B-TREE

Jyotirupa Basumatary (A125007)
Pragya Srivastava (A125013)
Rishita Rai (A125018)

Under the Guidance of
Dr. Ajaya Kumar Dash

Department of Computer Science and Engineering
IIIT Bhubaneswar

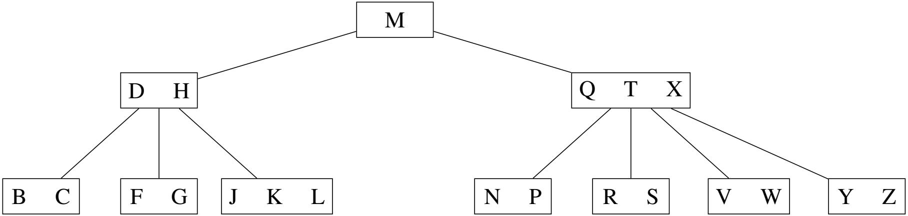
5 January 2026

Outline

Introduction to B-Tree

- A **B-tree** is a self-balancing tree data structure that maintains sorted data and supports **searching, sequential access, insertion, and deletion** in **logarithmic time**.
- It is specifically designed to work efficiently on **magnetic disks and secondary storage devices**, where minimizing disk access is critical.
- A B-tree generalizes the **Binary Search Tree** by allowing each node to have **more than two children**, which reduces the height of the tree and improves performance.

Example of B-Tree



Note:

If an internal node X contains $n[X]$ keys, then X has exactly $n[X] + 1$ children. Moreover, all leaf nodes in a B-Tree appear at the same depth.

Properties of B-Tree

Property 1:
Every node X has the following properties:

- $n[X]$ denotes the number of keys stored in node X.
- The $n[X]$ keys in node X are stored in non-decreasing (sorted) order:

$$key_1[X] \leq key_2[X] \leq \cdots \leq key_{n[X]}[X]$$

- If X is a leaf node, it contains only keys and no children.

Properties of B-Tree (contd..)

Property 2:

- If an internal node X contains $n[X]$ keys, then X has exactly $n[X] + 1$ children.
- These children are denoted as $C_1[X], C_2[X], \ldots, C_{n[X]+1}[X]$.

Properties of B-Tree (contd..)

Property 3:

- The keys $key_i[X]$ stored in an internal node X separate the ranges of keys stored in its subtrees.
- If node X contains the keys $key_1[X], key_2[X], \ldots, key_{n[X]}[X]$, then the keys in the corresponding subtrees satisfy:

$$k_1 \le key_1[X] \le k_2 \le key_2[X] \le \cdots \le k_{n[X]} \le key_{n[X]}[X] \le k_{n[X]+1}$$

Properties of B-Tree (contd..)

Property 4:

- All leaf nodes of a B-Tree appear at the same depth.
- Hence, a B-Tree is a **height-balanced** tree.

Properties of B-Trees (contd..)

Property 5:

- There are lower and upper bounds on the number of keys a node can contain.
- These bounds are expressed using a parameter $t \geq 2$, called the **minimum degree** of the B-Tree.
- **Lower bound:** Every node other than the root contains at least $t - 1$ keys and at least t children.
- **Upper bound:** Every node contains at most $2t - 1$ keys and at most $2t$ children.

Note

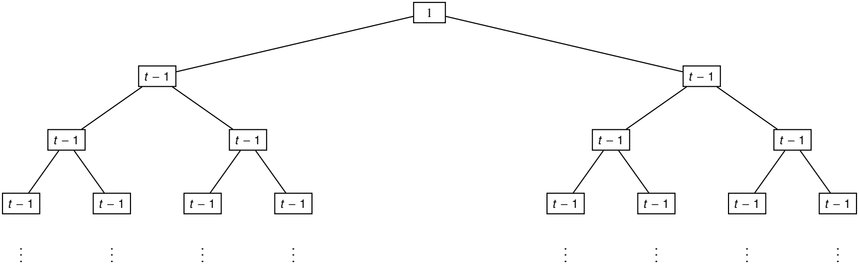
When the minimum degree $t = 2$, every internal node has either 2, 3, or 4 children. Such a B-Tree is called a **2–3–4 Tree**.

Theorem of Height Proof

Theorem:
If $n \geq 1$, then for any n-key B-Tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Proof: B-Tree (Height Proof)



Observation

This diagram represents a **minimum-key B-Tree** of height h, which is used to derive an upper bound on the height of the B-Tree.

Proof (contd..)

- From the above diagram, the minimum number of nodes at each depth is given by:

$$\begin{aligned}
\text{Depth } 0 &: \ 1 \\
\text{Depth } 1 &: \ 2 \\
\text{Depth } 2 &: \ 2t \\
\text{Depth } 3 &: \ 2t^2 \\
\text{Depth } i &: \ 2t^{i-1} \\
\text{Depth } h &: \ 2t^{h-1}
\end{aligned}$$

Proof (contd..)

Let T be a B-Tree of height h and minimum degree $t \geq 2$.

From the minimum-key B-Tree:

- Root contains exactly 1 key.
- Every non-root node contains exactly $t - 1$ keys (minimum case).
- The minimum number of nodes at depth i is

$$2t^{i-1}, \quad 1 \leq i \leq h$$

Hence, the minimum number of keys $n_{\min}$ in T is

$$n_{\min} = 1 + (t - 1)\sum_{i=1}^{h} 2t^{i-1}$$

Proof (contd..)

Rewrite the summation:

$$n_{\min} = 1 + 2(t-1)\sum_{i=0}^{h-1} t^i$$

Using the geometric series formula:

$$\sum_{i=0}^{h-1} t^i = \frac{t^h - 1}{t - 1}$$

Substituting:

$$n_{\min} = 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right)$$

$$n_{\min} = 1 + 2(t^h - 1)$$

$$n_{\min} = 2t^h - 1$$

Proof (contd..)

Since

$$n \geq n_{\min},$$

$$n \geq 2t^h - 1$$

$$n + 1 \geq 2t^h$$

$$t^h \leq \frac{n+1}{2}$$

Taking logarithm base t:

Result

$$h \leq \log_t\left(\frac{n+1}{2}\right)$$

Hence proved.

Operations on B-Tree

- Search
- Insert
- Delete

Search Operation in B-Tree

- Keys in each node are stored in sorted order.
- Search starts from the root node.
- If the key matches a node key, the search is successful.
- Otherwise, the search moves to the appropriate child subtree.
- If a leaf node is reached without a match, the search fails.

Search Algorithm

Algorithm: B-Tree-Search(x, k)

```
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
end while
if i ≤ n[x] and k = key_i[x] then
    return (x, i)
else if leaf[x] = TRUE then
    return NIL
else
    DISK-READ(C_i[x])
    return B-Tree-Search(C_i[x], k)
end if
```

Search (contd..)

Time Complexity

The search operation in a B-Tree takes

$$O(\log n)$$

time, where n is the number of keys in the B-Tree.

Insert Operation in B-Tree

- Insertion always starts at the **root** of the B-Tree.
- The new key is inserted into the appropriate **leaf node**.
- If the leaf node has fewer than $2t - 1$ keys, the key is inserted directly.
- If a node becomes **full**, it is split and the middle key is promoted to the parent node.
- Splitting may propagate upward and may create a new root.
- The B-Tree remains **balanced** after insertion.

Insertion Algorithm

procedure B-Tree-Insert(x, k)

```
find i such that x : keys[i] > k or i ≥ numkeys(x)
if x is a leaf then
    Insert k into x.key at position i
else
    if x.child[i] is full then
        Split x : child[i]
        if k > x : key[i] then
            i ← i + 1
        end if
    end if
    B-Tree-Insert(x : child[i]; k)
end if
```

Example: Insertion in a B-Tree

Given:

- B-Tree order: $t = 2$
- Minimum keys per node: $t - 1 = 1$
- Maximum keys per node: $2t - 1 = 3$
- Maximum number of children: $2t = 4$
- Keys to be inserted:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

Example (contd..)

Step 1: Insert 1

- The B-Tree is initially empty.
- A root node is created.
- Key 1 is inserted into the root.
- The root is also a leaf node.

$\boxed{1}$

Example (contd..)

Step 2: Insert 2

- The root node currently contains key 1.
- The root is not full (maximum keys = 3).
- Key 2 is inserted into the root in sorted order.
- The root remains a leaf node.

1	2

Example (contd..)

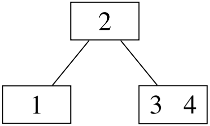
Step 3: Insert 3

- The root node currently contains keys 1 and 2.
- The root can hold up to 3 keys (since $2t - 1 = 3$).
- Key 3 is inserted into the root in sorted order.
- The root becomes full but no split is required yet.

1	2	3

Step 4: Insert 4

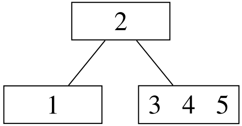
- The root node contains keys 1, 2, and 3 and is full.
- Before inserting 4, the root must be split.
- The middle key 2 is promoted to become the new root.
- The remaining keys form two child nodes: [1] and [3].
- Key 4 is inserted into the right child.



Example (contd..)

Step 5: Insert 5

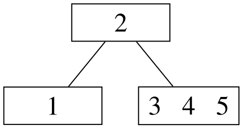
- The root contains key 2 with two children.
- Key 5 belongs to the right subtree of the root.
- The right child currently contains keys 3 and 4.
- The node is not full, so key 5 is inserted in sorted order.
- No split is required.



Example (contd..)

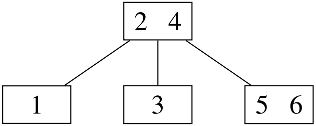
Step 5: Insert 5

- The root contains key 2 with two children.
- Key 5 belongs to the right subtree of the root.
- The right child currently contains keys 3 and 4.
- The node is not full, so key 5 is inserted in sorted order.
- No split is required.



Step 6: Insert 6

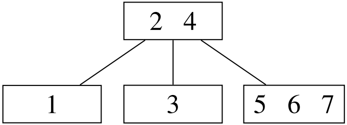
- The right child node contains keys 3, 4, and 5 and is full.
- Before inserting 6, the right child must be split.
- The middle key 4 is promoted to the root.
- The split creates two child nodes with keys [3] and [5].
- Key 6 is inserted into the rightmost child.



Example (contd..)

Step 7: Insert 7

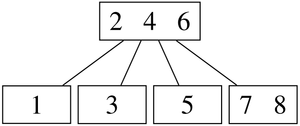
- The root node contains keys 2 and 4.
- Key 7 belongs to the rightmost subtree.
- The rightmost child currently contains keys 5 and 6.
- The node is not full, so key 7 is inserted in sorted order.
- No split is required.



Example (contd..)

Step 8: Insert 8

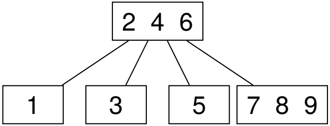
- The rightmost child contains keys 5, 6, and 7 and is full.
- Before inserting 8, this node must be split.
- The middle key 6 is promoted to the root.
- The split results in two nodes containing keys [5] and [7].
- Key 8 is inserted into the new rightmost child.



Example (contd..)

Step 9: Insert 9

- The root contains keys $[2, 4, 6]$.
- The root is not overfull (maximum 3 keys).
- Since $9 > 6$, it is inserted into the rightmost leaf.
- No split is required at this step.



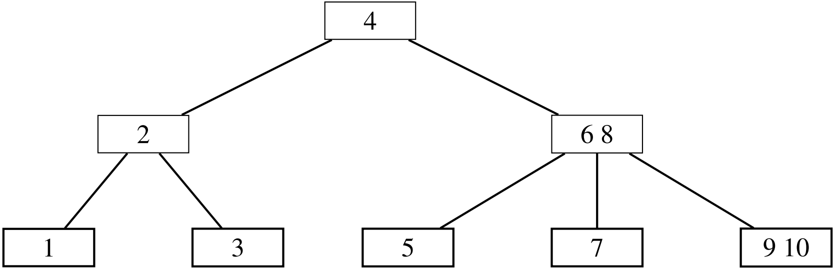
Example (contd..)

Step 10: Insert 10

- Insert key 10 into the rightmost leaf.
- The leaf becomes overfull: $[7, 8, 9, 10]$.
- The leaf is split and the middle key 8 is promoted.
- Promotion causes the root to overflow.
- The root is split and a new root is created.

Insert (contd..)

- Final Tree after Insertion from 1 to 10



Insert (contd..)

Time Complexity

The time complexity of inserting a key into a B-Tree with n keys and minimum degree t is:

$$\boxed{O(t \log_t n)}$$

If t is treated as a constant, the complexity becomes:

$$\boxed{O(\log n)}$$

Delete Operation in B-Tree

Why Deletion is Complex

Deletion in a B-Tree is more complex than insertion because the tree must **always satisfy the B-Tree properties**.

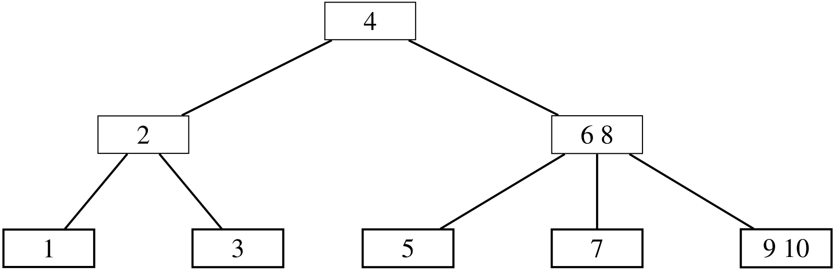
- The key to be deleted may be in a **leaf node** or an **internal node**.
- Before deleting a key, we ensure the node has at least t keys.
- If a node has fewer than t keys, it is **fixed before deletion**.
- Deletion maintains:
  - Minimum number of keys $(t - 1)$ in each node
  - Balanced height of the B-Tree

Various Cases of Deletion

Case 1: If the key k is in node x and x is a leaf node

- If the key k is present in node x and x is a leaf,
- then delete the key k directly from node x.

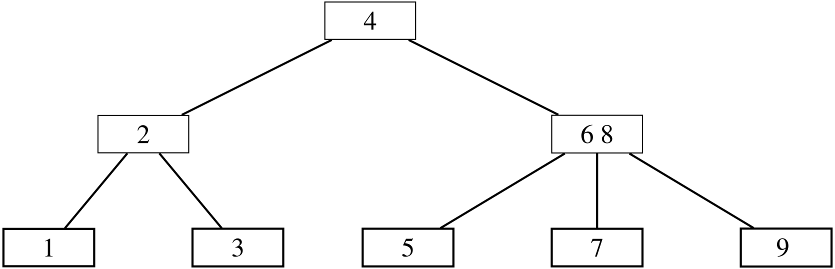
Example:



Delete Case 1 (contd..)

- Delete key $k = 10$ from the leaf node [9 10].
- After deletion, the leaf becomes [9].
- No rebalancing is required since the node still satisfies B-Tree properties.

After deletion, the tree is:



Deletion (contd..)

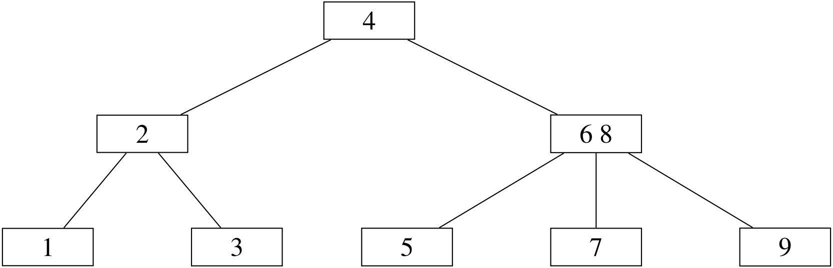
Case 2: If the key k is in an internal node x

- Let y be the child preceding key k and z be the child following k.
- If y has at least t keys:
  - Find the predecessor k' of k in subtree rooted at y.
  - Replace k with k' and recursively delete k'.
- Else if z has at least t keys:
  - Find the successor k' of k in subtree rooted at z.
  - Replace k with k' and recursively delete k'.
- Otherwise (both y and z have $t - 1$ keys):
  - Merge k and all keys of z into y.
  - Delete k recursively from the merged node.

Example of Predecessor

Step 1 (Original Tree)

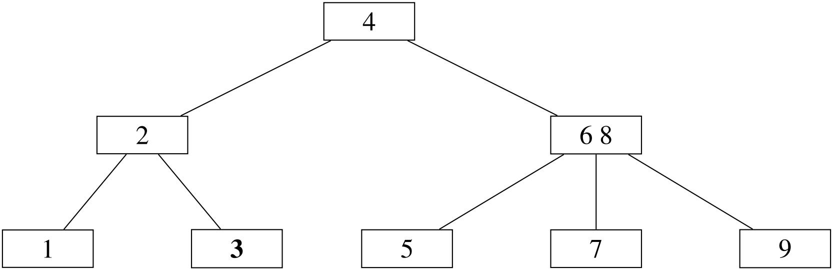
- Key to be deleted: $k = 4$
- The key is present in an internal node.



Delete Case 2 (contd..)

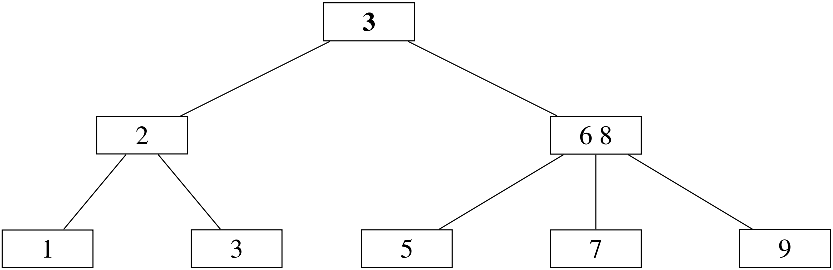
Step 2 (Find Predecessor)

- The predecessor of 4 is the largest key in its left subtree.
- The predecessor is 3.



Step 3 (Replacement)

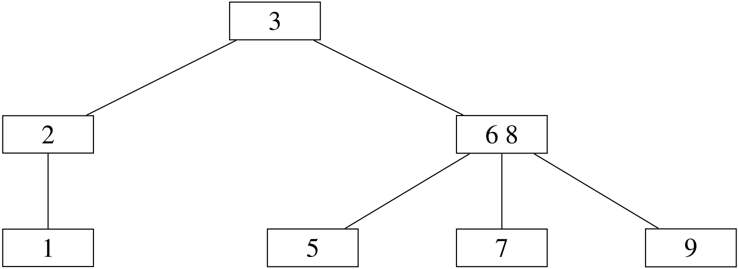
- Replace key 4 with its predecessor 3.
- Now the key 3 appears twice.



Delete Case 2 (contd..)

Step 4 (Final Tree)

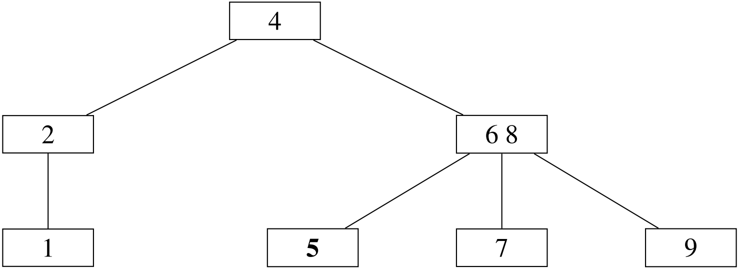
- Delete the duplicate key 3 from the leaf node.
- B-Tree properties are preserved.



Delete Case 2 (contd..)

Successor (Before Deletion)

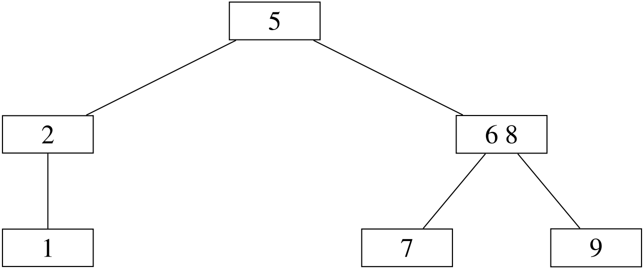
- Key $k = 4$ is in an internal node.
- Left child has minimum keys.
- Right child has at least t keys.



Delete Case 2 (contd..)

Successor (After Deletion)

- Replace key 4 with its successor 5.
- Delete 5 from the right subtree.
- B-Tree properties are preserved.



Deletion (contd..)

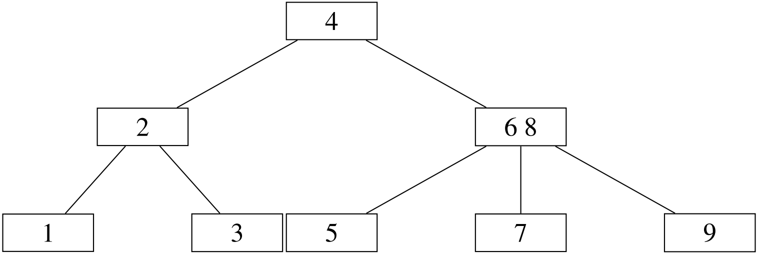
Case 3: If the key k is not present in internal node x

- Determine the child $x.c(i)$ that should contain k.
- If $x.c(i)$ has at least t keys:
  - Recursively delete k from $x.c(i)$.
- If $x.c(i)$ has only $t-1$ keys:
  - Borrow a key from an adjacent sibling if possible, or
  - Merge $x.c(i)$ with a sibling and a key from x.

Delete Case 3 (contd..)

Example (Minimum degree $t = 2$): Delete key 1

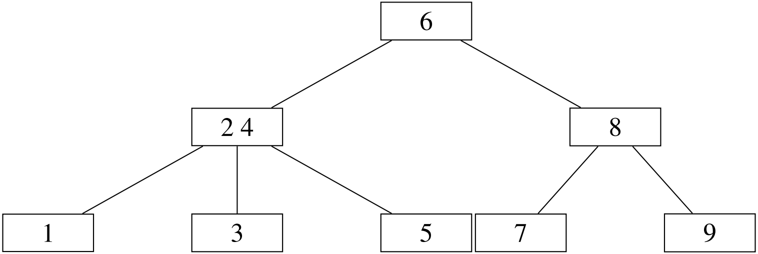
- Key 1 is not present in the root.
- It must be in the left subtree.
- The child node has only $t - 1 = 1$ key.



Delete Case 3 (contd..)

Case 3(a): Borrow from Sibling

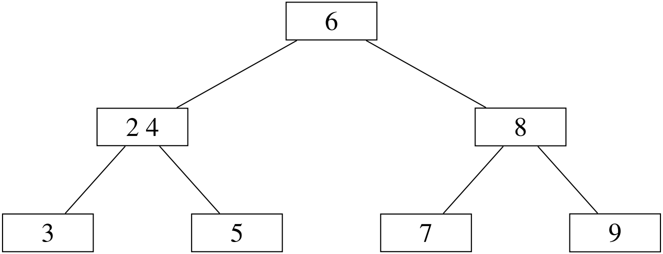
- The left child has only $t-1$ keys.
- Its right sibling has at least t keys.
- A key is borrowed via the parent.



Delete Case 3 (contd..)

Case 3: Recursive Delete

- The target child now has at least t keys.
- Key 1 is deleted safely from the leaf.
- B-Tree properties are preserved.



Delete Case 3 (contd..)

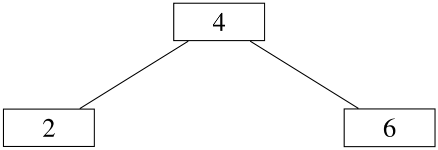
Case 3(b): Merge Required

- Key k is not present in internal node x.
- The child $x.c(i)$ has only $t - 1$ keys.
- Both immediate siblings of $x.c(i)$ also have $t - 1$ keys.
- Therefore, merging must be performed before descending.

Delete Case 3 (contd..)

Case 3(b): Before Merge

- Minimum degree $t = 2$.
- All children have only $t - 1 = 1$ key.



Delete Case 3 (contd..)

Case 3(b): Merge Step

- The parent key 4 is moved down.
- Nodes [2], 4, and [6] are merged.
- The parent node loses one key and one child.

Delete Case 3 (contd..)

Case 3(b): After Merge

- The merged node now contains $2t - 1$ keys.
- Recursive deletion of key k continues in the merged node.
- B-Tree properties are preserved.

2 4 6

Time Complexity of Deletion

Deletion Time Complexity

Deletion in a B-Tree of n keys and minimum degree t takes $O(h)$ time, where h is the height of the tree.
Since the height of a B-Tree is

$$h = O(\log_t n),$$

the overall time complexity of deletion is

$$O(\log n).$$

Applications of B-Trees

- **Database Systems:** B-Trees are widely used to implement database indexes for efficient searching, insertion, and deletion of records.
- **File Systems:** File systems use B-Trees to store and manage directory structures and metadata efficiently.
- **Disk-based Storage Systems:** B-Trees minimize disk I/O operations, making them ideal for secondary storage devices.
- **Multilevel Indexing:** B-Trees support multilevel indexing, allowing fast access to large datasets.
- **Range Queries:** Due to sorted keys, B-Trees efficiently support range-based queries.

Advantages vs Disadvantages of B-Trees

Advantages	**Disadvantages**
Always remains balanced	More complex to implement than binary search trees
Search, insertion, and deletion take $O(\log n)$ time	Insertion and deletion logic is complicated
Minimizes disk I/O operations	Requires more memory per node
Efficient for large datasets stored on disk	Not efficient for small datasets
Supports range queries efficiently	Higher constant factors compared to BSTs
Widely used in databases and file systems	Tree rebalancing increases implementation overhead

Conclusion

- B-Trees are self-balancing search trees designed for efficient access to large datasets.
- They guarantee logarithmic time complexity for search, insertion, and deletion operations.
- By keeping all leaves at the same depth, B-Trees maintain balanced structure at all times.
- Their ability to minimize disk I/O makes them ideal for databases and file systems.
- Due to these properties, B-Trees are widely used in real-world storage and indexing systems.

Thank You

Any Questions?