A REPORT
ON
B+ TREE

CSE300: Technical Writing and Presentation

Prepared By

Anup Halder Joy - 2005005
Afzal Hossan - 2005021
Asif Karim - 2005024

Computer Science and Engineering
Bangladesh University of Engineering and Technology

Contents

List of Figures

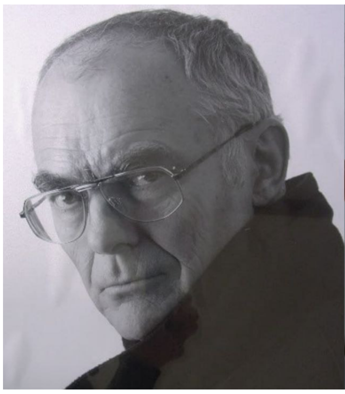
List of Tables

1 Definition

- A B+ tree is an m-ary tree with a large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.[5]
- A B+ Tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions
- The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

2 History

- The history of B+ trees dates back to their invention by Rudolf Bayer and Edward M. McCreight in 1972. The B+ tree was introduced as an improvement over the original B-tree data structure, which was also developed by Bayer and McCreight in 1970. Bayer and McCreight never explained what, if anything, the B stands for; Boeing, balanced, between, broad, bushy, and Bayer have been suggested. McCreight has said that ”the more you think about what the B in B-trees means, the better you understand B-trees”.



Rudolf Bayer

Edward M. McCreight

- The B+ tree was designed to address certain limitations of the B-tree, particularly in the context of file systems and database management systems. The key innovation of the B+ tree lies in the way it organizes and stores data.

3 Evolution of B+ Tree

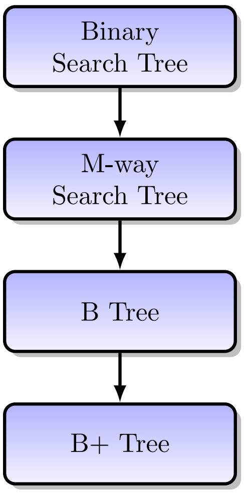


Figure 1: Evolution of B+ Tree

The evolution of the B+ tree from its predecessors marks a significant advancement in data structure design, particularly in the realm of database management systems.

- **Binary Search Tree:** Beginning with the Binary Search Tree (BST), which provided efficient searching but suffered from unbalanced structures leading to suboptimal performance in certain scenarios, such as highly skewed or sorted data distributions.

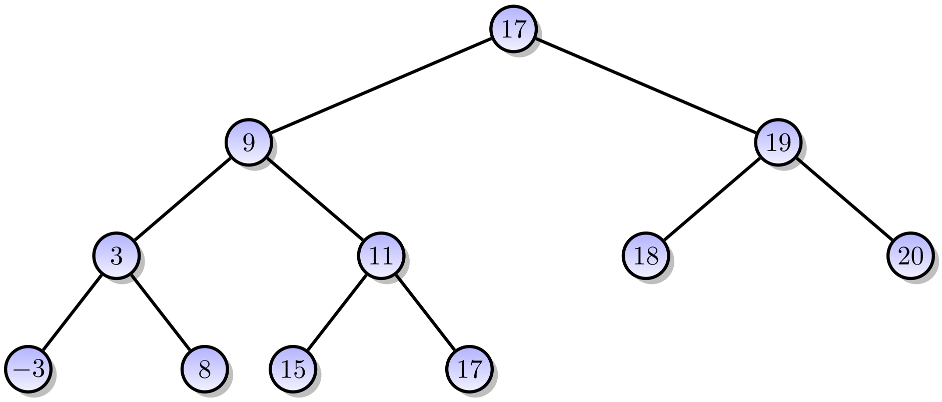


Figure 2: Binary Search Tree

- **M-way Tree:** The m-way tree addressed this imbalance by allowing multiple keys per node, improving balance and thus mitigating some of the inefficiencies of BSTs. However, it still faced limitations in disk-based storage systems, where frequent disk accesses could hamper performance.

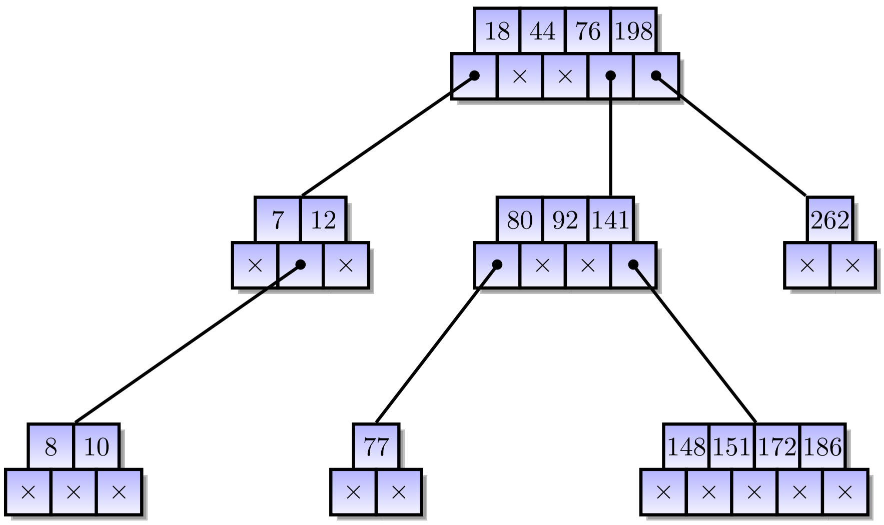


Figure 3: m-way Tree (m =5)

- **B Tree:** The B tree sought to remedy this by optimizing for external storage, but its design lacked efficient support for range queries and sequential access due to its internal structure, leading to suboptimal performance in certain database operations.

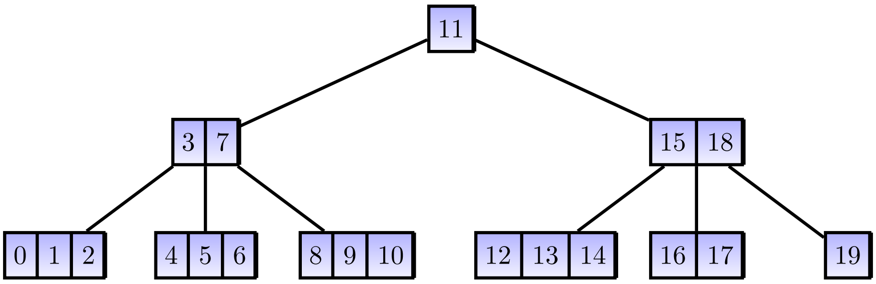


Figure 4: B Tree

- **B+ Tree:** Building upon all these challenges, the B+ tree refined this concept by separating internal nodes from leaf nodes, enhancing sequential access and range queries through its leaf-level linked lists while maintaining the balanced properties of its predecessors. This progression reflects a continual refinement in addressing the challenges of storage and retrieval in database systems, culminating in the robust and widely adopted B+ tree data structure.

4 Nodes of B+ Tree

4.1 Internal Node Structure

The structure of an internal node in a B+ tree is as follows:

- Sorted keys: Contains sorted keys that guide the search process.
- Child Pointers: Pointers to child nodes for navigating the tree.
- Variable Size: Can accommodate a variable number of keys and child pointers.
- Balance: Techniques like splitting and merging ensure balanced tree structure.

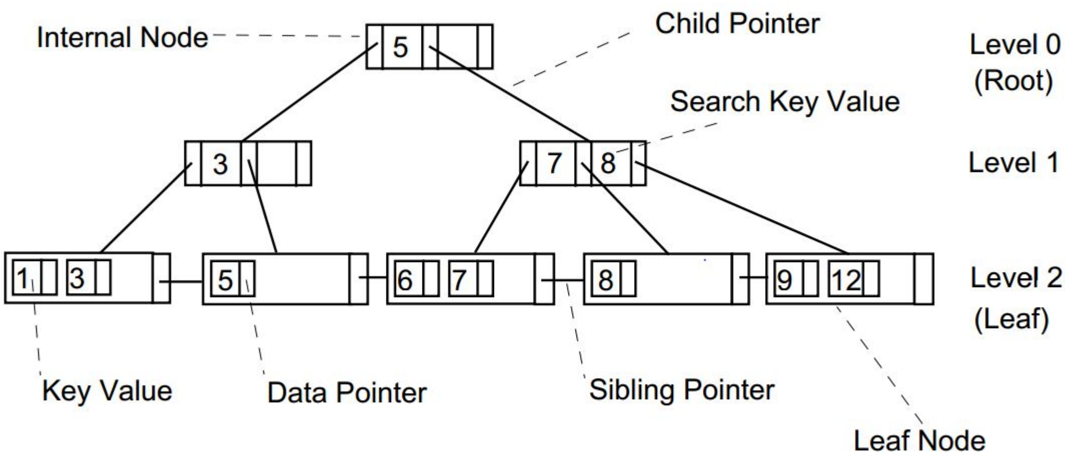


Figure 5: B+ Tree

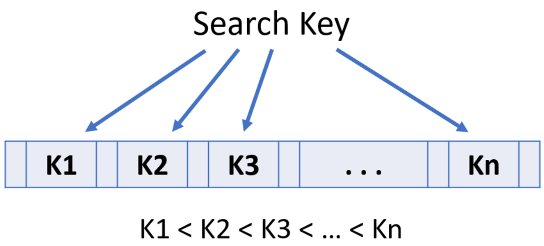


Figure 6: Internal Node Structure

4.2 Leaf Node Structure

The structure of a leaf node in a B+ tree is simple:

- Sorted Entries: Contains sorted keys and pointers to actual data entries.
- Data Entries: Stores actual data associated with the keys.
- Pointer to Next Leaf: Often linked to the next leaf node for efficient sequential access.
- Variable Size: Entries can be of fixed or variable size.
- Occupancy Management: Techniques like splitting and merging maintain optimal balance.

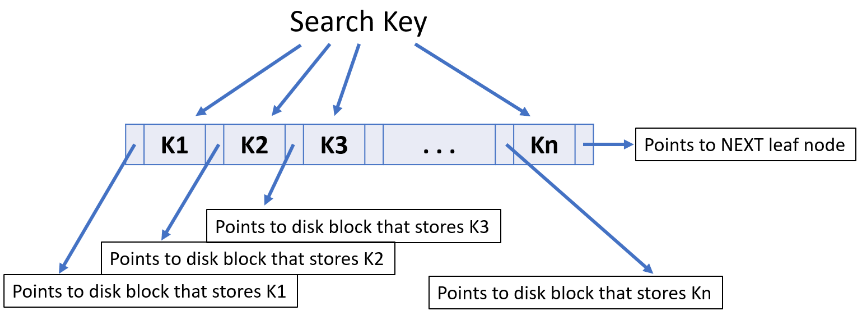


Figure 7: Leaf Node Structure

Visit here to learn more.

4.3 Node Bounds

Node bounds are crucial in B+ trees, ensuring balanced structure, efficient search, and optimal space utilization. By setting limits on the maximum number of keys and children a node can hold, these bounds prevent nodes from becoming excessively large or sparse. This balance enhances search efficiency by streamlining the traversal and narrowing down the search space.

In addition, node bounds significantly contribute to the efficiency of search operations within the tree. By limiting the number of keys stored in each node, the search process is simplified. With fewer keys to examine during traversal, search algorithms can quickly narrow down the search space and locate the desired data, leading to faster query response times.

Furthermore, node bounds provide clear guidelines for node splitting and merging procedures, essential for maintaining the tree's balance after insertions and deletions. When a node exceeds its maximum capacity, it must be split into two nodes, each adhering to the defined bounds. Similarly, if a node becomes too sparsely populated, it may need to be merged with its neighboring nodes to optimize space utilization.

In addition to preserving balance and facilitating operations, node bounds also contribute to space efficiency within the tree. By controlling the maximum number of keys and children per node, B+ trees can effectively utilize available memory resources without excessive waste or fragmentation. This efficient space management is particularly important in scenarios with limited memory or disk storage capacity.

Overall, node bounds are indispensable in B+ trees, playing a pivotal role in maintaining balance, optimizing search efficiency, guiding tree operations, and maximizing space utilization. Their careful definition and adherence are essential for the effective organization and performance of B+ tree-based data structures.

Node Type	**Min #Keys**	**Max #Keys**	**Min #Child**	**Max #Child**
Root Node	1	$M-1$	2[5]	M
Internal Node	$\lceil \frac{M}{2} \rceil - 1$	$M-1$	$\lceil \frac{M}{2} \rceil$	M
Leaf Node	$\lceil \frac{M}{2} \rceil - 1$	$M-1$	0	0

M = Order of B+ Tree

Table 1: Node bounds

5 Operations on B+ Tree

5.1 Insertion

Before adding an item to a B + tree, it is crucial to consider the following criteria:

1. The root node must have a minimum of two children.
2. Each node, excluding the root, can accommodate up to a maximum of m children and at least $\frac{m}{2}$ children.
3. Each node can hold a maximum of $m-1$ keys and a minimum of $\lceil \frac{m}{2} \rceil - 1$ keys.

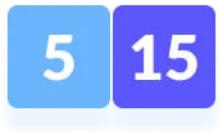
The insertion process follows these steps:

1. Locate the appropriate leaf node for insertion since every element is inserted into a leaf node.
2. Insert the key into the leaf node.
   - **Case I:**
     If the leaf node isn't at full capacity, insert the key in increasing order.
   - **Case II:**
     (a) If the leaf node reaches its full capacity, insert the key in increasing order and balance the tree by following these steps:
     (b) Split the node at the $\frac{m}{2}$th position.
     (c) Add the $\lceil \frac{m}{2} \rceil$th key to the parent node.
     (d) If the parent node is also full, repeat steps (b) to (c).

Let's see and example of insertion in order m = 3 B+ tree. The elements to be inserted are 5,15, 25, 35, 45.

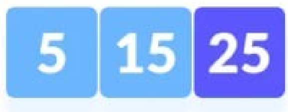
Insert 5

Insert 15

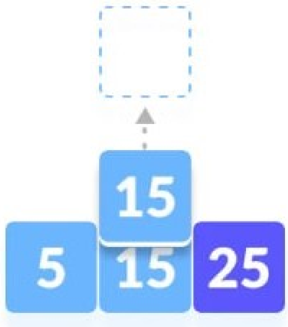


As the order m = 3, so a node can have maximum $m-1=2$ keys. After inserting 25 the node has 3 keys. Now split the node and send the $\lceil \frac{m}{2} \rceil = 2$th element(i.e. key 15) to the parents. And then attach the left half of the split node as left pointer of the parent node and right half as right pointer. Keep in mind to assign pointers if not already assigned.

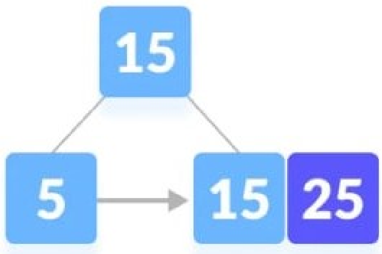
Insert 25



(i)



(ii)

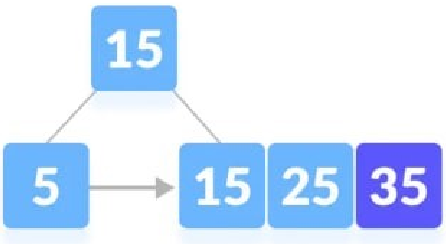


(iv)

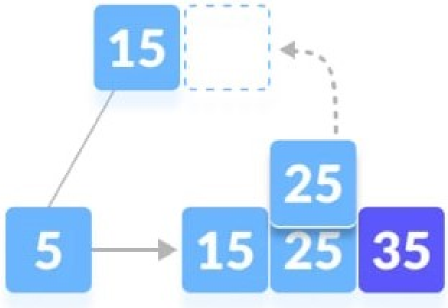
(iii)

Now we are going to insert 35 into the B + tree. At first we need to find out the position at leaf level where the new key need to be inserted. As 35 is greater than 15 we need to go to the right child of the root node. We would go left if smaller. And we will follow the process until we are at any leaf node.

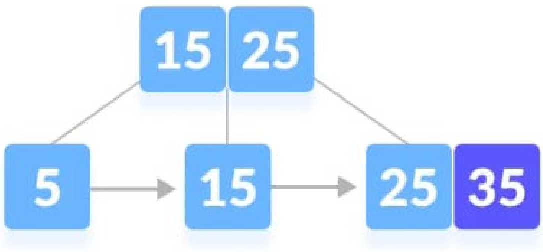
Insert 35



(i)



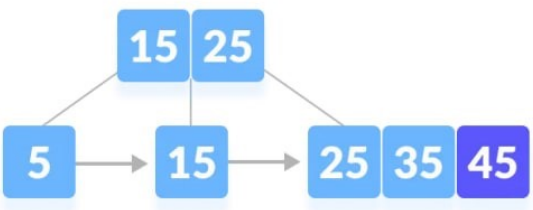
(ii)



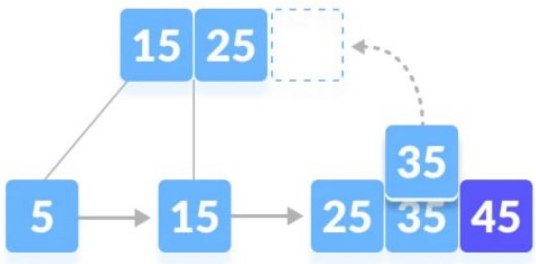
(iii)

Here we need to apply the 2nd case. After inserting 45 the leaf node is overflowing then we split the node as previous and send 35 to the parent node but the parent node already has 2 keys so it's also going to overflow. And we need to split this node also and then send 25 to the parent of the current node and assign it's left(if not already assigned) and right pointers.

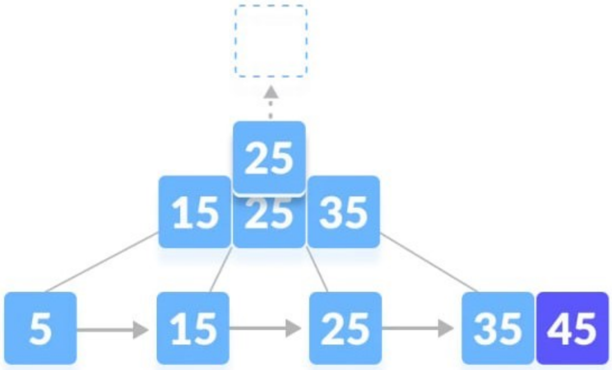
Insert 45



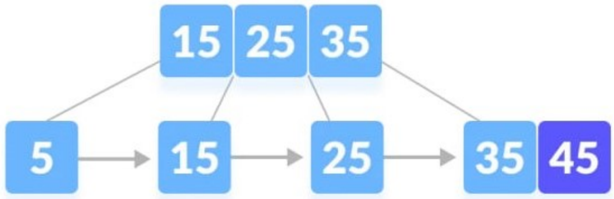
(i)



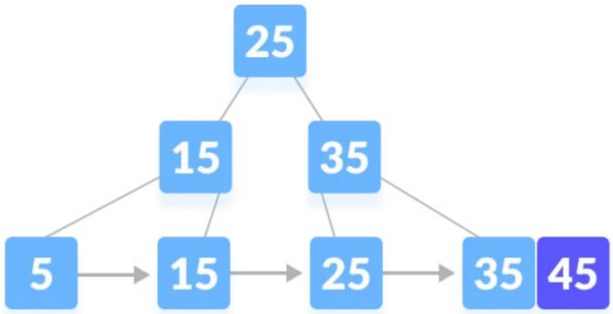
(ii)



(iv)



(iii)



(v)

5.2 Deletion from B+ Tree

Deleting an element in a B + tree consists of three main events:

- Searching the node where the key to be deleted exists.
- Deleting the key.
- Balancing the tree if required.

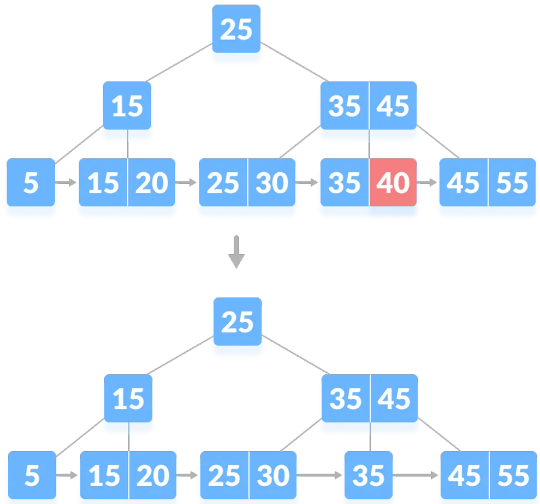
Underflow is a situation when there is less number of keys in a node than the minimum number of keys it should hold.
Before going through the steps below, one must know these facts about a B+ tree of degree m. A node can have-

1. a maximum of m children. (i.e. 3)
2. a maximum of m - 1 keys. (i.e. 2)
3. a minimum of $\lceil m/2 \rceil$ children. (i.e. 2)
4. a minimum of $\lceil m/2 \rceil - 1$ keys (except root node). (i.e., 1)

While deleting a key, we have to take care of the keys present in the internal nodes (i.e., indexes) as well because the values are redundant in a B+ tree. Search for the key to be deleted and then follow the following steps-

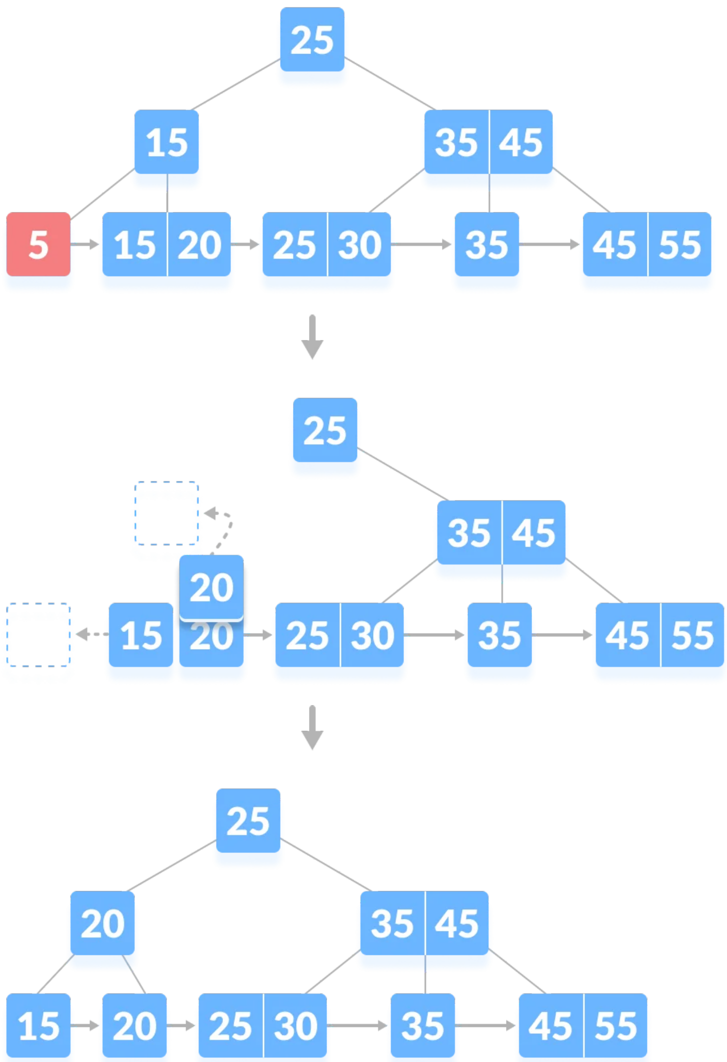
- **Case I:** The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:
  1. There is more than the minimum number of keys in the node. Simply delete the key.

Delete 40



2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.Deleting 5 from the tree below leads to this condition.

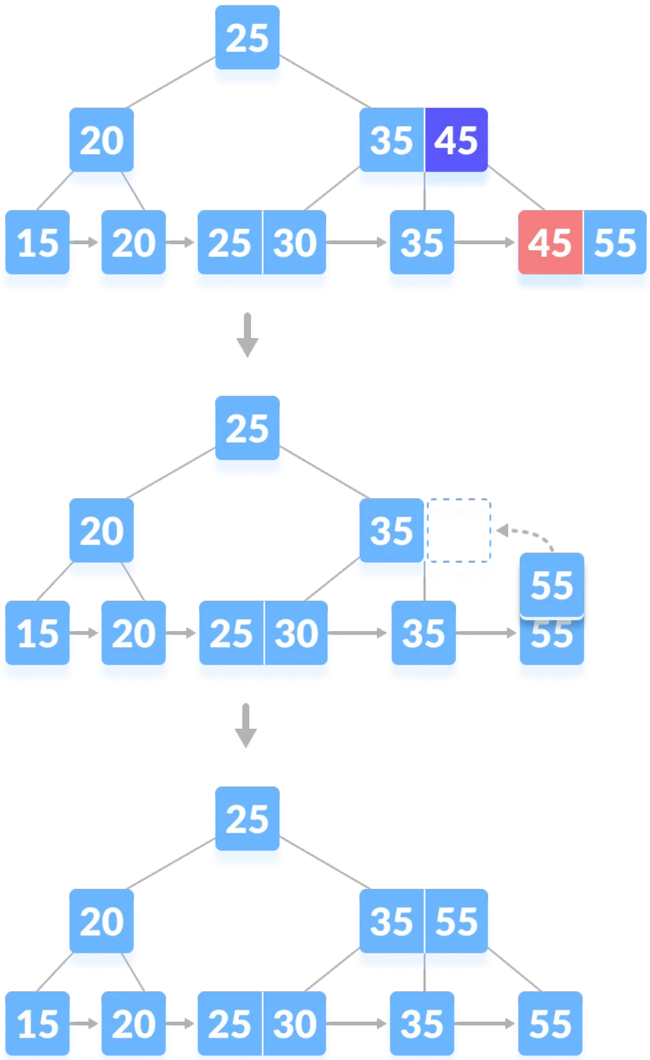
Delete 5



- **Case II:** The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

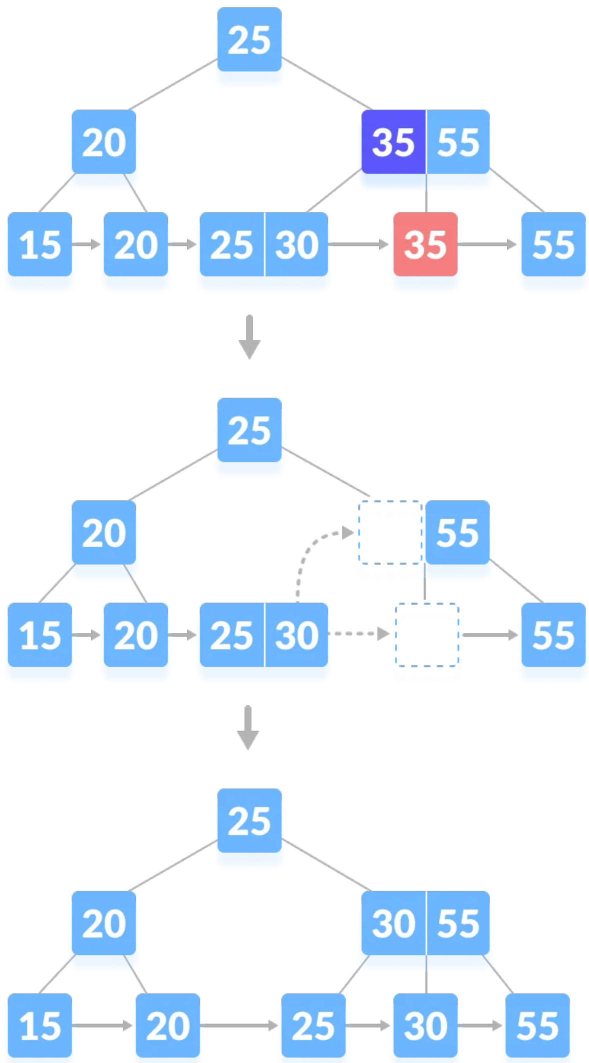
  1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well. Fill the empty space in the internal node with the inorder successor.Deleting 45 from the tree below leads to this condition.

Delete 45

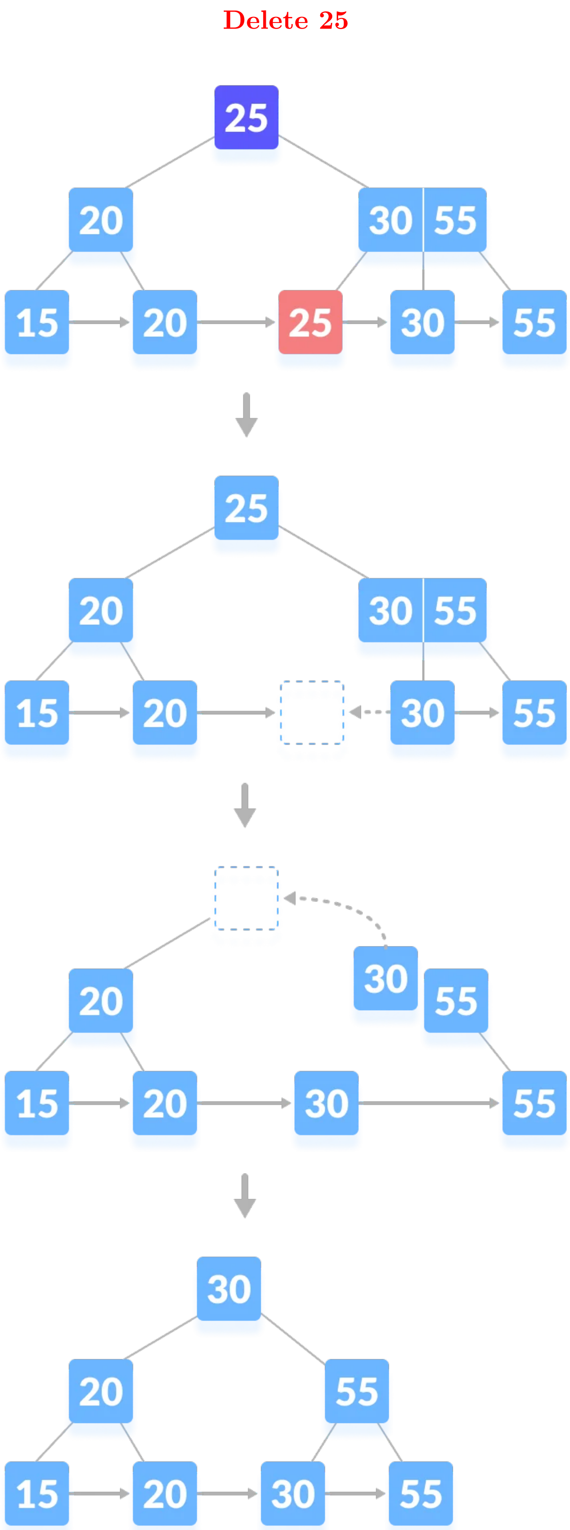


2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).Fill the empty space created in the index (internal node) with the borrowed key. Deleting 35 from the tree below leads to this condition.

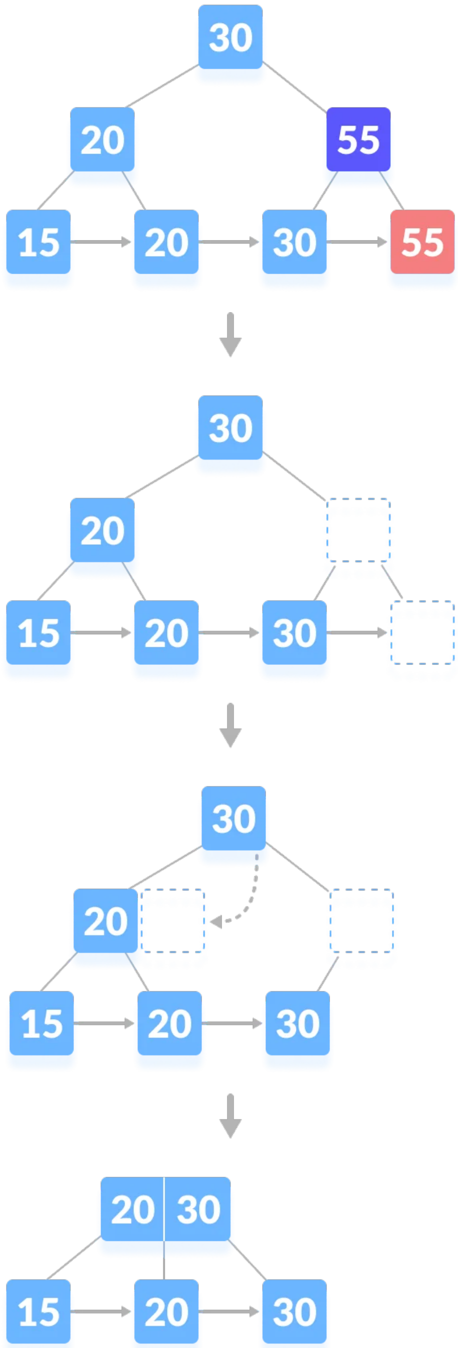
Delete 35



3. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.After deleting the key, merge the empty space with its sibling.Fill the empty space in the grand-parent node with the inorder successor. Deleting 25 from the tree below leads to this condition.



- **Case III:** In this case, the height of the tree gets shrinked. It is a little complicated.Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.



Pictures for simulating insertion and deletion are taken from www.programiz.com

5.3 Searching a key

When searching for a value k in a B+ tree, we start from the root and traverse down to find the leaf node that may contain the value k. At each internal node, we select the appropriate child node to follow.

An internal B+ tree node has at most $b \leq m$ children, where each child represents a different sub-interval. We choose the corresponding child via a linear search of the b entries. When we reach a leaf node, we perform a linear search of its n elements for the desired key.

Since we only traverse one branch of all the children at each level of the tree, we achieve a runtime of $O(\log N)$, where N is the total number of keys stored in the leaves of the B+ tree.

Function Description: Searches for a key k in the B+ tree starting from the root node root.

```
function search(k, root) is
    let leaf = leaf_search(k, root)
    for leaf_key in leaf.keys() do
        if k = leaf_key then
            return true
    return false
```

Function Description: Searches for a key k in the B+ tree starting from the node node and returns the leaf node where the key is located.

```
function leaf_search(k, node) is
    if node is a leaf then
        return node
    let p = node.children()
    let l = node.left_sided_intervals()
    assert |p| = |l| + 1
    let b = p.len()
    for i from 1 to m - 1 do
        if k <= l[i] then
            return leaf_search(k, p[i])
    return leaf_search(k, p[b])
```

6 Time Complexity of B+ Tree Operations

Consider a B+ tree having :

- m : Order of the B+ tree.
- n: Total number of elements (or keys) stored in the B+ tree.

B+ Tree Operation	**Time Complexity**
Insertion	$O(\log_m n)$
Deletion	$O(\log_m n)$
Search	$O(\log_m n)$

Table 2: Time Complexity

1. **Search Operation (Find):**
    - **Time Complexity:** $O(\log_m^n)$
    - **Explanation:** Searching in a B+ tree involves traversing from the root to the leaf level, which takes $O(\log_m^n)$ time, where n is the number of elements in the tree and m is the order of the tree. Since B+ trees are balanced, the height of the tree is logarithmic with respect to the number of elements and the order of the tree.
2. **Insertion Operation:**
    - **Time Complexity:** $O(\log_m^n)$
    - **Explanation:** Inserting a new element into a B+ tree involves searching for the correct position to insert the element ($O(\log_m^n)$) and potentially splitting nodes along the path to maintain the B+ tree properties. Splitting nodes may propagate up to the root, but the overall complexity remains $O(\log_m^n)$.
3. **Deletion Operation:**

- **Time Complexity:** $O(\log_m^n)$
- **Explanation:** Deleting an element from a B+ tree also involves searching for the element to delete ($O(\log_m^n)$) and potentially merging or redistributing nodes to maintain the B+ tree properties. Similar to insertion, the overall complexity remains $O(\log_m^n)$.

4. **Range Query Operation:**
   - **Time Complexity:** $O(\log_m^n + k)$
   - **Explanation:** Retrieving elements within a given range involves searching for the starting and ending points of the range ($O(\log_m^n)$) and traversing leaf nodes to collect elements falling within the range ($O(k)$, where k is the number of elements in the range).

5. **Splitting and Merging:**
   - **Time Complexity:** $O(1)$
   - **Explanation:** When splitting or merging nodes during insertion or deletion, the time complexity is constant per operation. However, these operations might propagate upwards, potentially affecting the entire height of the tree, but this is considered amortized constant time per operation.

7 Advantages of B+ Trees

- **Efficient Search and Range Queries:** B+ trees provide efficient search operations and are optimized for range queries. They maintain data in sorted order, enabling logarithmic time complexity for search operations and efficient retrieval of ranges of data [3].
- **Concurrency Control and Performance** B+ trees support efficient concurrency control mechanisms, making them suitable for concurrent database systems. They offer predictable performance characteristics and ensure consistency and isolation among concurrent transactions. If we use LOCKING for concurrency control, we need to update a single node at the same time, which can be done using B+ Tree very efficiently. If we use MULTI VERSION CONCURRENCY CONTROL(MVCC) , we have to update multiple nodes at the same time. It can be done using the persistence nature of B+ Tree. B+ Tree can be used as a persistent data structure that can remember its previous versions efficiently. We will only need to create logn more nodes every time that will be updated. [1].
- **Optimal Disk Access and Cache Efficiency:** B+ trees are designed to optimize disk I/O operations and maximize cache efficiency. They utilize node-based structures and node sizes that align well with the block size of storage devices, minimizing disk access and maximizing cache hits. [6].
- **Support for Large Datasets:** B+ trees are suitable for managing large datasets efficiently. They can handle a large number of keys while maintaining logarithmic search and update times.For instance, if the number of data n = 10^6 and order of tree is 100 then the height of the tree will be only 3. As a result , all the operations will be immensely fast. [6].
- **Ordered Structure for Sequential Access:** B+ trees maintain data in sorted order, facilitating efficient sequential access to keys. This property simplifies operations such as range scans and sequential processing of data [4].
- **Balanced Tree:** B+ trees are balanced trees, ensuring that the height of the tree remains minimal. This balance ensures that operations such as insertion, deletion, and search have consistent performance characteristics [2].

8 Real Life Application of B+ Tree

- **Database Indexing:**
  B+ trees are widely used in database management systems for indexing. They enable effective retrieval of data based on indexed characteristics, resulting in faster query execution times. For example, in a customer database, a B+ tree index on the "customerId" property may be used to quickly retrieve customer records by ID.
- **File Systems:**
  In file systems, B+ trees are used to manage and organize file information and directory hierarchies. They make it easier to store and retrieve file metadata, including size, creation date, and rights, and speed up file path lookup. For example, B+ trees are used for directory indexing in the Ext4 file system, which is widely found in Linux versions.

- **Distributed Databases:**
  In distributed databases, where data is spread across multiple nodes, B+ trees are employed for maintaining distributed indexes. These indexes allow for efficient query processing and data retrieval across distributed nodes while ensuring data consistency and fault tolerance.

- **Geo-spatial Databases:**
  B+ trees are well-suited for indexing Geo-spatial data such as coordinates, shapes, and spatial relationships. They enable efficient spatial queries such as range searches, nearest-neighbor searches, and spatial joins. Geo-spatial databases, used in applications such as Geographic Information Systems (GIS) and location-based services, often utilize B+ trees for spatial indexing.

- **File Sharing Networks:**
  B+ trees can be utilized in peer-to-peer file sharing networks for maintaining distributed indexes of available files. They facilitate efficient lookup and retrieval of files based on various attributes such as file name, size, and type, thereby enhancing the overall performance of the file sharing network.

- **Web Browsers:**
  B+ trees are employed in web browsers for managing bookmarks and history data. They enable quick retrieval of URLs based on search queries and facilitate efficient storage and retrieval of browsing history, thereby enhancing the browsing experience for users.

9 A Sample Implementation of B+ Tree Using Python

```python
import math

# Node creation
class Node:
    def __init__(self, order):
        self.order = order
        self.values = []
        self.keys = []
        self.nextKey = None
        self.parent = None
        self.check_leaf = False

    # Insert at the leaf
    def insert_at_leaf(self, leaf, value, key):
        if (self.values):
            temp1 = self.values
            for i in range(len(temp1)):
                if (value == temp1[i]):
                    self.keys[i].append(key)
                    break
                elif (value < temp1[i]):
                    self.values = self.values[:i] + [value] + self.values[i:]
                    self.keys = self.keys[:i] + [[key]] + self.keys[i:]
                    break
                elif (i + 1 == len(temp1)):
                    self.values.append(value)
                    self.keys.append([key])
                    break
        else:
            self.values = [value]
            self.keys = [[key]]

# B plus tree
class BplusTree:
    def __init__(self, order):
        self.root = Node(order)
        self.root.check_leaf = True

    # Insert operation
    def insert(self, value, key):
        value = str(value)
        old_node = self.search(value)
        old_node.insert_at_leaf(old_node, value, key)

        if (len(old_node.values) == old_node.order):
            node1 = Node(old_node.order)
            node1.check_leaf = True
```

```
            node1.parent = old_node.parent
            mid = int(math.ceil(old_node.order / 2)) - 1
            node1.values = old_node.values[mid + 1:]
            node1.keys = old_node.keys[mid + 1:]
            node1.nextKey = old_node.nextKey
            old_node.values = old_node.values[:mid + 1]
            old_node.keys = old_node.keys[:mid + 1]
            old_node.nextKey = node1
            self.insert_in_parent(old_node, node1.values[0], node1)

    # Search operation for different operations
    def search(self, value):
        current_node = self.root
        while(current_node.check_leaf == False):
            temp2 = current_node.values
            for i in range(len(temp2)):
                if (value == temp2[i]):
                    current_node = current_node.keys[i + 1]
                    break
                elif (value < temp2[i]):
                    current_node = current_node.keys[i]
                    break
                elif (i + 1 == len(current_node.values)):
                    current_node = current_node.keys[i + 1]
                    break
        return current_node

    # Find the node
    def find(self, value, key):
        l = self.search(value)
        for i, item in enumerate(l.values):
            if item == value:
                if key in l.keys[i]:
                    return True
                else:
                    return False
        return False

    # Inserting at the parent
    def insert_in_parent(self, n, value, ndash):
        if (self.root == n):
            rootNode = Node(n.order)
            rootNode.values = [value]
            rootNode.keys = [n, ndash]
            self.root = rootNode
            n.parent = rootNode
            ndash.parent = rootNode
            return

        parentNode = n.parent
        temp3 = parentNode.keys
        for i in range(len(temp3)):
            if (temp3[i] == n):
                parentNode.values = parentNode.values[:i] + \
                    [value] + parentNode.values[i:]
                parentNode.keys = parentNode.keys[:i +
                                                  1] + [ndash] + parentNode.keys[i + 1:]
                if (len(parentNode.keys) > parentNode.order):
                    parentdash = Node(parentNode.order)
                    parentdash.parent = parentNode.parent
                    mid = int(math.ceil(parentNode.order / 2)) - 1
                    parentdash.values = parentNode.values[mid + 1:]
                    parentdash.keys = parentNode.keys[mid + 1:]
                    value_ = parentNode.values[mid]
                    if (mid == 0):
                        parentNode.values = parentNode.values[:mid + 1]
                    else:
                        parentNode.values = parentNode.values[:mid]
                    parentNode.keys = parentNode.keys[:mid + 1]
                    for j in parentNode.keys:
                        j.parent = parentNode
                    for j in parentdash.keys:
                        j.parent = parentdash
                    self.insert_in_parent(parentNode, value_, parentdash)

    # Delete a node
```

```python
    def delete(self, value, key):
        node_ = self.search(value)

        temp = 0
        for i, item in enumerate(node_.values):
            if item == value:
                temp = 1

                if key in node_.keys[i]:
                    if len(node_.keys[i]) > 1:
                        node_.keys[i].pop(node_.keys[i].index(key))
                    elif node_ == self.root:
                        node_.values.pop(i)
                        node_.keys.pop(i)
                    else:
                        node_.keys[i].pop(node_.keys[i].index(key))
                        del node_.keys[i]
                        node_.values.pop(node_.values.index(value))
                        self.deleteEntry(node_, value, key)
                else:
                    print("Value not in Key")
                    return
        if temp == 0:
            print("Value not in Tree")
            return

    # Delete an entry
    def deleteEntry(self, node_, value, key):

        if not node_.check_leaf:
            for i, item in enumerate(node_.keys):
                if item == key:
                    node_.keys.pop(i)
                    break
            for i, item in enumerate(node_.values):
                if item == value:
                    node_.values.pop(i)
                    break

        if self.root == node_ and len(node_.keys) == 1:
            self.root = node_.keys[0]
            node_.keys[0].parent = None
            del node_
            return
        elif (len(node_.keys) < int(math.ceil(node_.order / 2)) and node_.check_leaf == False) or (len(node_.values) < int(math.ceil((node_.order - 1) / 2)) and node_.check_leaf == True):

            is_predecessor = 0
            parentNode = node_.parent
            PrevNode = -1
            NextNode = -1
            PrevK = -1
            PostK = -1
            for i, item in enumerate(parentNode.keys):

                if item == node_:
                    if i > 0:
                        PrevNode = parentNode.keys[i - 1]
                        PrevK = parentNode.values[i - 1]

                    if i < len(parentNode.keys) - 1:
                        NextNode = parentNode.keys[i + 1]
                        PostK = parentNode.values[i]

            if PrevNode == -1:
                ndash = NextNode
                value_ = PostK
            elif NextNode == -1:
                is_predecessor = 1
                ndash = PrevNode
                value_ = PrevK
            else:
                if len(node_.values) + len(NextNode.values) < node_.order:
                    ndash = NextNode
                    value_ = PostK
```

```
                else:
                    is_predecessor = 1
                    ndash = PrevNode
                    value_ = PrevK

            if len(node_.values) + len(ndash.values) < node_.order:
                if is_predecessor == 0:
                    node_, ndash = ndash, node_
                ndash.keys += node_.keys
                if not node_.check_leaf:
                    ndash.values.append(value_)
                else:
                    ndash.nextKey = node_.nextKey
                ndash.values += node_.values

                if not ndash.check_leaf:
                    for j in ndash.keys:
                        j.parent = ndash

                self.deleteEntry(node_.parent, value_, node_)
                del node_
            else:
                if is_predecessor == 1:
                    if not node_.check_leaf:
                        ndashpm = ndash.keys.pop(-1)
                        ndashkm_1 = ndash.values.pop(-1)
                        node_.keys = [ndashpm] + node_.keys
                        node_.values = [value_] + node_.values
                        parentNode = node_.parent
                        for i, item in enumerate(parentNode.values):
                            if item == value_:
                                p.values[i] = ndashkm_1
                                break
                    else:
                        ndashpm = ndash.keys.pop(-1)
                        ndashkm = ndash.values.pop(-1)
                        node_.keys = [ndashpm] + node_.keys
                        node_.values = [ndashkm] + node_.values
                        parentNode = node_.parent
                        for i, item in enumerate(p.values):
                            if item == value_:
                                parentNode.values[i] = ndashkm
                                break
                else:
                    if not node_.check_leaf:
                        ndashp0 = ndash.keys.pop(0)
                        ndashk0 = ndash.values.pop(0)
                        node_.keys = node_.keys + [ndashp0]
                        node_.values = node_.values + [value_]
                        parentNode = node_.parent
                        for i, item in enumerate(parentNode.values):
                            if item == value_:
                                parentNode.values[i] = ndashk0
                                break
                    else:
                        ndashp0 = ndash.keys.pop(0)
                        ndashk0 = ndash.values.pop(0)
                        node_.keys = node_.keys + [ndashp0]
                        node_.values = node_.values + [ndashk0]
                        parentNode = node_.parent
                        for i, item in enumerate(parentNode.values):
                            if item == value_:
                                parentNode.values[i] = ndash.values[0]
                                break

                if not ndash.check_leaf:
                    for j in ndash.keys:
                        j.parent = ndash
                if not node_.check_leaf:
                    for j in node_.keys:
                        j.parent = node_
                if not parentNode.check_leaf:
                    for j in parentNode.keys:
                        j.parent = parentNode

# Print the tree
```

```python
def printTree(tree):
    lst = [tree.root]
    level = [0]
    leaf = None
    flag = 0
    lev_leaf = 0

    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):
        x = lst.pop(0)
        lev = level.pop(0)
        if (x.check_leaf == False):
            for i, item in enumerate(x.keys):
                print(item.values)
        else:
            for i, item in enumerate(x.keys):
                print(item.values)
            if (flag == 0):
                lev_leaf = lev
                leaf = x
                flag = 1

#simulate
record_len = 3
bplustree = BplusTree(record_len)
bplustree.insert('5', '33')
bplustree.insert('15', '21')
bplustree.insert('25', '31')
bplustree.insert('35', '41')

printTree(bplustree)
bplustree.delete('15','21')
printTree(bplustree)

if(bplustree.find('5', '34')):
    print("Found")
else:
    print("Not found")
```

Find the code here

10 Discussion

Initially, grasping the concept of B+ trees seemed daunting due to their complex structure and operations. However, as we delved deeper into the topic, we gradually gained clarity and appreciation for their design principles.

Engaging in hands-on practice, including implementing B+ trees from scratch and working on practical exercises, significantly enhanced our understanding. It allowed us to visualize the inner workings of B+ trees and appreciate their efficiency in real-world scenarios.

While learning about B+ trees, we encountered challenges in understanding certain aspects such as node splitting, merging, and balancing. However, with perseverance and guidance, we overcame these challenges and strengthened our grasp of the concepts. Collaborating with peers and discussing concepts, challenges, and solutions greatly enriched our learning experience. Sharing insights, exploring different perspectives, and collaborating on the presentation fostered a supportive learning environment.

Learning about B+ trees has not only expanded our technical knowledge, but also sharpened our problem-solving and critical thinking skills. It has equipped us with a powerful tool for efficiently managing and accessing large datasets in diverse applications.

References

[1] Rakesh Agrawal, Michael J Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[2] Rudolf Bayer and Edward M McCreight. The design and analysis of b+ trees: A dynamic database indexing method. In *Proceedings of the 1972 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 121–141. ACM, 1972.

[3] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[4] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database system implementation*, volume 672. Prentice Hall Upper Saddle River, 2000.

[5] Shamkant B. Navathe, Ramez Elmasri. *Fundamentals of Database Systems.* Pearson Education, Upper Saddle River, N.J., 6th edition, 2010.

[6] Patrick E. O'Neil, Elizabeth J. O'Neil, and Gerhard Weikum. Modern b-tree techniques. *ACM Computing Surveys (CSUR)*, 24(2):123–160, 1992.