Implementation and Analysis of B+ Trees

November 5, 2023

Walde Vismay (2022MCB1283) ,
Shivam Zample (2022MCB1280) ,
Tejas Wagh (2022CSB1144)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
E Harshith Kumar Yadav

Summary: Our project entails the implementation and analysis of B+ Trees. The project explored the fundamental concepts, design, and performance analysis of B+ trees, a popular data structure in computer science and database management. B+ trees are widely used to optimize database indexing and enhance search and retrieval operations. The primary focus of the project was on the implementation of B+ trees. The team implemented the core operations of Btree and B+ trees, including insertion, deletion, and search, to create a functional data structure. This involved defining the structure of nodes, splitting and merging nodes to maintain balance, and ensuring proper ordering of data within the tree. Detailed explanations of how B+ trees optimize search and retrieval operations were provided. B+ trees offer logarithmic time complexity for searching, making them efficient for database applications.The project conducted a thorough performance analysis of B+ trees. It included evaluating the time complexity of search operation and comparing the results with Btree. This analysis showcased the advantages and limitations of using B+ trees in real-world scenarios.

---

1. Introduction

B+ trees are a fundamental and widely used data structure in computer science, particularly in the realm of database management systems. They were first introduced by Rudolf Bayer and Edward M. McCreight in 1972 and have since become a cornerstone in the organization of large datasets, improving data access efficiency and search performance. The primary motivation behind the development of B+ trees was to address the limitations of previous data structures like binary search trees, while also taking advantage of the benefits of balanced tree structures.

Key Characteristics of B+ Trees:

Balanced Structure: B+ trees are self-balancing data structures, which means that they automatically maintain their balance during insertions and deletions. This property ensures that the depth of the tree remains shallow, leading to consistent and efficient search operations.

Multiway Tree: Unlike binary search trees, which have two children per node, B+ trees have multiple child nodes per internal node. This multiway property minimizes the height of the tree, leading to logarithmic time complexity for search operations, which is essential for managing large datasets.

Ordered Data: Data in a B+ tree is organized in a sorted order, typically from left to right. This characteristic simplifies range queries, as traversing from one key to another only requires sequential access.

Leaf-Node Storage: In B+ trees, data entries are stored in leaf nodes, with internal nodes serving as index structures. This design decision streamlines data retrieval since searching always ends at a leaf node. Furthermore, leaf nodes are linked together, creating a linked list that facilitates range queries.

2. B+ Tree

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

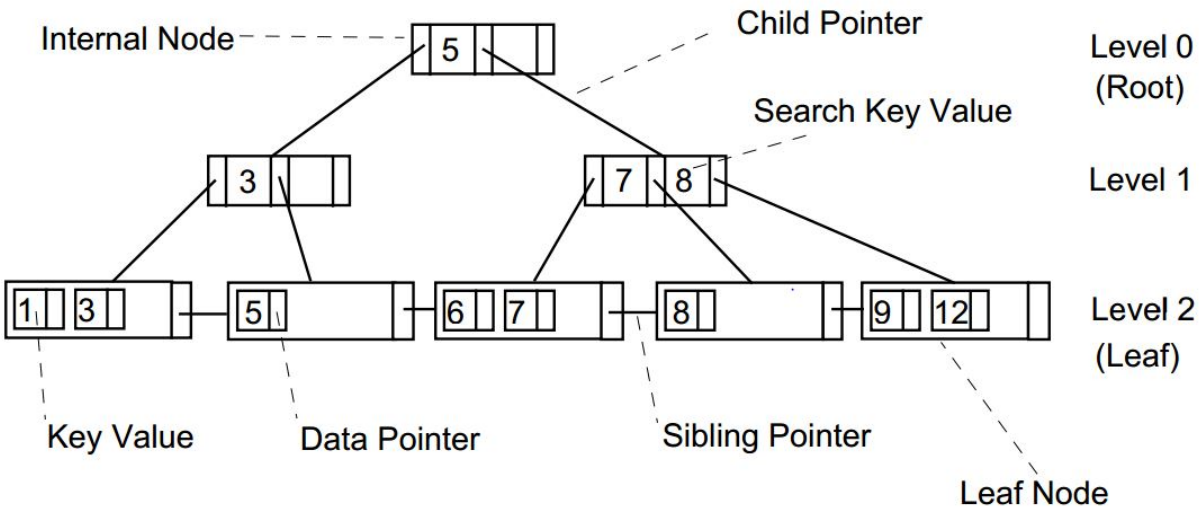

Figure 1: Structure of a B+ Tree

B+ Trees contain two types of nodes:

Internal Nodes: In B+ trees, internal nodes are nodes that serve as intermediaries between the root node and the leaf nodes. These nodes contain key values and pointers to child nodes. ,
Leaf Nodes: leaf nodes in a B+ tree serve as the endpoints of the tree structure and contain the actual data entries.

The Structure of the Internal Nodes of a B+ Tree of Order 'd' is as Follows

1. Each internal node is of the form: <P1, K1, P2, K2, ..., Pc-1, Kc-1, Pc> where c <= d and each Pi is a tree pointer (i.e points to another node of the tree) and, each Ki is a key-value (see Figure 2 for reference).

2. Every internal node has : K1 < K2 < .... < Kc-1
3. For each search field value 'X' in the sub-tree pointed at by Pi, the following condition holds: Ki-1 < X <= Ki, for 1 < I < c and, Ki-1 < X, for i = c (See diagram I for reference)
4. Each internal node has at most 'd' tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least ceil(d/2) tree pointers each.
6. If an internal node has 'c' pointers, c <= d, then it has 'c – 1' key values.

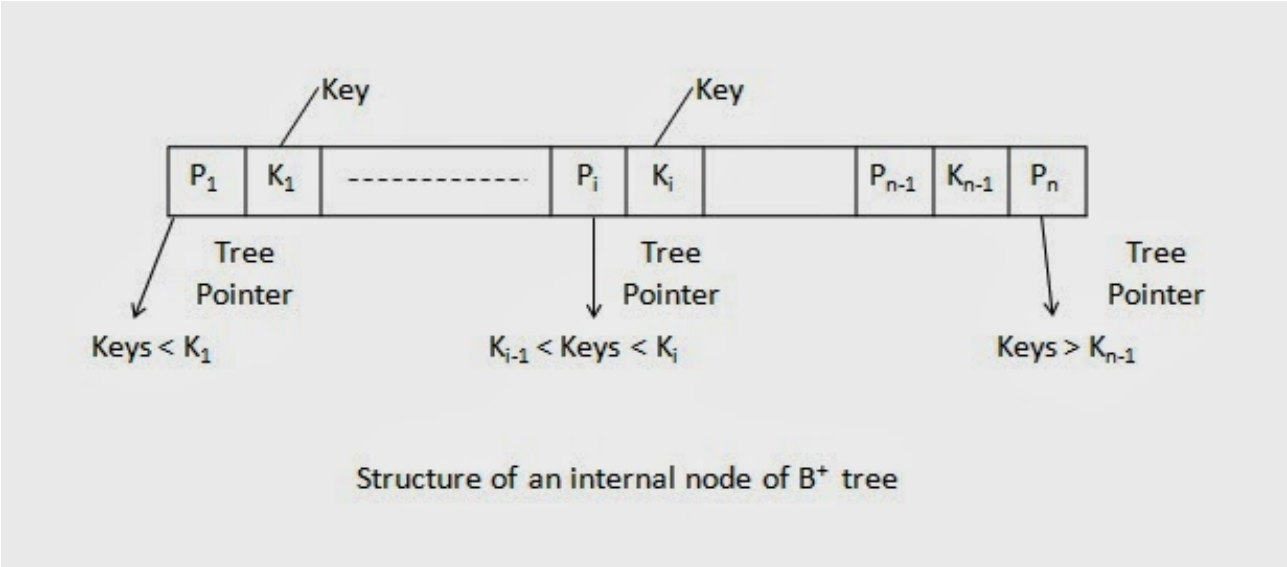

Figure 2: Structure of Internal Node

The Structure of the Leaf Nodes of a B+ Tree of Order 'd' is as Follows
1. Each leaf node is of the form: «K1, D1>, <K2, D2>, ..., <Kc-1, Dc-1>, Pnext> where c <= d and each Di is a data pointer (i.e points to actual record in the disk whose key value is Ki or to a disk file block containing that record) and, each Ki is a key value and, P next points to next leaf node in the B+ tree (see Figure 3 for reference).
2. Every leaf node has : K1 < K2 < .... < Kc-1, c <= d
3. Each leaf node has at least ceil(d/2) values.
4. All leaf nodes are at the same level.

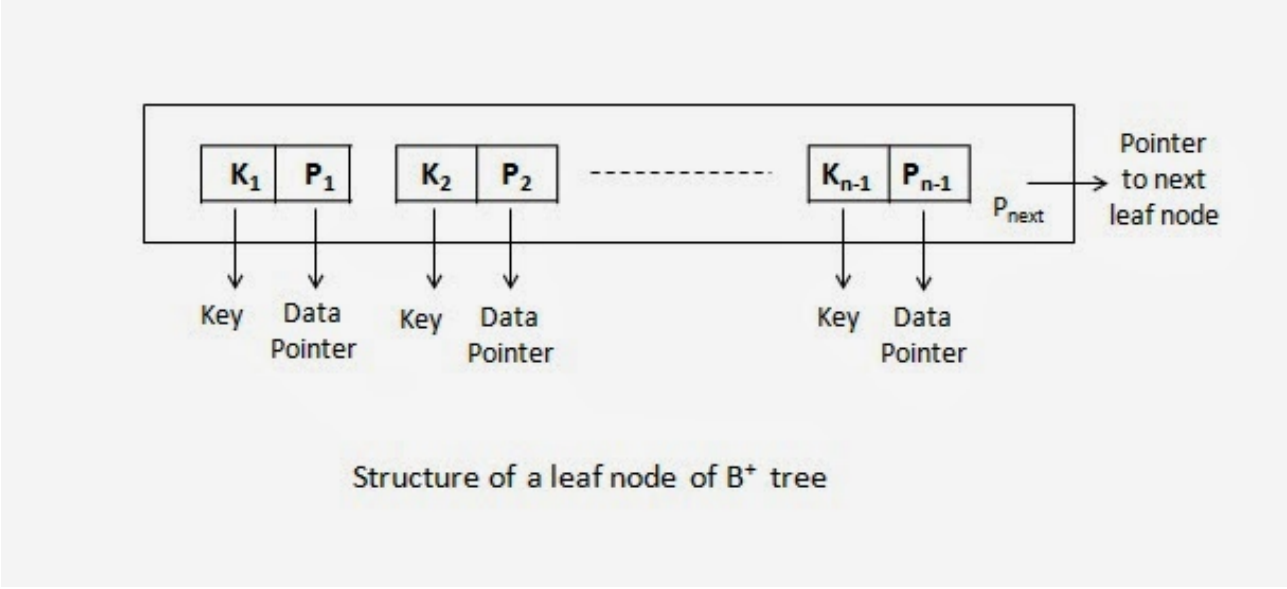

Figure 3: Structure of Leaf Node

3. Observations

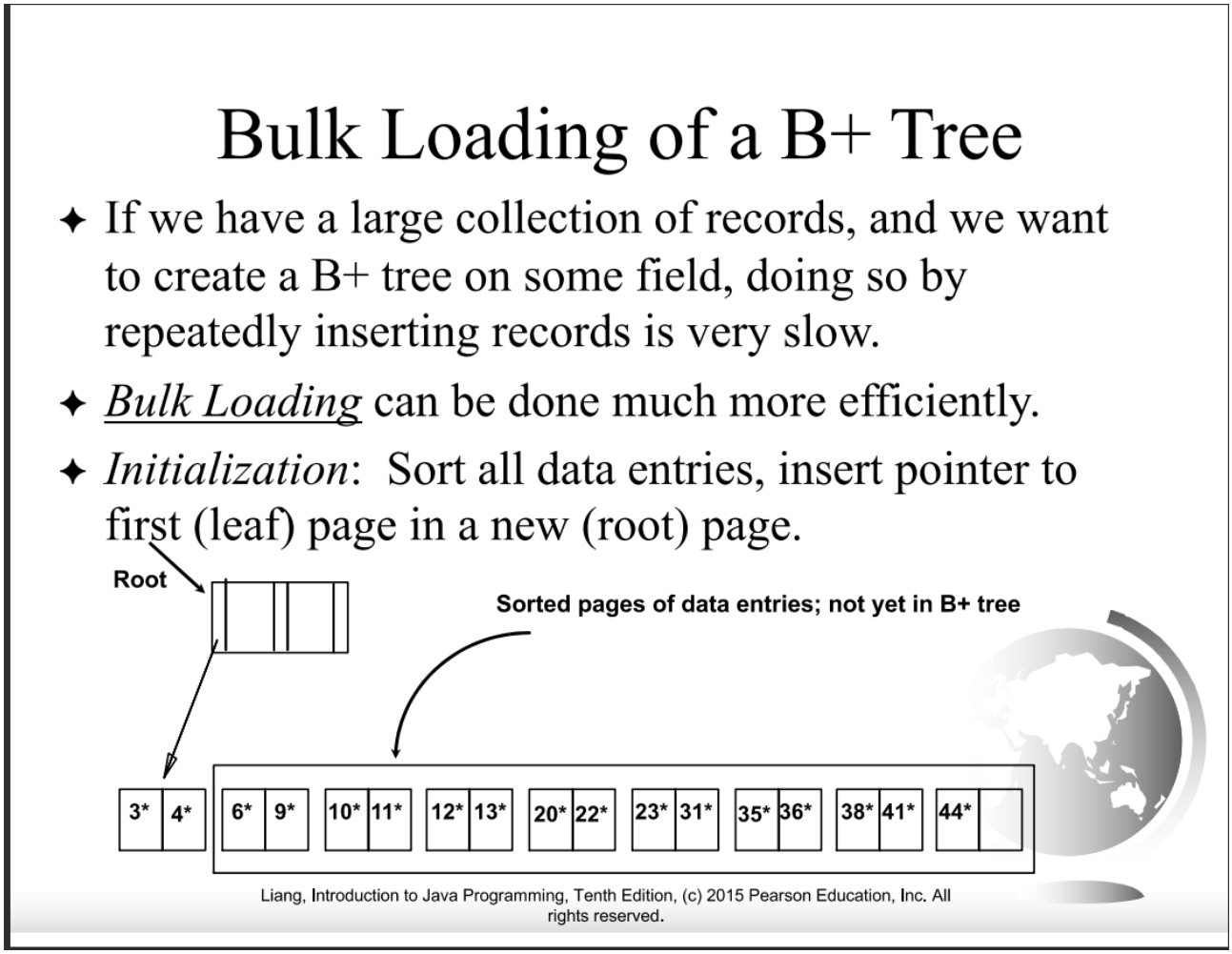

Figure 4: Bulk loading 1

Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level.
- When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts.

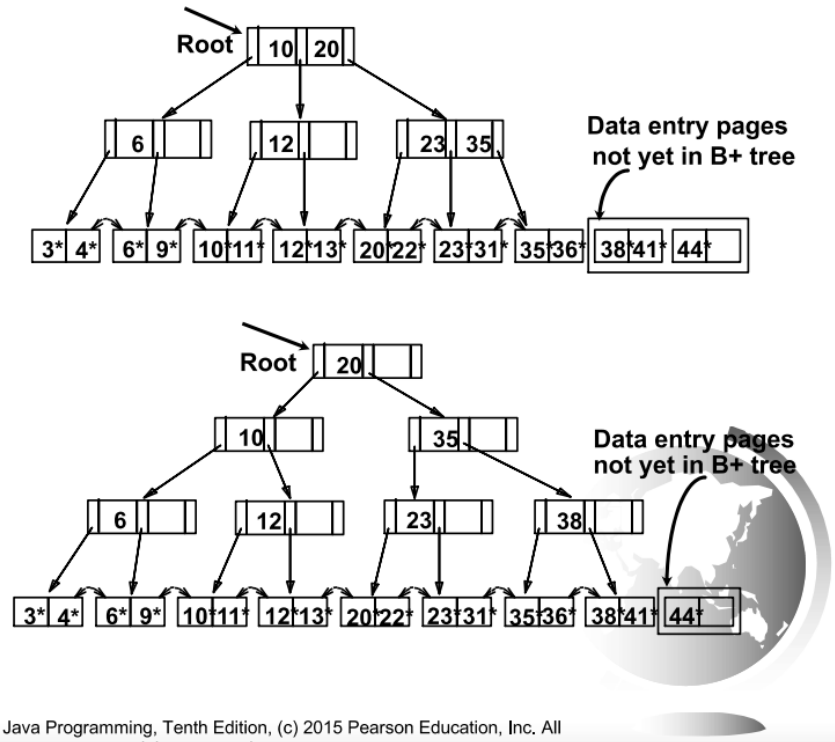

Figure 5: Bulk loading 2

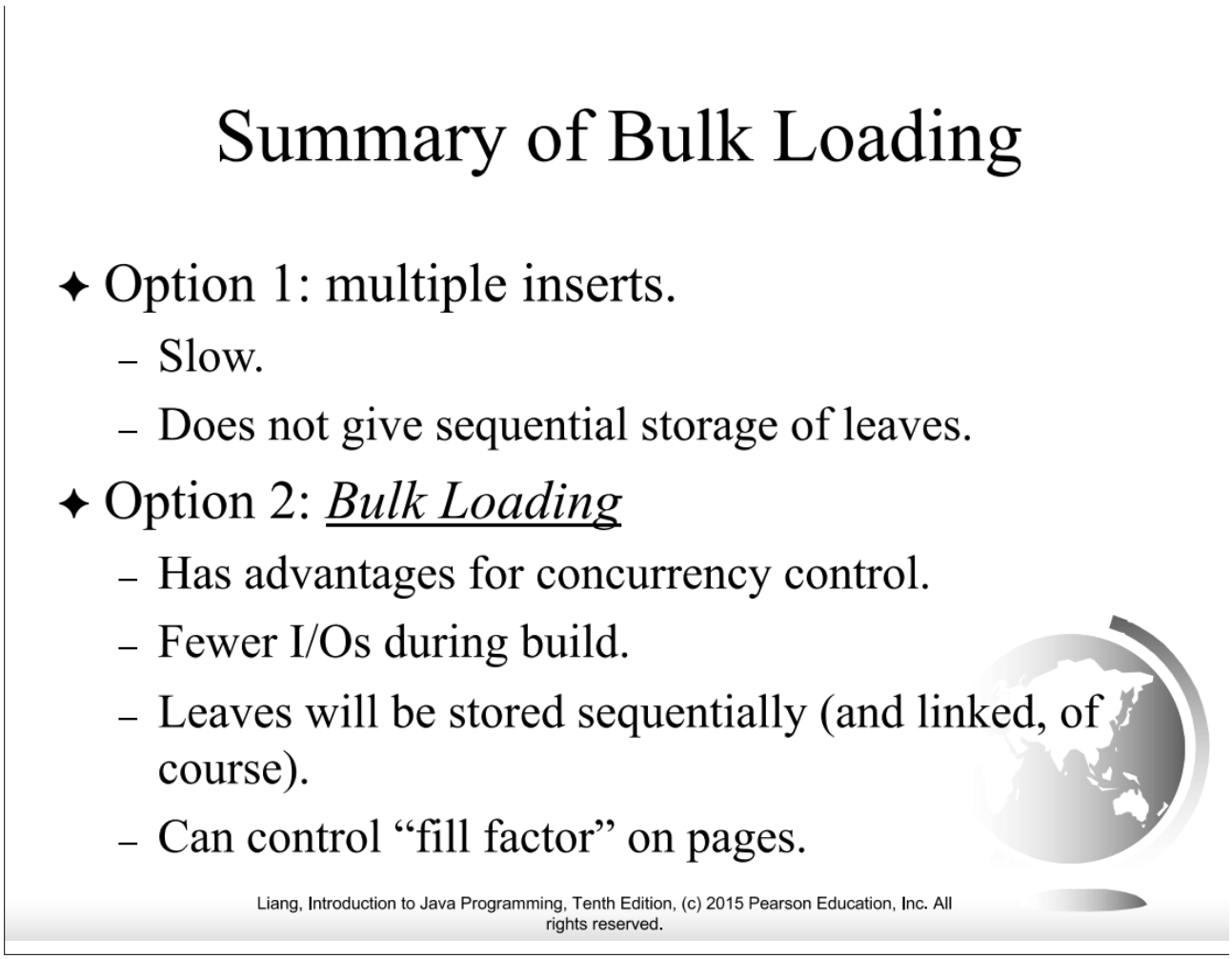

Figure 6: Bulk loading 3

3.1. Complexities

	Expected Complexity	Worst Case Complexity
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

4. Algorithms

4.1. Search

The algorithm for search finds the leaf node in which a given data entry belongs.To find a element (i) in tree search requires us to search within the node, which can be done with either a linear search or a binary search. We will be using linear search. Consider the sample B+ tree shown in Figure 9.10. This B+ tree is of order d=2. That is, each node contains between 2 and 4 entries. Each non-leaf entry is a (key value, node pointer) pair; at the leaf level, the entries are data records that we denote by k. To search for entry 5*, we follow the left-most child pointer, since 5 < 13. To search for the entries 14* or 15*, we follow the second pointer, since 13<= 14 < 17, and 13<=15 < 17. (We don't find 15* on the appropriate leaf, and we can conclude that it is not present in the tree.) To find 24*, we follow the fourth child pointer, since 24<=24< 30.

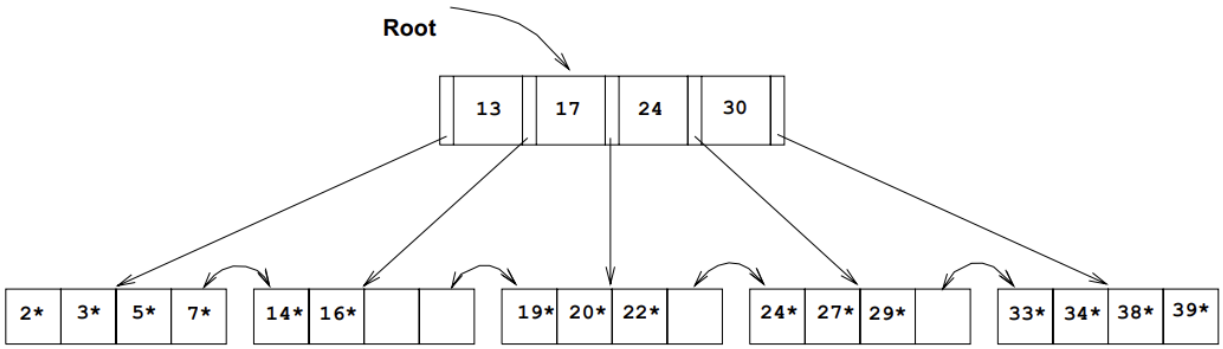

Figure 9.10 Example of a B+ Tree, Order d=2

Figure 7

```
func find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                          // searches from root
endfunc

func tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K_1 then return tree_search(P_0, K);
    else,
        if K ≥ K_m then return tree_search(P_m, K);        // m = # entries
        else,
            find i such that K_i ≤ K < K_{i+1};
            return tree_search(P_i, K)
endfunc
```

Figure 8: Algorithm for Search

4.2. Insertion

The algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. Pseudocode for the B+ tree insertion algorithm is given in Figure 9.11. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable newchildentry. If the (old) root is split, a new root node is created and the height of the tree increases

by one.

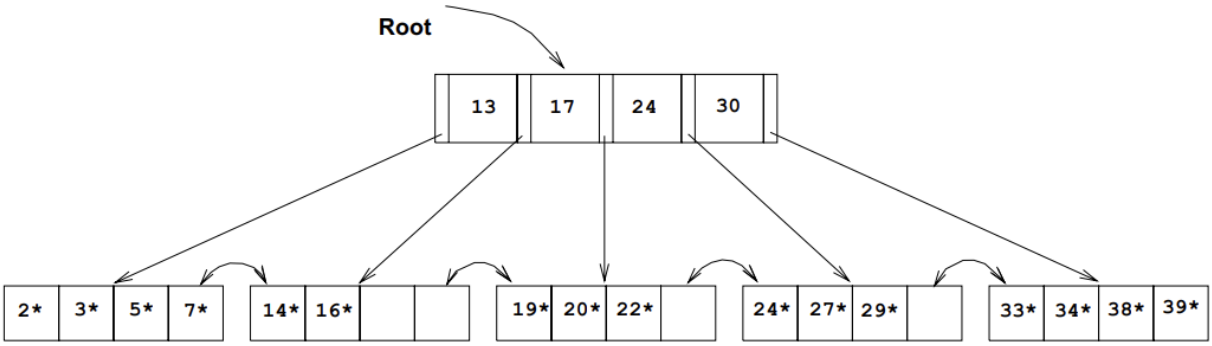

Figure 9: Example of a B+ Tree, Order d=2

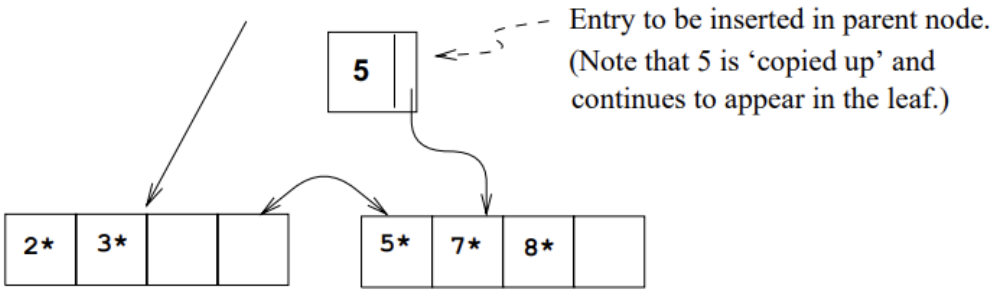

Figure 9.12 Split Leaf Pages during Insert of Entry 8*

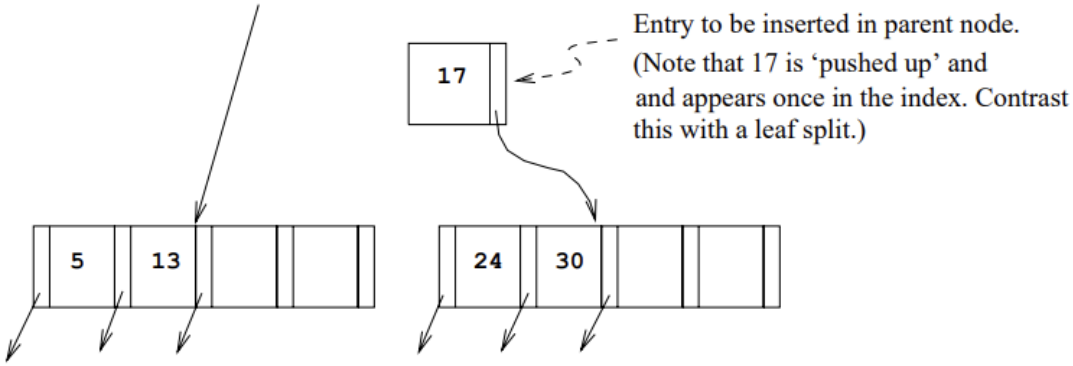

Figure 9.13 Split Index Pages during Insert of Entry 8*

Figure 10

Now, since the split node was the old root, we need to create a new root node to hold the entry that distinguishes the two split index pages. The tree after completing the insertion of the entry 8* is shown in Figure 9.14.

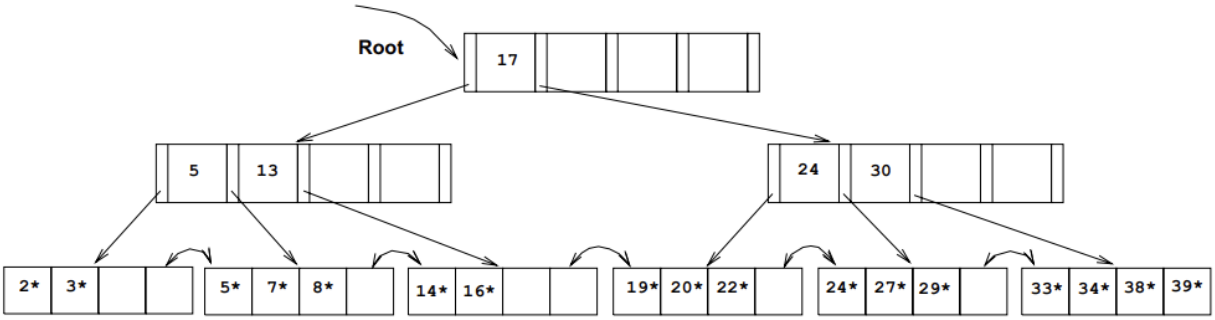

Figure 9.14 B+ Tree after Inserting Entry 8*

Figure 11: The tree after completing the insertion of the entry 8* is shown in Figure 9.14

```
proc insert (nodepointer, entry, newchildentry)
// Inserts entry into subtree with root '*nodepointer'; degree is d;
// 'newchildentry' is null initially, and null upon return unless child is split

if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};          // choose subtree
    insert(P_i, entry, newchildentry);                  // recursively, insert entry
    if newchildentry is null, return;                  // usual case; didn't split child
    else,                          // we split child, must insert *newchildentry in N
        if N has space,                                                  // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                                  // note difference wrt splitting of leaf page!
            split N:                          // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between N and N2
            newchildentry = & (⟨smallest key value on N2, pointer to N2⟩);
            if N is the root,                                  // root node was just split
                create new node with ⟨pointer to N, *newchildentry⟩;
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                                      // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                          // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & (⟨smallest key value on L2, pointer to L2⟩);
        set sibling pointers in L and L2;
        return;
endproc
```

Figure 12: Algorithm for Insert

Analysis of expected insertion cost

The expected insertion cost in a B+ tree is typically low and close to O(1) on average due to its self-balancing characteristics. When inserting a new element, a logarithmic O(log N) search cost is incurred to find the appropriate leaf node, where the actual insertion is generally O(1). Occasionally, node splitting and height adjustments may occur, incurring O(M) and O(log N) costs, respectively, but these costs are amortized over multiple insertions. As a result, B+ trees provide efficient and predictable insertion performance for managing large datasets while ensuring a balanced and optimized data structure.

4.3. Deletion

The algorithm for deletion takes an entry, finds the leaf node where it belongs, and deletes it. The pseudocode for the B+ tree deletion algorithm is given in Figure 15. B+ tree deletion involves locating and removing a specific key-value pair from the tree while maintaining its structure and balance. The process begins with a search to find the key in the leaf nodes. Once found, the key-value pair is removed. If the deletion causes a leaf node to become underfilled (i.e., it has fewer keys than the minimum required), the tree performs redistributions or node merges to maintain balance, possibly affecting the parent nodes. The tree's height adjustment is handled similarly to insertion, where node splits or merges can propagate up the tree. Deletion in a B+ tree is a complex process that ensures data integrity and tree balance, making it efficient for maintaining large datasets in databases.

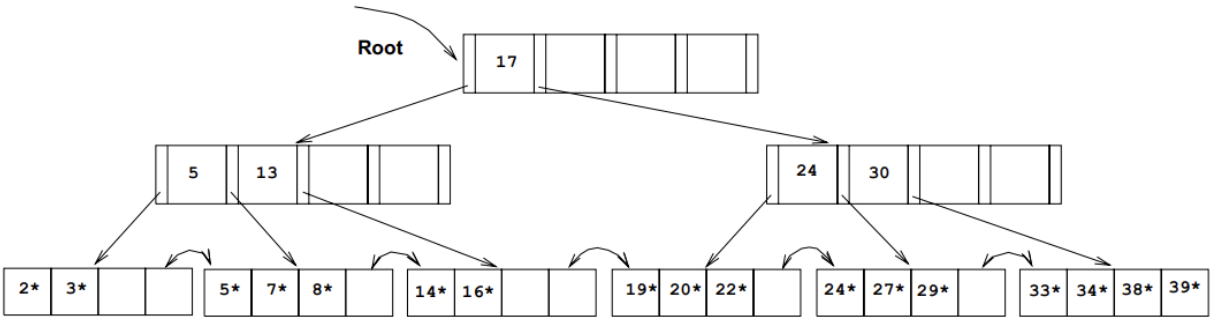

Figure 13: Initial B+ tree for deletion

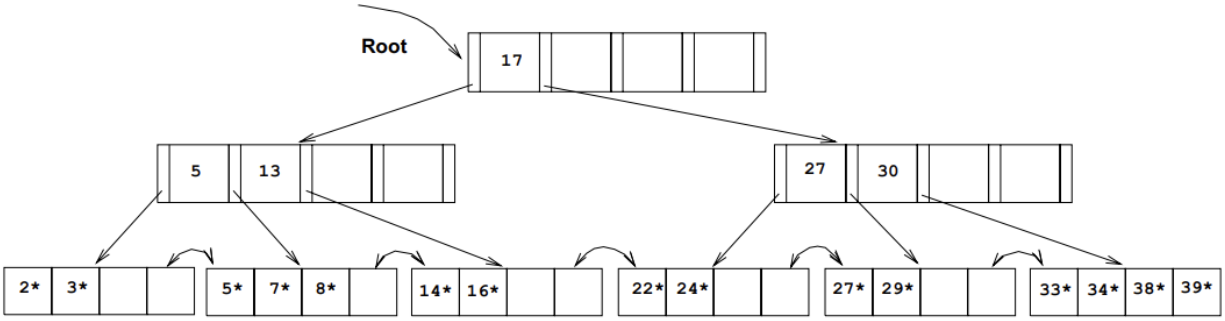

Figure 9.17 B+ Tree after Deleting Entries 19* and 20*

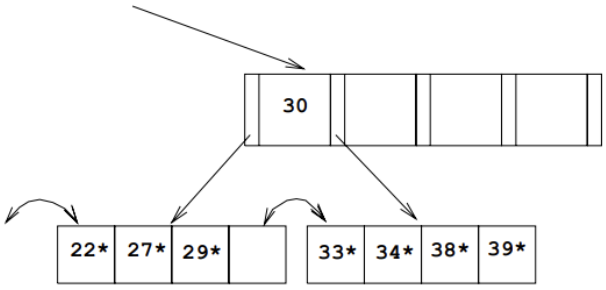

Figure 9.18 Partial B+ Tree during Deletion of Entry 24*

Figure 14

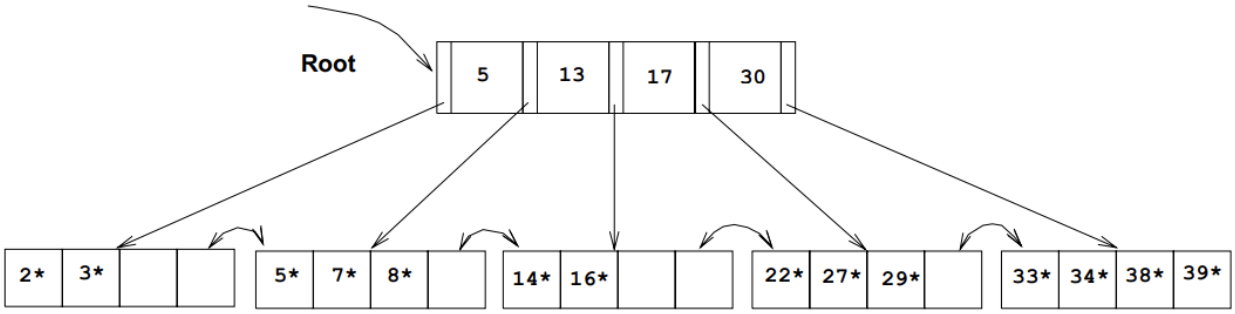

Figure 15: B+ Tree after Deleting Entry 24

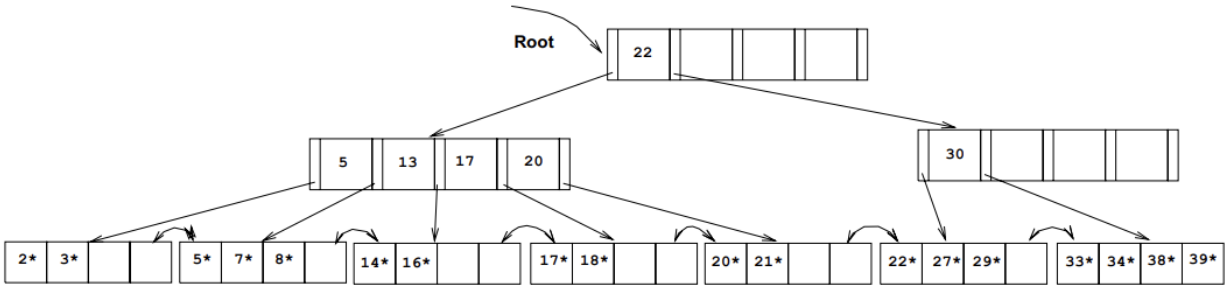

Figure 16: A B+ Tree during a Deletion

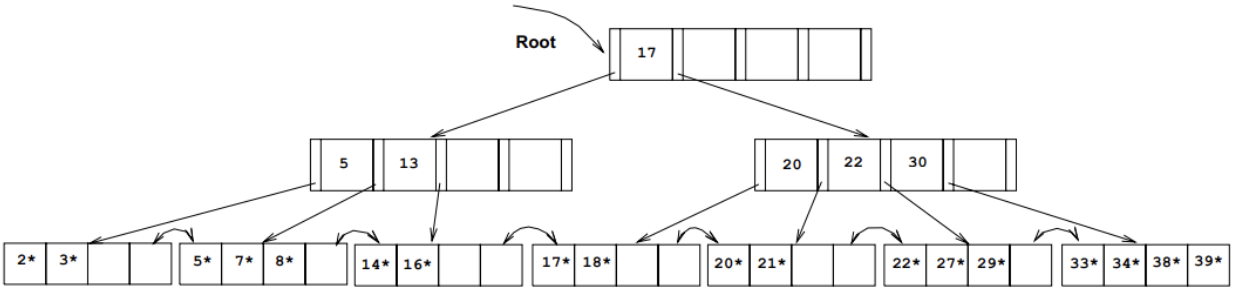

Figure 17: B+ Tree after Deletion

```
proc delete (parentpointer, nodepointer, entry, oldchildentry)
// Deletes entry from subtree with root '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};          // choose subtree
    delete(nodepointer, P_i, entry, oldchildentry);              // recursive delete
    if oldchildentry is null, return;                  // usual case: child not deleted
    else,                                  // we discarded child node (see discussion)
        remove *oldchildentry from N,      // next, check minimum occupancy
        if N has entries to spare,                                        // usual case
            set oldchildentry to null, return;        // delete doesn't go further
        else,                              // note difference wrt merging of leaf pages!
            get a sibling S of N:                  // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between N and S through parent;
                set oldchildentry to null, return;
            else, merge N and S                                  // call node on rhs M
                oldchildentry = & (current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                                            // usual case
        remove entry, set oldchildentry to null, and return;
    else,                                  // once in a while, the leaf becomes underfull
        get a sibling S of L;                          // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between L and S;
            find entry in parent for node on right;                   // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge L and S                                  // call node on rhs M
            oldchildentry = & (current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

Figure 18: Algorithm for Deletion

Analysis of expected deletion cost

The expected deletion cost in a B+ tree is typically low and close to O(1) on average due to the tree's self-balancing nature and its efficiency in handling deletions. A logarithmic O(log N) search cost is incurred to find the key to delete, while the actual deletion in the leaf node is generally O(1). Occasionally, node redistributions or merges may be required, incurring O(M) costs, and updates to parent nodes may be necessary, typically O(log N). Similar to insertions, height adjustments are managed as in insertions. Overall, B+ trees provide efficient and predictable deletion performance for managing large datasets in databases while preserving tree structure and balance, with occasional costs amortized over multiple deletions.

5. Conclusions

We have studied the methods of insertion, deletion and search in a B+ Tree and have analyzed the same. We have studied the advantages of B+ Trees over other balanced data structures like B Trees and self-adjusting trees. We have successfully implemented all three operations along with the operation to find the minimum element.Along with that we also realized the use of Bulk loading in minimizing the input output operations Possible future developments of this project would be

1. Dynamic B+ Trees
2. Apply data compression techniques to the B+ tree nodes to reduce storage requirements.
3. Combine B+ trees with other indexing structures like bitmap indexes or hash indexes to create hybrid indexing systems for improved query performance.

6. Applications

1. Faster operations on the tree (insertion, deletion, search)
2. B+ trees are widely used in database management systems to index data, allowing for efficient retrieval of records.
3. B+ trees are employed in distributed file systems like Hadoop HDFS and Google's GFS to maintain metadata about files and directories across a distributed cluster of servers.

7. Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant E Harshith Kumar Yadav for their guidance and valuable inputs provided during the project.

References

[1] GeeksforGeeks. Introduction of b+ tree, 2023.

[2] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[3] Wikipedia contributors. B+ tree — Wikipedia, the free encyclopedia, 2023.

[4] Liang Y, Daniel. Introduction to java programming. 2015.