AVL Trees
Dynamic Tree Balancing
Problems with BST

- With random insertions and deletions BST has $\Theta(\log N)$ times for search, insert and remove
- But worst case behaviour is $\Theta(N)$
- Problem is that BST’s can become unbalanced
- We need a **rebalance operation** on a BST to restore the balance property and regain $\Theta(\log N)$
- Rebalancing should be cheap enough that we could do it **dynamically** on every insert and remove
  - Preference is to have $\Theta(1)$ rebalance time
AVL Balance Definition

• A good balance conditions ensures the height of a tree with N nodes is $\Theta(\log N)$
  
  » That gives $\Theta(\log N)$ performance

• The following balance definition is used
  
  » The empty tree is balanced
  
  » For every node in a non-empty tree
  
  $|\text{height (left_sub_tree)} - \text{height (right_sub_tree)}| \leq 1$
Rebalancing

• Restructure the tree by moving the nodes around while preserving the order property

• The operation is called a **rotation**
  
  » Make use of the property that a node has one parent and two direct descendents
Single Rotations

LL rotation

RR rotation

Keys in A < K2
K2 < Keys in B < K1
K1 < Keys in C

Relationship to parent does not change

© Gunnar Gotshalks
Single LL Rotation Pseudocode

// Return pointer to root after rotation

rotate_LL ( oldRoot : Node ) : Node is
    Result ← oldRoot . left
    oldRoot . left ← Result . right
    Result . right ← oldRoot
    adjustHeight(old_root)
    adjustHeight(old_root.left)
    adjustHeight(Result)
end

// Example use of rotate_LL

parent . left ← rotate_LL ( parent . left)
parent . right ← rotate_LL ( parent . right)

Exercise
write rotate_RR
AdjustHeight Pseudocode

// Assume that every node contains a height attribute

adjustHeight ( root : Node ) is
  if root ≠ null then
    root . height ← 1 + max ( height ( root . left ) , height ( root . right ) )
  end
end
Single Rotations & Height

LL rotation

RR rotation

h(K1) = 1 + max (h(K2), h(C))
h(K2) = 1 + max (h(A), h(B))

If h(A) > h(B) & h(B) ≥ h(C) then rotate_LL reduces the height of the root

h(K2) = 1 + max (h(K1), h(A))
h(K1) = 1 + max (h(B), h(C))

If h(C) > h(B) & h(B) ≥ h(A) then rotate_RR reduces the height of the root
Single Rotations & Height – 2

\[ h(K1) = 1 + \max \left( h(K2), h(C) \right) \]
\[ h(K2) = 1 + \max \left( h(A), h(B) \right) \]

if \( h(A) > h(B) \land h(B) \geq h(C) \)
then \textit{rotate\_LL} reduces the height of the root

Proof – before rotation

\[ h(K2) = 1 + h(A) \]
\[ -- h(A) > h(B) \]
\[ h(K1) = 1 + h(K2) \]
\[ -- h(K2) > h(B) \geq h(C) \]
\[ h(K1) = 2 + h(A) \]

Before rotation \( h(\text{root}) = 2 + h(A) \)

After rotation \( h(\text{root}) = 1 + h(A) \)

Height of root has been reduced
if \( h(A) > h(B) \land h(B) \geq h(C) \)
then rotate\_LL reduces the height of the root

Proof (?) by diagram
Double Rotation – LR

Keys in A < K2
K2 < Keys in B1 < K3
K3 < Keys in B2 < K1
K1 < Keys in C

Relationship to parent does not change
Double Rotation – LR – Height

If \( h(K3) > h(A) \land h(A) \geq h(C) \) then rotate_LR reduces the height of root
Double Rotation – RL

Keys in A < K2
K2 < Keys in B1 < K3
K3 < Keys in B2 < K1
K1 < Keys in C

Relationship to parent does not change
Double Rotation – RL – Height

If \( h(K3) > h(C) \) \( \land \) \( h(C) \geq h(A) \)
then rotate_RL reduces the height of root
// Return pointer to root after rotation

rotate_RL (oldRoot : Node) : Node is
  rightChild ← oldRoot . right ; Result ← rightChild . left
  oldRoot . right ← Result . left ; rightChild . left ← Result . right
  Result . left ← oldRoot ; Result . right ← rightChild

adjustHeight (oldRoot)
adjustHeight (rightChild)
adjustHeight (Result)

end

// Example use of rotate_RL

parent . left ← rotate_RL (parent . left)
parent . right ← rotate_RL (parent . right)

Exercise
write rotate_LR
// Insert will do rotations, which changes the root of
// sub-trees. As a consequence, the recursive insert must
// return the root of the resulting sub-tree.

insert ( key : KeyType , data : ObjectType ) is
    newNode ← new Node ( key , data )
    root ← insertRec ( root , newNode )
    root ← rebalance ( root )     // Insertion may change
    adjustHeight ( root )         // height, which may
    // cause imbalance
end

Only one rebalance will occur but we do not know where
InsertRec Pseudocode

// Insert may do rotations, which changes the root of
// sub-trees. As a consequence, the recursive insert must
// return the root of the resulting sub-tree.

// Invariant – The tree rooted at root is balanced

insertRec ( root : Node , newNode : Node ) : Node is
  if root = Void then Result ← newNode
  else if root . key > newNode . key
    then root . left ← insertRec ( root . left , newNode )
    else root . right ← insertRec ( root . right , newNode )
  fi
  Result ← rebalance ( root ) ; adjustHeight ( Result )
  fi
end
// Assume that every node contains a height attribute

// Different definition for height for AVL trees.
// Height of leaf is 1 (Figure 10.10 p435) not 0 (page 273).
// By implication height of empty tree is 0 (see slides
// Tree Algorithms–11..15 on binary tree height).

height ( root : Node ) : Integer is
   if node = Void then Result ← 0
   else Result ← node . Height
   fi
   return
end
Rebalance Pseudocode

- Define 6 variables that have the height of the sub-trees of interest for rotations
  - If any of the pointers are void, height 0 is returned

![Diagram of AVL tree with variable definitions for LL and LR rotations.

- h_AL = height of subtree A
- h_BL = height of subtree B
- h_CL = height of subtree C

© Gunnar Gotshalks
Rebalance Pseudocode – 2

- Have the symmetric cases for the other 3 height variables
rebalance ( root : Node ) : Node is
  h_AL ← heightLL ( root ) ;  h_AR ← heightRR ( root )
  h_BL ← heightLR ( root ) ;  h_BR ← heightRL ( root )
  h_CL ← height( root . right) ;  h_CR ← height ( root . left)

  if       h_AL = h_BL  ∧  h_BL ≥ h_CL then Result ← rotate_LL ( root )
  elseif  h_AR = h_BR  ∧  h_BR ≥ h_CR then Result ← rotate_RR ( root )
  elseif  h_BL = h_AL  ∧  h_AL ≥ h_CL then Result ← rotate_LR ( root )
  elseif  h_BR = h_AR  ∧  h_AR ≥ h_CR then Result ← rotate_RL ( root )
  else Result ← root
fi
end

This follows the mathematical development in slides 8, 12, 14 and works correctly for insertion where the objective is to reduce the height of a subtree. See slides 29..32 for problems with remove.
Remove Difficulties

• Remove has to do two things
  » **Return the entry corresponding to the key**
  » **Rebalance the tree**
    > Means adjusting the pointers
    > Without a parent pointer, the path from the root to the node is a singly linked list
    > Need to keep track of the parent node of the root of the sub-tree to rebalance to adjust the pointer to the new sub-tree
    > Consequence is every step we have to look one level deeper than BST remove algorithm

• Rebalancing may occur at all levels
Remove Pseudocode

remove ( key : KeyType ) : EntryType is
    if root = Void then Result ← Void  // Entry not in tree
    elseif root . key = key then  // Root is a special case
        Result ← root . entry
        root ← removeNode ( root )
    else Result ← removeRec ( root , key )  // Try sub-trees
    fi

    // The following routines need look ahead. They are the
    // main change from BST remove.

    adjustHeight ( root )
    root ← rebalance ( root )
end
RemoveRec Pseudocode

// Require root ≠ null ∧ root .key ≠ key
// entry ∈ tree → entry ∈ root
// balanced ( tree (root ) )
// Ensure entry ∈ tree → Result = entry
// entry ∉ tree → Result = Void
// tree ( root ) may be unbalanced

removeRec ( root : Node , key : KeyType ) : EntryType is
if root . key > key then
   // Remove from the left sub-tree
else
   // Remove from the right sub-tree
fi
return
end
// Remove from the left sub-tree

if root . left = Void then Result ← Void
elseif root . left . key = key then
    Result ← root . left . entry
    root . left ← removeNode ( root . left )
else
    Result ← removeRec ( root . left , key )
    adjustHeight( root . left )
    root . left ← rebalance ( root . left )
fi
end
// Remove from the right sub-tree

if root . right = Void then Result ← Void
elseif root . right . key = key then
    Result ← root . right . entry
    root . right ← removeNode ( root . right )
else
    Result ← removeRec ( root . right, key )
    adjustHeight ( root . right )
    root . right ← rebalance ( root . right )
fi
end
// Require root ≠ Void
// Ensure Result is a balanced tree with root removed
Result = replacement node

removeNode ( root : Node ) : Node
    if root . left = Void then Result ← root . right
    elseif root . right = Void then Result ← root . Left
    else child ← root . left
        if child . right = Void then
            root . entry ← child . entry ; root . left ← child . left
        else root . left ←
            swap_and_remove_left_neighbour ( root , child )
        fi
        adjustHeight ( root )
    Result ← rebalance ( root )
    fi
end
Swap and Remove Left Neighbour

// Require child . right ≠ Void
// Ensure Result is a balanced tree with node removed
    Result = replacement node

swap_and_remove_left_neighbour ( parent , child : Node ) : Node

    if child . right . right ≠ Void then
        child . right ←
            swap_and_remove_left_neighbour ( parent , child . right )
    else
        parent . entry ← child . right . entry
        child . right ← child . right . left
    fi

    adjustHeight ( parent )
    Result ← rebalance ( parent )

end
Problem with Rebalance Pseudocode

- The pseudocode for rebalance in slide 21 is works correctly for inserting a node into an AVL tree.

  » But the pseudocode fails for the following remove example

![AVL tree before and after removal of node 21]
Problem with Rebalance Pseudocode

- What is the problem?
  - The case cannot occur on insertion – inserting 17 or 19 invokes a rebalance
  - Need to rebalance but the height will not change

Remove 21
Correct removal with rebalance is the following:

1. Remove 21
2. Rotate LR
3. Remove 21
4. Rotate LL
Rebalance Pseudocode Revised – 3

- Correct rebalance needs to have the following changes

  » Does the height of left and right sub-trees differ by more than 1?
    > If so, then continue rebalance.

  » The condition \( h(A) > h(B) \) does not hold (slide 8)
    > Need to change to \( h(A) \geq h(B) \)
      – If \( h(A) = h(B) \) then either rotateLL or rotateLR will restore balance but not change the height
Rebalance Pseudocode for Remove

rebalance ( root : Node ) : Node is
    h_AL ← heightLL ( root ) ;  h_AR ← heightRR ( root )
    h_BL ← heightLR ( root ) ;  h_BR ← heightRL ( root )
    h_CL ← height ( root . right) ;  h_CR ← height ( root . left)

    if      h_AL ≥ h_BL ∧ h_BL ≥ h_CL then Result ← rotate_LL ( root )
    elseif h_AR ≥ h_BR ∧ h_BR ≥ h_CR then Result ← rotate_RR ( root )
    elseif h_BL ≥ h_AL ∧ h_AL ≥ h_CL then Result ← rotate_LR ( root )
    elseif h_BR ≥ h_AR ∧ h_AR ≥ h_CR then Result ← rotate_RL ( root )
    else Result ← root
    fi
end

Note the ≥ instead of = to handle cases for remove.