The Rank Tree

A *rank tree* is a data structure that stores a sequence of elements, indexed from 0 to $n-1$, and that supports the following operations:

- Construct a rank tree from an array with n elements;
- `find(k)` returns the item at *rank* (index) k;
- `remove(k)` removes the item at rank k;
- `len` returns the current size (number of elements).

The rank tree stores the elements in a binary tree, one element per node. The indexed sequence is the *in-order sequence* of this tree. A node v also stores the *size of its subtree*, that is, the number of nodes in the subtree whose root is v. The `Node` class looks like this:

```
class _Node():
  def __init__(self, element, size=1, left=None, right=None):
    self.element = element
    self.size = size
    self.left = left
    self.right = right

  def recompute_size(self):
    ls = self.left.size if self.left else 0
    rs = self.right.size if self.right else 0
    self.size = ls + 1 + rs
```

Its `recompute_size()` method updates the `size` field based on the `size` fields of the children.

Fig. 1 shows a rank tree storing the sequence $1, 2, 3, \ldots, 12$. The red numbers are the `size` field of each node.

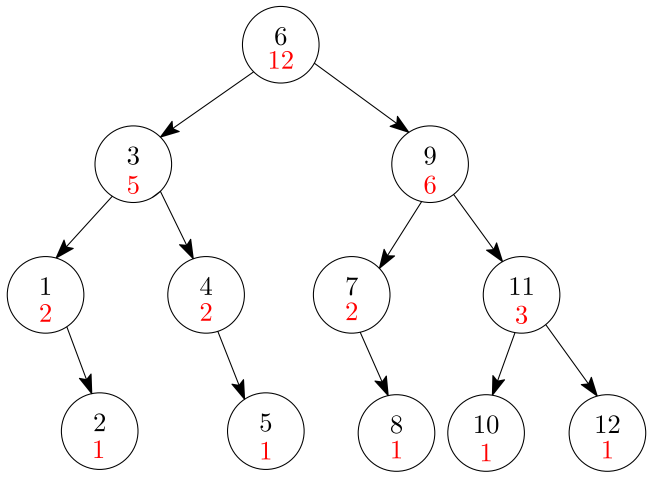


Figure 1: Black numbers are elements, red numbers indicate the `size` field of each node.

Size method. The rank tree's `__len__` method can be implemented easily: it just returns the size field of the root note. Its running time is $O(1)$:

```
  def __len__(self):
    return self._root.size if self._root else 0
```

Find method. The `find(k)` method follows a path starting at the root and ending at the node with index k. We implement this method recursively, using the method `_find(k)` of the `Node` class. It returns the node at index k *inside the subtree with root* `self` (so the indexing starts with the leftmost leaf of this subtree).

In in-order ordering, the index of the root node is equal to the size of its left subtree. So if `k == left.size`, the node we are looking for is the root itself. If `k < left.size`, the node we are looking for must be in the left subtree, where it is again at index `k`. In the last case, the node we are looking for is in the right subtree, and its index in the right subtree is `k - left.size - 1`, because all the elements of the left subtree and the root itself appear first in the sequence.

```
def _find(self, k):
  ls = self.left.size if self.left else 0
  if k < ls:
    return self.left._find(k)
  elif k == ls:
    return self
  else:
    return self.right._find(k - ls - 1)
```

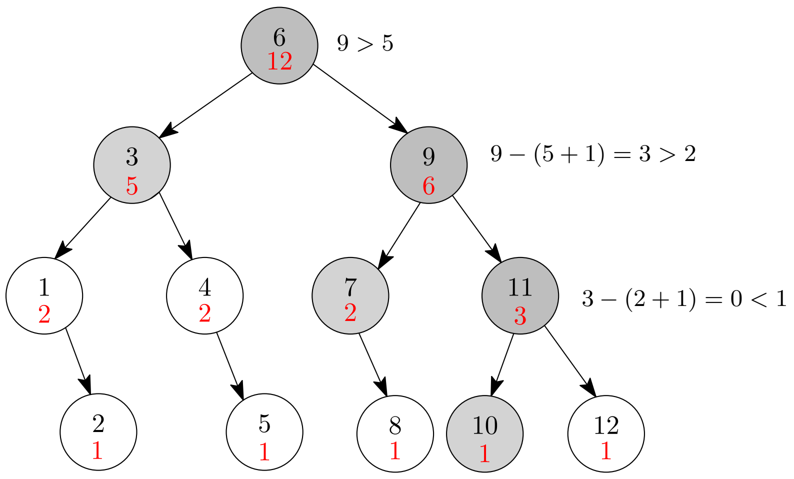


Figure 2: To execute `find(9)`, we visit the dark nodes, and check the lightly shaded nodes to obtain the size of the left subtree.

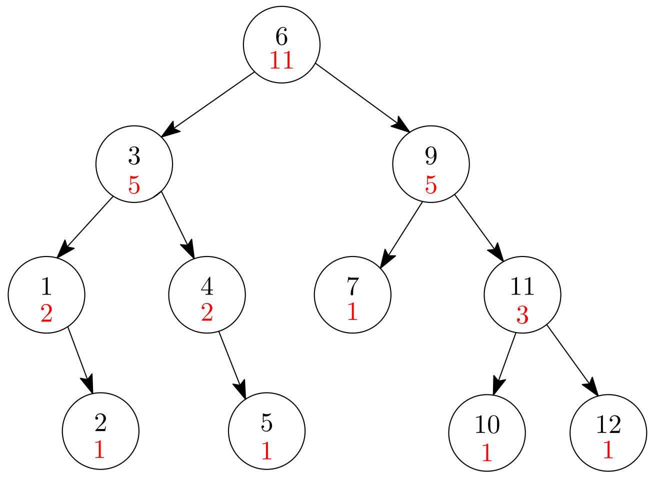
The running time of this method is clearly $O(h)$, where h is the height of the tree. (Recall that the *height* of a tree is the length of the longest path from the root to a leaf.)

Remove method. Let v be the node at index k. We distinguish three cases:

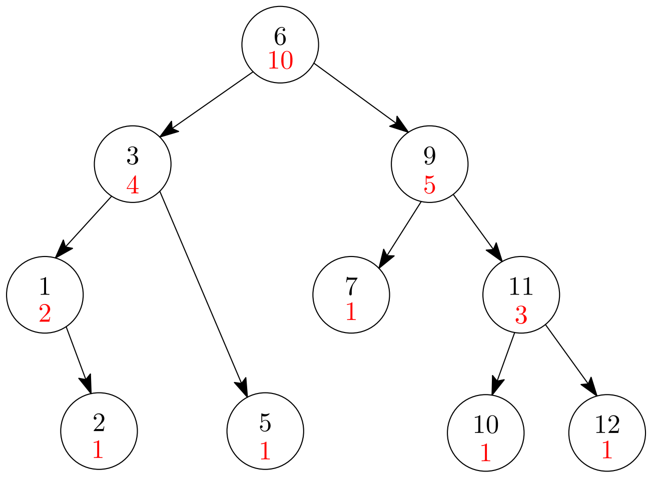
If v is a leaf, then we can remove it easily: We set the link from its parent to `None`. Fig. 3(a) shows the result of `remove(7)` on the tree of Fig. 2.

If v has one child, then we can replace the link from v's parent to v by a link from the parent to v's child. Fig. 3(b) shows the result of `remove(3)` on the previous tree. The link from element 3 to 4 has been replaced by a direct link from 3 to 5.

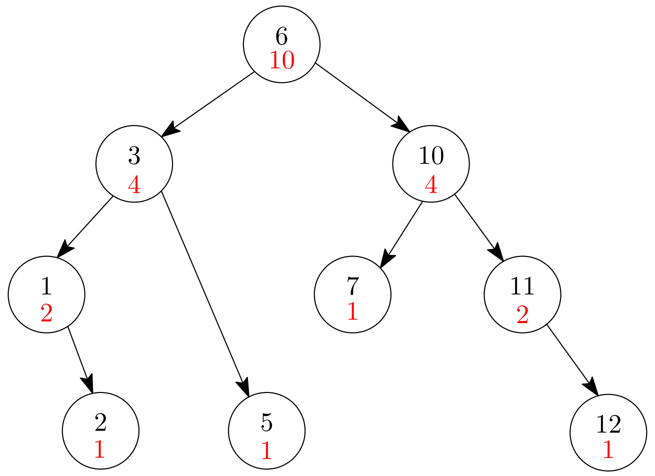
Finally, the hardest case is when v has two children. In this case we cannot remove the node v without seriously changing the structure of the tree. So we use a little trick: Instead of removing the node v at rank k, we remove the node u at rank $k+1$. So instead of deleting the element at index k from the sequence, we first swap it with the element to its right, and then delete the element at index $k+1$.



(a) After deleting the node with rank 7



(b) After deleting the node with rank 3



(c) After deleting the node with rank 6

Figure 3: Delete a node with given rank k

The good news is that the node u at rank $k+1$ is always the leftmost node in the right subtree of v. Since u is the leftmost node, it cannot have a left child. As a result the node u can be removed from the tree easily using one of the methods above.

To summarize: To remove v, we find the leftmost node u in v's right subtree, we transfer the element from node u to node v, and then we remove node u from the tree.

Consider, for instance, executing `remove(6)` on the tree of Fig. 3(b). The node v with rank 6 is the node containing the element 9. We instead find the leftmost node u of its right subtree—that is the node u containing the element 10. We transfer the element 10 into node v, and then remove node u, resulting in the tree of Fig. 3(c).

Removing a node will make some of the `size` fields in the tree invalid. More precisely, the `size` fields of all nodes that are ancestors of the deleted node have to be recomputed. We achieve this using recursion.

The running time of `remove(k)` is clearly $O(h)$, since it follows a path from the root to v, and then possibly further down to u. The total length of this path cannot be larger than h.

Rank tree construction. Finally, we need to show how to construct the initial rank tree. The constructor of `RankTree` should immediately build the tree from a given array:

```
class RankTree():
  def __init__(self, a):
    self._root = self._build_tree(a, 0, len(a) - 1)
```

We want the tree to be nicely balanced, so we need the left and right subtree of the root to be roughly equal in size. This immediately give us an algorithm for `_build_tree`: For a sequence with zero elements, we create an empty tree. Otherwise, we pick the element with the middle index and put it in the root. Then we recursively build a rank tree for the left half of the sequence, and for the right half of the sequence. These trees become the children of the root node:

```
  def _build_tree(self, a, lo, hi):
    if hi < lo:
      return None
    elif lo == hi:
      return _Node(a[lo])
    else:
      mid = (lo + hi) // 2
      lhs = self._build_tree(a, lo, mid - 1)
      rhs = self._build_tree(a, mid + 1, hi)
      t = _Node(a[mid], 1, lhs, rhs)
      t.recompute_size()
      return t
```

The running time of this procedure is $O(n)$. The easiest way to see this is to observe that `_build_tree` spends constant time if we do not count the recursive calls. We can "charge" this constant time to the element at rank `mid`. No element is charged twice, so the total running time is $O(n)$.

Alternatively, we can write the running time $T(n)$ of `_build_tree` for a sequence of length n using the recursive formula $T(n) = 2T(n/2) + O(1)$. The solution of this formula is again $O(n)$.

It remains to observe that the construction step builds a perfectly balanced tree of height $\lceil \log(n+1) \rceil - 1$. Since our rank tree does not have an insertion method, the height of the tree can never increase. As a result, we always have $h = O(\log n)$.

Analysis. We found the following running times (where h is the height of the tree):

- Constructing the tree: $O(n)$
- `find(k)`: $O(h) = O(\log n)$
- `remove(k)`: $O(h) = O(\log n)$
- `size()`: $O(1)$

We will later see how to add insertions to this tree that are also guaranteed to be done in time $O(\log n)$.