Analysis and Comparison of Binary and Interpolation Search Algorithms in a B-tree

SOTIRIOS SALAKOS* and NIKOLAOS PLOSKAS*, Department of Electrical and Computer Engineering, University of Western Macedonia, Greece

The consecutive increasing management of large amounts of information in various fields requires the development and construction of efficient database management systems. These systems implement storage, organization, and management of this data in order to perform multiple operations to solve complex and difficult problems. In order to achieve the aforementioned needs, indexing systems have been developed in the context of relational database management systems (RDBMSs). RDBMSs are composed of multiple mechanisms comprising a set of algorithms and data structures. This paper analyzes the performance of the B-tree data structure which laid the foundation as a structural and functional basis for the development of a whole category of tree indexing structures. B-tree indexes have applications in the largest modern RDBMs. Specifically, we study the effect of binary and interpolation search algorithmic techniques on the execution time of the insertion, deletion, and search functions of the B-tree. The computational experiments demonstrate the superiority of the interpolation search technique in the functional level of searching by primary key field.

CCS Concepts: • **Theory of computation** → **Data structures design and analysis**; **Data compression**; **Data structures and algorithms for data management**.

Additional Key Words and Phrases: data structures, B-tree, database management systems, interpolation search, binary search

ACM Reference Format:
Sotirios Salakos and Nikolaos Ploskas. 2021. Analysis and Comparison of Binary and Interpolation Search Algorithms in a B-tree. In *PCI '21: 25th Pan-Hellenic Conference on Informatics, November 26–28, 2021, Volos, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The B-tree data structure [2] [4] constitutes a structural and functional basis for the development and creation of many balanced tree data structures (B^{+}tree [4], B^{*}tree [4] [13], etc.). These structures are used extensively [4] [7] with multiple applications in modern relational database indexing [3] [5] and file management systems [15]. Therefore, there are numerous works and an increasing interest in the development of more efficient structures and the study of their efficient application in modern database and file management systems [1] [9] [10] [12] [14] [16].

In addition, computational studies are being conducted for optimizing execution time and memory usage at multiple operating levels of these systems. The studies analyzing the performance provided by the use of the interpolation search method in the B-tree data structure [6] [8] [11] are quite limited and do not provide a thorough comparison between the interpolation and binary search methods in a B-tree. This paper aims to provide experimental data that will fill this gap by giving detailed information on the execution time of the B-tree data structure at some fundamental functional levels using the aforementioned search techniques. This paper performs a comparison of the interpolation

*Both authors contributed equally to this research.

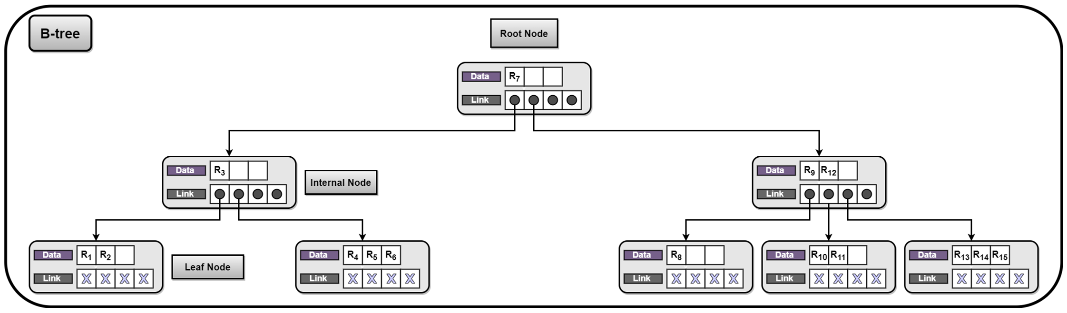

Fig. 1. B-tree Data Structure.

and binary search performance on the basic B-tree operations that are implemented. The interpolation search method was used in this work to optimize the execution time of the basic insertion, deletion, and search functions compared to the classical binary search technique. The main objective of the paper is to compare these search techniques through the implementation of an experimental process and to extract insights on the execution time of the B-tree data structure basic functions.

The structure of the paper is as follows. In Section 2 the structural and functional characteristics of the B-tree data structure are being analyzed. In Section 3, a computational study is presented comparing the performance of binary and interpolation search methods in a B-tree. Finally, the conclusions of this paper are outlined in Section 4.

2 B-TREE DATA STRUCTURE IMPLEMENTATION

In this Section, we describe and analyse the structural parts and the basic functions of the implemented B-tree data structure. In particular, the procedures that implement the basic insertion, deletion, and search functions and the structural elements (nodes) of the tree are listed.

2.1 Structure

Figure 1 describes the structural characteristics of the B-tree data structure which was implemented in the context of this paper. Each node of the B-tree consists of the following structural components:

- The dynamic array containing record addresses.
- The dynamic array containing B-tree node addresses.

Arrays can be resized to half of their capacity depending on the number of addresses that they contain and their maximum allowed capacity. This capability of arrays is used in the insertion and deletion operations to optimally manage the allocated memory that is used to store records and nodes addresses.

2.2 Insertion Process

The algorithmic process of inserting records addresses into the B-tree structure is implemented using three individual functions. These functions undertake a discrete part of the overall process which is functionally linked to the others for efficient completion of the process. The first function BTreeInsertData (Algorithm 1) takes over the initialization of the insertion process and has practical application when the structure does not contain record addresses. Thus, it handles the creation of the first root node of the structure. In case the capacity of the structure is non-zero, it calls the function BTreeInsertNode. The second function BTreeInsertNode (Algorithm 2) implements the reconstruction and reordering of the root, right, and left node-set after the root node has been split due to insufficient storage capacity of the records addresses. The above procedure applies in the case where a root node split was caused after the function BTreeInsertNodeInternal was called. The third function BTreeInsertNodeInternal (Algorithm 3) locates (by utilizing the function SearchBTreeNodeKey) the leaf node of the structure where the record address is to be inserted. Subsequently, the placement and reordering of the structure node are implemented in this function. If the node is full and no other records addresses can be stored, a split into two individual nodes is caused. Then half of the addresses from the first node are transferred to the second node. Finally, the middle element of the node before the split is transferred to the upper-level node. Further reconstruction and reordering of the nodes involved in the process follow. This process is repeated recursively up to the root node of the B-tree.

Algorithm 1: Status BTreeInsertData(B-tree data structure, Record address)

```
if B-tree data structure is empty then
    Creation of root node.
    Insertion of record address in the root node.
end
BTreeInsertNode()
```

Algorithm 2: Status BTreeInsertNode(B-tree data structure, Record address)

```
BTreeInsertNodeInternal()
if Current node is the root node and a node split was performed then
    Creation of left sub-node.
    Reconstruction of root, left, and right node.
end
```

Algorithm 3: B-tree node BTreeInsertNodeInternal(B-tree node, Record address, Median node record address)

```
SearchBTreeNodeKey()
if Current node is a leaf node then
    Insertion of record address in the current node.
else
    BTreeInsertNodeInternal()
    if The record address was placed in the next level node and a split was caused then
        Insertion of the next level's median record address in the current node.
        Reconstruction of current node.
    end
end
if Current node capacity is insufficient then
    Creation of right sub-node.
    Reconstruction of current and right node (split procedure).
end
```

2.3 Selection Process

The algorithmic process of finding a specific record address based on the primary key field of the record is implemented by a set of functions that perform this process. The first function BTreeSearchData (Algorithm 4) checks if a specific record address content exists in the structure and in case of success it calls the function BTreeSearch. The second function BTreeSearch (Algorithm 5) implements a process of searching and locating the record address by the node of the structure up to the last level. It uses the function SearchBTreeNodeKey which searches approximately at the node level for the location where the record address is or may be found. If the record address is not found at the search node, it locates the node at the lowest level where the address may be found. In the context of this paper, the function SearchBTreeNodeKey is implemented using the binary and interpolation search methods.

Algorithm 4: B-tree Record BTreeSearchData(B-tree data structure, Record address of primary key field)

```
if B-tree data structure is empty then
    Record address does not exist.
end
BTreeSearch()
```

Algorithm 5: B-tree Record BTreeSearch(B-tree node, Record address of primary key field)

```
SearchBTreeNodeKey()
if Record address was found in this node then
    Locates and exports the record address.
end
if Current node is an internal node then
    BTreeSearch()
end
```

2.4 Deletion Process

The algorithmic process of deleting records addresses in the B-tree data structure is implemented using four basic functions which are assisted by multiple additional functions. These functions take on a discrete part of the overall process which is functionally linked to the others for the efficient completion of the process. The first function BTreeDeleteData (Algorithm 6) performs two individual operations. The initial procedure involves checking the number of record addresses contained in the structure. Then, if the structure is not empty and consists of a single (root) node, the record address will be deleted from that node. The function SearchBTreeNodeKey is used to locate the address in the node. If the address is not found in the root node and if the structure contains additional nodes, the function BTreeDeleteNode (Algorithm 7) is called. The function BTreeDeleteNode implements the process of finding the address to be deleted at the internal node via the function SearchBTreeNodeKey. If found, it is stored for later deletion. Then, by calling the auxiliary function Locate_DeletionPos_LeftSubTreeMaxRecord, the maximum record address of the left sub-tree is moved to the position where the address to be deleted is located, thus changing positions. Then, if the node of the next level is an internal node, then the function BTreeDeleteNode is called iteratively. If the previous call causes a tree balancing problem, the auxiliary function BTreeDeleteNonLeaf is called to fix it. It therefore applies balancing techniques, reordering the nodes in the structure and reassigning the addresses that they contain. To implement these procedures, it uses multiple auxiliary functions depending on the case of imbalance. If it is not possible to restore balance, it recursively repeats the process up to the root node (if needed). Conversely, if the node of the next level is a leaf node, then the auxiliary function BTreeDeleteLeaf is called to implement the deletion of the record address from that node. The auxiliary function BTreeDeleteLeaf also implements reordering and reassignment techniques if the deletion causes a problem with the balance of the structure. In the case of unsuccessful balancing at the last level of the tree, the function BTreeDeleteNonLeaf takes over the repair of the problem.

Algorithm 6: Status BTreeDeleteData(B-tree data structure, Record address of primary key field, Variable to store the record address to be deleted)

```
if B-tree data structure is empty then
    Deletion cannot be achieved.
end
if Root node is a leaf node then
    SearchBTreeNodeKey()
    if The record address to be deleted is located at the current node then
        Stores record address to be deleted.
        if The Root node contains a single record address then
            Deletion of the node structure containing the record address.
        end
        ReplaceIndex()
    else
        Deletion cannot be achieved.
    end
end
BTreeDeleteNode()
```

Algorithm 7: B-tree node BTreeDeleteNode(B-tree node, Record address of primary key field, Variable to store the record address to be deleted)

```
SearchBTreeNodeKey()
if The record address to be deleted is located at the current internal node then
    Stores record address to be deleted.
    Locate_DeletionPos_LeftSubTreeMaxRecord()
    Replacement of the record address to be deleted with the maximum record address of the left sub-tree.
end
if Next level node is an internal node then
    BTreeDeleteNode()
    if The previous operation performed caused a problem in balancing the nodes of the structure then
        BTreeDeleteNonLeaf()
    end
else
    BTreeDeleteLeaf()
end
```

3 COMPUTATIONAL STUDY

In this section, a computational study is performed in order to compute the execution time of the basic functions of the B-tree data structure. These operations are insertion, deletion, and selection (based on primary key field) of records addresses. In particular, this is a comparative study of the binary and interpolation search methods with the aim of evaluating these techniques. It is important to clarify that the experiments were conducted on a main memory system and no data exchange and communication using disk systems were implemented. The computational comparison has been performed on an Intel Core i7-8700 with 64 GB of main memory, a clock of 3200 MHz, running under CentOS 8 Linux. In addition, all algorithms were implemented in C language. The measurements were made using the GCC 8.4.1 compiler and with -O3 flag enabled.

The algorithmic techniques of binary and interpolation search were applied in two individual measurement procedures. In each measurement process, the insertion, selection, and deletion operations of records addresses were performed sequentially. A set of 100, 000 primary fields (integers) was constructed and then a random reordering procedure of the set was repeatedly performed 100 times. More specifically, for each primary field in the dataset, another field was randomly selected with which to switch positions. This procedure was repeated 100 times for all individual fields. An additional non-primary integer field was then constructed which had the same value for all records. Consequently, each record is composed of the two aforementioned integer fields. The data created and prepared appropriately were randomly inserted into the structure and the total insertion operation completion time was measured. Then the selection (search) operation of each record based on the primary field was performed and the total selection time of all records was calculated. Finally, the deletion of all records that had been inserted in the structure was performed and the completion time of the overall deletion operation was measured. This measurement procedure was repeated 1, 000 times for each node capacity of the structure in a capacity range from 3 to 1, 000 records addresses with an increment factor of 1. For each node capacity, the average of 1, 000 measurements taken for each operation was calculated. In addition, the total average execution of each individual operation was calculated. More specifically, the average execution time of all individual executions (average execution time) per maximum node capacity of the structure was calculated for each individual operation. The above set of operations was implemented separately for the binary and interpolation search algorithmic techniques.

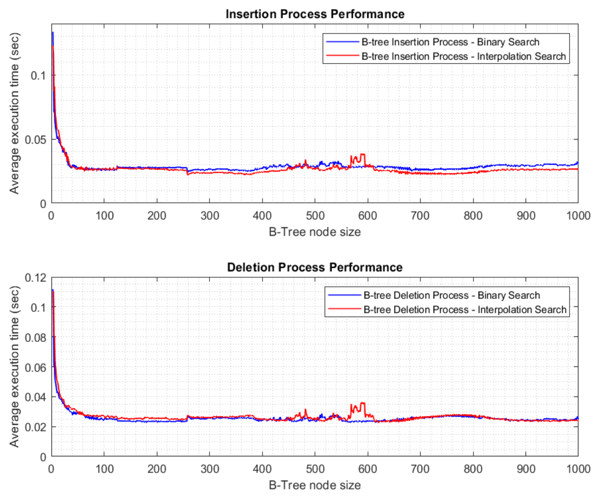

Fig. 2. Average Execution Time of the Insertion and Deletion Functions.

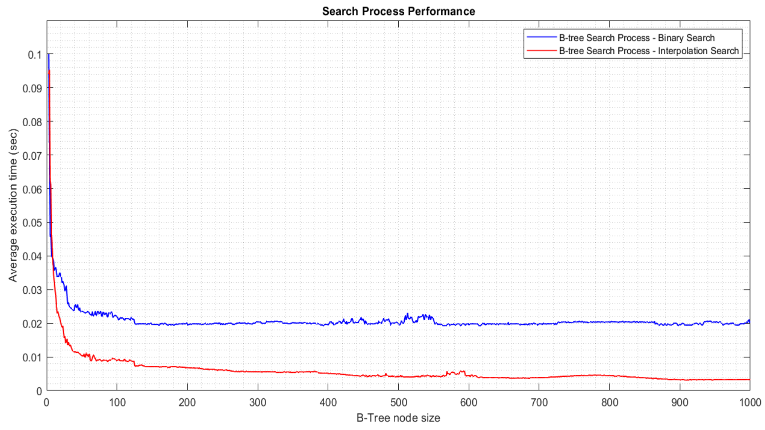

Fig. 3. Average Execution Time of the Search Function.

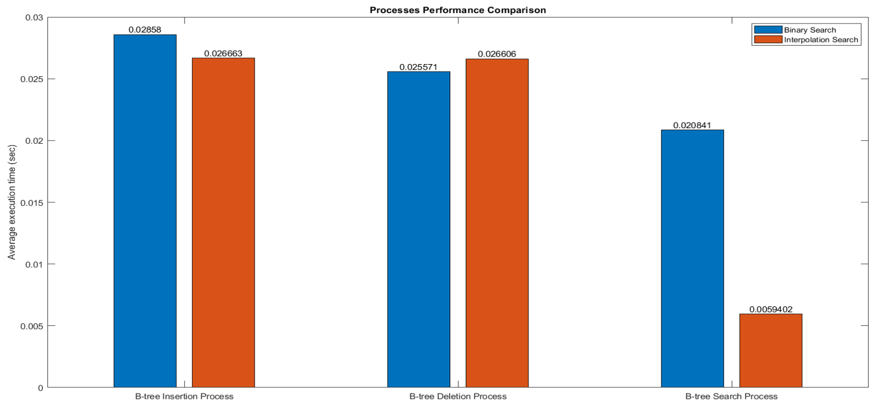

Fig. 4. Total Average Execution Time of the Insertion, Deletion and Search Functions.

Figures 2, 3, and 4 present the results of the computational study. Figures 2 and 3 show the average execution time of the basic insertion, deletion, and search operations in the node capacity range from 3 to 1, 000. In addition, Figure 4 presents the total average execution time per function. These figures further compare the performance at the application level of binary and interpolation search techniques.

The analysis and graphical representation of the measurement results demonstrate the reduction on the execution time achieved by using the interpolation search technique compared to the binary search technique. In particular, the reduction on the execution time concerns the process average completion time of searching for records addresses based on primary key field addresses per node capacity (Figures 2 and 3) and the total average completion time of the process (Figure 4). Specifically , using the interpolation search technique, a 3.5 speedup in the total average execution time was achieved compared to the binary search. Regarding the insertion and deletion functions, no significant time improvement in process completion was observed using the interpolation search technique because of the execution optimization. The interpolation search reduces the average completion time of the insertion and deletion operations per capacity but due to the quite high level of optimization of the compiler, the improvement that would have been achieved without the application of the optimization procedure is not so visible as the execution time is kept to a minimum level. Furthermore, the search function applied to the insertion and deletion is a small part of the overall time complexity of the operations and therefore does not significantly affect the average completion time. The splits and rearrangements of nodes affect the completion time more in compare with the part of searching.

4 CONCLUSION

This work focuses on the experimental analysis of the insertion, deletion, and search operations of primary field-based records addresses in the context of the B-tree data structure. In particular, the implementation of the binary search and interpolation search techniques is compared in the insertion, deletion, and search methods. The analysis of the results of the computational study revealed a significant reduction on the execution time of the search function using the interpolation search technique by a factor of 3.5 compared to the binary search. In the insertion and deletion operations, no significant reduction in the execution time was observed using the interpolation search method. The application of the interpolation search technique caused a small reduction on the execution time of the insertion and a slightly smaller increase on the deletion in comparison with the binary search. In addition, a fairly large improvement in the average execution time of the search function was observed for all measured capacities of the structure node using the interpolation search.

REFERENCES

[1] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. 2019. Engineering a high-performance GPU B-Tree. In *24th Symposium on Principles and Practice of Parallel Programming*. ACM, 145–157.

[2] Randall Bayer and Edward McCreight. 1972. Organization of large ordered indexes. *Acta Informatica* 1 (1972), 173–189.

[3] Rupali Chopade and Vinod Pachghare. 2020. MongoDB indexing for performance improvement. In *ICT Systems and Sustainability*. Springer, 529–539.

[4] Douglas Comer. 1979. Ubiquitous B-tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[5] Peter Fruhwirt, Peter Kieseberg, and Edgar Weippl. 2015. Using internal MySQL/InnoDB B-tree index navigation for data hiding. In *IFIP International Conference on Digital Forensics*. Springer, 179–194.

[6] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *2nd International Workshop on Data Management on New Hardware*. ACM, 1–10.

[7] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree techniques. In *27th International Conference on Data Engineering*. IEEE, 1370–1373.

[8] Ali Hadian and Thomas Heinis. 2019. Interp olation-friendly B-tr ees: Bridging the Gap Betw een AlgorithmicandLearnedInde xes. (2019).

[9] Bhagyashri Anand Jantkal and Santosh L Deshpande. 2017. Hybridization of B-Tree and HashMap for optimized search engine indexing. In *2017 International Conference On Smart Technologies for Smart Nation*. IEEE, 401–404.

[10] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *24th Symposium on Principles and Practice of Parallel Programming*. 327–339.

[11] Alexis Kaporis, Christos Makris, George Mavritsakis, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. 2010. ISB-tree: A new indexing scheme with efficient expected behaviour. *Journal of Discrete Algorithms* 8, 4 (2010), 373–387.

[12] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Transactions on Storage* 14, 1 (2018), 1–17.

[13] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[14] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th International Conference on Data Engineering*. IEEE, 302–313.

[15] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage* 9, 3 (2013), 1–32.

[16] Lei Yu, Ge Fu, Yan Jin, Xiaojia Xiang, Huaiyuan Tan, Hong Zhang, Xinran Liu, and Xiaobo Zhu. 2015. MPDBS: A multi-level parallel database system based on B-Tree. In *16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 1–7.