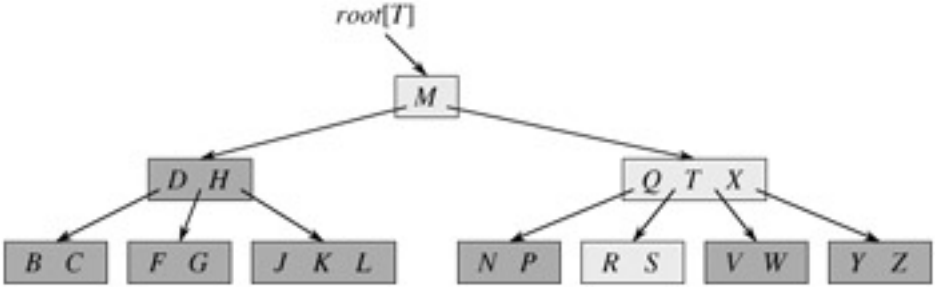
B-Trees (Cormen et. al. Chapter 18, also see Wikipedia: https://en.wikipedia.org/wiki/B-tree)

A balanced tree data structure keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithm time. B-trees are a good example of a data structure for external memory. It is commonly used in file systems and databases.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a handful to thousands, within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (or a **2-3 tree**), each internal node may have only 2 or 3 child nodes. B-trees are similar to red-black trees in that every **n**-node B-tree has height **O**(lg **n**), although the height of a B-tree can be considerably less than that of a red-black tree because its branching factor can be much larger. Therefore, B-trees can also be used to implement many dynamic-set operations in time **O**(log**n**).



A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which:

1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to m children
5. a leaf node contains no more than $m - 1$ keys

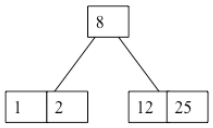
The number m should always be odd.

Example. Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45. We want to construct a B-tree of order 5.

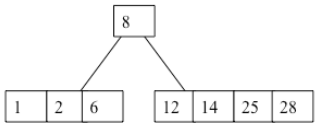
The first four items go into the root:

1	2	8	12

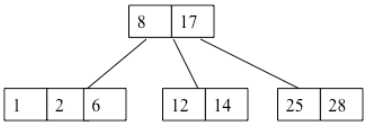
To put the fifth item in the root would violate condition 5; therefore, when 25 arrives, pick the middle key to make a new root.



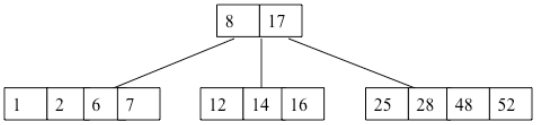
6, 14, 28 get added to the leaf nodes:



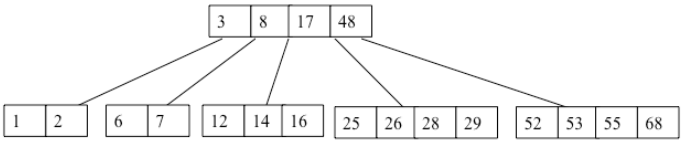
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

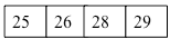


7, 52, 16, 48 get added to the leaf nodes

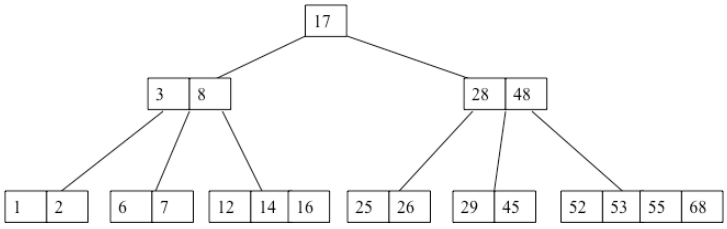


Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves



Adding 45 causes a split of 

and promoting 28 to the root then causes the root to split



	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)

Inserting an element to a B-tree (always to a leaf):

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Deleting an element from a B-tree (always have fewer element in a leaf as new elements will be inserted into leafs)

- If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.
- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf:
    - if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our leaf that needs a key
    - if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

The maximum number of nodes in a B-tree of order m and height h:

root $m - 1$
level 1 $m(m - 1)$
level 2 $m^2(m - 1)$
. . .
level h $m^h(m - 1)$

So, the total number of items is $(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) = [(m^{h+1} - 1)/ (m - 1)] (m - 1) = \boldsymbol{m^{h+1} - 1}$

When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

B-tree is often used for searching data stored in external disks (when the data is too large to be loaded entirely into the main memory). When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred. If we use a B-tree of order 101, say, we can transfer each node in one disc read operation (thus, we can set up the order as the data that can be loaded into main memory). A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory). Note that the depth of balanced BST has a

depth disk reads of $\log_2 10^8 \sim 700$.