9. B tree

1. Multiway balanced tree

Multiway balanced tree refers to a tree structure in which each node can have multiple child nodes, and the overall height of the tree is kept relatively balanced to ensure efficient lookup, insertion, and deletion operations. B-tree is a specific type of Multiway balanced tree that is widely used in disks and other external storage devices because it can efficiently manage large amounts of data and supports fast lookup, insertion, and deletion operations.

2. B tree

B-tree is a balanced tree data structure that is commonly used for efficient storage and management of large amounts of data on disk or other external storage devices. The key feature of a B-tree is that its **nodes can contain multiple keys**, and **the size of each node can be adjusted to fit the characteristics of the storage medium**. Each node in a B-tree has a list of keys and a list of pointers to child nodes, which can be leaf nodes or non-leaf nodes. The keys in each node are sorted in ascending order, and the number of keys in each node is within a predefined range, typically designed to be a multiple of the disk block size.

The **degree** of a B-tree is a fixed integer t greater than or equal to **2**. More specifically, the properties of a B-tree of degree **t** are:

1. The root must have at least 1 key, and at most $2t - 1$ keys.
2. Every node other than the root must have at least $t - 1$ keys, and at most $2t - 1$ keys.
3. If the root is not a leaf node, it must have at least 2 children.
4. Every internal node other than the root must have at least t children, and at most $2t$ children.

For example, if we have a B-tree of degree $t = 3$, then the root node has at least **1** key, at most **5** keys; each non-root node must have at least **2** keys and at most **5** keys; and each internal node must have at least **3** children and at most **6** children.

Note that the degree of a B-tree is related to its balance and height. A larger degree can lead to a shorter tree height and faster access times, but it also requires more memory per node.

3. Basic operations

The search operation in B-tree is similar to a binary search tree, but it can handle large amounts of data more efficiently. The **search** starts from the root node, and each node's keys are checked to determine which child node to search next. This process continues until the target key is found or it is determined that the target key does not exist.

The **insertion** operation in B-tree involves splitting and merging nodes to maintain the balance of the tree. Specifically, when a new key is inserted, the node where the key should be inserted is identified, and if the node is full, it is split into two nodes. The median key in the original node is promoted to the parent node, and the new node is inserted as a child of the parent node. This process is repeated recursively until the root node is reached, and if the root node is full, it is split into two nodes, and a new root node is created.

4. Exercises

1. Decide if a tree is a B tree or not:

See the book *Introduction_to_algorithms_3rd_edition*, **p.488-18.1 Definition of B-trees**

To keep things simple, we assume, as we have for binary search trees and red-black trees, that any "satellite information" associated with a key resides in the same node as the key. In practice, one might actually store with each key just a pointer to another disk page containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a ***B+-tree***, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:
   a. $x.n$, the number of keys currently stored in node x,
   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,
   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

   $$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

$t = 2$

ins: 1, 7, 5, 8, 10, 14, 12, 11, 15, 16, 17