

# B-Tree Psuedocode

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

## Definition of a B-Tree

Belowe, we rewrite definiition 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:  $x.key_0 \leq x.key_1 \leq \dots \leq x.key_{n-1}$
  - (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  $x.c_0, x.c_1, \dots, x.c_{n-1}, x.c_n$  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

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

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

## 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)
1. i = 0
2. while i < n and k > x.key[i]
3.     i = i + 1
4. if i < x.n and k == x.key[i]
5.     return (x, i)
6. elseif x.leaf
7.     return NIL
8. else DISK-READ(x.c[i])
9.     return B-TREE-TREE-SEARCH(x.c[i], k)

```

### Creating an empty B-tree

```

B-TREE-CREATE(T)
1. x = ALLOCATE-NODE()
2. x.leaf = TRUE
3. x.n = 0;
4. DISK-WRITE(x)
5. T.root = x

```

## Inserting into a B-tree

B-TREE-INSERT( $T, k$ )

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

B-TREE-SPLIT-ROOT( $T$ )

```
1.  s = ALLOCATE-NODE()
2.  s.leaf = FALSE
3.  s.n = 0
4.  s.c[0] = T.root
5.  T.root = s
6.  B-TREE-SPLIT-CHILD(s, 0)
7.  return s
```

B-TREE-SPLIT-CHILD( $x, i$ )

```
1.  y = x.c[i] //full node to split
2.  z = ALLOCATE-NODE() //z will take half of y
3.  z.leaf = y.leaf
4.  z.n = t - 1
5.  for j = 0 to t - 2          // z gets y's greatest keys
6.      z.key[j] = y.key[j + t]
7.  if not y.leaf
8.      for j = 0 to t - 1      // and its corresponding children
9.          z.c[j] = y.c[j + t]
10. y.n = t - 1                // y keeps t - 1 keys
11. for j = x.n downto i+1      // shift x's children to the right...
12.     x.c[j + 1] = x.c[j]
13. x.c[i + 1] = z              // to make room for z as a child
14. for j = x.n - 1 downto i    // shift the corresponding keys in x
15.     x.key[j + 1] = x.key[j]
16. x.key[i] = y.key[t - 1]    // insert y's median key
17. x.n = x.n + 1              // x has gained a child
18. DISK-WRITE(y)
19. DISK-WRITE(z)
20. DISK-WRITE(x)
```

```

B-TREE-INSERT-NONFULL(x, k)
1.  i = x.n - 1
2.  if x.leaf                                //inserting into a leaf
3.      while i >= 0 and k < x.key[i] //shift keys in x to make room for k
4.          x.key[i + 1] = x.key[i]
5.          i = i - 1
6.      x.key[i + 1] = k                      //insert k in x
7.      x.n = x.n + 1                        //now x has 1 more key
8.      DISK-WRITE(x)
9.  else while i >= 0 and k < x.key[i] //find the child where k belongs
10.     i = i - 1
11.     i = i + 1
12.     DISK-READ(x.c[i])
13.     if x.c[i].n = 2t - 1                //spl't the child if it is full
14.         B-TREE-SPLIT-CHILD(x, i)
15.         if k > x.key[i]                  //does k go into x.c[i] or x.c[i+1]
16.             i = i + 1
17.         DISK-READ(x.c[i])
18.         B-TREE-INSERT-NONFULL(x.c[i], k)

```