CS112: Data Structures

Lecture 7

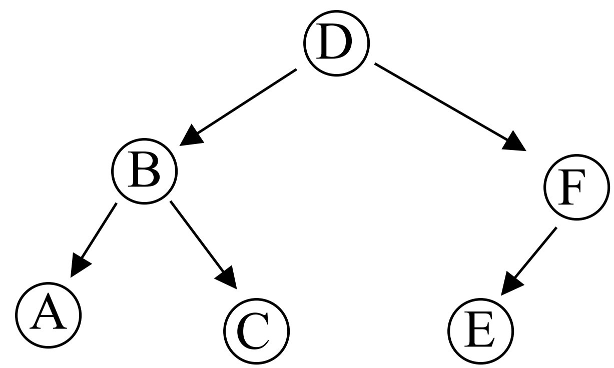
Hashing

Exam in 1 week

- **Exam 1 will be held:**
  - **Wednesday, June 29, 6 - 7:20 pm**
  - **In our normal lecture room**

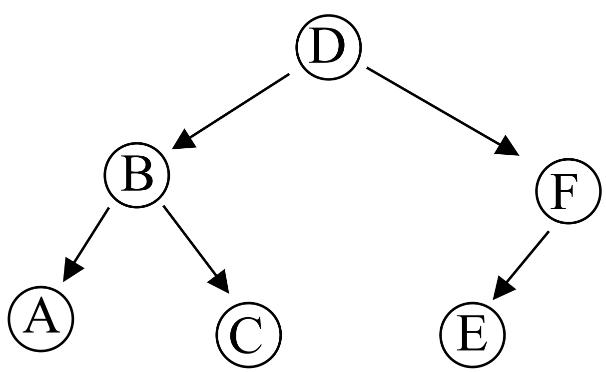
Review: Trees

- Nodes (vertices) and arcs (edges)
- Relationships:
  - Parent and Child
    - D and B, D and E, B and A, etx.
  - Root and Subtree
    - B and {B, C, A},
      D and {A, B, C, D, E, F},
      etc.



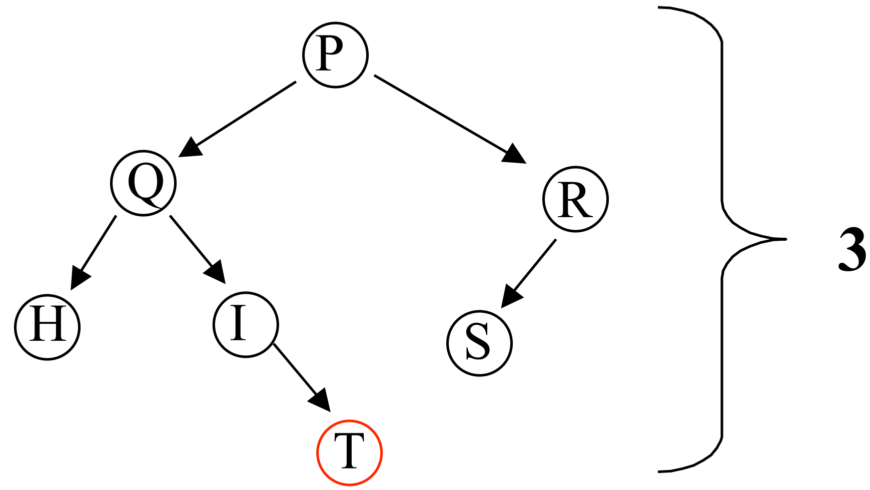
Trees

- Root (eg D) has no parents
- Leaf nodes have (A, C, and E) have no children
- All nodes except the root have a single parent
- There is exactly one path from root to any node

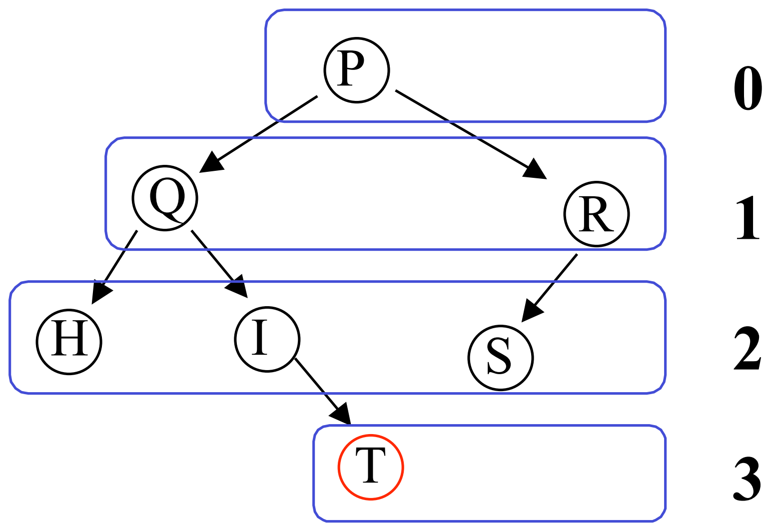


Trees

- **Height of tree**

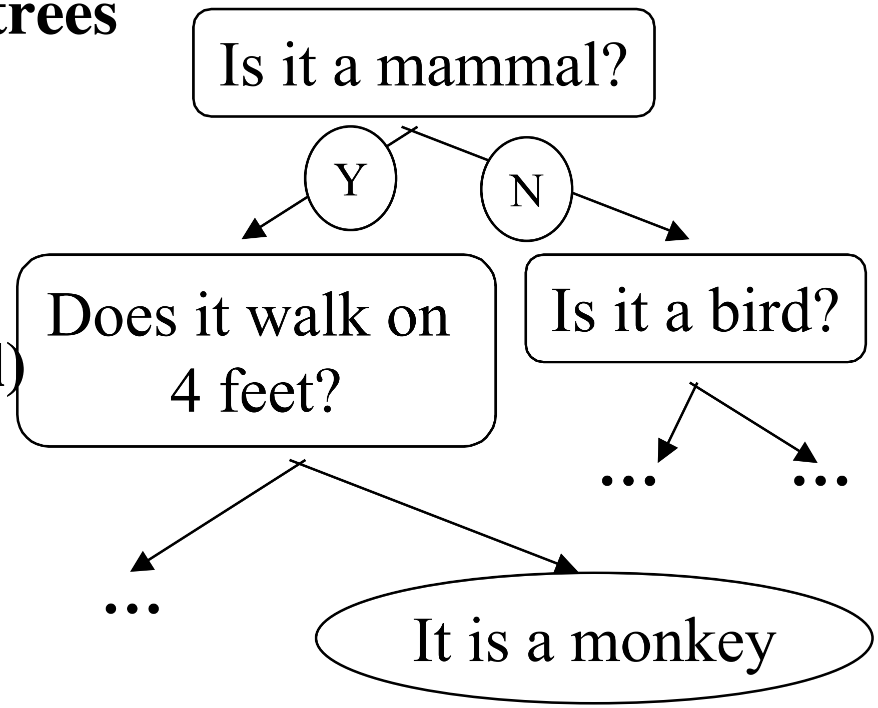


- **Depth of a node**

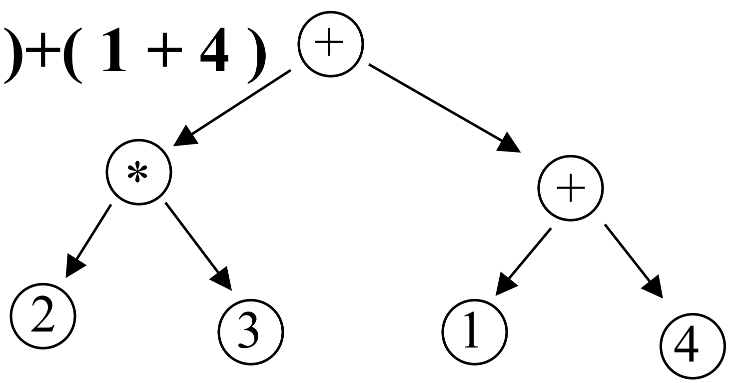


Binary tree

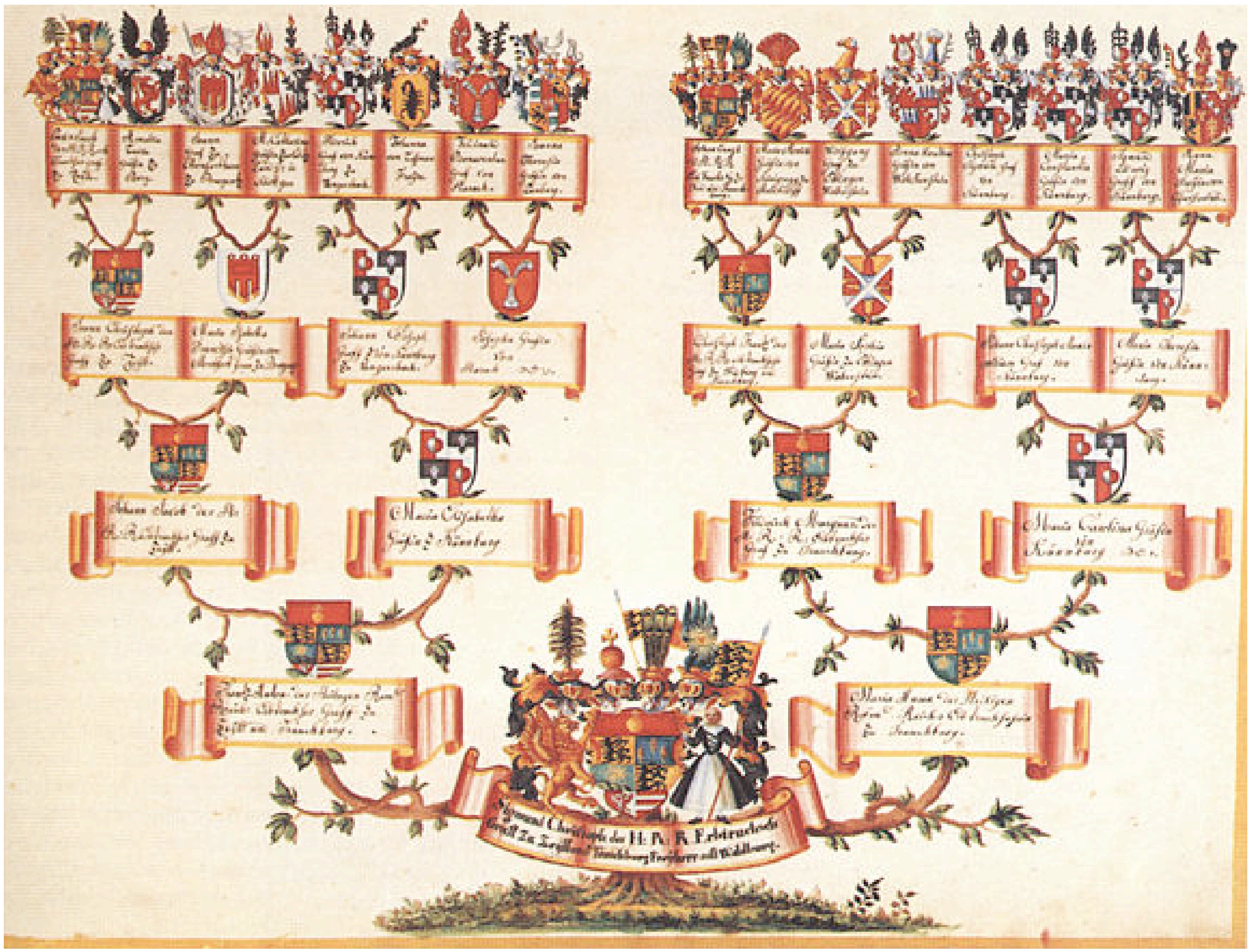
- each node has at most 2 subtrees
  - left and right subtree
- Examples of binary trees
  - 20 questions game (after animal/vegetable/mineral)
  - Arithmetic expressions



(2*3)+(1 + 4)



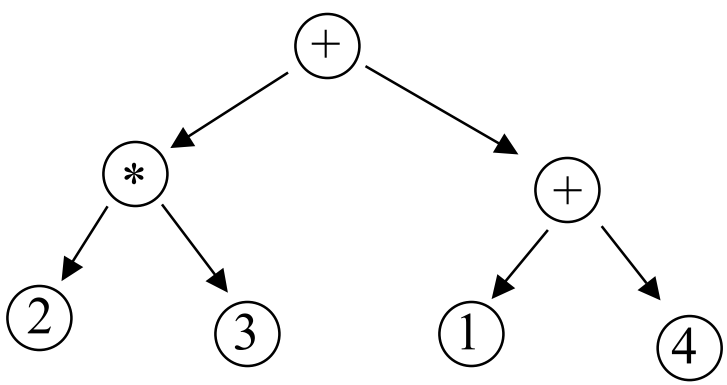
Family Tree

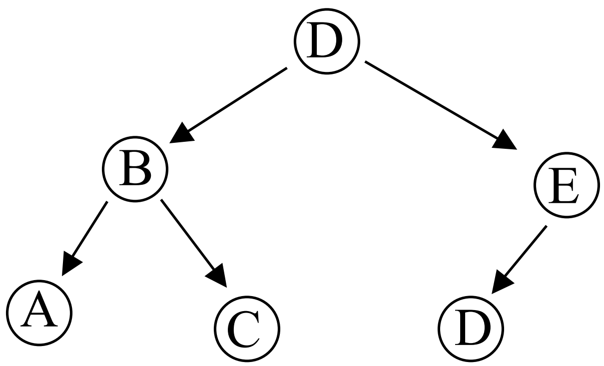


Source: Wikimedia Commons

Binary tree

- Strict binary tree
  - only 0 or 2 subtrees
  - why not “only 2 subtrees”?
- Complete binary tree
  - every level but last is full,
  - last filled left-to-right





Recursive Data Structures

- **Recursive definition of a binary tree**
  - **empty (i.e. null)**
  - **not empty**
    - **the root**
    - **a left subtree, which is a binary tree**
    - **a right subtree, which is a binary tree**

Recursive functions

- **Common form of function on a tree is recursive**

f(tree):

if (tree = = null) return

else return (data, f(tree.lst), f(tree.rst))

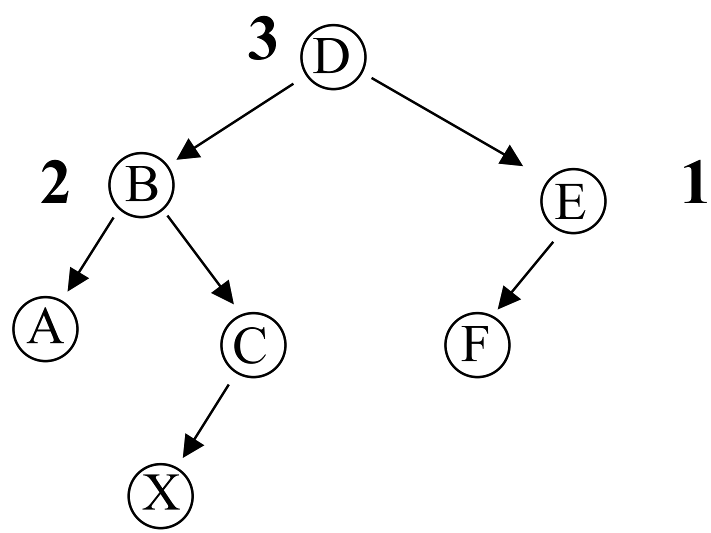
Where is a value and

is a function

Recursive functions
height

```
height(tree):
   if (tree = = null) return -1
     else  return  1 + max   (  height  (tree.lst),
                                height   (tree.rst))
```

Recursive functions
height



Recursive functions nodeCount

```
nodeCount(tree):
  if (tree = = null) return 0
    else  return 1 + sum  (nodeCount(tree.lst),
                           nodeCount(tree.rst))
```

Recursive functions
Sum

```
sum(tree):
    if (tree = = null) return
    else return      (tree.data,
                       sum( tree.lst ),
                       sum( tree.rst ))
```

Recursive functions
has0

```
has0(tree):
    if (tree = = null) return
    else return     (tree.data,
                       has0( tree.lst ),
                       has0( tree.rst ))
```

Recursive functions
has0

```
has0(tree):
    if (tree = = null) return false
    else return  or  (tree.data = = 0,
                      has0( tree.lst ),
                      has0( tree.rst ))
```

Static vs NonStatic

- **Problem in Java:**
  - **Null is not an object, so can't send it a message, so can't do**

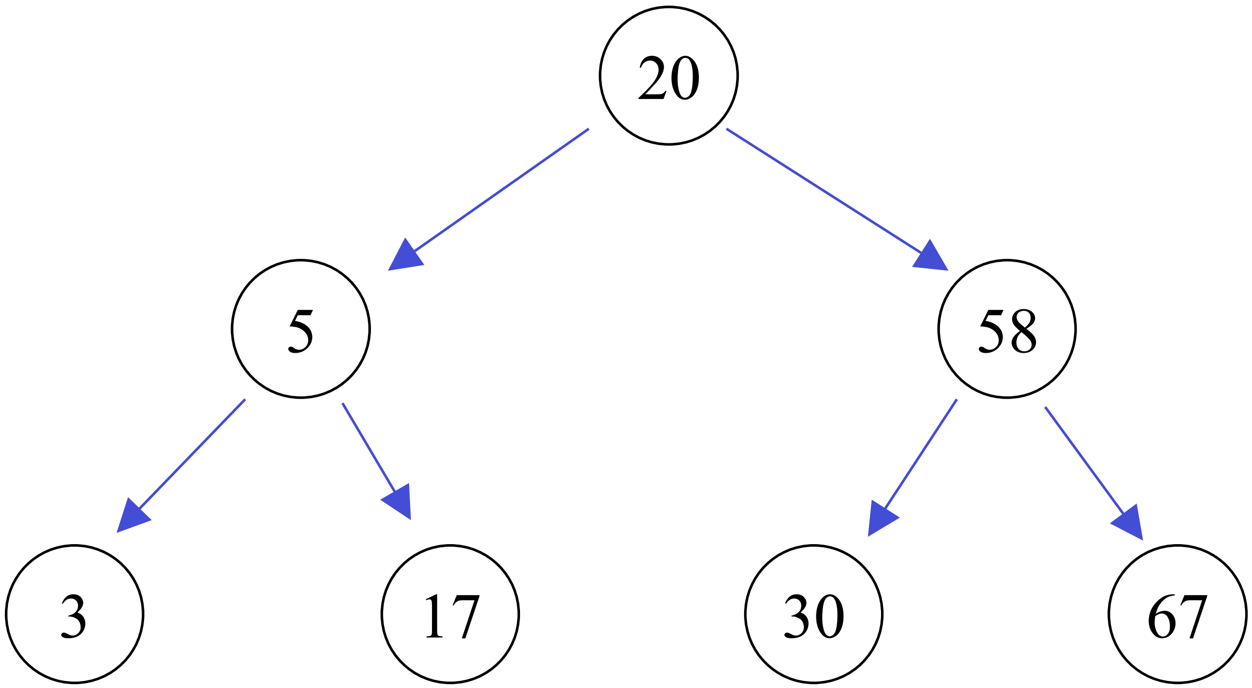
```
class TreeNode{
        int maxData( ){
                if (this = = null){ …
```

Back to: Add / Delete / Search

- **Basic task:**
  - **Set of data items**
    - **E.g. "Al", "Bob", "Cindy"**
  - **Operations:**
    - **Add an item**
    - **Delete an item**
    - **Search for an item**
- **Goal: minimize**

  worst case O(add + delete + search)

Binary Search Tree



Binary Search Tree

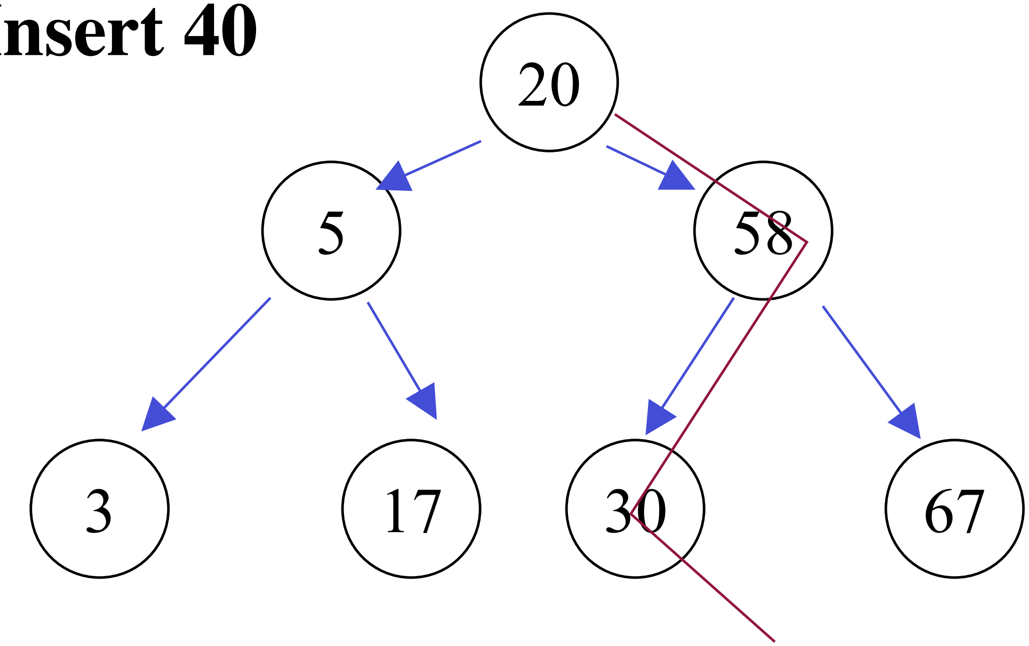
- data at a node is > any data in left subtree
- data at a node is < any data in right subtree
- Therefore, to print a BST in data order:
  - Print left subtree in data order
  - Print data
  - Print right subtree in data order

Search

- Searching a BST is easy
  - if node = null, search fails
  - if node.data equals target, found
  - if target < node.data, search on left subtree
  - else search on right subtree

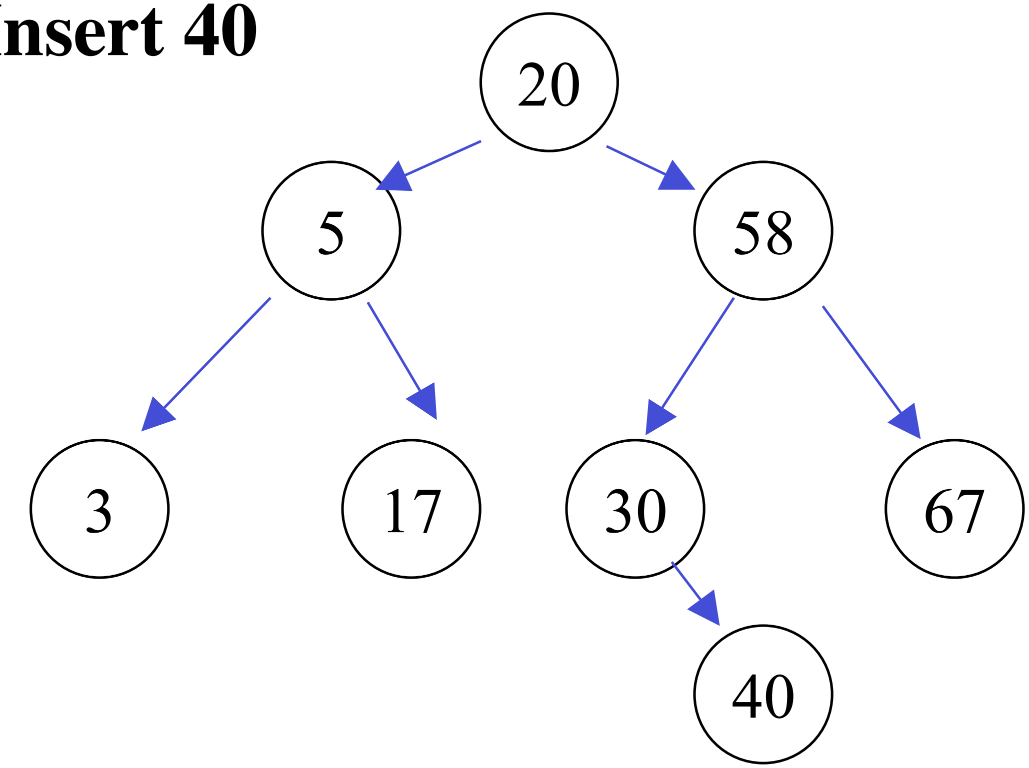
Insert

- **Search, fail, insert where failed**
  - **Insert 40**



Insert

- **Search, fail, insert where failed**
  - **Insert 40**

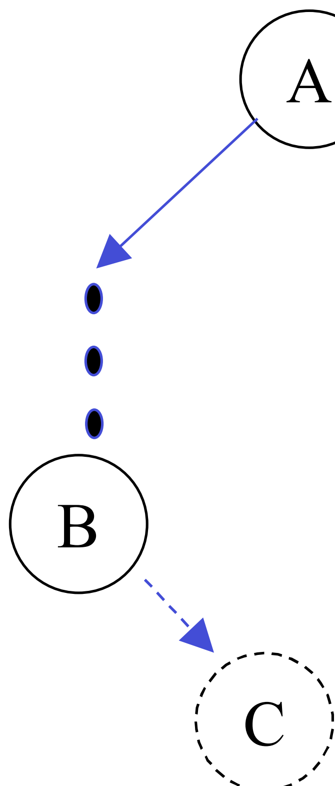


Delete

- Three cases
  - node to delete had no children => delete it
  - node to delete has 1 child => replace node with child
  - node to delete has 2 children

Deleting node with 2 children

- **Observation: for node with left child, inorder predecessor has no right child**



If C exists, C>B and C < A

So B cannot be inorder predecessor of A

Deleting node with 2 children

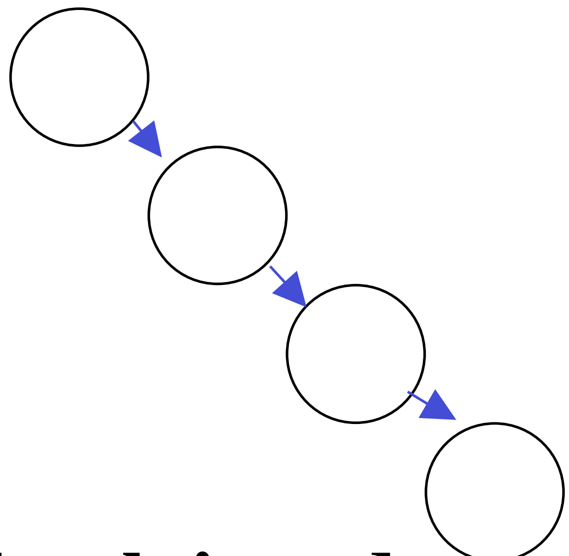
- Replace data at node with data of inorder predecessor
- Delete inorder predecessor (which must have either 0 or 1 child)

Cost of using BST

- Search: O(depth)
  - what is depth of tree?
  - with n nodes, best depth is log n
  - but worst depth is n

Binary Search Trees

- Problem: insertion & deletion can give tree of any shape - even



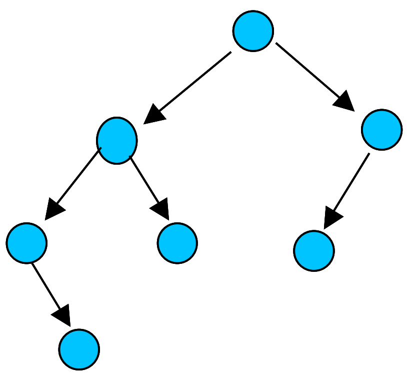
- Worst case depth is order n, not logn

•Goal: O(log n) complexity

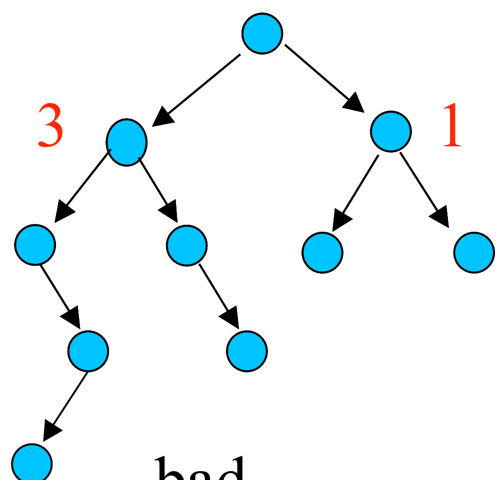
- **Goal: to be able to maintain a list with all operations at worst O(log(# nodes))**
  - **Insert, delete, search**
- **Binary search tree is O(depth) but depth is, worst case, #nodes**
- **AVL tree is like Binary search tree but depth is roughly log(#nodes)**

AVL Trees

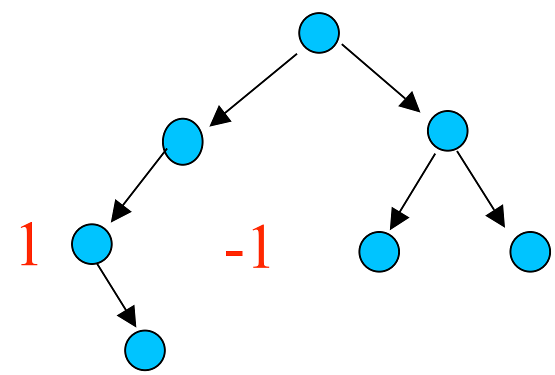
- **Binary Search Tree**
  - **Inorder traversal = data order**
- **Almost balanced**
  - **At every node, subtree heights same +/- 1**



good



bad

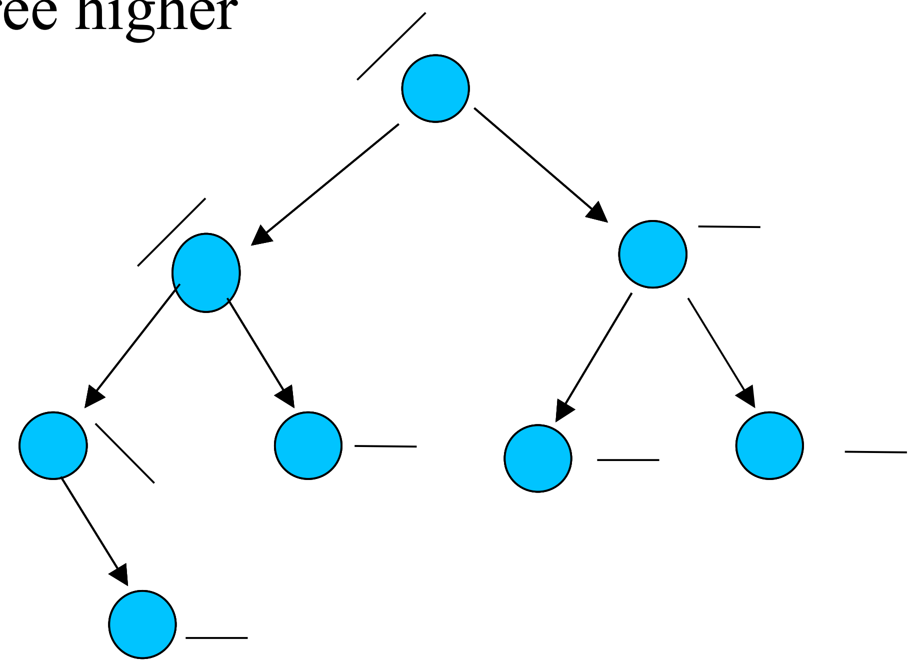


bad

Labeling an AVL Tree

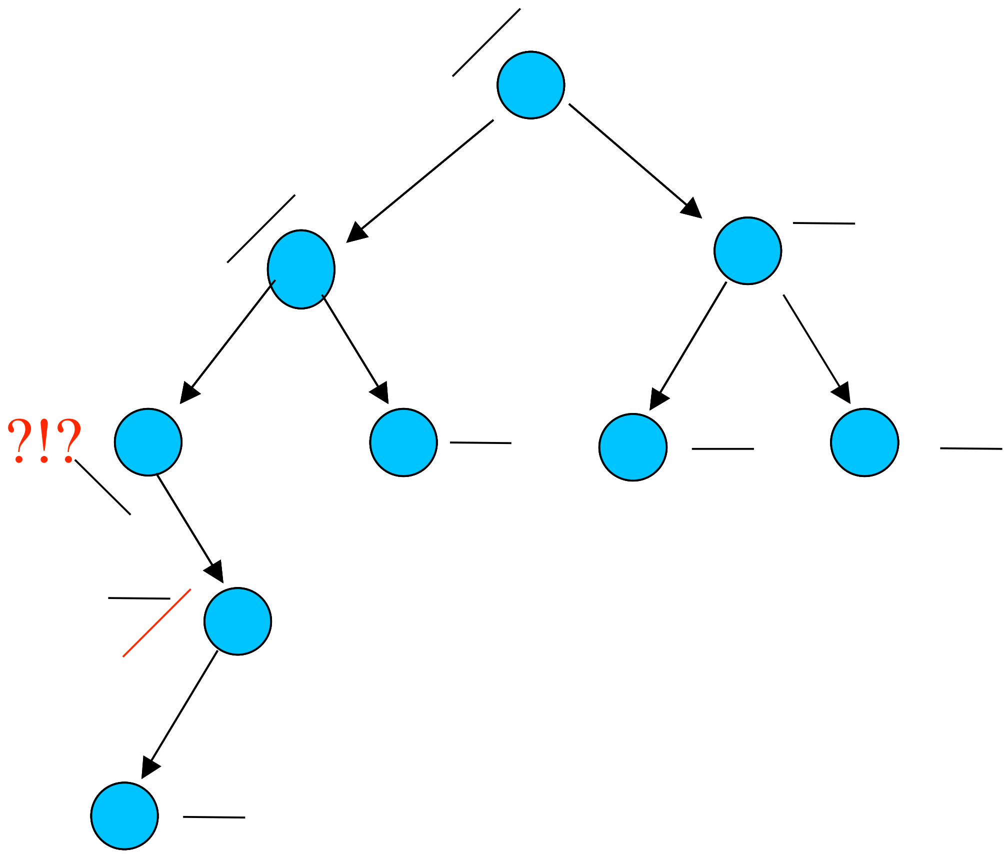
- **Label each node as**

  \- left & right subtrees equally high
  \ right subtree higher
  / left subtree higher



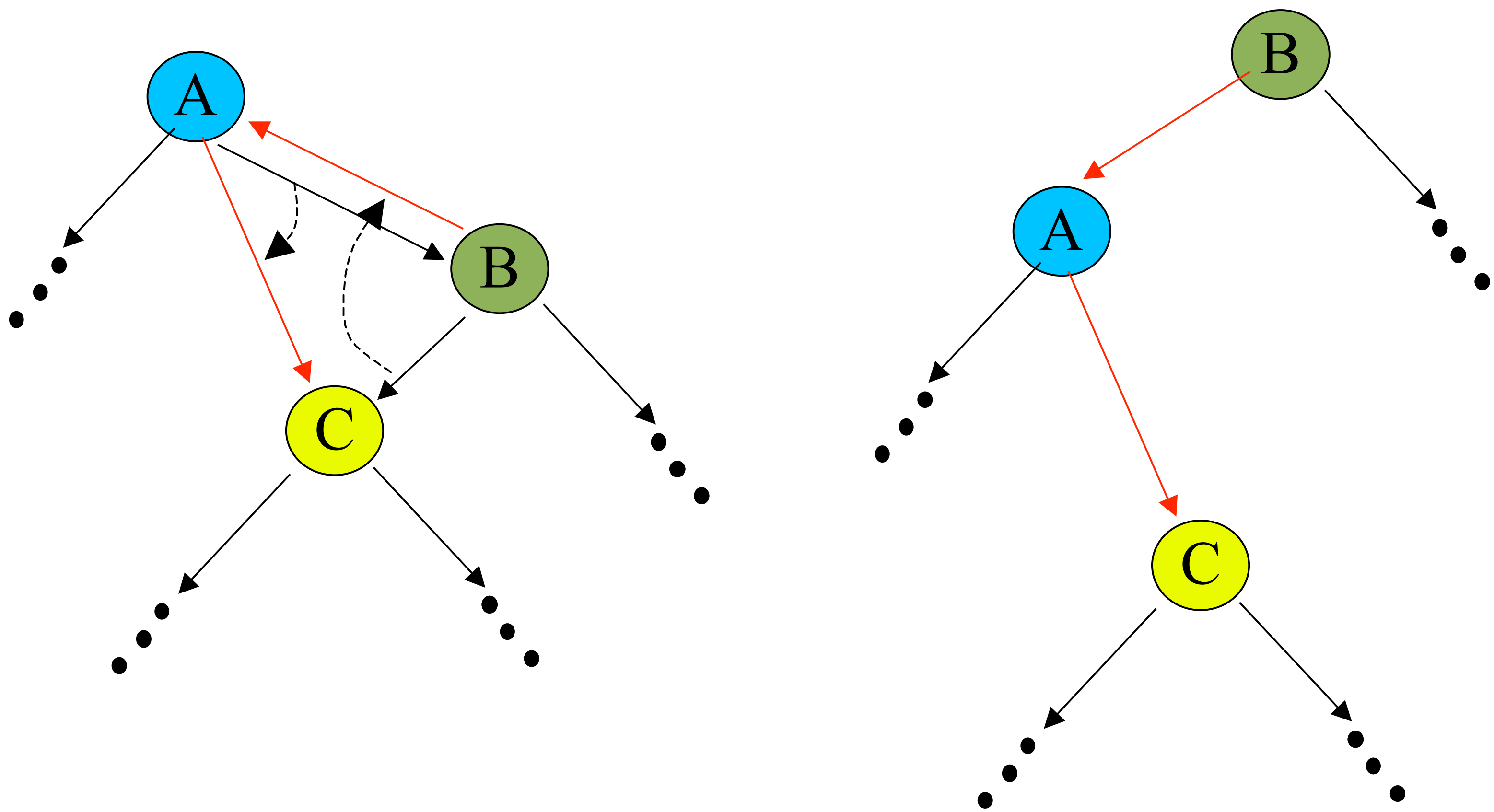
Rebalancing

- **Problem: insert/delete -> not balanced**



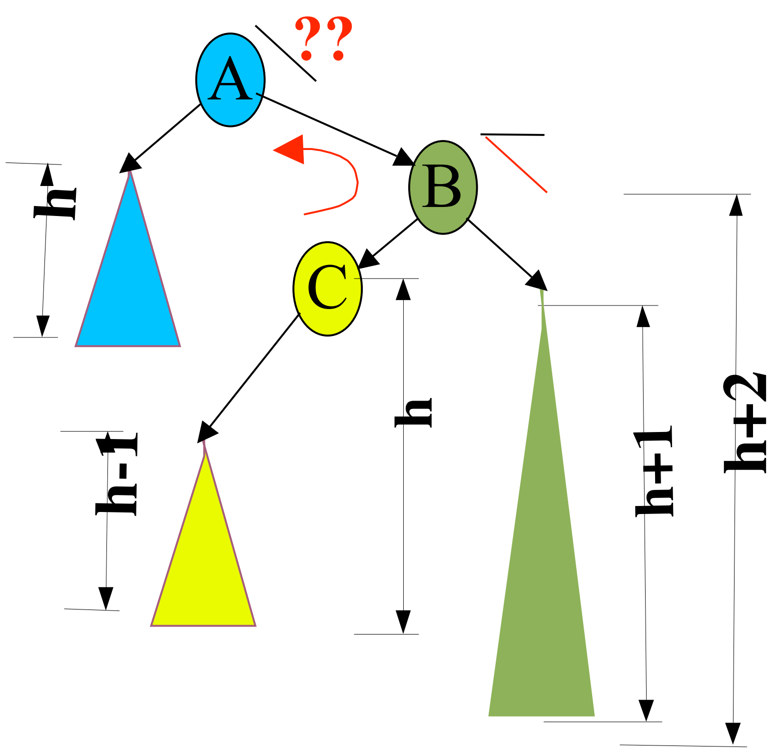
Rebalancing

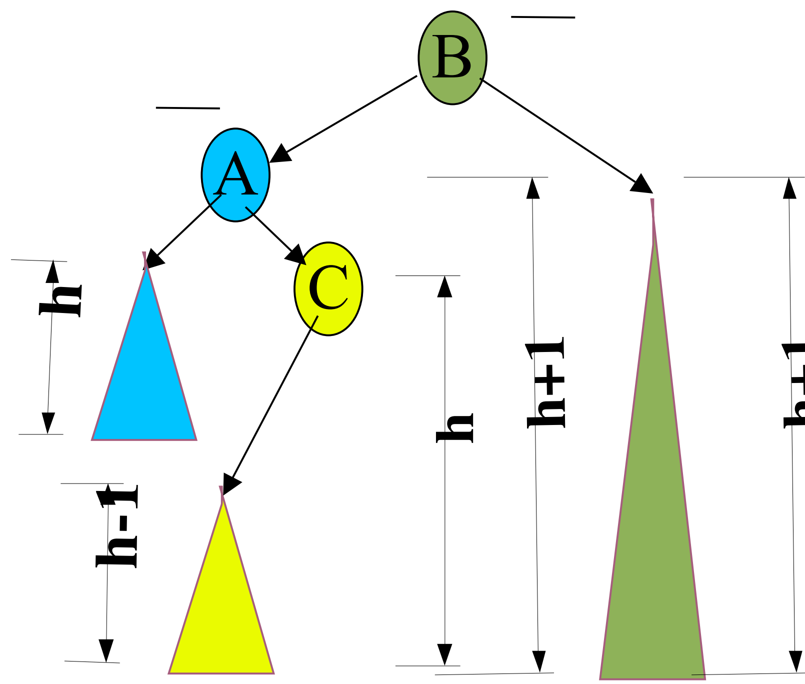
- **Solution: Rotation**



Rebalancing

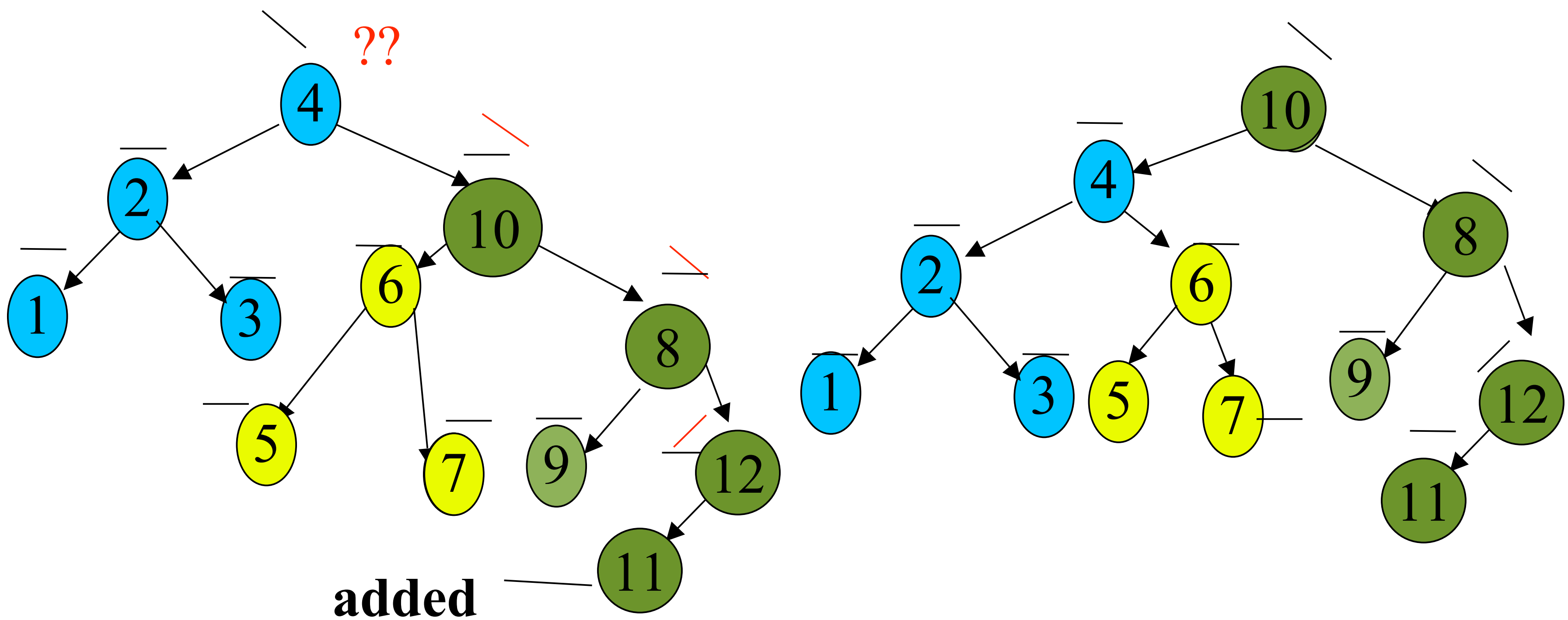
- **Solution: Rotation**
  - **Highside child of A has same label as A**





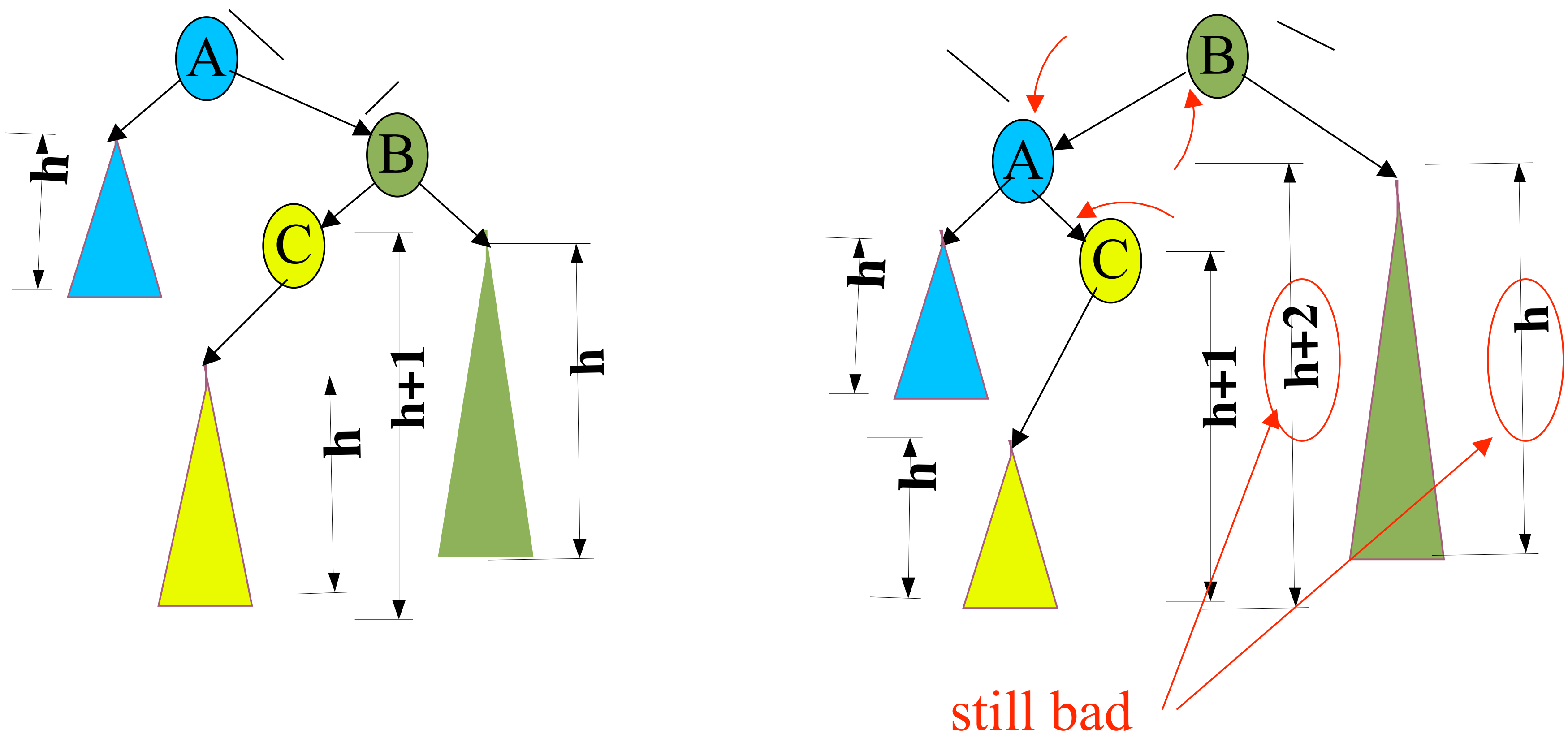
Rebalancing

- **Solution: Rotation**
  - **Highside child of A has same label as A**



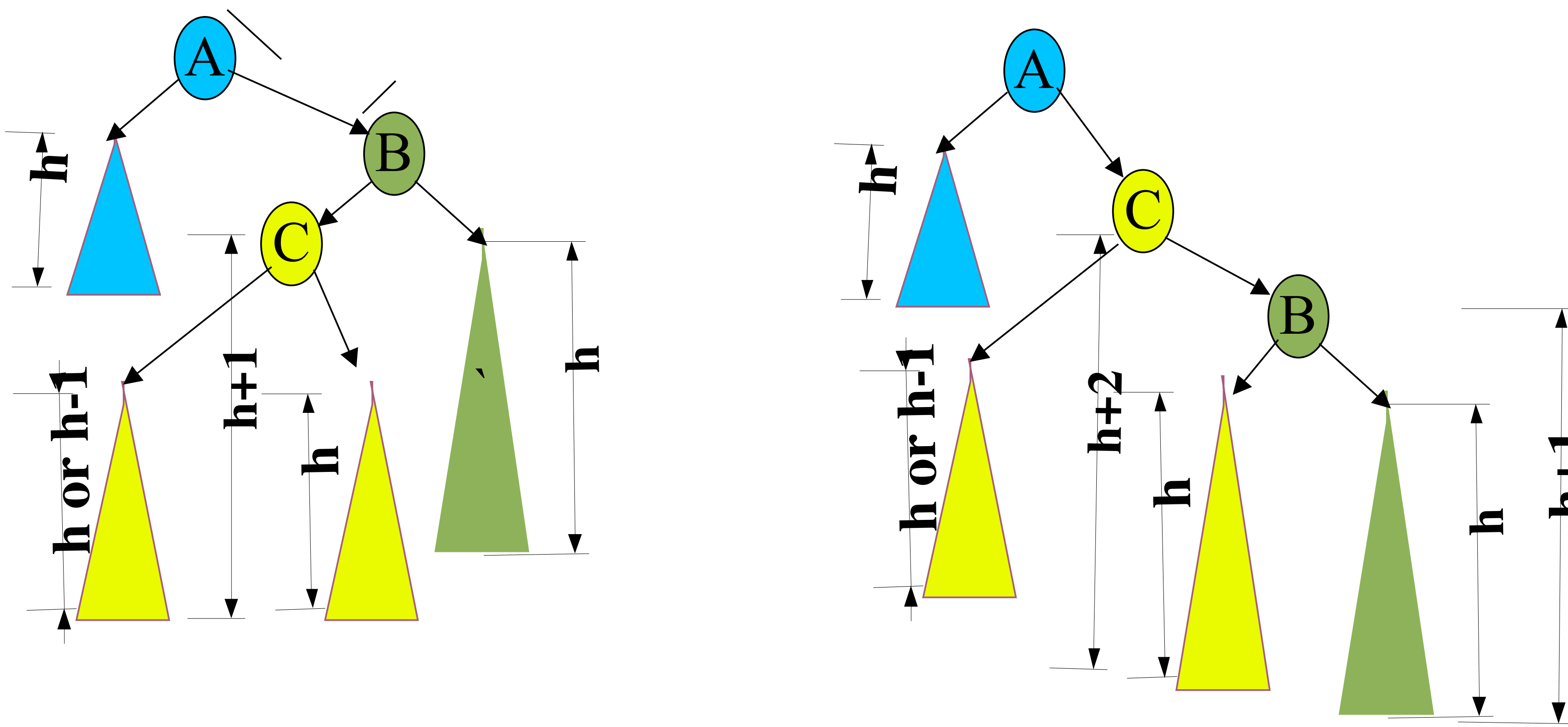
Rebalancing

- **Solution: Rotation**
  - **Highshide child of A has opposite label from A**



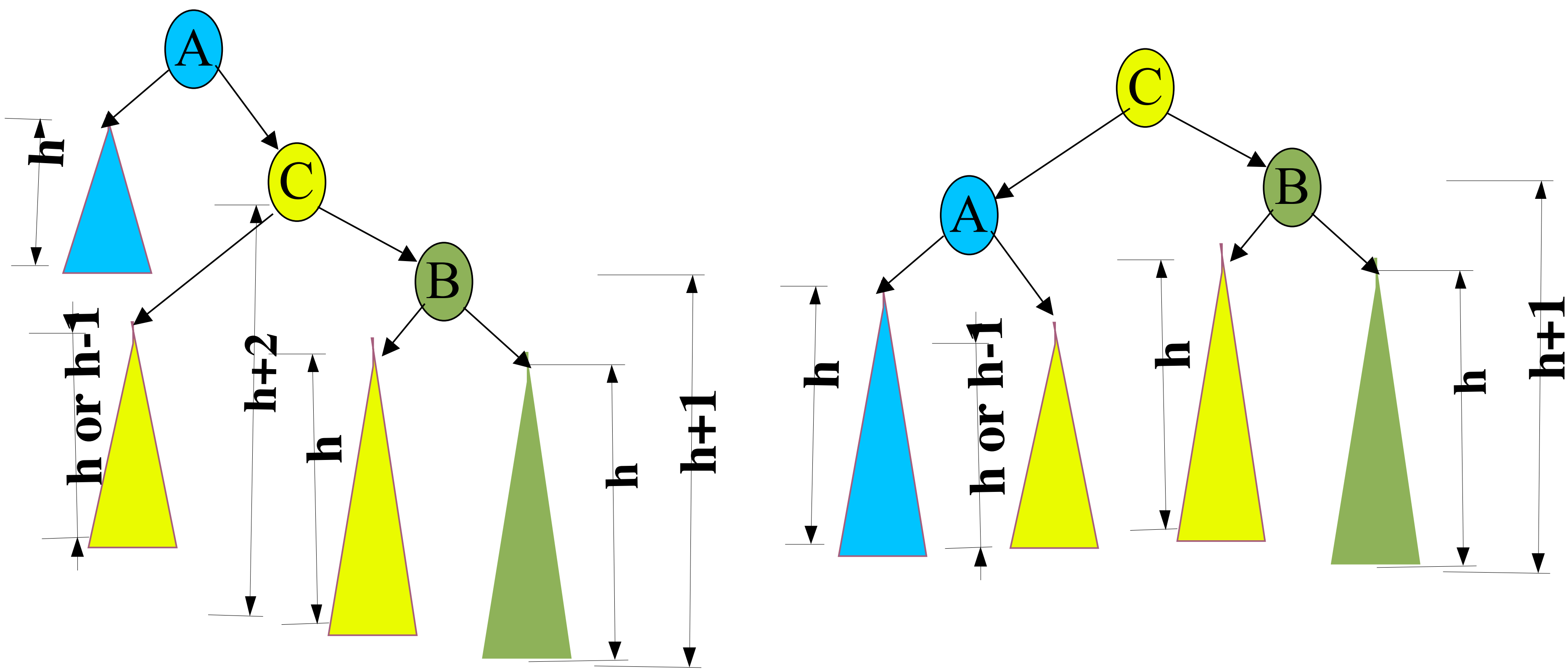
Rebalancing

- **Solution: Rotate BC First**



Rebalancing

- **Solution: Then Rotate AC**



New: Hashing

- Suppose we want to store a set of numbers
  - add number to set, delete from set, test if in set should all be O(1)
- If range of numbers is small, e.g. 0 .. 9, we can use a boolean array

0	1	2	3	4	5	6	7	8	9
t	f	f	f	t	t	f	t	f	f

- What if range of numbers is large, e.g. 0…500,0000?
  - but only a small number of numbers, e.g. 10

Hashing

- **If we use array of 500,000 elements, they will nearly all be false.**
  - **Divide the 500,000 into blocks of say 1000**
  - **500 blocks so unlikely that two of our 10 numbers will fall in one block**
  - **So for each block an object:**
    - **boolean: did any number fall in this block?**
    - **int: if so, which one**

Hashing

- Array of 500 objects
  - Insert n: put in object at index n/1000
  - Lookup n: look in object at index n/1000
    - is any number in this object?
    - is it the right number?
  - All O(1)

Hash Function

- What if numbers not random, eg likely to be near each other?
  - convert n to index in some other way, e.g. index = n mod 500
  - In general, function that makes each index equally likely: “makes hash out of any pattern in the numbers” -
- Hash function: converts data to hash code
- Mapping function: converts hash code to array index. (Why separate this?)

Collisions

- Even with 500 indices for 10 numbers, it is possible that more than one number will hash to same index
- As we reduce number of indices probability of collision grows
- => must be some way to handle collisions

Linear Probing

- **On insert n, if already data at hash(n), try hash(n)+1, hash(n)+2, …**
- **On lookup n, look at hash(n), hash(n)+1, hash(n)+2, … until**
  - **find n**
  - **find empty object**

Problem: clumping

- **Say 10 indices.**
  Let P(i) = P(next number goes in index i)
- **When objects empty, P(i) = .1 for all i**
- **Suppose number in index 3.**
  - **P(3) = 0, P(4) = .2**
- **Suppose numbers in index 3 & 4**
  - **P(3) = P(4) = 0, P(5) = .3**

Quadratic Probing

- **If hash(n) full try hash(n)+1, hash(n)+4, hash(n)+9, … hash(n)+j^2**
- **Does not have clumping effect**
- **Does have problem that it only tries at most half the indices**

Chaining

- **Instead of moving to other indices on collision, have a linked list of items at each index**

Complexity

- **Worst case: O(n)**
  - all items hash to same index
- **Average: depends on load factor $\alpha = n / size$**

alpha	linear	quadratic	chaining
.1	1.06	1.06	1.05
.5	1.5	1.4	1.3
.8	3	2	1.4
.9	5.5	2.6	1.45
. 99	50.5	4.6	1.5

Built-in Hashing in Java

- The class java.util.HashMap<K, V>
  – Mapping from (unique) key to a value
  – Note: generic with two class parameters:
    – K: class of keys
    – V: class of values
  – E.g. Driver's license ID (String) => Driver object (name, address, etc.): java.util.HashMap<String, Driver>