B-Tree Pseudocode

The pseudocode for BTree with arrays starting at 0 instead of 1.

Definition of a B-Tree

Below, we rewrite definition of B-Tree to use arrays starting with 0 instead of 1.

A B-tree is a rooted tree (whose root is T.root) having the following properties.

1. Every node x has:

   (a) $x.n$, the number of keys currently stored in x

   (b) $x.n$ keys: $x.key_0 \leq x.key_1 \leq \ldots \leq x.key_{n-1}$

   (c) $x.leaf$, a Boolean which is true if x is a leaf and otherwise false

2. Each internal node x also contains $x.n+1$ points $x.c_0, x.c_1, \ldots, x.c_{n-1}, x.c_n$ to its children. Leaf nodes have no children so their c_i values are null.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree. If k_i is any key stored in the subtree with root $x.c_i$, then

$$k_0 \leq x.key_0 \leq k_1 \leq x.key_1 \leq \ldots \leq x.key_{n-1} \leq k_n$$

4. All leafs have the same depth, which is the height h of the tree.

5. The value $t \geq 2$, is called the minimum degree of the B-tree, helps define the structure of the tree as follows:

   (a) Every note other than the root must have at least $t-1$ keys. Every internal node other than the root has at least t children. If the tree is non-empty, the root must have at least one key.

   (b) Every node may contain at most $2t-1$ keys. Thus an internal node may have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

Some observations

- The root of the B-tree is always in main memory, so we never need to perform a DISK-READ on the root but we do need to perform a DISK-WRITE when the root node is changed.

- Any nodes that are passed as parameters already have had a DISK-READ operation performed on them.
- All procedures are "one-pass" algorithms that proceed downward from the root, without having to back up.
- We will access the key values and the child pointers using arrays inside the node x. They would need to be declared so they can hold a full node, so $2t-1$ keys and $2t$ child pointers: $x.key[0:2t-1]$ and $x.c[0:2t]$ but we will access only the following values:

  $x.key[0]\ldots x.key[x.n-1]$ and $x.c[0]\ldots x.c[x.n]$

B-tree operations

Searching in a B-tree

- Generalization of binary tree search except at node x we make an $(x.n+1)$-way branching decision.
- The return value is a 2-tuple (y,i) consisting of a node y and an index i such that $y.key_i = k$, where we are searching for key k. If not found, the search returns NIL.

```
B-TREE-SEARCH(x, k)
i = 0
while i < n and k > x.key[i]
      i = i + 1
if i < x.n and k == x.key[i]
      return (x, i)
elseif x.leaf
      return NIL
else  DISK-READ(x.c[i])
      return B-TREE-TREE-SEARCH(x.c[i], k)
```

Creating an empty B-tree

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0;
DISK-WRITE(x)
T.root = x
```

Inserting into a B-tree

```
B-TREE-INSERT(T, k)
 r = T.root
 if r.n == 2t-1
      s = B-TREE-SPLIT-ROOT(T)
      B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-SPLIT-ROOT(T)
s = ALLOCATE-NODE()
s.leaf = FALSE
s.n = 0
s.c[0] = T.root
T.root = s
B-TREE-SPLIT-CHILD(s, 0)
return s
```

```
B-TREE-SPLIT-CHILD(x, i)
y = x.c[i] //full node to split
z = ALLOCATE-NODE() //z will take half of y
z.leaf = y.leaf
z.n = t - 1
for j = 0 to t - 2           // z gets y's greatest keys
   z.key[j] = y.key[j + t]
if not y.leaf
   for j = 0 to t - 1        // and it's corresponding children
      z.c[j] = y.c[j + t]
y.n = t - 1                  // y keeps t - 1 keys
for j = x.n downto i+1       // shift x's children to the right...
   x.c[j + 1] = x.c[j]
x.c[i + 1] = z               // to make room for z as a child
for j = x.n - 1 downto i     // shift the corresponding keys in x
   x.key[j + 1] = x.key[j]
x.key[i] = y.key[t - 1]      // insert y's median key
x.n = x.n + 1                // x has gained a child
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n - 1
if x.leaf                          //inserting into a leaf
    while i >= 0 and k < x.key[i]  //shift keys in x to make room for k
        x.key[i + 1] = x.key[i]
        i = i - 1
    x.key[i + 1] = k              //insert k in x
    x.n = x.n + 1                 //now x has 1 more key
    DISK-WRITE(x)
else while i >= 0 and k < x.key[i] //find the child where k belongs
         i = i - 1
     i = i + 1
     DISK-READ(x.c[i])
     if x.c[i].n = 2t - 1          //split the child if it is full
         B-TREE-SPLIT-CHILD(x, i)
         if k > x.key[i]           //does k go into x.c[i] or x.c[i+1]
             i = i + 1
             DISK-READ(x.c[i])
     B-TREE-INSERT-NONFULL(x.c[i], k)
```