Data Structure and Algorithms [CO2003]

Chapter 7 - AVL Tree

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Outcomes

- **L.O.3.1** - Depict the following concepts: binary tree, complete binary tree, balanced binary tree, AVL tree, multi-way tree, etc.
- **L.O.3.2** - Describe the strorage structure for tree structures using pseudocode.
- **L.O.3.3** - List necessary methods supplied for tree structures, and describe them using pseudocode.
- **L.O.3.4** - Identify the importance of "blanced" feature in tree structures and give examples to demonstate it.
- **L.O.3.5** - Identiy cases in which AVL tree and B-tree are unblanced, and demonstrate methods to resolve all the cases step-by-step using figures.

Outcomes

- **L.O.3.6** - Implement binary tree and AVL tree using C/C++.
- **L.O.3.7** - Use binary tree and AVL tree to solve problems in real-life, especially related to searching techniques.
- **L.O.3.8** - Analyze the complexity and develop experiment (program) to evaluate methods supplied for tree structures.
- **L.O.8.4** - Develop recursive implementations for methods supplied for the following structures: list, tree, heap, searching, and graphs.
- **L.O.1.2** - Analyze algorithms and use Big-O notation to characterize the computational complexity of algorithms composed by using the following control structures: sequence, branching, and iteration (not recursion).

AVL Tree Concepts

Definition
AVL Tree is:

- A Binary Search Tree,
- in which the heights of the left and right subtrees of the root differ by at most 1, and
- the left and right subtrees are again AVL trees.

Discovered by G.M.Adel'son-Vel'skii and E.M.Landis in 1962.

AVL Tree is a Binary Search Tree that is balanced tree.

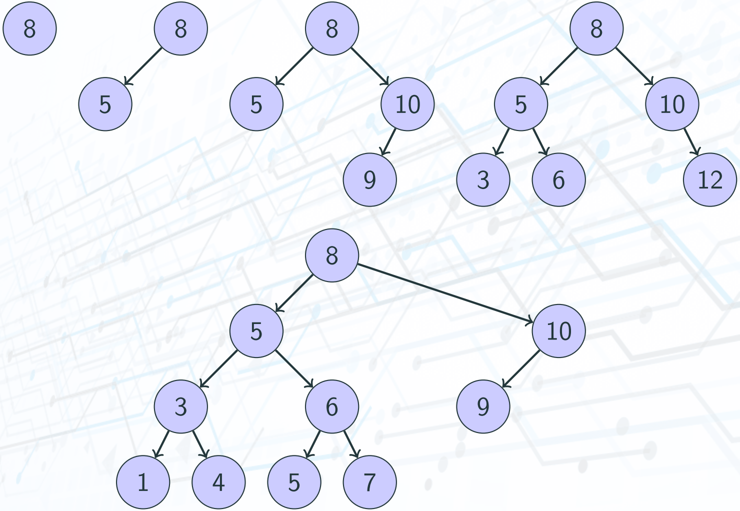
A binary tree is an AVL Tree if

- Each node satisfies BST property: key of the node is greater than the key of each node in its left subtree and is smaller than or equals to the key of each node in its right subtree.
- Each node satisfies balanced tree property: the difference between the heights of the left subtree and right subtree of the node does not exceed one.

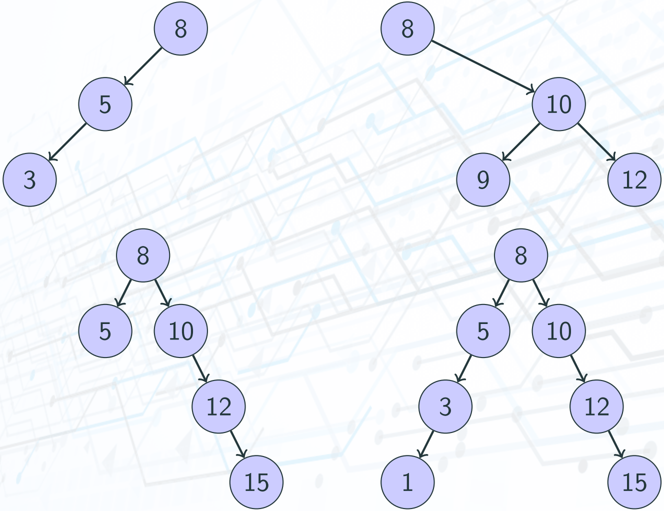
Balance factor

- left_higher (LH): $H_L = H_R + 1$
- equal_height (EH): $H_L = H_R$
- right_higher (RH): $H_R = H_L + 1$

(H_L, H_R: the heights of left and right subtrees)


8
8
5
8
5
10
9
8
5
10
3
6
12
8
5
10
3
6
9
1
4
5
7

Non-AVL Trees



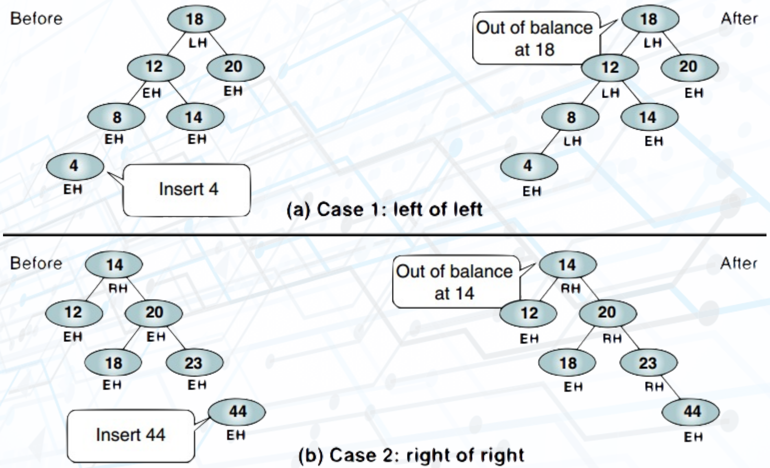
Why AVL Trees?

- When data elements are inserted in a BST in sorted order: 1, 2, 3, ...
  BST becomes a degenerate tree.
  Search operation takes O(n), which is inefficient.
- It is possible that after a number of insert and delete operations, a binary tree may become unbalanced and inscrease in height.
- AVL trees ensure that the complexity of search is $O(log_2 n)$.

AVL Balance

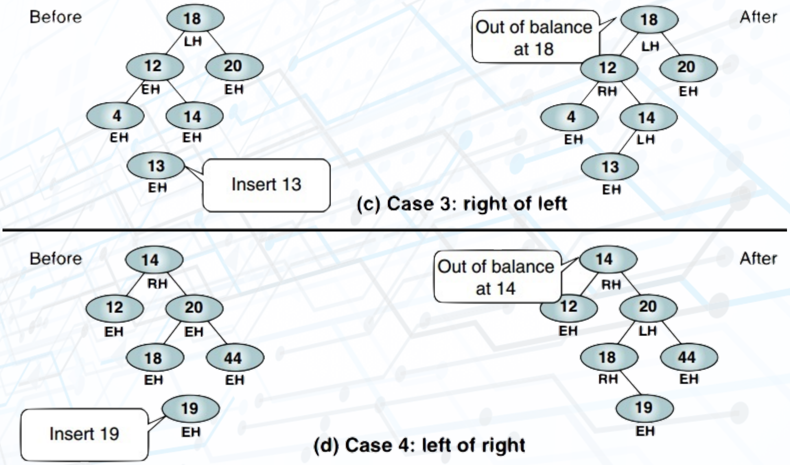
- When we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced.
  $\rightarrow$ rebalance the tree.
- Four unbalanced tree cases:
  - left of left: a subtree of a tree that is left high has also become left high;
  - right of right: a subtree of a tree that is right high has also become right high;
  - right of left: a subtree of a tree that is left high has become right high;
  - left of right: a subtree of a tree that is right high has become left high;

Unbalanced tree cases



(Source: Data Structures - A Pseudocode Approach with C++)

Unbalanced tree cases



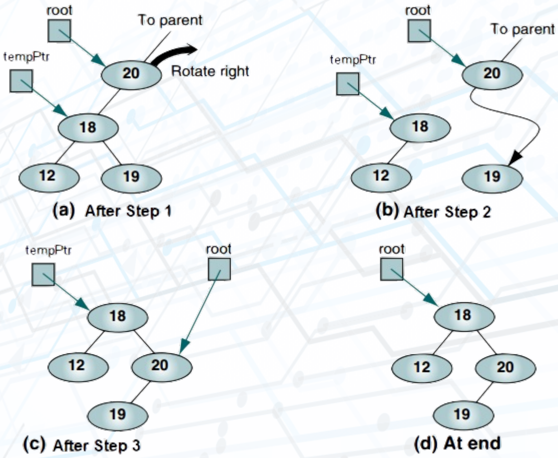
(Source: Data Structures - A Pseudocode Approach with C++)

Algorithm rotateRight(ref root <pointer>)
Exchanges pointers to rotate the tree right.

Pre: `root` is pointer to tree to be rotated
Post: node rotated and root updated

tempPtr = root->left
root->left = tempPtr->right
tempPtr->right = root
Return tempPtr

End rotateRight



(Source: Data Structures - A Pseudocode Approach with C++)

Rotate Left

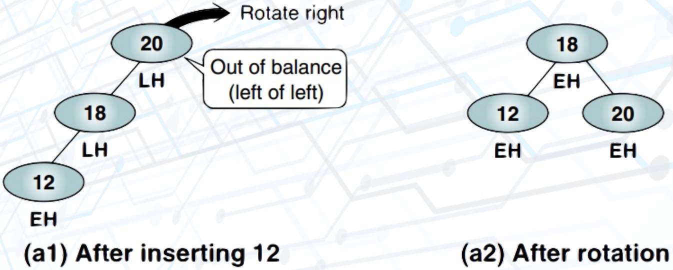
Algorithm rotateLeft(ref root <pointer>)
Exchanges pointers to rotate the tree left.

Pre: `root` is pointer to tree to be rotated
Post: node rotated and root updated

tempPtr = root->right
root->right = tempPtr->left
tempPtr->left = root
Return tempPtr

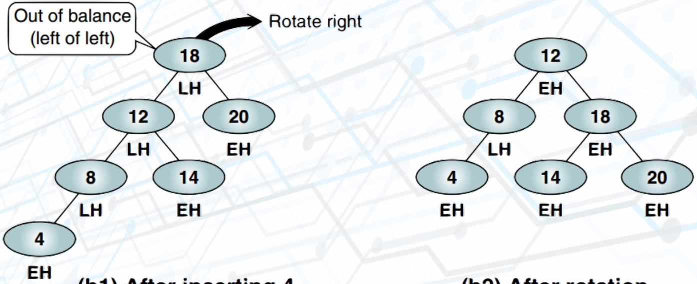
End rotateLeft

Out of balance condition created by a left high subtree of a left high tree
$\rightarrow$ balance the tree by rotating the out of balance node to the right.



(Source: Data Structures - A Pseudocode Approach with C++)

Balancing Trees - Case 1: Left of Left



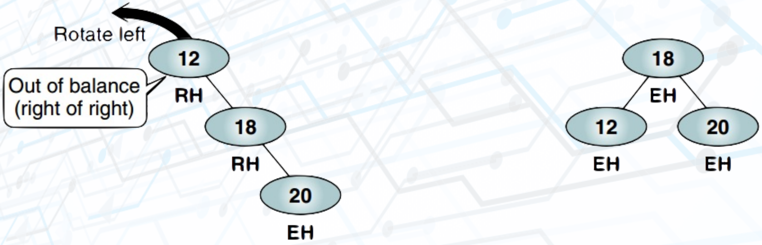
(b1) After inserting 4

(b2) After rotation

(Source: Data Structures - A Pseudocode Approach with C++)

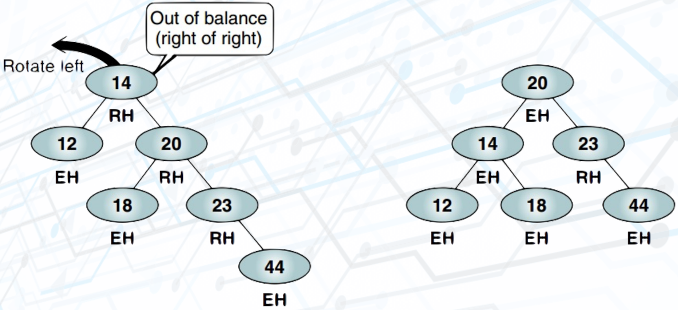
Balancing Trees - Case 2: Right of Right

Out of balance condition created by a right high subtree of a right high tree
$\rightarrow$ balance the tree by rotating the out of balance node to the left.



(Source: Data Structures - A Pseudocode Approach with C++)

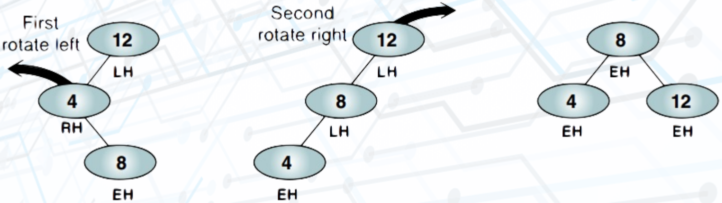
Balancing Trees - Case 2: Right of Right



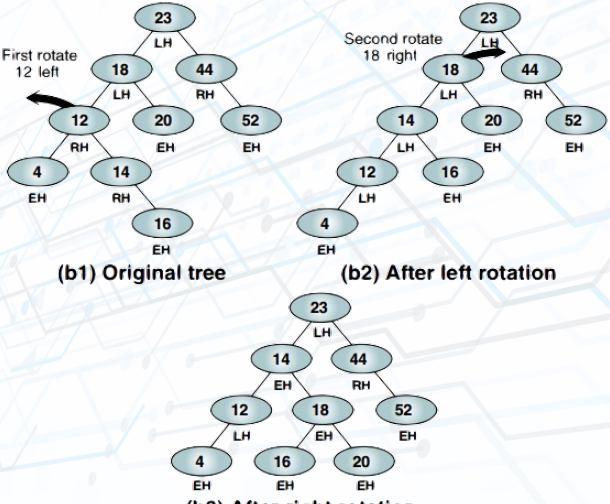
(Source: Data Structures - A Pseudocode Approach with C++)

Out of balance condition created by a right high subtree of a left high tree
→ balance the tree by two steps:

1. rotating the left subtree to the left;
2. rotating the root to the right.



(Source: Data Structures - A Pseudocode Approach with C++)



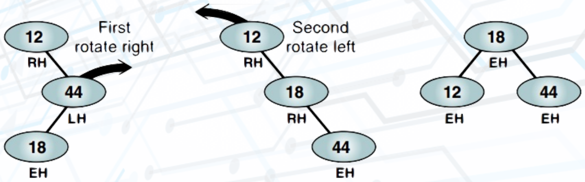
(b1) Original tree

(b2) After left rotation

(b3) After right rotation

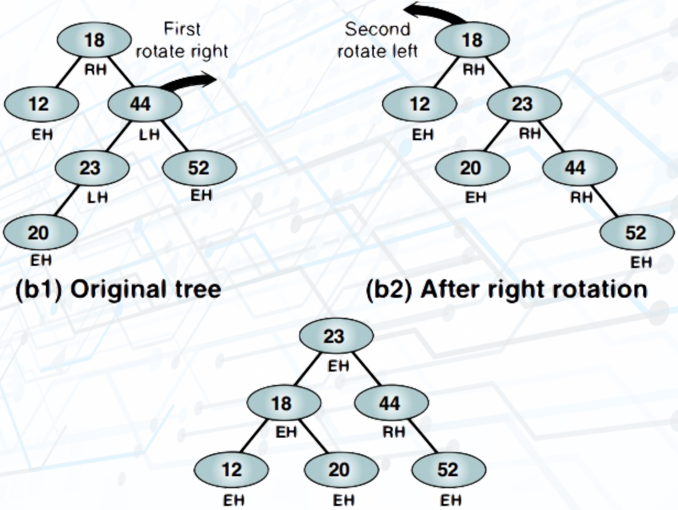
Out of balance condition created by a left high subtree of a right high tree
$\rightarrow$ balance the tree by two steps:

1. rotating the right subtree to the right;
2. rotating the root to the left.



(Source: Data Structures - A Pseudocode Approach with C++)

Balancing Trees - Case 4: Left of Right



(b1) Original tree

(b2) After right rotation

(b3) After left rotation

AVL Tree Operations

```
node
  data <dataType>
  left <pointer>
  right <pointer>
  balance <balance_factor>
end node

avlTree
  root <pointer>
end avlTree
```

```
// General dataTye:
dataType
  key <keyType>
  field1 <...>
  field2 <...>
  ...
  fieldn <...>
end dataType
```

Note: Array is not suitable for AVL Tree.

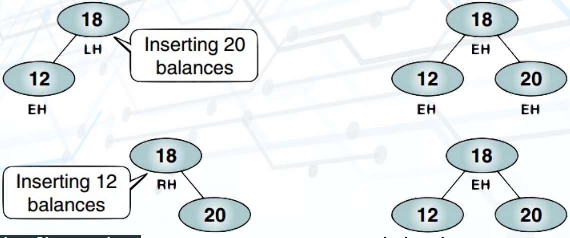
AVL Tree Operations

- Search and retrieval are the same for any binary tree.
- AVL Insert
- AVL Delete

AVL Insert

- Insert can make an out of balance condition.

- Otherwise, some inserts can make an automatic balancing.



AVL Insert Algorithm

Algorithm AVLInsert(ref root <pointer>, val newPtr <pointer>, ref taller <boolean>)
Using recursion, insert a node into an AVL tree.

Pre: `root` is a pointer to first node in AVL tree/subtree
`newPtr` is a pointer to new node to be inserted
Post: `taller` is a Boolean: `true` indicating the subtree height has increased, `false` indicating same height
Return `root` returned recursively up the tree

AVL Insert Algorithm

```
// Insert at root
if root null then
    root = newPtr
    taller = true
    return root
end
```

```
if newPtr->data.key < root->data.key then
    root->left = AVLInsert(root->left, newPtr, taller)
    // Left subtree is taller
    if taller then
        if root is LH then
            root = leftBalance(root, taller)
        else if root is EH then
            root->balance = LH
        else
            root->balance = EH
            taller = false
        end
    end
```

```
else
    root->right = AVLInsert(root->right, newPtr, taller)
    // Right subtree is taller
    if taller then
        if root is LH then
            root->balance = EH
            taller = false
        else if root is EH then
            root->balance = RH
        else
            root = rightBalance(root, taller)
        end
    end
end
return root
```

Algorithm leftBalance(ref root <pointer>, ref taller <boolean>)
This algorithm is entered when the left subtree is higher than the right subtree.

Pre: `root` is a pointer to the root of the [sub]tree
`taller` is `true`
Post: `root` has been updated (if necessary)
`taller` has been updated

AVL Left Balance Algorithm

leftTree = root->left

// Case 1: Left of left. Single rotation right.

if *leftTree is LH* **then**

    root = rotateRight(root)
    root->balance = EH
    leftTree->balance = EH
    taller = false

AVL Left Balance Algorithm

```
else
    rightTree = leftTree->right
    if rightTree->balance = LH then
        root->balance = RH
        leftTree->balance = EH
    else if rightTree->balance = EH then
        leftTree->balance = EH
    else
        root->balance = EH
        leftTree->balance = LH
    end
    rightTree->balance = EH
    root->left = rotateLeft(leftTree)
    root = rotateRight(root), taller = false
end
return root
```

Algorithm rightBalance(ref root <pointer>, ref taller <boolean>)
This algorithm is entered when the right subtree is higher than the left subtree.

Pre: `root` is a pointer to the root of the [sub]tree
`taller` is `true`
Post: `root` has been updated (if necessary)
`taller` has been updated

rightTree = root->right

// Case 1: Right of right. Single rotation left.

if *rightTree is RH* **then**

  root = rotateLeft(root)

  root->balance = EH

  rightTree->balance = EH

  taller = false

AVL Right Balance Algorithm

```
else
    leftTree = rightTree->left
    if leftTree->balance = RH then
        root->balance = LH
        rightTree->balance = EH
    else if leftTree->balance = EH then
        rightTree->balance = EH
    else
        root->balance = EH
        rightTree->balance = RH
    end
    leftTree->balance = EH
    root->right = rotateRight(rightTree)
    root = rotateLeft(root), taller = false
end
return root
```

The AVL delete follows the basic logic of the binary search tree delete with the addition of the logic to balance the tree. As with the insert logic, the balancing occurs as we back out of the tree.

Algorithm AVLDelete(ref root <pointer>, val deleteKey <key>, ref shorter <boolean>, ref success <boolean>)
This algorithm deletes a node from an AVL tree and rebalances if necessary.

Pre: `root` is a pointer to the root of the [sub]tree
`deleteKey` is the key of node to be deleted
Post: node deleted if found, tree unchanged if not found
`shorter` is `true` if subtree is shorter
`success` is `true` if deleted, `false` if not found
Return pointer to root of (potential) new subtree

```
if tree null then
    shorter = false
    success = false
    return null
end
if deleteKey < root->data.key then
    root->left = AVLDelete(root->left, deleteKey, shorter, success)
    if shorter then
        root = deleteRightBalance(root, shorter)
    end
else if deleteKey > root->data.key then
    root->right = AVLDelete(root->right, deleteKey, shorter, success)
    if shorter then
        root = deleteLeftBalance(root, shorter)
    end
```

```
// Delete node found – test for leaf node
else
    deleteNode = root
    if no right subtree then
        newRoot = root->left
        success = true
        shorter = true
        recycle(deleteNode)
        return newRoot
    else if no left subtree then
        newRoot = root->right
        success = true
        shorter = true
        recycle(deleteNode)
        return newRoot
```

AVL Delete Algorithm

```
else
    // ... else
        exchPtr = root->left
        while exchPtr->right not null do
            exchPtr = exchPtr->right
        end
        root->data = exchPtr->data
        root->left = AVLDelete(root->left, exchPtr->data.key, shorter, success)
        if shorter then
            root = deleteRightBalance(root, shorter)
        end
    end
end
Return root
End AVLDelete
```

Algorithm deleteRightBalance(ref root <pointer>, ref shorter <boolean>)
The (sub)tree is shorter after a deletion on the left branch. Adjust the balance factors and if necessary balance the tree by rotating left.

Pre: tree is shorter
Post: balance factors updated and balance restored
`root` updated
`shorter` updated

if *root LH* **then**
    root->balance = EH
else if *root EH* **then**
    root->balance = RH
    shorter = false

Delete Right Balance

```
else
    rightTree = root->right
    if rightTree LH then
        leftTree = rightTree->left
        if leftTree LH then
            rightTree->balance = RH
            root->balance = EH
        else if leftTree EH then
            root->balance = LH
            rightTree->balance = EH
        else
            root->balance = LH
            rightTree->balance = EH
        end
        leftTree->balance = EH
        root->right = rotateRight(rightTree)
```

```
else
    // ...

    else
        if rightTree not EH then
            root->balance = EH
            rightTree->balance = EH
        else
            root->balance = RH
            rightTree->balance = LH
            shorter = false
        end
        root = rotateLeft(root)
    end
end
return root
```

Algorithm deleteLeftBalance(ref root <pointer>, ref shorter <boolean>)
The (sub)tree is shorter after a deletion on the right branch. Adjust the balance factors and if necessary balance the tree by rotating right.

Pre: tree is shorter
Post: balance factors updated and balance restored
`root` updated
`shorter` updated

if *root RH* **then**
  root->balance = EH
else if *root EH* **then**
  root->balance = LH
  shorter = false

```
else
    leftTree = root->left
    if leftTree RH then
        rightTree = leftTree->right
        if rightTree RH then
            leftTree->balance = LH
            root->balance = EH
        else if rightTree EH then
            root->balance = RH
            leftTree->balance = EH
        else
            root->balance = RH
            leftTree->balance = EH
        end
        rightTree->balance = EH
        root->left = rotateLeft(leftTree)
```

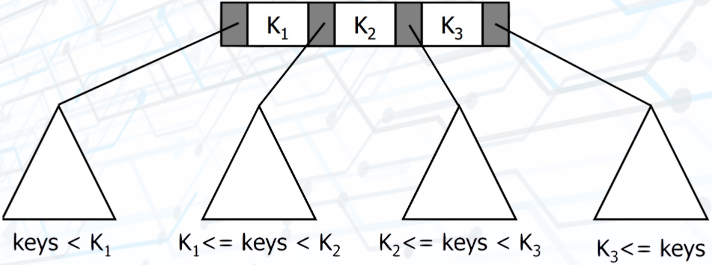
Delete Left Balance

```
else
    // ... else
        if leftTree not EH then
            root->balance = EH
            leftTree->balance = EH
        else
            root->balance = LH
            leftTree->balance = RH
            shorter = false
        end
        root = rotateRight(root)
    end
end
return root
End deleteLeftBalance
```

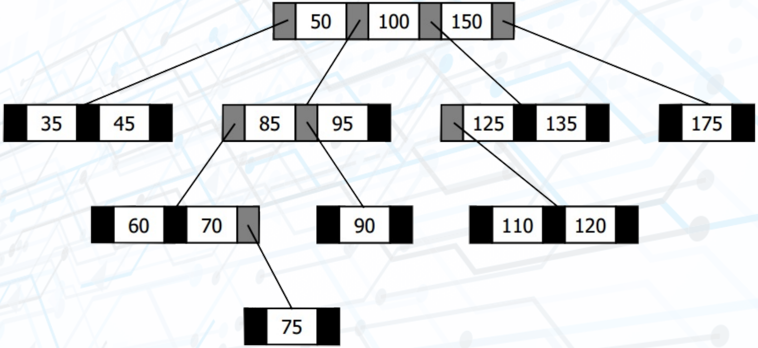
Multiway Trees

Tree whose outdegree is not restricted to 2 while retaining the general properties of binary search trees.

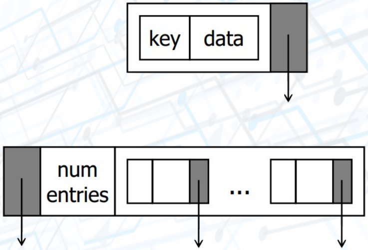
- Each node has `m - 1` data entries and `m` subtree pointers.
- The key values in a subtree such that:
  - $\geq$ the key of the left data entry
  - $<$ the key of the right data entry.



M-Way Search Trees



M-Way Node Structure



entry
    key <key type>
    data <data type>
    rightPtr <pointer>
end entry

node
    firstPtr <pointer>
    numEntries <integer>
    entries <array[1 .. m-1] of entry>
end node

B-Trees

- M-way trees are unbalanced.
- Bayer, R. & McCreight, E. (1970) created B-Trees.

A B-tree is an m-way tree with the following additional properties ($m \geq 3$):

- The root is either a leaf or has at least 2 subtrees.
- All other nodes have at least $\lceil m/2 \rceil - 1$ entries.
- All leaf nodes are at the same level.

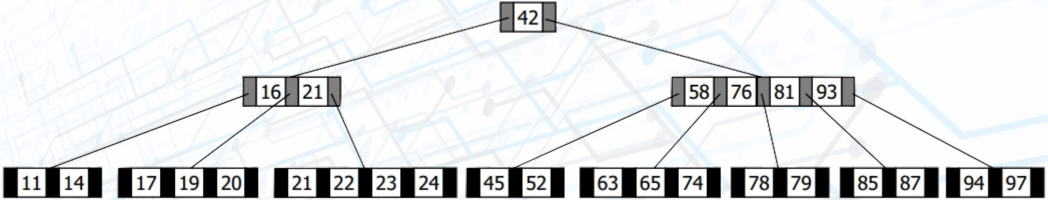
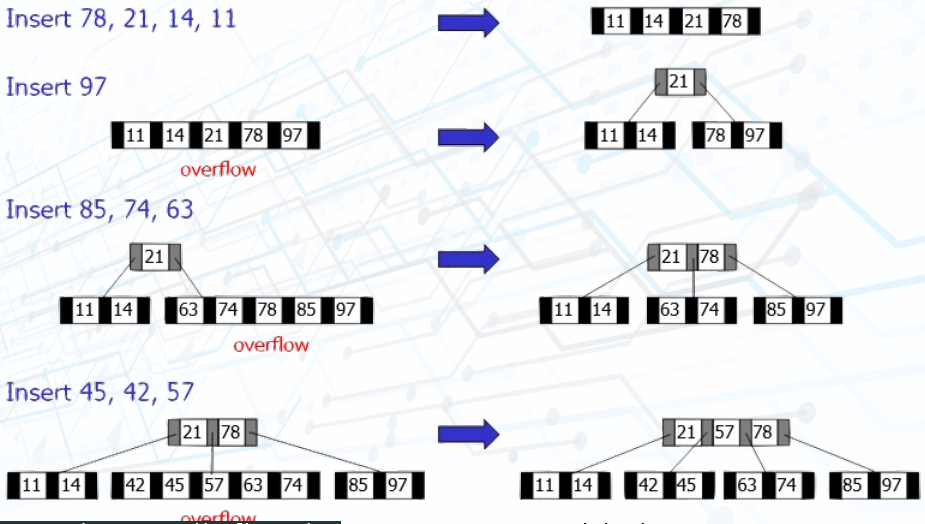

42
16
21
58
76
81
93
11
14
17
19
20
21
22
23
24
45
52
63
65
74
78
79
85
87
94
97

Figure 1: $m = 5$

B-Tree Insertion

- Insert the new entry into a leaf node.
- If the leaf node is overflow, then split it and insert its median entry into its parent.

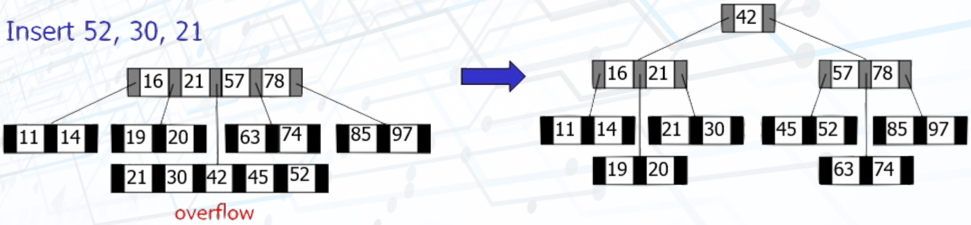
B-Tree Insertion



B-Tree Insertion

Insert 20, 16, 19

Insert 52, 30, 21



Algorithm BTreeInsert(ref root <pointer>, val data <record>)

Inserts data into B-tree. Equal keys placed on right branch.

Pre: `root` is a pointer to the B-tree. May be `null`.

Post: `data` inserted

Return pointer to B-tree root.

taller = insertNode(root, data, upEntry)

if *taller* **then**

  // Tree has grown. Create new root.

  allocate(newPtr)

  newPtr−>entries[1] = upEntry

  newPtr−>firstPtr = root

  newPtr−>numEntries = 1

  root = newPtr

end

return root

Algorithm insertNode (ref root <pointer>, val data <record>, ref upEntry <entry>)
Recursively searches tree to locate leaf for data. If node overflow, inserts median key's data into parent.
Pre: `root` is a pointer to tree or subtree. May be `null`.
Post: `data` inserted
`upEntry` is overflow entry to be inserted into parent.
Return tree taller <boolean>.

if *root null* **then**
    upEntry.data = data
    upEntry.rightPtr = null
    taller = true

B-Tree Insertion

```
else
    entryNdx = searchNode(root, data.key)
    if entryNdx > 0 then
        subTree = root−>entries[entryNdx].rightPtr
    else
        subTree = root−>firstPtr
    end
    taller = insertNode(subTree, data, upEntry)
    if taller then
        if node full then
            splitNode(root, entryNdx, upEntry), taller = true
        else
            insertEntry(root, entryNdx, upEntry), taller = false
            root−>numEntries = root−>numEntries + 1
        end
    end
end
return taller
End insertNode
```

Algorithm searchNode(val nodePtr <pointer>, val target <key>)
Search B-tree node for data entry containing key <= target.
Pre: `nodePtr` is pointer to non-null node.
`target` is key to be located.
Return index to entry with key <= target.
0 if key < first entry in node

```
if target < nodePtr−>entry[1].data.key then
    walker = 0
else
    walker = nodePtr−>numEntries
    while target < nodePtr−>entries[walker].data.key do
        walker = walker - 1
    end
end
return walker
```

Algorithm splitNode(val node <pointer>, val entryNdx <index>, ref upEntry <entry>)

Node has overflowed. Split node. No duplicate keys allowed.

Pre: `node` is pointer to node that overflowed.

`entryNdx` contains index location of parent.

`upEntry` contains entry being inserted into split node.

Post: `upEntry` now contains entry to be inserted into parent.

minEntries = minimum number of entries

allocate (rightPtr)

// Build right subtree node

if *entryNdx <= minEntries* **then**

  fromNdx = minEntries + 1

else

```
else
|   fromNdx = minEntries + 2
end
toNdx = 1
rightPtr->numEntries = node->numEntries – fromNdx + 1
while fromNdx <= node->numEntries do
|   rightPtr->entries[toNdx] = node->entries[fromNdx]
|   fromNdx = fromNdx + 1
|   toNdx = toNdx + 1
end
node->numEntries = node->numEntries−rightPtr->numEntries
if entryNdx <= minEntries then
|   insertEntry(node, entryNdx, upEntry)
else
```

B-Tree Insertion

```
else
    insertEntry(rightPtr, entryNdx−minEntries, upEntry)
    node->numEntries = node->numEntries− 1
    rightPtr->numEntries = rightPtr->numEntries + 1
end
// Build entry for parent
medianNdx = minEntries + 1
upEntry.data = node->entries[medianNdx].data
upEntry.rightPtr = rightPtr
rightPtr->firstPtr = node->entries[medianNdx].rightPtr
return
End splitNode
```

B-Tree Insertion

Algorithm insertEntry(val node <pointer>, val entryNdx <index>, val newEntry <entry>)

Inserts one entry into a node by shifting nodes to make room.

Pre: `node` is pointer to node to contain data.

`entryNdx` is index to location for new data.

`newEntry` contains data to be inserted.

Post: data has been inserted in sequence.

shifter = node->numEntries + 1

while *shifter > entryNdx + 1* **do**

  node->entries[shifter] = node->entries[shifter - 1]

  shifter = shifter - 1

end

node->entries[shifter] = newEntry

node->numEntries = node->numEntries + 1

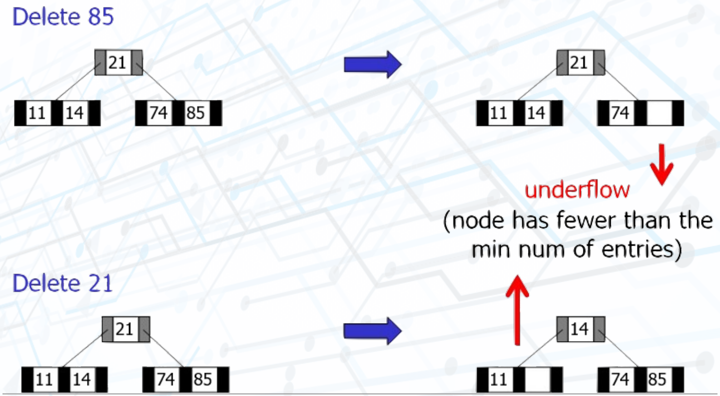
return

- It must take place at a leaf node.
- If the data to be deleted are not in a leaf node, then replace that entry by the largest entry on its left subtree.

Delete 78

Delete 63

B-Tree Deletion



Reflow

For each node to have sufficient number of entries:

- Balance: shift data among nodes.
- Combine: join data from nodes.

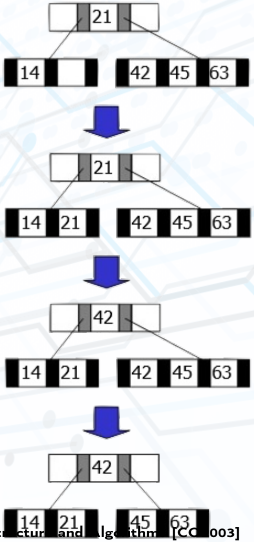
Borrow from right

Original node

Rotate parent data down

Rotate data to parent

Shift entries left



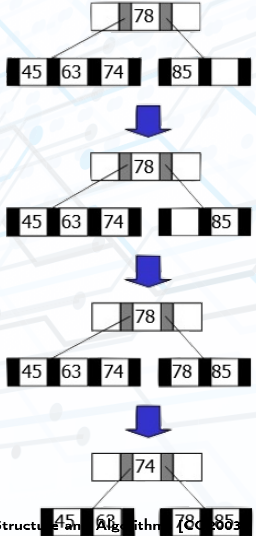
Borrow from left

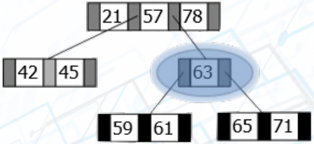
Original node

Shift entries right

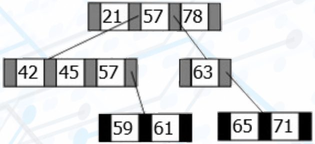
Rotate parent data down

Rotate data up

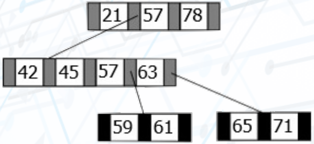




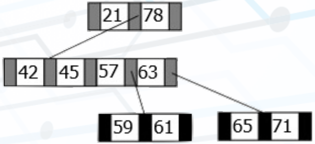
1. After underflow



2. After moving root to subtree

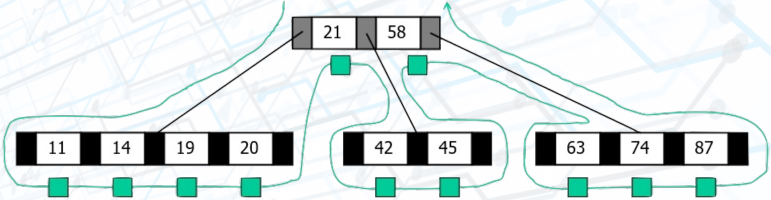


3.After moving right entries



4. After shifting root

B-Tree Traversal



B-Tree Traversal

Algorithm BTreeTraversal (val root <pointer>)

Processes tree using inorder traversal.

Pre: `root` is pointer to B-Tree.

Post: Every entry has been processed in order.

```
scanCount = 0, ptr = root−>firstPtr
while scanCount <= root−>numEntries do
    if ptr not null then
        BTreeTraversal(ptr)
    end
    scanCount = scanCount + 1
    if scanCount <= root−>numEntries then
        process (root−>entries[scanCount].data)
        ptr = root−>entries[scanCount].rightPtr
    end
end
```

Algorithm BTreeSearch(val root <pointer>, val target <key>, ref node <pointer>, ref entryNo <index>)

Recursively searches a B-tree for the target key.

Pre: `root` is pointer to a tree or subtree

`target` is the data to be located

Post:

if found – –

node is pointer to located node

entryNo is entry within node

if not found – –

node is null and entryNo is zero

Return found <boolean>

B-Tree Search

```
if target < first entry then
    return BTreeSearch (root->firstPtr, target, node, entryNo)
else
    entryNo = root->numEntries
    while target < root->entries[entryNo].data.key do
        entryNo = entryNo - 1
    end
    if target = root->entries[entryNo].data.key then
        found = true
        node = root
    else
        return BTreeSearch (root->entries[entryNo].rightPtr, target, node, entryNo)
    end
end
return found
End BTreeSearch
```

- B*Tree: the minimum number of (used) entries is two thirds.
- B+Tree:
  - Each data entry must be represented at the leaf level.
  - Each leaf node has one additional pointer to move to the next leaf node.