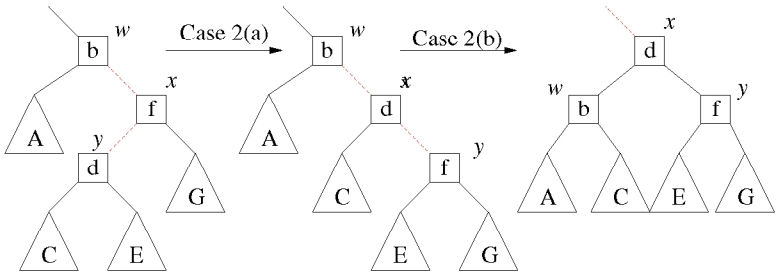

Figure 87: Case 2.

Lecture X02: BB-trees

Reading: This is not covered in our readings.

BB-trees: We introduced B-trees earlier, and mentioned that they are good for storing data on disks. However, B-trees can be stored in main memory as well, and are an alternative to AVL trees, since both guarantee $O(\log n)$ wost case time for dictionary operations (not amortized, not randomized, not average case). Unfortunately, implementing B-trees is quite a messy programming task in general. BB-trees are a special binary tree implementation of 2-3 trees, and one of their appealing aspects is that they are very easy to code (arguably even easier than AVL trees).

Recall that in a 2-3 tree, each node has either 2 or 3 children, and if a node has j children, then it has $j - 1$ key values. We can represent a single node in a 2-3 tree using either 1 or 2 nodes in a binary tree, as illustrated below.

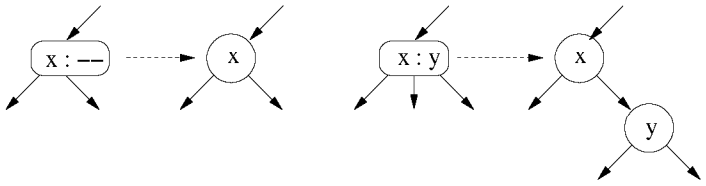

Figure 88: Binary representation of 2-3 tree nodes.

When two nodes of a BB-tree are used to represent a single node of a 2-3 tree, this pair of nodes is called *pseudo-node*. When we represent a 2-3 tree using this binary representation, we must be careful to keep straight which pairs of vertices are pseudo-nodes. To do this, we create an additional field in each node that contains the *level* of the node in the 2-3 tree. The leaves of the 2-3 tree are at level 1, and the root is at the highest level. Two adjacent nodes in a BB-tree (parent and right child) that are of equal level form a single pseudo-node.

The term "BB-tree" stands for "binary B-tree". Note that in other textbooks there is a data structure called a "bounded balance" tree, which also goes by the name BB-tree. Be sure you are aware of the difference. BB-trees are closely related to red-black trees, which are a binary representation of 2-3-4 trees. However, because of their additional structure the code for BB-trees is quite a bit simpler than the corresponding code for red-black trees.

As is done with skip-lists, it simplifies coding to create a special *sentinel* node called `nil`.

Rather than using `null` pointers, we store a pointer to the node `nil` as the child in each leaf. The level of the sentinel node is 0. The left and right children of `nil` point back to `nil`.

The figure below illustrates a 2-3 tree and the corresponding BB-tree.

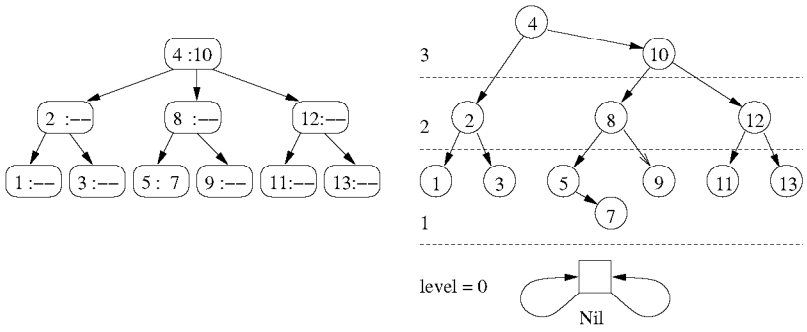

Figure 89: BB-tree corresponding to a 2-3 tree.

Note that the edges of the tree can be broken into two classes, *vertical* edges that correspond to 2-3 tree edges, and *horizontal* edges that are used to join the two nodes that form a pseudo-node.

BB-tree operations: Since a BB-tree is essentially a binary search tree, find operations are no different than they are for any other binary tree. Insertions and deletions operate in essentially the same way they do for AVL trees, first insert or delete the key, and then retrace the search path and rebalance as you go. Rebalancing is performed using rotations. For BB-trees the two rotations go under the special names `skew()` and `split`. Their intuitive meanings are:

`skew(p)`: replace any horizontal left edge with a horizontal right edge by a right rotation at p.

`split(p)`: if a pseudo-node is too large (i.e. more than two consecutive nodes at the same level), then split it by increasing the level of every other node. This is done by making left rotations along a right path of horizontal edges.

Here is their implementation. Split only splits one set of three nodes.

BB-tree Utilities skew and split

```
BBNode skew(BBNode p) {
    if (p.left.level == p.level) {
        q = p.left
        p.left = q.right
        q.right = p
        return q
    }
    else return p
}
BBNode split(BBNode p) {
    if (p.right.right.level == p.level) {
        q = p.right
        p.right = q.left
        q.left = p
        q.level++
        return q
```

```
        }
        else return p
    }
```

Insertion: Insertion performs in the usual way. We walk down the tree until falling out, and insert the new key at the point we fell out. The new node is at level 1. We return up the search path and rebalance. At each node along the search path it suffices to perform one skew and one split.

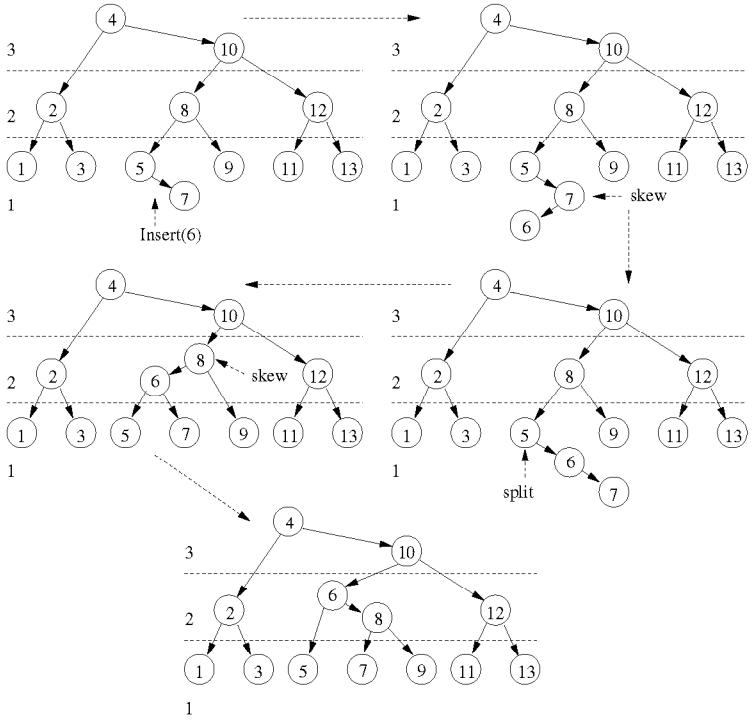

Figure 90: BB-tree insertion.

BB-Tree Insertion

```
BBNode insert(int x, BBNode t) {
    if (t == nil) {                          // empty tree
        t = new BBNode(x,1,nil,nil)          // new node a level 1
    }
    else {
        if (x < t.data)
            t.left = insert(x, t.left)       // insert on left
        else if (x > t.data)
            t.right = insert(x, t.right)     // insert on right
        else                                 // duplicate key
            ...error: duplicate key...
        t = skew(t)                          // rebalance
        t = split(t)
    }
```

```
    return t
}
```

Deletion: As usual deletion is more of a hassle. We first locate the node to be deleted. We replace it's key with an appropriately chose key from level 1 (which need not be a leaf) and proceed to delete the node with replacement key. We retrace the search path towards the root rebalancing along the way. At each node p that we visit there are a number of things that can happen.

(a) If p's child is 2 levels lower, then p drops a level. If p is part of a pseudo-node of size 2, then both nodes drop a level.

(b) Apply a sufficient number of skew operations to align the nodes at this level. In general 3 may be needed: one at p, one at p's right child, and one at p's right-right grandchild.

(c) Apply a sufficient number of splits to fix things up. In general, two may be needed: one at p, and one at p's right child.

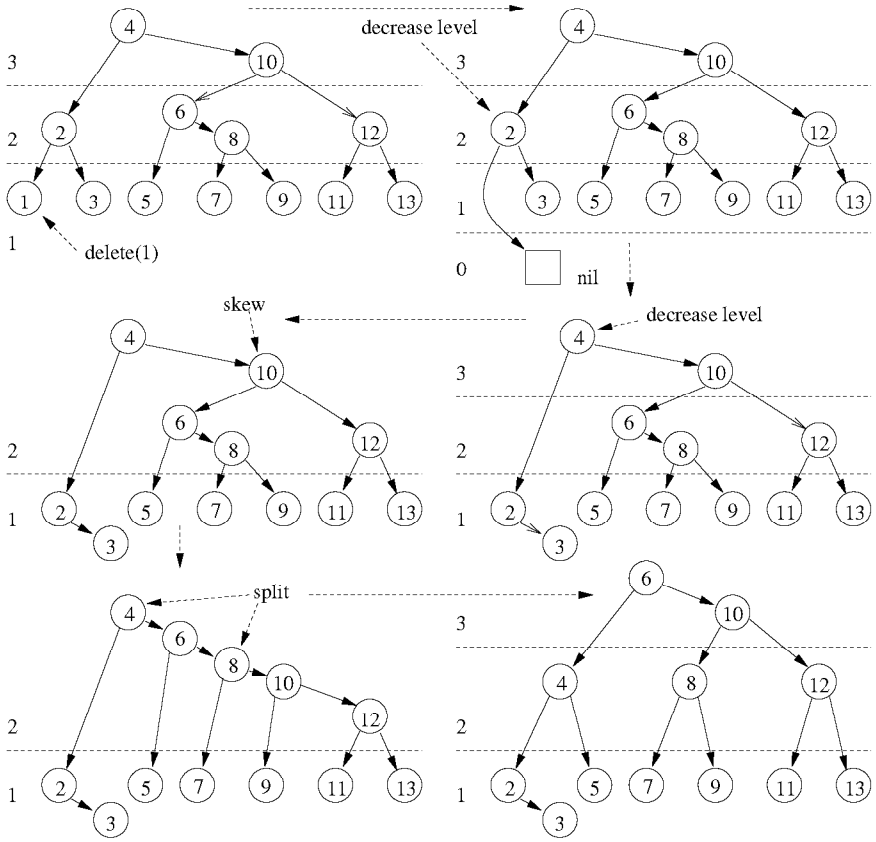

Figure 91: BB-tree deletion.

Important note: We use 3 global variables: `nil` is a pointer to the sentinel node at level 0, `del` is a pointer to the node to be deleted, `repl` is the replacement node. Before calling this routine from the main program, initialize `del = nil`.

BB-Tree Deletion

```
BBNode delete(int x, BBNode t) {
    if (t != nil) {                     // search tree for repl and del
        repl = t
        if (x < t.data)
            t.left = delete(x, t.left)
        else {
            del = t
            t.right = delete(x, t.right)
        }
        if (t == repl) {                // if at bottom remove item
            if ((del != nil) && (x == del.data)) {
                del.data = t.data
                del = nil
                t = t.right             // unlink replacement
                delete repl             // destroy replacement
            }
            else
                ...error: deletion of nonexistent key...
        }
                                        // lost a level?
        else if ((t.left.level < t.level - 1) ||
                (t.right.level < t.level - 1)) {
            t.level--                   // drop down a level
            if (t.right.level > t.level) {
                t.right.level = t.level
            }
            t = skew(t)
            t.right = skew(t.right)
            t.right.right = skew(t.right.right)
            t = split(t)
            t.right = split(t.right)
        }
    }
    return t
}
```