# Chapter 8.9.1 Order 3 B-Trees

### Introduction to B-trees

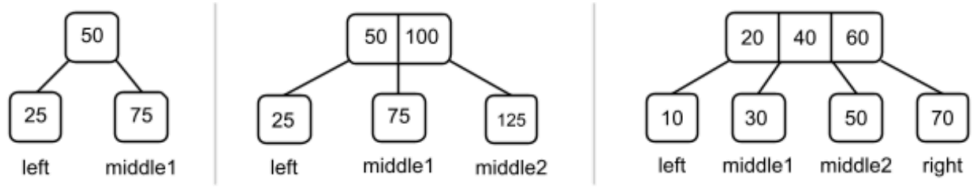
In a binary tree, each node has one key and up to two children. A ***B-tree*** with order K is a tree where nodes can have up to K-1 keys and up to K children. The ***order*** is the maximum number of children a node can have. Ex: In a B-tree with order 4, a node can have 1, 2, or 3 keys, and up to 4 children. B-trees have the following properties:

- All keys in a B-tree must be distinct.
- All leaf nodes must be at the same level.
- An internal node with N keys must have N+1 children.
- Keys in a node are stored in sorted order from smallest to largest.
- Each key in a B-tree internal node has one left subtree and one right subtree. All left subtree keys are < that key, and all right subtree keys are > that key.

## 2-3-4 Tree Node Labels

- The keys in a 2-3-4 tree node are labeled as A, B, and C
- The child nodes of a 2-3-4 tree internal node are labeled as left, middle1, middle2, and right.
- If a node contains 1 key, then keys B and C, as well as children middle2 and right, are not used.
- If a node contains 2 keys, then key C, as well as the right child, are not used.
- A 2-3-4 tree node containing exactly 3 keys is said to be full, and uses all keys and children

Figure 8.9.1: 2-3-4 child subtree labels.

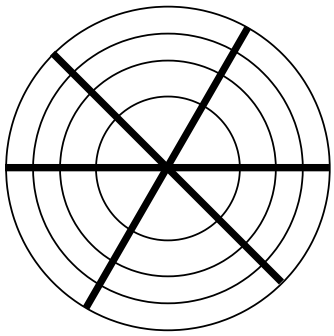



## 2-3-4 Tree Search Algorithm

- Given a key, a search algorithm returns the first node found matching that key, or returns null if a matching node is not found.
- Searching a 2-3-4 tree is a recursive process that starts with the root node.
- If the search key equals any of the keys in the node, then the node is returned.
- Otherwise, a recursive call is made on the appropriate child node.
- Which child node to use depends on the value of the search key in comparison to the node's keys.

| Condition | Child node to search |
|---|---|
| key < node's A key | left |
| node has only 1 key or key < node's B key | middle1 |
| node has only 2 keys or key < node's C key | middle2 |
| none of the above | right |

```
BTreeSearch(node, key) {
   if (node is not null) {
      if (node has key) {
         return node
      }
      if (key < node -->A) {
         return BTreeSearch(node -->left, key)
      }
      else if (node -->B is null || key < node -->B) {
         return BTreeSearch(node -->middle1, key)
      }
      else if (node -->C is null || key < node -->C) {
         return BTreeSearch(node -->middle2, key)
      }
      else {
         return BTreeSearch(node -->right, key)
      }
   }
   return null
}
```

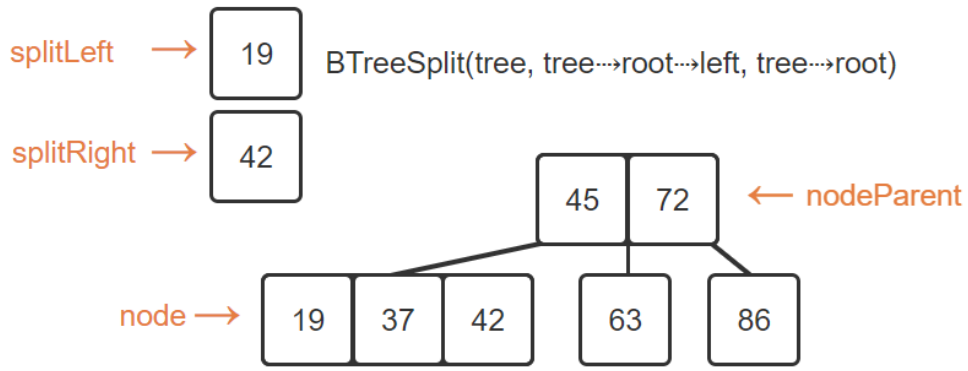


```
//BTreeSplit
BTreeSplit(tree, node, nodeParent){

    if (node is not null){
        return null;
    }

    splitLeft = new BTreeNode(node-->A, node-->left, node-->middle);
    splitRight = new BTreeNode(node-->C, node-->middle2, node-->right);

    if ( nodeParent is not null){
        BTreeInsertKeyWithChildren(nodeParent, node-->B, splitLeft, splitRight);

    }
    else {
        nodeParent = new BTreeNode(node-->B, splitLeft, splitRight);
        tree-->root = nodeParent;
    }
    return nodeParent;

}
```

45 72

37

```
BTreeSplit(tree, node, nodeParent) {
    if (node is not full) {
        return null
    }

    splitLeft = new BTreeNode(node⇢A, node⇢left, node⇢middle1)
    splitRight = new BTreeNode(node⇢C, node⇢middle2, node⇢right)
    if (nodeParent is not null) {
        BTreeInsertKeyWithChildren(nodeParent, node⇢B, splitLeft, splitRight)
    }
    else {
        nodeParent = new BTreeNode(node⇢B, splitLeft, splitRight)
        tree⇢root = nodeParent
    }
    return nodeParent
}
```



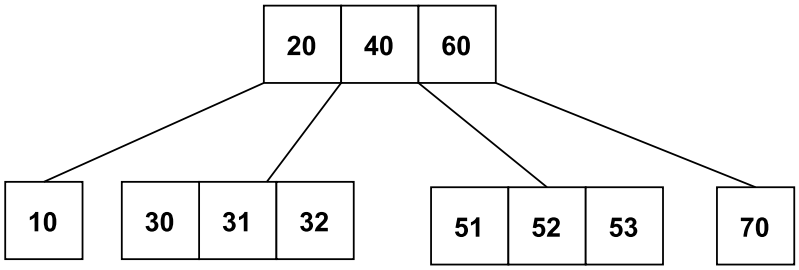

Since nodeParent is not null, the key 37 moves from node into nodeParent and the two newly allocated children are attached to nodeParent as well.

```
//BTreeInsertKeyWithChildren
BTreeInsertKeyWithChildren (parent, key, leftChild, rightChild){

    if( key < parent-->A){
        parent--> C = parent-->B;
        parent--> B = parent-->A;
        parent--> A = key;
        parent--> right = parent -->middle2;
        parent--> middle2 = parent-->middle1;
        parent--> middle1 = rightChild;
        parent--> left = leftChild;
    }
    else if (parent-->B is null || key < parent--> B){
        parent--> C = parent --> B;
        parent--> B = key;
        parent--> right = parent--> middle2;
        parent--> middle2 = rightChild;
        parent--> middle1 = leftChild;
    }
    else{
        parent--> C = key;
        parent--> right = rightChild;
        parent--> middle2 = leftChild;
    }
}
```

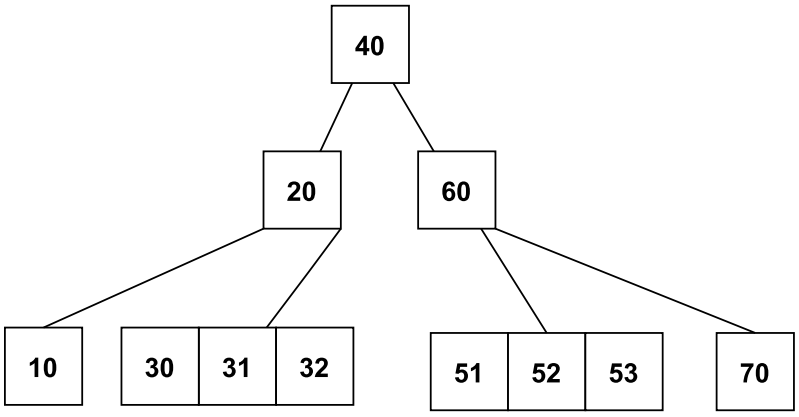
# Inserting a key into a leaf node

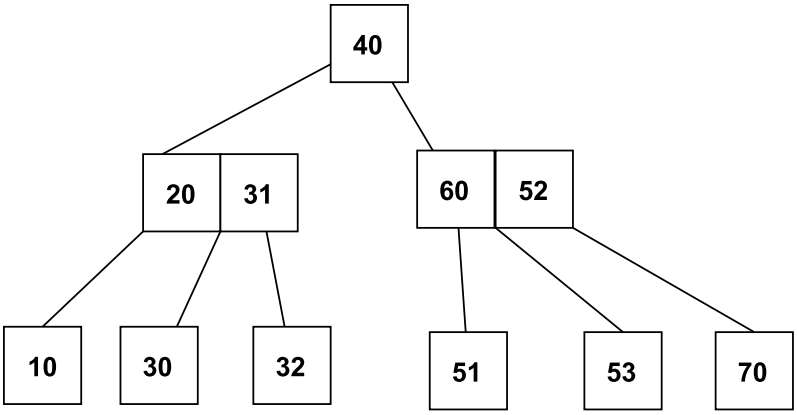
- A new key is always inserted into a non-full leaf node.
- The table below describes the 4 possible cases for inserting a new key into a non-full leaf node.

| Condition | Outcome |
|---|---|
| New key equals an existing key in node | No insertion takes place, and the node is not altered. |
| New key is < node's first key | Existing keys in node are shifted right, and the new key becomes node's first key. |
| Node has only 1 key or new key is < node's middle key | Node's middle key , if present, becomes last key, and new key becomes node's middle key. |
| None of the above | New key becomes node's last key. |


20
40
60
10
30
31
32
51
52
53
70

38


40
20
60
10
30
31
32
51
52
53
70


40
20
31
60
52
10
30
32
51
53
70

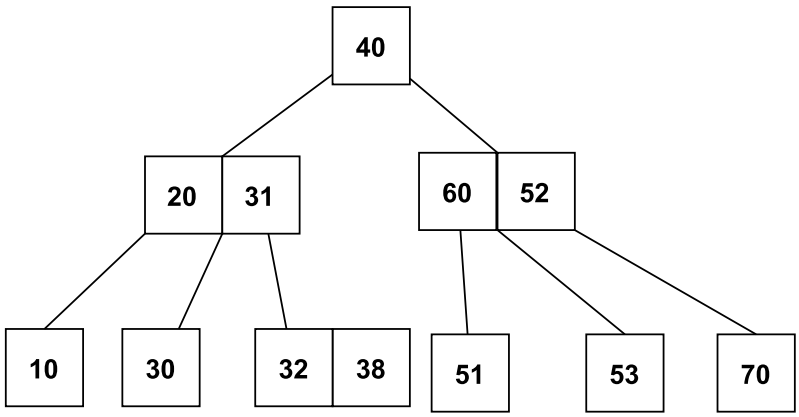
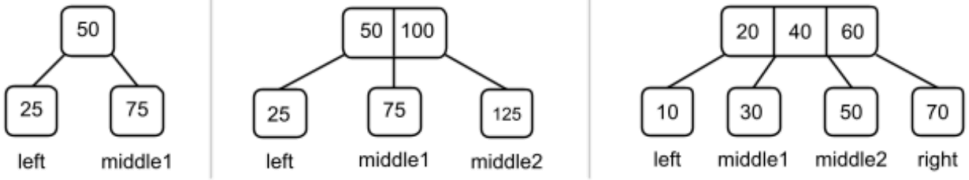
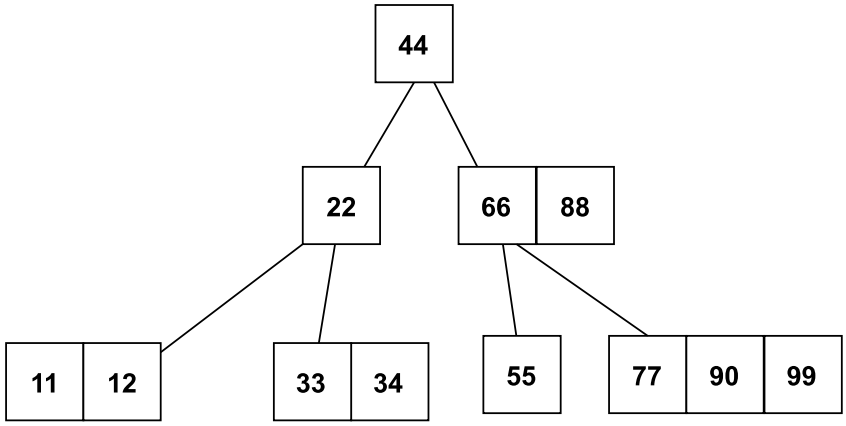
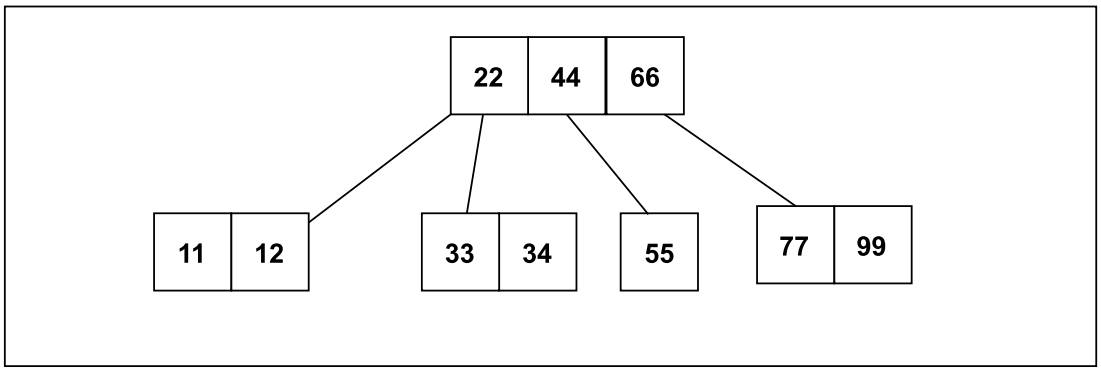


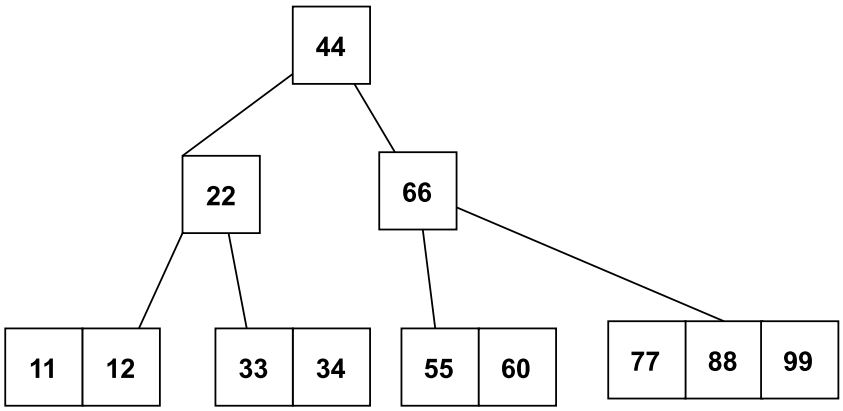

Figure 8.9.1: 2-3-4 child subtree labels.



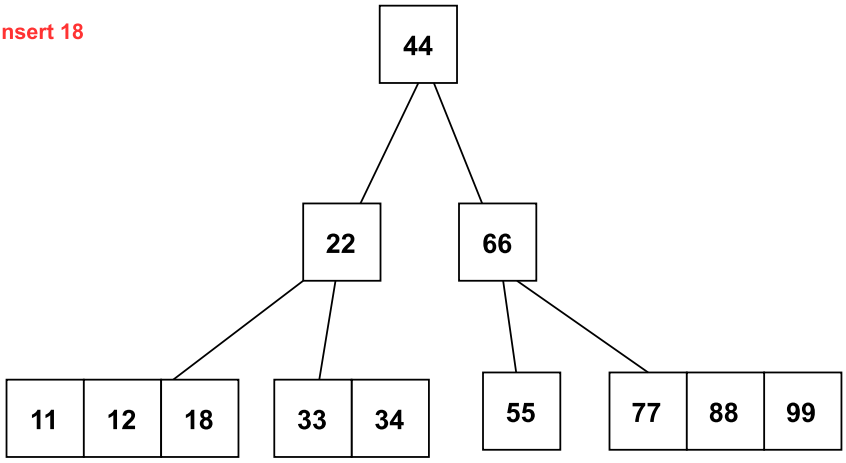


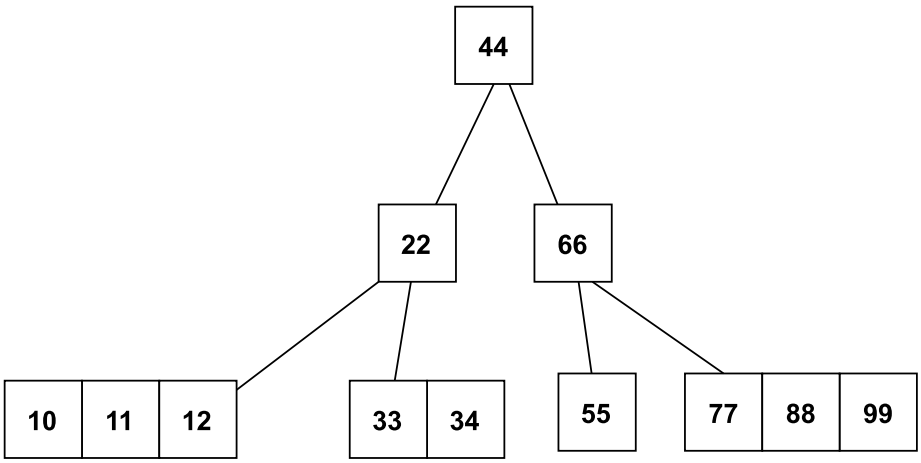

Insert 60



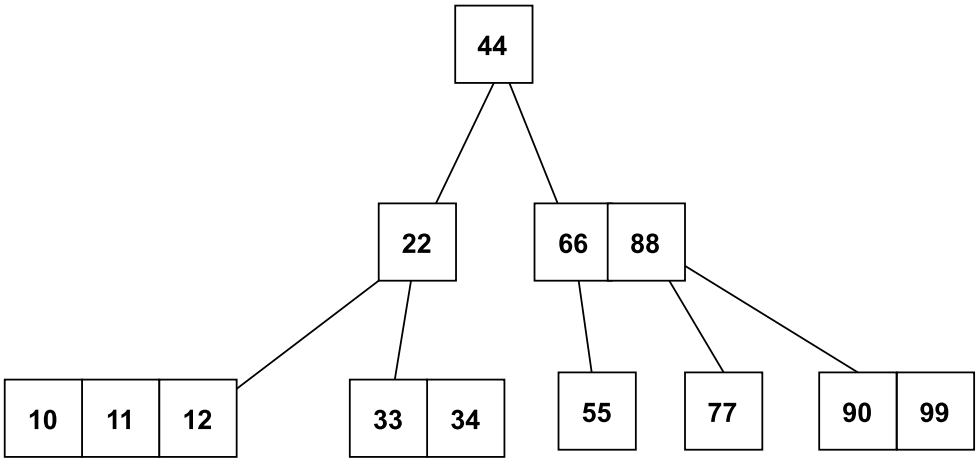

Insert 18



Insert 10




Insert 90




# Merge Algorithm

- A B-Tree merge operates on a node with 1 key and increases the node's keys to 2 or 3 using either a rotation or fusion
- A node's 2 adjacent siblings are checked first during a merge, and if either has 2 or more keys, a key is transferred via a rotation
- Such a rotation increases the number of keys in the merged node from 1 to 2.
- If all adjacent siblings of the node being merged have 1 key, then fusion is used to increase the number of keys in the node from 1 to 3.
- The merge operation can be performed on any node that has 1 key and a non-null parent node with at least 2 keys

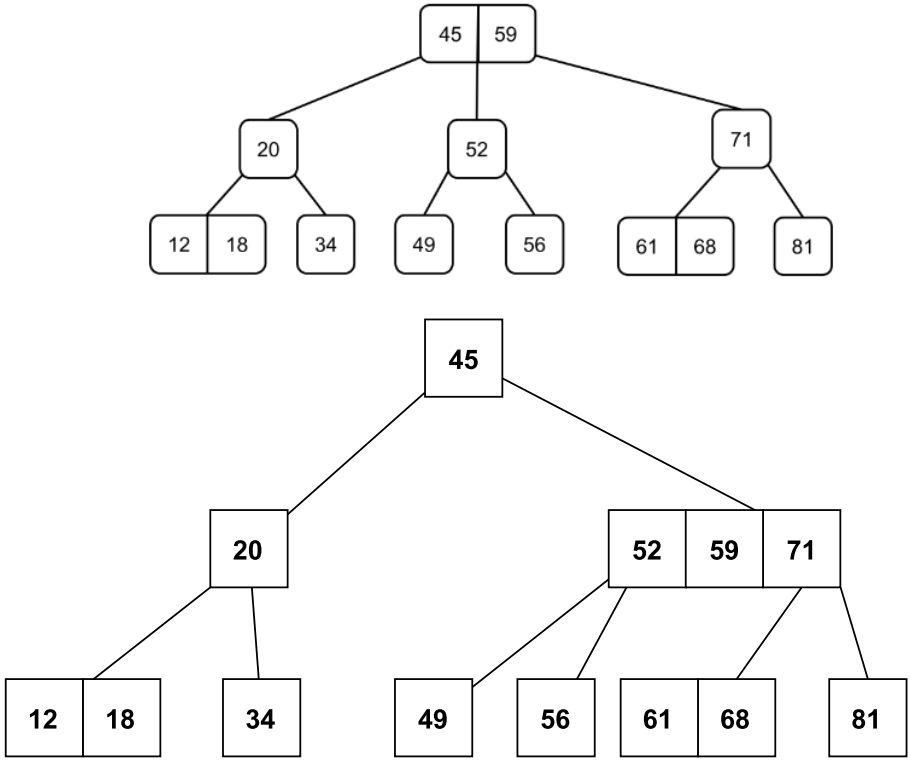
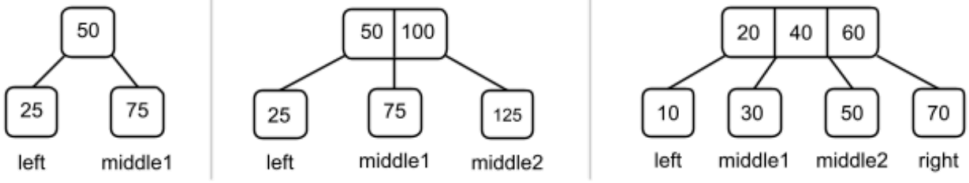
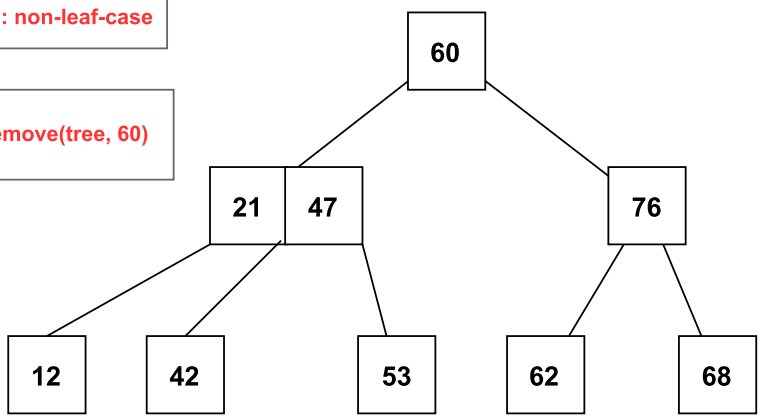



Figure 8.9.1: 2-3-4 child subtree labels.

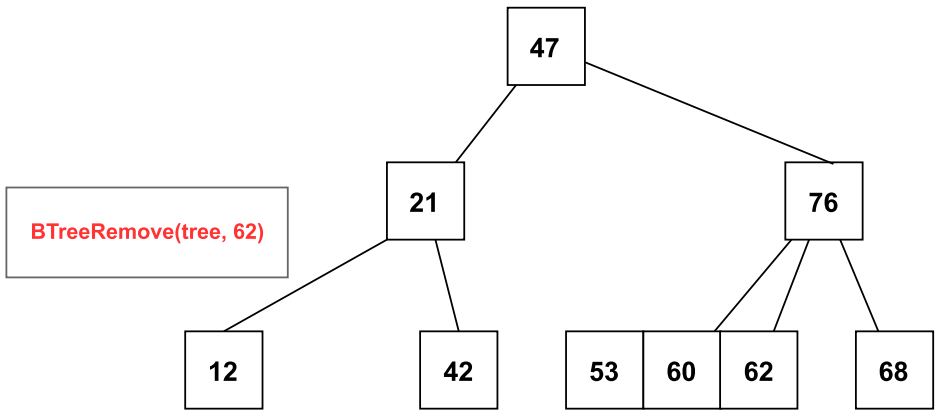


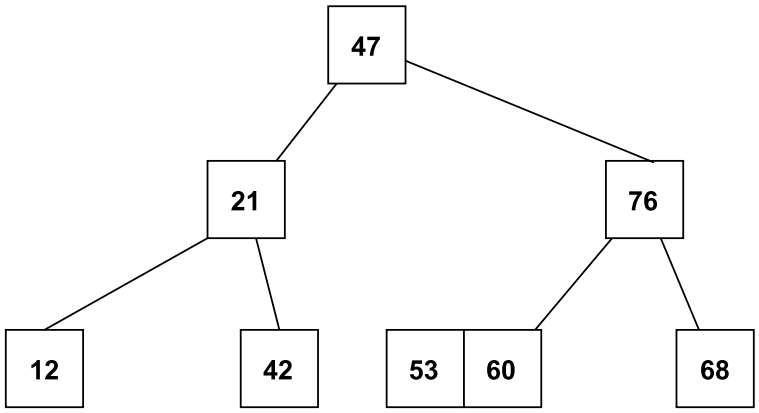

**8.13.5 BTree Remove Algorithm: non-leaf-case**

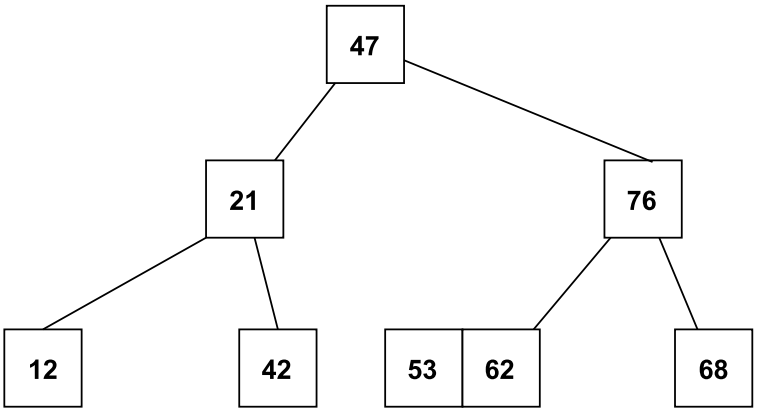
**BTreeRemove(tree, 60)**

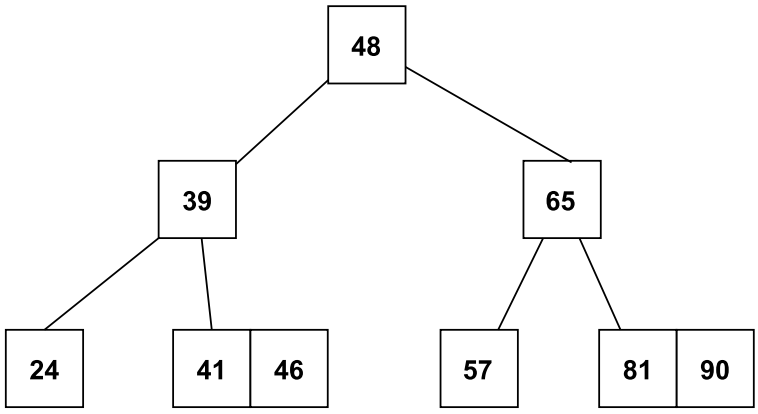


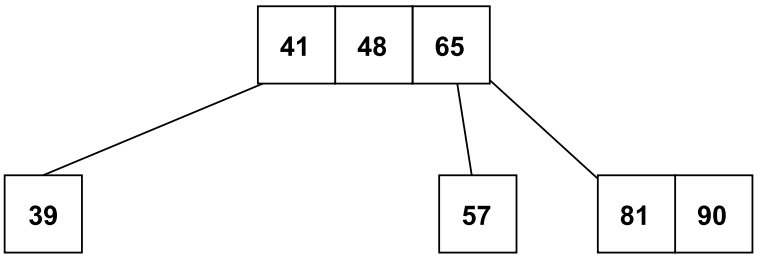

- When deleting 60, the process is more complex due to the key being found in an internal node
- The key 62 is a suitable replacement for 60, but 62 must be recursively removed before the swap.
- After the recursive removal completes, 60 is replaced with 62


BTreeRemove(tree, 62)
47
21
76
12
42
53
60
62
68


47
21
76
12
42
53
60
68


47
21
76
12
42
53
62
68


48
39
65
24
41
46
57
81
90


41
48
65
39
57
81
90