B-Trees (aka Multiway)

A multiway tree is one where you can store more than one value in a node. Now the question becomes, if you can store more than one value in a node, how do you arrange subtrees?

If a node contains k items, (let these be I_1, I_2, ... and I_k, in numeric order), then that node will contain k+1 subtrees. (Let these subtrees be S_1, S_2, ... and S_{k+1}.) In order to create a search tree with a reasonable order, we have to place constraints on the values stored in each subtree. In particular we have:

All values in S_1 are less than I_1.
All values in S_2 are less than I_2, but greater than I_1.
All values in S_3 are less than I_3, but greater than I_2.
...
All values in S_k are less than I_k, but greater than I_{k-1}.
All values in S_{k+1} are greater than I_k.

From this point on, if I refer to a subtree to the left of a value, I_m, I am referring to the subtree directly to the left of the value, S_m. Also, I will refer to S_{m+1} as the subtree to the right of I_m.

Here is an example of a multiway tree:

```
                      20, 40
             /                |                 \
        5, 8, 13, 17          30                50, 70, 80
       /      |    \          /                   /    |     |    \
     2      10   14      22, 25             43    60   71   81,87
```

2-4 Trees

A 2-4 Tree is a specific type of multitree. Here are the specifications for a valid 2-4 Tree:

1) Every node must have in between 2 and 4 children. (Thus, each internal node must store in between 1 and 3 values.)

2) All external nodes (the null children of leaf nodes) have the same depth. (This does imply that all leaf nodes have the same depth as well.)

Here is an example of a 2-4 Tree:

```
                 10,20,30
            /      /        \         \
          3,7     15        22,28    40, 50, 60
       /  | \     / \      / |  \     /  /    \    \
       1  5  8   12  17  21  26 29 35  45   55   65,70
```

In a normal 2-4 Tree insert, compare the value to insert, (4 in this example) to the values in the root node. Based upon which two values the new value to insert falls in between, insert the item into the appropriate subtree. This is less than 10, so it must go into the subtree to the left ot 10. Using the same logic, we will insert the value in the subtree to the right of 3:

```
                 10,20,30
            /      /        \         \
          3,7     15        22,28    40, 50, 60
      /   | \    / \      / |  \     /  /    \    \
      1  4,5  8 12  17  21  26 29 35  45   55   65,70
```

Problems with Insert

Now the problem with this insert method is that the node that a value gets inserted in may already be full (contain 3 values already.) Now, we could just insert the value into a new level of the tree, but the problem with this is that not ALL of the external nodes will have the same depth after this. Instead, what we will do is attempt to push a value up to the parent of the inserted node. Consider inserting 18 into the following tree:

```
           10, 20
   /          |          \
3, 7     13,15,17     22
```

Initially, we'd have the following situation:

```
           10, 20
   /          |          \
3, 7   13,15,17,18   22
```

Now, the idea here is that you can take a value from the overloaded node and send it to the parent. You can send either of the middle values. The default convention is to send the third value:

```
           10,17, 20
   /         /      \     \
3, 7   13,15      18   22
```

When you send this value up, you will have "split" the other three values into two different node groups. This allows for the proper number of children and fixes the situation.

Other Problems with Insert

In the situation above, the parent node was able to "accept" the 17. However, consider a parent node with three values in it and a similar insert (12):

```
            10,20, 30
   /        /      \     \
5     11,14,17   25  32,37
```

Initially, we have:

```
            10,20, 30
   /        /       \       \
5   11,12,14,17   25     32,37
```

Then, using the rule stated above, we have:

```
            10,14, 20, 30
   /      /     |    \       \
5   11,12,   17   25     32,37
```

Now, this looks like a mess: too many parent nodes and subtrees!!!

We'll just have to repeat the process discussed above on the new node that has overflowed. (A node overflows when a 4th value is inserted into it.) Luckily, it's a simple matter to rearrage each sub tree as necessary. (This is because whether we "add" another level to the tree or not, the number of "gaps" between existing nodes stays the same.)

```
             20
          /       \
      10,14        30
    /   |   \      /  \
   5  11,12 17   25   32,37
```

Thus, the way in which a 2-4 Tree changes in height is when an insert propagates an overflowed element all the way to the top node of the tree. This propagation can stop at any level of the tree and is simply dictated by the number of nodes in each ancestor of the initially inserted node.

Here is one more example:

```
              50
        /              \
      25              70, 90
     /  \            /   |     \
   20    27        60   80    91, 93, 94
```

Insert 99:

```
              50
        /              \
      25              70, 90, 94
     /  \            /   /     \       \
   20    27        60  80  91,93     99
```

Implementation Ideas

1) Each node is represented by 2 arrays:
    a) An array of values
    b) An array of references to Nodes.

2) Each node stores
    a) A linked list of values
    b) A linked list of references to Nodes.

Advantage of the first design:

1) Easier to access each item in a node and modify values in a node.

Advantage of the second design:

1) Never have extra values or references stored in a node.