B-Tree Implementation and Automated Data Insertion

Manish Kolla, Ashwin Gopalakrishnan
Design & Analysis: Algorithms (CS 4520)
Department of Computer Science
Atlanta, GA, United States
mkolla1@student.gsu.edu, agopalakrishnan3@student.gsu.edu

Abstract **Propelled by Professor Shiraj's insightful lectures on B-trees, our curiosity piqued regarding this data structure that bears a resemblance to the familiar binary tree. However, we soon discovered that B-trees are a distinct and far more efficient approach for search, insertion, and deletion operations. This project, undertaken for the Design Analysis and Algorithms (CS 4520) course under the guidance of Professor Shiraj Pokharel, delves into the intricacies of B-trees and their construction from scratch. Our project encompasses sequential element insertion, element removal, and efficient search enabled by indexing. A unique aspect of our project lies in its ability to automate data addition and indexing when a user uploads a dataset in the form of a CSV file or an Excel spreadsheet. All B-tree operations are performed with a time complexity of O(log n), ensuring remarkable efficiency.**

I. Introduction

The primary reason for choosing this project for this class is our professor citing this as an example for a long time which has led us with a curiosity to learn about this tree and its operations and discover its underlying uniqueness and efficiency when compared to Binary trees. After watching a bundle of YouTube videos and reading multiple online resources which includes documentation and some non-credible resources, we have understood the idea behind the B tree and the advantage of B trees over any other traditional trees. We decided to implement the automation of parsing the excel sheet or csv file when uploaded and converting it to a B tree without the need of manual insertion of elements.

We wanted to investigate and analyze the B tree structure and algorithm. B-trees are versatile data structures that find applications in various domains where efficient sorting, searching, and retrieval of data are essential, particularly when dealing with large datasets and storage systems. Their self-balancing property ensures that operations remain efficient even as data is inserted or deleted.

II. Materials and Methods

Resources Used:

1. Google Colab Notebook
2. A sample employee dataset

Time Complexity of operations:

1. Insertion of elements: O(logn)
2. Removal of elements: O(logn)
3. Searching of elements: O(logn)
4. Deletion of elements: O(logn)
5. Traversal methods: O(n)

Steps followed in the sequential process of building and implementing this project:

1. Understanding and history of B Trees.
2. Basic implementation of B tree structure (with determination of max number of keys in each node)
3. Implementation of Insertion of nodes
4. Implementation of Searching of nodes
5. Implementation of removal of nodes
6. Implementation of automated csv/excel parsing and indexing.

Understanding and history of B-Trees:

In the realm of computer science, B-trees stand as a fundamental data structure designed to efficiently manage large datasets. Introduced by Rudolf Bayer and Edward M. McCreight during their tenure at Boeing Research Labs, B-trees were conceived to address the challenges posed by voluminous indexes that exceeded the capacity of main memory. Their seminal paper, "Organization and Maintenance of Large Ordered Indices," first circulated in July 1970 and subsequently published in Acta Informatica, laid the foundation for this groundbreaking data structure.

However, even after this publication, neither of them explained the real meaning of the letter B in the term B Trees. Despite that McCreight has said that, “the more you think about what the B in B-trees means, the better you understand B-trees.” Which left everyone curious about the letter B.

Soon, after doing some research in B trees, we understood that Binary tree is also an order 2 of B tree with maximum of 2 children to each node. Throughout this process of self-clarification, we have written some of the detailed differences between B Trees and binary trees. After a more in-depth understanding of the differences, we were able to explain the explain the differences between B-Trees and Binary Trees.

Feature	B- Trees	Binary Trees
Nodes	Each node can have multiple children up to a maximum number defined.	Each node can have at most 2 children.
Keys per node	Each node can hold a range of values	Each node can only hold one key
Pointers	Each node contains pointers to its children, internal nodes, and keys	Each node contains pointers to its left and right child only
Height	Usually, B trees are dense, but are not as tall as binary trees for massive datasets	Binary trees have more depth for larger datasets due to limitation of maximum keys in each node
Insertion Time Complexity	Best and Worst: O(logn)	Best: O(logn), Worst: O(n)
Removal Time Complexity	Best and Worst: O(logn)	Best: O(logn), Worst: O(n)
Search Time Complexity	Best and Worst: O(logn)	Best: O(logn), Worst: O(n)
Space Complexity	O(n)	O(n)
Usage and Applications	Databases and computer networking	General purpose DS and Huffman coding

Whereas n represents the number of total nodes.

Besides these dissimilarities with Binary trees, there are also some unique properties in B trees which makes them more efficient and faster for bigger databases. According to Knuth's definition, B trees of order t can be defined as:

1. Every node has at most t children.
2. Every internal node has at least ⌈t/2⌉ children.
3. The root node has at least two children, unless it is a leaf.
4. All leaves appear on the same level.
5. A non-leaf node with k children contains k−1 keys.

Besides these properties there are a few more properties which it must also satisfy for a B tree:

1. All nodes (including root) may contain at most (2*m – 1) keys.
2. The number of children of a node is equal to the number of keys in it plus 1.
3. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
4. The minimum height of a B tree can be calculated by this formula:

$$H_{min} = \lceil \log_m(n+1) \rceil - 1$$

The maximum height of a B tree can be calculated by this formula:

$$H_{max} = \lfloor \log_t(\frac{n+1}{2}) \rfloor$$

Where,

$t = \lceil \frac{m}{2} \rceil$

n = number of nodes

t = min number of children a non-leaf node can have

m = maximum number of children a node can have

Algorithm Design

Basic Implementation of a B-Tree

A B-tree is a self-balancing tree data structure that is used to organize data efficiently in logarithmic time. The basic implementation of a B-tree is as follows:

1. Defining the edge cases
2. Implementing the insertion function
3. Implementing the search operation
4. Implementing the removal operation

Each node in the B-tree is represented by the BTreeNode class. This class encapsulates the following attributes:

keys: An array that stores the node's keys in sorted order.
children: An array that holds pointers to the node's child nodes.
Leaf: A boolean flag indicating whether the node is a leaf node, meaning it has no children.
t: The minimum degree, which defines the minimum number of keys a node can hold.

The insert() Method:

Adding Keys to the BTree subsequently

The insert () method on the BTree class is responsible for inserting new keys into the tree. It begins by checking if the root node is full, meaning it contains utmost 2t-1 keys. If so, it creates a new root node with the old root as its child and invokes the split_child() method on the new root to split the old root into parts. This effectively increases the height of the tree. Finally, it recursively calls the insert_non_full() method on the root to insert the new key.

The insert_non_full() Method:

Inserting Keys into Non-Full Nodes (keys less than 2t-1)

The insert_non_full() method on the BTreeNode class handles the insertion of keys into nodes that are not yet full. For leaf nodes, the key is simply appended to the end of the keys array, and the array is sorted. However, for internal nodes, the method employs a more intricate approach. It

first identifies the appropriate child to insert into by traversing through the keys array. If the identified child is full, it splits the child using the split_child() method and then recursively inserts into the appropriate child.

The split_child() Method:
Balancing Node Overflow

The split_child() method addresses node overflow by splitting a full child node into two roughly equal halves. It creates a new node to store the greater half of the keys and children, extracts the median key from the full child, and inserts it into the parent node. Finally, it inserts the new node alongside the old child into the parent's children array removing the duplicate element which has been moved to the parent node from the child node. This new parent node will be inserted at index childIndex + 1, which ensures that the new node is placed after the original child node in the parent's child list.

The search() Method:
Searches for a value in the B tree in logarithmic time

The search algorithm exploits the balanced structure of a B-tree to achieve efficient logarithmic time complexity. It begins at the root node, then recursively narrows down the search space by rapidly selecting the correct child subtree using binary search at each internal node. This quickly guides the search to reach a leaf node while traversing a minimal number of levels. Finally, a linear scan over the leaf node locates the search key, if it exists in the tree, otherwise false is returned. By minimizing the traversal path to a leaf and avoiding repeated linear scans of nodes, the search operates in log(n) time even for large trees.

The delete() Method:
Deleting the elements from the tree and balancing

The deletion method in this B-tree implementation is an algorithm that maintains the tree's balance after removing a key. It involves locating the key, handling its removal differently based on whether it's in a leaf or an internal node, and then carefully rebalancing the tree if necessary. This rebalancing includes borrowing keys from sibling nodes or merging nodes. The method ensures that the tree maintains its essential properties, keeping operations efficient, which is crucial for the effectiveness of B-trees in handling large data sets.

Traversal Methods:
Depth First Search and Breadth First Search

The B-tree implementation offers four distinct traversal methods, each catering to specific needs. Pre-order prioritizes the current node's data, processing its keys before diving into its children. Post-order tackles all children first, making it ideal when child processing is crucial. In-order, specifically designed for B-trees, delivers a sorted key sequence by visiting children around the current node's key. Lastly, breadth-first explores the tree layer by layer, perfect for level-wise analysis or visualization. This diverse set of traversals showcases the B-tree's flexibility, allowing users to access and process data in ways that best suit their specific goals.

The file() function:
Converts dataset into Btree and helps user manipulate data

This program takes a CSV/Excel file and builds a B-tree data structure. Users can then search for elements, remove elements, or exit the program. The program also supports various tree traversals like DFS (pre-order, in-order, post-order) and BFS (level-order). It removes elements from both the B-tree and the original dataset for accuracy. Searching retrieves the entire row containing the element, not just a boolean. Exiting prints the final B-tree after all manipulations. Users only need to provide the file path with extension. The program automatically recognizes CSV/Excel files and builds the B-tree with a chosen minimum degree. It is built user-friendly.

III. Critique

B-trees are effective in use with small datasets and storage systems. While it is efficient, it also has drawbacks. One is the complexity in implementation. The implementation of B-Trees is notably complex due to their structure involving multiple child nodes and the requirement for maintaining balance. This complexity can lead to difficulties in both development and maintenance, increasing the likelihood of bugs and making optimization a challenging task. A second drawback is performance with large datasets and cache inefficiency. Since nodes may not be located close to each other in memory, this can lead to frequent cache misses, reducing performance. Additionally, the overhead of balancing and restructuring the tree can become significant in large-scale implementations. Lastly, implementing concurrency control in B-Trees is difficult. Ensuring safe and efficient access and modification by multiple threads or processes without causing data corruption, deadlocks, or race conditions requires sophisticated and often complex concurrency mechanisms.

Regarding optimization, our algorithm in this project is as optimized as possible. There are not many, if any, optimizations that we can make. The time complexity is the most efficient as it can be. We are content with the efficiency of this algorithm.

REFERENCES

[1] GeeksforGeeks, “Introduction of B-tree,” GeeksforGeeks, https://www.geeksforgeeks.org/introduction-of-b-tree-2/ (accessed Dec. 5, 2023).

[2] J. khatri lamba, “5.28 B-tree deletion in data structures | DSA tutorials,” YouTube, https://www.youtube.com/watch?v=GKa_t7fF8o0 (accessed Dec. 5, 2023).

[3] A.Bari, “10.2 B-trees and B+ trees.how thry are useful in databases,” YouTube, https://www.youtube.com/watch?v=aZjYr87r1b8 (accessed Dec. 5, 2023).

[4]“BTrees”, Wikipedia https://en.wikipedia.org/wiki/Btree#:~:text=B%2Dtrees%20were%20invented%20by,could%20fit%20in%20main%20memory (accessed Dec. 5, 2023).