```python
    def find_pairs(node, path):
    count = 0
    for i in range(len(path)):
        ancestor = path[i][0]
        direction = path[i][1]
        if direction == 'right' and node.key < ancestor.key:
            count++
        if direction == 'left' and node.key > ancestor.key:
            count++
    return count + find_pairs(node.left, path.append([node, 'left'])) +
        find_pairs(node.right, path.append([node, 'right']))
```

```cpp
template <class KeyType, class ValueType>
class Node { // variables could be public, or could add getters/setters
    int numberKeys;
    KeyType key[2]; // space for up to 2 keys
    ValueType value[2]; // space for the corresponding values
    Node* subtree[3]; // pointers to the up to 3 subtrees
}
```

Inserting a new key:

- Search for the key. If found, deal with duplicates. Otherwise, we've found the leaf.
- Insert the key into the leaf. If the leaf had only one key, we're done.
- Otherwise, fix the new leaf:
- To fix a node with 3 keys `[a b c]`

  1. If there is a parent, move `b` into it.
  2. If there is not a parent, make a new root for the tree, assign it height of `[a b c].height + 1`, move `b` into it.
  3. Create a node with just `a` as the key and make it the left child of the `b` node.
  4. Create a node with just `c` as the key and make it the left child of the `b` node.

Deleting a key:

1. Search for the key (call this the `target`).
2. If the `target` is in a leaf, move to step 5.
3. If the `target` is not in a leaf, search for its `successor` in a leaf.
4. Swap the `target` key with its `successor` key. (Leave the heights in the original nodes).
5. Now that the `target` key is in a leaf, delete it.
6. Now that the `target` key has been deleted, call `fix_empty()` on the leaf.
7. If the leaf still has a key, `fix_empty()` will do nothing. Otherwise it will propagate the changes needed to fix the tree.
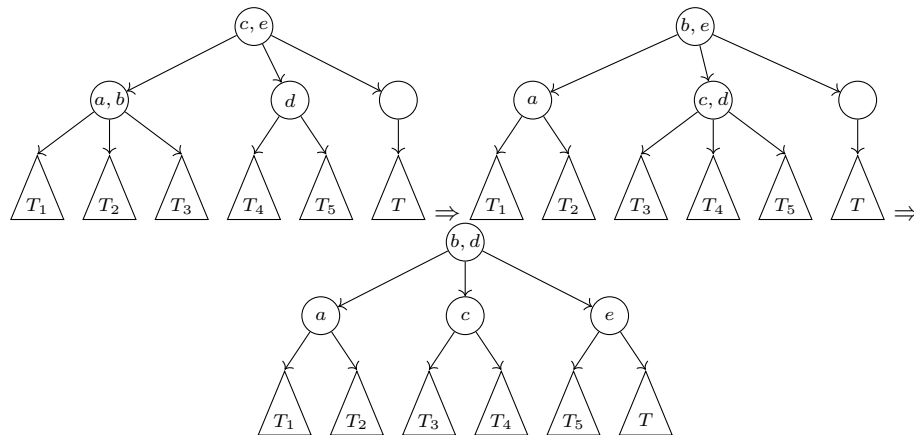
`fix_empty()`:

1. If the node still has a key, do nothing.
2. If the empty node has a sibling with two keys, perform a rotation in order to borrow a key from the sibling.
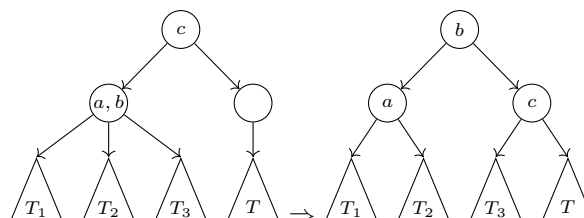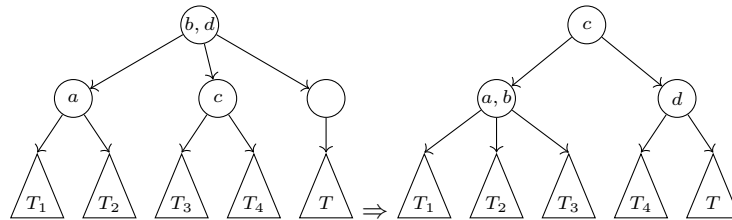
In this instance don't adjust the heights.
3. If the empty node has two siblings and one has two keys, it's similar. Either the same thing can be done (if the sibling is adjacent) or, if it's two hops away:

4. If no sibling of the empty node has two keys, then the next case is that the empty node has two siblings. In that case the empty node borrows from the parent:

5. Lastly, if the node has only one sibling with only one key, the parent is merged with the sibling into a node with two keys, and the empty is pushed up:
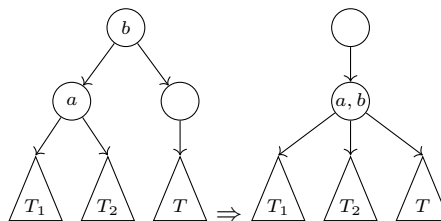
6. In each case, only swap key values. Keep heights the same.

K-sorting an array:

```
ksorted(A[], k):
    B = new 23tree()
    for i from 0 to k:
        B.insert(A[i])
    for i from 0 to n − k:
        A[i] = B.extractMin()
        B.insert(A[i+k])
    for i from n−k to n−1:
        A[i] = B.extractMin()
    return A[]
```

2

Other Stuff:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1; \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

$$y = \log_b x \rightarrow x = b^y$$

$$\log_b b = 1$$

$$\log_b b^x = x$$

$$b^{\log_b x} = x$$

$$\log_b(x^r) = r \log_b x$$

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y$$

$$a^n a^m = a^{n+m}$$

$$(a^n)^m = a^{nm}$$

$$\frac{d}{dx}(a^x) = \ln a$$

$$(fg)' = f'g + fg'$$

$$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$

$$\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$$

$$\frac{d}{dx}(e^{g(x)}) = g'(x)e^{g(x)}$$

$$\frac{d}{dx}(\ln g(x)) = \frac{g'(x)}{g(x)}$$