B-Trees

A B-tree is a balanced tree designed to be stored on a disk. Disks can only read and write (large) fixed-size blocks of data at once. A B-tree stores multiple keys in each of its nodes so that (1) a single disk read can access a lot of keys, and (2) the branching factor of the tree is very high (in practice, 1000+), so that a tree of small height can store a huge number of keys, any of which can be accessed with just a few disk operations.

Formally, a B-tree is a rooted tree $T$ (with root $\text{root}(T)$), each of whose nodes can contain a variable number of keys, and the following properties:

1. Each node $x$ of the tree contains the following data:
   - $n(x)$ – the number of keys in node $x$
   - leaf($x$) – a boolean, true iff $x$ is a leaf of the tree
   - keys $k_1(x) \ldots k_{n(x)}(x)$, stored in sorted order
   - child pointers $c_1(x) \ldots c_{n(x)+1}(x)$. Note that a non-leaf node with $n$ keys always has $n+1$ children.

2. The equivalent to the BST property for B-trees is as follows. Let $c_i(x)$ be the pointer to the subtree that lies between keys $k_{i-1}(x)$ and $k_i(x)$. Then (assuming no duplicate keys) every key in the subtree rooted at $c_i(x)$ lies strictly between $k_{i-1}(x)$ and $k_i(x)$.

   For example, if $k_2(x) = 5$, and $k_3(x) = 17$, all the keys in the subtree rooted at $c_3(x)$ would lie between 6 and 16.

   The keys in the subtree rooted $c_1(x)$ are all < $k_1(x)$, while those in the subtree rooted at $c_{n(x)+1}(x)$ are all > $k_{n(x)}(x)$.

3. A B-tree has some height $h$ (which we define as the total number of levels including the root). Every leaf of the tree is at the same depth $h$.

4. A B-tree is parameterized by a value $t \geq 2$, called its minimum degree. Every B-tree with minimum degree $t$ must satisfy the following two degree invariants:

   (a) min-degree: Each node of the tree except the root must contain at least $t - 1$ keys (and hence must have at least $t$ children if it is not a leaf). The root must have at least 1 key (two children).

   (b) max-degree: Each node of the tree must contain no more than $2t - 1$ keys. A node with exactly $2t - 1$ keys is called “full.” (In practice, this limit derives from the size of the disk block used to store a node. Note that $2t - 1$ is always an odd number.)

The min-degree invariant, together with the fact that all leaves of a B-tree are at the same depth, implies that a B-tree with $n$ keys has height $O(\log n)$. Hence, any operation that traverses a path from a root to a leaf (find, insert, delete) takes worst-case $O(\log n)$ disk operations.
Searching a B-tree is very similar to searching a binary search tree. The pseudocode looks something like the following:

```plaintext
find(x, k)
  if k is in x
    return k (and any associated data)
  else if leaf(x)
    return “not found”
  else
    locate i such that k lies between k_{i-1}(x) and k_i(x)
    find(c_i(x), k)
```

We start the search by calling `find(root[T], k)`.

To insert a new key in a B-tree, we find the appropriate leaf for the key (as above) and put it there. However, we must not fail if the target leaf is currently full! We therefore introduce the `splitting` operation on a full tree node `x` as follows:

```plaintext
split(x)
  m ← k_t(x)  // x has 2t − 1 keys, so k_t is median
  create node x_ℓ from keys k_1(x) . . . k_{t-1}(x)
  create node x_r from keys k_{t+1}(x) . . . k_{2t-1}(x)
  place m in x’s parent
  place pointers to x_ℓ and x_r to left and right of m
```

One subtlety of this procedure is that we cannot simply jump from `x` to its parent – B-tree nodes contain no parent pointer. Hence, we must know `x`’s parent when we split it. To make this fact explicit, we should rewrite the pseudocode as `split(x, p)`, where `p` is `x`’s parent.

If `x` is the root of the tree, we must create a new node to hold the median key after the split. This node becomes the parent of the two nodes created from the old root and so is the new root of the tree. Hence, B-trees grow up, not down.

The other problem with splitting is that `x`’s parent may already be full and so may be unable to accept the median key from `x`. The `INSERT` procedure must work around this problem.

We describe `INSERT` on a B-tree in terms of a recursive procedure `DoInsert(x, k)`. Calling `INSERT(T, x)` simply calls `DoInsert(root[T], k)`. Insertion maintains the following invariant:

Whenever `DoInsert(x, k)` is called, either `x` is the root, or `x`’s parent is not full.
\textbf{DoInsert}(x, k)
\begin{itemize}
  \item if \( x \) is full
    \begin{itemize}
      \item \( m \leftarrow k_t(x) \) \quad \triangleright \text{median key}
      \item \text{SPLIT}(x) \quad \triangleright \text{succeeds by invariant}
    \end{itemize}
  \item if \( k < m \)
    \begin{itemize}
      \item \( x \leftarrow x_\ell \)
    \end{itemize}
  \item else
    \begin{itemize}
      \item \( x \leftarrow x_r \)
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item if \text{leaf}(x)
    \begin{itemize}
      \item place \( k \) in \( x \) \quad \triangleright x \text{ not full – it would have been split}
    \end{itemize}
  \item else
    \begin{itemize}
      \item \( y \leftarrow \) appropriate child of \( x \), as in \text{FIND}
      \item \text{DoInsert}(y, k) \quad \triangleright x \text{ not full, so invariant holds}
    \end{itemize}
\end{itemize}

Again, we should probably rewrite \text{DoInsert} to explicitly track the parent \( p \) of the current node, as we did with \text{SPLIT}. Here’s a modified version that tracks the parent:

\textbf{DoInsert}(x, p, k)
\begin{itemize}
  \item if \( x \) is full
    \begin{itemize}
      \item \( m \leftarrow k_t(x) \) \quad \triangleright \text{median key}
      \item \text{SPLIT}(x, p) \quad \triangleright \text{succeeds by invariant}
    \end{itemize}
  \item if \( k < m \)
    \begin{itemize}
      \item \( x \leftarrow x_\ell \)
    \end{itemize}
  \item else
    \begin{itemize}
      \item \( x \leftarrow x_r \)
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item if \text{leaf}(x)
    \begin{itemize}
      \item place \( k \) in \( x \) \quad \triangleright x \text{ not full – it would have been split}
    \end{itemize}
  \item else
    \begin{itemize}
      \item \( y \leftarrow \) appropriate child of \( x \), as in \text{FIND}
      \item \text{DoInsert}(y, x, k) \quad \triangleright x \text{ not full, so invariant holds}
    \end{itemize}
\end{itemize}