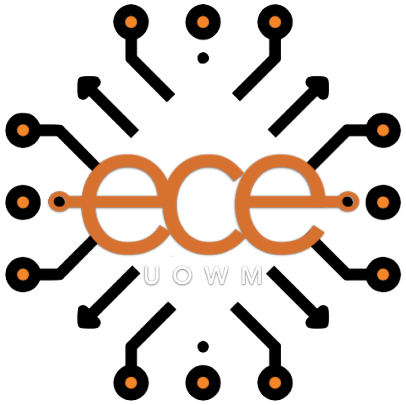


University of Western Macedonia

Department of Electrical & Computer Engineering

Analysis and comparison of data structures for database indexing

Sotirios Salakos

(AM: 1385)

Supervisor: Nikolaos Ploskas, Assistant Professor

Intelligent Systems & Optimization Laboratory

Monday, October 24, 2022

Περίληψη

Η διαδοχική εξέλιξη ενός περιβάλλοντος τεχνολογιών νέφους (cloud), απομακρυσμένων ψηφιακών επικοινωνιών και τεχνολογιών παροχής και διασύνδεσης δεδομένων και υπηρεσιών απαιτεί την ανάπτυξη και κατασκευή αποδοτικών συστημάτων διαχείρισης βάσεων δεδομένων (ΣΔΒΔ) που υλοποιούν την αποθήκευση, τη διανομή και τη διαχείριση μεγάλων συνόλων δεδομένων. Αυτά τα συστήματα διαχείρισης βάσεων δεδομένων αποτελούνται από πολλαπλά υποσυστήματα διαχείρισης αρχείων και επιπέδων δομικών τμημάτων συστήματος που υλοποιούν την αποτελεσματική και γρήγορη διαχείριση των αποθηκευμένων συνόλων δεδομένων μέσω συνόλων λειτουργιών συναλλαγών. Τα συστήματα ευρετηριοποίησης (indexing) έχουν αναπτυχθεί στο πλαίσιο των ΣΔΒΔ για την υλοποίηση αυτής της λειτουργικής αποδοτικότητας, λειτουργώντας ως υποσυστήματα συστημάτων αρχείων που συνδέονται δομικά και λειτουργικά με ένα μεγάλο και πολύπλοκο σύνολο άλλων υποσυστημάτων βάσεων δεδομένων. Τα συστήματα αυτά αποτελούνται επίσης από πολλαπλούς μηχανισμούς που περιλαμβάνουν σύνολα δομών δεδομένων ευρετηριοποίησης και αλγορίθμων. Η παρούσα διπλωματική εργασία συνθέτει μια υλοποίηση των δομών ευρετηρίου B-tree, B^+-tree, B-Hash Map και B^+-Hash Map και μια πλήρη θεωρητική και υπολογιστική ανάλυση των δομών αυτών σε λειτουργικό και δομικό επίπεδο. Η υπολογιστική ανάλυση, αξιολόγηση και σύγκριση της λειτουργικής αποδοτικότητας και της χρονικής απόδοσης των δομών ευρετηρίου B-tree, B^+-tree, B-Hash Map και B^+-Hash Map που αναπτύχθηκαν πραγματοποιήθηκε μέσω ενός συνόλου υπολογιστικών διαδικασιών σε πραγματικά και κατασκευασμένα σύνολα δεδομένων. Κατά συνέπεια, η παρούσα μελέτη παρέχει μια αρκετά πλήρη, περιεκτική και λεπτομερή θεωρητική και υπολογιστική ανάλυση αυτών των υλοποιημένων - αναπτυγμένων δομών και λειτουργιών ευρετηρίων όσον αφορά την αποδοτικότητα και τη χρονική απόδοση της διαχείρισης χρονικών πόρων και πόρων μνήμης.

Λέξεις κλειδιά: Βάσεις δεδομένων, Ευρετηριοποίηση, B-δένδρο, B+-δένδρο, Hash Map.

Abstract

The consecutive evolution of a digital communication, remote data and services provision and interconnection and cloud technologies environment requires the development and construction of efficient database management systems (DBMSs) that implement storage, distribution and management of large datasets. These DBMSs are composed of multiple file sub-systems and system structural components layers that implement the efficient and fast management of the stored datasets through transnational operations sets. Indexing systems have been developed in the context of DBMSs in order to implement this functional efficiency, operating as file sub-systems that are structurally and functionally connected to a large and complex set of other database sub-systems. These systems are also composed of multiple mechanisms comprising indexing data structures sets and algorithms. This thesis composes an implementation of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures and a complete theoretical and computational analysis of those structures functional and structural levels. The computational analysis, evaluation and comparison of the developed B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures functional efficiency and time performance was carried out through a set of computational processes on real and synthetic datasets. Consequently, this study provides a quite complete, comprehensive and detailed theoretical and computational analysis of these implemented - developed indexes structure and functions performance in terms of time and memory resources management efficiency and time performance.

Keywords: Databases, Indexing, B-tree, B+tree, Hash Map.

Copyright

Student Signature

Contents

List of figures

List of algorithms

Chapter 1

Introduction

1.1 Thesis subject and content

Observing that there are not sufficient dynamic, efficient, refactorable, maintainable and general-purpose implementations in C of in-memory B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures combined with a complete, qualitative and concurrently simple theoretical and computational analysis, the conducted study aims to fill partially this gap. In particular, this study aims to provide a complete and solid theoretical analysis of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map indexes structure, functions and performance in terms of time and memory resources management efficiency in conjunction with a set of computational processes and computational analysis, providing functional metric data on real and synthetic data.

The main incentives and objectives of conducting this study are the implementation and development in C of open-source software packages that provide a set of dynamic, efficient, fast and qualitative B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index data structures that are specially designed to operate as structural and functional parts of a RDBMS in-memory file system indexing sub-system (system structures simulation). This packages also includes sets of unit testing function tools and integration tests in order to be provided qualitative, dynamic, refactorable and maintainable software packages (software quality assurance). Furthermore, another important and basic reason for the conduction of this study is the theoretical and computational analysis of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures structural properties and functional efficiency in terms of time and memory resources management (time performance). Moreover the efficiency and time performance of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures is analyzed through a set of computational experiments on real and synthetic data in order to provide metric data for meta-analysis, comparison and evaluation. Finally, this

thesis aims to offer a set of open-source and free packages that could be utilized for the implementation and development of software products and services parts and for research and educational purposes.

1.2 Content structure

The thesis structure is as follows. In Chapter 2, the structural and functional theoretical analysis of the B-tree index data structure is performed. In Chapter 3, we analyze the structural and functional theoretical analysis of the B$^+$-tree index data structure, while in Chapter 4, we analyze the structural and functional theoretical analysis of the B-Hash Map and B$^+$-Hash Map index data structures. In Chapter 5 we present the theoretical analysis of the conducted computational processes and the analysis, evaluation and comparison of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures functional performance metric data (results) on real and synthetic data. Finally, the summary, inferences and conclusions of this study are outlined in Chapter 6.

Chapter 2

Theoretical analysis and implementation of the B-tree data structure

2.1 B-tree index structural properties and characteristics

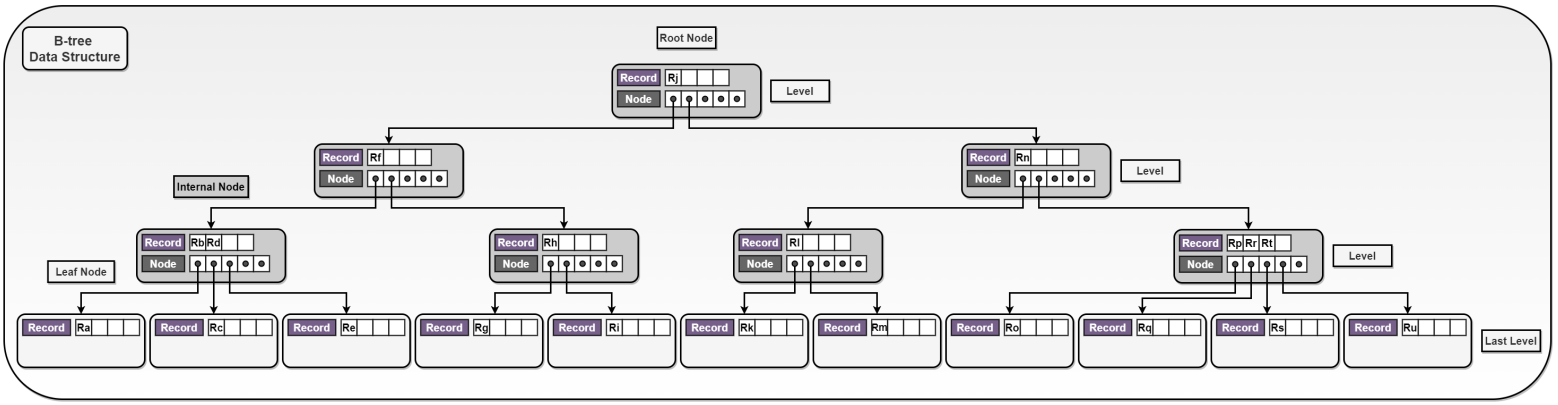
2.1.1 B-tree index structure implementation and development theoretical base

The development of the implemented B-tree data structure is based on the theoretical definition, formulation and analysis of the B-tree index structure and functionality of the works in [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]. This study is based on the aforementioned works and applies modifications at some basic structural and functional B-tree structure levels. Furthermore, it is important to clarify that this B-tree index structure implementation composes a structural and functional approximation of these studies utilizing them as a theoretical base in order to develop and implement an efficient and fast B-tree index data structure. This data structure will be analyzed and evaluated in terms of performance.

Moreover, this implementation is based on the development of the B-tree index structure and the conduction of a computational study in order to analyze and evaluate the approximate average execution time performance of the basic B-tree data structure insertion, deletion and selection operations in the context of our previous work in [14].

2.1.2 B-tree index structure

Figure 2.1: B-tree data structure

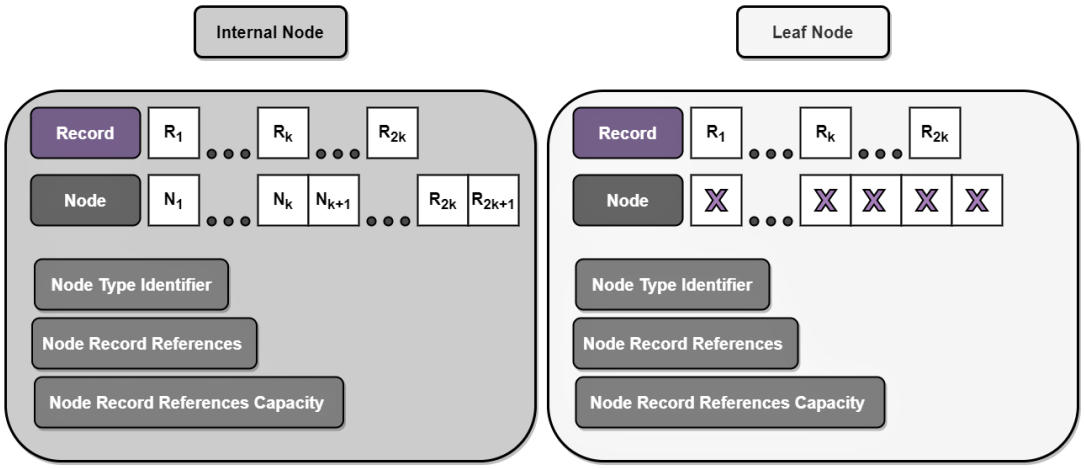


The B-tree index data structure consists of nodes which are basic and initial structural parts - blocks of the tree. The structural parts, characteristics and properties of each individual tree node are based on the node type. There are two separate node types, the intermediate - internal nodes and the leaf nodes. The B-tree structure shown in Fig. 2.1 is formed and structurally organized in levels that are composed of node sets. The levels set of the structure that are at a height - depth higher than the last level is composed of internal B-tree nodes. The last level of the structure consists exclusively and completely of leaf nodes which are all located at the same last tree level. Each path from the root node to any leaf node has the same length - height. The node that is at the first level of the tree is defined as the root node and is a potential internal node in case that the first level of the tree is not identical to the last tree level, in which case the root node is a leaf node. The basic property and characteristic of each node and level is the height - depth, which is defined as the path - set of nodes or levels from the root node (first - top level) to the associated node (B-tree node height). Approaching the height property from a different perspective, we can define it as the total number of transitions to be made between connected nodes of different levels of the B-tree in order to move from the root node to some leaf node of the last - bottom structure level (B-tree height). Another B-tree characteristic is the branching factor, which is basically the maximum number of node references (next level linked nodes) that each node can contain - store.

This structural organization and formation of the B-tree nodes defines the property of the structural tree balance. The structural balance is implemented through a set of algorithmic techniques of nodes and stored record references rearrangement and reorganization, which are incorporated in all of the basic insertion, deletion, search - selection and modification – update functions. Consequently, the B-tree is structurally self-balanced as it transforms and modifies its structure depending on its functionality.

2.1.3 B-tree index node structure

Figure 2.2: B-tree node structure

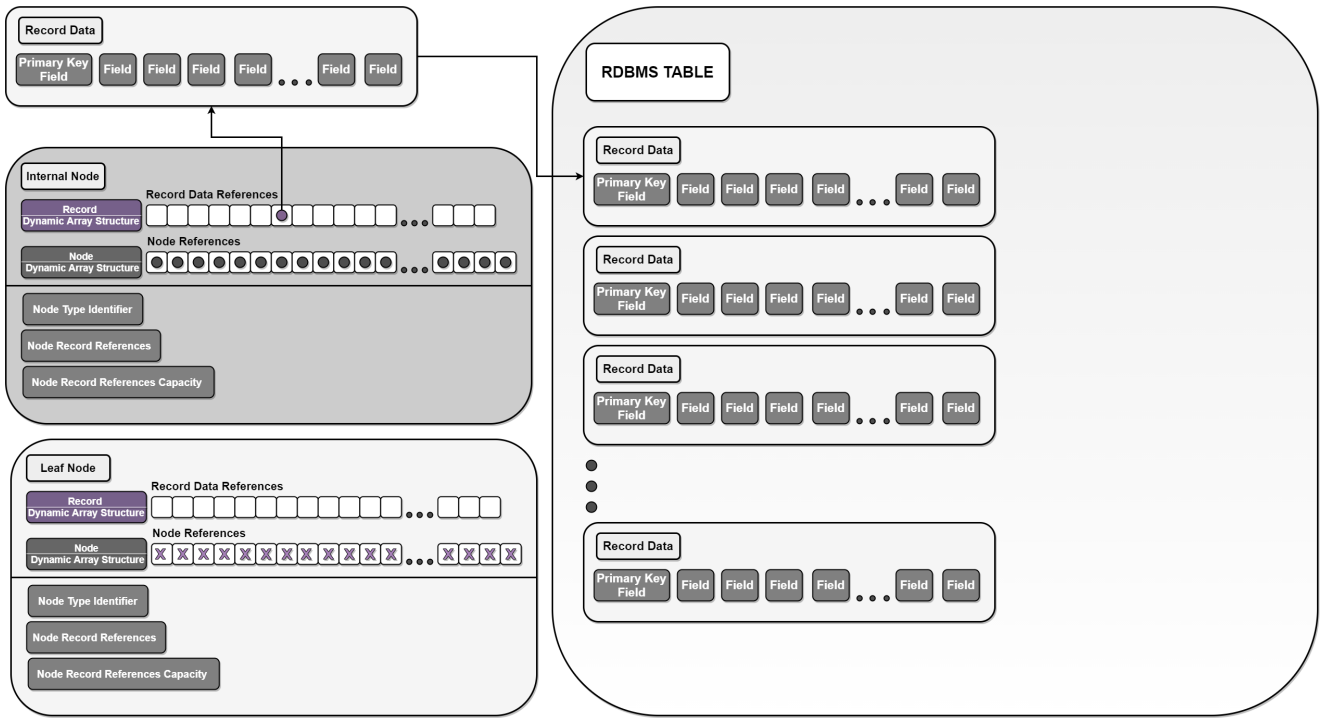


The B-tree node structure shown in Fig. 2.2 consists of the following structural parts – blocks and is governed by the listed characteristics and properties:

- B-tree is a multi-valued tree data structure since each node can store multiple data items - records. The parameter k constitutes the sets of records that can be stored in a B-tree node.
- Each internal node can contain multiple ordered record and node references.
- Each leaf node can contain exclusively multiple ordered record references.
- Each internal node can store at most $2k$ record and $2k + 1$ node references, where $k \geq 1$.
- Each leaf node can store at most $2k$ record references.

- Each node can theoretically store between 1 and $2k$ record references. Furthermore, each internal node can store between 1 and $2k + 1$ node references.
- Each node contains between k and $2k$ record references except the root node, which contains between 1 and $2k$ record references.
- In this implementation, each internal node is composed of two semi-dynamic array data structures that store record and node references. Each node semi-dynamic array can modify (increase and decrease) its capacity in order to reduce the memory allocation and usage.
- Each internal node contains three variables that specify the node type, the number of stored array structure references and the maximum capacity of the reference array structure.
- In this implementation, each internal node (except the root node) can store approximately between k and $2k$ record and between $k + 1$ and $2k + 1$ node references.
- In this implementation, each leaf node is composed of two semi-dynamic array data structures that store record and node references and modify (increase and decrease) its capacity in order to reduce the memory allocation and usage. The array of node references is completely empty and has not available allocated memory.
- Each leaf node contains three variables that specify the node type, the number of stored array structure references and the maximum capacity of the reference array structure.
- In this implementation, each leaf node (except the root leaf node) can approximately store between k and $2k$ record references.
- The default record array capacity of a new internal node is approximately at most k and the node array capacity is $k + 1$.
- The default record array capacity of a new leaf node is approximately at most k and the node array capacity is 0.
- The record array capacity is approximately at most k and the node array capacity is at most $k + 1$ depending on the size (1 to k) of the stored record references.
- The record array capacity is approximately at most $2k$ and the node array capacity is at most $2k + 1$ depending on the size ($k + 1$ to $2k$) of the stored record references.

Figure 2.3: B-tree node structure and system architecture



In this implementation, each record block is composed of a data field set. This set consists of fields - attributes of different data types. Each record has a unique identifier, the primary key field that separates them from the other records in the RDBMS table to which each record element belongs. The B-tree is a clustered - primary key index structure as it stores the records references based on the records primary key fields. The available valid records primary key fields data types of the developed B-tree structure can be either integer or string. The structure, design and architecture of the B-tree system is represented in Fig. 2.3. The implemented B-tree index structure system is based on a main memory (RAM) system and each record and node data reference constitutes a link to one or more virtual memory blocks - frames (set of memory elements) that store the data.

2.1.4 B-tree nodes number and height approximation

According to the theoretical analysis of the B-tree index structural and functional parameters in [1] [2] [6] [7] [8], the parameter n ($n \geq 0$) is defined as the total number of the B-tree stored record references and can be approximately approached and calculated by the relations:

$$n_{min} = 2(k+1)^{h-1} - 1 \tag{2.1}$$

$$n_{max} = (2k+1)^{h} - 1 \tag{2.2}$$

$$2(k+1)^{h-1} - 1 \leq n \leq (2k+1)^{h} - 1 \tag{2.3}$$

The parameter h ($h \geq 0$) is defined as the B-tree structure height - total B-tree nodes levels and can be approximately calculated by the relations:

$$h_{min} = log_{(2k+1)}(n+1) \tag{2.4}$$

$$h_{max} = 1 + log_{(k+1)}(\frac{n+1}{2}) \tag{2.5}$$

$$log_{(2k+1)}(n+1) \leq h \leq 1 + log_{(k+1)}(\frac{n+1}{2}) \tag{2.6}$$

According to [8], each B-tree node, except the root, contains on average $2ku_k$ record and $2ku_k + 1$ node references. Based on this work, the total average B-tree structure height can be approximately calculated by the relation:

$$h_{avg} \cong log_{(2ku_k+1)}(n), \quad 2k \leq n \quad and \quad h = 1, \quad 2k > n \tag{2.7}$$

Furthermore, if the root node contains k record references then a more theoretically ideal B-tree structure height approximation (approach of the optimal height reduction) can be calculated by the relation:

$$h_{avg} \cong 1 + log_{(2ku_k+1)}(\frac{n}{k+1}), \quad 2k \leq n \quad and \quad h = 1, \quad 2k > n \tag{2.8}$$

The u_k parameter ($u_k \cong ln_2 \cong 0.69315, 0.66666 \leq u_k \leq 0.69315$) is constant and is defined as the storage utilization, i.e., the ratio of the record references to the record references allocated memory slots in the B-tree nodes set [7] [8].

Based on the theoretical analysis of the aforementioned studies, we can approximately calculate the total B-tree nodes d ($d \geq 0$) by the relations:

$$d_{min} = 1 + \frac{2}{k}((k+1)^{h-1} - 1) \tag{2.9}$$

$$d_{max} = \frac{1}{2k}((2k+1)^{h} - 1) \tag{2.10}$$

$$1 + \frac{2}{k}((k+1)^{h-1} - 1) \leq d \leq \frac{1}{2k}((2k+1)^{h} - 1) \tag{2.11}$$

Based on [8], the average bottom level leaf nodes set of the B-tree structure d_{ln} can be approximately approached more accurate using a tighter relation:

$$d_{ln} \cong \frac{n+1}{(2k+2)(H(2k+2) - H(k+1))}, \quad 2k \leq n \quad and \quad d_{ln} = 1, \quad 2k > n \tag{2.12}$$

$$H(m) = \sum_{i=1}^{m}(\frac{1}{i}), \quad m \geq 1 \tag{2.13}$$

According to the same work, the parameter p is defined as the average number or the possibility of leaf and internal nodes splits per insertion process in the B-tree structure and can be calculated by the following relation:

$$p = \frac{1}{2ln(2)k} \cong \frac{1}{1.3863k} \leq \frac{1}{k} \tag{2.14}$$

Equation 2.14 is not theoretically identical to the deletion operation. As already mentioned, the deletion operation is not the same function with the insertion function from a functional (algorithmic perspective). The deletion function implements a set of rearrangement and reconstruction algorithmic methods in order to keep the B-tree nodes and stored record references set structure stable and balanced as opposed to the insertion function that implements split algorithmic processes in order to keep the structural balance. However, the deletion operation tends to keep the existing and prior structure and arrangement of the tree nodes.

Consequently, this relation cannot be used to calculate the average number of nodes and record references rearrangement and reconstruction per deletion as well as the number of insertion splits is calculated.

An average approach of the parameter d based on relations 2.11 – 2.14 can be approximately calculated by the following relation:

$$d \cong p \cdot n + h, \quad 2k \leq n \quad and \quad d = 1, \quad 2k > n \tag{2.15}$$

Equation 2.15 is based on the node (leaf - internal node) split, rearrangement, reconstruction and structural re-organization algorithms of the insertion and deletion functions in order to structurally re-balance the B-tree index structure. The insertion function implements the node split and the records - nodes references semi-dynamic array structures rearrangement - reconstruction algorithmic sub-functions. Each individual discrete split sub-function creates a new B-tree node. Consequently, each B-tree node except the first - leaf root node is created by the insertion node split and rearrangement - reconstruction sub-functions. Furthermore, each internal root node split algorithmic sub-function creates two nodes, the right sub-node and the new root node. The internal node split process is quite rare since it is implemented when the insertion path from the root to the leaf level is composed of a different linked nodes set that each node contains $2k$ record references. Each root node split increases the B-tree height by one. Consequently, the total root node split processes are $h-1$ because the first leaf root node is not created by a node split. Equation 2.15 provides a theoretical approximation of the total B-tree structure nodes as the B-tree nodes set is created based on the nodes splits set that is related to the parameters p and n.

The relation 2.17 provides a more complete, optimal and accurate average approach of the parameter d. The total average bottom level leaf nodes of the structure can be approximately calculated by Equation 2.12. Based on the previous relation the total stored record references in the upper internal B-tree levels (internal nodes set) can be approached as $d_{ln} - 1$ for n stored record references in the structure. From a theoretical perspective, the upper internal nodes levels of the structure constitute a sub-tree if the bottom leaf nodes level extracted - detached from the overall tree structure. Equation 2.12 can be applied to this extracted sub-tree (tree that is composed of all the internal levels nodes) that contains $d_{ln} - 1$ stored record references in order to calculate the total average bottom level leaf nodes of this individual subset of nodes (tree). This process is implemented recursively for each individual sub-tree up to the root node in order to calculate the total average bottom levels leaf nodes of each sub-tree. These leaf nodes sets constitute the total average B-tree nodes.

$$\begin{aligned} &T(b) = 1, \quad b = 1 \\ &T(b) = dl(T(b+1) - 1), \quad 1 < b < h \\ &T(b) = dl(n), \quad b = h \end{aligned} \tag{2.16}$$

Consequently, based on relation 2.16, the total average nodes d of the B-tree can be approached as the sum of all sub-trees leaf levels nodes by the equation:

$$\begin{aligned} &d = \sum_{i=0}^{h-2} (T(b-i)) + T(1), \quad h > 2 \\ &d = dl(n) + 1, \quad h = 2 \\ &d = 1, \quad h = 1 \end{aligned} \tag{2.17}$$

According to Equations 2.12 – 2.17, the parameter d_{in} is defined as the total average number of internal B-tree nodes and can be approximately calculated by the following 2.18 relation:

$$d_{in} \cong d - d_{ln} \tag{2.18}$$

As already defined, the total number of B-tree nodes d is the sum of the B-tree internal and leaf levels nodes.

The total average record references that are stored in the internal B-tree nodes r_i 2.19 can be approximately approached by the relation:

$$r_i \cong d_{ln} - 1 \tag{2.19}$$

The total average record references that are stored in the leaf B-tree nodes r_l 2.20 can be approximately approached by the relation:

$$r_l \cong n - r_i \tag{2.20}$$

2.2 B-tree index structure basic functional levels

2.2.1 B-tree index structure functions

Indexing systems are composed of multiple special designed structural components, data structures and mechanisms [15] [16] [17] [18] [19] [20] [21]. The indexing systems implement a set of functions and transactions through a set of index structures for the efficient, fast, accurate and secure management of the RDBMs in-memory and on-disk file systems relational tables stored records sets. The B-tree data structure is a basic and fundamental index tree structure that is used extensively in the RDBMs indexing sub-systems and composes the structural and functional base for the development and implementation of multiple efficient tree index structures set and their variants [2] [3] [22] [23]. These structures are used in multiple modern storage management - file system engines, such as RDBMs (MySQL [24] [25] [26] [27] [28] [29], PostgreSQL [30] [31] [32], Oracle Database [33], SQLite [34] [35] etc), software applications, operating systems (BTRFS Linux file system [36] [37]), indexing - caching systems (Redis etc) and in other scientific and technological fields with specially designed implementations.

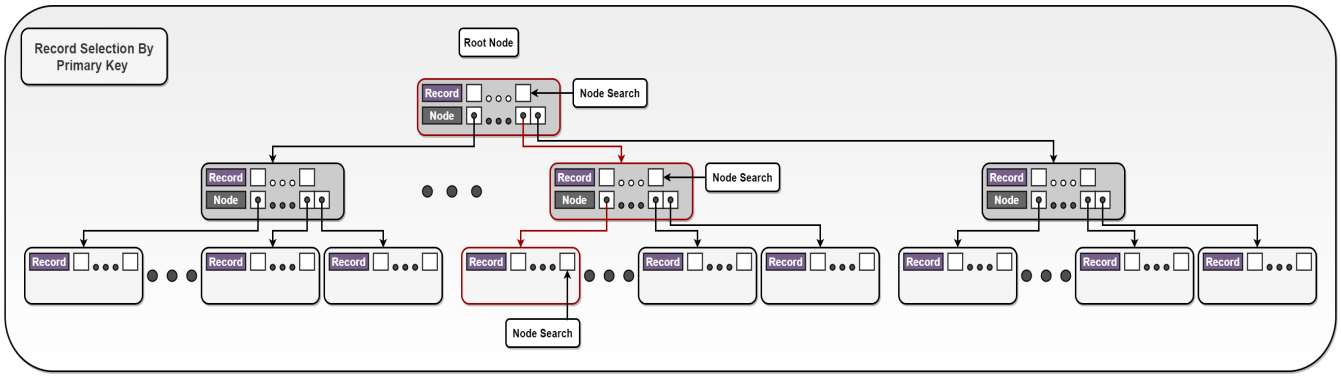
The B-tree index structure functions that are analyzed and implemented in this work are the record references sets insertion, deletion and selection functions:

- Records selection by record primary key field.
- Records selection by multiple record fields.
- Records insertion and deletion based on the record primary key field.

The B-tree visualization tool in [38] can be used in order to theoretically simulate the insertion, deletion and selection - search B-tree functions with high precision and accuracy.

2.2.2 Records selection by primary key fields

Figure 2.4: Records selection by record primary key field



The selection function listed in Alg. 1 and schematically represented in Fig. 2.4 recursively implements a set of node selection - search algorithmic sub-functions. Each individual sub-function in Alg. 3 is applied to a B-tree index structure node in order to locate (based on a primary key field) the node in the B-tree structure and the position of the stored record reference in the B-tree node semi-dynamic array structures or the location of the next level linked node that the record reference could be stored. In this implementation, the sub-function in Alg. 3 implements the record reference location and selection in the node semi-dynamic array structure using optionally both the specially

designed binary - BTree_Binary_Search() and interpolation - BTree_Interpolation_Search() search algorithms. The sub-function in Alg. 2 implements the node selection - search sub-function to locate and select the node and stored record reference in the B-tree index structure levels nodes sets applying this process on a nodes path from the root node to the bottom leaf nodes level.

The binary and interpolation search algorithms are quite efficient and fast to the location and selection in the B-tree nodes semi-dynamic array structures of the record references and next level linked nodes that the record references could be stored. The binary search algorithm has an average theoretical time complexity - performance $O(log_2(n))$ and the interpolation search has $O(log_2(log_2(n))$ average time complexity [39]. Consequently, these algorithms are used extensively as the node-level location and selection functions of the B-tree index structure and its variants.

Algorithm 1: BTreeFastSearchData_ByPrimaryKey function

```
Returned item: Selected record
 BTreeFastSearchData_ByPrimaryKey(
 B-tree structure item,
 Primary key field of the record to be selected
 )
if B-tree data structure is empty then
    Return null item.
end
Return BTreeFastSearch_Tool() item.
```

Algorithm 2: BTreeFastSearch_Tool function

Returned item: Selected record
BTreeFastSearch_Tool(
B-tree node item,
Primary key field of the record to be selected
)

SearchBTreeNode_Record_ByPrimaryKey()

if *record with the specific primary key field is located in the current node* **then**
    Return record item.
end

if *Current node is internal* **then**
    Return BTreeFastSearch_Tool() item.
end

Return null item.

Algorithm 3: SearchBTreeNode_Record_ByPrimaryKey function

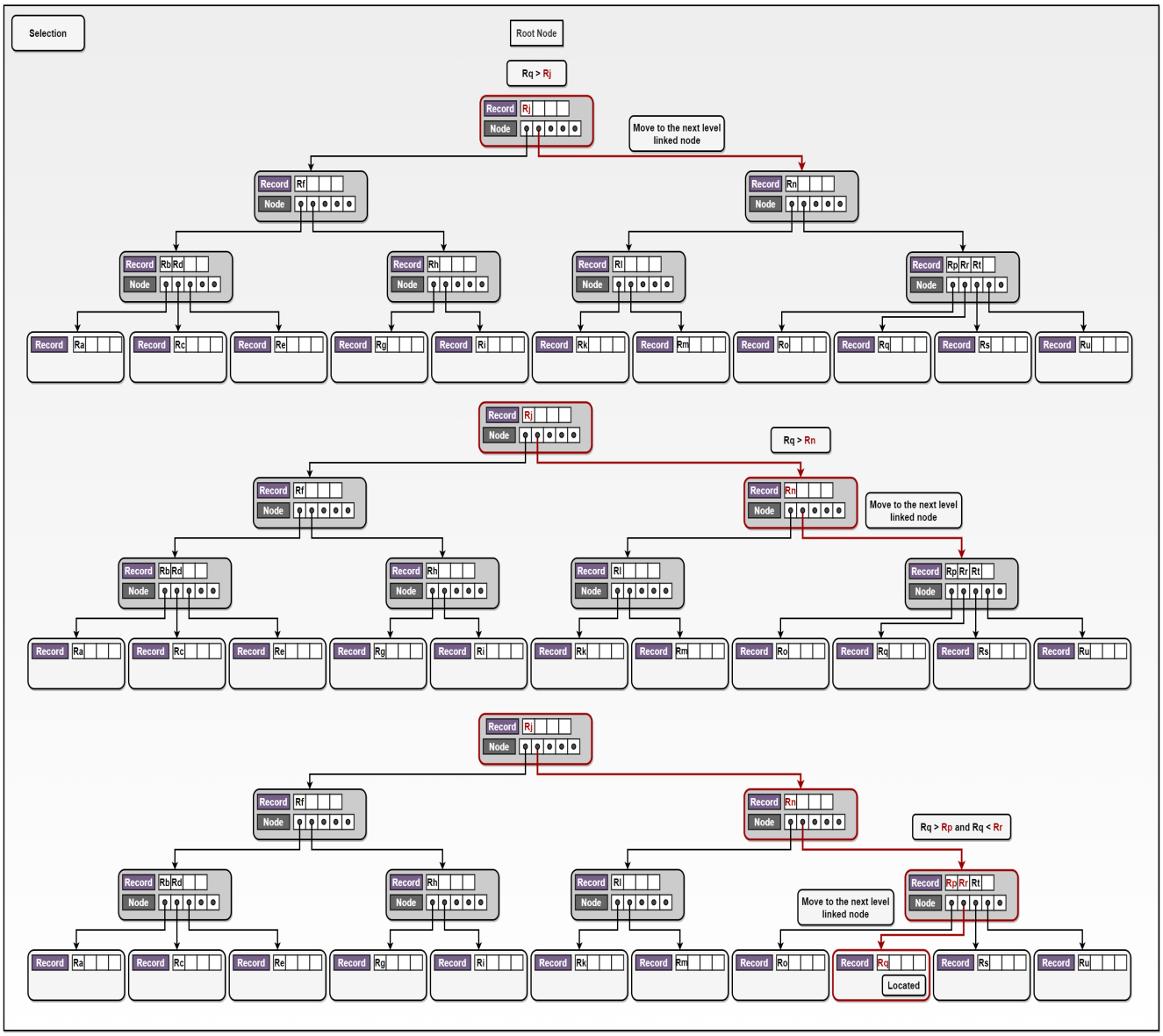
Returned item: Node or semi-dynamic node array position that can be located the record to be selected
SearchBTreeNode_Record_ByPrimaryKey(
Stored records in the node Semi-dynamic array structure,
Node Semi-dynamic array structure item,
Primary key field of the record to be selected
)

Return BTree_Binary_Search() or **BTree_Interpolation_Search()** item.

Furthermore, the linear search method can be implemented and applied to the node search sub-function even though it is not an efficient search method due to its average and worst theoretical time complexity, which is $O(n)$. For this reason, we opted not to implement this method in our code but we just present its implementation for completeness.

The record references location - selection in the B-tree index structure nodes semi-dynamic array structures based on a primary key field shown in Fig. 2.5 utilizes the linear search algorithm in order to provide a more complete, accurate and simple description of the selection process. The Figure shows the selection function of the record reference with record primary key field R_q.

Figure 2.5: Selection of the record reference with record primary key field R_q



The total average linked nodes set - path from the root node to the leaf nodes level composes the total average B-tree structure height [8]. The selection sub-function in Alg. 3 applied to that nodes set semi-dynamic array structures that the record references are stored in order to implement the record reference location and selection using the binary and interpolation search algorithms. The average time complexity of the binary and interpolation search algorithms is $O(log_2(n))$ and $O(log_2(log_2(n))$ [39].

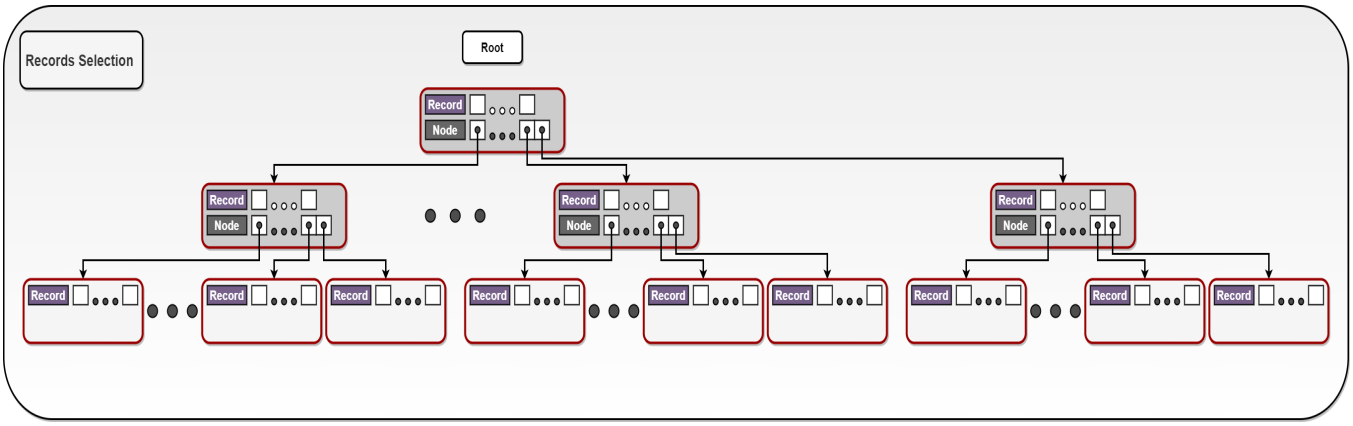
Consequently, the theoretical average time complexity of the selection function in Alg. 1 and Alg. 2 can be calculated by the relations 2.21 and 2.22, respectively:

$$O(\ log_{(2)}(2k) \cdot log_{(2ku_k+1)}(n)\ ), \quad using\, binary\, search \tag{2.21}$$

$$O(\ log_{(2)}(log_{(2)}(2k)) \cdot log_{(2ku_k+1)}(n)\ ), \quad using\, interpolation\, search \tag{2.22}$$

2.2.3 Records selection by multiple fields (constraints)

Figure 2.6: Records selection by multiple record fields



The selection function in Alg. 4, shown in Fig. 2.6, of multiple non primary and primary record references based on a set of individual discrete conditions (constraints) compose a complete and full scan - selection function of the B-tree index structure nodes and stores record references set in order to locate and collect these data to gather and store them in a data structure. In this implementation, Alg. 5 implements the search procedure.

Furthermore, we use a specially designed Double Linked List data structure which stores and contains the selected record references set in both ascending or descending order of the selection process. The selection function uses the node transition sub-function in Alg. 5 as a basic functional part of the overall selection process that implements the recursive ascending transition from a B-tree node to an other level linked B-tree node scanning and selecting the record references that are stored in the semi-dynamic array of this node based on a selection constraints set. The selection function has also been implemented to store the selected data in descending order - layout using a Double Linked List function that stores the selected record references to the position of the List structure head node.

In general, the overall selection process can be implemented recursively based on some full scan search - traversal algorithmic technique as the inorder, preorder, postorder and other full (complete) tree traversal techniques. In the context of this implementation the inorder tree traversal algorithmic technique is used.

Algorithm 4: BTreeSelectRecordData_ASC function

Returned item: Selection process status
BTreeSelectRecordData_ASC(
B-tree structure item,
Double Linked List item that the selected records set will be stored,
Record field attribute that compose the selection condition
)

if *B-tree data structure is empty* **then**
    Return unsuccessful selection status.
end

BTreeSelectRecordData_ASC_Tool()

if *Double Linked List data structure is empty* **then**
    Return unsuccessful selection status.
end

Return successful selection status.

Algorithm 5: BTreeSelectRecordData_ASC_Tool function

Returned item: Void item
BTreeSelectRecordData_ASC_Tool(
B-tree node item,
Double Linked List item that the selected records set will be stored,
Record field attribute that composes the selection condition
)

if *Current node is leaf* **then**

Node semi-dynamic array structure traversal to locate and select the record based on the specified condition.

Each selected record stored ascending (to the end of the List) using the **InsertBListNode_Last()** function in the selection Double Linked List.

Return void item.

end

while *Node Semi-dynamic array structure traversal is not completed* **do**

BTreeSelectRecordData_ASC_Tool()

Record location and selection in the semi-dynamic array structure based on the specified condition.

Each selected record stored ascending (to the end of the List) using the **InsertBListNode_Last()** function in the selection Double Linked List.

end

BTreeSelectRecordData_ASC_Tool()

As already analyzed, the selection function is implemented as a continuously recursive scan process of the n record references that are stored in the B-tree structure nodes set in order to locate and select a subset of the stored record references set based on the selection conditions. Consequently, the theoretical average time complexity of the selection function is $O(n)$.

2.2.4 Records insertion based on primary key fields

The insertion function in Alg. 6 is composed of multiple recursively linked sub-functions in Alg. 7 and Alg. 8. Each individual sub-function implements a discrete functional part of the overall insertion process. The insertion sub-functions use the split node algorithm as a basic B-tree index structure balancing technique.

The sub-function in Alg. 7 implements the root node split process and the reconstruction and rearrangement of the record and node references semi-dynamic array structures. The node (root node) splits (Fig. 2.9– 2.12) into two distinct structural parts - sub-nodes and the right half stored record and node references of the node semi-dynamic array structures transferred to the new node. The middle record reference that is stored in the initial node (root node) array structure transferred and stored in the record reference semi-dynamic array structure of the upper level linked node (parent node). In this case that the splitting node is the root node a new root node is created in order to be transferred the middle record reference. The insertion sub-functions in Alg. 7 and Alg. 8 use the split node algorithm to reconstruct, rearrange and organize the B-tree nodes set structure in order to restore the structural balance.

Alg. 8 implements the insertion - storage of a record reference in a leaf node. The record reference is stored to the leaf node if the leaf node record semi-dynamic array structure has available allocated memory - capacity to store the inserted record (Fig. 2.7 and Fig. 2.8). If the record semi-dynamic array structure of the leaf node has not available memory - capacity to store the record reference the leaf node splits and a reconstruction and rearrangement process is caused (Fig. 2.9– 2.12. This process is repeated recursively up to the root node in order to structurally re-balance and stabilize the B-tree index structure.

Fig. 2.7 and 2.8 analyze and describe the record reference insertion in a leaf node record semi-dynamic array structure with available allocated storage memory - capacity:

Figure 2.7: Record reference insertion process in a leaf node with available record semi-dynamic array structure capacity - part 1

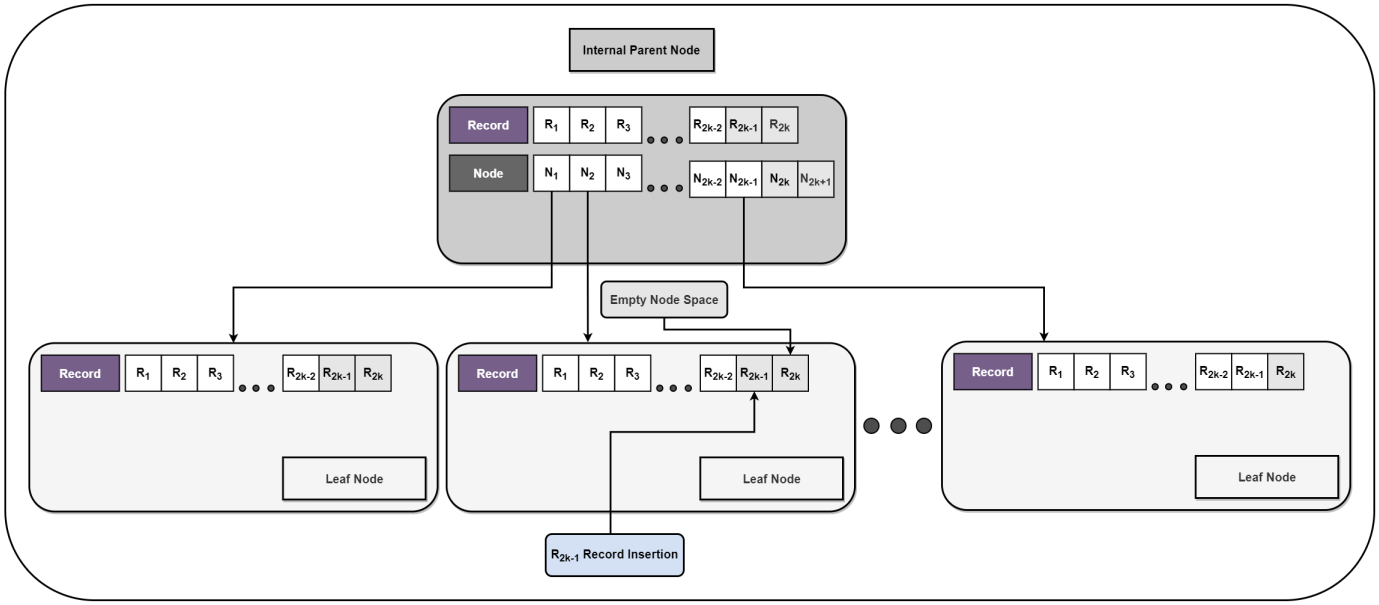


Figure 2.8: Record reference insertion process in a leaf node with available record semi-dynamic array structure capacity - part 2

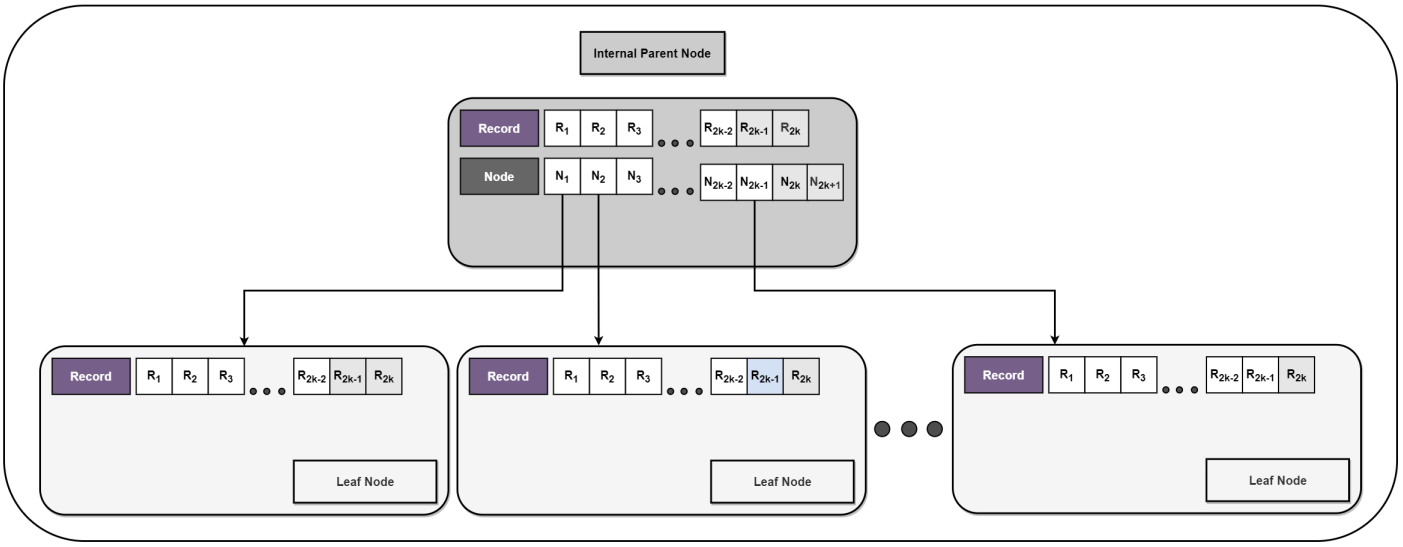


Fig. 2.9– 2.12 analyze and describe the record reference insertion in a leaf node record semi-dynamic array structure without available allocated storage memory - capacity and the leaf node split process:

Figure 2.9: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 1

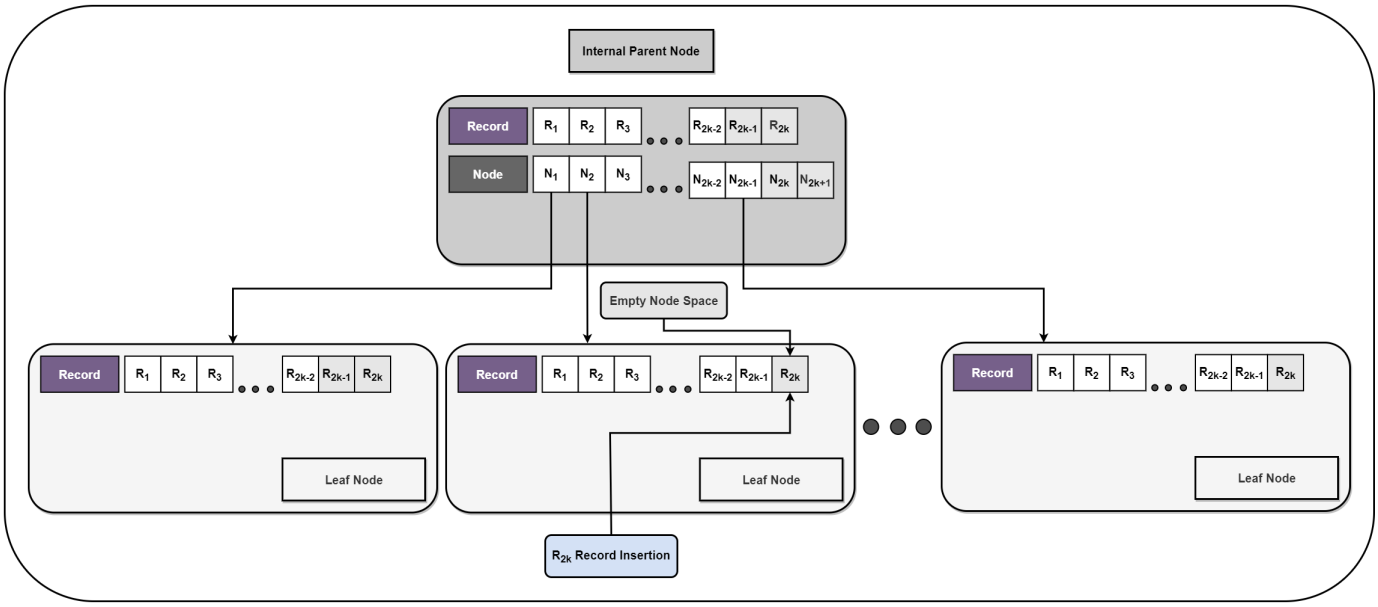


Figure 2.10: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 2

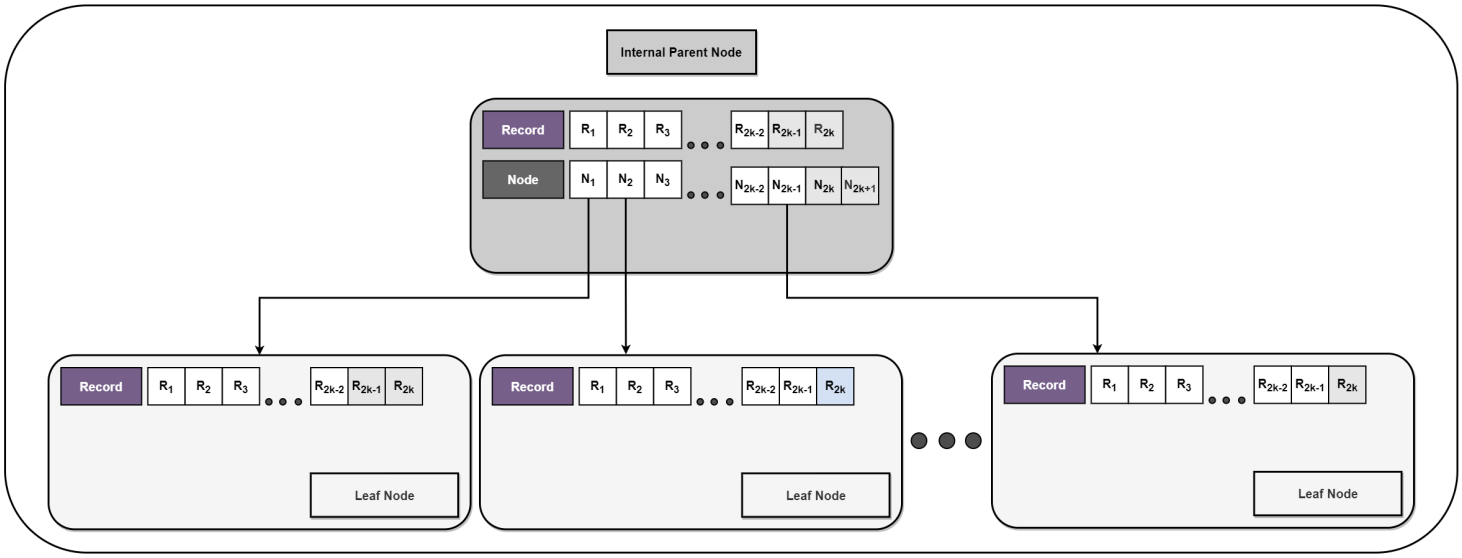


Figure 2.11: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 3

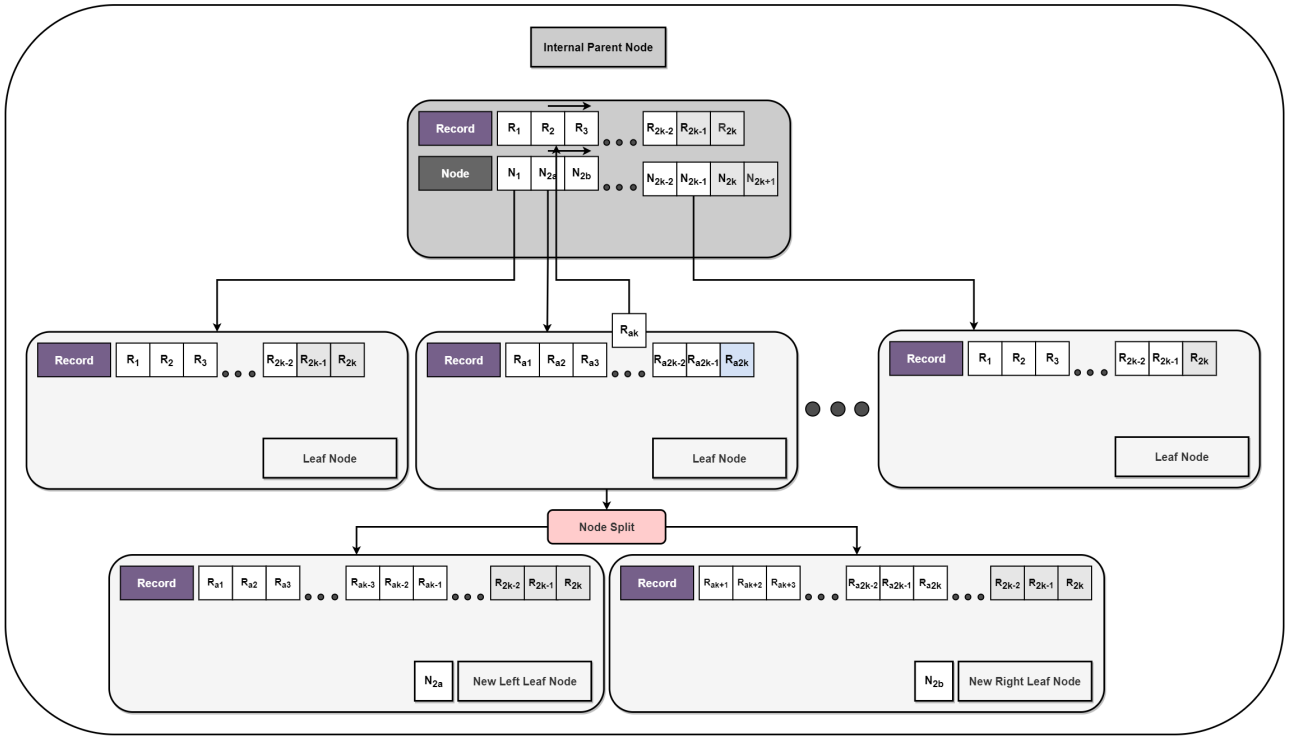
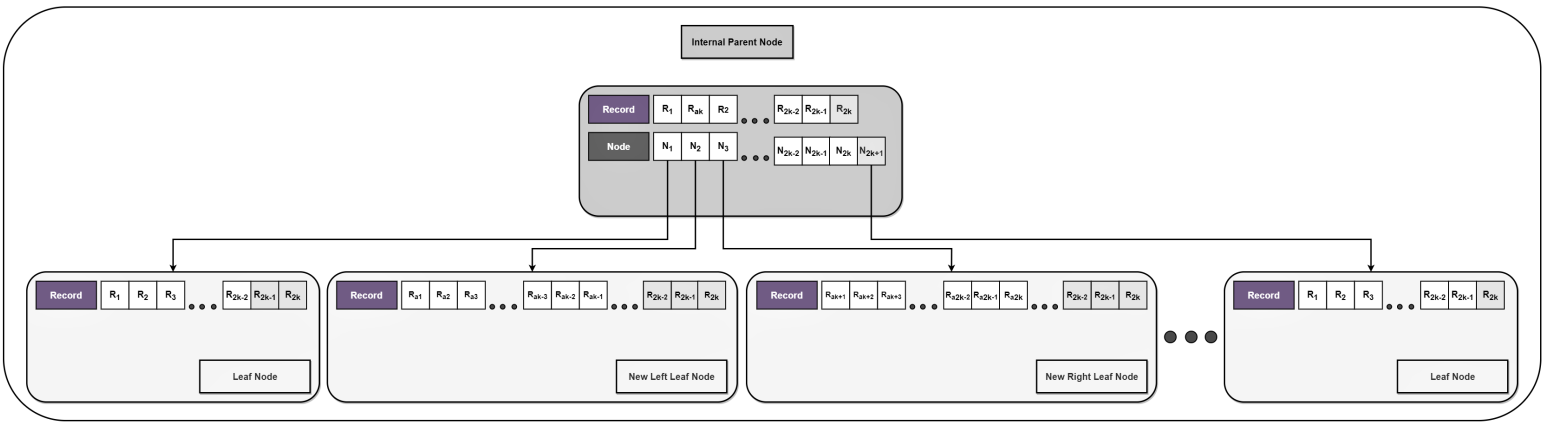


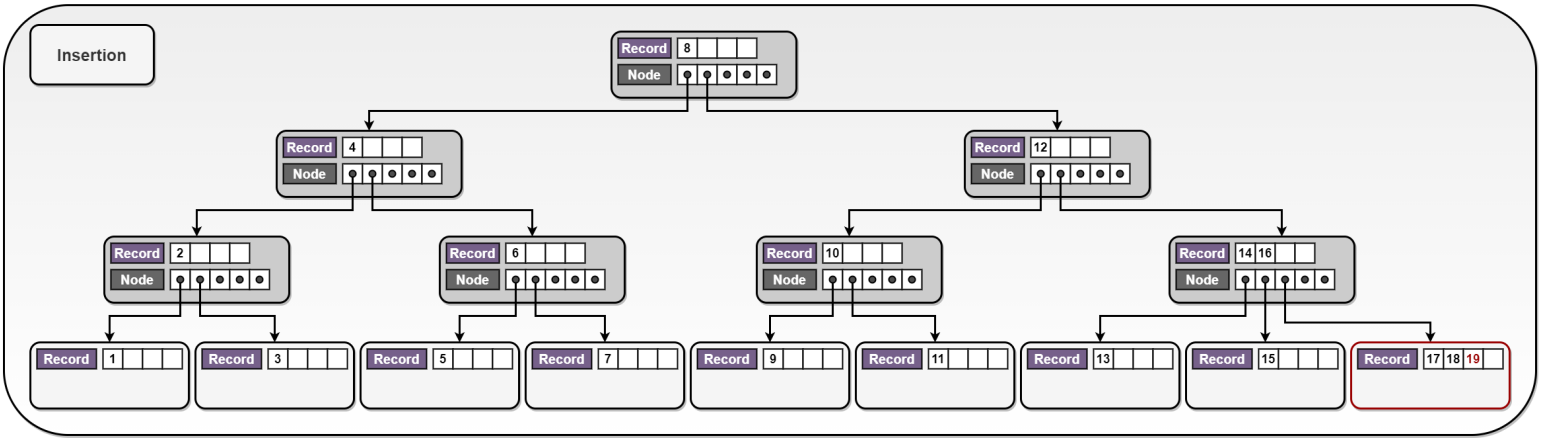
Figure 2.12: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 4



The insertion functions in Alg. 7 and 8 of the implemented B-tree index structure are based on the node and node semi-dynamic array structures split, reconstruction and rearrangement algorithmic method patterns:

- Case 1 - record reference insertion in a leaf node record semi-dynamic array structure with available allocated storage memory - capacity (Fig. 2.13):

Figure 2.13: Record reference 19 insertion process in a leaf node with available record semi-dynamic array structure capacity



- Case 2 - record reference insertion in a leaf node record semi-dynamic array structure without available allocated storage memory - capacity (leaf node split process) and the linked upper level node (parent node) has available capacity (Fig. 2.14 and 2.15):

Figure 2.14: Record reference 20 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has available capacity - part 1

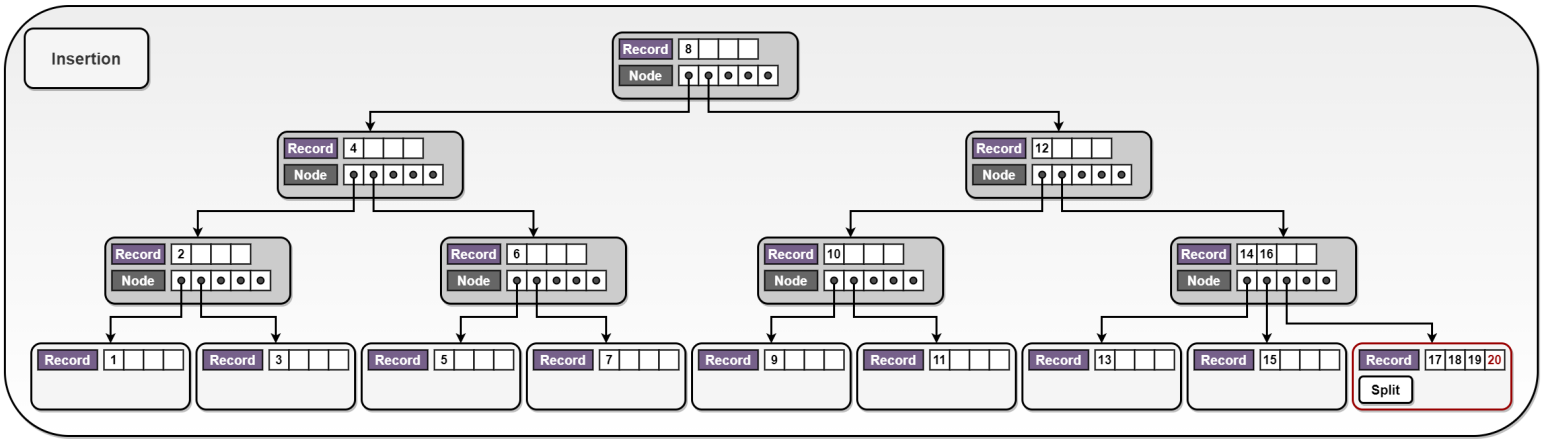
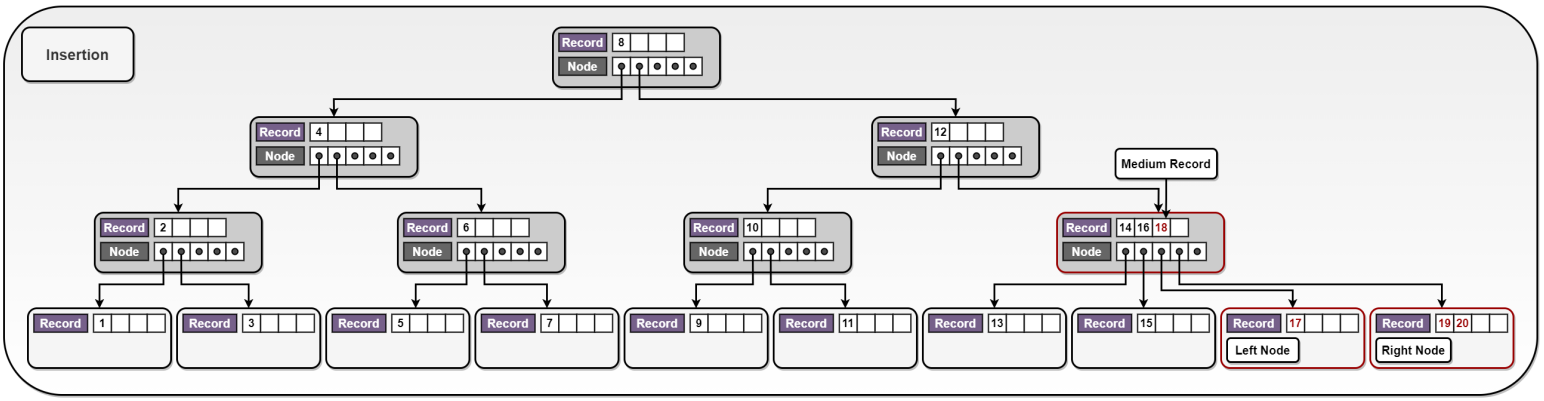


Figure 2.15: Record reference 20 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has available capacity - part 2



- Case 3 - record reference insertion in a leaf node record semi-dynamic array structure without available allocated storage memory - capacity (leaf node split process) and the linked upper level node (parent node) has not available capacity (linked upper level node split) (Fig. 2.16– 2.19):

Figure 2.16: Record reference 22 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has not available capacity (parent node node split process) - part 1

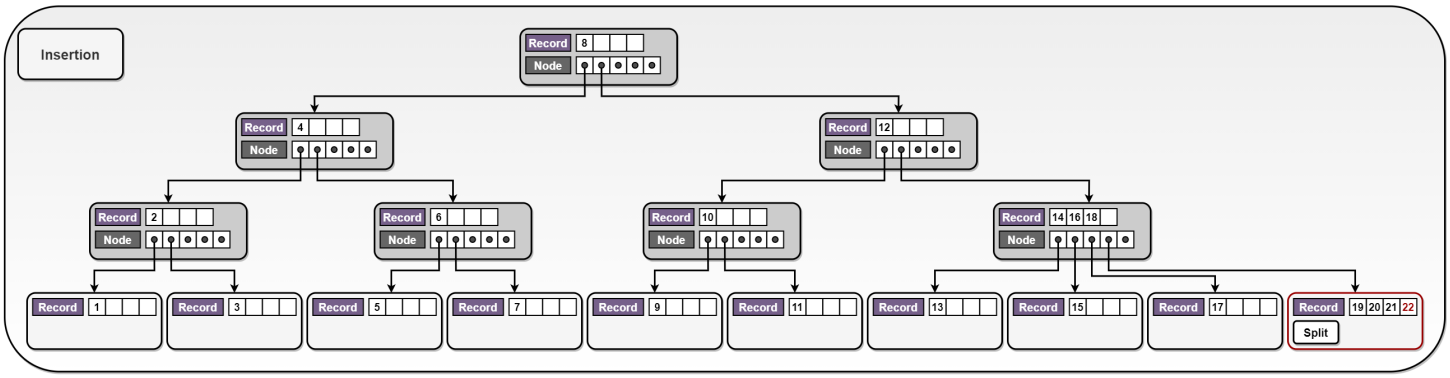


Figure 2.17: Record reference 22 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has not available capacity (parent node node split process) - part 2

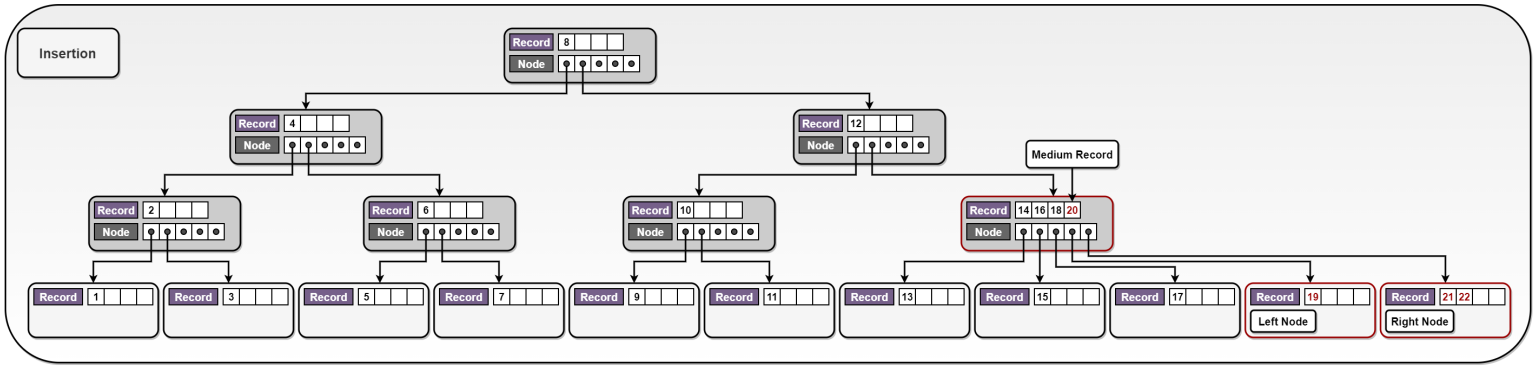


Figure 2.18: Record reference 22 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has not available capacity (parent node node split process) - part 3

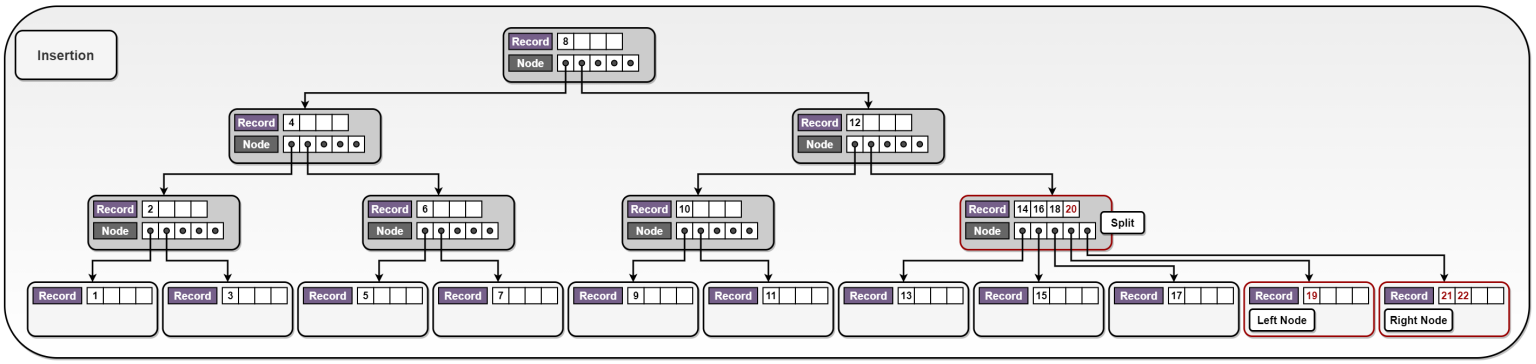
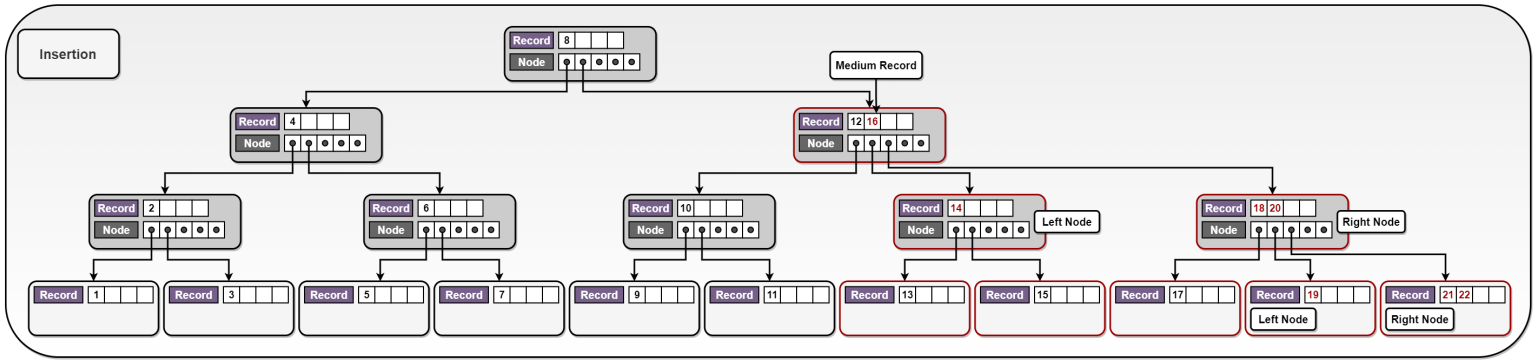


Figure 2.19: Record reference 22 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has not available capacity (parent node node split process) - part 4



- Case 4 - root node split process (root is internal - leaf node) (Fig. 2.20– 2.23):

Figure 2.20: Record reference 10 insertion process in a leaf node without available record semi-dynamic array structure capacity. The parent root node has also not available capacity (parent node split) - part 1

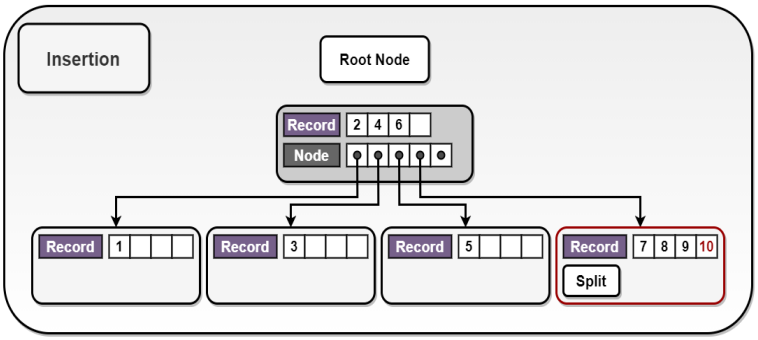


Figure 2.21: Record reference 10 insertion process in a leaf node without available record semi-dynamic array structure capacity. The parent root node has also not available capacity (parent node split) - part 2

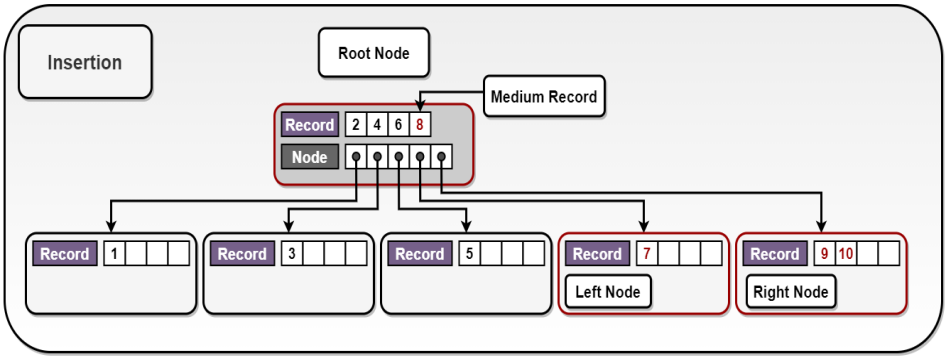


Figure 2.22: Record reference 10 insertion process in a leaf node without available record semi-dynamic array structure capacity. The parent root node has also not available capacity (parent node split) - part 3

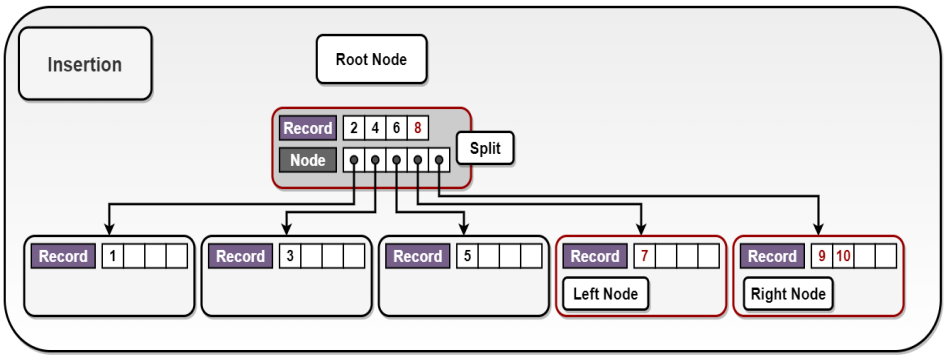
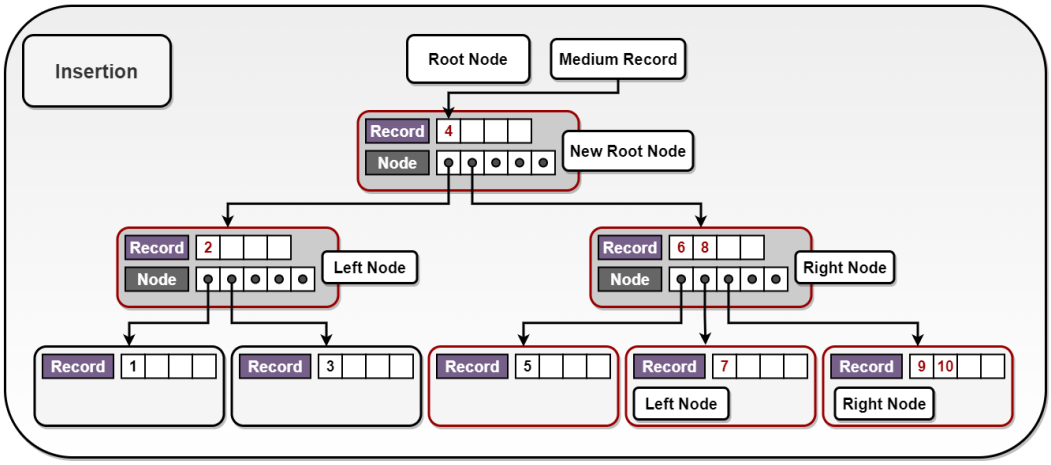


Figure 2.23: Record reference 10 insertion process in a leaf node without available record semi-dynamic array structure capacity. The parent root node has also not available capacity (parent node split) - part 4



Algorithm 6: BTreeInsertData function

Returned item: Insertion process status
BTreeInsertData(
B-tree structure item,
Record reference - data to be inserted
)

if *B-tree data structure is empty* **then**
    Root node construction - creation.
    Record reference insertion - storage in the root node.
    Return successful insertion status.
end

BTreeInsertNode_RootBreakTool()

if *BTreeInsertNode_RootBreakTool() is unsuccessful* **then**
    Return unsuccessful insertion status.
end

Return successful insertion status.

Algorithm 7: BTreeInsertNode_RootBreakTool function

Returned item: Insertion process status
BTreeInsertNode_RootBreakTool(
B-tree structure item,
Record reference - data to be inserted
)

BTreeInsertNode_Tool()

if *Record reference has already been stored - inserted in the B-tree structure (duplicate stored record reference)* **then**
    Return unsuccessful insertion status.
end

if *Current node is the root node and a node break - split process was performed* **then**
    Left sub-node creation.
    Reconstruction - rearrangement of the root, left and right nodes set
    record and node references semi-dynamic array structures that the split process was implemented.
end

Return successful insertion status.

Algorithm 8: BTreeInsertNode_Tool function

Returned item: Node split process - right sub-node item
BTreeInsertNode_Tool(
Current node item,
Record reference - data to be inserted,
Node split process - record references semi-dynamic array structure middle record,
Node record references semi-dynamic array structure capacity - size,
Duplication identifier of the inserted record reference
)

SearchBTreeNode_Record_ByPrimaryKey()

if *Record reference duplication* **then**
    Duplication identifier status update - unsuccessful insertion process.
    Storage obstruction - deallocation of the duplicate record reference.
    Return null node item.
end

if *Current node is a leaf node* **then**
    Record reference insertion - storage in the current leaf node.
else
    BTreeInsertNode_Tool()
    if *Next level linked node split process was implemented* **then**
        Storage - insertion of the next level linked node (split node) middle record reference in the current node.
        Reconstruction - rearrangement of the current node references and record references semi-dynamic array structures.
    else
        Return null node item.
    end
end

if *Current node record references semi-dynamic array structure has not available storage capacity* **then**
    Current node split (right sub-node creation) and node semi-dynamic array structures reconstruction - rearrangement.
    Return right sub-node item.
end

Return null node item.

The total average algorithmic operations - steps of the insertion function can be approximately be approached based on the average B-tree structure height as shown in Equation 2.7. Consequently, the theoretical average time complexity of the insertion function in Alg. 6–8 can be approximately calculated by the relation 2.23:

$$O(\ log_{(2ku_k+1)}(n)\ ) \tag{2.23}$$

2.2.5 Records deletion based on primary key fields

The deletion function is composed of multiple recursively linked sub-functions (functional levels) that each individual sub-function implements a discrete algorithmic part of the overall deletion process. Especially the deletion of a record reference and record data by a primary key field in the B-tree index structure is based on 4 different connected functional parts.

Alg. 9 implements the record reference location and deletion in the root node in the case that the root node is leaf and the B-tree index structure is composed of a single node. Furthermore implements the record reference location, deletion and the reconstruction and rearrangement (re-balance) of the B-tree nodes and stored record references sets utilizing Alg. 10 in case that the the B-tree is composed of multiple nodes and stored record references. Alg. 10 implements the record reference location, deletion and the reconstruction, rearrangement and re-balancing of the structure nodes and nodes stored record references sets utilizing Alg. 11 and 12 sub-functions. This sub-function composes the basic deletion method as it functionally links all the individual functional parts of the overall deletion process. The deletion process is separated in two functional parts, the record reference deletion that is stored in a leaf node and in an internal node. In particular, Alg. 10 implements the record reference location and deletion in the internal nodes set of the upper B-tree levels using the BTree_LeftSubTree_MaxRecord() sub-function that replaces the record reference to be deleted which is stored in the internal node with the maximum record reference of the leftmost leaf node path. Then implements the record reference deletion in the leaf node. Alg. 10 seeks the internal nodes path from the root node to the leaf node level using Alg. 3. In the case that the record reference is located or transferred by the previous internal node deletion process in a leaf node,

Alg. 11 sub-function is used to delete the record reference from the leaf node using a set of sub-function that apply multiple algorithmic methods in order to implement the deletion and balancing, reconstruction and rearrangement of the structure nodes and nodes dynamic array structures. Furthermore, Alg. 10 implements recursively the reconstruction - rearrangement of the B-tree nodes and the nodes stored record references based on Alg. 12 in order to structural re-balance the B-tree index structure. The algorithmic parts of the sub-functions in Alg. 11 and 12 are theoretically analyzed bellow. The record reference deletion on the bottom leaf nodes level is functionally based on the sub-function in Alg. 11. Alg. 11 implements the record reference location, deletion and the re-balancing, reconstruction and rearrangement of the B-tree nodes, the stored record and node references semi-dynamic array structures and the stored references in these structures on the leaf nodes level. In the case that the structural balance cannot be restored - recovered on the leaf nodes level, Alg. 12 is used in order to implement the nodes re-balancing, reconstruction and rearrangement process on the upper internal nodes levels of the B-tree to re-balance the tree. The sub-function in Alg. 11 and 12 uses a set of algorithmic tools and functions to reconstruct and re-balance the tree index:

- The sub-function BTree_ReplaceRecord() implements the deletion in a leaf node that contains multiple record references. The sub-function BTree_ReplaceRecord() implements the structural re-balancing of the B-tree on the bottom leaf level.

Figure 2.24: Deletion of the record 16 in a leaf node that contains multiple record references - part 1

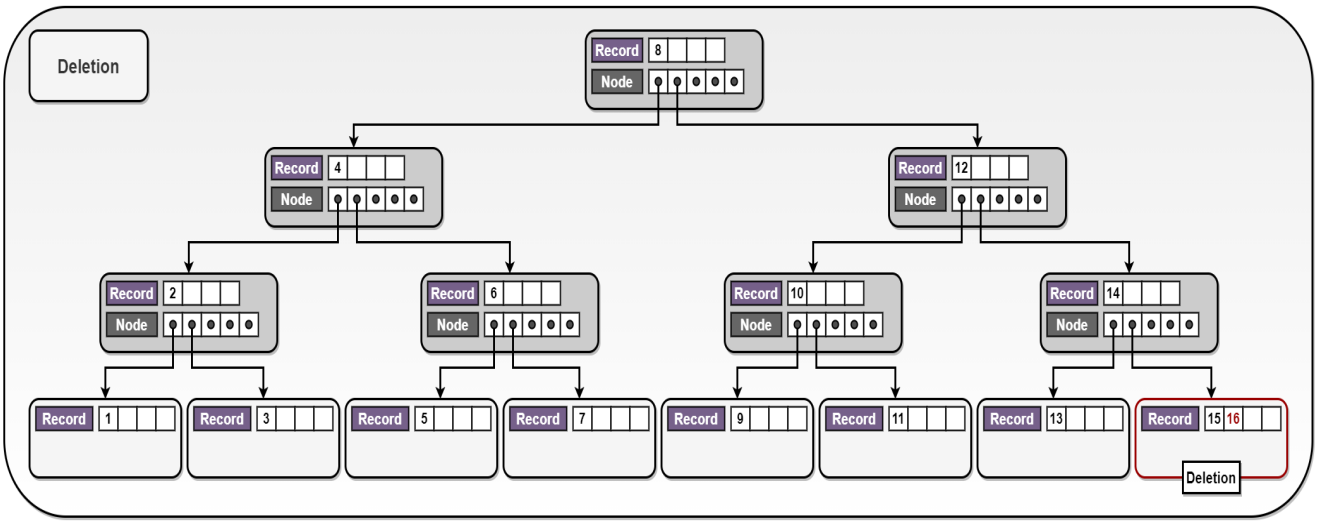
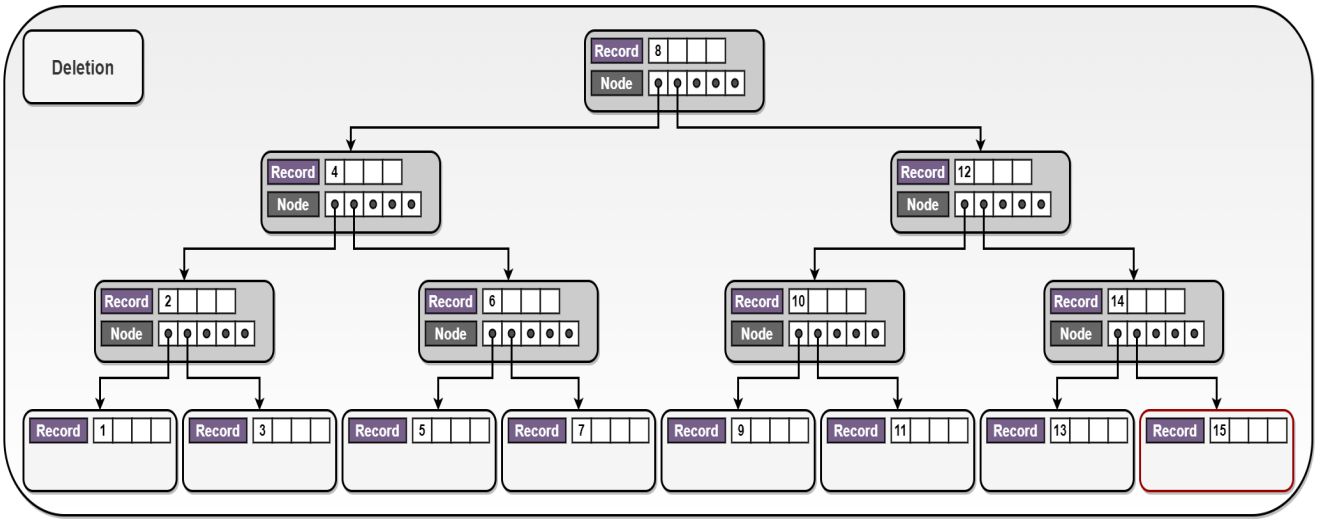
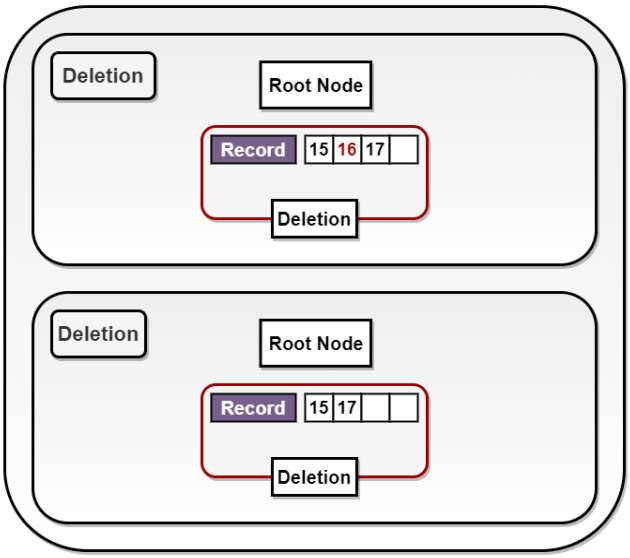


Figure 2.25: Deletion of the record 16 in a leaf node that contains multiple record references - part 2



- The sub-function BTree_ReplaceRecord() implements the deletion in a leaf root node that contains multiple record references. The sub-function BTree_ReplaceRecord() completely implements the structural re-balancing of the B-tree on the bottom leaf level.

Figure 2.26: Deletion of the record 16 in a leaf root node that contains multiple record references



- The sub-functions BTree_RebalanceLeftNode() and BTree_RebalanceRightNode() implement the deletion in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references. The sub-functions BTree_RebalanceLeftNode() and BTree_RebalanceRightNode() completely implement the structural re-balancing of the B-tree on the bottom leaf level.

Figure 2.27: Deletion of the record 16 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references - part 1

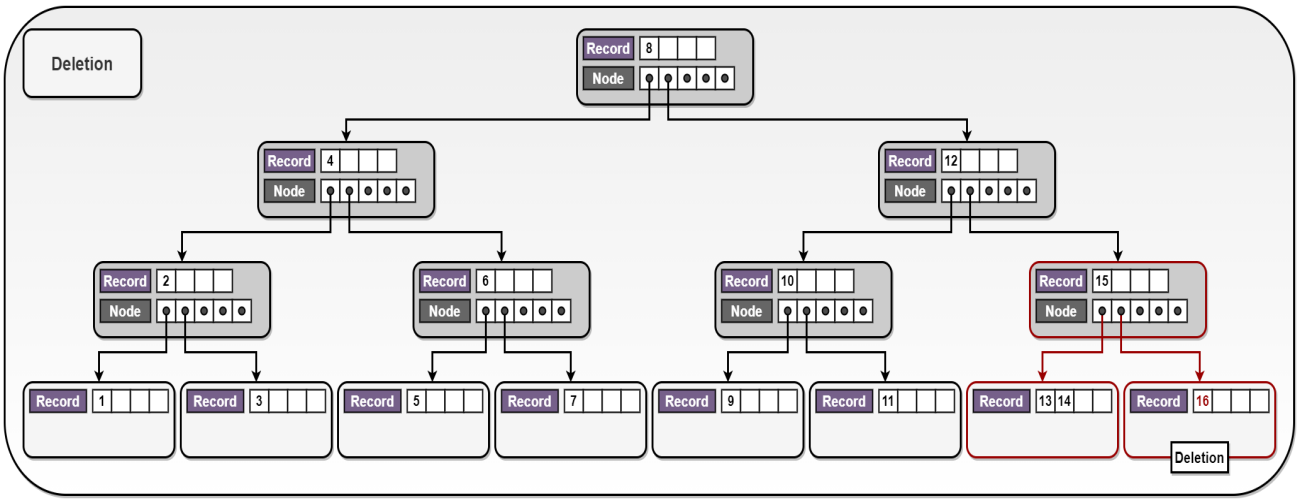
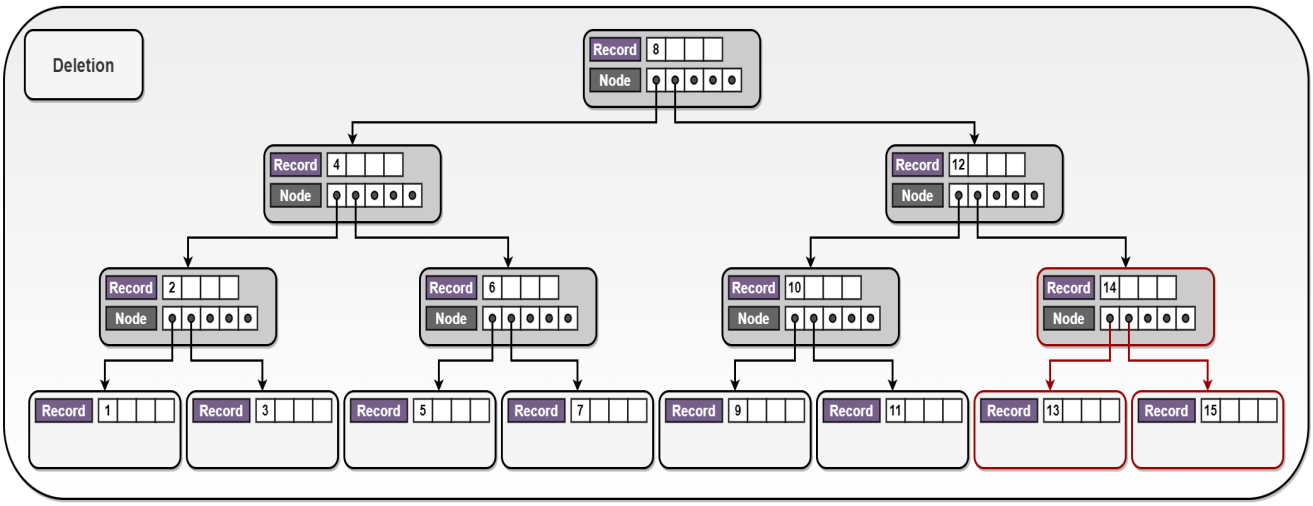


Figure 2.28: Deletion of the record 16 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references - part 2



- The sub-functions BTree_MergeSingleNodeRight() and BTree_MergeSingleNodeLeft() implement the deletion in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains a single record reference. The sub-functions BTree_MergeSingleNodeRight() and BTree_MergeSingleNodeLeft() completely implement the structural re-balancing of the B-tree on the bottom leaf level.

Figure 2.29: Deletion of the record 15 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains a single record reference - part 1

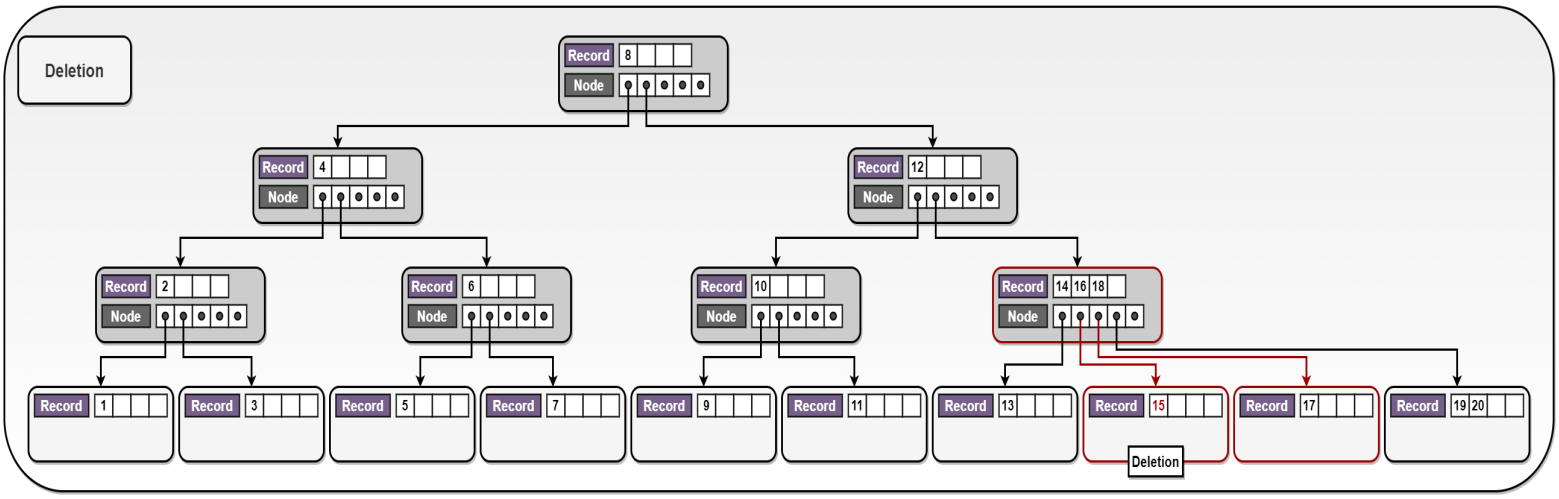
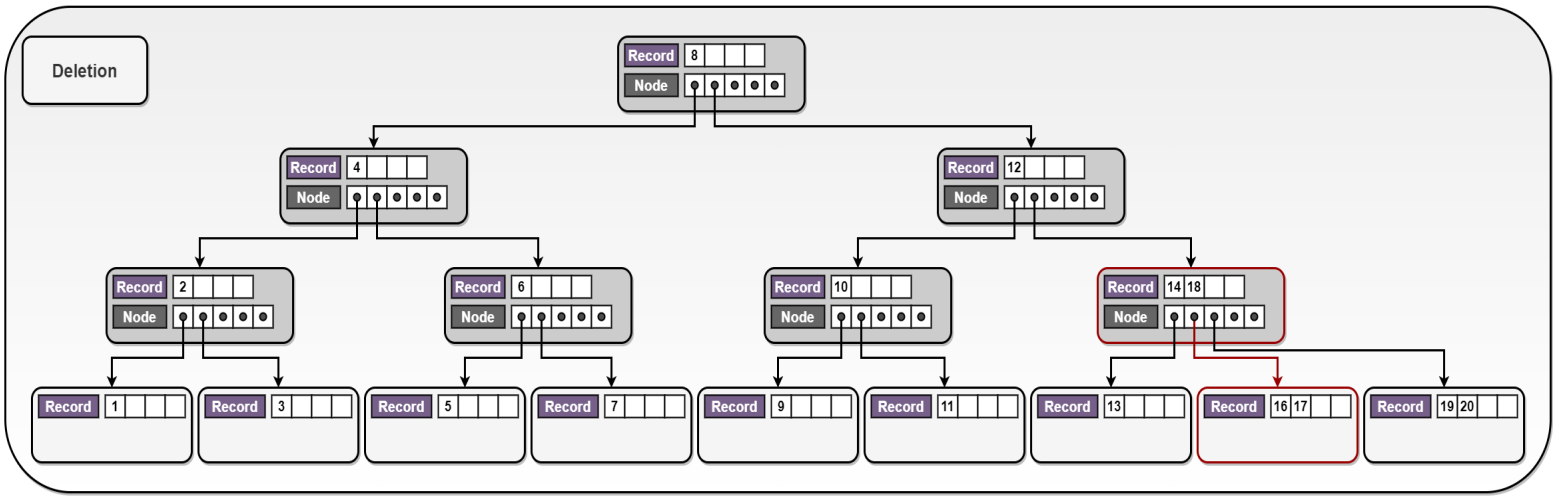


Figure 2.30: Deletion of the record 15 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains a single record reference - part 2



- The sub-functions BTree_SwapLeftNode() and BTree_SwapRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked node (parent node) and the left - right side node contain multiple record references. The sub-functions BTree_SwapLeftNode() and BTree_SwapRightNode() completely implement the structural re-balancing of the B-tree on the bottom leaf level.

Figure 2.31: Deletion of the record 15 in a leaf node that contains a single record reference and the upper level linked node (parent node) and the left - right side node contain multiple record references - part 1

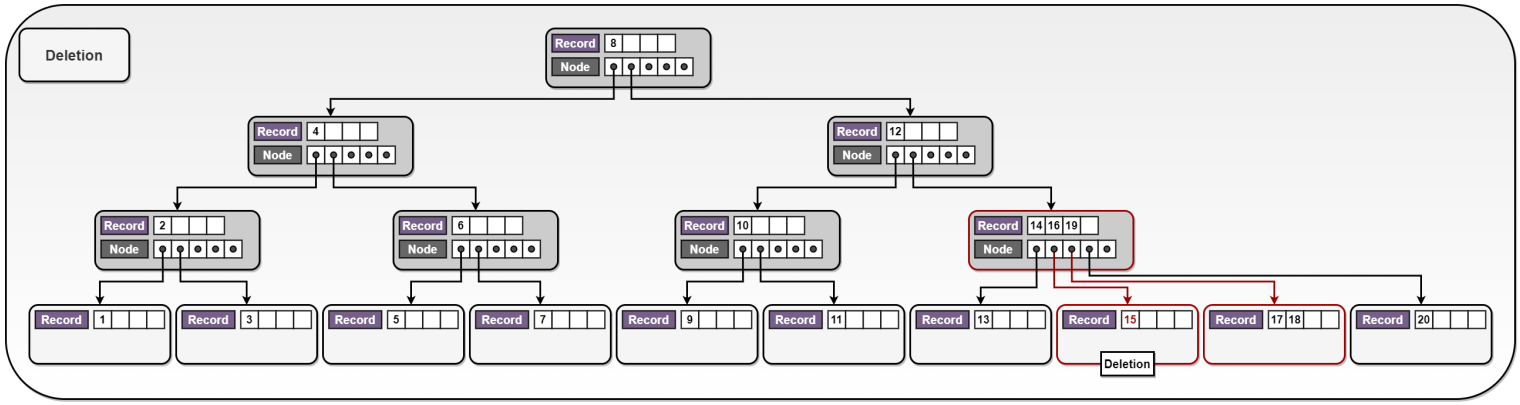
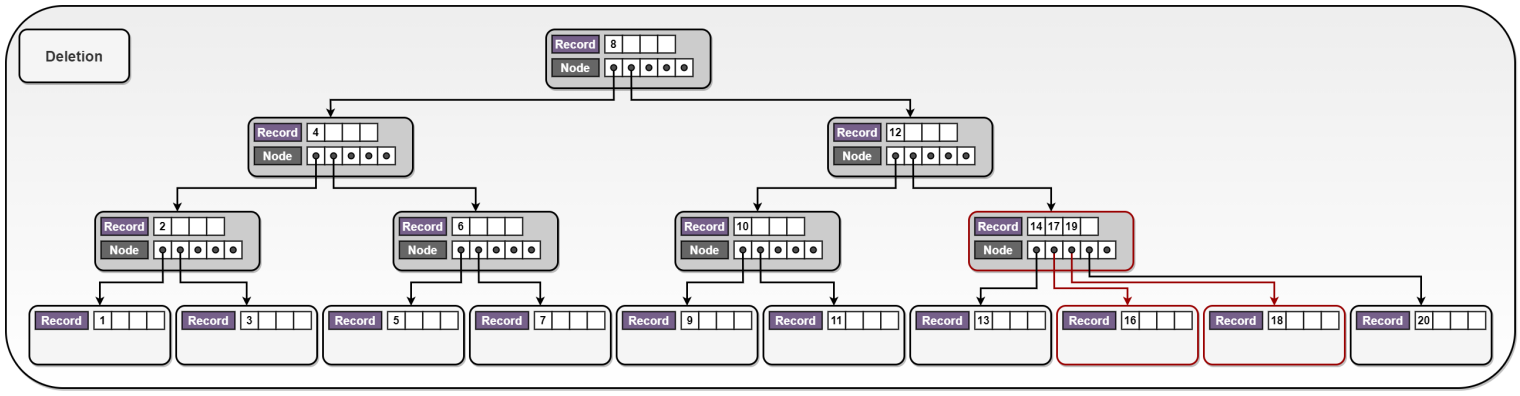


Figure 2.32: Deletion of the record 15 in a leaf node that contains a single record reference and the upper level linked node (parent node) and the left - right side node contain multiple record references - part 2



- The sub-functions BTree_MergeLeftNode() and BTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. The sub-functions BTree_MergeLeftNode() and BTree_MergeRightNode() implement the record reference deletion in the leaf node and the nodes reconstruction on the bottom leaf nodes level in order to structurally re-balance the B-tree. In this case that the structural B-tree balance cannot be restored - recovered on the leaf nodes level a nodes re-balancing and reconstruction recursive process is being implemented to the upper nodes levels. If this problematic nodes structural balancing case is caused up to the root node the sub-functions BTree_MergeLeftNodeRecursive() and the BTree_MergeRightNodeRecursive() are used recursively to fix this problem from the leaf to the root node level. The structural balance finally restored - recovered on the root node level.

Figure 2.33: Deletion of the record 15 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference - part 1

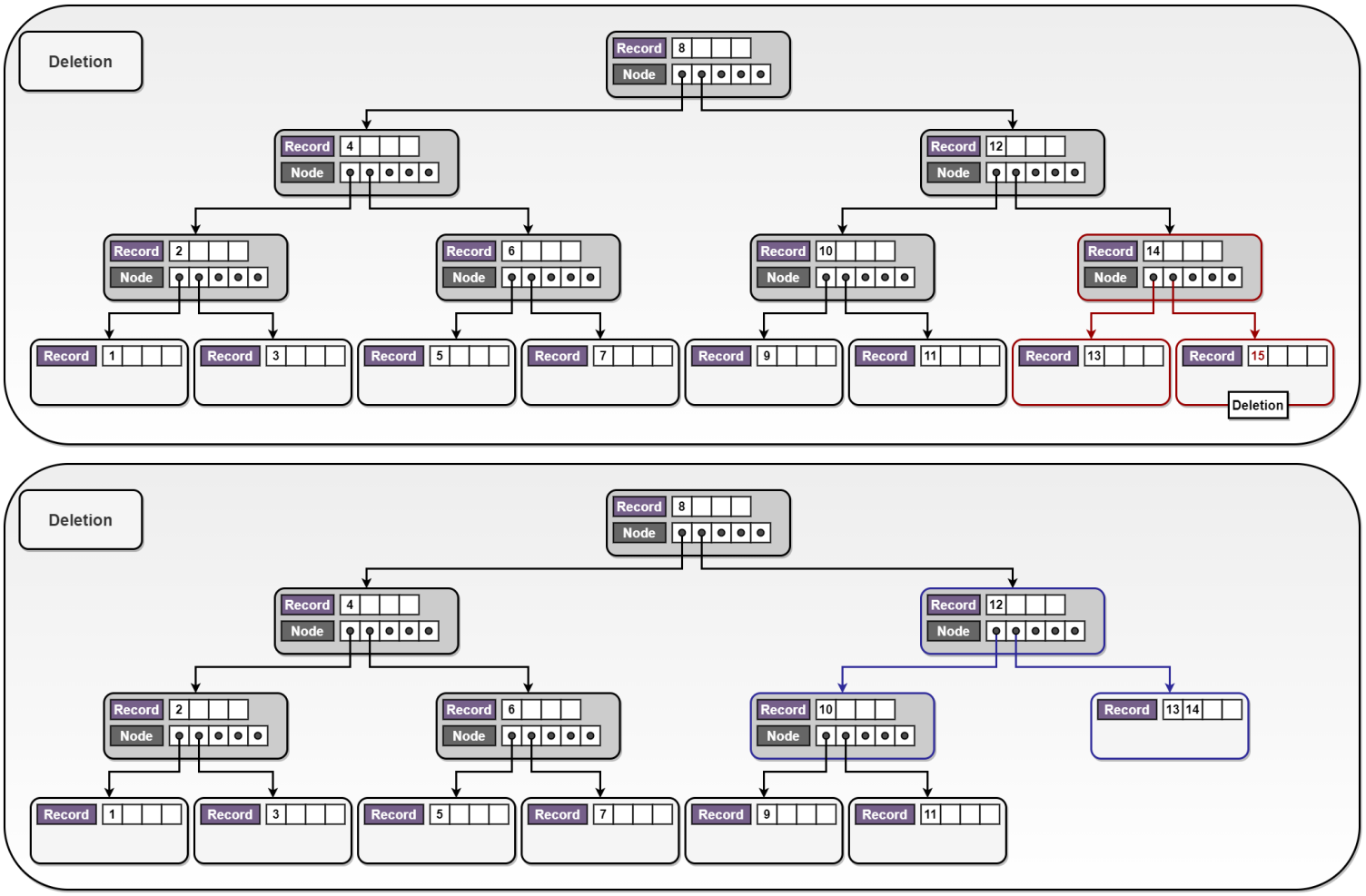
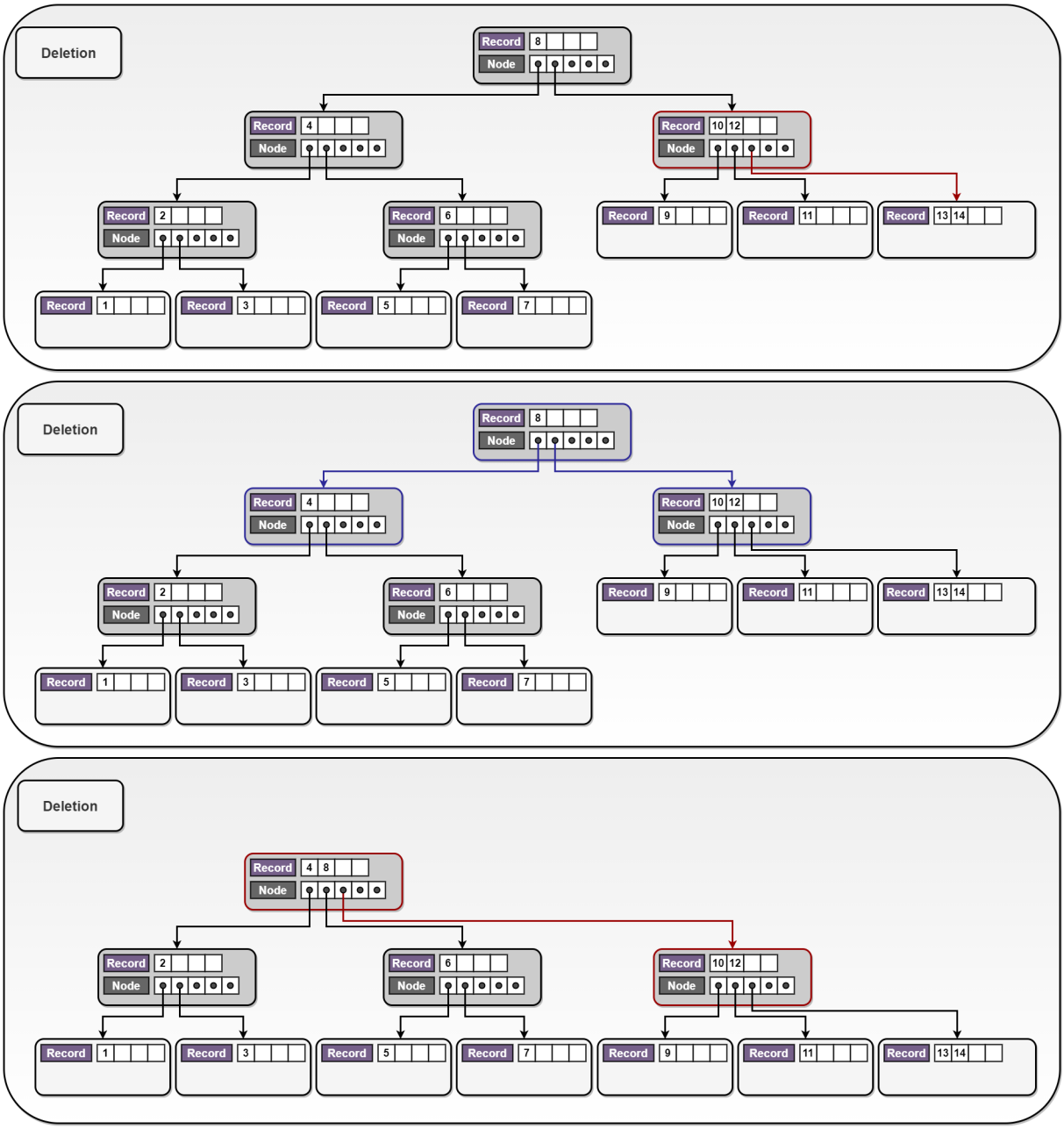
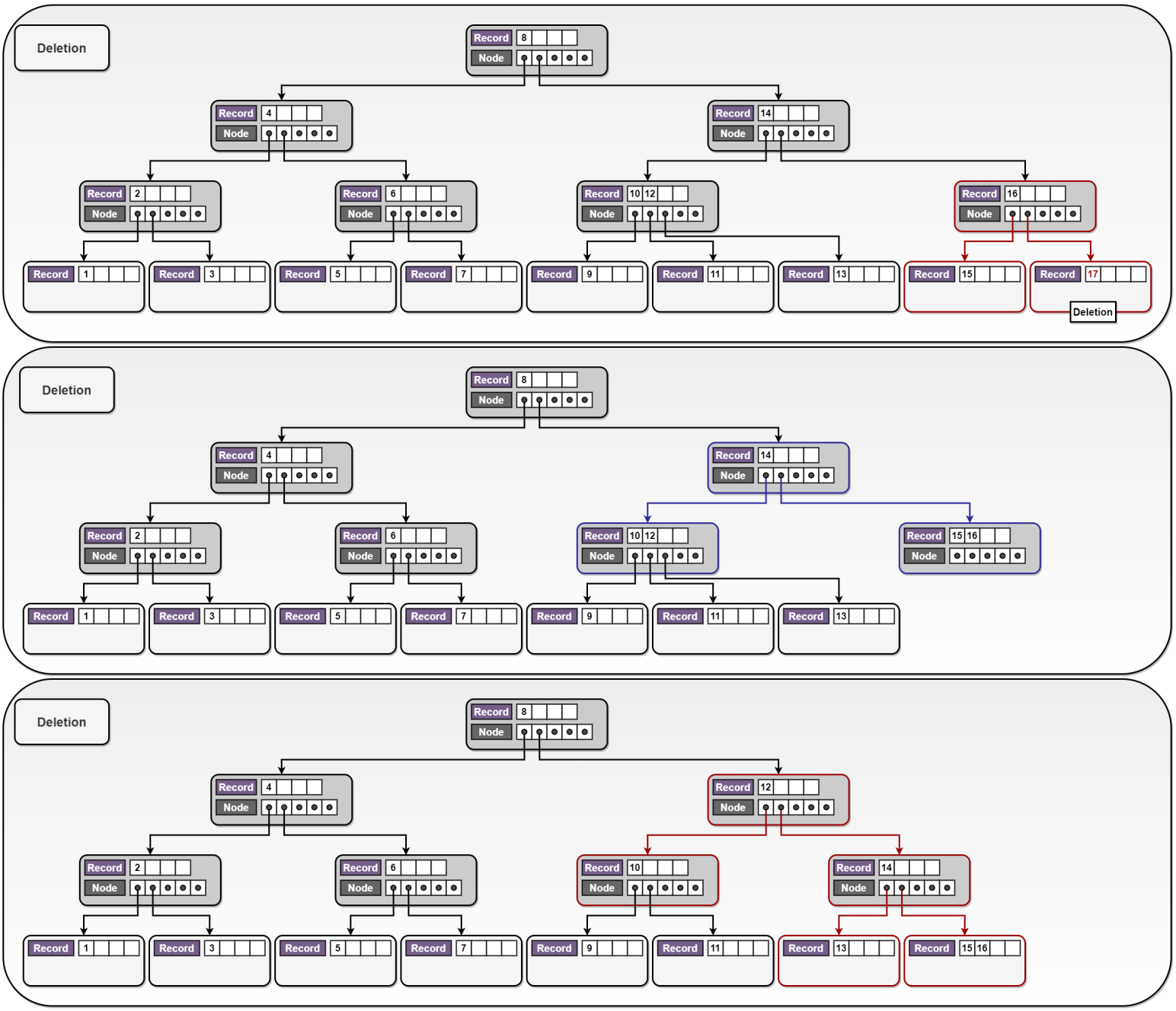


Figure 2.34: Deletion of the record 15 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference - part 2



- The sub-functions BTree_MergeLeftNode() and BTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Furthermore the sub-functions BTree_ReplaceLeftNodeRecursive() and BTree_ReplaceRightNodeRecursive() implement the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has a single record reference and the left - right side node contains multiple record references.

Figure 2.35: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has a single record reference and the left - right side node contains multiple record references



- The sub-functions BTree_MergeLeftNode() and BTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Furthermore the sub-functions BTree_ReplaceSingleLeftNodeRecursive() and BTree_ReplaceSingleRightNodeRecursive() implement the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference.

Figure 2.36: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference - part 1

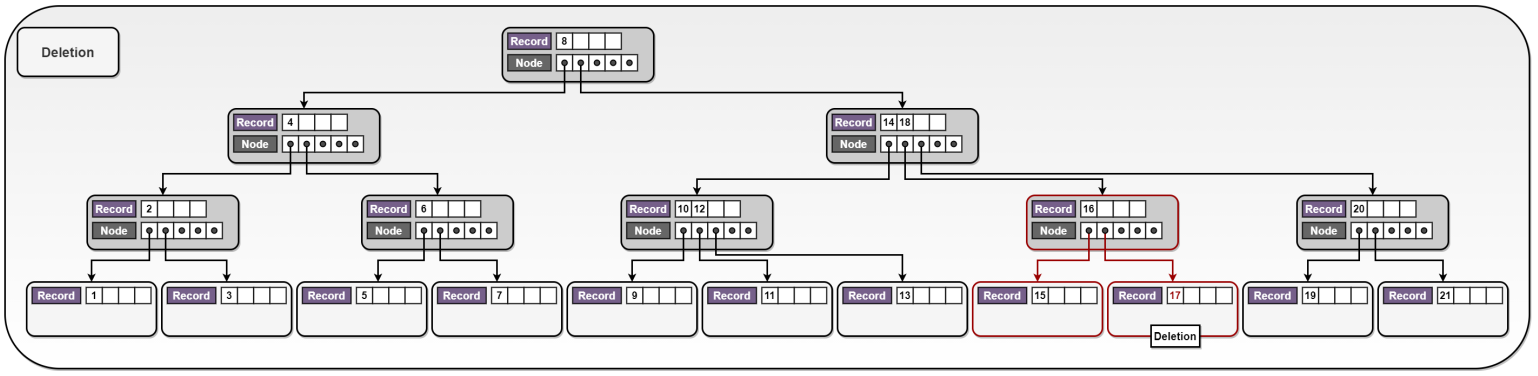


Figure 2.37: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference - part 2

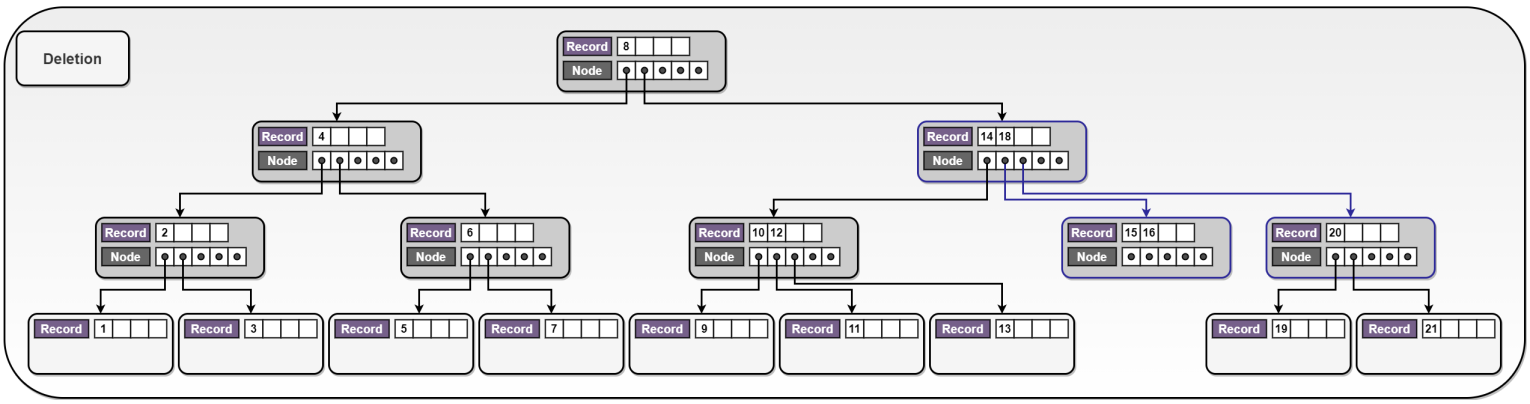
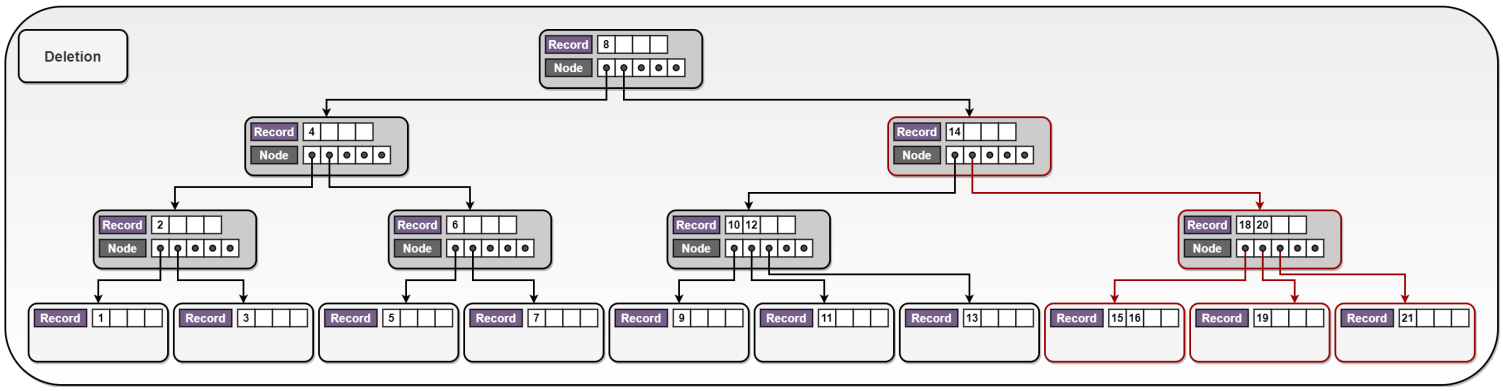


Figure 2.38: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference - part 3



- The sub-functions BTree_MergeLeftNode() and BTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Furthermore the sub-functions BTree_ReplaceMultipleLeftNodeRecursive() and BTree_ReplaceMultipleRightNodeRecursive() implement the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references.

Figure 2.39: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contains a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references - part 1

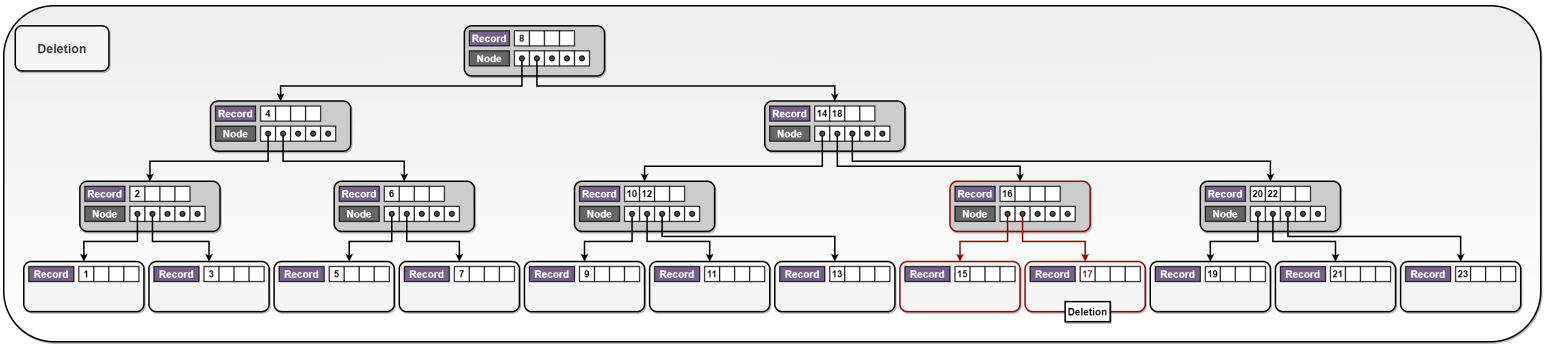


Figure 2.40: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contains a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references - part 2

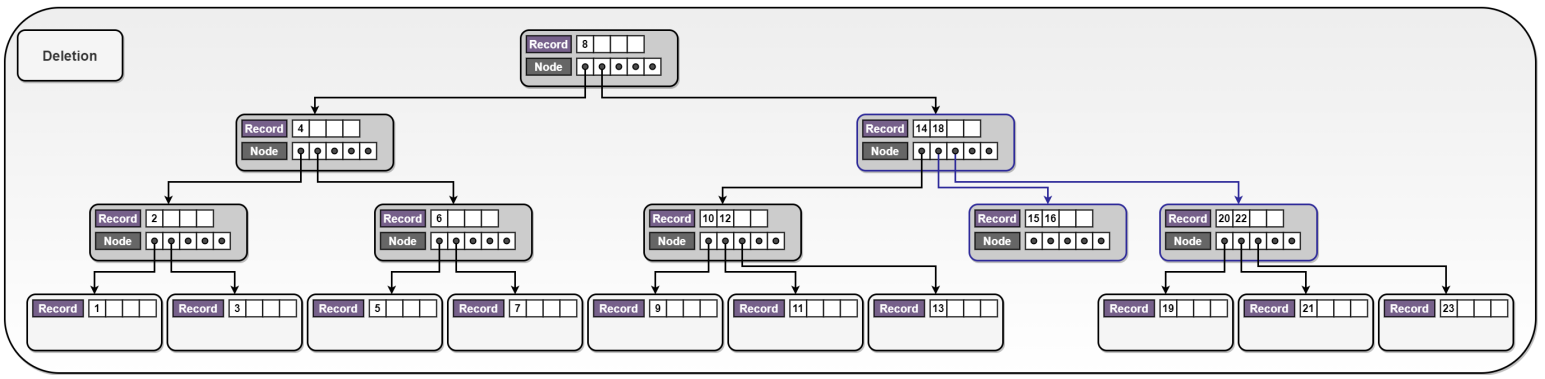
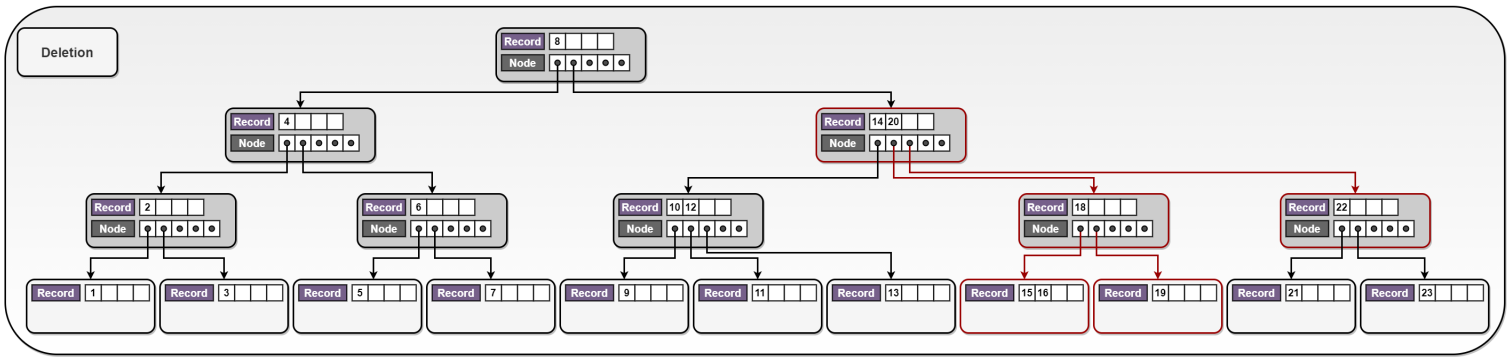


Figure 2.41: Deletion of the record 17 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contains a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references - part 3



The record reference deletion on the upper internal nodes levels is functionally based on the sub-functions in Alg. 11, Alg. 12 and BTree_LeftSubTree_MaxRecord() as already mentioned and analyzed. The BTree_LeftSubTree_MaxRecord() sub-function is used to remove the located record reference from the internal node and transfer it to the maximum record reference of the leftmost leaf node path. The record reference that is already stored in this leaf node is transferred - stored to the first internal node that that the record reference to be deleted was located. Then a leaf nodes level deletion process is implemented by the sub-function in Alg. 11. The B-tree index reconstruction and re-balancing is implemented by the function in Alg. 12. The basic point is that the internal node deletion is transformed to a leaf node deletion process using the BTree_LeftSubTree_MaxRecord().

Figure 2.42: Deletion on an internal node - part 1

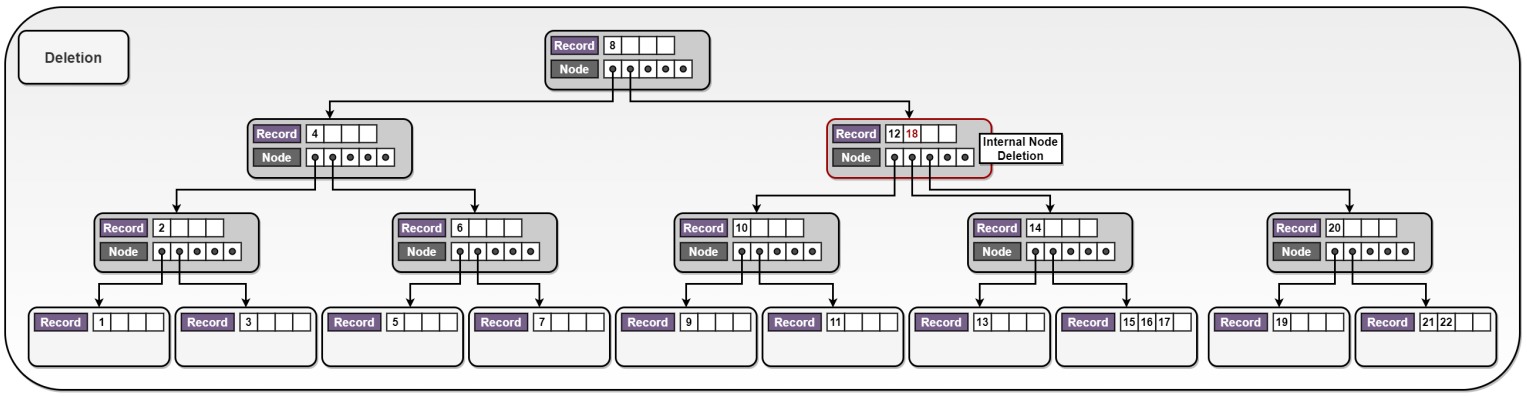


Figure 2.43: Deletion on an internal node - part 2

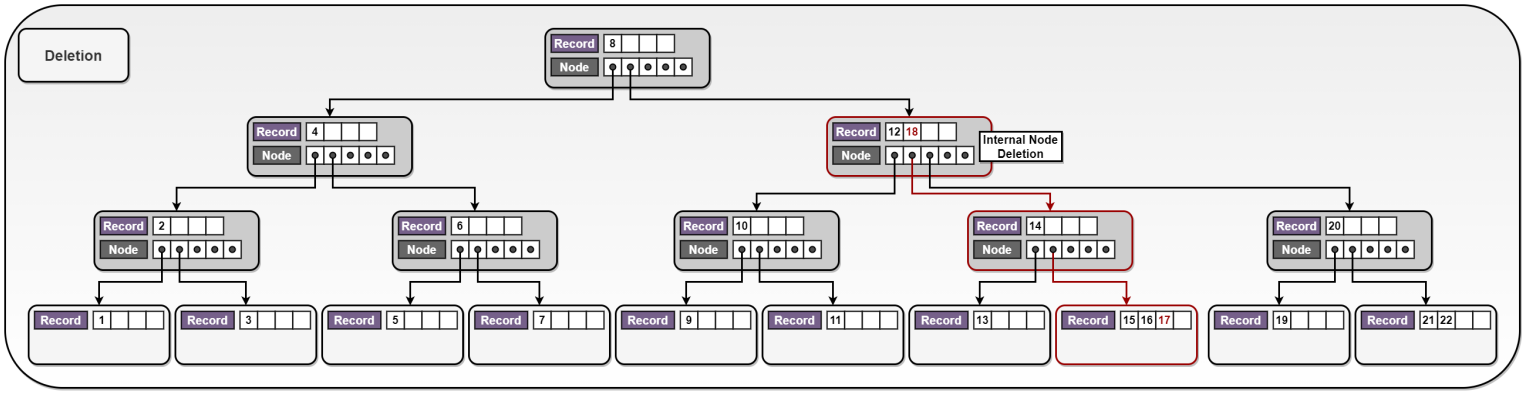


Figure 2.44: Deletion on an internal node - part 3

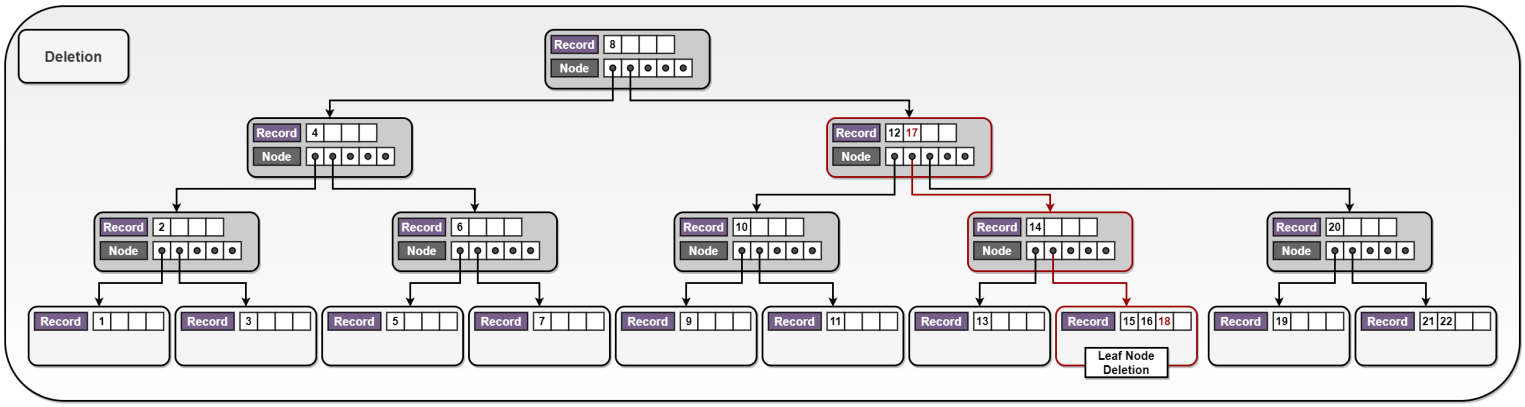
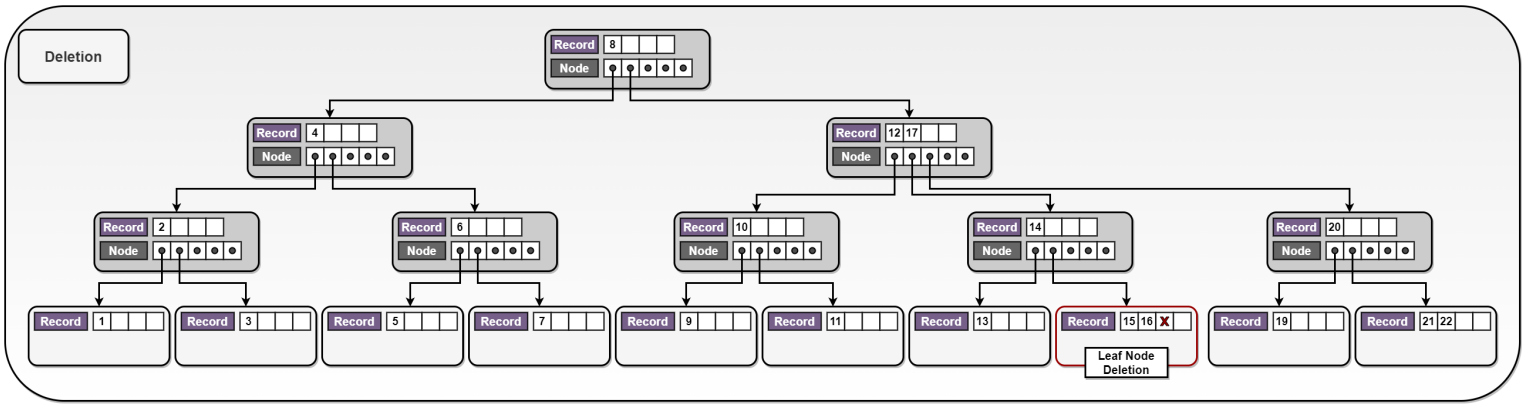


Figure 2.45: Deletion on an internal node - part 4



Algorithm 9: BTreeDeleteData function

Returned item: Deletion process status
BTreeDeleteData(
B-tree structure item,
Primary key field of the record to be deleted,
Item to store the removed record reference
)

if *B-tree data structure is empty* **then**
    Deletion cannot be implemented.
    Return unsuccessful deletion status.
end
if *Root node is a leaf node* **then**
    SearchBTreeNode_Record_ByPrimaryKey()
    if *Record reference to be deleted is located in the current root leaf node* **then**
        if *Root node contains a single record reference* **then**
            Remove the stored record reference from the current node.
            Deletion of the current root node.
            Return successful deletion status.
        end
        BTree_ReplaceRecord()
        Return successful deletion status.
    end
    Return unsuccessful deletion status.
end
BTreeDeleteNode()
if *Record reference to be deleted is not stored in the B-tree structure* **then**
    Deletion cannot be implemented.
    Return unsuccessful deletion status.
end
Return successful deletion status.

Algorithm 10: BTreeDeleteNode function

Returned item: B-tree node item
BTreeDeleteNode(
Next level node item,
Primary key field of the record to be deleted,
Item to store the removed record reference,
Maximum record - node semi-dynamic array structures capacity of each node,
Flag to specify if the balancing process has been activated to an internal node,
Flag to specify if the internal node record deletion procedure has been activated,
Flag to specify if the record to be deleted exists in the structure
)

SearchBTreeNode_Record_ByPrimaryKey()

if *The record reference to be deleted is located at the current internal node* **then**

BTree_LeftSubTree_MaxRecord()

Replacement - swap of the record reference to be deleted with the maximum leaf node record reference of the left sub-tree path.

Activation of the internal node deletion process - deletion of the transferred record reference (to be deleted) in this leaf node.

end

if *Next level node is an internal node* **then**

BTreeDeleteNode()

if *The structural balance recovery - restoration cannot be implemented on the previous level. The upper level node re-balancing and reconstruction process has been activated* **then**

BTreeDelete_NonLeafNode()

end

Return Current node item.

end

BTreeDelete_LeafNode()

Return Current node item.

Algorithm 11: BTreeDelete_LeafNode function

Returned item: Void item
BTreeDelete_LeafNode(
Upper level - parent node item,
Primary key field of the record to be deleted,
Item to store the removed record reference,
Position of the stored record reference to be deleted in the next level leaf node,
Maximum record - node semi-dynamic array structures capacity of each node,
Flags to specify the internal node balancing and record deletion processes and the record to be deleted existence in the structure
)

SearchBTreeNode_Record_ByPrimaryKey()

if *The record reference to be deleted is located at the current leaf node* **then**

  if *The current leaf node contains multiple record references* **then**

    BTree_ReplaceRecord()
    Return void item.

  end

  if *The current upper level - parent node and each of the leaf and left - right side nodes contain a single record reference* **then**

    BTree_MergeRightNode() or **BTree_MergeLeftNode()**
    Activation of the structural nodes re-balancing process to the upper level.
    Return void item.

  end

  if *The current upper level - parent node and the leaf node contain a single record reference and the left - right side node contains multiple record references* **then**

    BTree_RebalanceRightNode() or **BTree_RebalanceLeftNode()**
    Return void item.

  end

  if *The current upper level - parent node contains multiple record references, the leaf node contains a single record reference and the left - right side node contains multiple record references* **then**

    BTree_SwapRightNode() or **BTree_SwapLeftNode()**
    Return void item.

  end

  if *The current upper level - parent node contains multiple record references, the leaf node contains a single record reference and the left - right side node contains a single record reference* **then**

    BTree_MergeSingleNodeRight() or **BTree_MergeSingleNodeLeft()**
    Return void item.

  end

end

Return void item.

Algorithm 12: BTreeDelete_NonLeafNode function

Returned item: Void item
BTreeDelete_NonLeafNode(
Current upper level node item,
Position of the next level linked node in the node references semi-dynamic array structure that the nodes reconstruction - re-balancing process was implemented,
Maximum record - node semi-dynamic array structures capacity of each node,
Flag to specify the internal node balancing process
)

if *The current upper level - parent node contains a single record reference and the next level left - right side node of the previous reconstructed node contains a single record reference* **then**

BTree_MergeLeftNodeRecursive() or **BTree_MergeRightNodeRecursive()**
Activation of the structural nodes re-balancing process to the upper level.
Return void item.

end

if *The current upper level - parent node contains a single record reference and the next level left - right side node of the previous reconstructed node contains multiple record references* **then**

BTree_ReplaceLeftNodeRecursive() or **BTree_ReplaceRightNodeRecursive()**
Return void item.

end

if *The current upper level - parent node contains multiple record references and the next level left - right side node of the previous reconstructed node contains a single record reference* **then**

BTree_ReplaceSingleLeftNodeRecursive() or
BTree_ReplaceSingleRightNodeRecursive()
Return void item.

end

if *The current upper level - parent node contains multiple record references and the next level left - right side node of the previous reconstructed node contains multiple record references* **then**

BTree_ReplaceMultipleLeftNodeRecursive() or
BTree_ReplaceMultipleRightNodeRecursive()
Return void item.

end

The total average algorithmic operations - steps of the deletion function can be approximately calculated based on the average B-tree structure height shown in Equation 2.7. Consequently, the theoretical average time complexity of the deletion function in Alg. 9– 12 can be approximately calculated by the relation 2.24:

$$O(\ log_{(2ku_k+1)}(n)\ ) \tag{2.24}$$

Chapter 3

Theoretical analysis and implementation of the B^+-tree data structure

3.1 B^+-tree index structural properties and characteristics

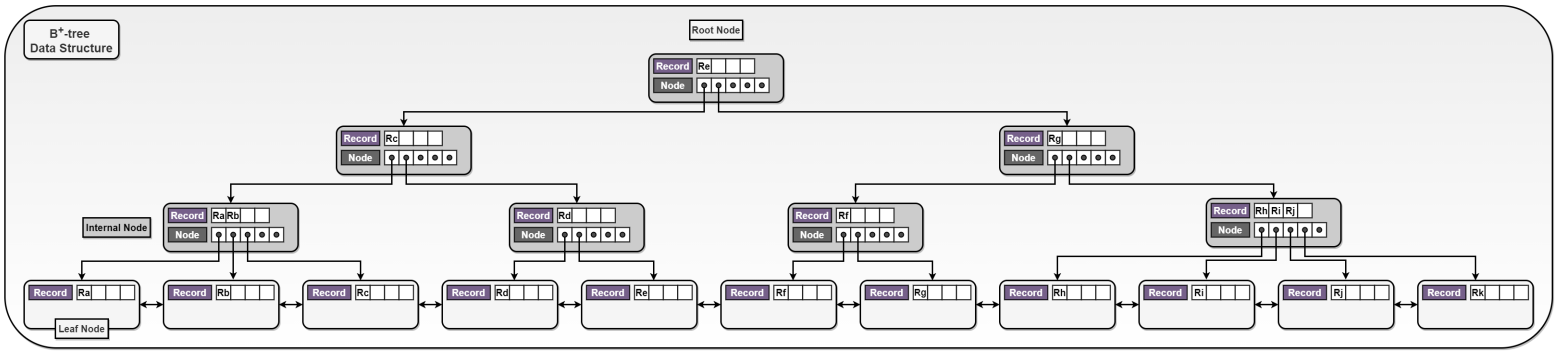
3.1.1 B^+-tree index structure implementation and development base

Similar to the basis of the general fundamental B-tree data structure, the development and implementation of the B^+-tree data structure is based on the structural and functional analysis and definition of the works in [6] [2] [3] [40] [20] [41] [42] [17] [18] [19] [43] [13] [39] [44] [45] [46] [47]. This study is based on the aforementioned works and applies modifications at some basic structural and functional B^+-tree structure levels. This B^+-tree index structure implementation composes a structural and functional approximation of these studies utilizing them as a theoretical base in order to develop, construct and implement an efficient and fast B^+-tree index data structure.

Furthermore, the structural and functional characteristics and properties of the B^+-tree are based on the B-tree data structure as the B^+-tree constitutes an advanced and specialized structural and functional tree variant of the B-tree. Consequently, the B^+-tree has been developed and constructed in the context of the previous B-tree index implementation. The B^+-tree data structure will be analyzed, evaluated and compared with the B-tree in terms of efficiency and time performance.

3.1.2 B⁺-tree index structure

Figure 3.1: B⁺-tree data structure



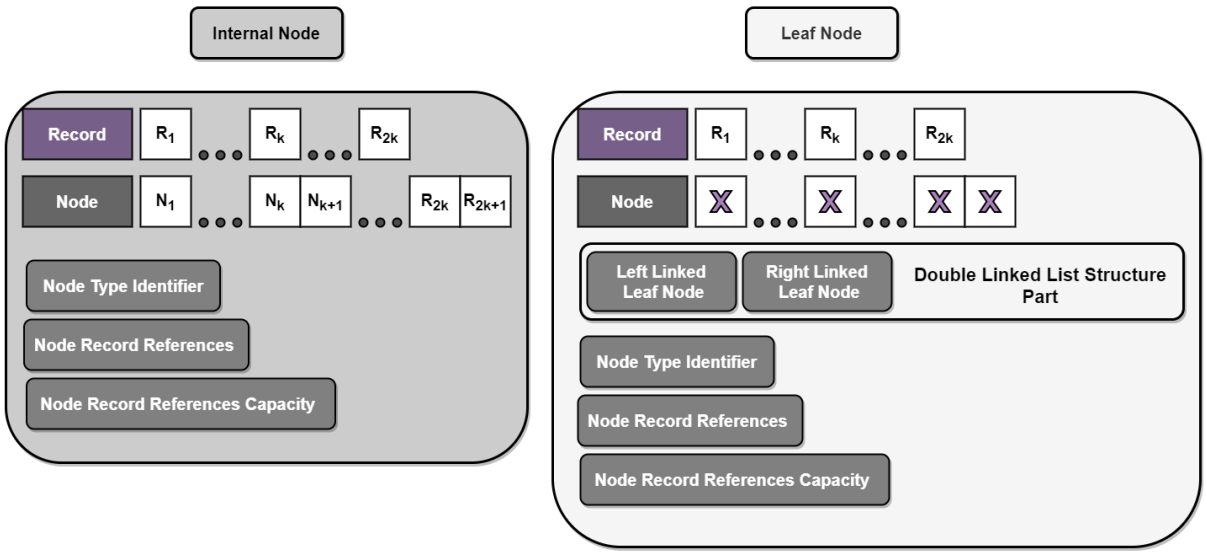
The B⁺-tree index data structure consists of nodes which are basic and initial structural parts - blocks of the tree. The structural parts, characteristics and properties of each individual tree node are based on the node type. There are two separate node types, the intermediate - internal nodes and the leaf nodes.

The B⁺-tree structure, shown in Fig. 3.1, is formed and structurally organized in levels that are composed of node sets. The levels set of the structure that are at a height - depth higher than the last bottom level consists of internal B⁺-tree nodes. The last level of the structure consists exclusively and completely of leaf nodes which are all located at the same last tree level and the leaf nodes set is linked in the form of a Double Linked List structure. Each path from the root node to any leaf node has the same length - height. The node that is at the first level of the tree is defined as the root node and is a potential internal node in case that the first level of the tree is not identical to the last tree level, in which case the root node is a leaf node. The basic property and characteristic of each node and level is the height or depth which is defined as the path - set of nodes or levels from the root node (first - top level) to the bottom last leaf nodes level (B⁺-tree node height). Approaching the height property from a different perspective, we can define it as the total number of transitions to be made between connected nodes of different levels of the B⁺-tree in order to move from the root node to some leaf node of the last structure level (B⁺-tree height). Another B⁺-tree characteristic is the branching factor which is basically the maximum number of the node references (next level linked child nodes) that each node can contain - store. This structural organization and formation of the B⁺-tree nodes defines the property of the structural tree balance. Each leaf node is contained in the last bottom leaf level of the tree structure. The structural balance is implemented through a set of

algorithmic techniques of nodes and nodes semi-dynamic array structures reconstruction, rearrangement and reorganization which are incorporated in all of the basic insertion, deletion and search - selection functions. Furthermore an algorithmic subset of the basic B$^+$-tree structure operations is functionally identical to that of the B-tree. Consequently, the B$^+$-tree is structurally self-balanced as it transforms and modifies its structure depending on its functionality.

3.1.3 B$^+$-tree index node structure

Figure 3.2: B$^+$-tree node structure



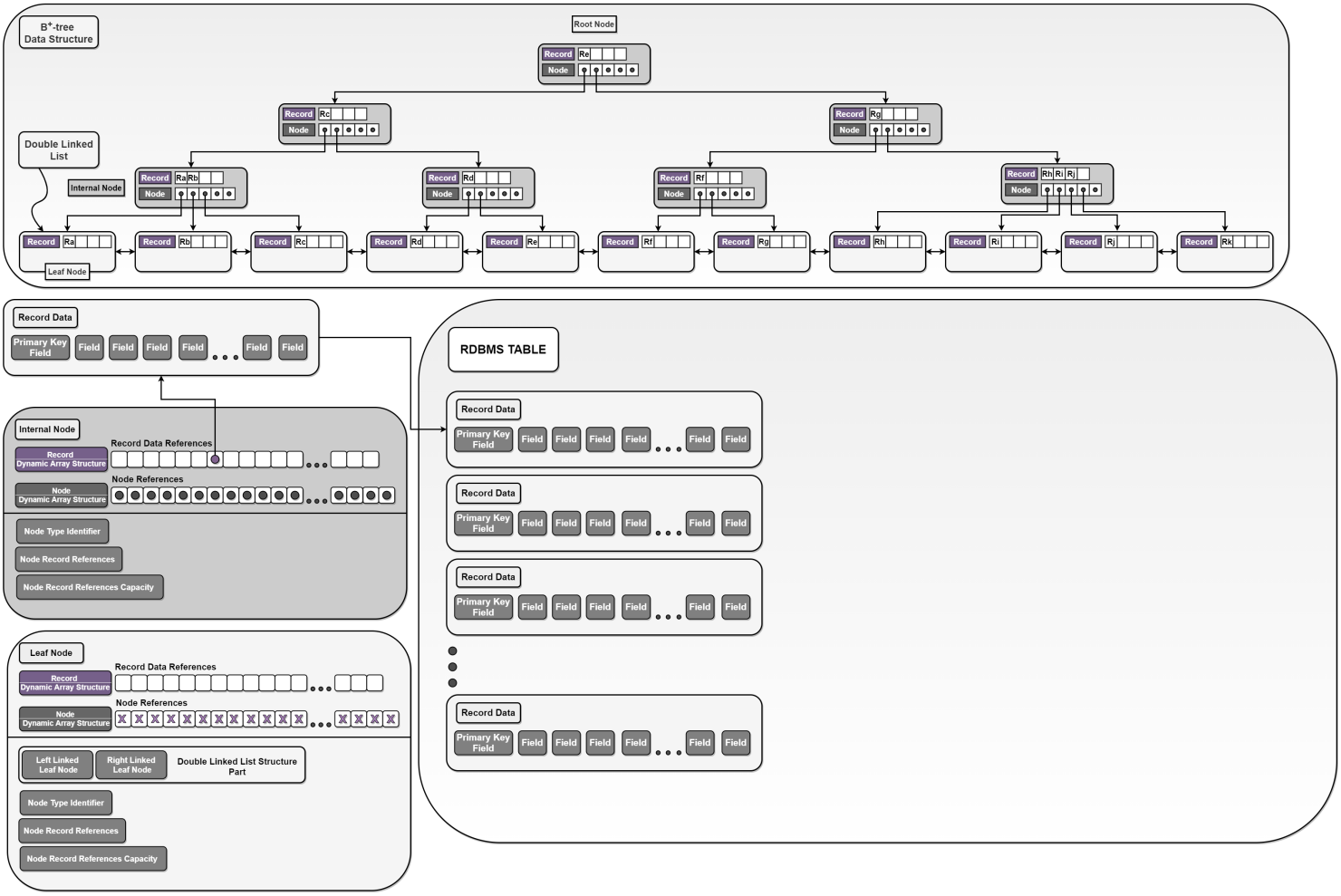
The B$^+$-tree node structure shown in Fig. 3.2 consists of the following structural parts – blocks and is governed by the listed characteristics and properties:

- B$^+$-tree is a multi-valued tree data structure since each node can store multiple data items - records. The k parameter constitutes the multiple different sets of records that can be stored in a B$^+$-tree node.
- In this implementation and theoretically each internal node can contain - store multiple ordered record and node references.
- Each leaf node can contain exclusively multiple ordered record references.
- Each internal node can store at most $2k$ record and $2k + 1$ node references, where $k \geq 1$.

- Each leaf node can store at most $2k$ record references.
- Each node can theoretically store between 1 and $2k$ record references. Furthermore, each internal node can store between 1 and $2k + 1$ node references.
- Each node contains between k and $2k$ record references except the root node, which contains between 1 and $2k$ record references.
- In this implementation, each internal node is composed of two semi-dynamic array data structures that store record and node references. Each node semi-dynamic array can modify (increase and decrease) its capacity in order to reduce the memory allocation and usage.
- Each internal node contains three variables that specify the node type, the number of stored array structure references and the maximum capacity of the reference array structure.
- In this implementation, each internal node (except the root node) can store approximately between k and $2k$ record and between $k + 1$ and $2k + 1$ node references.
- In this implementation, each leaf node consists of two semi-dynamic array data structures that store record and node references and modify (increase and decrease) its capacity in order to reduce the memory allocation and usage. The array of node references is completely empty and has not available allocated memory.
- Each leaf node contains three variables that specify the node type, the number of stored array structure references and the maximum capacity of the reference array structure.
- In this implementation, each leaf node contains references of the left and right linked leaf nodes of this node in the form of a Double Linked List structure.
- In this implementation, each leaf node (except the root leaf node) can approximately store between k and $2k$ record references.
- The default record array capacity of a new internal node is approximately at most k and the node array capacity is $k + 1$.
- The default record array capacity of a new leaf node is approximately at most k and the node array capacity is 0.
- The record array capacity is approximately at most k and the node array capacity is at most $k + 1$ depending on the size (1 to k) of the stored record references.

- The record array capacity is approximately at most $2k$ and the node array capacity is at most $2k + 1$ depending on the size ($k + 1$ to $2k$) of the stored record references.

Figure 3.3: B⁺-tree node structure and system architecture



In this implementation, each record block consists of a data field set. This set is composed of fields - attributes of different data types. Each record has a unique identifier, the primary key field that separates them from the other records in the RDBMS table to which each record is stored and belongs. The B⁺-tree is a clustered - primary index structure as it stores a record references set based on the records primary key fields. The available valid primary key field data types of the developed B⁺-tree structure can be either integer or string.

The structure, design and architecture of the B⁺-tree index system is represented in Fig. 3.3. The implemented B⁺-tree index structure system is based on a main memory (RAM) system and each record and node data reference is a link to a set of memory components - blocks that store the data.

3.1.4 B⁺-tree nodes number and height approximation

According to the related cited works the parameter h ($h \geq 0$) is defined as the theoretical B⁺-tree structure height (total B⁺-tree nodes levels) and can be approximately calculated by the relations:

$$h_{min} = log_{(2k+1)}(\frac{n}{k}) \tag{3.1}$$

$$h_{max} = 1 + log_{(k+1)}(\frac{n}{2k}) \tag{3.2}$$

$$log_{(2k+1)}(\frac{n}{k}) \leq h \leq 1 + log_{(k+1)}(\frac{n}{2k}) \tag{3.3}$$

Based on relation 3.3, the total theoretical average B⁺-tree structure height can be approximately calculated by the relation:

$$h_{avg} \cong \frac{(h_{min} + h_{max})}{2} \tag{3.4}$$

Based on the theoretical property that each B⁺-tree leaf node (except the root leaf node) can store and contain approximately at least k and at most $2k$ record references, the total theoretical average bottom level leaf nodes set number of the B⁺-tree structure d_{ln} can be approximately approached via the relation:

$$\begin{aligned} d_{ln} &\cong \frac{\sum_{i=k}^{2k} \left(\frac{n}{i} \right)}{\left(\frac{k}{s} + 1 \right)}, \quad 2k < n \\ d_{ln} &= 1, \quad 2k \geq n \end{aligned} \tag{3.5}$$

The parameter s ($s > 0$) is the increment step - rate of the i parameter.

Each leaf node of the Double Linked List structure except the last list tail leaf node stores - contains exactly one record reference a copy of which is also stored in an internal B⁺-tree node. The record references set that is stored in the internal nodes is also stored in the leaf nodes set of the B⁺-tree and it is equal to $d_{ln} - 1$ (Equation 3.5). The B⁺-tree structure can be theoretically transformed to a B-tree if the bottom last leaf nodes level of the B⁺-tree is removed - extracted. That created B-tree structure contains - stores $d_{ln} - 1$ record references and has theoretical average height that can be approximately calculated by the relations 3.3 and 3.4. Based on these data and utilizing the B-tree theory the total and internal B⁺-tree structure nodes can be calculated via this new created B-tree. In this way the B⁺-tree average height, internal - leaf nodes and record references storage distribution in internal and leaf nodes can be calculated.

3.2 B⁺-tree index structure basic functional levels

3.2.1 B⁺-tree index structure functions

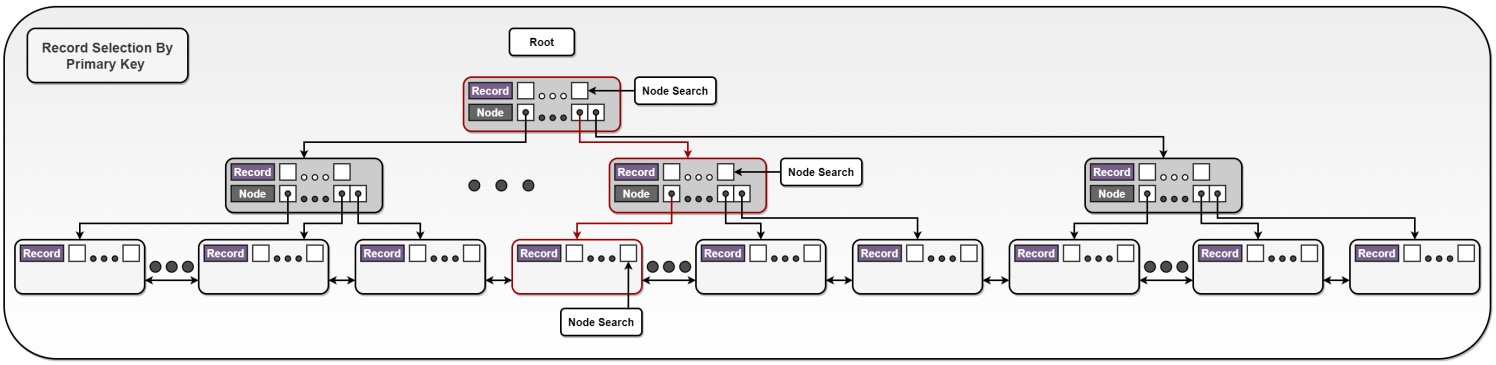
The structural and functional parts of the B⁺-tree index were implemented as a base of the B-tree index structure in the development environment of the B-tree structure using the same tools. The basic implemented B⁺-tree algorithmic functions that are analyzed in the context of this work, are:

- Records selection by record primary key field.
- Records selection by multiple record fields.
- Records insertion and deletion based on the record primary key field.

The B⁺-tree visualization tool in [48] can be used in order to theoretically simulate the insertion, deletion and selection - search B⁺-tree functions with sufficient accuracy.

3.2.2 Records selection by primary key fields

Figure 3.4: Records selection by record primary key field



The overall selection function BplusTreeFastSearchData_ByPrimaryKey() shown in Fig. 3.4 that implements the location and selection of a record in some B⁺-tree node by the primary key field of the record is composed of a sub-functions algorithmic set. This sub-functions set that implements the node and tree scope selection - search is the same as that of the B-tree index structure.

The node selection sub-function SearchBplusTreeNode_Record_ByPrimaryKey() implements the location and selection of a record in the B⁺-tree index structure node or the location of the next level linked node that the record could be stored based on the primary key field of the record. The tree scope selection BplusTreeFastSearchData_ByPrimaryKey() and BplusTreeFastSearch_Tool() utilizes the node search sub-function in order to scan the B⁺-tree structure nodes paths from the root node to the leaf nodes level in order to locate

and select the record with the specific primary key field identifier. Furthermore as the B-tree, the B$^+$-tree structure SearchBplusTreeNode_Record_ByPrimaryKey() function use the binary and interpolation search algorithms in order to implement the record location and selection in the node record references semi-dynamic array structure.

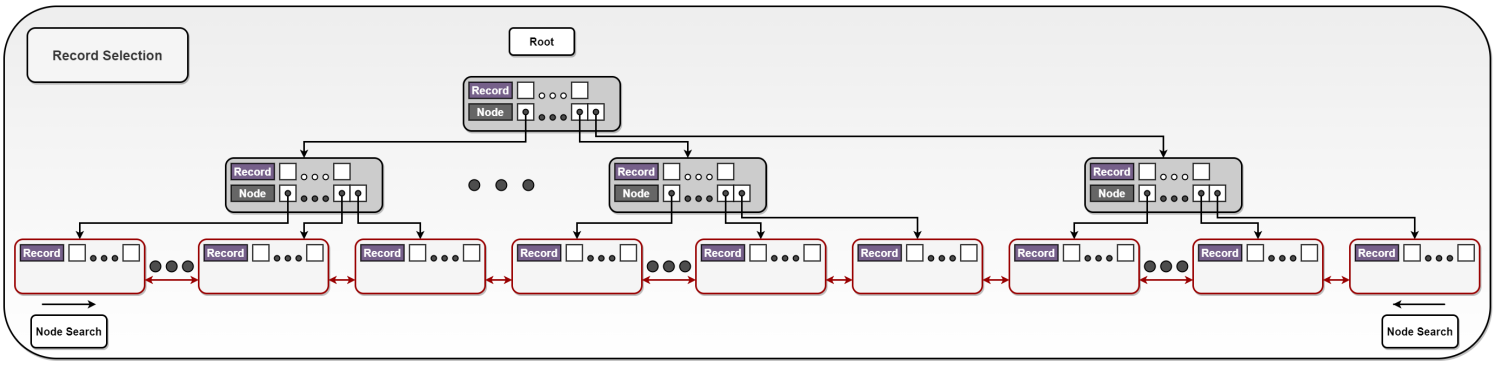
The theoretical average time complexity of the record selection function based on a primary key field can be calculated by the relations 3.6 and 3.7:

$$O(\ log_{(2)}(2k) \cdot (1 + log_{(2ku_k+1)}(d_{ln} - 1))\ ), \quad using\, binary\, search \tag{3.6}$$

$$O(\ log_{(2)}(log_{(2)}(2k)) \cdot (1 + log_{(2ku_k+1)}(d_{ln} - 1))\ ), \quad using\, interpolation\, search \tag{3.7}$$

3.2.3 Records selection by multiple fields

Figure 3.5: Records selection by multiple record fields



The overall selection function shown in Fig. 3.5 of multiple record references composes a complete and full scan function of all the B$^+$-tree leaf nodes stored record references in order to locate and collect - select these records based on the selection conditions set to gather and store them in a data structure. This selection process is implemented via the functions BplusTreeSelectRecordData_ASC() and BplusTreeSelectRecordData_DESC() by applying a complete iterative scan and selection of the records set that is stored

and contained in the bottom leaf nodes level set in the form of a Double Linked List structure. In order to store the selected records set an auxiliary specially designed Double Linked List data structure is used. This list stores the selected record references in both ascending or descending order based on the primary key field of each record. Furthermore to implement the bottom leaf nodes level scan the BplusTreeLocateLeftLeafNode() and BplusTreeLocateRightLeafNode() sub-functions are used for the transition from the B$^+$-tree root node to the head or tail node of the list structure that the iterative selection process is applied.

The selection process can also be implemented as a recursive scan - selection of all the B$^+$-tree nodes and nodes semi-dynamic array structures stored records (B-tree selection). Nevertheless this recursive selection is not generally efficient in terms of time and memory consumption in compare with the iterative scan of the leaf level Double Linked List nodes as must be scanned more nodes and records and there is an additional time and memory consummation of the recursion stack.

In both selection methods the theoretical average time complexity of the selection function can be approached as $O(n)$.

3.2.4 Records insertion based on primary key fields

The insertion function in Alg. 13 consists of multiple recursively linked sub-functions listed in Alg. 14 and 15. Each individual sub-function implements a discrete functional part of the overall insertion process. The insertion sub-functions use the split node algorithm as a basic B$^+$-tree index structure balancing technique.

The record insertion - storage function implementation is based on the node split algorithmic method. This algorithm is used to store the record in a B$^+$-tree leaf node and re-balance - reconstruct the tree nodes that the insertion process affected in order to recover the structural balance. The node split algorithm is applied on the internal and leaf nodes sets of the tree in order to implement the reconstruction, rearrangement and re-balancing of the B$^+$-tree index on a macroscopic level. There are two different node split algorithms which are applied on the tree nodes depending on the node type (internal or leaf node). The internal node-block splitting and the node semi-dynamic array structures

reconstruction and rearrangement of the stored record - node references sets is completely identical to the B-tree node split algorithm. The leaf node-block splitting algorithm (shown in Fig. 3.8 – 3.11) is based on the property - technique of storing and maintaining all the records at the last - bottom leaf nodes level of the B^+-tree structure and differs from the basic internal node B^+-tree node splitting technique. The main difference of the leaf node split related to the internal node split is that the middle stored record of the record semi-dynamic array structure of the splitting node is kept stored in the left node part (sub-node) after the node split and the right half records set transferred from the initial node to the new right node part (sub-node).

The sub-function in Alg. 14 implements the root node split process and the reconstruction and rearrangement of the record and node references semi-dynamic array structures. The node (root node) splits (shown in Fig. 3.16and 3.15) into two distinct structural parts - sub-nodes and the right half stored record and node references of the node semi-dynamic array structures transferred to the new node. The middle record reference that is stored in the initial node (root node) array structure transferred and stored in the record reference semi-dynamic array structure of the upper level linked node (parent node). In this case that the splitting node is the root a new root node is created in order to be transferred the middle record reference. The insertion sub-functions in Alg. 14 and 15 use the split node algorithm to reconstruct, rearrange and organize the B^+-tree nodes set structure in order to restore the structural balance.

The sub-function in Alg. 15 implements the insertion and storage of a record reference in a leaf node. The record reference is stored to the leaf node if the leaf node record references semi-dynamic array structure has available allocated memory - capacity to store the inserted record (shown in Fig. 3.6 – 3.12). If the record semi-dynamic array structure of the leaf node has not available memory - capacity to store the record reference the leaf node splits and a reconstruction and rearrangement process is caused (shown in Fig. 3.8 – Fig. 3.14). This process is repeated recursively up to the root node (Fig. 3.15) in order to structurally re-balance and stabilize the B^+-tree structure.

Fig. 3.6 and 3.7 analyze and describe the record reference insertion in a leaf node record semi-dynamic array structure with available allocated storage memory - capacity:

Figure 3.6: Record reference insertion process in a leaf node with available record semi-dynamic array structure capacity - part 1

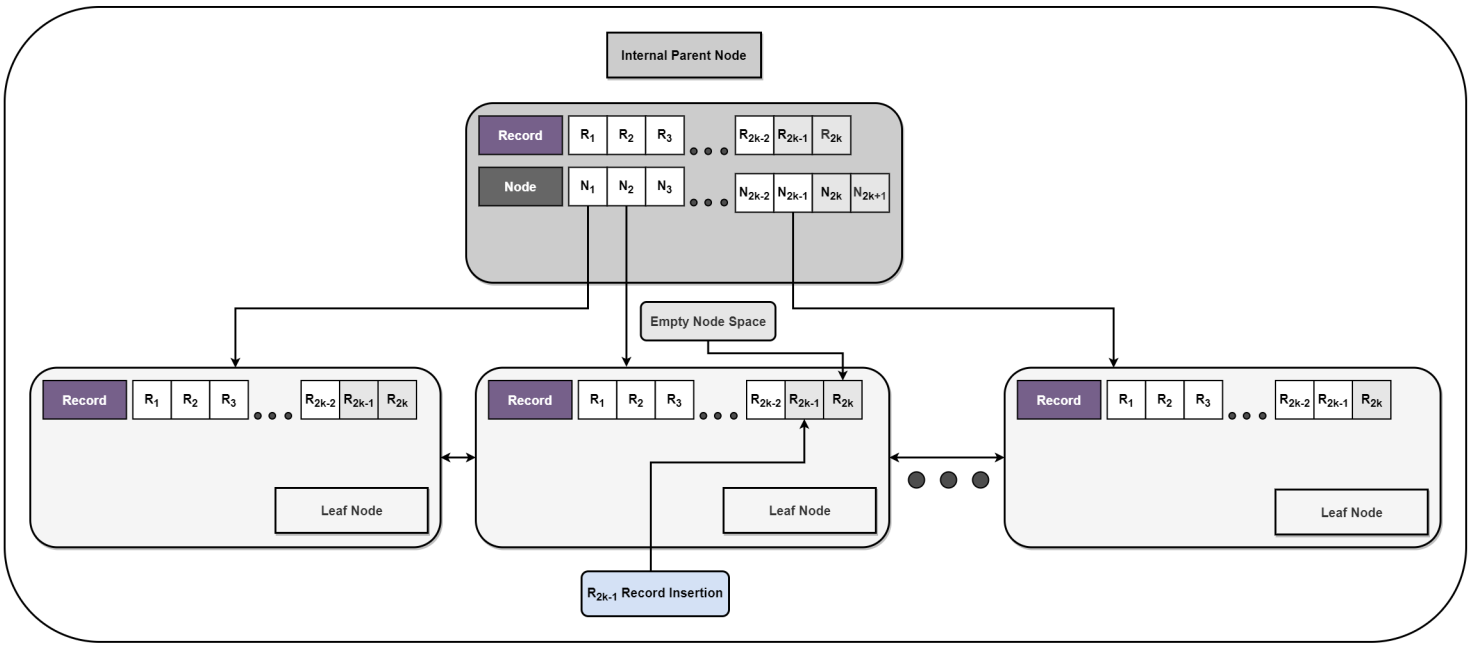


Figure 3.7: Record reference insertion process in a leaf node with available record semi-dynamic array structure capacity - part 2

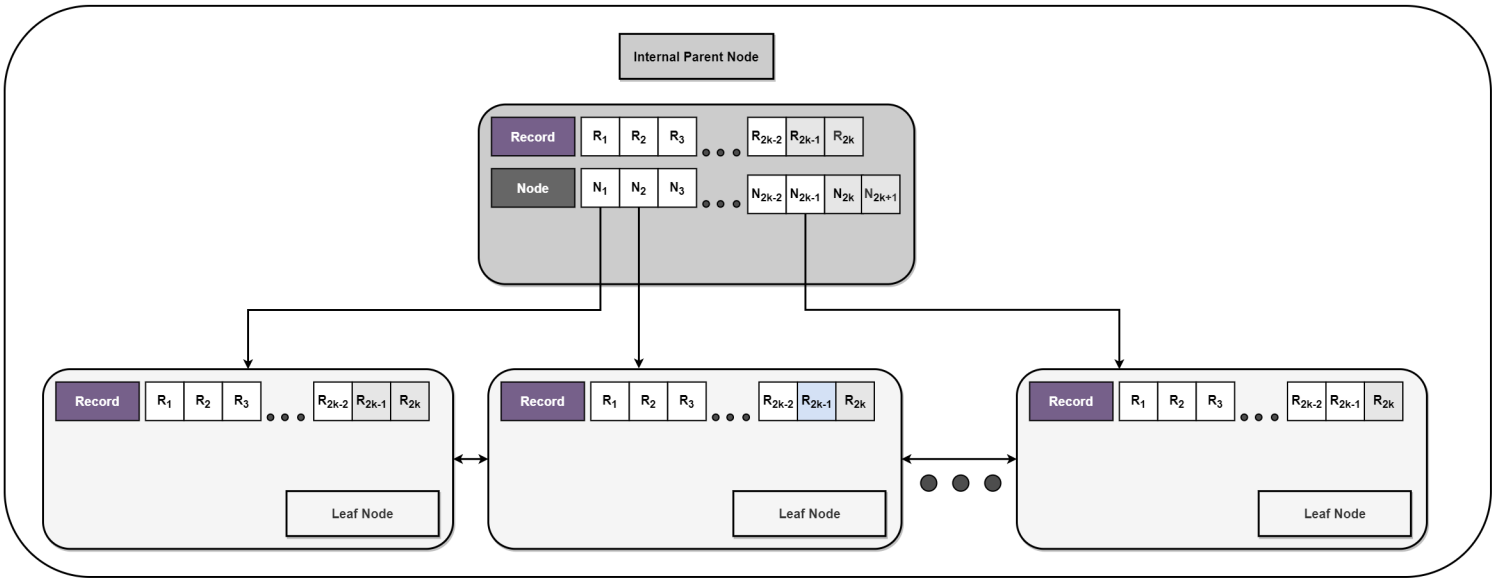


Fig. 3.8 – 3.11 analyze and describe the record reference insertion in a leaf node record semi-dynamic array structure without available allocated storage memory - capacity and the leaf node split process:

Figure 3.8: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 1

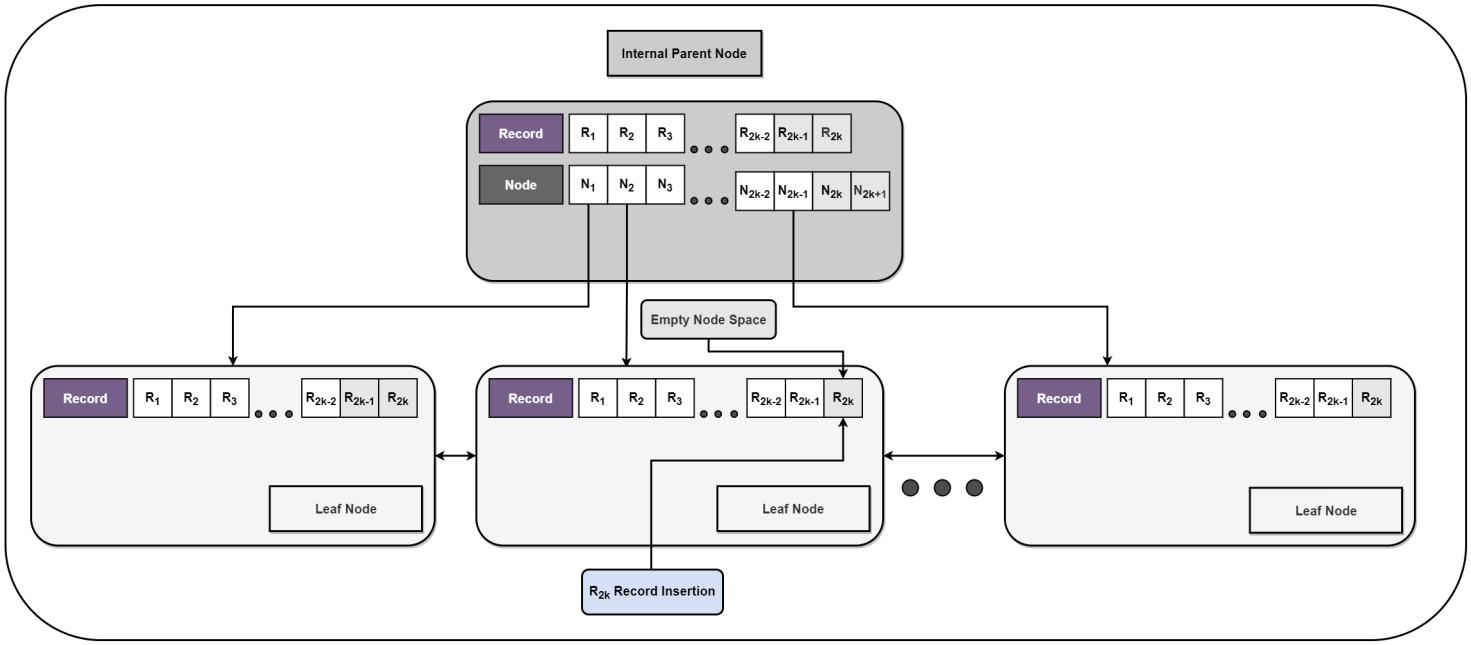


Figure 3.9: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 2

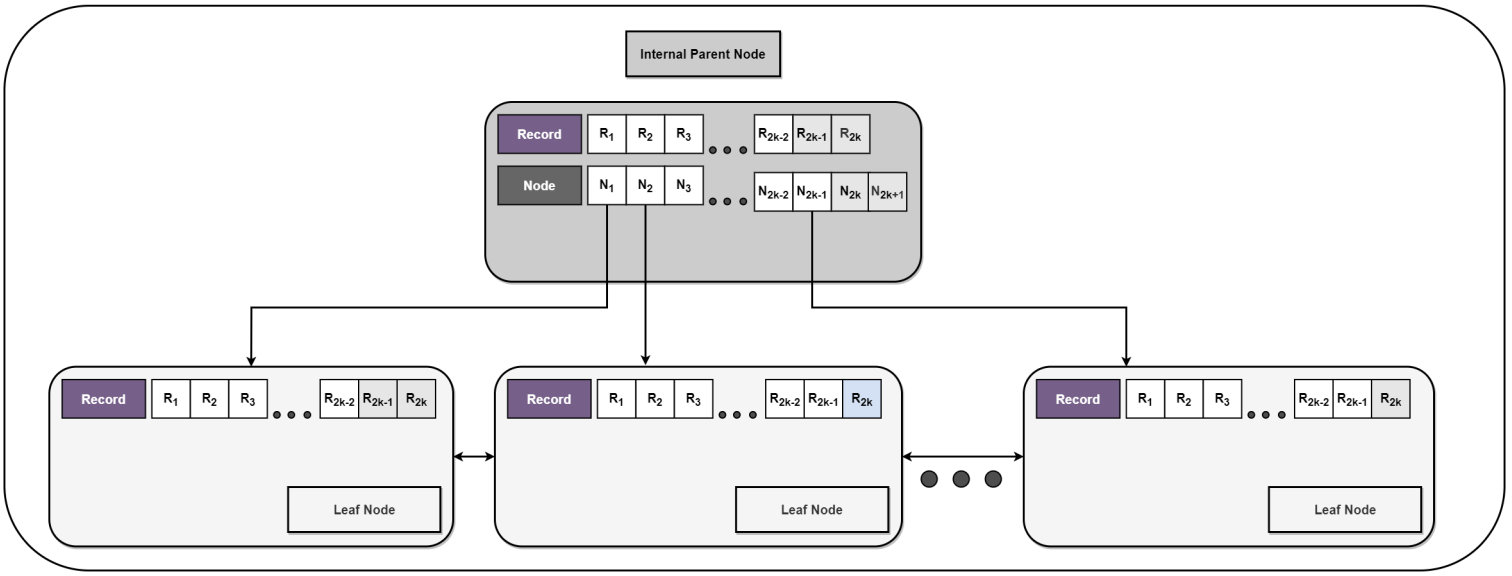


Figure 3.10: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 3

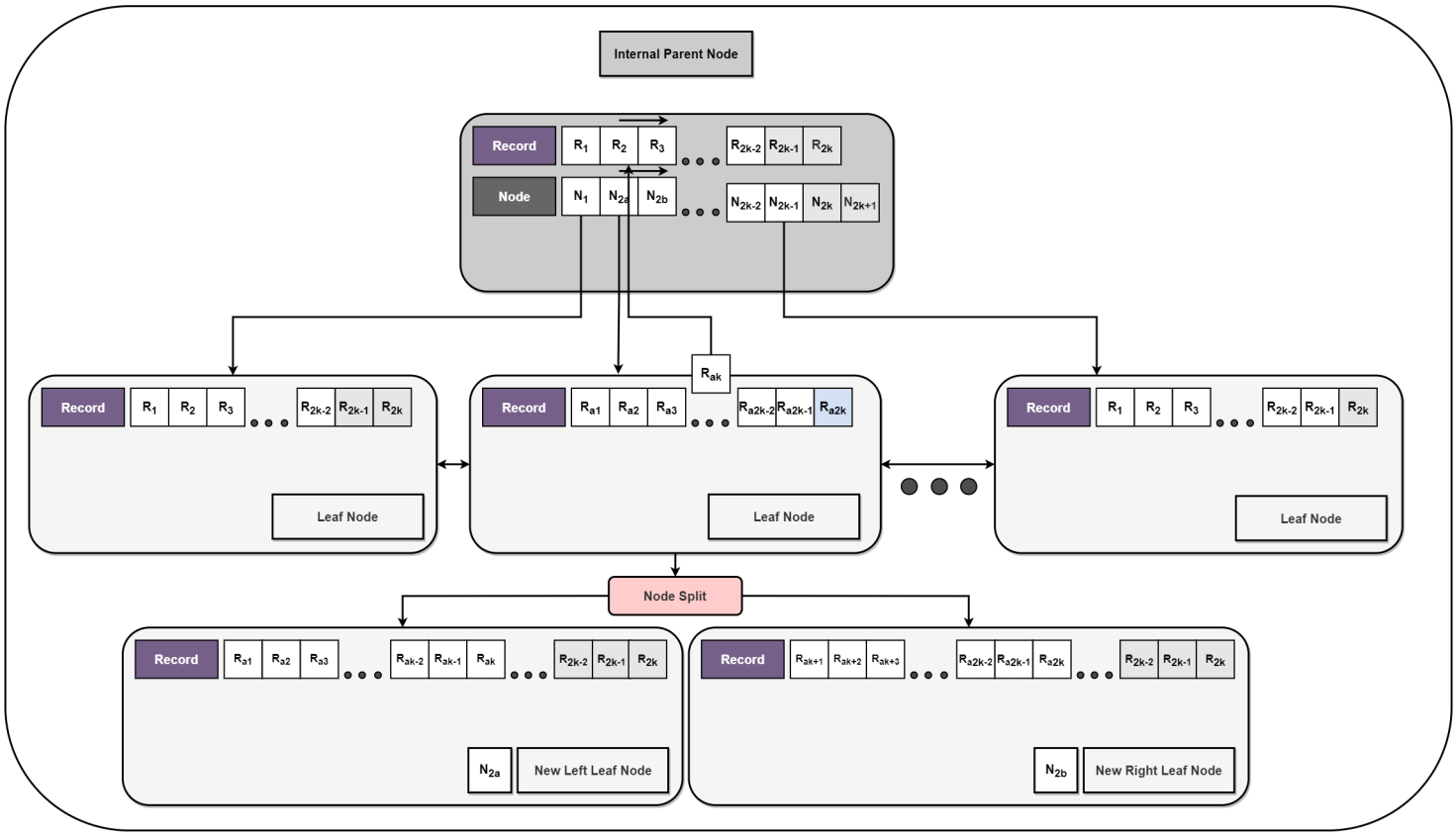
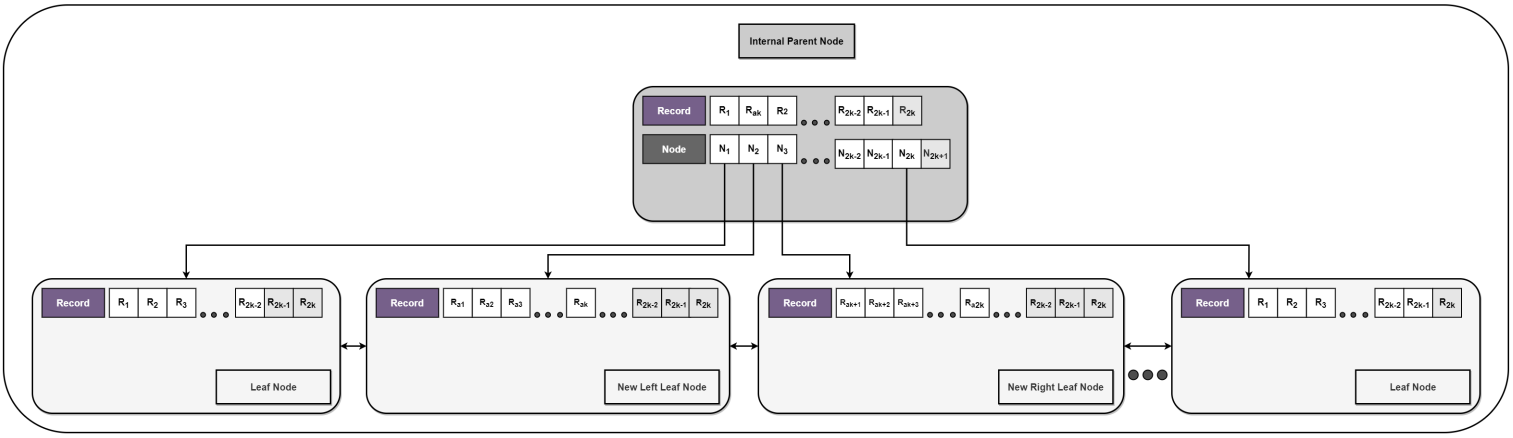


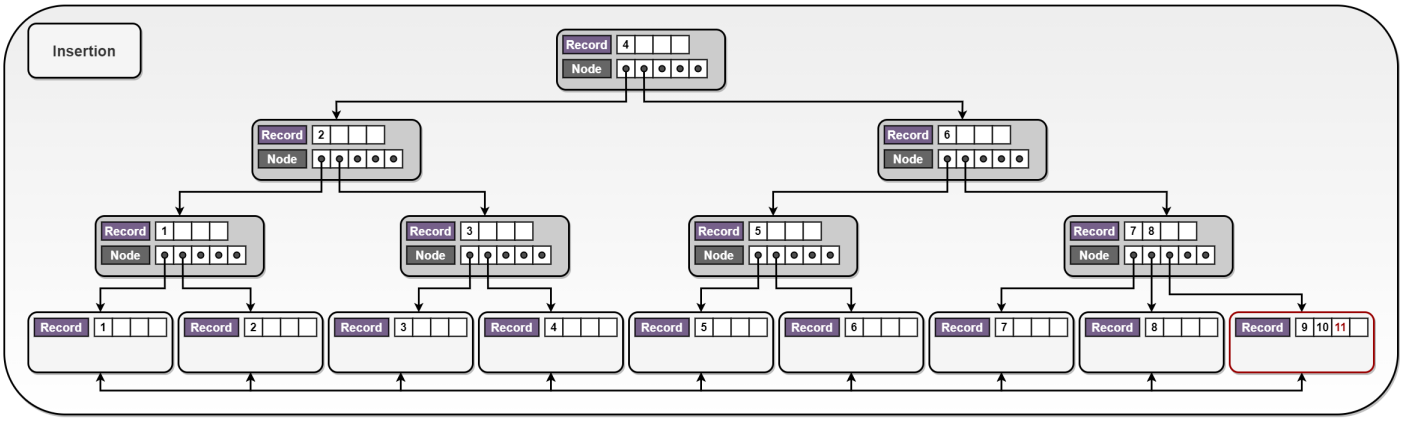
Figure 3.11: Record reference insertion process in a leaf node without available record semi-dynamic array structure capacity and node split - part 4



The implemented B$^+$-tree insertion functions in Alg. 13 – 15 are based on the node and node semi-dynamic array structures split, reconstruction and rearrangement algorithmic methods:

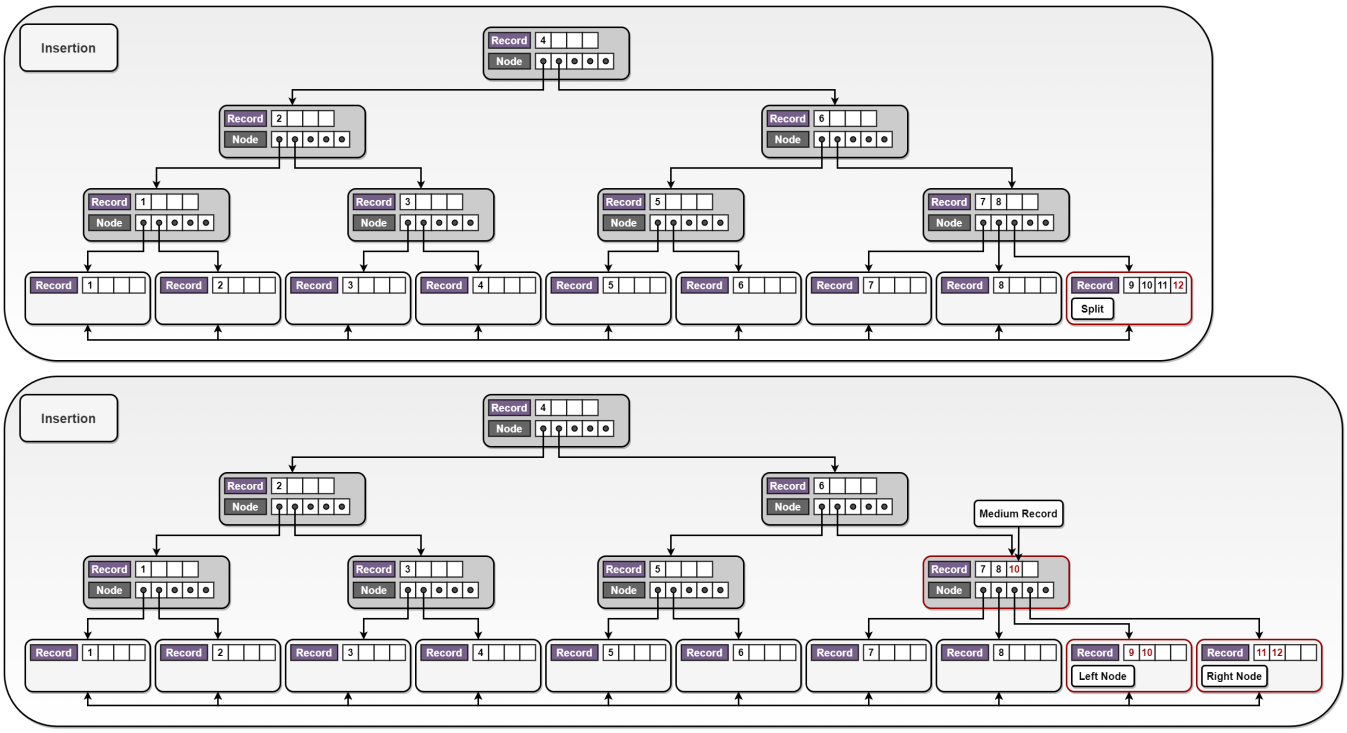
- Case 1 - record reference insertion in a leaf node record semi-dynamic array structure with available allocated storage memory - capacity (Fig. 3.12):

Figure 3.12: Record reference 11 insertion process in a leaf node with available record semi-dynamic array structure capacity



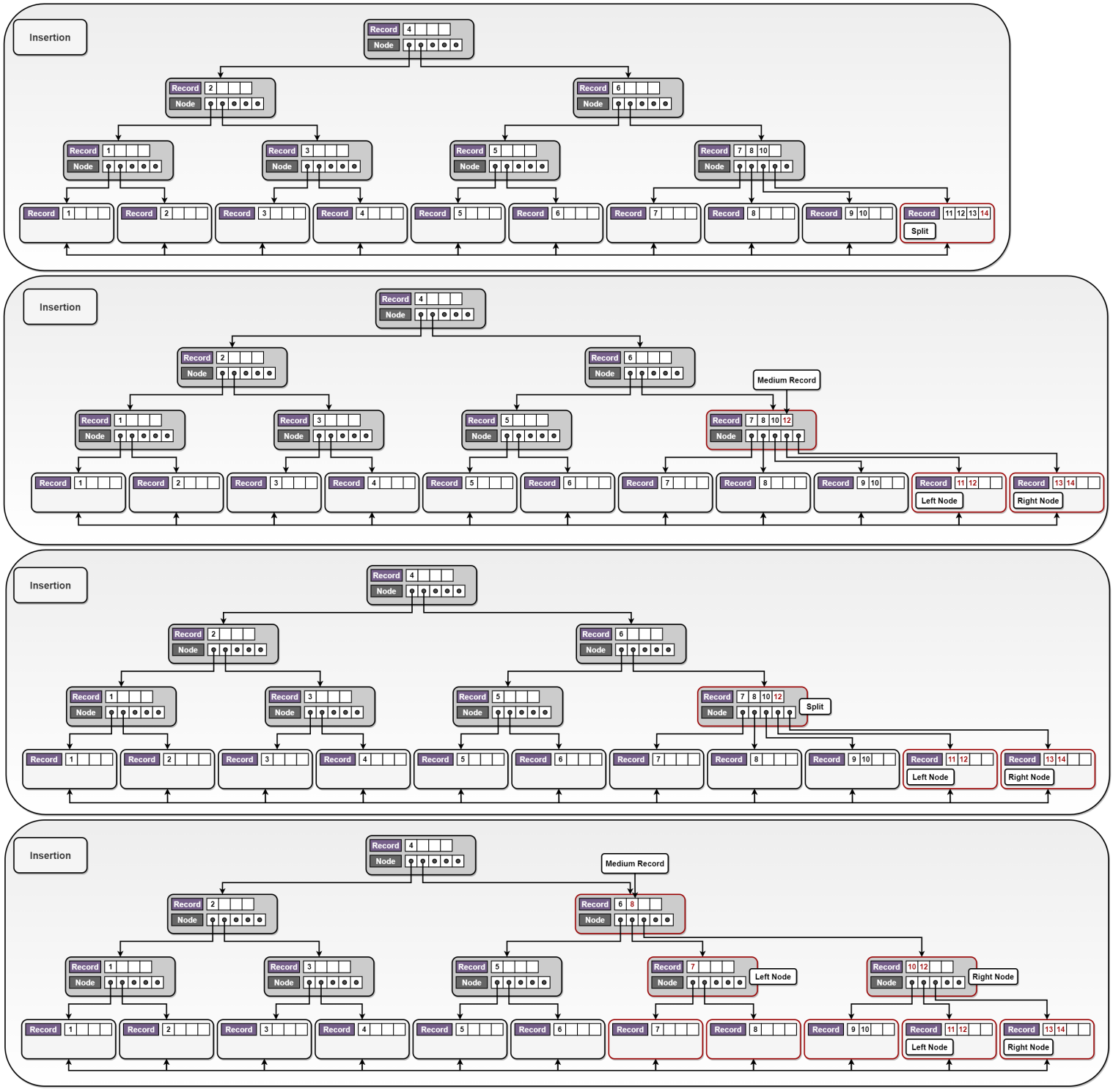
- Case 2 - record reference insertion in a leaf node record array without available allocated storage memory - capacity (leaf node split process) and the linked upper level node (parent node) has available capacity (Fig. 3.13):

Figure 3.13: Record reference 12 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has available capacity



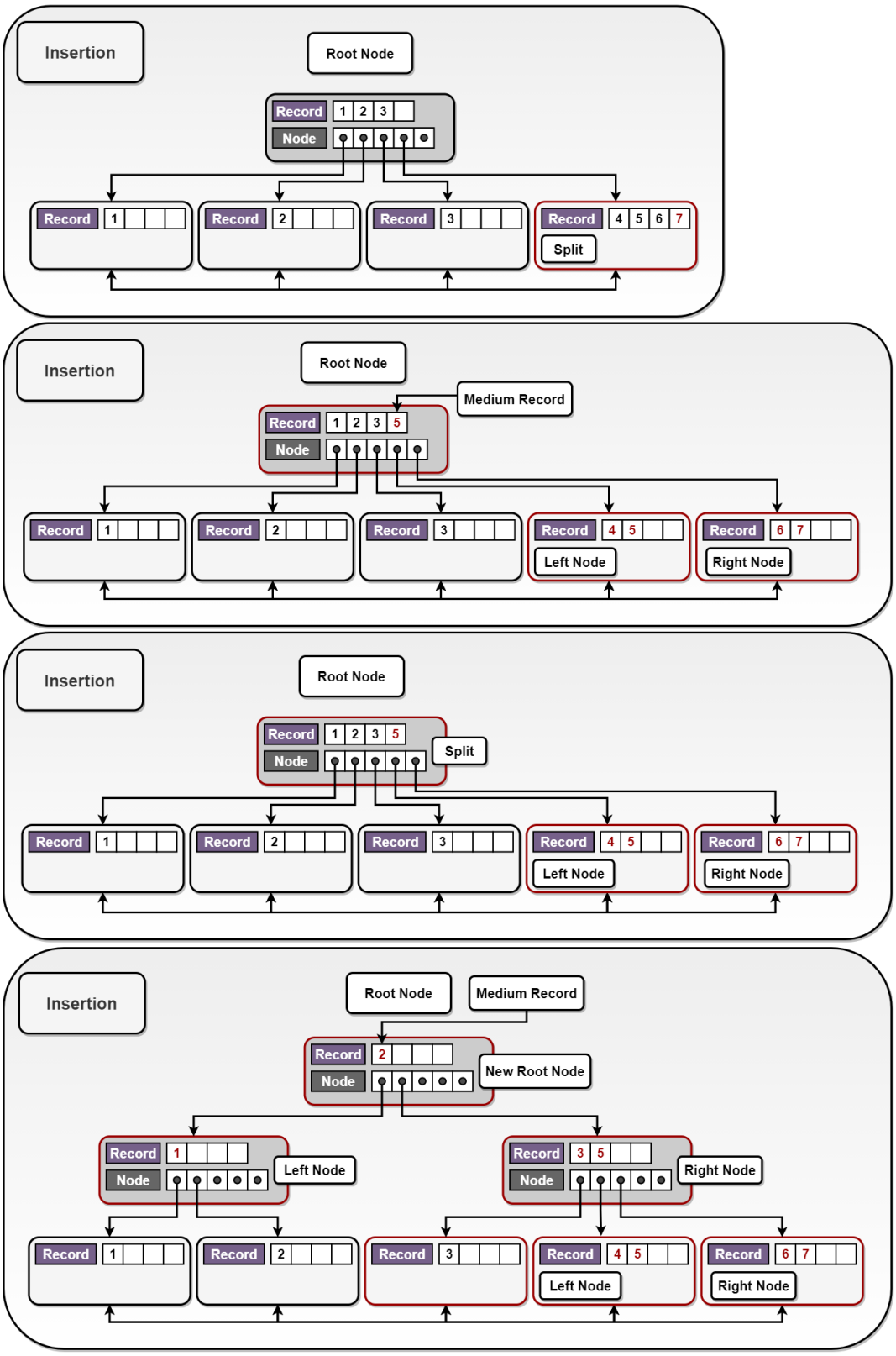
- Case 3 - record reference insertion in a leaf node record semi-dynamic array structure without available allocated storage memory - capacity (leaf node split process) and the linked upper level node (parent node) has not available capacity (linked upper level node split) (Fig. 3.14):

Figure 3.14: Record reference 14 insertion process in a leaf node without available record semi-dynamic array structure capacity and node split. The upper level linked node (parent node) record references semi-dynamic array structure has not available capacity (parent node node split process)



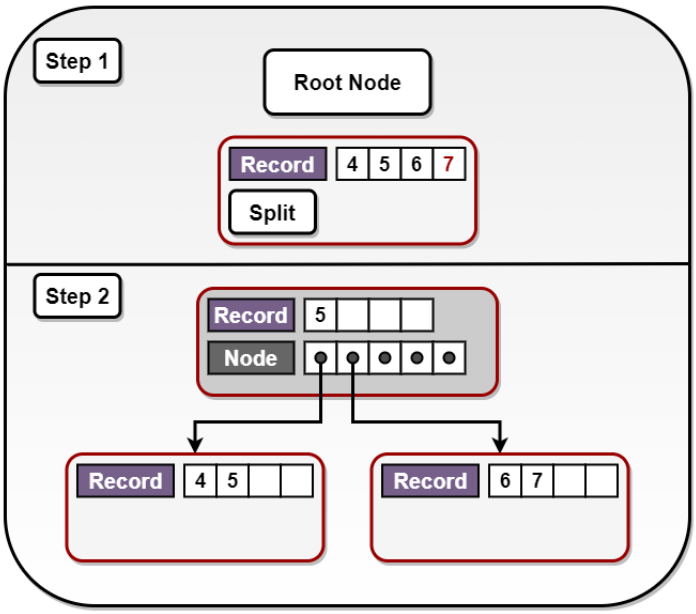
- Case 4 - root node split process (root is internal node) (Fig. 3.15):

Figure 3.15: Record reference 7 insertion process in a leaf node without available record semi-dynamic array structure capacity. The parent node has also not available capacity (parent node split). In case that all of the upper levels linked nodes has not available storage capacity - memory the split process is being implemented up to the root node



- Case 5 - root node split process (root is leaf node) (Fig. 3.16):

Figure 3.16: Record reference 7 insertion process in the leaf root node without available record semi-dynamic array structure capacity



Algorithm 13: BplusTreeInsertData function

Returned item: Insertion process status
BplusTreeInsertData(
B^+-tree structure item,
Record reference - data to be inserted
)

if *B^+-tree data structure is empty* **then**
    Root node construction - creation.
    Record reference insertion - storage in the root node.
    Return successful insertion status.
end

BplusTreeInsertNode_RootBreakTool()

if *BTreeInsertNode_RootBreakTool() is unsuccessful* **then**
    Return unsuccessful insertion status.
end

Return successful insertion status.

Algorithm 14: BplusTreeInsertNode_RootBreakTool function

Returned item: Insertion process status
BplusTreeInsertNode_RootBreakTool(
B^+-tree structure item,
Record reference - data to be inserted,
Node record references semi-dynamic array structure capacity - size
)

BplusTreeInsertNode_Tool()

if *Record reference has already been stored - inserted in the B^+-tree structure (duplicate stored record reference)* **then**
    Return unsuccessful insertion status.
end

if *Current node is the root node and a node break - split process was performed* **then**
    Left sub-node creation.
    Reconstruction - rearrangement of the root, left and right nodes set
    record and node references semi-dynamic array structures that the split process was implemented depending on the node type (leaf - internal node split).
end

Return successful insertion status.

Algorithm 15: BplusTreeInsertNode_Tool function

Returned item: Node split process - right sub-node item
BplusTreeInsertNode_Tool(
Current node item,
Record reference - data to be inserted,
Node split process - record references semi-dynamic array structure middle record,
Node record references semi-dynamic array structure capacity - size,
Duplication identifier of the inserted record reference
)

SearchBplusTreeNode_Record_ByPrimaryKey()

if *Record reference duplication* **then**
    Duplication identifier status update - unsuccessful insertion process.
    Storage obstruction - deallocation of the duplicate record reference.
    Return null node item.
end

if *Current node is a leaf node* **then**
    Record reference insertion - storage in the current leaf node
    if *Current leaf node record references semi-dynamic array structure has not available storage capacity* **then**
        Current leaf node split (right sub-node creation) and node semi-dynamic array structures reconstruction - rearrangement.
        Return right sub-node item.
    end
    Return null node item.
end

BplusTreeInsertNode_Tool()

if *Next level linked node split process was implemented* **then**
    Storage - insertion of the next level linked node (split node) middle record reference in the current node.
    Reconstruction - rearrangement of the current node references and record references semi-dynamic array structures.
else
    Return null node item.
end

if *Current internal node record references semi-dynamic array structure has not available storage capacity* **then**
    Current internal node split (right sub-node creation) and node semi-dynamic array structures reconstruction - rearrangement.
    Return right sub-node item.
end

Return null node item.

The total average algorithmic operations - steps of the insertion function can be approximately be approached based on the average B$^+$-tree structure height. Consequently, the theoretical average time complexity of the insertion function in Alg. 13 – 15 can be approximately calculated by the relation 3.8:

$$O(\ 1 + log_{(2ku_k+1)}(d_{ln} - 1)\ ) \tag{3.8}$$

3.2.5 Records deletion based on primary key fields

The B^+-tree index structure record reference deletion function and the deletion of the record that is stored in the RDBMS relational table to which the B^+-tree index structure is linked consists of multiple sub-functions (functional levels). These recursively functionally connected sub-function (functional levels) implement a part of the overall deletion process.

Alg. 16 implements the record reference location and deletion in the root node in the case that the root node is leaf and the leaf root node has not lower level linked child nodes. Furthermore implements the record reference location and deletion and the nodes - nodes semi-dynamic array structures reconstruction and rearrangement (re-balancing) utilizing Alg. 17 in case that the the B^+-tree is composed of multiple leaf and internal node structural parts. Alg. 17 implements the location and removal of the record reference to be deleted that is stored in the B^+-tree index structure and the physical record data deletion of the table that the B^+-tree index structure is linked. Furthermore implements the B^+-tree nodes and nodes internal semi-dynamic array structures reconstruction - rearrangement in order to re-balance the tree index. This reconstruction can be achieved using Alg. 18 and 19. Alg. 17 constitutes the basic deletion method as it functionally links all the individual functional parts of the overall deletion process. The deletion process is separated in multiple functional parts. The leaf node record reference deletion and the reconstruction of the leaf and internal nodes set that were affected by the deletion, to re-balance the index. Alg. 18 deletes - removes the record reference that is stored in a leaf node and implements the leaf and upper level linked nodes set reconstruction that are structurally affected from the deletion process. It therefore restores the structural balance of the B^+-tree index structure at the bottom leaf nodes level in the case that the tree can be directly re-balanced at the leaf level. In the case that the B^+-tree index structure can not be re-balanced at the bottom leaf nodes level, Alg. 19 is used to reconstruct the internal nodes set on some upper internal nodes level in order to structurally reconstruct and re-balance the tree. These functions in Alg. 18 and 19 use a sub-functions set to implement the B^+-tree nodes reconstruction and re-balancing:

- The sub-function BplusTree_ReplaceRecord() implements the deletion in a leaf node that contains multiple record references. The sub-function BplusTree_ReplaceRecord() implements the structural re-balancing of the B^+-tree on the bottom leaf nodes level.

Figure 3.17: Deletion of the record 6 in a leaf node that contains multiple record references - case 1

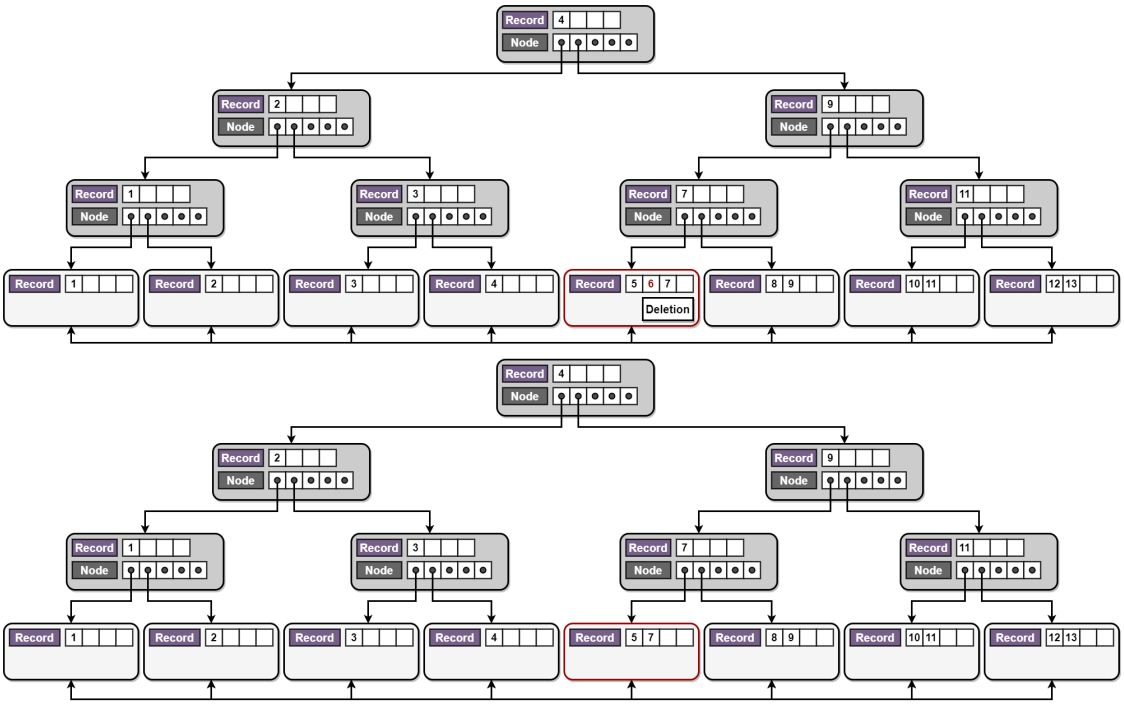


Figure 3.18: Deletion of the record 7 in a leaf node that contains multiple record references - case 2

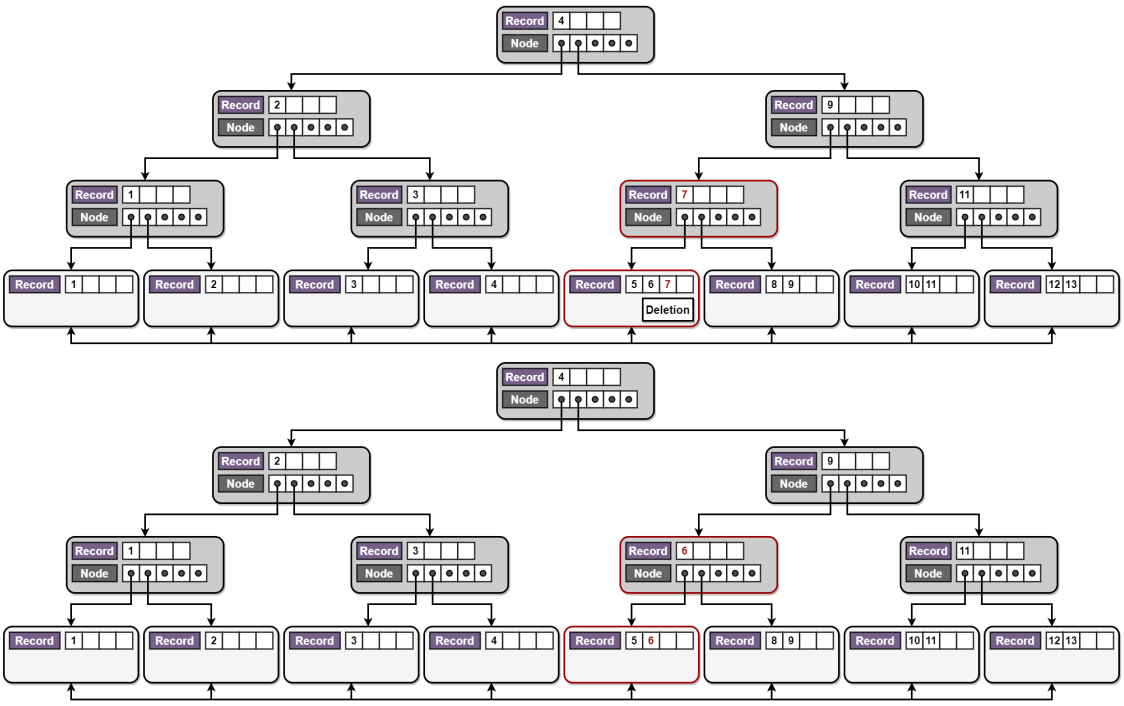
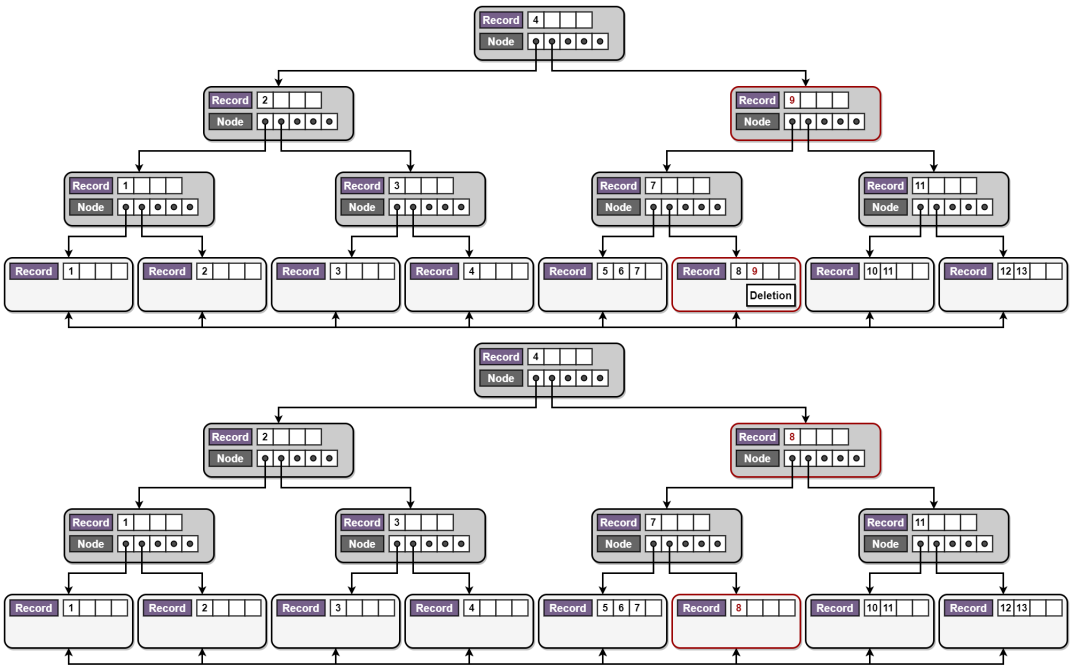


Figure 3.19: Deletion of the record 9 in a leaf node that contains multiple record references - case 3



- The sub-functions BplusTree_RebalanceLeftNode() and BplusTree_RebalanceRightNode() implement the deletion in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references. These sub-functions implement the structural re-balancing of the B$^+$-tree on the bottom leaf nodes level.

Figure 3.20: Deletion of the record 5 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references - case 1

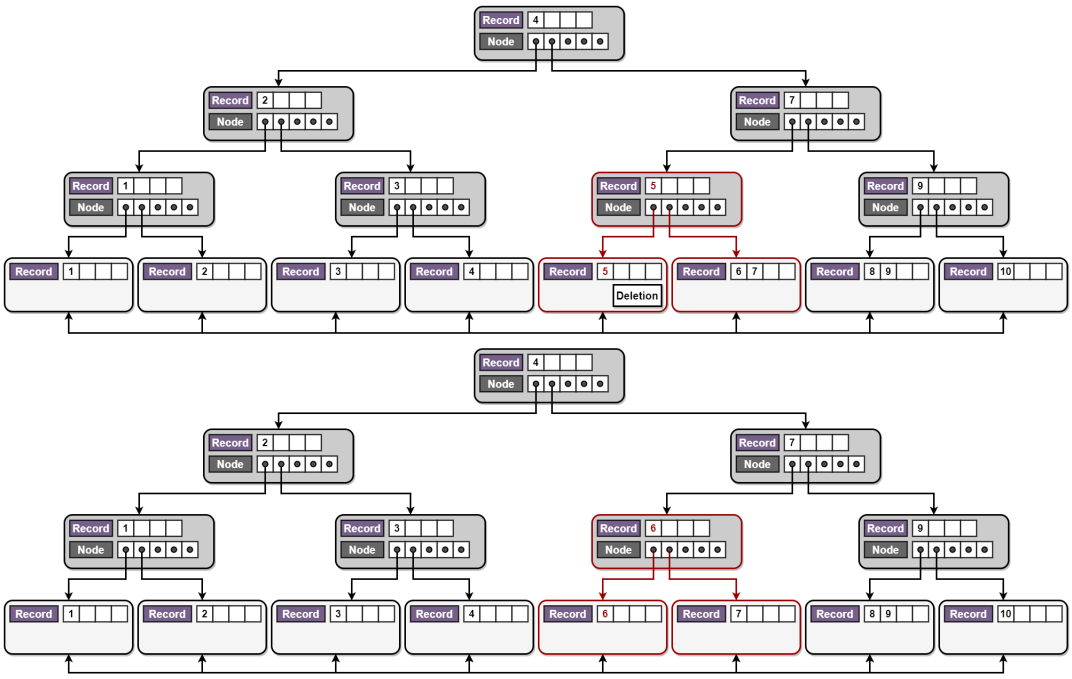
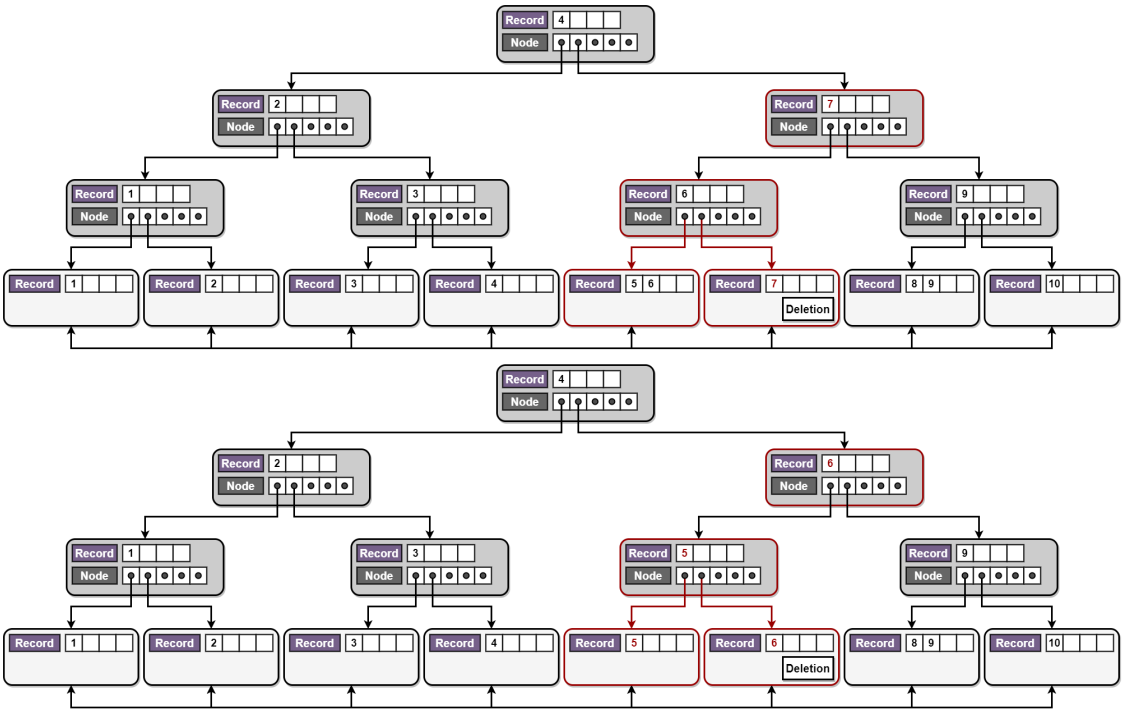


Figure 3.21: Deletion of the record 7 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains a single record reference and the left - right side node contains multiple record references - case 2



- The BplusTree_ReplaceRecord_Right_to_Left() and BplusTree_ReplaceRecord_Left_to_Right() implement the deletion in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains multiple record references or a single record reference. These sub-functions implement the structural re-balancing of the B^+-tree on the bottom leaf nodes level.

Figure 3.22: Deletion of the record 6 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains a single record reference - case 1

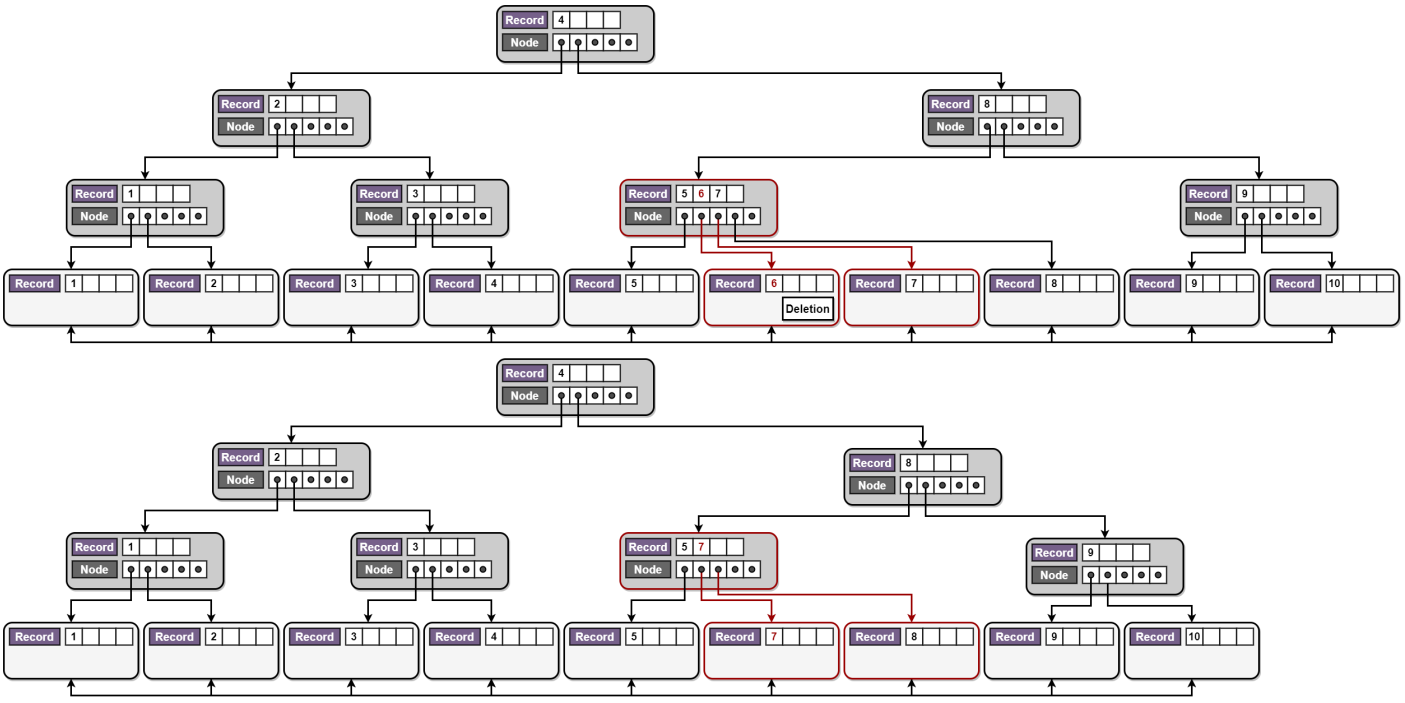


Figure 3.23: Deletion of the record 8 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains a single record reference - case 2

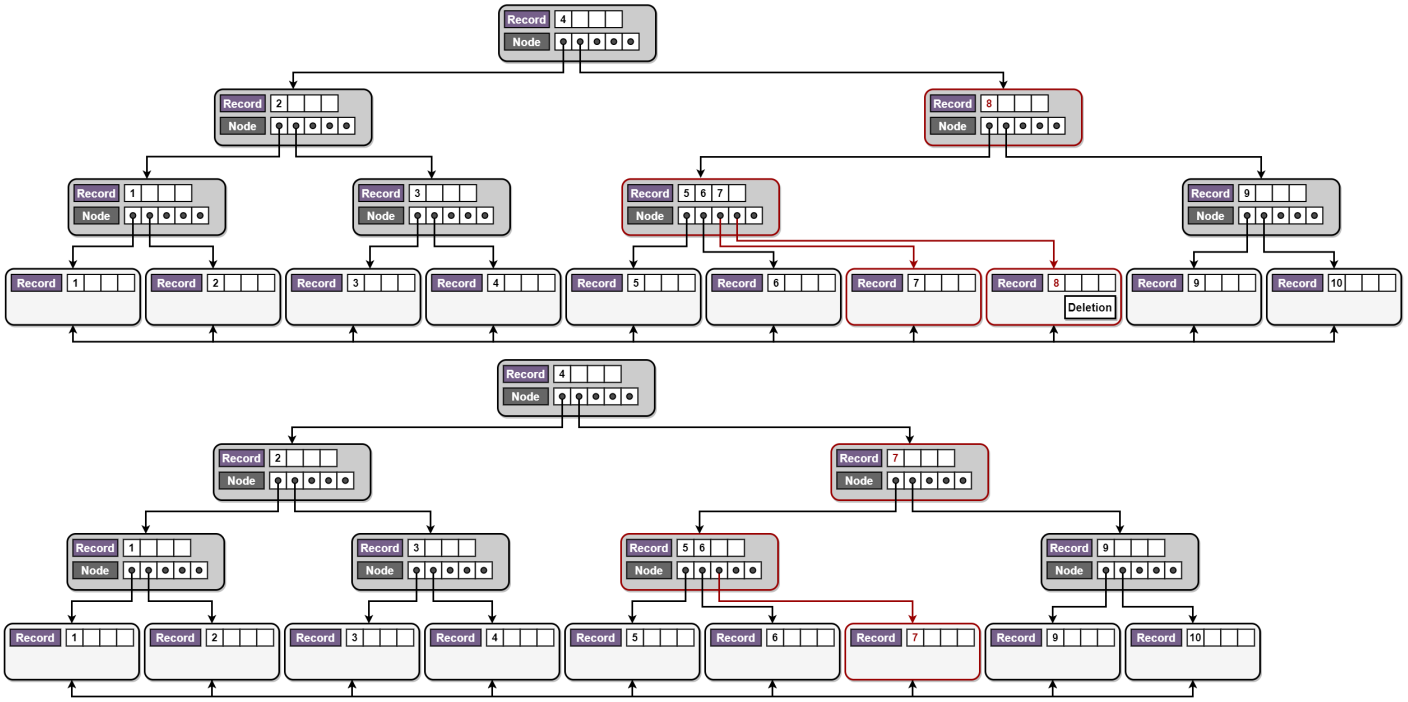


Figure 3.24: Deletion of the record 8 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains multiple record references - case 1

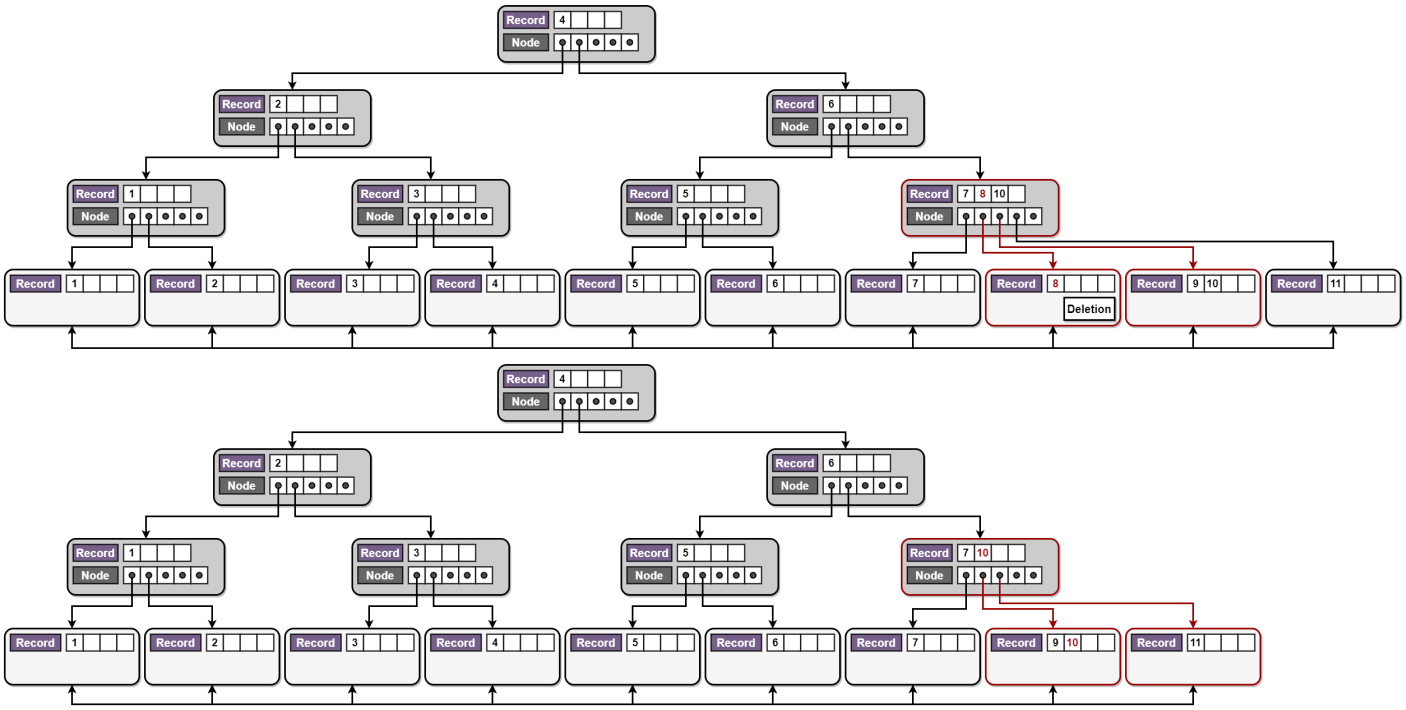
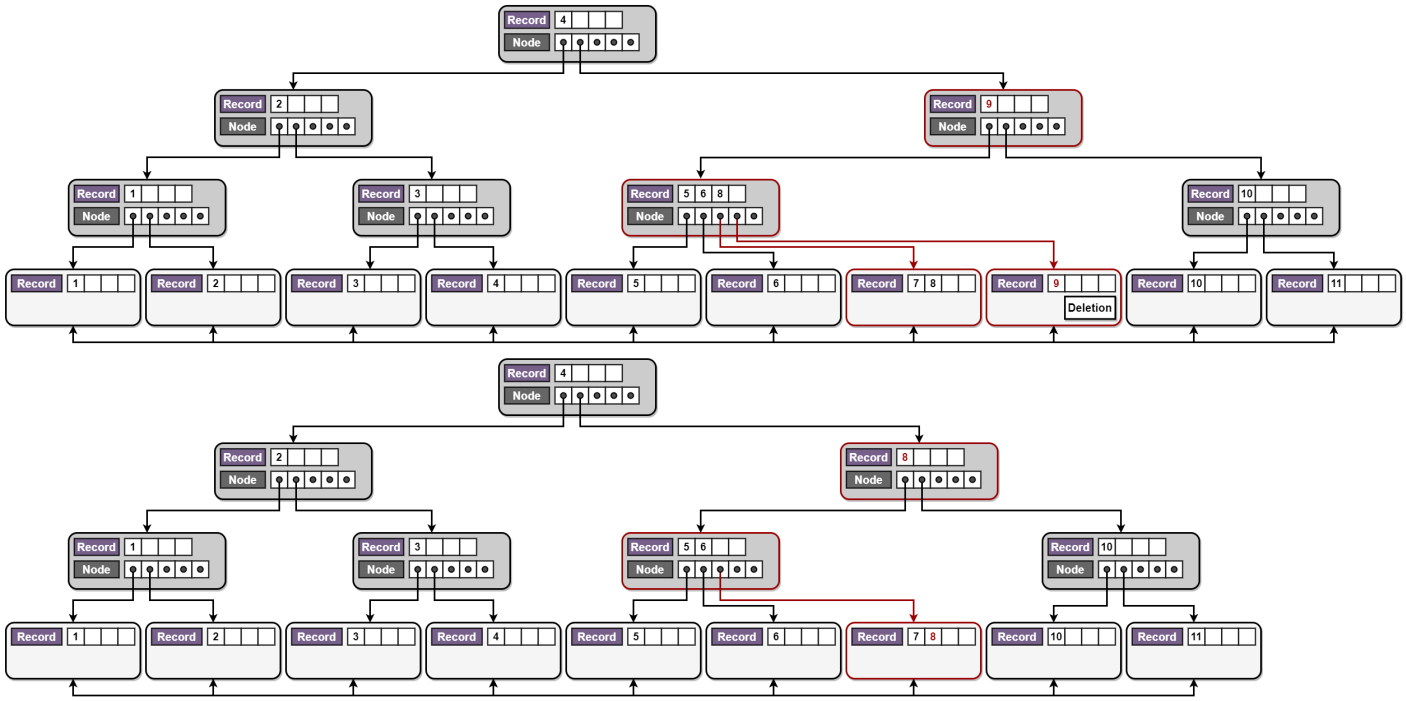


Figure 3.25: Deletion of the record 9 in a leaf node that contains a single record reference, the upper level linked node (parent node) contains multiple record references and the left - right side node contains multiple record references - case 2



- The sub-functions BplusTree_MergeLeftNode() and BplusTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. These sub-functions implement the record reference deletion in the leaf node and the nodes reconstruction on the bottom leaf nodes level in order to structurally re-balance the B^+-tree. In this case that the structural B^+-tree balance cannot be restored - recovered on the leaf nodes level a nodes re-balancing and reconstruction recursive process is being implemented to the upper nodes levels.
  If this problematic nodes structural balancing case is caused up to the root node the sub-functions BplusTree_MergeLeftNodeRecursive() and the BplusTree_MergeRightNodeRecursive() are used recursively to fix this problem from the leaf to the root node level. The structural balance finally restored - recovered on the root node level. These functions implement the same reconstruction process with those functions of the B-tree structure.

Figure 3.26: Deletion of the record 5 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference - case 1

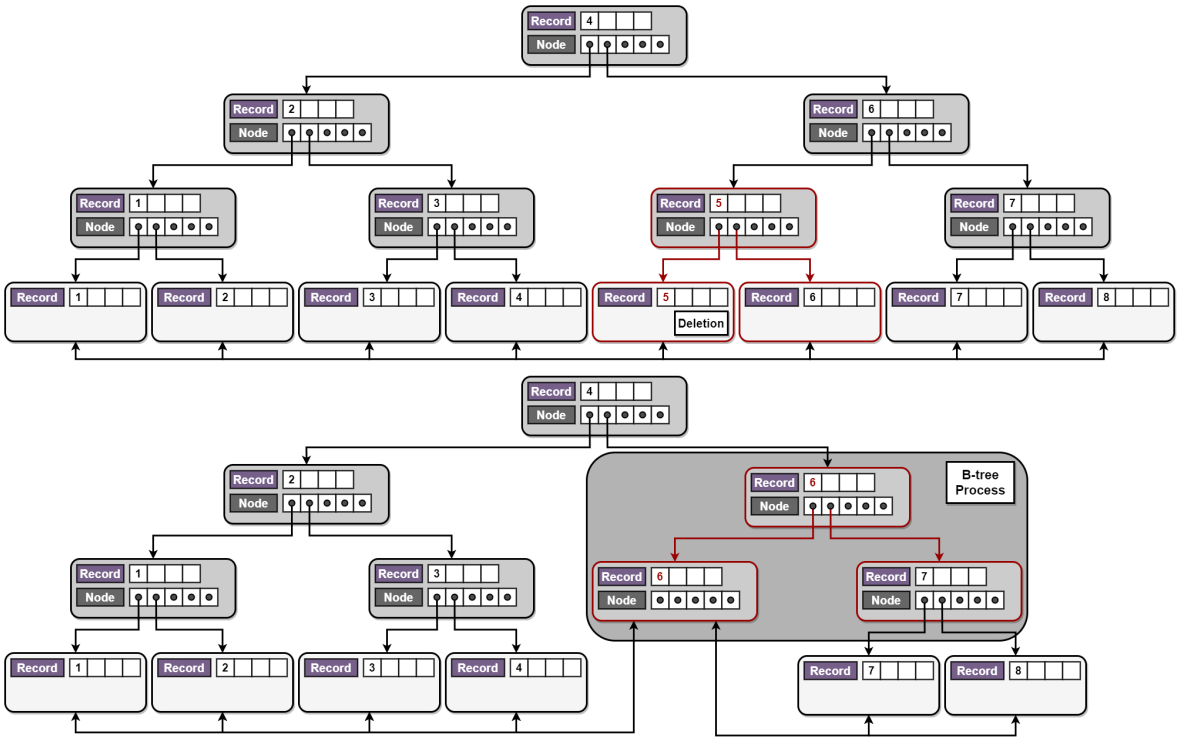
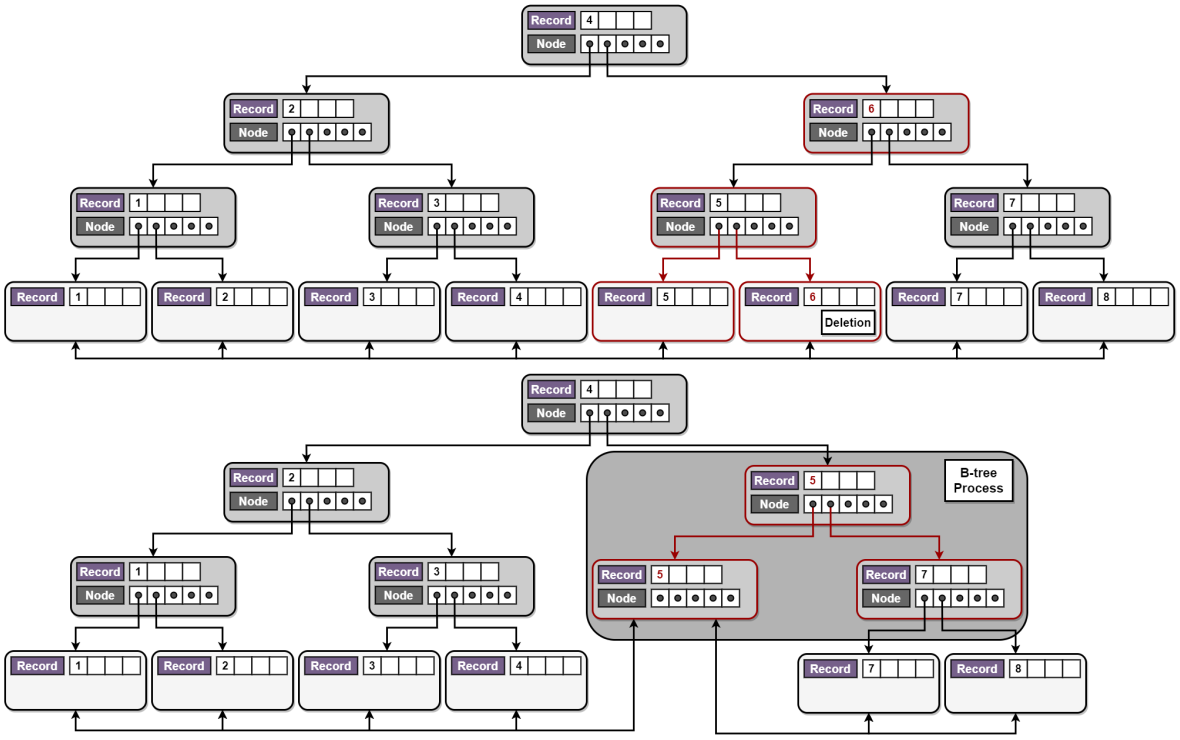


Figure 3.27: Deletion of the record 6 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference - case 2



- The sub-functions BplusTree_MergeLeftNode() and BplusTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Furthermore the sub-functions BplusTree_ReplaceLeftNodeRecursive() and BplusTree_ReplaceRightNodeRecursive() implement the B^+-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has a single record reference and the left - right side node contains multiple record references. These functions implement the same reconstruction process with those functions of the B-tree structure.

Figure 3.28: Deletion of the record 5 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B⁺-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has a single record reference and the left - right side node contains multiple record references - case 1

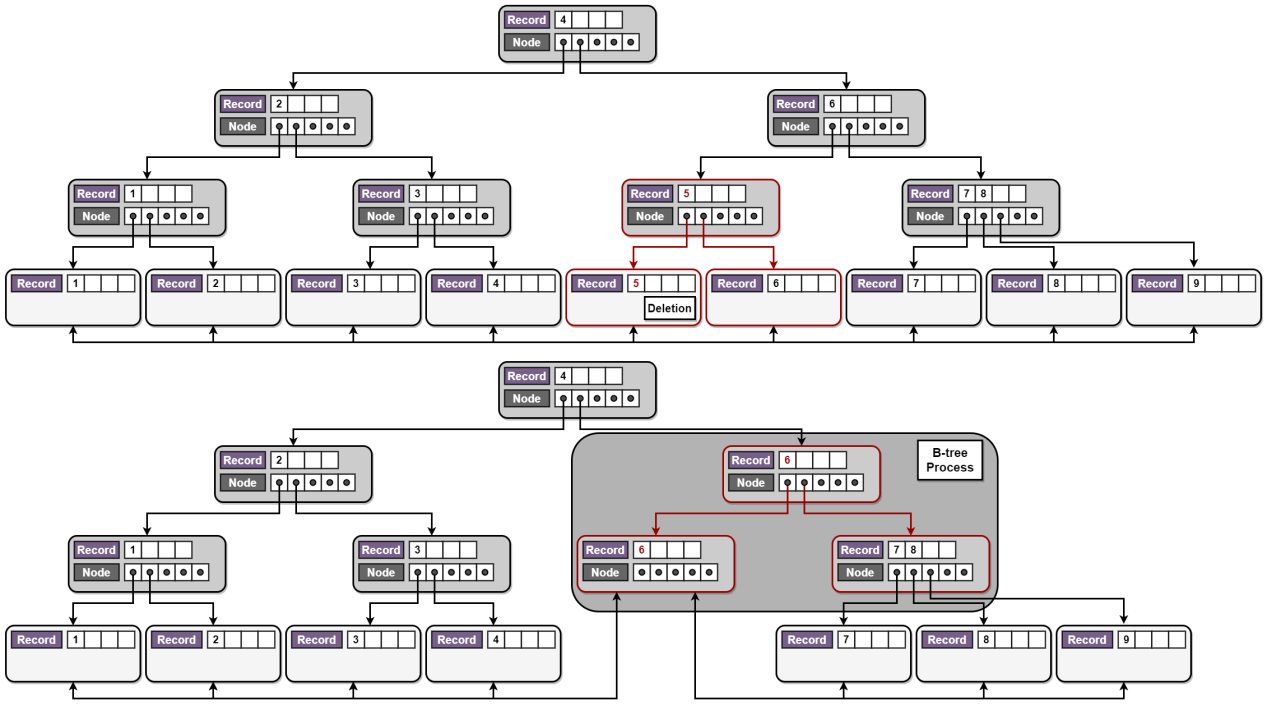
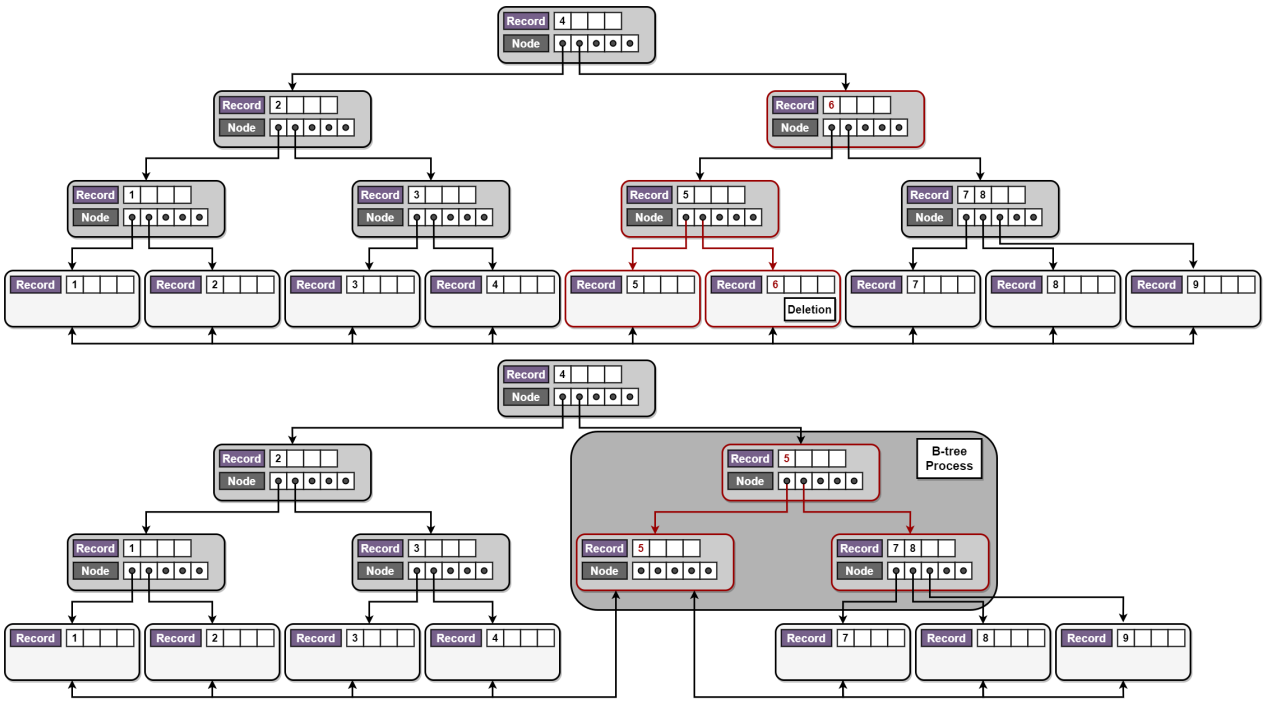


Figure 3.29: Deletion of the record 6 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B⁺-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has a single record reference and the left - right side node contains multiple record references - case 2



- The sub-functions BplusTree_MergeLeftNode() and BplusTree_MergeRightNode() implement the deletion in a leaf node that contain a single record reference and the upper level linked (parent node) and the left - right side node contains a single record reference. Furthermore the sub-functions BplusTree_ReplaceSingleLeftNodeRecursive() and BplusTree_ReplaceSingleRightNodeRecursive() implement the B^+-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference. These functions implement the same reconstruction process with those functions of the B-tree structure.

Figure 3.30: Deletion of the record 5 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference - case 1

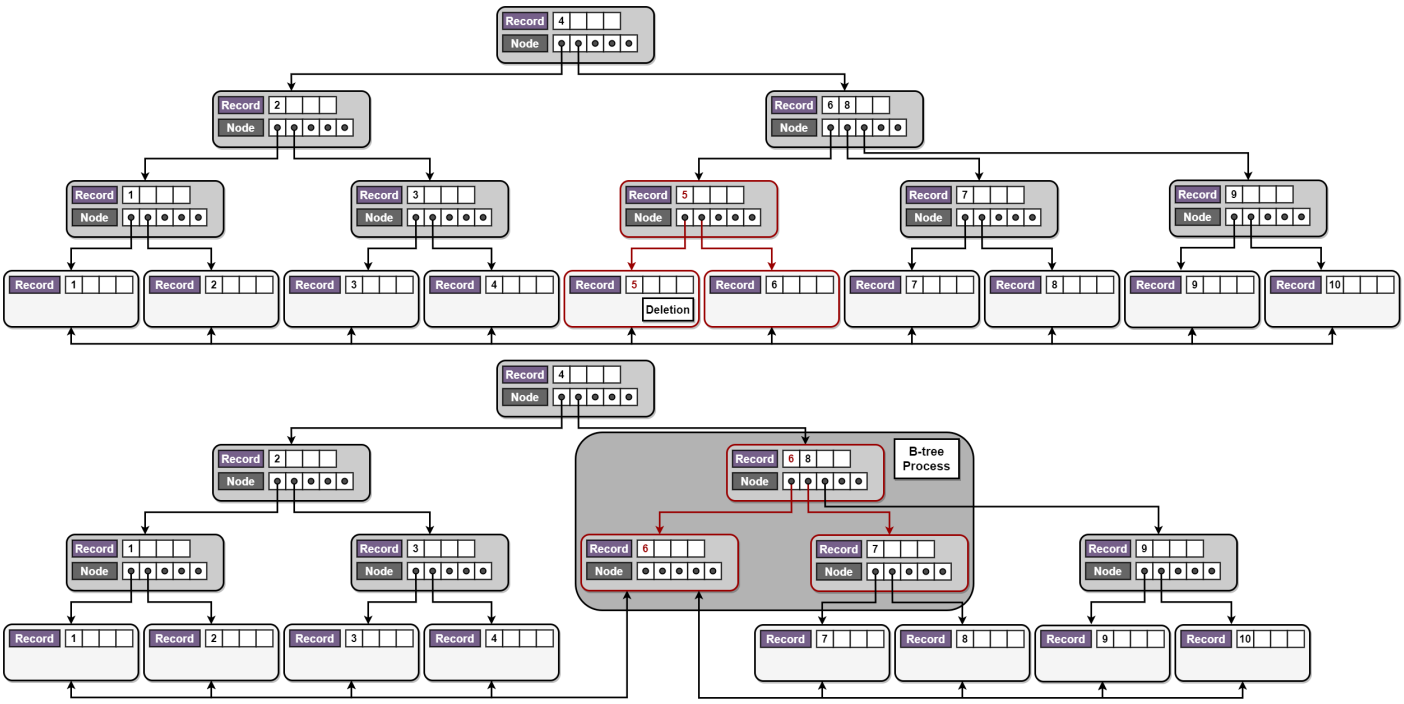
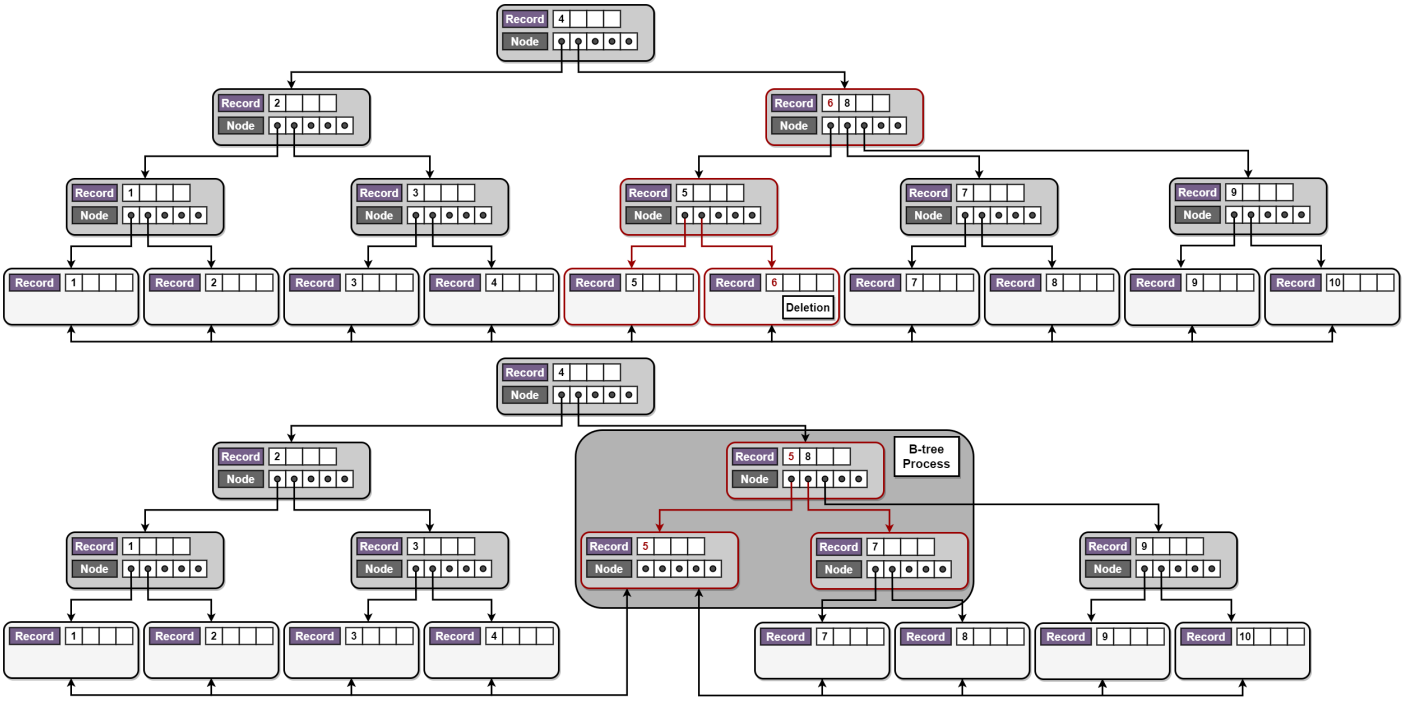


Figure 3.31: Deletion of the record 6 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains a single record reference - case 2



- The sub-functions BplusTree_MergeLeftNode() and BplusTree_MergeRightNode() implement the deletion in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Furthermore the sub-functions BplusTree_ReplaceMultipleLeftNodeRecursive() and BplusTree_ReplaceMultipleRightNodeRecursive() implement the B^+-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references. These functions implement the same reconstruction process with those functions of the B-tree structure.

Figure 3.32: Deletion of the record 5 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B$^+$-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references - case 1

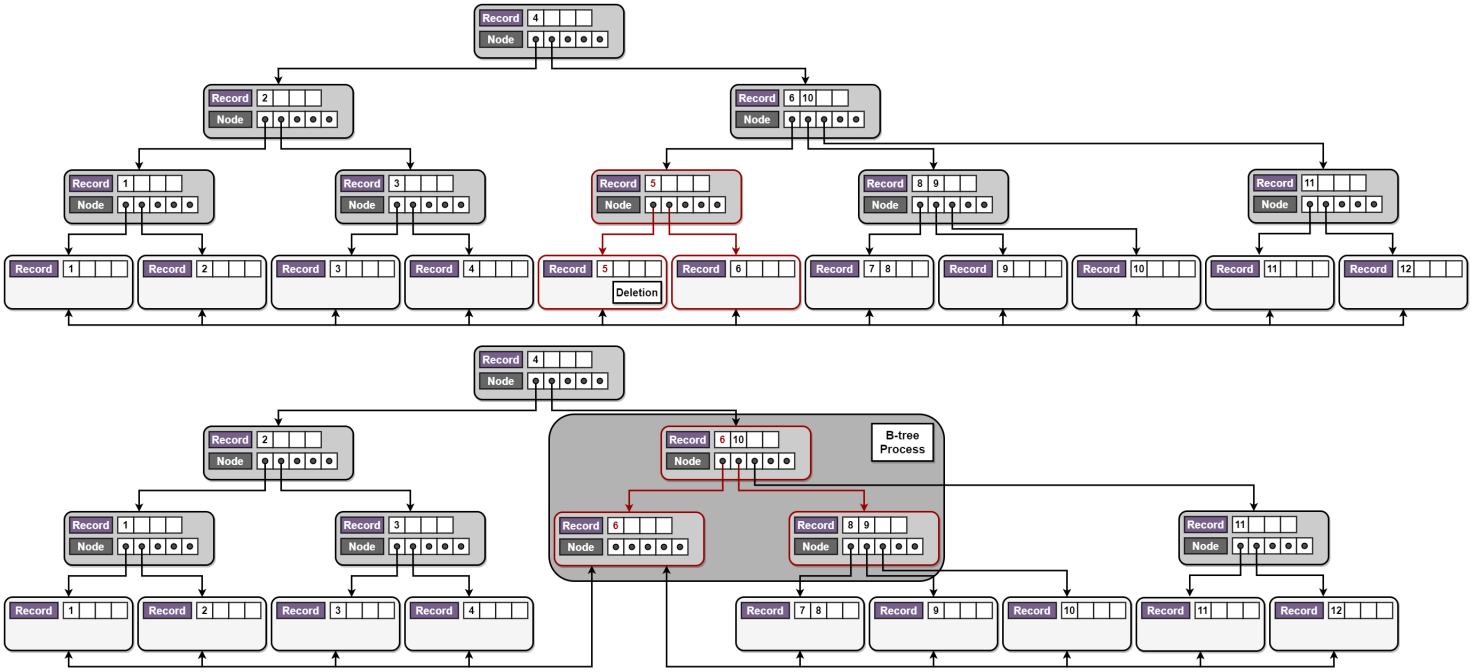
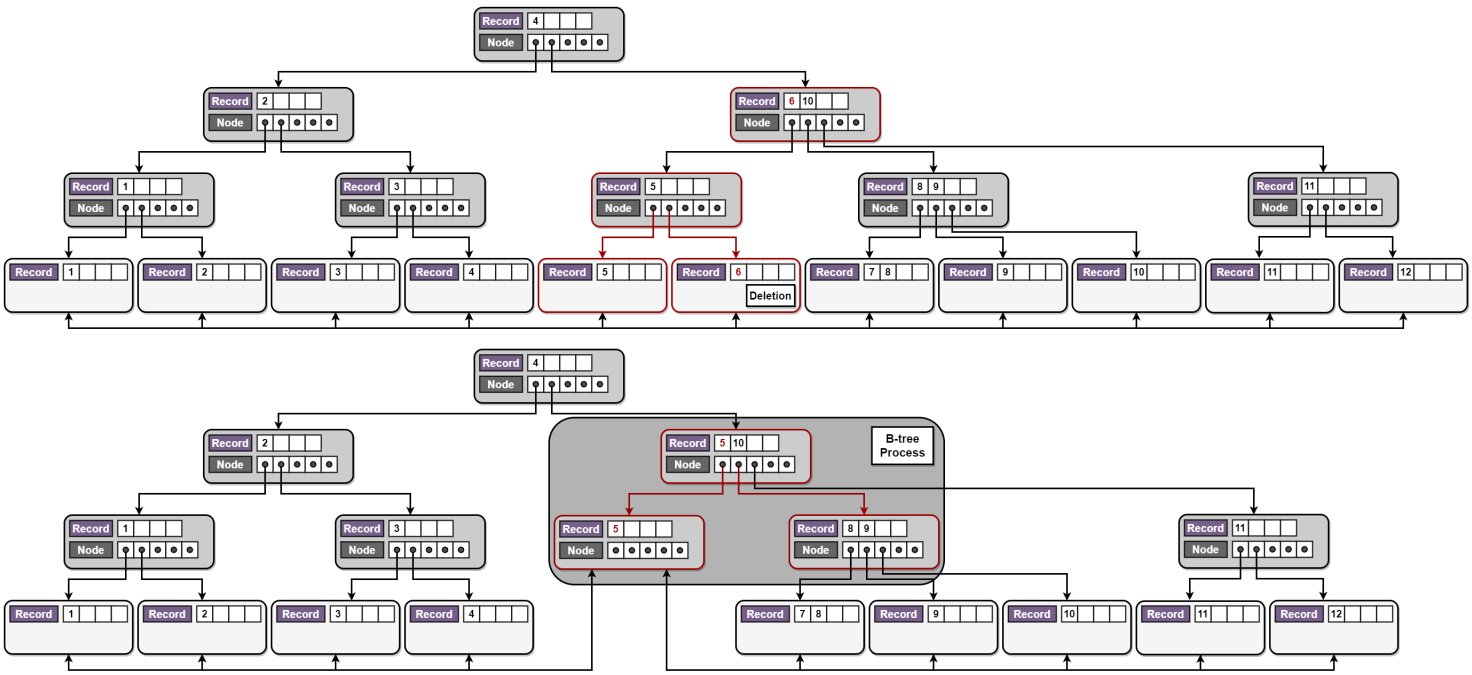


Figure 3.33: Deletion of the record 6 in a leaf node that contains a single record reference and the upper level linked (parent node) and the left - right side node contain a single record reference. Implementation of the B$^+$-tree nodes re-balancing - reconstruction on an internal nodes level in the case that the upper level linked (parent node) of the previous level reconstructed nodes set has multiple record references and the left - right side node contains multiple record references - case 2



The record references set of a B$^+$-tree index structure is stored in the leaf nodes and a subset of them is stored in both leaf and internal nodes. Therefore the deletion of a stored record is implemented at the bottom leaf nodes level and at the upper levels in case that the balance can not be restored at the leaf level using Alg. 18 and 19 as all the record references are stored in the last tree structure level. In case that the record to be deleted is stored in both leaf and internal node is also removed from the internal node.

Algorithm 16: BplusTreeDeleteData function

```
Returned item: Deletion process status
 BplusTreeDeleteData(
 B+-tree structure item,
 Primary key field of the record to be deleted,
 Item to store the removed record reference
 )
if B+-tree data structure is empty then
    Deletion cannot be implemented.
    Return unsuccessful deletion status.
end
if Root node is a leaf node then
    SearchBplusTreeNode_Record_ByPrimaryKey()
    if Record reference to be deleted is located in the current root leaf node then
        if Root node contains a single record reference then
            Remove the stored record reference from the current node.
            Deletion of the current root node.
            Return successful deletion status.
        end
        BplusTree_ReplaceRecord()
        Return successful deletion status.
    end
    Return unsuccessful deletion status.
end
BplusTreeDeleteNode()
if Record reference to be deleted is not stored in the B+-tree structure then
    Deletion cannot be implemented.
    Return unsuccessful deletion status.
end
Return successful deletion status.
```

Algorithm 17: BplusTreeDeleteNode function

Returned item: B^+-tree node item

BplusTreeDeleteNode(
Next level node item,
Primary key field of the record to be deleted,
Item to store the removed record reference,
Internal node storage position of the record to be deleted,
Maximum record - node semi-dynamic array structures capacity of each node,
Flag to specify if the balancing process has been activated to an internal node,
Flag to specify if the internal node record deletion procedure has been activated,
Flag to specify if the record to be deleted exists in the structure
)

SearchBplusTreeNode_Record_ByPrimaryKey()

if *The record reference to be deleted is located at the current internal node* **then**

Storage of the internal node and the semi-dynamic array structure that the record to be deleted is stored in the B^+-tree.

Activation of the internal node deletion process.

end

if *Next level node is an internal node* **then**

BplusTreeDeleteNode()

if *The structural balance recovery - restoration cannot be implemented on the previous level. The upper level node re-balancing and reconstruction process has been activated* **then**

BplusTreeDelete_NonLeafNode()

end

Return Current node item.

end

BplusTreeDelete_LeafNode()

Return Current node item.

Algorithm 18: BplusTreeDelete_LeafNode function

Returned item: Void item
BplusTreeDelete_LeafNode(
Upper level - parent node item,
Primary key field of the record to be deleted,
Item to store the removed record reference,
Position of the stored record reference to be deleted in the next level leaf node,
Internal node storage position of the record to be deleted,
Maximum record - node semi-dynamic array structures capacity of each node,
Flags to specify the internal node balancing and record deletion processes and the record to be deleted existence in the structure
)

SearchBplusTreeNode_Record_ByPrimaryKey()

if *The record reference to be deleted is located at the current leaf node* **then**
  if *The current leaf node contains multiple record references* **then**
    BplusTree_ReplaceRecord()
    Return void item.
  end
  if *The current upper level - parent node and each of the leaf and left - right side nodes contain a single record reference* **then**
    BplusTree_MergeRightNode() or **BplusTree_MergeLeftNode()**
    Remove the record reference from the internal node that is stored and reconstructs the node if the record is also stored in an internal node.
    Activation of the structural nodes re-balancing process to the upper level.
    Return void item.
  end
  if *The current upper level - parent node and the leaf node contain a single record reference and the left - right side node contains multiple record references* **then**
    BplusTree_RebalanceRightNode() or **BplusTree_RebalanceLeftNode()**
    Remove the record reference from the internal node that is stored and reconstructs the node if the record is also stored in an internal node.
    Return void item.
  end
  if *The current upper level - parent node contains multiple record references, the leaf node contains a single record reference and the left - right side node contains multiple record references or a single record reference* **then**
    BplusTree_ReplaceRecord_Left_to_Right() or
    BplusTree_ReplaceRecord_Right_to_Left()
    Remove the record reference from the internal node that is stored and reconstructs the node if the record is also stored in an internal node.
    Return void item.
  end
end

Return void item.

Algorithm 19: BplusTreeDelete_NonLeafNode function

Returned item: Void item
BplusTreeDelete_NonLeafNode(
Current upper level node item,
Position of the next level linked node in the node references semi-dynamic array structure that the nodes reconstruction - re-balancing process was implemented,
Maximum record - node semi-dynamic array structures capacity of each node,
Flag to specify the internal node balancing process
)

if *The current upper level - parent node contains a single record reference and the next level left - right side node of the previous reconstructed node contains a single record reference* **then**

> **BplusTree_MergeLeftNodeRecursive()** or
> **BplusTree_MergeRightNodeRecursive()**
> Activation of the structural nodes re-balancing process to the upper level.
> **Return** void item.

end

if *The current upper level - parent node contains a single record reference and the next level left - right side node of the previous reconstructed node contains multiple record references* **then**

> **BplusTree_ReplaceLeftNodeRecursive()** or
> **BplusTree_ReplaceRightNodeRecursive()**
> **Return** void item.

end

if *The current upper level - parent node contains multiple record references and the next level left - right side node of the previous reconstructed node contains a single record reference* **then**

> **BplusTree_ReplaceSingleLeftNodeRecursive()** or
> **BplusTree_ReplaceSingleRightNodeRecursive()**
> **Return** void item.

end

if *The current upper level - parent node contains multiple record references and the next level left - right side node of the previous reconstructed node contains multiple record references* **then**

> **BplusTree_ReplaceMultipleLeftNodeRecursive()** or
> **BplusTree_ReplaceMultipleRightNodeRecursive()**
> **Return** void item.

end

The total average algorithmic operations - steps of the deletion function can be approximately be approached based on the average B$^+$-tree structure height. Consequently, the theoretical average time complexity of the deletion function in Alg. 16 – 19 can be approximately calculated by the relation 3.9:

$$O(\ 1 + log_{(2ku_k+1)}(d_{ln} - 1)\ ) \tag{3.9}$$

Chapter 4

B-Hash and B$^+$-Hash Map index structures

4.1 B-Hash and B$^+$-Hash Map indexes structural properties and characteristics

The implemented B-Hash and B$^+$-Hash Map index data structures compose the RDBMS file system in-memory and on-disk sub-systems tree hash structures simulation that are designed for indexing of relational database tables. Furthermore the developed B-Hash and B$^+$-Hash Map index structures are based on the existing RDBMSs on disk and in memory file systems indexing structures architecture and functionality, the fundamental hashing theory in [40] [10] [12] [49] [50] [51] [52] [53] [13] and on the implemented B-tree and B$^+$-tree index structures.

The implemented B-Hash - B$^+$-Hash Map index structure consists of a dynamic array data structure that each individual array node contains a B-tree - B$^+$-tree index structure that is linked to a RDBMS table and stores a record references subset of that table. The B-tree and B$^+$-tree structures set of the B-Hash Map - B$^+$-Hash index stores the total record references of the relational database table to which the tree hash map index structure is connected. Consequently the implemented B-Hash - B$^+$-Hash Map index constitutes a classic hash map index structure of B-tree - B$^+$-tree index structures that store all the record references of the relational table records . Furthermore the hash index is functionally and structurally developed, implemented and designed as a RAM - main memory system index structure and not as a disk (non volatile) based memory system indexing structure.

Fig. 4.1 and 4.2 represent the B-Hash and B$^+$-Hash Map index structures architecture and structural design.

Figure 4.1: B-Hash Map index structure architecture

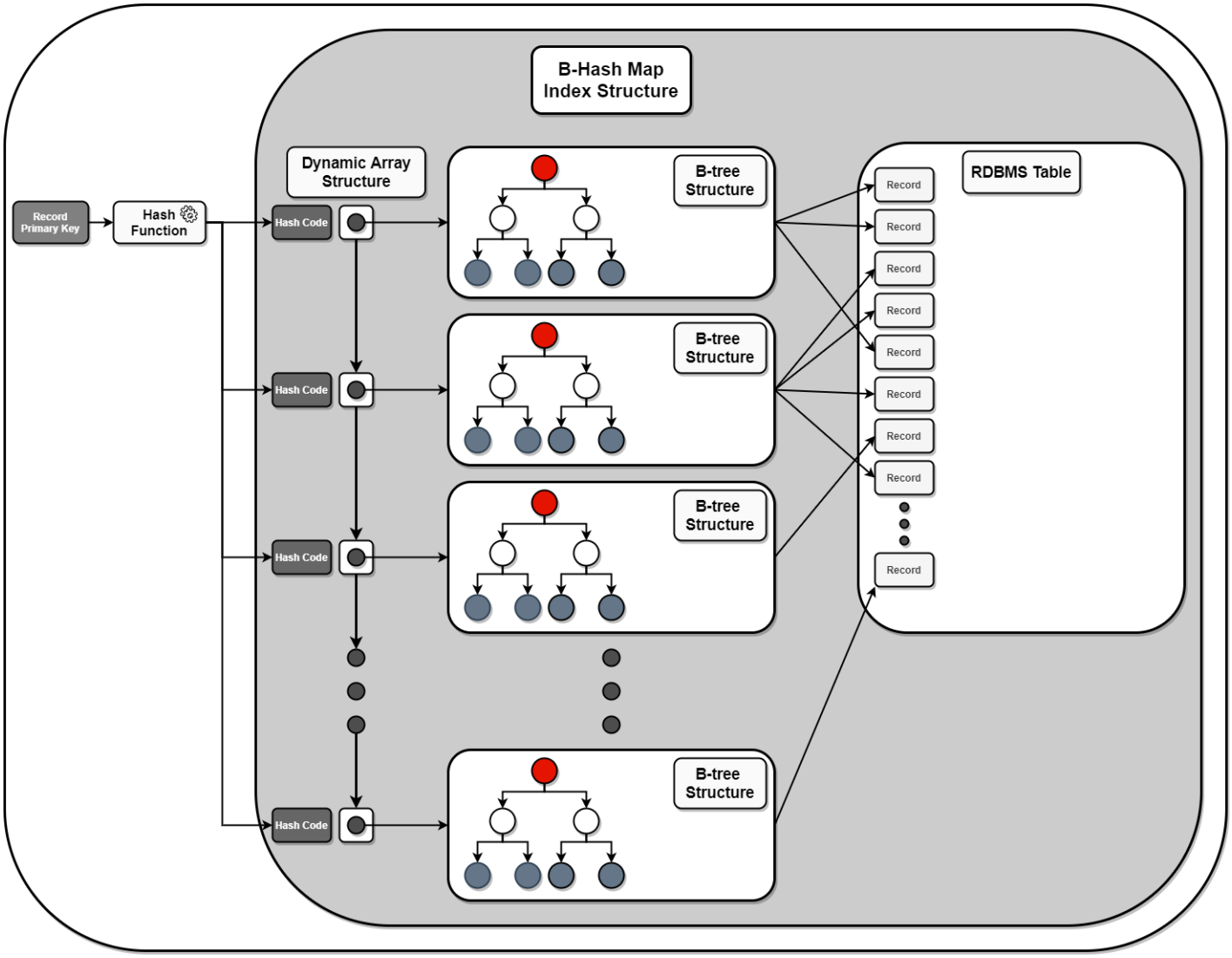
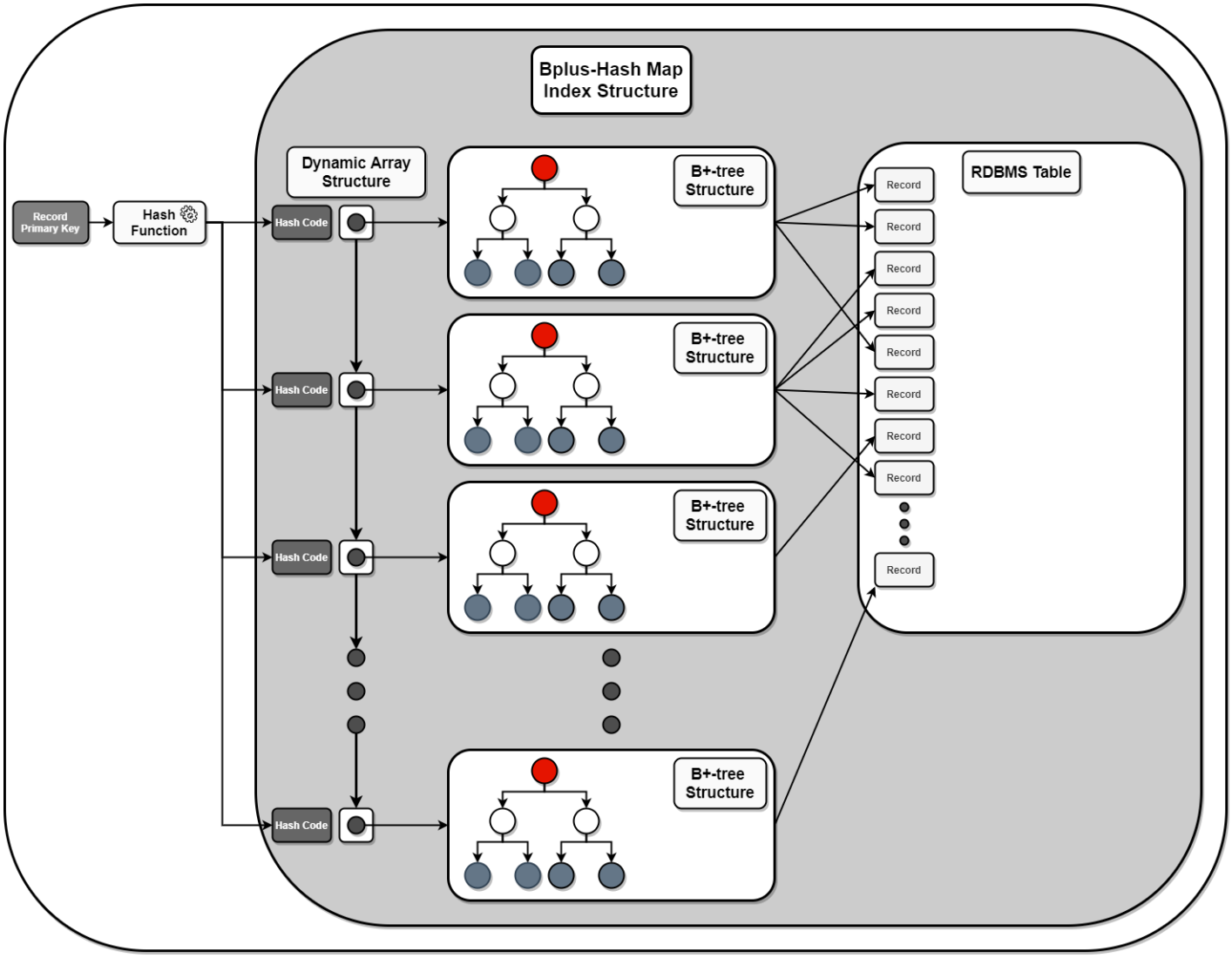


Figure 4.2: B⁺-Hash Map index structure architecture



4.2 B-Hash and B$^+$-Hash Map index structures basic functional levels

The basic implemented indexing B-Hash - B$^+$-Hash Map structure functions that are analyzed and developed in this work are the record references insertion, deletion and selection functions. The algorithmic functions, analyzed and developed in the context of the implemented B-Hash and B$^+$-Hash Map index structures, are:

- Records selection by record primary key field.
- Records selection by multiple record fields.
- Records insertion and deletion based on the record primary key field.

The B-Hash and B$^+$-Hash Map index structures are implemented to store the record references with primary keys of string and integer type. The functions of the B-Hash and B$^+$-Hash Map index structures that store record references with records string primary key fields utilize the Bernstein's DJB2 string hash function in order to create a hash code for each record primary key that will be stored in the tree hash indexes. The DJB2 string hash function is a quite efficient and effective string hash function that decreases and stabilize the B-Hash and B$^+$-Hash Map index structures functions collision. However, it is possible to be used any string hash function (which is suitable for this use) instead of the default DJB2 function for each case that requires special functionality and design. Furthermore the B-Hash and B$^+$-Hash Map index structures that are structurally and functionally based on the integer type primary key use the simple hash function that creates a numeric hash code from the division modulo of the integer record primary key and the hash map array structures nodes set (set of B-tree - B$^+$-tree index structures).

4.2.1 Records insertion based on primary key fields

The functions in Alg. 20 and 21 implement the record insertion - storage in the B-Hash and B$^+$-Hash Map index structures based on the primary key hash functions and using the B-tree and B$^+$-tree index structures insertion sub-functions in Alg. 6 and 13. Fig. 4.3 and 4.4 represent the B-Hash and B$^+$-Hash Map index structures insertion functions operation.

Figure 4.3: B-Hash Map index structure record insertion function

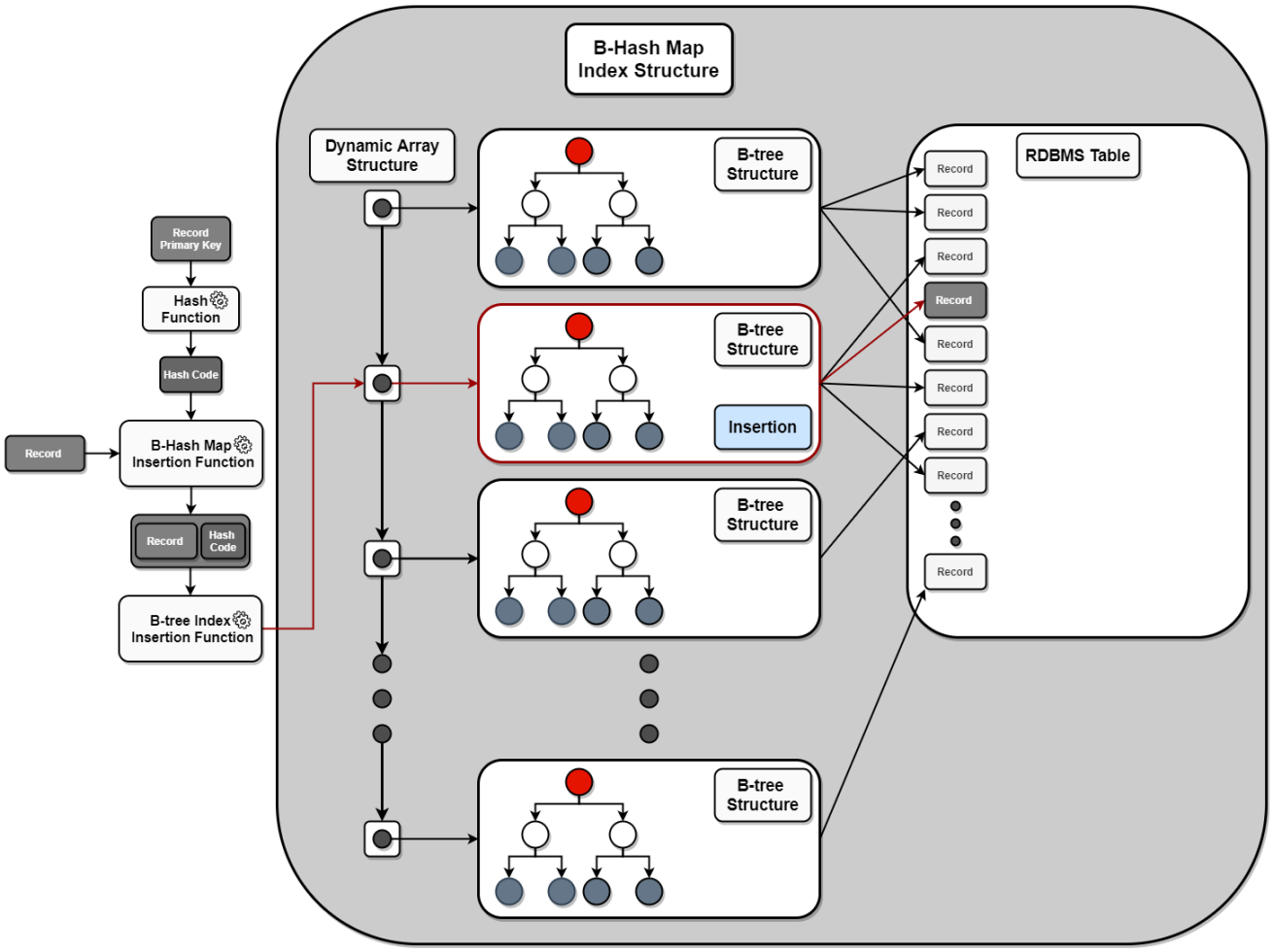
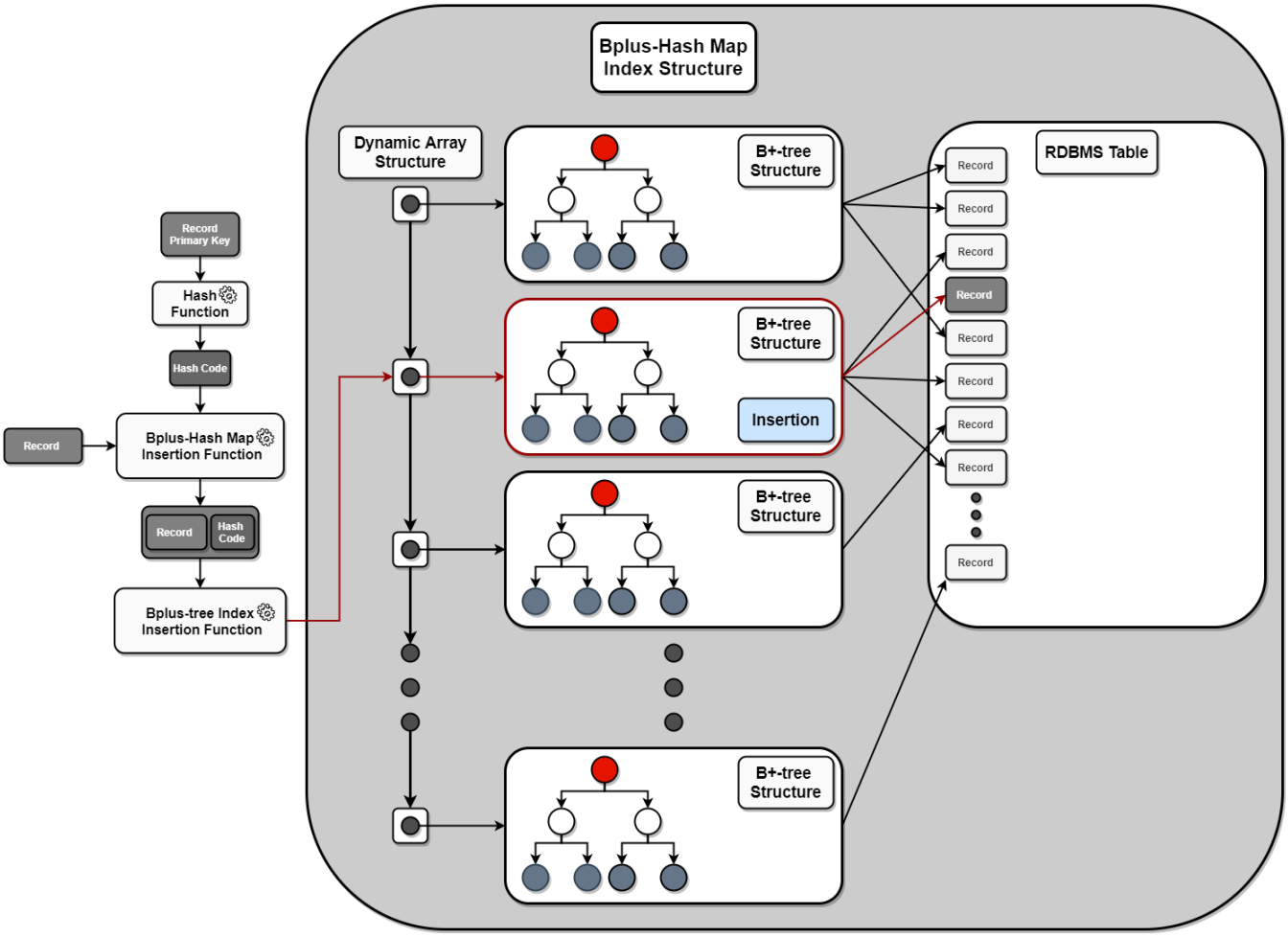


Figure 4.4: B⁺-Hash Map index structure record insertion function



Algorithm 20: BHashMapInsertData function

Returned item: Insertion process status
BHashMapInsertData(
B-Hash Map index structure item,
Record reference - data to be inserted,
Hash code of the record primary key
)

if *B-Hash Map index structure is not constructed and is uninitialized* **then**
    Return unsuccessful insertion status.
end

BTreeInsertData()
if *B-tree insertion was completed successfully* **then**
    Return successful insertion status.
end

Return unsuccessful insertion status.

Algorithm 21: BplusHashMapInsertData function

Returned item: Insertion process status
BplusHashMapInsertData(
B^+-Hash Map index structure item,
Record reference - data to be inserted,
Hash code of the record primary key
)

if *B^+-Hash Map index structure is not constructed and is uninitialized* **then**
    Return unsuccessful insertion status.
end

BplusTreeInsertData()
if *B^+-tree insertion was completed successfully* **then**
    Return successful insertion status.
end

Return unsuccessful insertion status.

4.2.2 Records deletion based on primary key fields

Alg. 22 and 23 implement the record deletion - removal from the B-Hash and B$^+$-Hash Map index structures based on the primary key hash functions and using the B-tree and B$^+$-tree index structures deletion sub-functions in Alg. 9 and 16. Fig. 4.5 and 4.6 represent the B-Hash and B$^+$-Hash Map index structures deletion functions operation.

Figure 4.5: B-Hash Map index structure record deletion function

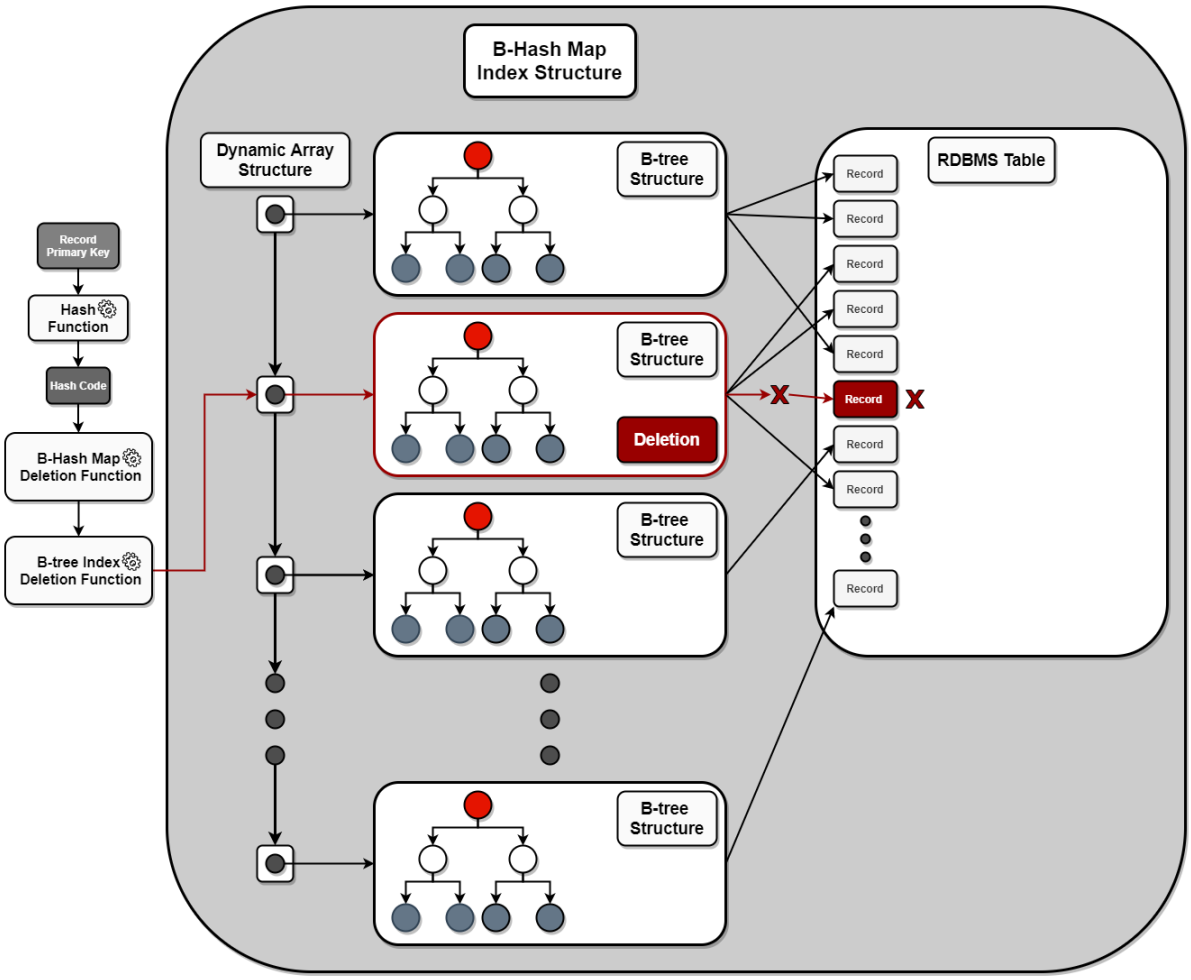
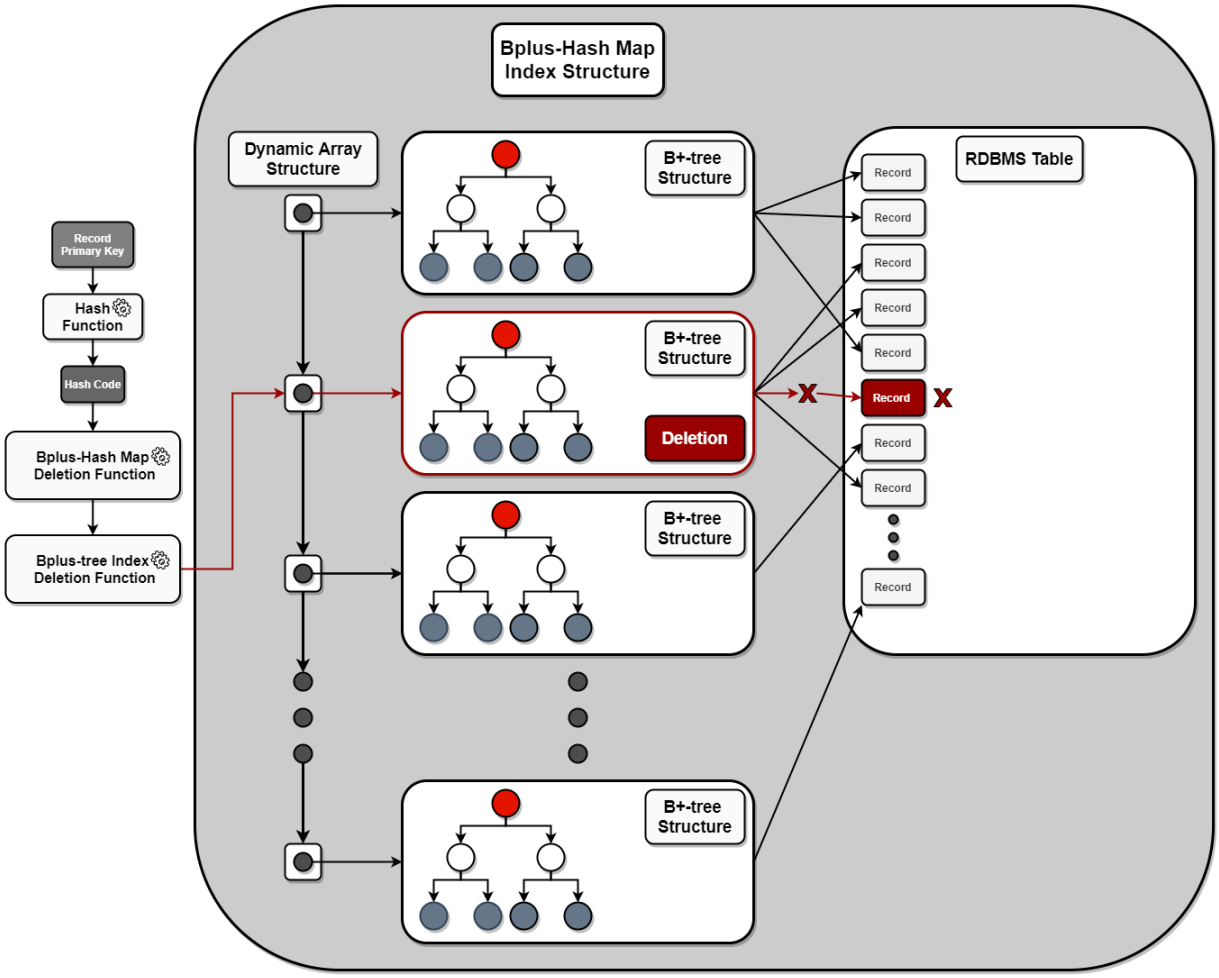


Figure 4.6: B⁺-Hash Map index structure record deletion function



Algorithm 22: BHashMapDeleteData function

Returned item: Deletion process status
BHashMapDeleteData(
B-Hash Map index structure item,
Record reference - data primary key to be deleted,
Hash code of the record primary key,
Deleted - removed record **)**

if *B-Hash Map index structure is not constructed and is uninitialized* **then**
  Return unsuccessful deletion status.
end
if *B-Hash Map is empty or B-Hash Map B-tree index structure that the deletion will be implemented has not stored records* **then**
  Return unsuccessful deletion status.
end
BTreeDeleteData()
if *B-tree deletion was completed successfully* **then**
  Return successful deletion status.
end
Return unsuccessful deletion status.

Algorithm 23: BplusHashMapDeleteData function

Returned item: Deletion process status
BplusHashMapDeleteData(
B^+-Hash Map index structure item,
Record reference - data primary key to be deleted,
Hash code of the record primary key,
Deleted - removed record
)

if *B^+-Hash Map index structure is not constructed and is uninitialized* **then**
  Return unsuccessful deletion status.
end
if *B^+-Hash Map is empty or B^+-Hash Map B^+-tree index structure that the deletion will be implemented has not stored records* **then**
  Return unsuccessful deletion status.
end
BplusTreeDeleteData()
if *B^+-tree deletion was completed successfully* **then**
  Return successful deletion status.
end
Return unsuccessful deletion status.

4.2.3 Records selection by the records primary key fields

Alg. 24 and 25 implement the record location - selection in the B-Hash and B$^+$-Hash Map index structures based on the primary key hash functions and using the B-tree and B$^+$-tree index structures record references selection by records primary key fields sub-functions. Fig. 4.7 and 4.8 represent the B-Hash and B$^+$-Hash Map index structures selection functions operation.

Figure 4.7: B-Hash Map index structure record selection function by primary key

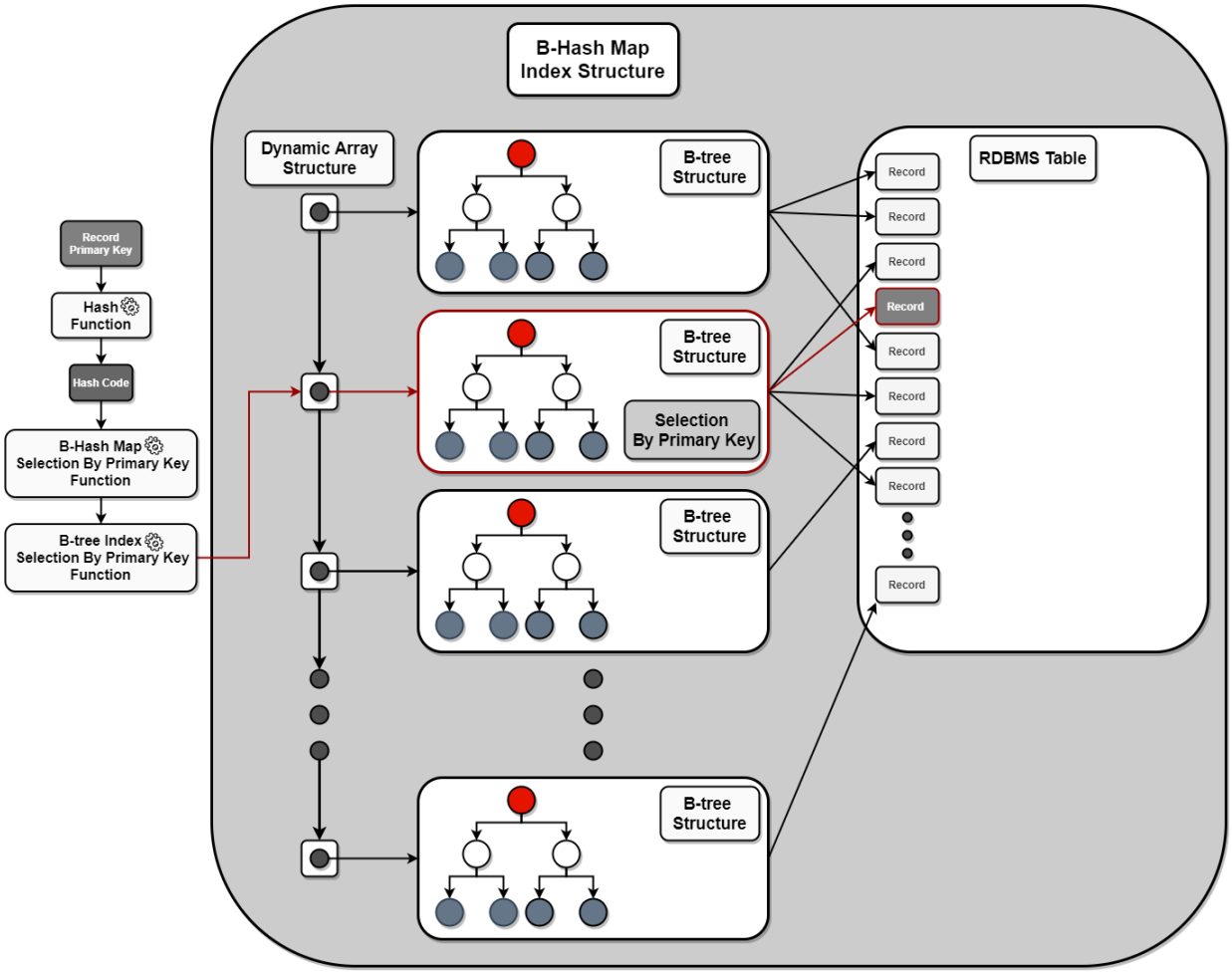
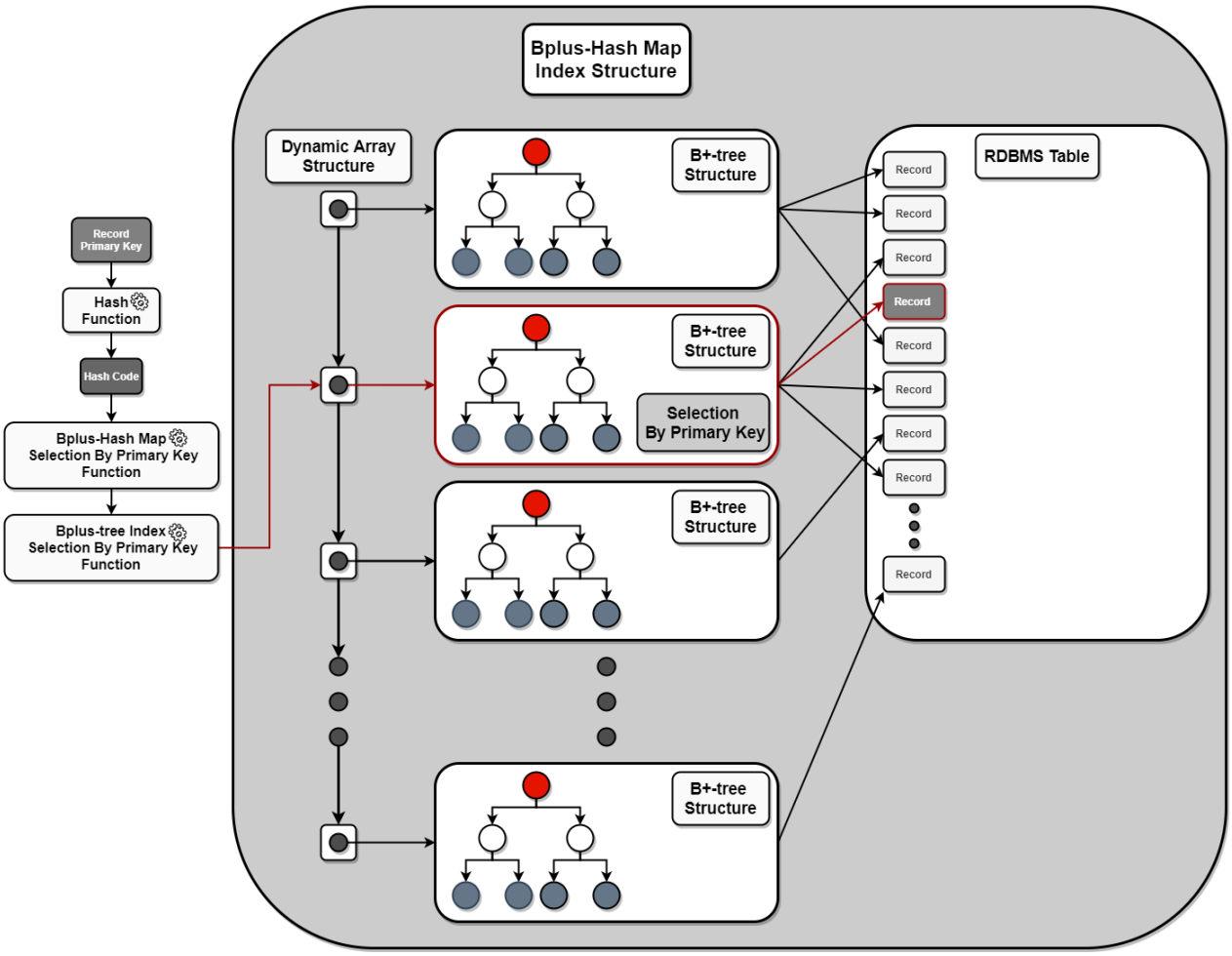


Figure 4.8: B⁺-Hash Map index structure record selection function by primary key



Algorithm 24: BHashMapSelectData_ByPrimaryKey function

Returned item: Selected record
BHashMapSelectData_ByPrimaryKey(
B-Hash Map index structure item,
Record reference - data primary key to be selected,
Hash code of the record primary key
)

if *B-Hash Map index structure is not constructed and is uninitialized* **then**
  Return null record item.
end
if *B-Hash Map is empty or B-Hash Map B-tree index structure that the selection will be implemented has not stored records* **then**
  Return null record item.
end
BTreeFastSearchData_ByPrimaryKey()
if *B-tree selection was completed successfully* **then**
  Return selected record.
end
Return null record item.

Algorithm 25: BplusHashMapSelectData_ByPrimaryKey function

Returned item: Selected record
BplusHashMapSelectData_ByPrimaryKey(
B^+-Hash Map index structure item,
Record reference - data primary key to be selected,
Hash code of the record primary key
)

if *B^+-Hash Map index structure is not constructed and is uninitialized* **then**
  Return null record item.
end
if *B^+-Hash Map is empty or B^+-Hash Map B^+-tree index structure that the selection will be implemented has not stored records* **then**
  Return null record item.
end
BplusTreeFastSearchData_ByPrimaryKey()
if *B^+-tree selection was completed successfully* **then**
  Return selected record.
end
Return null record item.

4.2.4 Records selection by multiple records fields (selection conditions)

Alg. 26 and 27 implement the record references set location - selection in the B-Hash and B$^+$-Hash Map index structures based on a selection constraints - conditions set using the B-tree and B$^+$-tree index structures selection sub-functions. Fig. 4.9 and 4.10 represent the B-Hash and B$^+$-Hash Map index structures full scan - selection functions operation.

Figure 4.9: B-Hash Map index structure records selection function by a selection constraints set

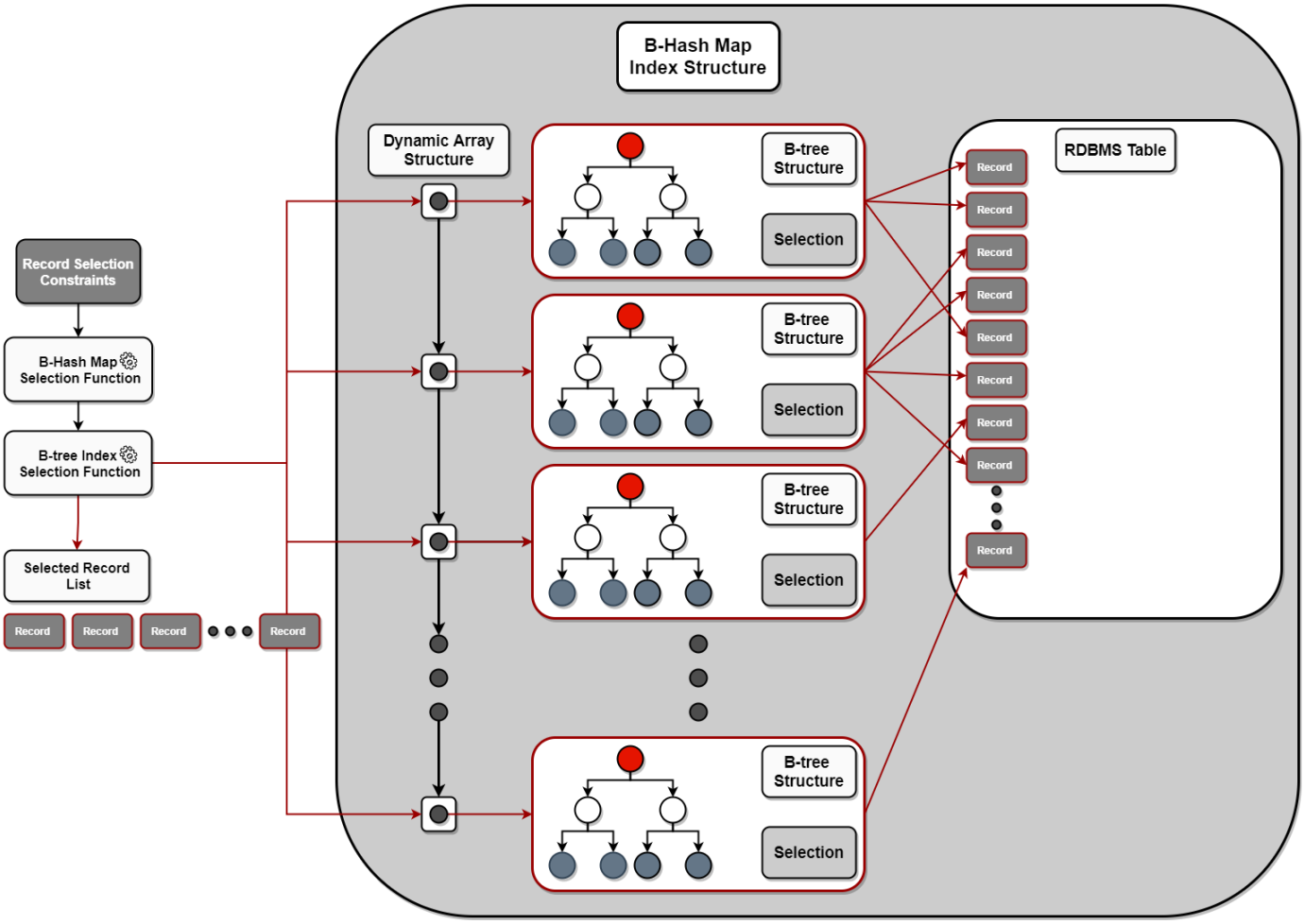
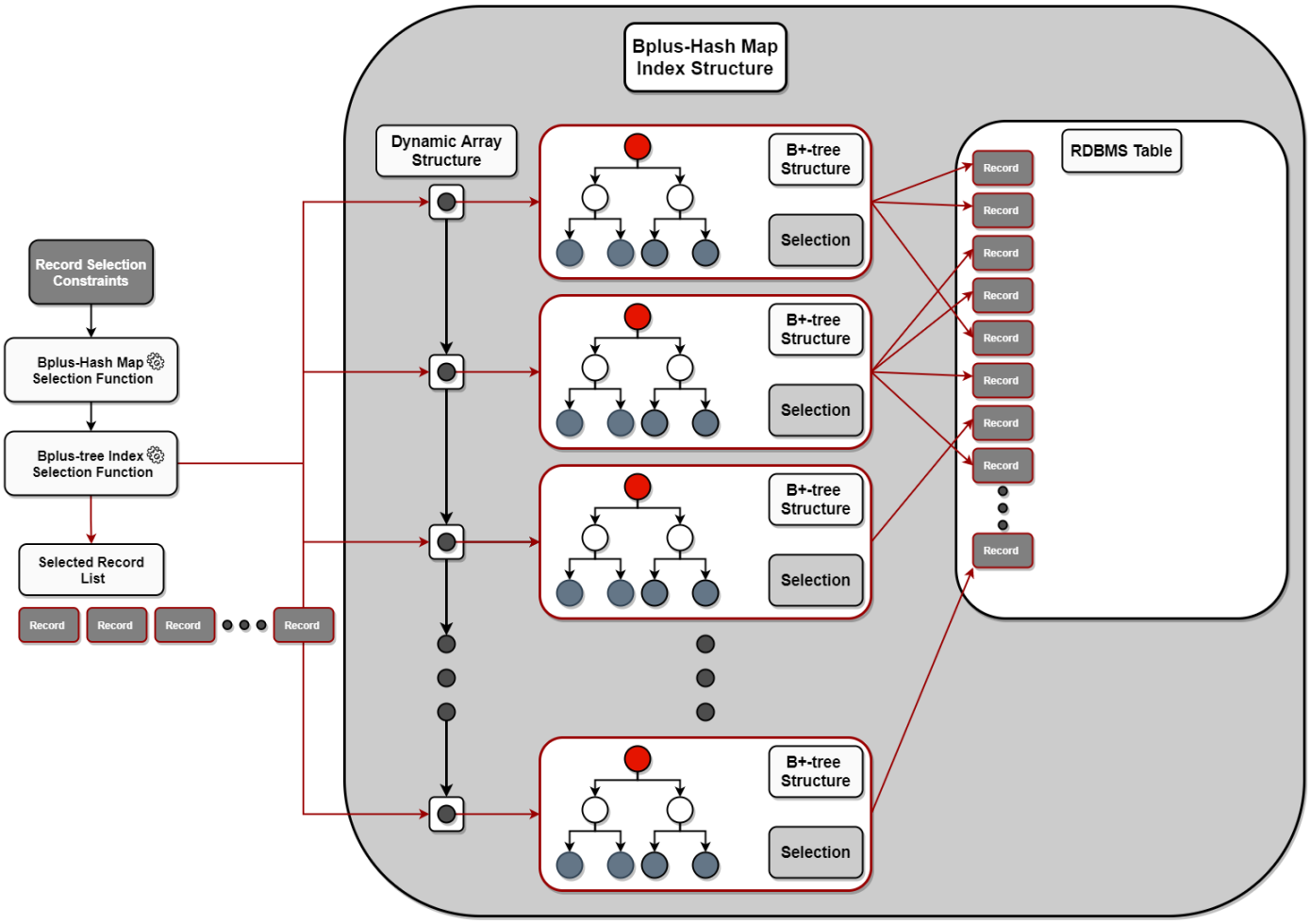


Figure 4.10: B⁺-Hash Map index structure records selection function by a selection constraints set



Algorithm 26: BHashMapSelectData function

Returned item: Selection process status
BHashMapSelectData(
B-Hash Map index structure item,
Double Linked List structure to store the selected records,
Selection constraint,
Selection method
)

if *B-Hash Map index structure is not constructed and is uninitialized* **then**
    Return unsuccessful selection status.
end

if *B-Hash Map is empty* **then**
    Return unsuccessful selection status.
end

Iterative scan - records selection of all B-Hash Map dynamic array structure
B-tree index structures based on the selection constraint.

BTreeSelectRecordData_ASC() or **BTreeSelectRecordData_DESC**
based on the selection method for each B-tree index structure selection process.

if *Double Linked List structure that stores the selected records is not empty* **then**
    Return successful selection status.
end

Return unsuccessful selection status.

Algorithm 27: BplusHashMapSelectData function

Returned item: Selection process status
BplusHashMapSelectData(
B$^+$-Hash Map index structure item,
Double Linked List structure to store the selected records,
Selection constraint,
Selection method
)

if *B$^+$-Hash Map index structure is not constructed and is uninitialized* **then**
  Return unsuccessful selection status.
end

if *B$^+$-Hash Map is empty* **then**
  Return unsuccessful selection status.
end

Iterative scan - records selection of all B$^+$-Hash Map dynamic array structure B$^+$-tree index structures based on the selection constraint.

BplusTreeSelectRecordData_ASC() or **BplusTreeSelectRecordData_DESC** based on the selection method for each B$^+$-tree index structure selection process.

if *Double Linked List structure that stores the selected records is not empty* **then**
  Return successful selection status.
end

Return unsuccessful selection status.

Each B-tree - B$^+$-tree index structure record reference insertion, deletion and selection function requires the load - transfer of the index structure nodes, stored record references and records data from the secondary storage disk system to the main memory system for processing applying a set of selection, insertion and deletion transactions on the disk. These read - write and delete disk operations are not efficient and are quite time consuming especially when a large B-tree - B$^+$-tree index structure part or the whole structure must be located and transferred from the disk memory system to the main memory in order to be implemented a set of transactions - table management functions. The B-Hash and B$^+$-Hash Map hybrid index tree hash structures use the basic hash map structure properties combined with the B-tree and B$^+$-tree index structures to reduce the disk access operations and increase the insertion, deletion and selection functions efficiency in terms of time performance and memory management (usage). This can be implemented through the records references and data storage - distribution in a set of B-tree - B$^+$-tree index structures of the hash index based on the records primary key hash codes that are created by the hash functions for each table record. Consequently each B-Hash and B$^+$-Hash Map insertion, deletion and selection function requires a quite smaller index structure (nodes and stored

references) and data part location and transfer from the disk to main memory structures of the RDBMS in memory file system in compare with the B-tree - B⁺-tree indexes. Thus avoiding the main memory - RAM overload and the memory leak and corruption that can be destructive for the data storage and maintenance. Therefore the B-Hash and B⁺-Hash Map index structures are more functionally efficient on disk-based and in-memory file systems related to the B-tree - B⁺-tree index structures due to the more efficient distribution of the stored record references sets in the B-tree and B⁺-tree index structures of the B-Hash and B⁺-Hash Map indexes. This reduces the real completion time of the B-Hash and B⁺-Hash Map index structures insertion, deletion, update and selection operations even though these structures functions have approximately quite similar (depending on the set of B-Hash and B⁺-Hash Map B-tree and B⁺-tree index structures) theoretical time complexity related to the B-tree and B⁺-tree index structures.

Considering that the B-Hash - B⁺-Hash Map index structure contains n stored records and the hash map dynamic array structure consists of b individual B-tree - B⁺-tree index structure nodes. Then each hash map dynamic array structure node B-tree - B⁺-tree index structure contains $n_b = \frac{n}{b}$ stored record references. This is the perfect records storage distribution in the B-tree - B⁺-tree index structure node of the B-Hash - B⁺-Hash Map index dynamic array structure. The records storage distribution is mainly based on the hash function that creates the hash codes for each record primary key (collision stabilization - balance) and on the record primary key type that is stored in the hash map structure array nodes. Therefore the perfect hashing - storage distribution is a structural state that can be approached in order to have the best possible B-Hash - B⁺-Hash Map index structure functionality. There is a large set of hash functions with different properties and functional effectiveness that are suitable in different problems.

The B-Hash and B⁺-Hash Map index structures insertion, deletion and selection functions theoretical best case (record references set perfect storage distribution in the B-Hash - B⁺-Hash Map B-tree and B⁺-tree index structures) average time complexity is approximately:

$$O(\ log_{(2ku_k+1)}(\frac{n}{b})\ ) \tag{4.1}$$

Chapter 5

Computational study

5.1 Development environment

The B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures and functions were developed and implemented in C programming language utilizing the CLion and Visual Studio IDEs. Furthermore a set of integration and unit tests is provided and applied to the index structures functions for the structures functionality test coverage and quality assurance combining them with the Valgrind Memcheck and Massif tools for inefficient and problematic memory usage - management detection.

The software packages of the developed and implemented B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures and a set of quality assurance unit, integration tests and testing tools are provided completely documented on **GitHub**.

5.2 Computational process

The computational study was performed as an analysis, evaluation and comparison of the B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures insertion, deletion and selection functions time performance through a set of computational experiments. The computational experiments were applied on constructed and real data.

Furthermore, the computational study has been performed on a 32-core Intel Xeon CPU E5-2630 v3 2.40GHz with 128 GB of main memory, a clock of 3.2 GHz, an L1 code cache of 32 KB per core, an L1 data cache of 32 KB per core, an L2 cache of 256 KB per core, an L3 cache of 20 MB and a memory bandwidth of 41.6 GB/s, running under Ubuntu 16.04.6 LTS.

5.2.1 Computational process on synthetic data

This computational process was conducted as an average time performance approximation of the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures insertion, deletion and selection functions execution time in a set of synthetic data.

Each individual synthetic dataset is an RDBMS table stored record that is structurally composed of a record primary key field and an auxiliary record field of integer type. The set of these $100,000$ records is stored in a dynamic array structure in an ascending sorted arraignment - order based on the record primary key field. Then a random reordering - rearrangement process of this input data set was repeatedly performed 100 times. Specifically each record that is stored in the dynamic array structure switches position with a randomly selected record that is stored in the array structure. This procedure was repeated 100 times for the whole stored records set rearrangement in order to randomly be constructed the set of $100,000$ records.

This $100,000$ records set references are stored in the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures that are linked to the RDBMS relational table that contains these records utilizing the record insertion function of each individual index structure. Then, the record references set insertion process time is measured and stored. In addition, the B-tree and B^+-tree index structures internal and leaf nodes, the record references that are stored in the leaf and internal nodes of the index structures and the structures height are measured and stored. The average nodes, nodes stored record references and height of the B-Hash and B^+-Hash Map B-tree and B^+-tree index structures are measured. After the insertion process completion, the record references set is selected from the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures based on the primary key field of each record using the selection function and the total selection process time is measured and stored. As all the record references are stored in the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures, the complete - full scan and selection process of all the stored record references of the index structures is preformed based on the auxiliary record field that is the same for all the records. The full selection process time is also measured and stored. Finally, a rearrangement - reconstruction process of the dynamic array structure records set is implemented. Then, the deletion of all record references and records that had been inserted in the structures was performed and the completion time of the overall deletion operation was measured and stored.

This computational process was repeated $10,000$ times for each node capacity of the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures in a capacity range from 11 to 501 stored record references with an increment factor of 10. For each node capacity, the average of $10,000$ measurements taken for each operation was calculated. The B-Hash and B^+-Hash Map index structures are composed of 10 B-tree and B^+-tree index structures. In addition, the total average execution time of each individual function was calculated.

The above set of computational - measurement experiments was implemented separately for the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures in discrete computational experiments. Each computation was implemented using both the records sets with records that contain primary key fields of integer and string type. The structures that are functionally and structurally based on an record integer primary key field use the interpolation search algorithm for the nodes stored records location. The structures that are based on a record string primary key field use the binary search algorithm for the nodes stored records location.

Computational study results of insertion and deletion functional processes average time performance

Fig. 5.1 represents the average time performance of the B-tree index structure insertion and deletion functional processes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.1: Functional process of B-tree index structure insertion and deletion average time performance

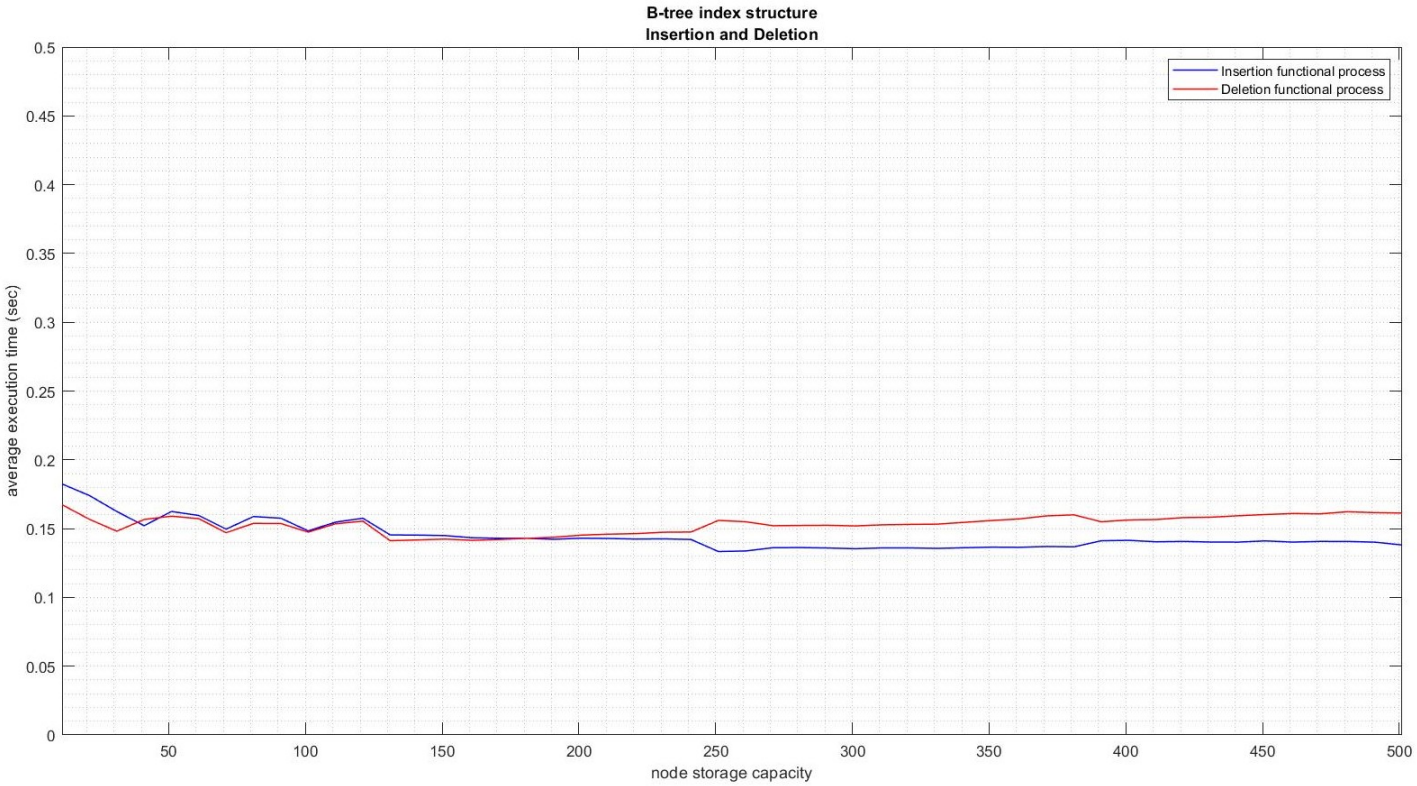


Fig. 5.2 represents the average time performance of the B-tree index structure insertion and deletion functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.2: Functional process of B-tree index structure insertion and deletion average time performance

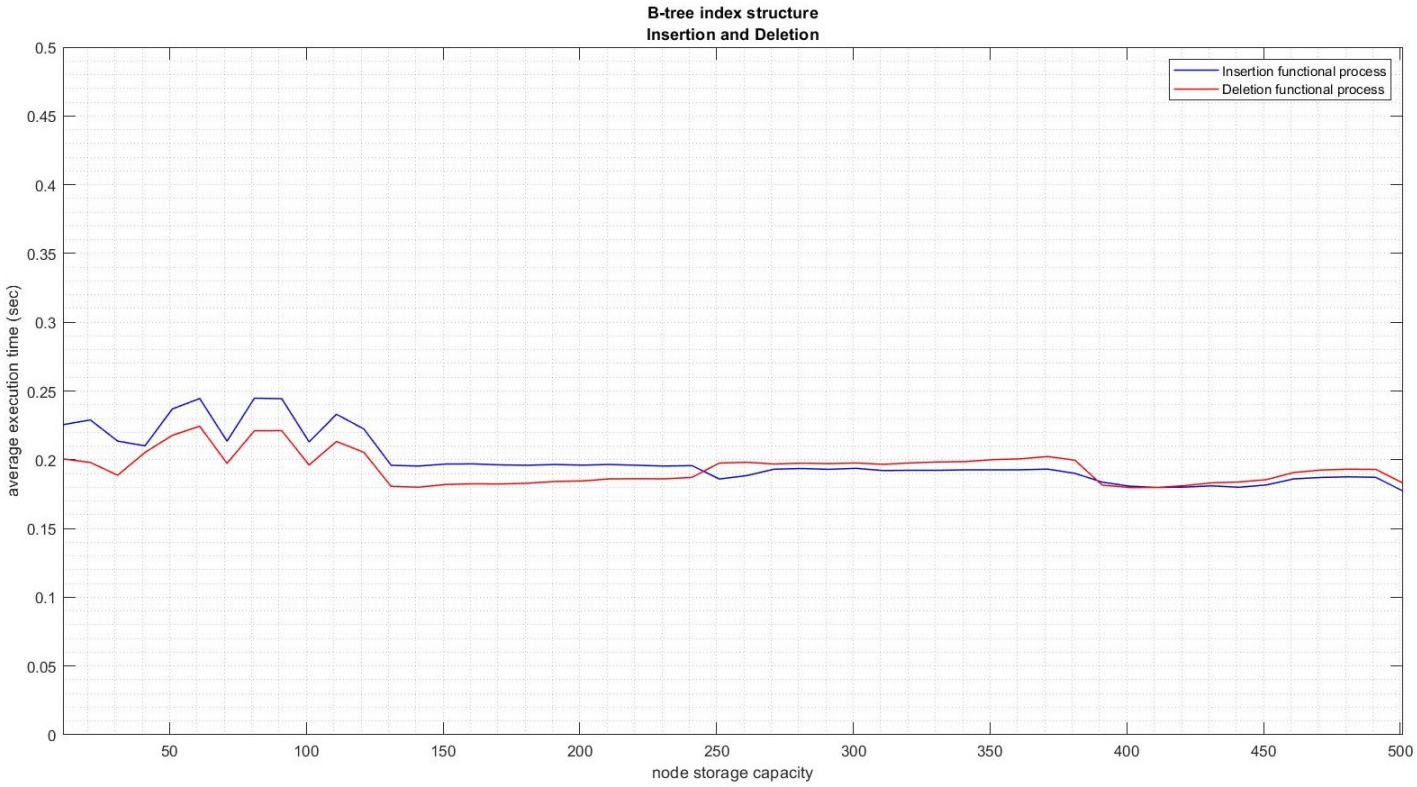


Fig. 5.3 represents the average time performance of the B$^+$-tree index structure insertion and deletion functional processes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.3: Functional process of B$^+$-tree index structure insertion and deletion average time performance

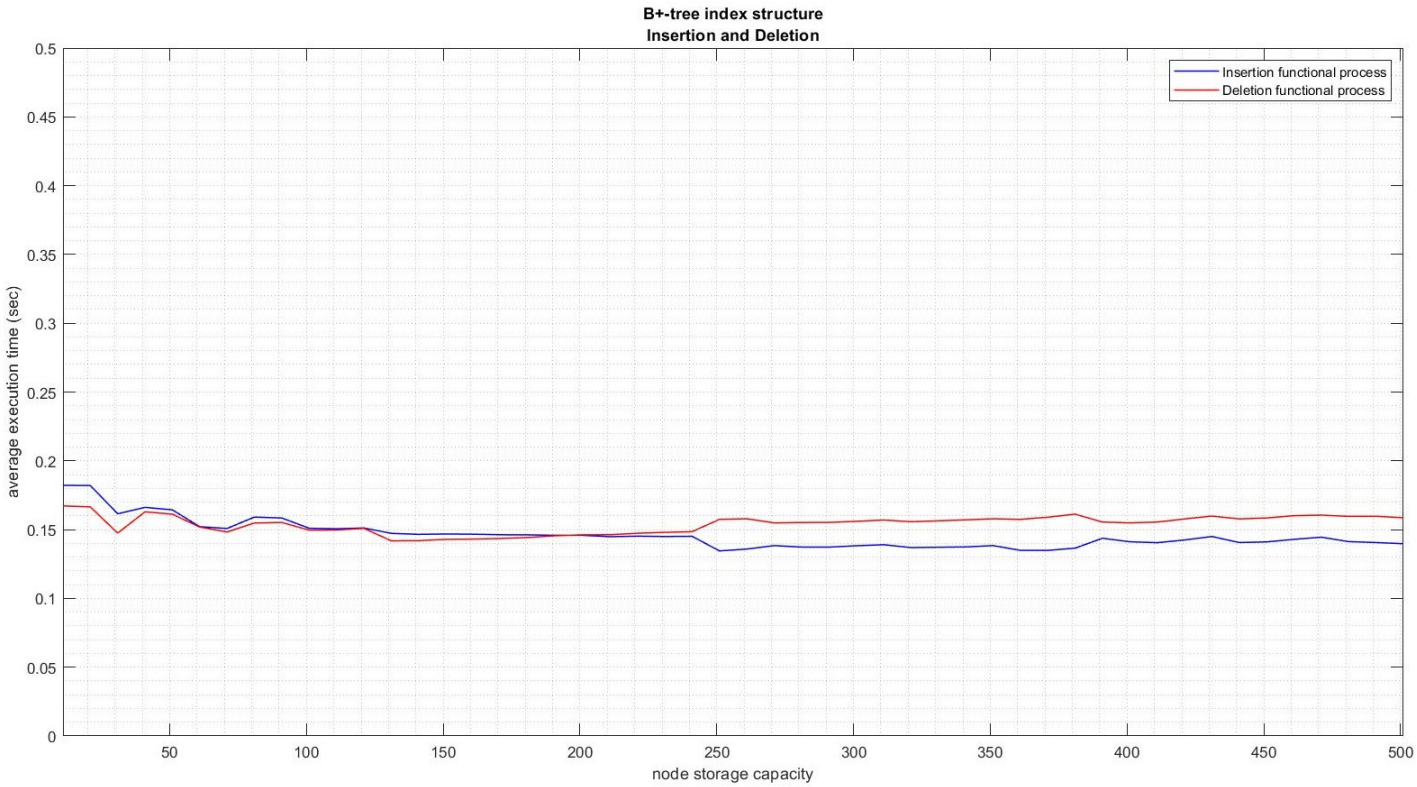


Fig. 5.4 represents the average time performance of the B$^+$-tree index structure insertion and deletion functional processes. The utilized records set consists of records with primary key fields of string type.

Figure 5.4: Functional process of B$^+$-tree index structure insertion and deletion average time performance

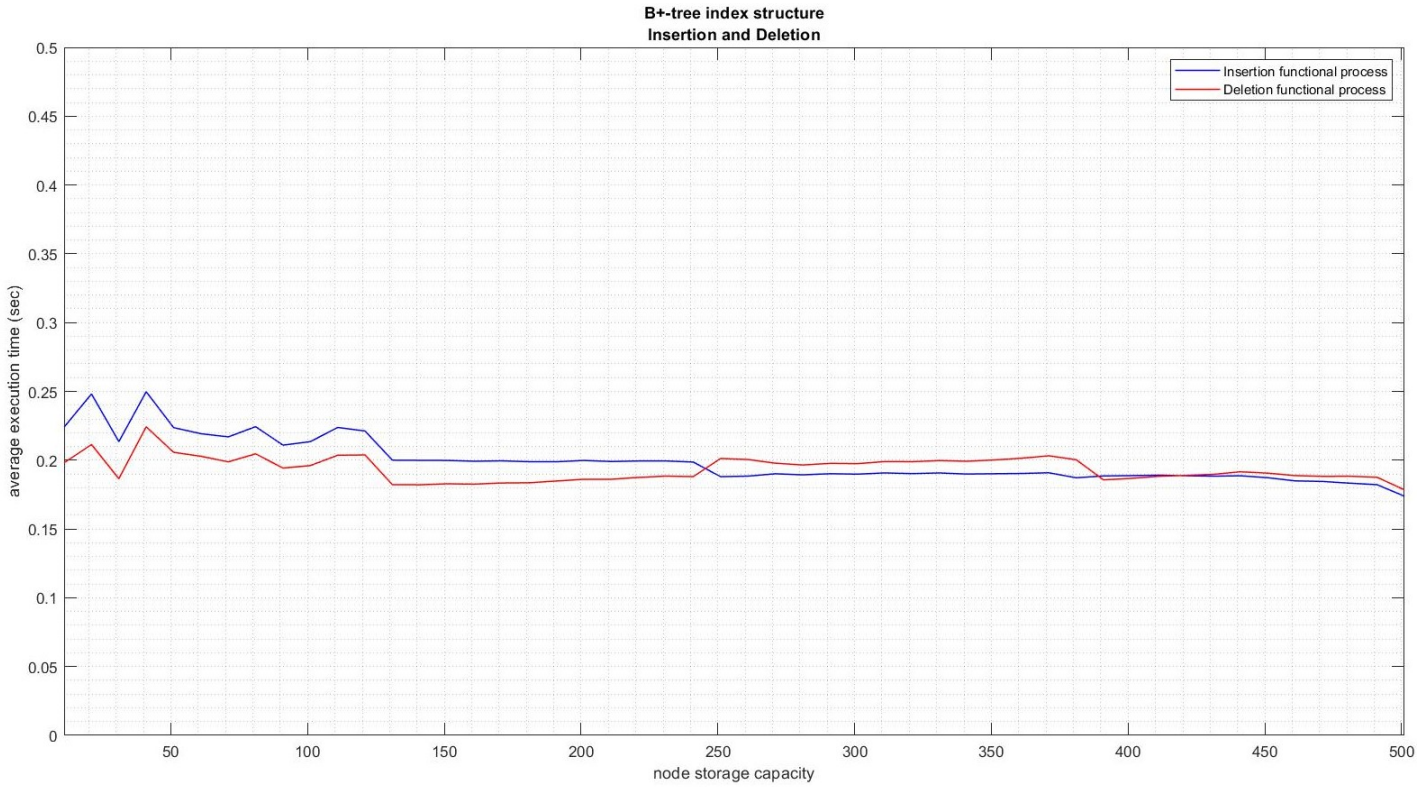


Fig. 5.5 represents the average time performance of the B-Hash Map index structure insertion and deletion functional processes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.5: Functional process of B-Hash Map index structure insertion and deletion average time performance

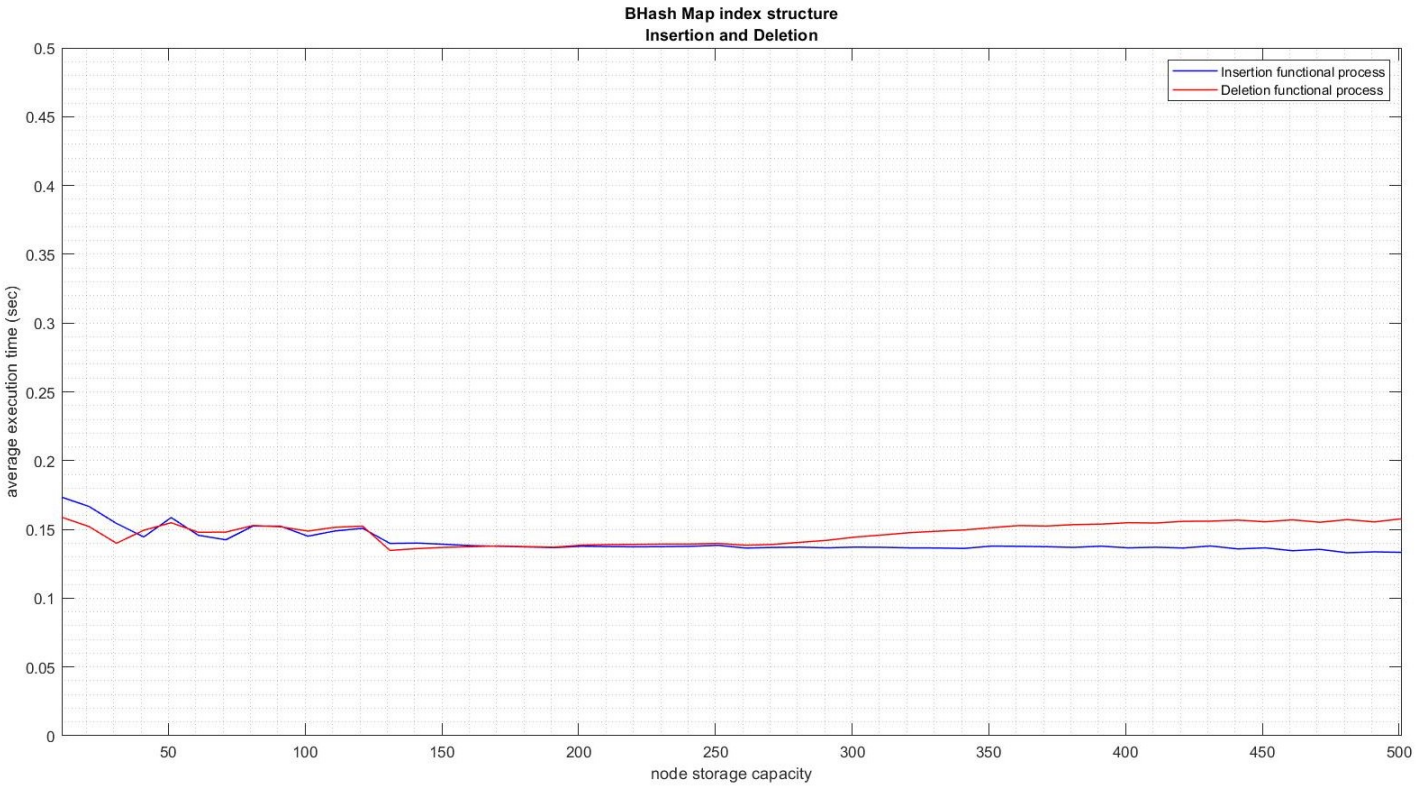


Fig. 5.6 represents the average time performance of the B-Hash Map index structure insertion and deletion functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.6: Functional process of B-Hash Map index structure insertion and deletion average time performance

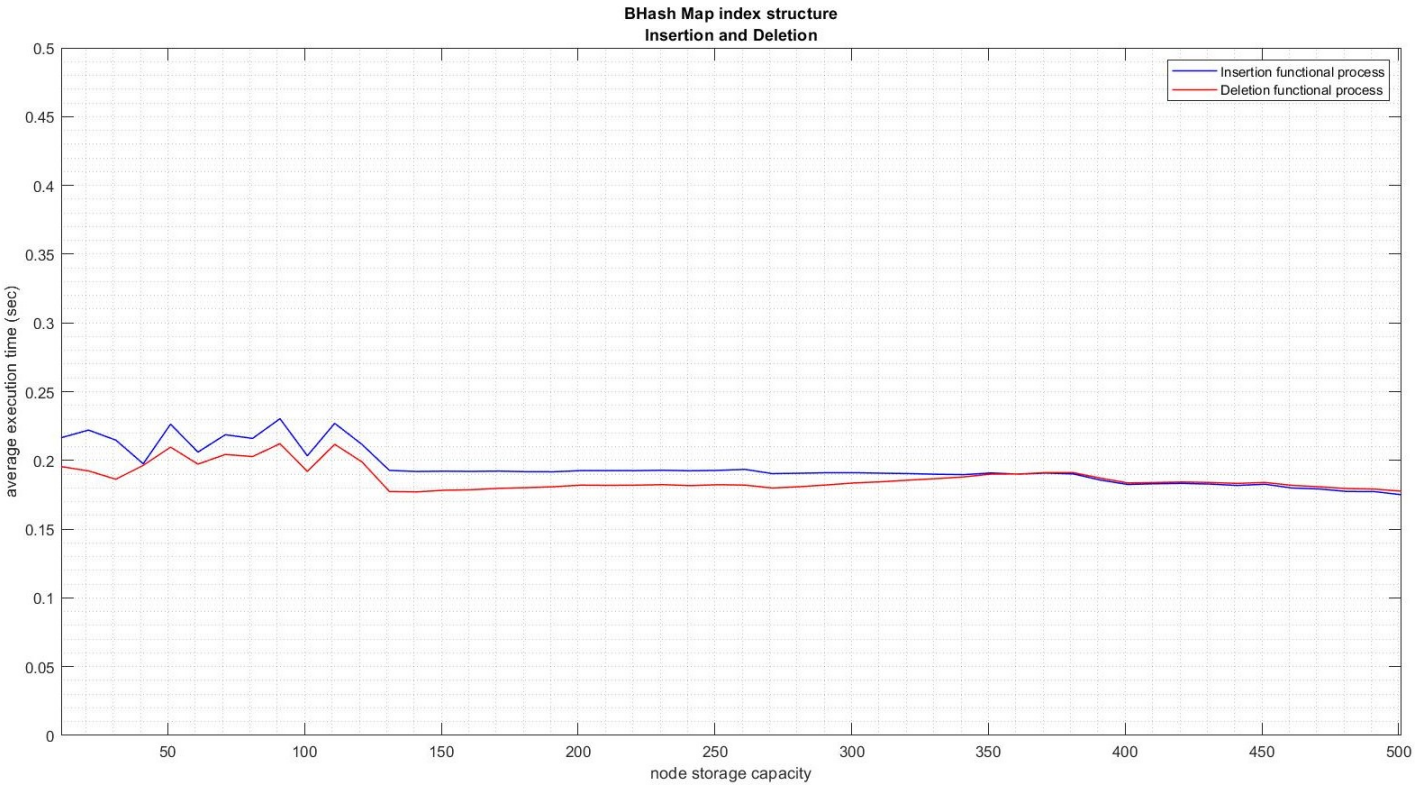


Fig. 5.7 represents the average time performance of the B$^+$-Hash Map index structure insertion and deletion functional processes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.7: Functional process of B$^+$-Hash Map index structure insertion and deletion average time performance

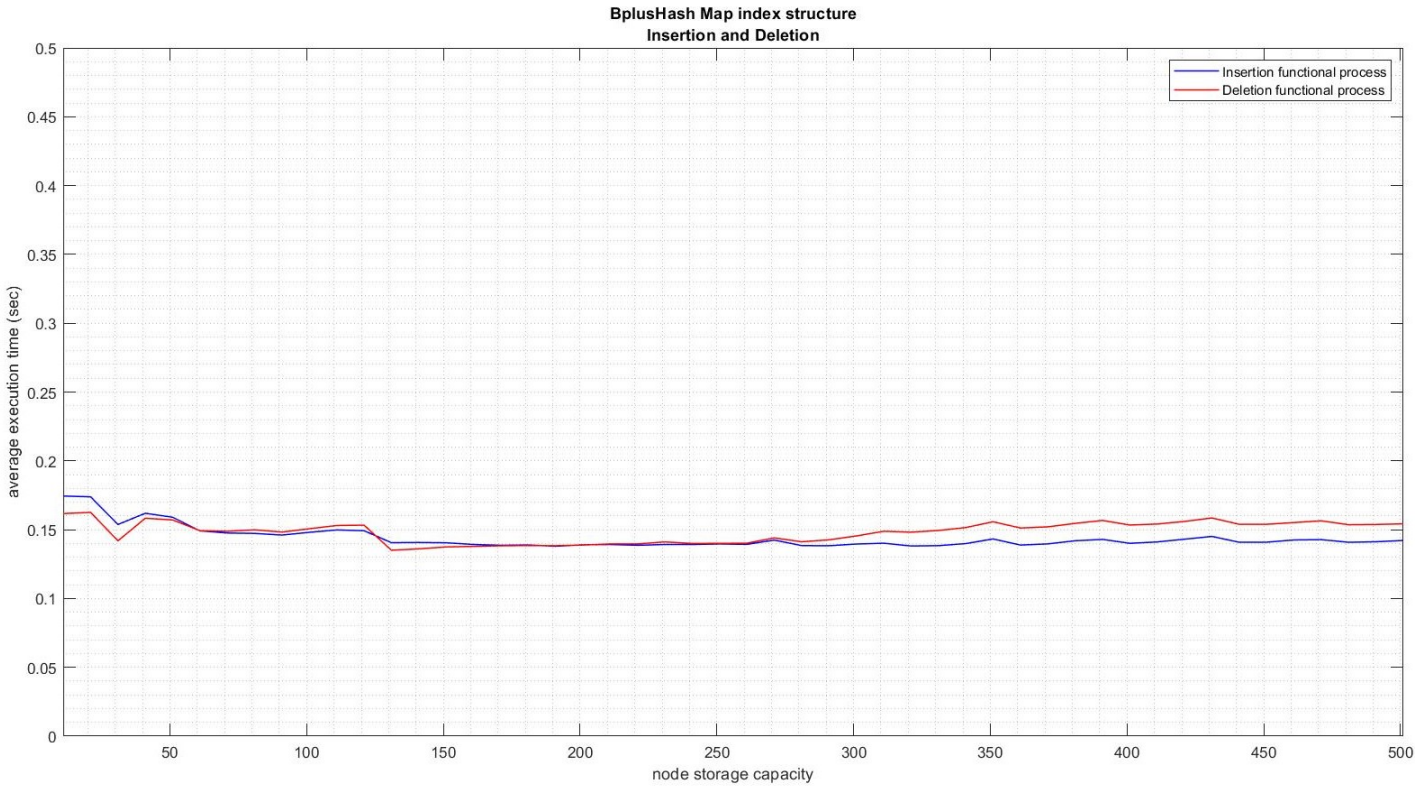


Fig. 5.8 represents the average time performance of the B⁺-Hash Map index structure insertion and deletion functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.8: Functional process of B⁺-Hash Map index structure insertion and deletion average time performance

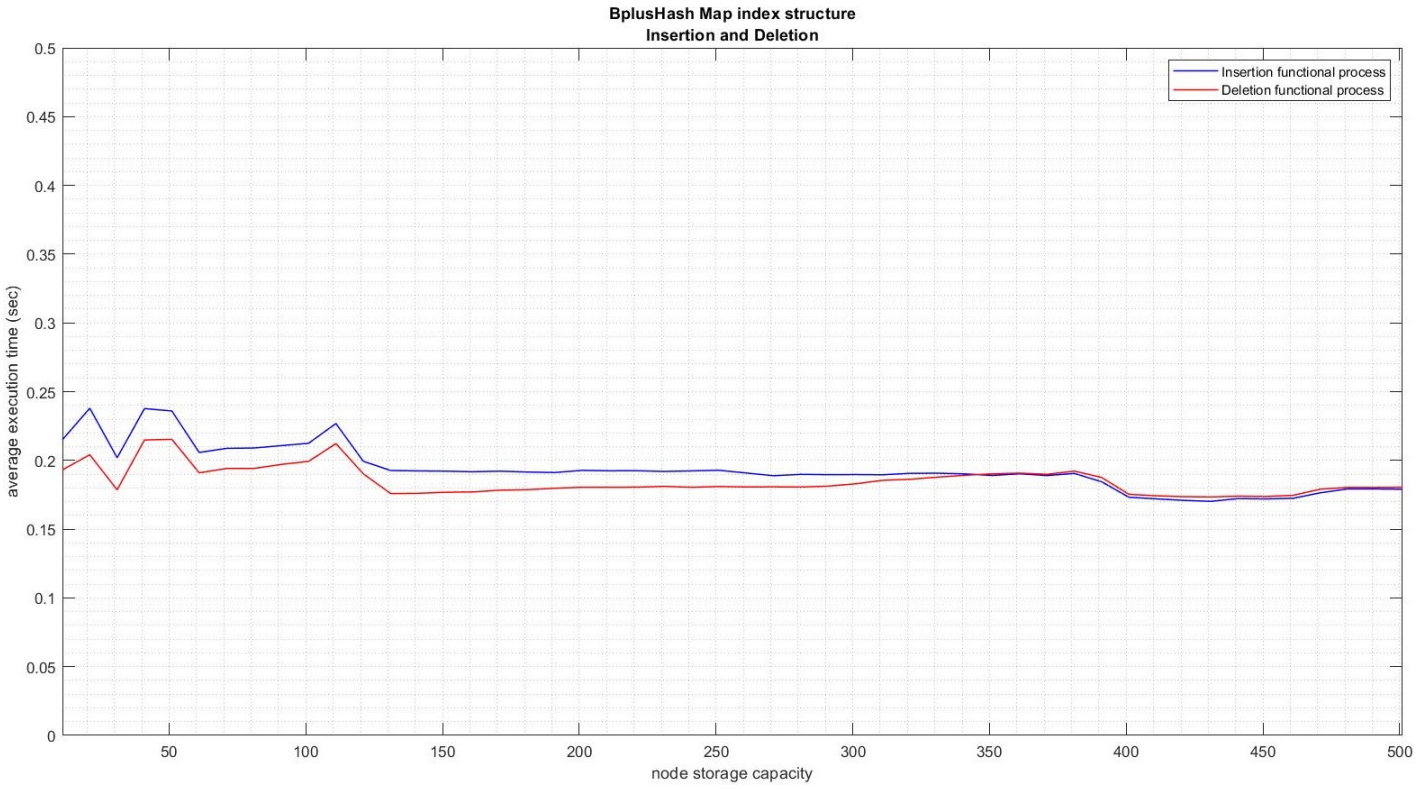


Fig. 5.9 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures insertion functional processes. The utilized records sets are composed of records with primary key fields of integer and string type.

Figure 5.9: Functional processes of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures insertion average time performance

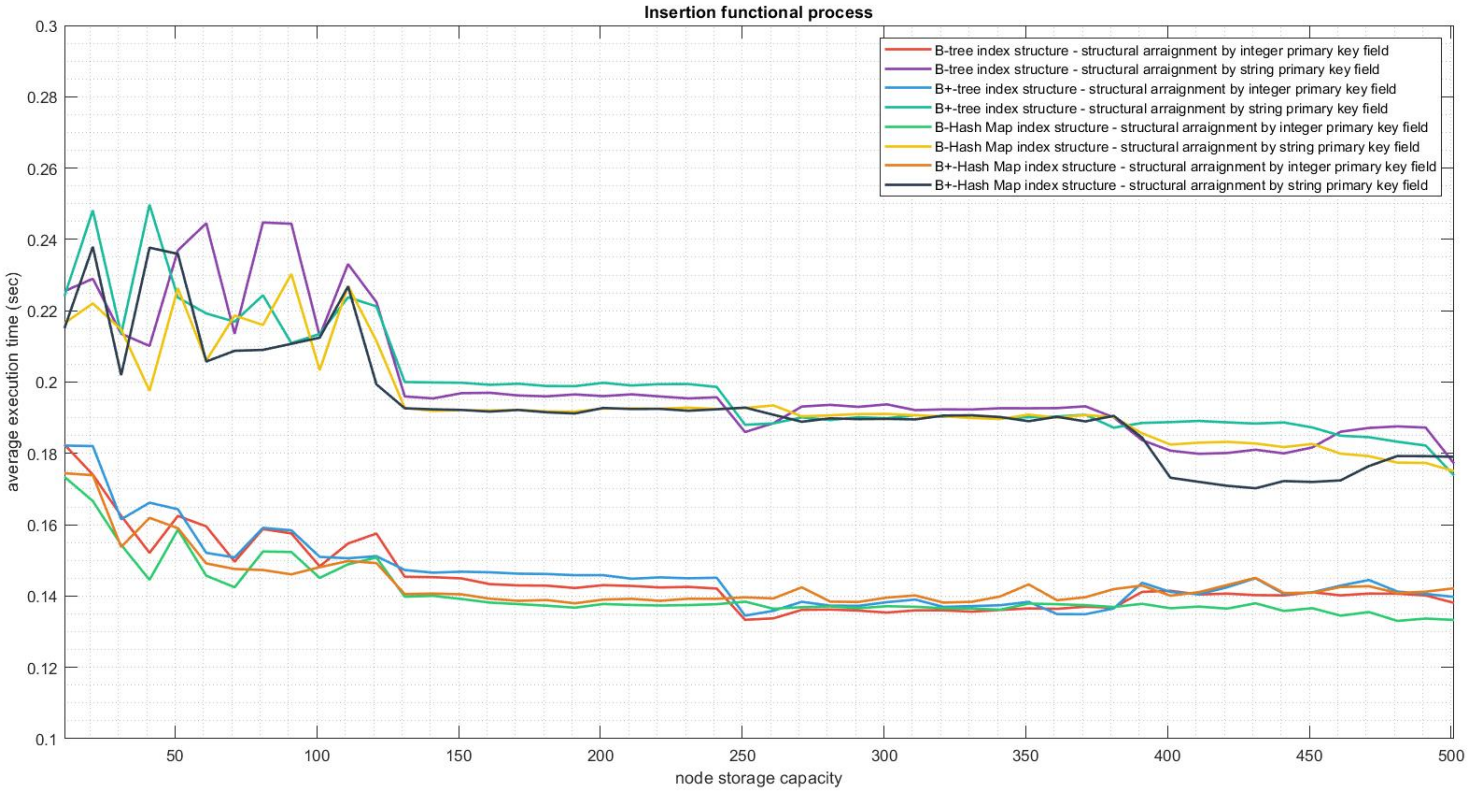
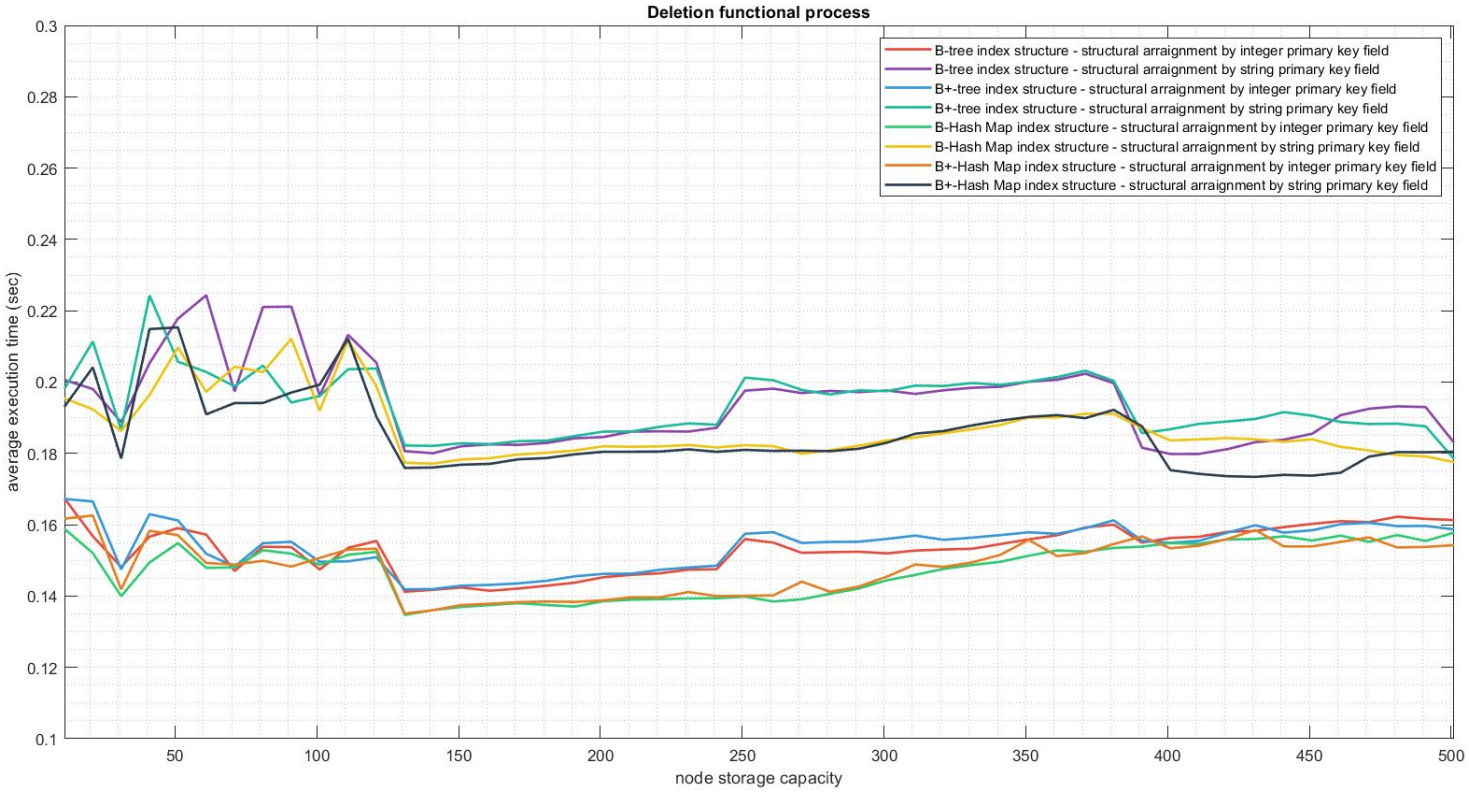


Fig. 5.10 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures deletion functional processes. The utilized records sets are composed of records with primary key fields of integer and string type.

Figure 5.10: Functional processes of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures deletion average time performance



Computational study results of selection by primary key field and full scan - selection functional processes average time performance

Fig. 5.11 represents the average time performance of the B-tree index structure selection by primary key field and full scan - selection functional processes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.11: Functional process of B-tree index structure selection by primary key field and full scan - selection average time performance

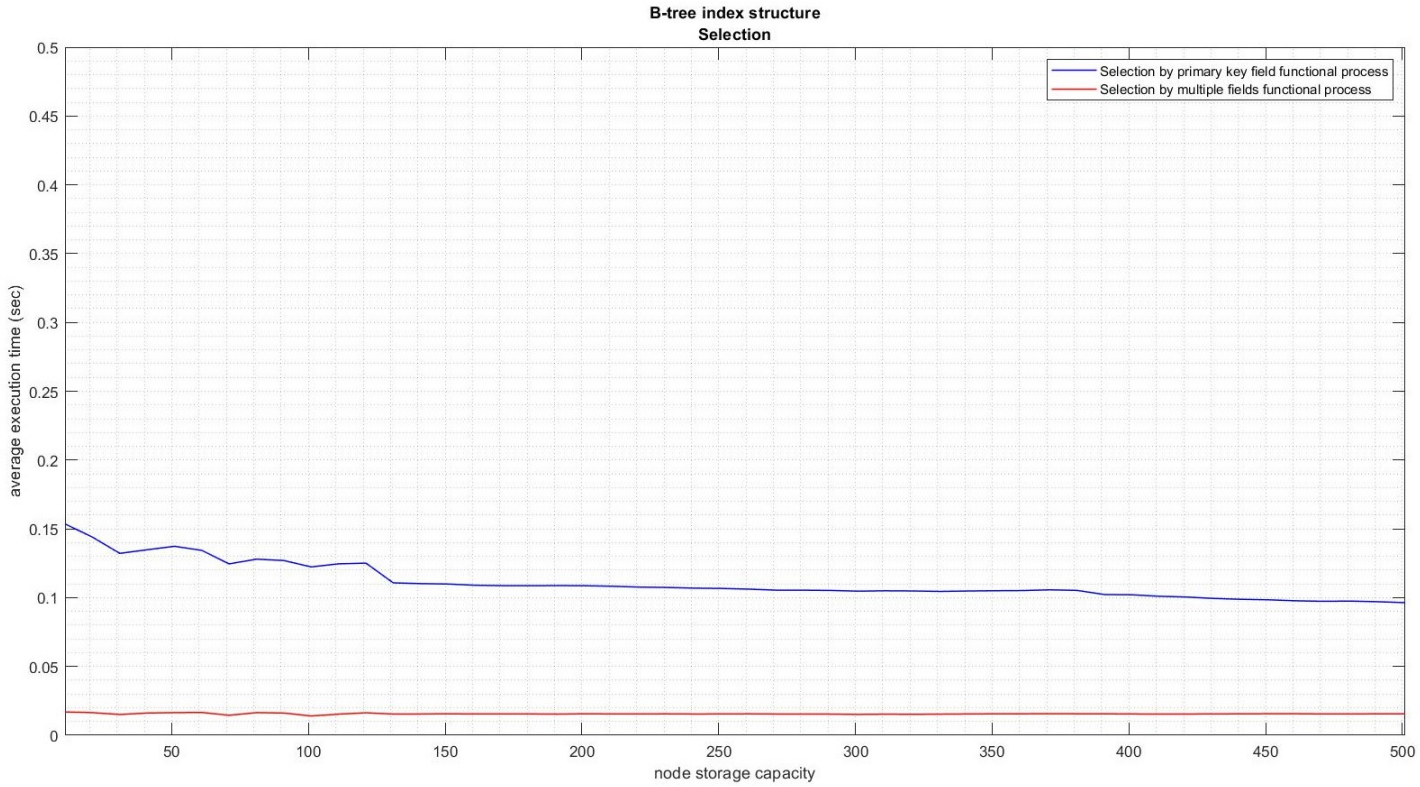


Fig. 5.12 represents the average time performance of the B-tree index structure selection by primary key field and full scan - selection functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.12: Functional process of B-tree index structure selection by primary key field and full scan - selection average time performance

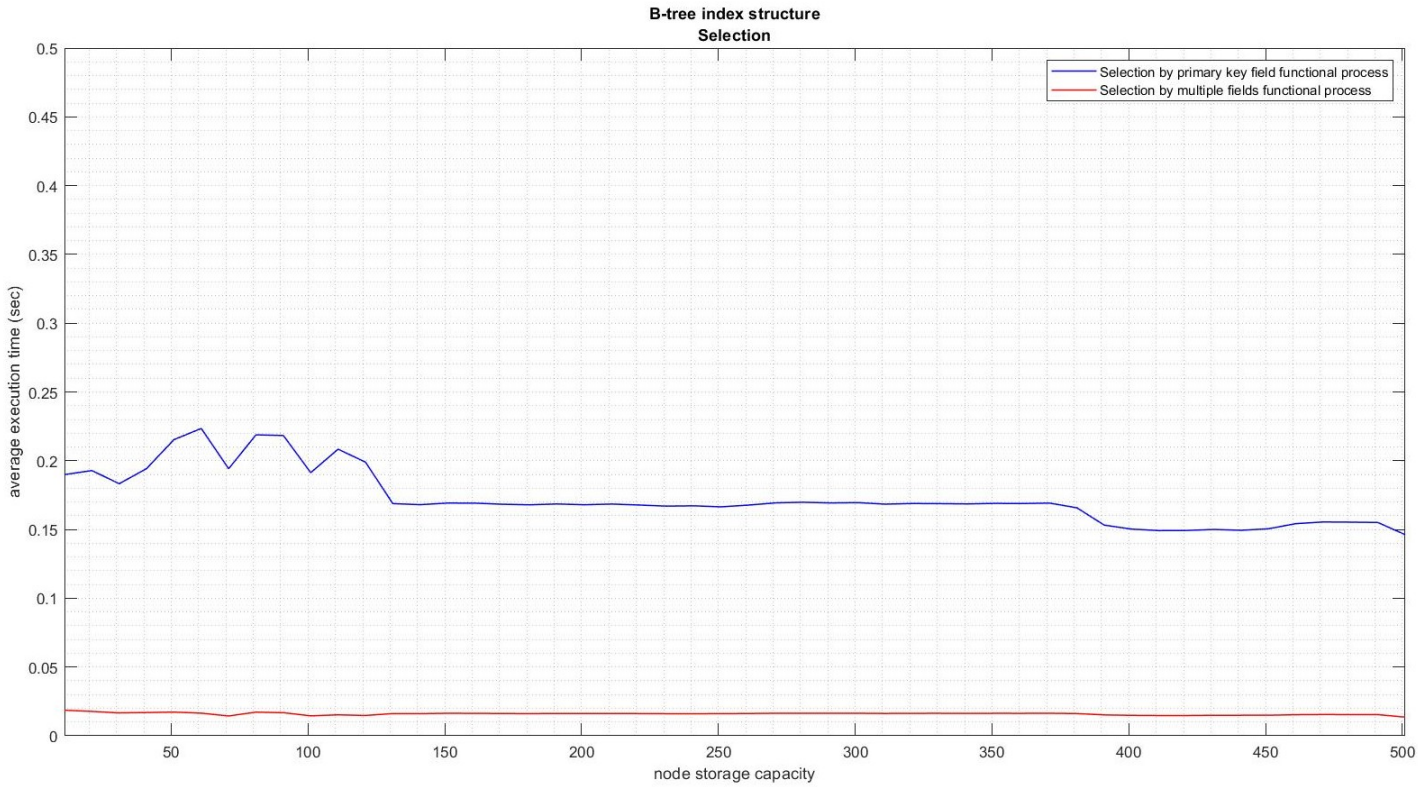


Fig. 5.13 represents the average time performance of the B$^+$-tree index structure selection by primary key field and full scan - selection functional processes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.13: Functional process of B$^+$-tree index structure selection by primary key field and full scan - selection average time performance

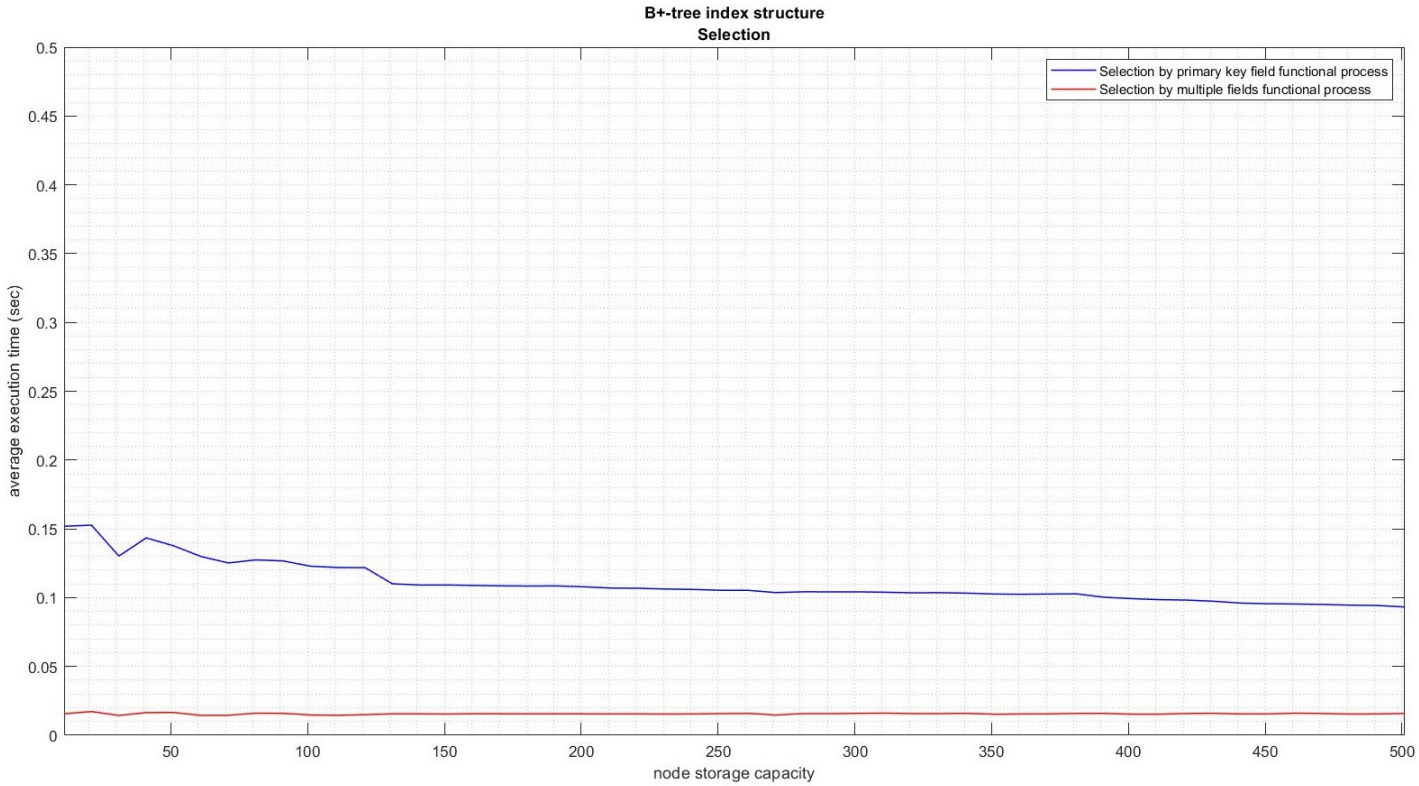


Fig. 5.14 represents the average time performance of the B$^+$-tree index structure selection by primary key field and full scan - selection functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.14: Functional process of B$^+$-tree index structure selection by primary key field and full scan - selection average time performance

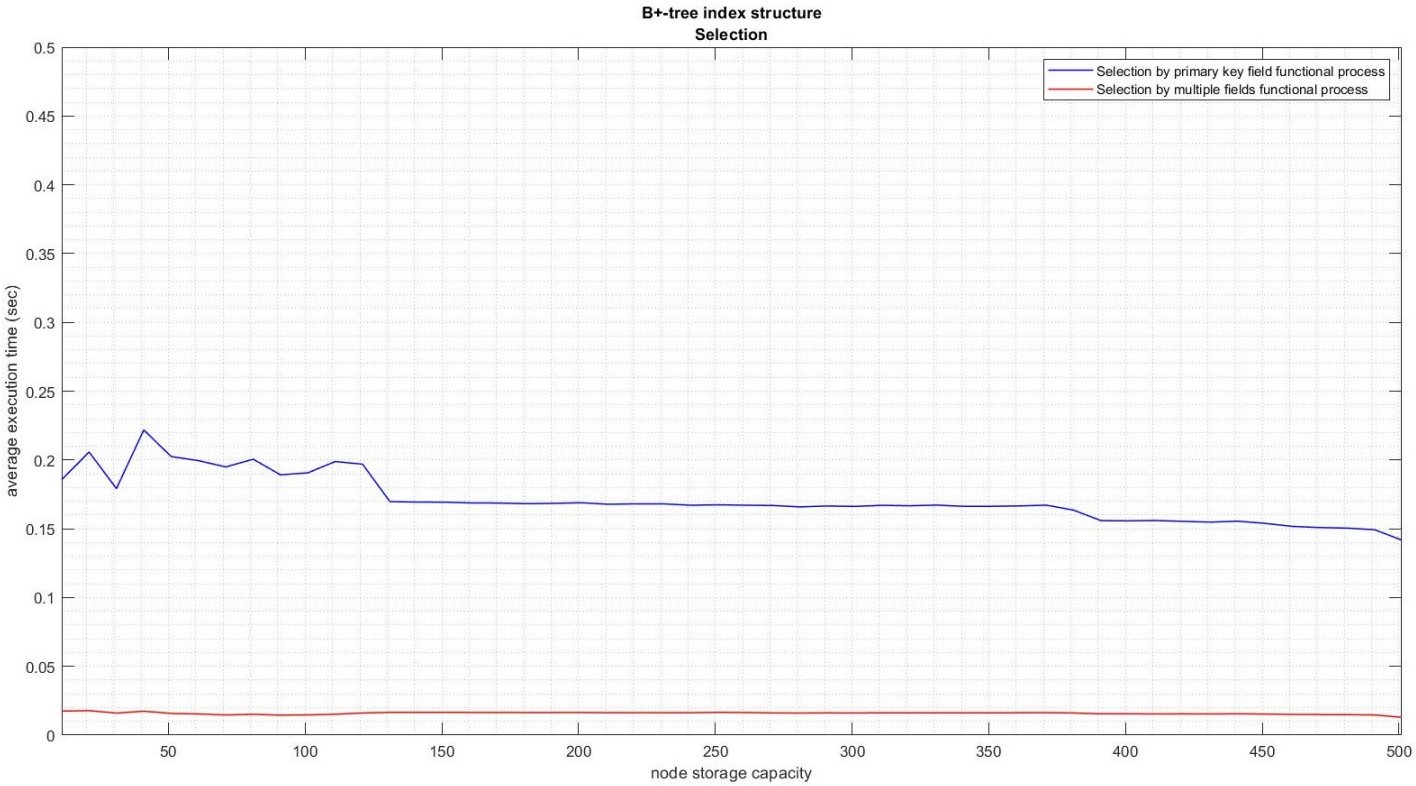


Fig. 5.16 represents the average time performance of the B-Hash Map index structure selection by primary key field and full scan - selection functional processes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.15: Functional process of B-Hash Map index structure selection by primary key field and full scan - selection average time performance

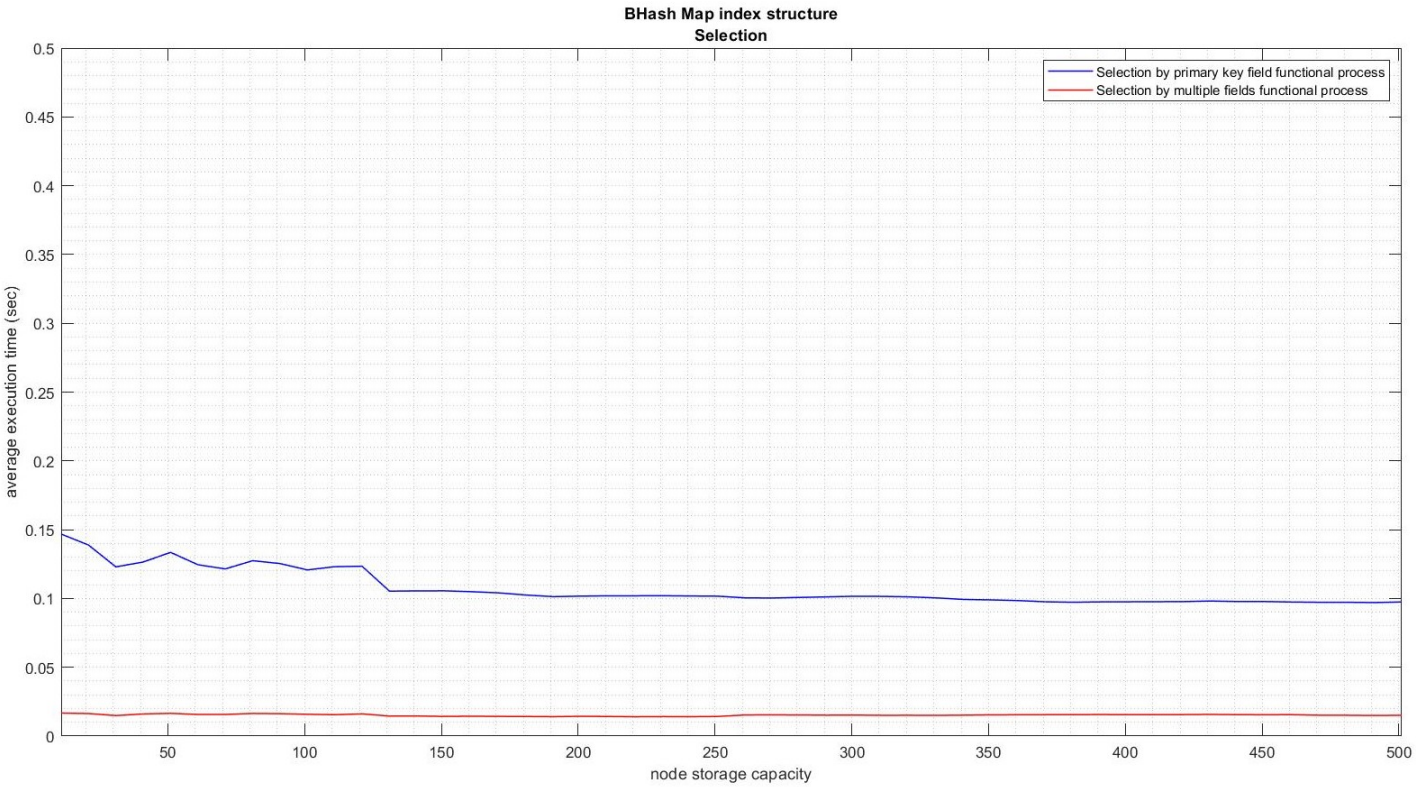


Fig. 5.53 represents the average time performance of the B-Hash Map index structure selection by primary key field and full scan - selection functional processes. The utilized records set consists of records with primary key fields of string type.

Figure 5.16: Functional process of B-Hash Map index structure selection by primary key field and full scan - selection average time performance

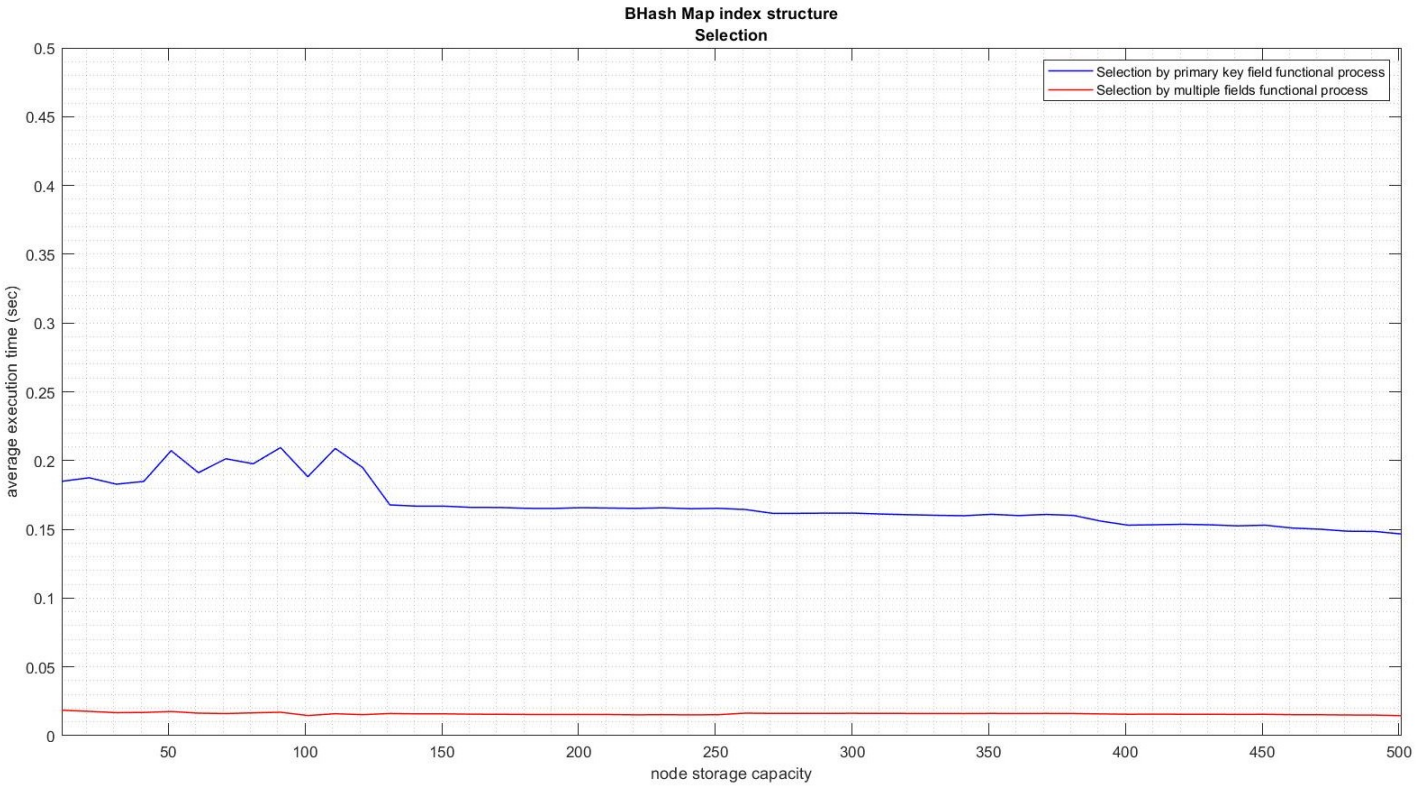


Fig. 5.17 represents the average time performance of the B⁺-Hash Map index structure selection by primary key field and full scan - selection functional processes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.17: Functional process of B⁺-Hash Map index structure selection by primary key field and full scan - selection average time performance

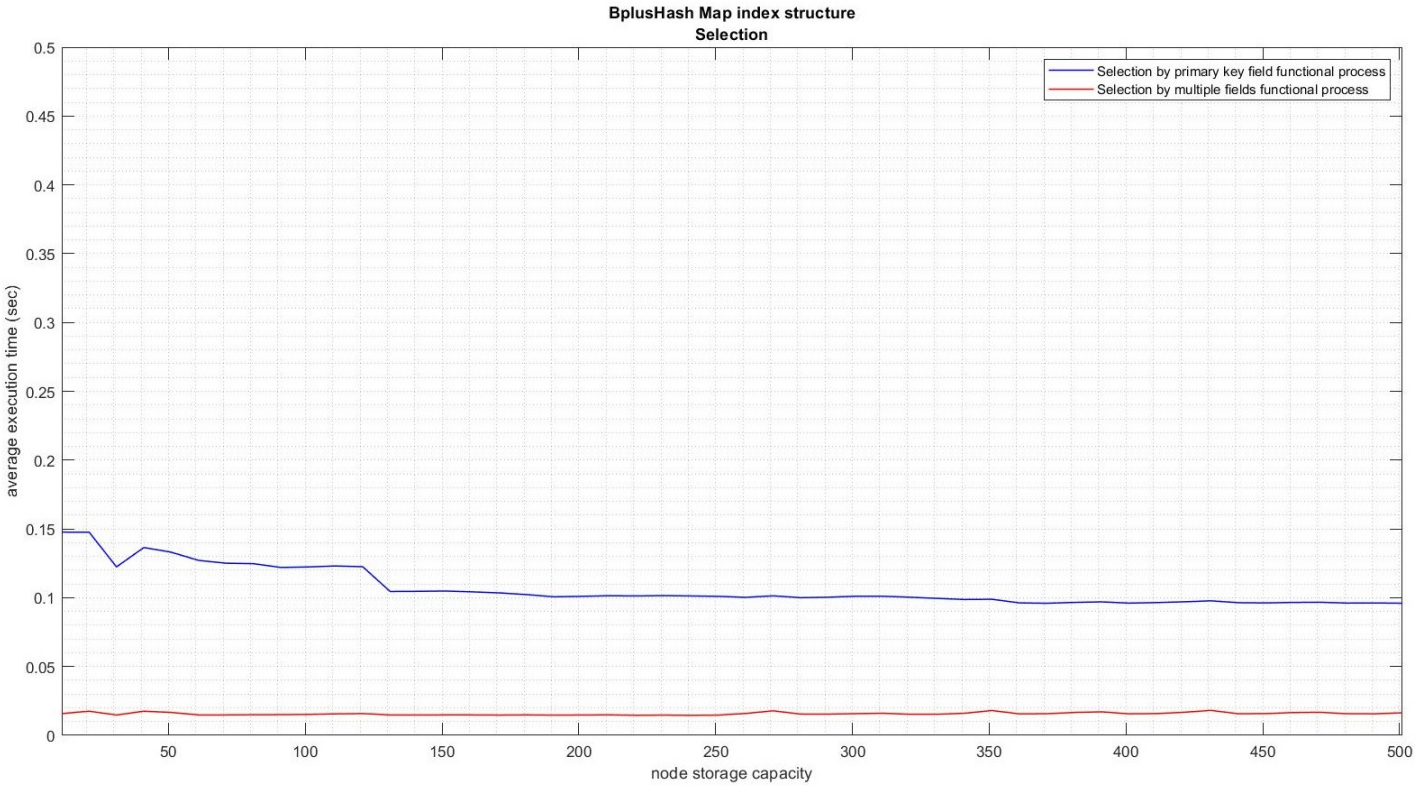


Fig. 5.18 represents the average time performance of the B⁺-Hash Map index structure selection by primary key field and full scan - selection functional processes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.18: Functional process of B⁺-Hash Map index structure selection by primary key field and full scan - selection average time performance

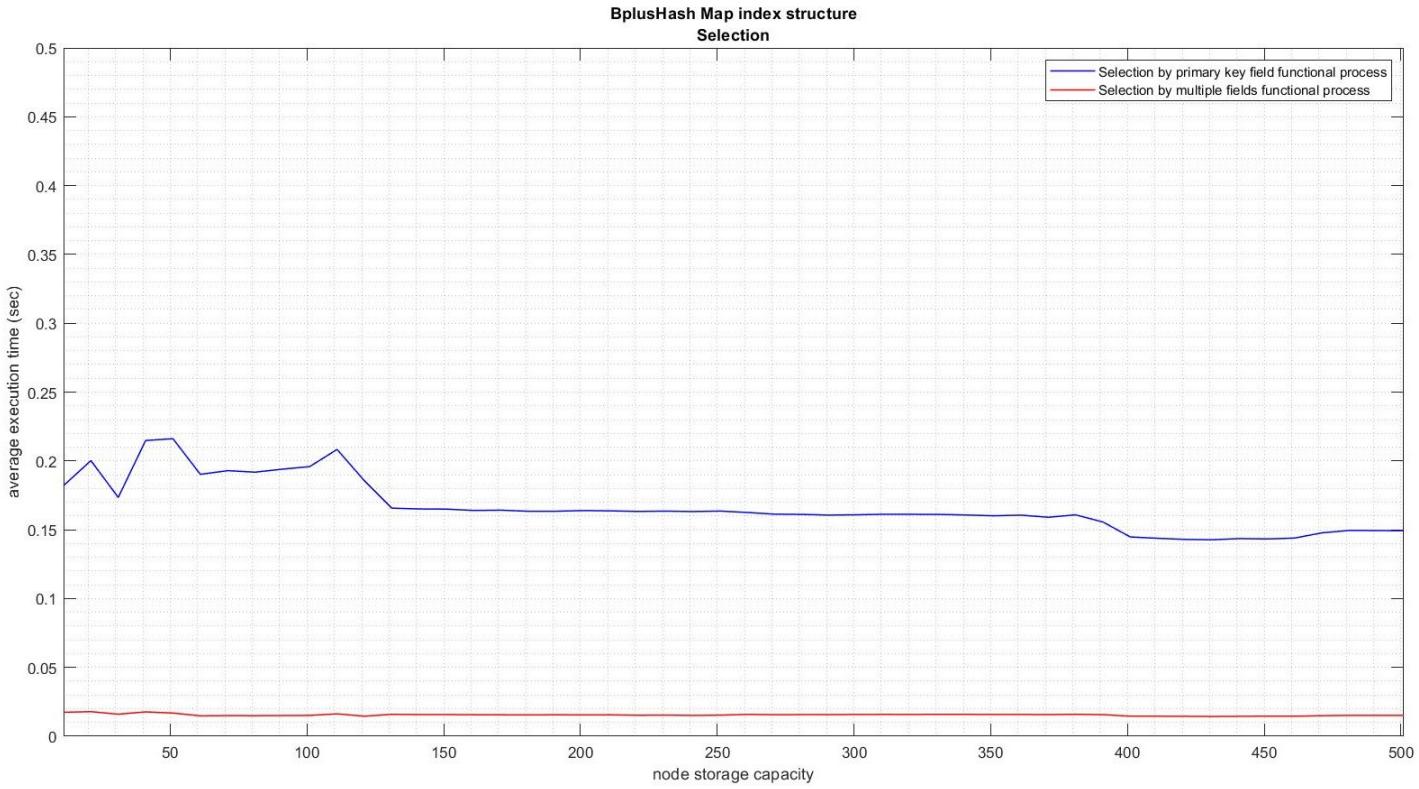


Fig. 5.19 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures selection by primary key field functional processes. The utilized records sets are composed of records with primary key fields of integer and string type.

Figure 5.19: Functional processes of B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures selection by primary key field average time performance

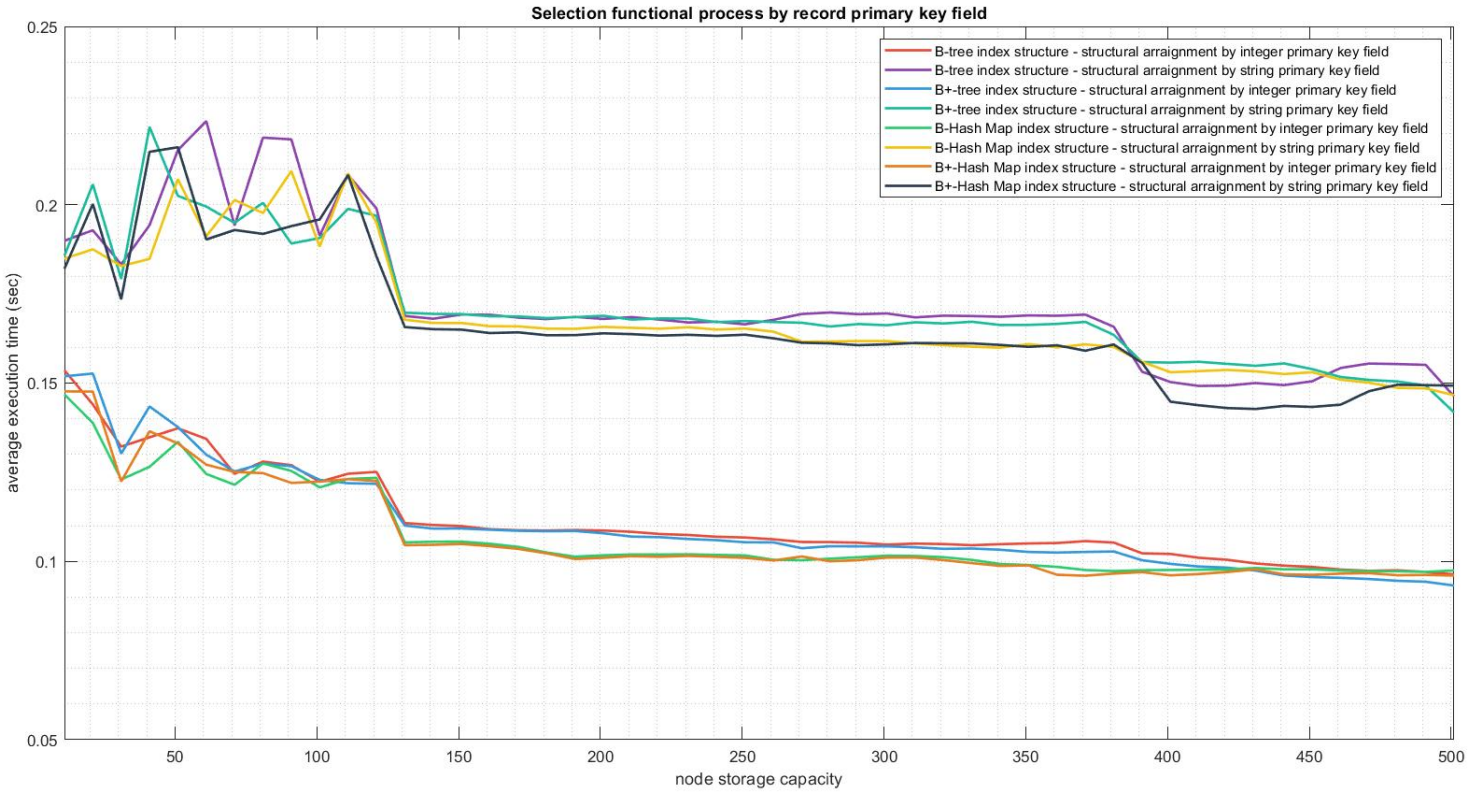
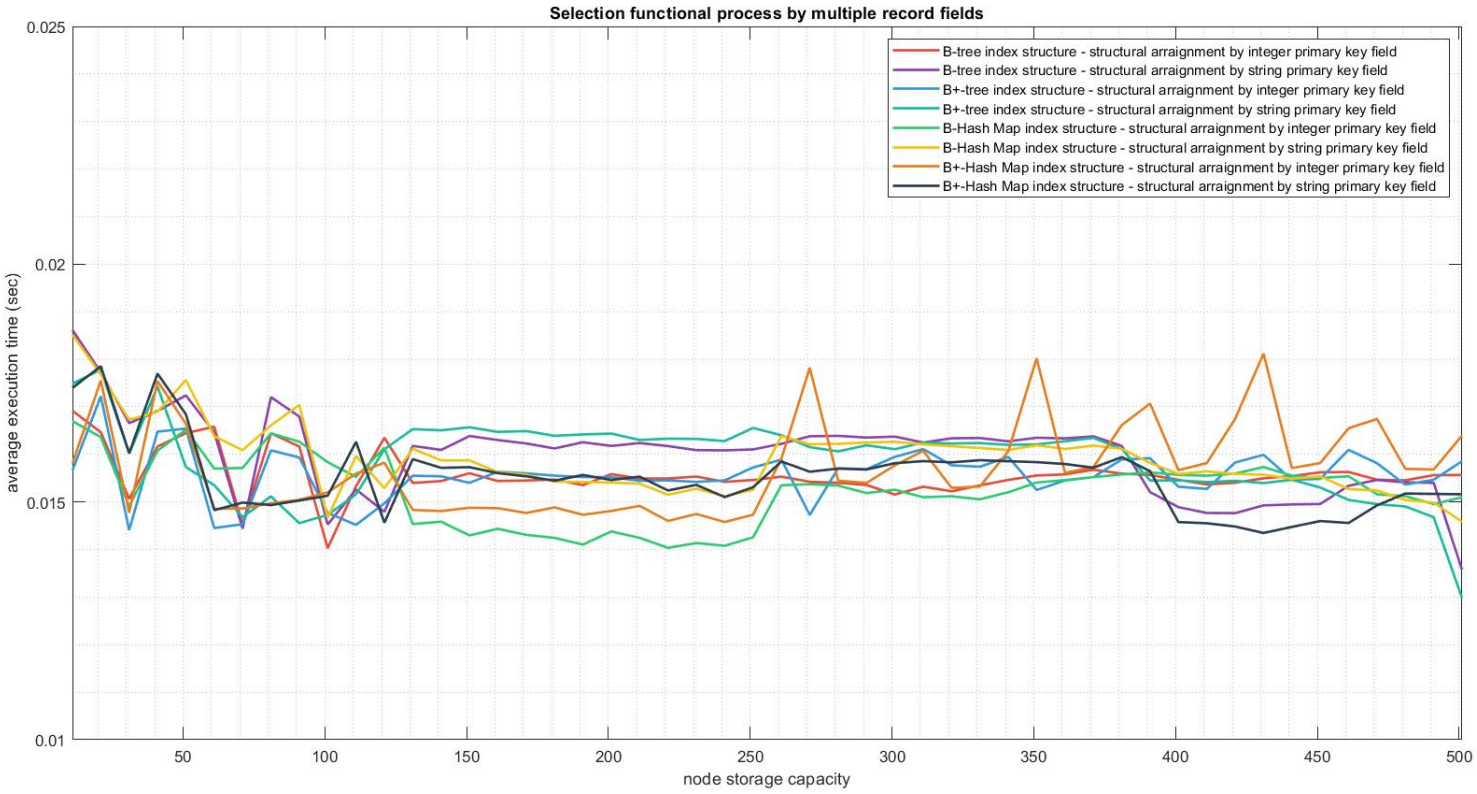


Fig. 5.20 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures full records scan and selection by multiple record fields functional processes. The utilized records sets are composed of records with primary key fields of integer and string type.

Figure 5.20: Functional processes of B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures full records scan and selection by multiple record fields average time performance



Computational study results of average internal and leaf nodes distribution in the index structures

Fig. 5.21 represents the average structural distribution of the B-tree index structure nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.21: Average structural distribution of the B-tree index structure nodes in internal and leaf nodes

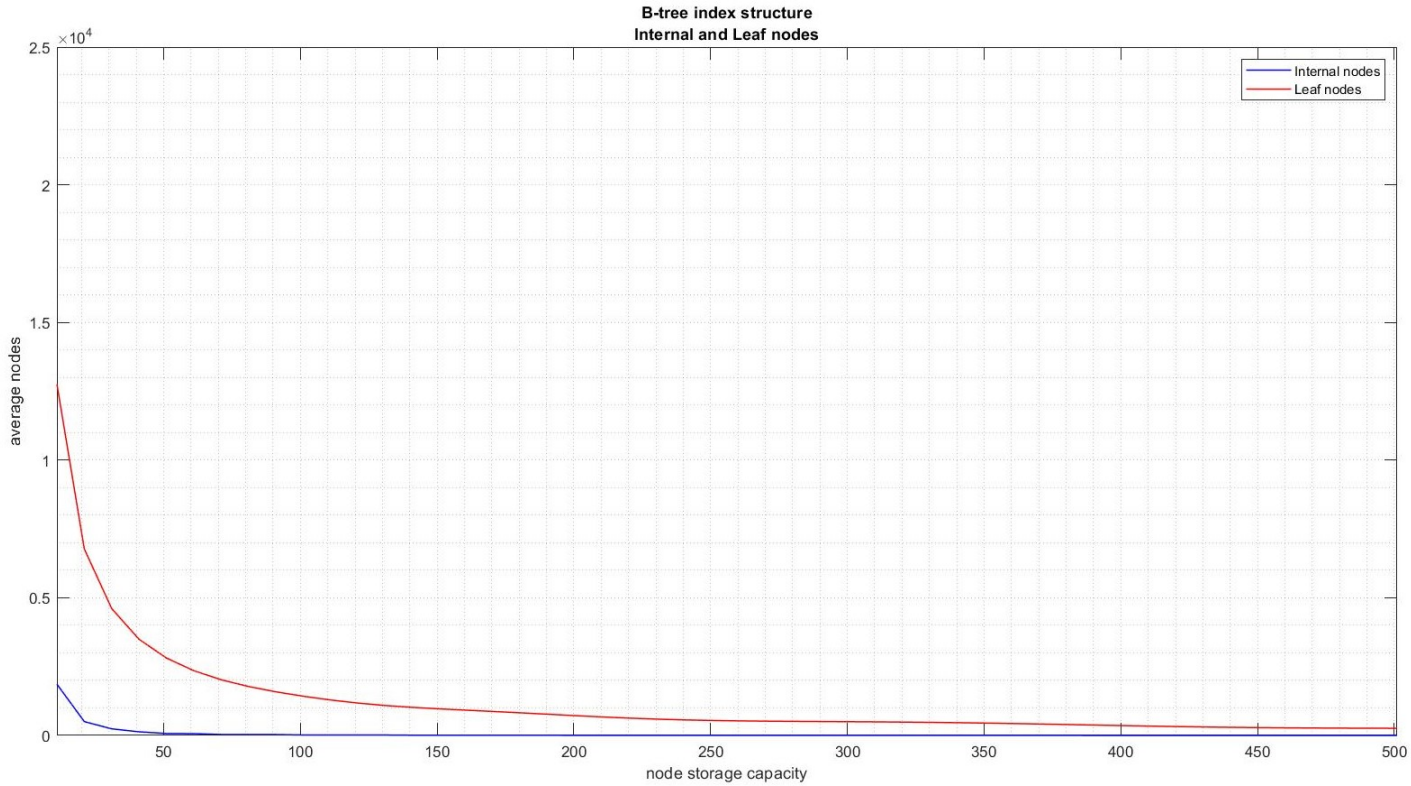


Fig. 5.22 represents the average structural distribution of the B-tree index structure nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.22: Average structural distribution of the B-tree index structure nodes in internal and leaf nodes

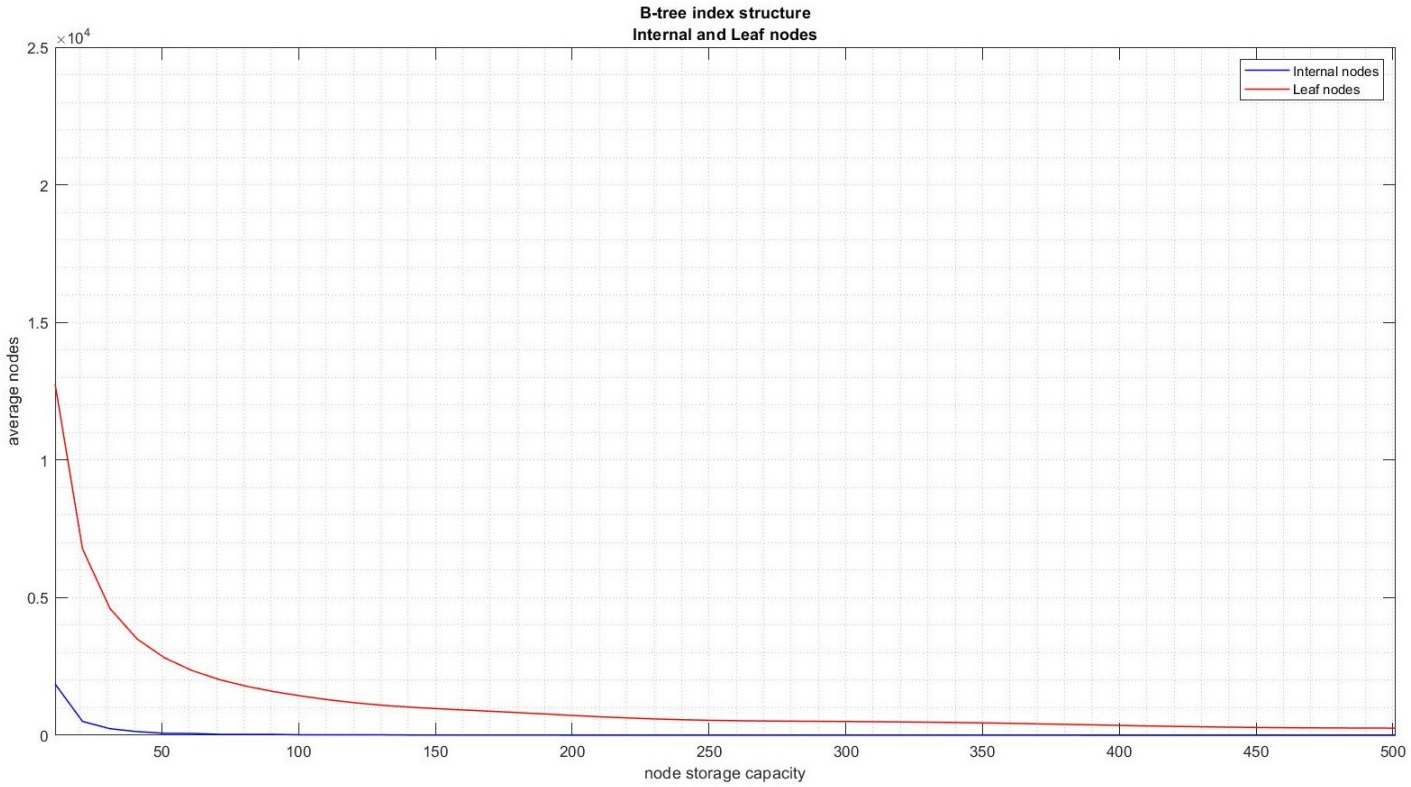


Fig. 5.23 represents the average structural distribution of the B^+-tree index structure nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.23: Average structural distribution of the B^+-tree index structure nodes in internal and leaf nodes

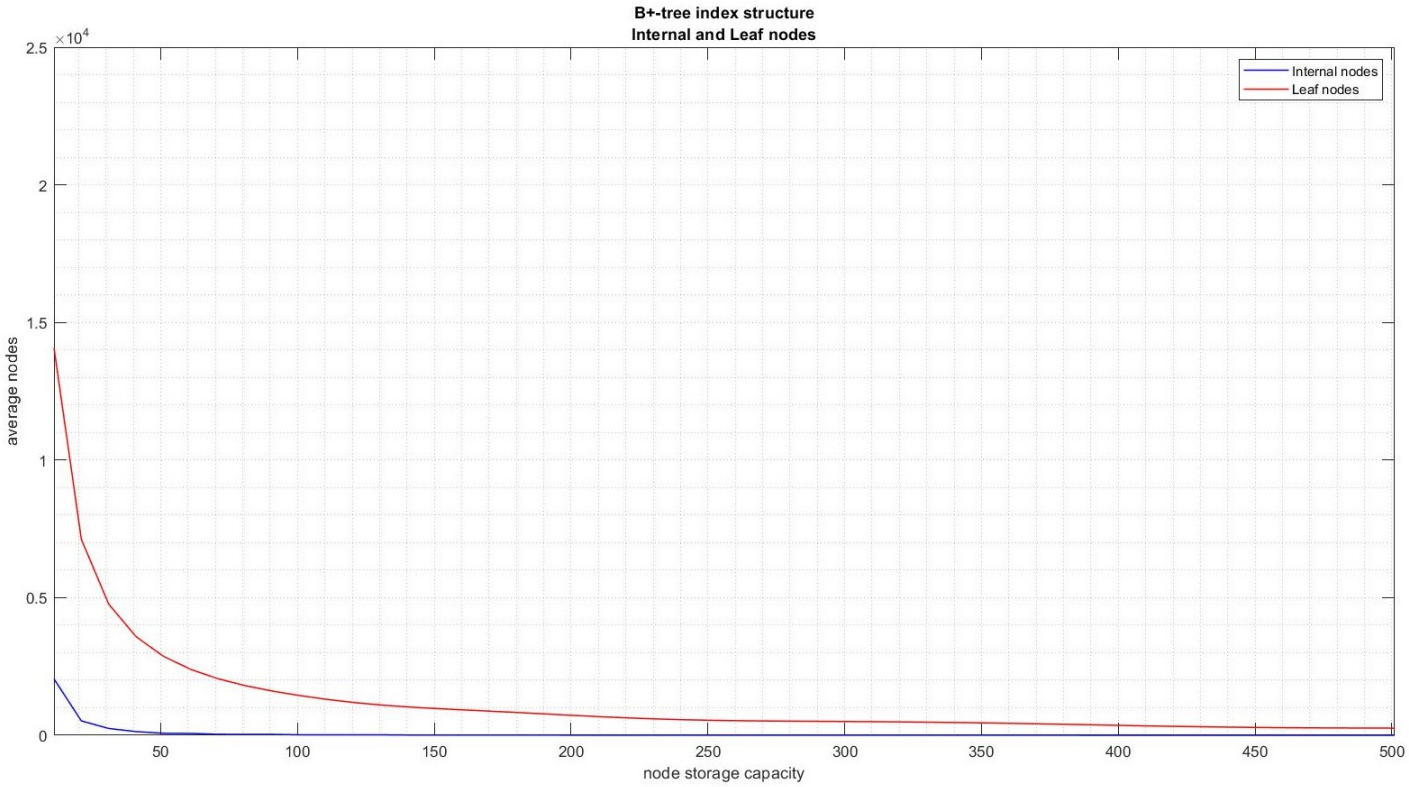


Fig. 5.24 represents the average structural distribution of the B$^+$-tree index structure nodes in internal and leaf nodes. The utilized records set consists of records with primary key fields of string type.

Figure 5.24: Average structural distribution of the B$^+$-tree index structure nodes in internal and leaf nodes

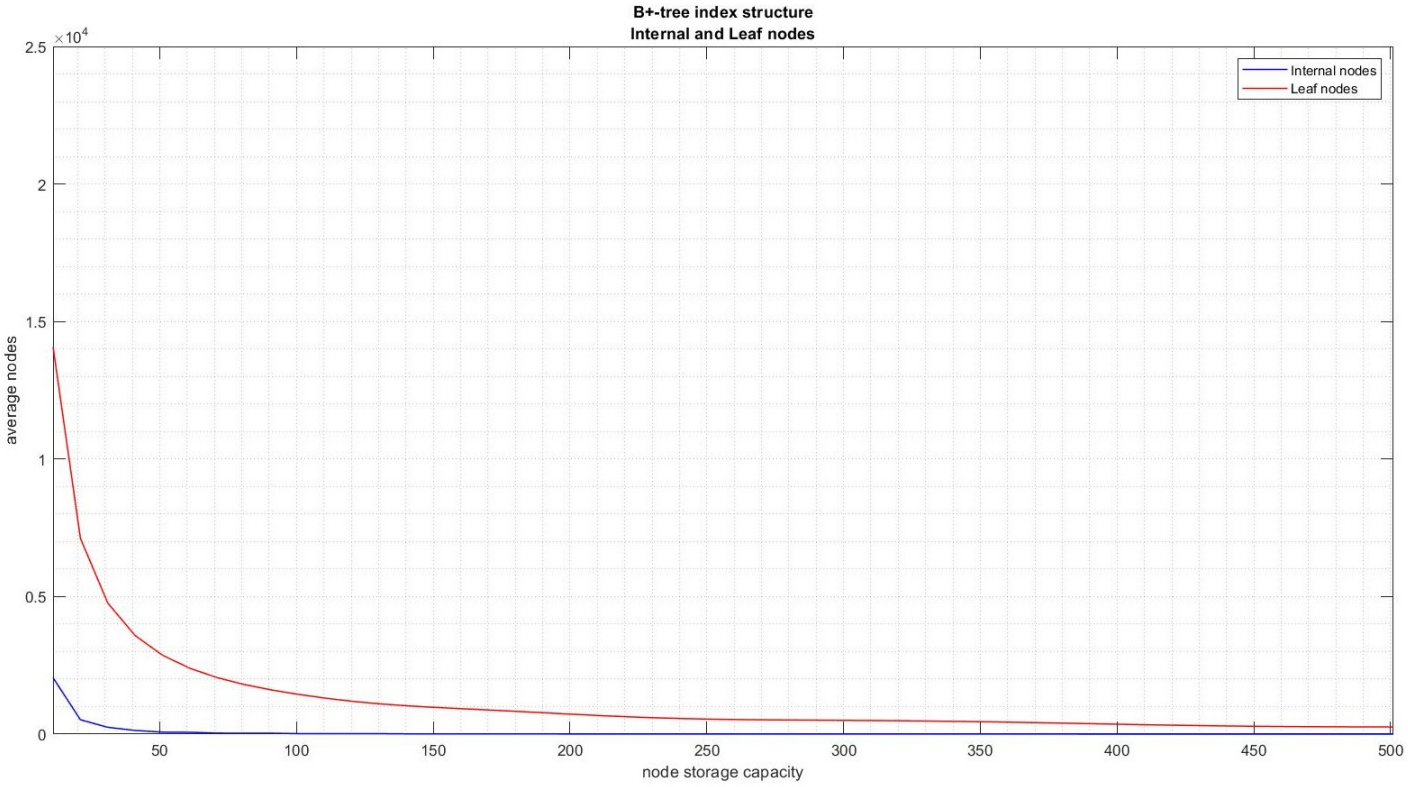


Fig. 5.25 represents the average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.25: Average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes

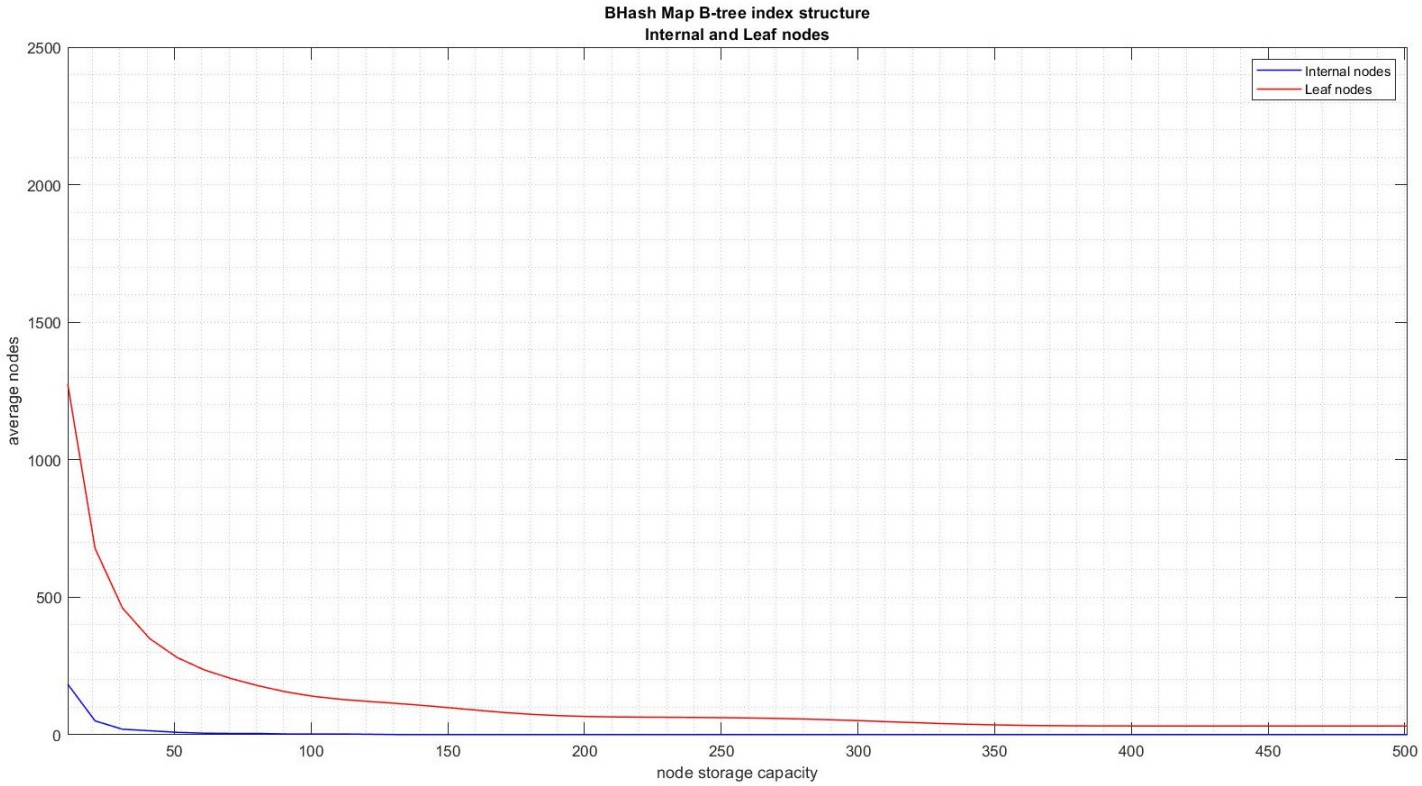


Fig. 5.26 represents the average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.26: Average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes

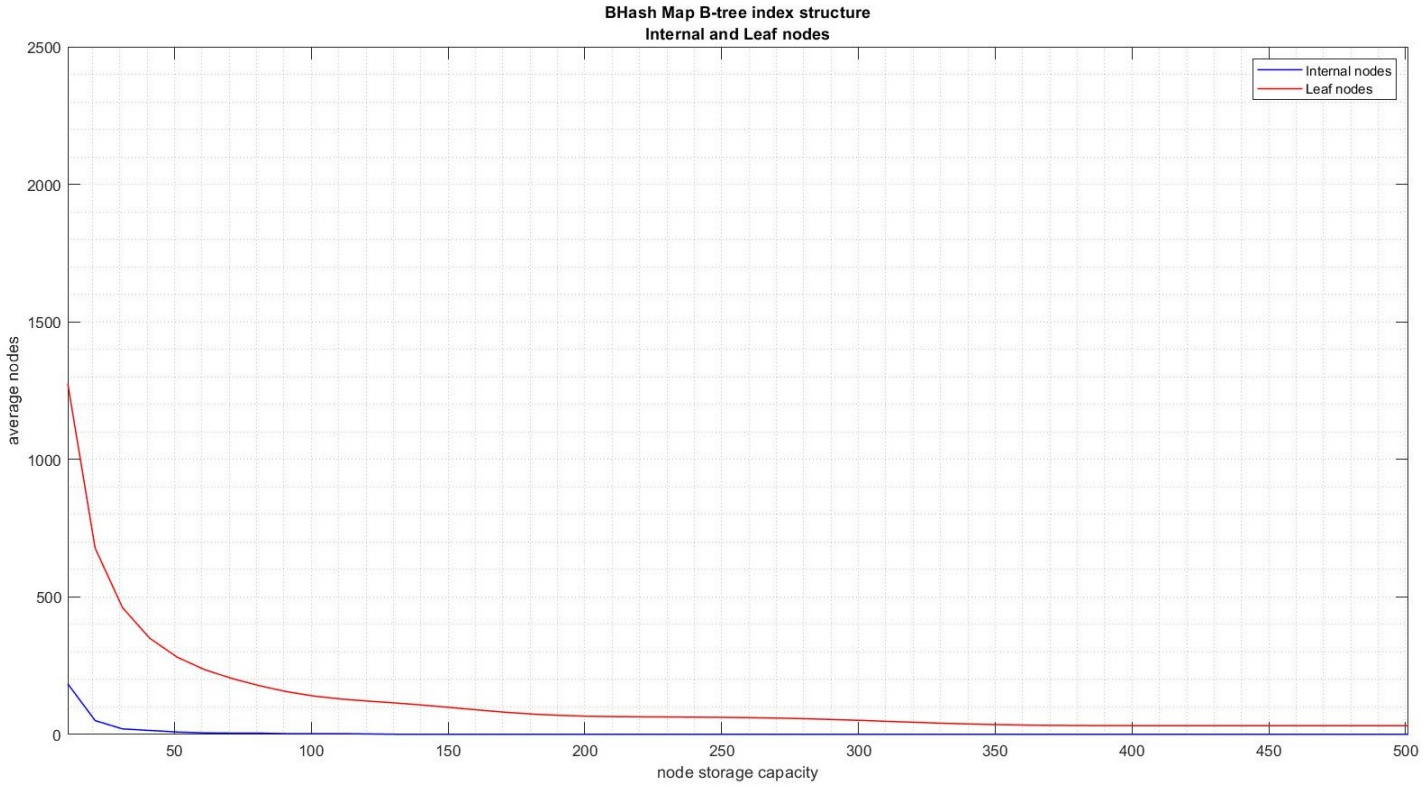


Fig. 5.27 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.27: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes

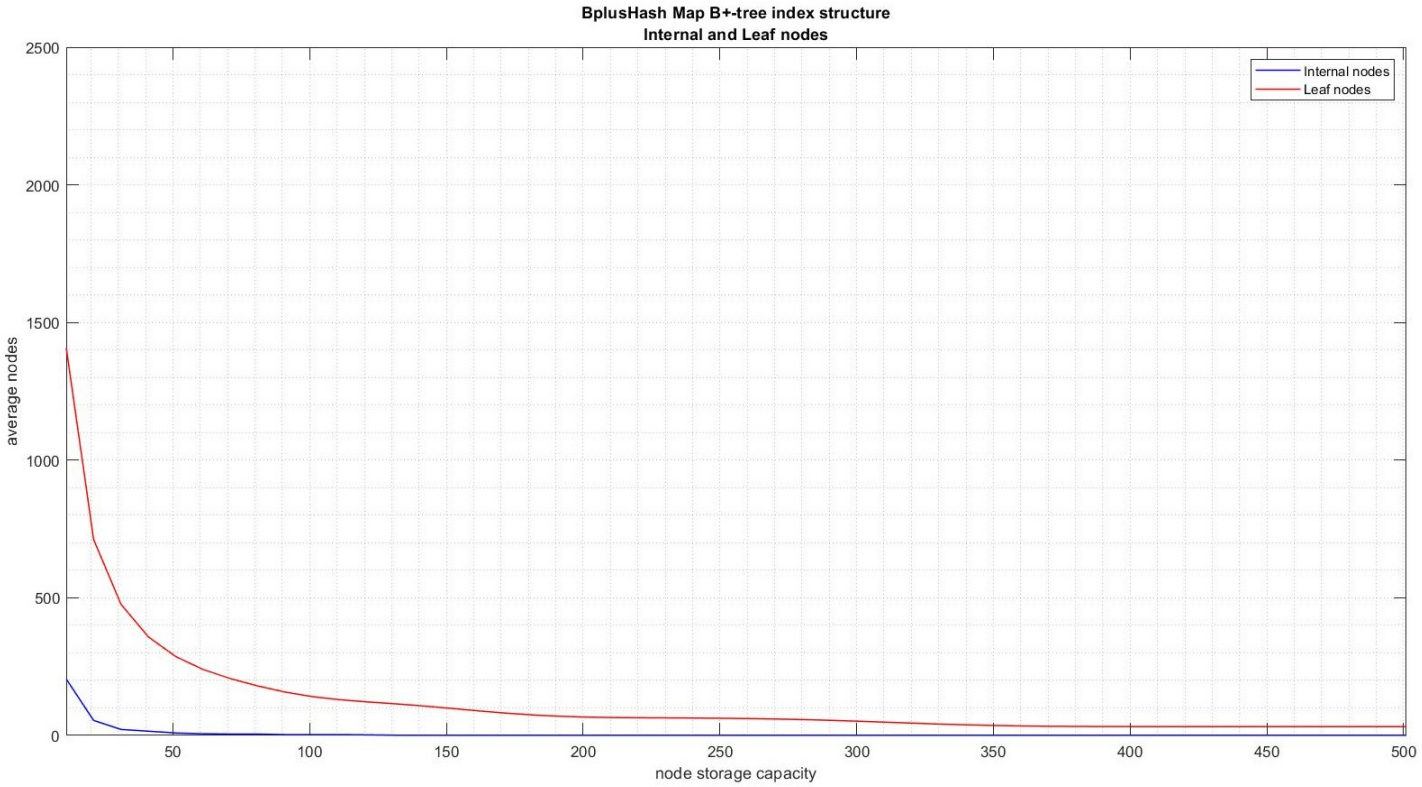
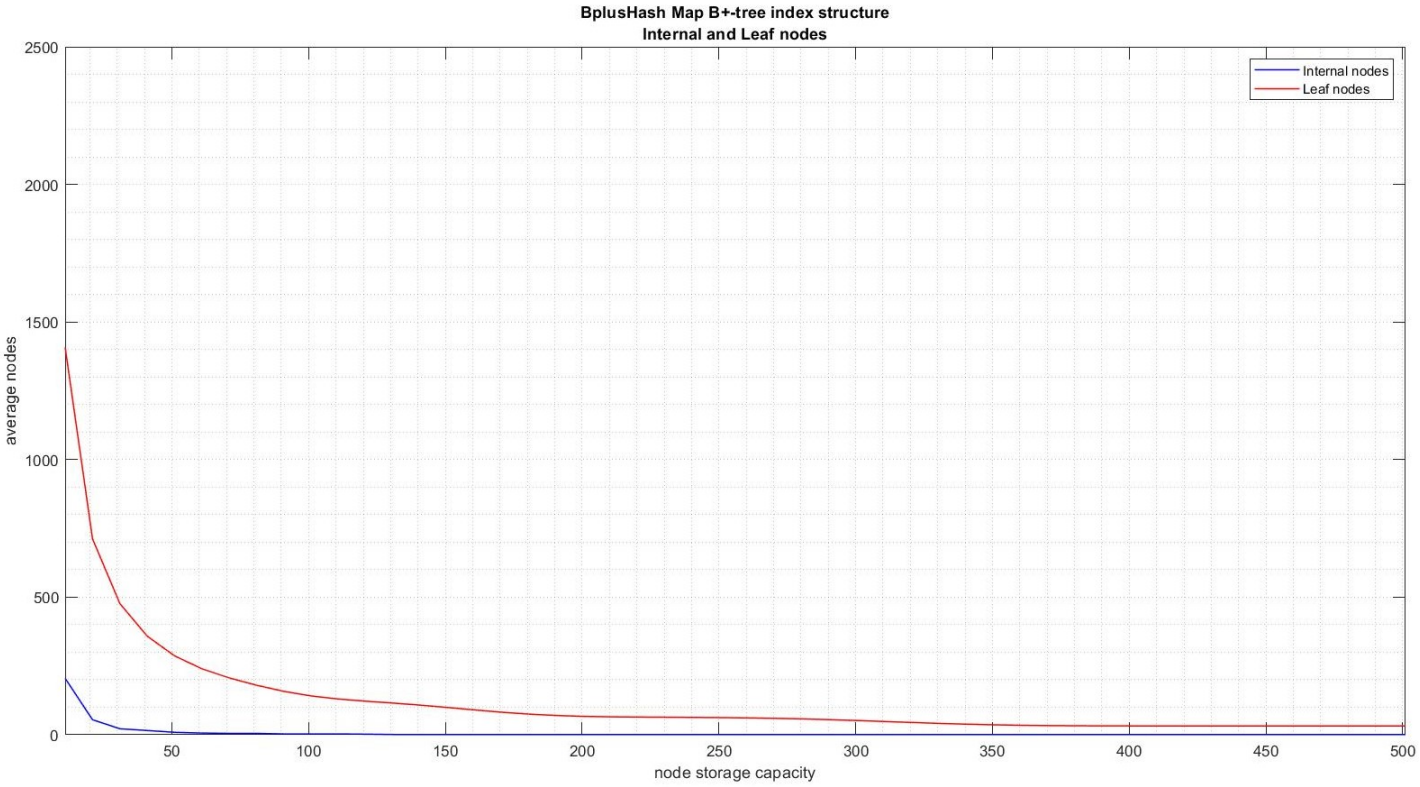


Fig. 5.28 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.28: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes



Computational study results of average internal and leaf nodes stored records distribution in the index structures

Fig. 5.29 represents the average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes. The utilized records set consists of records with primary key fields of integer type.

Figure 5.29: Average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes

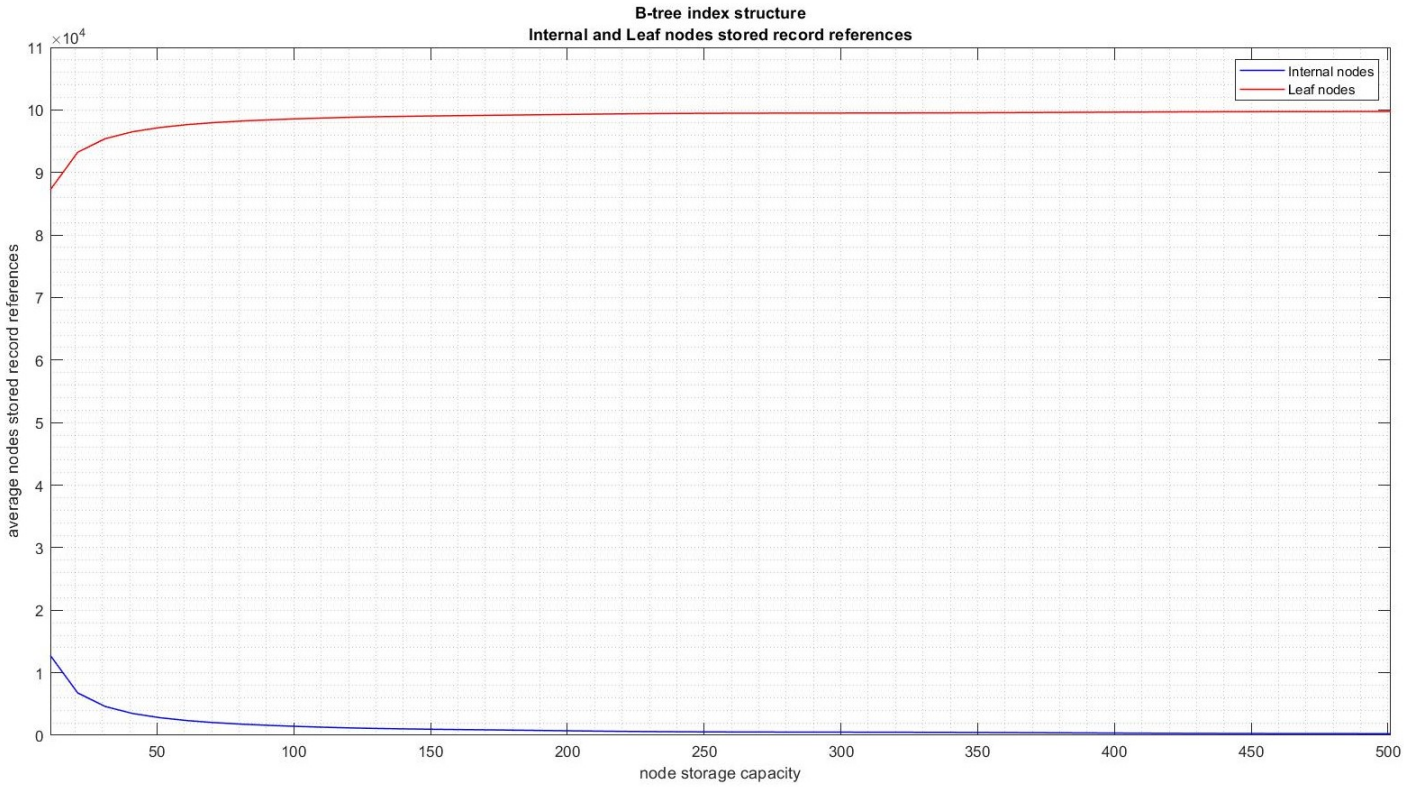


Fig. 5.30 represents the average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.30: Average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes

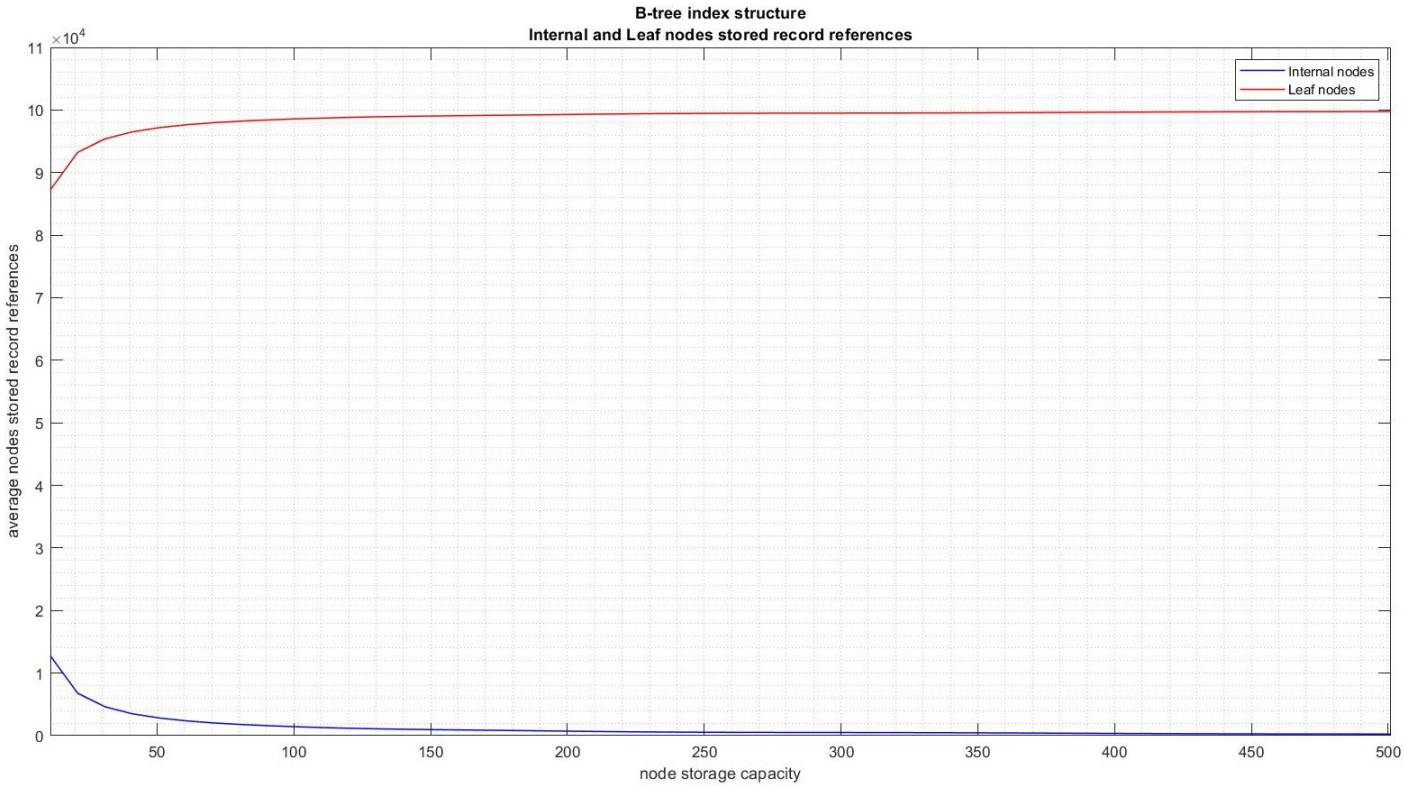


Fig. 5.31 represents the average structural distribution of the B$^+$-tree index structure nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.31: Average structural distribution of the B$^+$-tree index structure nodes stored records in internal and leaf nodes

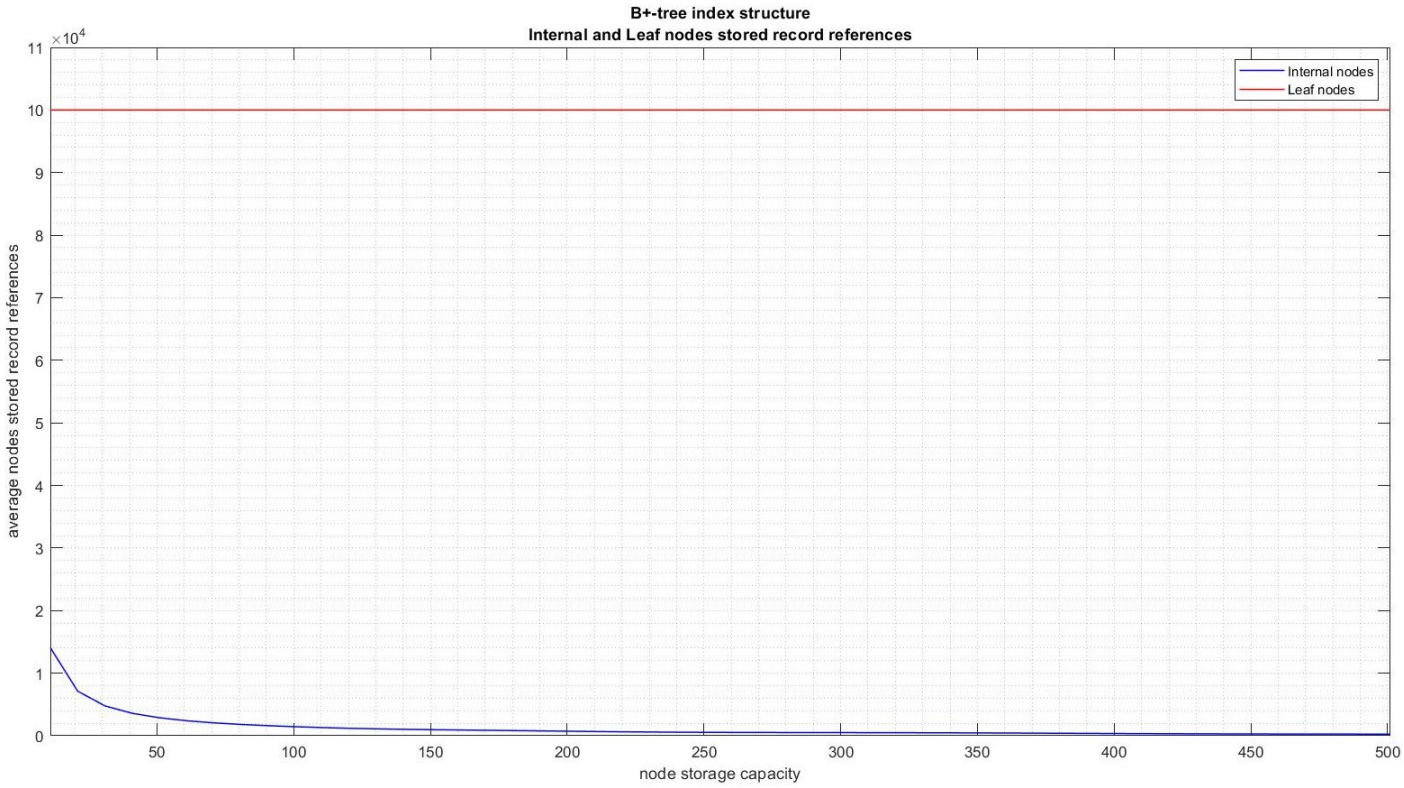


Fig. 5.32 represents the average structural distribution of the B$^+$-tree index structure nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.32: Average structural distribution of the B$^+$-tree index structure nodes stored records in internal and leaf nodes

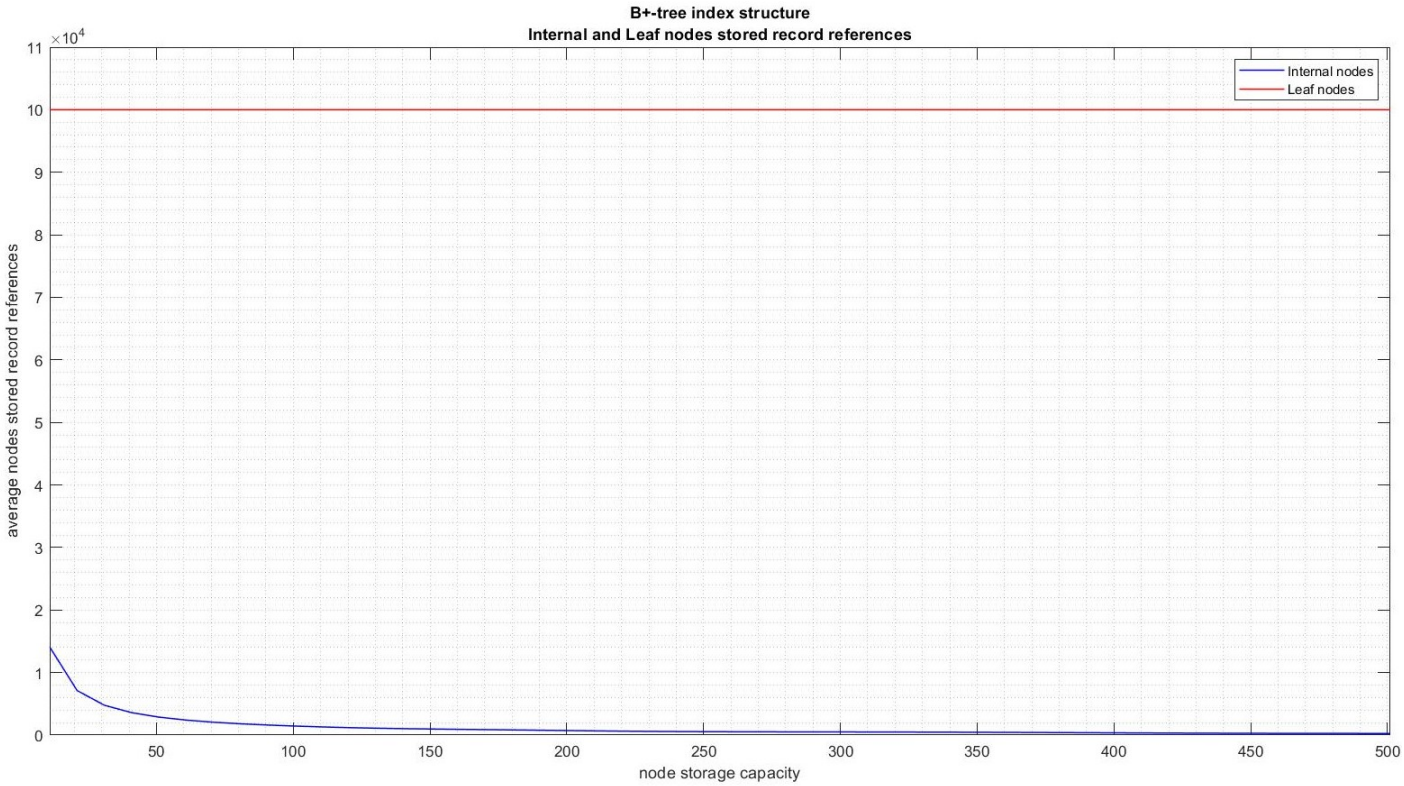


Fig. 5.33 represents the average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.33: Average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes

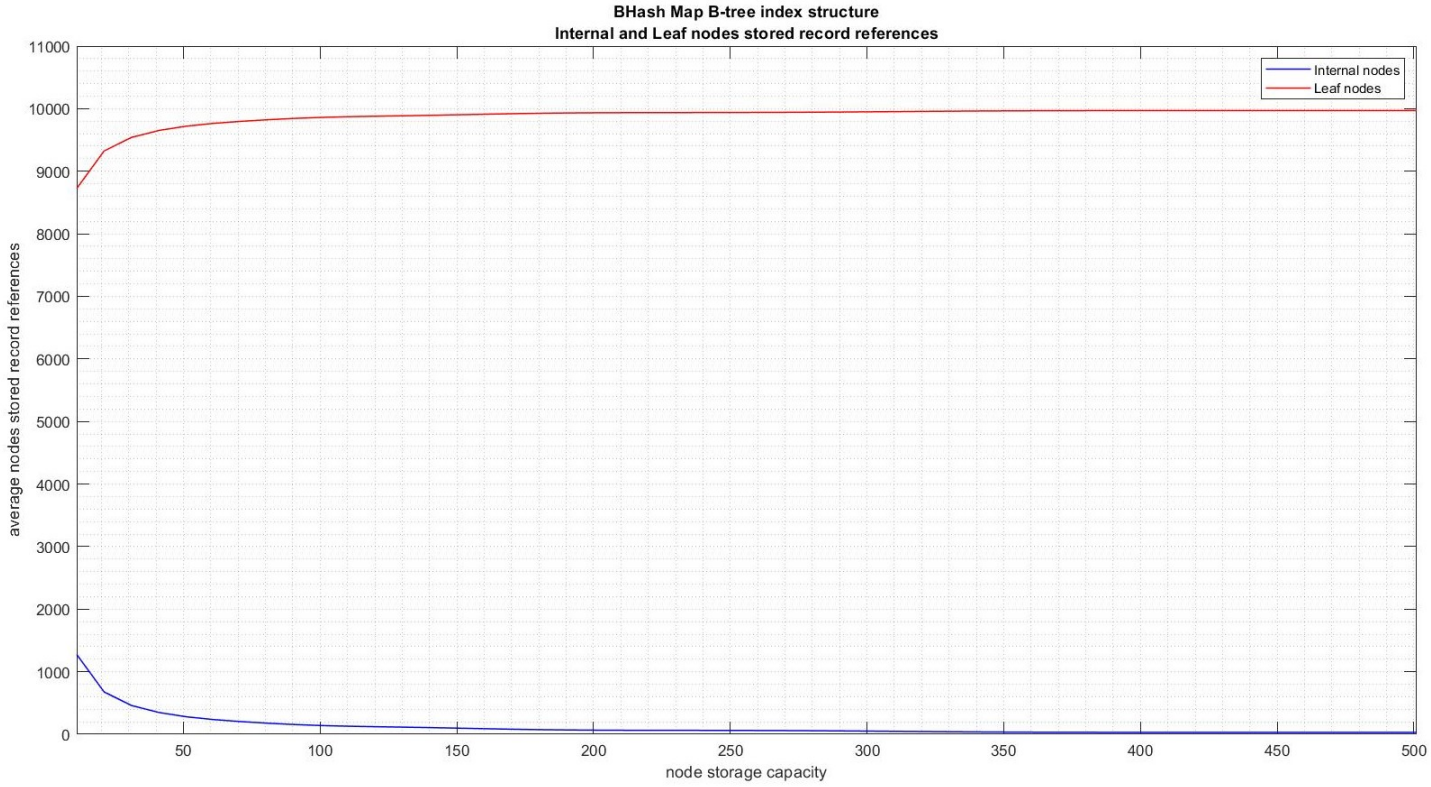


Fig. 5.34 represents the average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes. The utilized records set consists of records with primary key fields of string type.

Figure 5.34: Average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes

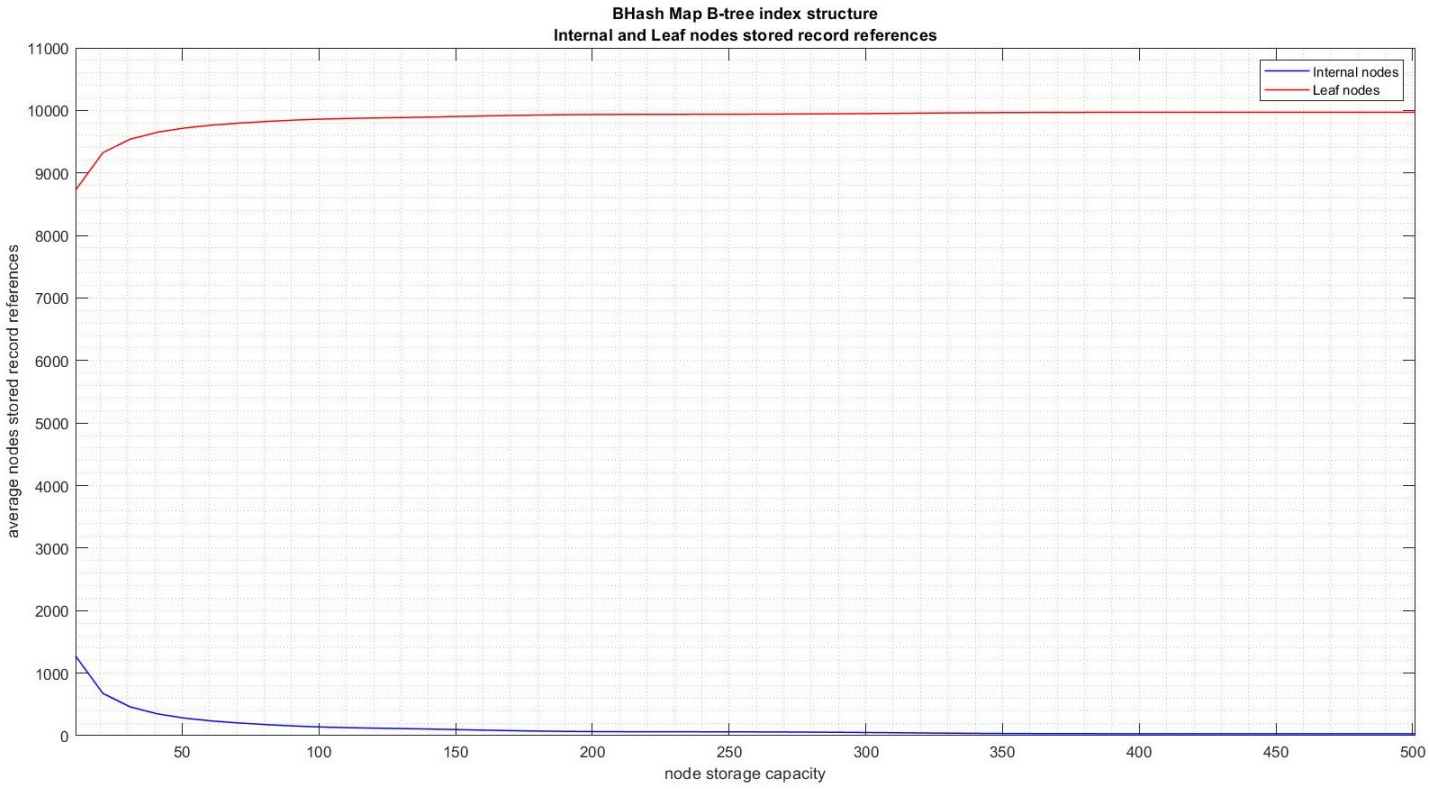


Fig. 5.35 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.35: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes

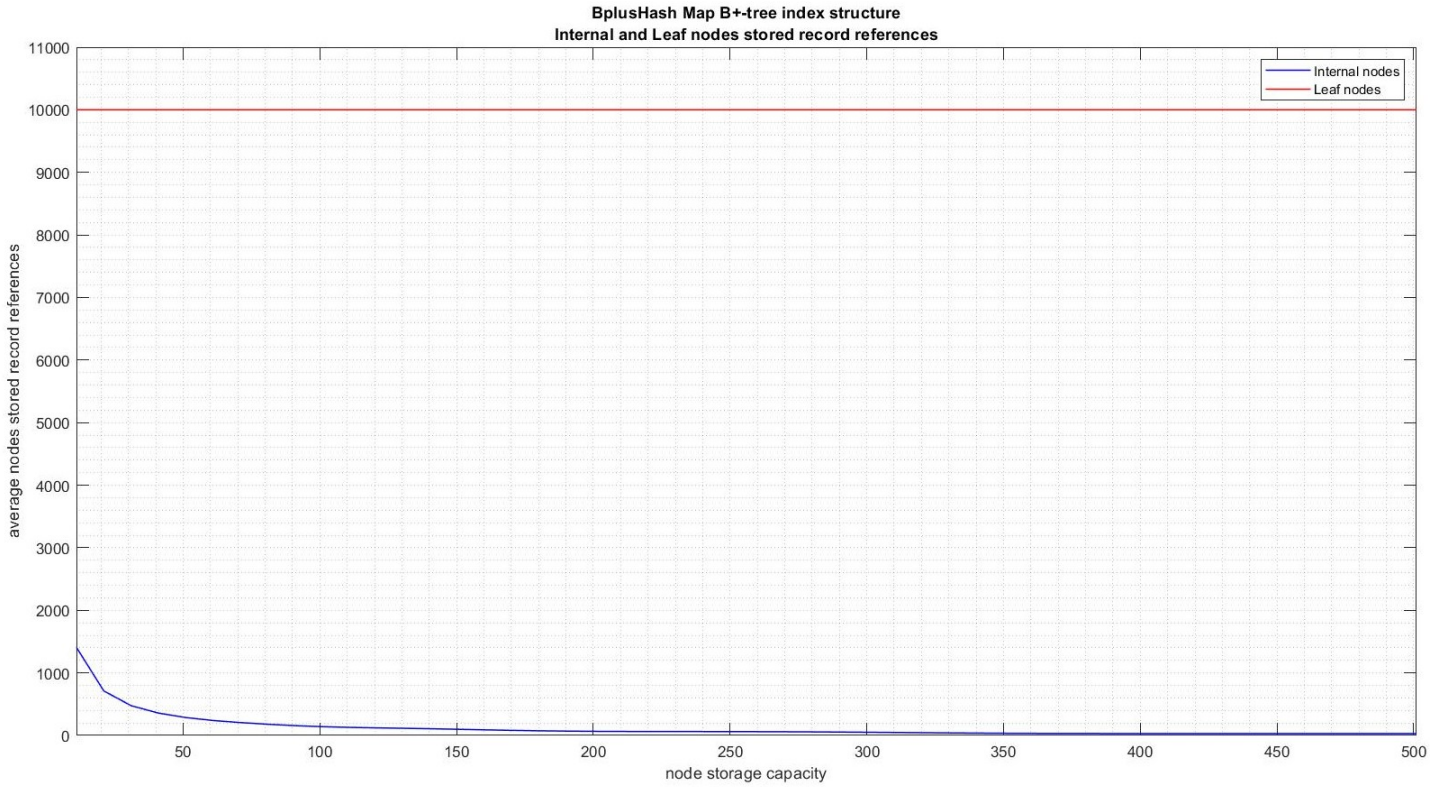
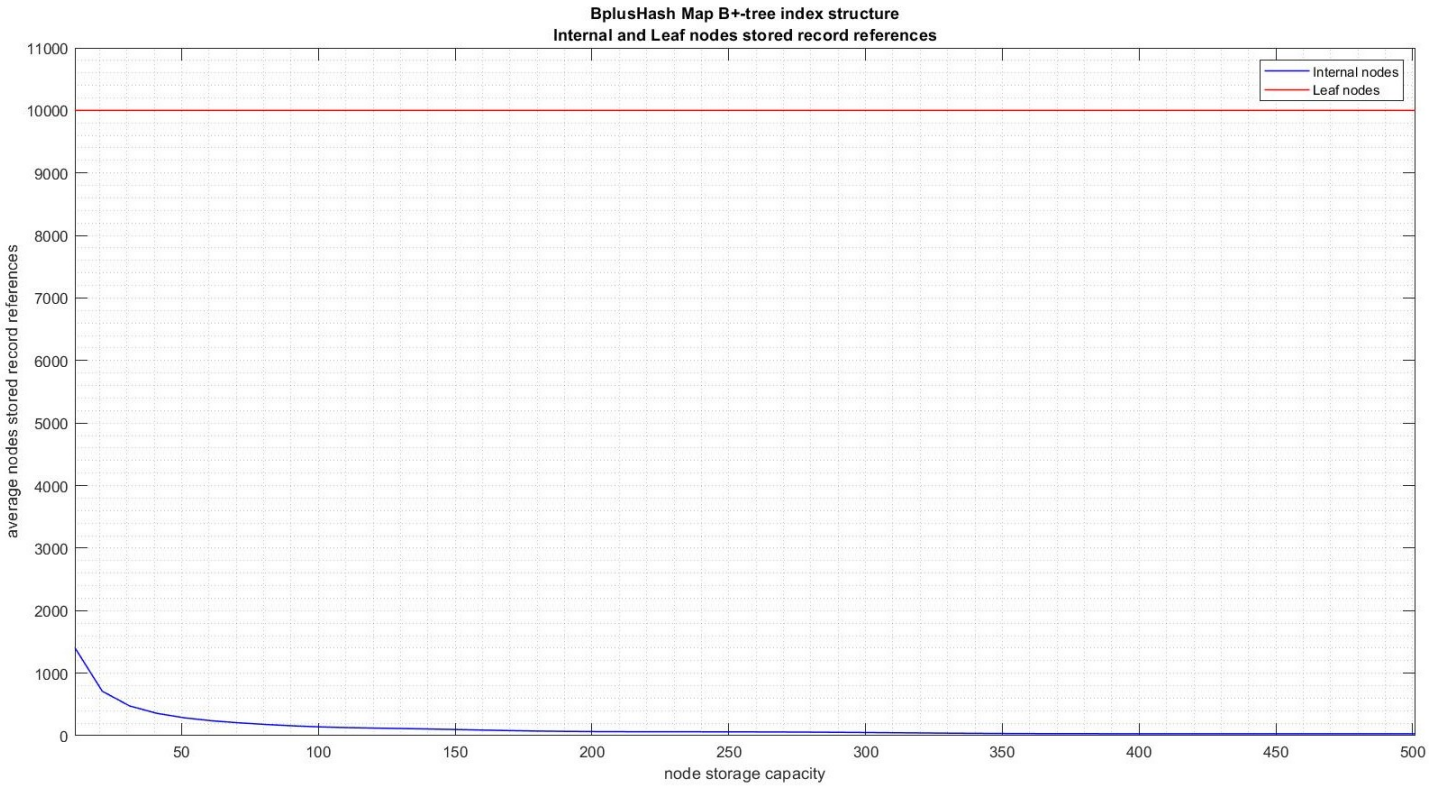


Fig. 5.36 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes. The utilized records set is composed of records with primary key fields of string type.

Figure 5.36: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes



Computational study results of average height

Fig. 5.37 represents the average B-tree index structure height. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.37: Average B-tree index structure height

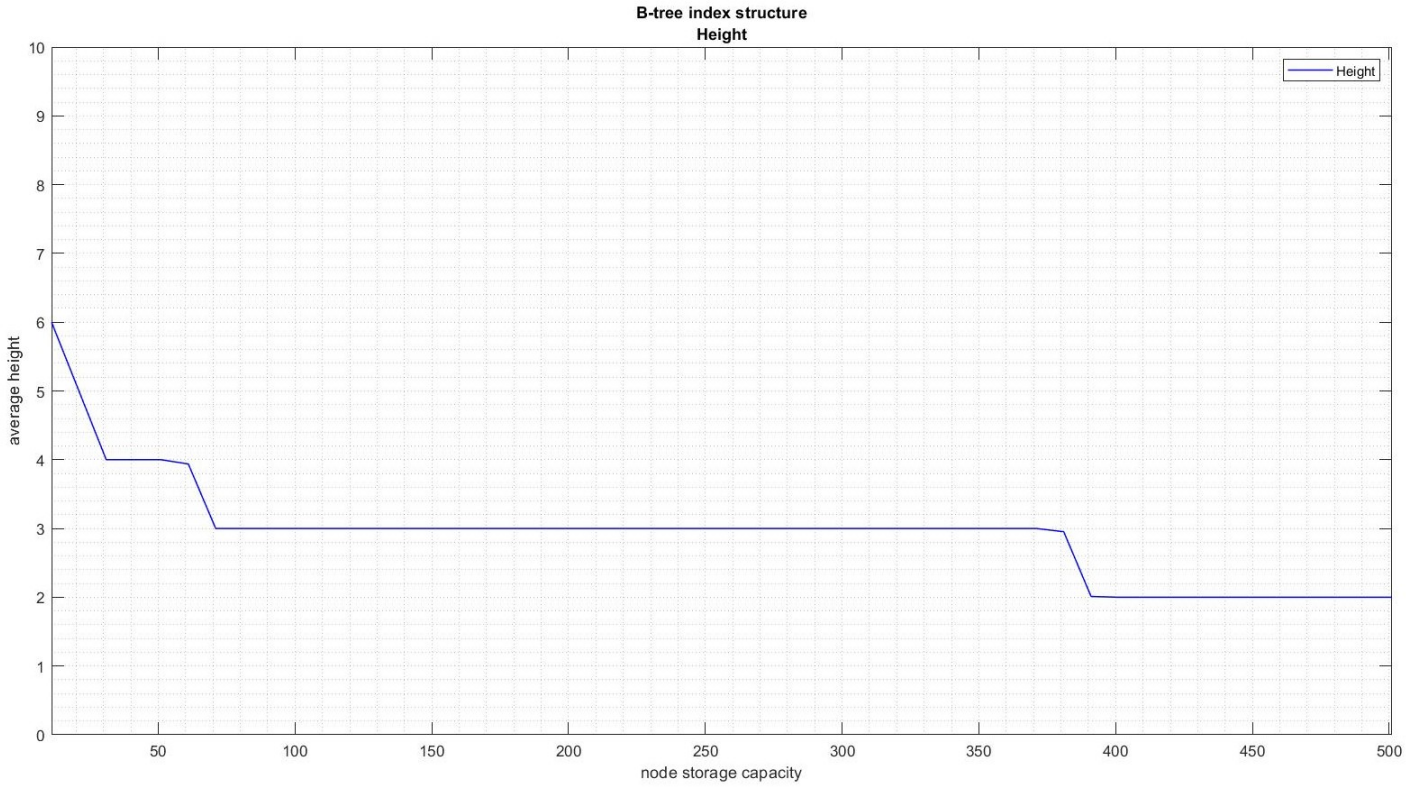


Fig. 5.38 represents the average B-tree index structure height. The utilized records set is composed of records with primary key fields of string type.

Figure 5.38: Average B-tree index structure height

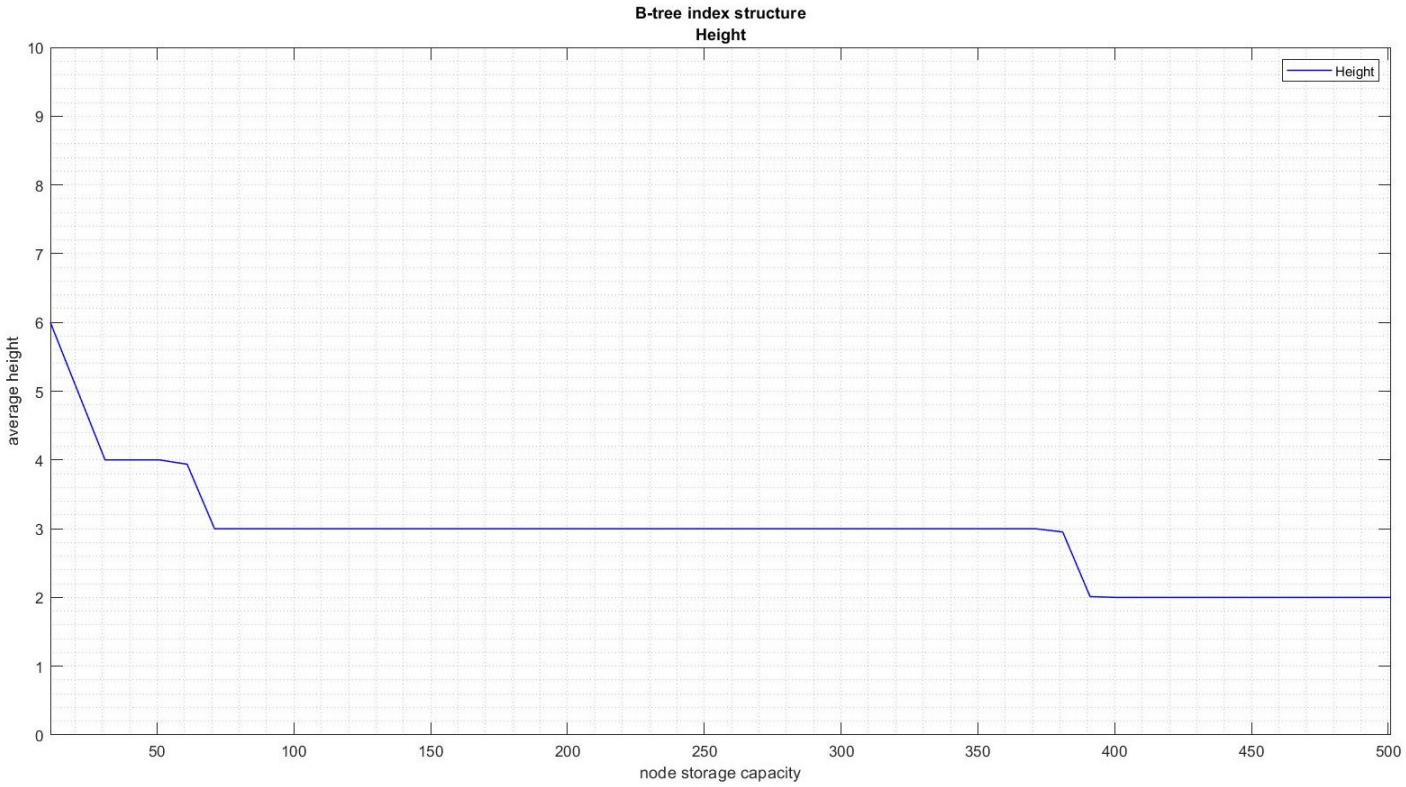


Fig. 5.39 represents the average B^+-tree index structure height. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.39: Average B^+-tree index structure height

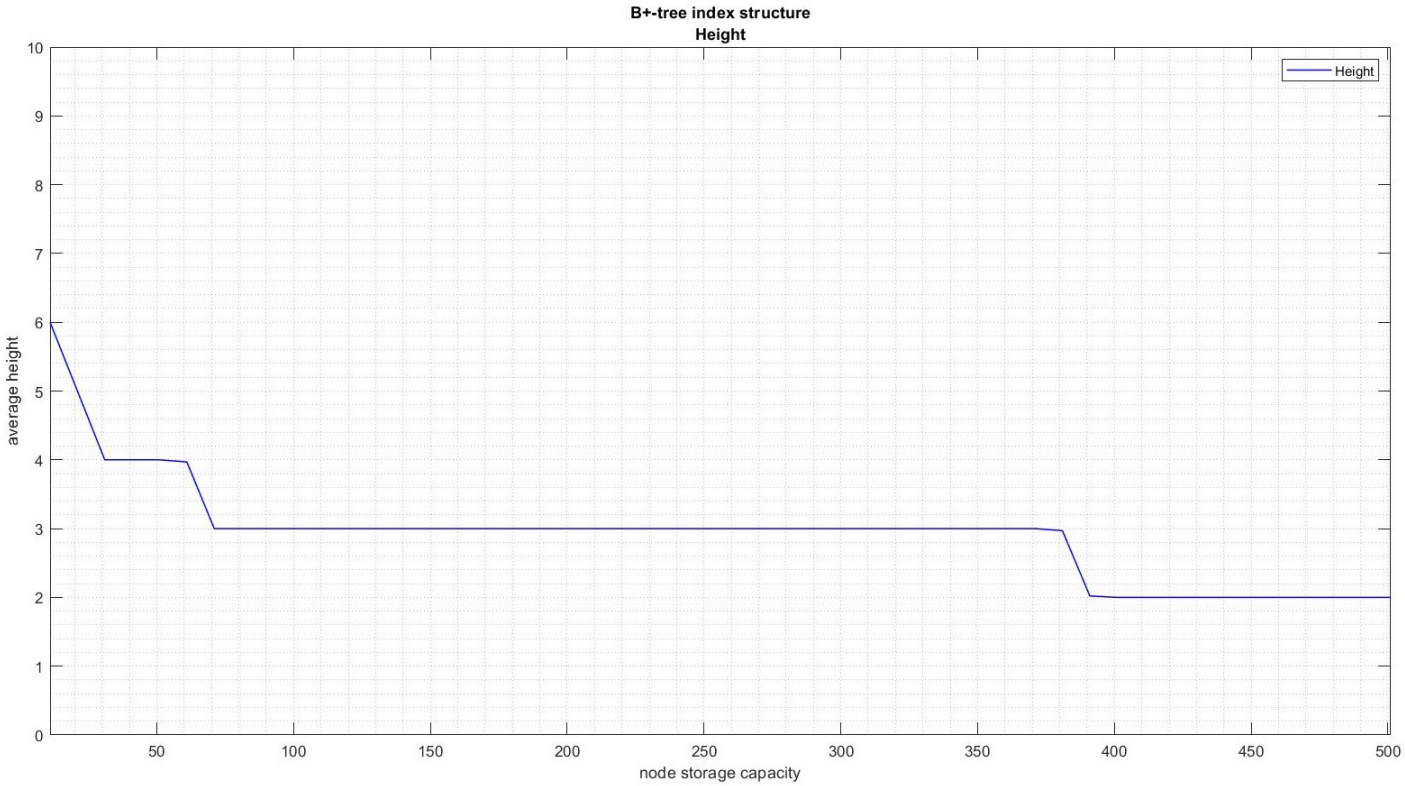


Fig. 5.40 represents the average B^+-tree index structure height. The utilized records set consists of records with primary key fields of string type.

Figure 5.40: Average B^+-tree index structure height

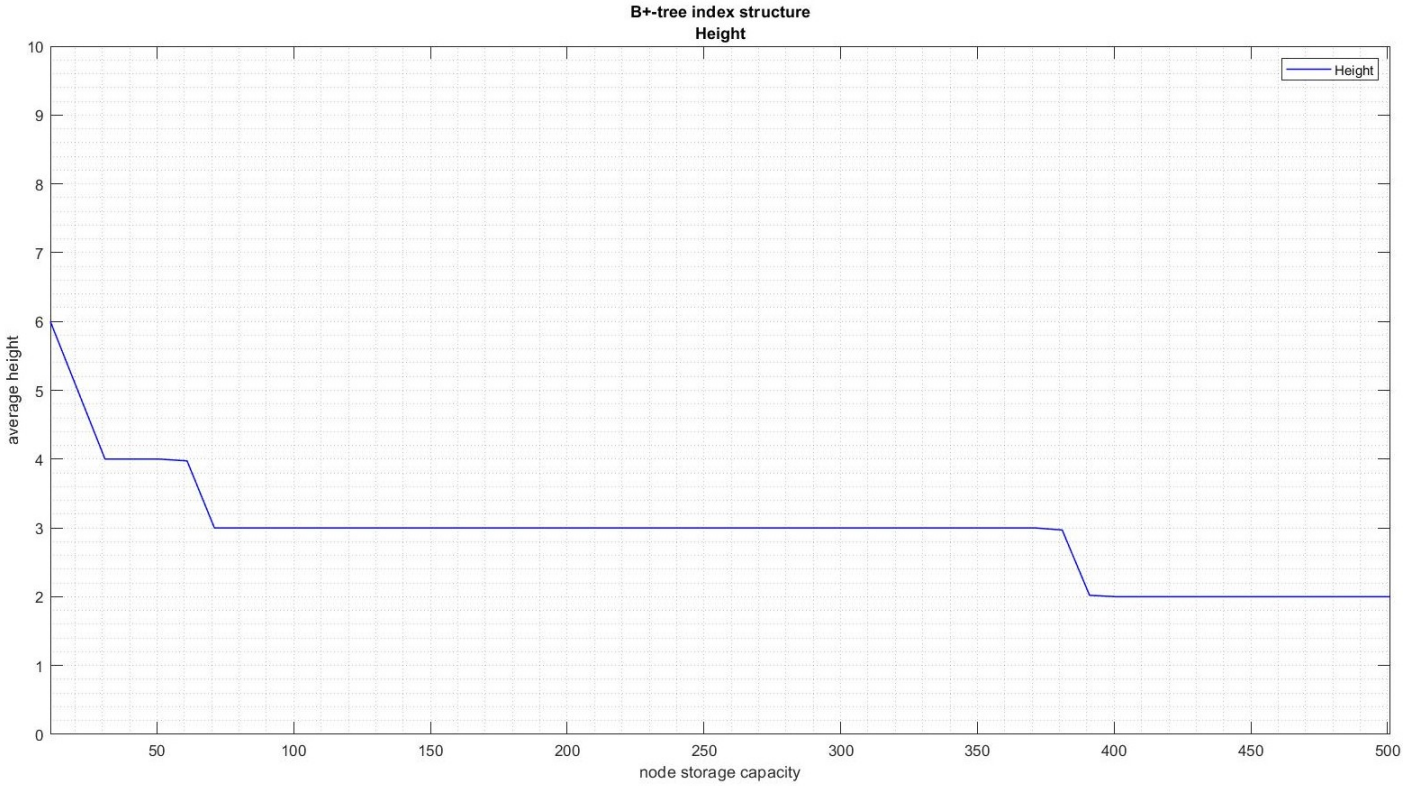


Fig. 5.41 represents the average B-Hash Map index structure B-tree height. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.41: Average B-Hash Map index structure B-tree height

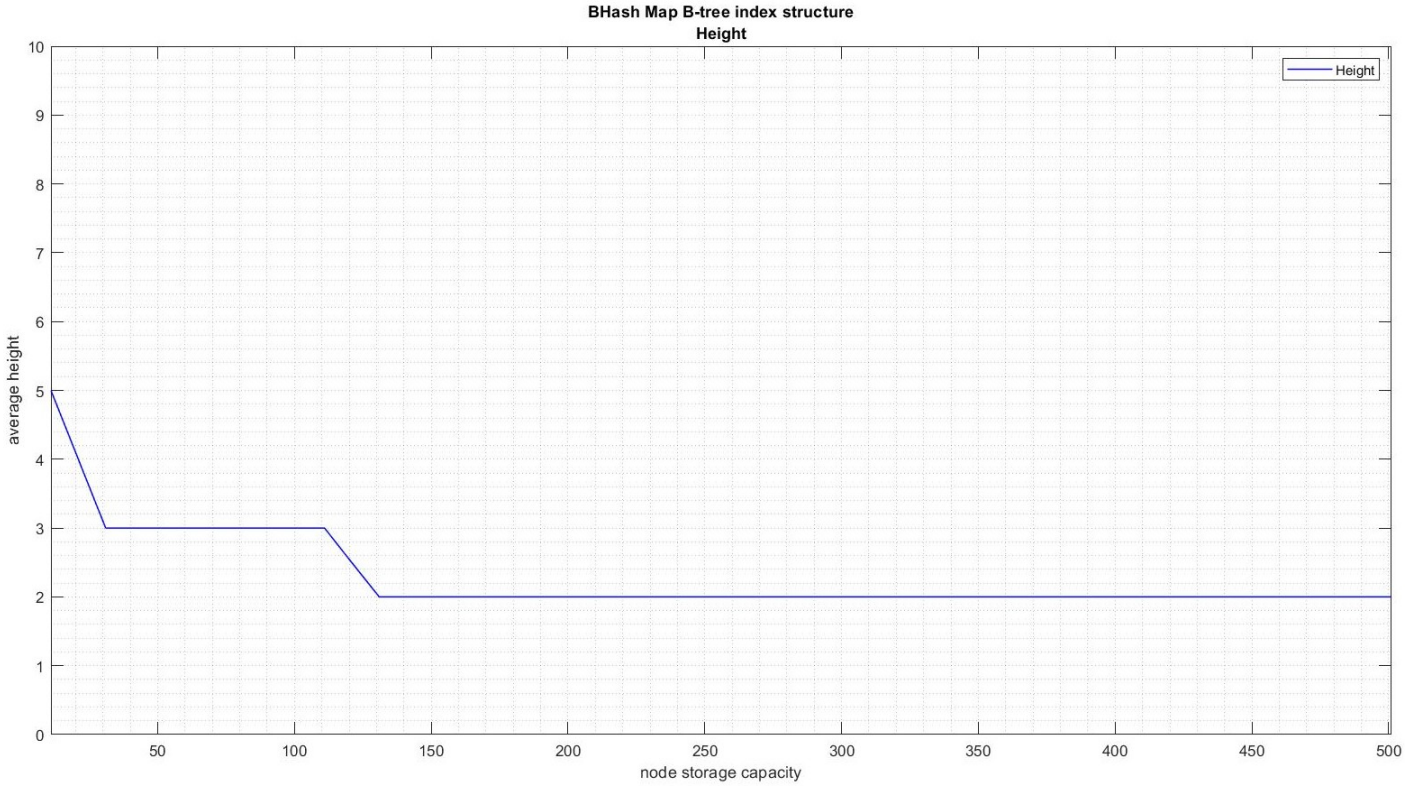


Fig. 5.42 represents the average B-Hash Map index structure B-tree height. The utilized records set is composed of records with primary key fields of string type.

Figure 5.42: Average B-Hash Map index structure B-tree height

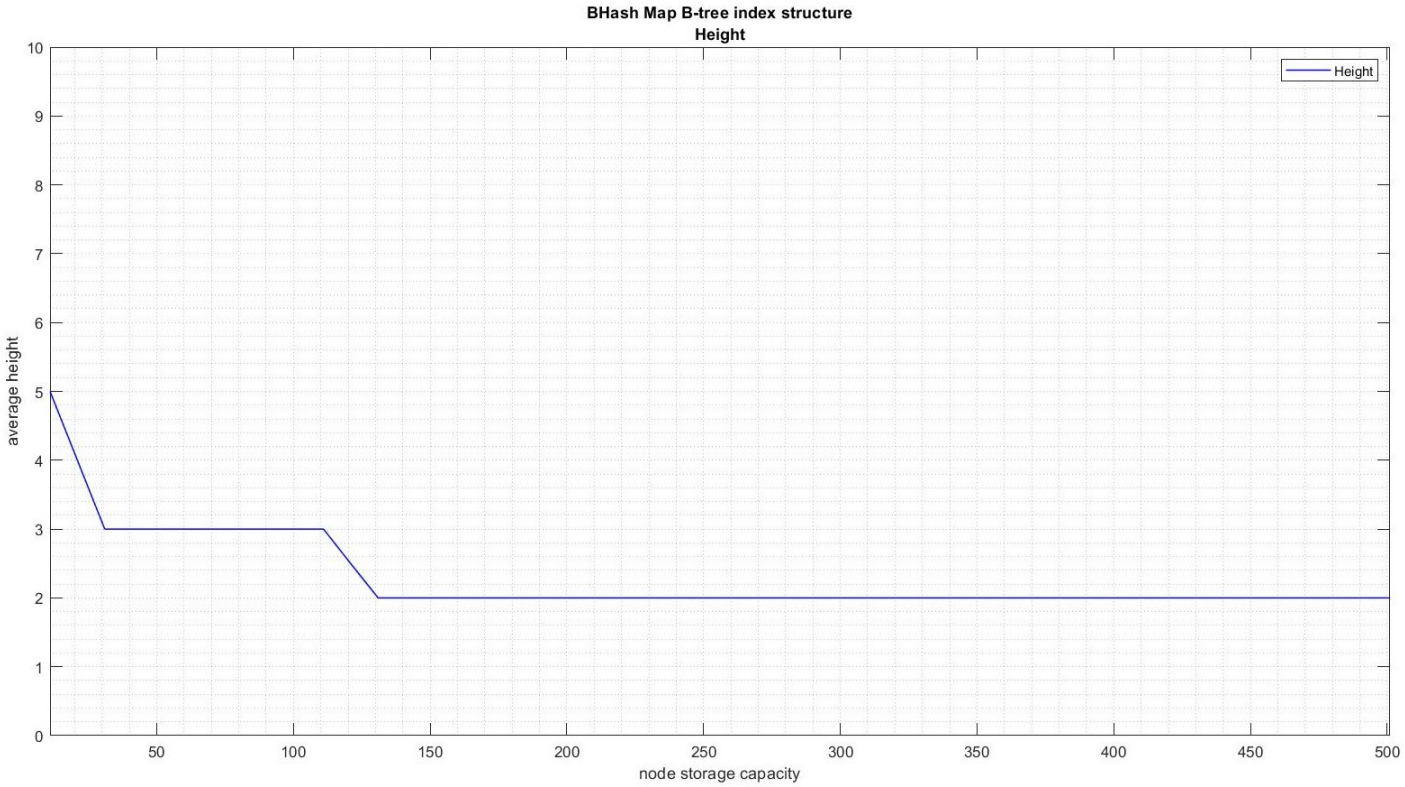


Fig. 5.43 represents the average B$^+$-Hash Map index structure B$^+$-tree height. The utilized records set is composed of records with primary key fields of integer type.

Figure 5.43: Average B$^+$-Hash Map index structure B$^+$-tree height

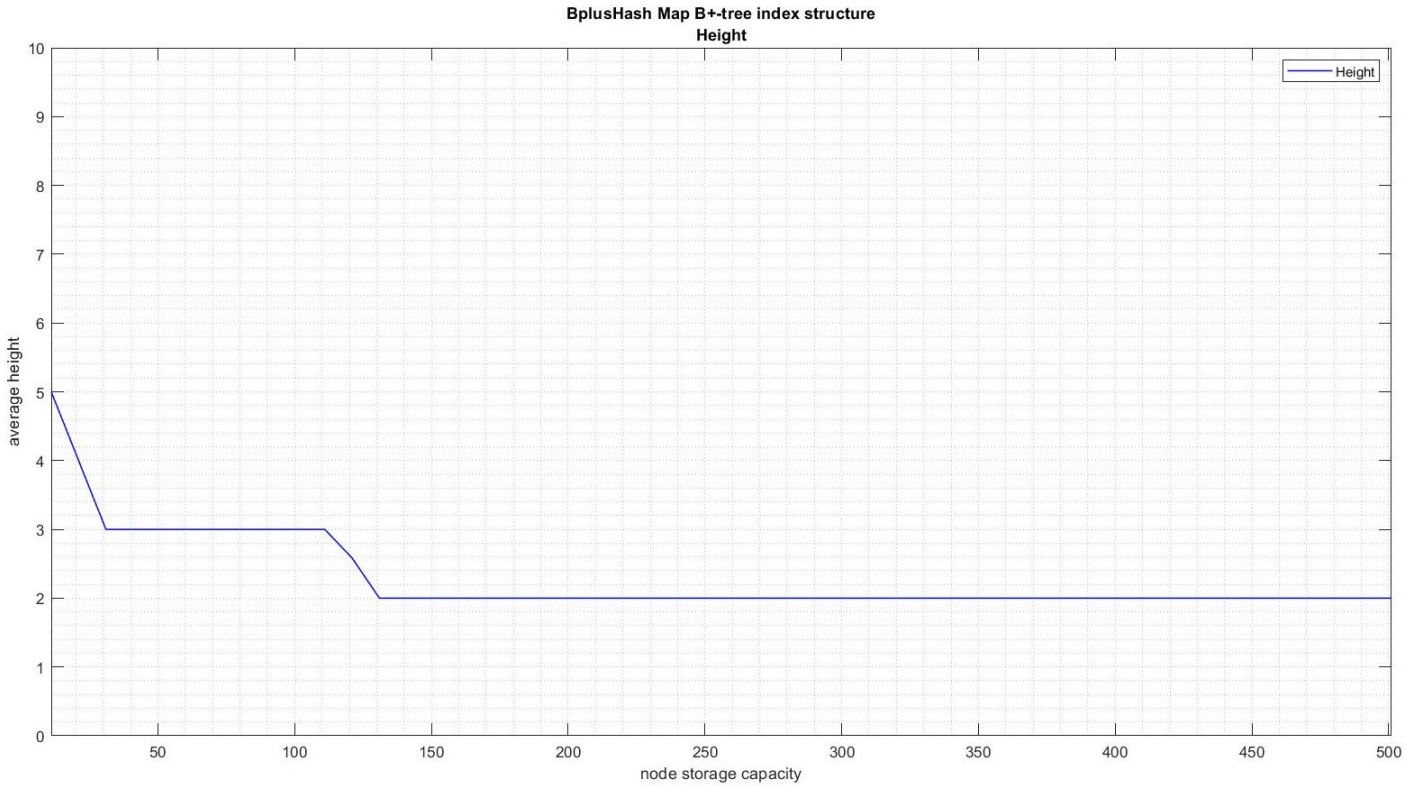
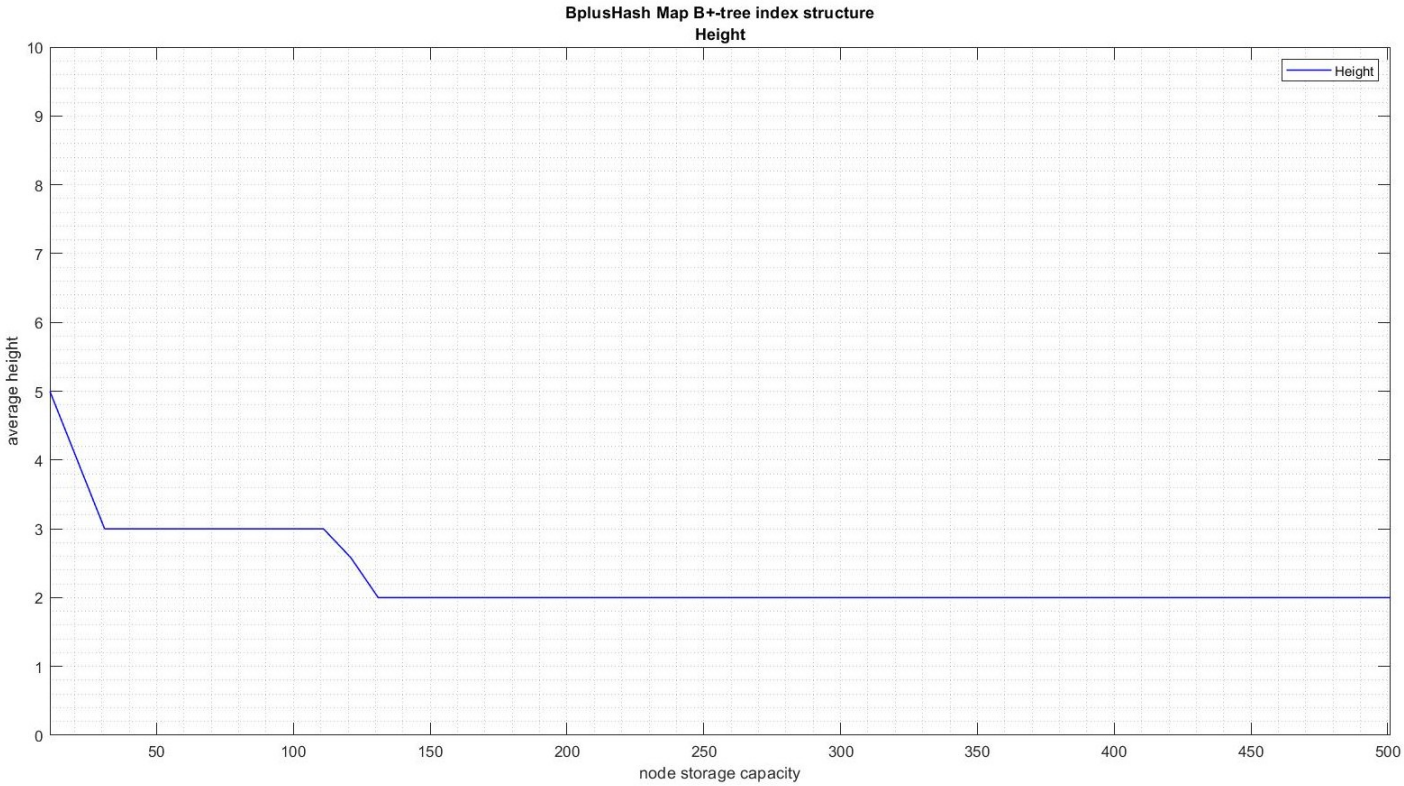


Fig. 5.44 represents the average B$^+$-Hash Map index structure B$^+$-tree height. The utilized records set consists of records with primary key fields of string type.

Figure 5.44: Average B$^+$-Hash Map index structure B$^+$-tree height



5.2.2 Computational process on real data

This computational process was conducted as an average time performance approximation of the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures insertion, deletion and selection functions execution time in a set of real data.

The utilized dataset consists of real anonymized data and constitutes a RDBMS relational table stored records set of $1,056,320$ bank transaction operations. These stored transnational operations are processes of crediting and debiting funds to customers bank accounts and funds transfer between banking companies - institutions. The dataset was constructed and minimized in order to be stored as a RDBMS table to which the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures are linked and store the relational table record references set. The RDBMS table stored records are composed of the following data parts:

- The transaction identifier - primary key field (string type).
- The customer bank account identifier (string type).
- The transaction operation type (string type).
  - Debit operation.
  - Credit operation.
- The transaction operation (string type).
  - Cash withdrawal operation.
  - Remittance to another bank operation.
  - Credit in cash operation.
  - Collection from another bank operation.
  - Credit card withdrawal operation.
- The transaction operation full date-time (string type).

A subset of $500,000$ records objects (part of the $1,056,320$ records dataset) is stored in a dynamic array structure in an ascending sorted arraignment - order based on the stored records transaction operation real date-time.

This $500,000$ records set references are stored in the B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures that are linked to the RDBMS relational table that contains these records utilizing the record insertion function of each individual index structure. Then the record references set insertion process time is measured and stored. In addition the internal and leaf nodes, the record references that are stored in the leaf and internal nodes of the B-tree and B$^+$-tree index structures and the structures height are measured and stored. For the B-Hash and B$^+$-Hash Map index structures B-tree and B$^+$-tree substructures the average nodes, nodes stored record references and height are measured. After the insertion process the record references set is selected from the B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures based on the primary key field of each record and the total selection process time is measured and stored. As all the record references are stored in the B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures the complete - full scan and selection process of all the stored record references of the index structures is preformed based on the auxiliary record field that is the same for all the records. The full selection process time is also measured and stored. Then a random reordering - rearrangement process of the input data set was repeatedly performed 100 times. Specifically each record that is stored in the dynamic array structure switches position with a randomly selected record that is stored in the array structure. This procedure was repeated 100 times for the whole stored records set rearrangement in order to randomly be constructed the set of $500,000$ records. Then the deletion of all records that had been inserted in the structures was performed and the completion time of the overall deletion operation was measured and stored. After the deletion process the dataset array structure is reconstructed - rearranged to its initial state to be reused.

This computational process was repeated $5,000$ times for each node capacity of the B-tree, B$^+$-tree, B-Hash and B$^+$-Hash Map index structures in a capacity range from 11 to 501 stored record references with an increment factor of 10. For each node capacity, the average of $5,000$ measurements taken for each function was calculated. The B-Hash and B$^+$-Hash Map index structures are composed of 10 B-tree and B$^+$-tree index structures. In addition, the total average execution time of each individual function was calculated.

The above set of computational - measurement processes was implemented separately for the B-tree, B^+-tree, B-Hash and B^+-Hash Map index structures in discrete computational procedures.

Computational study results of insertion and deletion functional processes average time performance

Fig. 5.45 represents the average time performance of the B-tree index structure insertion and deletion functional processes.

Figure 5.45: Functional process of B-tree index structure insertion and deletion average time performance

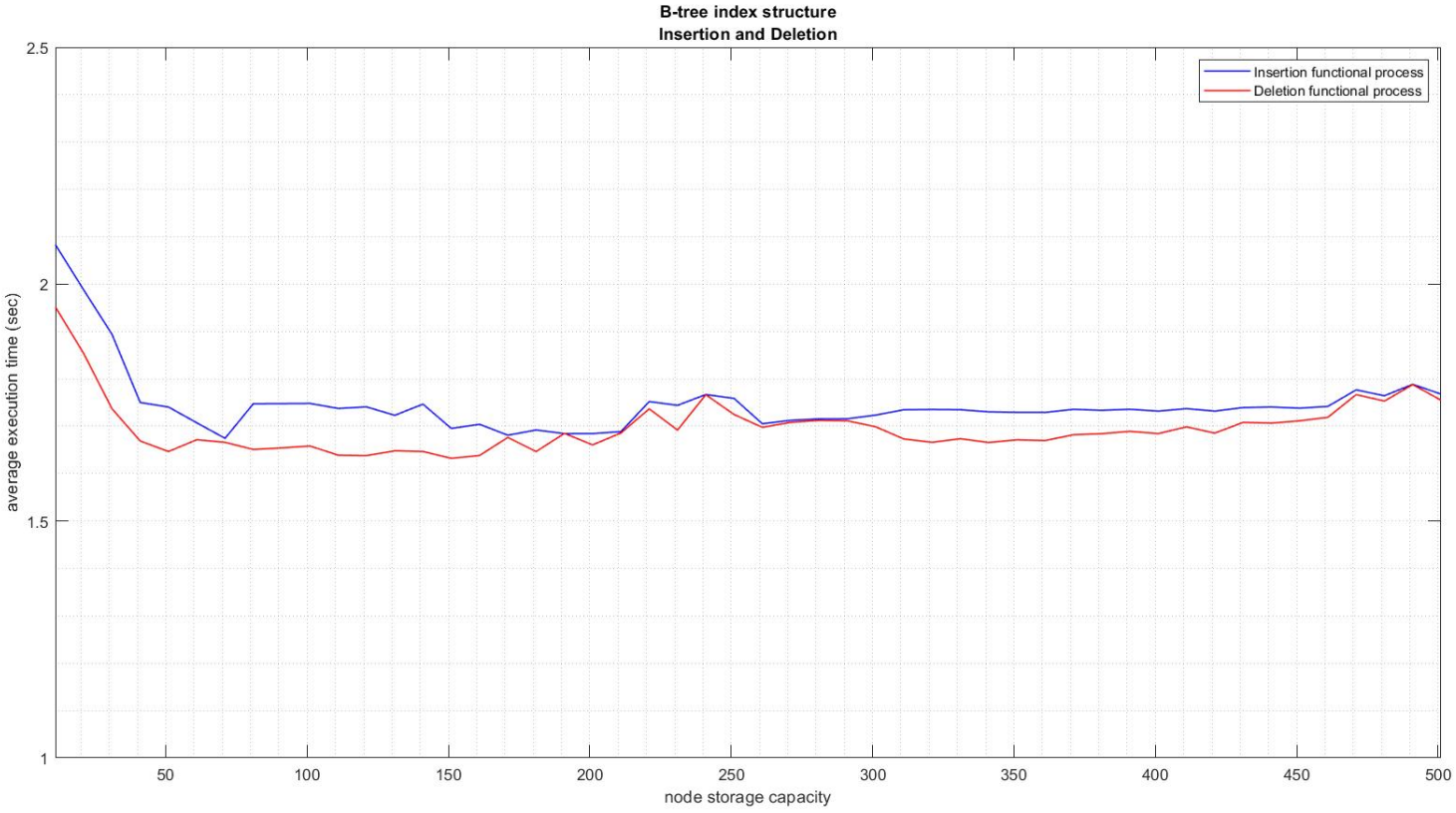


Fig. 5.46 represents the average time performance of the B^+-tree index structure insertion and deletion functional processes.

Figure 5.46: Functional process of B^+-tree index structure insertion and deletion average time performance

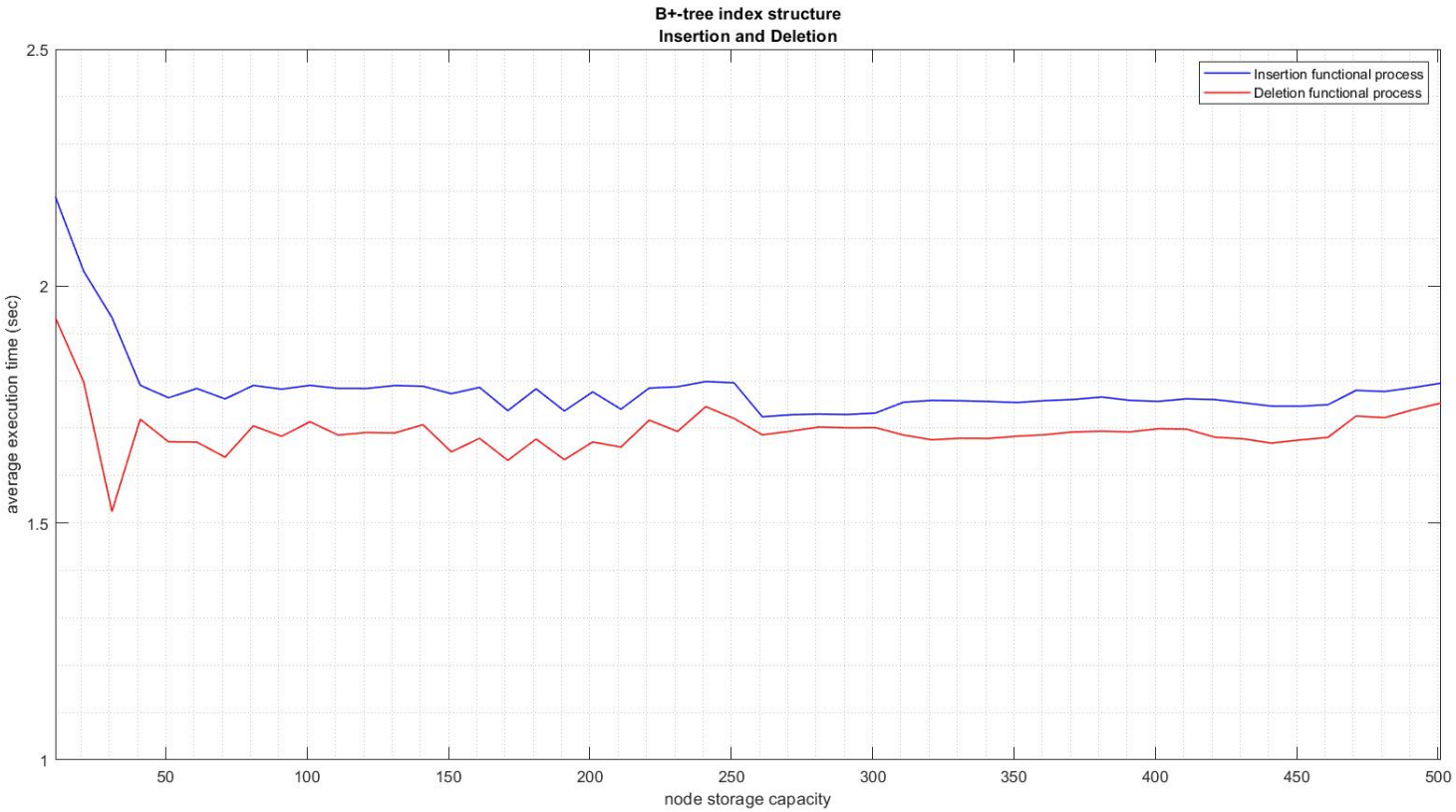


Fig. 5.47 represents the average time performance of the B-Hash Map index structure insertion and deletion functional processes.

Figure 5.47: Functional process of B-Hash Map index structure insertion and deletion average time performance

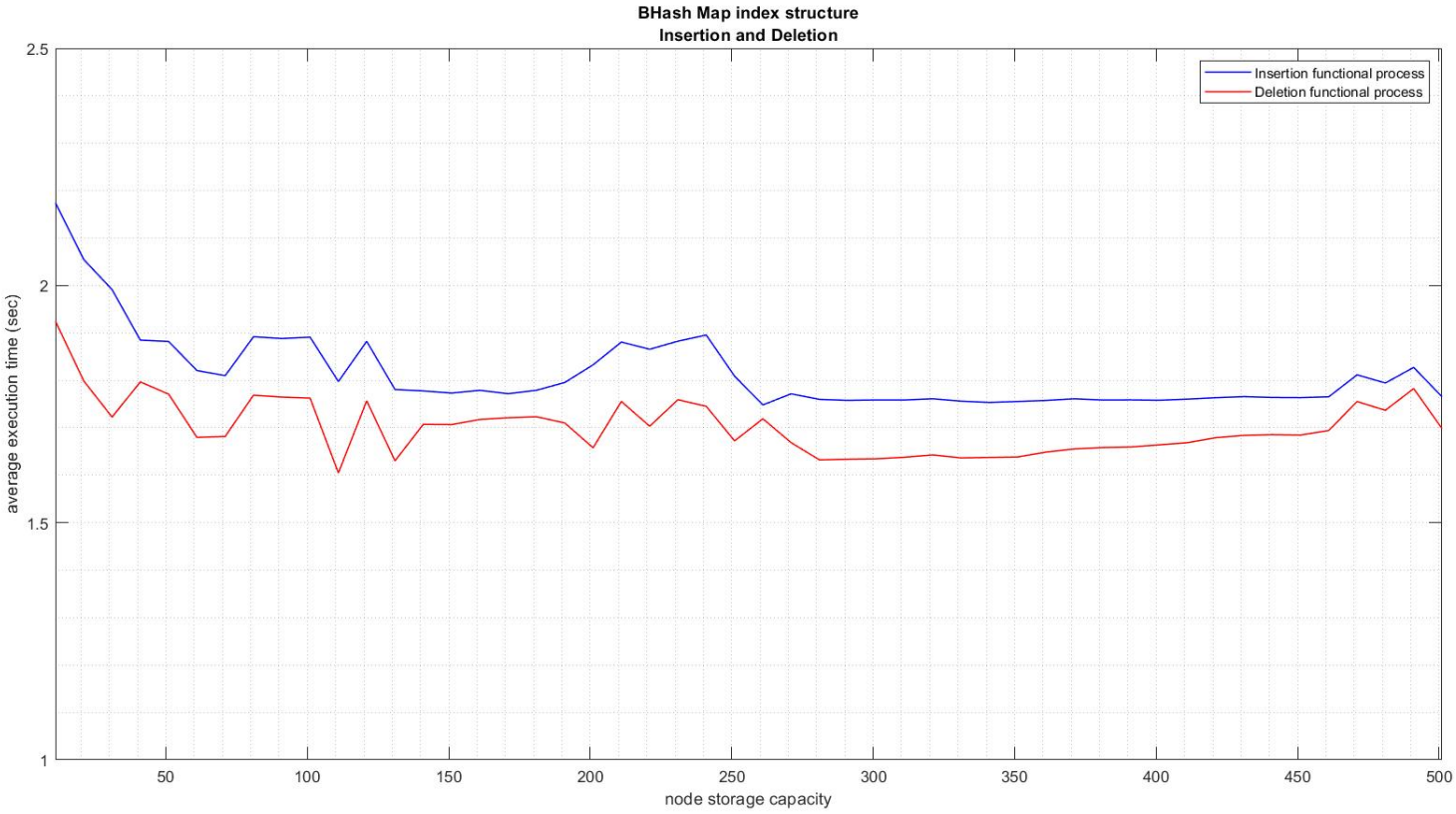


Fig. 5.48 represents the average time performance of the B$^+$-Hash Map index structure insertion and deletion functional processes.

Figure 5.48: Functional process of B$^+$-Hash Map index structure insertion and deletion average time performance

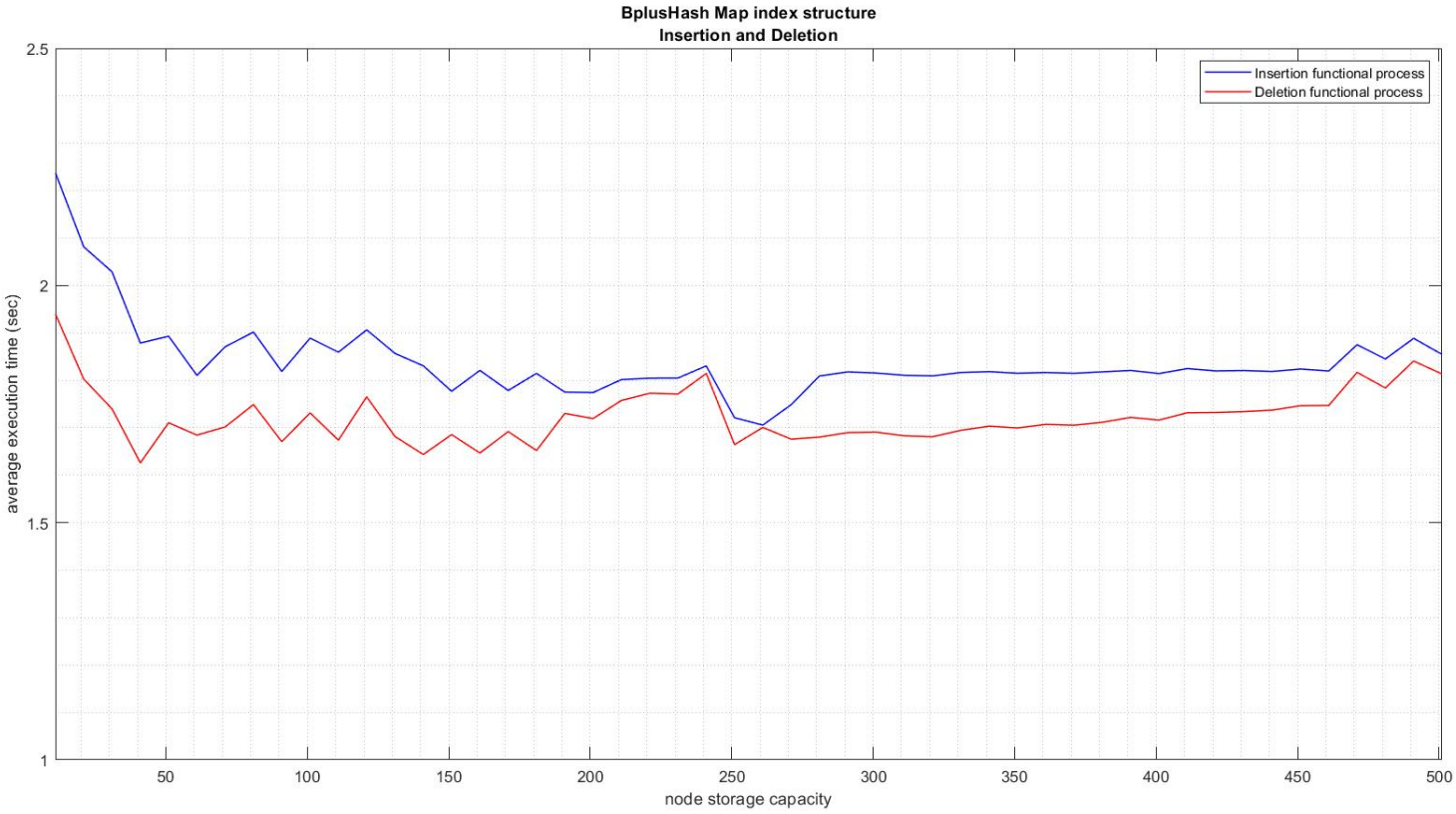


Fig. 5.49 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures insertion functional processes.

Figure 5.49: Functional processes of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures insertion average time performance

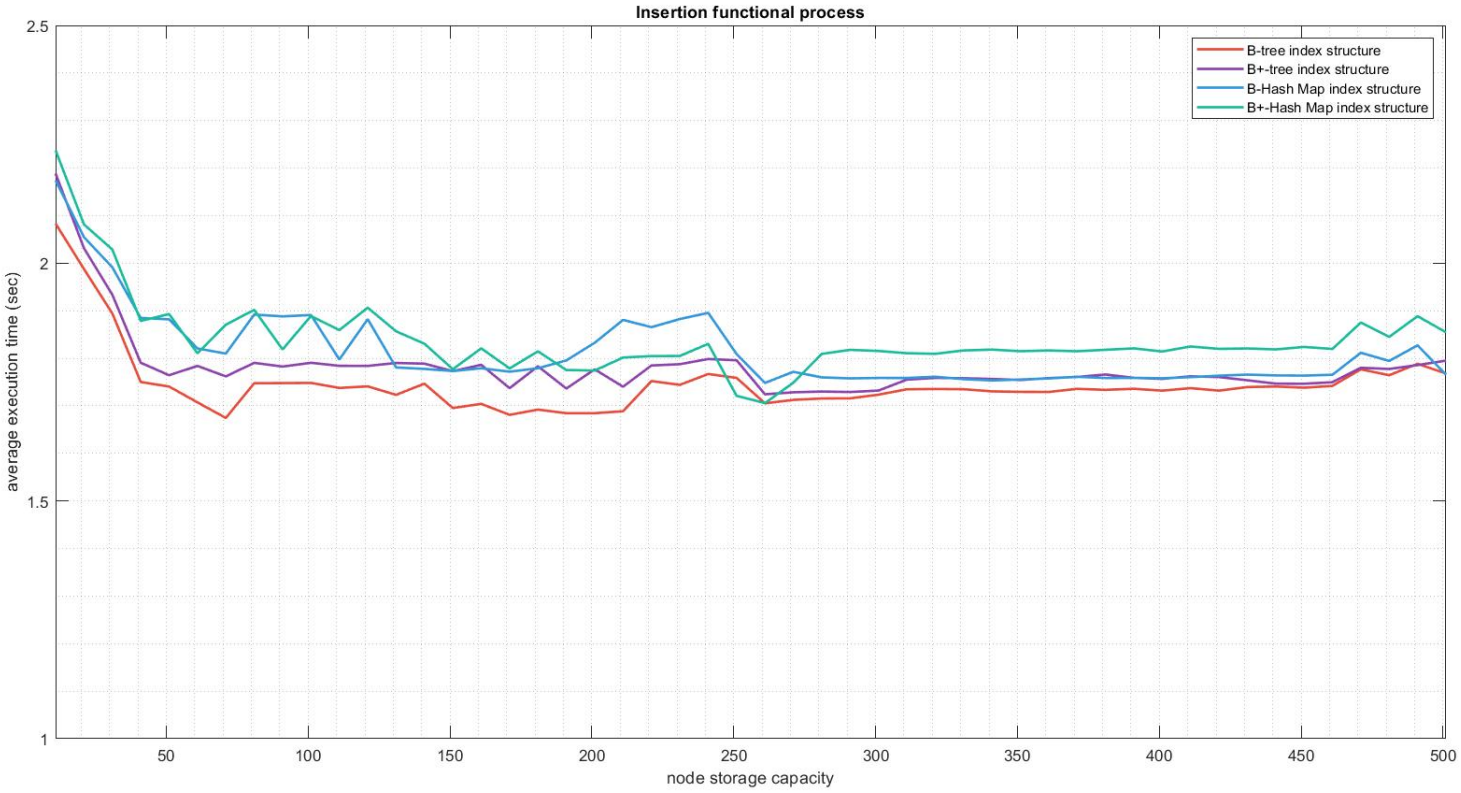
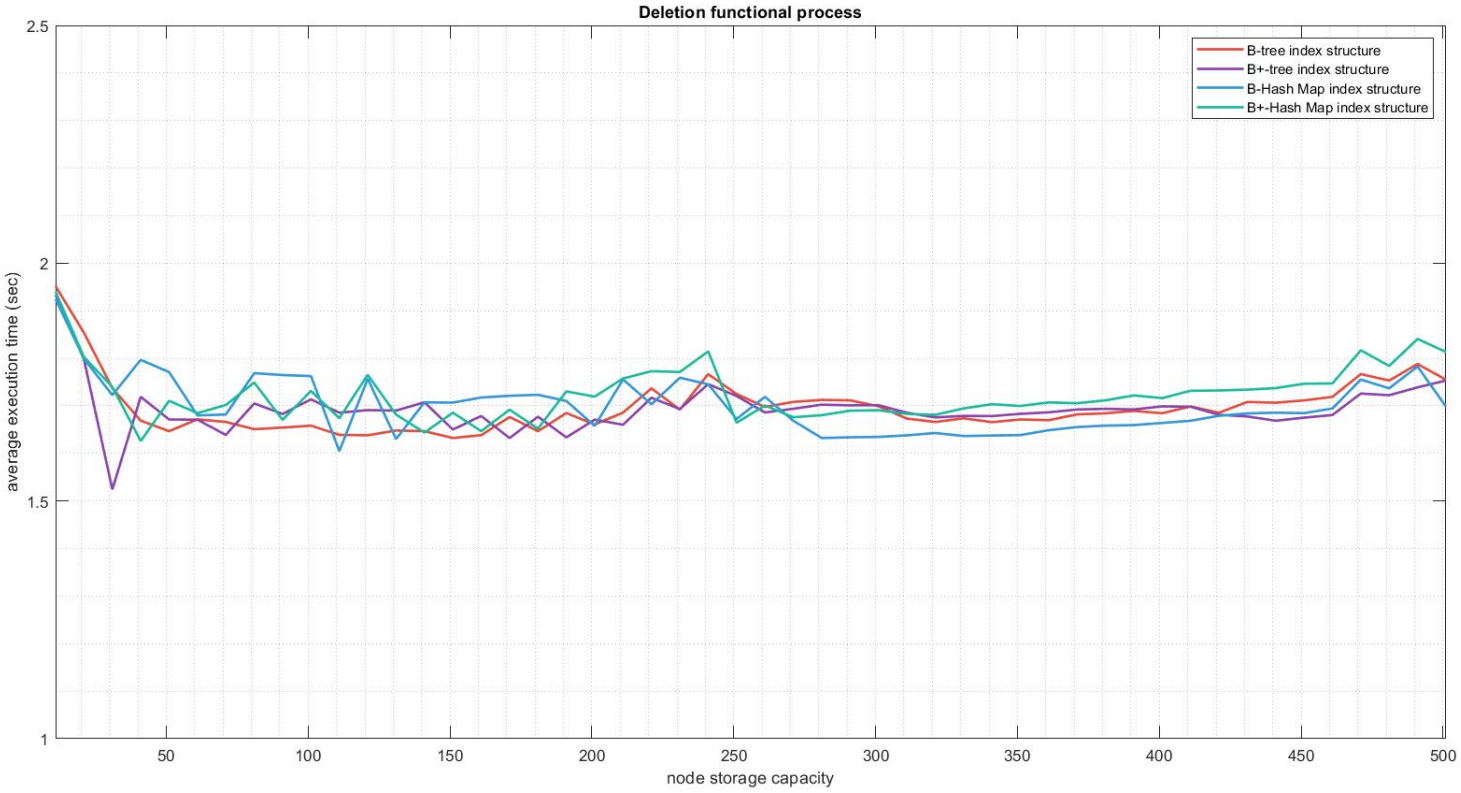


Fig. 5.50 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures deletion functional processes.

Figure 5.50: Functional processes of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures deletion average time performance



Computational study results of selection by primary key field and full scan - selection functional processes average time performance

Fig. 5.51 represents the average time performance of the B-tree index structure selection by primary key field and full scan - selection functional processes.

Figure 5.51: Functional process of B-tree index structure selection by primary key field and full scan - selection average time performance

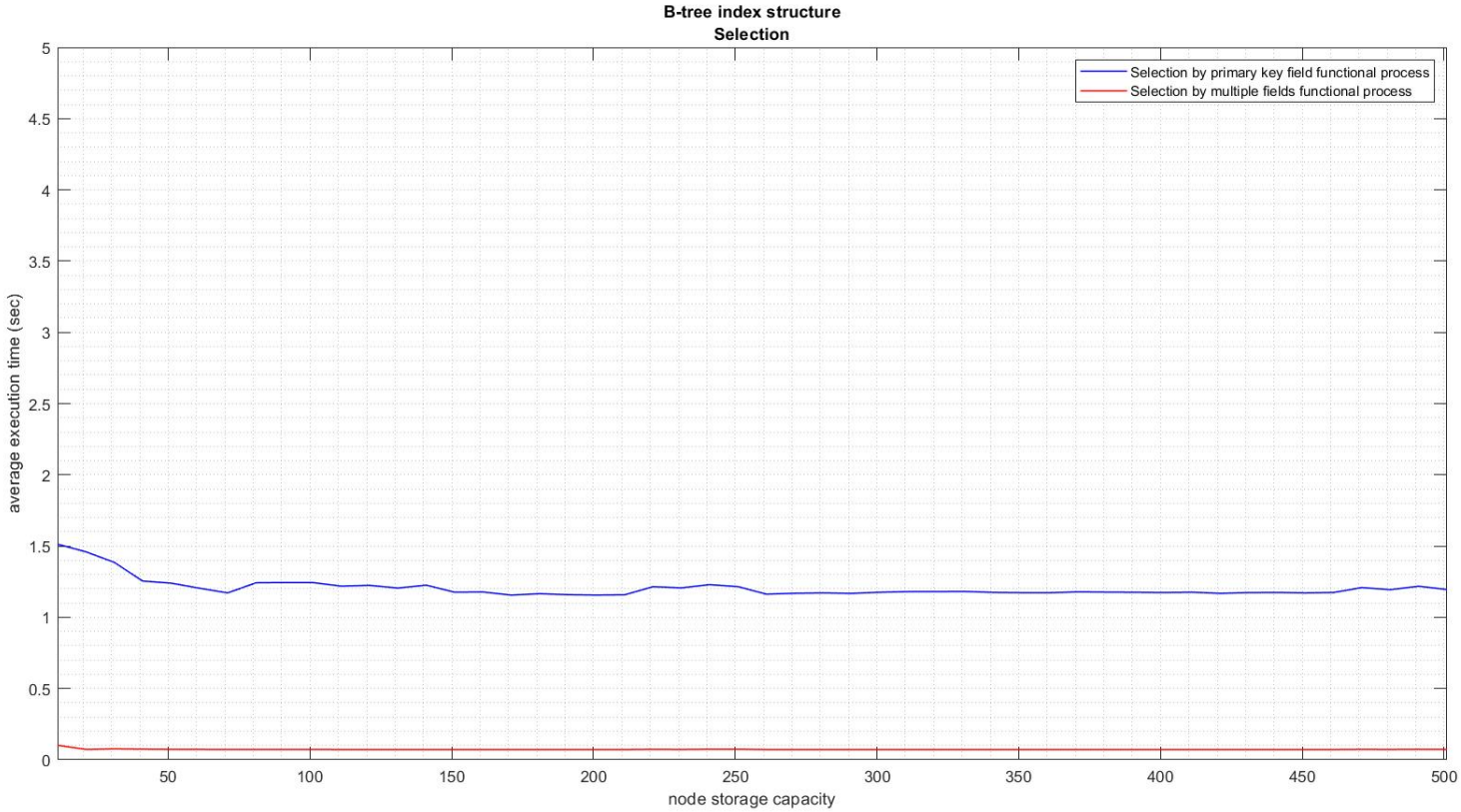


Fig. 5.52 represents the average time performance of the B$^+$-tree index structure selection by primary key field and full scan - selection functional processes.

Figure 5.52: Functional process of B$^+$-tree index structure selection by primary key field and full scan - selection average time performance

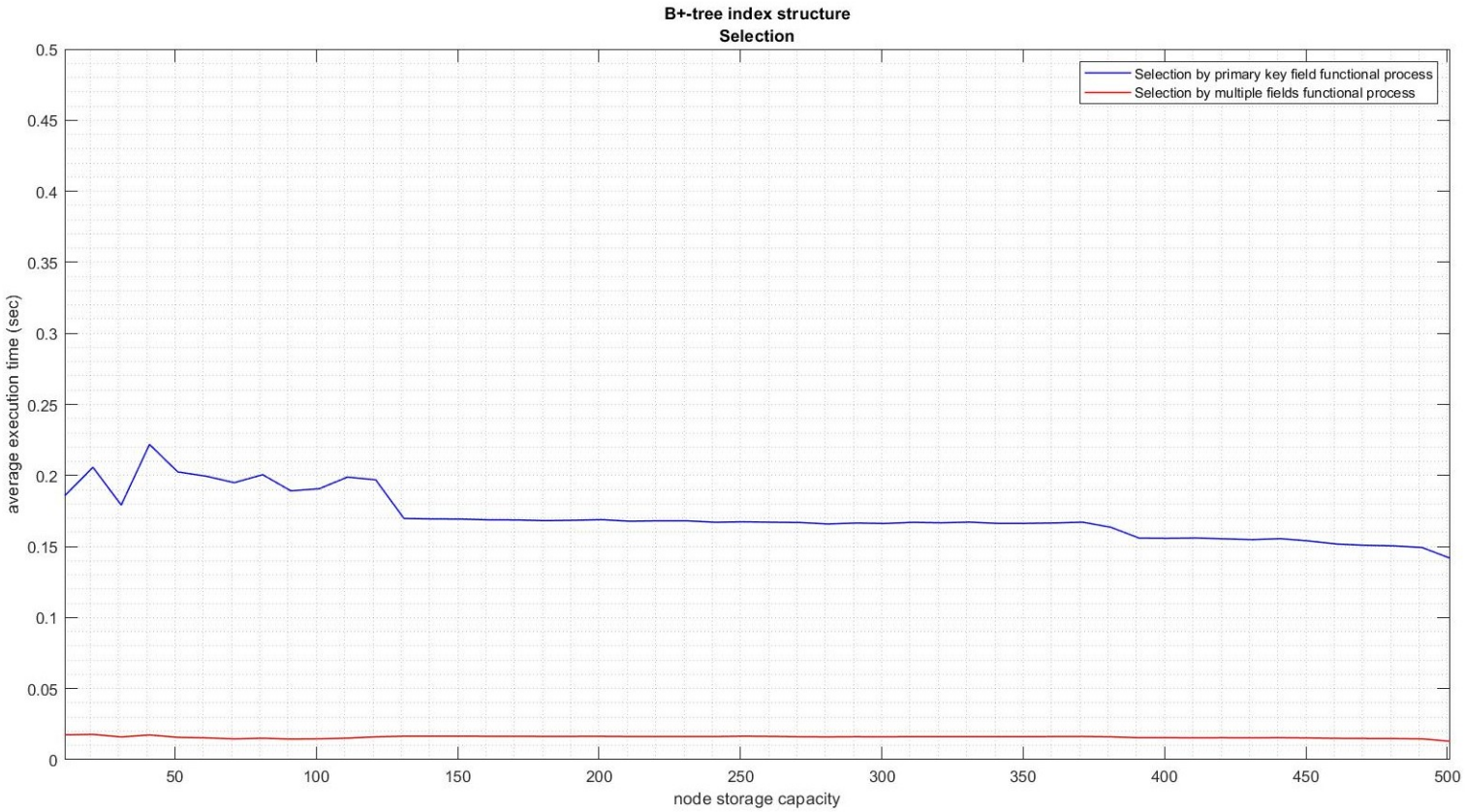


Fig. 5.53 represents the average time performance of the B-Hash Map index structure selection by primary key field and full scan - selection functional processes.

Figure 5.53: Functional process of B-Hash Map index structure selection by primary key field and full scan - selection average time performance

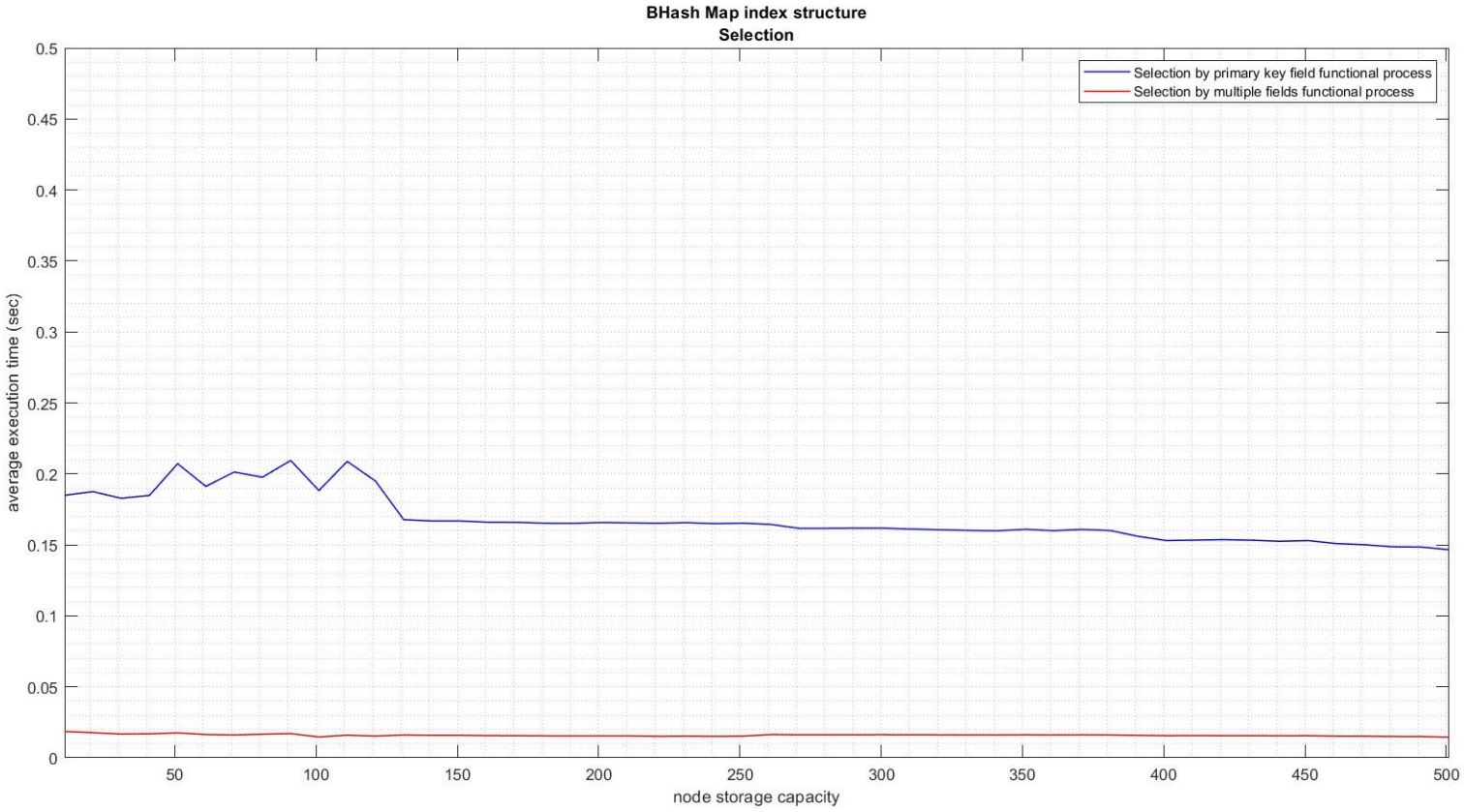


Fig. 5.54 represents the average time performance of the B$^+$-Hash Map index structure selection by primary key field and full scan - selection functional processes.

Figure 5.54: Functional process of B$^+$-Hash Map index structure selection by primary key field and full scan - selection average time performance

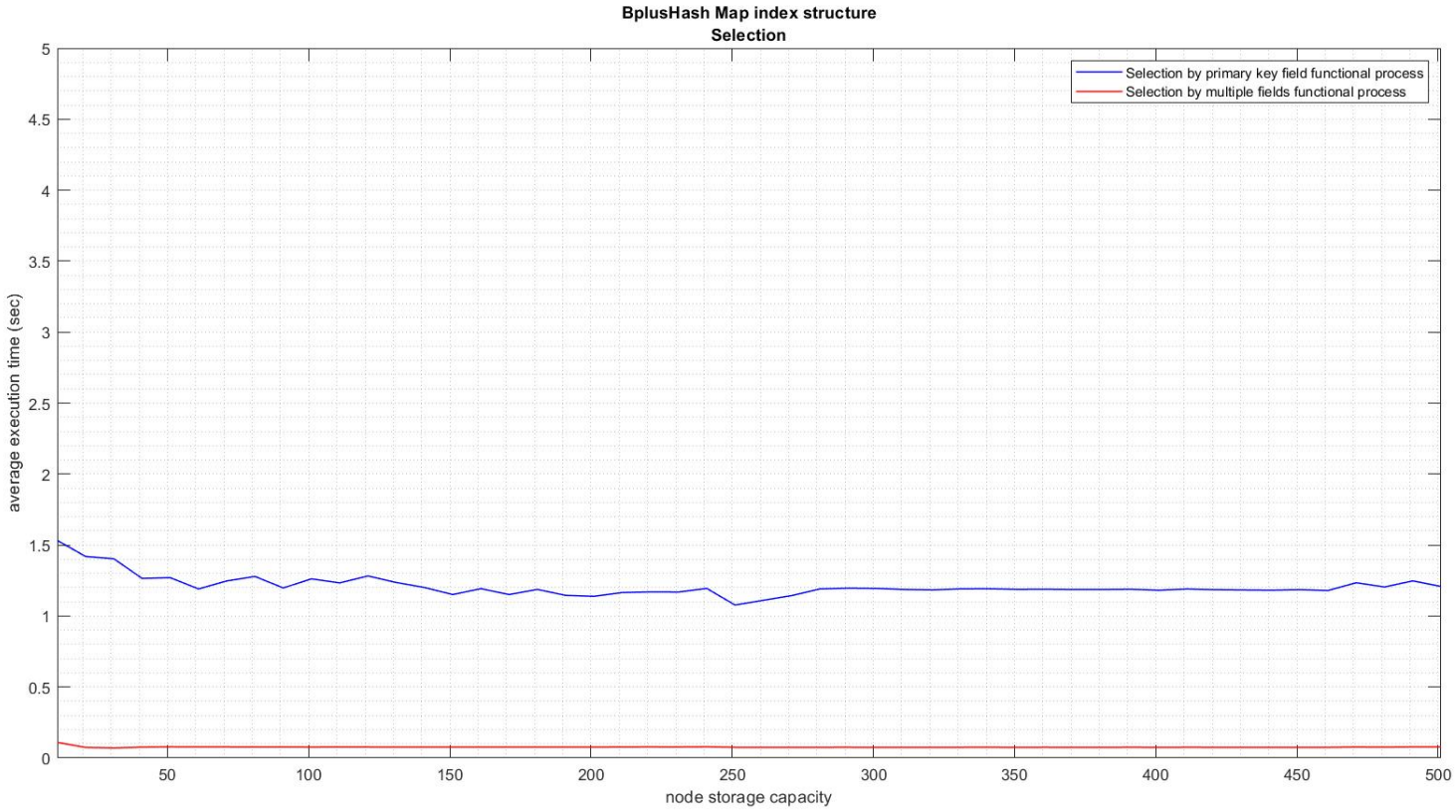


Fig. 5.55 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures selection by primary key field functional processes.

Figure 5.55: Functional processes of B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures selection by primary key field average time performance

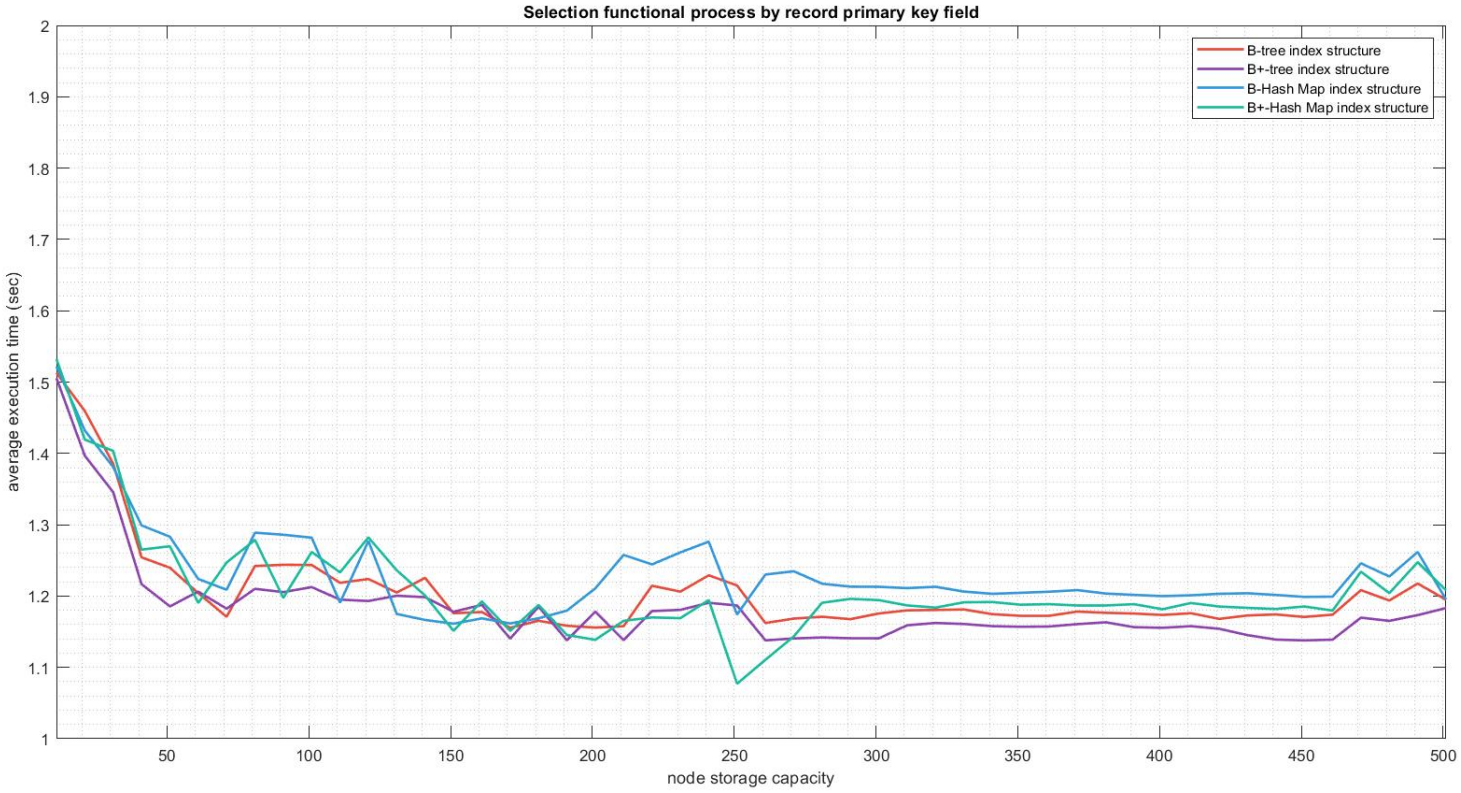
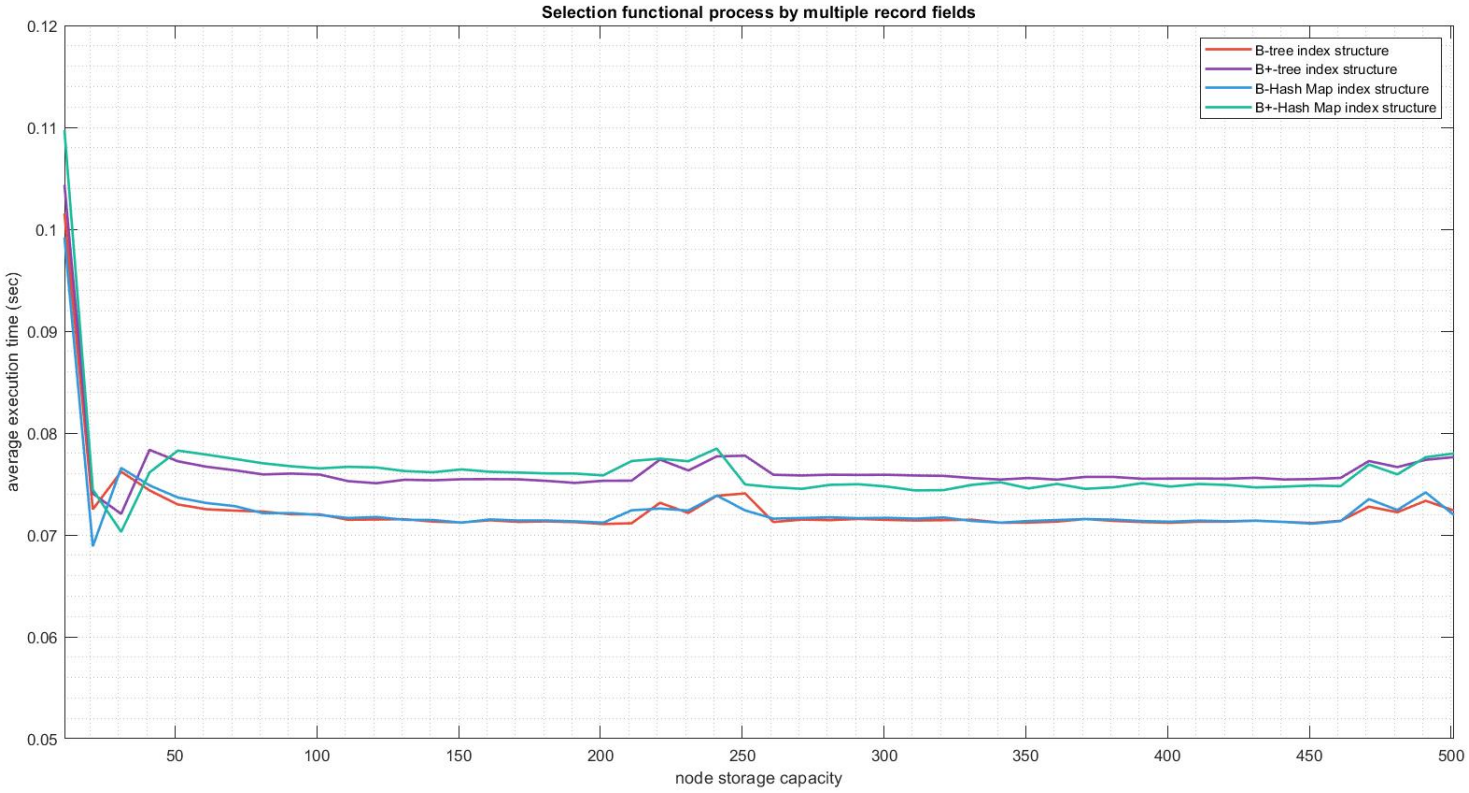


Fig. 5.56 represents the average time performance of the B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures full records scan and selection by multiple record fields functional processes.

Figure 5.56: Functional processes of B-tree, B$^+$-tree, B-Hash Map and B$^+$-Hash Map index structures full records scan and selection by multiple record fields average time performance



Computational study results of average internal and leaf nodes distribution in the index structures

Fig. 5.57 represents the average structural distribution of the B-tree index structure nodes in internal and leaf nodes.

Figure 5.57: Average structural distribution of the B-tree index structure nodes in internal and leaf nodes

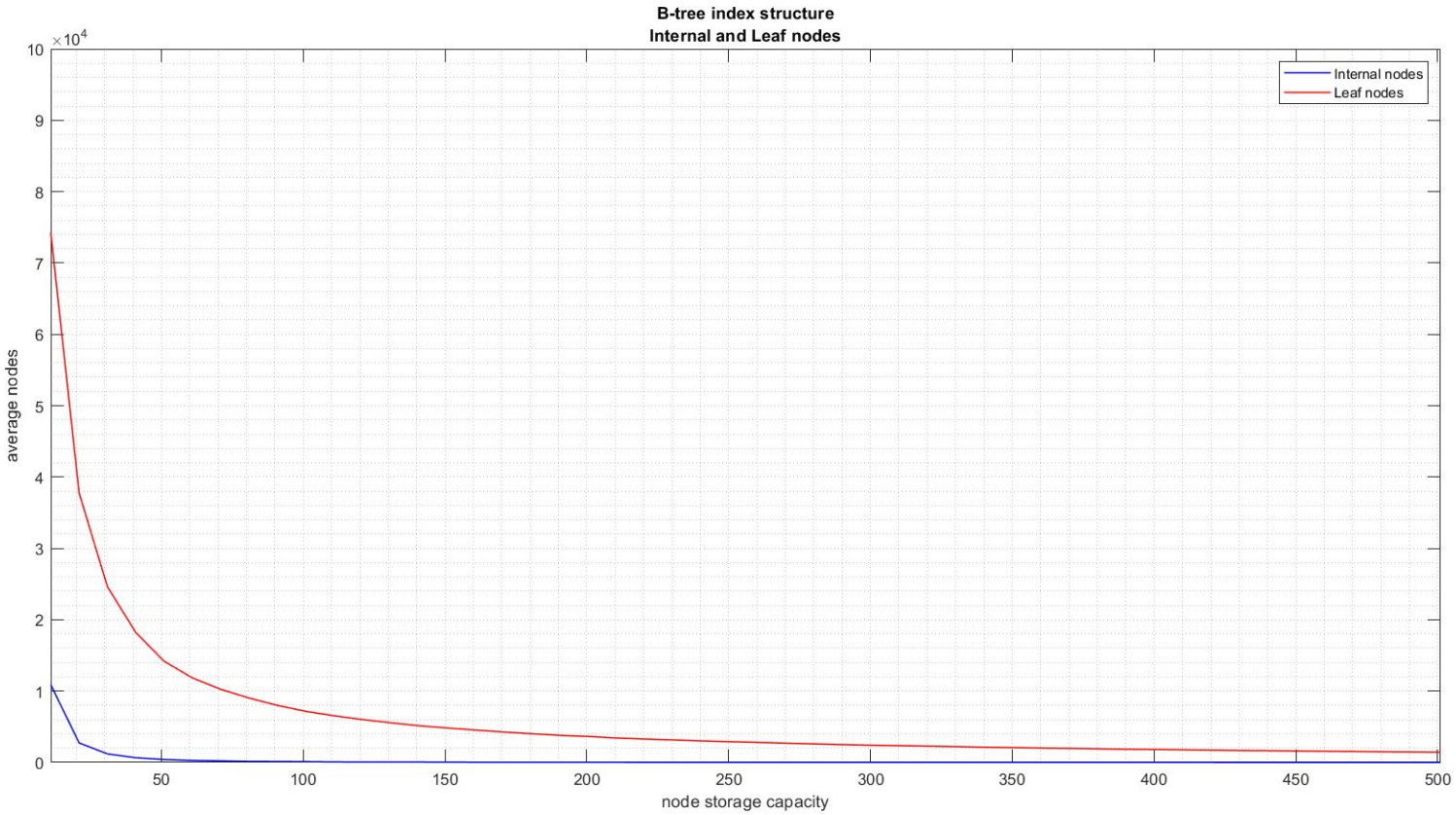


Fig. 5.58 represents the average structural distribution of the B$^+$-tree index structure nodes in internal and leaf nodes.

Figure 5.58: Average structural distribution of the B$^+$-tree index structure nodes in internal and leaf nodes

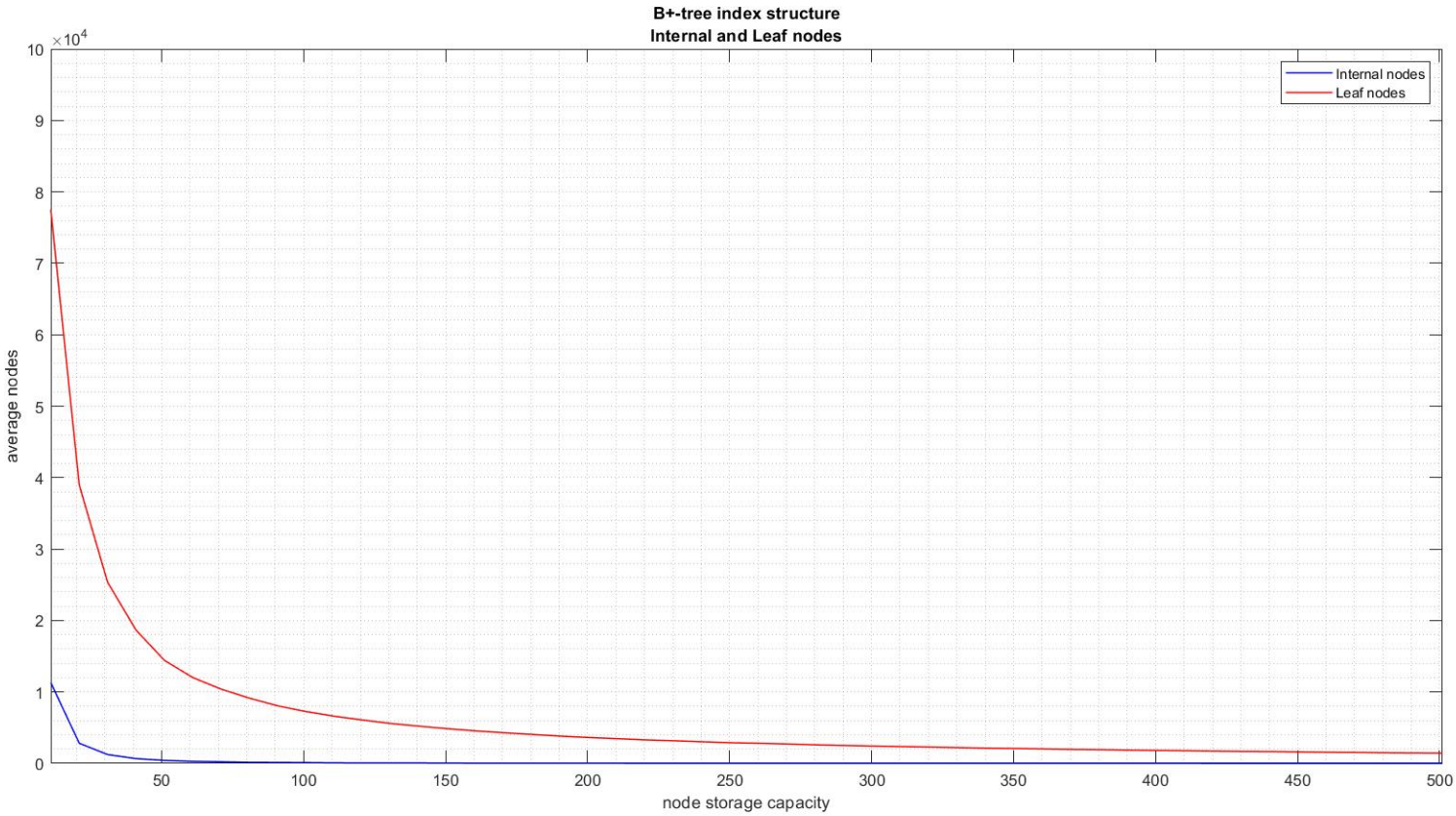


Fig. 5.59 represents the average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes.

Figure 5.59: Average structural distribution of the B-Hash Map index structure B-tree nodes in internal and leaf nodes

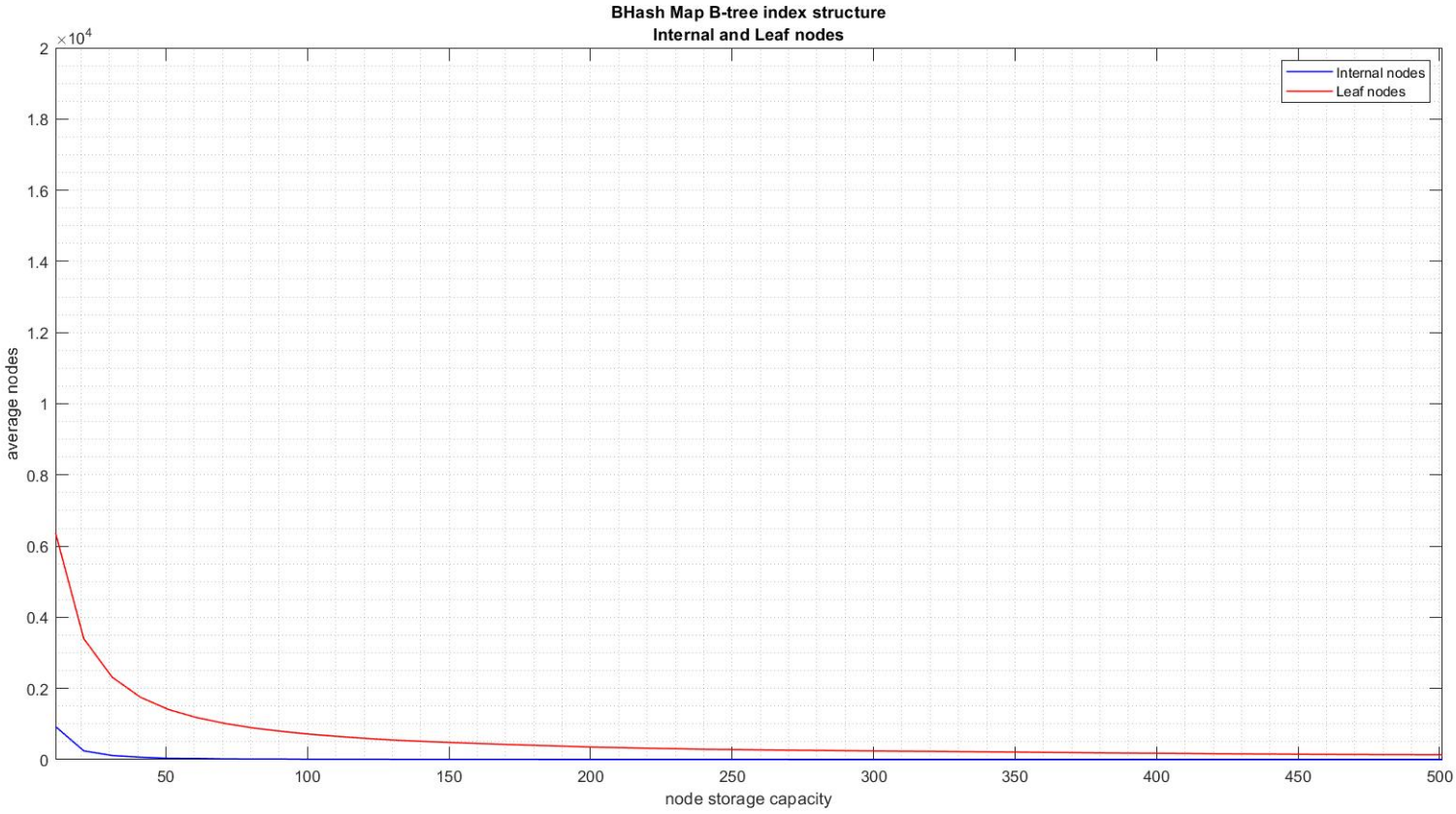
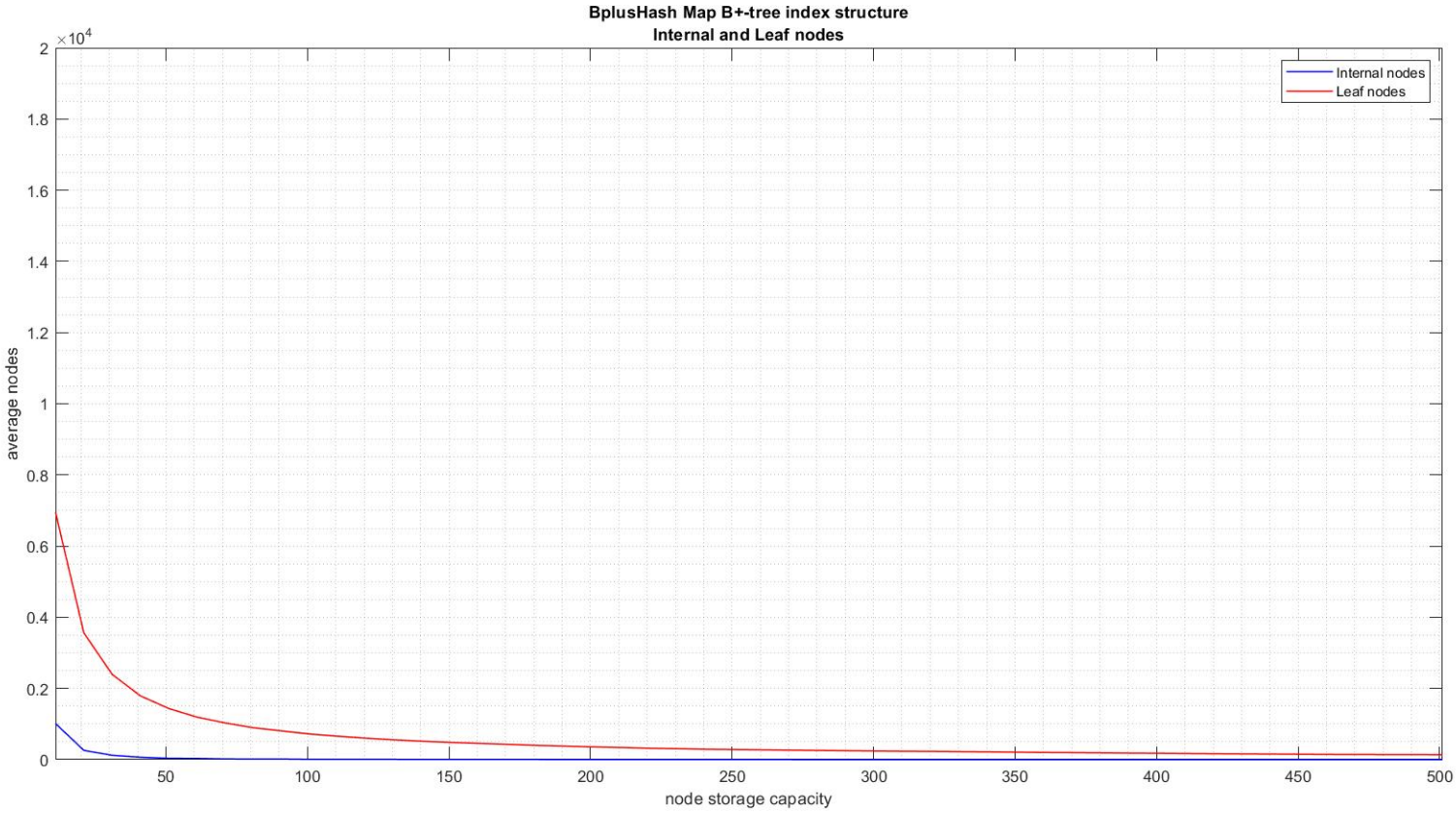


Fig. 5.60 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes.

Figure 5.60: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes in internal and leaf nodes



Computational study results of average internal and leaf nodes stored records distribution in the index structures

Fig. 5.61 represents the average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes.

Figure 5.61: Average structural distribution of the B-tree index structure nodes stored records in internal and leaf nodes

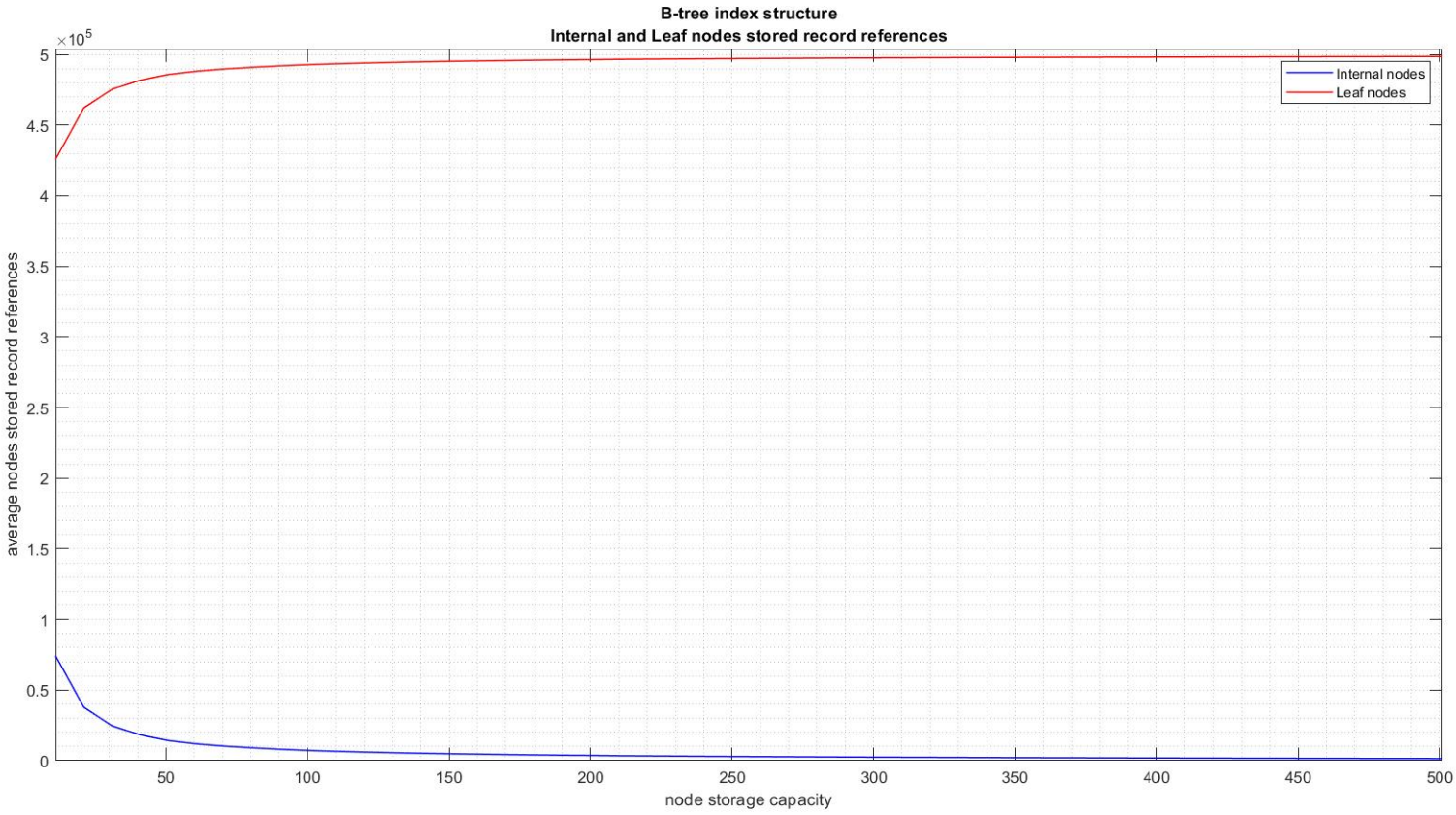


Fig. 5.62 represents the average structural distribution of the B^+-tree index structure nodes stored records in internal and leaf nodes.

Figure 5.62: Average structural distribution of the B^+-tree index structure nodes stored records in internal and leaf nodes

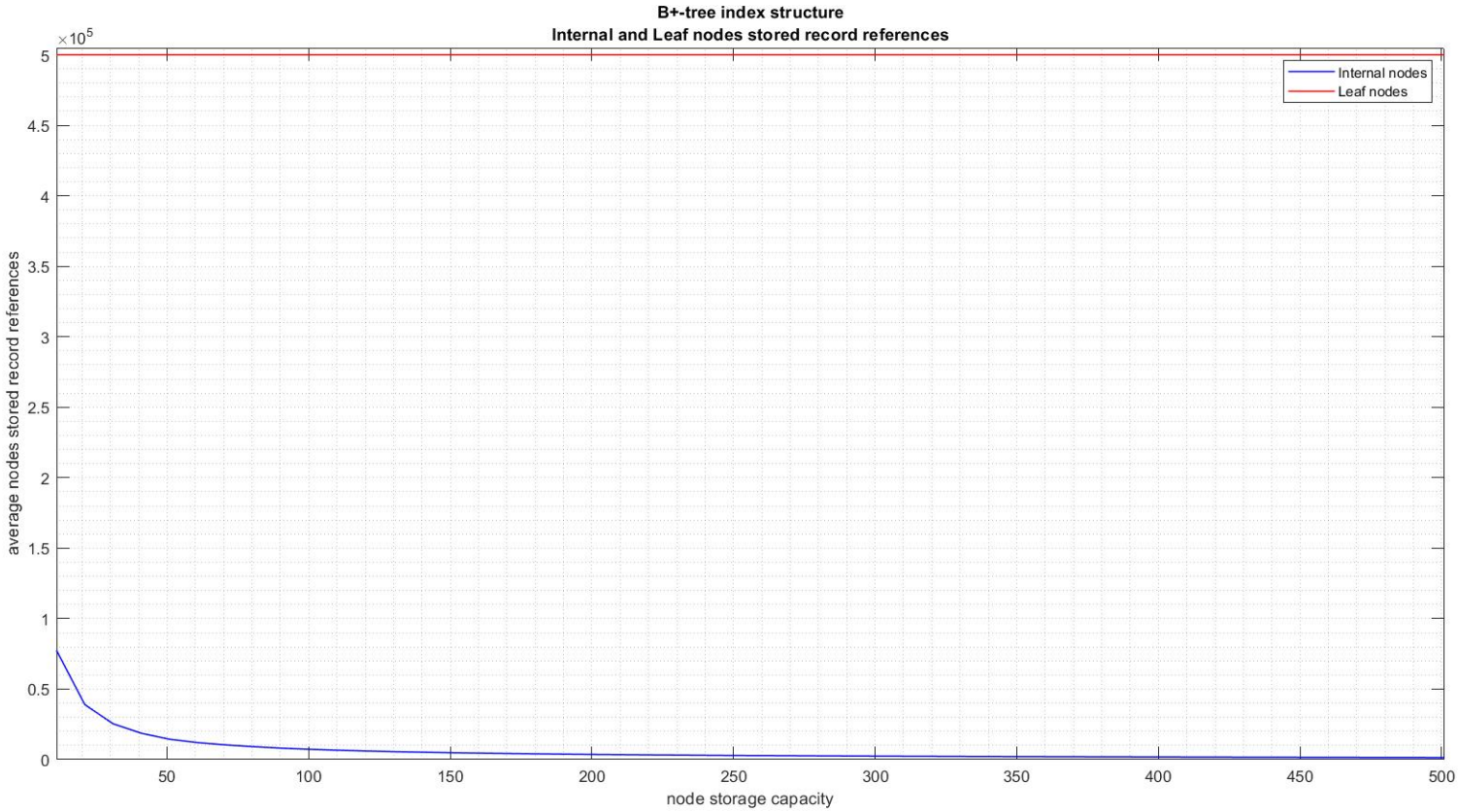


Fig. 5.63 represents the average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes.

Figure 5.63: Average structural distribution of the B-Hash Map index structure B-tree nodes stored records in internal and leaf nodes

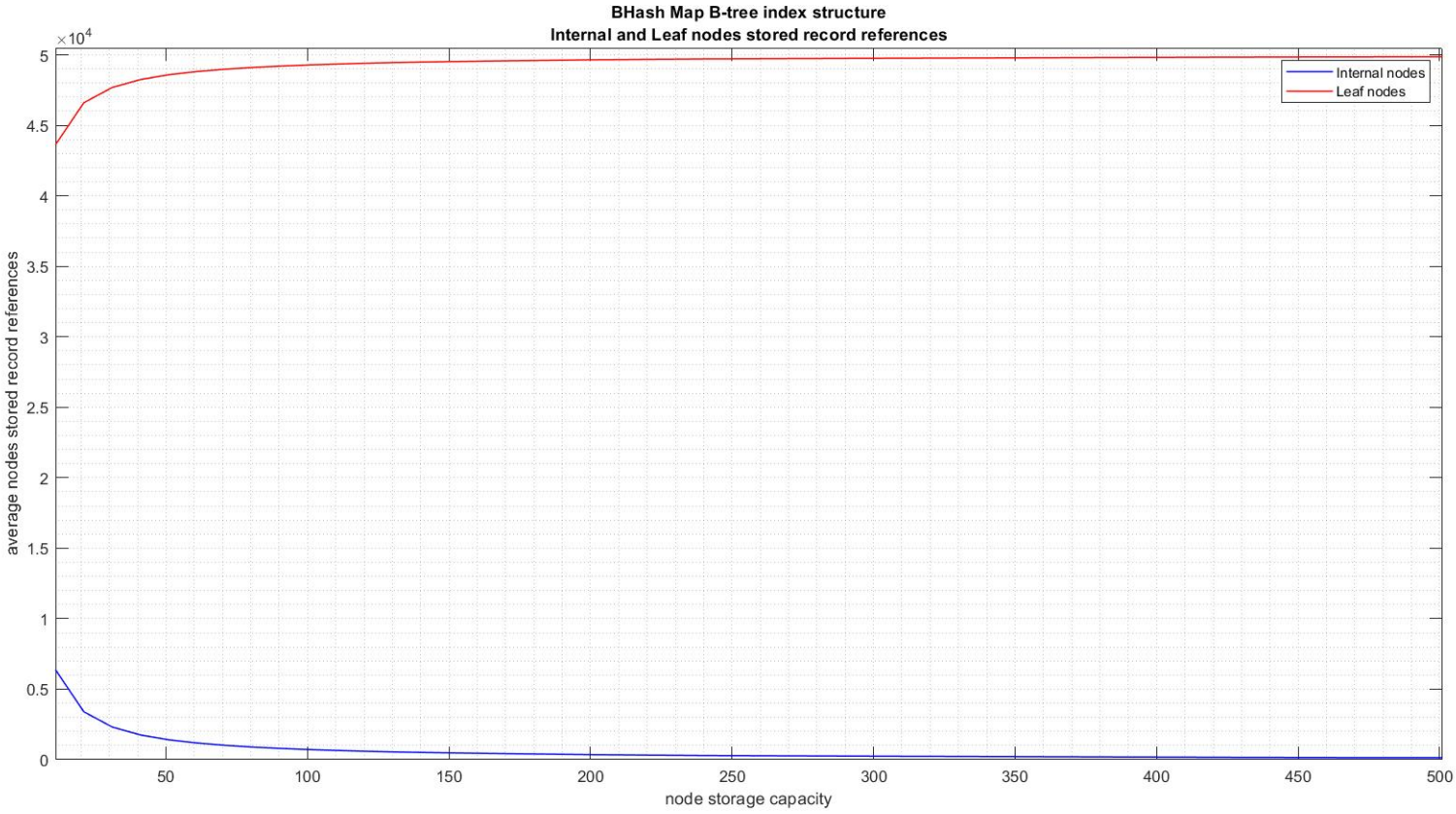
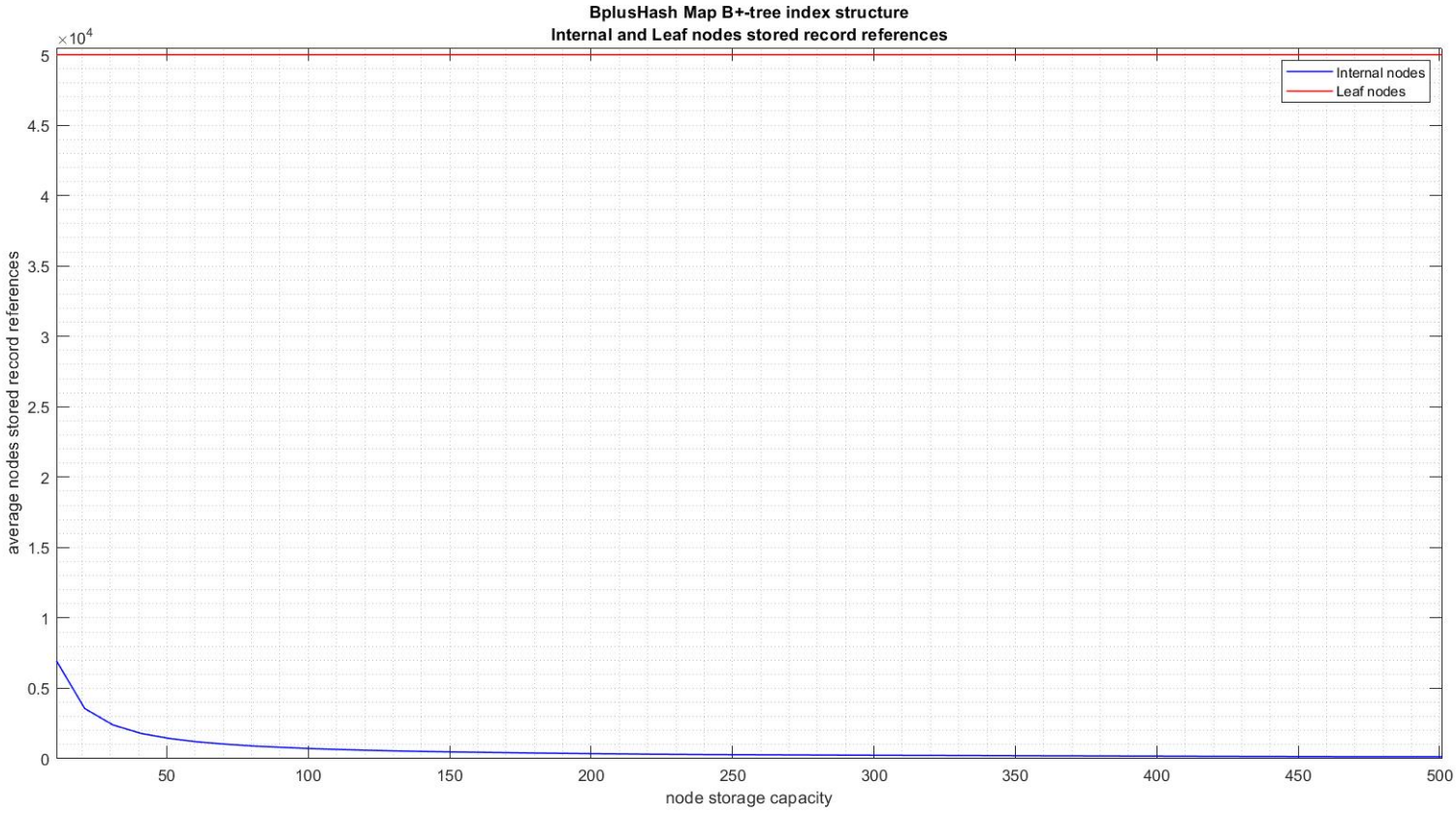


Fig. 5.64 represents the average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes.

Figure 5.64: Average structural distribution of the B⁺-Hash Map index structure B⁺-tree nodes stored records in internal and leaf nodes



Computational study results of average height

Fig. 5.65 represents the average B-tree index structure height.

Figure 5.65: Average B-tree index structure height

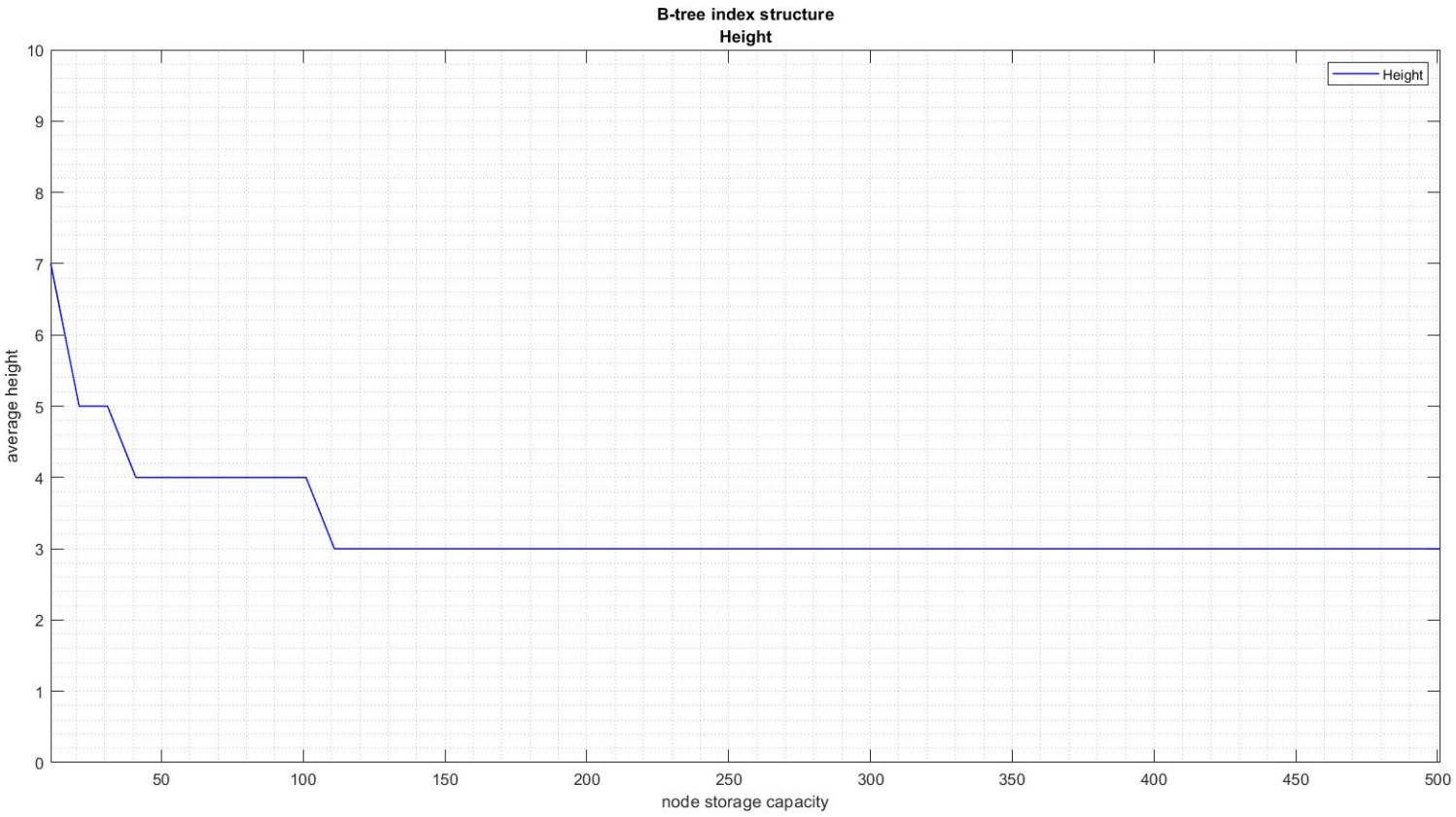


Fig. 5.66 represents the average B^+-tree index structure height.

Figure 5.66: Average B^+-tree index structure height

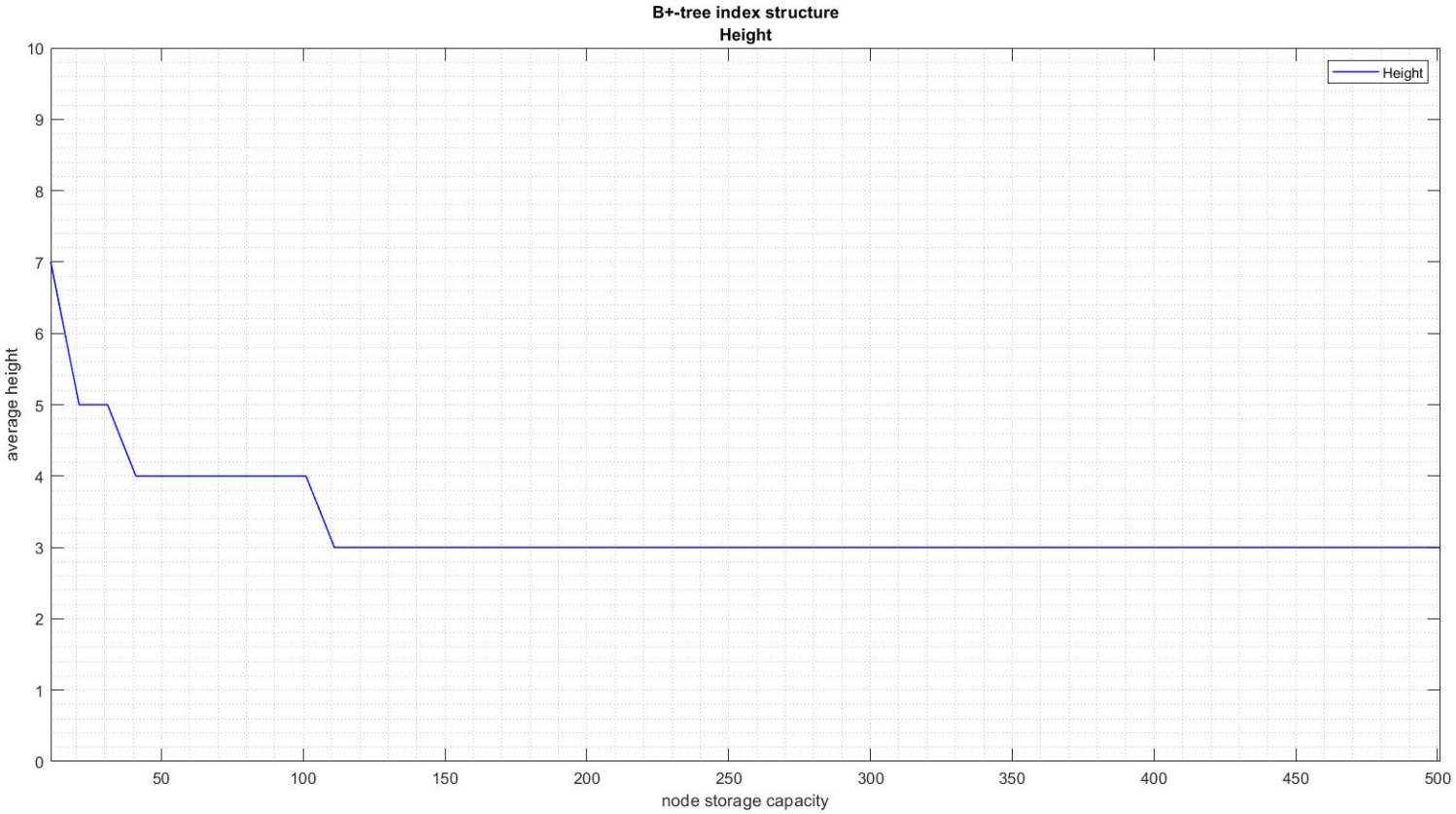


Fig. 5.67 represents the average B-Hash Map index structure B-tree height.

Figure 5.67: Average B-Hash Map index structure B-tree height

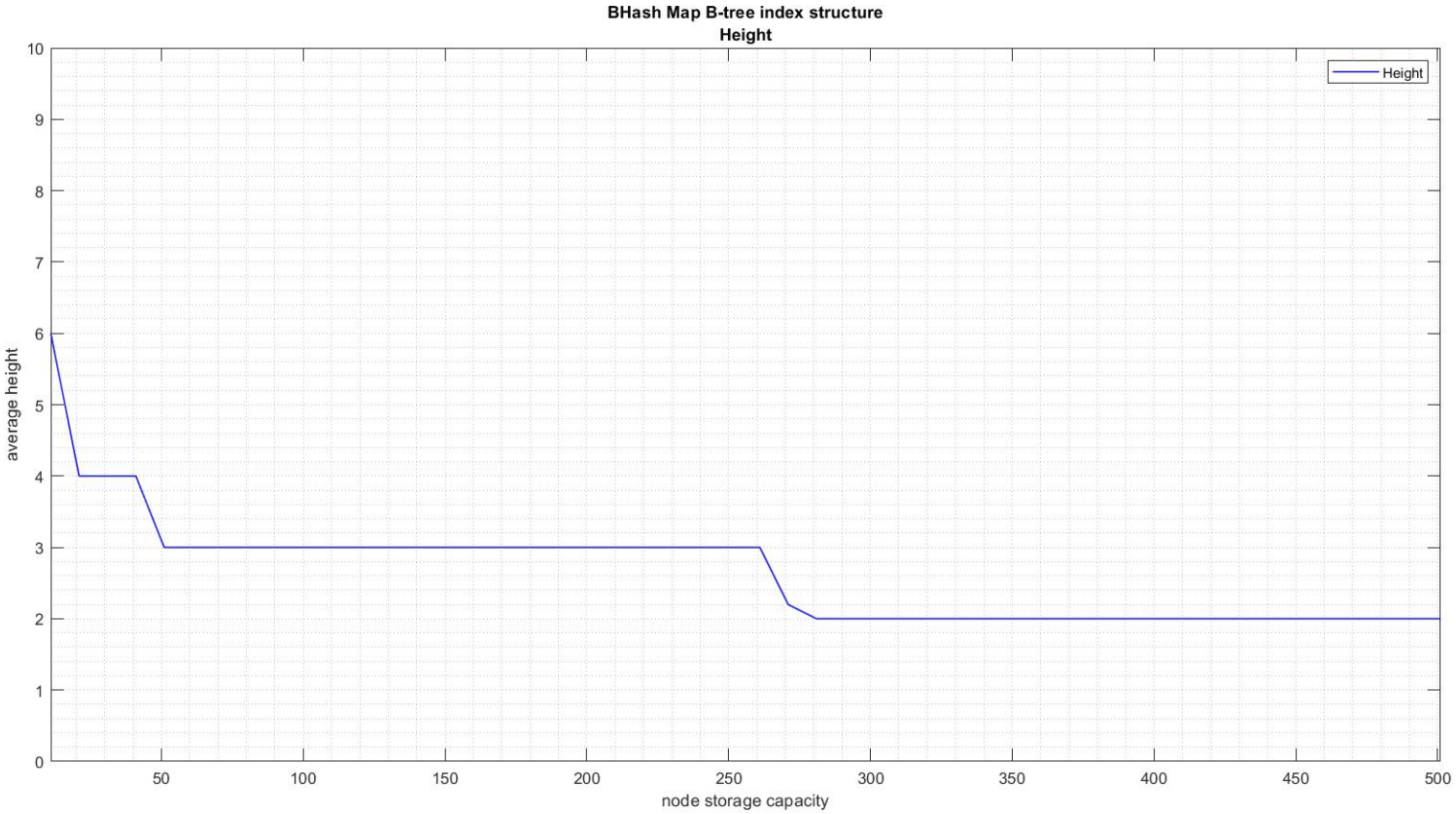
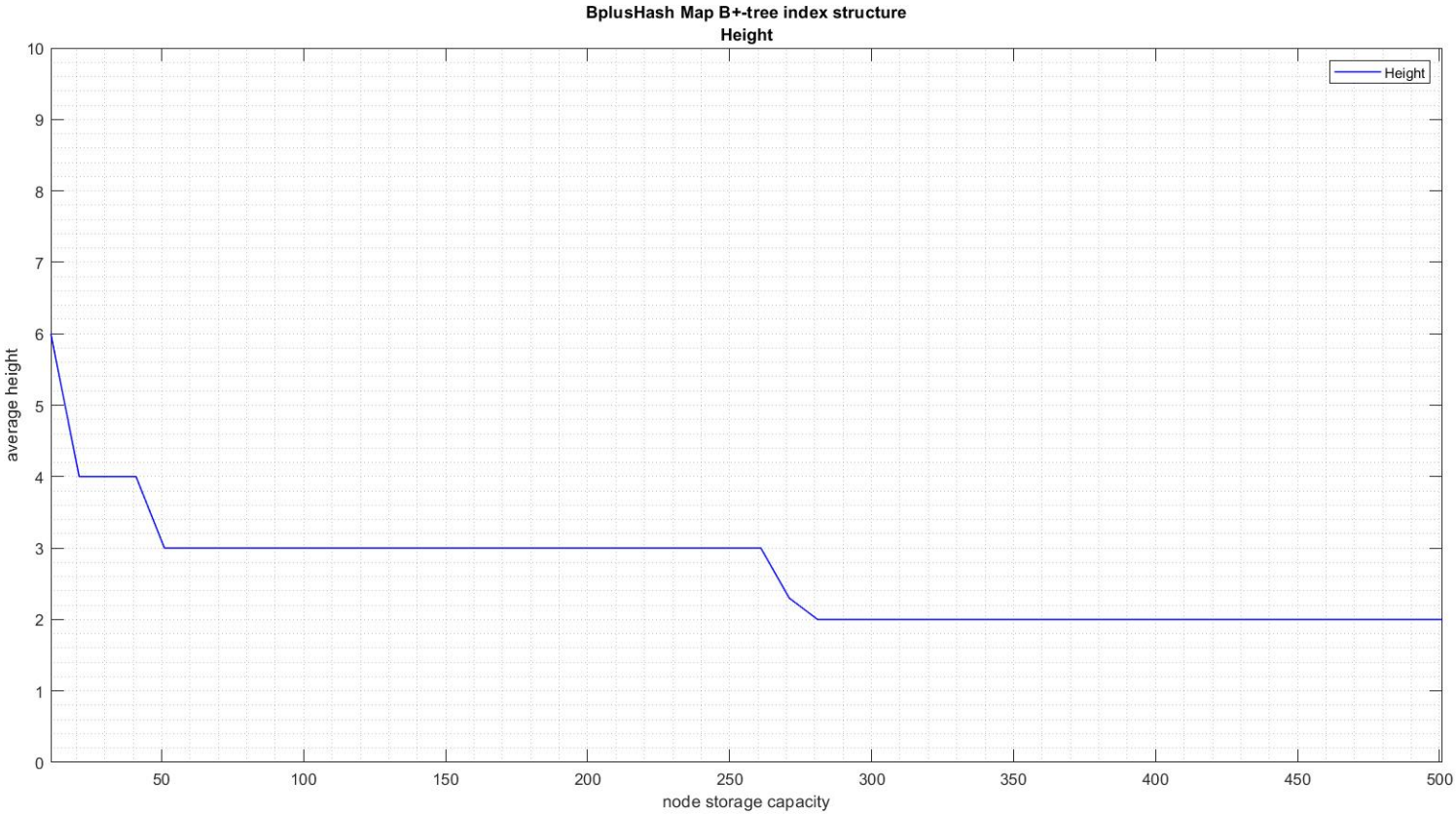


Fig. 5.68 represents the average B$^+$-Hash Map index structure B$^+$-tree height.

Figure 5.68: Average B$^+$-Hash Map index structure B$^+$-tree height



5.3 Analysis and evaluation of the computations results

5.3.1 Theoretical analysis

In-memory system B-tree and B^+-tree index structures

The node size - capacity increment in the stored record references incrementally affects the rate and analogy of the B-tree and B^+-tree index structures width-wise structural expansion in association with the structural expansion in height. Increasing the capacity of each individual node component increases the record references that can be stored in each node by reducing the total number of nodes required to store the record references set that the indices contain. Consequently there is a correlation of the node capacity with the record references set storage distribution in the nodes set, the nodes structural distribution in internal and leaf nodes and the B-tree and B^+-tree index structures height. The node capacity increment affects the gradual reduction and eventually the balancing - stabilization of the index structures height.

The dynamic construction and structural mutability associated with the distribution, layout - arrangement and organization of the structures nodes and the nodes stored record references are based on the functional property of structural stabilization – balancing of the B-tree and B^+-tree index structures insertion and deletion operations. Therefore the dynamic mutability of the node capacity in stored record references affects the structural formulation of the B-tree and B^+-tree index structures.

Furthermore, increasing the node capacity in stored record references increases the probability of the average simple record reference insertion - storage in a node (it is not required the node split) of the B-tree and B^+-tree index structures and therefore increases the set of record references stored by simple storage procedure in the B-tree and B^+-tree index structures nodes reducing the node splits and the structural reordering, rearrangement and reconstruction at the nodes and tree structural levels. This reduction in nodes splits causes a reduction in the B-tree and B^+-tree index structures average height as the total number of internal and leaf nodes created by the node splits is reduced. Moreover the leaf nodes set is increased with a quite higher rate related to the internal nodes as each internal node creation (node split) requires multiple leaf nodes splits (creations). So we

can conclude that the B-tree and B^+-tree index structures are composed of quite more leaf nodes than internal nodes. Consequently, the nodes size - capacity of the B-tree and B^+-tree index structures has an impact on the efficiency and speed in terms of time resources management and time performance of the insertion function. The node capacity increment implies the average increase in the time efficiency and speed of the insertion function.

The node capacity increment in stored record references increases the probability of the average simple deletion of a record reference from a node of the B-tree and B^+-tree index structures. Therefore increases the record references set deleted - removed by simple deletion processes from the B-tree and B^+-tree index structures nodes reducing the nodes structural reordering, rearrangement and reconstruction operations. Reducing the set of nodes rearrangement and reconstruction - rebuilding procedures and their application rate has an effect on maintaining the distribution - proportion of the B-tree and B^+-tree index structures nodes in leaf and internal nodes and on keeping the height reduction rate at a low level. Furthermore the deletion function requires on average a quite larger set of nodes rearrangements - reconstructions related to the insertion function in order to structurally re-balance and stabilize the B-tree and B^+-tree index structures. Consequently, the nodes capacity of the B-tree and B^+-tree index structures has an impact on the efficiency and speed in terms of time resources management and time performance of the deletion function. The node capacity increment implies the average increase in the time efficiency and speed of the deletion function.

The node capacity increment causes the reduction of node splitting, nodes structural rearrangements - reconstructions (tree re-balancing operations) while increasing the set of functional algorithmic steps for the implementation and completion of each individual operation.

The record references location and selection in the nodes internal dynamic array storage structures is based on the binary and interpolation node-side search functions. Reducing the B-tree and B^+-tree index structures height reduces the total number of node-level record references location and selection operations as it reduces the nodes transitions (vertical nodes paths size from the root node to the bottom leaf nodes level of the structures). This reduces the average number of nodes (nodes set) where the localization-selection operations are applied. In parallel, the node capacity increment has impact on the branching factor increment. These factors reduce the average time of the record references selection function based on primary key field as they reduce the average time of the stored record references

node-side location - selection operations and the set of node-side location - selection operations at the structure tree level. This optimization is partially compensated by the node storage capacity increment as the average number of stored record references contained in each node is increased reducing the efficiency and time performance of the node-size location selection operations.

Therefore, the storage size - capacity increment of the B-tree and B^+-tree index structures nodes affects the time resources management efficiency optimization of the primary key field-based record references selection function.

The average height and internal nodes set reduction, the leaf nodes set increment and the general B-tree index nodes set reduction causes the recursive transition-crossing (scan and selection) operations between different structural levels nodes decreasing the recursion stack operations set and the stack memory usage (depth). In addition, the node storage capacity increment in record references affects the record references storage gathering on fewer nodes with higher capacity, reducing the memory gaps (unused allocated memory) in the in-memory (main memory - RAM) system. This increases the fast and efficient management (usage) of the allocated memory by using the required memory and reducing the location - selection time of the nodes and nodes semi-dynamic array structures memory components - memory structural parts that the stored record references sets are contained. This time reduction of the nodes location and selection in memory is caused as the nodes and the stored record references sets of nodes are more dense stored and distributed in the memory system in more continuous and larger memory chunks as dynamic array structures. The location and transition in less and continuous large memory components - chunks (B-tree index structure nodes) which are randomly distributed in memory (larger memory gaps between the nodes memory parts) is time efficient and fast as the location and transition operations in continuous memory is quite faster in compare to the location and transition operations in more and smaller memory components (nodes with small storage capacity) that the memory is randomly distributed and less continuous.

Consequently, the node capacity increment of the B-tree index structure affects the time resources management efficiency optimization of the complete B-tree stored records references set scan - selection function.

This full scan - selection function of the B^+-tree index structure stored record references set is based on the structural property of the B^+-tree that contains the stored record references set in a set of Double Linked List structure leaf nodes at the bottom leaf nodes level of the tree. In this case a complete iterative scan - selection process is implemented in the Double Linked List structure nodes that store the record references. Therefore the B^+-tree index structure full selection function has a little better theoretical time performance in compare with the B-tree because there are not recursive transfer between nodes of different structural levels (recursion stack operations). Moreover the B^+-tree nodes storage distribution in the memory system affects the selection function time and memory resources management efficiency and time performance.

Consequently, the node capacity increment of the B^+-tree index structure affects the time resources management efficiency optimization of the complete records references set scan - selection function.

As the nodes capacity of the B-tree and B^+-tree index structures increases in proportion to the total number of stored record references of the structures, the structures tend to transform into semi-dynamic array structures losing their structural and functional properties. This causes the gradual destabilization and reduction of the B-tree and B^+-tree index structures functional performance in terms of time and memory resource management efficiency.

On-disk file system B-tree and B^+-tree index structures

On real RDBMS file systems as MySQL, PostgreSQL and SQLite on-disk file systems the B-tree - B^+-tree indexes are constructed to be utilized and operate as disk-based memory system structures that are stored on disk. Each B-tree - B^+-tree index structure record insertion, deletion, update and selection operation requires the load - transfer of the index structure nodes, stored record references and records physical data from the storage disk system to the main memory (RAM) system. The exchange of this data between the main and disk memory systems for processing applying a set of transactions and operations that affects the stored data on disk requires a large set of read - write and delete disk operations. These read - write and delete disk operations (disk access operations) are quite time consuming and inefficient in terms of time and memory resources management, especially when a large B-tree - B^+-tree index structure part or the whole structure must

be located and transferred from the disk memory system to the main memory in order to be implemented a set of transactions - operations.

The RDBMS file systems indexing sub-systems B^+-tree index structures store the records sets - record references sets at the last bottom leaf nodes level and the upper internal nodes levels store only the node references and nodes metadata. As opposed to the B^+-tree the B-tree index structures store the records sets - record references sets in both leaf and internal nodes and the internal nodes also store the node references and a set of nodes metadata. Each node is a memory component on the disk memory system that consists of fixed size - capacity allocated memory chunks. The leaf nodes can store a quite larger set of records sets - record references sets in compare to the internal nodes which store both the records and nodes references sets. Therefore, a typical leaf node can store a large set of records in a set of less disk memory components related to the internal node which can not fit and store the same set of records utilizing the same memory components. This causes the internal nodes memory components distribution - scatter in remote disk memory system parts (disk locations) requiring more disk access operations for the reconstruction, rearrangement and transfer of an internal nodes from the disk to main memory compared to a leaf node. As already mentioned, the B^+-tree stores the records data only in the leaf nodes and as opposed to the B^+-tree the B-tree index structures store the records data in both leaf and internal nodes. This affects the efficiency of the B-tree insertion, deletion, update and selection functions in terms of time and memory resources management reducing the functional performance of the B-tree related to the B^+-tree because of the larger set of disk accesses that the B-tree implements through its functions. Consequently, in practice, the B^+-tree index structure is more functionally efficient and fast related to the B-tree.

B-Hash Map and B^+-Hash Map index structures

Based on the theoretical analysis of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures it can be concluded that the insertion, deletion, update and selection functions of the B-Hash Map and B^+-Hash Map index structures have generally and approximately higher performance in terms of average time and memory resource management efficiency related to the B-tree and B^+-tree indexes corresponding functions on disk-based and in-memory file systems. Furthermore as the B-tree and B^+-tree index structures have approximately the same average time performance on in-memory file systems, the B-Hash Map and B^+-Hash Map index structures that are composed of B-tree and B^+-tree indexes

nodes have proportional functional performance.

5.3.2 Computational processes results on constructed and real data

In this implementation, the theoretical analysis and the conducted computational processes data analysis on constructed and real data shown in Fig. 5.9 – 5.56 and in set of the other related figures, demonstrate that the implemented and developed insertion, deletion and selection functions of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures are on average quite consistent with the existing theory and theoretical analysis.

Furthermore, the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures height, the internal and leaf nodes and the records references sets that are stored in the internal and leaf nodes computational data shown in the related figures provided by the conducted computational processes are also correspond to the related provided theory and theoretical analysis.

As indicated through the theoretical analysis, the average time performance of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures functions on real dataset is quite higher but approximately analogous to the average time performance of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures functions on synthetic data (records set with records primary key fields of string type).

Chapter 6

Summary and inferences

Realizing and observing that there are not sufficient dynamic, efficient, refactorable - maintainable and general-purpose implementations in C of in-memory (RAM) system B-tree, B^+-tree, B-Hash Map and B^+-Hash Map indexes structures combined with a complete, qualitative and concurrently simple theoretical and computational analysis, the conducted study aims to fill partially this gap. In this study, an open-source C programming language software package was implemented, developed and provided (based on the cited studies and researches) that includes a set of quite dynamic, efficient, refactorable, maintainable, testable and general-purpose in-memory system B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures combined with a complete theoretical and computational analysis of the indexes structure, functions and functional efficiency in terms of time and memory resources. The theoretical analysis of the B-tree, B^+-tree, B-Hash Map and B^+-Hash Map index structures functional performance was carried out and implemented through a computational processes set on real and synthetic data providing metric data for meta-analysis, evaluation and comparison. The theoretical analysis of the computational processes metric data demonstrates that the implemented in-memory B-tree and B^+-tree index structures functions have quite similar average time performance. Furthermore, it can be concluded that the implemented in-memory B-Hash Map and B^+-Hash Map index structures functions are quite faster and more efficient in terms of time resources management related to the B-tree and B^+-tree index structures functions. Finally, the provided packages are available on **GitHub** for meta-analysis, evaluation, modification, refactoring, testing and further development in order to be implemented and created a more dynamic, complete, maintainable and stable software package.

References

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, (New York, NY, USA), p. 107–141, Association for Computing Machinery, 1970.

[2] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[3] S. E. Fischbeck, "The ubiquitous b-tree: volume ii," 1987.

[4] C. E. Langenhop and W. E. Wright, "A model of the dynamic behavior of b-trees," *Acta informatica*, vol. 27, no. 1, pp. 41–59, 1989.

[5] G. Held and M. Stonebraker, "B-trees re-examined," *Communications of the ACM*, vol. 21, no. 2, pp. 139–143, 1978.

[6] C. Jiang-Hsing and G. D. Knott, "An analysis of b-trees and their variants," *Information Systems*, vol. 14, no. 5, pp. 359–370, 1989.

[7] P. Koruga and M. Baca, "Analysis of b-tree data structure and its usage in computer forensics," in *Central European Conference on Information and Intelligent Systems*, p. 423, Faculty of Organization and Informatics Varazdin, 2010.

[8] W. E. Wright, "Some average performance measures for the b-tree," *Acta Informatica*, vol. 21, no. 6, pp. 541–557, 1985.

[9] A. C.-C. Yao, "On random 2–3 trees," *Acta Informatica*, vol. 9, no. 2, pp. 159–170, 1978.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[11] C. Thomas H, L. Charles E, R. Ronald L, S. Clifford, *et al.*, "Introduction to algorithms," 2016.

[12] H. Knebl, "Algorithms and data structures," *Cham: Springer Nature Switzerland AG*, 2020.

[13] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

[14] S. Salakos and N. Ploskas, "Analysis and comparison of binary and interpolation search algorithms in a b-tree," in *25th Pan-Hellenic Conference on Informatics*, pp. 74–78, 2021.

[15] S. Sippu and E. Soisalon-Soininen, "Transaction processing."

[16] T. Lahdenmaki and M. Leach, *Relational Database Index Design and the Optimizers*. USA: Wiley-Interscience, 2005.

[17] H. Korth, S. Sudarshan, and P. Abraham Silberschatz, *Database System Concepts*. McGraw-Hill Education, 2010.

[18] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill Education, 2019.

[19] R. Elmasri, "Fundamentals of database systems seventh edition," 2021.

[20] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill higher education, McGraw-Hill Education, 2003.

[21] H. Garcia-Molina, *Database Systems: The Complete Book*. Pearson Education, 2008.

[22] G. Graefe and H. Kuno, "Modern b-tree techniques," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 1370–1373, IEEE, 2011.

[23] G. Graefe, "Modern b-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[24] "Mysql innodb storage engine and index system." `https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html`.

[25] P. Kieseberg, S. Schrittwieser, P. Frühwirt, and E. Weippl, "Analysis of the internals of mysql/innodb b+ tree index navigation from a forensic perspective," in *2019 International Conference on Software Security and Assurance (ICSSA)*, pp. 46–51, IEEE, 2019.

[26] P. Frühwirt, P. Kieseberg, and E. Weippl, "Using internal mysql/innodb b-tree index navigation for data hiding purposes,"

[27] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, "Innodb database forensics: Enhanced reconstruction of data manipulation queries from redo logs," *Information Security Technical Report*, vol. 17, no. 4, pp. 227–238, 2013.

[28] P. Frühwirt, M. Huber, M. Mulazzani, and E. R. Weippl, "Innodb database forensics," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 1028–1036, IEEE, 2010.

[29] M.-A. Manu, "Mysql database engines review, analysis, compilation and customization," 2013.

[30] "Postgresql documentation." `https://www.postgresql.org/docs/current/`.

[31] H. Dombrovskaya, B. Novikov, and A. Bailliekova, *PostgreSQL Query Optimization*. Springer, 2021.

[32] E. Inersjö, "Comparing database optimisation techniques in postgresql: Indexes, query writing and the query optimiser," 2021.

[33] D. Kuhn, S. R. Alapati, and B. Padfield, *Expert Oracle Indexing and Access Paths: Maximum Performance for Your Database*. Springer, 2016.

[34] "Architecture of sqlite system." `https://www.sqlite.org/arch.html`.

[35] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[36] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[37] W. A. Bhat and M. A. Wani, "Forensic analysis of b-tree file system (btrfs)," *Digital Investigation*, vol. 27, pp. 57–70, 2018.

[38] D. Galles, "B-tree visualization." `https://www.cs.usfca.edu/~galles/visualization/BTree.html`, 2011.

[39] A. Levitin, *Introduction to the Design & Analysis of Algorithms*. Always learning, Pearson, 2012.

[40] S. Saha and S. Shukla, *Advanced Data Structures: Theory and Applications*. CRC Press, 2019.

[41] R. A. Baeza-Yates, "Expected behaviour of b+-trees under random insertions," *Acta Informatica*, vol. 26, no. 5, pp. 439–471, 1989.

[42] C. A. Shaffer, *A practical introduction to data structures and algorithm analysis*. Prentice Hall Upper Saddle River, NJ, 1997.

[43] S. Groppe, *Data management and query processing in semantic web databases*. Springer Science & Business Media, 2011.

[44] S. Sippu and E. Soisalon-Soininen, *Transaction processing: Management of the logical database and its underlying physical structure*. Springer, 2015.

[45] V. Srinivasan and M. J. Carey, "Performance of b+ tree concurrency control algorithms," *The VLDB Journal*, vol. 2, no. 4, pp. 361–406, 1993.

[46] M. T. . Ling Liu, *Encyclopedia of Database Systems*. Springer Science+Business Media, LLC, part of Springer Nature 2018, 2018.

[47] S. D. Viglas, "Adapting the b+-tree for asymmetric i/o," in *East European Conference on Advances in Databases and Information Systems*, pp. 399–412, Springer, 2012.

[48] D. Galles, "B+-tree visualization." `https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`, 2011.

[49] K. Mehlhorn, P. Sanders, and P. Sanders, *Algorithms and data structures: The basic toolbox*, vol. 55. Springer, 2008.

[50] A. Drozdek, *Data Structures and algorithms in C++*. Cengage Learning, 2012.

[51] T. Mailund, *The joys of hashing: hash table programming with C*. Apress, 2019.

[52] S. S. Skiena, *The algorithm design manual*, vol. 2. Springer, 1998.

[53] R. Sedgewick, *Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching*. Pearson Education, 1998.