**B-Trees and 2-3-4 Trees**

The 2-3 tree was the inspiration for the more general B-tree which allows up to n children per node

The B-tree was designed for building indexes to very large databases stored on a hard disk

The 2-3-4 tree is a specialization of the B-tree

**B-Trees**

Diagram

Description automatically generated

The max number of children is the order of the B-tree, which we represent as the variable “order”

n = 5 (number of children in a B-tree) 🡪 order of the B-tree

The number of data items in a node is 1 less than the number of children (the order)

Other than the root, each node has between order/2 and order children

The data items in each node are in increasing order

h =

The first link from a node connects it to a subtree with values smaller than the parent’s smallest value  
The last link from a node connects it to a subtree with values greater than the parent’s largest value  
The other links are to subtrees with values between each pair of consecutive values in the parent node

B-trees were developed to store indexes to databases on disk storage

* disk storage is broken into blocks
* the nodes of a B-tree are sized to fit in a block
  + If data items are 1KB and block is 1MB, then 1024 items are stored into each B-tree node
    - We keep links in each node too, so order should be kind of 512
* each disk access to the index retrieves exactly one B-tree node
* the time to retrieve a block off the disk is large compared to the time to process it in memory
* by making tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index

Assuming a block can store a node for a B-tree of order 200, each node would store at least 100, at most 199 items

This enables 100 million items to be accessed in a B-tree of height 4

Usually 4-5 levels are enough for a B-tree structure, so disk accesses are very few

Insertion

Diagram

Description automatically generated

Diagram

Description automatically generated

If a leaf to receive the insertion is full, it is split into 2 nodes, each containing approximately half the items, and the middle item is passed up to the split node’s parents

If the parent is full, it is split and its middle item is passed up to its parent, and so on

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Root has just 1 element, it is possible for the root only. For other nodes, minimum is 2 (n/2).

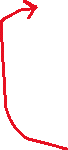
**IMPLEMENTING THE B-TREE**

Text, letter

Description automatically generated

Text, application, email

Description automatically generated



Text

Description automatically generated

Graphical user interface, text

Description automatically generated

We made root to leaf traversal. At each node on this path, we perform a search operation and follow the link until the leaf. When going back on this recursion tree, we make the corrections, new child parent insertions. These insertions can be done in constant time if the place is available. But to be able to put the element in between the array, everything should be shifted at the right. Running time depends on the location and number of elements available in this node (n/2 🡨🡪 n-1).

For these operations, it requires O(order) time. If we consider order to be constant, all search and insertions are constant time. So overall running time depends on the height of the tree (logarithmic) :

🡪 n: number of elements

Since order is big, number of accesses is quite low.

“index” in the insert method above is the position in the data array

if the root is null part is considered in helper public method which calls insert method above with just an item and it is public.

Diagram

Description automatically generatedText, letter

Description automatically generated

Text, letter

Description automatically generated

**REMOVAL**

Removing an item is a generalization of removing an item from a 2-3 tree

The simplest removal is deletion from a leaf

When an item is removed from an interior node, it must be replaced by its inorder predecessor (or successor) in a leaf

If removing an item from a leaf results in the leaf being less than half full, redistribution needs to occur

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

If we have 16, 16 (middle element) would go up next to the 10 and we are done but here:

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

**B+ Trees (extension of B-Trees)**

The B-tree was developed to create indexes for databases

* the Node is stored on a disk block
* the Node pointers are pointers to disk blocks instead of memory addresses
* the E is a key-value pair where the value is also a pointer to a disk block

Since in the leaf nodes all child pointers are null, there is a significant waste of space

A B+ tree addresses this wasted space

In a B+ tree,

* the leaves contain the keys and pointers to their corresponding values
* the internal nodes contain only keys and pointers to the children
* the parent’s value is repeated as the first value
* there are order pointers and order values (number of pointers and number of values are same)
  + in B-tree number of children is 1 more than the number of items

Diagram

Description automatically generated with medium confidence

1 node for level 1,  
2 nodes for level 2,  
6 nodes for level 3

At the leaf level you can see all the keys.

If you want to see corresponding value for the key 5, you have to look leaf 5’s value. Inner 5s hold address of next B+ node.