B-Trees continued.

More B-tree operations

Creating an empty B-tree

To initialize a B-tree, we need simply to build an empty root node:

```
B-Tree-Create (T)
        x = allocate-node ();
        leaf[x] = True
        n[x] = 0
        Disk-Write (x)
        root[T] = x
```

This assumes there is an `allocate-node` function that returns a node with *key*, *c*, *leaf* fields, etc., and that each node has a unique "address" on the disk.

Clearly, the running time of `B-Tree-Create` is $O(1)$, dominated by the time it takes to write the node to disk.

Inserting a key into a B-tree

Inserting into a B-tree is a bit more complicated than inserting into an ordinary binary search tree. We have to find a place to put the new key. We would prefer to put it in the root, since that is kept in RAM and so we don't have to do any disk accesses. If that node is not full (i.e., $n[x]$ for that node is not $2t$-1), then we can just stick the new key in, shift around some pointers and keys, write the results back to disk, and we're done. Otherwise, we will have to split the root and do something with the resulting pair of nodes, maintaining the properties of the definition of a B-tree.

Here is the general algorithm for insertinging a key k into a B-tree T. It calls two other procedures, `B-Tree-Split-Child`, that splits a node, and `B-Tree-Insert-Nonfull`, that handles inserting into a node that isn't full.

```
B-Tree-Insert (T, k)
        r = root[T]
        if n[r] = 2t - 1 then
                // uh-oh, the root is full, we have to split it
                s = allocate-node ()
                root[T] = s     // new root node
                leaf[s] = False // will have some children
                n[s] = 0        // for now
                c_1[s] = r // child is the old root node
                B-Tree-Split-Child (s, 1, r) // r is split
                B-Tree-Insert-Nonfull (s, k) // s is clearly not full
        else
                B-Tree-Insert-Nonfull (r, k)
        endif
```

Let's look at the non-full case first: this procedure is called by `B-Tree-Insert` to insert a key into a node that isn't full. In a B-tree with a large minimum degree, this is the common case. Before looking at the pseudocode, let's look at a more English explanation of what's going to happen:

To insert the key *k* into the node *x*, there are two cases:

- *x* is a leaf node. Then we find where *k* belongs in the array of keys, shift everything over to the left, and stick *k* in there.
- *x* is not a leaf node. We can't just stick *k* in because it doesn't have any children; children are really only created when we split a node, so we don't get an unbalanced tree. We find a child of *x* where we can (recursively) insert *k*. We read that child in from disk. If that child is full, we split it and figure out which one *k* belongs in. Then we recursively insert *k* into this child (which we know is non-full, because if it were, we would have split it).

Here's the algorithm:

```
B-Tree-Insert-Nonfull (x, k)
        i = n[x]

        if leaf[x] then

                // shift everything over to the "right" up to the
                // point where the new key k should go

                while i >= 1 and k < $key_i$[x] do
                        $key_{i+1}$[x] = $key_i$[x]
                        i--
                end while

                // stick k in its right place and bump up n[x]

                $key_{i+1}$[x] = k
                n[x]++
        else

                // find child where new key belongs:

                while i >= 1 and k < $key_i$[x] do
                        i--
                end while

                // if k is in $c_i$[x], then k <= $key_i$[x] (from the definition)
                // we'll go back to the last key (least i) where we found this
                // to be true, then read in that child node

                i++
                Disk-Read ($c_i$[x])
                if n[$c_i$[x]] = 2t - 1 then

                        // uh-oh, this child node is full, we'll have to split it

                        B-Tree-Split-Child (x, i, $c_i$[x])

                        // now $c_i$[x] and $c_{i+1}$[x] are the new children,
                        // and $key_i$[x] may have been changed.
                        // we'll see if k belongs in the first or the second

                        if k > $key_i$[x] then i++
                end if

                // call ourself recursively to do the insertion
                B-Tree-Insert-Nonfull ($c_i$[x], k)
        end if
```

Now let's see how to split a node. When we split a node, we always do it with respect to its parent; two new nodes appear and the parent has one more child than it did before. Again, let's see some English before we have to look at the pseudocode:

We will split a node *y* that is the *i*th child of its parent *x*. Node *x* will end up having one more child we'll call *z*, and we'll make room for it in the $c_i[x]$ array right next to *y*.

We know *y* is full, so it has 2*t*-1 keys. We'll "cut" *y* in half, copying $key_{t+1}[y]$ through $key_{2t-1}[y]$ into the first *t*-1 keys of this new node *z*.

If the node isn't a leaf, we'll also have to copy over the child pointers $c_{t+1}[y]$ through $c_{2t}[y]$ (one more child than keys) into the first *t* children of *z*.

Then we have to shift the keys and children of *x* over one starting at index *i+1* to accomodate the new node *z*, and then update the *n*[] counts on *x*, *y* and *z*, finally writing them to disk.

Here's the pseudocode:

```
B-Tree-Split-Child (x, i, y)
        z = allocate-node ()

        // new node is a leaf if old node was

        leaf[z] = leaf[y]

        // we since y is full, the new node must have t-1 keys

        n[z] = t - 1

        // copy over the "right half" of y into z

        for j in 1..t-1 do
                $key_j$[z] = $key_{j+t}$[y]
        end for

        // copy over the child pointers if y isn't a leaf

        if not leaf[y] then
                for j in 1..t do
                        $c_j$[z] = $c_{j+t}$[y]
                end for
        end if

        // having "chopped off" the right half of y, it now has t-1 keys

        n[y] = t - 1

        // shift everything in x over from i+1, then stick the new child in x;
        // y will half its former self as $c_i$[x] and z will
        // be the other half as $c_{i+1}$[x]

        for j in n[x]+1 downto i+1 do
```

```
                $c_{j+1}[x] = c_j[x]$
        end for
        $c_{i+1} = z$

        // the keys have to be shifted over as well...

        for j in n[x] downto i do
                $key_{j+1}[x] = key_j[x]$
        end for

        // ...to accomodate the new key we're bringing in from the middle
        // of y (if you're wondering, since (t-1) + (t-1) = 2t-2, where
        // the other key went, its coming into x)

        $key_i[x] = key_t[y]$
        n[x]++

        // write everything out to disk

        Disk-Write (y)
        Disk-Write (z)
        Disk-Write (x)
```

Note that this is the only time we ever create a child. Doing a split doesn't increase the height of a tree, because we only add a sibling to existing keys at the same level. Thus, the only time the height of the tree ever increases is when we split the root. So we satisfy the part of the definition that says "each leaf must occur at the same depth."

Example of Insertion

Let's look at an example of inserting into a B-tree. For preservation of sanity, let *t* = 2. So a node is full if it has 2(2)-1 = 3 keys in it, and each node can have up to 4 children. We'll insert the sequence `5 9 3 7 1 2 8 6 0 4` into the tree:

```
Step 1: Insert 5
                                 ___
                                |_5_|

Step 2: Insert 9
B-Tree-Insert simply calls B-Tree-Insert-Nonfull, putting 9 to the
right of 5:
                                 _______
                                |_5_|_9_|

Step 3: Insert 3
Again, B-Tree-Insert-Nonfull is called
                               ___ _______
                              |_3_|_5_|_9_|

Step 4: Insert 7
Tree is full.  We allocate a new (empty) node, make it the root, split
the former root, then pull 5 into the new root:
                                 ___
                                |_5_|
                             __ /   \__
                            |_3_|  |_9_|

Then insert we insert 7; it goes in with 9
```

```
                                 ___
                                |_5_|
                             __ /   \_______
                            |_3_|  |_7_|_9_|

Step 5: Insert 1
It goes in with 3

                                 ___
                                |_5_|
                         ___ __ /   \_______
                        |_1_|_3_|  |_7_|_9_|

Step 6: Insert 2
It goes in with 3

                                 ___
                                |_5_|
                               /     \
                       ___ __ /___    \_______
                      |_1_|_2_|_3_|  |_7_|_9_|

Step 7: Insert 8
It goes in with 9


                                 ___
                                |_5_|
                               /     \
                       ___ __ /___    \___________
                      |_1_|_2_|_3_|  |_7_|_8_|_9_|

Step 8: Insert 6
It would go in with |7|8|9|, but that node is full.  So we split it,
bringing its middle child into the root:


                                _______
                               |_5_|_8_|
                              /    |   \
                     ___ ____/__  _|_   \__
                    |_1_|_2_|_3_||_7_| |_9_|

Then insert 6, which goes in with 7:
                                _______
                            ___|_5_|_8_|__
                           /       |      \
                  ___ ____/__    __|_____  \__
                 |_1_|_2_|_3_|  |_6_|_7_|  |_9_|

Step 9: Insert 0

0 would go in with |1|2|3|, which is full, so we split it, sending the middle
child up to the root:
                             ___________
                            |_2_|_5_|_8_|
                          _/    |   |    \_
                        _/      |   |      \_
                      _/_     __|   |_______  \___
                     |_1_|   |_3_| |_6_|_7_| |_9_|

Now we can put 0 in with 1
                             ___________
                            |_2_|_5_|_8_|
                          _/    |   |    \_
                        _/      |   |      \_
                  ___ _/_     __|   |_______ \___
                 |_0_|_1_|   |_3_| |_6_|_7_| |_9_|
```

Step 10: Insert 4
It would be nice to just stick 4 in with 3, but the B-Tree algorithm
requires us to split the full root. Note that, if we don't do this and
one of the leaves becomes full, there would be nowhere to put the middle
key of that split since the root would be full, thus, this split of the
root is necessary:

```
                                 ___
                                |_5_|
                            ___/     \___
                           |_2_|     |_8_|
                         _/    |     |    \_
                       _/      |     |      \_
                 ___ _/_     __|     |_______ \___
                |_0_|_1_|   |_3_|   |_6_|_7_| |_9_|
```

Now we can insert 4, assured that future insertions will work:

```
                                 ___
                                |_5_|
                            ___/     \___
                           |_2_|     |_8_|
                         _/    |     |    \_
                       _/      |     |      \_
                 ___ _/_    ___|___  |________\_____
                |_0_|_1_|  |_3_|_4_| |_6_|_7_| |_9_|
```