# Chapter 3: Trees

In computer science, a tree is a fundamental data structure that represents hierarchical relationships among data elements. In this chapter, we will discuss several types of trees, including the basic tree structure, the Binary Search Tree (BST), the K-Dimensional Tree (K-D Tree), the AVL Tree, the Red-Black Tree, the B-Tree, and the B+ Tree.

## Introduction to Tree

In data structures, a **tree** is a hierarchical structure that represents a collection of connected nodes, each of which can have zero or more child nodes. Refer to Fig. 1 for an example of a tree. There is no specific policy for constructing a tree. Nodes are added to the tree as required by the hierarchical structure.

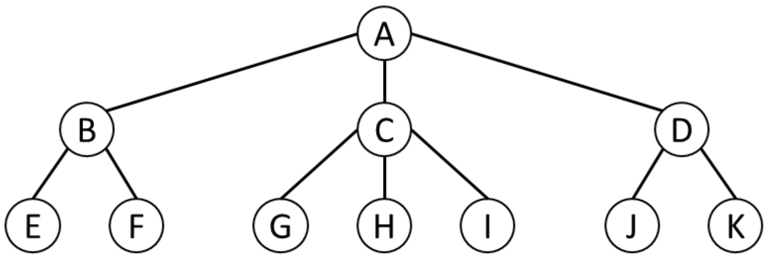
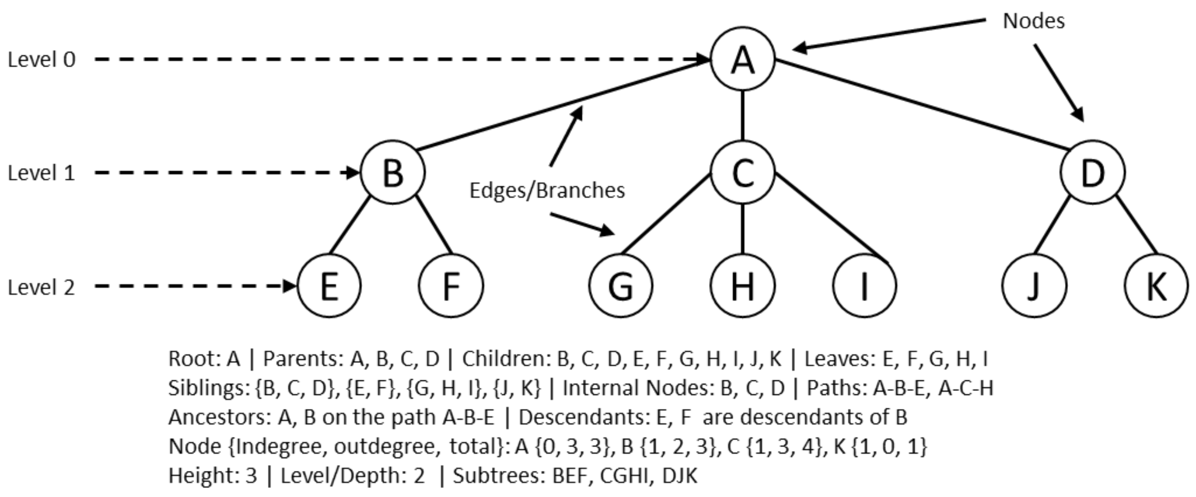


Fig. 1: A tree.

## Introduction to Tree Terms

- **Nodes**: Each item in a tree is a node. A node contains a value or data and may link to other nodes.
- **Edges** or **Branches**: Branches are connections between nodes that represent a parent-child relationship.
- **Degree**: The degree of a node is the total number of edges connected to it.
- **Indegree**: Indegree refers to the number of edges coming *into* a node.
- **Outdegree**: Outdegree is the number of edges going *out* from a node.
- **Root** or **Root Node**: The topmost node in the tree. It has no parent.
- **Leaf** or **Leaf Node**: Node without children. It represents the endpoint of a branch in the tree.

- **Internal Node**: A node with at least one child; it is not a leaf node.
- **Parent**: A node with an outdegree greater than zero.
- **Child**: A node with an indegree of one.
- **Siblings**: Nodes that share the same parent node.
- **Ancestor**: A node that lies on the path from the root to a given node, including the root and any other nodes leading up to that node.
- **Descendant**: A node that can be reached by moving downwards from a given node, including its children, children's children, and so on.
- **Path**: A sequence of nodes and edges that connect a starting node to an end node. In trees, this path is unique for each node due to the hierarchical structure.
- **Level**: The depth or distance of a node from the root. The root is at level 0, its children are at level 1, and so on.
- **Height**: The length of the longest path from a node to a leaf node. The height of a tree is the height of the root node. By definition, the height of an empty tree is -1.
- **Depth**: The distance from the root node to a specific node. It is effectively the same as the level of the node.
- **Subtree**: A smaller tree structure within a larger tree, consisting of a node and all its descendants. Any node in a tree, along with its children, can form a subtree.



Root: A | Parents: A, B, C, D | Children: B, C, D, E, F, G, H, I, J, K | Leaves: E, F, G, H, I
Siblings: {B, C, D}, {E, F}, {G, H, I}, {J, K} | Internal Nodes: B, C, D | Paths: A-B-E, A-C-H
Ancestors: A, B on the path A-B-E | Descendants: E, F are descendants of B
Node {Indegree, outdegree, total}: A {0, 3, 3}, B {1, 2, 3}, C {1, 3, 4}, K {1, 0, 1}
Height: 3 | Level/Depth: 2 | Subtrees: BEF, CGHI, DJK

Fig. 2: Basic tree concepts.

- **Binary Tree:** A binary tree is a type of tree in which each node has at most two child nodes, often referred to as the left and right children.
- **Strict Binary Tree**: Every node has either 0 or 2 children.
- **Full Binary Tree:** A full binary tree is essentially a strict binary tree in which all leaf nodes are at the same level.
- **Complete Binary Tree**: All levels are fully filled, except possibly the last, and nodes are as far left as possible.

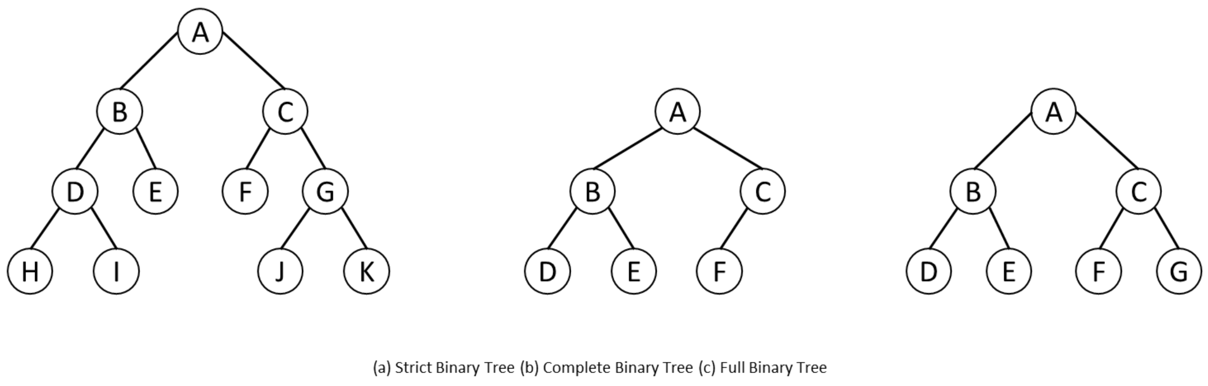

A
B
C
D
E
F
G
H
I
J
K
(a) Strict Binary Tree (b) Complete Binary Tree (c) Full Binary Tree

Fig. 3: Binary trees.

- **Binary Search Tree (BST)**: A binary tree with an ordering property—left children contain values less than the parent, and right children contain values greater than or equal to the parent.

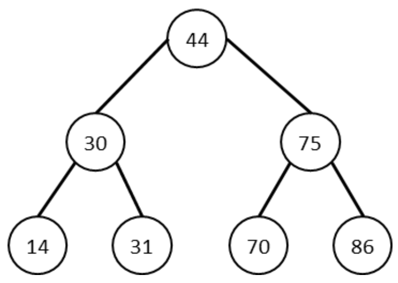


Fig. 4: Binary Search Tree.

## BST Operations

### Add Node to BST

The algorithm iteratively finds the correct position for a new node in a Binary Search Tree (BST) by traversing from the root node down through the tree. Starting at the root, the algorithm compares the new node's value with each node's value, moving left if the new value is smaller and right if it's larger. This traversal continues until it finds an empty position where the new node can be added as either a left or right child, maintaining the BST property where left children are smaller and right children are larger than their parent nodes. This iterative approach allows for efficient insertion without the need for recursion, making it suitable for trees of any height.

**Algorithm for adding a node to a BST:**

1. **Start at the Root**: Begin at the root node of the BST.
2. **Compare Node Values**:
   - If the tree is empty, create a new node and set it as the root.
   - Otherwise, compare the value to be added with the value of the current node.
3. **Navigate the Tree**:
   - If the value to be added is less than the current node's value, move to the left child.
   - If the value to be added is greater than or equal to the current node's value, move to the right child.
4. **Insert the Node**:
   - If the appropriate child (left or right) is `null`, insert the new node in that position.
   - If the appropriate child is not `null`, repeat the comparison and navigation steps (steps 2 and 3) with the child node as the current node.
5. **End of Algorithm**: The node is added when an appropriate position is found.

## Delete Node from BST

To delete a node from a Binary Search Tree (BST), we need to consider three possible cases:

1. **The node to be deleted has no children (leaf node)**: Simply remove the node.
2. **The node to be deleted has one child**: Remove the node and link its parent to its only child.

3. **The node to be deleted has two children**: Find the in-order successor (or in-order predecessor), replace the node’s value with the successor’s value, and then delete the successor (which will now be a simpler case of having at most one child).

**Algorithm for deleting a node from a BST**

1. **Find the Node**: Start from the root and search for the node to delete, comparing the value with the current node’s value.
2. **Check for 0 or 1 Child**:
    - If the node has no children (leaf node), remove it by setting its parent's pointer to `null`.
    - If the node has one child, replace it with its child (either left or right).
3. **Check for 2 Children**:
    - Find the in-order successor (the smallest node in the right subtree) or in-order predecessor (the largest node in the left subtree).
    - Replace the value of the node to be deleted with the in-order successor or predecessor.
    - Recursively delete the in-order successor or predecessor (which will be a case with at most one child).
4. **End the operation**: Once the node is deleted, the tree structure is updated.

## Search a value in a BST

To search for a value in a Binary Search Tree (BST), we take advantage of the BST property where for any node:

- The left subtree contains only nodes with values **less than** the node’s value.
- The right subtree contains only nodes with values **greater than or equal to** the node’s value.

**Algorithm for searching a value in a BST:**

1. **Start at the Root**: Begin with the root node of the BST.
2. **Compare the Value**:
    - If the target value is equal to the current node’s value, return the node (found).

   - If the target value is less than the current node's value, search in the left subtree.
   - If the target value is greater than the current node's value, search in the right subtree.
3. **Recursive Search**: Continue this comparison and navigate through the tree until the target value is found or a leaf node is reached.
4. **End Condition**: If the current node is `null`, it means the value is not present in the tree.

**Time Complexity:**

- **Best-case**: O(log n) when the tree is balanced.
- **Worst-case**: O(n) when the tree is skewed (like a linked list).

## Find the Smallest Node in a BST

To find the **smallest node** in a Binary Search Tree (BST), we can use the property of the BST: In a BST, the smallest value is always the leftmost node. So, we keep traversing the left child until we reach the leftmost node.

**Algorithm to find the smallest node in a BST:**

1. **Start at the Root**: Begin at the root node of the BST.
2. **Traverse Left**: Move to the left child of the current node.
   - Repeat this step until the left child is `null`.
3. **Return the Leftmost Node**: Once the leftmost node is reached, return that node as it contains the smallest value.
4. **End the Operation**: The traversal ends when the leftmost node is found.

**Time Complexity:**

- **Best-case**: O(1) if the root itself is the smallest node.
- **Worst-case**: O(h), where `h` is the height of the tree (the maximum number of left children). In a skewed tree, this can be O(n). In a balanced tree, it is O(log n).

## Find the Largest Node in a BST

To find the **largest node** in a Binary Search Tree (BST), we can utilize the BST property where the largest value is always the rightmost node. So, we keep traversing the right child until we reach the rightmost node.

**Algorithm to find the largest node in a BST:**

1. **Start at the Root**: Begin at the root node of the BST.
2. **Traverse Right**: Move to the right child of the current node.
   - Repeat this step until the right child is `null`.
3. **Return the Rightmost Node**: Once the rightmost node is reached, return that node as it contains the largest value.
4. **End the Operation**: The traversal ends when the rightmost node is found.

**Time Complexity:**

- **Best-case**: O(1) if the root itself is the largest node.
- **Worst-case**: O(h), where `h` is the height of the tree (the maximum number of right children). In a skewed tree, this can be O(n). In a balanced tree, it is O(log n).

## Pre-order Traversal

In **pre-order traversal**, the nodes are visited in the following order:

1. Visit the current node (root).
2. Recursively perform a pre-order traversal on the left subtree.
3. Recursively perform a pre-order traversal on the right subtree.

This traversal is often used for copying a tree or creating a deep copy of the tree structure.

**Algorithm for pre-order traversal:**

1. **Start at the Root**: Begin with the root node of the BST.
2. **Visit the Current Node**: Process the current node (i.e., print its value, store it, etc.).
3. **Traverse Left Subtree**: Recursively perform pre-order traversal on the left subtree.
4. **Traverse Right Subtree**: Recursively perform pre-order traversal on the right subtree.
5. **End Condition**: The traversal ends when all nodes are visited (i.e., the recursive calls reach `null`).

**Time Complexity:**

- **Best-case and Worst-case**: O(n), where `n` is the number of nodes in the tree. This is because every node is visited exactly once.

## In-order Traversal

In **in-order traversal**, the nodes are visited in the following order:

1. Recursively traverse the left subtree.
2. Visit the current node (root).
3. Recursively traverse the right subtree.

In the case of a Binary Search Tree (BST), the **in-order traversal** produces the nodes in **ascending order** because for every node:

- All nodes in the left subtree have smaller values.
- All nodes in the right subtree have larger or equal values.

**Algorithm for in-order traversal:**

1. **Start at the Root**: Begin with the root node of the BST.
2. **Traverse Left Subtree**: Recursively perform in-order traversal on the left subtree.
3. **Visit the Current Node**: Process the current node (i.e., print its value, store it, etc.).
4. **Traverse Right Subtree**: Recursively perform in-order traversal on the right subtree.
5. **End Condition**: The traversal ends when all nodes are visited (i.e., the recursive calls reach `null`).

**Time Complexity:**

- **Best-case and Worst-case**: O(n), where `n` is the number of nodes in the tree. Every node is visited exactly once during the traversal.

## Post-order Traversal

In **post-order traversal**, the nodes are visited in the following order:

1. Recursively traverse the left subtree.
2. Recursively traverse the right subtree.

3. Visit the current node (root).

This traversal is useful when we need to process the nodes after visiting both subtrees, such as in deletion or freeing memory for the tree.

**Algorithm for post-order traversal:**

1. **Start at the Root**: Begin with the root node of the BST.
2. **Traverse Left Subtree**: Recursively perform post-order traversal on the left subtree.
3. **Traverse Right Subtree**: Recursively perform post-order traversal on the right subtree.
4. **Visit the Current Node**: Process the current node (i.e., print its value, store it, etc.).
5. **End Condition**: The traversal ends when all nodes are visited (i.e., the recursive calls reach `null`).

**Time Complexity:**

- **Best-case and Worst-case**: O(n), where `n` is the number of nodes in the tree. Every node is visited exactly once during the traversal.

## Balanced vs Unbalanced Tree

Fig. 5 shows two tree structures created using the values 44, 30, and 14. The tree in Fig. 5(a) is known as an unbalanced tree, as all nodes are heavily leaning to the left. The same values can be rearranged to form a complete tree, resulting in a balanced structure. Fig. 5(b) illustrates a balanced tree created from these same values.

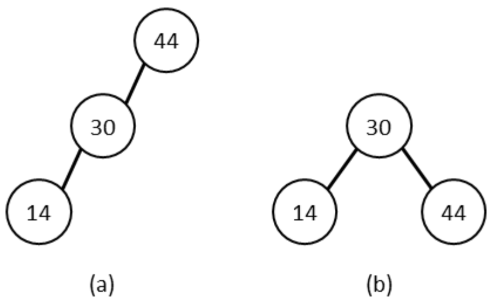


Fig. 5: (a) Unbalanced tree (b) Balanced tree

The time complexity for common operations, such as adding a node, deleting a node, and searching for a value in a balanced tree, is O($logn$). However, in an unbalanced tree, this time complexity is likely to degrade to O($n$), as operations may require traversing most or all nodes in a single path.  This implies that balancing a tree is essential to ensure optimal performance of common operations.

## When is a Tree considered balanced?

A tree is considered balanced when the heights of its subtrees differ by no more than one level.

## AVL Trees

An **AVL tree** is a type of self-balancing binary search tree (BST) named after its inventors, Adelson-Velsky and Landis. In an AVL tree, the height difference (or balance factor) between the left and right subtrees of any node is restricted to at most one. This balance is maintained after every insertion or deletion operation, ensuring that the tree remains balanced.

- **Balance Factor**: For any node in an AVL tree, the balance factor (the difference between the heights of the left and right subtrees) is -1, 0, or +1. This constraint keeps the tree balanced, maintaining a height of O($logn$).

  The **balance factor** of a node in an AVL tree is calculated by taking the difference between the heights of its left and right subtrees:

  Balance Factor = Height of Left Subtree − Height of Right Subtree

- **Rotations**: To maintain balance after an insertion or deletion, the tree may perform one or more rotations. These rotations are categorized as:
  - **Single Right Rotation (LL Rotation)**
  - **Single Left Rotation (RR Rotation)**
  - **Left-Right Rotation (LR Rotation)**
  - **Right-Left Rotation (Right of Left or RL Rotation)**

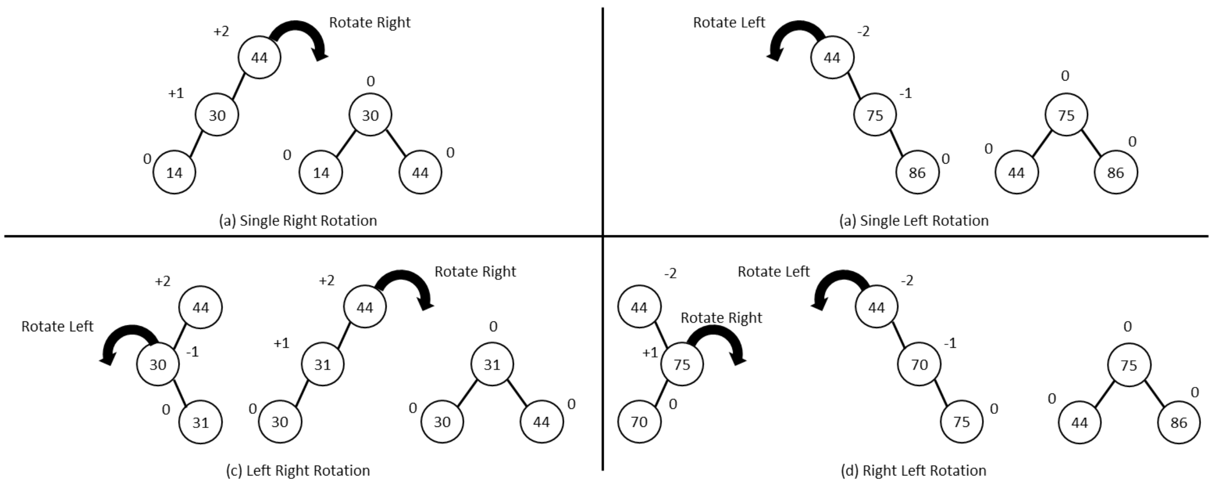


Fig. 6: Balancing Trees.

Use the link AVL Tree Simulator to access the simulator and experiment with the provided values to observe rotations and the construction of an AVL tree.

**Single Right Rotation:** 18 20 12 14 8 5

**Single Left Rotation:** 14 20 12 18 23 44

**Left-Right Rotation:** 23 18 44 20 52 12 14 4 16

**Right-Left Rotation:** 18 12 44 52 23 20

### Pros and Cons of AVL Trees

While the balanced structure of the AVL tree makes search operations efficient, the complex rotations can slow down insertion operations.

## Red and Black Trees

Red-Black Trees perform faster for insertion and deletion operations because they involve fewer rotations than AVL Trees.

### Properties:

1. **Node Colour**: Every node is either red or black.
2. **Root**: The root node is always black.
3. **Red Rule**: Red nodes cannot have red children (no two consecutive red nodes).
4. **Black Depth**: Every path from a node to its descendant null nodes has the same number of black nodes.

5. **Leaf Nodes**: Null leaf nodes (null pointers) are considered black.

## Insertion Algorithm for Red-Black Tree

1. **Insert the Node**:
    - Insert the new node as you would in a standard Binary Search Tree (BST).
    - Colour the new node red.
2. **Check for Violations**:
    - If the new node is the **root**, recolour it to black.
    - If the parent of the new node is black, the tree remains balanced (no further action is needed).
    - If the parent is red, there is a violation of the Red Rule (two consecutive red nodes).
3. **Fix Violations**:
    - **Case 1: The Red Uncle Case**:
        - If the uncle of the new node is red, recolour the parent and uncle to black and the grandparent to red.
        - Set the grandparent as the new current node and repeat the check for violations.
    - **Case 2: The Black Uncle with Triangle (LR/RL) Case**:
        - If the uncle is black and the new node forms a "triangle" with its parent (i.e., left-right or right-left configuration), perform a rotation to make it a line.
        - Rotate the parent in the opposite direction of the "triangle." (For example, if left-right, perform a left rotation.)
        - After rotation, proceed to Case 3.
    - **Case 3: The Black Uncle with Line (LL/RR) Case**:
        - If the uncle is black and the new node, parent, and grandparent form a "line" (left-left or right-right configuration), perform a rotation to maintain balance.

    - Rotate the grandparent in the opposite direction of the "line." Recolour the nodes as needed, making the parent black and the grandparent red.

4. **Repeat**:
   - Continue the fix process until all Red-Black Tree properties are restored.

Use the link Red-Black Tree Simulator to access the simulator and experiment with the values:  10, 18, 7, 15, 16, 26, 17, 23, 28, 20, 20 to observe rotations and construction of a Red and Black tree.

### Red-Black Tree Deletion Algorithm

Refer to the article on the Red-Black Tree Deletion Algorithm for guidance. Test the algorithm with the following cases:

- **Case 1:** Either the node to be deleted or its parent is red. Construct a Red-Black Tree with the values: 20, 15, 25, 10. First, attempt to delete node 10, then restore the tree and delete node 15. Note that node 15 is black, and node 10 is red.
- **Case 2:** Both the parent and child nodes are black, while the sibling is black with at least one red child. Construct a Red-Black Tree with the values: 20, 30, 40, 35, and 50. Attempt to delete node 20. Note that nodes 20, 30, and 40 are black, while nodes 35 and 50 are red.
- **Case 3:** The node to be deleted is black, and its sibling is red. Construct a Red-Black Tree with the values: 10, 5, 20, 15, 25, 30. Attempt to delete node 5. Note that nodes 5, 10, 15, and 25 are black, while nodes 20 and 30 are red.

## K-D Trees

In a traditional binary search tree, each node contains a single key, allowing us to easily determine placement: if a new value is less than the node's key, we move to the left child; if it’s greater than or equal, we move to the right. This approach ensures that every node has a clear position based on a single comparison. But what if each node held multiple keys? How would we decide where to place new values, and would the tree still function as a binary search tree? This raises the question of whether a node in a binary search tree can truly have multiple keys.

The answer is yes; a node can hold multiple keys. In this case, each key represents a separate dimension. For instance, a typical binary search tree node with one key is considered one-dimensional. If a node contains two keys, it becomes two-dimensional. More generally, if a node has *k* keys, it is referred to as a *k*-dimensional node. A binary search tree built with such nodes, where each node can store multiple dimensions, is called a *k-d tree*. This concept extends the traditional binary search tree, allowing for more efficient searching and organization of data in multi-dimensional spaces.

## K-D Tree Operations

### Inserting a Node in a K-D Tree

Let's assume we have a set of points in a 2D space, where each point has two values: the x and y coordinates. Each coordinate represents one dimension, so each point has two dimensions in total. When constructing a binary search tree from these points, we cycle through the dimensions as we traverse down the tree.

For example, when comparing a new point with the root at Level 0, we compare the x coordinate of the root with the x coordinate of the new point. If the x coordinate of the new point is less than that of the root, we move to the left; otherwise, we move to the right. If there is no child in the appropriate position, we insert the new point there.

However, if a point already exists at Level 1, we compare the y coordinate of the new point with the y coordinate of the node at Level 1. If the y coordinate of the new point is less than that of the node, we traverse to the left; otherwise, we traverse to the right. If the child at Level 1 does not have a child at the appropriate pointer, we attach the new point there. If a child exists, we move to the child at Level 2 and compare the x coordinate of the new point with the x coordinate of the node at Level 2. If the x coordinate of the new point is less than that of the node at Level 2, we traverse to the left; otherwise, we traverse to the right.

Thus, as we traverse down the tree, we alternately switch between the x and y coordinates to find the appropriate position and attach the new point there. Fig. 9 shows the K-D tree generated from the given points.

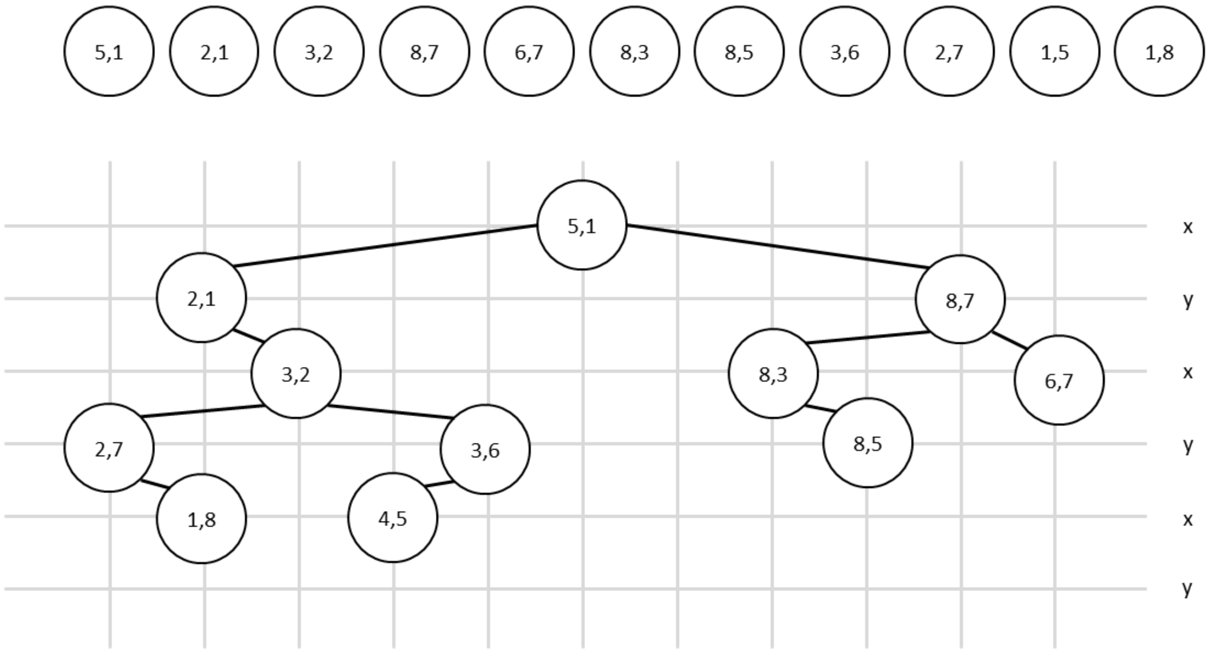


Fig. 9: K-D Tree after insertion of nodes.

## Finding the Minimum of a Dimension in a K-D Tree

To find the minimum in the given dimension of a K-D tree, the following algorithm can be used:

1. Start from the root.
2. If the dimension at the current level matches the given dimension, the required minimum will lie on the left side if there is a left child.
3. If the dimension at the current level is different, the minimum may be in either the left or right subtree, or the current node itself may also be the minimum.

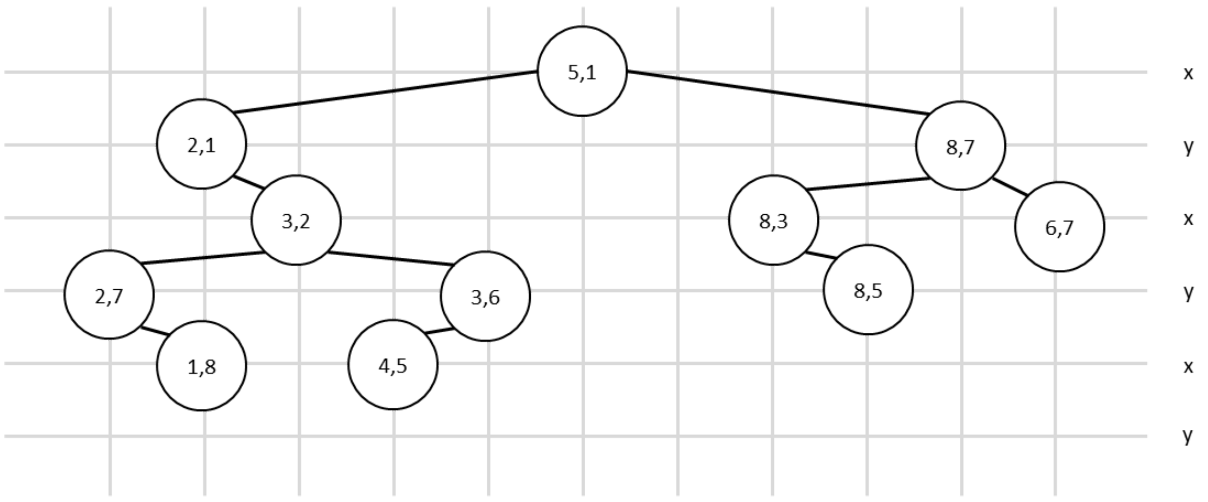


Fig. 10: Finding minimum of K-D tree.

Assuming we want to search for the minimum in the x dimension, we begin at the root. As the dimension of the root is x, we traverse left, where the minimum is likely to lie. This brings us to the node (2, 1). At node (2, 1), the dimension is y, so we need to search both the left and right subtrees. Since (2, 1) doesn't have a left subtree, we continue our search in the right subtree, reaching the node (3, 2). The dimension of (3, 2) is x, so we proceed only in the left subtree, leading us to the node (2, 7). At node (2, 7), the dimension is y, and we search for the minimum in both the left and right subtrees. With no left subtree at (2, 7), we move to the right subtree and reach (1, 8). As (1, 8) has no left or right subtrees, our search concludes here. We return (1, 8) from the right subtree of (2, 7). After comparing the x-coordinates of nodes (2, 7) and (1, 8), we conclude that node (1, 8) is the minimum. Therefore, the value returned from the left subtree of node (3, 2) is (1, 8), which is also the value returned from the right subtree of node (2, 1). Finally, node (1, 8) is compared with node (2, 1), and after comparing their x-coordinates, we confirm that node (1, 8) is the minimum, which is returned from the left subtree of node (5, 1).

Let’s now search for the minimum in the y dimension. We begin at the root. Since the root’s dimension is x, we continue our search in both the left and right subtrees. This leads us to node (2, 1) in the left subtree and node (8, 7) in the right subtree. The dimension of node (2, 1) is y, so the minimum is expected to lie in the left subtree. However, since node (2, 1) doesn't have a left subtree, our search for the minimum in the left subtree ends here. We return node (2, 1) as the minimum point in the y dimension from the left subtree of node (5, 1). Next, we search the right subtree, starting at node (8, 7). Since the dimension of node (8, 7) is y, we move to its left subtree, where the minimum is likely to be found. This leads us to node (8, 3). The dimension of node (8, 3) is x, so we search both its left and right subtrees. Since node (8, 3) doesn’t have a left subtree, we proceed to the right subtree, reaching node (8, 5). As node (8, 5) has no left or right subtrees, our search in the right subtree of node (8, 3) ends here. We return node (8, 5) from the right subtree of node (8, 3). After comparing the y-coordinates of nodes (8, 3) and (8, 5), since the y-coordinate of node (8, 3) is smaller, we return (8, 3) from the left subtree of node (8, 7). From the left subtree of node (5, 1), we get node (2, 1), and from the right subtree, we get the minimum node (8, 3). After comparing the y-coordinates of all three nodes (2, 1), (5, 1), and (8, 3), we conclude that node (2, 1) is the minimum in the y-dimension.

💡 Note that since nodes (2, 1) and (5, 1) have the same y-coordinate value, the algorithm may return (5, 1) as the minimum node in the y-dimension depending upon the implementation.

## Deleting a Node from a K-D Tree

To delete a node from a K-D tree, the following algorithm can be used:

1. Locate the node
2. If the node is found, proceed according to the applicable case.
    a. Case 1:
        i. If the node to be deleted is a leaf node, simply remove the node from the tree.
    b. Case 2:
        i. If the node to be deleted has a right subtree, find the minimum value in the current node’s dimension within the right subtree. Replace the node with the found minimum, then recursively delete the minimum in the right subtree.
    c. Case 3:
        i. If the node to be deleted has a left subtree, find the minimum value in the current node's dimension within the left subtree. Replace the node with the found minimum, and then recursively delete the minimum in the left subtree. Finally, make the new left subtree the right child of the current node.

Consider removing the node (4, 5). As node (4, 5) is a leaf node, it can be deleted directly from the tree. The overall structure of the K-D tree remains unaffected.

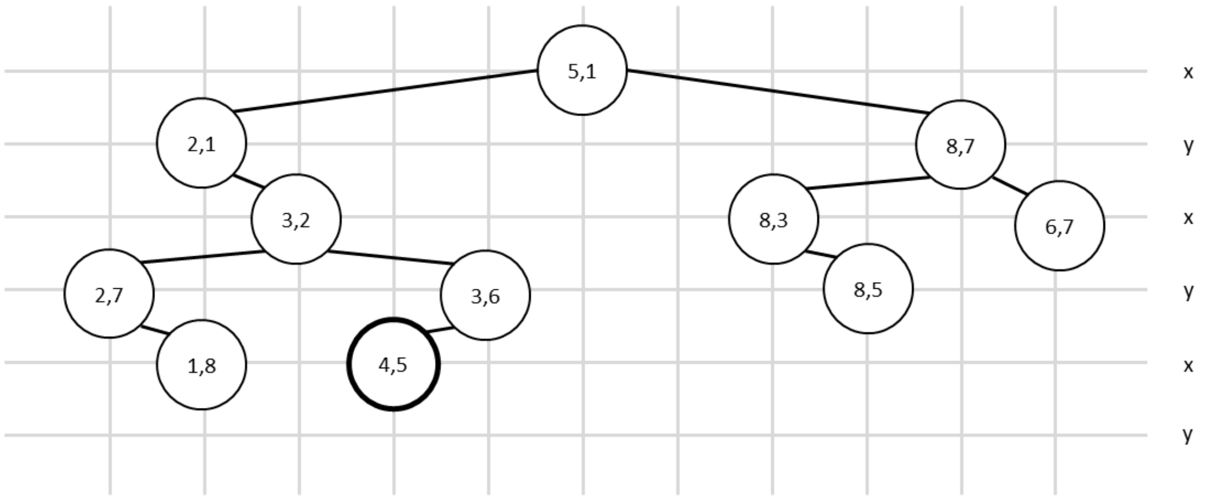


Fig. 11: Deleting a leaf node from K-D tree.

Now, let's consider removing the node (5, 1). Refer to Fig. 12(a). As node (5, 1) has a right subtree, we need to find the minimum value for the x dimension within this right subtree according to the deletion algorithm. Refer to Fig. 12(b). Our search begins at node (8, 7). Since the dimension of node (8, 7) is y, we need to find the minimum x value in both its left and right subtrees. This leads us to node (8, 3) in the left subtree and node (6, 7) in the right subtree. The dimension of node (8, 3) is x, so we attempt to traverse to its left. However, as node (8, 3) has no left subtree, our search stops there, and we return node (8, 3) from the left subtree of node (8, 7). Similarly, we return node (6, 7) from the right subtree because it is a leaf node. Comparing nodes (8, 7), (8, 3), and (6, 7), we determine that (6, 7) has the smallest x value. Refer to Fig. 12(c) and (d). We replace the value of node (5, 1) with (6, 7). Refer to Fig. 12(e).

Next, we begin a search from node (8, 7) to locate node (6, 7) for removal. We find node (6, 7) to the right of node (8, 7). As node (6, 7) is a leaf node, it can be deleted directly without affecting the overall structure of the K-D tree. Refer to Fig. 12(f). It shows the state of the K-D tree after deleting node (6, 7) from the right subtree of node (8, 7).

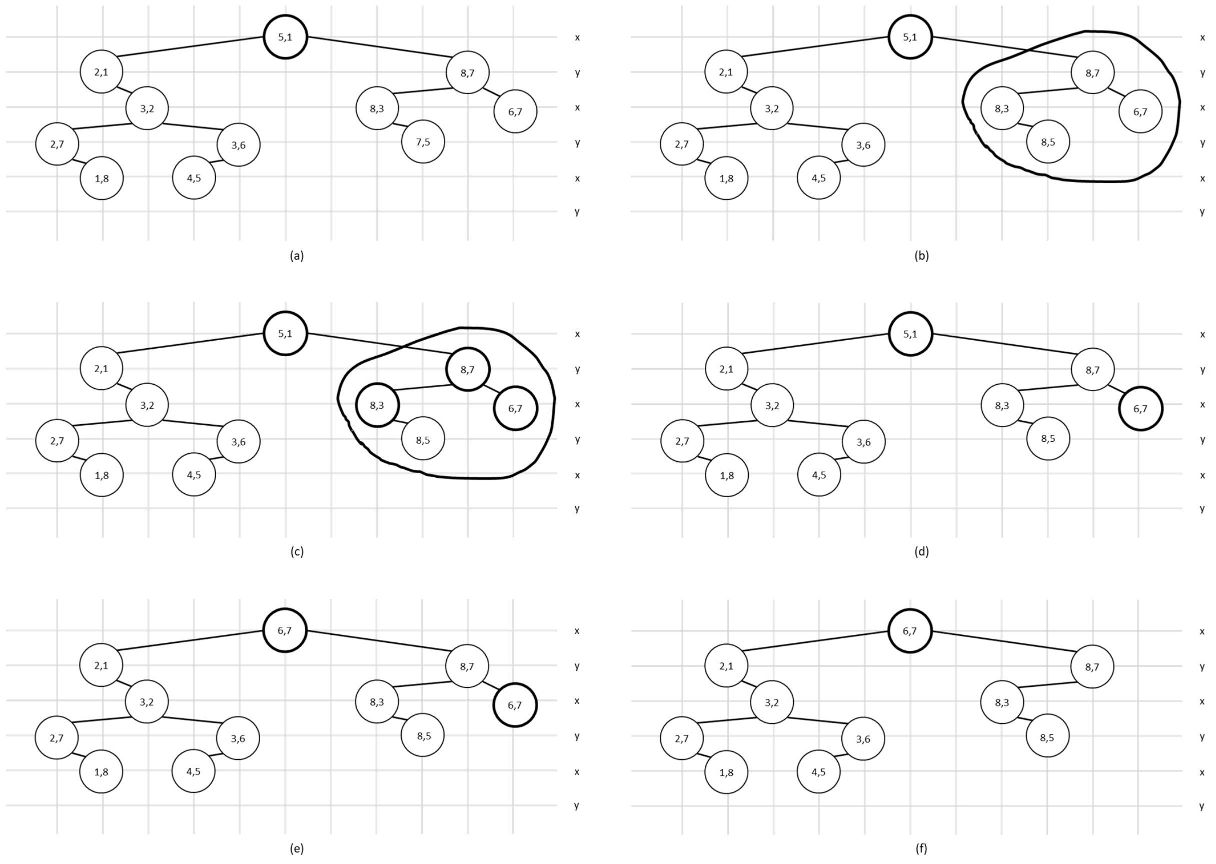


Fig. 12: Deleting a node having the right subtree.

Finally, let's try to delete the node (8, 7). Refer to Fig. 13(a). The node is neither a leaf nor does it have a right subtree, but it does have a left subtree. According to the algorithm (Case 3), we need to find the node in the left subtree with the minimum value in the y dimension. This leads us to the node (8, 3). As the dimension of node (8, 3) is x, we must search for the minimum y value in both its left and right subtrees. With no left subtree, we proceed to the right subtree, which brings us to the node (8, 5). The node (8, 5) has no left or right subtrees, so we return it from the right subtree. Now, we have two nodes, (8, 3) and (8, 5), to compare. Based on their y-coordinates, we identify node (8, 3) as the minimum in the y dimension. Refer to Fig. 13(b). We then replace the value of node (8, 7) with (8, 3). Refer to Fig. 13(c). Next, we need to delete the original node (8, 3). As this node has only a right child, we copy the value (8, 5) to node (8, 3). Finally, we delete the node (8, 5). Refer to Fig. 13(d). After deleting node (8, 5), the K-D tree appears as shown in Fig. 13(e). The deletion process is not complete yet (according to Case 3 of the deletion algorithm); one final adjustment is necessary: the node (7, 5), initially attached to the left of node (8, 3), must be placed on the right side to complete the deletion process.

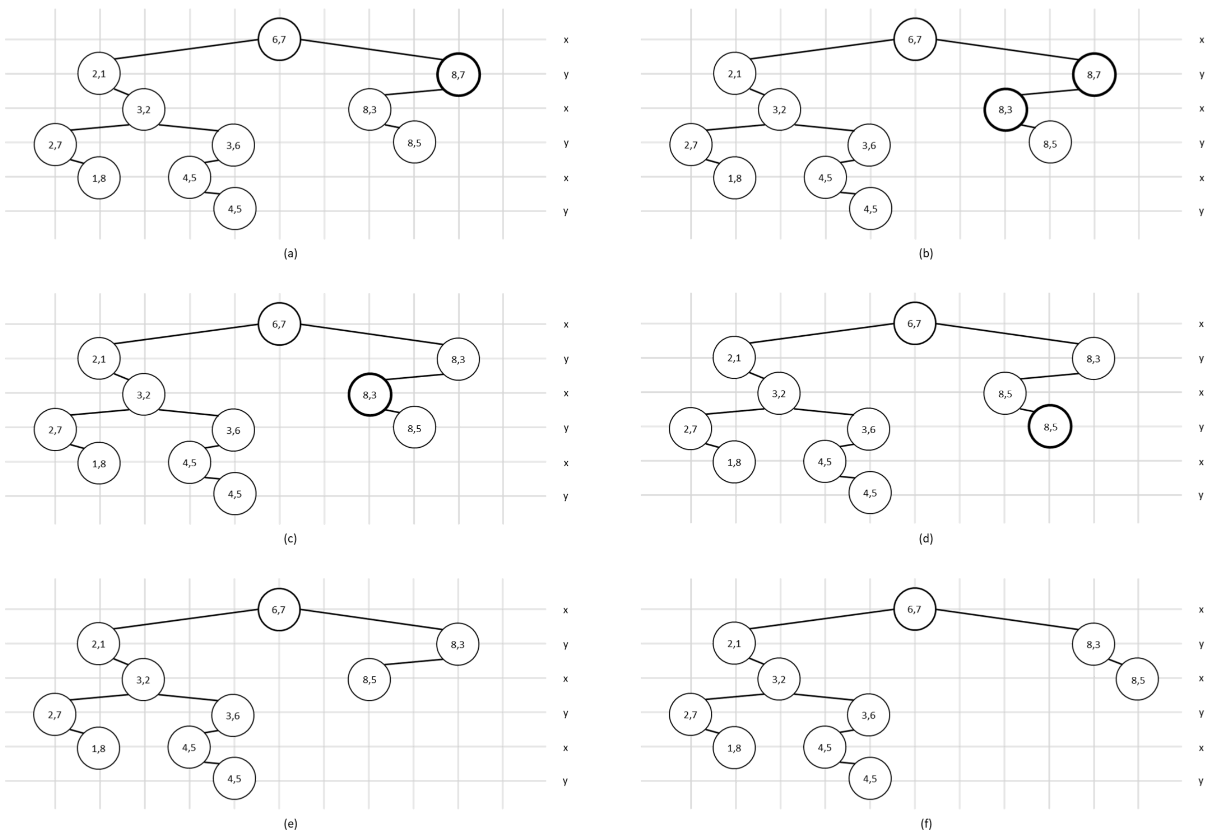


Fig. 13: Deleting a node having the left subtree.

## Balancing a K-D Tree

Balancing a K-D Tree involves rearranging nodes to ensure a balanced structure, improving query performance. Here's a general algorithm for balancing a K-D Tree:

1. **Sort the Points**: Sort the points based on the current dimension (axis).
2. **Find the Median**: Find the median point of the sorted array along the current dimension.
3. **Create a Node**: Make the median point the root node of the subtree.
4. **Recursively Build Subtrees**: Recursively build the left and right subtrees by repeating the process, alternating the splitting dimension at each level.

This process ensures the tree remains balanced, with each node split along a different dimension, which helps optimize search and insertion operations.

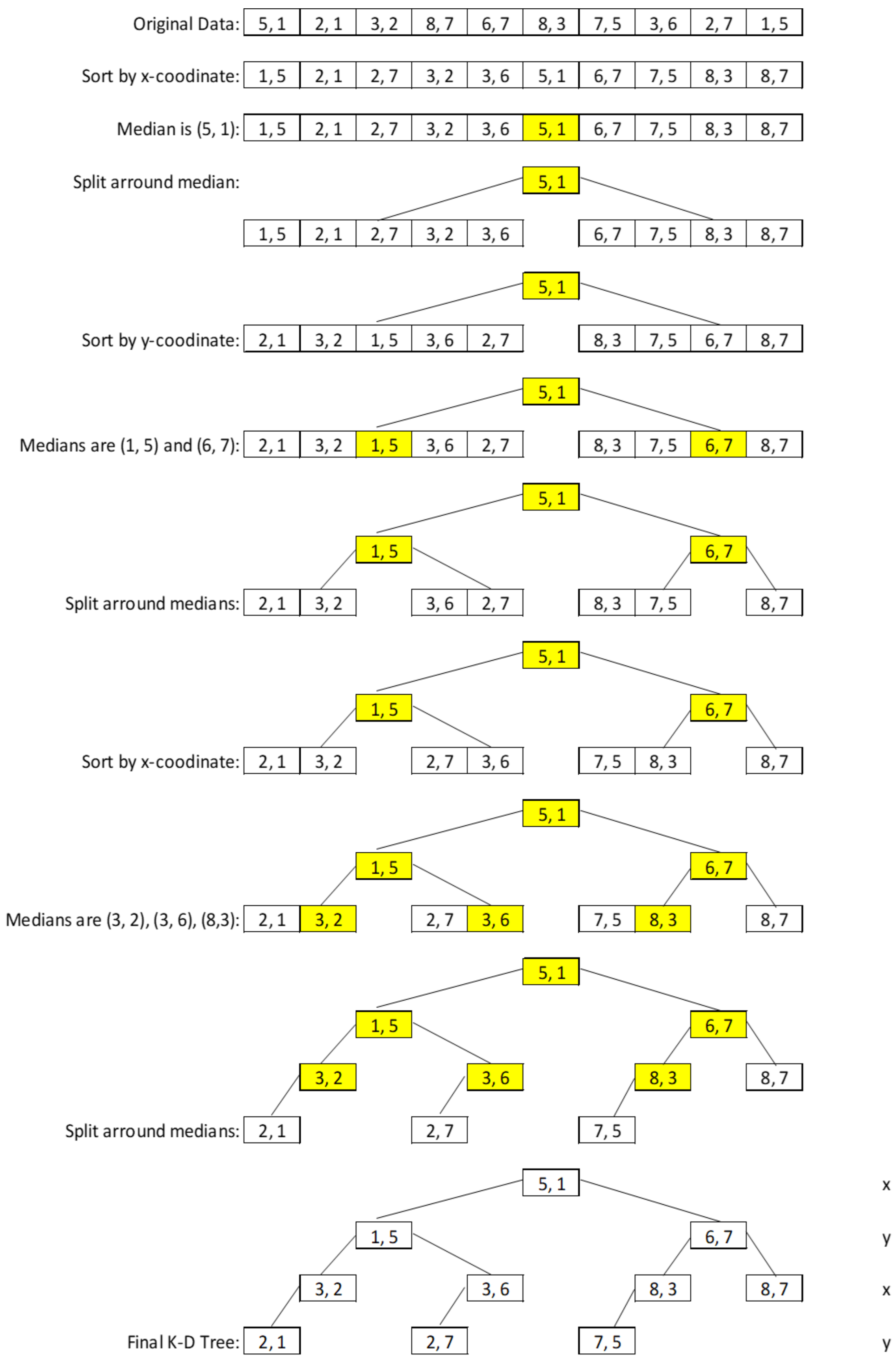


Fig. 14: Balancing a K-D tree

## Revisiting Binary Search Tree

A **binary search tree** is a tree in which each node has at most **two children**, known as the **left child** and **right child**. Because this is a two-branch structure, a binary search tree is also referred to as a **two-way search tree**. Thus, each binary tree node stores a single key and can point to two children.

### Can a node store multiple keys and have more than two children?

Yes, a node can store multiple keys and have more than two children in data structures. Such trees are called **multi-way search trees**.

### The degree of multi-way search tree

The **degree** of a multi-way search tree defines the maximum number of children a node can have. In a binary search tree, each node can have up to two children; hence, the degree of a binary search tree is 2. If a node of a tree can have a maximum of $m$ children, then the degree of such a search tree is $m$. A node of such a tree can have up to $m-1$ keys.

### How would a node of such a tree appear?

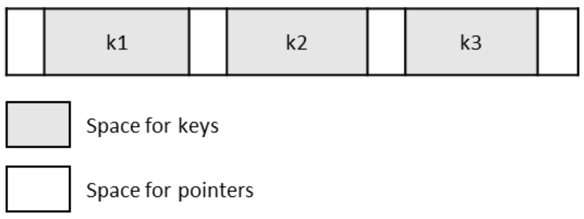


Fig. 15: Node of an order 4 multi-way tree.

### Limitations of general multi-way search trees

There is no specific policy for inserting a key in a multi-way search tree. While the tree structure shown in Fig. 16(a) is possible, the tree structure in Fig. 16(b) is also valid.

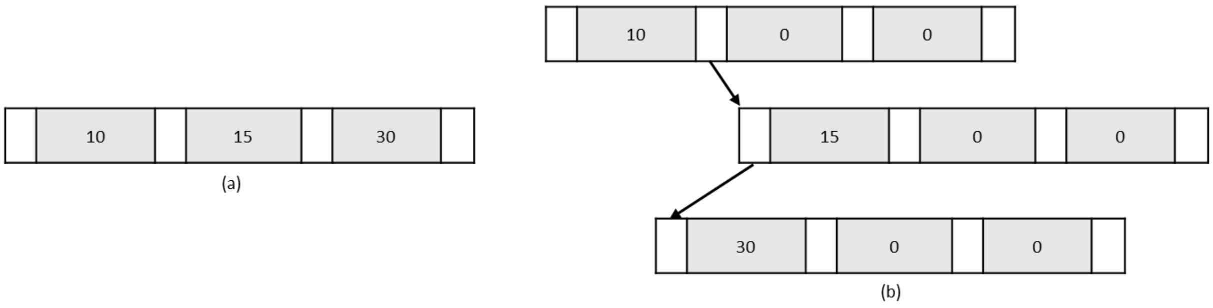


Fig. 16: (a) Full Node Example and (b) Non-Full Node Example of 4-Way Search Trees.

The arrangement shown in Fig. 16(b) is highly inefficient. If there are n values and only a single value is stored per node, the time complexity for the search operation will be O(n). We therefore need a stricter policy for constructing a multi-way tree. B-trees and B+ trees provide a stricter policy for inserting keys into the nodes.

https://www.youtube.com/watch?v=K1a2Bk8NrYQ

Use B Tree Simulator and B+ Tree Simulator to experiment out with the construction of B tree and
B+ tree.