B-Trees

B-trees are balanced search trees designed to work well on magnetic disks or direct-access secondary storage devices. B-trees differ significantly from red-black trees in that B-tree nodes may have many children, from a handful to thousands.

It often takes more time to access a page of information and read it from a disk than it takes for the computer to examine all the information read. For this reason, we look at the two principal components of the running time:

- the number of disk accesses, and
- the CPU time.

We model these disk operations in our pseudo-code as follows. Let $x$ be a pointer to an object. Disk-Read($x$) to read object $x$ into main memory; Disk-Write($x$) to write object $x$ to the disk.

1 Definition

A B-tree is a rooted tree having the following properties.

1. Every node $x$ has the following fields:
   - $n[x]$, the number of keys currently stored in node $x$
   - the $n[x]$ keys themselves, stored in nondecreasing order: $key_{1}[x] \leq key_{2}[x] \leq \ldots \leq key_{n[x]}[x]$, and

![Figure 1: B-Tree where $t = 2$ and the sequence 9,0,8,1,7,2,6,3,5,4 inserted.](image-url)
• $leaf[x]$, a boolean value that is TRUE if $x$ is a leaf and FALSE if $x$
is an internal node.

2. If $x$ is an internal node, it also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their $c_i$ fields are undefined.

3. 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

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

4. Every leaf has the same depth, which is the tree’s height $h$.

5. 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 $t \geq 2$ called the minimum degree of the B-tree:

• 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 non-empty, the root must have at least one key.
• 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.

**THEOREM** If $n \geq 1$, then for any $n$-key B-tree $T$ of height $h$ and minimum degree $t \geq 2$, then

$$h \leq \log_t \frac{n+1}{2}.$$  

2 Operations

The operations for B-trees include B-Tree-Search, B-Tree-Create, B-Tree-Insert, B-Tree-Delete, etc. We assume that:

• the root of the B-tree is always in main memory.
• any nodes that are passed as parameters must already have had a Disk-Read operation performed on them.
2.1 Search

B-TREE-SEARCH($x, k$)
1 $i \leftarrow 1$
2 while $i \leq n[x]$ and $k > \text{key}_i[x]$
3 do $i \leftarrow i + 1$
4 if $i \leq n[x]$ and $k = \text{key}_i[x]$
5 then return ($x, i$)
6 if leaf[$x$]
7 then return NIL
8 else Disk-Read($c_i[x]$)
9 return B-TREE-SEARCH($c_i[x], k$)

2.2 Creation

B-TREE-CREATE($T$)
1 $x \leftarrow \text{Allocate-Node}()$
2 leaf[$x$] $\leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 Disk-Write($x$)
5 root[$T$] $\leftarrow x$

2.3 Insertion

A fundamental operation used during insertion is the splitting of a full node $y$ (having $2t - 1$ keys) around its median key $\text{key}_i[y]$ into two nodes having $t - 1$ keys each. The median key moves up into $y$’s parent, which must be nonfull prior to the splitting of $y$. 

Figure 2: The arguments for the B-TREE-SPLIT-CHILD algorithm.
Figure 3: The result of child splitting.

\[ \text{B-TREE-SPLIT-CHILD}(x, i, y) \]
1 \[ z \leftarrow \text{ALLOCATE-NODE}() \]
2 \[ \text{leaf}[z] \leftarrow \text{leaf}[y] \]
3 \[ n[z] \leftarrow t - 1 \]
4 \[ \text{for } j \leftarrow 1 \text{ to } t - 1 \]
5 \[ \text{do } \text{key}_{j}[z] \leftarrow \text{key}_{j+1}[y] \]
6 \[ \text{if not } \text{leaf}[y] \]
7 \[ \text{then for } j \leftarrow 1 \text{ to } t \]
8 \[ \text{do } c_{j}[z] \leftarrow c_{j+1}[y] \]
9 \[ n[y] \leftarrow t - 1 \]
10 \[ \text{for } j \leftarrow n[x] + 1 \text{ downto } i + 1 \]
11 \[ \text{do } c_{j+1}[x] \leftarrow c_{j}[x] \]
12 \[ c_{i+1}[x] \leftarrow z \]
13 \[ \text{for } j \leftarrow n[x] \text{ downto } i \]
14 \[ \text{do } \text{key}_{j+1}[x] \leftarrow \text{key}_{j}[x] \]
15 \[ \text{key}_{i}[x] \leftarrow \text{key}_{i}[y] \]
16 \[ n[x] \leftarrow n[x] + 1 \]
17 \[ \text{DISK-WRITE}(y) \]
18 \[ \text{DISK-WRITE}(z) \]
19 \[ \text{DISK-WRITE}(x) \]

Inserting a key \( k \) into a B-tree \( T \) of height \( h \) is done in a single pass down the tree, requiring \( O(h) \) disk accesses. The CPU time required is \( O(th) = O(t \log_t n) \).
B-Tree-Insert($T, k$)
1 $r \leftarrow \text{root}[T]$
2 if $\text{n}[r] = 2t - 1$
3 then $s \leftarrow \text{ALLOCATE-NODE}()$
4 \hspace{1em} $\text{root}[T] \leftarrow s$
5 \hspace{1em} $\text{leaf}[s] \leftarrow \text{FALSE}$
6 \hspace{1em} $n[s] \leftarrow 0$
7 \hspace{1em} $c_1[s] \leftarrow r$
8 $\text{B-Tree-Split-Child}(s, 1, r)$
9 $\text{B-Tree-Insert-Nonfull}(s, k)$
10 else $\text{B-Tree-Insert-Nonfull}(r, k)$

B-Tree-Insert-Nonfull($x, k$)
1 $i \leftarrow n[x]$
2 if $\text{leaf}[x]$
3 then while $i \geq 1$ and $k < \text{key}_i[x]$
4 \hspace{1em} $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$
5 \hspace{1em} $i \leftarrow i - 1$
6 \hspace{1em} $\text{key}_{i+1}[x] \leftarrow k$
7 \hspace{1em} $n[x] \leftarrow n[x] + 1$
8 $\text{Disk-Write}(x)$
9 else while $i \geq 1$ and $k < \text{key}_i[x]$
10 \hspace{1em} $\text{Disk-Read}(c_i[x])$
11 \hspace{1em} $i \leftarrow i + 1$
12 $\text{Disk-Write}(x)$
13 if $n[c_i[x]] = 2t - 1$
14 \hspace{1em} $\text{B-Tree-Split-Child}(x, i, c_i[x])$
15 \hspace{1em} if $k > \text{key}_i[x]$
16 \hspace{2em} $i \leftarrow i + 1$
17 $\text{B-Tree-Insert-Nonfull}(c_i[x], k)$

2.4 Deletion

1. If the key $k$ is in node $x$ and $x$ is a leaf, delete the key $k$ from $x$.

2. If the key $k$ is in node $x$ and $x$ is an internal node, do the following:
   
   • If the child $y$ that precedes $k$ in node $x$ and $x$ has at least $t$ keys, then find the predecessor $k'$ of $k$ in the subtree rooted at $y$. Recursively delete $k'$, and replace $k$ by $k'$ in $x$. 

Figure 4: See Fig 18.6 (page 446) – create a new node for root if root is $2t - 1$
Figure 5: Inserting the sequence 9,0,8,1,7,2,6,3,5,4 into a B-Tree.
Symmetrically, if the child \( z \) that follows \( k \) in node \( x \) has at least \( t \) keys, then find the successor \( k' \) of \( k \) in the subtree rooted at \( z \). Recursively delete \( k' \), and replace \( k \) by \( k' \) in \( x \).

Otherwise, if both \( y \) and \( z \) have only \( t - 1 \) keys, merge \( k \) and all of \( z \) into \( y \), so that \( x \) loses both \( k \) and the pointer to \( z \), and \( y \) now contains \( 2t - 1 \) keys. Then free \( z \) and recursively delete \( k \) from \( y \).

3. If the key \( k \) is not present in internal node \( x \), determine the root \( c_i[x] \) of the appropriate subtree that must contain \( k \), if \( k \) is in the tree at all. If \( c_i[x] \) has only \( t - 1 \) keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least \( t \) keys. Then, finish by recursing on the appropriate child of \( x \).

(a) If \( c_i[x] \) has only \( t - 1 \) keys but has a sibling with \( t \) keys, give \( c_i[x] \) an extra key by moving a key from \( x \) down into \( c_i[x] \), moving a key from \( c_i[x] \)'s immediate left or right sibling up into \( x \), and moving the appropriate child from the sibling into \( c_i[x] \).

(b) If \( c_i[x] \) and both of \( c_i[x] \)'s immediate siblings have \( t - 1 \) keys, merge \( c_i \) with one sibling., which involves moving a key from \( x \) down into the new merged node to become the median key from that node.