Advantages:

In such implementation, X values are the keys (and at the same time the values stored in the treap), and Y values are called priorities. Without priorities, the treap would be a regular binary search tree by X. One set of X values could correspond to a lot of different trees, some of them degenerate (for example, in the form of a linked list), and therefore extremely slow (the main operations would have *O(N)* complexity).

At the same time, priorities allow uniquely specifying the tree that will be constructed (of course, it does not depend on the order in which values are added), which can be proven using the corresponding theorem. If you choose the priorities randomly, you will get non-degenerate trees on average, which will ensure *O(logN)* complexity for the main operations. Hence another name for this data structure - is the randomized binary search tree.

With treap, we can keep dynamic arrays as we can search the k-th value in an array at any time in O(log n). All the operations stated before are made in O(log n) (except rotations which are made in O(1)).

Uniqueness:

Each subtree is a treap of size k-1, so they are unique by induction.

Major Operations:

There are some several operations that can be done with Treap:

§ Insert (X, Y) in *O(logN)*.

§ Search (X) in *O(logN)*.

§ Erase/delete (X) in *O(logN)*.

§ Build (*X1*, ..., *XN*) in *O(N)*.

§ Union (*T1*, *T2*) in *O(Mlog(N/M))*.

§ Intersect (*T1*, *T2*) in *O(Mlog(N/M))*.

§ Split (T, X)

§ Merge (*T1*, *T2*)

§ Security reasons. A treap can not contain information on the history.

§ Efficient sub-tree sharing. The fastest algorithms for set operations I have seen use treaps

Implicit Treaps

Implicit treap is a simple modification of the regular treap which is a very powerful data structure. In fact, implicit treap can be considered as an array with the following procedures implemented (all in O(logN) in the online mode):

Inserting an element in the array in any location

Removal of an arbitrary element

Finding sum, minimum/maximum element, etc. on an arbitrary interval

In addition, painting on an arbitrary interval

Reversing elements on an arbitrary interval

The idea is that the keys should be indices of the elements in the array. But we will not store these values explicitly (otherwise, for example, inserting an element would cause changes of the key in O(N) nodes of the tree).

Note that the key of a node is the number of nodes less than it (such nodes can be present not only in its left subtree but also in the left subtrees of its ancestors). More specifically, the implicit key for some node T is the number of vertices cnt(T→L) in the left subtree of this node plus similar values cnt(P→L)+1 for each ancestor P of the node T, if T is in the right subtree of P.

Now it's clear how to calculate the implicit key of the current node quickly. Since in all operations we arrive at any node by descending in the tree, we can just accumulate this sum and pass it to the function. If we go to the left subtree, the accumulated sum does not change, if we go to the right subtree it increases by cnt(T→L)+1.