B Tree

B Tree is a Self-balancing search tree that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems. B-trees were invented by Rudolf Bayer and Edward M.McCreight while working at Boeing Research Labs, for the purpose of efficiently managing index pages for large random access files. The basic assumption was that indexes would be so voluminous that only small chunks of the tree could fit in main memory.

A B-tree of order *m* is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with *k* children contains $k - 1$ keys.
5. All leaves appear in the same level and carry no information.

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

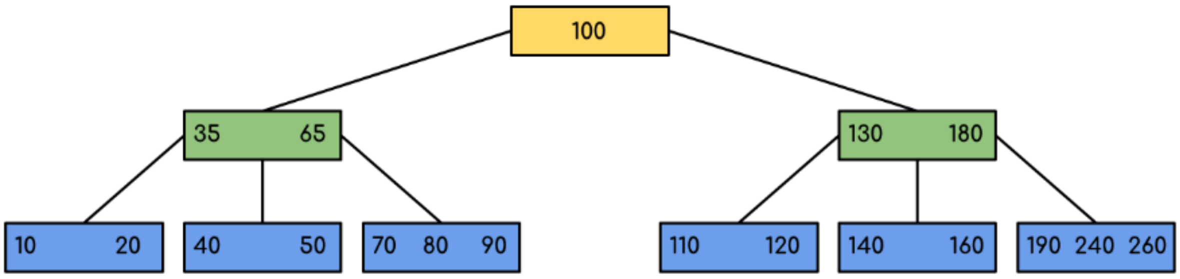
Internal nodes
Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and a **minimum** of *L* children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between $L-1$ and $U-1$). *U* must be either $2L$ or $2L-1$; therefore each internal node is at least half full. The relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there's room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.
The root node
The root node's number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree with no children at all.
Leaf nodes
In Knuth's terminology, leaf nodes do not carry any information. The internal nodes that are one level above the leaves are what would be called "leaves" by other authors: these nodes only store keys (at most m-1, and at least $m/2$-1 if they are not the root) and pointers to nodes carrying no information.

A B-tree of depth $n+1$ can hold about *U* times as many items as a B-tree of depth *n*, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

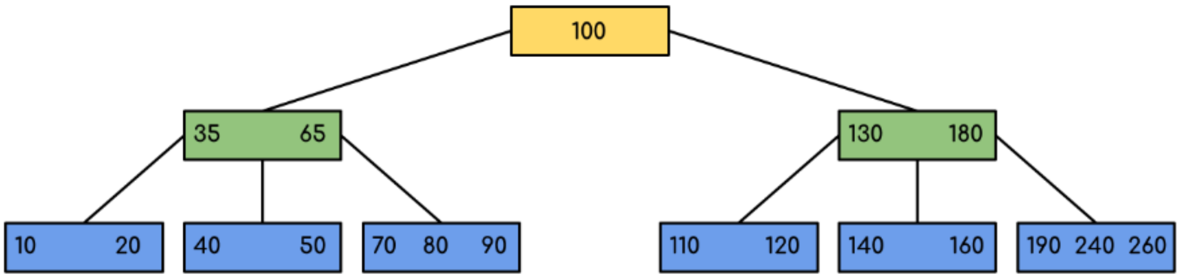
Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree except leaf nodes.

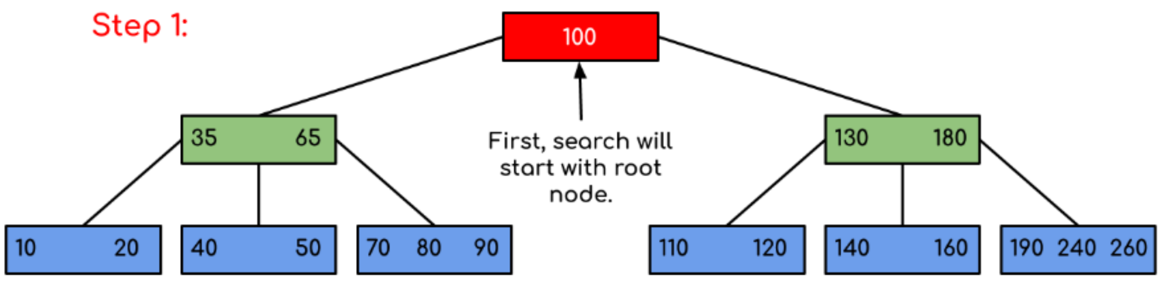
Properties of B-Tree:

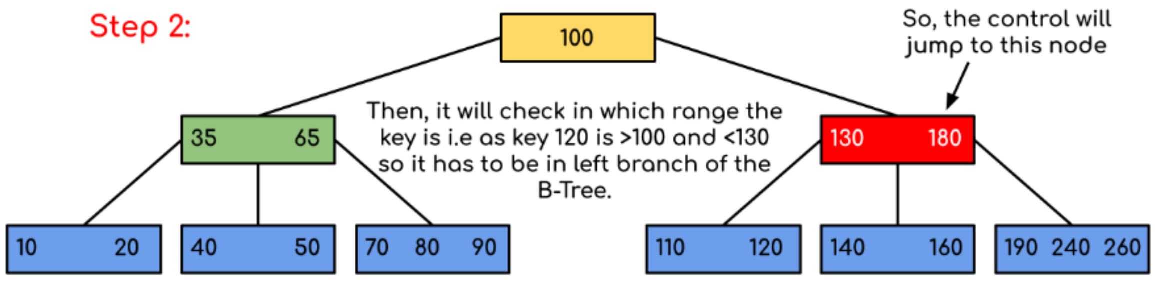
1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).



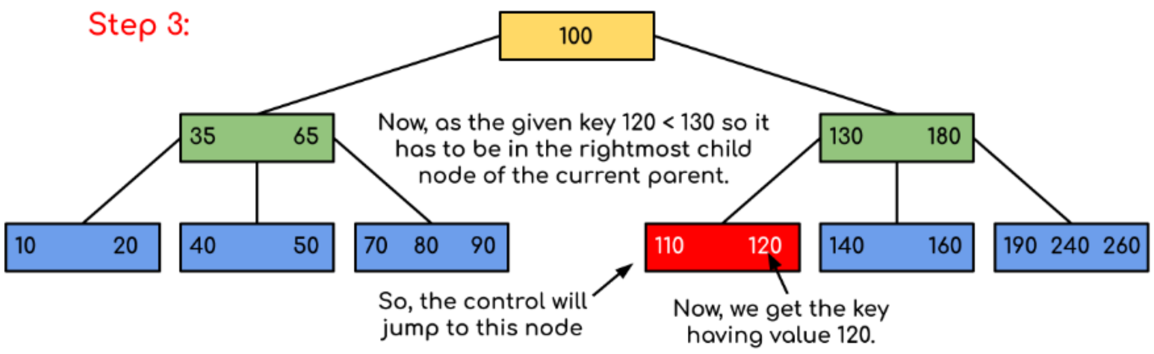
Searching 120 in the given B-Tree







Step 2:
100
So, the control will jump to this node
35 65
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260



Step 3:
100
35 65
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

Insertion

1. Initialize x as root.

2) While x is not leaf, do following

..**a)** Find the child of x that is going to be traversed next. Let the child be y.

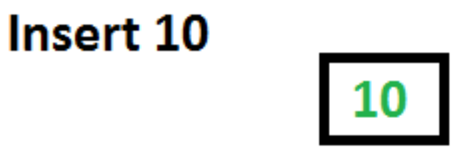
..**b)** If y is not full, change x to point to y.

..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

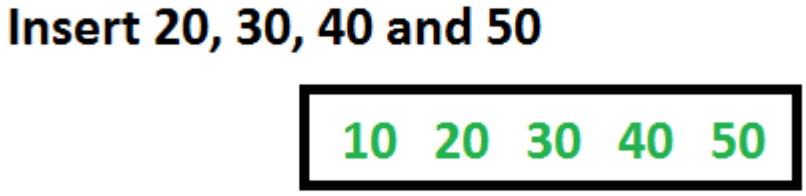
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Let us understand the algorithm with an example tree of minimum degree ‘t’ as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

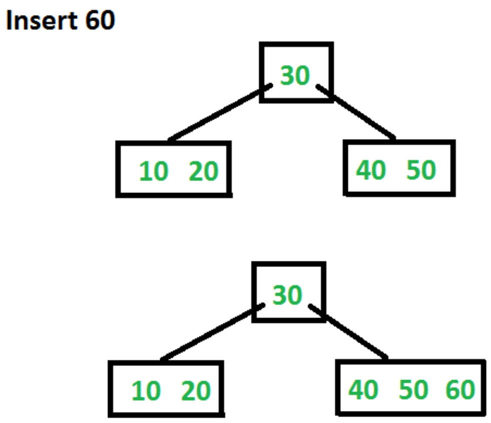
Initially root is NULL. Let us first insert 10.



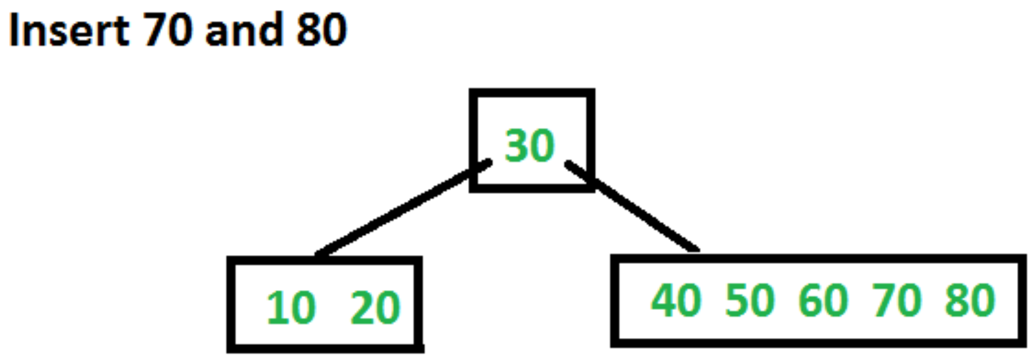
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2*t – 1 which is 5.



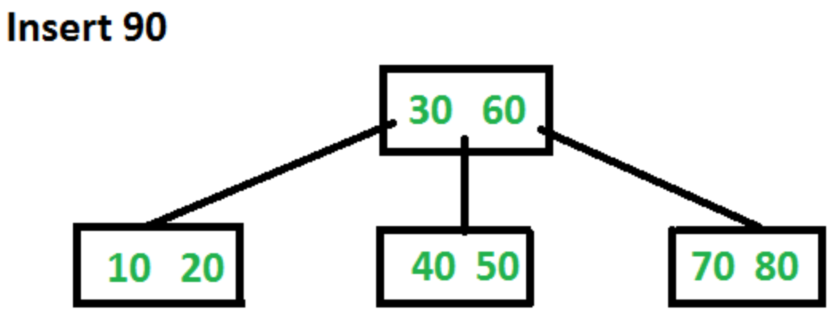
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.



Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



Delete Operation in B-Tree

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).
1.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) 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 x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

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

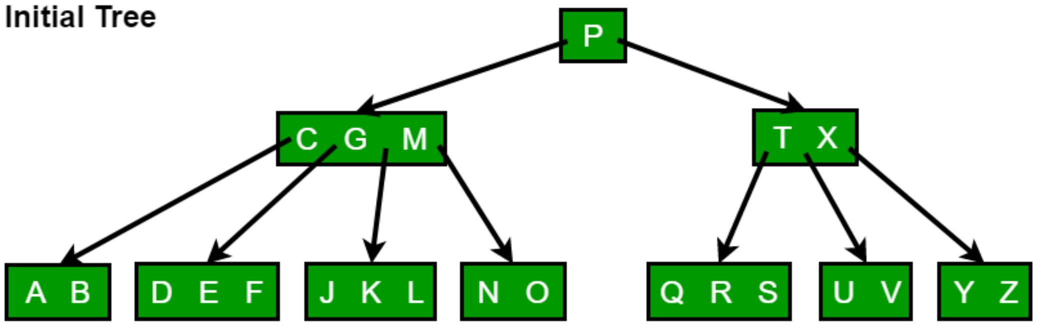
a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

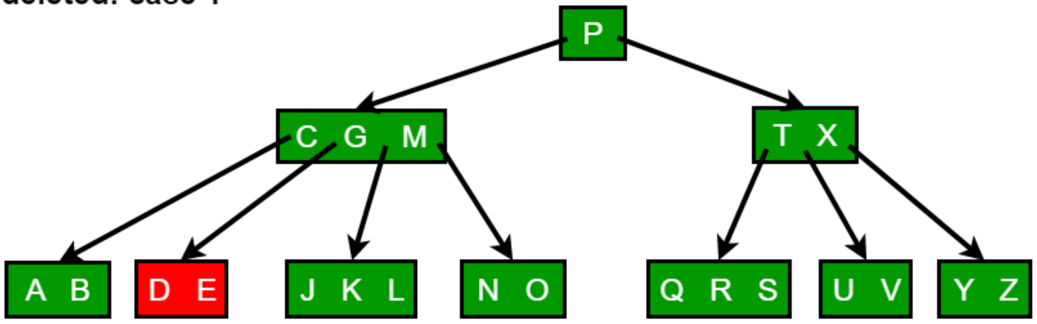
c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor

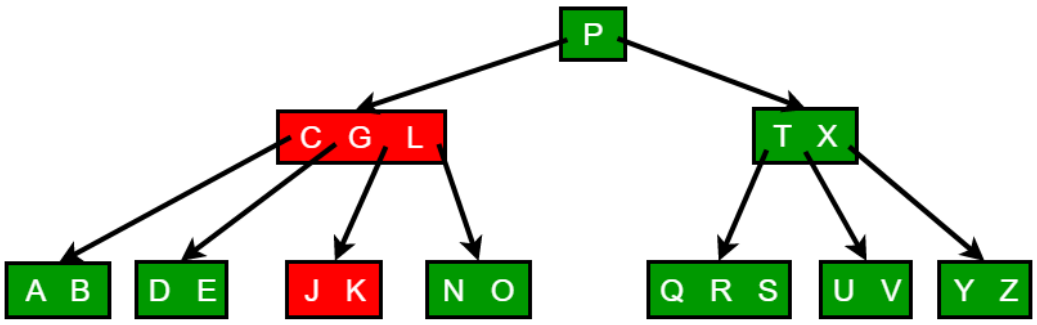
(a) Initial Tree



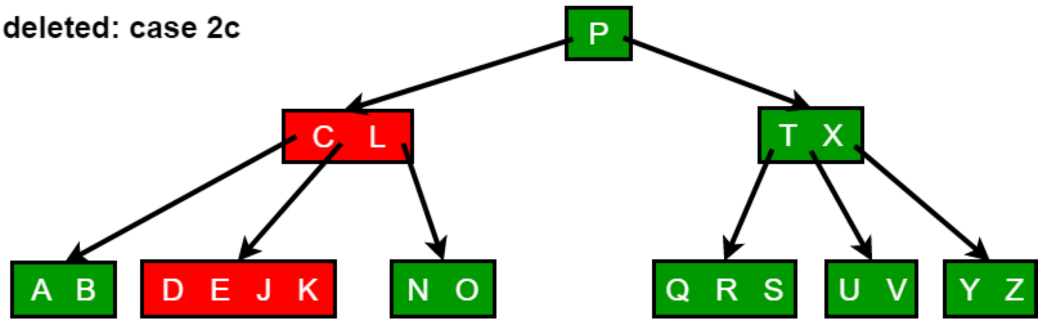
(b) F deleted: case 1



(c) M deleted: case 2a



(d) G deleted: case 2c



P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z