Trees

- Abstract data type used for data organization

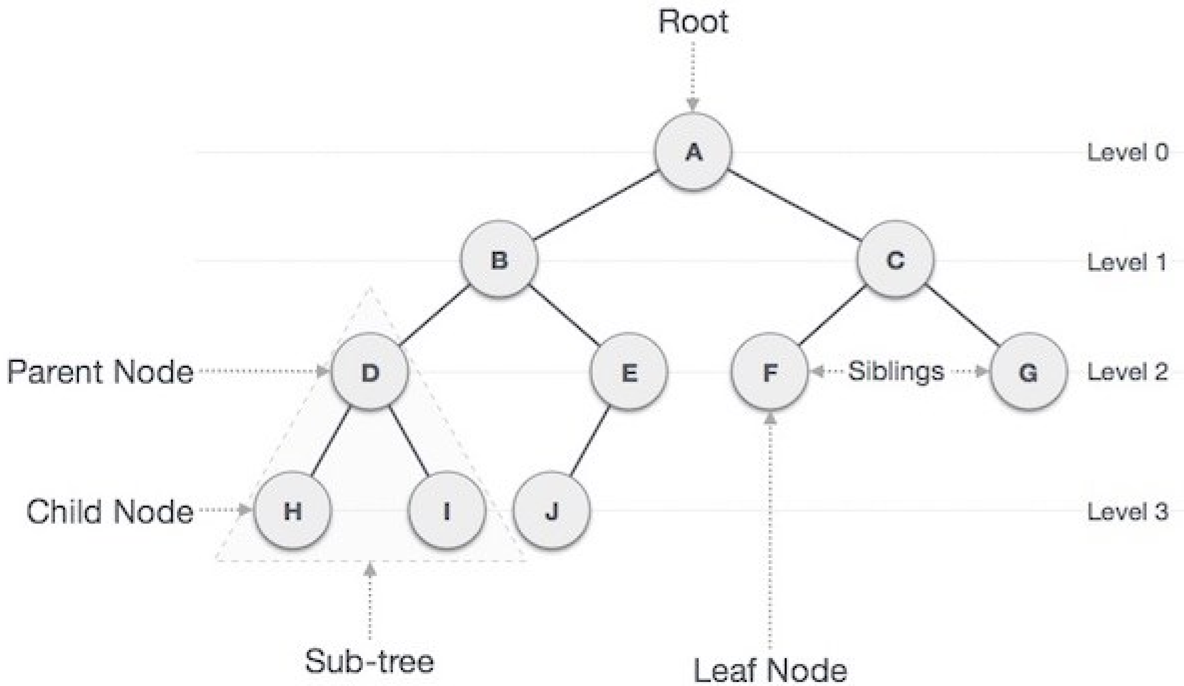
Uses in Computer Science:

- Expression trees
  (A – B) * (C + D)

```
              *
        /           \
      -              +
   /     \       /      \
  A       B    C         D
```

- Search trees
- Find data faster
- Index into large files or databases
- Game trees
- Keep possible next moves in tree (e.g. checkers, chest)
- Postponed obligations
- Encoding/Decoding messages
- Huffman codes
- Priority Queues
- Items have priorities
- Tree data structures allows quickest access to highest priority items (e.g. P.Q. to hold events for CPU)

Basic Trees Concepts and Terminology

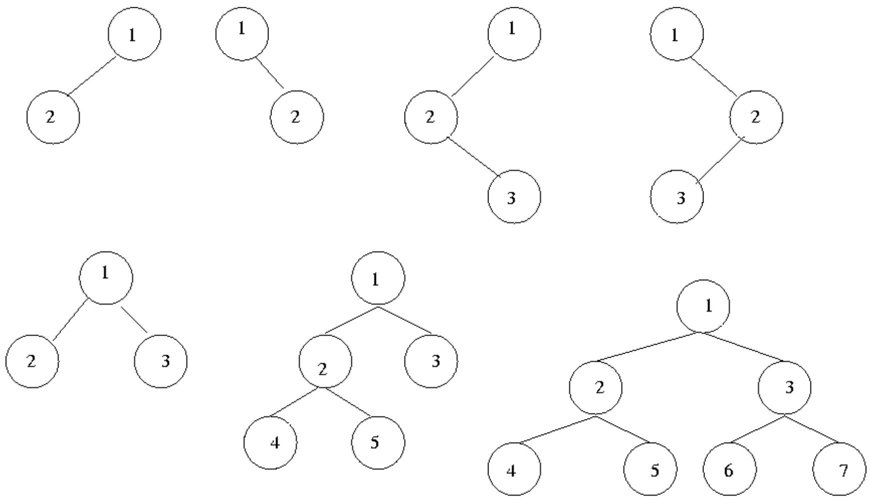


- A data structure made up of nodes and edges without having any cycle
- All trees are graphs but not all graphs are trees
- NOT a tree: anything with cycles (e.g. A-> A or B->C->E->D->B), undirected cycle, two non-connected parts (A->B and C->D->E)

Tree Anatomy:
- Root – the top node in a tree.
- Child – a node directly connected to another node when moving away from the root.
- Parent – the converse notion of a child.
- Siblings – a group of nodes with the same parent.
- Descendant – a node reachable by repeated proceeding from parent to child.
- Ancestor – a node reachable by repeated proceeding from child to parent.
- Leaf/External node – a node with no children.
- Branch/Internal node – a node with at least one child.
- Edge – the connection between one node and another.
- Path – a sequence of nodes and edges connecting a node with a descendant.
- Level – the level of a node is defined as: 1 + the number of edges between the node and the root.
- Height of node – the height of a node is the number of edges on the longest path between that node and a leaf.
- Height of tree – the height of a tree is the height of its root node.
- Depth – the depth of a node is the number of edges from the tree's root node to the node

Binary Tree

- Empty or has 1 node with 2 children, each a Binary Tree (recursive definition)



Complete Binary Tree

- Binary Tree that has leaves (on a single level or on 2 adjacent levels) such that leaves on the bottom most level are as far left as possible
- All levels are full (except possibly last)
- Are these Complete Binary trees?

Complete Binary Tree

Sequential:

- Use array
- Root – A[1]
- A[i]'s left child – A[2*i]
- A[i]'s right child – A(2*i+1]
- A[i]'s parent – A[i/2]

```
         X
       /   \
      B     C
    /  \   / \
   E    F D   G
```

0	1	2	3	4	5	6	7
	X	B	C	E	F	D	G

```
         4
       /   \
      7     9
    /  \   / \
   1    0 2   6
```

?

0	1	2	3	4	5	6	7

Problems?

- What happens when a tree is long and thin and right heavy?
- For a tree of height 3, you need an array A[15] – $2^{(3+1)}$ -1
- For a tree of height 8, you need an array A[511] – $2^{(8+1)}$ – 1

Linked:

- Use pointers

```
typedef struct NodeTag{
  itemType Item
  struct NodeTag *LLink
  struct NodeTag *RLink
} TreeNode
```

Tree Traversals

- Level Order: Level by level, left (Breadth First)
- Pre-order: <u>Root</u>, Left, Right (Depth First)
- In-order: Left, <u>Root</u>, Right (Depth First)
- Post-order: Left, Right, <u>Root</u> (Depth First)

```
           A
        /     \
      B         C
    /   \     /   \
   D     E   F     G
```

Level Order: A B C D E F G

How? Use Queues (iterative)

```
insert root on Q
while Q not Empty
  Remove item
  Visit it
  Insert item's left child on Q
  Insert item's right child on Q
```

Q

A								A
B	C							B
C	D	E						C
D	E	F	G					D, E, F, G

Pre-order: A B D E C F G
In-order: D B E A F C G
Post-order: D E B F G C A

Iterative Preorder algorithm: Use Stack

```
insert root on S
while S not Empty
  Pop item
  Visit it
  Push item's right child on S
  Push item's left child on S
```

Recursive Preorder algorithm:

```
PreOrder (T)
  if (!Empty(T))
    Visit (T)
    PreOrder(T->LChild)
    PreOrder(T->RChild)
```

Recursive in-order and post-order?

Binary Search Trees (BTSs)

- Binary Tree such that each node X
- (keys in X's left subtree) < (key in X) < (keys in X's right subtree)

```
          30
        /     \
      15       50
     /  \         \
    10  20         70
       /             \
      17              80
```

Search: <- do it
Insert: always a leaf <-
Delete:

- If leaf, easy
- If 1 child, promote child
- If 2 children, promote a descendant
- Copy largest descendent in left subtree OR smallest descendent in right subtree
- Delete "copied" from old subtree

Delete 80?
Delete 50 (promote70)
Delete 30 (promote 20 or 50)

Optimally Balanced Trees?

Complete BST Search time?
Worse case: O(logn)
Why?
Height is proportional to logn
n is number of nodes, on a complete binary tree $n = 2^{(h+1)} – 1$ where h is the height of the tree
therefore $h = \log_2(n+1) – 1$
and the math says $\log(n+1) < \log n + 1$ so $h < \log n$
any Complete BST is similar h = floor(logn)

Degenerate BST?

Search time and insert time is O (n) at worst

Height is proportional to n

- Keep trees optimally balanced for quickest search
- Problem: algorithm to re-blance tree after insert is O(n)

AVL Trees

- Adelson, Velskii, Landis – 1962
- BSTs such that

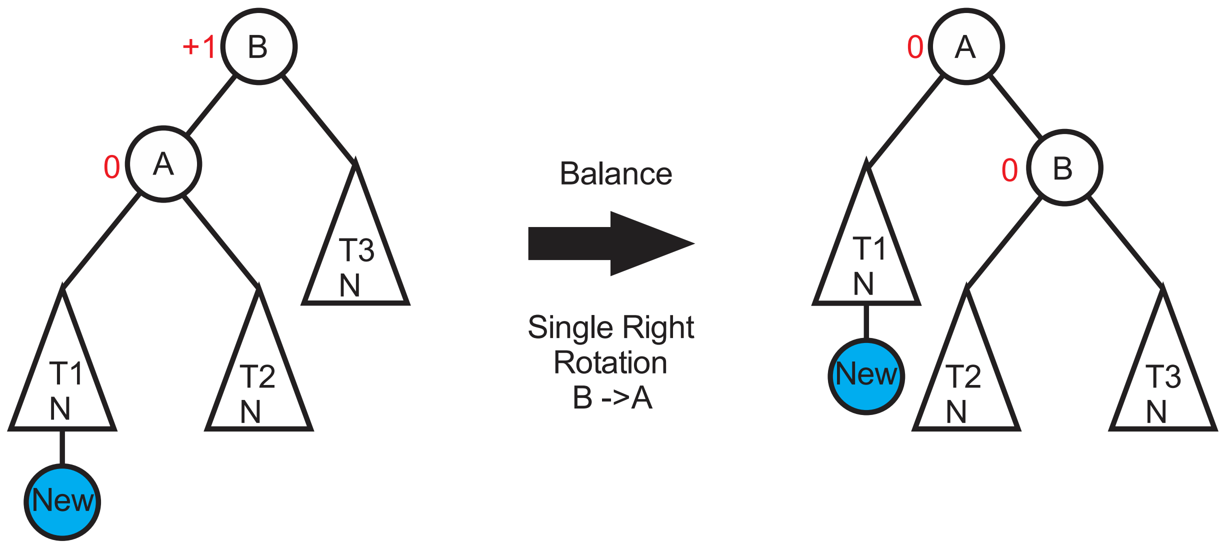
$$for\ every\ node, (height\ of\ leftsubtree) - (height\ of\ right\ subtree \begin{cases} -1 \\ 0 \\ 1 \end{cases}$$

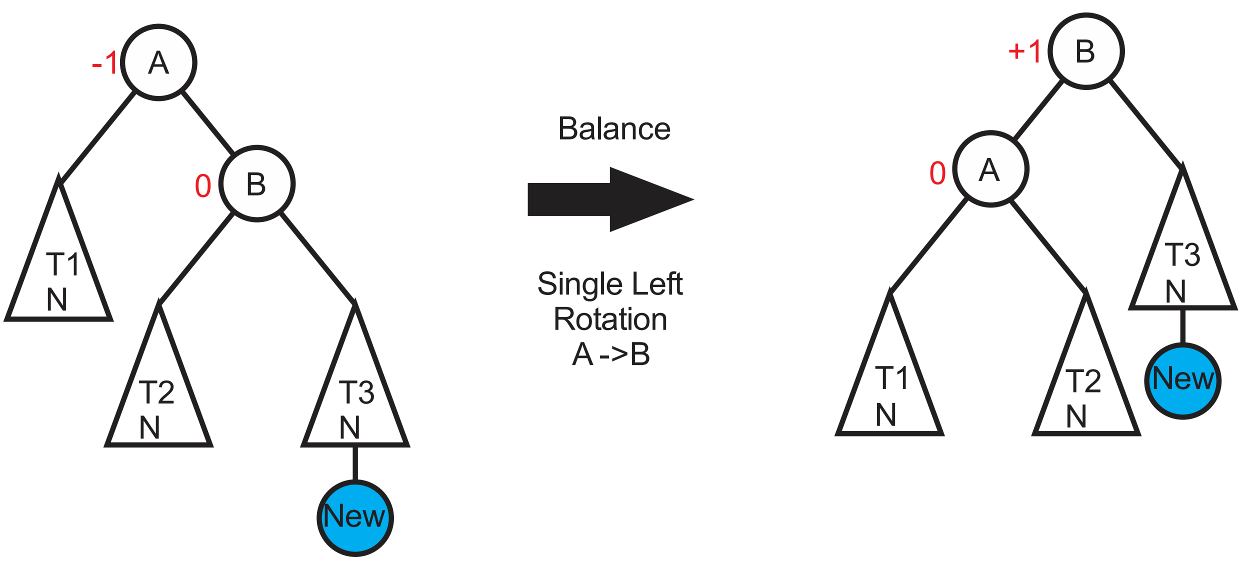
- almost balanced trees
- search/insert/delete time of O(logn)

Inserting

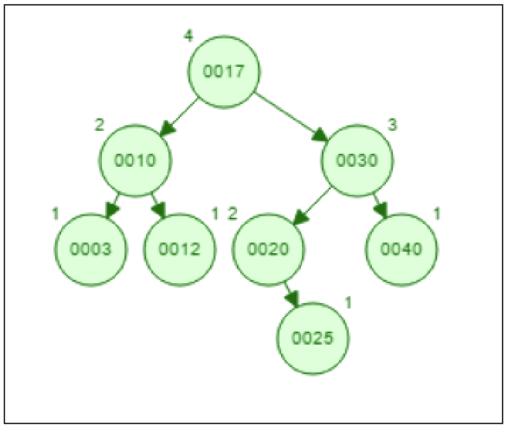
- as usual for BST ... then … may need to rebalance
- 4 ways to insert and cause imbalance
- re-balance preserves in-order traversal of tree
- only re-balance smallest subtree possible

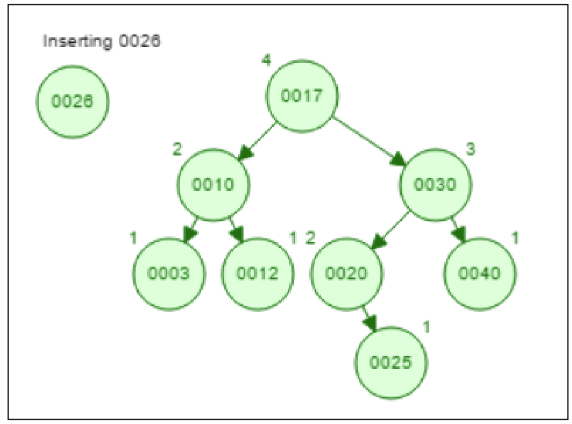
AVL Single Rotation

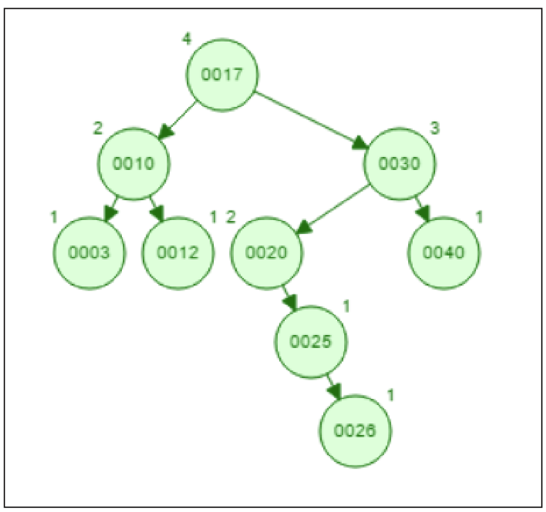


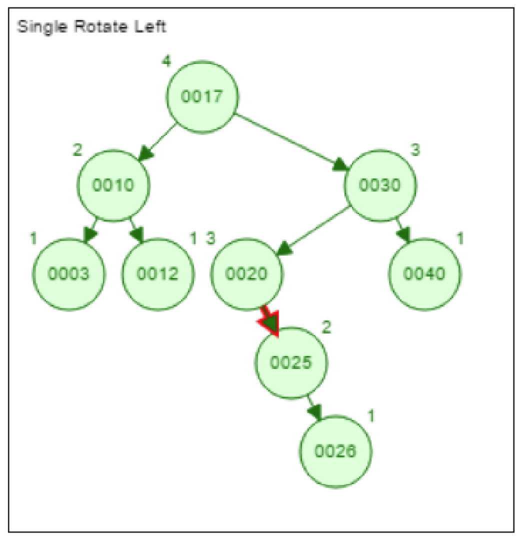


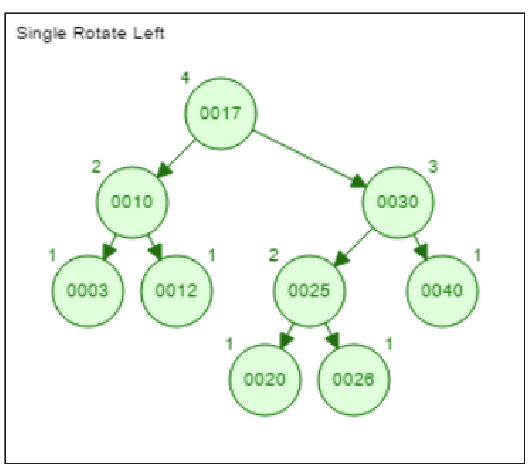
Single Rotation Example



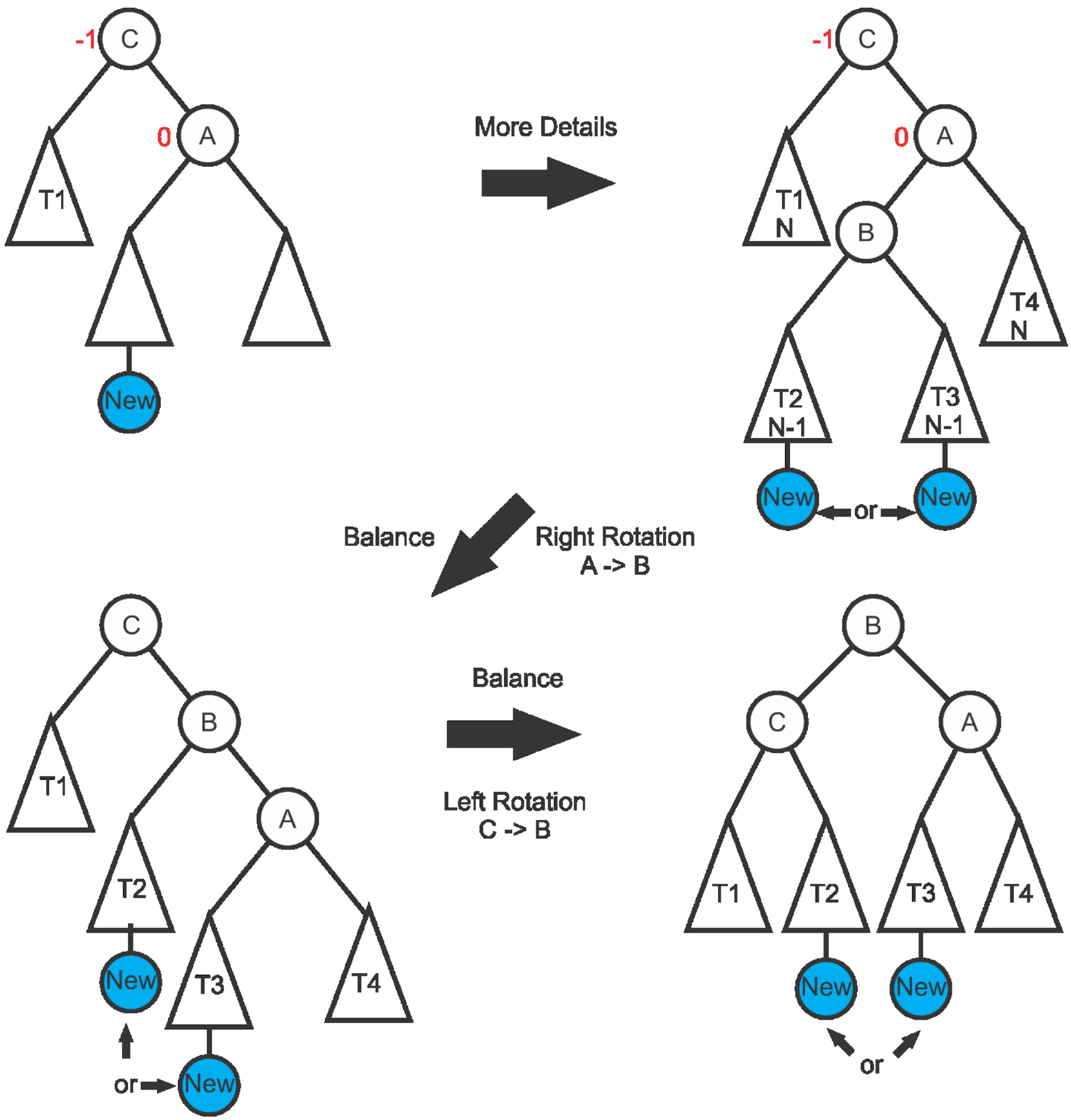




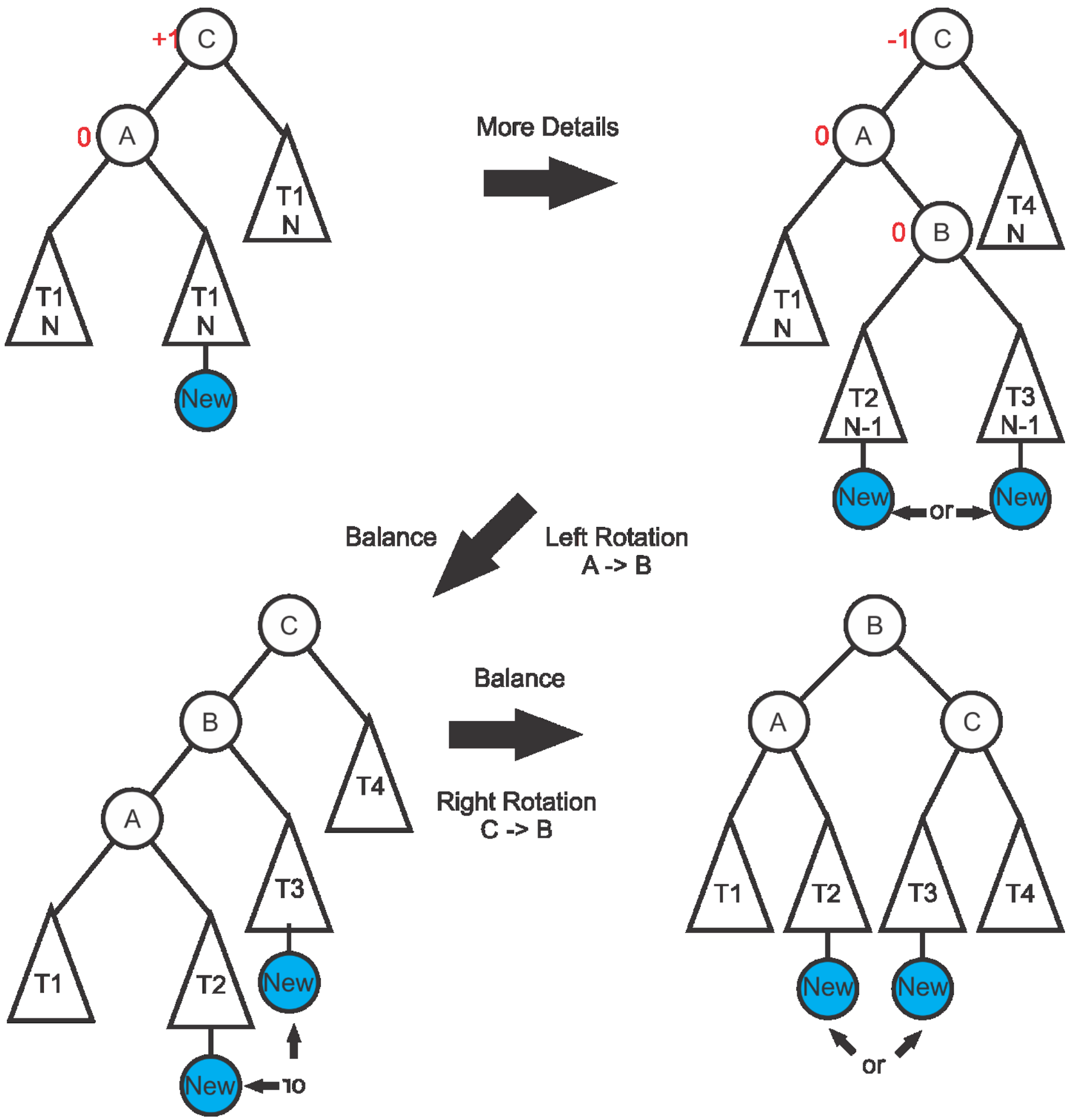




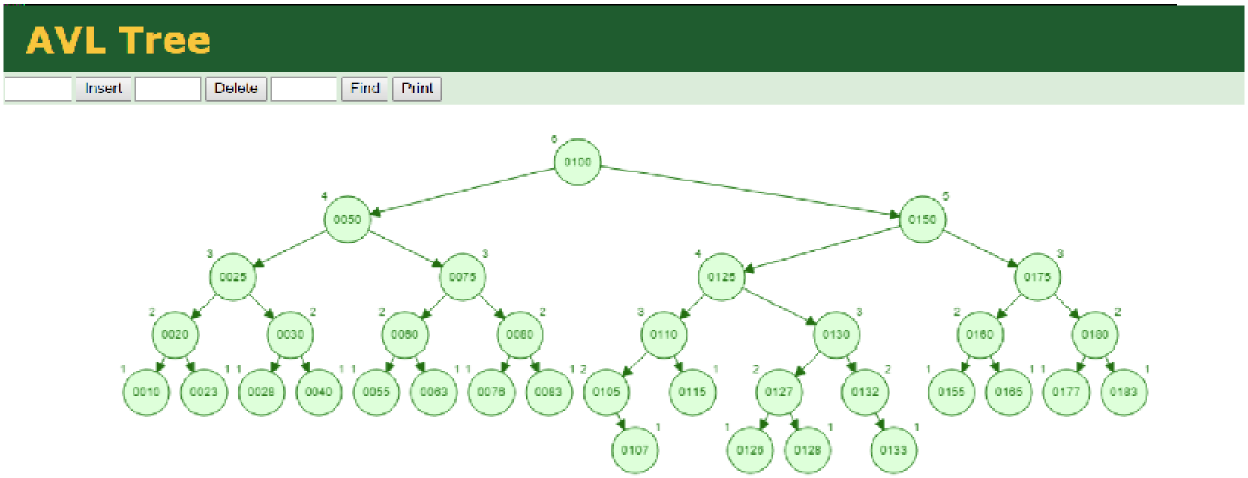
AVL Double Left Rotation

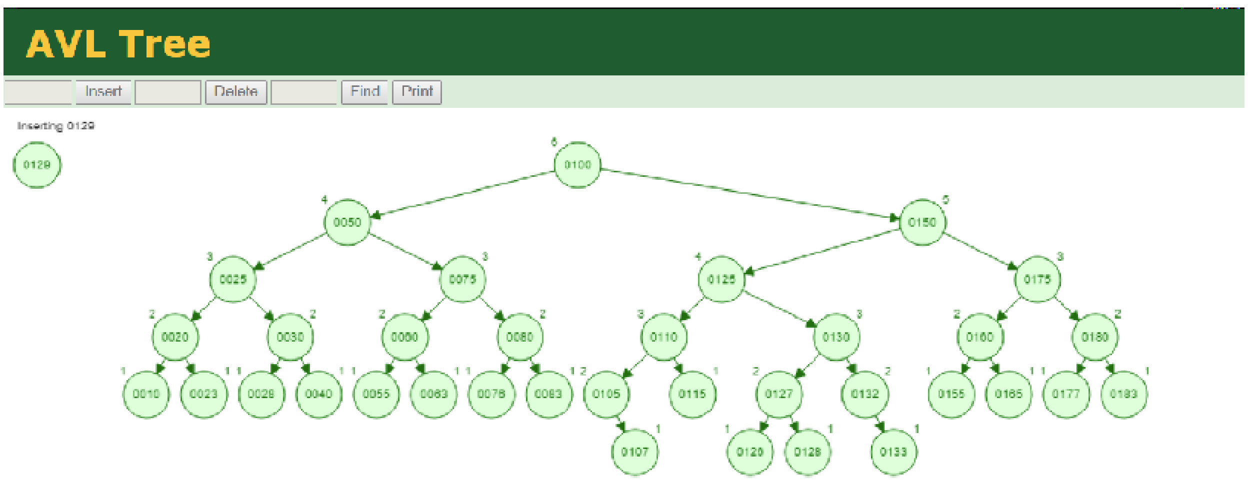


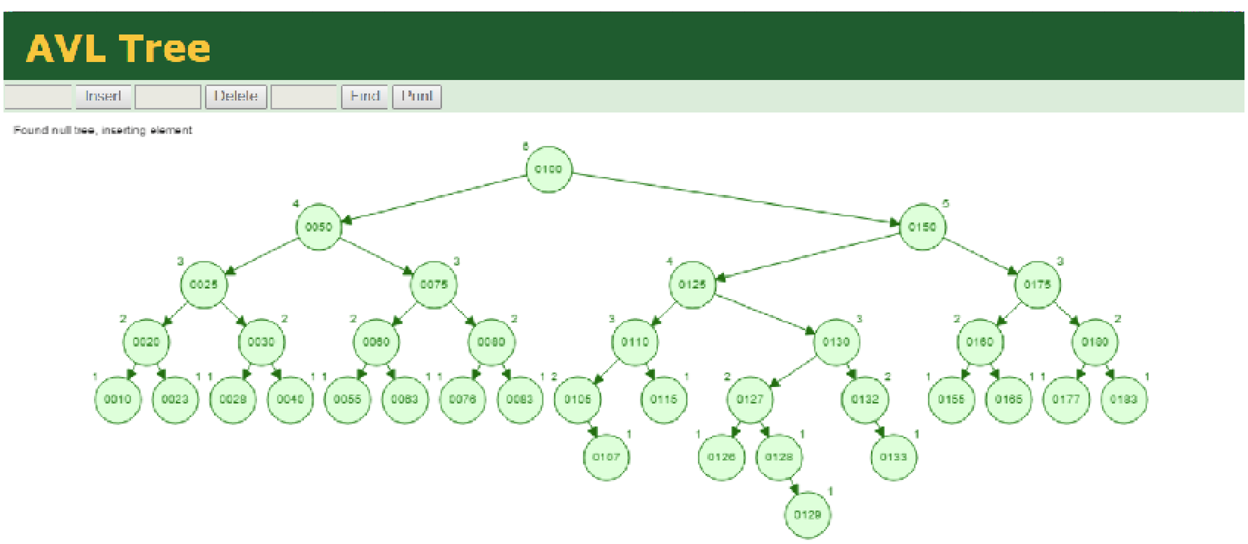
AVL Double Right Rotation

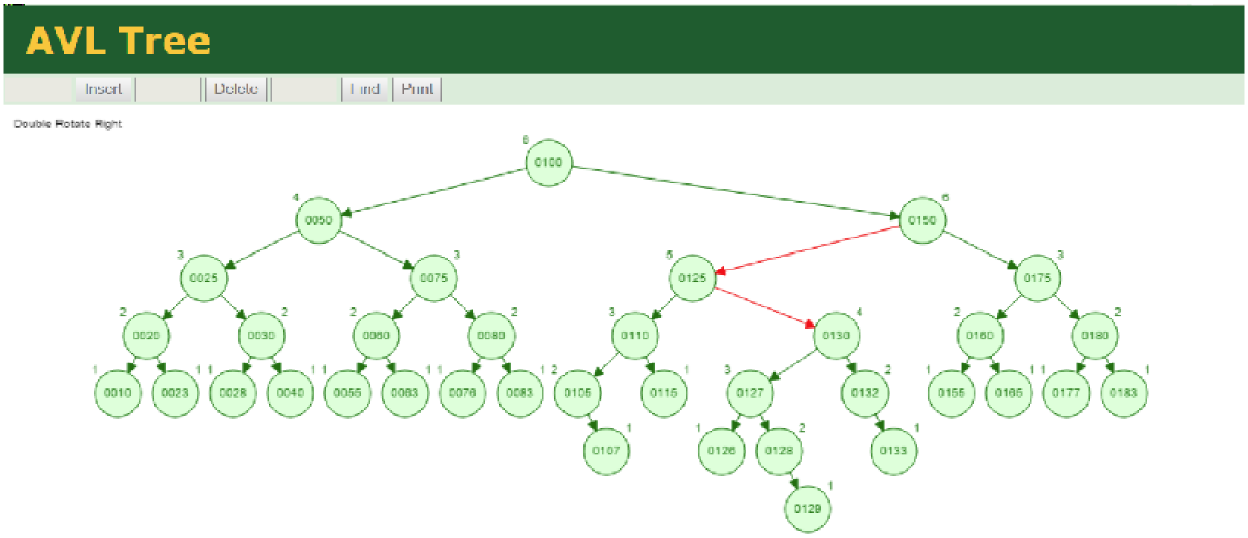


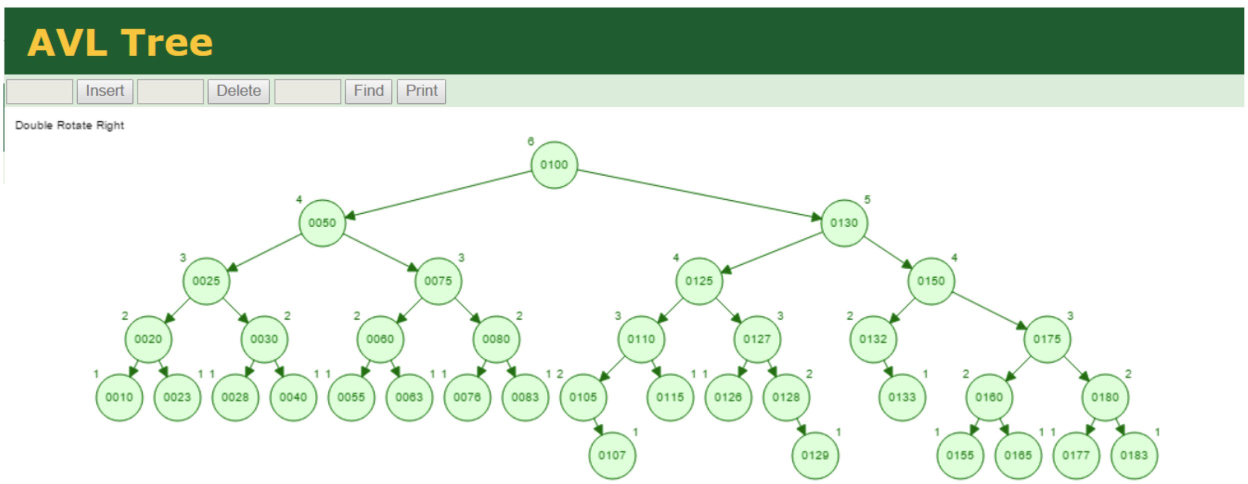
Double Rotation Examples







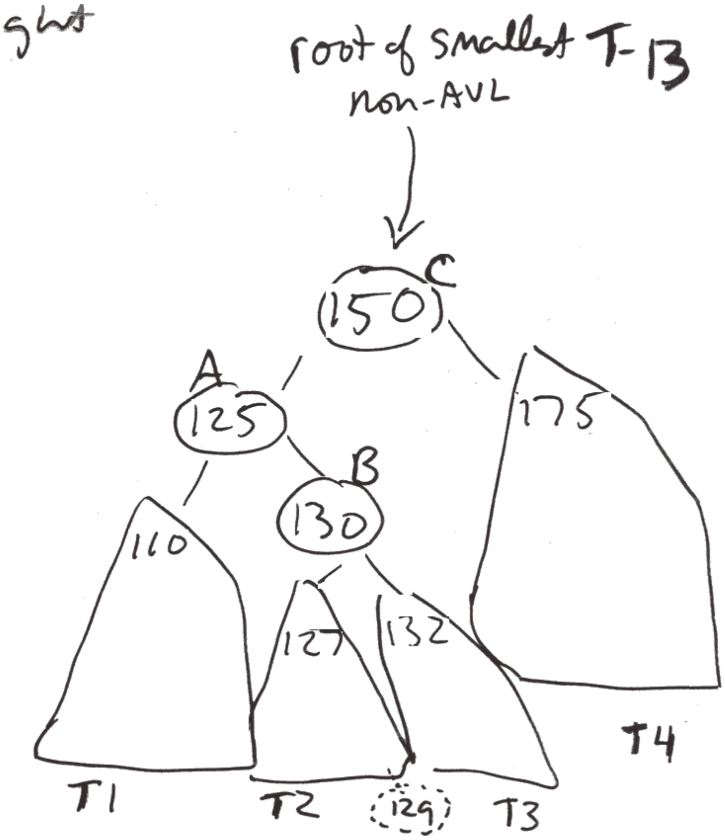




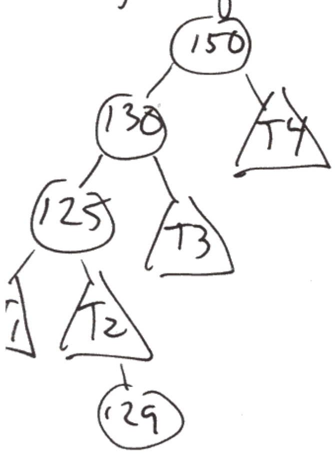
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Break down

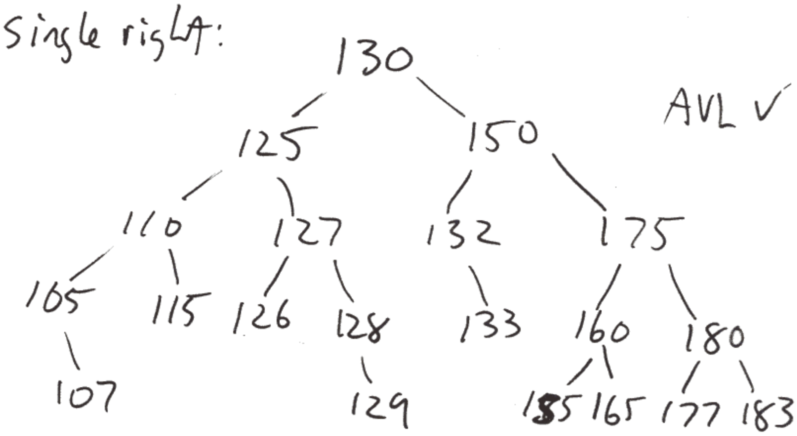
Matches Double Right
(left, right)



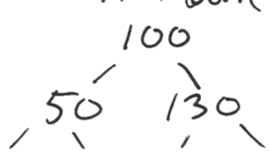
single left:



single right:

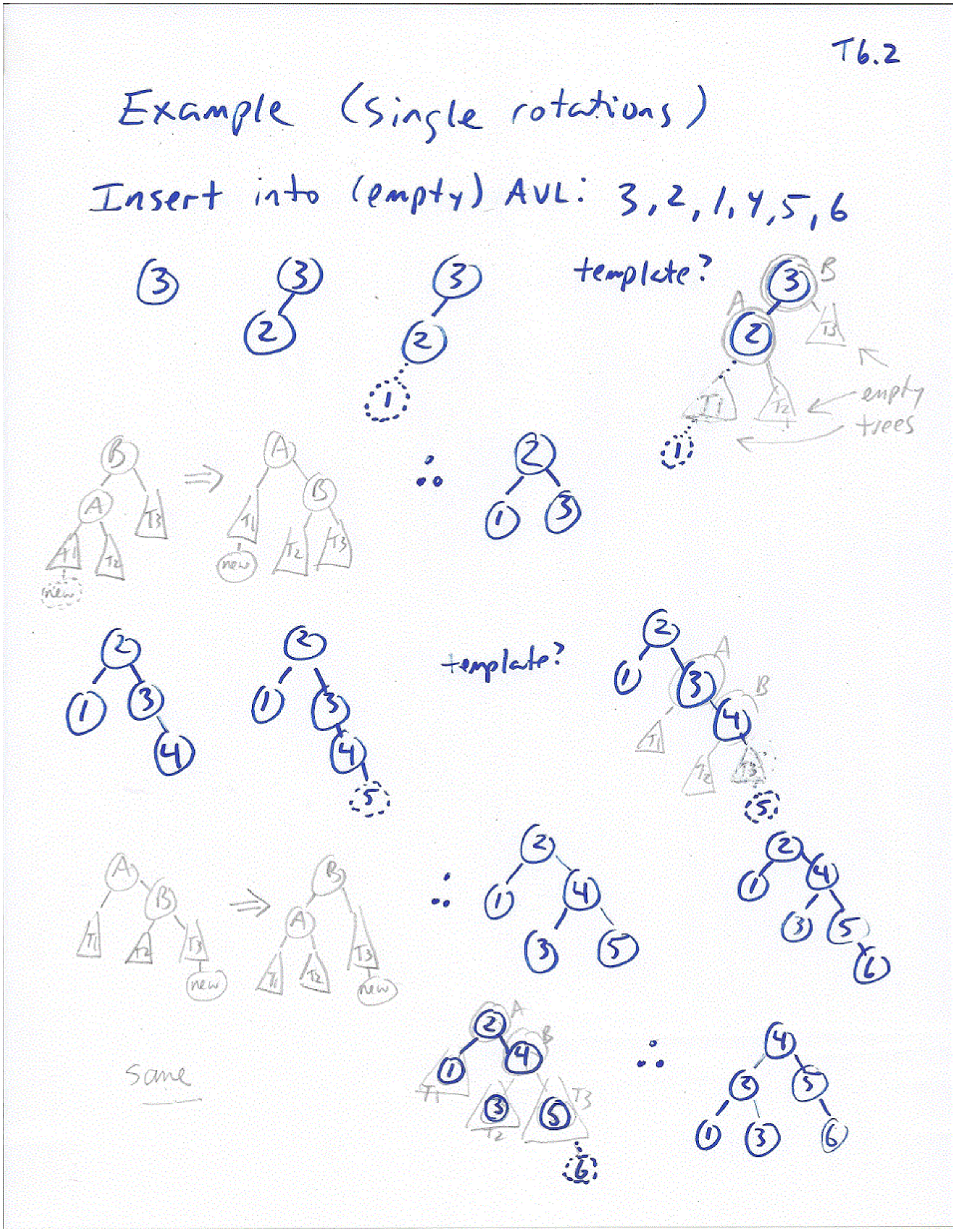


Now put it back in main tree

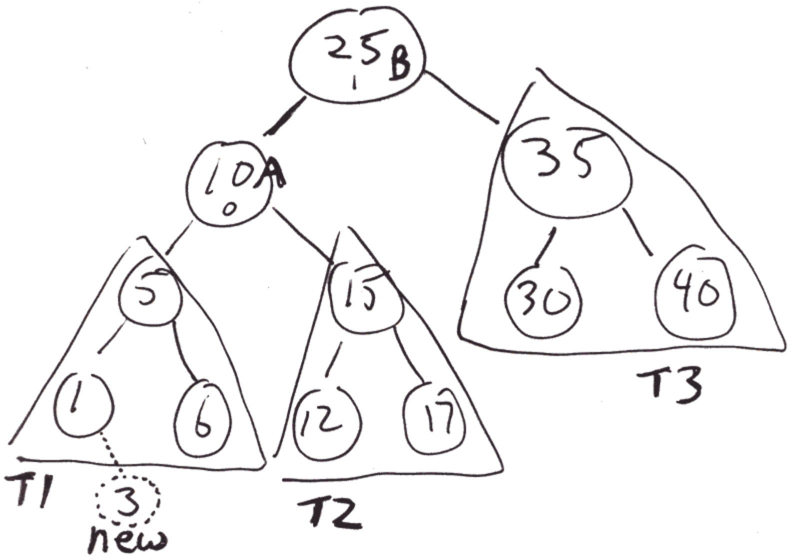


& whole tree AVL.

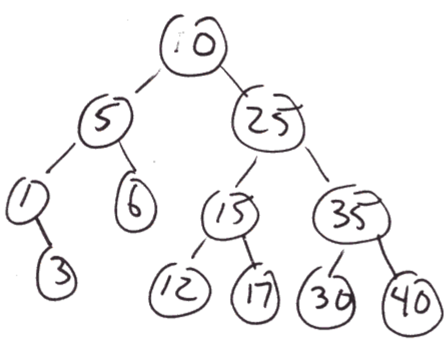
More examples



Matches single right rotation

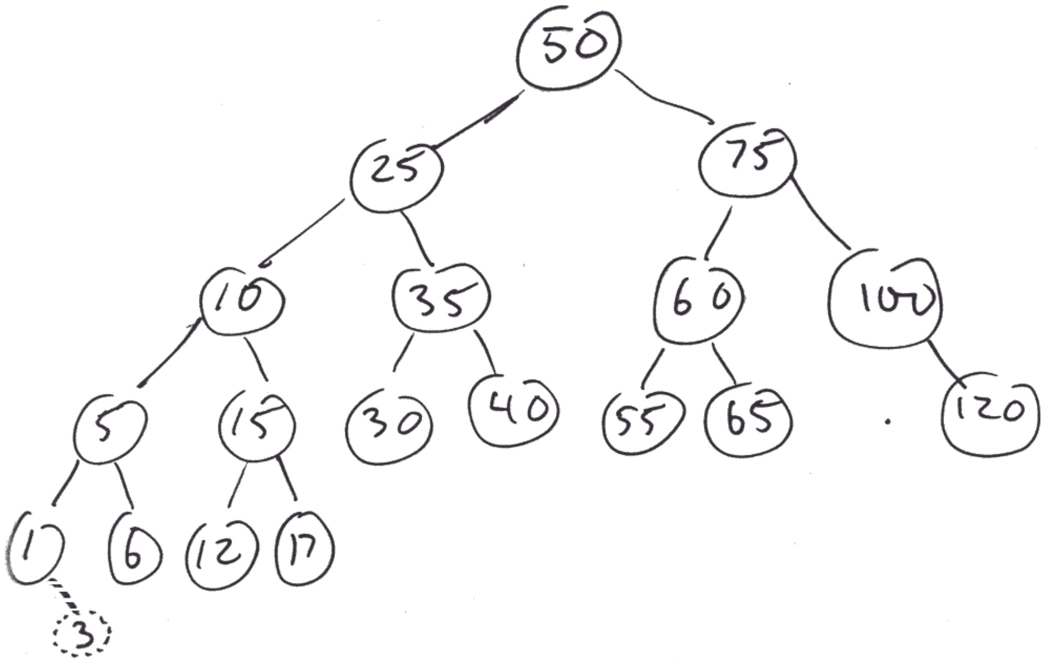


⇒ Balance



Subtree now AVL ⇒ whole tree AVL

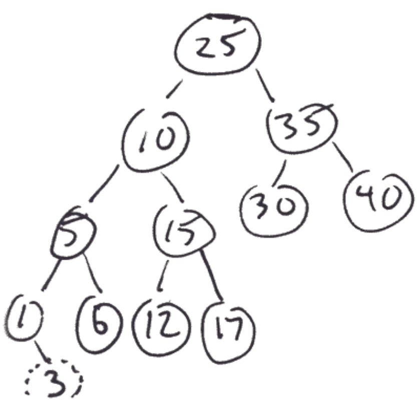
Remember to put subtree back in tree!



- AVL? yes (BST & property)

- insert 80 - still AVL

- insert 3 (goes to right of 1)
  - AVL? no.
  - smallest non-AVL subtree?
    (rooted at 25) rotate as before



How do you find smallest non-AVL subtree?

- Start at inserted node
- Calculate its balance factor (B.F.)
- Work up ancestors toward root
  - for each node, recalculate B.F
  - if find |B.F.| >= 2 then it's the root of smallest non -AVL subtree

NOTE: the insert algorithm most change tehse B.F.s. If find |B.F.| >= 2 stop and rotate, other wise go to root, re-calculating

Deleting:

- The usual delete by copy (normal BST delete)
  - copy node "X" into node we're deleting
  - delete node X (leaf or 1-child)
- Update B.F.s from X's parent up to root for each node with |B.F.| >= 2: rotate to restore balance

Delete

- Keeps going until gets to root, rebalancing if necessary (unlike insert, which stops after first re-balance)

AVL Trees – insert/delete/search are all O(logn)