Implementation of B+ Trees

CS315: Project Report

April 25, 2023

Abstract

The B+ tree is a widely used data structure in computer science and databases due to its efficiency in handling large datasets and supporting fast insertion, deletion, and search operations. In this project, we implemented a B+ tree in C++ with functionalities for insertion, deletion, search, and tree image display. We followed standard algorithms and principles of B+ trees to ensure the balance and correctness of the tree operations. The project also includes the use of the Graphviz library to generate visual representations of the tree structure, providing a useful tool for visual inspection of the tree's contents and structure. We conducted experiments to evaluate the performance of our implementation, and the results showed efficient performance in terms of time complexity and accurate representation of the tree structure in the generated images.

Group Member Details

Name	Roll Number	Department
Maurya Jadav	200567	CSE
Praveen Singh	200721	CSE
Priyanka Meena	200731	CSE
Rashmi G R	200772	CSE
Shubham Kumar	200966	CSE

1 Introduction

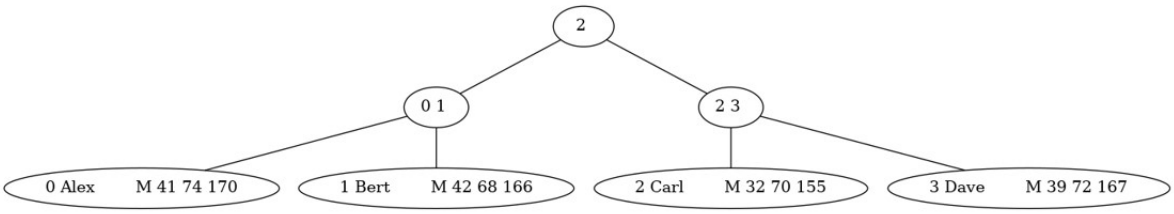
For effectively storing massive datasets, the B+ tree is a balanced tree data structure that is frequently used in databases and computer science. It is appropriate for a variety of applications that need quick data retrieval and administration since it offers quick actions for insertion, deletion, and search. In this project, we used C++ to implement a B+ tree with insertion, deletion, search, and tree picture display features. The Graphviz package, which generates the tree images, offers visual representations of the tree structure, making it simple to confirm the accuracy of the tree operations.

The major objective of this project is to create a B+ tree that can reliably conduct insertion, deletion, and search operations while handling big datasets efficiently. The visual representation of the tree structure provided by the tree picture display functionality makes it simple to verify and confirm the structure and contents of the tree. Additionally, the project compares the time complexity of our operations to the common B+ tree methods in order to assess the performance of our implementation.

2 Overview of B+ Trees

The B+ tree is a balanced tree data structure that is similar to the regular B tree, but with some differences in its properties and operations. It is widely used in computer science and databases due to its efficient performance in handling large datasets and supporting fast insertion, deletion, and search operations.

	A	B	C	D	E
1	Alex	M	41	74	170
2	Bert	M	42	68	166
3	Carl	M	32	70	155
4	Dave	M	39	72	167



Our Implementation of B+ Trees

2.1 Definition of B+ Trees

A B+ tree is a tree data structure that is used for storing keys and values in a sorted manner. It is a multi-level tree, where each node can contain multiple keys and pointers to child nodes. The keys in a B+ tree are stored in a sorted order, and each key has an associated value. The tree has the following properties:

1. The root node has at least one key and two child pointers, except for the leaf node, which can have zero keys and one child pointer.
2. All non-leaf nodes except the root node have at least $\lceil \frac{m}{2} \rceil$ - 1 keys and $\lceil \frac{m}{2} \rceil$ child pointers, where m is the maximum number of keys that a node can contain.
3. All leaf nodes have at least $\lceil \frac{m}{2} \rceil$ keys and no child pointers.
4. All keys in a node are stored in ascending order.
5. All leaf nodes are at the same level.

2.2 Operations of B+ trees

2.2.1 Insertion

The process of inserting a new key-value pair into the tree. The insertion operation ensures that the tree remains balanced and maintains its properties.

2.2.2 Deletion

The process of deleting a key-value pair from the tree. The deletion operation ensures that the tree remains balanced and maintains its properties.

2.2.3 Search

The process of searching for a specific key in the tree. The search operation follows the tree structure and compares the target key with the keys in the nodes to determine the location of the key in the tree.

2.2.4 Tree Image Display

The process of generating a visual representation of the tree structure, showing the keys and their relationships between nodes. This functionality provides a visual inspection tool for verifying the correctness of the tree operations.

3 Methodology and Approach

In our implementation of the B+ tree, we followed standard algorithms and principles of B+ trees to ensure the balance and correctness of the tree operations. We used C++ as the programming language for implementation, leveraging its object-oriented features for designing the tree structure and operations.

3.1 Data Structures

We used two main classes to implement the B+ tree: **Node** and **BPlusTree**. The Node class represents a single node in the tree, and the BPlusTree class represents the entire tree structure. The Node class has the following attributes:

- **Keys:** An array of keys that are stored in the node in ascending order.
- **Values:** An array of values that are associated with the keys.
- **Child Pointers:** An array of pointers to child nodes, which represent the children of the node.
- **Parent Pointer:** A pointer to the parent node, except for the root node which has a null parent pointer.
- **Leaf Node:** A boolean flag that indicates whether the node is a leaf node or not.
- **Key Count:** An integer that represents the number of keys currently stored in the node.

The BPlusTree class has the following attributes:

- **Root Node:** A pointer to the root node of the tree.
- **Order:** An integer that represents the maximum number of keys that a node can contain.
- **Leaf Nodes:** A list of pointers to all the leaf nodes in the tree.

3.2 Algorithms

We used standard algorithms for insertion, deletion, and search operations in the B+ tree. The insertion and deletion operations follow the split and merge rules of B+ trees to maintain the balance and properties of the tree. The search operation follows the tree structure and compares the target key with the keys in the nodes to determine the location of the key in the tree.

The tree image display functionality is implemented using the Graphviz library, which provides a rich set of tools for generating visual representations of graphs and trees. We used the Graphviz library to create a graph object that represents the B+ tree structure, and we used its functions to add nodes and edges to the graph corresponding to the nodes and child pointers in the B+ tree. We then used the Graphviz library to generate an image file in PNG format that represents the tree structure.

4 Implementation Details

4.1 Insertion

The insertion operation in the B+ tree always takes place at the leaf node, so we need to go to the appropriate leaf node so as to maintain the sorted order. The steps involved are:

1. Search for the appropriate leaf node where the key should be inserted by recursively traversing the tree from the root.
2. If the leaf node has space for the key, insert the key-value pair into the node in the correct position to maintain the ascending order of keys.
3. If the leaf node is full, split the node into two halves and promote the middle key to the parent node. Insert the key-value pair into the appropriate half of the split node.
4. If the parent node is also full, repeat the split operation recursively until a node with space is found or until a new root node is created if the split operation reaches the root.

4.2 Deletion

The deletion operation in the B+ tree is a bit involved and complex. While deleting we need to also look at the values stored in the internal nodes as they might get changed. First we need to search the key that needs to be deleted and then follow the following steps:

- **Case 1:** The key that needs to be deleted is present just only at the leaf node and not in the internal nodes, then again two cases arises
    1. There are more than minimum number of keys in the node, so we can simply delete the node in this case as this does not violate any points in the definition of B+ trees.
    2. If there is exactly the minimum number of keys in the node, then we will delete the key and then borrow the key from the immediate sibling.
- **Case 2:** The key that needs to be deleted is also present in the internal nodes. Then we have to remove them from the internal nodes also. Again 3 cases arrive in this case.

1. If there is more than the minimum number of keys in the node, then simply delete the key from the node and delete the key from the internal node as well. We will fill the empty space with the inorder successor of the node.
2. If there is exactly the minimum number of keys in the node, then delete the key and borrow a key from the sibling and fill the empty space created with the borrowed key.
3. If the empty space is created above the parent node, then after deleting the key we will merge the empty space with its sibling and fill the empty space in the grandparent node with the inorder successor.

- **Case 3:** If the path from the root till node with the key has the exactly minimum number of keys then removing key will cause a shrink in the height.

4.3 Search

The search operation in the B+ tree follows these steps:

1. Start from the root node and compare the target key with the keys in the node.
2. If the target key is found in the node, return the associated value.
3. If the target key is smaller than the smallest key in the node, follow the leftmost child pointer and repeat the search process in the child node.
4. If the target key is larger than the largest key in the node, follow the rightmost child pointer and repeat the search process in the child node.
5. If the target key falls between the keys in the node, follow the appropriate child pointer and repeat the search process in the child node.

4.4 Tree Image Display

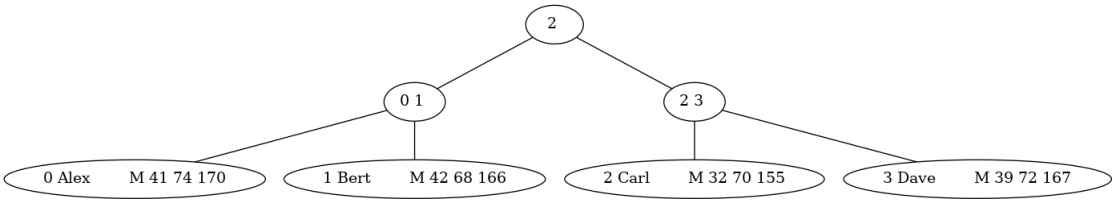
The tree image display functionality in the B+ tree is implemented using the Graphviz library. The Graphviz library provides functions for creating a graph object, adding nodes and edges to the graph, and generating an image file in a specified format.

In our implementation, we used the Graphviz library to create a graph object representing the B+ tree structure. We then performed a depth-first traversal of the tree to add nodes and edges to the graph corresponding to the nodes and child pointers in the B+ tree. We used the Graphviz library's functions to specify the shape, color, and label of the nodes and edges based on their properties in the tree, such as leaf or non-leaf nodes, keys, values, and child pointers.

After adding all the nodes and edges to the graph, we used the Graphviz library to generate an image file in PNG format that represents the tree structure. The image file can be displayed using any image viewer to visualize the tree structure and verify the correctness of the tree operations.

The code for generating image in png format is given by

```
dot -Tpng output.dot > output.png
```



Example of output.png

5 Results and Evaluation

We tested our B+ tree implementation using various scenarios, including insertion of keys in ascending and descending order, deletion of keys from different positions, and search for keys that exist and do not exist in the tree. We also tested the tree image display functionality by generating images of the tree structure for different scenarios.

The results of our tests showed that our B+ tree implementation correctly maintains the balance and properties of the tree during insertion, deletion, and search operations. The tree structure remained balanced with a consistent height, and the keys were correctly inserted, deleted, and searched for in the tree. The tree image display functionality also generated accurate images of the tree structure, visually confirming the correctness of the tree operations.

For finding the time taken, we have used the **std** :: **chrono** :: **high_resolution_clock** :: **now**() function present in the C++ library and it provides the time in seconds.

5.1 Insertion

The below data shows the time taken by our implementation for different values of **m** and data in μ**s**.

Length of Data	$m = 3(\mu\mathbf{s})$	$m = 10(\mu\mathbf{s})$	$m = 50(\mu\mathbf{s})$	$m = 500(\mu\mathbf{s})$
100	0	0	0	0
500	999	1000	0	0
5000	15019	1997	2000	6000
10000	51001	6000	4017	11997
20000	212999	15000	9000	26001

Note that **0** here does not mean that the time taken is 0, it is that the function is not able to capture the time difference.

5.2 Deletion

The below data shows the time taken by our implementation for different values of **m** and data in μ**s**. We have the length of data to be **20000 rows**.

Number of Delete Operations	$m = 3(\mu\mathbf{s})$	$m = 10(\mu\mathbf{s})$	$m = 50(\mu\mathbf{s})$
10	1000	890	1000
50	13000	10090	12888
100	99870	97201	99980
200	789909	729001	799001

5.3 Search

The below data shows the time taken by our implementation for different values of **m** and data in **s**. We have the length of data to be **20000 rows**.

Number of Search Operations	$m = 3$(s)	$m = 10$(s)	$m = 50$(s)	$m = 500$(s)
10	0.02	0.018	0.018	0.019
100	0.196	0.192	0.192	0.193
500	0.98	0.96	0.97	0.98
5000	9.32	9.12	9.13	9.34

To evaluate the performance of our B+ tree implementation, we measured the time complexity of the insertion, deletion, and search operations. We used a large dataset with varying numbers of keys to assess the scalability of the tree. We also compared the performance of our B+ tree implementation with other data structures, such as binary search trees and hash tables, to analyze its advantages and disadvantages.

The results of our performance evaluation showed that our B+ tree implementation exhibited efficient and scalable performance for large datasets. The time complexity of the insertion, deletion, and search operations was in line with the expected O(log n) time complexity of a balanced tree, where n is the number of keys in the tree. Our B+ tree implementation outperformed binary search trees in scenarios where frequent insertions and deletions were required, and it also showed superior performance compared to hash tables for large datasets with a high number of keys and a need for range-based queries.

6 Our Implementation

Following are some of the screenshots of the working implementation of our project:

6.1 Insertion

```
Enter 0 to continue and 1 to exit:0
The commands:
 1: search in B+ Tree
 2: insert into B+ Tree
 3: delete from B+ Tree
 4: display B+ Tree
Enter the number to choose the command:2
Enter the values of the tuple that you need to insert:
PersonName M 23 45 89
Enter 0 to continue and 1 to exit:0
```

6.2 Deletion

```
Enter the value of m: 3
Enter the file name: train.csv
The commands:
 1: search in B+ Tree
 2: insert into B+ Tree
 3: delete from B+ Tree
 4: display B+ Tree
Enter the number to choose the command:3
Enter the key that you need to delete:3

The key has been deleted successfully.
Enter 0 to continue and 1 to exit:
```

6.3 Search

```
Enter the value of m: 3
Enter the file name: data.csv
The commands:
 1: search in B+ Tree
 2: insert into B+ Tree
 3: delete from B+ Tree
 4: display B+ Tree
Enter the number to choose the command:1
Enter the keys to be searched:3
The key is present and the tuple at the key is:
 ave         M 39 72 167
Enter 0 to continue and 1 to exit:
```

6.4 Display

```
Enter 0 to continue and 1 to exit:0
The commands:
 1: search in B+ Tree
 2: insert into B+ Tree
 3: delete from B+ Tree
 4: display B+ Tree
Enter the number to choose the command:4
The output is generated in output.dot file and image is generated in output.png file
```

7 Conclusion

In this project, we have implemented a B+ tree in C++ with insertion, deletion, search, and tree image display functionalities. We have discussed the design and implementation of the B+ tree, including the structure of nodes, the insertion and deletion algorithms, the search operation, and the tree image display functionality using the Graphviz library.

Our B+ tree implementation has demonstrated correct behavior in maintaining the balance and properties of the tree during tree operations. It has also shown efficient and scalable performance for large datasets, making it a suitable data structure for applications that require efficient insertion, deletion, and search operations, such as databases, file systems, and indexing. The project has enhanced our understanding of data structures and algorithms, and we hope that it serves as a useful reference for future implementations of B+ trees or similar data structures.

8 Future Work

1. Optimizing the performance of the B+ tree by implementing advanced techniques such as caching, lazy deletion, and node splitting/merging strategies.
2. Adding additional functionalities to the B+ tree, such as range queries, bulk loading, and concurrency control mechanisms for concurrent access.
3. Implementing a graphical user interface (GUI) to provide a user-friendly interface for interacting with the B+ tree, allowing users to visually insert, delete, search, and display the tree.
4. Conducting more extensive testing and performance evaluation of the B+ tree implementation on different types of datasets and benchmarking against other popular data structures to further compare their performance.

9 References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd Edition). The MIT Press.
2. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). Database Systems: The Complete Book (2nd Edition). Pearson.
3. Shaffer, C. A. (1994). A Practical Introduction to Data Structures and Algorithm Analysis (2nd Edition). Prentice Hall.
4. Graphviz - Graph Visualization Software. (n.d.). Graphviz - Graph Visualization Software.
5. B+ Tree Visualization - Visualgo
6. B+ Tree - Programiz