

UNIT 7

Trees

(5 Hours) — Data Structures and Algorithms

A tree is a non-linear hierarchical data structure that organises data in parent-child relationships rather than in a sequence. This unit covers tree terminology, binary trees and the binary search tree, searching, insertion and deletion, the three standard traversals, the notions of height, level and depth, self-balancing AVL trees and their rotation algorithm, and three important applications: Huffman coding, game trees, and B-Trees.

7.1 Introduction to Trees

Unlike arrays, linked lists, stacks and queues — which are all linear structures — a tree is a non-linear structure. It consists of a finite set of nodes connected by edges, such that there is one specially designated node called the root, and every other node is connected by exactly one path from the root. A tree with n nodes always has exactly $n - 1$ edges and contains no cycles.

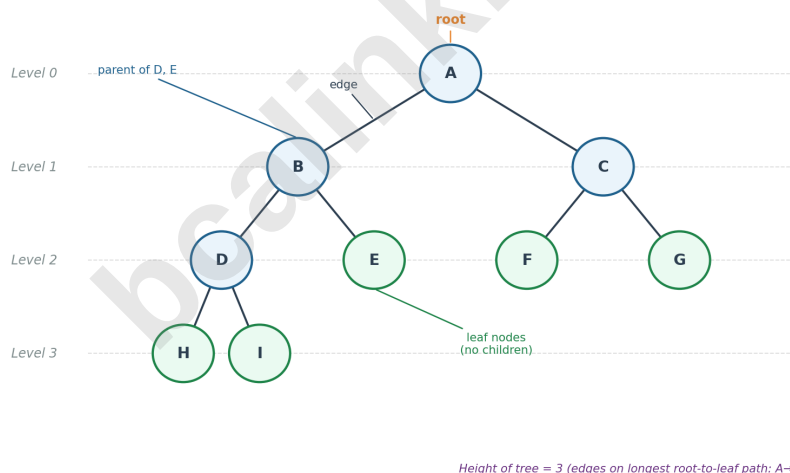


Fig 7.1 — A tree showing root, edges, parent, leaf nodes, levels and height.

7.1.1 Basic Terminology

Term	Meaning
Root	The topmost node of the tree; it has no parent
Edge	The link connecting a parent node to a child node
Parent	A node that has one or more nodes directly below it
Child	A node directly below and connected to a parent

Term	Meaning
Siblings	Nodes that share the same parent
Leaf (terminal node)	A node with no children
Internal node	A node with at least one child (a non-leaf node)
Subtree	A node together with all of its descendants
Degree of a node	The number of children the node has
Path	A sequence of nodes connected by edges from one node to another
Ancestor / Descendant	Any node on the path up to the root / any node in a node's subtree

7.1.2 Binary Tree

A binary tree is a tree in which every node has at most two children, referred to as the left child and the right child. Binary trees are by far the most widely used form of tree in computing, because the two-way branching maps naturally onto comparison-based decisions.

- Full (strict) binary tree — every node has either 0 or exactly 2 children.
- Complete binary tree — all levels are completely filled except possibly the last, which is filled from left to right.
- Perfect binary tree — all internal nodes have 2 children and all leaves are at the same level.
- Skewed binary tree — every node has only one child, making the tree degenerate into a structure resembling a linked list.

Note: A binary tree of height h can hold at most $2^{h+1} - 1$ nodes, and a tree with n nodes has a minimum possible height of $\text{floor}(\log_2 n)$. This logarithmic relationship is what makes tree-based searching efficient.

7.1.3 Binary Search Tree (BST)

A Binary Search Tree is a binary tree that maintains an ordering property: for every node, all keys in its left subtree are smaller than the node's key, and all keys in its right subtree are larger. This property must hold at every node, not just at the root.

```
struct TNode {
    int key;
    struct TNode *left, *right;
};
typedef struct TNode *TNODEPTR;
```

7.2 Basic Operations in a Binary Tree

The fundamental operations supported by a binary search tree are searching for a key, inserting a new key, deleting an existing key, and traversing the tree. All of these exploit the recursive structure of the tree.

Operation	Description	Average Time	Worst Time
search(key)	Locate the node containing the given key	$O(\log n)$	$O(n)$
insert(key)	Add a new key while preserving the BST property	$O(\log n)$	$O(n)$
delete(key)	Remove a key and restructure the tree	$O(\log n)$	$O(n)$
traverse()	Visit every node exactly once	$O(n)$	$O(n)$
findMin() / findMax()	Leftmost / rightmost node of the tree	$O(\log n)$	$O(n)$

Note: The worst case $O(n)$ occurs when the tree becomes skewed — for example when keys are inserted in already-sorted order. Balanced trees such as AVL (Section 7.5) guarantee $O(\log n)$ even in the worst case.

7.3 Tree Search, Insertion and Deletion

7.3.1 Searching in a BST

Searching starts at the root and compares the target key with the current node's key at each step. If the target is smaller, the search continues in the left subtree; if larger, in the right subtree. Each comparison discards roughly half of the remaining nodes.

```
TNODEPTR search(TNODEPTR root, int key) {
    if (root == NULL || root->key == key)
        return root; /* found, or absent */
    if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}
```

7.3.2 Insertion in a BST

Insertion follows exactly the same path a search would take. When the search reaches an empty position (NULL), that is precisely where the new node belongs, so a new node is created and linked there. A new key is therefore always inserted as a leaf.

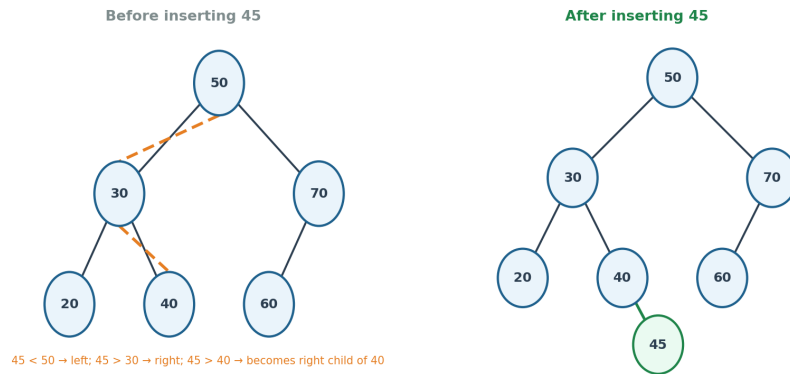


Fig 7.2 — Inserting 45: the search path $50 \rightarrow 30 \rightarrow 40$ ends at an empty right link of 40, where the new node is attached.

```

TNODEPTR insert(TNODEPTR root, int key) {
    if (root == NULL) { /* empty spot found */
        TNODEPTR temp = getnode();
        temp->key = key;
        temp->left = temp->right = NULL;
        return temp;
    }
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    return root; /* duplicates ignored */
}

```

Example 7.1 — Building a BST

Insert the keys 50, 30, 70, 20, 40, 60 into an initially empty BST, in that order.

Solution

50 becomes the root (tree was empty).

$30 < 50 \Rightarrow$ left child of 50.

$70 > 50 \Rightarrow$ right child of 50.

$20 < 50$, then $20 < 30 \Rightarrow$ left child of 30.

$40 < 50$, then $40 > 30 \Rightarrow$ right child of 30.

$60 > 50$, then $60 < 70 \Rightarrow$ left child of 70.

7.3.3 Deletion from a BST

Deletion is the most intricate BST operation because the tree must remain a valid BST afterwards. There are three distinct cases, depending on how many children the node being deleted has.

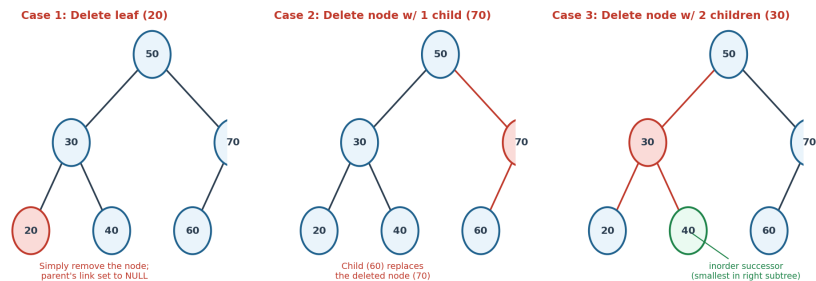


Fig 7.3 — The three cases of BST deletion: leaf node, node with one child, and node with two children.

- Case 1 — Leaf node (no children): simply remove the node and set the parent's corresponding link to NULL.
- Case 2 — One child: the node's only child is promoted to take its place; the parent's link is redirected to that child, and the node is freed.
- Case 3 — Two children: the node cannot simply be removed. Instead its key is replaced by its inorder successor (the smallest key in its right subtree) or its inorder predecessor, and then that successor node — which has at most one child — is deleted recursively using Case 1 or Case 2.

```

TNODEPTR findMin(TNODEPTR root) {
    while (root->left != NULL) root = root->left;
    return root;
}
TNODEPTR delete(TNODEPTR root, int key) {
    if (root == NULL) return NULL;
    if (key < root->key)
        root->left = delete(root->left, key);
    else if (key > root->key)
        root->right = delete(root->right, key);
    else { /* node found */
        if (root->left == NULL) { /* Case 1 or 2 */
            TNODEPTR t = root->right;
            free(root); return t;
        }
        if (root->right == NULL) {
            TNODEPTR t = root->left;
            free(root); return t;
        }
        TNODEPTR succ = findMin(root->right); /* Case 3 */
        root->key = succ->key;
        root->right = delete(root->right, succ->key);
    }
    return root;
}

```

7.4 Binary Tree Traversals

Traversal means visiting every node of a tree exactly once in a systematic order. Because a tree is non-linear, there is more than one sensible order. The three classical depth-first traversals differ only in when the root is visited relative to its subtrees.

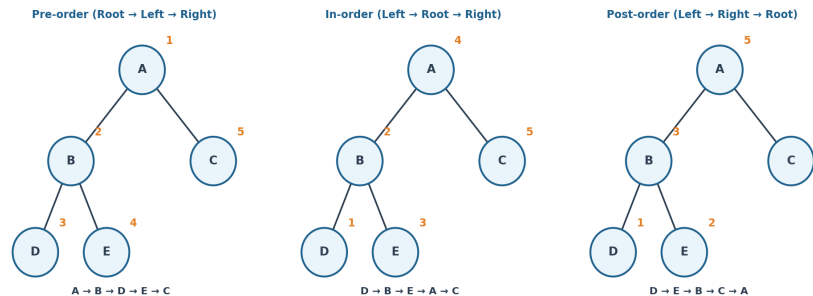


Fig 7.4 — The same binary tree traversed in pre-order, in-order and post-order; the numbers show the visiting sequence.

Traversal	Order of visiting	Output for Fig 7.4
Pre-order	Root → Left → Right	A B D E C
In-order	Left → Root → Right	D B E A C
Post-order	Left → Right → Root	D E B C A

```
void preorder(TNODEPTR root) {
    if (root == NULL) return;
    printf("%d ", root->key); /* visit root first */
    preorder(root->left);
    preorder(root->right);
}

void inorder(TNODEPTR root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->key); /* visit root in the middle */
    inorder(root->right);
}

void postorder(TNODEPTR root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->key); /* visit root last */
}
```

Note: In-order traversal of a BST always produces the keys in sorted ascending order. This is a direct consequence of the BST property and is often used to verify that a tree is a valid BST.

7.4.1 Uses of Each Traversal

- Pre-order — used to create a copy of a tree, and to produce prefix (Polish) notation from an expression tree.
- In-order — used to retrieve BST keys in sorted order, and to produce infix notation from an expression tree.
- Post-order — used to delete or free a tree safely (children are freed before their parent), and to produce postfix (Reverse Polish) notation.

Example 7.2 — Traversal of a BST

Give the in-order traversal of the BST built in Example 7.1 (keys 50, 30, 70, 20, 40, 60).

Solution

In-order visits Left → Root → Right at every node, giving:

20 30 40 50 60 70

As expected, the keys come out in sorted ascending order.

7.5 Tree Height, Level and Depth

These three closely related terms are often confused, so it is worth stating them precisely. They all describe positions and sizes within the hierarchy (refer back to Fig 7.1).

Term	Definition	Example (Fig 7.1)
Level of a node	The number of edges on the path from the root to that node; the root is at level 0	B is at level 1; H is at level 3
Depth of a node	Same as level — distance measured downward from the root	depth(D) = 2
Height of a node	The number of edges on the longest path from that node down to a leaf	height(B) = 2; height(E) = 0
Height of the tree	The height of its root node, i.e. the longest root-to-leaf path	height = 3 (A → B → D → H)
Leaf height	Always 0, since a leaf has no descendants	height(F) = 0

Note: Depth is measured downward from the root, while height is measured upward from the leaves. An empty tree is conventionally given height -1, and a single-node tree has height 0.

```
int height(TNODEPTR root) {
    if (root == NULL) return -1; /* empty tree */
    int lh = height(root->left);
    int rh = height(root->right);
    return 1 + (lh > rh ? lh : rh);
}
```

7.6 Balanced Trees: AVL Trees

A plain BST degrades badly when keys arrive in sorted order: it becomes skewed and search time falls from $O(\log n)$ to $O(n)$. A balanced tree automatically restructures itself to keep its height close to $\log n$, guaranteeing efficient operations regardless of insertion order.

The AVL tree (named after Adelson-Velsky and Landis, 1962) was the first self-balancing binary search tree. It is a BST with one extra condition:

- For every node, the heights of its left and right subtrees may differ by at most 1.

7.6.1 Balance Factor

The balance factor (BF) of a node is defined as the height of its left subtree minus the height of its right subtree:

```
BF(node) = height(node->left) - height(node->right)
```

A tree is a valid AVL tree if and only if every node has a balance factor of -1 , 0 , or $+1$. If an insertion or deletion makes any balance factor ± 2 , the tree must be rebalanced by performing rotations.

7.6.2 Balancing Algorithm — The Four Rotation Cases

When a node becomes unbalanced, exactly one of four situations applies, determined by where the offending insertion occurred relative to the unbalanced node. Two cases need a single rotation and two need a double rotation.

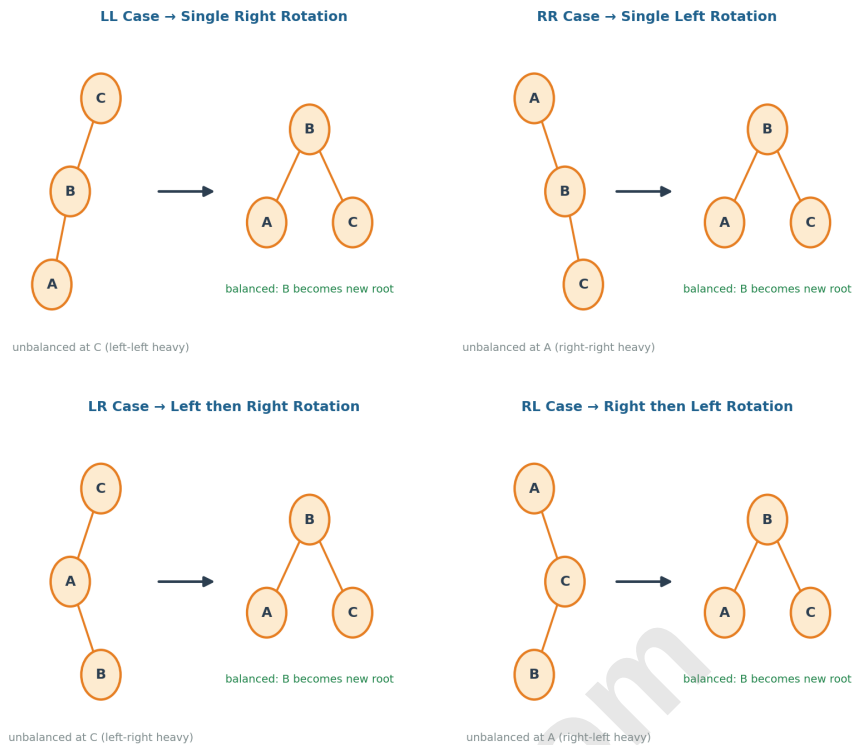


Fig 7.5 — The four AVL rebalancing cases: LL, RR, LR and RL.

Case	Condition (imbalance at node N)	Fix required
LL (Left-Left)	$BF(N) = +2$ and the insertion went into the left subtree of N's left child	Single Right Rotation about N
RR (Right-Right)	$BF(N) = -2$ and the insertion went into the right subtree of N's right child	Single Left Rotation about N
LR (Left-Right)	$BF(N) = +2$ and the insertion went into the right subtree of N's left child	Left rotation on left child, then Right rotation on N
RL (Right-Left)	$BF(N) = -2$ and the insertion went into the left subtree of N's right child	Right rotation on right child, then Left rotation on N

The balancing algorithm is applied while returning from a recursive insertion: at each node on the path back to the root, the height is updated, the balance factor is recomputed, and if it falls outside $\{-1, 0, +1\}$ the appropriate rotation is performed. Each rotation takes $O(1)$ time and only a constant number of pointers change.

```

TNODEPTR rightRotate(TNODEPTR y) {
    TNODEPTR x = y->left;
    TNODEPTR T2 = x->right;
    x->right = y; /* rotate */
    y->left = T2;
    updateHeight(y);
    updateHeight(x);
    return x; /* x is the new root */
}

TNODEPTR leftRotate(TNODEPTR x) {
    TNODEPTR y = x->right;
    TNODEPTR T2 = y->left;
    y->left = x;
    x->right = T2;
    updateHeight(x);
    updateHeight(y);
    return y;
}

```

Example 7.3 — AVL insertion causing a rotation

Insert the keys 30, 20, 10 into an empty AVL tree and rebalance.

Solution

Insert 30 → root. Insert 20 → left child of 30; $BF(30) = +1$, still valid.

Insert 10 → left child of 20. Now $BF(30) = +2$ — unbalanced.

The insertion went into the left subtree of 30's left child ⇒ LL case.

Perform a single right rotation about 30: node 20 becomes the new root, with 10 as its left child and 30 as its right child.

Resulting tree is balanced with all balance factors = 0.

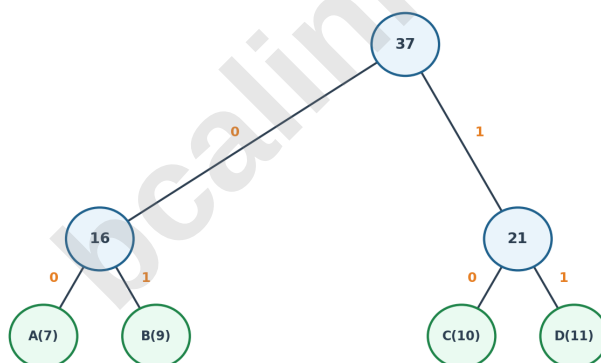
Note: Because the AVL condition keeps the height at $O(\log n)$, search, insertion and deletion are all guaranteed $O(\log n)$ in the worst case — the main advantage over an ordinary BST.

7.7 The Huffman Algorithm

Huffman coding is a lossless data-compression technique that assigns variable-length binary codes to characters based on how frequently they occur. Frequently occurring characters get short codes and rare characters get longer ones, so the total encoded length is minimised. The codes are derived from a binary tree called the Huffman tree.

7.7.1 The Algorithm

- Step 1: Create a leaf node for every character, each labelled with its frequency, and place all nodes in a min-priority queue.
- Step 2: Remove the two nodes with the smallest frequencies from the queue.
- Step 3: Create a new internal node whose frequency is the sum of those two, making them its left and right children. Insert this new node back into the queue.
- Step 4: Repeat Steps 2–3 until only one node remains — this is the root of the Huffman tree.
- Step 5: Assign 0 to every left edge and 1 to every right edge. The code for a character is the sequence of bits read along the path from the root down to that character's leaf.



Codes: A=00 (freq 7), B=01 (freq 9), C=10 (freq 10), D=11 (freq 11) — read root-to-leaf bits

Fig 7.6 — A Huffman tree built from character frequencies; codes are read from root to leaf (left = 0, right = 1).

Because every character sits at a leaf, no code is ever a prefix of another code. This prefix property is what makes the compressed bit stream unambiguously decodable without any separators between codes.

Property	Detail
Type	Greedy algorithm producing an optimal prefix code
Data structure used	Min-heap (priority queue) plus a binary tree

Property	Detail
Time complexity	$O(n \log n)$ for n distinct characters
Key guarantee	Minimum possible total encoded length for the given frequencies
Applications	ZIP and GZIP compression, JPEG, MP3, fax transmission

Example 7.4 — Building Huffman codes

Characters A, B, C, D have frequencies 7, 9, 10 and 11. Build the Huffman tree and give the codes.

Solution

Queue: A:7, B:9, C:10, D:11.

Step 1: take the two smallest, A:7 and B:9 → new node 16. Queue: C:10, D:11, 16.

Step 2: take C:10 and D:11 → new node 21. Queue: 16, 21.

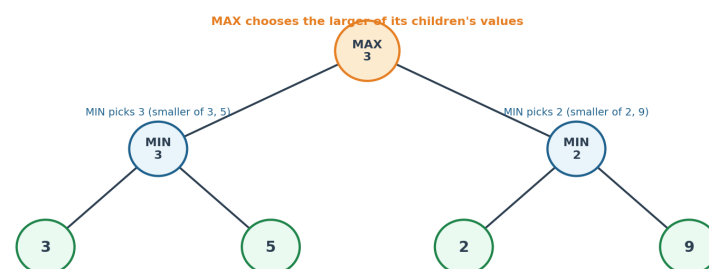
Step 3: take 16 and 21 → root node 37. Only one node remains — done.

Reading paths (left = 0, right = 1): A = 00, B = 01, C = 10, D = 11.

Here all frequencies are similar, so all codes came out 2 bits long; a skewed frequency distribution would produce codes of differing lengths.

7.8 Game Tree

A game tree is a tree that represents all possible sequences of moves in a two-player game such as tic-tac-toe, checkers or chess. The root is the current board position, each edge is a legal move, and each node is the position reached after that move. Leaf nodes are terminal or evaluated positions and carry a numeric score.



Leaf nodes are terminal board positions with a static evaluation score

Fig 7.7 — A game tree evaluated with the minimax algorithm; values propagate upward from the leaves to the root.

7.8.1 The Minimax Algorithm

Levels of a game tree alternate between the two players, and minimax evaluates the tree bottom-up on the assumption that both players play optimally:

- MAX levels represent the computer's own turn; it chooses the child with the largest value, since it wants to maximise its score.
- MIN levels represent the opponent's turn; the opponent is assumed to choose the child with the smallest value, minimising the computer's score.
- Leaf nodes are scored by a static evaluation function that estimates how favourable that board position is.
- The value computed at the root tells the computer which first move guarantees the best outcome against optimal play.

Note: Real game trees are far too large to explore completely — chess has roughly 10^{120} possible games. Practical engines therefore search only to a limited depth and use alpha-beta pruning to skip branches that cannot affect the final decision.

7.9 B-Tree

A B-Tree is a self-balancing search tree in which a node may hold many keys and have many children, rather than just one key and two children. It is designed for systems that read and write large blocks of data — above all, databases and file systems stored on disk.

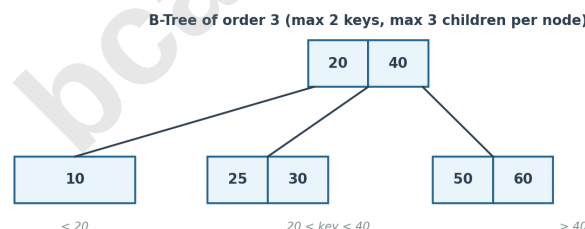


Fig 7.8 — A B-Tree; each node stores multiple keys, and all leaves lie at the same level.

7.9.1 Properties of a B-Tree of Order m

- Every node holds at most $m - 1$ keys and at most m children.
- Every node except the root holds at least $\text{ceil}(m/2) - 1$ keys, so nodes are kept at least half full.
- The keys within each node are stored in sorted order, and the child pointers between them separate the key ranges.
- All leaf nodes appear at the same level, so the tree is always perfectly height-balanced.

- The tree grows and shrinks from the root, not from the leaves: height increases only when the root splits.

Insertion places a key into the appropriate leaf; if that node overflows, it is split in two and the median key is pushed up into the parent, possibly cascading to the root. Deletion may cause a node to underflow, which is repaired by borrowing a key from a sibling or merging two nodes.

7.9.2 Why B-Trees Are Used for Disk Storage

Reading a block from disk is far slower than comparing keys in memory, so the cost of a search is dominated by the number of nodes visited, not the number of comparisons. By packing hundreds of keys into a single node sized to match a disk block, a B-Tree keeps its height extremely small — often just 3 or 4 levels for millions of records — so very few disk reads are needed.

Feature	Binary Search Tree / AVL	B-Tree
Keys per node	Exactly 1	Many (up to $m - 1$)
Children per node	At most 2	Many (up to m)
Typical height	$O(\log_2 n)$ — relatively tall	$O(\log_m n)$ — very short
Balance	AVL: via rotations	Via node splitting and merging
Designed for	Data held in main memory	Data held on disk / external storage
Main application	In-memory dictionaries and sets	Database indexes, file systems (NTFS, ext4)

Note: The B+ Tree, a common variant, stores all actual records in the leaf level and links the leaves together, making range queries and sequential scans especially efficient. Most relational database indexes are B+ Trees.

Key Takeaways

Unit 7 — Key Takeaways

- ✓ A tree is a non-linear hierarchical structure with one root and $n - 1$ edges for n nodes; a binary tree restricts each node to at most two children.
- ✓ A Binary Search Tree keeps all smaller keys in the left subtree and all larger keys in the right subtree, giving $O(\log n)$ search, insert and delete on average.
- ✓ BST insertion always adds a leaf; deletion has three cases, the hardest being a node with two children, which is replaced by its inorder successor.
- ✓ The three depth-first traversals are pre-order (Root-Left-Right), in-order (Left-Root-Right) and post-order (Left-Right-Root); in-order on a BST yields sorted keys.
- ✓ Level and depth are measured downward from the root, while height is measured upward from the leaves; the height of a tree is the height of its root.
- ✓ An AVL tree keeps every node's balance factor in $\{-1, 0, +1\}$, restoring balance with LL, RR, LR or RL rotations, guaranteeing $O(\log n)$ worst-case performance.
- ✓ Huffman coding is a greedy algorithm that builds a binary tree from character frequencies to produce an optimal prefix code for lossless compression.
- ✓ A game tree represents all possible move sequences and is evaluated with minimax, where MAX levels maximise and MIN levels minimise the propagated score.
- ✓ A B-Tree stores many keys per node and keeps all leaves at the same level, minimising disk accesses — making it the standard structure for database indexes and file systems.

— *End of Unit 7* —