```
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']))
```

```
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:

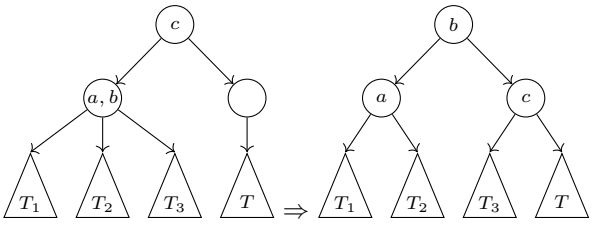
- Search for the key. If found, deal with duplicates. Otherwise, we've found the leaf.
- Insert the key into the leaf. If the leaf had only one key, we're done.
- Otherwise, fix the new leaf:
- To fix a node with 3 keys `[a b c]`
    1. If there is a parent, move `b` into it.
    2. If there is not a 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.

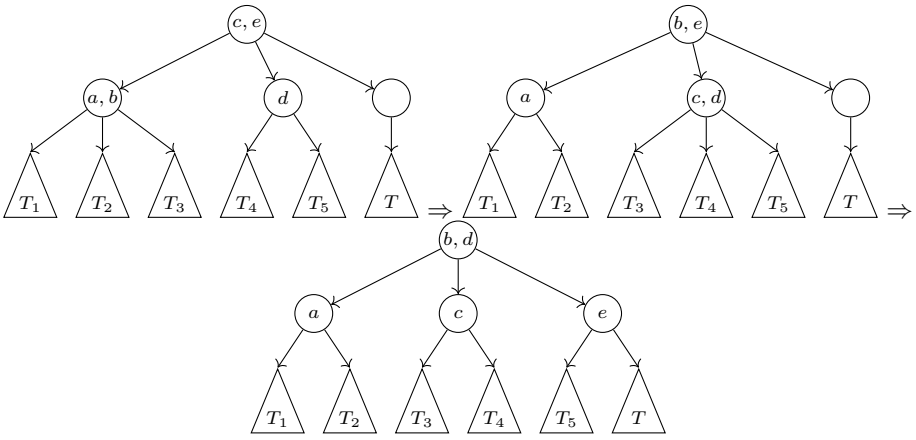
`fix_empty()`:

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.

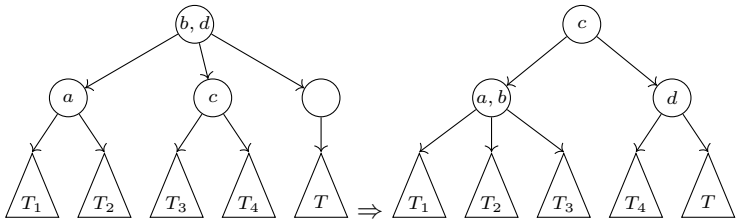


In this instance don't adjust the heights.

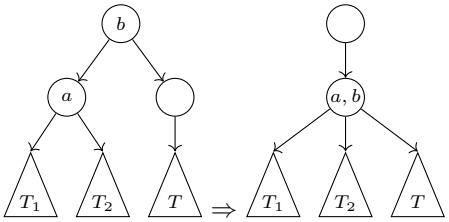
3. If the empty node has two siblings and one has two keys, it's similar. Either the same thing can be done (if the sibling is adjacent) or, if it's two hops away:



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[]
```

Insertion Sort:

```
Insertion_Sort(A):
    for j = 2 to A.length:
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Loop Invariant: At the start of each iteration of the for loop of lines 1-8, the subarray `A[1..j-1]` consists of the elements originally in `A[1..j-1]`, but in sorted order.

Running Time: $O(n^2)$

MergeSort:

```
MergeSort(A):
    n = len(A)
    if n > 1:
        left = A[0..n/2]
        right = A[(n/2)+1..n-1]
        lsort = MergeSort(left)
        rsort = MergeSort(right)
        return Merge(lsort, rsort)
    return A

Merge(A,B):
    i = 0
    j = 0
    C = []
    A[len(A)] = infinity
    B[len(B)] = infinity
    while A[i] < infinity or B[j] < infinity:
        if B[j] < A[i]:
            C[i + j] = B[j]
            j += 1
        else:
            C[i + j] = A[i]
            i += 1
    return C
```

QuickSort:

```
QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q - 1)
        Quicksort(A, q + 1, r)

Partition(A, p, r):
    x = A[r]
    i = p - 1
```

```
    for j = p to r - 1:
        if A[j] <= x:
            i += 1
            swap(A[i], A[j])
    swap(A[i + 1], A[r])
    return i + 1
```

Heaps:

```
Build_Max_Heap(A):
    A.heap-size = A.length
    for i = A.length/2 downto 1:
        Max_Heapify(A,i)

Max_Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l <= A.heap-size and A[l] > A[i]:
        largest = l
    else:
        largest = i
    if r <= A.heap-size and A[r] > A[largest]:
        largest = r
    if largest != i:
        swap(A[i], A[largest])
        Max_Heapify(A, largest)

Parent(i):
    return i/2

Left(i):
    return 2*i

Right(i):
    return 2*i + 1

Insert(A, k):
    A.heap-size += 1
    A[A.heap-size] = -infinity
    Heap_Increase_Key(A, A.heap-size, key)

Heap_Increase_Key(A, i, key):
    A[i] = key
    while i > 1 and A[Parent(i)] < A[i]:
        swap(A[i], A[Parent(i)])
        i = Parent(i)

Extract_Max(A):
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size -1
    Max-Heapify(A,1)
    return max
```

BFS:

```
BFS(G,s):
    Q = queue()
    dist = [infinity]
    pred = [null]
    color = [white]
    color[s] = gray
    dist[s] = 0
    Q.enqueue(s)
    while !Q.is_empty():
        u = Q.dequeue()
        color[u] = black
        for v in N(u):
            if color[v] == white:
                color[v] = gray
                pred[v] = u
                dist[v] = dist[u] + 1
                Q.enqueue(v)
```

BFS Theorem 1: When `BFS(s)` terminates, $\forall v \in V$ `dist[v]` is the shortest path distance from s to v. Furthermore: `v, pred[v], pred[pred[v]],...,s` is such a shortest path in reverse.

BFS Theorem 2: If vertices s and t are connected, when `BFS(s)` terminates, the color of t is black. If s and t are not connected, when `BFS(s)` terminates, the color of t is white.

BFS Theorem 3: (main loop invariant) At any stage of the algorithm `BFS(s)` at step 4, the queue Q contains vertices in at most two adjacent levels l and $l+1$.

1. All vertices in level $< l$ are black
2. All vertices in level l that are not gray are black
3. All vertices in level $l+1$ that are not gray are white
4. All vertices in level $> l+1$ are white
5. In Q, all vertices in level l are ahead of those in level $l+1$

BFS Theorem 4: When `BFS(s)` terminates, $\forall v$, `dist[v]` $= \delta_s(v)$.

Djikstra:

```
Djikstra(G,s):
    dist = [infinity]
    pred = [null]
    S = []
    dist[s] = 0
    Q = priority_queue(all vertices)
    while !Q.is_empty():
        u = Q.extract_min()
        S.append(u)
        for v in N(u):
            Relax(u,v)
            Q.decrease_key(v)
```

```
Relax(u,v):
    if dist[v] > dist[u] + w(u,v):
        dist[v] = dist[u] + w(u,v)
        pred[v] = u
```

Running Time: $O((m+n)\log n)$

Theorem: When Djikstra's algorithm terminates, $\forall v$, `dist[v]` $= \delta_s(v)$

Bellman-Ford:

```
Bellman-Ford(G, s):
    dist = [infinity]
    pred = [null]
    dist[s] = 0
    for u in G.V:
        for v in N(u):
            Relax(u,v)
```

Running time: $O(mn)$

DFS:

```
DFS(G):
    color = [white]
    pred = [null]
    disc = [null]
    finish = [null]
    time = 0
    for u in G.V:
        DFS_Visit(u)

DFS_Visit(u):
    color[u] = gray
    disc[u] = time
    time += 1
    for v in N(V):
        if color[v] == white:
            pred[v] = s
            DFS_Visit(v)
    color[u] = black
    finish[u] = time
    time += 1
```

Running Time: $O(m+n)$

Parentheses Theorem: In any DFS forest, for vertices u, v either:

1. $[d[u], f[u]]$ is disjoint from $[d[v], f[v]]$ and neither is the descendant of the other or
2. $[d[u], f[u]]$ contains $[d[v], f[v]]$ and v is a descendant of u

White Path Theorem: v is a descendant of $u \iff$ there is a white path from u to v when `DFS_Visit(u)` is called.

Topological Sort:

Theorem: Perform `DFS(G)`:

(1) G is acyclic $\iff$ there is no back edge

(2) If G is acyclic, then decreasing order of finish times is a topological order

Other Stuff:

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

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

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

$\log_b b = 1$

$\log_b b^x = x$

$b^{\log_b x} = x$

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

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

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

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

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

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

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

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

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

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

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