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)

Although simple, it's usually good enough, but not reliable

This section

- How to get a BST 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 BST perfectly (or almost perfectly) balanced
- In a balanced tree of *n* items, height is O(*log n*)
  - Perfectly balanced tree, height = log n, exactly
  - Balanced tree, height = O(log n).
- Therefore build a 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 binary tree
- The only difference is that it is O(log n)

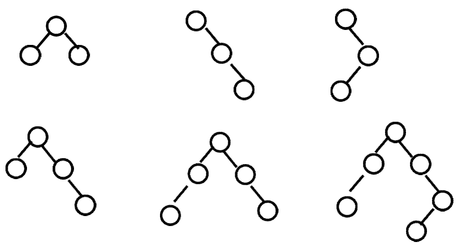
AVL Trees

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.

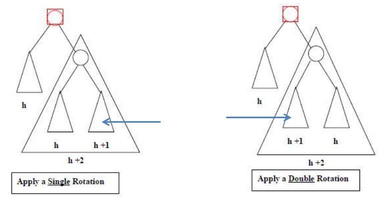
Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences **146**: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady **3**:1259–1263, 1962.

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



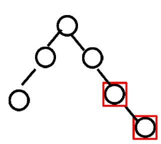
Non-AVL Trees caused by...

Outside insertion Inside insertion

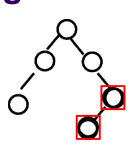


Symmetrical case is handled identically!

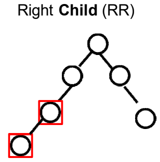
Unbalanced tree Categories



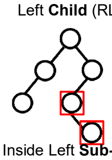
Outside Right **Sub-tree**
Right **Child** (RR)



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



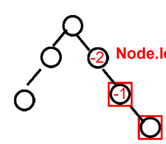
Outside Left **Sub-tree**
Left **Child** (LL)



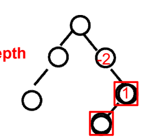
Inside Left **Sub-tree**
Right **Child** (LR)

Unbalanced tree Categories

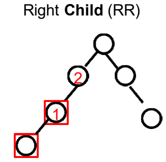
Counter = Node.left.depth – node.right.depth



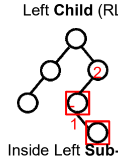
Outside Right **Sub-tree**
Right **Child** (RR)



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

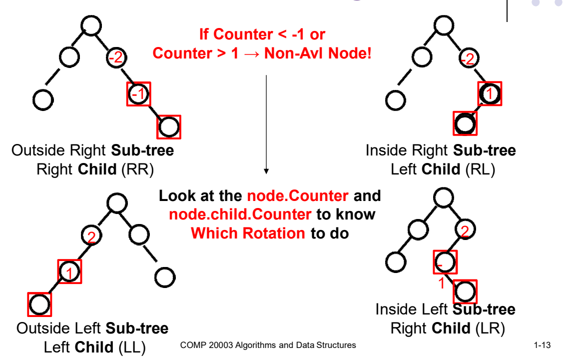


Outside Left **Sub-tree**
Left **Child** (LL)

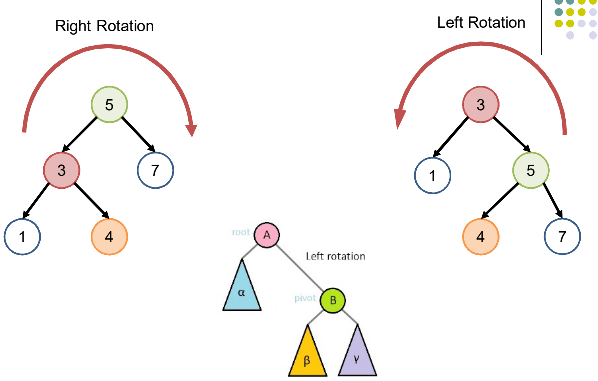


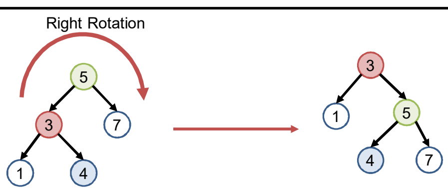
Inside Left **Sub-tree**
Right **Child** (LR)

Unbalanced tree Categories



Rotation



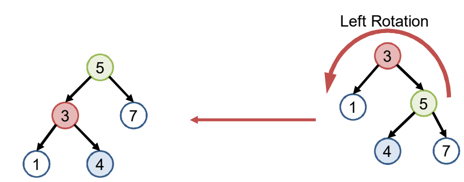


RotateR(node)
```
{
    //Auxiliary variables
    left = node.Left;
    leftRight = left.Right;
    parent = node.Parent;

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

RotateR(5)
```
{
    //Auxiliary variables
    left = 3;
    leftRight = 4;
    parent = Null;

    //Operations
    3.Parent = Null;
    3.Right = 5;
    5.Left = 4;
    5.Parent = 3;
}
```

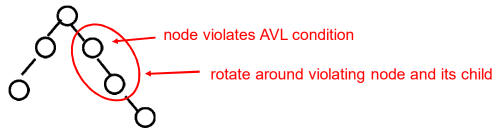
RotateL(node)
```
{
    //Auxiliary variables
    right = node.Right;
    rightLeft = right.Light;
    parent = node.Parent;

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

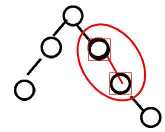
RotateL(3)
```
{
    //Auxiliary variables
    right = 5;
    rightLeft = 4;
    parent = Null;

    //Operations
    5.Parent = Null;
    5.Left = 3;
    3.Right = 4;
    3.Parent = 5;
}
```

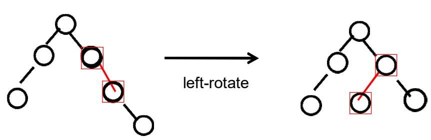
How does rotation help balance a tree?



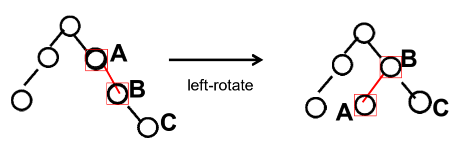
How does rotation help balance a tree?



How does rotation help balance a tree?

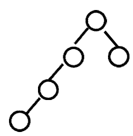


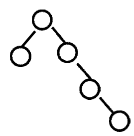
How does rotation help balance a tree?



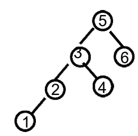
- LeftLeft/RightRight-rotation (single) :
  - Take non-AVL node:
    - Rotate Child and node
    - Keep ordered subtree!

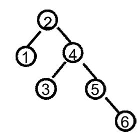
Which Rotation should we apply?



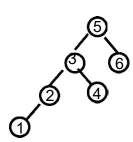


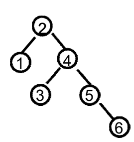
Excercise: Rotate? If so, do it...



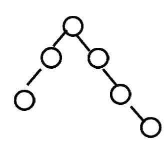


Excercise: Rotate? If so, do it...





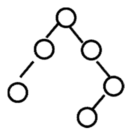
How does rotation help balance a tree?



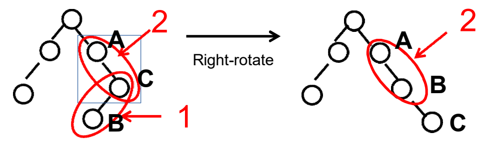
We have shown that: in these cases (LL/RR), Rotation rebalances the tree.

How does rotation help balance a tree?

What about
in these cases
(LR/RL)?

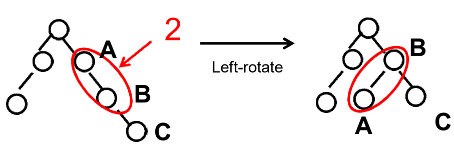


RL and LR: Double rotation

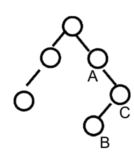


Right Left (RL) double Rotation:
- First rotation swaps Grandchild and child (Right Rotation)

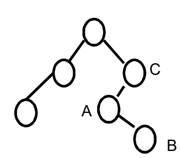
RL and LR: Double rotation



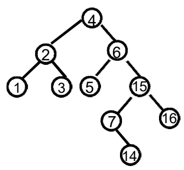
Right Left (RL) double Rotation:
- Second rotation swaps Parent and child (Left Rotation) , as before



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 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 A. Gunawardena, Carnegie Mellon Institute
http://www.youtube.com/watch?v=EsgAUiXbOBo (25 minutes)

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

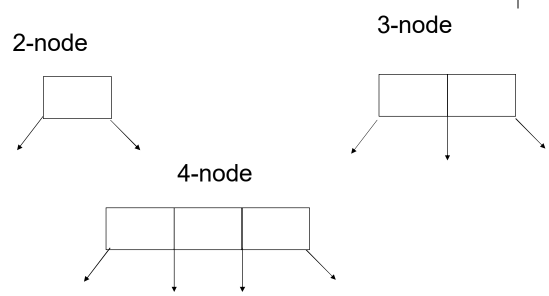
Balanced Trees (so far)

- 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 height grow slowly

2-3-4 Tree nodes



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 $\log_2 n$
- 2-3-4 also known as B-trees of order 4

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.
- Only leaf contain data, internal nodes are only intervals
- Leaves may include pointer to next leaf, to speed up sequential access

Red-Black Trees

- Red-black trees implement a 2-3-4 tree as a binary search tree, using rotation to keep 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

- Search:
  - Search as for BST
  - "Splay" the searched 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:

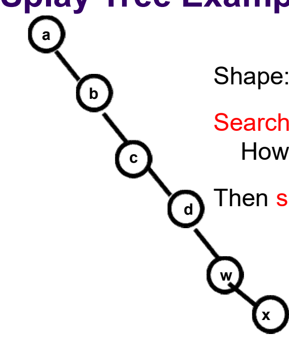
- The tree gets bushier.
- Highly accessed nodes are closer to the root

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

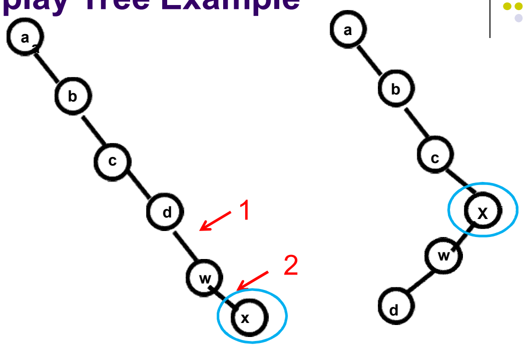


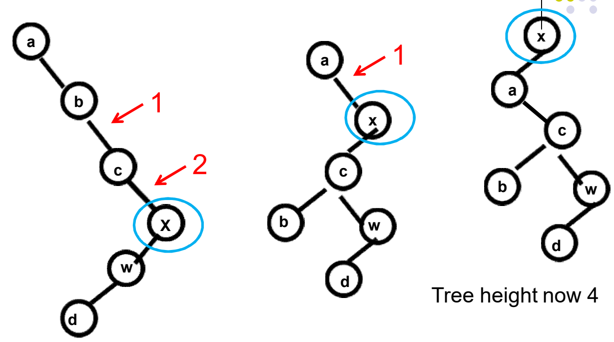
Shape: a stick, height 5

Search for node x:
How many comparisons?

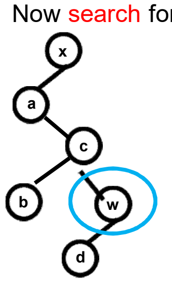
Then splay x to root.

Splay Tree Example





Now search for w



Good things about balanced trees

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 behaviour,
- 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 behaviour