Database systems: LabExercise

Khalid Belhajjame

B Trees

For this assignment, you are requested to implement a B Tree index structure.

A B–tree of *order* m (m ≥ 3) is an m-way search tree that is either empty or of height h ≥ 1 and satisfies the following properties:

1. Each node contains at most m – 1 keys.
2. Each node, except the root, contains at least ⌈m/2⌉ – 1 keys. The root may contain any number n of keys with n ≤ m – 1.
3. A node is either a leaf node or has j + 1 children, where j is the number of keys of the node.
4. All leaves appear on same level.
5. Each node has the following structure: $[p_0, k_1*, p_1, k_2*, p_2, \dots k_j*, p_j]$ where:
   - The keys are sorted: $k_1 < \dots < k_j$.
   - p_i is a pointer to another node of the tree structure, and is undefined in the leaves.
   - Let $K(p_i)$ be the set of keys stored in the subtree pointed by p_i. For each non-leaf node, the following properties hold:
     - $\forall y \in K(p_0), y<k_1$
     - $\forall y \in K(p_i), k_i <y<k_{i+1}, i=1,\dots j-1$
     - $\forall y \in K(p_j), y>k_j$

The *height* h of a B–tree is the number of nodes in a path from the root to a leaf node.

Search. The search of a key k starts at the root node. If the key is not in the root, and h > 1, the search continues as follows:

1. If $k_i < k < k_{i+1}, 1 \leq i \leq m$, then the search continues in the subtree p_i.

2. If $k_m < k$, then the search continues in the subtree p_m.

3. If $k < k_1$, then the search continues in the subtree p_0.

If the key value is not found in a leaf node, the search is unsuccessful, otherwise the search cost is ≤ h.

Insertion. The insertion of a key k into a B–tree is also quite simple. First, a search is made for the leaf node which should contain the key k. An unsuccessful search determines the leaf node Q_1 where k should be inserted.

If the node Q_1 contains less than $m - 1$ keys, then k is inserted and the operation terminates. Otherwise, if Q_1 is full, it will be split into two nodes, with the first half of the m keys that remain in the old node Q_1, the second half of the keys that go into a new adjacent node Q_2, and the median key, together with the pointer to Q_2, that is inserted into the father node Q of Q_1, repeating the insertion operation in this node. This splitting and moving up process may continue if necessary up to the root, and if this must be split, a new root node will be created and this increases the height of the B–tree by one.

Note that the growth is at the *top* of the tree, and this is an intrinsic characteristic of a B–tree to ensure the important properties that it always have all the leaves at the same level, and each node different from the root is at least 50% full.

Alternatively, if an adjacent brother node of Q_1 is not full, the insertion can be performed without splitting Q_1 by applying a *rotation* technique, as explained for the deletion operation.

Deletion. Key deletion is slightly more complicated. If the key to be deleted does not occur in a leaf, we replace it with the largest value in its left subtree and then proceed to delete that value from the node that originally contained it. In a B-tree, the largest value in any value's left subtree is guaranteed to be in leaf. Therefore wherever the value to be deleted initially resides, the following deletion algorithm always begins at a leaf.

Furthermore, after a deletion if the leaf node p has less than $\lceil m/2 \rceil - 1$ keys, it has to be regrouped with an adjacent brother node in order to respect the definition of B–tree, using one of the following techniques: *merging* or *rotation*.

The node p is *merged* with one of its adjacent brother nodes which contains $\lceil m/2 \rceil - 1$ keys operating in a way that is exactly the inverse to the process of division.

Merging is illustrated by Figure 1. If key 70 is deleted from node Q_1, it becomes underfull and it is merged with the brother to the right Q_2. The key 71 separating the two nodes in the ancestor Q is no longer necessary and it too is added to the single remaining leaf Q_1, so the tree will become that shown in Figure 2.

The elimination of 71 from Q can cause a further underflow by requiring the merging of Q with one of its adjacent brothers. In such a case, the process is applied recursively and terminates upon encountering a node that does not need be merged or if the root node is used. If the root node contains a single key, as a result of the merging, it becomes empty, and is removed. The result is that the B–tree shrinks from the top. Thus the deletion process reverses the effects of the insertion process.

When the *merging* of the node p with one of its adjacent brothers is not possible, then the *rotation* technique is applied.

Rotation is illustrated by Figure 3a. If key 70 is deleted from Q_2, it becomes underfull and a rotation is performed to borrow the maximum key 65 from the brother to the left Q_1. The key 65 is moved in the ancestor node Q and replace the key 66 which is moved in Q_2 as the new smallest key value. The tree will become that shown in Figure 3b.

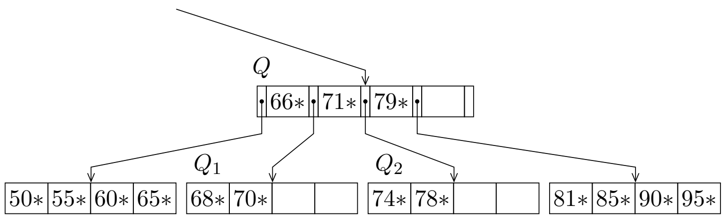

Figure 1: Example of a B-Tree.

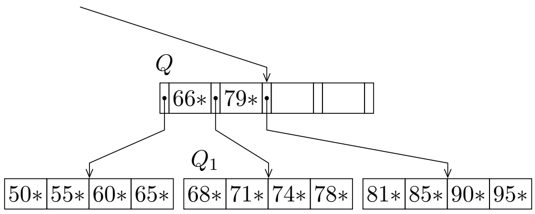

Figure 2: The B-Tree illustrated in Figure 1 after the deletion of the record 70*

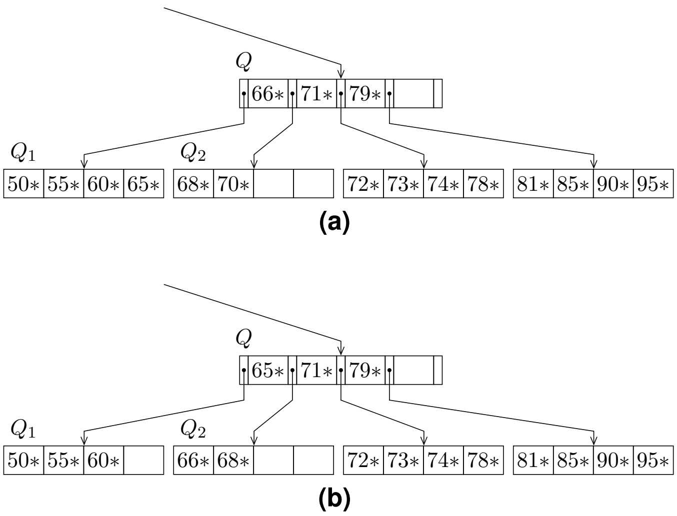

Figure 3: A rotation example

Data Loading. The initial B–tree structure depends on the order in which the keys are loaded. For example, the B–tree in Figure 4 is the result of loading the keys in the following order:

10, 15, 30, 27, 35, 40, 45, 37, 20, 50, 55, 46, 71, 66, 74, 85, 90, 79, 78, 95, 25, 81, 68, 60, 65.

If the keys to load are sorted, the result is a B–tree with the leaves filled at 50%, except the last one, as shown in Figure 5 for the keys: 50, 55, 66, 68, 70, 71, 72, 73, 79, 81, 85, 90, 95.

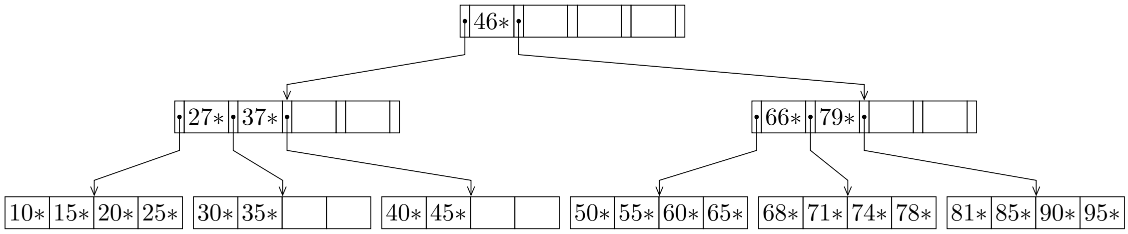

Figure 4. The resulting B–tree from loading a non sorted set of keys

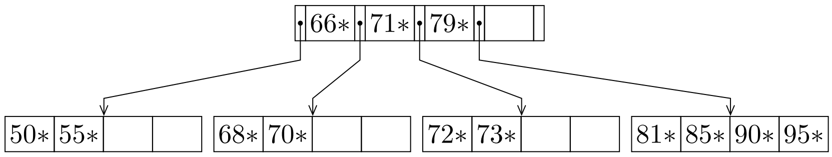

Figure 5. The resulting B–tree from loading a sorted set of keys

Requestd work

Your job is:

1. Understand B tree, and the way the search, insertion and deletion operations work
2. Implement the search
3. Implement insertion operation.
4. Devise test cases to ensure that the operations you implemented behave correctly.

Logistics

You will work in teams of two. Please find quickly some one to team up with.