ChatGPT

2-3 Trees Tutorial

Introduction to 2-3 Trees

A **2-3 tree** is a type of balanced search tree that extends the concept of binary search trees. Unlike a binary tree node which can only have one data value and two children, a 2-3 tree node can hold *more than one value* and accordingly have *more than two children*. Specifically, every internal node in a 2-3 tree is either:

- a **2-node** (holding one data element and having two children), or
- a **3-node** (holding two data elements and having three children) [1] [2].

In other words, a 2-3 tree node with one key has two child pointers, and a node with two keys has three child pointers (hence the name *2-3 tree*). All data in the node is kept in sorted order (if a node has two keys, they are stored in increasing order) [3]. Figure 1 shows a conceptual 3-node containing two keys `5` and `10`, with three children pointers:

```
    [ 5 | 10 ]
   /    |    \
 (L)  (M)    (R)
```

Figure 1: A 3-node with keys 5 and 10. All values in the left subtree (L) are `< 5`*, values in the middle subtree (M) lie between 5 and 10, and values in the right subtree (R) are* `> 10` [4].

Because nodes can have multiple children, 2-3 trees are a subset of the more general *B-trees*. In fact, a 2-3 tree is essentially a B-tree of order 3 [1] (maximum 2 keys per node and 3 children). This multi-way structure, combined with special insertion rules, keeps the tree **balanced** at all times. Being balanced means that all leaf nodes are always at the same depth (the tree grows **outward** rather than becoming skewed) [5] [6]. Thus, 2-3 trees avoid the pathological cases of plain binary search trees and guarantee efficient operations. In a 2-3 tree, **search, insertion, and deletion** all run in O(log *n*) time in the worst case [7], unlike an unbalanced BST which can degrade to O(*n*) in the worst case.

Key differences from binary trees: In summary, 2-3 trees differ from binary trees in that nodes may contain multiple keys and have multiple children, and the tree self-balances after updates. There are no "long chains" of nodes as in a skewed BST; 2-3 trees remain shallow and bushy because they automatically redistribute values among nodes to maintain balance. This comes at the cost of more complex insertion (and deletion) algorithms, which we will explore step-by-step. However, the search logic extends naturally from binary search trees, as we will see next.

2-3 Tree Properties and Structure

Let's outline the fundamental properties of a 2-3 tree:

- **Multi-way Nodes:** Each node can be a 2-node (1 key, 2 children) or a 3-node (2 keys, 3 children) [1] [2]. No other node types persist in a valid 2-3 tree (a node with three keys, sometimes called a 4-node, may appear *temporarily* during insertion or deletion, but it is immediately split and not kept in the final tree structure [8]).

- **Balanced Height:** The tree is always height-balanced. All leaf nodes are at the same level (distance from the root) [9] [6]. This ensures that the longest path from root to leaf is not much larger than the shortest path, guaranteeing logarithmic search time.

- **Ordered Keys (Search Tree Property):** The keys within each node are stored in sorted order, and they divide the node's children into ranges just like a binary search tree generalization. For a 2-node with key `a`: all keys in the left child subtree are `< a` and all keys in the right child subtree are `> a`. For a 3-node with keys `a` and `b` (where `a < b`): all keys in the left subtree are `< a`, all keys in the middle subtree are between `a` and `b`, and all keys in the right subtree are `> b` [4]. This invariant allows efficient directed search.

- **Leaf-Oriented Operations:** New keys are always added into a **leaf node** rather than creating new nodes elsewhere [10]. If a leaf is "full" (already has two keys) and another key needs to be added, the node will temporarily hold three keys just for the moment of insertion. Such an overflow triggers a **split** operation (described later) to restore the 2-3 tree properties by redistributing the keys. Similarly, deletion (which we will not cover in this basic tutorial) involves ensuring nodes do not drop below the minimum number of keys, often by merging or borrowing keys between siblings. The key idea is that *all structural adjustments happen by splitting or merging internal nodes*, rather than by unbalancing the tree.

With these properties in mind, we can proceed to see how fundamental operations work on a 2-3 tree. We begin with searching, which is conceptually straightforward, and then move on to insertion, which is more involved.

Searching in a 2-3 Tree

Searching in a 2-3 tree is similar to searching in a binary search tree, except that at each node you may have to make a three-way decision (for a 3-node) instead of a two-way decision. The search algorithm uses the keys in each node to decide which child subtree to follow, narrowing down the location of the target key.

Search algorithm description: Starting from the root, at each node compare the target key *K* to the key(s) in that node:

- If the node is a **2-node** (one key `a`):
- If `K == a`, you found the key (success).
- If `K < a`, continue search in the left child.

- If `K > a`, continue search in the right child [11].

- If the node is a **3-node** (two keys `a` and `b` with `a < b`):

- If `K` equals `a` *or* `b`, you found it.
- If `K < a`, go down to the left child.
- If `a < K < b` (i.e. between the two keys), go down to the middle child.
- If `K > b`, go down to the right child [12].

This process repeats until either the key is found in some node or you reach a leaf node where it *would* exist but isn't present (in which case the search concludes as unsuccessful).

Below is pseudocode for the search operation in a 2-3 tree:

```
function search(node, K):
    if node is NULL:
        return False  // empty tree or reached a null child

    // Check current node's keys
    if K equals node.leftKey:
        return True
    if node.hasTwoKeys() and K equals node.rightKey:
        return True

    // Determine which subtree to search next
    if K < node.leftKey:
        return search(node.leftChild, K)
    else if not node.hasTwoKeys():  // only one key in node
        // K is greater than leftKey (since not returned earlier), go right
        return search(node.rightChild, K)
    else if K < node.rightKey:
        // K is between leftKey and rightKey
        return search(node.middleChild, K)
    else:
        // K is greater than rightKey
        return search(node.rightChild, K)
```

This is essentially an extension of binary search tree logic. The additional comparisons handle the middle range when a node has two keys [13]. Each comparison directs the search to one of the node's children. Thanks to the 2-3 tree's balanced structure, the height of the tree is O(log *n*), so this recursive (or iterative) search visits O(log *n*) nodes in the worst case.

Search example: Suppose we have the 2-3 tree shown in Figure 2 (which will be built in the insertion example later), and we want to search for the key `15`:

Figure 2: A sample 2-3 tree after several insertions (keys 1,5,7,10,15,20,25).

1. Start at the root `[10]` (a 2-node). The root's key is 10. Since 15 > 10, we take the right branch to the child node `[20]`.
2. At node `[20]` (also a 2-node with key 20), we compare 15 to 20. Now 15 < 20, so we follow the left branch to the left child.
3. The left child of `[20]` is the leaf node `[15]`. We examine it and find the key 15, completing the search successfully.

If instead we were searching for a key not in the tree (say 17), the search would follow the same path and end up at the leaf `[15]`. Upon discovering that 17 is not present in this leaf node, the algorithm would conclude that 17 is not in the tree (since we've reached a leaf with no further children to explore).

Insertion in a 2-3 Tree

Insertion is the most interesting operation on 2-3 trees because we must carefully preserve the tree's balanced structure and 2-3 node properties. The goal is to always insert the new key into a leaf and, if any node ends up with too many keys, **split** that node and propagate any extra key upwards. This might cause cascading splits up to the root, but the result will be a valid 2-3 tree.

High-level idea: To insert a new key, we first perform a search to find the correct leaf where the key should go (just as in a BST, the new key's sorted position will be at some leaf). Then:

- If that leaf currently has only one key (it's a 2-node), we can safely insert the new key into that node, making it a 3-node. **No further adjustments** are needed, since we haven't violated any properties – the leaf now has two keys, which is allowed [14].

- If the target leaf is already a 3-node (has two keys), then adding the new key will give it three keys (a temporary *4-node*). A node with three keys violates the 2-3 tree rules, so we must **split** this node into two nodes and promote one of the keys to the parent:

- Take the three keys (the two existing ones and the new key) and sort them. Let's call them *leftKey*, *midKey*, *rightKey* in sorted order.
- The node is split into two nodes: the "left" node retains *leftKey*, and a new "right" node is created containing *rightKey*. The middle key (*midKey*) is **promoted** to the parent node above [15].

- The parent will receive *midKey* as an extra data element and a new child pointer to the newly created right node. Essentially, the 4-node at the leaf is resolved by distributing its keys between two 2-nodes and pushing the middle key up one level [16].

- After promoting the middle key to the parent, the parent might now become a 3-node or 4-node itself. If the parent was a 2-node, it simply becomes a 3-node with the added key and extra child (no further split needed) [17]. But if the parent was already a 3-node (two keys), then adding one more makes it a 4-node as well, requiring *another split*. In that case, we repeat the same splitting process on the parent: split it into two nodes and promote its middle key up to the next level [18]. This splitting and promotion can cascade upward recursively. In the worst case, it reaches the root: if the root splits (from a 3-node to a temporary 4-node), the middle key is promoted and becomes a new root, increasing the tree's height by one [19]. This is the *only* way a 2-3 tree grows in height (it grows upward, not by adding leaf levels arbitrarily).

- Throughout insertion, the tree maintains the search order property and balance. Splitting ensures that no node ends up with more than 2 keys, and promoting keys upward ensures the leaf levels remain at equal depth.

To clarify these rules, let's look at pseudocode for insertion. This pseudocode outlines a common approach: recursively insert the key into the appropriate leaf, and use the function's return value to handle splits (the recursive approach is one way to implement the upward propagation of splits):

```
function insert(node, K):
    // Insert key K into the subtree rooted at 'node'.
    if node is null:
        // Tree is empty, create a new leaf node
        return new Node(K)  // Node with K as a single key

    if node is a leaf:
        // Insert K into this leaf node
        insertKeyIntoNode(node, K)  // (maintaining sorted order of keys)
        if node.keyCount <= 2:
            return node         // no split needed, return the node itself
        else:
            // Node is overfull (3 keys) – split into two nodes
            let leftKey  = node.keys[0]    // smallest
            let midKey   = node.keys[1]    // middle
            let rightKey = node.keys[2]    // largest
            // Create a new node for the rightKey
            let newNode = new Node(rightKey)
            // Reduce current node to a 1-key node (leftKey)
            node.keys = [leftKey]
            // Return a special structure indicating a split:
            // midKey should be inserted into the parent, and newNode as right
child
            return (midKey, newNode)
    else:
        // Internal node: descend to correct child
        if K < node.leftKey:
            result = insert(node.leftChild, K)
```

```
            childIndexForResult = "left"
        else if (node.isTwoNode() or K < node.rightKey):
            // K goes to middle child in a 3-node, or the only child in a 2-node
            result = insert(node.middleChild, K)
            childIndexForResult = "middle"
        else:
            // K >= rightKey
            result = insert(node.rightChild, K)
            childIndexForResult = "right"

        // After recursive insert, check if a split happened in the child
        if result is not a split:
            return node   // no split from child, tree is unchanged at this
level
        else:
            // result is a tuple (midKey, newNode)
            (promotedKey, newNode) = result
            // Insert the promoted key into the current node (parent) with
newNode as a child
            insertKeyIntoNode(node, promotedKey, position=childIndexForResult)
            attachChild(node, newNode, position=childIndexForResult + "Right")
            if node.keyCount <= 2:
                return node    // parent had room, absorbed promotedKey, done
            else:
                // Current node overflow, split this node as well
                let leftKey  = node.keys[0]
                let midKey   = node.keys[1]
                let rightKey = node.keys[2]
                // Split into left node and right node around midKey
                let leftNode = new Node(leftKey)
                let rightNode = new Node(rightKey)
                // Distribute children: node.leftChild, node.middleChild ->
leftNode;
                //                     node.middleChildRight, node.rightChild ->
rightNode
                // (child pointers adjusted accordingly)
                return (midKey, rightNode)  // promote midKey upward, with
rightNode as new child
```

Note: The pseudocode above is written in a hybrid style for clarity rather than strict syntax. The function `insertKeyIntoNode` handles inserting a key into a node's key list in sorted order, and `attachChild` inserts a child pointer in the correct position. The return value of `insert` can be either a normal node (when no split occurred in the recursive call) or a special tuple `(promotedKey, newNode)` indicating that the child node split and yielded a `promotedKey` that needs to be inserted into the current node along with a new right-sibling child `newNode`. The logic then tries to insert `promotedKey` into the current node and splits

again if needed, potentially propagating further up. This approach ensures that splits are handled **bottom-up** after reaching the insertion leaf [20] [21].

In simpler terms, the insertion algorithm can be summarized as follows:

1. **Find the Leaf:** Navigate down from the root to find the leaf node where the new key belongs (this is just a search to find the correct position).
2. **Insert into Leaf:** If the leaf has only one key, insert the new key into the leaf (it will now have two keys, which is fine) [14].
3. **Handle Leaf Overflow:** If the leaf already had two keys (full):
4. Split this leaf into two nodes, each taking one of the two smallest/largest keys, and promote the middle key to the parent node [20].
5. **Insert into Parent:** Insert the promoted middle key into the parent node in the appropriate position. The parent will now have an additional child (the new node from the split).
6. **Handle Parent Overflow:** If the parent now has three keys (overflow):
7. Split the parent as well, in the same manner, and promote its middle key up to *its* parent.
8. **Repeat Upward:** Continue this split-and-promote process moving up the tree until no node is overfull. In practice, this may reach the root. If the root splits, create a new root containing the promoted key, and the old root is split into two children of the new root [19].
9. **Done:** The tree remains balanced and valid. All leaves are still at the same level, possibly one level deeper if the root was split (height increases by 1 in that case, which is a rare event).

Crucially, notice that unlike a binary search tree insertion which always creates a new leaf node, a 2-3 tree insertion **adds the new key into an existing leaf node** (or nodes, in case of splits). The tree grows upward when necessary, not downward [22]. This strategy is what keeps the tree balanced at all times.

Insertion Example (Step-by-Step)

Let's walk through a concrete example of building a 2-3 tree by inserting a sequence of keys. We will insert the keys in the following order: **5, 10, 15, 20, 25, 1, 7**. For each insertion, we'll show the state of the tree and explain any splits that occur. (Empty boxes like `[ ]` denote nodes, and `[a | b]` denotes a node with two keys `a` and `b`.)

- **Start with an empty tree:** (no nodes yet)

- **Insert 5:** The tree is empty, so the new key `5` becomes the key of a new root node.
  Tree after inserting 5:

  ```
  [5]
  ```

  Explanation: We created a leaf node containing `5`. It is also the root. (Root is a 2-node with one key.)

- **Insert 10:** We start at the root `[5]`. Since `10 > 5` and the root has no right child yet, we insert `10` into the root node (which is a leaf). Now the root node has two keys `5` and `10`.
  Tree after inserting 10:

```
[5 | 10]
```

Explanation: The root was a 2-node and had space for a second key, so we simply added `10` into it in sorted order. No splitting was needed. The root is now a 3-node containing two keys (5 and 10).

- **Insert 15:** We start at the root. The root is `[5 | 10]` (a leaf with two keys). To insert `15`, we would normally place it in this node, but the node is **full** (already has two keys). This triggers a split. We take the keys {5, 10, 15} (including the new key) and find the middle key. Sorted, they are {5, 10, 15}, so the middle key is **10**. Key 10 will be promoted up. The node will split into two leaf nodes: one holding the smaller key **5** and the other holding the larger key **15**. The promoted key 10 becomes the key of a new root node.
  Tree after inserting 15 (after splitting):

```
   [10]         <-- new root
  /    \
[5]    [15]     <-- two leaves after split
```

Explanation: The old root `[5 | 10]` split because it overflowed. We created a new root `[10]`. The old root's keys were split into two nodes: left child `[5]` and right child `[15]`. The middle key 10 rose to become the root key. Now the tree has increased in height by 1 (the tree height was 1, now it's 2). All leaves (5 and 15) are at the same depth. This demonstrates how a 2-3 tree grows upward when the root splits [19].

- **Insert 20:** Start at root `[10]`. Compare `20` with `10`: `20 > 10`, so we go to the right child. The right child is `[15]` (a leaf with one key 15). Since `20 > 15`, it belongs in this node. The node `[15]` has space (only one key), so we insert `20` here without any split.
  Tree after inserting 20:

```
   [10]
  /    \
[5]    [15 | 20]
```

Explanation: The root remains `[10]`. The left leaf is `[5]`. The right leaf, which was `[15]`, now has two keys and is `[15 | 20]`. No structural changes (splits) were needed because we inserted into a leaf that had room. Leaves are still all at depth 1 from the root.

- **Insert 25:** Start at root `[10]`. `25 > 10`, so go right. At the right child `[15 | 20]`, we find a leaf with two keys (15 and 20) – it's full. We need to insert `25` here, causing an overflow. We take {15, 20, 25} and determine the middle key: sorted, they are {15, 20, 25}, so the middle is **20**. We split node `[15 | 20]` into two nodes `[15]` and `[25]`, and promote the middle key **20** up to the parent. The parent of this leaf is the root `[10]`. The root currently has one key (10) and will accommodate the promoted key 20. After inserting 20 into the root, the root becomes a 3-node `[10 | 20]` with three children.

Tree after inserting 25:

```
     [10 | 20]        <-- root has two keys now
    /    |     \
 [5]   [15]   [25]    <-- leaves
```

Explanation: The right leaf split into two leaves `[15]` and `[25]`, and 20 moved up into the root. The root was a 2-node and had room for another key, so no split was needed at the root level [17]. The root is now a 3-node containing `[10 | 20]`. It has three children: left child `[5]`, middle child `[15]`, and right child `[25]`. All leaves ([5], [15], [25]) are at the same depth (one level below the root).

- **Insert 1:** Start at root `[10 | 20]`. Compare `1` with the root's keys: `1 < 10` (it's less than the first key 10), so we go to the **left child**. The left child is `[5]` (a leaf with one key 5). Insert `1` into this leaf. `[5]` has space for a second key, so after insertion it becomes `[1 | 5]`.
  Tree after inserting 1:
  ```
  text
        [10 | 20]
       /    |     \
  [1 | 5]  [15]   [25]
  ```
  Explanation: The leftmost leaf now has two keys (1 and 5). No splitting was required because it had room. The overall tree structure hasn't changed otherwise.

- **Insert 7:** Finally, insert key `7`. Start at the root `[10 | 20]`. We compare `7` to 10 and 20: `7 < 10`, so we again go to the left child. The left child is currently `[1 | 5]` (a leaf with two keys, 1 and 5). Now, `7` is greater than 5, so logically it would be placed after 5 in this node – but the node is full. We must split this leaf. Take keys {1, 5, 7}, sorted they are {1, 5, 7}, with middle key **5**. Split `[1 | 5]` into two nodes: left node `[1]` and right node `[7]`, and promote the middle key 5 to the parent (which is the root). The root `[10 | 20]` now tries to absorb the key 5. But the root currently has two keys (10 and 20) – it's a 3-node and cannot have a third key directly. So inserting 5 into the root causes the **root to overflow** into a temporary 4-node `[5 | 10 | 20]`. We must split the root as well. Take {5, 10, 20}, the middle key is **10**. The root splits into two nodes: left internal node `[5]` and right internal node `[20]`, and key 10 is promoted up. However, since the root was splitting, promoting 10 actually creates a **new root** `[10]`. The new root `[10]` will have two children: the left child is `[5]` and the right child is `[20]`. We also need to update their subtrees: after the leaf split, we had leaf `[1]` and leaf `[7]` under what was the left part, and leaves `[15]` and `[25]` under the right part. Organizing the children:
  - The new left internal node `[5]` becomes the parent of the two leftmost leaves `[1]` and `[7]`.
  - The new right internal node `[20]` becomes the parent of the two rightmost leaves `[15]` and `[25]`.

    Tree after inserting 7 (final tree):

```
       [10]          <-- new root
      /    \
    [5]    [20]      <-- level 1 internal nodes
   /   \   /   \
 [1]  [7] [15] [25]  <-- leaves at level 2
```

Explanation: This final insertion was the most involved. First, the leaf `[1 | 5]` split into `[1]` and `[7]`, promoting 5. The root overflowed when 5 was added, causing the root to split as well. Key 10 became the new root key. The tree's height increased by 1 (from 2 to 3 levels total). All leaves (`[1]`, `[7]`, `[15]`, `[25]`) are now at depth 2, and all internal nodes are either 2-nodes or 3-nodes. The tree remains balanced and valid. We have successfully maintained the 2-3 tree properties throughout the insertions.

This step-by-step example demonstrates how 2-3 tree insertions ensure the tree remains balanced by splitting nodes that overflow. Every insertion ended with a valid 2-3 tree, and at no point did we allow a node to permanently have 3 keys or a leaf to be at a different depth than others. We also see that *insertion in a 2-3 tree can cause multiple splits*, but these splits propagate upward and happen in a controlled way.

Conclusion

In this tutorial, we introduced 2-3 trees and walked through their basic operations. A 2-3 tree is a balanced multi-way search tree where each internal node can have 2 or 3 children (and 1 or 2 keys). We saw that search in a 2-3 tree is a straightforward generalization of binary search tree search, guided by one or two comparisons at each node. The insertion algorithm is more involved: new keys are always inserted at the leaves, and any node that temporarily gets three keys is split, with the middle key moving up to maintain the 2-3 tree invariants. This guarantees the tree stays balanced and bushy, giving O(log *n*) performance for operations.

Note: We limited our discussion to insertion (and search). *Deletion* in 2-3 trees is also possible and follows a symmetric logic (ensuring nodes don't underflow by merging or redistributing keys), but it is more complex and was omitted here to focus on core concepts. In practice, understanding 2-3 trees provides a foundation for more advanced balanced trees. For instance, 2-3 trees are closely related to **B-trees of order 3** and share DNA with **2-3-4 trees** and **red-black trees** in how they maintain balance. However, even on their own, 2-3 trees offer a clean, if somewhat intricate, example of a self-balancing search tree that guarantees efficient operations through careful management of node capacities [7] . By working through the example and pseudocode, you should now have a solid understanding of how 2-3 trees function and how they differ from binary trees in managing data for optimal search performance.

---

[1] [5] [8] [11] [12] [14] [17] [18] [19] 2–3 tree - Wikipedia
https://en.wikipedia.org/wiki/2%E2%80%933_tree

[2] [3] [6] [7] [10] [13] 2-3 Trees | (Search, Insert and Deletion) - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/2-3-trees-search-and-insert/

[4] [9] [15] [16] [20] [21] [22] 12.5. 2-3 Trees — CS3 Data Structures & Algorithms
https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/TwoThreeTree.html