B-Trees

AVL trees and other binary search trees are suitable for organizing data that is entirely contained within computer memory. When the amount of data is too large to fit entirely in memory, i.e., when it is *external* data, these structures perform poorly. This is because of the enormous disparity in access times between memory accesses and disk accesses. The big-O analysis may show that a search is (O(log n)) in the worst case, but it hides the fact that the constants may be extremely large.

The problem is that disk accesses are typically 1000 times slower than in-memory accesses. For example, consider a disk revolving at 10,800 RPM, current state of the art. One revolution takes 1/180th second. Since on average, we need to seek one half a rotation, we need 1/360th of a second to begin reading the data, or about 3 msec. In 3 msec., a typical CPU can execute about 75,000 instructions. If we use a binary search tree as a disk's file index, then every node access requires time at least equivalent to 75,000 instructions!

One solution is to increase the degree of the tree, so that the tree is flatter and fewer disk accesses are needed. More time is spent in the CPU to find the right node, but that is small compared to disk I/O. In 1970, R. Bayer and E. McCreight discovered (or invented, depending on your point of view) a multi-way tree called a B-tree. The strategy underlying the B-tree idea is to flatten the tree, making it a high degree, and pay the bill by doing more work inside each internal node, possibly a search through a list of children, to find the correct child. Even resorting to a linear search of an array of children's keys is worth the effort, since it saves the equivalent of hundreds of thousands of instructions.

The textbook defines a slight variation on the original B-tree idea.

Definition. A ***B-tree of order m*** and ***L*** is an m-ary tree with the following properties:

1. The data items are stored in the leaves.
2. The interior nodes store up to m-1 keys, K_1, K_2, ..., K_{m-1}. Key K_i is the smallest key in subtree T_{i+1} of the node.
3. For all keys in a given node, $K_1 < K_2 < \ldots < K_{m-1}$.
4. All interior nodes except the root have between ceil(m/2) and m children.
5. If the root is not a leaf node, then it has between 2 and m children.
6. All leaf nodes are at the same depth.
7. If the tree has at least ceil(L/2) data items then all leaf nodes have between ceil(L/2) and L children.

The figure below shows an internal node of a B-tree of order m. It has m subtree pointers, P_0 through P_{m-1}.

T_1	K_1	T_2	K_2	T_3	.....	T_{m-1}	K_{m-1}	T_m

Note about terminology.

The term "leaf node" in the above definition refers to the nodes that hold data items exclusively. These leaf nodes are a different type of node from the interior nodes, which store pointers and keys. The definition, part 5, states that the root of the tree may be a leaf node or an interior node. When the first few items are inserted into the tree, there are not enough of them to require an interior node. Therefore, the tree consists of a single leaf node. As soon as there are enough data items to warrant two leaf nodes, an interior node is created with pointer to the two leaf nodes.

Determining Values for m and L

The design of a B-tree should take into consideration the size of a disk block, the size of a disk block address, the size of a data item, and the size of the key. For example, suppose a disk block is 4096 bytes, disk addresses are 4 bytes long, a data item is a 512 byte record, and a key is 32 bytes long. Since a node has m-1 keys and m children, we need 32(m-1) + 4m bytes per interior node. Solving for the largest m such that 32(m-1) + 4m = 36m-32 <= 4096, we get m = 114. Since data items are 512 bytes each and a disk block is 4096 bytes, 8 data items fit in a disk block. It follows that the way to minimize disk reads is to let m = 114 and L = 8. Each time a block is accessed, all keys for a given node are loaded, and the appropriate subtree can be determined. A more accurate analysis should really be done, to account for actual search time, if m starts getting very large (because keys are small compared to block sizes).

In general, let:

- p be the number of bytes in a pointer,
- B be the number of bytes in a disk block,
- k be the number of bytes in a key, and
- d be the number of bytes in a data item.

Then an internal node has pm + k(m-1) bytes and a leaf node has d bytes. Setting

$$B = pm + k(m-1) = (p+k)m - k$$

we get

$$m = \text{floor}((B+k)/(p+k))$$

as the value for m, and setting

$$B = Ld$$

we get

$$L = \text{floor}(B/d)$$

for L.

B-Tree Search

To search using a B-tree requires either linear searching or binary searching through the keys in each node. If binary search is used, each node requires O(log m) comparisons. When the leaf node s finally reached, the data items can be searched linearly or using binary search. The number of disk accesses is equal to the height of the tree, which is $O(\log_m n)$.

B-Tree Insertion

Inserting into a B-tree is much more complex than into an AVL tree. This is because the requirements are more stringent. The number of children must be controlled in both leaf nodes and interior nodes. An insertion can cause a leaf node to exceed its child limit. When this happens, the node must be split into two nodes each with fewer children. This split in turn might cause the parent node's child limit to be exceeded, and so on. The algorithm is roughly:

Search the tree for the place to insert x. If x is already in the tree, return.

If the leaf node has fewer than L children, insert x into the appropriate position and return. Otherwise, split the leaf into two leaves, one having *ceil*((L+1)/2) and the other having *floor*((L+1)/2) items.

```
Recursively do the following:
If the parent has fewer than m children,
    insert the new leaf into the parent node and update the keys of the parent so that
    there is a new key for the new leaf.
Otherwise,
    split the parent into two nodes, a left and a right, making the first ceil(m/2) nodes
    children of the left parent node and the remaining children into the
    right parent node.
    If the parent node has a parent, then insert the newly created node into the parent
    of the parent, and let the parent be the parent of the parent, and repeat these steps.
    If not, this is a root. Create a new root and make the two nodes children of the new root.
```

Notice that the tree grows in height only as a result of the root node being split. This is an extremely rare event. The tree reaches a height of h (measured as the number of edges from the root to the leaf nodes) as a result of h-1 splitting operations. A tree of height h of order m and L has at least $2\cdot(\lceil m/2\rceil)^{h-1}\cdot\lceil L/2\rceil$ data items. For example, if m = 20, L = 40 and h = 4, the tree has at least 40,000 data items and the root split just 3 times! What is the most data items in this tree? If a tree has 40,000 items, what is the least height it would have, given m and L?

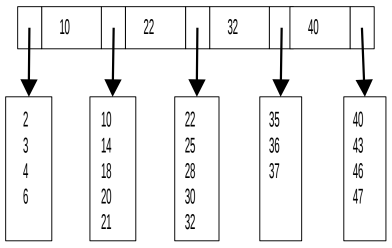
B-Tree Deletion

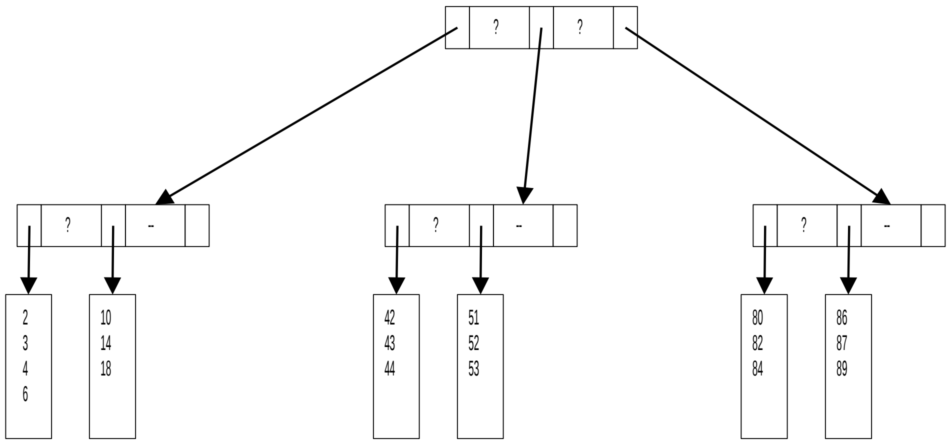
Deleting a node from a B-tree is as complex as B-tree insertion. A worst-case deletion may require deleting nodes all the way up to the root. If it makes it that far, the root is deleted and the height of the tree is diminished by 1.

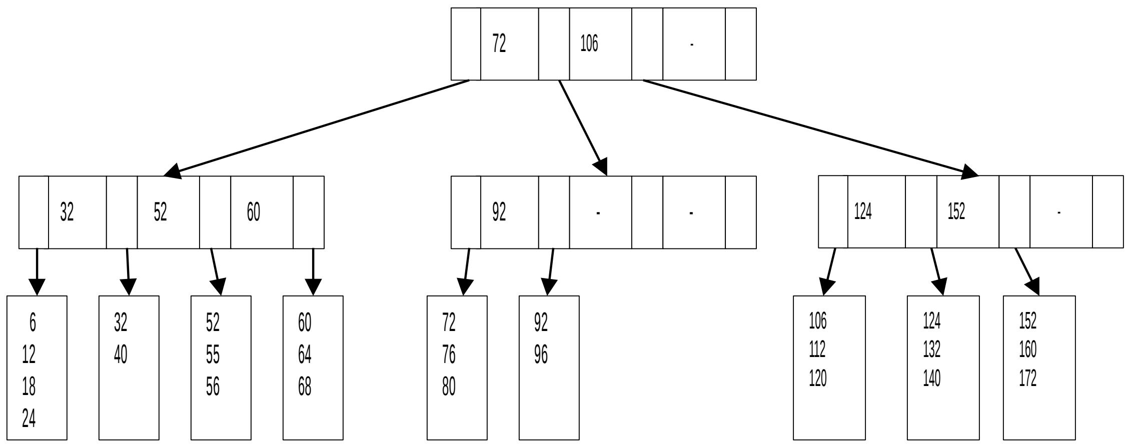
The problem with deletion is that a node may have too few children after a deletion. In that case, nodes are merged. Merging is the opposite of splitting. The node is merged with the node to its left, unless it is the leftmost node, in which case it is merged with the node to its right. If the total number of nodes in the combined node is greater than L, then merging cannot be done, in which case data is borrowed from the adjacent node to replace the item that was deleted.

If merging takes place, then the gap in the sequence of child trees is filled in by moving all sibling trees to the left to fill in the place of the node that was deleted. If the parent node now has too few children, i.e., less than ceil(m/2), then the parent node is merged with a sibling of the parent, combining their children into a single node. The method of doing this is very straightforward. If this merging cannot be done, then borrowing is used instead, taking a child tree from the neighbor to replace the subtree that was merged.

This process can repeat all the back to the root. If the root has fewer than ceil(m/2) children, it is okay as long as it has at least two children. If it has only one child node, the root is deleted and this child node becomes the new root.






72
106
-
32
52
60
92
-
-
124
152
-
6
12
18
24
32
40
52
55
56
60
64
68
72
76
80
92
96
106
112
120
124
132
140
152
160
172