FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Data structures
Treaps

1 Treaps

A disadvantage of binary search trees is that the height of the tree depends on the order in which items were added to the tree. In the worst case, the tree collapses into a linked list which causes the operations to cost linear time instead of the optimal logarithmic time.

A Treap is a randomized binary search tree that uses randomness to try to keep the height of the tree proportional to the logarithm of the number of elements it contains. There is still no guarantee however.

Just like a normal binary search tree, the elements are stored in the nodes. For an internal node, we can find all the child nodes with a smaller key in the left sub tree and all the child nodes with a larger key in the right sub tree. In addition to the key value, each node also stores a priority value. This value is picked randomly when the node is created. In addition to the BST property, a Treap also maintains a heap property on the priority values: the priority for any non-leaf node must be greater than or equal to the priority of its children. This means that the node with the highest priority can be found in the root and this is true for every sub tree. You can find an example of a Treap in Figure 1. Here the green values represent the keys and the blue values represent the priorities.

The operations on a Treap are a combination of BST operations and operations on a heap:

- **Search**: Exactly the same as on a BST, the priorities are simply ignored.
- **Add**: The priority is picked at random. The node is inserted in the tree just like it would be inserted in a binary search tree. Finally, the heap condition is checked. The priority of the element is compared with the priority of the parent node and if the heap condition is not met, a rotation is performed that switches the role of the parent and the child. When performing the rotation, we have to make sure that the BST property is still intact. Figure 2 shows the two options for the rotation. If the node (B) is the left child, we have to perform a right rotation. The node (B) moves up and the parent (A) becomes the right child. The right child (T2) of the original node (B) becomes the left child of the original parent node (A). Similarly, if the node (B) is the right child, we have to perform a left rotation. This is repeated until the heap condition is satisfied, in the worst case, the new node has to be rotated all the way up to the root.

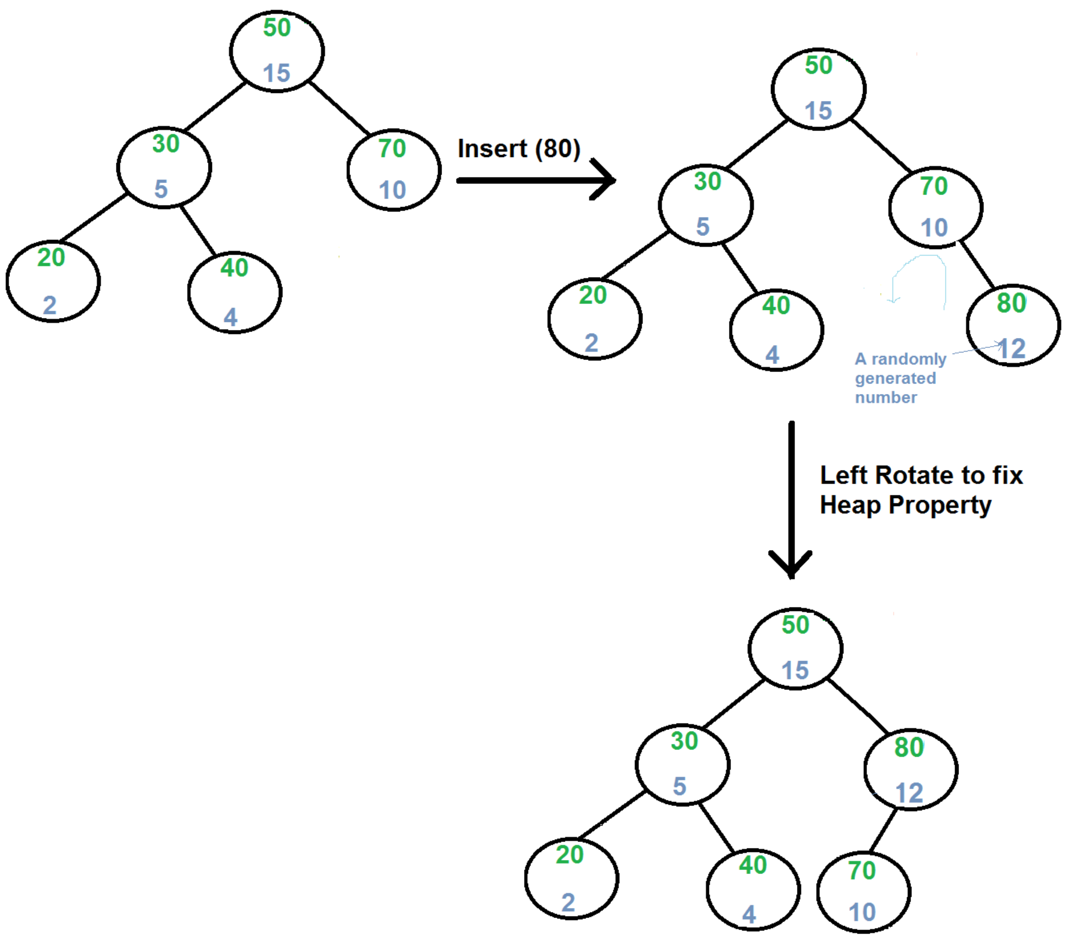

Figure 1: An example Treap.

2 Assignment

Implement the *add* method in the *Treap* class. No changes are needed outside this method. The main program contains code to add elements to the treap, to check the structure of the Treap and to create a visual representation. An example output is shown below. For each node we show the value with the priority between brackets. Note that the structure of the tree will depend on the randomly chosen priorities.

```
Height: 5

                         7 (98)
              /                            \
             3 (96)                          8 (93)
       /           \                           \
      1 (77)         6 (59)                      9 (10)
 /          \       /
0 (24)        2 (45)   4 (41)
                         \
                          5 (11)
```

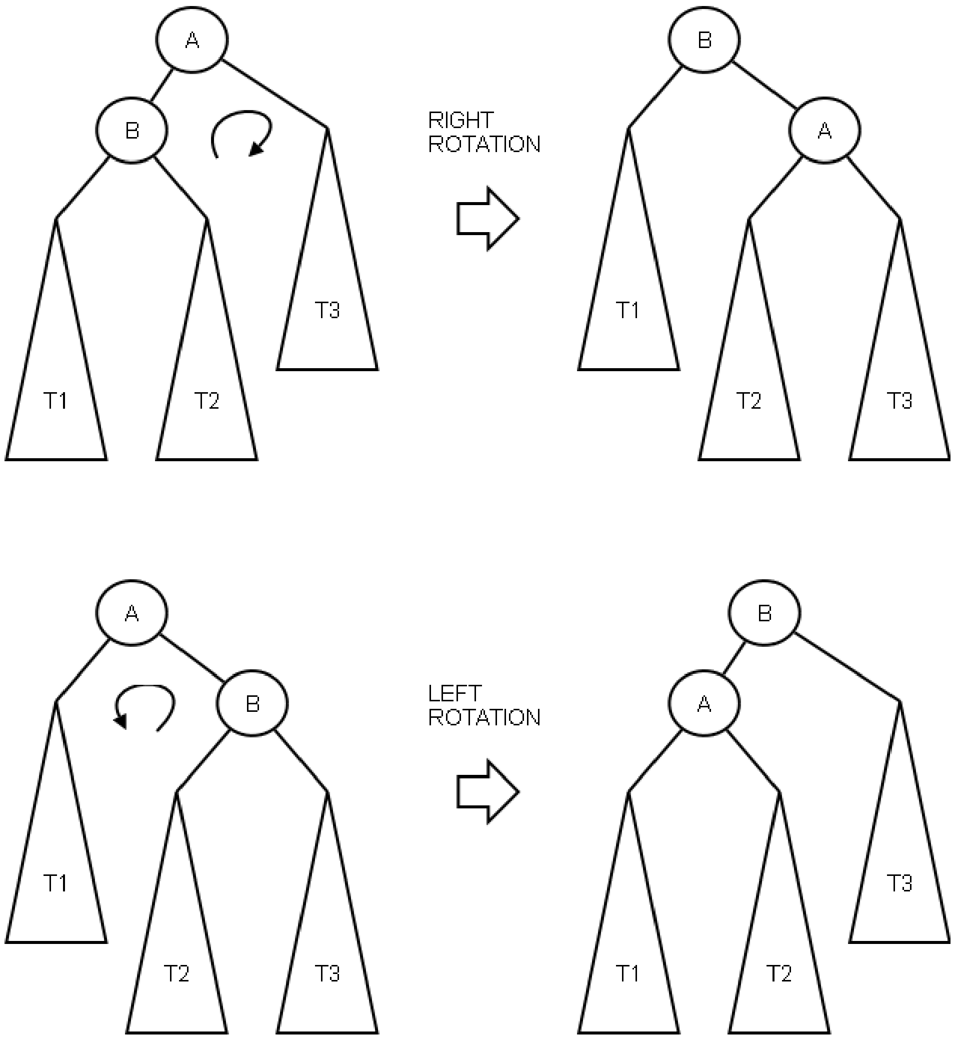

Figure 2: Left and right rotations.