Introduction of B+ Tree

Last Updated : 08 Mar, 2024

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree structure of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Features of B+ Trees

Balanced: B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.

Multi-level: B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.

Ordered: B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.

Fan-out: B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.

Cache-friendly: B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.

Disk-oriented: B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

Why Use B+ Tree?

B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.

B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Difference Between B+ Tree and B Tree

Some differences between B+ Tree and B Tree are stated below.

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	

Nodes store both keys and data values

Leaf Nodes

Leaf nodes form a linked list for efficient range-based queries

Leaf nodes do not form a linked list

Order

Higher order (more keys)

Lower order (fewer keys)

Key Duplication

Typically allows key duplication in leaf nodes

Usually does not allow key duplication

Disk Access

Better disk access due to sequential reads in a linked list structure

More disk I/O due to non-sequential reads in internal nodes

Applications

Database systems, file systems, where range queries are common

In-memory data structures, databases, general-purpose use

Performance

Better performance for range queries and bulk data retrieval

Balanced performance for search, insert, and delete operations

Memory Usage

Requires more memory for internal nodes

Requires less memory as keys and values are stored in the same node

Implementation of B+ Tree
In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record. From the above discussion, it is apparent that a B+ tree, unlike a B-tree, has two orders, ‘a’ and ‘b’, one for the internal nodes and the other for the external (or leaf) nodes.

Structure of B+ Trees
1

B+ Trees contain two types of nodes:

Internal Nodes: Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
Leaf Nodes: Leaf Nodes are the nodes that have n pointers.
The Structure of the Internal Nodes of a B+ Tree of Order ‘a’ is as Follows
Each internal node is of the form: <P1, K1, P2, K2, ….., Pc-1, Kc-1, Pc> where c <= a and each Pi is a tree pointer (i.e points to another node of the tree) and, each Ki is a key-value (see diagram-I for reference).
Every internal node has : K1 < K2 < …. < Kc-1
For each search field value ‘X’ in the sub-tree pointed at by Pi, the following condition holds: Ki-1 < X <= Ki, for 1 < I < c and, Ki-1 < X, for i = c (See diagram I for reference)
Each internal node has at most ‘aa tree pointers.
The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
If an internal node has ‘c’ pointers, c <= a, then it has ‘c – 1’ key values.
Structure of Internal Node
Structure of Internal Node

The Structure of the Leaf Nodes of a B+ Tree of Order ‘b’ is as Follows

Each leaf node is of the form: <<K1, D1>, <K2, D2>, ….., <Kc-1, Dc-1>, Pnext> where c <= b and each Di is a data pointer (i.e points to actual record in the disk whose key value is Ki or to a disk file block containing that record) and, each Ki is a key value and, Pnext points to next leaf node in the B+ tree (see diagram II for reference).
Every leaf node has : K1 < K2 < …. < Kc-1, c <= b
Each leaf node has at least \ceil(b/2) values.
All leaf nodes are at the same level.
Structure of Lead Node
Structure of Lead Node

Diagram-II Using the Pnext pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

Tree Pointer
Tree Pointer

Searching a Record in B+ Trees
1
Searching in B+ Tree

Let us suppose we have to find 58 in the B+ Tree. We will start by fetching from the root node then we will move to the leaf node, which might contain a record of 58. In the image given above, we will get 58 between 50 and 70. Therefore, we will we are getting a leaf node in the third leaf node and get 58 there. If we are unable to find that node, we will return that ‘record not founded’ message.

Insertion in B+ Trees
Insertion in B+ Trees is done via the following steps.

Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
Insert the key into the leaf node in increasing order if there is no overflow.
For more, refer to Insertion in a B+ Trees.

Deletion in B+Trees
Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

For more, refer to Deletion in B+ Trees.

Advantages of B+Trees
A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels. This accentuates the significant improvement made to the search time for

any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
Data stored in a B+ tree can be accessed both sequentially and directly.
It takes an equal number of disk accesses to fetch records.
B+trees have redundant search keys, and storing search keys repeatedly is not possible.
Disadvantages of B+ Trees
The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.
Application of B+ Trees
Multilevel Indexing
Faster operations on the tree (insertion, deletion, search)
Database indexing
Conclusion
In conclusion, B+ trees are an essential component of contemporary database systems since they significantly improve database performance and make efficient data management possible.

FAQs on B+ Trees
Q.1: What is a B+ Tree?
Answer:

B+ Tree is balanced binary search tree that can simply be a B Tree, where data is stored in keys, not key-value pairs.

Q.2: What is the advantage of the B+ Tree?
Answer:

The height of the B+ Trees is mostly balanced and is comparatively lesser than B-Trees.

Q.3: Where are B+ Trees used?
Answer:

B+ Trees are often used for disk-based storage systems.

Insertion in a B+ tree
Last Updated : 04 Apr, 2024
Prerequisite: Introduction of B+ trees
In this article, we will discuss that how to insert a node in B+ Tree. During insertion following properties of B+ Tree must be followed:

Each node except root can have a maximum of M children and at least ceil(M/2) children.
Each node can contain a maximum of M – 1 keys and a minimum of ceil(M/2) – 1 keys.
The root has at least two children and atleast one search key.
While insertion overflow of the node occurs when it contains more than M – 1 search key values.
Here M is the order of B+ tree.

Steps for insertion in B+ Tree
Every element is inserted into a leaf node. So, go to the appropriate leaf node.
Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow go ahead with the following steps mentioned below to deal with overflow while maintaining the B+ Tree properties.
Properties for insertion B+ Tree
Case 1: Overflow in leaf node

Split the leaf node into two nodes.
First node contains ceil((m-1)/2) values.
Second node contains the remaining values.
Copy the smallest search key value from second node to the parent node.(Right biased)

Below is the illustration of inserting 8 into B+ Tree of order of 5:

Case 2: Overflow in non-leaf node

Split the non leaf node into two nodes.
First node contains ceil(m/2)-1 values.
Move the smallest among remaining to the parent.
Second node contains the remaining keys.

Below is the illustration of inserting 15 into B+ Tree of order of 5:

Example to illustrate insertion on a B+ tree

Problem: Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.
Solution:
Step 1: The order is 3 so at maximum in a node so there can be only 2 search key values. As insertion happens on a leaf node only in a B+ tree so insert search key value 6 and 16 in increasing order in the node. Below is the illustration of the same:

Step 2: We cannot insert 26 in the same node as it causes an overflow in the leaf node, We have to split the leaf node according to the rules. First part contains ceil((3-1)/2) values i.e., only 6. The second node contains the remaining values i.e., 16 and 26. Then also copy the smallest search key value from the second node to the parent node i.e., 16 to the parent node. Below is the illustration of the same:

Step 3: Now the next value is 36 that is to be inserted after 26 but in that node, it causes an overflow again in that leaf node. Again follow the above steps to split the node. First part contains ceil((3-1)/2) values i.e., only 16. The second node contains the remaining values i.e., 26 and 36. Then also copy the smallest search key value from the second node to the parent node i.e., 26 to the parent node. Below is the illustration of the same:
The illustration is shown in the diagram below.

Step 4: Now we have to insert 46 which is to be inserted after 36 but it causes an overflow in the leaf node. So we split the node according to the rules. The first part contains 26 and the second part contains 36 and 46 but now we also have to copy 36 to the parent node but it causes overflow as only two search key values can be accommodated in a node. Now follow the steps to deal with overflow in the non-leaf node.
First node contains ceil(3/2)-1 values i.e. ’16’.
Move the smallest among remaining to the parent i.e ’26’ will be the new parent node.
The second node contains the remaining keys i.e ’36’ and the rest of the leaf nodes remain the same. Below is the illustration of the same:

AVL Tree Data Structure
Last Updated : 24 Feb, 2025
An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The absolute difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node. The balance factor for all nodes must be less than or equal to 1.
Every AVL tree is also a Binary Search Tree (Left subtree values Smaller and Right subtree values grater for every node), but every BST is not AVL Tree. For example, the second diagram below is not an AVL Tree.
The main advantage of an AVL Tree is, the time complexities of all operations (search, insert and delete, max, min, floor and ceiling) become O(Log n). This happens because height of an AVL tree is bounded by O(Log n). In case of a normal BST, the height can go up to O(n).
An AVL tree maintains its height by doing some extra work during insert and delete operations. It mainly uses rotations to maintain both BST properties and height balance.
There exist other self-balancing BSTs also like Red Black Tree. Red Black tree is more complex, but used more in practice as it is less restrictive in terms of left and right subtree height differences.
Example of an AVL Tree:
The balance factors for different nodes are : 12 :1, 8:1, 18:1, 5:1, 11:0, 17:0 and 4:0. Since all differences are less than or equal to 1, the tree is an AVL tree.

AVL tree
AVL tree

Example of a BST which is NOT AVL:
The Below Tree is NOT an AVL Tree as the balance factor for nodes 8, 4 and 7 is more than 1.

BST-Unbalanced
Not an AVL Tree

Operations on an AVL Tree:
Searching : It is same as normal Binary Search Tree (BST) as an AVL Tree is always a BST. So we can use the same implementation as BST. The advantage here is time complexity is O(Log n)
Insertion : It does rotations along with normal BST insertion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after insertion
Deletion : It also does rotations along with normal BST deltion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after deletion.
Please refer Insertion in AVL Tree and Deletion in AVL Tree for details.

Rotating the subtrees (Used in Insertion and Deletion)
An AVL tree may rotate in one of the following four ways to keep itself balanced while making sure that the BST properties are maintained.

Left Rotation:

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.

Left-Rotation in AVL tree

Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.

avl-tree
Right-Rotation in AVL Tree

Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

Left-Right Rotation in AVL tree

Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

Right-Left Rotation in AVL tree

Advantages of AVL Tree:
AVL trees can self-balance themselves and therefore provides time complexity as O(Log n) for search, insert and delete.
It is a BST only (with balancing), so items can be traversed in sorted order.
Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
AVL tree is relatively less complex to understand and implement compared to Red Black Trees.
Disadvantages of AVL Tree:
It is difficult to implement compared to normal BST and easier compared to Red Black
Less used compared to Red-Black trees. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
Applications of AVL Tree:
AVL Tree is used as a first example self balancing BST in teaching DSA as it is easier to understand and implement compared to Red Black

Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max.
Red Black tree is more commonly implemented in language libraries like map in C++, set in C++, TreeMap in Java and TreeSet in Java.
AVL Trees can be used in a real time environment where predictable and consistent performance is required.

Insertion in an AVL Tree
Last Updated : 22 Feb, 2025
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Example of AVL Tree:

The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

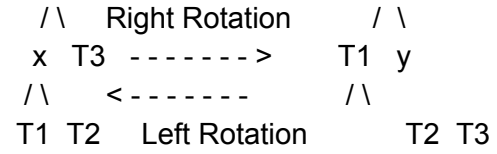
Example of a Tree that is NOT an AVL Tree:

The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees? Most of the BST operations (e.g., search, max, min, insert, delete, floor and ceiling) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log(n)) after every insertion and deletion, then we can guarantee an upper bound of O(log(n)) for all these operations. The height of an AVL tree is always O(log(n)) where n is the number of nodes in the tree.

Insertion in AVL Tree:
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

Left Rotation
Right Rotation
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)



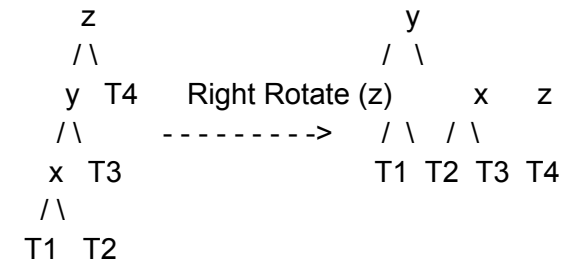
Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
Steps to follow for insertion:

Let the newly inserted node be w

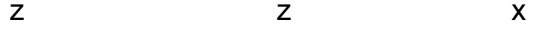
Perform standard BST insert for w.
Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.
Following are the possible 4 arrangements:
y is the left child of z and x is the left child of y (Left Left Case)
y is the left child of z and x is the right child of y (Left Right Case)
y is the right child of z and x is the right child of y (Right Right Case)
y is the right child of z and x is the left child of y (Right Left Case)
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with z becomes the same as it was before insertion.

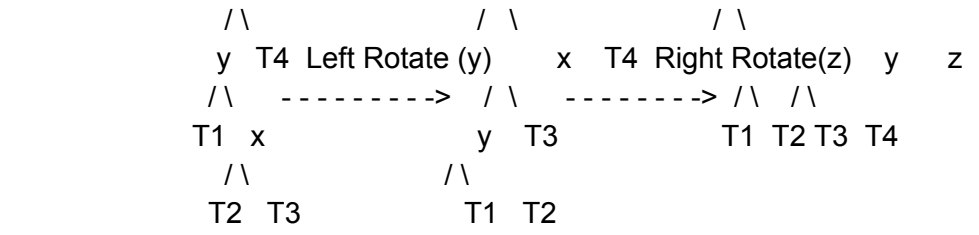
Try it on GfG Practice
redirect icon
1. Left Left Case

T1, T2, T3 and T4 are subtrees.

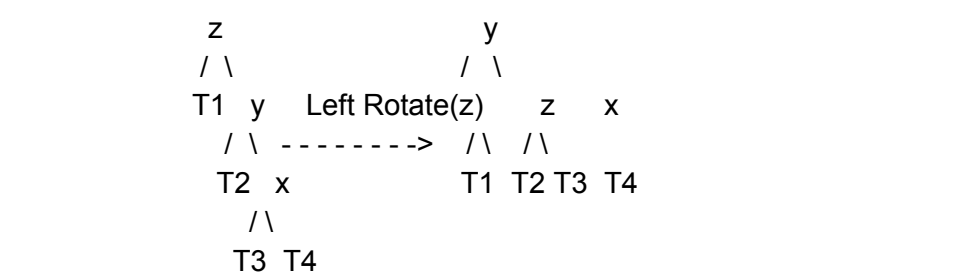


2. Left Right Case





3. Right Right Case



4. Right Left Case

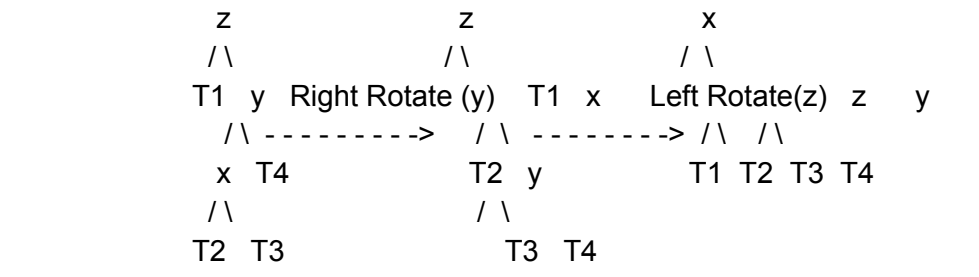


Illustration of Insertion at AVL Tree

avlinsert1

avlinsert2-jpg

avlinsert3

avlinsert4

avlinsert5

Approach: The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

Perform the normal BST insertion.
The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
Get the balance factor (left subtree height – right subtree height) of the current node.
If the balance factor is greater than 1, then the current node is unbalanced and we are either in the Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in the left subtree root.
If the balance factor is less than -1, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.
Below is the implementation of the above approach:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

# A utility function to get the
# height of the tree
def height(node):
    if not node:
        return 0
    return node.height

# A utility function to right rotate
# subtree rooted with y
def right_rotate(y):
    x = y.left
    T2 = x.right
```

```
    # Perform rotation
    x.right = y
    y.left = T2

    # Update heights
    y.height = 1 + max(height(y.left), height(y.right))
    x.height = 1 + max(height(x.left), height(x.right))

    # Return new root
    return x

# A utility function to left rotate
# subtree rooted with x
def left_rotate(x):
    y = x.right
    T2 = y.left

    # Perform rotation
    y.left = x
    x.right = T2

    # Update heights
    x.height = 1 + max(height(x.left), height(x.right))
    y.height = 1 + max(height(y.left), height(y.right))

    # Return new root
    return y

# Get balance factor of node N
def get_balance(node):
    if not node:
        return 0
    return height(node.left) - height(node.right)

# Recursive function to insert a key in
# the subtree rooted with node
def insert(node, key):

    # Perform the normal BST insertion
    if not node:
        return Node(key)

    if key < node.key:
        node.left = insert(node.left, key)
```

```
    elif key > node.key:
        node.right = insert(node.right, key)
    else:
        # Equal keys are not allowed in BST
        return node

    # Update height of this ancestor node
    node.height = 1 + max(height(node.left), height(node.right))

    # Get the balance factor of this ancestor node
    balance = get_balance(node)

    # If this node becomes unbalanced,
    # then there are 4 cases

    # Left Left Case
    if balance > 1 and key < node.left.key:
        return right_rotate(node)

    # Right Right Case
    if balance < -1 and key > node.right.key:
        return left_rotate(node)

    # Left Right Case
    if balance > 1 and key > node.left.key:
        node.left = left_rotate(node.left)
        return right_rotate(node)

    # Right Left Case
    if balance < -1 and key < node.right.key:
        node.right = right_rotate(node.right)
        return left_rotate(node)

    # Return the (unchanged) node pointer
    return node

# A utility function to print preorder
# traversal of the tree
def pre_order(root):
    if root:
        print(root.key, end=" ")
        pre_order(root.left)
        pre_order(root.right)
```

```
# Driver code
root = None

# Constructing tree given in the above figure
root = insert(root, 10)
root = insert(root, 20)
root = insert(root, 30)
root = insert(root, 40)
root = insert(root, 50)
root = insert(root, 25)

# The constructed AVL Tree would be
#         30
#        /  \
#      20   40
#     /  \    \
#    10  25   50

print("Preorder traversal :")
pre_order(root)
```

Output
Preorder traversal :
30 20 10 25 40 50
Time Complexity: O(log(n)), For Insertion
Auxiliary Space: O(Log n) for recursion call stack as we have written a recursive method to insert

The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the AVL insert remains the same as the BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Comparison with Red Black Tree:
The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then the AVL tree should be preferred over Red Black Tree.

Insertion, Searching and Deletion in AVL trees containing a parent node pointer

Last Updated : 21 May, 2024
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. The insertion and deletion in AVL trees have been discussed in the previous article. In this article, insert, search, and delete operations are discussed on AVL trees that also have a parent pointer in their structure.

Definition of AVL tree node:

```
# Python code

class AVLwithParent:
    def __init__(self):
        # Pointer to the left and the right subtree
        self.left = None
        self.right = None

        # Stores the data in the node
        self.key = None

        # Stores the parent pointer
        self.par = None

        # Stores the height of the current tree
        self.height = None

# This code is contributed by princekumaras
```

Representation of the Node:

Below is the example of an AVL tree containing a parent pointer:

Insert Operation: The insertion procedure is similar to that of a normal AVL tree without a parent pointer, but in this case, the parent pointers need to be updated with every insertion and rotation accordingly. Follow the steps below to perform insert operation:

Perform standard BST insert for the node to be placed at its correct position.
Increase the height of each node encountered by 1 while finding the correct position for the node to be inserted.
Update the parent and child pointers of the inserted node and its parent respectively.

Starting from the inserted node till the root node check if the AVL condition is satisfied for each node on this path.
If w is the node where the AVL condition is not satisfied then we have 4 cases:
Left Left Case: (If the left subtree of the left child of w has the inserted node)
Left Right Case: (If the right subtree of the left child of w has the inserted node)
Right Left Case: (If the left subtree of the right child of w has the inserted node)
Right Right Case: (If the right subtree of the right child of w has the inserted node)
Below is the implementation of the above approach:

```
class AVLwithparent:
    def __init__(self, key):
        # Initialize a node with key, left and right child, parent, and height.
        self.left = None
        self.right = None
        self.key = key
        self.par = None
        self.height = 1

# Function to update the height of a node based on its children's heights
def update_height(root):
    if root is not None:
        left_height = root.left.height if root.left else 0  # Get the height of the left child
        right_height = root.right.height if root.right else 0  # Get the height of the right child
        root.height = max(left_height, right_height) + 1  # Update the height of the current node

# Left-Left Rotation (LLR) to balance the AVL tree
def LLR(root):
    # Perform a left rotation and then a right rotation
    # to balance the tree when there's an imbalance in the left subtree
    tmpnode = root.left
    root.left = tmpnode.right
    if tmpnode.right:
        tmpnode.right.par = root
    tmpnode.right = root
    tmpnode.par = root.par
    root.par = tmpnode
    if tmpnode.par:
        if root.key < tmpnode.par.key:
            tmpnode.par.left = tmpnode
        else:
            tmpnode.par.right = tmpnode
```

```
    update_height(root)
    update_height(tmpnode)
    return tmpnode

# Right-Right Rotation (RRR) to balance the AVL tree
def RRR(root):
    # Perform a right rotation and then a left rotation
    # to balance the tree when there's an imbalance in the right subtree
    tmpnode = root.right
    root.right = tmpnode.left
    if tmpnode.left:
        tmpnode.left.par = root
    tmpnode.left = root
    tmpnode.par = root.par
    root.par = tmpnode
    if tmpnode.par:
        if root.key < tmpnode.par.key:
            tmpnode.par.left = tmpnode
        else:
            tmpnode.par.right = tmpnode
    update_height(root)
    update_height(tmpnode)
    return tmpnode

# Left-Right Rotation (LRR) to balance the AVL tree
def LRR(root):
    # Perform a right rotation on the left child and then a left rotation on the root
    root.left = RRR(root.left)
    return LLR(root)

# Right-Left Rotation (RLR) to balance the AVL tree
def RLR(root):
    # Perform a left rotation on the right child and then a right rotation on the root
    root.right = LLR(root.right)
    return RRR(root)

# Function to insert a key into the AVL tree and balance the tree if needed
def insert(root, parent, key):
    if root is None:
        root = AVLwithparent(key)  # Create a new node if the current node is None
        root.par = parent  # Set the parent of the new node
    elif root.key > key:
        # Insert the key into the left subtree and balance the tree if needed
        root.left = insert(root.left, root, key)
```

```
        left_height = root.left.height if root.left else 0
        right_height = root.right.height if root.right else 0
        if abs(left_height - right_height) == 2:
            if key < root.left.key:
                root = LLR(root)
            else:
                root = LRR(root)
    elif root.key < key:
        # Insert the key into the right subtree and balance the tree if needed
        root.right = insert(root.right, root, key)
        left_height = root.left.height if root.left else 0
        right_height = root.right.height if root.right else 0
        if abs(left_height - right_height) == 2:
            if key < root.right.key:
                root = RLR(root)
            else:
                root = RRR(root)
    update_height(root)  # Update the height of the current node after insertion
    return root  # Return the root of the updated subtree

# Function to print the nodes of the AVL tree in preorder
def print_preorder(root):
    if root:
        parent_key = root.par.key if root.par else "NULL"  # Get the key of the parent node or
"NULL" if it's None
        print(f"Node: {root.key}, Parent Node: {parent_key}")  # Print the node and its parent
        print_preorder(root.left)  # Print the left subtree in preorder
        print_preorder(root.right)  # Print the right subtree in preorder

# Main function to demonstrate AVL tree operations
if __name__ == "__main__":
    root = None  # Initialize the root of the AVL tree as None
    # Insert keys into the AVL tree
    root = insert(root, None, 10)
    root = insert(root, None, 20)
    root = insert(root, None, 30)
    root = insert(root, None, 40)
    root = insert(root, None, 50)
    root = insert(root, None, 25)
    # Print the AVL tree in preorder
    print_preorder(root)
```

Output

```
Node: 30, Parent Node: NULL
```

Node: 20, Parent Node: 30
Node: 10, Parent Node: 20
Node: 25, Parent Node: 20
Node: 40, Parent Node: 30
Node: 50, Parent Node: 40

Time Complexity: O(log N), where N is the number of nodes of the tree.
Auxiliary Space: O(1)

Search Operation: The search operation in an AVL tree with parent pointers is similar to the search operation in a normal Binary Search Tree. Follow the steps below to perform search operation:

Start from the root node.
If the root node is NULL, return false.
Check if the current node’s value is equal to the value of the node to be searched. If yes, return true.
If the current node’s value is less than searched key then recur to the right subtree.
If the current node’s value is greater than searched key then recur to the left subtree.
Below is the implementation of the above approach:

```
# Python program for the above approach

# AVL tree node
class AVLwithparent:
    def __init__(self, key, parent=None):
        self.left = None
        self.right = None
        self.key = key
        self.par = parent
        self.height = 1

# Function to update the height of
# a node according to its children's
# node's heights
def update_height(root):
    if root is not None:

        # Store the height of the
```

```
        # current node
        val = 1

        # Store the height of the left
        # and the right subtree
        if root.left is not None:
            val = root.left.height + 1

        if root.right is not None:
            val = max(val, root.right.height + 1)

        # Update the height of the
        # current node
        root.height = val

# Function to handle Left Left Case
def llr(root):
    # Create a reference to the
    # left child
    tmp_node = root.left

    # Update the left child of the
    # root to the right child of the
    # current left child of the root
    root.left = tmp_node.right

    # Update parent pointer of the left
    # child of the root node
    if tmp_node.right is not None:
        tmp_node.right.par = root

    # Update the right child of
    # tmp_node to root
    tmp_node.right = root

    # Update parent pointer of tmp_node
    tmp_node.par = root.par

    # Update the parent pointer of root
    root.par = tmp_node

    # Update tmp_node as the left or
    # the right child of its parent
    # pointer according to its key value
```

```
    if tmp_node.par is not None and root.key < tmp_node.par.key:
        tmp_node.par.left = tmp_node
    else:
        if tmp_node.par is not None:
            tmp_node.par.right = tmp_node

    # Make tmp_node as the new root
    root = tmp_node

    # Update the heights
    update_height(root.left)
    update_height(root.right)
    update_height(root)
    update_height(root.par)

    # Return the root node
    return root

# Function to handle Right Right Case
def rrr(root):
    # Create a reference to the
    # right child
    tmp_node = root.right

    # Update the right child of the
    # root as the left child of the
    # current right child of the root
    root.right = tmp_node.left

    # Update parent pointer of the right
    # child of the root node
    if tmp_node.left is not None:
        tmp_node.left.par = root

    # Update the left child of the
    # tmp_node to root
    tmp_node.left = root

    # Update parent pointer of tmp_node
    tmp_node.par = root.par

    # Update the parent pointer of root
    root.par = tmp_node
```

```
    # Update tmp_node as the left or
    # the right child of its parent
    # pointer according to its key value
    if tmp_node.par is not None and root.key < tmp_node.par.key:
        tmp_node.par.left = tmp_node
    else:
        if tmp_node.par is not None:
            tmp_node.par.right = tmp_node

    # Make tmp_node as the new root
    root = tmp_node

    # Update the heights
    update_height(root.left)
    update_height(root.right)
    update_height(root)
    update_height(root.par)

    # Return the root node
    return root

# Function to handle Left Right Case
def lrr(root):
    root.left = rrr(root.left)
    return llr(root)

# Function to handle Right Left Case
def rlr(root):
    root.right = llr(root.right)
    return rrr(root)

# Function to insert a node in
# the AVL tree
def insert(root, parent, key):
    if root is None:

        # Create and assign values
        # to a new node
        root = AVLwithparent(key, parent)

    else:
        if root.key > key:

            # Recur to the left subtree
```

```
    # to insert the node
    root.left = insert(root.left, root, key)

    # Stores the heights of the
    # left and right subtree
    first_height = 0
    second_height = 0

    if root.left is not None:
        first_height = root.left.height

    if root.right is not None:
        second_height = root.right.height

    # Balance the tree if the
    # current node is not balanced
    if abs(first_height - second_height) == 2:

        if root.left is not None and key < root.left.key:

            # Left Left Case
            root = llr(root)
        else:

            # Left Right Case
            root = lrr(root)

elif root.key < key:

    # Recur to the right subtree
    # to insert the node
    root.right = insert(root.right, root, key)

    # Store the heights of the left
    # and right subtree
    first_height = 0
    second_height = 0

    if root.left is not None:
        first_height = root.left.height

    if root.right is not None:
        second_height = root.right.height
```

```
            # Balance the tree if the
            # current node is not balanced
            if abs(first_height - second_height) == 2:
                if root.right is not None and key < root.right.key:

                    # Right Left Case
                    root = rlr(root)
                else:

                    # Right Right Case
                    root = rrr(root)

    # Update the height of the
    # root node
    update_height(root)

    # Return the root node
    return root

# Function to find a key in AVL tree
def avl_search(root, key):
    # If root is None
    if root is None:
        return False

    # If found, return True
    elif root.key == key:
        return True

    # Recur to the left subtree if
    # the current node's value is
    # greater than key
    elif root.key > key:
        return avl_search(root.left, key)

    # Otherwise, recur to the
    # right subtree
    else:
        return avl_search(root.right, key)

# Driver Code
if __name__ == "__main__":
    root = None
```

```
    # Function call to insert the nodes
    root = insert(root, None, 10)
    root = insert(root, None, 20)
    root = insert(root, None, 30)
    root = insert(root, None, 40)
    root = insert(root, None, 50)
    root = insert(root, None, 25)

    # Function call to search for a node
    found = avl_search(root, 40)
    if found:
        print("Value found")
    else:
        print("Value not found")
```

Output
value found
Time Complexity: O(log N), where N is the number of nodes of the tree
Auxiliary Space: O(1)

Delete Operation: The deletion procedure is similar to that of a normal AVL tree without a parent pointer, but in this case, the references to the parent pointers need to be updated with every deletion and rotation accordingly. Follow the steps below to perform the delete operation:

Perform the delete procedure as in a normal BST.
From the node that has been deleted, move towards the root.
At each node on the path, update the height of the node.
Check for AVL conditions at each node. Let there be 3 nodes: w, x, y where w is the current node, x is the root of the subtree of w which has greater height and y is the root of the subtree of x which has greater height.
If the node w is unbalanced, there exists one of the following 4 cases:
Left Left Case (x is left child of w and y is left child of x)
Left Right Case (x is left child of w and y is right child of x)
Right Left Case (x is right child of w and y is left child of x)
Right Right Case (x is right child of w and y is right child of x)
Below is the implementation of the above approach:

```
class AVLwithparent:
    def __init__(self, key, parent=None):
        self.left = None
        self.right = None
```

```
        self.key = key
        self.par = parent
        self.height = 1


def print_preorder(root):
    if root:
        print("Node:", root.key, ", Parent Node:", root.par.key if root.par else "NULL")
        print_preorder(root.left)
        print_preorder(root.right)


def update_height(root):
    if root:
        val = 1
        if root.left:
            val = root.left.height + 1
        if root.right:
            val = max(val, root.right.height + 1)
        root.height = val


def llr(root):
    tmp_node = root.left
    root.left = tmp_node.right
    if tmp_node.right:
        tmp_node.right.par = root
    tmp_node.right = root
    tmp_node.par = root.par
    root.par = tmp_node
    if tmp_node.par and root.key < tmp_node.par.key:
        tmp_node.par.left = tmp_node
    elif tmp_node.par:
        tmp_node.par.right = tmp_node
    root = tmp_node
    update_height(root.left)
    update_height(root.right)
    update_height(root)
    update_height(root.par)
    return root


def rrr(root):
    tmp_node = root.right
```

```
    root.right = tmp_node.left
    if tmp_node.left:
        tmp_node.left.par = root
    tmp_node.left = root
    tmp_node.par = root.par
    root.par = tmp_node
    if tmp_node.par and root.key < tmp_node.par.key:
        tmp_node.par.left = tmp_node
    elif tmp_node.par:
        tmp_node.par.right = tmp_node
    root = tmp_node
    update_height(root.left)
    update_height(root.right)
    update_height(root)
    update_height(root.par)
    return root


def lrr(root):
    root.left = rrr(root.left)
    return llr(root)


def rlr(root):
    root.right = llr(root.right)
    return rrr(root)


def balance(root):
    first_height = 0
    second_height = 0
    if root.left:
        first_height = root.left.height
    if root.right:
        second_height = root.right.height
    if abs(first_height - second_height) == 2:
        if first_height < second_height:
            rightheight1 = 0
            rightheight2 = 0
            if root.right.right:
                rightheight2 = root.right.right.height
            if root.right.left:
                rightheight1 = root.right.left.height
            if rightheight1 > rightheight2:
```

```
                root = rlr(root)
            else:
                root = rrr(root)
        else:
            leftheight1 = 0
            leftheight2 = 0
            if root.left.right:
                leftheight2 = root.left.right.height
            if root.left.left:
                leftheight1 = root.left.left.height
            if leftheight1 > leftheight2:
                root = llr(root)
            else:
                root = lrr(root)
    return root


def insert(root, parent, key):
    if root is None:
        root = AVLwithparent(key, parent)
    elif root.key > key:
        root.left = insert(root.left, root, key)
        first_height = 0
        second_height = 0
        if root.left:
            first_height = root.left.height
        if root.right:
            second_height = root.right.height
        if abs(first_height - second_height) == 2:
            if root.left and key < root.left.key:
                root = llr(root)
            else:
                root = lrr(root)
    elif root.key < key:
        root.right = insert(root.right, root, key)
        first_height = 0
        second_height = 0
        if root.left:
            first_height = root.left.height
        if root.right:
            second_height = root.right.height
        if abs(first_height - second_height) == 2:
            if root.right and key < root.right.key:
                root = rlr(root)
```

```
            else:
                root = rrr(root)
        update_height(root)
        return root


def delete(root, key):
    if root:
        if root.key == key:
            if root.right is None and root.left is not None:
                if root.par:
                    if root.par.key < root.key:
                        root.par.right = root.left
                    else:
                        root.par.left = root.left
                    update_height(root.par)
                root.left.par = root.par
                root.left = balance(root.left)
                return root.left
            elif root.left is None and root.right is not None:
                if root.par:
                    if root.par.key < root.key:
                        root.par.right = root.right
                    else:
                        root.par.left = root.right
                    update_height(root.par)
                root.right.par = root.par
                root.right = balance(root.right)
                return root.right
            elif root.left is None and root.right is None:
                if root.par:
                    if root.par.key < root.key:
                        root.par.right = None
                    else:
                        root.par.left = None
                    update_height(root.par)
                root = None
                return None
            else:
                tmp_node = root
                tmp_node = tmp_node.right
                while tmp_node.left:
                    tmp_node = tmp_node.left
                val = tmp_node.key
```

```
            root.right = delete(root.right, tmp_node.key)
            root.key = val
            root = balance(root)
        elif root.key < key:
            root.right = delete(root.right, key)
            root = balance(root)
        elif root.key > key:
            root.left = delete(root.left, key)
            root = balance(root)
        update_height(root)
    return root


# Driver Code
if __name__ == "__main__":
    root = None

    # Function call to insert the nodes
    root = insert(root, None, 9)
    root = insert(root, None, 5)
    root = insert(root, None, 10)
    root = insert(root, None, 0)
    root = insert(root, None, 6)

    # Print the tree before deleting node
    print("Before deletion:")
    print_preorder(root)

    # Function Call to delete node 10
    root = delete(root, 10)

    # Print the tree after deleting node
    print("After deletion:")
    print_preorder(root)
```

Output

```
Before deletion:
Node: 9, Parent Node: NULL
Node: 5, Parent Node: 9
Node: 0, Parent Node: 5
Node: 6, Parent Node: 5
Node: 10, Parent Node: 9
After deletion:
Node: 6, Parent Node: NULL
```

Node: 5, Parent ...

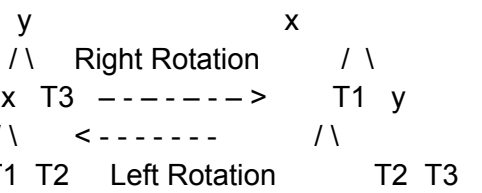
Time Complexity: O(log N), where N is the number of nodes of the tree
Auxiliary Space: O(1)

Deletion in an AVL Tree
Last Updated : 11 Oct, 2024

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

Left Rotation
Right Rotation
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



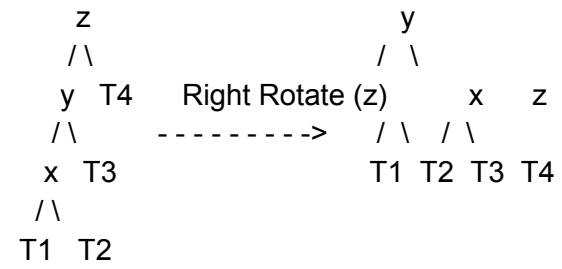
Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
Let w be the node to be deleted

Perform standard BST delete for w.
Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.
Following are the possible 4 arrangements:
y is left child of z and x is left child of y (Left Left Case)
y is left child of z and x is right child of y (Left Right Case)
y is right child of z and x is right child of y (Right Right Case)
y is right child of z and x is left child of y (Right Left Case)

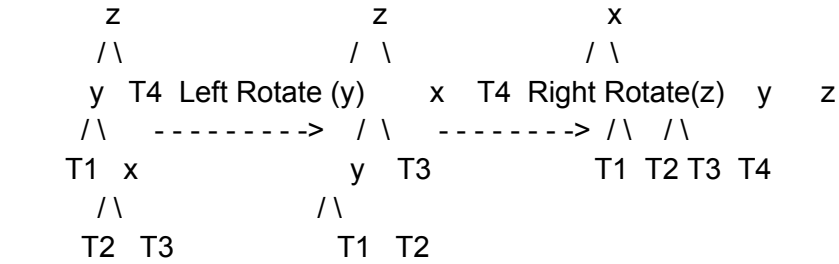
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See this video lecture for proof)

a) Left Left Case

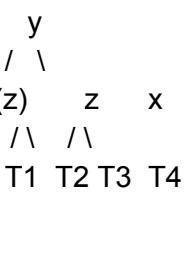
T1, T2, T3 and T4 are subtrees.



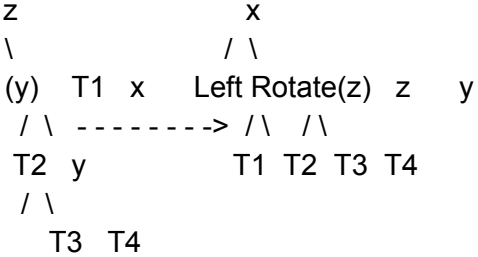
b) Left Right Case



c) Right Right Case



d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example:

avl-delete1avl-delete1

A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

Following is the implementation for AVL Tree Deletion. The following implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer (or reference) to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

Perform the normal BST deletion.
The current node must be one of the ancestors of the deleted node. Update the height of the current node.
Get the balance factor (left subtree height – right subtree height) of the current node.
If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.
Try it on GfG Practice
redirect icon

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

def height(N):
```

```
    if N is None:
        return 0
    return N.height

def right_rotate(y):
    x = y.left
    T2 = x.right

    # Perform rotation
    x.right = y
    y.left = T2

    # Update heights
    y.height = max(height(y.left),
                   height(y.right)) + 1
    x.height = max(height(x.left),
                   height(x.right)) + 1

    # Return new root
    return x

def left_rotate(x):
    y = x.right
    T2 = y.left

    # Perform rotation
    y.left = x
    x.right = T2

    # Update heights
    x.height = max(height(x.left),
                   height(x.right)) + 1
    y.height = max(height(y.left),
                   height(y.right)) + 1

    # Return new root
    return y

def get_balance(N):
    if N is None:
        return 0
    return height(N.left) - height(N.right)

def insert(node, key):
```

```
    # 1. Perform the normal BST insertion
    if node is None:
        return Node(key)

    if key < node.key:
        node.left = insert(node.left, key)
    elif key > node.key:
        node.right = insert(node.right, key)
    else:  # Duplicate keys not allowed
        return node

    # 2. Update height of this ancestor node
    node.height = max(height(node.left),
                      height(node.right)) + 1

    # 3. Get the balance factor of this node
    # to check whether this node became
    # unbalanced
    balance = get_balance(node)

    # If this node becomes unbalanced, then
    # there are 4 cases

    # Left Left Case
    if balance > 1 and key < node.left.key:
        return right_rotate(node)

    # Right Right Case
    if balance < -1 and key > node.right.key:
        return left_rotate(node)

    # Left Right Case
    if balance > 1 and key > node.left.key:
        node.left = left_rotate(node.left)
        return right_rotate(node)

    # Right Left Case
    if balance < -1 and key < node.right.key:
        node.right = right_rotate(node.right)
        return left_rotate(node)

    return node

def min_value_node(node):
```

```
    current = node

    # loop down to find the leftmost leaf
    while current.left is not None:
        current = current.left

    return current

def delete_node(root, key):
    # STEP 1: PERFORM STANDARD BST DELETE
    if root is None:
        return root

    # If the key to be deleted is smaller
    # than the root's key, then it lies in
    # left subtree
    if key < root.key:
        root.left = delete_node(root.left, key)

    # If the key to be deleted is greater
    # than the root's key, then it lies in
    # right subtree
    elif key > root.key:
        root.right = delete_node(root.right, key)

    # if key is same as root's key, then
    # this is the node to be deleted
    else:
        # node with only one child or no child
        if root.left is None or root.right is None:
            temp = root.left if root.left else root.right

            # No child case
            if temp is None:
                root = None
            else:  # One child case
                root = temp

        else:
            # node with two children: Get the
            # inorder successor (smallest in
            # the right subtree)
            temp = min_value_node(root.right)
```

```
            # Copy the inorder successor's
            # data to this node
            root.key = temp.key

            # Delete the inorder successor
            root.right = delete_node(root.right, temp.key)

    # If the tree had only one node then return
    if root is None:
        return root

    # STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root.height = max(height(root.left),
                      height(root.right)) + 1

    # STEP 3: GET THE BALANCE FACTOR OF THIS
    # NODE (to check whether this node
    # became unbalanced)
    balance = get_balance(root)

    # If this node becomes unbalanced, then
    # there are 4 cases

    # Left Left Case
    if balance > 1 and get_balance(root.left) >= 0:
        return right_rotate(root)

    # Left Right Case
    if balance > 1 and get_balance(root.left) < 0:
        root.left = left_rotate(root.left)
        return right_rotate(root)

    # Right Right Case
    if balance < -1 and get_balance(root.right) <= 0:
        return left_rotate(root)

    # Right Left Case
    if balance < -1 and get_balance(root.right) > 0:
        root.right = right_rotate(root.right)
        return left_rotate(root)

    return root

def pre_order(root):
```

```
    if root is not None:
        print("{0} ".format(root.key), end="")
        pre_order(root.left)
        pre_order(root.right)

# Driver Code
if __name__ == "__main__":
    root = None

    # Constructing tree given in the
    # above figure
    root = insert(root, 9)
    root = insert(root, 5)
    root = insert(root, 10)
    root = insert(root, 0)
    root = insert(root, 6)
    root = insert(root, 11)
    root = insert(root, -1)
    root = insert(root, 1)
    root = insert(root, 2)

    print("Preorder traversal of the "
          "constructed AVL tree is")
    pre_order(root)

    root = delete_node(root, 10)

    print("\nPreorder traversal after"
          " deletion of 10")
    pre_order(root)
```

Output
Preorder traversal of the constructed AVL tree is
5 1 0 -1 2 9 6 10 11
Preorder traversal after deletion of 10
5 1 0 -1 2 9 6 11
Output:

Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take

constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(log n). So time complexity of AVL delete is O(log n).
Auxiliary Space: O(log n) for recursion call stack as we have written a recursive method to delete

Summary of AVL Trees

These are self-balancing binary search trees.
Balancing Factor ranges -1, 0, and +1.
When balancing factor goes beyond the range require rotations to be performed
Insert, delete, and search time is O(log N).
AVL tree are mostly used where search is more frequent compared to insert and delete operation.

Introduction
Documents stored in a MongoDB database can vary widely. Some might be relatively small and contain only a few entries, like items in a shopping list. Others might be highly complex, containing dozens of fields of different types, arrays holding multiple values, and even other documents nested within the larger structure.

Regardless of how complex your documents are or how many you have, most often you won't need to review the data in all of them at once. Instead, you'll more likely want to only retrieve documents that satisfy one or more particular conditions. Similar to how you would find your holiday destination by selecting a range of filters on a booking website, such as distance from the seaside, pet-friendliness, a pool, and nearby parking, you can precisely query MongoDB to find exactly the documents you need. MongoDB provides a robust query mechanism for defining filtering criteria when retrieving documents.

In this tutorial, you'll learn how to query MongoDB collections using a different range of filters and conditions. You will also learn what cursors are and how to use them within the MongoDB shell.

Prerequisites
To follow this tutorial, you will need:

A server with a regular, non-root user with sudo privileges and a firewall configured with UFW. This tutorial was validated using a server running Ubuntu 20.04, and you can prepare your server by following this initial server setup tutorial for Ubuntu 20.04.
MongoDB installed on your server. To set this up, follow our tutorial on How to Install MongoDB on Ubuntu 20.04.

Your server's MongoDB instance secured by enabling authentication and creating an administrative user. To secure MongoDB like this, follow our tutorial on How To Secure MongoDB on Ubuntu 20.04.
Familiarity with MongoDB CRUD operations and retrieving objects from collections in particular. To learn how to use MongoDB shell to perform CRUD operations, follow the tutorial How To Perform CRUD operations in MongoDB.
Note: The linked tutorials on how to configure your server, install, and then secure MongoDB installation refer to Ubuntu 20.04. This tutorial concentrates on MongoDB itself, not the underlying operating system. It will generally work with any MongoDB installation regardless of the operating system as long as authentication has been enabled.

Step 1 — Preparing the Sample Database

To explain how to create queries in MongoDB — including how to filter documents with multiple fields, nested documents, and arrays — this guide uses an example database containing a collection of documents that describe the five highest mountains in the world.

To create this sample collection, connect to the MongoDB shell as your administrative user. This tutorial follows the conventions of the prerequisite MongoDB security tutorial and assumes the name of this administrative user is AdminSammy and its authentication database is admin. Be sure to change these details in the following command to reflect your own setup, if different:

```
mongo -u AdminSammy -p --authenticationDatabase admin
```

When prompted, enter the password you set when you created your administrative user. After providing the password, your prompt will change to a greater-than (>) sign:

Note: On a fresh connection, the MongoDB shell will automatically connect to the test database by default. You can safely use this database to experiment with MongoDB and the MongoDB shell.

Alternatively, you could also switch to another database to run all of the example commands given in this tutorial. To switch to another database, run the use command followed by the name of your database:

```
use database_name
```

To understand how MongoDB filters documents with multiple fields, nested documents and arrays, you'll need sample data complex enough to allow exploring different types of queries. As mentioned previously, this guide uses a sample collection of the five highest mountains in the world.

The documents in this collection will follow this format. This example document describes Mount Everest:

Mount Everest document

```
{
   "name": "Everest",
   "height": 8848,
   "location": ["Nepal", "China"],
   "ascents": {
      "first": {
         "year": 1953,
      },
      "first_winter": {
         "year": 1980,
      },
      "total": 5656,
   }
}
```

This document contains the following fields and values:

name: the peak's name
height: the peak's elevation, in meters
location: the countries in which the mountain is located. This field stores values as an array to allow for mountains located in more than one country
ascents: this field's value is another document. When one document is stored within another document like this, it's known as an embedded or nested document. Each ascents document describes successful ascents of the given mountain. Specifically, each ascents document contains a total field that lists the total number of successful ascents of each given peak. Additionally, each of these nested documents contain two fields whose values are also nested documents:
first: this field's value is a nested document that contains one field, year, which describes the year of the first overall successful ascent
first_winter: this field's value is a nested document that also contains a year field, the value of which represents the year of the first successful winter ascent of the given mountain
The reason why the first ascents are represented as nested documents even though only the year is included now is to make it easier to expand the ascent details with more fields in the future, such as the summiters' names or the expedition details.

Run the following insertMany() method in the MongoDB shell to simultaneously create a collection named peaks and insert five sample documents into it. These documents describe the five tallest mountain peaks in the world:

```
db.peaks.insertMany([
   {
      "name": "Everest",
      "height": 8848,
      "location": ["Nepal", "China"],
      "ascents": {
```

```
        "first": {
            "year": 1953
        },
        "first_winter": {
            "year": 1980
        },
        "total": 5656
    }
},
{
    "name": "K2",
    "height": 8611,
    "location": ["Pakistan", "China"],
    "ascents": {
        "first": {
            "year": 1954
        },
        "first_winter": {
            "year": 1921
        },
        "total": 306
    }
},
{
    "name": "Kangchenjunga",
    "height": 8586,
    "location": ["Nepal", "India"],
    "ascents": {
        "first": {
            "year": 1955
        },
        "first_winter": {
            "year": 1986
        },
        "total": 283
    }
},
{
    "name": "Lhotse",
    "height": 8516,
    "location": ["Nepal", "China"],
    "ascents": {
        "first": {
            "year": 1956
```

```
            },
            "first_winter": {
               "year": 1988
            },
            "total": 461
         }
      },
      {
         "name": "Makalu",
         "height": 8485,
         "location": ["China", "Nepal"],
         "ascents": {
            "first": {
               "year": 1955
            },
            "first_winter": {
               "year": 2009
            },
            "total": 361
         }
      }
])
```

The output will contain a list of object identifiers assigned to the newly-inserted objects.

Output

```
{
      "acknowledged" : true,
      "insertedIds" : [
            ObjectId("610c23828a94efbbf0cf6004"),
            ObjectId("610c23828a94efbbf0cf6005"),
            ObjectId("610c23828a94efbbf0cf6006"),
            ObjectId("610c23828a94efbbf0cf6007"),
            ObjectId("610c23828a94efbbf0cf6008")
      ]
}
```

You can verify that the documents were properly inserted by running the find() method with no arguments, which will retrieve all the documents you just added:

```
db.peaks.find()
```

Output

```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
```

. . .
With that, you have successfully created the list of example documents of mountains that will serve as the test data for creating queries. Next, you'll learn how to query with conditions referring to individual fields.

Step 2 — Querying Individual Fields

At the end of the previous step, you used MongoDB's find() method to return every document from the peaks collection. A query like this won't be very useful in practice, though, as it doesn't filter any documents and always returns the same result set.

You can filter query results in MongoDB by defining a specific condition that documents must adhere to in order to be included in a result set. If you have followed the How To Perform CRUD operations in MongoDB tutorial, you have already used the most basic filtering condition: the equality condition.

As an example, run the following query which returns any documents whose name value is equal to Everest:

```
db.peaks.find(
    { "name": "Everest" }
)
```

The second line — { "name": "Everest" } — is the query filter document, a JSON object specifying the filters to apply when searching the collection in order to find documents that satisfy the condition. This example operation tells MongoDB to retrieve any documents in the peaks collection whose name value matches the string Everest:

Output

```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
```

MongoDB returned a single document, as there is only one Mt. Everest in the peaks collection.

The equality condition specifies a single value that MongoDB will attempt to match against documents in the collection. MongoDB provides comparison query operators that allow you to specify other conditions that also refer to a single field, but filter documents in ways that are more complex than searching for exact matches.

A comparison operator consists of the operator itself, a single key preceded by a dollar sign ($), and the value the query operator will use to filter documents.

To illustrate, run the following query which searches for any documents whose name value does not equal Everest:

```
db.peaks.find(
```

```
    { "name": { $ne: "Everest" } }
)
```
This time, the query filter document includes { $ne: "Everest" }. $ne is the comparison operator in this example, and it stands for “not equal”. The peak name, Everest, appears again as the value for this operator. Because this query is searching for documents whose name value is not equal to Everest, it returns four documents:

Output
```
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2", "height" : 8611, "location" : [ "Pakistan", "China" ], "ascents" : { "first" : { "year" : 1954 }, "first_winter" : { "year" : 1921 }, "total" : 306 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6006"), "name" : "Kangchenjunga", "height" : 8586, "location" : [ "Nepal", "India" ], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 1986 }, "total" : 283 } }
. . .
```
The $in operator allows you to write queries that will return documents with values matching one of multiple values held in an array.

The following example query includes the $in operator, and will return documents whose name value matches either Everest or K2:

```
db.peaks.find(
    { "name": { $in: ["Everest", "K2"] } }
)
```
Instead of a single value, the value passed to the $in operator is an array of two peak names in square braces. MongoDB returns two documents, just as expected:

Output
```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2", "height" : 8611, "location" : [ "Pakistan", "China" ], "ascents" : { "first" : { "year" : 1954 }, "first_winter" : { "year" : 1921 }, "total" : 306 } }
```
The examples so far have queried the name field with text values. You can also filter documents based on numerical values.

The following example query searches for documents whose height value is greater than 8500:

```
db.peaks.find(
    { "height": { $gt: 8500 } }
)
```
This query includes the $gt operator, which stands for greater than. By passing it the value 8500, MongoDB will return documents whose height value is greater than 8500:

Output
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2", "height" : 8611, "location" : ["Pakistan", "China"], "ascents" : { "first" : { "year" : 1954 }, "first_winter" : { "year" : 1921 }, "total" : 306 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6006"), "name" : "Kangchenjunga", "height" : 8586, "location" : ["Nepal", "India"], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 1986 }, "total" : 283 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse", "height" : 8516, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1956 }, "first_winter" : { "year" : 1988 }, "total" : 461 } }
MongoDB offers a number of comparison query operators in addition to the ones outlined in this section. For a full list of these operators, see the official documentation on the subject.

Now that you know how to use equality conditions and comparison operators on a single document field, you can move onto learning how to join multiple conditions together in a single query.

Step 3 — Using Multiple Conditions

Sometimes, filtering based on a single document field is not enough to precisely select documents of interest. In such cases, you might want to filter documents using multiple conditions at once.

There are two ways to connect multiple conditions in MongoDB. The first is to use a logical AND conjunction to select documents in the collection matching all the conditions, or the logical OR to select documents matching at least one condition from the list.

In MongoDB, the AND conjunction is implicit when using more than one field in the query filter document. Try selecting a mountain that matches the name Everest and the exact height of 8848 meters:

```
db.peaks.find(
   { "name": "Everest", "height": 8848 }
)
```

Notice that the syntax is similar to the equality condition example from the previous step, but this time two fields appear in the query filter document. MongoDB checks for equality on both fields and requires both to match the requested values in order for a document to be selected:

Output

{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
In this case, a single document is returned, but if you try changing the height to any other numerical value the result set will be empty since any returned documents must match both conditions. For instance, the following example will not return any output to the shell:

```
db.peaks.find(
   { "name": "Everest", "height": 9000 }
)
```

This implicit AND can be made explicit by including the $and logical query operator followed by a list of conditions that returned documents must satisfy. The following example is essentially the same query as the previous one, but includes the $and operator instead of an implicit AND conjunction:

```
db.peaks.find(
   { $and: [{"name": "Everest"}, {"height": 8848}] }
)
```

This time the JSON object containing the $and query operator is the query filter document itself. Here, the comparison operator takes two separate equality conditions that appear in the list, one for name matches and the latter for height matches.

In order to select documents matching any of the chosen conditions rather than all of them, you can instead use the $or operator:

```
db.peaks.find(
   { $or: [{"name": "Everest"}, {"name": "K2"}] }
)
```

When using the $or operator, a document only needs to satisfy one of the two the equality filters:

Output
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2", "height" : 8611, "location" : ["Pakistan", "China"], "ascents" : { "first" : { "year" : 1954 }, "first_winter" : { "year" : 1921 }, "total" : 306 } }
Although each of this example’s conditions are single-field equality conditions, both the $and and $or operators can contain any valid query filter documents. They can even include nested AND/OR condition lists.

Joining multiple filters together using $and and $or operators as outlined in this step can be very helpful with retrieving fine-grained query results. However, the examples so far have all used

query filter documents that filter based on individual values. The next step outlines how to query against values stored in an array field.

Step 4 — Querying for Array Values
Sometimes a single field may contain multiple values stored in an array. In our example with mountain peaks, location is such a field. Because mountains often span more than one country, like Kangchenjunga in Nepal and India, a single value may not always be enough for this field.

In this step, you'll learn how to construct query filters that match items in array fields.

Let's start by trying to select documents representing mountains that are in Nepal. For this example, though, it's okay if the mountain has multiple locations listed, as long as one of them is Nepal:

```
db.peaks.find(
    { "location": "Nepal" }
)
```
This query uses an equality condition that tells MongoDB to return documents whose location value exactly matches the given string value, Nepal, similar to the previous examples that used the name field. MongoDB will select any documents in which the requested value appears in any place in the arrays:

Output
```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6006"), "name" : "Kangchenjunga", "height" : 8586, "location" : [ "Nepal", "India" ], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 1986 }, "total" : 283 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse", "height" : 8516, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1956 }, "first_winter" : { "year" : 1988 }, "total" : 461 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu", "height" : 8485, "location" : [ "China", "Nepal" ], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 2009 }, "total" : 361 } }
```
For this query, MongoDB returned the four documents in which Nepal appears in the location field.

However, what if you wanted to find mountains located in both China and Nepal? To do this, you could include an array in the filter document, rather than a single value:

```
db.peaks.find(
    { "location": ["China", "Nepal"] }
)
```

Even though there are four mountains in Nepal and China in the database, there is only one in which the countries are listed in the order given in this query, so this query returns a single document:

Output
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu", "height" : 8485, "location" : ["China", "Nepal"], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 2009 }, "total" : 361 } }
Notice that the value of the location field for Makalu is identical to the query's filter document. When you supply an array as the value for the equality condition like this, MongoDB will retrieve documents where the location field matches the query filter exactly, including the order of elements inside the array. To illustrate, run the query again but swap China with Nepal:

```
db.peaks.find(
    { "location": ["Nepal", "China"] }
)
```
Output
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse", "height" : 8516, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1956 }, "first_winter" : { "year" : 1988 }, "total" : 461 } }
Now, two other mountains are returned, but Makalu is not.

Using the equality condition like this is not helpful in cases where you care only about elements in an array (regardless of their order) rather than an exact match. Fortunately, MongoDB allows you to retrieve documents containing more than one array element anywhere in an array using the $all query operator.

To illustrate, run the following query:

```
db.peaks.find(
    { "location": { $all: ["China", "Nepal"] } }
)
```
The $all operator will ensure that documents will be checked whether their location array contains both China and Nepal inside in any order. MongoDB will return all three mountains in a single query:

Output
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }

{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse", "height" : 8516, "location" : ["Nepal", "China"], "ascents" : { "first" : { "year" : 1956 }, "first_winter" : { "year" : 1988 }, "total" : 461 } }
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu", "height" : 8485, "location" : ["China", "Nepal"], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 2009 }, "total" : 361 } }
This step outlined how to use arrays in query filter documents to retrieve documents with more than one value in a single field. If you want to query data held within a nested document, you'll need to use the special syntax required for such an operation. Continue onto the next step to learn how to do this.

Step 5 — Querying Fields in Nested Documents
Recall that the example database documents include an ascent field that holds various details about each mountain's first ascents an array. This way, the data about the first ascent, the winter ascent, and the total number of ascents is cleanly grouped inside a single nested document. This step explains how you can access fields within a nested document when building queries.

Review the sample Everest document once more:

The Everest document
```
{
   "name": "Everest",
   "height": 8848,
   "location": ["Nepal", "China"],
   "ascents": {
      "first": {
         "year": 1953,
      },
      "first_winter": {
         "year": 1980,
      },
      "total": 5656,
   }
}
```
Accessing the name and height fields was straightforward, as a single value resides under these keys. But say you wanted to find the total number of ascents for a given peak. The ascents field contains more data than just the total number of ascents inside. There is a total field, but it's not part of the main document, so there's no way to access it directly.

To solve this issue, MongoDB provides a dot notation to access fields in nested documents.

To illustrate how MongoDB's dot notation works, run the following query. This will return all the mountains in the collection that have been ascended more than 1000 times, using the $gt operator highlighted previously:

```
db.peaks.find(
   { "ascents.total": { $gt: 1000 } }
)
```

Mt. Everest is the only mountain in the collection with more than 1000 ascents, so only its document will be returned:

Output

```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first_winter" : { "year" : 1980 }, "total" : 5656 } }
```

While the { $gt: 1000 } query filter with $gt operator is familiar, notice how this query accesses the total field held within the document stored in the ascents field. In nested documents, the access path to any given field is constructed with dots indicating the action of going inside the nested object.

So, ascents.total means that MongoDB should first open the nested document that the ascents field points to and then find the total field within it.

The notation works with multiple nested documents as well:

```
db.peaks.find(
   { "ascents.first_winter.year": { $gt: 2000 } }
)
```

This query will return any documents describing mountains that were first ascended in winter only after the year 2000:

Output

```
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu", "height" : 8485, "location" : [ "China", "Nepal" ], "ascents" : { "first" : { "year" : 1955 }, "first_winter" : { "year" : 2009 }, "total" : 361 } }
```

As before, the ascents.first_winter.year notation means MongoDB first finds the ascents field and finds the nested documents there. It then goes into another nested document, first_winter, and finally retrieves the year field from within it.

The dot notation can be used to access any depth of nested documents in MongoDB.

By now you will have a good understanding of how to access data from nested documents and how to filter query results. You can move on to learning how to limit the list of fields returned by your queries.

Step 6 — Returning a Subset of Fields

In all the examples so far, whenever you queried the peaks collection, MongoDB returned one or more full documents. Oftentimes, you'll only need information from a handful of fields. As an example, you might only want to find the names of the mountains in the database.

This isn't just a matter of legibility, but also of performance. If only a small part of a document is needed, retrieving whole document objects would be an unnecessary performance burden on the database. This may not be a problem when working with small datasets like this tutorial's examples, but it becomes an important consideration when working with many large, complex documents.

As an example, say you're only interested in mountain names stored in the peaks collection, but the ascent details or location are not important this time. You could limit the fields your query will return by following the query filter document with a projection.

A projection document is a JSON object where keys correspond to the fields of the queried documents. Projections can be either constructed as inclusion projections or exclusion projections. When the projection document contains keys with 1 as their values, it describes the list of fields that will be included in the result. If, on the other hand, projection keys are set to 0, the projection document describes the list of fields that will be excluded from the result.

Run the following query, which includes the by-now familiar find() method. This query's find() method includes two arguments, instead of one. The first, {}, is the query filter document. Here it's an empty JSON object, meaning it won't apply any filtering. The second argument, { "name": 1 }, describes the projection and means that the query results will only include each document's name field:

```
db.peaks.find(
   {},
   { "name": 1 }
)
```

After running this example query, MongoDB returns the following results:

Output

```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest" }
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2" }
{ "_id" : ObjectId("610c23828a94efbbf0cf6006"), "name" : "Kangchenjunga" }
{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse" }
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu" }
```

Notice that the returned documents are simplified, and contain only the name and _id fields. MongoDB always includes the _id key, even if it's not explicitly requested.

To illustrate how to specify what fields to exclude, run the following query. It will return data from each document, but will exclude the ascents and location fields:

```
db.peaks.find(
   {},
   { "ascents": 0, "location": 0 }
)
```
MongoDB returns all five mountains once again, but this time only the name, height, and _id fields are present:

Output
```
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848 }
{ "_id" : ObjectId("610c23828a94efbbf0cf6005"), "name" : "K2", "height" : 8611 }
{ "_id" : ObjectId("610c23828a94efbbf0cf6006"), "name" : "Kangchenjunga", "height" : 8586 }
{ "_id" : ObjectId("610c23828a94efbbf0cf6007"), "name" : "Lhotse", "height" : 8516 }
{ "_id" : ObjectId("610c23828a94efbbf0cf6008"), "name" : "Makalu", "height" : 8485 }
```
Note: When specifying projections, you cannot mix inclusions and exclusions. You either have to specify the list of fields to include, or a list of fields to exclude.

There is, however, one exception to this rule. MongoDB allows you to exclude the _id field from a result set even when the query has an inclusion projection applied. To suppress the _id field, you can append "_id": 0 to the projection document. The following example is similar to the previous example query, but will exclude every field, including _id, except for the name field:

```
db.peaks.find(
   {},
   { "_id": 0, "name": 1 }
)
```
Output
```
{ "name" : "Everest" }
{ "name" : "K2" }
{ "name" : "Kangchenjunga" }
{ "name" : "Lhotse" }
{ "name" : "Makalu" }
```
Projections can also be used to include or exclude fields in nested documents. Say, for example, that you want to know each mountain’s first winter ascent and the total number of ascents, both of which are nested within the ascents field. Additionally, you want to return each mountain’s name. To do this, you could run a query like this:

```
db.peaks.find(
   {},
   { "_id": 0, "name": 1, "ascents": { "first_winter": 1, "total": 1 } }
)
```
Notice how the projection is specified for the ascents fields and how it follows the structure of the nested document, being a nested projection itself. By using "first_winter": 1, "total": 1 this query tells the database to include only these two fields from the nested document and no other.

The returned documents will contain only the requested fields:

Output
{ "name" : "Everest", "ascents" : { "first_winter" : { "year" : 1980 }, "total" : 5656 } }
{ "name" : "K2", "ascents" : { "first_winter" : { "year" : 1921 }, "total" : 306 } }
{ "name" : "Kangchenjunga", "ascents" : { "first_winter" : { "year" : 1988 }, "total" : 461 } }
{ "name" : "Makalu", "ascents" : { "first_winter" : { "year" : 2009 }, "total" : 361 } }
Limiting the size of returned documents to only a subset of fields can be helpful with making result sets more readable and can even improve performance. The next step outlines how to limit the number of documents returned by a query, and also details how to sort the data returned by a query.

Step 7 — Using Cursors to Sort and Limit Query Results

When retrieving objects from a large collection, there may be times when you want to limit the number of results or perhaps sort them in a particular order. For example, a popular approach for shopping sites is to sort products by their price. MongoDB uses cursors which allow you to limit the number of documents returned in a query result set and also sort the results in ascending or descending order.

Recall this example query from Step 1:

```
db.peaks.find()
```

You may recall that the result set returned by this query includes all the data from each document in the peaks collection. While it may seem like MongoDB returns all the objects from the peaks collection, this is not the case. What MongoDB actually returns is a cursor object.

A cursor is a pointer to the result set of a query but it is not the result set itself. It's an object that can be iterated, meaning that you can request the cursor to return the next document in line, and only then will the full document be retrieved from the database. Until that happens, the cursor only points to the next document on the list.

With cursors, MongoDB can ensure that the actual document retrieval happens only when it's needed. This can have significant performance implications when the documents in question are large or many of them are requested at once.

To illustrate how cursors work, run the following operation which includes both the find() and count() methods:

```
db.peaks.find().count()
```

MongoDB will respond with 5:

Output
```
5
```

Under the hood, the find() method finds and then returns a cursor, and then the count() method is called on that cursor. This lets MongoDB know that you're interested in the object count and not the documents themselves. This means that documents won't be a part of the results — all the database will return is the count. Using methods on the cursor object to further modify the query before retrieving documents from the cursor, you can ensure only the database operations that you ask for will be performed on the collection.

Note: When executing queries, the MongoDB shell automatically iterates over the returned cursors 20 times so as to display the first 20 results on the screen. This is specific to the MongoDB shell. When working with MongoDB programmatically, it won't immediately retrieve any results from a cursor.

Another MongoDB method that uses cursors to alter a result set is the limit() method. As its name implies, you can use limit() to limit the number of results a query will return.

Run the following query which will retrieve only three mountain peaks from the collection:

```
db.peaks.find(
   {},
   { "_id": 0, "name": 1, "height": 1 }
).limit(3)
```

MongoDB shell will respond with three objects rather than five, even though the query isn't filtering any data:

Output

```
{ "name" : "Everest", "height" : 8848 }
{ "name" : "K2", "height" : 8611 }
{ "name" : "Kangchenjunga", "height" : 8586 }
```

The limit(3) method applied on the cursor tells the cursor to stop returning further documents after reaching the first 3. Using the limit() cursor method like this with large collections will help to ensure that you only retrieve the results you need and no more.

By default, MongoDB will return objects in the order of their insertion, but you might want to alter that behavior. Say you're interested in finding the three lowest mountain peaks held in the database. You could run the following query:

```
db.peaks.find(
   {},
   { "_id": 0, "name": 1, "height": 1 }
).limit(3).sort({ "height": 1 })
```

The added sort({ "height": 1 }) causes the result set to differ from the previous example:

Output

```
{ "name" : "Makalu", "height" : 8485 }
```

{ "name" : "Lhotse", "height" : 8516 }
{ "name" : "Kangchenjunga", "height" : 8586 }
Again, only three mountain peaks are returned. However, this time they have been sorted ascending from the one with the lowest height value.

The sort() method on the cursor accepts a JSON object — height — as an argument, similar to the projection document. It also accepts the list of keys that will be used to sort against. The accepted value is either 1 for ascending or -1 for descending sort order for each key.