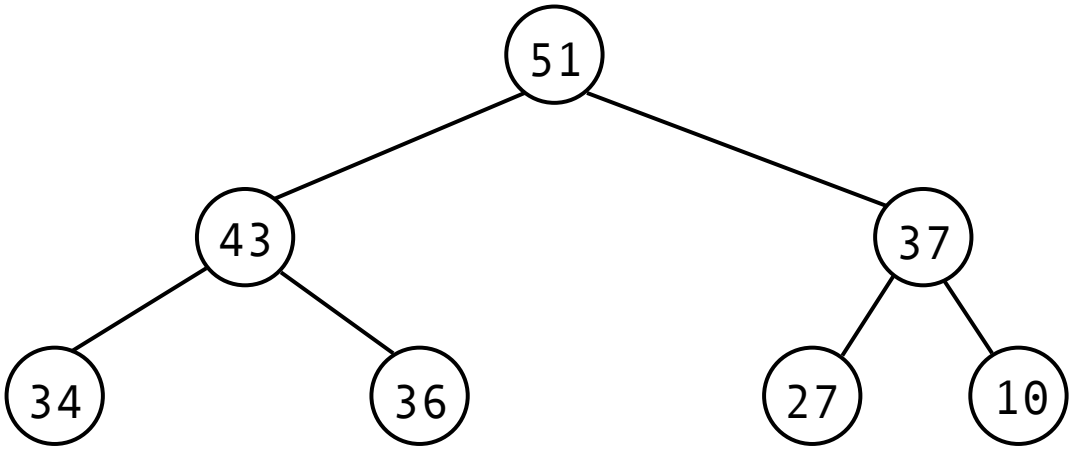
95 82 70

Heapsort



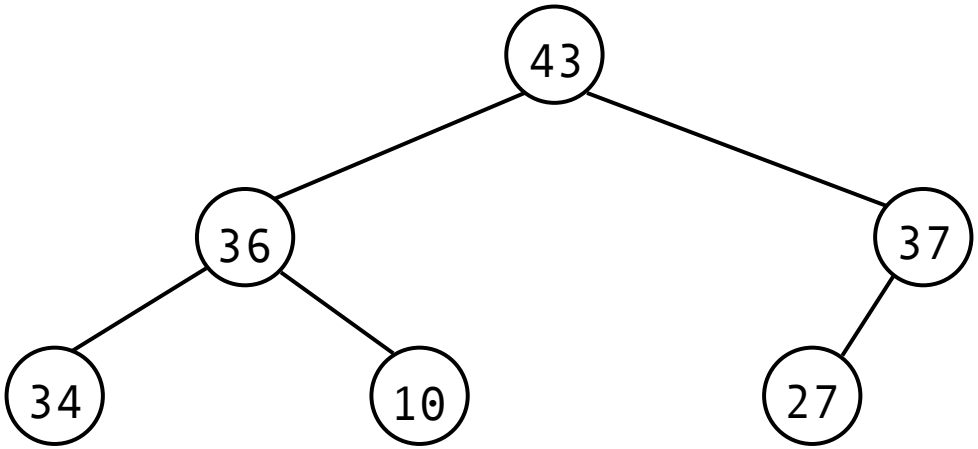
95 82 70 63

Heapsort



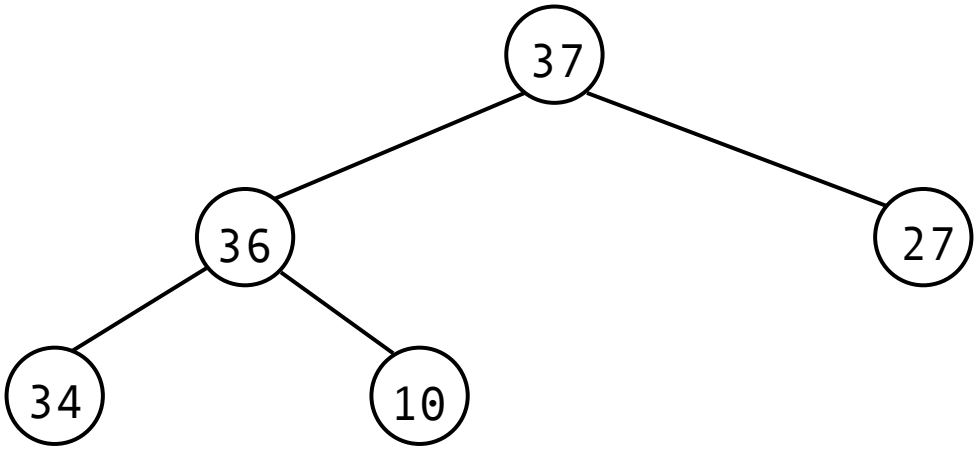
95 82 70 63 55

Heapsort



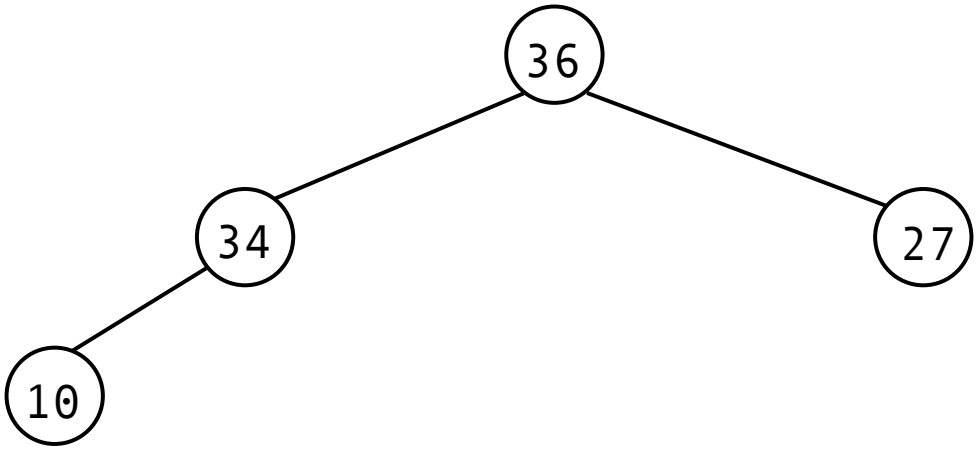
95 82 70 63 55 51

Heapsort



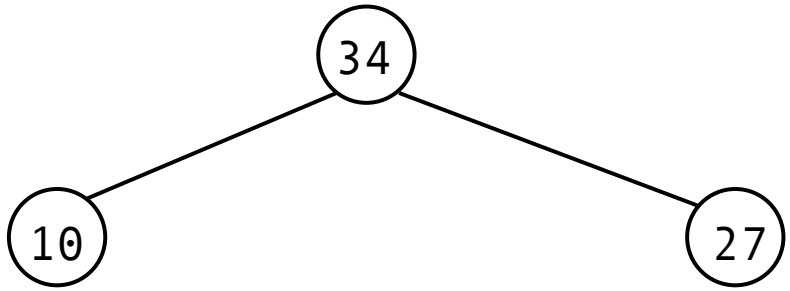
95 82 70 63 55 51 43

Heapsort



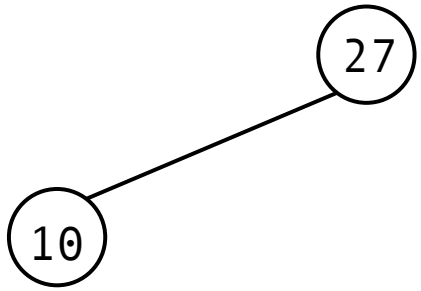
95 82 70 63 55 51 43 37

Heapsort



95 82 70 63 55 51 43 37 36

Heapsort



95 82 70 63 55 51 43 37 36 34

Heapsort

95 82 70 63 55 51 43 37 36 34 27

Heapsort

95 82 70 63 55 51 43 37 36 34 27 10

HeapSort

`insert` and `remove` are both $O(\log n)$ and each are performed n times, so heapsort is $O(n \log n)$ (loglinear).

Trickling up and down are quite expensive though, with lots of copying and swapping. Heapsort implementations apply optimisations such as reducing the number of swaps when trickling up, and allowing the heap to become temporarily disordered during a batch of insertions then restoring the heap condition in one go.

Next time

Balanced trees of various kinds.