Vu Nham Nguyen Nguyen

Data Structure and Algorithm

B TREE

SEMINAR REPORT

Table of content

[**I.**](#_heading=h.gjdgxs) 132

[**II.**](#_heading=h.30j0zll) 2

[**III.**](#_heading=h.1fob9te) 2

[**IV.**](#_heading=h.3znysh7) 132

[**V.**](#_heading=h.3znysh7) 133

[**VI.** **APPLICATION** 1](#_heading=h.tyjcwt)4

[**VI.**](#_heading=h.tyjcwt) **REFERENCE** 16

1. **INTRODUCTION**

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like [AVL](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/) and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.

1. **HISTORY**

B-trees were invented by [Rudolf Bayer](https://en.wikipedia.org/wiki/Rudolf_Bayer) and [Edward M. McCreight](https://en.wikipedia.org/wiki/Edward_M._McCreight) while working at [Boeing Research Labs](https://en.wikipedia.org/wiki/Boeing), for the purpose of efficiently managing index pages for large random access files. The basic assumption was that indexes would be so voluminous that only small chunks of the tree could fit in main memory. Bayer and McCreight's paper, Organization and maintenance of large ordered indices, was first circulated in July 1970 and later published in [Acta Informatica](https://en.wikipedia.org/wiki/Acta_Informatica)

Bayer and McCreight never explained what, if anything, the B stands for: Boeing, balanced, broad, bushy, and Bayer have been suggested. McCreight has said that "the more you think about what the B in B-trees means, the better you understand B-trees."

1. **OPERATION**
2. **Search Operation in B – Tree**

Searching in B – Tree is similar to the search in Binary Search Trees. If the key is found, we will print out “The key exists”. Otherwise, if the current position is leaf node and does not contain the key which we need, we just inform “The key does not exist”.

**Algorithm:** **You want to find x**.

* Start searching from the root and recursively traverse down the tree.
* Within a node having n-1 values and n children, indexed as value[**0**], value[**1**], … up to value[**n-2**] , using loop:
  + Find the position I that key >= value[i]
  + If value[i] = key, print out the result.
  + If value[i] not equal to key, traverse down the child node of value[i]
* If k is not found in the tree, print out nothing.

**Example and demo:** Search 60 in B – TreeChart, line chart

Description automatically generated

Chart

Description automatically generated

**Code**

Graphical user interface, text, application

Description automatically generated

**Run code:**

Graphical user interface, text

Description automatically generated with medium confidence

1. **Insert Operation in B – tree**

Insertion is an operation to add a new value to the tree. During the insertion, the tree can grow in height if it satisfies the conditions. Unlike Binary Search Trees, we have a predefined range on the number of keys that a node can contain. Therefore, insert operation consists of two cases:

* Case 1: The node is not full.
* Case 2: If the node is full then split it before inserting (Preemptive Split).The operation splits the node in half and moves a key up; hence this is the reason why B – tree grows up and Binary Search Trees grow down.

1. **Preemptive Split**

**Algorithm**

To insert a new key x, we do these following steps:

* Check if root is null, we initialize x as root and then return. Otherwise, checking the number of keys in the root whether it is full or not, if it is full, we will split the root. On the other hand, we search for an appropriate child node to insert.
* Update number of keys after inserting.
* After searching an appropriate child node, we check the number of keys in that node. If the node is full, we will split this node into two halves and one middle key.
* Push the middle key upwards and add it to the parent node. Then, we check these two new children nodes and insert the key into the appropriate child.
* If the node is not full, we insert the key into the node in ascending order.

Let us understand this algorithm with an example tree of minimum degree “t” as 2 (maximum degree is 4), therefore, the maximum number of keys a node is 2 \* t – 1 which is 3.

**Ex1**: Insert 90

Diagram

Description automatically generated with medium confidence

**Ex2:** Insert 100

Diagram

Description automatically generated

**Ex3**: Insert 200A picture containing chart

Description automatically generated

Now, this child node “**2**” already has the maximum number of keys. We need to split this node into two parts and the middle key “**150**” will go up to the parent node and be filled in key[**2**] of the root node. Making these two parts will create two new child nodes.

Chart

Description automatically generated with low confidence

**Ex4**: Insert 92

Now, this child node “**1**” already have the maximum number of keys. We need to split this node into two parts and the middle key “**90**” will go up to the parent node. Chart

Description automatically generated

The keys behind the position “**1**” are moved to the right node step by step. Then, the children behind position “**1**” are also moved to the end. Making these two parts will create two new child nodes.

The key “**90**” is filled in key[**1**] of the root node.A picture containing text, clock

Description automatically generated

Text

Description automatically generated

**Code**

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

**\*\* Run code**

**Ex1**: Input: 50, 80

Insert: 90Graphical user interface, text

Description automatically generated

**Ex2**: Input: 50, 80, 90

Insert: 100

Text

Description automatically generated

**Ex3**: Input: 50, 80, 100, 120, 150, 180

Insert: 200

A screenshot of a computer

Description automatically generated with medium confidence

**Ex4:** Input: 50 80 90 100 120 85 95

Insert: 92

A screenshot of a computer

Description automatically generated with low confidence

\*\***Not Preemptive Split: a special case for 2-3 tree (B – tree order 3)**

**Algorithm**

* The same as the above insertion algorithm. But after inserting, we will check the number of keys in the node and decide to split or not.
* If there are more than the maximum number of keys, we will split the node into two parts and one middle key. If the maximum degree is odd, two parts that are separated will be equal. Conversely, the left part or the right part can contain more keys.
* Push the middle key upwards and add it to the parent node. Then, we make these two new children nodes. If there is overflow in the parent node, we will continue to split until the overflow disappears.
* If there are fewer or equal to the maximum number of keys, the program finishes.

With some examples there is a tree of maximum degree “t” as 3. The maximum number of keys a node is t – 1 which is 2.

**Ex1**: Insert 90

Diagram

Description automatically generated

**Ex2**: Insert 100

Chart, waterfall chart

Description automatically generated

Insert 120. Since the child node “2” is full. After inserting 120, it overflows and it will first split into two parts and push the middle key “100” (95 < 100 < 120) upwards. Then, add 100 to the root node and make these two new nodes as its child (95 and 120).

A picture containing text, sign, indoor, clock

Description automatically generated

Now, the root is overflowing. It will continue to split until the overflow disappears.

A picture containing text, sign

Description automatically generated

1. **COMPLEXITY**
2. **Space complexity**

The complexity of space in a B-tree is O(n), which n is the number of elements in the tree.

1. **Time complexity**

The complexity of time of searching, insertion, deletion operations in B-tree is O(logn) in average case, and O(logn) in worst case, which n is the number of elements in the tree.

Let’s explain why they are O(logn). We come first with the searching function. Firstly, we need to define some variable, ‘m’ is the max degree of a B-tree; ‘n’ is the number of elements in a tree. Overall, big-O notation considers the worst case scenario, the searching operation depends on the height of the tree. Then we have to consider the worst height of tree which is the highest height = . And then we perform the binary search in one node in each height, so that is logm. Multiply two of them we have the total of logm\*, but the m is a constant so logm is constant too. Which means the big-O notation is O(logn). We can do similarly with the insertion and deletion operations.

1. **Comparison**

As we have learned that the Binary Search Tree has space and time complexity is better than 2-3 trees but just slightly better. Because the number of comparisons in 2-3 trees is more and then redundancy in each node makes the space bigger. So in terms of efficiency, a Binary Search Tree is better than 2-3 trees. Then 2-3 trees are better than 2-3-4 trees with the similar above explanation. Moreover, if you didn’t know, the 2-3 tree is B-tree order of 3 and 2-3-4 tree is B-tree order of 4. So B-tree is not better than Binary Search Tree just in terms of efficiency. But in real life problems when you need to access a disk then B-tree is better because of its minimum height possible.

1. ***Example***

Because B-tree is suitable for storage systems that read and write large blocks of data. They are particularly well suited to on-disk storage. The searching, insertion, deletion all take the logarithmic time.

It is common to use in file systems. So why is B-tree a common use for file systems? Because it was designed flat, having many values in a node which also means it is having a low height. Therefore, it reduces the amount of data that has to be sent (I/O operations). Reducing tree depth also implies limiting I/O, because each node traversing in a binary tree is an I/O operation. Because each node does not have to be completely filled, the tradeoff is complexity and storage inefficiency. As a result, B-trees are suitable for file systems.

What about real life applications? We have known the advantages of B-tree, so it will be well suitable for ‘store’, maybe just at this moment with all we have learned. Just imagine how you modify the tree, for example, a store of food. For restaurants, they can ‘insert’ the dishes they serve while , for customers, they can search for food that they want. So it will have the searching and inserting for food quickly, but also it depends how you modify the tree too, such as a node can be a type of food or a restaurant.

[**IV.** **ADVANTAGE AND DISADVANTAGE**](#_heading=h.3znysh7)

1. **Advantages of B-tree**

* In every node of B-tree, it keeps the keys sorted in order (ascending or descending).
* Due to its flat structure, we can easily and efficiently traverse through the data while storing a million items.
* It is also a balanced tree then why the worst case is the same as the average case.
* Compared to hash tables, B-Tree allows for ordered sequential access, which makes it more efficient overall.
* B-tree and its index are stored in a file so B-tree doesn’t have to be rebuilt every time when needed in a program.

1. **Disadvantages of B-tree**

* When executing a search, a B-tree that does not fill RAM requires more disk searches on average than a hash table.
* The implementation of a B-tree is far more difficult than that of a hash table.
* When the keys are dense, an array's random access is quicker and uses less memory than a b-tree. The implementation of a B-tree is significantly more difficult than that of an array.
* When compared to other balanced trees, B-trees are inefficient when stored in RAM. Many keys/values must be moved around while inserting or removing items. If a b-tree has a low branching factor and a node fits a cache line, it has a possibility of being efficient in RAM. Cache misses may be reduced as a result of this.

1. **APPLICATION**

B-Tree algorithms are good for accessing **pages** (or blocks) of stored information which are then copied into main memory for processing. In the worst case, they are designed to do dynamic set operations in O(log n) time because of their high **"branching factor"** (hundreds or thousands of keys on any node). It is this branching factor that makes B-Trees so efficient for **block storage/retrieval,** since a large branching factor greatly reduces the height of the tree and thus the number of disk accesses needed to find any key! The total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ... Also, B-Tree is the default index type for most storage engines.

For example: a B-Tree with a branching factor of 1001 and a height of 2 can store over 1 Billion keys! And since the root node can be kept in main memory, you only need 2 disk accesses to find any key in this tree!

B-Tree is used to **index the data** in multilevel layers and provides fast access to the actual data stored in a self-balancing fashion on the disks since the access to value stored in a large database is very time consuming.

Each node in the B-Tree stores not only keys, but also a record pointer for each key to the actual data being stored. (Could also potentially store the record in the B-Tree itself). To find the data you want, search the B-Tree using the key and then use the pointer to retrieve the data (No additional disk access is required if the record is stored in the node) process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in the worst case. List indexing and sparse-array indexing have serious performance disadvantages. However, if we use B-Tree to index this database, it will be searched in O(log n) time in the worst case. The only serious competitor with B-Tree indexing could be hashing.

**Databases and file systems** (SQL SERVER, QUERIES…): It is a useful algorithm for databases and file systems for handling a bulky amount of data.

* Dictionary - range queries, say if you want to find all entries where the key is between two values in a sorted order. For range queries like between XX and YY.
* Map folders/directories.
* Recording and checking a list of keys, spellcheck: You want to read a list of correct words, and then check every word typed against those words. Storing all the words in a tree of some sort makes the lookups quicker - scanning through an entire list each time would be slow (scaling with the number of words), while a look-up in a tree scales with the logarithm of the number - roughly speaking. In a similar way, you could use a tree to compare two lists - put the first in a B-Tree, then check each item in the other against the tree.

1. **REFERENCE**

[Geeksforgeeks](https://www.geeksforgeeks.org)

[StackOverflow](https://stackoverflow.com)

[Jenny’s lectures CS/IT NET&JRF](https://www.youtube.com/watch?v=GKa_t7fF8o0)

[Nargish Gupta](https://www.youtube.com/watch?v=ffVZ02TKeMc&t=312s)

[saurabhschool](https://www.youtube.com/watch?v=k5J9M5_IMzg&list=WL&index=1)

[Wikipedia](https://en.wikipedia.org/wiki/B-tree)

[Oracle](https://blog.toadworld.com/2017/05/08/how-oracle-b-tree-indexes-work?hs_amp=true)

[Viska Gupta](https://guptavikas.wordpress.com/2012/12/17/b-tree-index-in-mysql/amp/?fbclid=IwAR0RDXL4MMTjws2RfLMLsPHIXdmiUQ0jbSydYR9DU2zwdsaET75yGzRXL4Q)

[DZone](https://dzone.com/articles/database-btree-indexing-in-sqlite)

[Javatpoint](https://www.javatpoint.com/b-tree)

[Quora](https://www.quora.com/What-are-some-real-world-applications-using-B-tree)

[CodeProject](https://www.codeproject.com/Articles/808055/Introduction-to-B-trees-Concepts-and-Applications)

<https://cs.stackexchange.com>