COMP20003
Algorithms and Data Structures
Balanced Trees

Nir Lipovetzky
Department of Computing and
Information Systems
University of Melbourne
Semester 2

So far…

- Dictionary search with slow look-up or insertion:
  - Lists, sorted and unsorted
  - Array, unsorted
  - Sorted array has log n lookup, but n^2 build
- Binary search tree: good average case, but very bad worst case.

Balanced trees

- Binary search tree:
  - Average case insertion and search: log n
  - Worst case for both: O(n)
- So, nice and simple, usually good enough, but not reliable.

This section

- How to get a binary search tree to stay balanced…
- … or almost balanced…
- … no matter what order the data are inserted.
- Note: this material is not covered in Skiena. It is essential knowledge for any computer scientist, however, and *is* examinable.

Balanced trees

- Idea: make a binary search tree perfectly (or almost perfectly) balanced.
- In a balanced tree of *n* items, the height will be O(*log n*).
  - Perfectly balanced tree, height = log n, exactly.
  - Balanced tree, height = O(log n).
  - Therefore build in balanced tree is O(n log n)
  - Search is O(log n).

Balanced tree implementations

- AVL trees
- 2-3-4 trees
- B+ trees
- Red-black trees

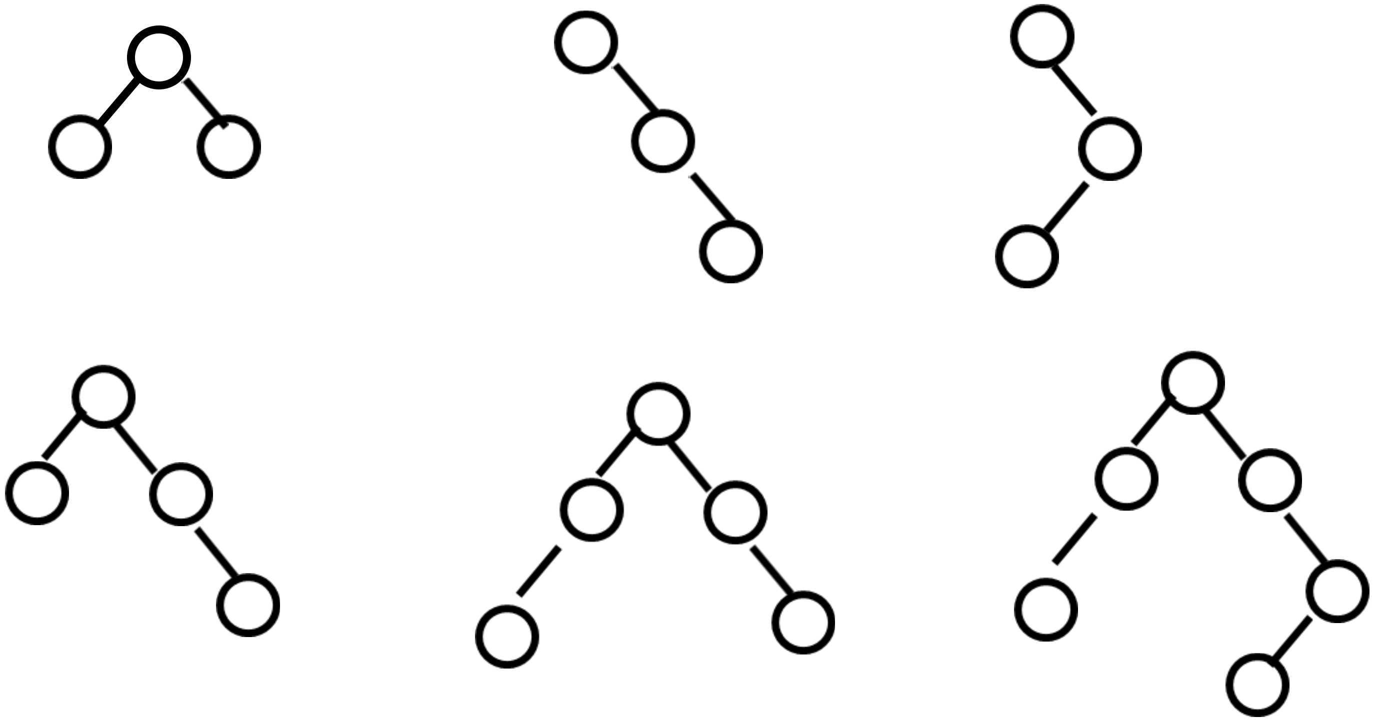
Balanced Trees and Binary Search Trees

- In balanced trees, during insertion there are mechanisms for making sure the tree does not grow unbalanced.
- At the same time, the bst ordering is preserved.
- So, search in a balanced tree is exactly the same as in a bst.
- The only difference is that it is O(log n).

AVL Trees

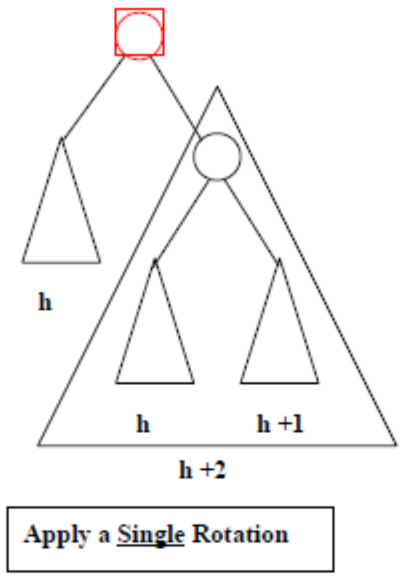
- The first balanced tree.
- Insert node + Keep track of height of subtrees of *every* node.
  - Balance node every time difference between subtree heights is >1.
  - Basic balancing operation: Rotation.
- Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences **146**: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady **3**:1259–1263, 1962.

Do these trees satisfy the AVL condition? Why / why not?

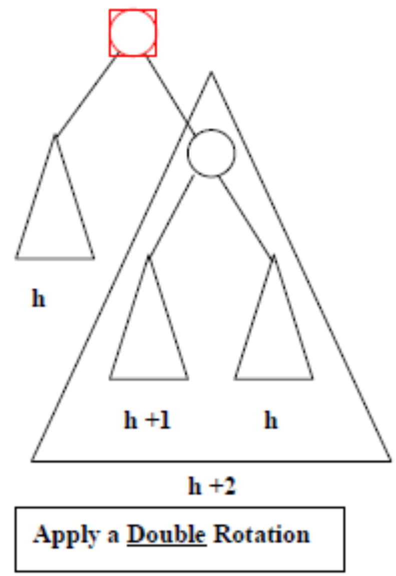


Non-AVL Trees caused by...

Outside insertion

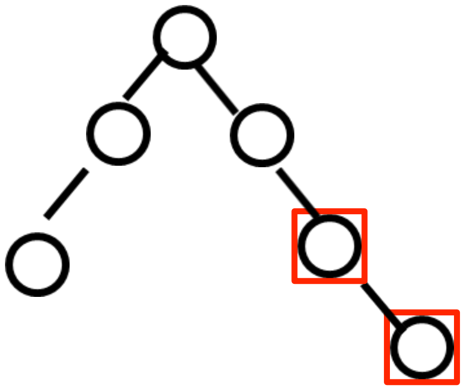


Inside insertion

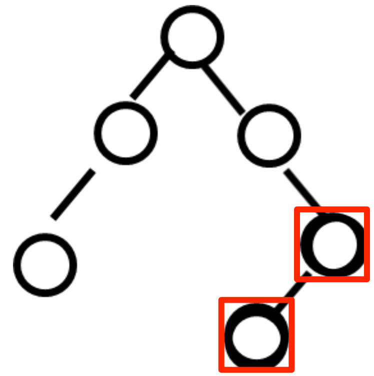


Symmetrical case is handled identically!

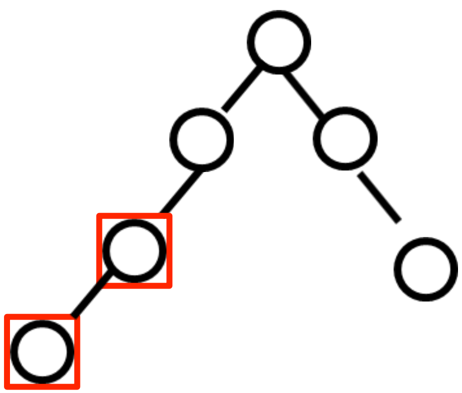
Unbalanced tree Categories



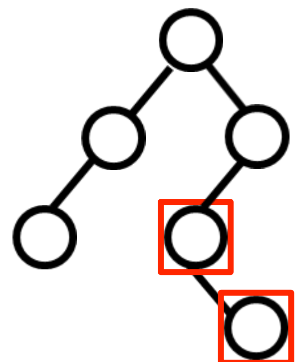
Outside Right **Sub-tree**
Right **Child** (RR)



Inside Right **Sub-tree**
Left **Child** (RL)



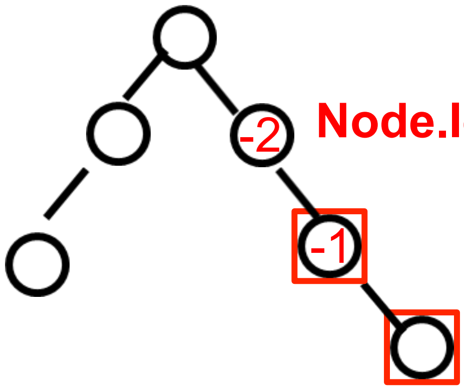
Outside Left **Sub-tree**
Left **Child** (LL)



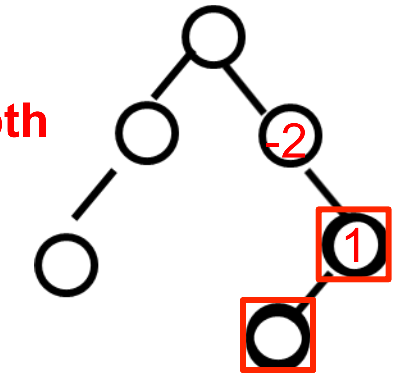
Inside Left **Sub-tree**
Right **Child** (RL)

Unbalanced tree Categories

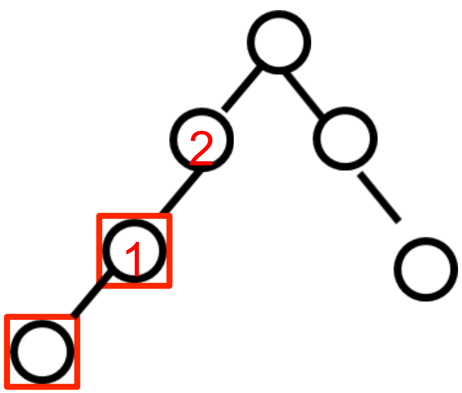
Counter =
Node.left.depth – node.right.depth



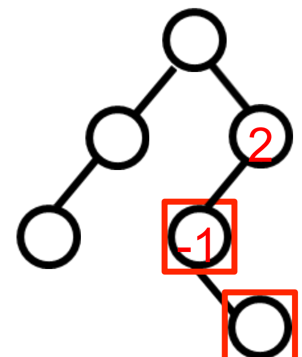
Outside Right **Sub-tree**
Right **Child** (RR)



Inside Right **Sub-tree**
Left **Child** (RL)



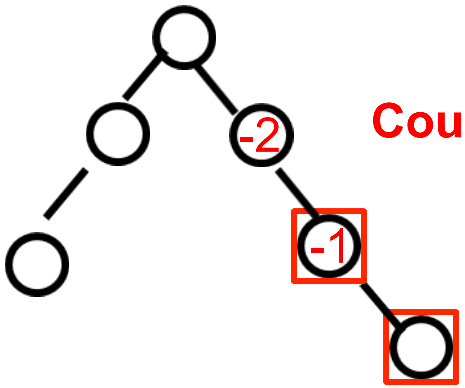
Outside Left **Sub-tree**
Left **Child** (LL)



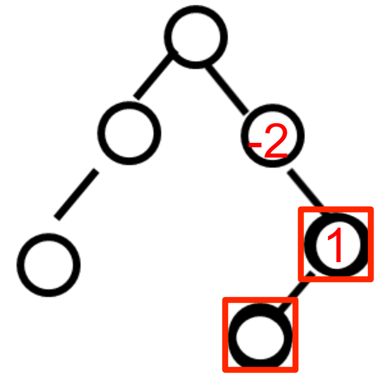
Inside Left **Sub-tree**
Right **Child** (RL)

Unbalanced tree Categories

If Counter < -1 or
Counter > 1 → Non-Avl Node!

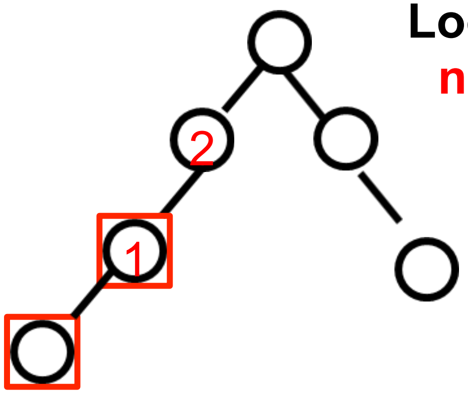


Outside Right **Sub-tree**
Right **Child** (RR)

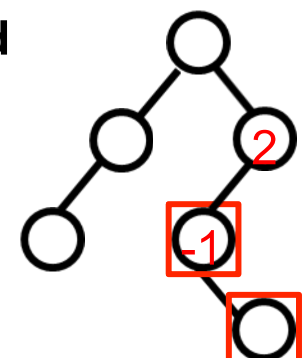


Inside Right **Sub-tree**
Left **Child** (RL)

Look at the node.Counter and node.child.Counter to know Which Rotation to do



Outside Left **Sub-tree**
Left **Child** (LL)



Inside Left **Sub-tree**
Right **Child** (RL)

Correcting operation: Rotation

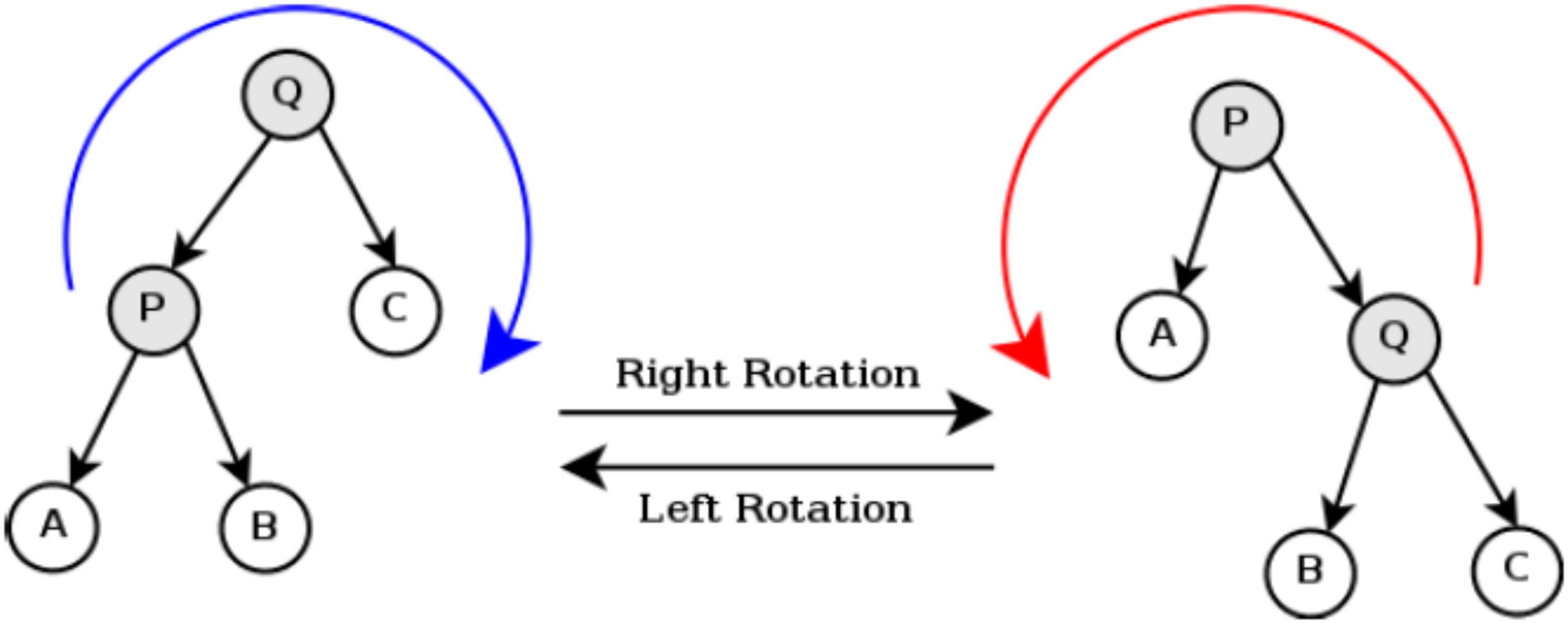
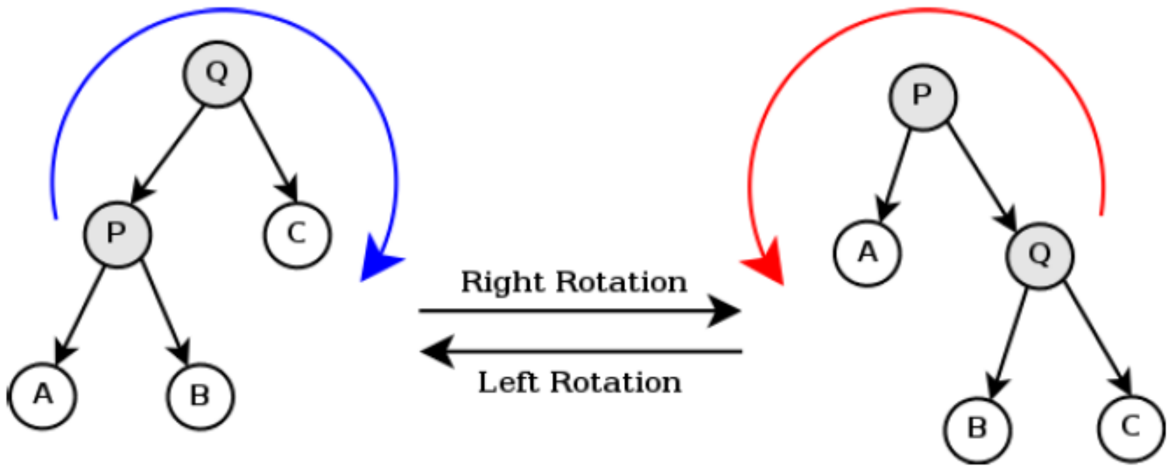


Image from Wikipedia: Tree rotation

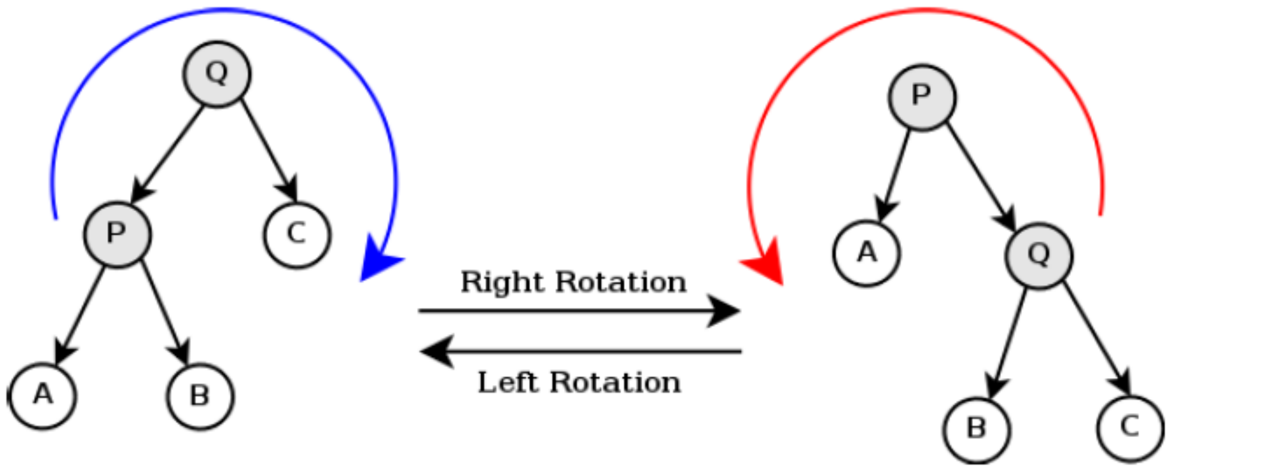



```
RotateR(node)
{
    left = node.Left;
    leftRight = left.Right;
    parent = node.Parent;

    left.Parent = parent;
    left.Right = node;
    node.Left = leftRight;
    node.Parent = left;
}
```

```
RotateR(Q)
{
    left = P;
    leftRight = B;
    parent = Null;

    P.Parent = Null;
    P.Right = Q;
    Q.Left = B;
    Q.Parent = P;
}
```



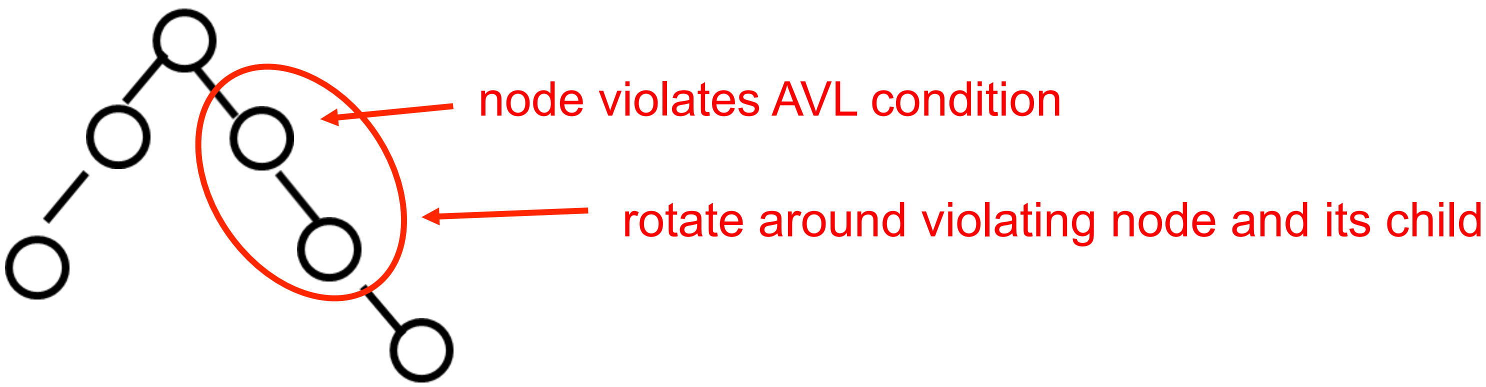

```
RotateL(node)
{
    right = node.Right;
    rightLeft = right.Light;
    parent = node.Parent;

    right.Parent = parent;
    right.Left = node;
    node.Right = rightLeft;
    node.Parent = right;
}
```

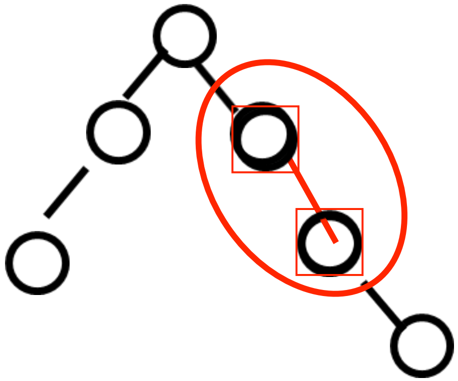
```
RotateL(P)
{
    right = Q;
    rightLeft = B;
    parent = Null;

    Q.Parent = Null;
    Q.Left = P;
    P.Right = B;
    P.Parent = Q;
}
```

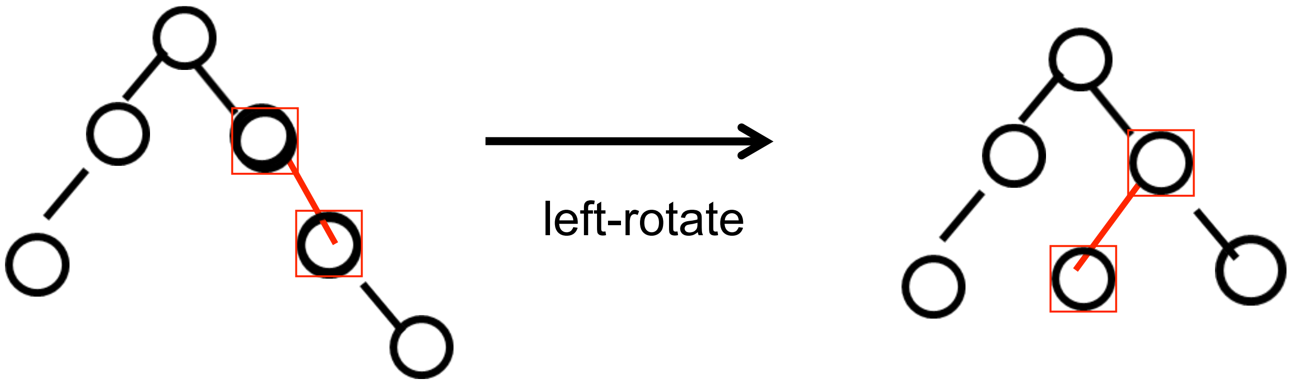
How does rotation help balance a tree?



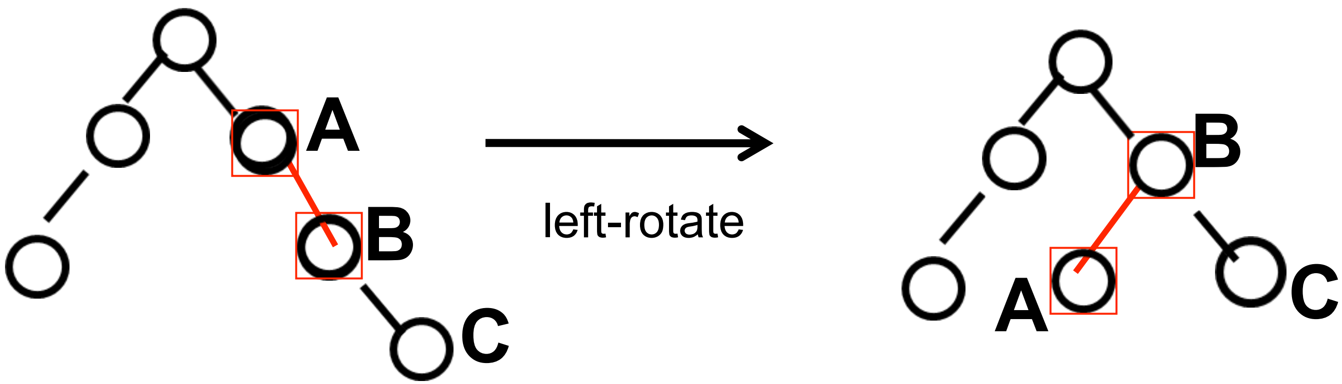
How does rotation help balance a tree?



How does rotation help balance a tree?



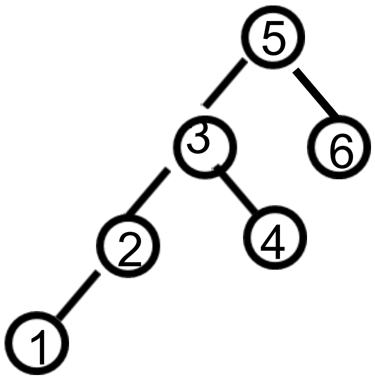
How does rotation help balance a tree?

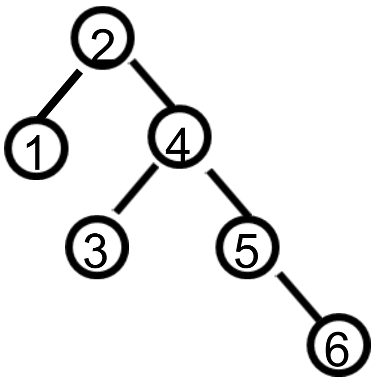


- LeftLeft/RightRight-rotation (single) :
  - Take non-AVL node:
    - Rotate Child and node
    - Keep ordered subtree!

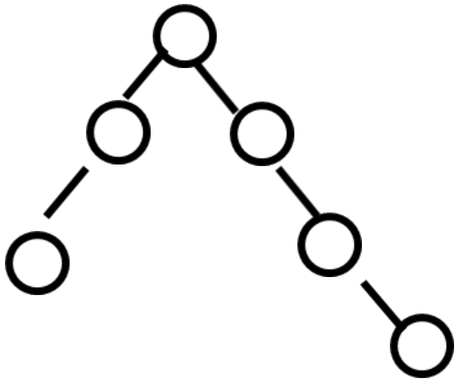
Which Rotation should we apply?

Excercise: Rotate? If so, do it...





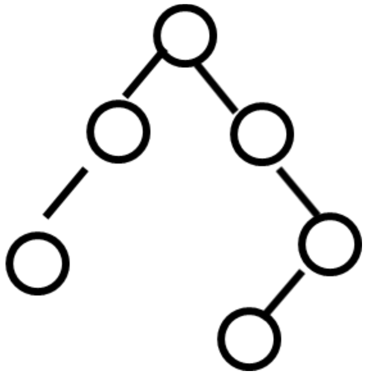
How does rotation help balance a tree?



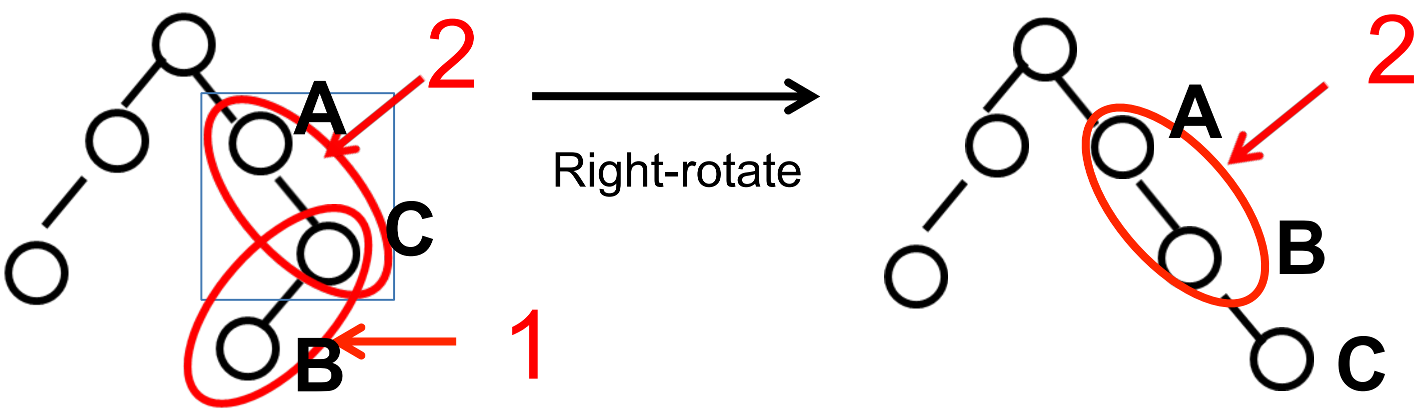
We have shown that:
in these cases (LL/RR),
Rotation rebalances
the tree.

How does rotation help balance a tree?

What about
in these cases
(LR/RL)?



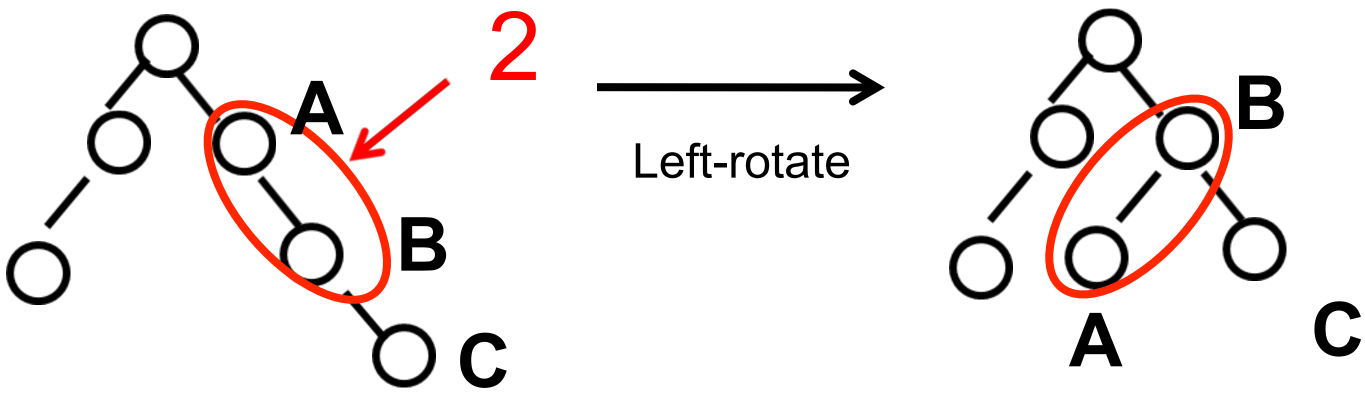
RL and LR: Double rotation



Right Left (RL) double Rotation:

- First rotation swaps Grandchild and child (Right Rotation)

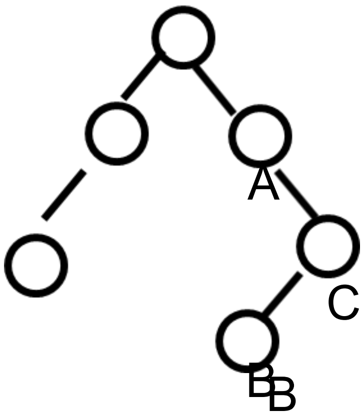
RL and LR: Double rotation

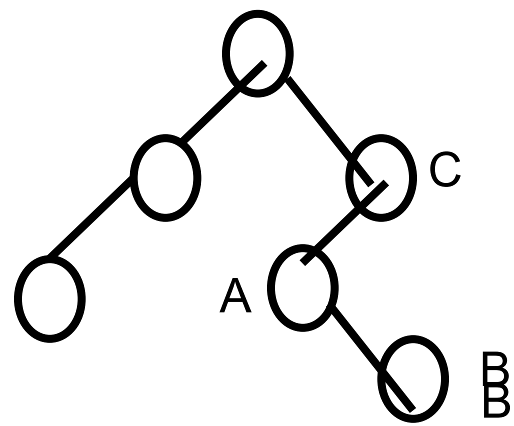


Right Left (RL) double Rotation:

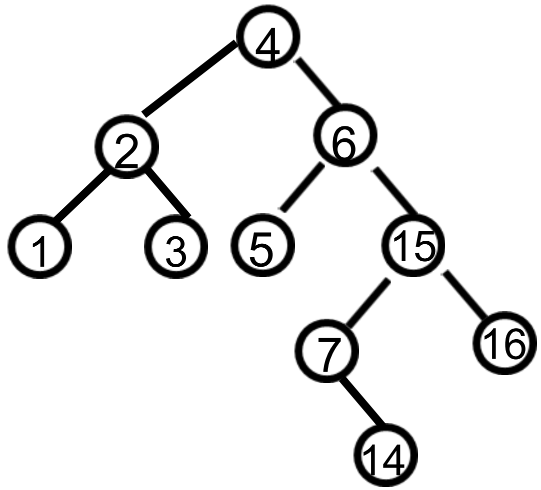
- Second rotation swaps Parent and child (Left Rotation) , as before

Why not just left rotate?





Excercise: Rotate? If so, do it...



- Note that since rotations preserve the bst ordering of the tree, search is the same as for a bst.

AVL Trees

- Good features:
  - Tree is always reasonably balanced.
  - Actually height $\leq$ 1.44 $\log_2 n$.
  - Therefore complexity for any search is O(?).
- Less than ideal features:
  - Very fiddly to code: must keep track of insertion path and size of all subtrees.
  - Balancing adds time (but constant time).

```
node* insert ( node* tree, node* new_node )
{
    if ( tree == NULL )
      tree = new_node;
    else if ( new_node->key < tree->key ) {
      tree->left = insert ( tree->left, new_node );
      /* Fifty lines of left balancing code */
    }
   else {
     tree->right = insert ( tree->right, new_node );
     /* Fifty lines of right balancing code */
   }
   return tree;
}
```

Other resources for AVL trees

- Tutorial on AVL trees by Ananda Gunawardena, Carnegie Mellon Institute: http://www.youtube.com/watch?v=EsgAUiXbOBo (25 minutes)

- Interactive Demo! https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Balanced Trees (so far)

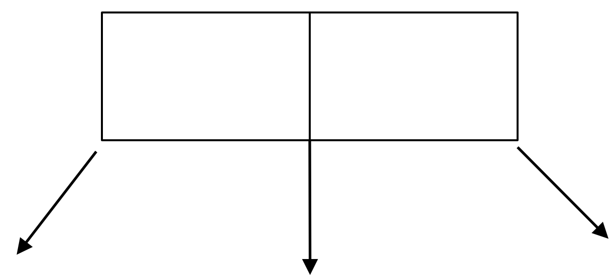
- AVL trees use rotation to keep the tree balanced.
- Rotations are a general operation, used in other situations, not just AVL trees.

2,3,4-Trees: Overview

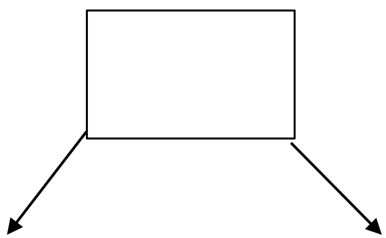
- Trees do not have to be binary!
- Nodes in 2,3,4-Trees have:
  - 1, 2, or 3 keys
  - 2, 3, or 4 pointers, correspondingly.
- Items are inserted *only into leaf nodes*.
- When 4-nodes are full – split to accommodate new items.
- Tree grows in height slowly.

2-3-4 Tree nodes

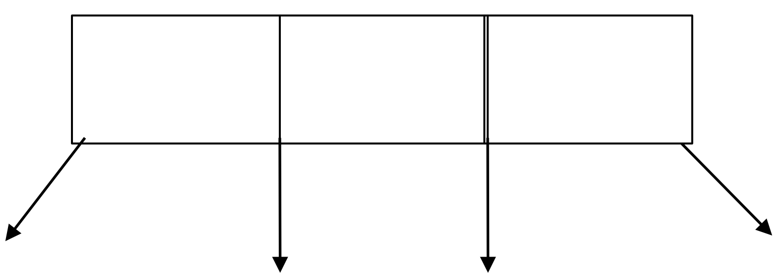
3-node



2-node



4-node



2,3,4-Trees

- Note that tree remains balanced *even when items are inserted in sorted order.*
- Height of tree: between $\log_4 n$ and $\log_2 n$

B+-Trees

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- B+-trees: generalization of the 2,3,4 tree.
- Nodes have many pointers:
  - Typically 256-512
  - Depth of tree is $\log_{(\text{very large number})} n$
- Used for storing large databases on disk, where accesses are very expensive.

Red-Black Trees

- Red-black trees implement a 2-3-4 tree as a binary search tree, using node rotation to keep the balance.
- Beyond the scope of this subject.
- An excellent description is found in Sedgewick, Algorithms in C, Parts 1-4, Section 13.4.

Splay Trees

- A splay tree is a self-adjusting tree.
- Insertion:
  - Insert as for bst.
  - “Splay” new node to the root.
- Splay: do a series of rotations, that bring the node closer to the root.

Splay Trees

- A splay tree is a self-adjusting tree.
- Search:
  - Search as for bst
  - “Splay” the searched-for node to the root.
- Note: might be O(n) search in a stick tree, but then splaying bushes out the tree.

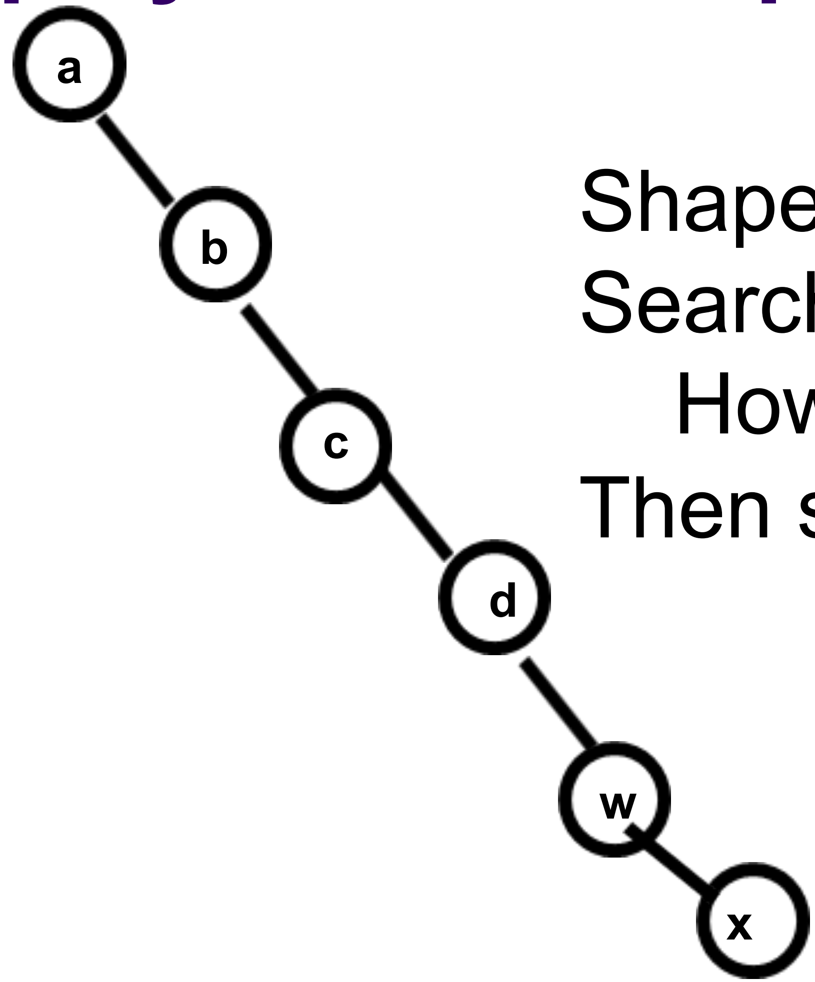
Splay Trees

- Overall:
  - A single search might take linear time.
  - BUT over time:
    - The tree gets bushier.
    - More highly accessed nodes are closer to the root.

Splay Trees

- Splay tree analysis: amortized over a series of searches.
- Cope well with non-uniform access.
- Sleator and Tarjan, *Self-Adjusting Binary Search Trees*, *JACM* **32(3)**, 1985, 652-686.

Splay Tree Example



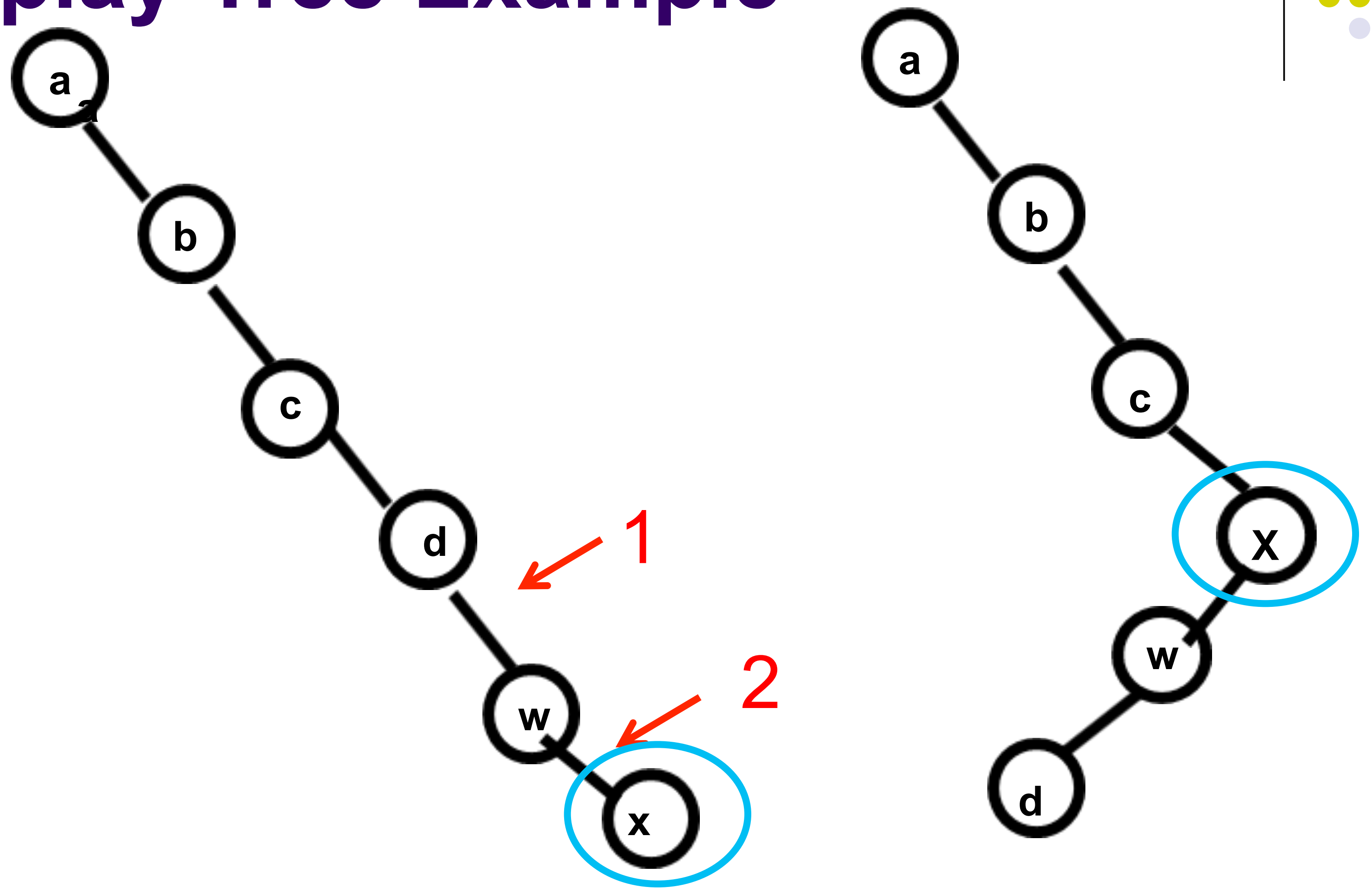
Shape: a stick, height 5

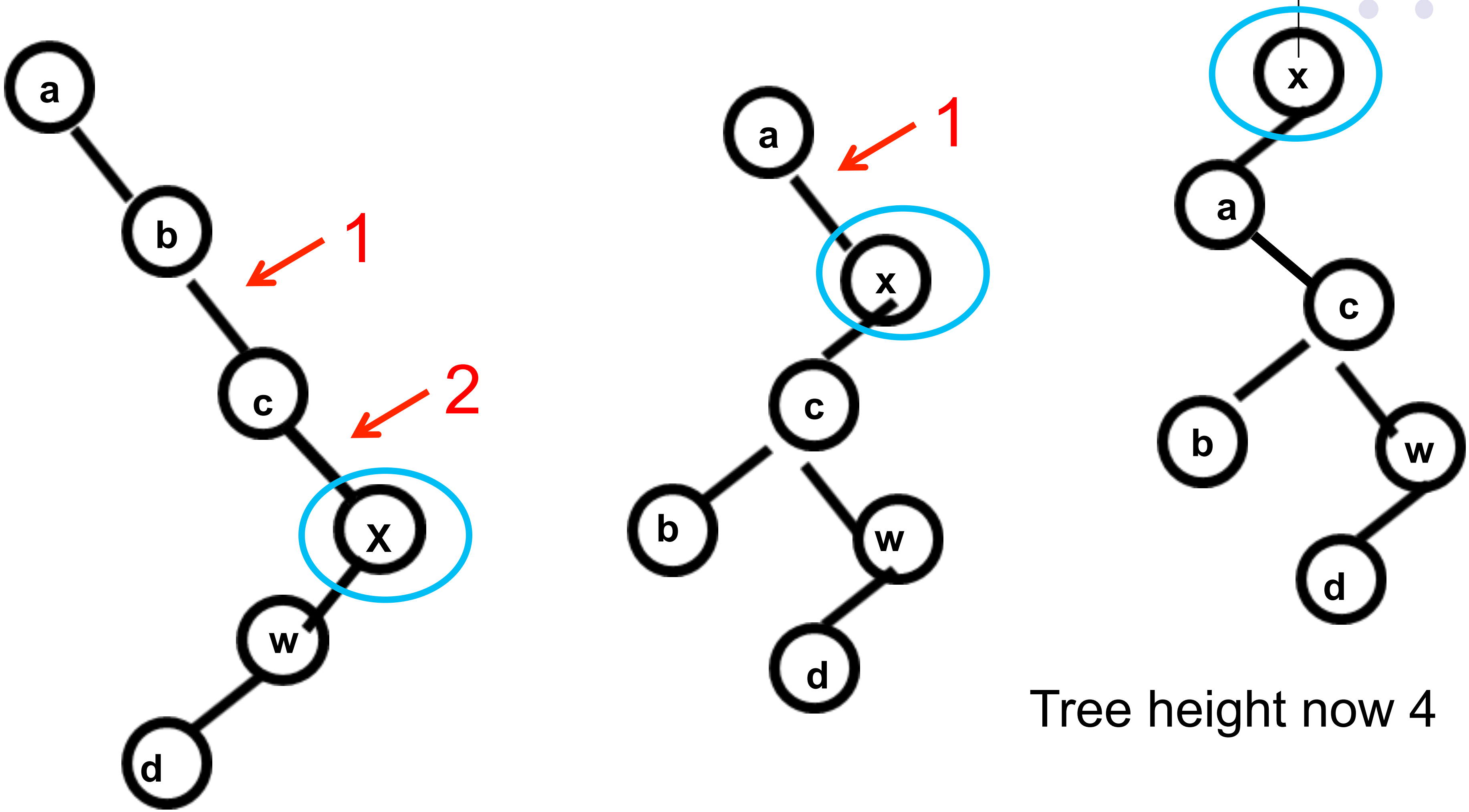
Search for node x:

How many comparisons?

Then splay x to root.

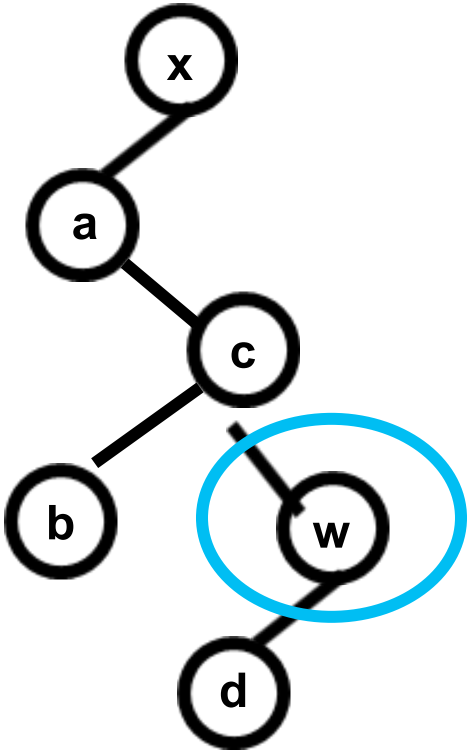
Splay Tree Example





Tree height now 4

Now search for w.



Good things about balanced trees

- Tree is balanced:
  - Always relatively balanced for AVL, 2-3-4, B.
  - On average balanced for splay trees.
- O(log n) search for AVL, 2-3-4, B.

Skip Lists: A Probabilistic Alternative to Balanced Trees

- Skip lists are lists pretending to be balanced trees.
- They have excellent log n search behavior, BUT…
- … they are a probabilistic algorithm.
  - There is an extremely high probability that a skip list search will complete in log n time.
  - But there is always an infinitesimal probability of worst case linear behavior.