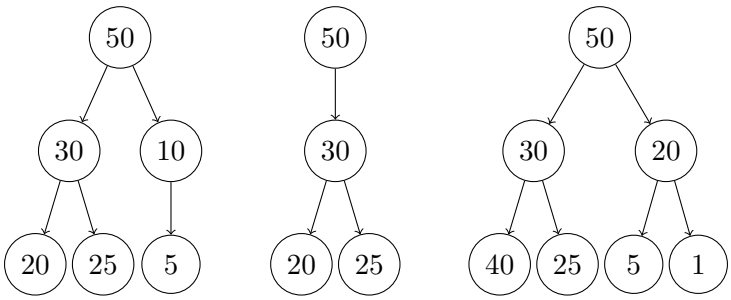

Figure 1: The first tree is a binary heap; it doesn't matter that the ten on the second row is less than elements on the third row, the defining propery is that nodes are bigger than their children and smaller than their parents. Similarly, it doesn't matter that the third row isn't complete, since this is the lowest row. The second tree isn't complete, there are gaps on the second row even though there are nodes on the third row. The third tree doesn't satisfy the heap property, the 40 is bigger than the 30 above it.

16 - binary heap

A binary heap is a data structure which allows elements to be easily added and removed while keeping track of the largest element. They are important because they are used to implement priority queues in scheduling, for example, in a router. We have actually already seen another potential application, in Dijskra's algorithm the lowest distance node is always evaluated next and a binary heap could be used to track which node is the lowest. Here, for definiteness, we will consider a tree designed to keep track of the highest node, rather than the lowest as in the the Dijstra example, but, of course, this isn't a significant difference.

A binary heap is a complete binary tree; a binary tree means, as ever, that all the nodes have up to two child nodes. A complete tree is one where all the layers except the lowest layer are full. A binary heap also has the *heap* property which states that every node is smaller than, or equal to, its parent. These definitions are illustrated in Fig. 1.

Of course, in a binary heap, because of the heap property, the root node is always the biggest element. It turns out that storing data in a binary heap is an efficient way to keep track of the biggest element if elements are continually being added and removed, and, in particular if the biggest element is often removed, as it is in a priority queue.

Of course, in a binary heap, because of the heap property, the root node is always the biggest element. It turns out that storing data in a binary heap is an efficient way to keep track of the biggest element if elements are continually being added and removed, and, in particular if the biggest element is often removed, as it is in a priority queue.

Let's first of all consider adding a new item to a binary heap. For neatness imagine we have the rule that we fill the lowest level from left to right. To add a new item the new node is inserted in the first available place and then swapped upwards until the heap property is restored. In other words, if the new node is larger than its parent it is swapped with its parent and this it continued, with the new node bubbling upwards, until the new node is smaller than its parent, or is the root. This is illustrated in Fig. 2. To create a heap out of a list of items, add the items to the list one-by-one using this algorithm, this is sometimes called *heapifying*.

Next lets consider removing the top node. If the top node is removed a new node needs to be

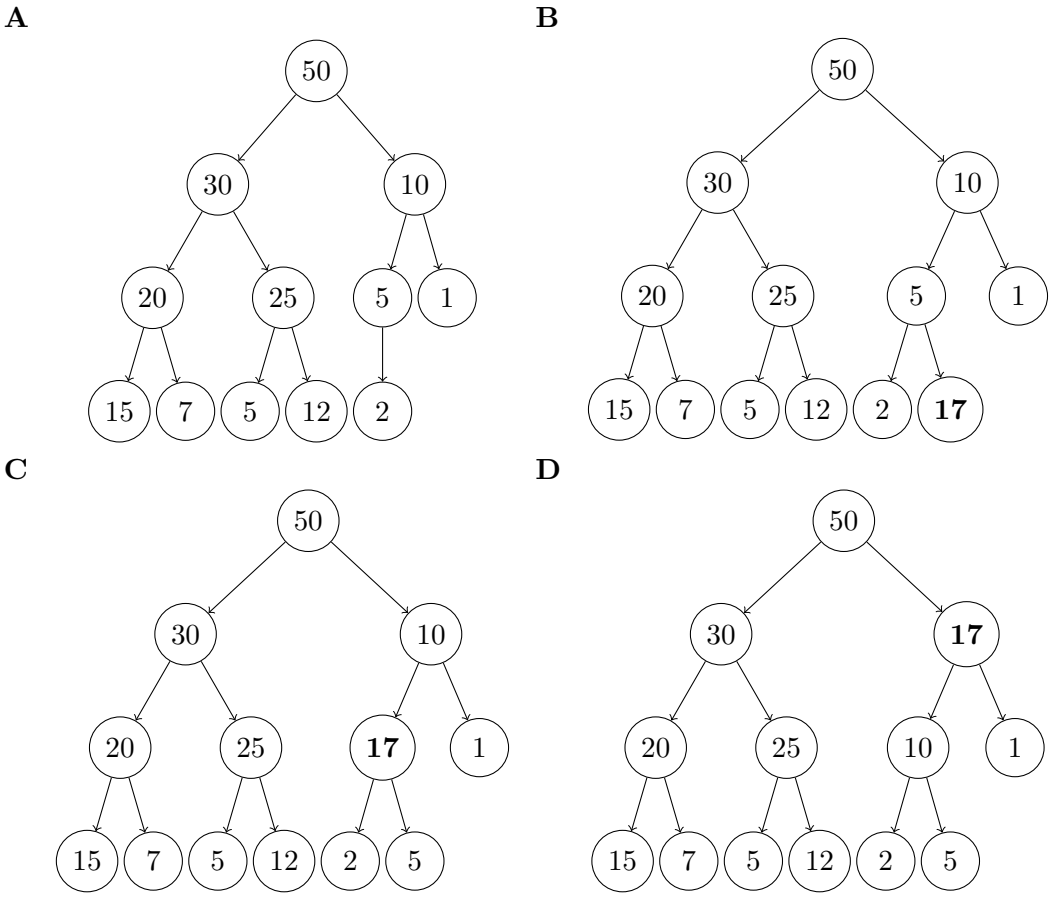

Figure 2: **A** shows the tree before the new node is added; in **B** the new node is added to the first available slot and in **C** and **D** it bubbles upwards to its correct position.

put in its place; to maintain the completeness property this is the last node on the lowest layer, the location the last node was added. This node is moved to the top. Now the heap property has to be re-implimented by swapping this node down to its correct place, to do this, as long as it is smaller than either of its children it is swapped with the larger of its two children. An example is shown in Fig. 3.

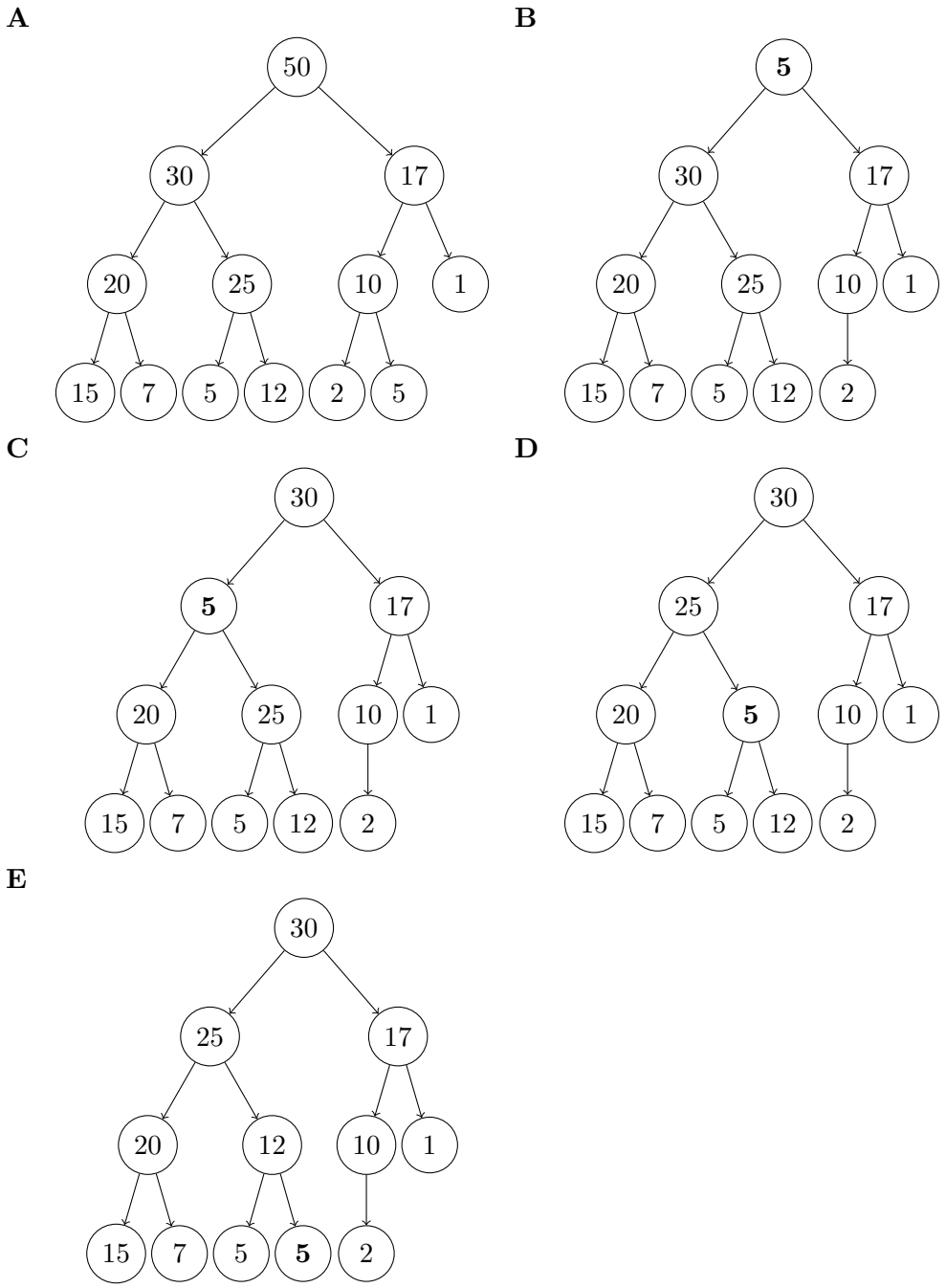

Figure 3: **A** shows the tree before the root node is deleted; in **B** the node is removed and replace with the five from the lowest layer and in **C** to **E** this five sinks back downwards to its correct position.