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

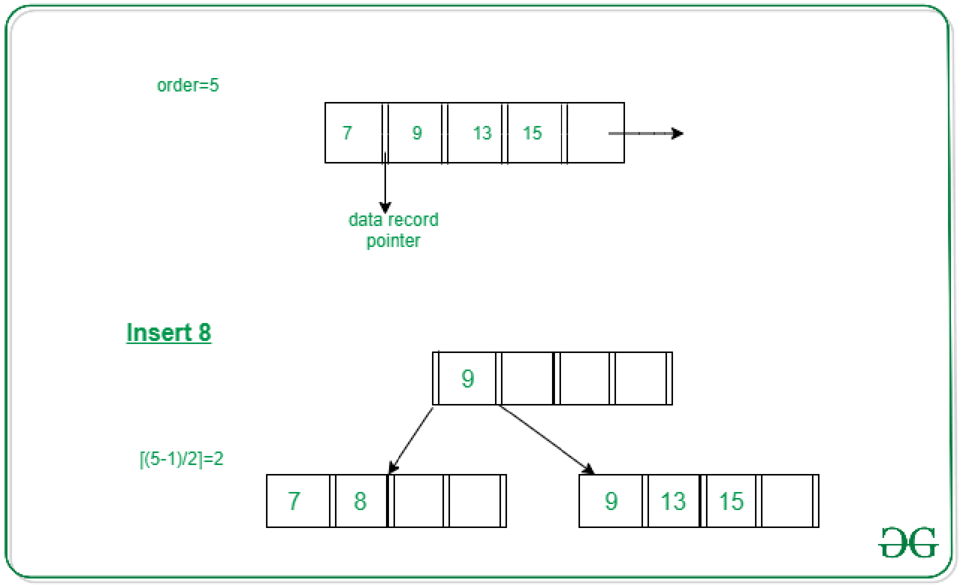
1. Every element is inserted into a leaf node. So, go to the appropriate leaf node.
2. Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow go ahead with the following steps mentioned below to deal with overflow while maintaining the B+ Tree properties.

Properties for insertion B+ Tree

Case 1: Overflow in leaf node

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

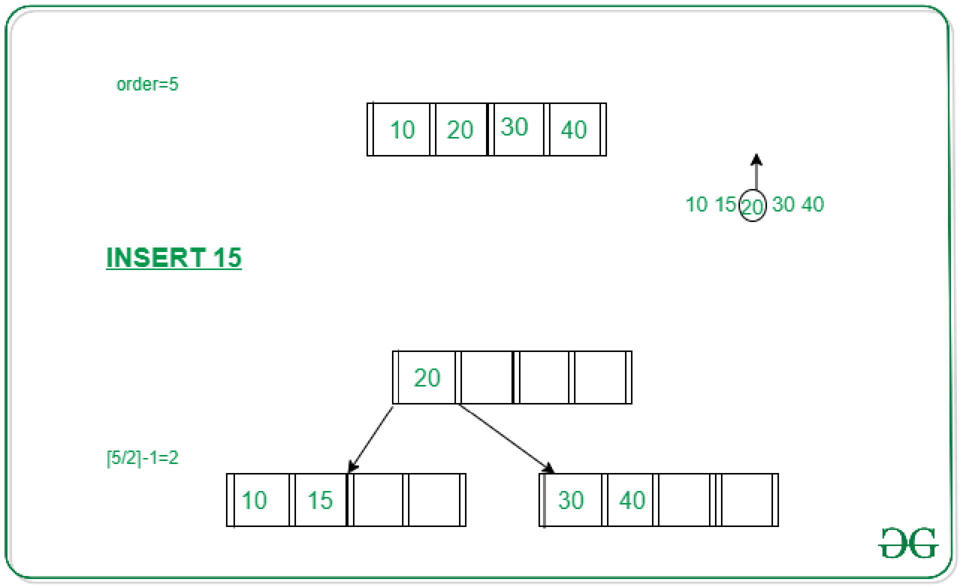
Below is the illustration of inserting 8 into B+ Tree of order of 5:



Case 2: Overflow in non-leaf node

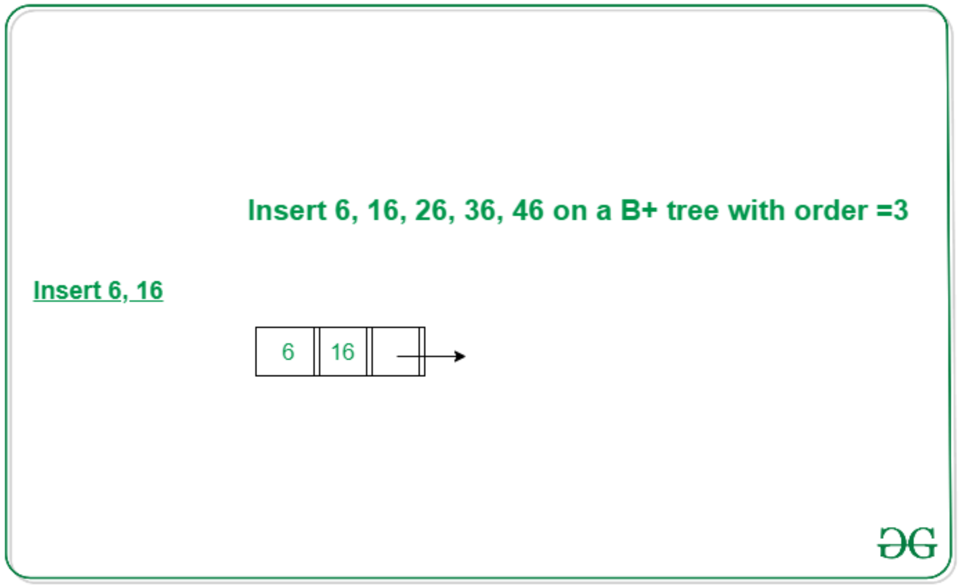
1. Split the non leaf node into two nodes.
2. First node contains ceil(m/2)-1 values.
3. Move the smallest among remaining to the parent.
4. Second node contains the remaining keys.

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

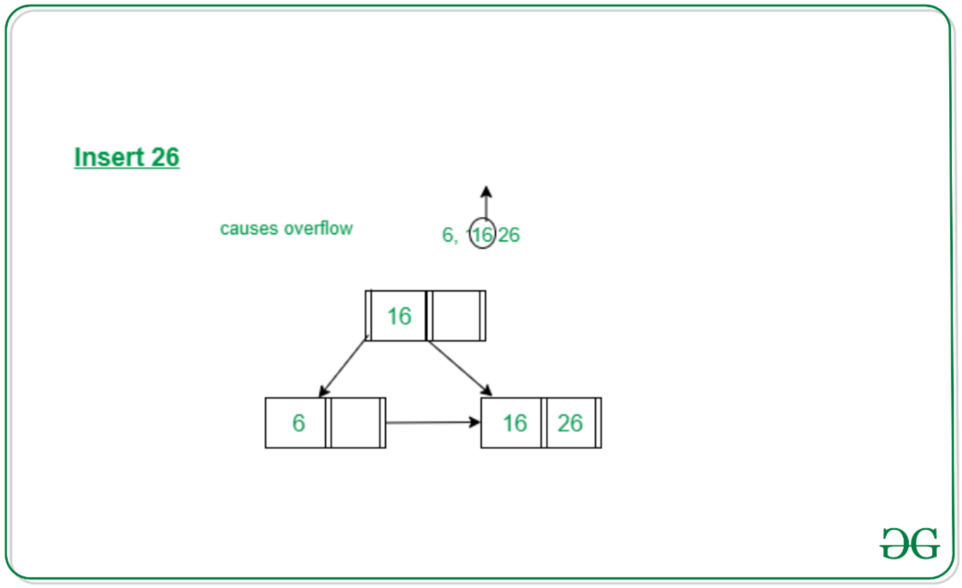


Example to illustrate insertion on a B+ tree

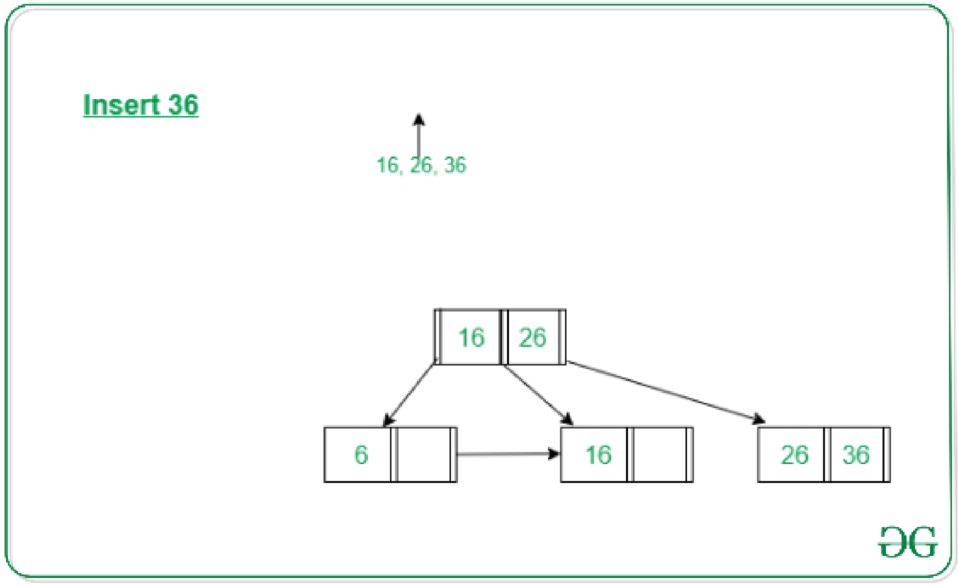
Problem: Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.
Solution:
Step 1: The order is 3 so at maximum in a node so there can be only 2 search key values. As insertion happens on a leaf node only in a B+ tree so insert search key value **6 and 16** in increasing order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



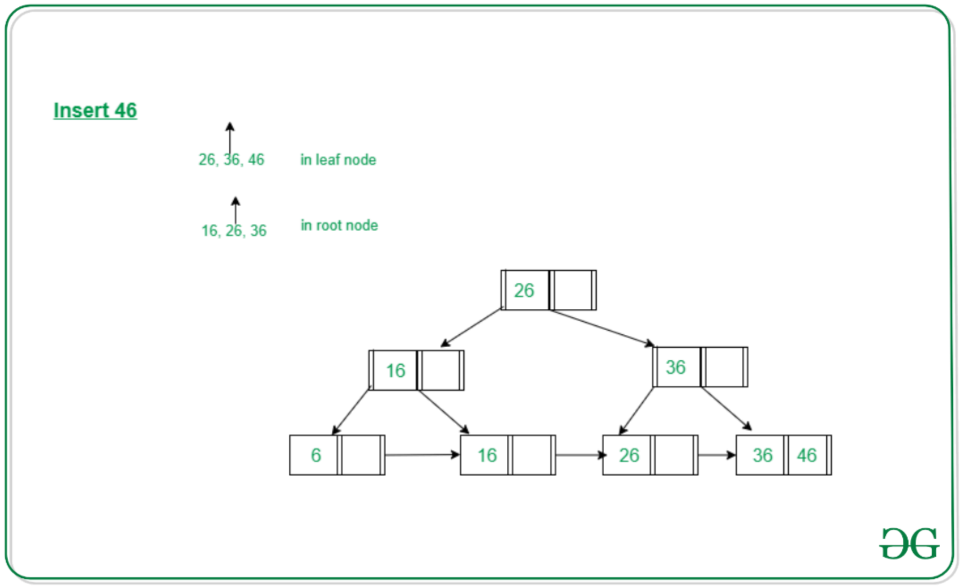
Step 2: We cannot insert **26** in the same node as it causes an overflow in the leaf node, We have to split the leaf node according to the rules. First part contains **ceil((3-1)/2)** values i.e., only **6**. The second node contains the remaining values i.e., **16** and **26**. Then also copy the smallest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



Below is the python implementation of B+ tree:

C++ Java Python3 C# JavaScript

```
# Python3 program for implementing B+ Tree

import math

# Node creation
class Node:
    def __init__(self, order):
        self.order = order
        self.values = []
        self.keys = []
        self.nextKey = None
        self.parent = None
        self.check_leaf = False

    # Insert at the leaf
    def insert_at_leaf(self, leaf, value, key):
        if (self.values):
            temp1 = self.values
            for i in range(len(temp1)):
                if (value == temp1[i]):
                    self.keys[i].append(key)
                    break
                elif (value < temp1[i]):
                    self.values = self.values[:i] + [value] + self.values[i:]
                    self.keys = self.keys[:i] + [[key]] + self.keys[i:]
                    break
                elif (i + 1 == len(temp1)):
                    self.values.append(value)
                    self.keys.append([key])
                    break
        else:
            self.values = [value]
            self.keys = [[key]]


# B plus tree
class BplusTree:
    def __init__(self, order):
        self.root = Node(order)
        self.root.check_leaf = True

    # Insert operation
    def insert(self, value, key):
        value = str(value)
        old_node = self.search(value)
        old_node.insert_at_leaf(old_node, value, key)

        if (len(old_node.values) == old_node.order):
            node1 = Node(old_node.order)
            node1.check_leaf = True
            node1.parent = old_node.parent
            mid = int(math.ceil(old_node.order / 2)) - 1
            node1.values = old_node.values[mid + 1:]
            node1.keys = old_node.keys[mid + 1:]
            node1.nextKey = old_node.nextKey
            old_node.values = old_node.values[:mid + 1]
            old_node.keys = old_node.keys[:mid + 1]
            old_node.nextKey = node1
            self.insert_in_parent(old_node, node1.values[0], node1)
    # Search operation for different operations
    def search(self, value):
        current_node = self.root
        while(current_node.check_leaf == False):
            temp2 = current_node.values
            for i in range(len(temp2)):
                if (value == temp2[i]):
                    current_node = current_node.keys[i + 1]
                    break
                elif (value < temp2[i]):
                    current_node = current_node.keys[i]
                    break
                elif (i + 1 == len(current_node.values)):
                    current_node = current_node.keys[i + 1]
                    break
        return current_node

    # Find the node
    def find(self, value, key):
        l = self.search(value)
        for i, item in enumerate(l.values):
            if item == value:
                if key in l.keys[i]:
                    return True
                else:
                    return False
        return False

    # Inserting at the parent
    def insert_in_parent(self, n, value, ndash):
        if (self.root == n):
            rootNode = Node(n.order)
            rootNode.values = [value]
            rootNode.keys = [n, ndash]
            self.root = rootNode
            n.parent = rootNode
            ndash.parent = rootNode
            return

        parentNode = n.parent
        temp3 = parentNode.keys
        for i in range(len(temp3)):
            if (temp3[i] == n):
                parentNode.values = parentNode.values[:i] + \
                    [value] + parentNode.values[i:]
                parentNode.keys = parentNode.keys[:i +
                                                 1] + [ndash] + parentNode.keys[i + 1:]
                if (len(parentNode.keys) > parentNode.order):
                    parentdash = Node(parentNode.order)
                    parentdash.parent = parentNode.parent
                    mid = int(math.ceil(parentNode.order / 2)) - 1
                    parentdash.values = parentNode.values[mid + 1:]
                    parentdash.keys = parentNode.keys[mid + 1:]
                    value_ = parentNode.values[mid]
                    if (mid == 0):
                        parentNode.values = parentNode.values[:mid + 1]
                    else:
                        parentNode.values = parentNode.values[:mid]
                    parentNode.keys = parentNode.keys[:mid + 1]
                    for j in parentNode.keys:
                        j.parent = parentNode
                    for j in parentdash.keys:
                        j.parent = parentdash
                    self.insert_in_parent(parentNode, value_, parentdash)

# Print the tree
def printTree(tree):
    lst = [tree.root]
    level = [0]
    leaf = None
    flag = 0
    lev_leaf = 0

    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):
        x = lst.pop(0)
        lev = level.pop(0)
        if (x.check_leaf == False):
            for i, item in enumerate(x.keys):
                print(item.values)
        else:
            for i, item in enumerate(x.keys):
                print(item.values)
                if (flag == 0):
                    lev_leaf = lev
                    leaf = x
                    flag = 1


record_len = 3
bplustree = BplusTree(record_len)
bplustree.insert('5', '33')
bplustree.insert('15', '21')
bplustree.insert('25', '31')
bplustree.insert('35', '41')
bplustree.insert('45', '10')

printTree(bplustree)

if(bplustree.find('5', '34')):
    print("Found")
else:
    print("Not found")
```

Output

```
['15']
['25']
['35']
['45']
['5']
Not found
```

Time complexity: O(log n)

Auxiliary Space: O(log n)