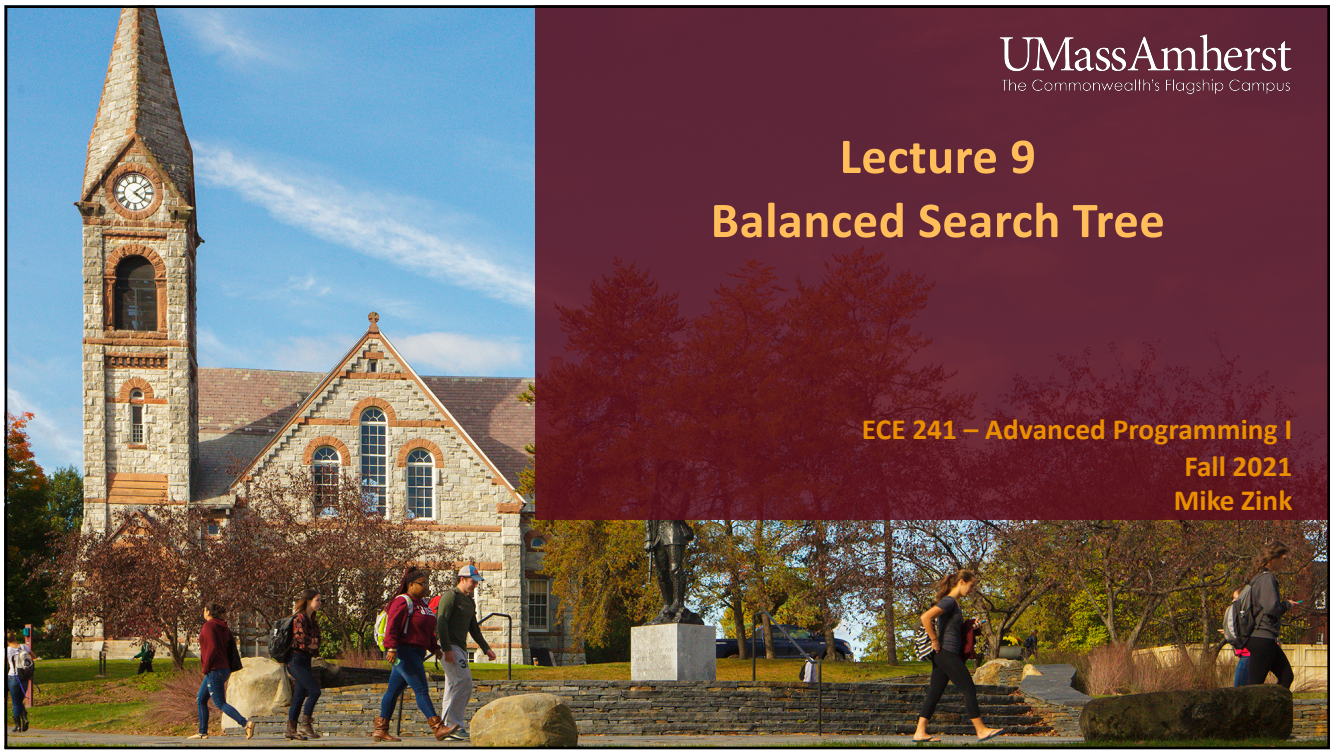


0

UMassAmherst

Overview

- Balance Binary Search Tree

1

UMassAmherst

Objective

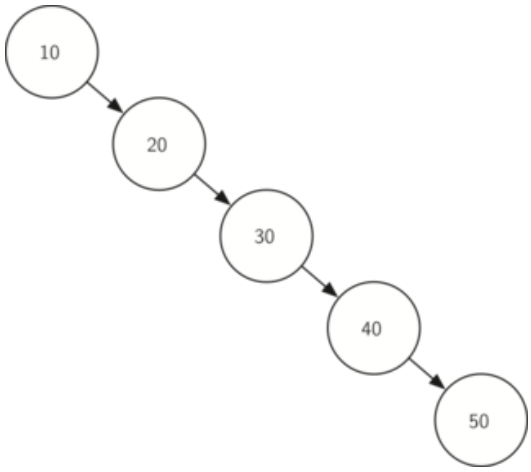
- Understand the principles of binary search trees that assure that tree remains balanced at all times

2

UMassAmherst

Binary Search Tree Problem

- Unfortunately, search tree of height *n* can be constructed by inserting keys in sorted order
- In this case, performance of `put` method is *O(n)*
- Similar for `get,in,del`



3

UMassAmherst

Balanced Binary Search Tree

- Special kind of binary search tree
- Automatically assures that tree remains balanced at all times
- Tree is called AVL tree, names after inventors: G. M. Adelson-Velskii and E. M. Landis

UMassAmherst

Balanced Binary Search Tree

- AVL tree implements Map ADT just like regular binary search tree
- Difference lies in its performance
- Need to keep track of balance:
  - Height of left and right subtree of each node

balanceFactor = height(leftSubTree) – height(rightSubTree)

UMassAmherst

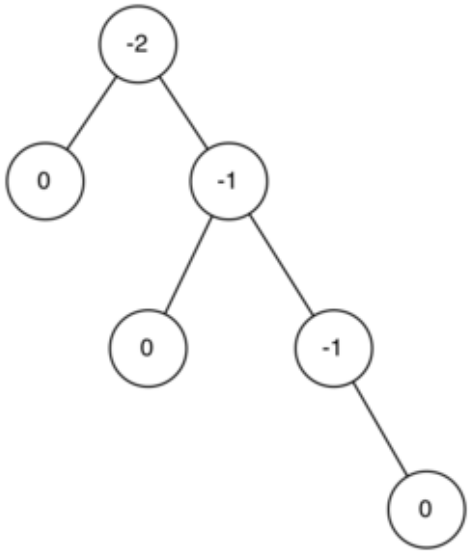
Balanced Binary Search Tree

- Left-heavy: Balance factor > 0
- Right-heavy: Balance factor < 0
- Perfectly in balance: Balance factor = 0
- Definitions: Tree is balanced if balance factor is -1, 0, or 1
- Outside that range tree needs to be brought back in balance

UMassAmherst

Balanced Binary Search Tree

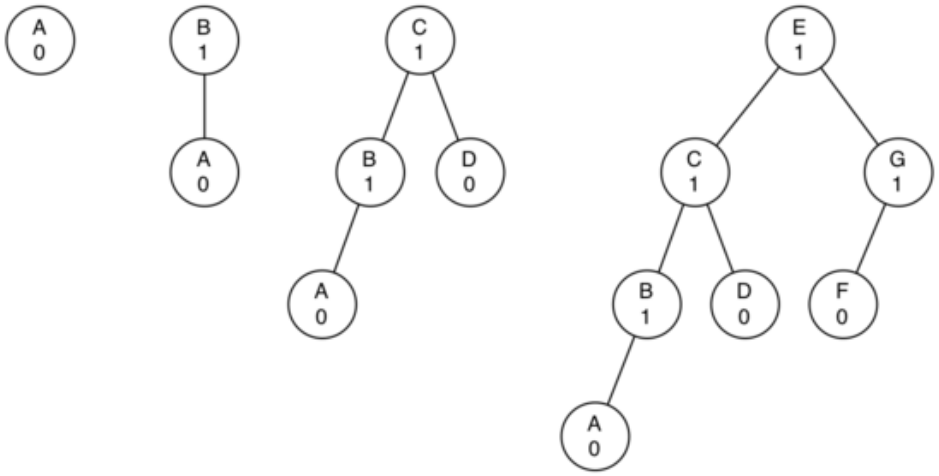
- Unbalance, right-heavy tree
- Balance factor at each node



UMassAmherst

AVL Tree Performance

- Most unbalanced (worst case) left-heavy tree



UMassAmherst

AVL Tree Performance

- Number of nodes in tree:

Height	Nodes
0	1
1	1 + 1 = 2
2	1 + 1 + 2 = 4
3	1 + 2 + 4 = 7

- General: $N_h = 1 + N_{h-1} + N_{h-2}$

UMassAmherst

AVL Tree Performance

- Similarity to Fibonacci sequence:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_i = F_{i-1} + F_{i-2}$ for all $i \geq 2$
- With "golden ratio": $F_i = \phi^i/\sqrt{5}$

UMassAmherst

AVL Tree Performance

- With approximation: $N_h = F_{h+2} - 1, h \geq 1$
- $N_h = \frac{\phi^{h+2}}{\sqrt{5}} - 1$
- $logN_h + 1 = (H+2)log\phi - \frac{1}{2}log5$
- $h = \frac{logN_h + 1 - 2log\phi + \frac{1}{2}log5}{log\phi}$
- $h = 1.44logN_h$ => $O(log\ N)$

UMassAmherst

AVL Tree Implementation

- Base implementation on binary search tree:
  - New keys will be added as leaf nodes
    - Balance factor for leaf node = 0
  - Must update balance factor for parent
  - If new node == right child balance factor of parent reduced by 1
  - If new node == left child balance factor of parent increased by 1

UMassAmherst

AVL Tree Implementation

- relation can be applied recursively to the grandparent of new node
- updating balance factors:
  - The recursive call has reached the root of the tree.
  - The balance factor of the parent has been adjusted to zero. Once balance factor is zero, balance of its ancestor nodes does not change.

UMassAmherst

AVL Tree Implementation

- Implement the AVL tree as a subclass of `BinarySearchTree`.
- override the `_put` method
- new `updateBalance` helper method.

UMassAmherst

AVL Tree Implementation

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
       else:
                currentNode.leftChild =
TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.leftChild)
   else:
        if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
       else:
                currentNode.rightChild =
TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)
```

UMassAmherst

AVL Tree Implementation

- updateBalance method is where most of the work is done.
- updateBalance first checks if current node is out of balance enough to require rebalancing
- If current node does not require rebalancing => balance factor of parent is adjusted
- If balance factor of the parent is non-zero then the algorithm continues

UMassAmherst

AVL Tree Implementation

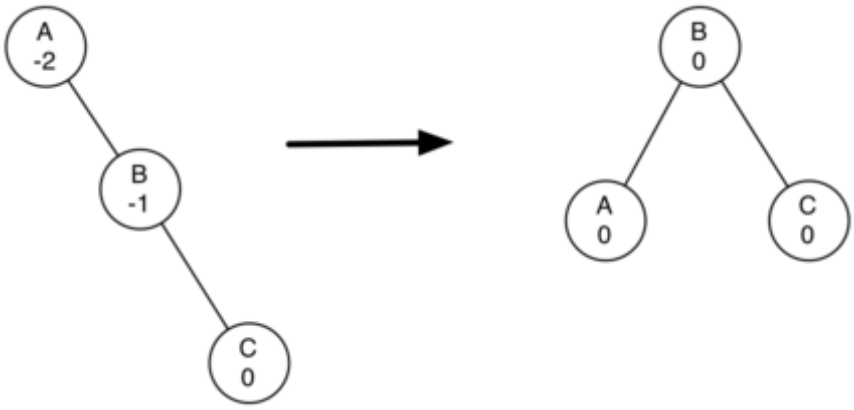
```
def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
      return
    if node.parent != None:
        if node.isLeftChild():
            node.parent.balanceFactor += 1
        elif node.isRightChild():
            node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
            self.updateBalance(node.parent)
```

UMassAmherst

AVL Tree Rebalancing

- How to perform rebalancing
- => **rotations** on the tree



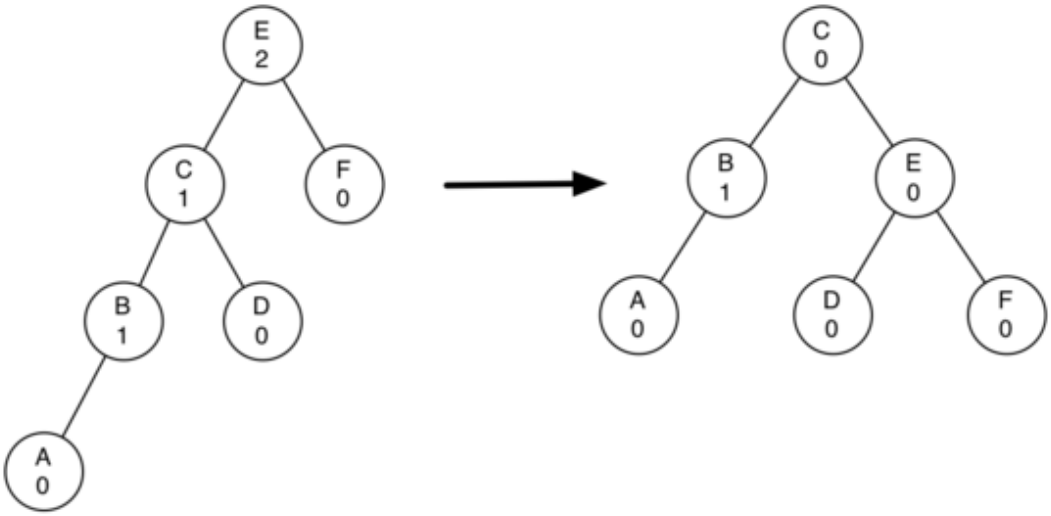
UMassAmherst

AVL Tree Left Rotation

- Promote right child (B) to be root of subtree
- Move old root (A) to be left child of new root
- If new root (B) already had left child then make it right child of new left child (A)
- While procedure if fairly easy in concepts, implementation is tricky

UMassAmherst

AVL Tree Right Rotation



UMassAmherst

AVL Tree Right Rotation

- Promote the child (C) to be root of subtree
- Move old root (E) to be right child of new root
- If new root (C) already had a right child (D) then make it left child of new right child (E)

UMassAmherst

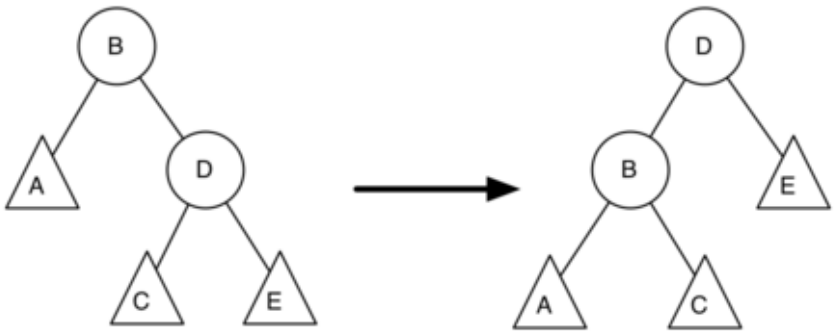
AVL Tree Rotate Implementation

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
            rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 +
max(rotRoot.balanceFactor, 0)
```


UMassAmherst

AVL Tree Balance Factors

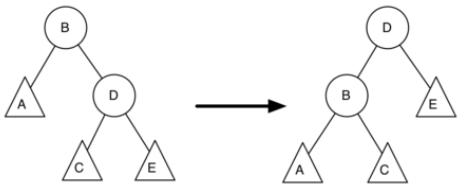
- How to update balance factors without completely recalculating heights of new subtrees?



UMassAmherst

AVL Tree Balance Factors

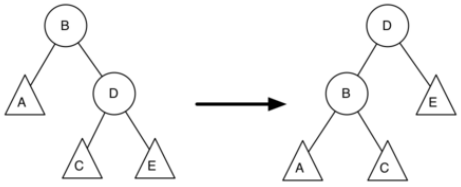
- B and D are pivotal nodes; A, C, E are their subtrees
- Let h_x be height at subtree rooted at node x:
  - $newBal(B) = h_A - h_C$
  - $oldBal(B) = h_A - h_D$



UMassAmherst

AVL Tree Balance Factors

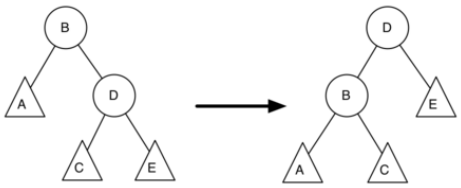
- D can also be given by $1 + max(h_C, h_E)$
- h_C and h_E *have not changed*
- => $oldBal(B) = h_A - (1 + max(h_C, h_E))$



UMassAmherst

AVL Tree Balance Factors

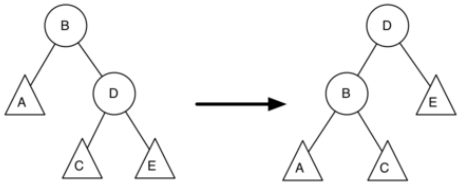
- $newBal(B) - oldBal(B) = h_A - h_C - h_A - (1 + max(h_C, h_E))$
- $newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))$
- $newBal(B) - oldBal(B) = (1 + max(h_C, h_E)) - h_C$



UMassAmherst

AVL Tree Balance Factors

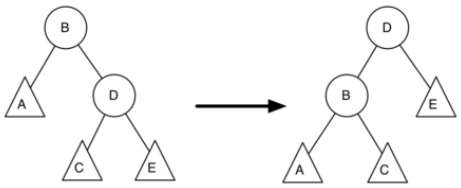
- With $max(a,b) - c = max(a - c, b - c)$
- $newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))$
- $newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$



UMassAmherst

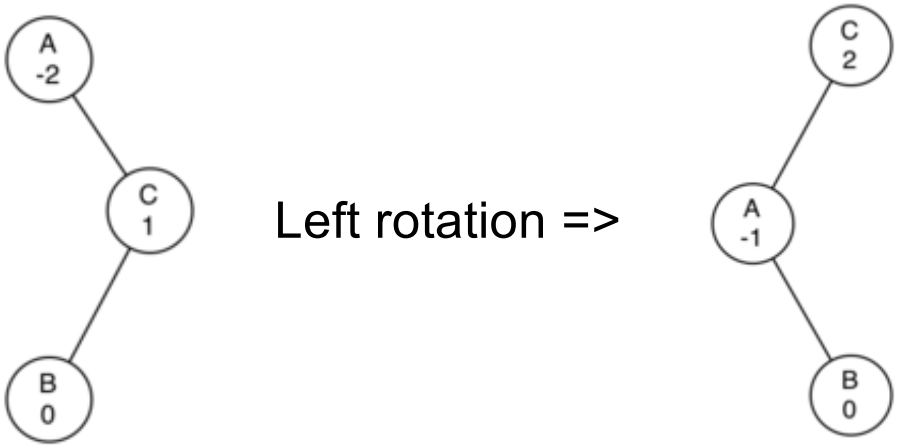
AVL Tree Balance Factors

- Since $h_E - h_C$ = - *oldBal(D)* and *max(-a,-b) = -min(a,b)*
- *newBal(B) = oldBal(B)* + 1 + *max(0, - oldBal(D))*
- *newBal(B) = oldBal(B)* + 1 - *min(0, oldBal(D))*



UMassAmherst

AVL Tree - Not Done Yet



UMassAmherst

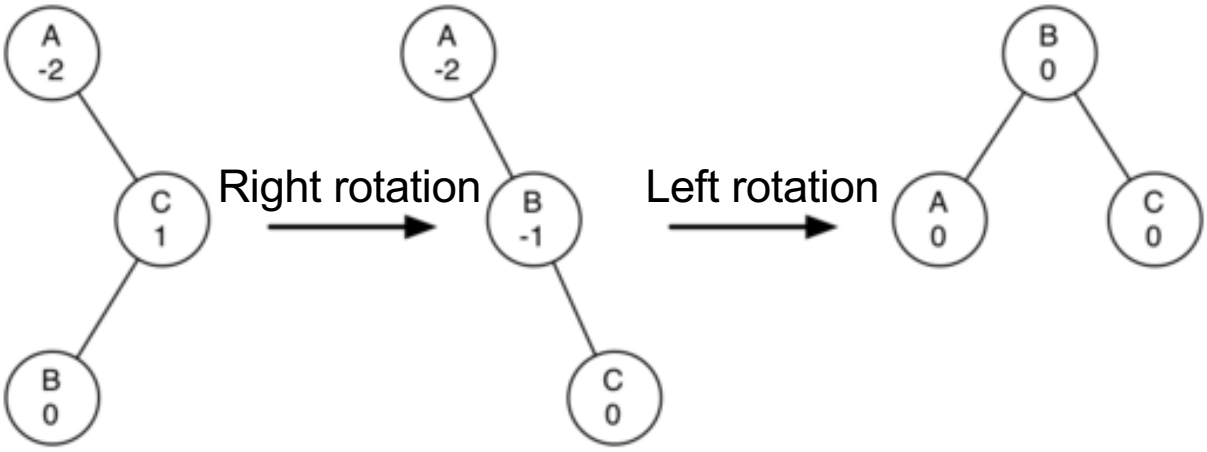
AVL Tree - Not Done Yet

- To correct problem:
  - If a subtree needs left rotation, first check balance factor of right child. If right child is left heavy then do a right rotation on right child, followed by original left rotation.
  - If a subtree needs right rotation, first check balance factor of left child. If left child is right heavy then do a left rotation on left child, followed by the original right rotation.

30

UMassAmherst

AVL Tree - Not Done Yet



31

UMassAmherst

AVL Tree – Rebalance

```
def rebalance(self,node):
  if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
           self.rotateRight(node.rightChild)
           self.rotateLeft(node)
      else:
           self.rotateLeft(node)
  elif node.balanceFactor > 0:
        if node.leftChild.balanceFactor < 0:
           self.rotateLeft(node.leftChild)
           self.rotateRight(node)
      else:
           self.rotateRight(node)
```


UMassAmherst

AVL Tree – Analysis

- Keeping the tree in balance all times => get runs in *O(log n)* time.
- Insertion (put):
  - New node inserted as leaf => *log n*
  - Balance => *0(1)*

UMassAmherst

Search – Summary

Operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	*O(n)*	*O(1)*	*O(n)*	*O(log n)*
get	*O(log n)*	*O(1)*	*O(n)*	*O(log n)*
in	*O(log n)*	*O(1)*	*O(n)*	*O(log n)*
del	*O(n)*	*O(1)*	*O(n)*	*O(log n)*

UMassAmherst

Next Steps

- Next lecture on Thursday
- HW3 due on Thursday
- Project 1 due on 10/14

UMassAmherst
The Commonwealth's Flagship Campus