Lecture 13

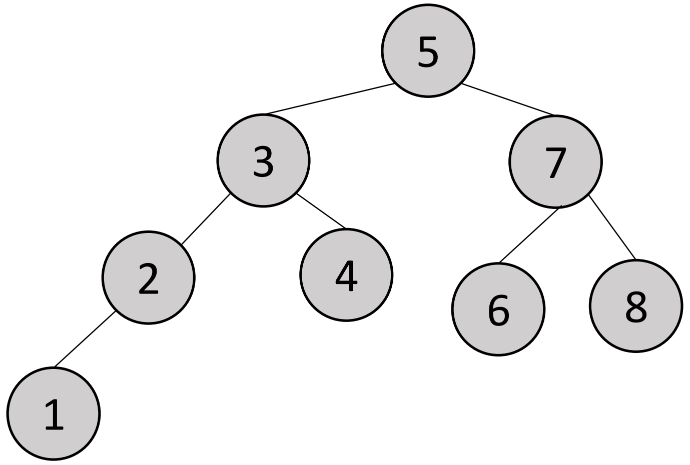
Multiway Trees

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Maintenance Operations

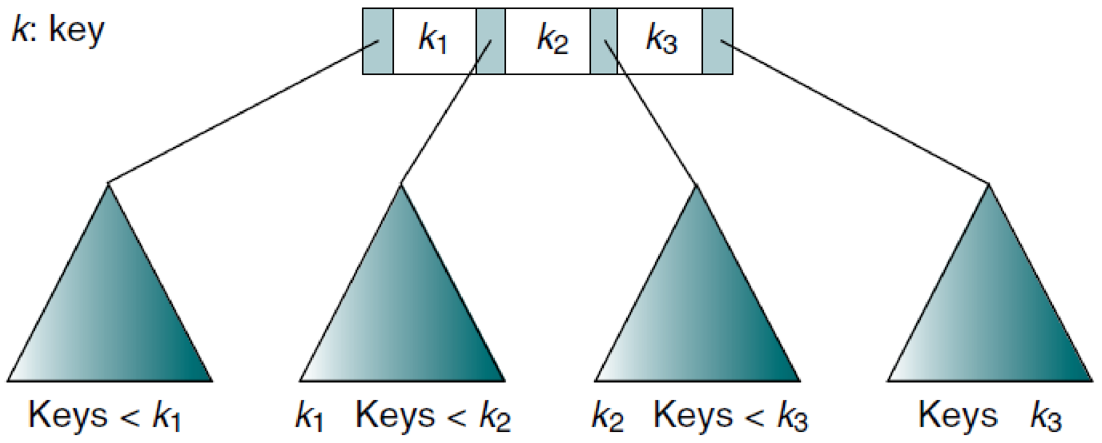
- Binary trees grow in height as the number of entries increases.
  - 1,000 entries → at least 10.
  - 100,000 entries → at least 17.
- If the binary tree is unbalanced, the height can be much larger, leading to inefficiencies.
- To limit the height of the tree, one option is multiway trees where each node can have multiple subtrees.
- A node can have multiple children.
- Multiway trees maintain the essential properties of binary search trees, such as ordering and efficient searching.

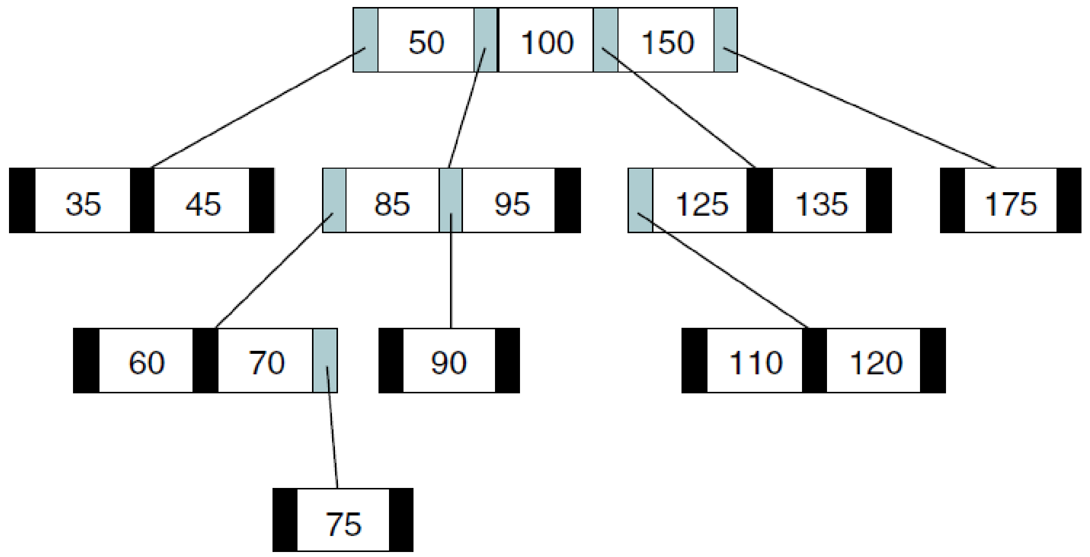


M-way Search Trees

There are five key properties:

1. Each node can have between 0 and m subtrees, where m is the order of the tree.
2. A node with k subtrees contains exactly k – 1 data entries.
3. Keys in a node and its subtrees follow a specific order:
    1. First subtree: All keys are less than the key value in the first data entry.
    2. For other subtrees, keys are greater than or equal to their respective parent entry.
4. The keys in the data entries of a node are sorted in non-decreasing order.
5. Each subtree of an m-way tree is itself an m-way tree.

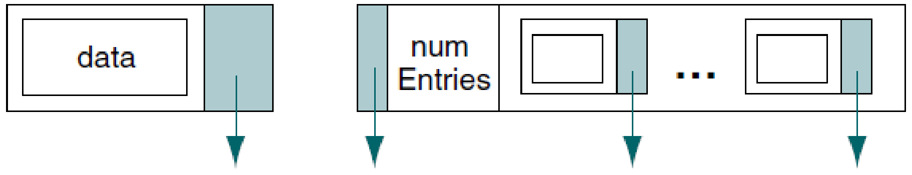




M-way Node Structure

- To implement an m-way tree node as described, we need to define two primary structures:
  - Entry Structure: Represents an individual data entry within the node. Each entry stores the key data (or a pointer to the data if it's stored externally) and the pointer to the right subtree.
  - Node Structure: Represents a node of the m-way tree, containing: A pointer to the first subtree, a count of the number of entries in the node, an array of entries that can hold up to m−1 entries.
- The binary search tree is an *m*-way tree of order 2.

```
entry
    data
    rightPtr
end entry
node
    firstPtr
    numEntries
    entries    array of entry
end node
```



(a) Entry (b) Node

B-trees

- The B-tree is a balanced m-way search tree that overcomes the imbalance issue in m-way trees.
- Widely used in databases and file systems because it ensures efficient search, insert, and delete operations.

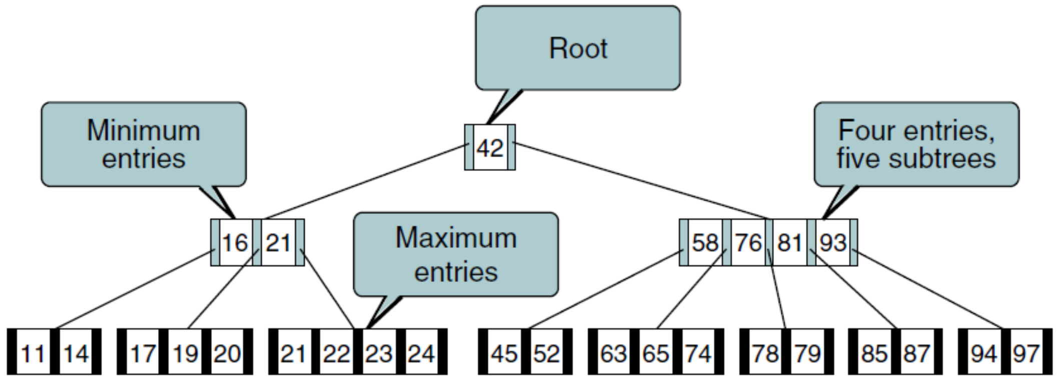
1. The root is either a leaf or it has 2 … m subtrees.

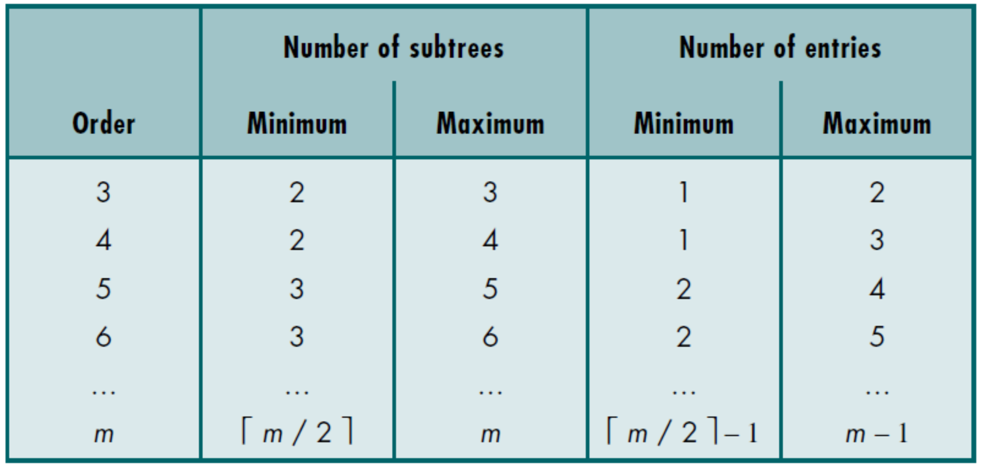
2. All internal nodes have at least $\lceil m/2 \rceil$ nonnull subtrees and at most m nonnull subtrees.

3. All leaf nodes are at the same level; that is, the tree is perfectly balanced.

4. A leaf node has at least $\lceil m/2 \rceil$ – 1 and at most m – 1 entries.

A B-Tree of order 5



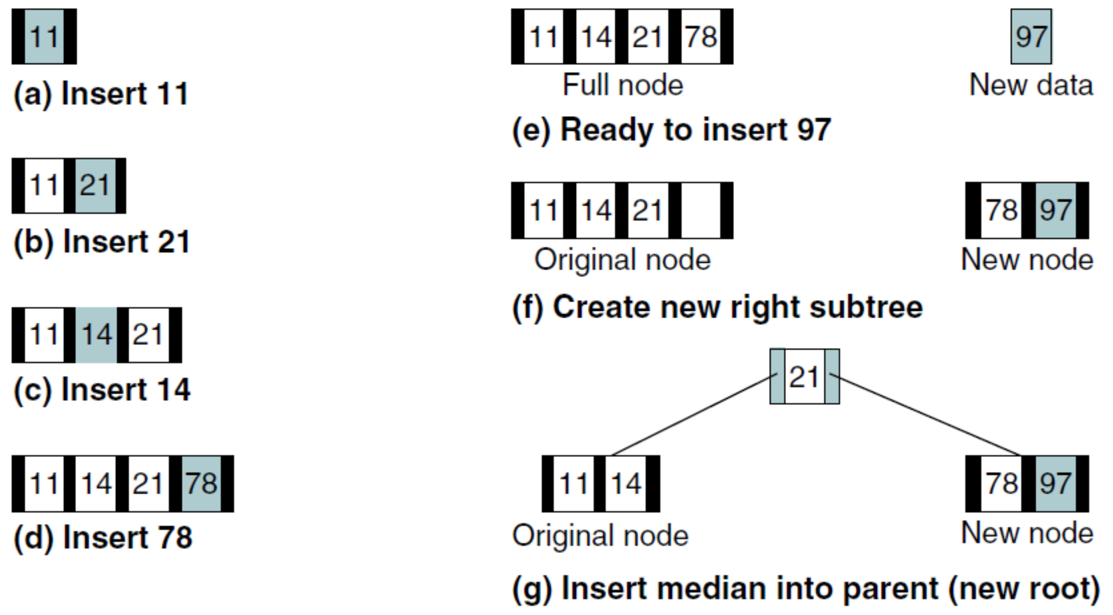


	Number of subtrees		Number of entries	
Order	Minimum	Maximum	Minimum	Maximum
3	2	3	1	2
4	2	4	1	3
5	3	5	2	4
6	3	6	2	5
…	…	…	…	…
m	$\lceil m/2 \rceil$	m	$\lceil m/2 \rceil - 1$	$m - 1$

B-tree Insertion

- A B-tree grows from the bottom up.
- **Step 1: Locate the Appropriate Leaf Node**:
  - Start at the root and navigate down the tree.
  - Use the key comparisons to decide which child node to visit until you reach the correct leaf node.
- **Step 2: Check for Overflow**:
  - If the leaf node is not full insert the new key and data into the node in sorted order.
  - If the leaf node is full (has m−1 entries), an overflow condition occurs, and the node must be split.
- **Step 3, for overflow: Splitting the Node**:
  - Split the full node into two nodes:
    - Create a new node.
    - Move the right half of the entries from the full node to the new node.
  - Identify the median entry (middle key) of the full node:
    - The median entry is promoted to the parent node.
  - Insert the new entry into the appropriate node (original or new) based on its key value.

A B-Tree of order 5, again

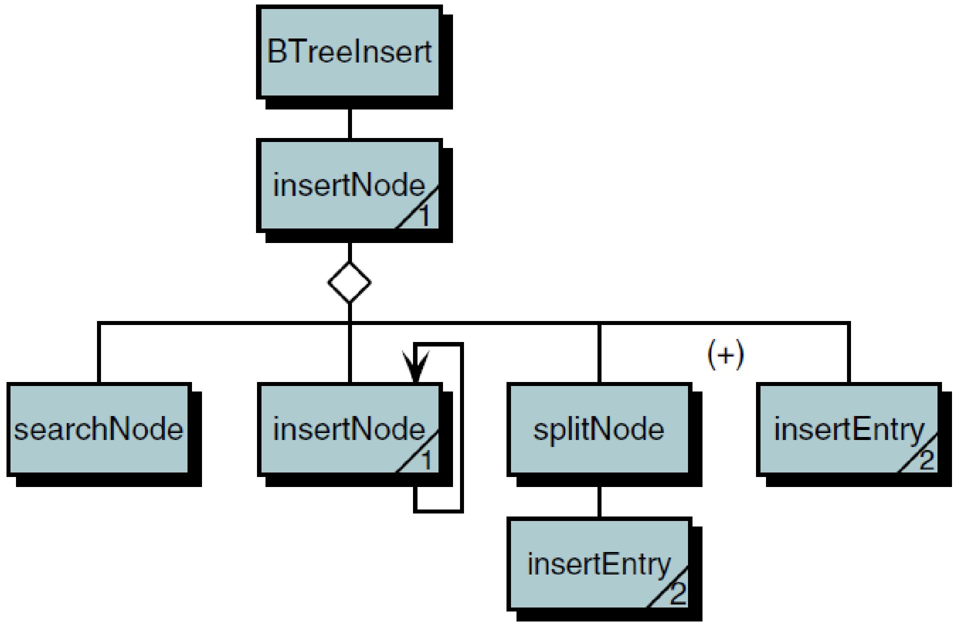


Step 4: Insert the Median into the Parent Node:
- If the parent node is not full, simply insert the median entry in sorted order.
- If the parent node is full, repeat the splitting process recursively.

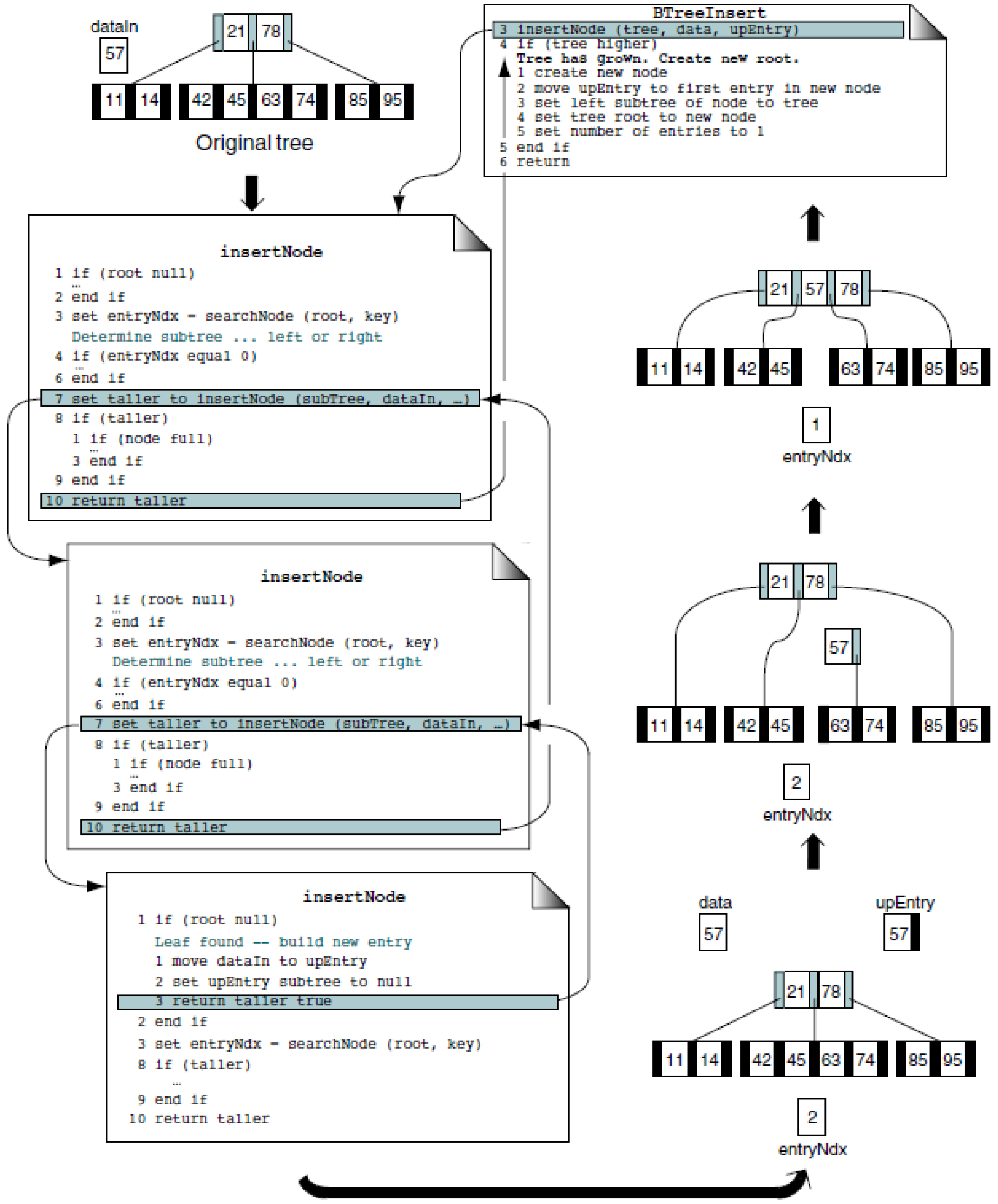
Special Case: Root Splitting:
- If splitting occurs at the **root**, a new root node is created.
- The median entry becomes the only entry in the new root, and the two split nodes become its children.
- This increases the height of the B-tree by one level.

B-tree Insertion

```
Algorithm BTreeInsert (tree,  data)
Inserts data into B-Tree. Equal keys placed on right branch.
   Pre     tree is reference to B-Tree; may be null
   Post    data inserted
1  if (tree null)
   Empty tree. Insert first node.
   1   create new node
   2   set left subtree of node to null
   3   move data to first entry in new node
   4   set subtree of first entry to null
   5   set tree root to address of new node
   6   set number of entries to 1
2  end if
   Insert data in existing tree
3  insertNode (tree, data, upEntry)
4  if (tree higher)
       Tree has grown; create new root
   1   create new node
   2   move upEntry to first entry in new node
   3   set left subtree of node to tree
   4   set tree root to new node
   5   set number of entries to 1
   6   end if
end BTreeInsert
```



B-tree Insertion



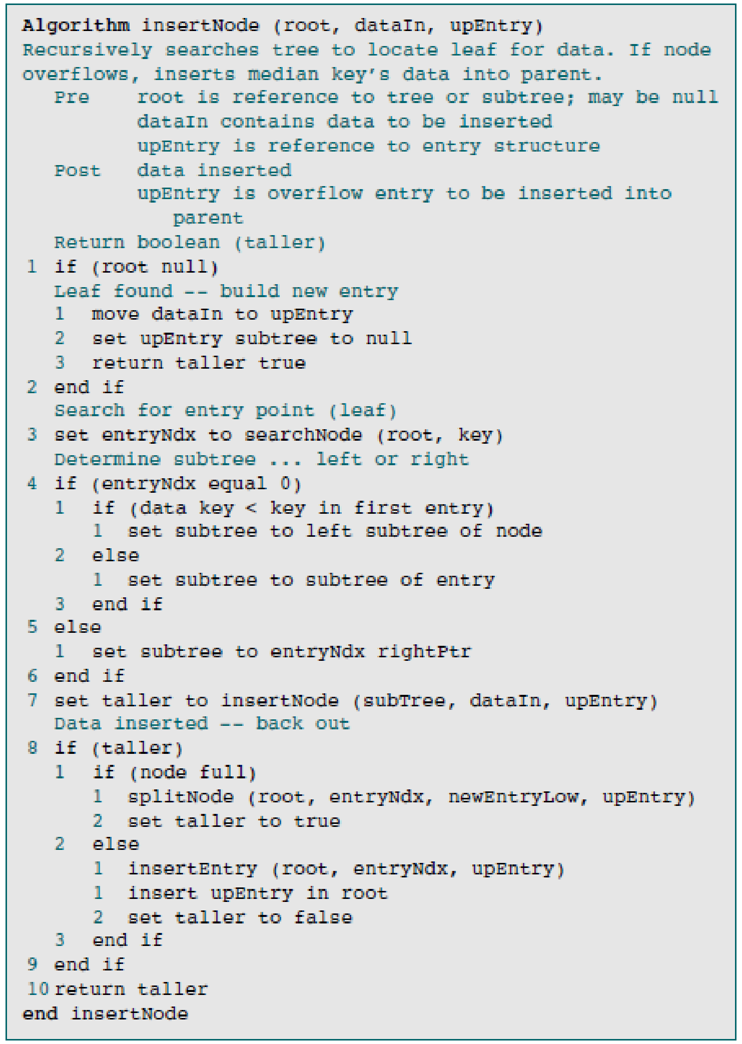
```
BTreeInsert
3 insertNode (tree, data, upEntry)
4 if (tree higher)
  Tree has grown. Create new root.
  1 create new node
  2 move upEntry to first entry in new node
  3 set left subtree of node to tree
  4 set tree root to new node
  5 set number of entries to 1
5 end if
6 return
```

```
insertNode
 1 if (root null)
   …
 2 end if
 3 set entryNdx = searchNode (root, key)
   Determine subtree ... left or right
 4 if (entryNdx equal 0)
   …
 6 end if
 7 set taller to insertNode (subTree, dataIn, …)
 8 if (taller)
   1 if (node full)
     …
   3 end if
 9 end if
10 return taller
```

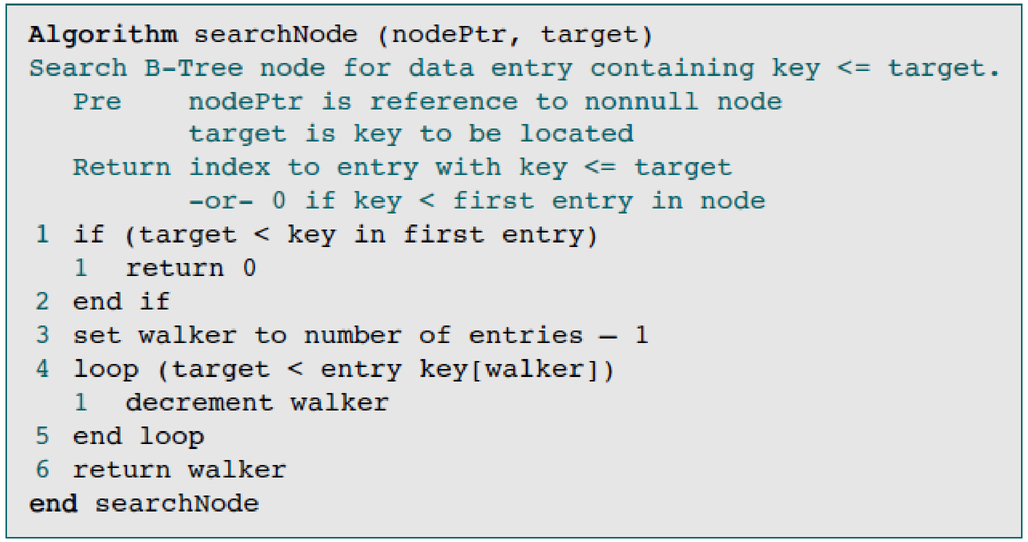
```
insertNode
 1 if (root null)
   …
 2 end if
 3 set entryNdx = searchNode (root, key)
   Determine subtree ... left or right
 4 if (entryNdx equal 0)
   …
 6 end if
 7 set taller to insertNode (subTree, dataIn, …)
 8 if (taller)
   1 if (node full)
     …
   3 end if
 9 end if
10 return taller
```

```
insertNode
 1 if (root null)
   Leaf found -- build new entry
   1 move dataIn to upEntry
   2 set upEntry subtree to null
   3 return taller true
 2 end if
 3 set entryNdx = searchNode (root, key)
 8 if (taller)
     …
 9 end if
10 return taller
```

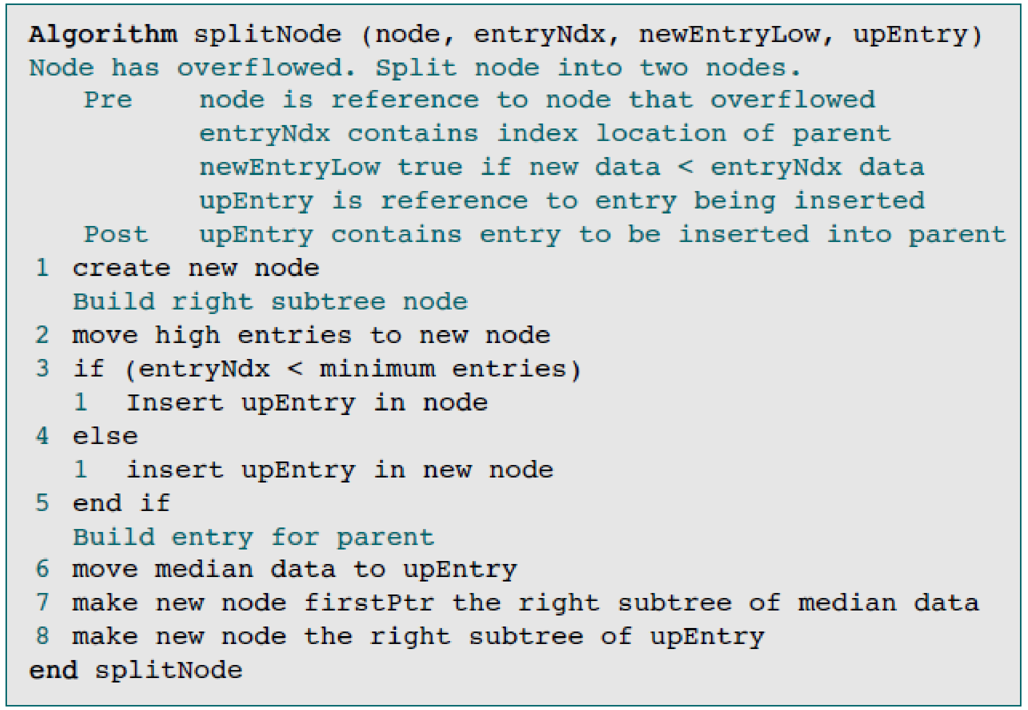
B-tree Insertion



```
Algorithm insertNode (root, dataIn, upEntry)
Recursively searches tree to locate leaf for data. If node
overflows, inserts median key's data into parent.
   Pre     root is reference to tree or subtree; may be null
           dataIn contains data to be inserted
           upEntry is reference to entry structure
   Post    data inserted
           upEntry is overflow entry to be inserted into
              parent
   Return boolean (taller)
1  if (root null)
   Leaf found -- build new entry
   1  move dataIn to upEntry
   2  set upEntry subtree to null
   3  return taller true
2  end if
   Search for entry point (leaf)
3  set entryNdx to searchNode (root, key)
   Determine subtree ... left or right
4  if (entryNdx equal 0)
   1  if (data key < key in first entry)
      1  set subtree to left subtree of node
   2  else
      1  set subtree to subtree of entry
   3  end if
5  else
   1  set subtree to entryNdx rightPtr
6  end if
7  set taller to insertNode (subTree, dataIn, upEntry)
   Data inserted -- back out
8  if (taller)
   1  if (node full)
      1  splitNode (root, entryNdx, newEntryLow, upEntry)
      2  set taller to true
   2  else
      1  insertEntry (root, entryNdx, upEntry)
      1  insert upEntry in root
      2  set taller to false
   3  end if
9  end if
10 return taller
end insertNode
```

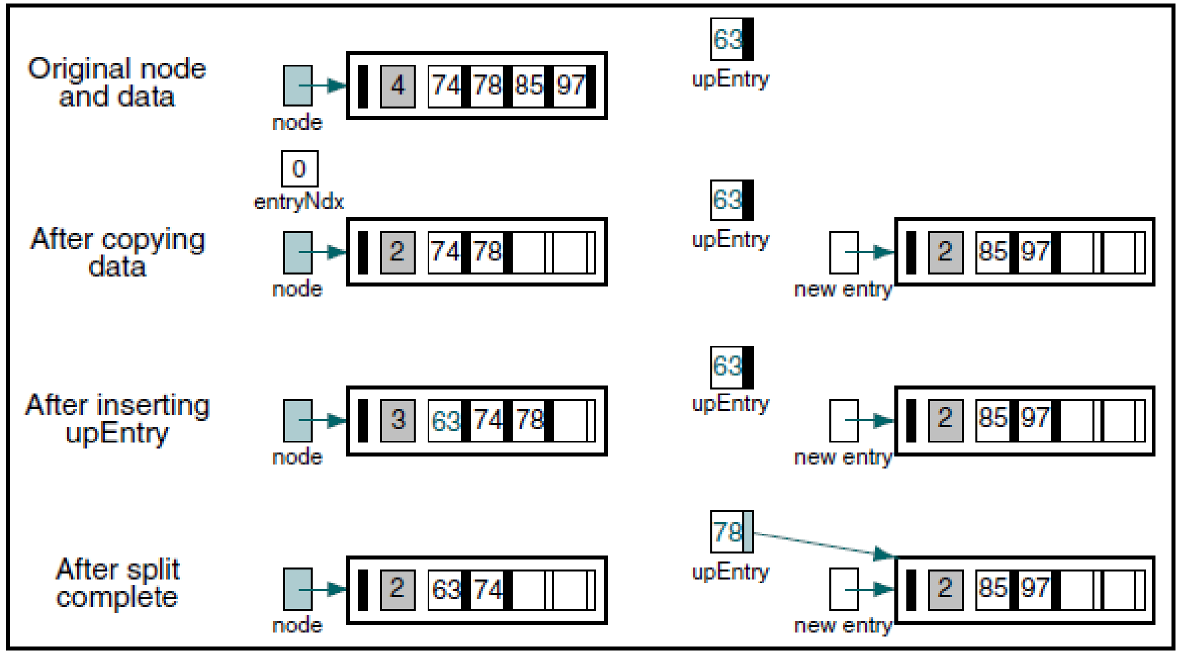


```
Algorithm searchNode (nodePtr, target)
Search B-Tree node for data entry containing key <= target.
   Pre     nodePtr is reference to nonnull node
           target is key to be located
   Return index to entry with key <= target
           -or- 0 if key < first entry in node
1  if (target < key in first entry)
   1  return 0
2  end if
3  set walker to number of entries – 1
4  loop (target < entry key[walker])
   1  decrement walker
5  end loop
6  return walker
end searchNode
```

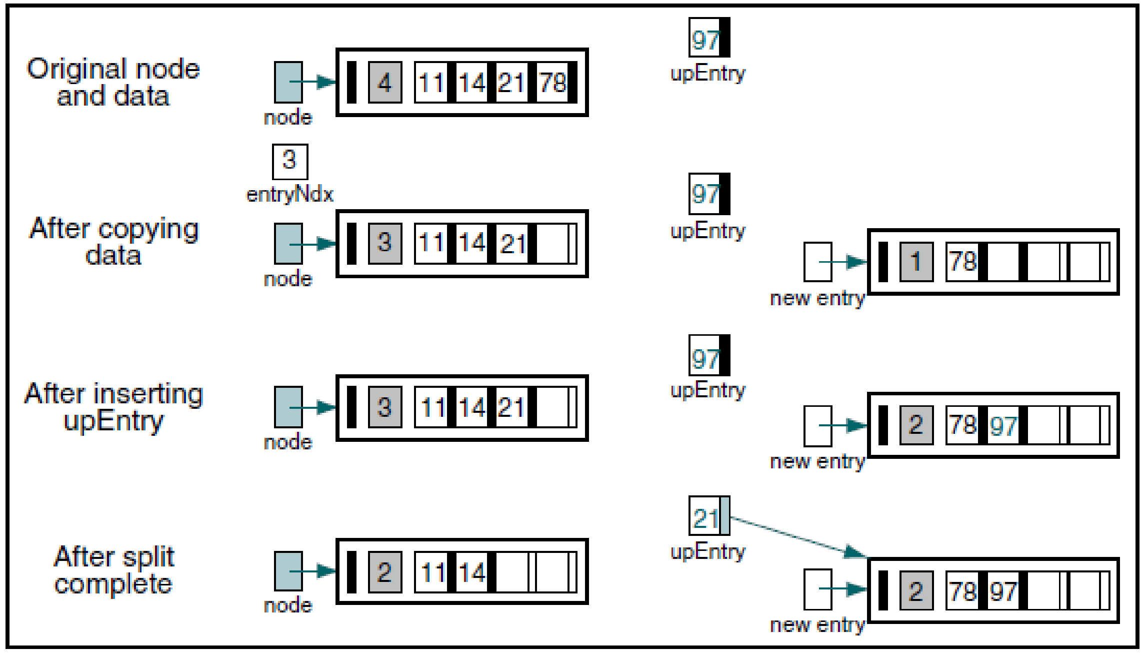


```
Algorithm splitNode (node, entryNdx, newEntryLow, upEntry)
Node has overflowed. Split node into two nodes.
    Pre    node is reference to node that overflowed
           entryNdx contains index location of parent
           newEntryLow true if new data < entryNdx data
           upEntry is reference to entry being inserted
    Post   upEntry contains entry to be inserted into parent
1  create new node
   Build right subtree node
2  move high entries to new node
3  if (entryNdx < minimum entries)
   1  Insert upEntry in node
4  else
   1  insert upEntry in new node
5  end if
   Build entry for parent
6  move median data to upEntry
7  make new node firstPtr the right subtree of median data
8  make new node the right subtree of upEntry
end splitNode
```

Split Node B-tree Order of 5

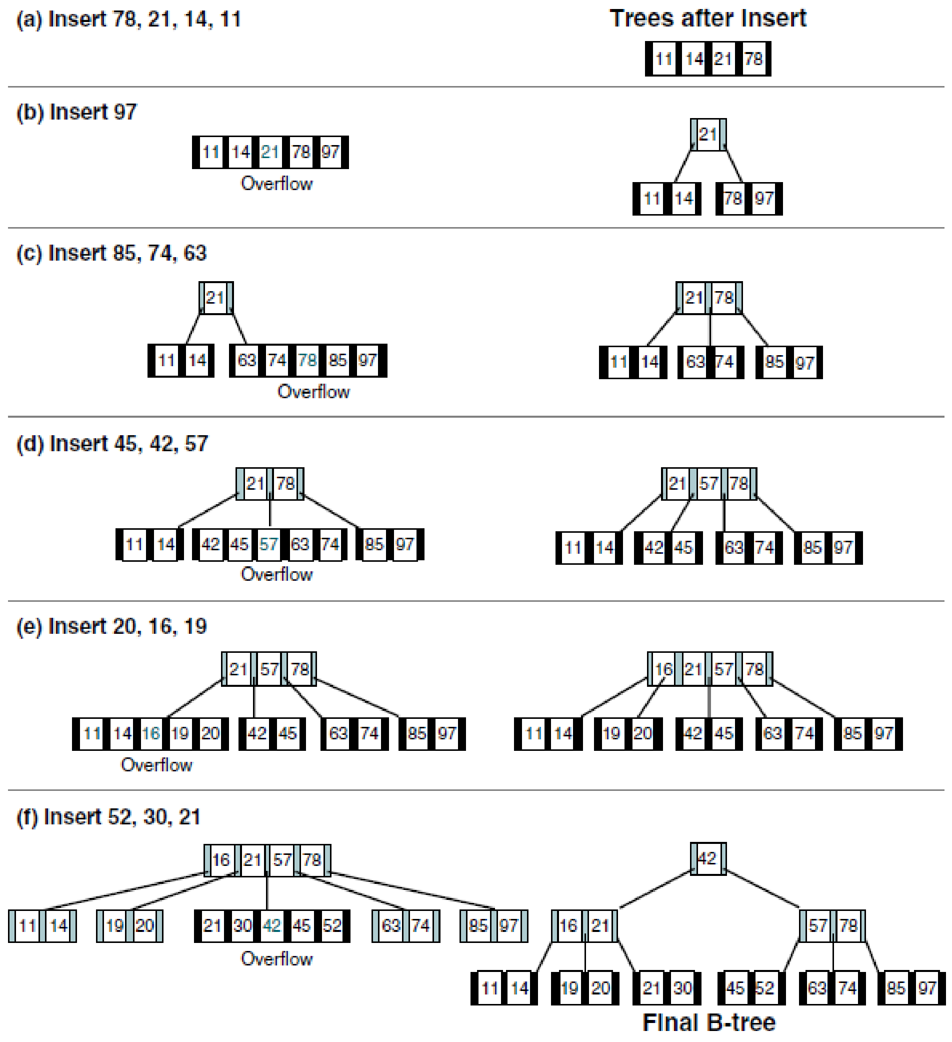


(a) New entry ≤ median



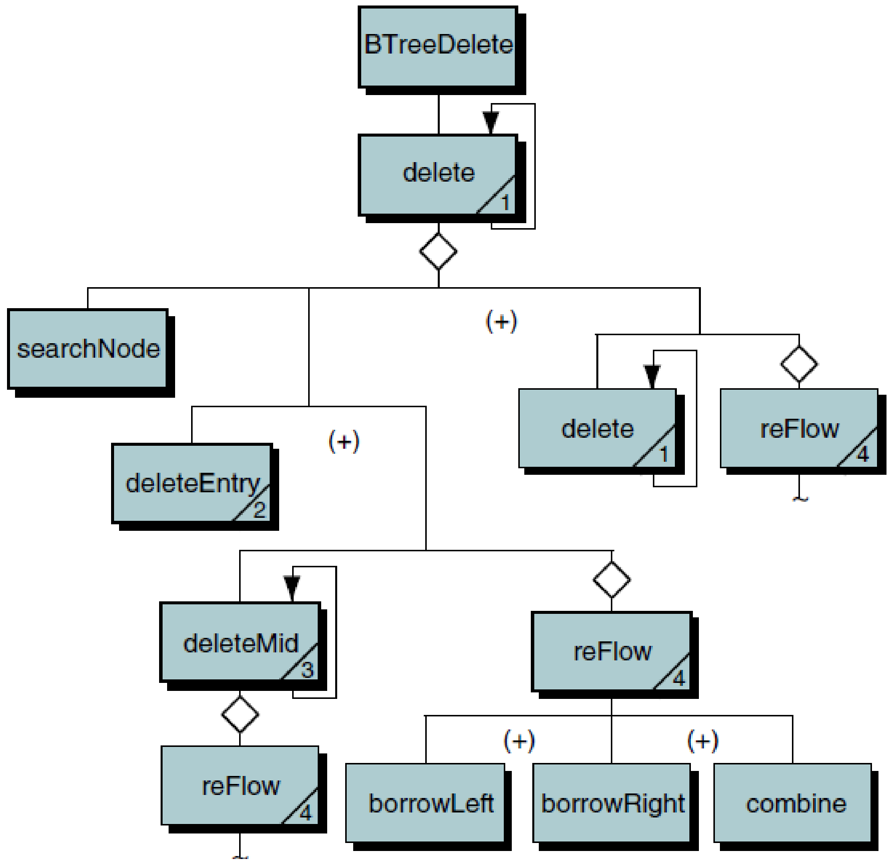
(b) New entry > median

Insertion Summary



B-tree Deletion

- Deletion in a B-tree is more complex than insertion due to the need to maintain the structural properties of the tree. The three key challenges during deletion are:
    - **Ensuring the Key Exists**: Verify that the key to be deleted is actually present in the B-tree.
    - **Handling Underflow**: If deleting a key causes a node to have fewer than the required minimum number of entries (underflow), structural corrections are needed.
    - **Deleting from an Internal Node**: Since deletions occur at leaf nodes, if the key is in an internal node, we must replace it with an appropriate key from the leaf level.



B-tree Delete

```
Algorithm BTreeDelete (tree, dltKey)
Delete entry with key target from B-tree.
   Pre    tree is a reference to a B-tree
          dltKey is the key of the entry to be deleted
   Post   data deleted or false returned
   Return success (found) or failure (not found)
 1 if (tree empty)
   1 return false
 2 end if
 3 delete (tree, dltKey, success)
 4 if (success)
   1 if (tree number of entries zero)
        Tree is shorter--delete root
      1 set tree to left subtree
   2 end if
 5 end if
 6 return success
end BTreeDelete
```

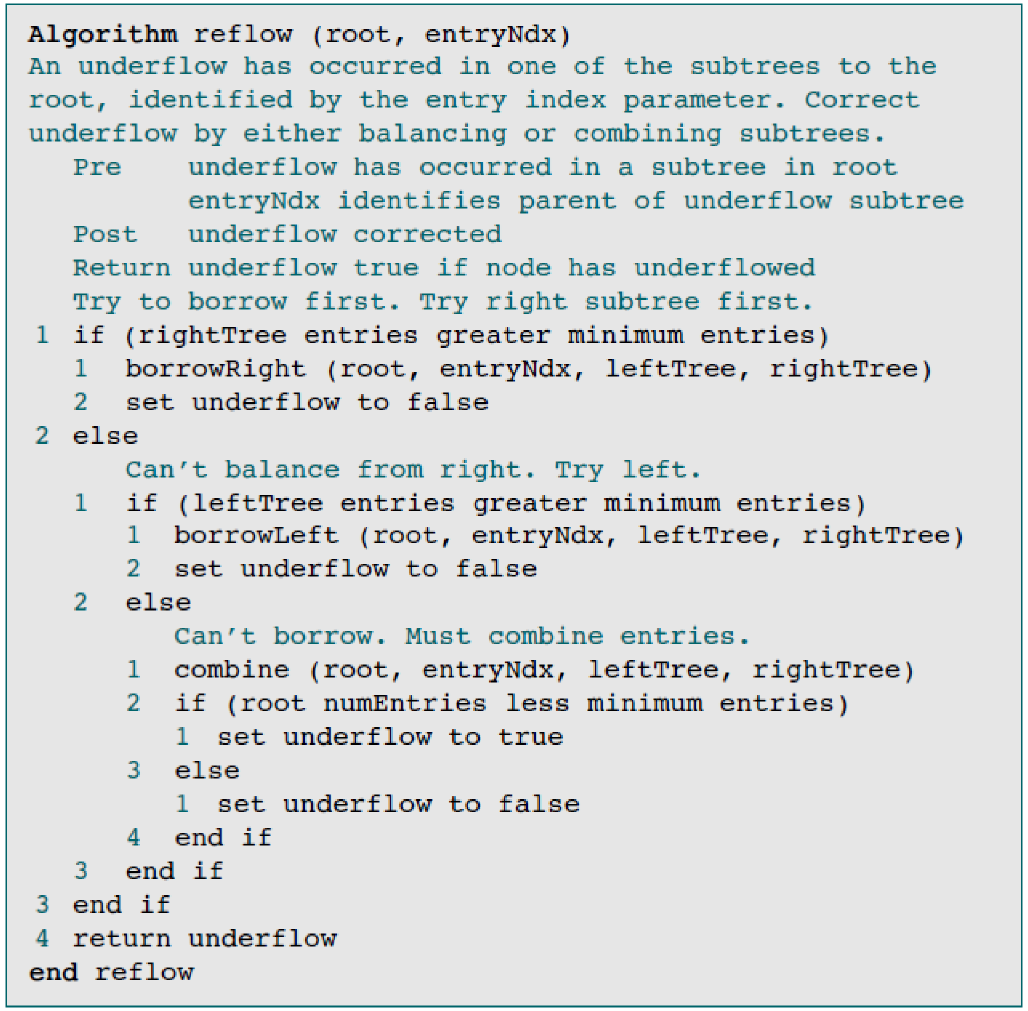
```
Algorithm delete (root, deleteKey, success)
Recursively locates node containing deleteKey; deletes data.
   Pre    root is a reference to a nonnull B-Tree
          deleteKey is key to entry to be deleted
          success is reference to boolean
   Post   data deleted--success true; or success false
          underflow--true or false
   Return success true or false
1 if (root null)
     Leaf node found--deleteKey key does not exist
  1 set success false
  2 return false
2 end if
3 set entryNdx to searchNode (root, deleteKey)
4 if (deleteKey found)
     Found entry to be deleted
  1 set success to true
  2 if (leaf node)
     1 set underflow to deleteEntry (root, entryNdx)
  3 else
        Entry is in internal node
     1 if (entryNdx > 0)
        1 set leftPtr to rightPtr of previous entry
     2 else
        1 set leftPtr to root firstPtr
     3 end if
     4 set underflow to deleteMid (root, entryNdx, leftPtr)
     5 if (underflow)
        1 set underflow to reFlow (root, entryNdx)
     6 end if
  4 end if
5 else
  1 if (deleteKey less key in first entry)
     1 set subtree to root firstPtr
  2 else
        deleteKey is in right subtree
     1 set subtree to entryNdx rightPtr
  3 end if
  4 set underflow to delete (subtree, deleteKey, success)
  5 if (underflow)
     1 set underflow to reFlow (root, entryNdx)
  6 end if
6 end if
7 return underflow
end delete
```

Reflow

When a B-tree underflows during deletion, we restore balance using reflow, which involves two methods: balance and combine.

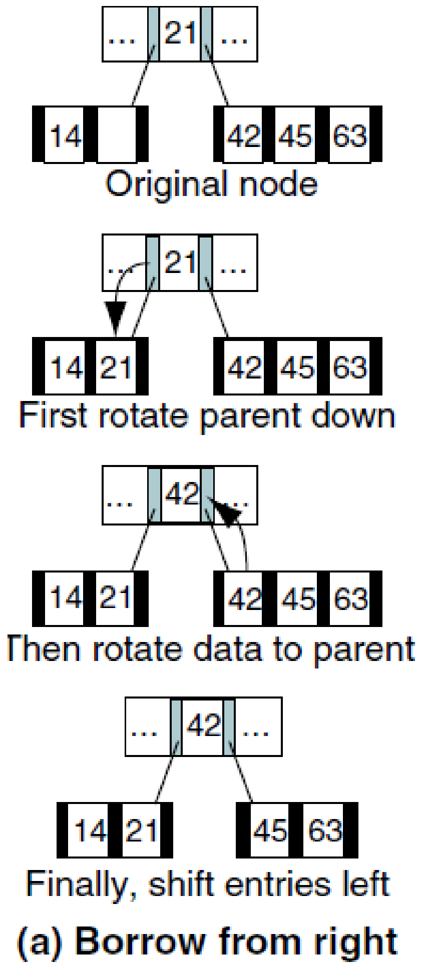
1. **Balance** (Preferred):
    - Borrow an entry from a sibling with extra entries.
    - Move a parent key down and shift a sibling key up to maintain structure.
2. **Combine** (Fallback):
    - If siblings have only minimum entries, merge the underflowed node, a sibling, and the parent key into one node.
    - Remove the empty node and adjust parent pointers.

Balance preserves the tree's structure, while combine merges nodes when balancing isn't possible.

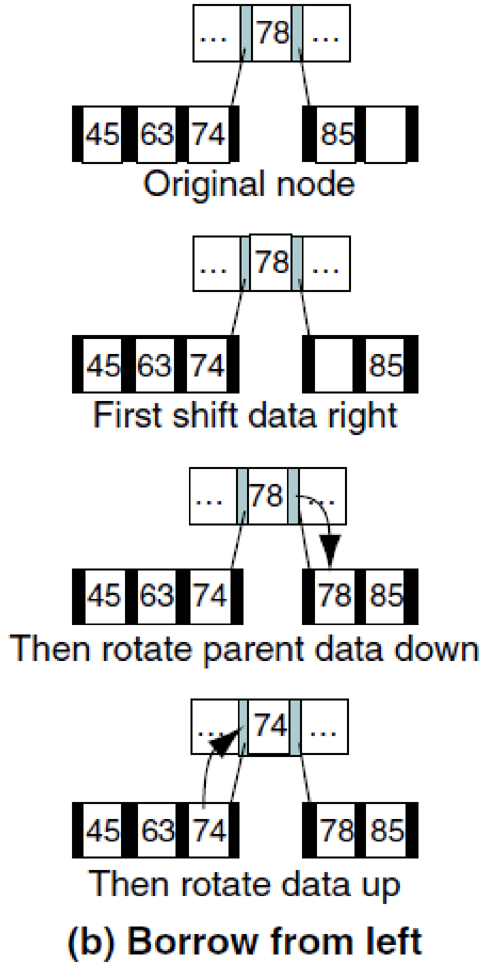


```
Algorithm reflow (root, entryNdx)
An underflow has occurred in one of the subtrees to the
root, identified by the entry index parameter. Correct
underflow by either balancing or combining subtrees.
   Pre     underflow has occurred in a subtree in root
           entryNdx identifies parent of underflow subtree
   Post    underflow corrected
   Return underflow true if node has underflowed
   Try to borrow first. Try right subtree first.
1  if (rightTree entries greater minimum entries)
   1  borrowRight (root, entryNdx, leftTree, rightTree)
   2  set underflow to false
2  else
      Can't balance from right. Try left.
   1  if (leftTree entries greater minimum entries)
      1  borrowLeft (root, entryNdx, leftTree, rightTree)
      2  set underflow to false
   2  else
         Can't borrow. Must combine entries.
      1  combine (root, entryNdx, leftTree, rightTree)
      2  if (root numEntries less minimum entries)
         1  set underflow to true
      3  else
         1  set underflow to false
      4  end if
   3  end if
3  end if
4  return underflow
end reflow
```

Borrowing

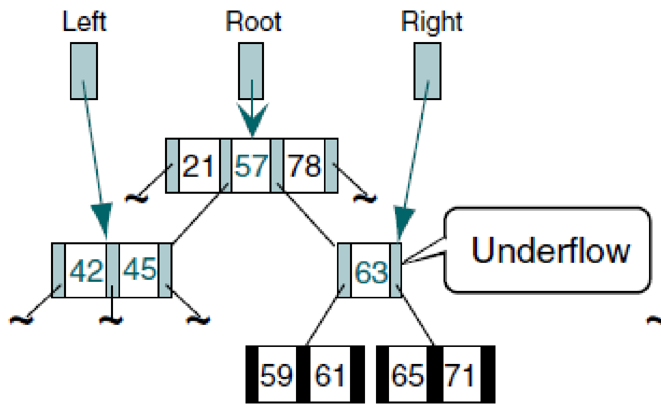


... 21 ...
14
42 45 63
Original node
... 21 ...
14 21
42 45 63
First rotate parent down
... 42 ...
14 21
42 45 63
Then rotate data to parent
... 42 ...
14 21
45 63
Finally, shift entries left
(a) Borrow from right

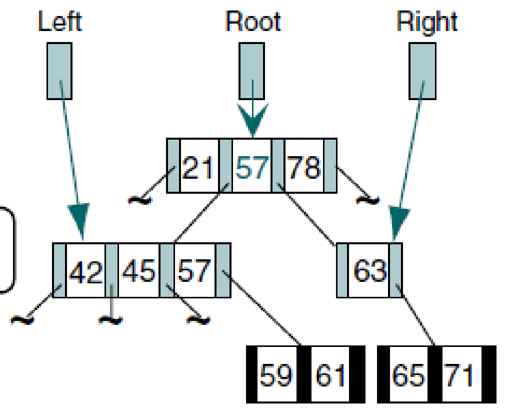


... 78 ...
45 63 74
85
Original node
... 78 ...
45 63 74
85
First shift data right
... 78 ...
45 63 74
78 85
Then rotate parent data down
... 74 ...
45 63 74
78 85
Then rotate data up
(b) Borrow from left

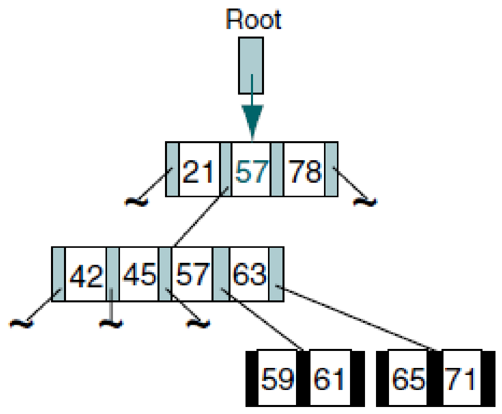
Combining



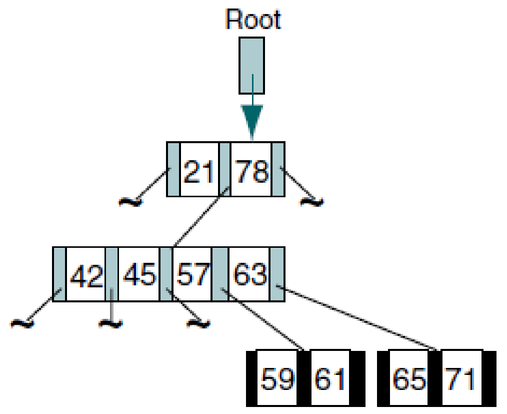
(a) After underflow



(b) After moving root to subtree



(c) After moving right entries



(d) After shifting root

Deletion Summary

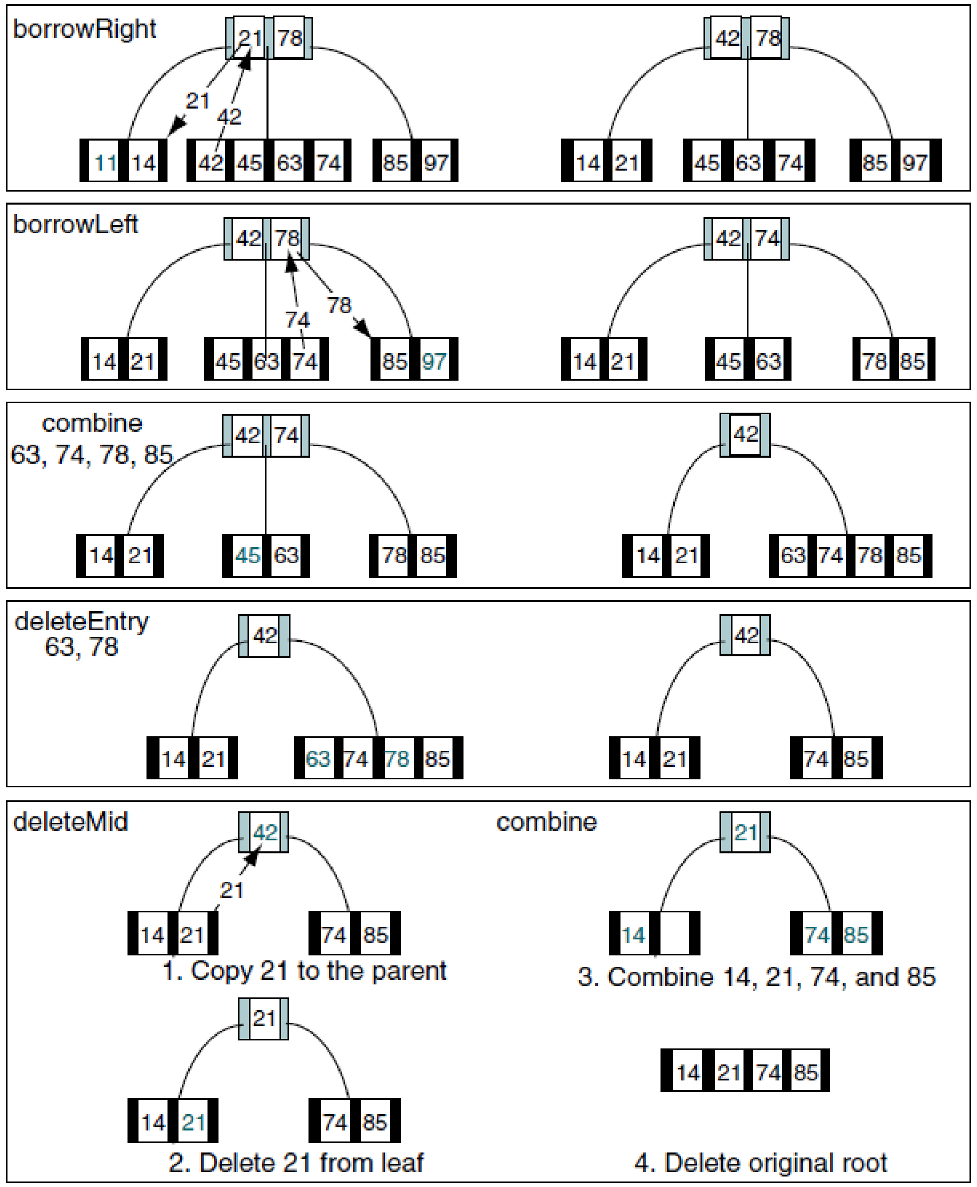
(a) Delete 11

(b) Delete 97

(c) Delete 45

(d) Delete 63 and 78

(e) Delete 42


borrowRight
21
42
borrowLeft
78
74
combine
63, 74, 78, 85
deleteEntry
63, 78
deleteMid
21
1. Copy 21 to the parent
2. Delete 21 from leaf
combine
3. Combine 14, 21, 74, and 85
4. Delete original root