CS112: Data Structures

Lecture 06

Add / Delete / Search

Trees

Review: 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)

Unordered Array

Add Alice

0	Carol
1	Anne
2	Bob
3	
4	

numberOfNames

3

→

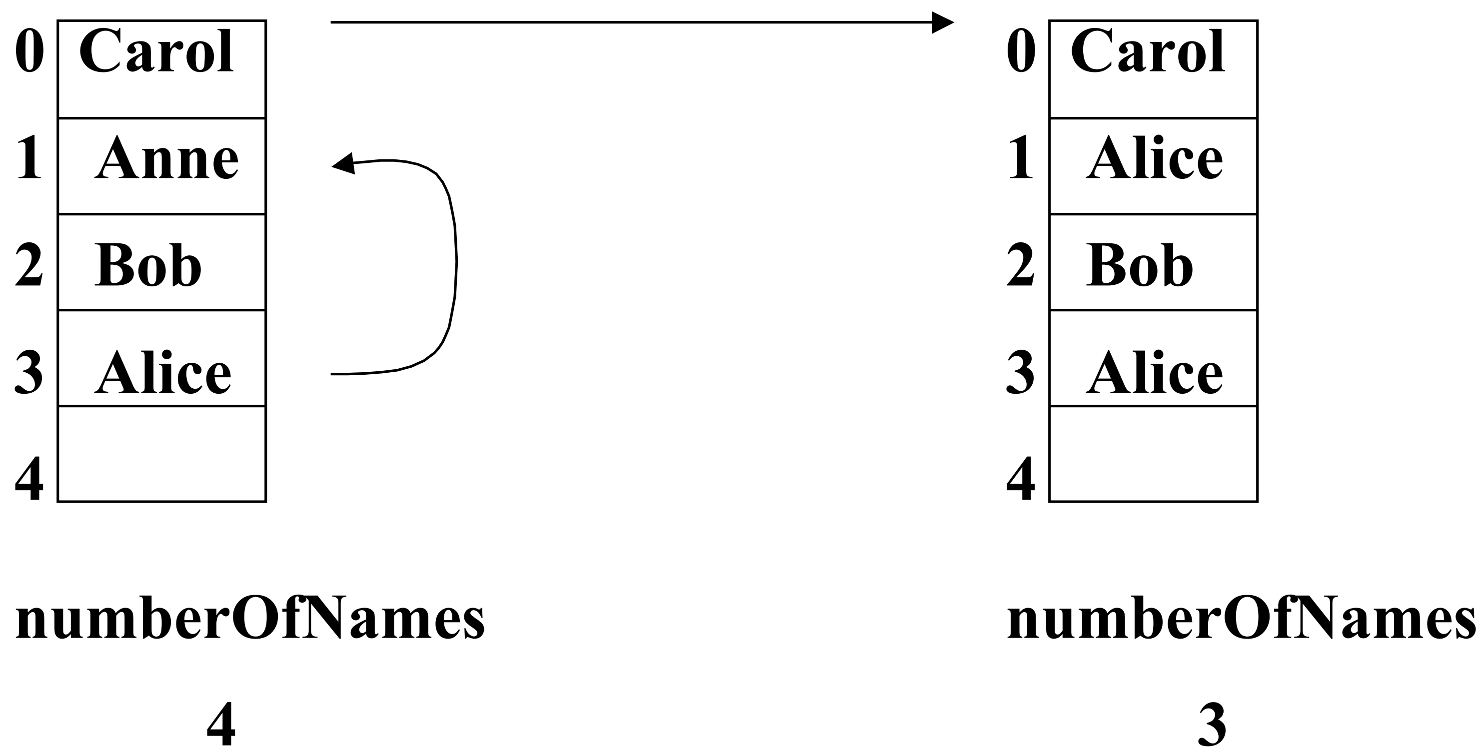
0	Carol
1	Anne
2	Bob
3	Alice
4	

numberOfNames

4

Unordered Array

Remove Anne

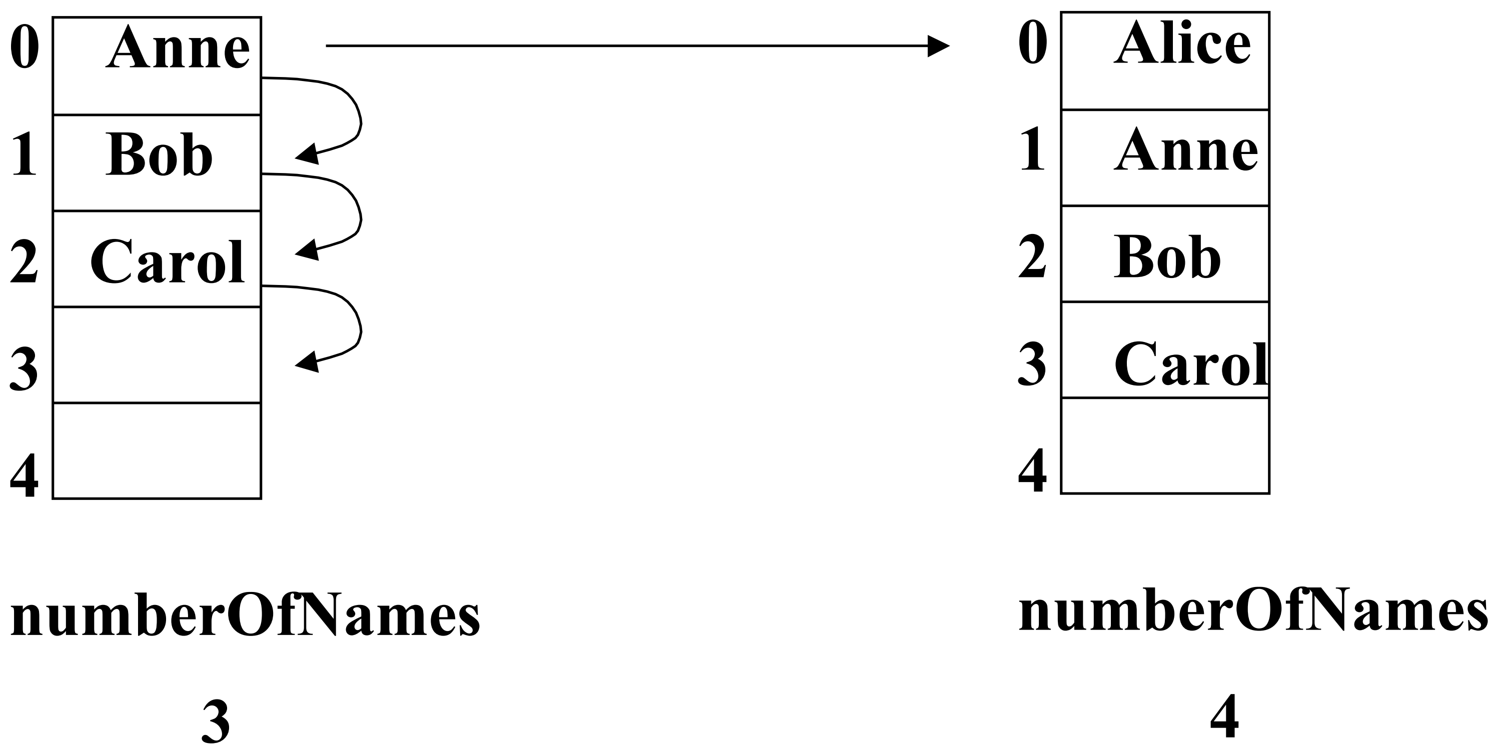


Unordered Array

- Insert O(1) if there’s space
- Delete O(1) (move last element)
- Search O(n) where n is size of set
- Overall O(n)

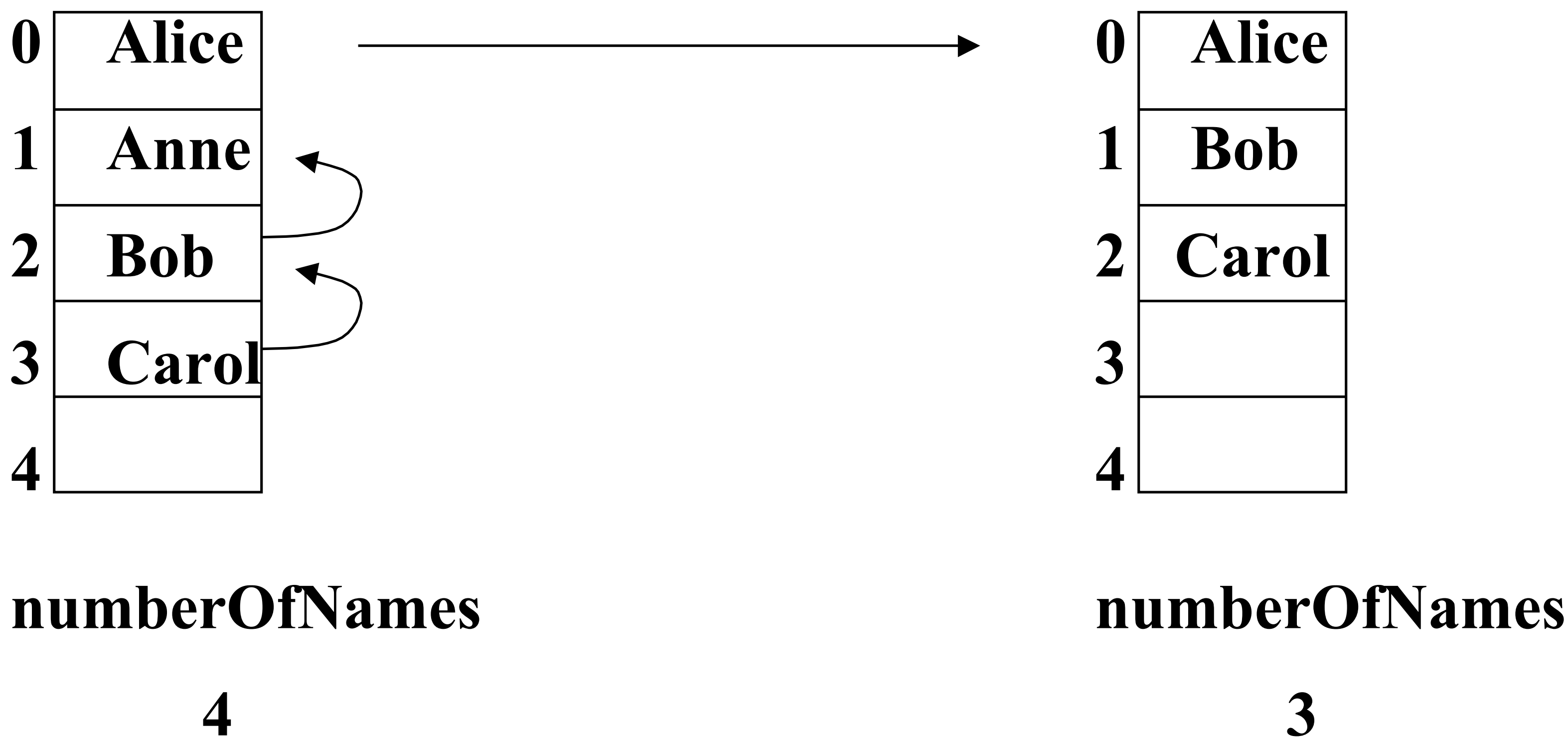
Ordered Array

Add Alice



Ordered Array

Remove Anne



Searching an ordered array
Binary search

- requires sorted values
- each comparison rules out half of the remaining elements
- O(log(n)) - we will prove this later
- Find A, find R

A	D	E	F	G	H	J	M	P	T
0	1	2	3	4	5	6	7	8	9

Searching an array Performance

- Search among 1 <u>B</u>illion entries
- Check 1 million entries per second
  - Sequential search
    - 1 billion operations needed
    - Requires 1000 seconds - about 20 minutes
  - Binary search
    - 30 operations needed
    - Requires 30 microseconds
    - 30 million times faster

Searching an array Performance

- **Search among 1 Million entries**
- **Check 1 million entries per second**
  - **Sequential search**
    - **1 million operations needed**
    - **Requires 1 second**
  - **Binary earch**
    - **20 operations needed**
    - **Requires 20 microseconds**
    - **50,000 times faster**

Searching + Insert Performance

- 1 billion entries, process 1 million/sec
- Insert O(n)
  - 1 billion operations needed
  - Requires 1000 seconds - about 20 minutes
- Binary search O(log n)
  - 30 operations needed
  - Requires 30 microseconds
- Together: 1000.00003 seconds

Searching + Insert Performance

- **1 million entries, process 1 million/sec**
- **Insert O(n)**
  - **1 million operations needed**
  - **Requires 1 second**
- **Binary search O(log n)**
  - **20 operations needed**
  - **Requires 20 microseconds**
- **Together: 1.00002 seconds**

Ordered Array

- **Insert O(n)**
- **Delete O(n)**
- **Search O(log n) Binary Search**
- **Overall O(n)**

Unordered Linked List

- **Insert O(1)**
- **Delete O(1)**
- **Search O(n)**
- **Overall O(n)**

Ordered Linked List

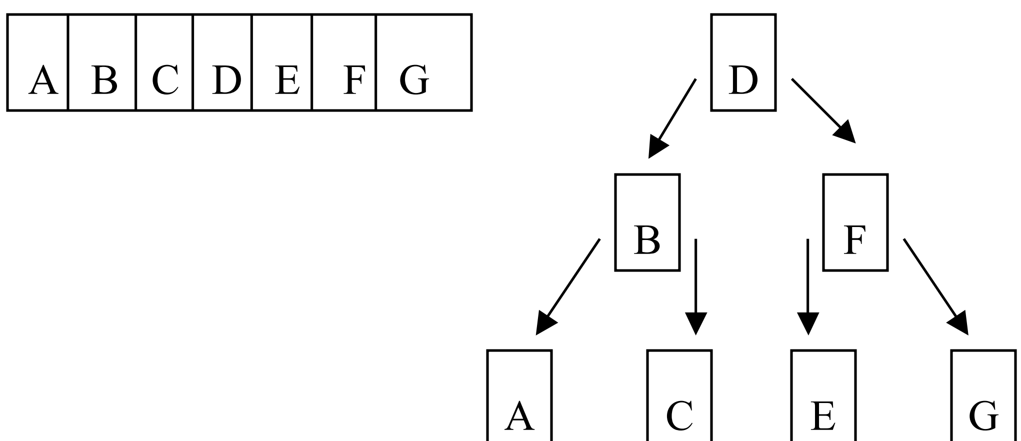
- Insert O(n)
- Delete O(1)
- Search O(n)
  - Jump to A[middle] is O(n) not O(1) so linear search faster than binary search
- Overall O(n)

Links Speed Up Add/Delete

- **Idea: Use linked list to make add / delete faster**
- **Problem: Search of linked list is O(n)**
  - **Why not binary search on linked list?**

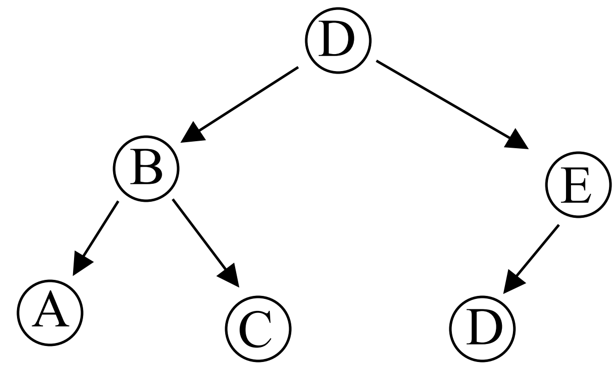
Links Speed Up Add/Delete

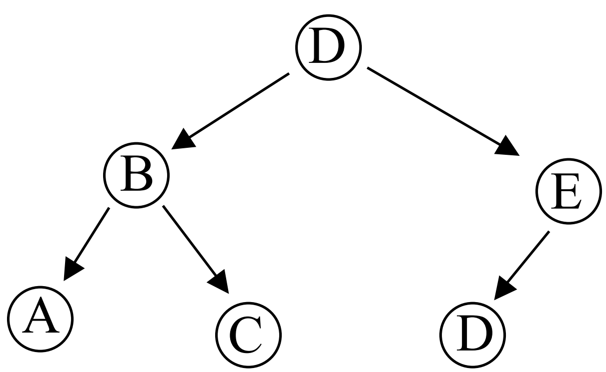
- **Idea: Use linked list to make add / delete faster**
- **Problem: Search of linked list is O(n)**
  - **Why not binary search on linked list?**
- **Idea: links to *two* places**

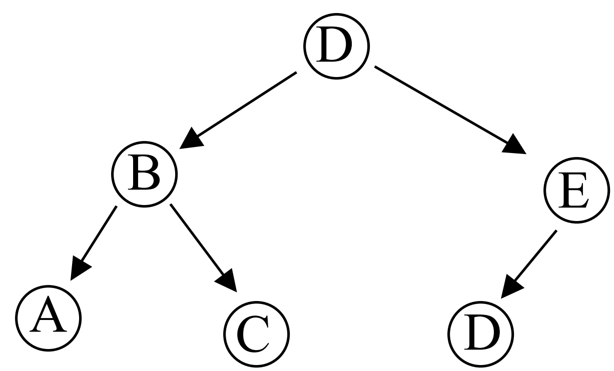


New Trees

- Nodes and arcs (edges)
- Relationships:
  - Parent and Child
  - Root and Subtree

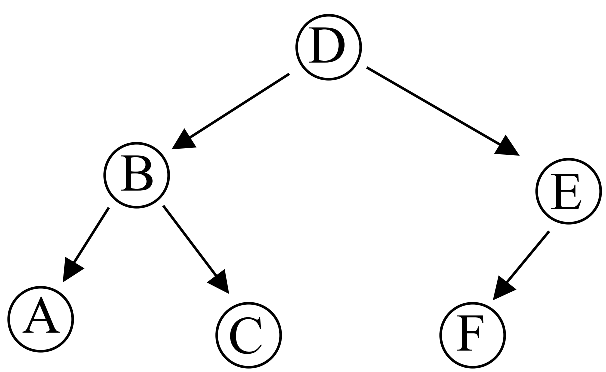






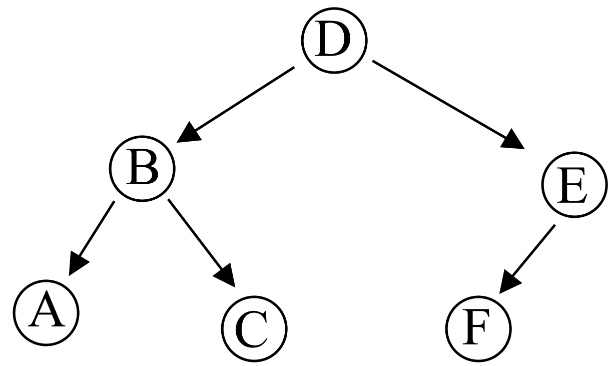
Trees

- Root has no parents
- Leaf node has no children
- All nodes except the root have a single parent
- There is exactly one path from root to any node

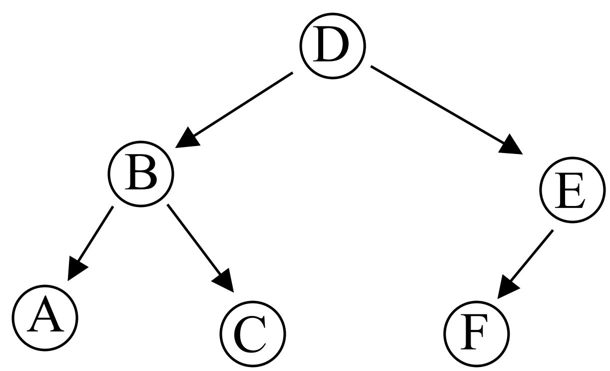


Trees

- **Height of tree**

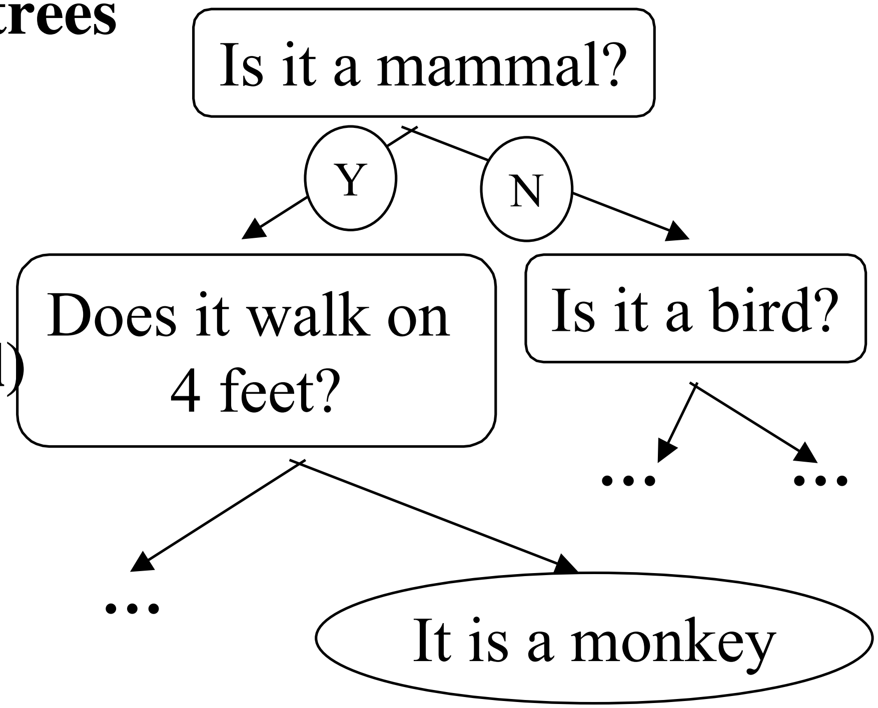


- **Depth of a node**

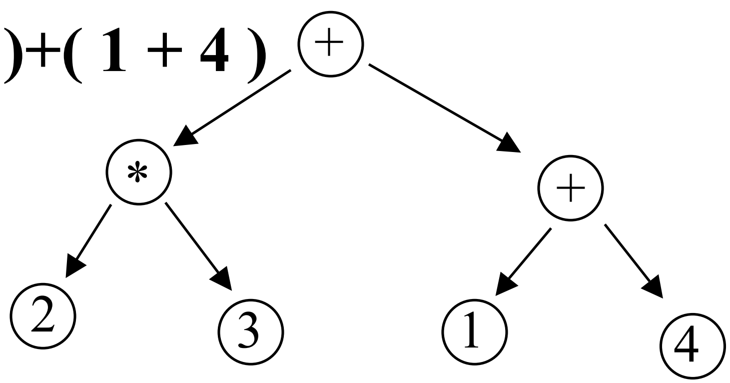


Binary tree

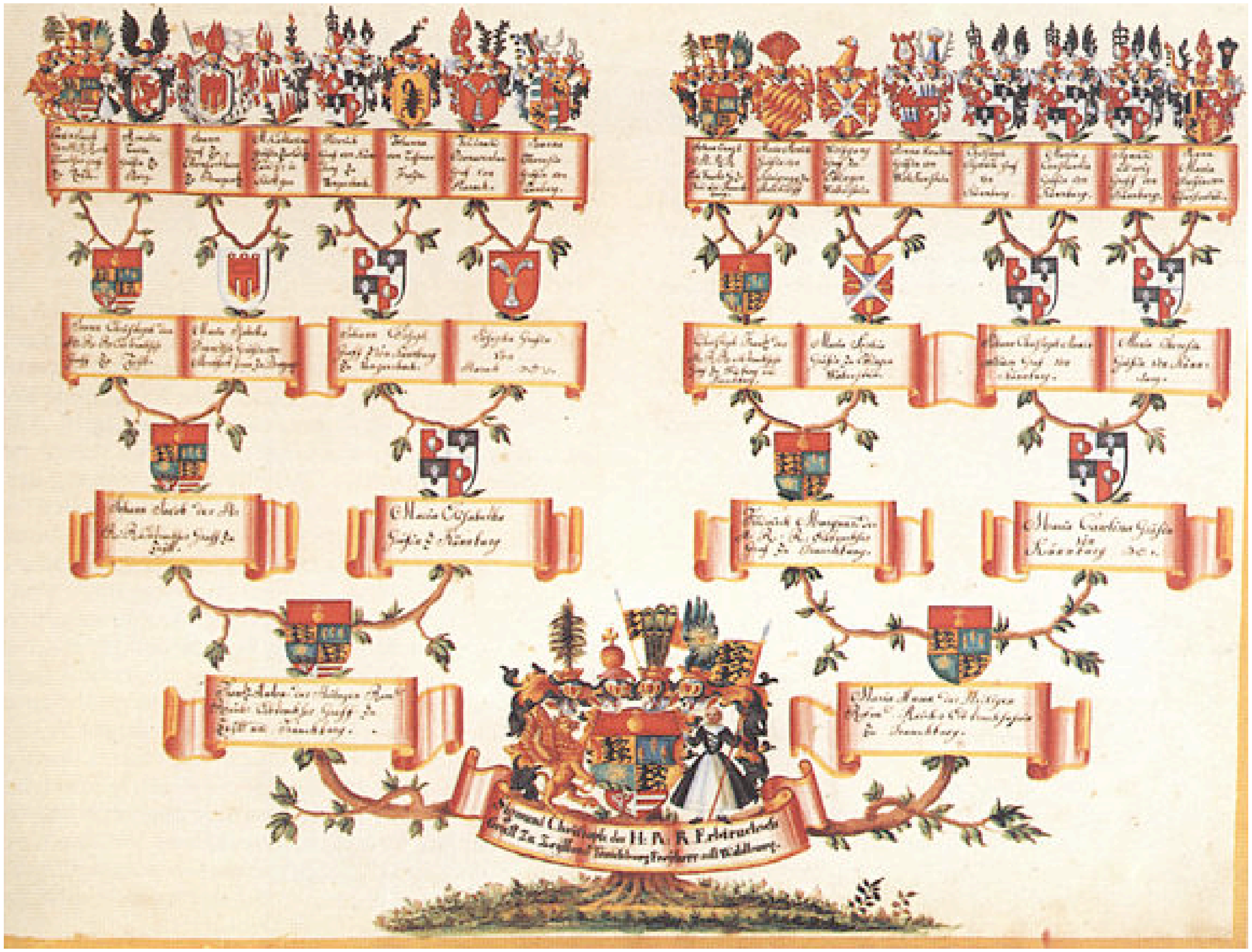
- **each node has at most 2 subtrees**
  - **left and right subtree**
- **Examples of binary trees**
  - **20 questions game (after animal/vegetable/mineral)**
  - **Arithmetic expressions**



(2*3)+(1 + 4)



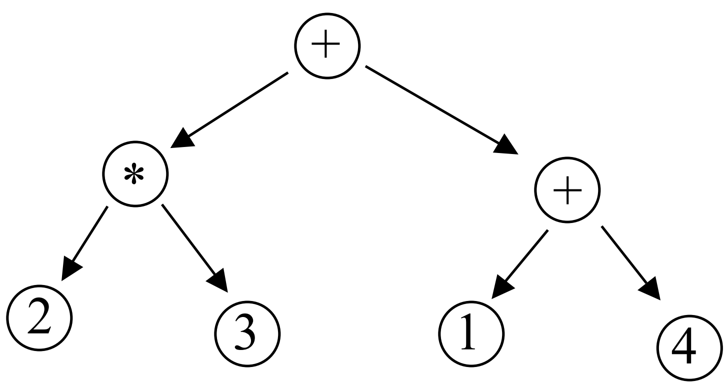
Family Tree

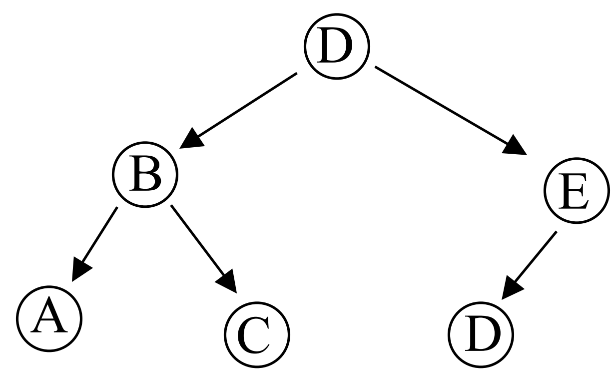


Source: Wikimedia Commons

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))

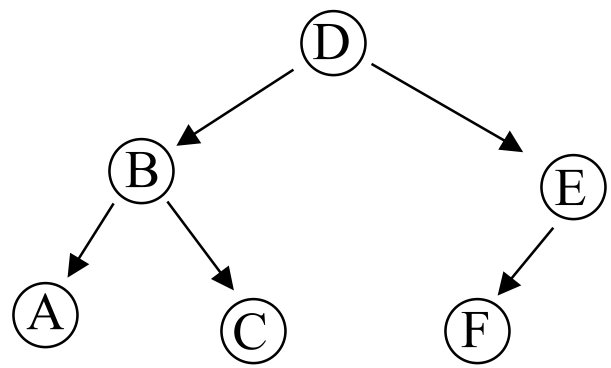
Where is a value and

is a function

Recursive functions height

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

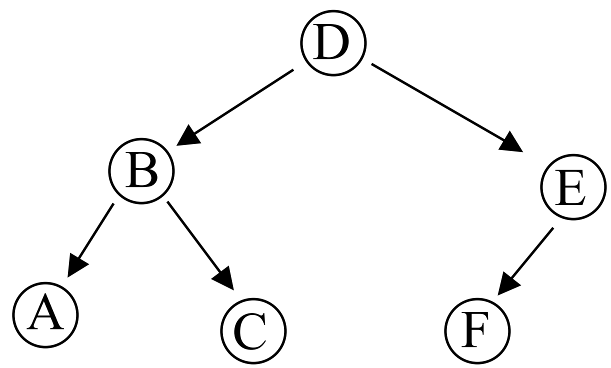
Recursive functions
height



Recursive functions nodeCount

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

Recursive functions nodeCount



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(TreeNode node){
                if (node = = null){ …
```

- **See TreeNode.java**

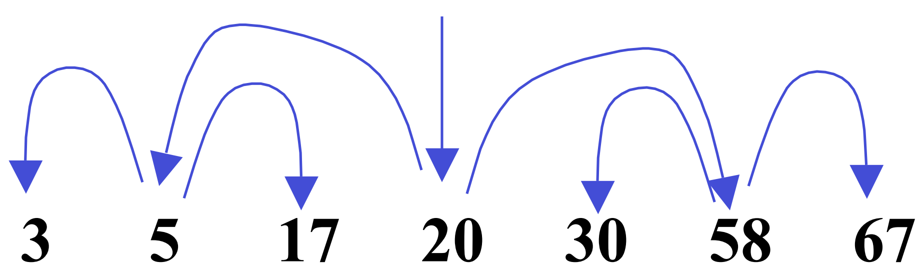
Back to: Add / Delete / Search

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

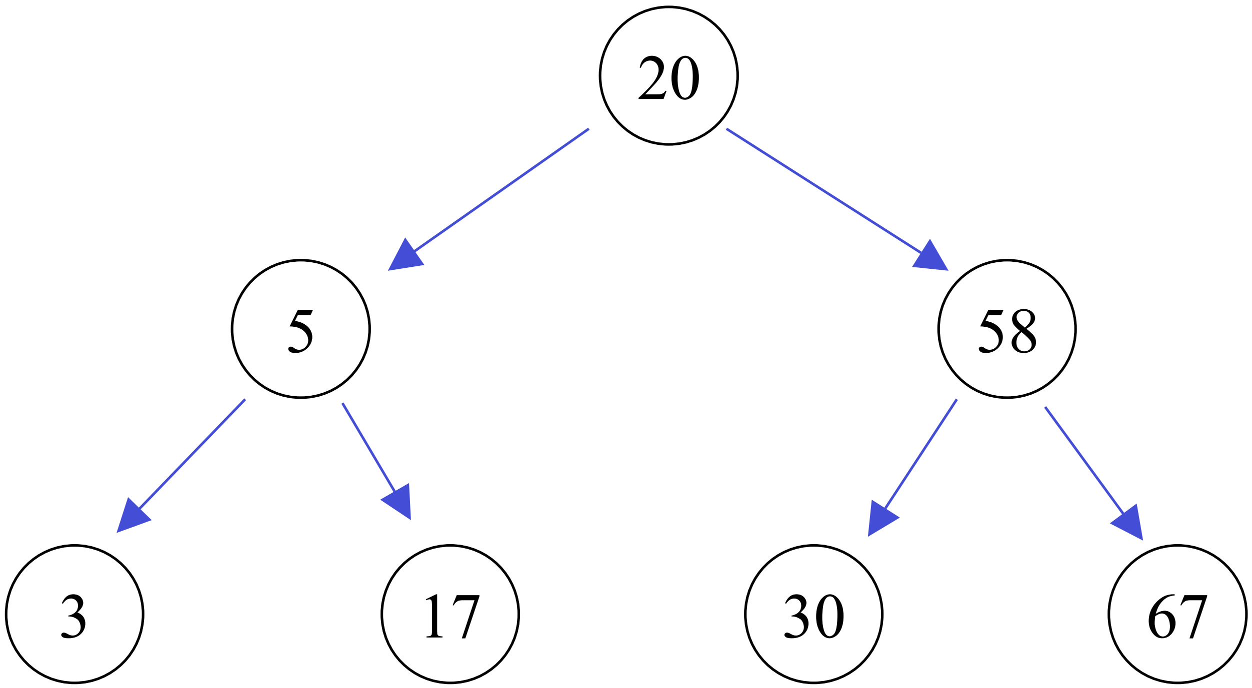
  worst case O(add + delete + search)

Binary Search Trees

- Why can't we do binary search on a linked list?
  - can't jump to middle
- Suppose we could jump to middle
  - use more pointers



This looks like a tree!



Binary Search Tree

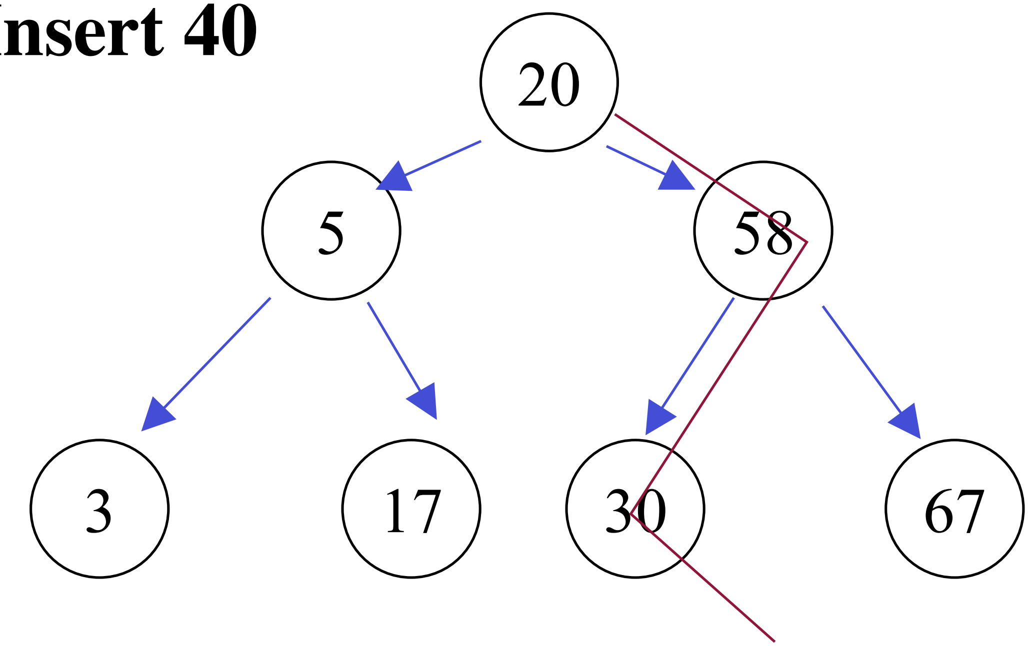
- data at a node is > any data in left subtree
- data at a node is < any data in right subtree
- Therefore, to print a BST in data order:
  - Print left subtree in data order
  - Print data
  - Print right subtree in data order

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

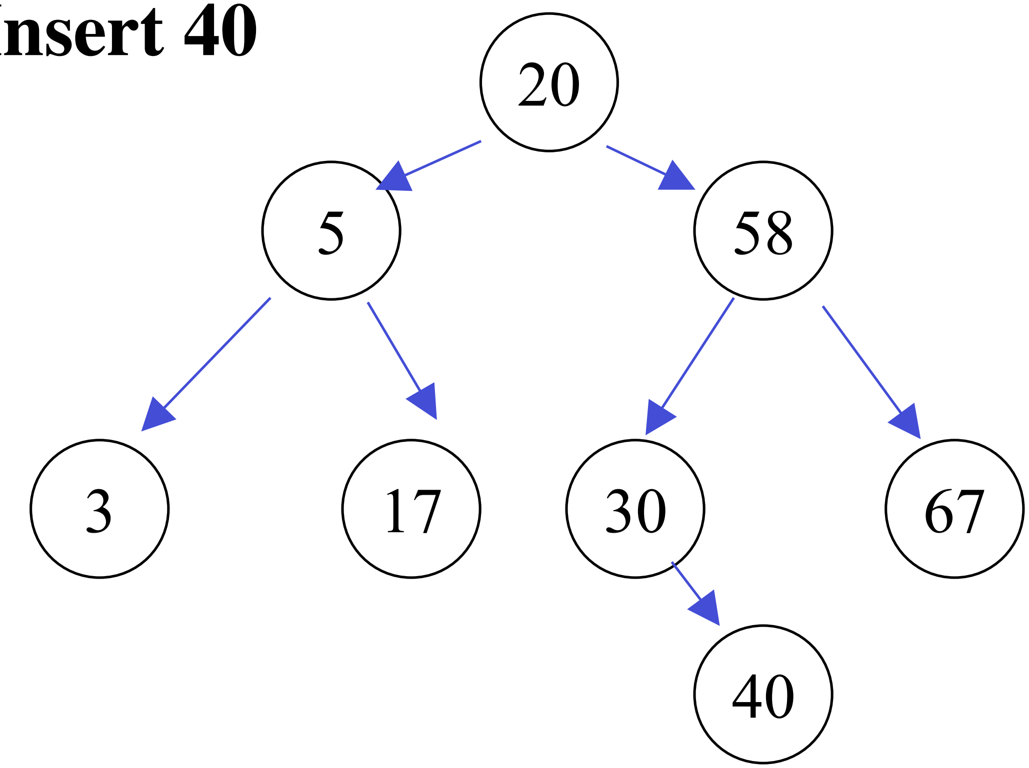
Insert

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



Insert

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

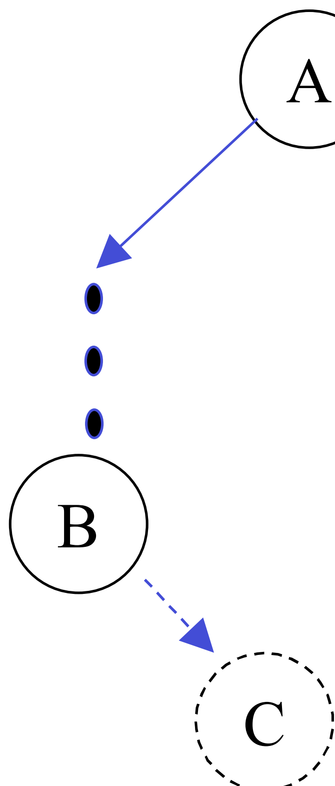


Delete

- 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

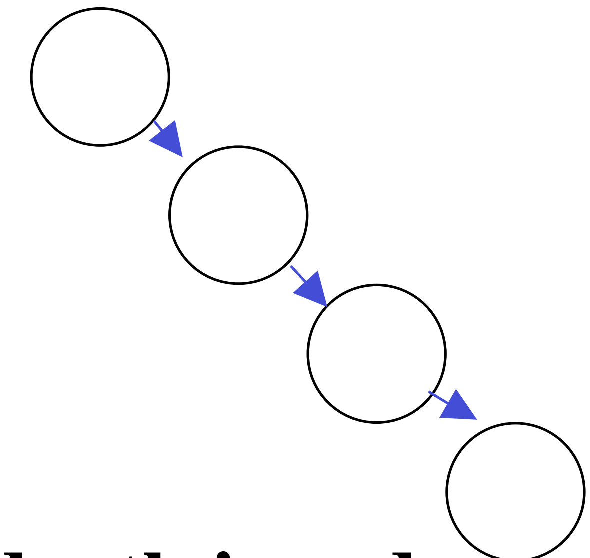
- **Replace data at node with data of inorder predecessor**
- **Delete inorder predecessor (which must have either 0 or 1 child)**

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**
  - **wait - I thought binary search was O(log n) worst case!?!**

Binary Search Trees

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



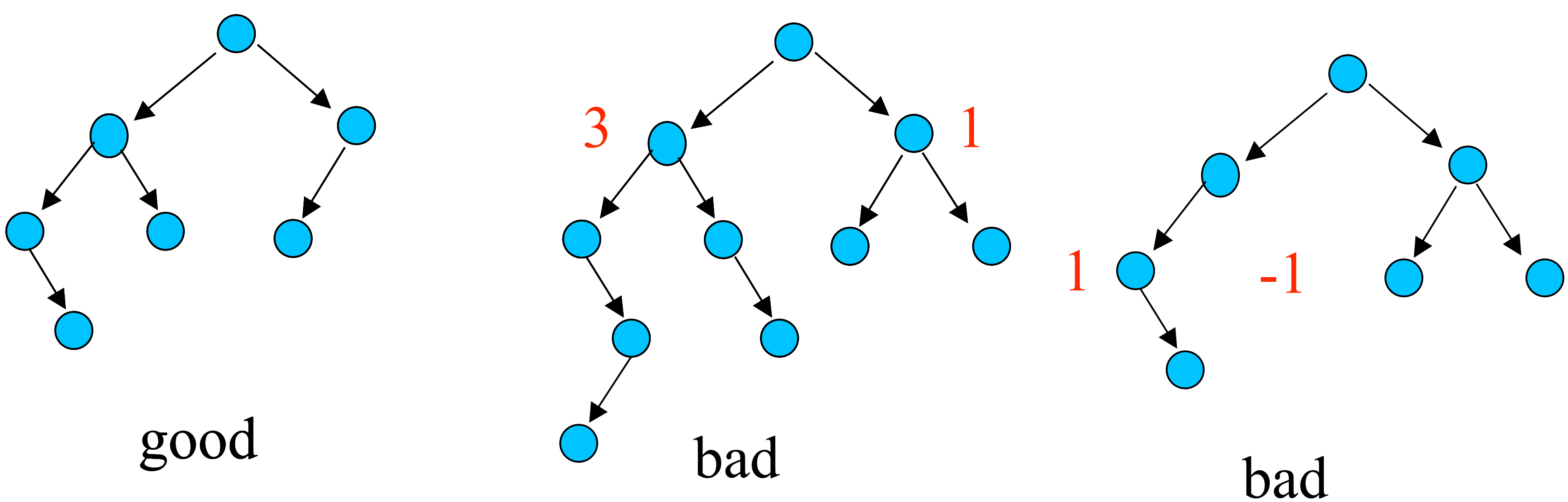
- **Worst case depth is order n, not logn**

•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

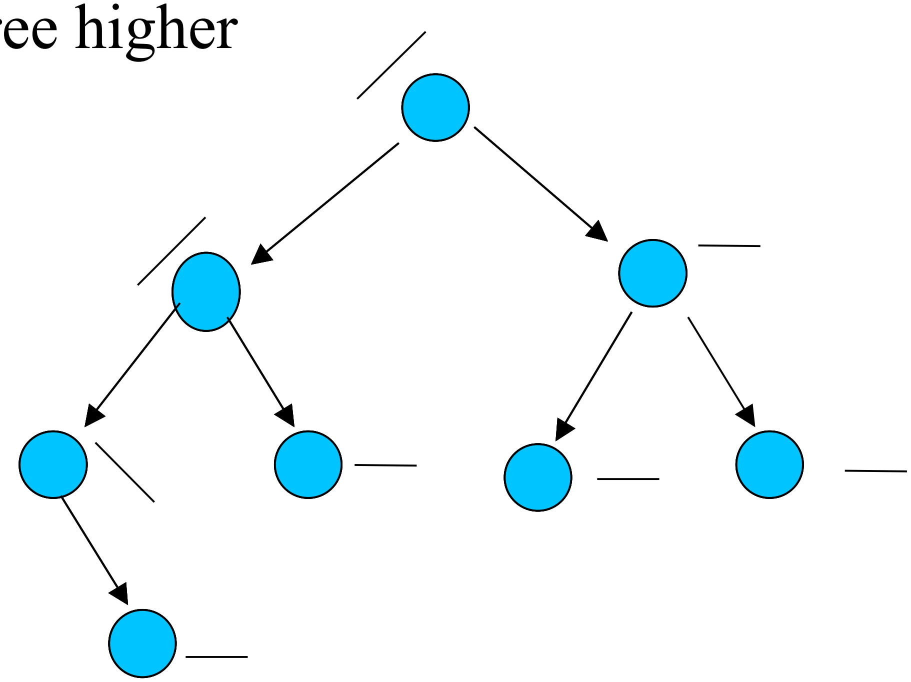
- **Binary Search Tree**
  - **Inorder traversal = data order**
- **Almost balanced**
  - **At every node, subtree heights same +/- 1**



Labeling an AVL Tree

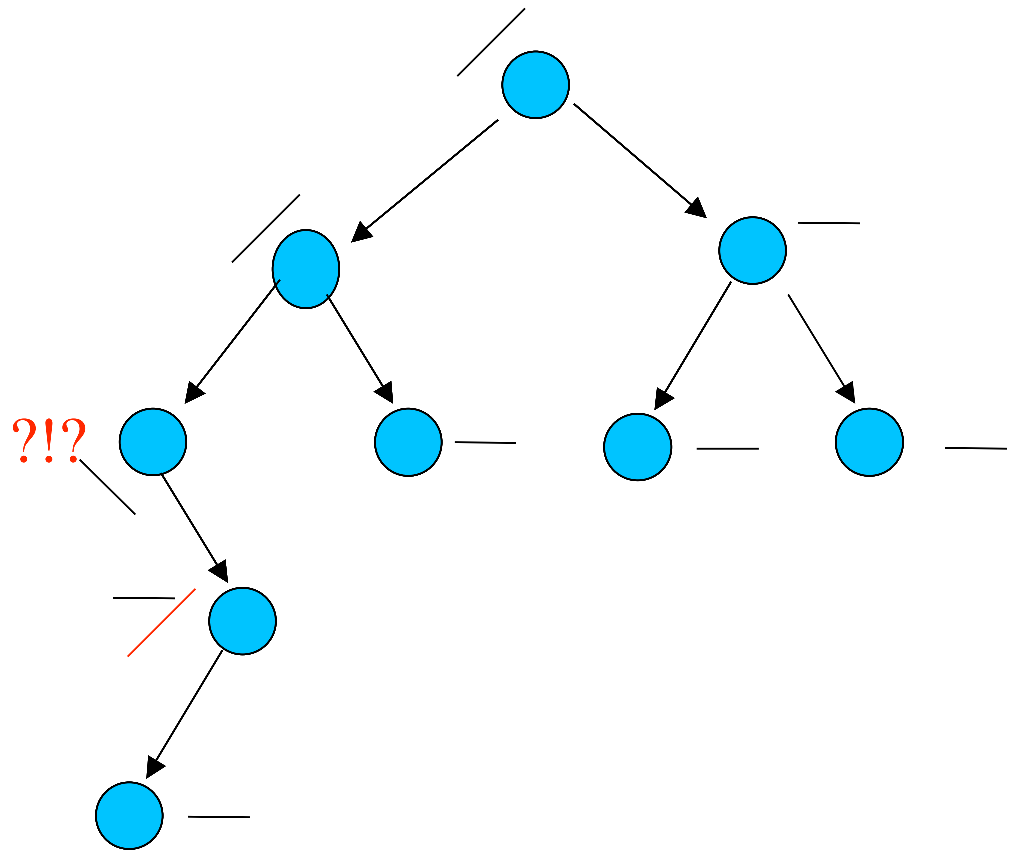
- **Label each node as**

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



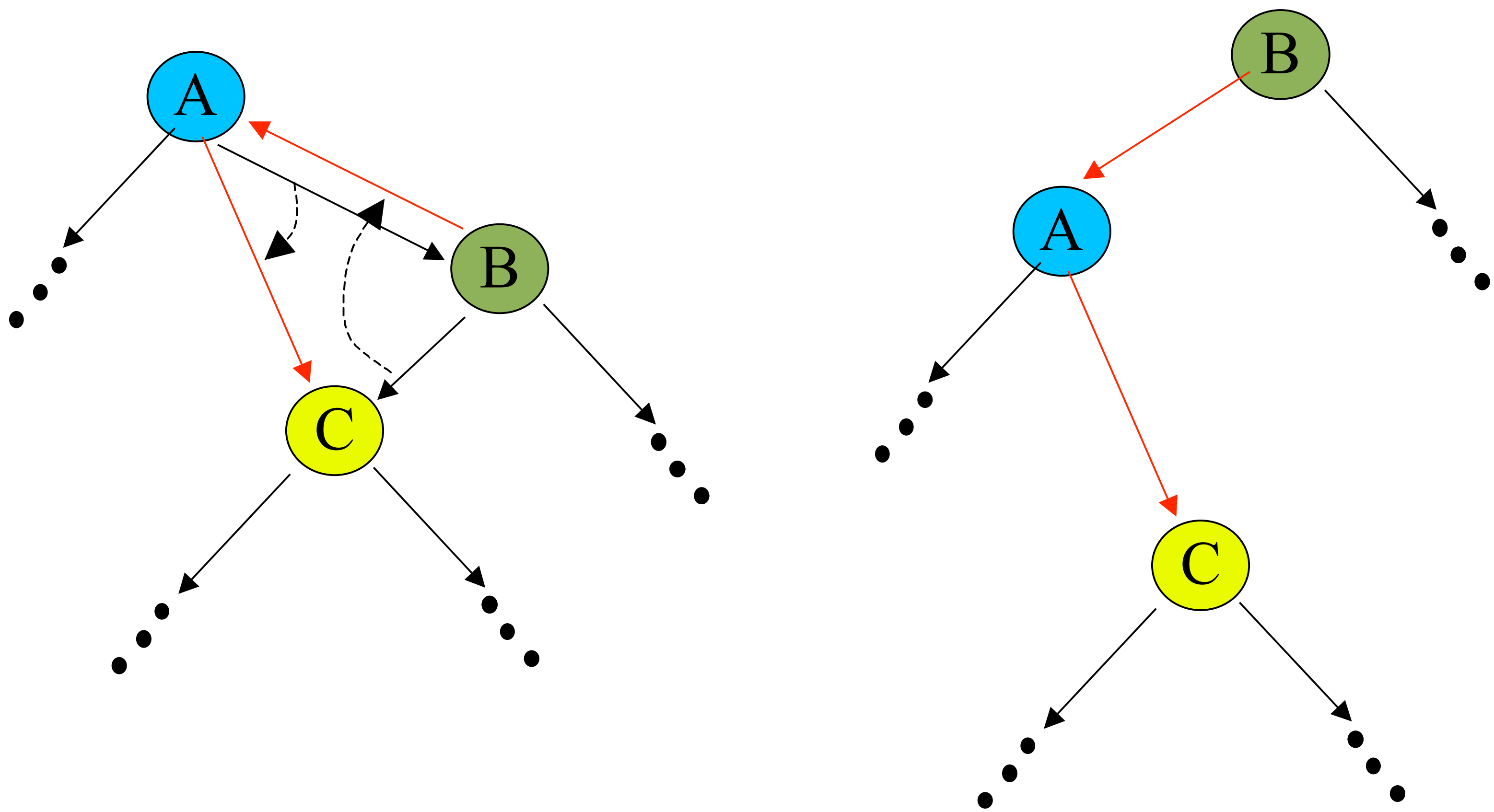
Rebalancing

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



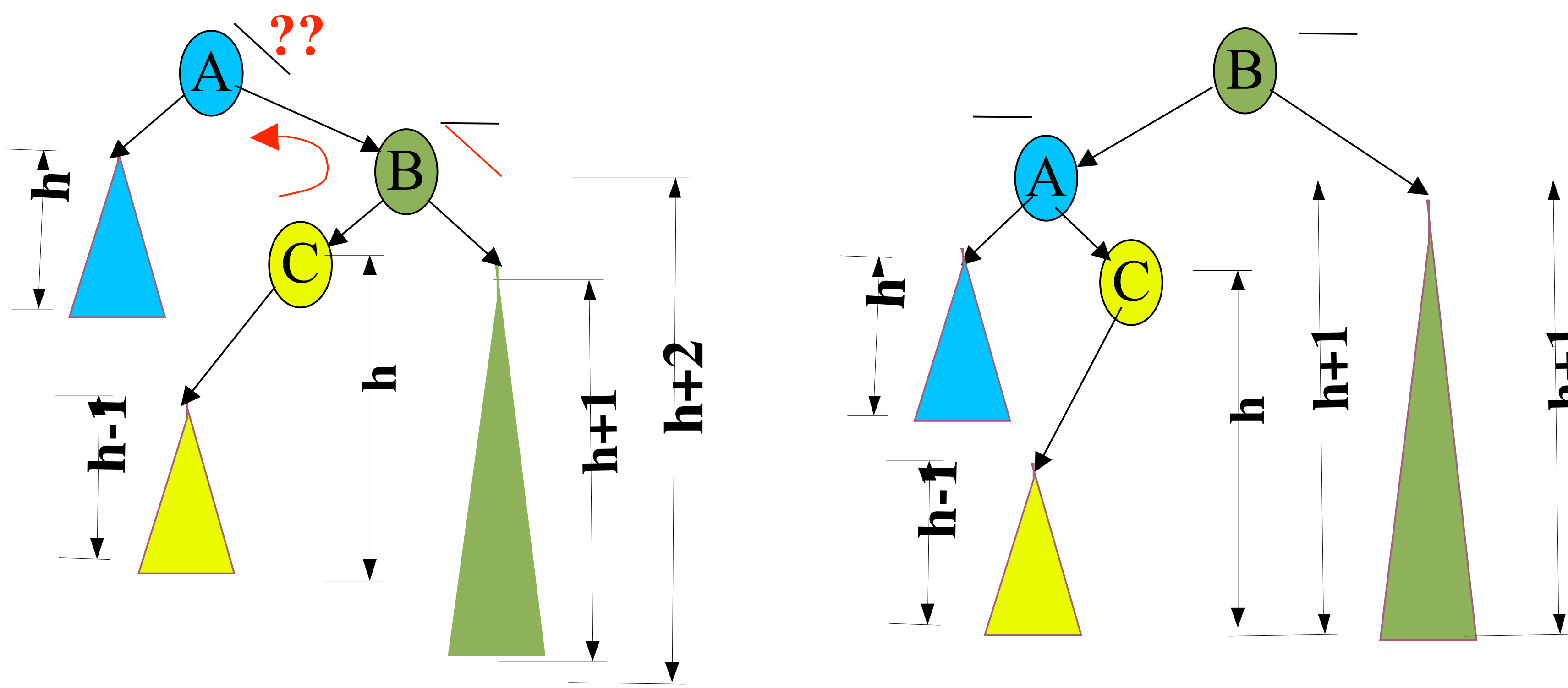
Rebalancing

- **Solution: Rotation**



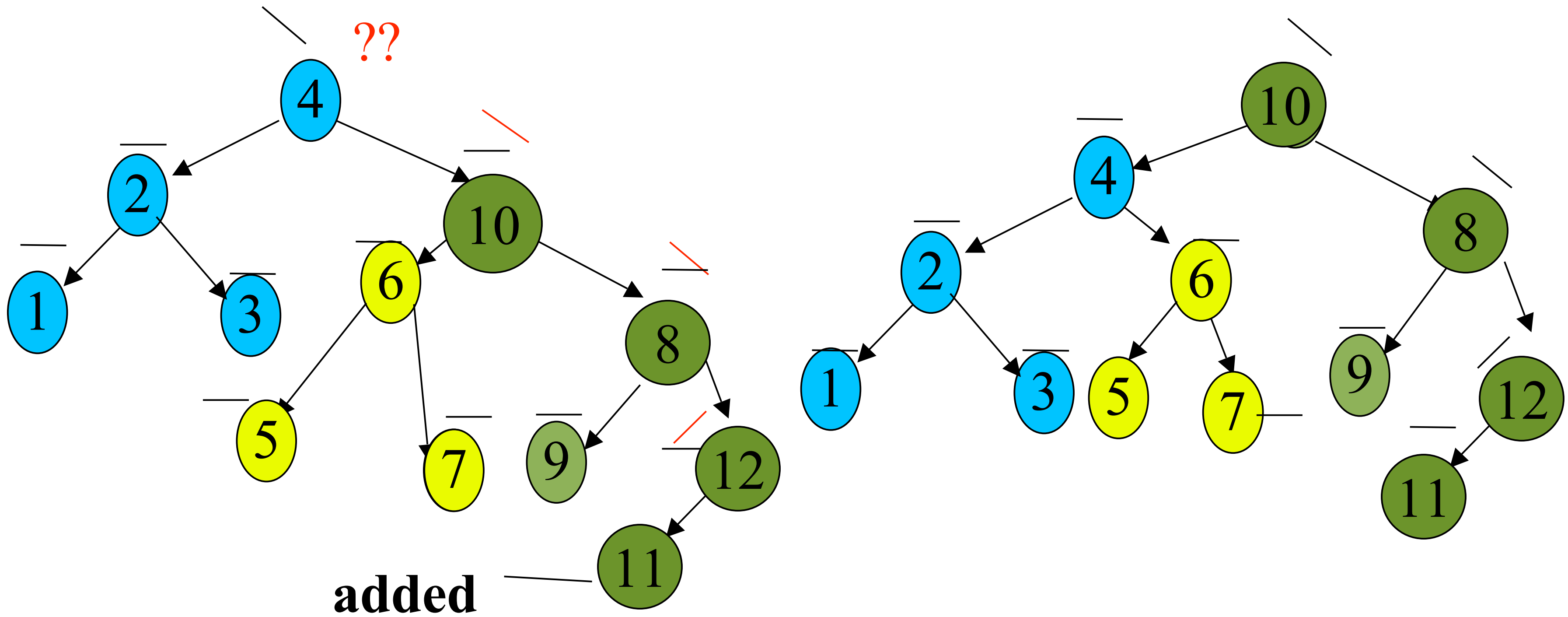
Rebalancing

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



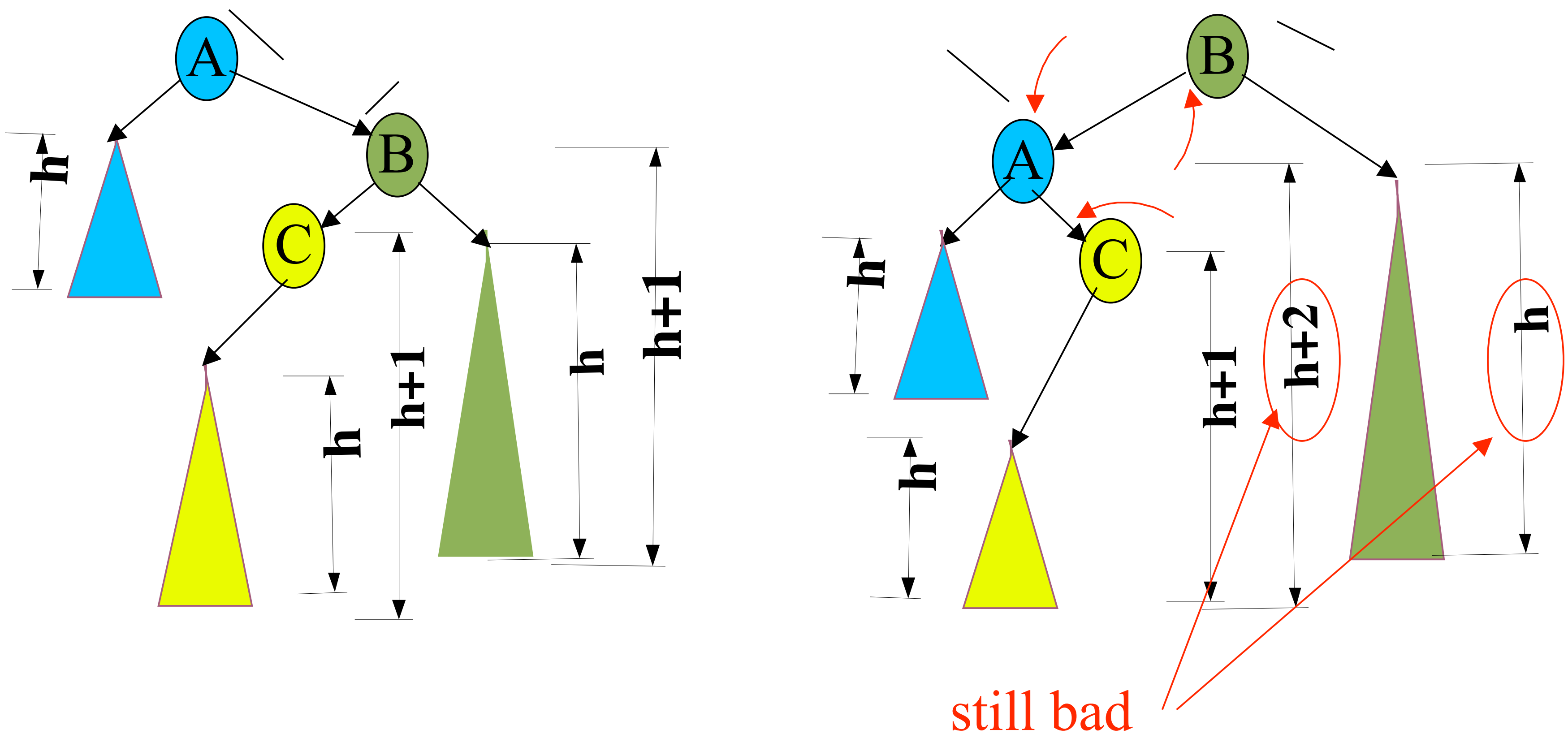
Rebalancing

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



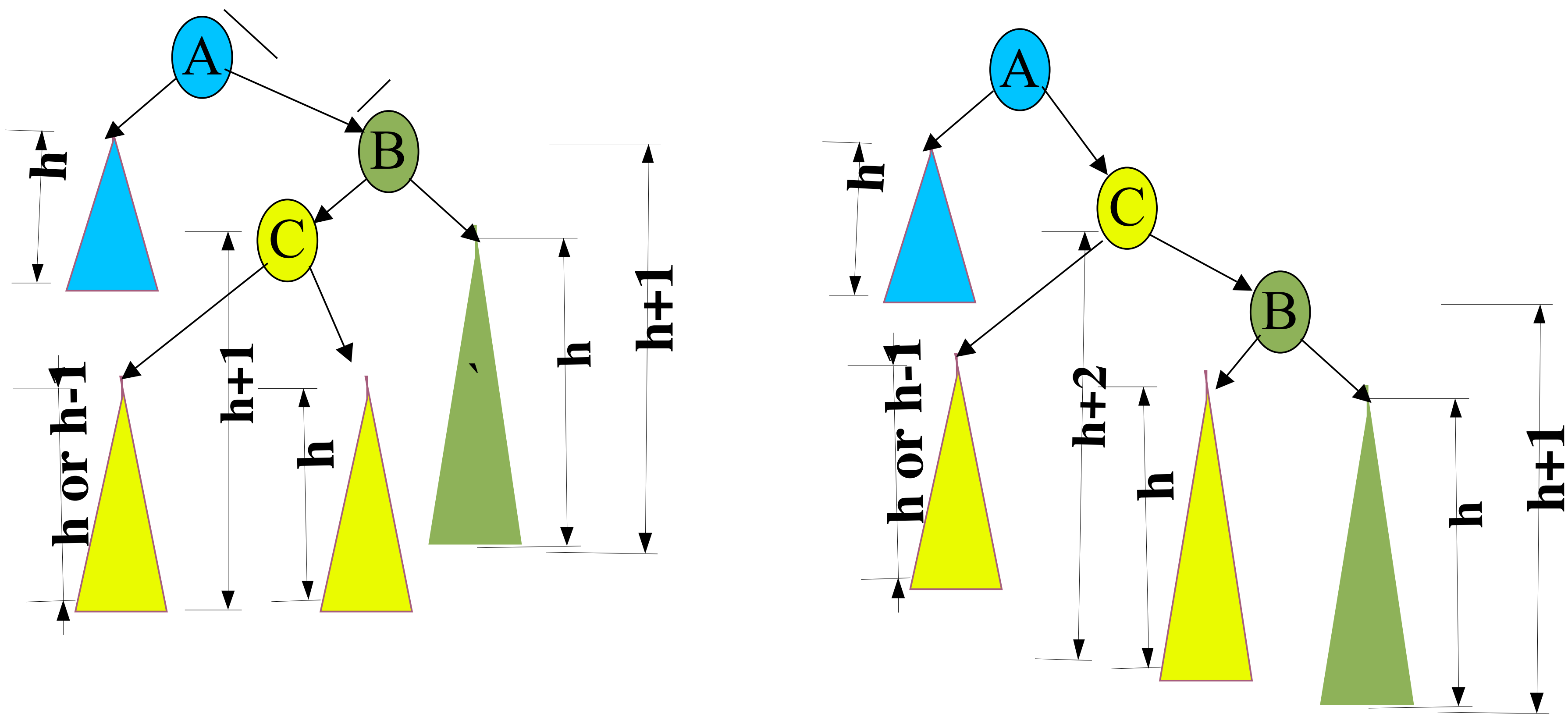
Rebalancing

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



Rebalancing

- **Solution: Rotate BC First**



Rebalancing

- **Solution: Then Rotate AC**