BTree Implementation for Student Database

CPE-593 Final Project Report

By: Zachary Fazal & Zachary Salisbury

I pledge my honor that I've abided by the Stevens Honor System.

-Zachary Fazal, Zachary Salisbury

Table of Contents

Abstract

Managing and accessing information among large sets of data, such as those in databases can pose serious challenges. Most notably, many data structures have a difficult time maintaining efficiency in accessing and modifying information that resides in a large data set. In this paper, the background and makeup of a BTree data structure will be discussed. Efficiency difficulties previously mentioned will be addressed, as well as how implementing a BTree data structure can help alleviate these issues. Lastly, an application of BTree to structure and access information from a dataset of student records/information will be demonstrated. Specifically, the database information used, the BTree structure created by the group, and results will be presented.

Introduction

Tree structures and searches became a very commonly used method of structuring data in the mid-20th century. The first of its kind, the binary search tree was conceived in 1960 by P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hubbard as a way to provide faster lookups, insertions, and deletions of data [1]. Binary search trees rely on the principle that the data structure behaves as a tree with connected nodes, where each parental node has 2 children, extending from an original root node all the way to the final level containing leaf nodes. Data which is stored as a key within a node is also sorted in a hierarchy within the tree, which allows traversal down a branch of the tree rather than traversing through an entire data set. By creating algorithms for the structure that didn't require traversing the entire data set, an efficient structure was established with an average runtime efficiency of O(log(N)) for all functions, and a worst-case runtime efficiency of O(N) [1].

However, as datasets became larger and more complex, the use of a binary search tree in some (and eventually many) cases became insufficient in providing efficient data sorting and searching. Specifically, the limiting factors of a binary search tree such as using memory to perform operations or having a maximum of only 2 children per parental node made it difficult to implement into databases containing large sets of data. As a result, Rudolf Bayer and Edward McCreight theorized and created a more consistently efficient structure designed to handle large sets of sorted data in 1970 known as a BTree [2]. The BTree built off of the principles of the binary search tree, but maximized efficiency with volume by allowing parental nodes to have up to M children

instead of 2 and compacting nodes with multiple keys. The outcome was a data structure that now had logarithmic efficiency as both an average and worst-case scenario, and could handle much larger data sets since it can be used with disks/files instead of memory [2].

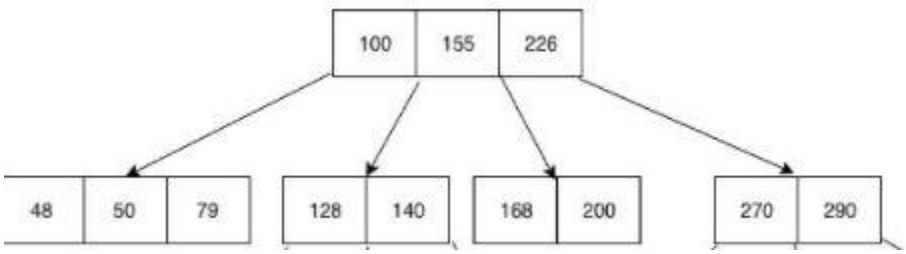
Theory

_____As previously mentioned, standard BTrees build off principles of a binary search tree structure, but have certain qualities that make it not only unique, but improved. To be defined as a BTree, the tree structure must satisfy the following properties (we assume in this case the BTree is of order m):

- Every node is allowed up to m children
- The order m of the tree is consistent with the maximum number of children per node
- If the node is not a leaf node nor a root node (known as an internal node), the node may have a minimum of m/2 child nodes and a maximum of m child nodes
- Non-leaf nodes contain one less keys than the number of children it has
- The root node must contain a minimum of two child nodes if doesn't act as a leaf node
- Leaf nodes all appear in the same level of the tree, but do not carry information*

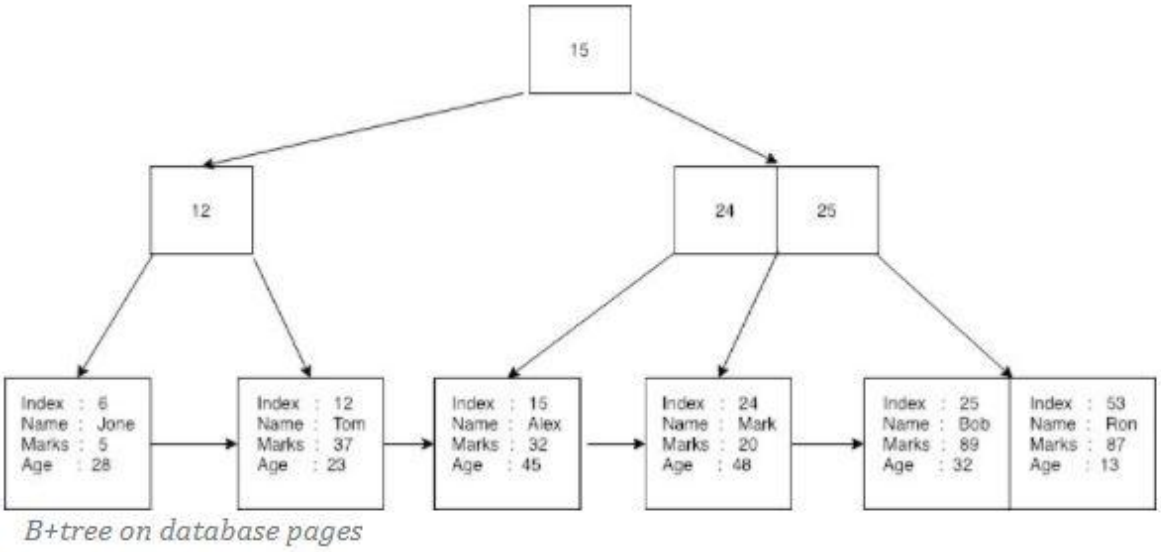
For purposes of this paper, leaf nodes are defined as nodes which exist one level below the lowest level key-carrying nodes*. Furthermore, internal nodes, which we previously defined as non-leaf and non-root nodes, play an important role in BTrees. They not only carry keys, but help sort the tree using values associated with a key known as separation values. Generally, the separation value is the median of the range of values of the child nodes, and must be an existing value associated with a key stored in the tree.

Figure 1



Examining figure 1, the internal node contains the three keys associated with separation values 100, 155 and 226. The four children of this internal node are divided such that values to the left of the separation value will be smaller, and those to the right will be larger (48,50, and 79 are left of 100, while 128 and 140 are right of 100). If figure 1 is to be assumed as a standard BTree with the bottom row being leaf nodes, then none of those values would contain keys. However, using BTree variations such as a B+Tree or B/B+Tree with Indexing are different in that leaf nodes may not only store keys, but also pointers to adjacent leaf nodes to permit sequential access to information. Copies of the keys that reside in the leaf nodes are used as internal nodes, with a key's specific value serving as a separator value. An example of this structure is shown in figure 2 below (note that the tree can be sorted by variables other than the index that reside in the key).

Figure 2 [3]



There are three main functions/operations involved with a BTree structure: search/traversal, insertion and deletion. For the most part, these operations mimic those done in other tree structures such as a binary tree, but there are some notable differences that will be discussed. The reason that BTree or BTree variations are so popular to implement in large data sets/databases is that for all three operations, both the average and the worst-case runtime are on the order O(log(N)) [3].

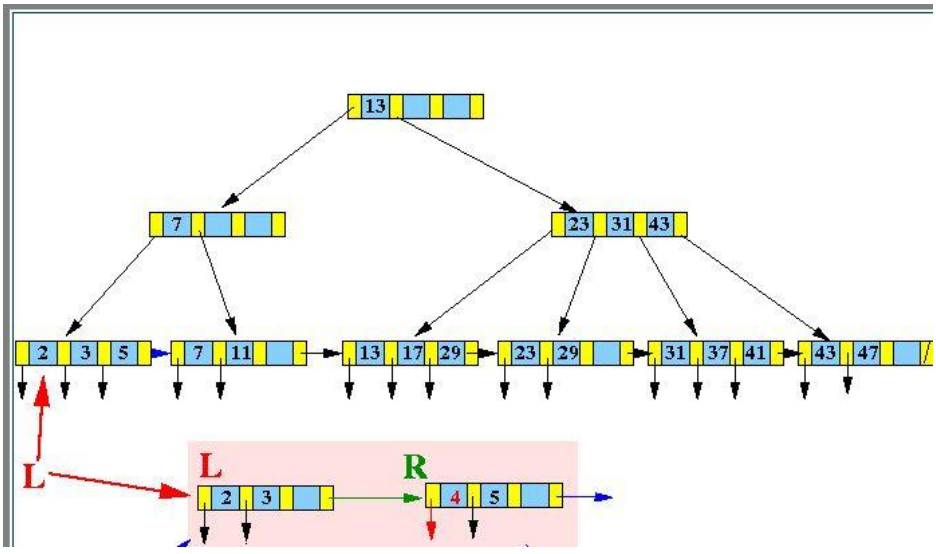
Search Algorithm

Starting at the root node of the tree, the search algorithm recursively traverses the tree from root to leaf. Following each iteration, the “field of view” for the search reduces to the next subtree that satisfies the value range associated with the key being searched. The root of the subtree in the search is replaced each time with the node that satisfies the correct range that the key lies in. The operation is repeated and the “field of view” is continually narrowed until a leaf node is reached, which either will or will not point to a match for the key being searched. Because the algorithm is performed recursively and thus follows a single branch of the tree, the runtime efficiency of this algorithm is on the order of O(log(N)), where N is the number of elements in the tree.

Insertion Algorithm

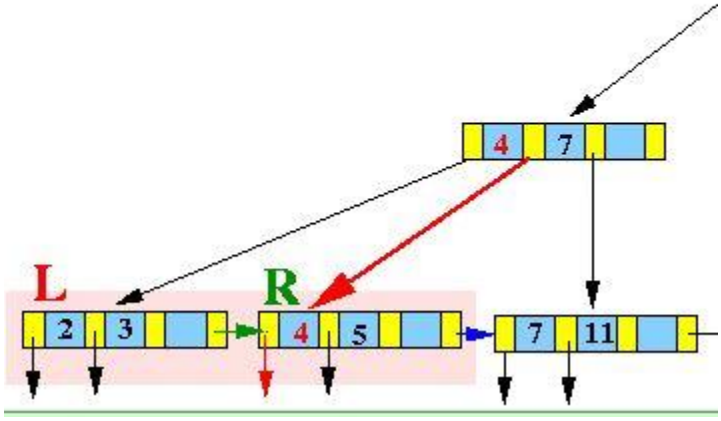
To insert a new element into the BTree, a traversal or search must be done to determine the location in the tree for the new key. Once the appropriate location is reached, there are two scenarios that can transpire during insertion: the node that the key will be inserted into isn’t full, or the node is full (also referred to as an overflow condition). If the node is **not** full, the key is inserted into the node (between adjacent lesser and greater keys) and the algorithm is finished. If the node **is** full and no additional keys can be added, the node must be split in order to make space for the new key. For example, for the B+Tree in figure 3 below, the key associated with value 4 is trying to be inserted into the node with keys 2,3,5, but since the maximum key size has been reached in that node, the node must be split into two child nodes.

Figure 3 [4]



Once the node is split up, the tree must reorganize itself to maintain symmetry, which in some cases entails altering key separation values. In figure 3, the addition of a key with value 4 would result in 3 children stemming from the parent node containing one key with value 7, which by rule is not permitted. To rectify this, an additional key with a separation value is added to the parent node, which is shown in figure 4 below. Subsequent splits can occur at levels all the way up to and including the root if the nodes are full or if rules are not met, such as non-leaf nodes containing one less key than children. As with the search algorithm, the average and worst-case runtime efficiency of this algorithm is also on the order of O(log(N)).

Figure 4 [4]



Deletion Algorithm

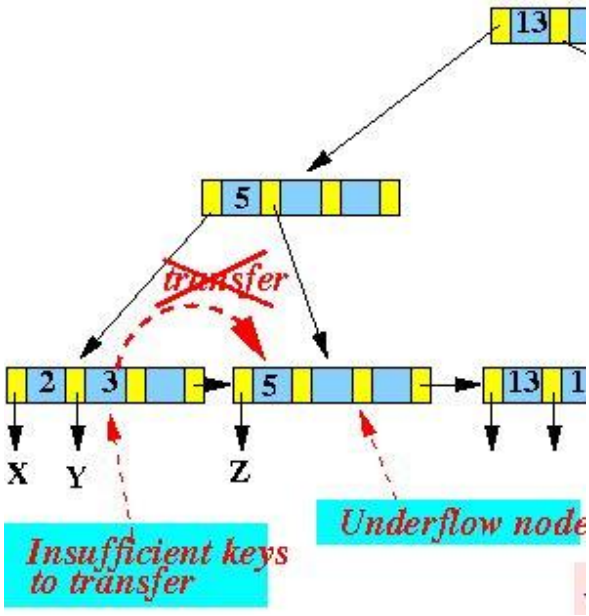
As with insertion, deletion also requires traversing the BTree until the correct key is reached. In a standard BTree, removing a key from the tree will remove it entirely from the tree. However, in a B+ Tree or BTree Index, deleting a key will delete the key, but not the index associated with the key if the index is being used in an internal node as a separation variable.

When deleting a key from a standard BTree, there are two scenarios that may occur: deleting a key whose value is located at a leaf, or deleting a key at an internal node. Deleting a search key whose value is located at a leaf node is simple if an underflow condition does not occur, which will be discussed shortly. On the other hand,

deleting a key located at an internal node of a standard BTree entails extra steps. First, a new key must take its place and serve as a separation value. This median value is usually the highest value from the left child, or the lowest value from the right child. If this is completed without any nodes reaching an underflow condition, the deletion is complete.

If a search key is deleted from a node and the node falls below the required number of keys, an underflow condition occurs. The result of an underflow condition is either the transfer of a key from the sibling node (which would change the parent separation value), or the merging of sibling child nodes if there aren't enough keys to be able to transfer. The result of a merging of nodes creates a child node that meets capacity requirements, and removes any pointers associated with the now-empty node. Figure 5 (shown below) demonstrates this process with the key pointing to value 11 having been removed from the tree.

Figure 5 [5]



Following a merge between sibling nodes, it may be necessary to merge internal nodes at either one or multiple levels if the parent node has an underflow of child nodes. Doing so would require the same transfers or merges, but in this case with internal nodes. Merges and transfers of keys in the BTree can occur all the way up to and including the root node in order to rebalance the tree.

Application and Implementation

Although there are a plethora of online resources that detail how one can code a BTree using C++, these examples are built with little attention toward efficiency. They consistently make the mistake of copying data when passing it as function arguments, rather than referencing it. They also ignore using block size to decide what degree the BTree should have in order to work at an optimal level. What is even more disappointing is how basic and useless the applications are for these online examples. They all deal with storing integer values. To transform these BTrees to an efficient and useful one proved to be a tall task. They did supply us with the basic class structure involved with making a BTree: a BTreeNode class and a BTree class, which acted as a friend to the former. They were also helpful in supplying useful code for the underlying algorithms behind BTree traversals, searches, and most importantly insertions and removals. Insertion utilized three different functions to insert new values into the BTree: insert, insertnonfull, and splitchild. Remove was the most complex action utilizing nine different functions: remove, removefromleaf, removefromnonleaf, getpred, getsucc, fill, borrowfrompred, borrowfromnext, and merge. The actual uses of each of these functions are pretty self explanatory, and the algorithms behind them are covered in the above section. Despite the fact that we were able to start off already having this code, a thorough understanding of exactly how each function worked was essential as we reformatted them to be able to support a student database.

Primary BTree

Our first objective was to create a BTree capable of storing student information rather than just integers. To do this we created a struct called student that contained the following variables: first name, last name, ID number, date of birth, major, and GPA. It was crucial for the BTree to sort the data, so a “primary key” needed to be specified so the BTree knew what variable to look at when sorting students. Rather than hard coding the BTree to only look at a specific variable, we opted to make it generalized so that it would be capable of sorting based on any given variable within the struct.

To accomplish this goal, utilizing templates was crucial. Our template was composed of Record, PrimaryKeyType, and PrimaryKey. Record represented the object being stored in the BTree (in this case students), PrimaryKey represented the variable that the Records should be sorted by, and PrimaryKeyType represented the variable type of the PrimaryKey. The deployment of these templates throughout all BTree and BTreeNode methods yielded a generalized BTree capable of sorting any given struct by any of its given variables.

To further improve efficiency, any method taking a Record or PrimaryKeyType as a parameter was altered to take references of these data types instead in an effort to

avoid making unnecessary copies of data in memory. Additionally, to find the optimal degree of the BTree we had to use block size to determine how many records could fit in a single block. This would be the max number of records per node of the BTree.

Figure 6

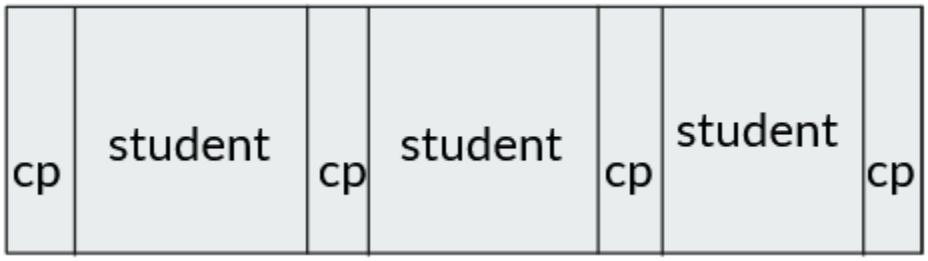


Figure 6 shows the structure of a node with three records. Because pointers are 8 bytes, and there will always be one more child pointer than records, the equation in Figure 7 is able to calculate how many records can fit in one block.

Figure 7

```
constexpr static uint32_t rec_per_block = (blocksize - 8) / (sizeof(Record) + 8);
```

On Windows 10 the block size is 4096 bytes and on Linux the block size is 1024 bytes.

Index BTree

Although this primary BTree can hold any given struct, sort them by any given variable of that struct, and can perform searches, insertions, and removals all in log(n) time, it still has some shortcomings. The BTree can only locate a record in log(n) time if the specified primary key of that record is given. For example, if you have a Btree of students sorted by ID number, you can only search for students by their ID number. In order to quickly look one up by their last name, you would have to create a whole other BTree, this time sorted by last name. This would take up a lot of memory. And if you wanted to be able to look up students based on other variables as well you would have to create even more BTrees, taking up even more memory. So the way to get around this problem is to create an Index BTree that fills its nodes with pointers to the students in the primary BTree rather than with the students themselves. This saves an incredible amount of memory, but is very complicated to pull off. It is crucial to the integrity of the index BTree that its pointers are adjusted for any insertion or deletion of a record in the primary BTree, otherwise it will contain pointers to random places in memory and will serve zero purpose. It is important to note that when insertions and deletions take place the records and nodes are subject to some very complex and extensive manipulation, particularly when removing from non-leafs, performing mergers, or splitting children.

The only way we could think of creating this Index BTree was to create two new classes for it: Index_BTree and Index_BTreeNode. These classes would contain

methods practically identical to those in BTree and BTreeNode with a few modifications. These modifications would account for three things. First, the index nodes would be containing pointers to records rather than just records. Second, the index BTree would be sorting its record pointers based on a different variable than the primary BTree (which sorts based on the template argument: PrimaryKey). To account for this, two new template arguments needed to be added: IndexKeyType and IndexKey. IndexKey would be the variable that the index BTree would use to sort, and IndexKeyType would be the data type of the IndexKey. Third, since the size of a pointer to a record is less than the size of a record, the equation to determine the number of elements that could fit in a block needed to change to for the index BTree. Figure 8 shows the new equation.

Figure 8

```
constexpr static uint32_t rec_per_block = (blocksize - 8) / (8 + 8);
```

The Index BTree is initialized in the BTree class right when the primary BTree is initialized. Consequently, when initializing the primary BTree in main, the keys that each BTree will be sorting by must be included along with their data types and the struct. Figure 9 shows a BTree named t being initialized in main. It will produce a primary BTree of students sorted by their ID number and an index BTree of pointers to those students sorted by their last name.

Figure 9

```
BTree<student, int, &student::ID, string, &student::lastName> t;
```

Creating an Index BTree is the easy part. The hard part is making sure it can account for any change in the primary BTree. Any time a single key's location in the primary BTree changes, it must be accounted for in the index BTree, otherwise the records of the index BTree will be in the wrong order, rendering it useless. To accomplish this we had to be able to obtain the exact location of a given record pointer within the index BTree. To do this, the search method of the Index_BTree class had to be modified to return a struct called "location_in_IndexBTree" containing two values: node (a pointer to the Index_BTreeNode) and index(the index of the intended record pointer in the node). Now any time a key in the BTree was moved to a new location it could be accounted for. Figure 10 shows the code that does this.

Figure 10

```
location_in_IndexBTree<Record, PrimaryKeyType, PrimaryKey, IndexKeyType, IndexKey> i_tree_key_loc = i_tree.search(keys[i].*IndexKey);
Index_BTreeNode<Record, PrimaryKeyType, PrimaryKey, IndexKeyType, IndexKey> i_node = *i_tree_key_loc.node;
i_node.keys[i_tree_key_loc.index] = &keys[i+1];
keys[i + 1] = keys[i];
```

Initially, the method that this code is taken from was trying to make the value of keys[i + 1] equal to the record located at keys[i] (line 4). Now, before this happens, the location of keys[i] in the index BTree is found (line 1). Then, the pointer at that location is changed to &keys[i + 1] (line 3). So now after all 4 lines are completed, although the

pointer in the index BTree has changed, it will still be pointing at the same Record, thus maintaining the order in the index BTree.

For insertions, the insert function of the index BTree is called at the end of the insert function of the primary BTree. Deletions are slightly more complicated. First off, the removal of a record must be done to the index BTree before it is done to the primary BTree, otherwise (due to the pointer manipulation shown in Figure 10 being applied to all of the removal algorithms) the index BTree will end up containing two of the same record and no longer contain the record that needs to be removed. It will instead be called to remove one of the duplicates, but will not know which one to delete, which can lead to some obvious problems. Additionally, during the removal process there can be many records that change positions, including the one that needs to be removed. In this case, because the removal happens in the index BTree first, when line 1 in figure 10 is reached, it will return null, as the record will no longer exist in the index BTree. This creates a null pointer exception in line 2 of Figure 10. To combat this, in all of the removal functions, any time a key changes position and a search for it is conducted in the index BTree, it includes an if statement to see if the search returned null. This can be observed in Figure 11:

Figure 11

```
location_in_IndexBTree<Record, PrimaryKeyType, PrimaryKey, IndexKeyType, IndexKey> i_tree_key_loc = i_tree.search(keys[idx].*IndexKey);
if (i_tree_key_loc.node != nullptr) {
    Index_BTreeNode<Record, PrimaryKeyType, PrimaryKey, IndexKeyType, IndexKey> i_node = *i_tree_key_loc.node;
    i_node.keys[i_tree_key_loc.index] = &(child->keys[minRec - 1]);
}

child->keys[minRec - 1] = keys[idx];        //key change
```

If the search does return null, then the record being moved has already been deleted from the index BTree, so the index BTree is left alone.

Results and Conclusion

We were able to successfully create a templated BTree capable of handling any given struct sorted by any given variable. We were also successful in implementing an index BTree that could account for any changes made in its primary BTree. In our repo we included code for just the BTree which can perform searches, insertions, and deletions in log(n) time. We also included code for the BTree with Index BTree which can carry out searches in log(n), but performs insertions and deletions in log(n)^2 time. This is because in order to account for changes in the primary BTree, every time a record is moved a search must be conducted to locate where the index BTree must be adjusted. Although we would have hoped to accomplish this task in log(n) time, we still recognize the advantage of being able to search a database using more than just a single type of key without doubling or tripling the amount of memory required (which would be the case if more regular BTrees were created rather than index BTrees). The BTree and IndexBTree code also includes a simple little demo in main where a

database of 100 students is created by reading data from the CSV “Student Database Data Table” and, to show off the BTree’s generalization, a second database of 100 faculty members is created by reading data from the CSV “Faculty Database Data Table.” The user can play around with the databases and utilize the different functions. This project taught us a lot more than just how BTrees work. We also gained valuable insight in the utilization and capabilities of templates and dealing with pointers and references in C++.

References

1. https://en.wikipedia.org/wiki/Binary_search_tree
2. https://en.wikipedia.org/wiki/B-tree#In_filesystems
3. https://dzone.com/articles/database-btree-indexing-in-sqlite#:~:text=The%20index%20is%20the%20reference,time%20and%20gets%20the%20record.
4. http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/B-tree=insert.html
5. http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/B-tree=delete.html
6. https://dzone.com/articles/database-btree-indexing-in-sqlite
7. https://cstack.github.io/db_tutorial/parts/part7.html
8. https://use-the-index-luke.com/sql/anatomy/the-tree
9. https://www.geeksforgeeks.org/delete-operation-in-b-tree/