B-Tree

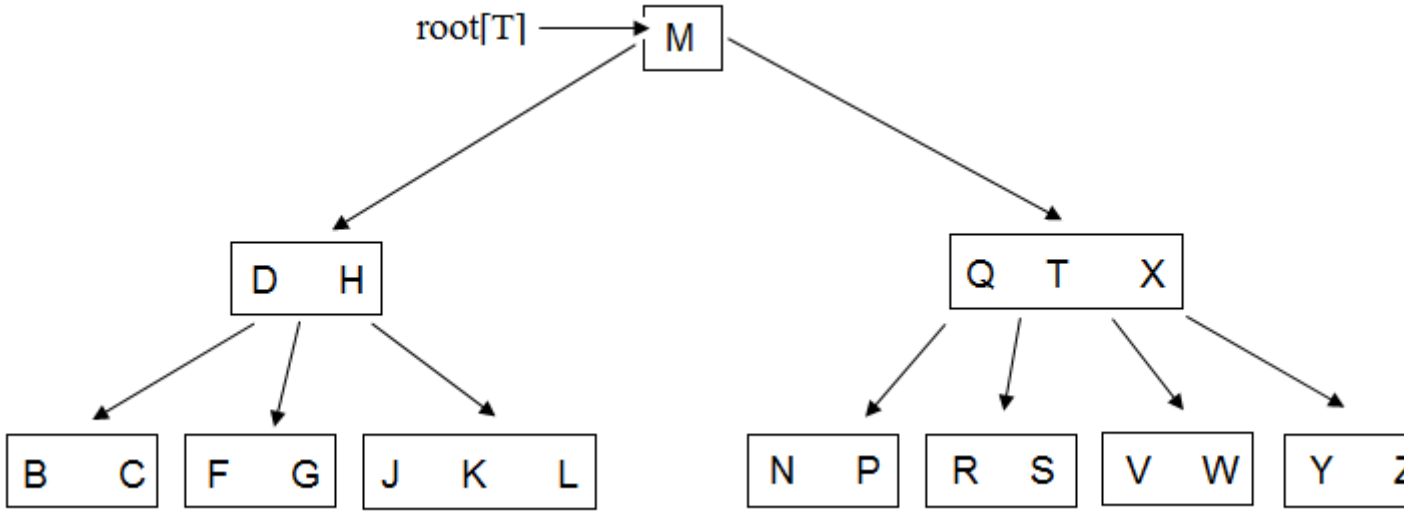
B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices.

B-tree nodes may have many children, from a handful to thousands. That is called the ″branching factor″ of a B-tree.

Every n-node B-tree has height O(lg n), therefore, B-tree can be used to implement many dynamic-set operations in time O(lg n).

B-trees generalize binary search trees in a natural manner.

- If a B-tree node x contains n[x] keys, then x has n[x]+1 children
- The keys in node x are used as dividing points separating the range of keys handled by x into n[x]+1 subranges, each handled by one child of x.
- When searching for a key in a B-tree, we make an (n[x]+1) way decision based on comparisons with the n[x] keys stored at node x.



Definition of B-tree

A B-tree T is a rooted tree having the following properties :

(i) Every node x has the following fields :

a. n[x], the number of keys currently stored in node x,

b. the n[x] keys themselves, stored in non-decreasing order:

$key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$, and

c. leaf[x], a boolean value that is TRUE if x is a leaf and FALSE if it is an internal node.

(ii) If x is an internal node, it contains n[x]+1 pointers $c_1[x], c_2[x], \ldots, c_{n[x]}[x]$ to its children.

Leaf nodes have no children, so their c_i fields are undefined.

(iii) The keys $key_i[x]$ separate the ranges of keys stored in each subtree:

if k_i is any key stored in the subtree with root $c_i[x]$ then, $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$.

(iv) Every leaf has the same depth, which is the tree's height h.

(v) There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \ge 2$ called the minimum degree of the B-tree :

a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node can contain at most 2t - 1 keys. Therefore, an internal node can have at most 2t children. We say that a node is full if it contains exactly 2t - 1 keys.

Basic operations on B-tree :

In basic operations, we present the details of the operations B-TREE-SEARCH, B-TREECREATE, and B-TREE-INSERT.In these procedures, we adopt two conventions:

• The root of the B-tree is always in main memory, so that a DISK-READ on the root is never required; a DISK-WRITE of the root is required, however, whenever the root node is changed.

• Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures we present are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

1. Searching a B-tree - Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way" branching decision at each node, we make a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, we make an (n[x] + 1)-way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH procedure defined for binary search trees. B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH(root[T], k). If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that keyi[y] = k. Otherwise, the value NIL is returned.

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] and k > key_i[x]
do i ← i + 1
if i ≤ n[x] and k = key_i[x]
then return (x, i)
if leaf [x]
then return NIL
else DISK-READ(c_i[x])
return B-TREE-SEARCH(c_i[x], k)
```

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a path downward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $\Theta(h) = \Theta(\log_t n)$, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $n[x] < 2t$, the time taken by the while loop of lines 2-3 within each node is O(t), and the total CPU time is $O(th) = O(t \log_t n)$.

2. <u>Creating an empty B-tree</u> -

To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in O(1) time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE()
leaf[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
root[T] ← x
```

B-TREE-CREATE requires O(1) disk operations and O(1) CPU time.

Source:

http://www.learnalgorithms.in/#