CSCI 104
Splay Trees

Mark Redekopp

Sources / Reading

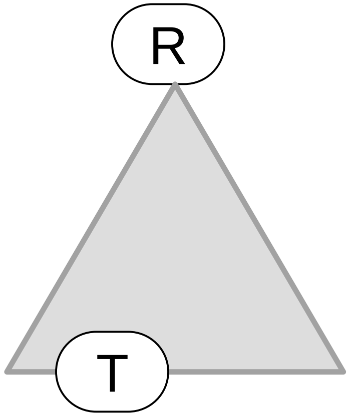
- Material for these slides was derived from the following sources
  - https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf
  - http://digital.cs.usu.edu/~allan/DS/Notes/Ch22.pdf
  - http://www.cs.umd.edu/~meesh/420/Notes/MountNotes/lecture10-splay.pdf
- Nice Visualization Tool
  - https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Splay Tree Intro

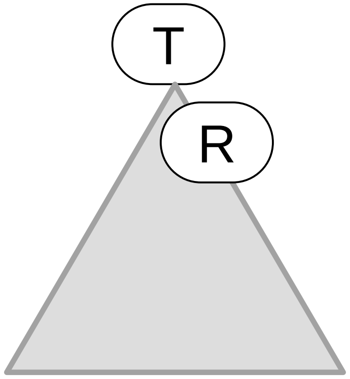
- Another map/set implementation (storing keys or key/value pairs)
  - Insert, Remove, Find
- Recall…To do m inserts/finds/removes on an AVLTree w/ n elements would cost?
  - O(m*log(n))
- Splay trees have a worst case find, insert, delete time of…
  - O(n)
- However, they guarantee that if you do m operations on a splay tree with n elements that the total time is
  - O(m*log(n)) **[i.e. amortized time is O(log(n))]**
- They have a further benefit that recently accessed elements will be near the top of the tree
  - In fact, the most recently accessed item is always at the top of the tree

Splay Operation

- Splay means "spread"
- As you search for an item or after you insert an item we will perform a series of splay operations
- These operations will cause the desired node to always end up at the top of the tree
  - A desirable side-effect is that accessing a key multiple times within a short time window will yield fast searches because it will be near the top
  - See next slide on principle of locality



If we search for or insert T…



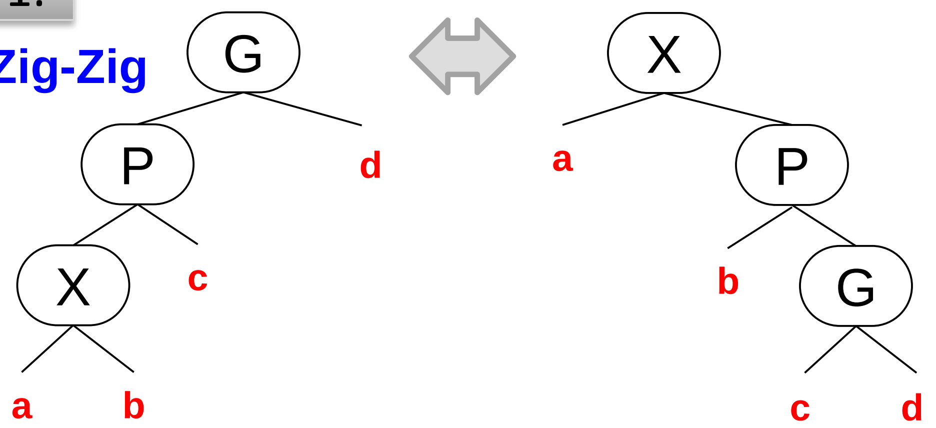
…T will end up as the root node with the old root in the top level or two

Principle of Locality

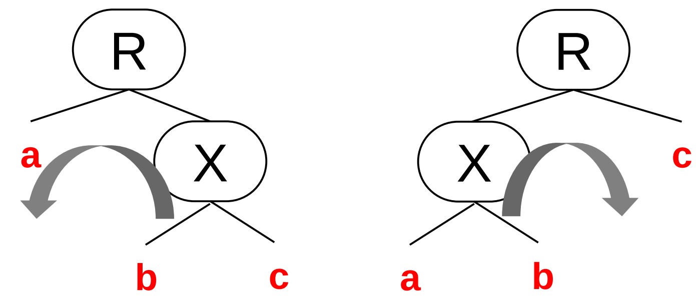
- 2 dimensions of this principle: space & time
- Spatial Locality – Future accesses will likely cluster near current accesses
  - Instructions and data arrays are sequential (they are all one after the next)
- Temporal Locality – Future accesses will likely be to recently accessed items
  - Same code and data are repeatedly accessed (loops, subroutines, if(x > y) x++;
  - 90/10 rule: Analysis shows that usually 10% of the written instructions account for 90% of the executed instructions
- Splay trees help exploit temporal locality by guaranteeing recently accessed items near the top of the tree

Splay Cases

1. **Zig-Zig**



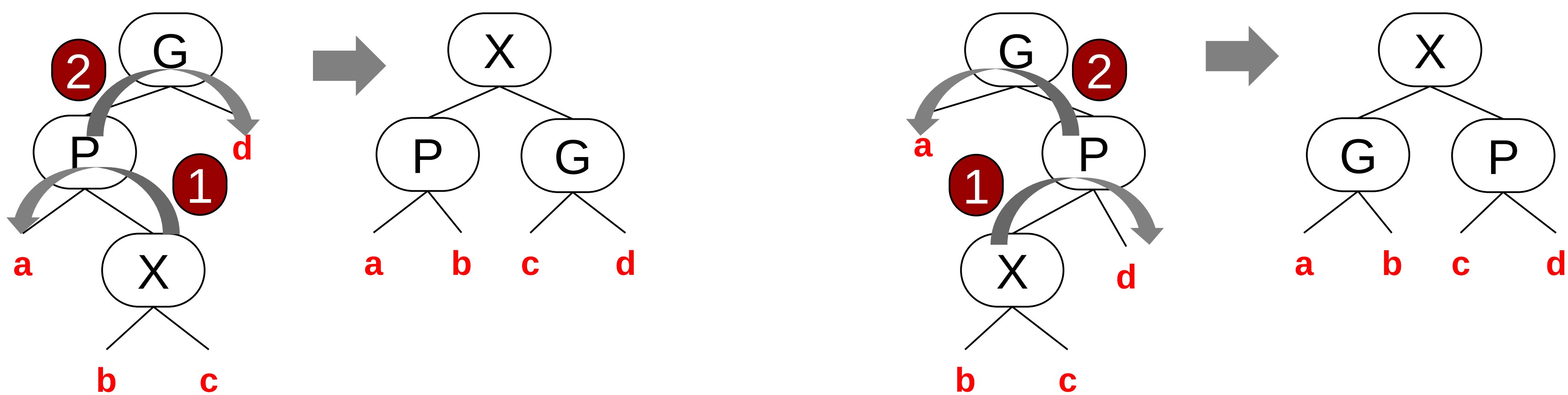
3. **Root/Zig Case (Single Rotation)**



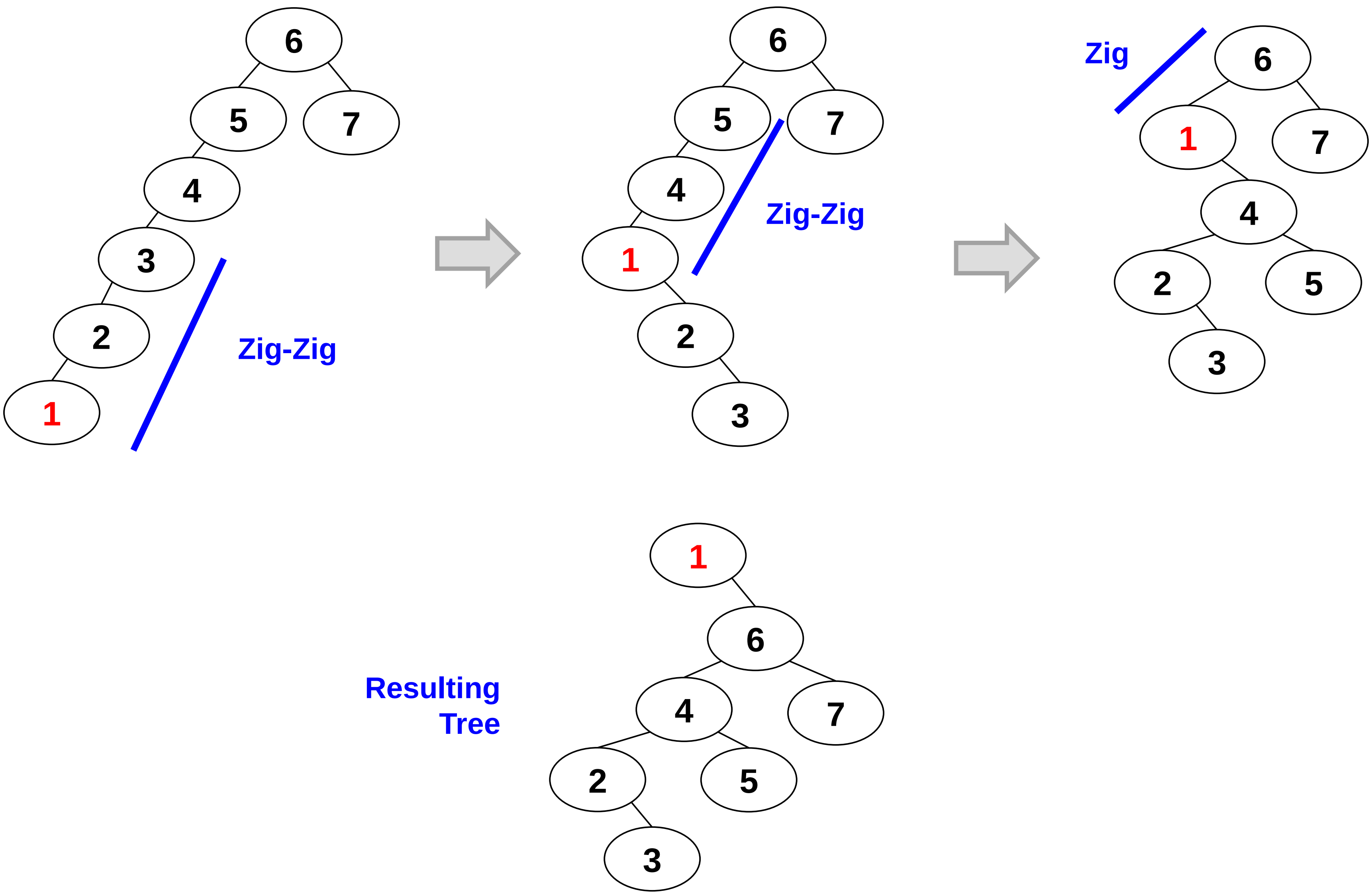
Left rotate of X,R

Right rotate of X,R

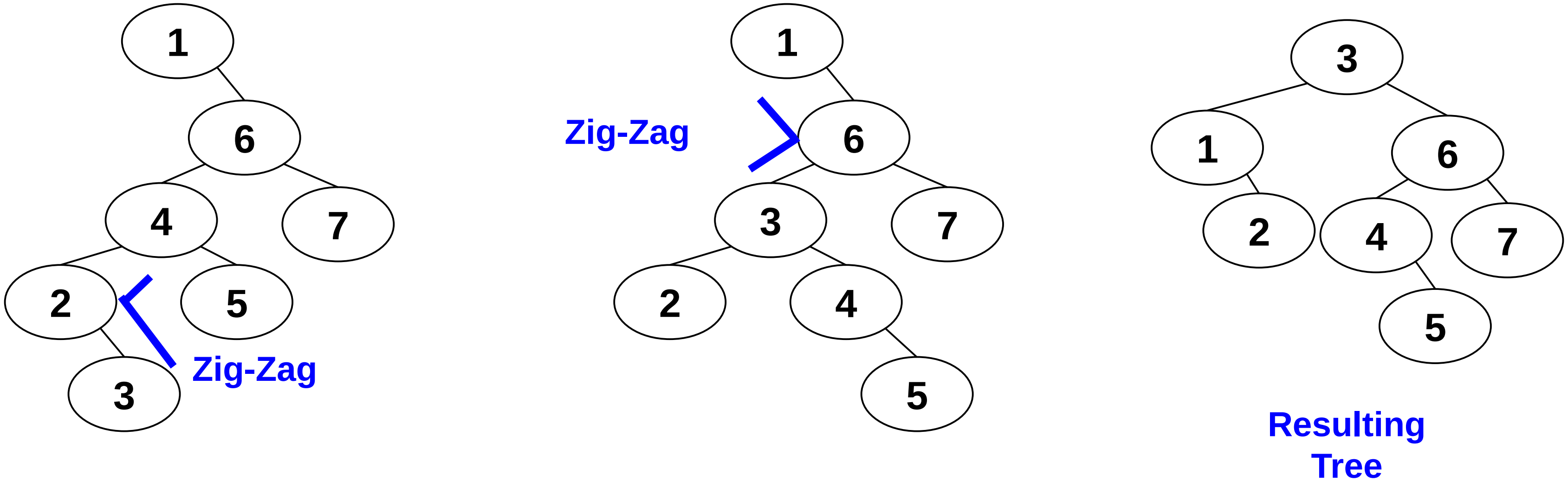
2. **Zig-Zag**



Find(1)



Find(3)

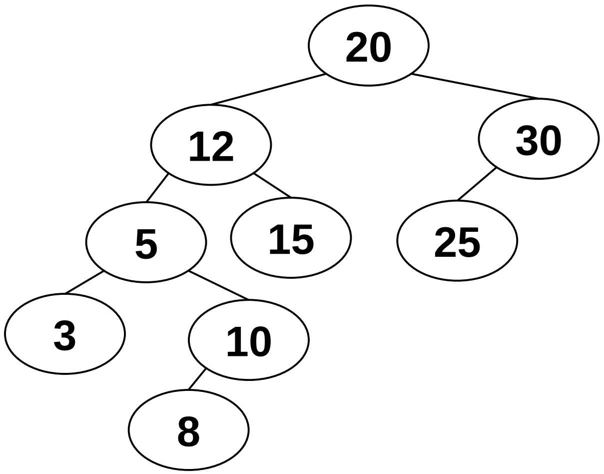


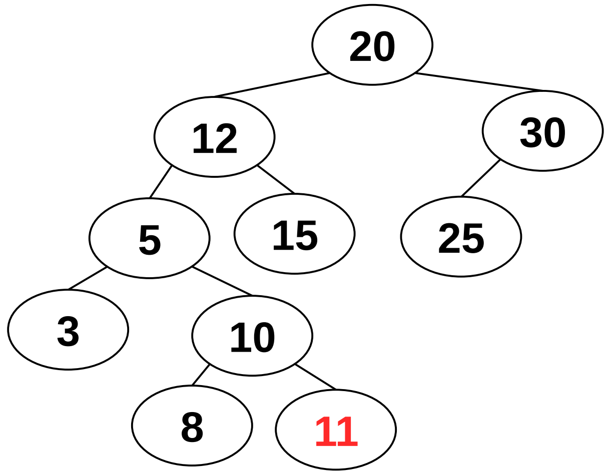
- Notice the tree is starting to look at lot more balanced

Worst Case

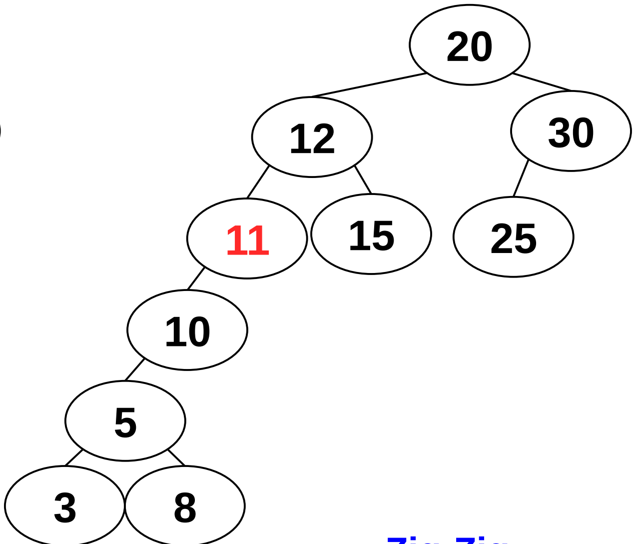
- Suppose you want to make the amortized time (averaged time over multiple calls to find/insert/remove) look bad, you might try to always access the ________________ node in the tree
  - Deepest
- But splay trees have a property that as we keep accessing deep nodes the tree starts to balance and thus access to deep nodes start by costing O(n) but soon start costing O(log n)

Insert(11)

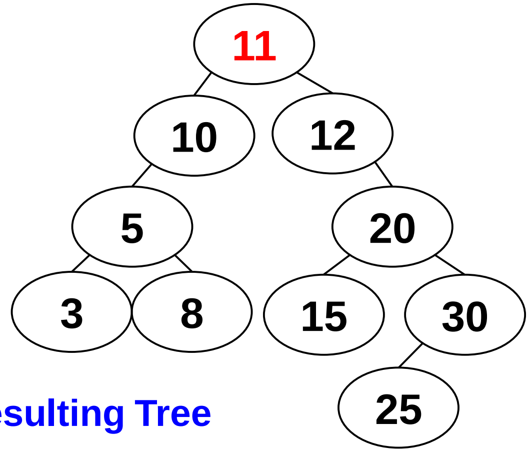




Zig-Zig

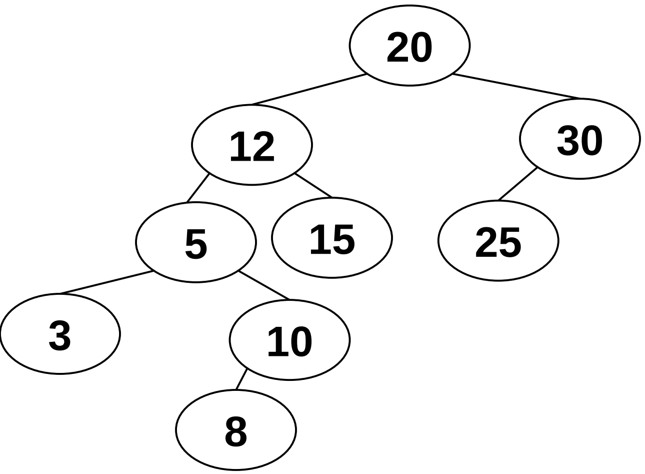


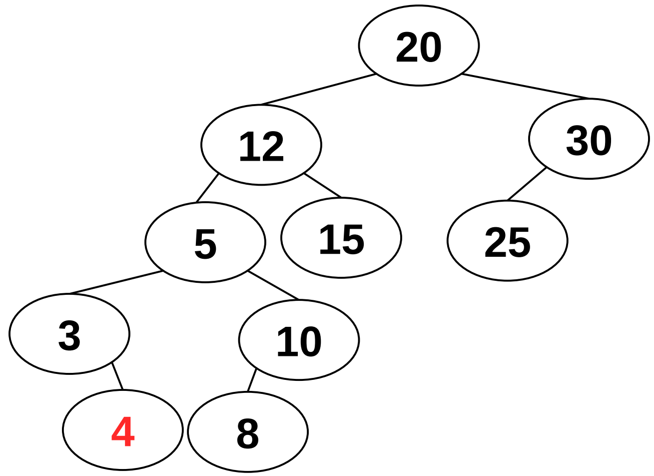
Zig-Zig



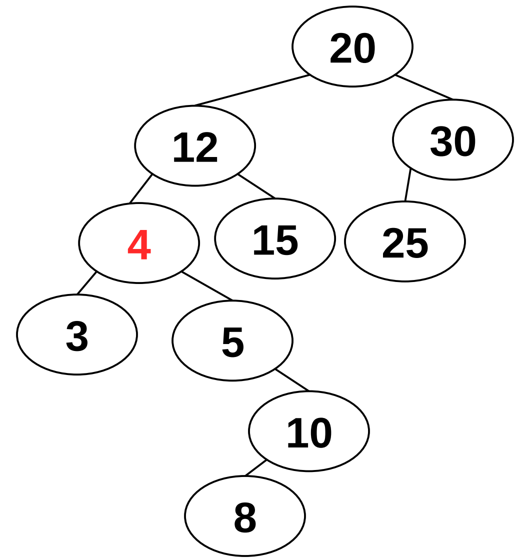
Resulting Tree

Insert(4)

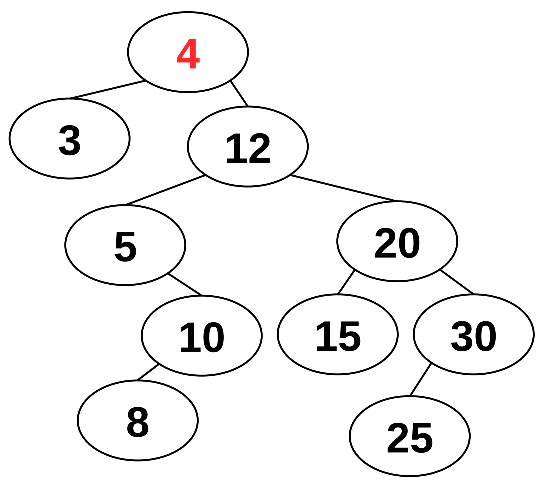




Zig-Zag



Zig-Zig



Resulting Tree

Activity

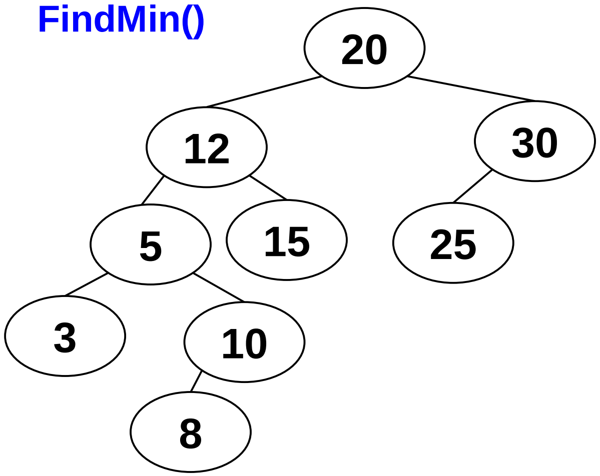
- Go to
  - https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
  - Try to be an adversary by inserting and finding elements that would cause O(n) each time

Splay Tree Supported Operations

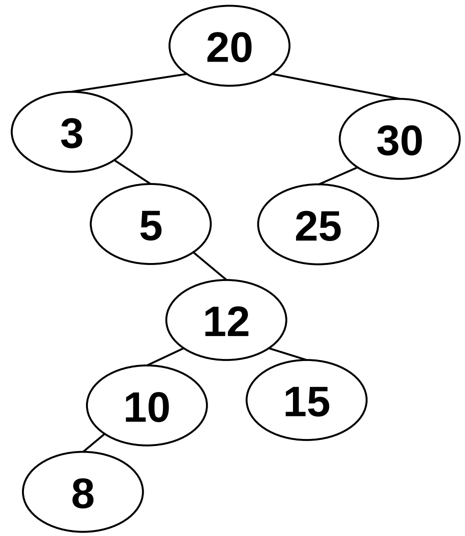
- Insert(x)
  - Normal BST insert, then splay x
- Find(x)
  - Attempt normal BST find(x) and splay last node visited
    - If x is in the tree, then we splay x
    - If x is not in the tree we splay the leaf node where our search ended
- FindMin(), FindMax()
  - Walk to far left or right of tree, return that node's value and then splay that node
- DeleteMin(), DeleteMax()
  - Perform FindMin(), FindMax() [which splays the min/max to the root] then delete that node and set root to be the non-NULL child of the min/max
- Remove(x)
  - Find(x) splaying it to the top, then overwrite its value with is successor/predecessor, deleting the successor/predecessor node

FindMin() / DeleteMin()

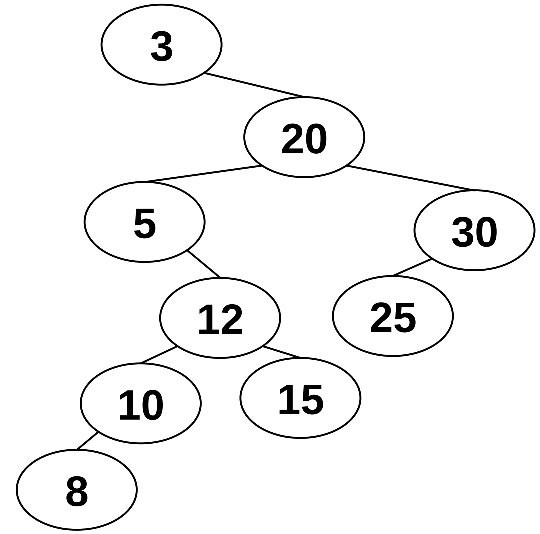
FindMin()



Zig-Zig

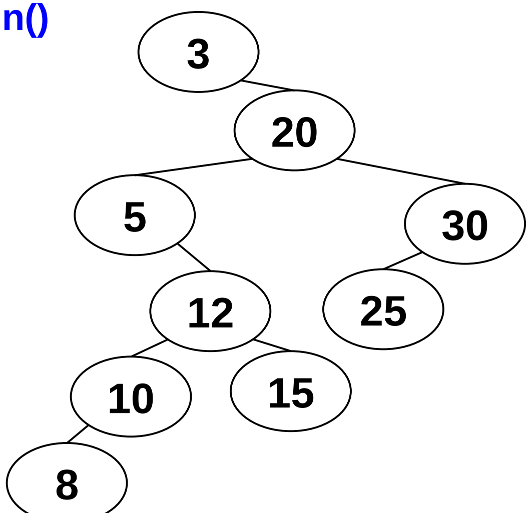


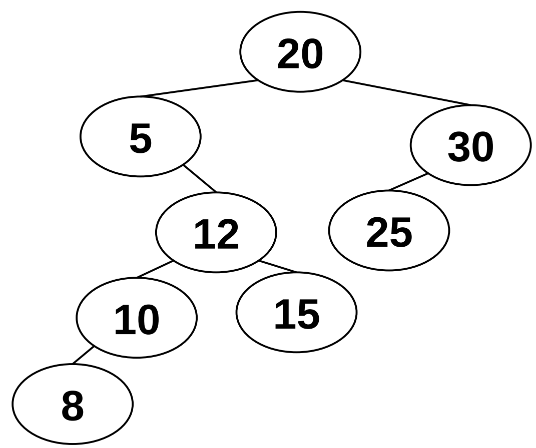
Zig



Resulting Tree

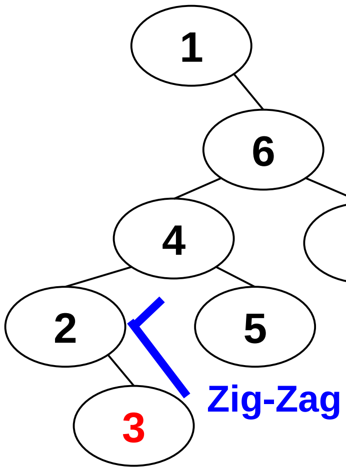
DeleteMin()

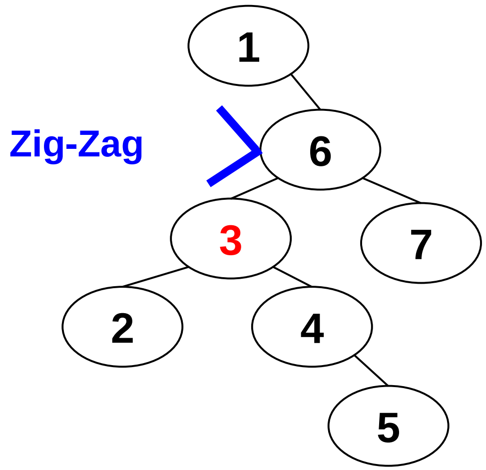


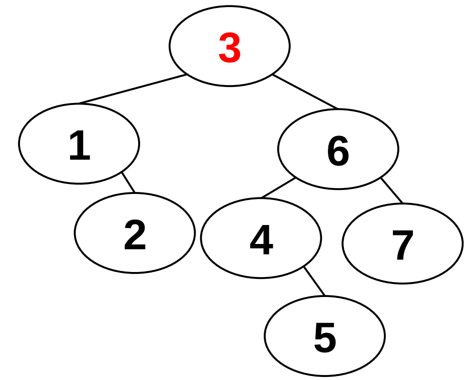


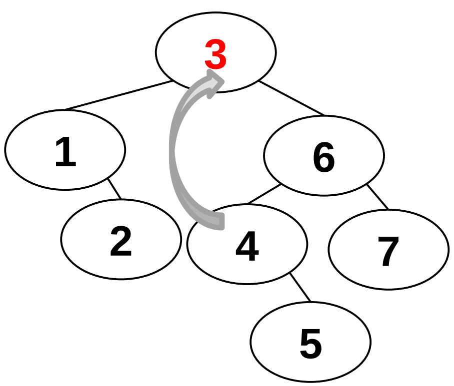
Resulting Tree

Remove(3)

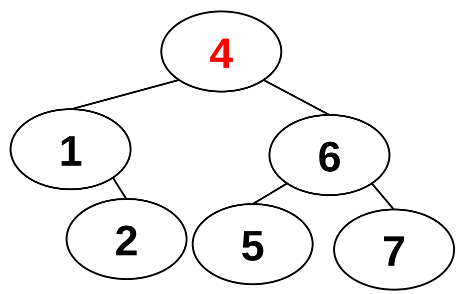








Copy successor or predecessor to root

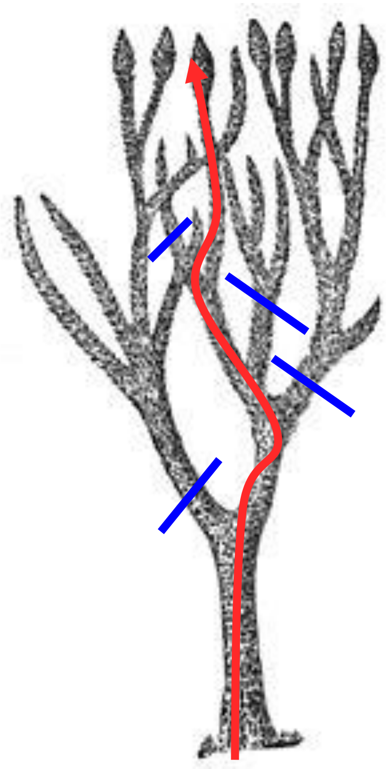


Delete successor (Remove node or reattach single child)

Resulting Tree

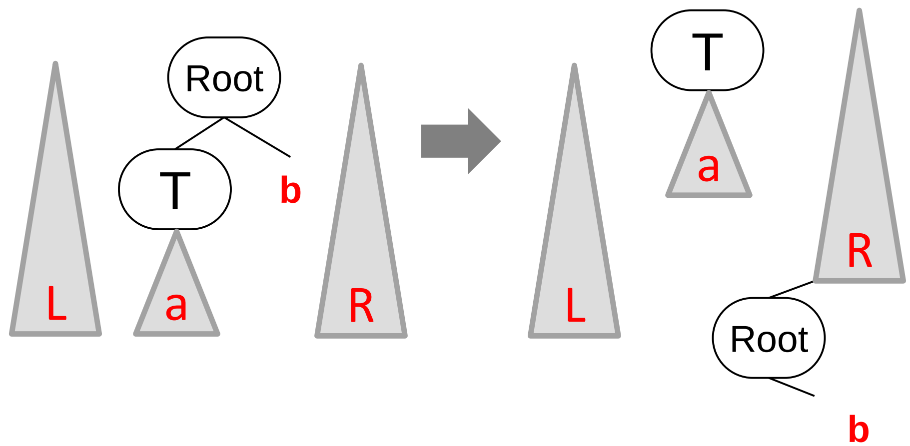
Top Down Splaying

- Rather than walking down the tree to first find the value then splaying back up, we can splay on the way down
- We will be "pruning" the big tree into two smaller trees as we walk, cutting off the unused pathways

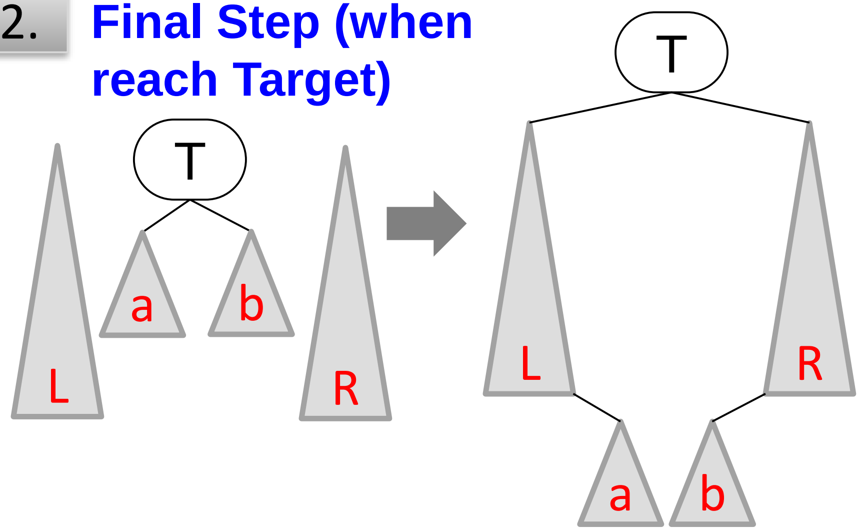


Top-Down Splaying

1. **Zig (If Target is in 2nd level)**

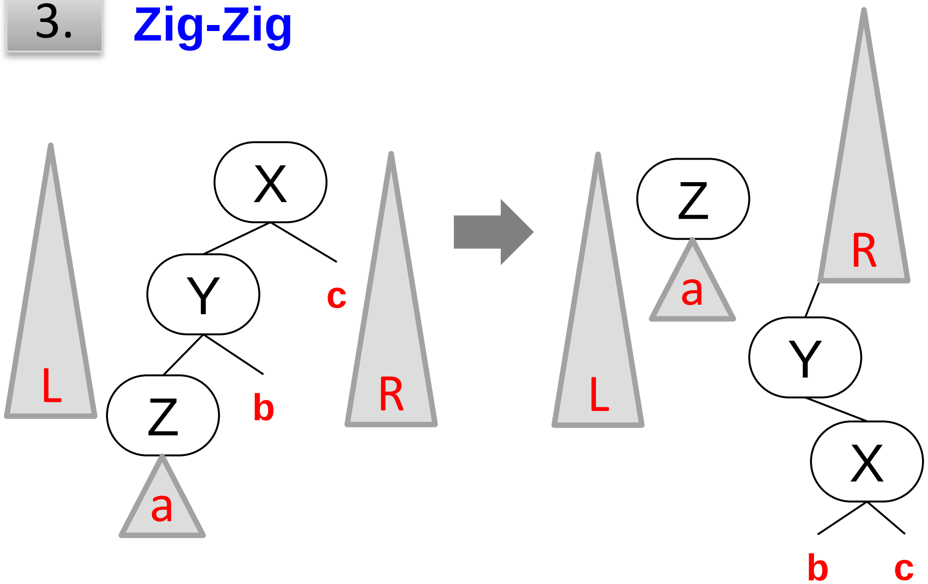


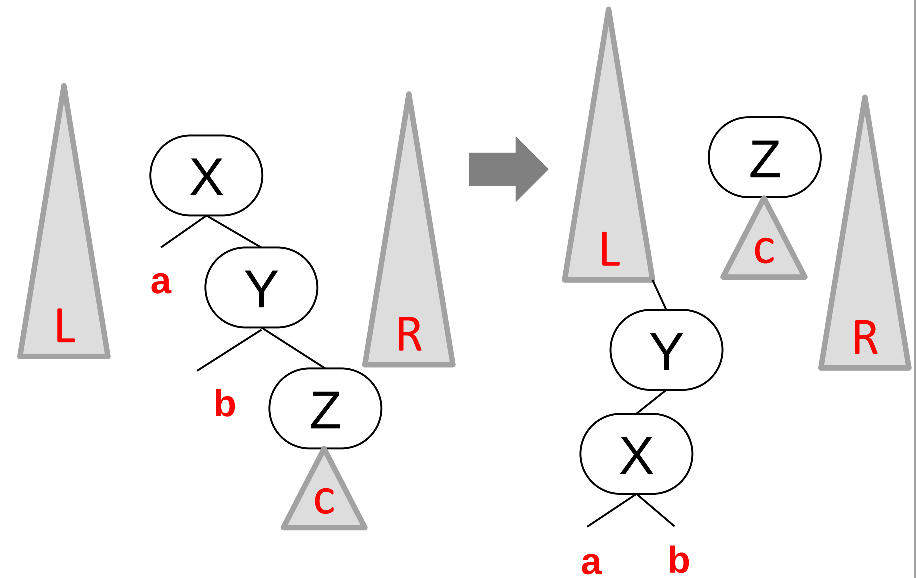
2. **Final Step (when reach Target)**



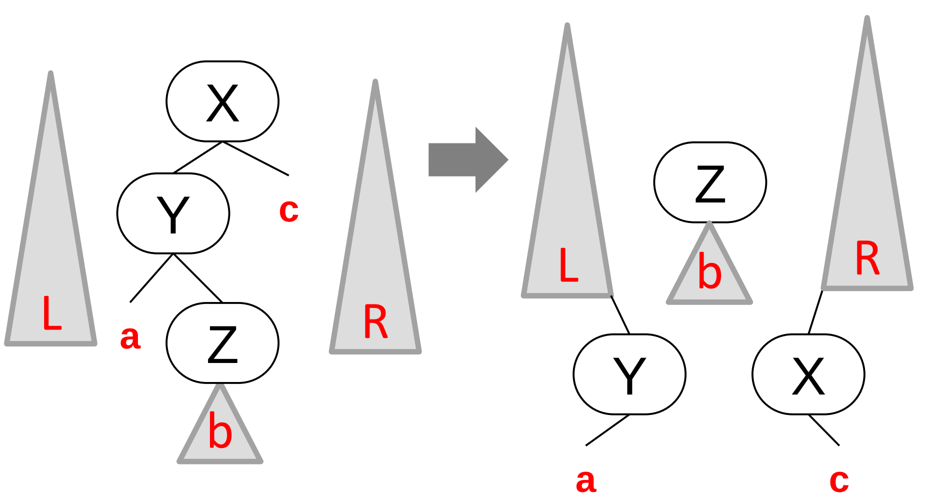
Top-Down Splaying

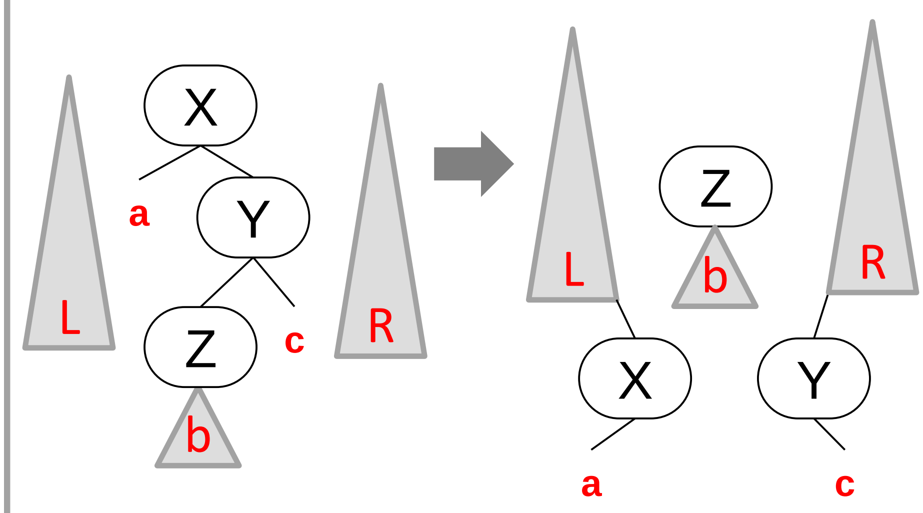
3. Zig-Zig



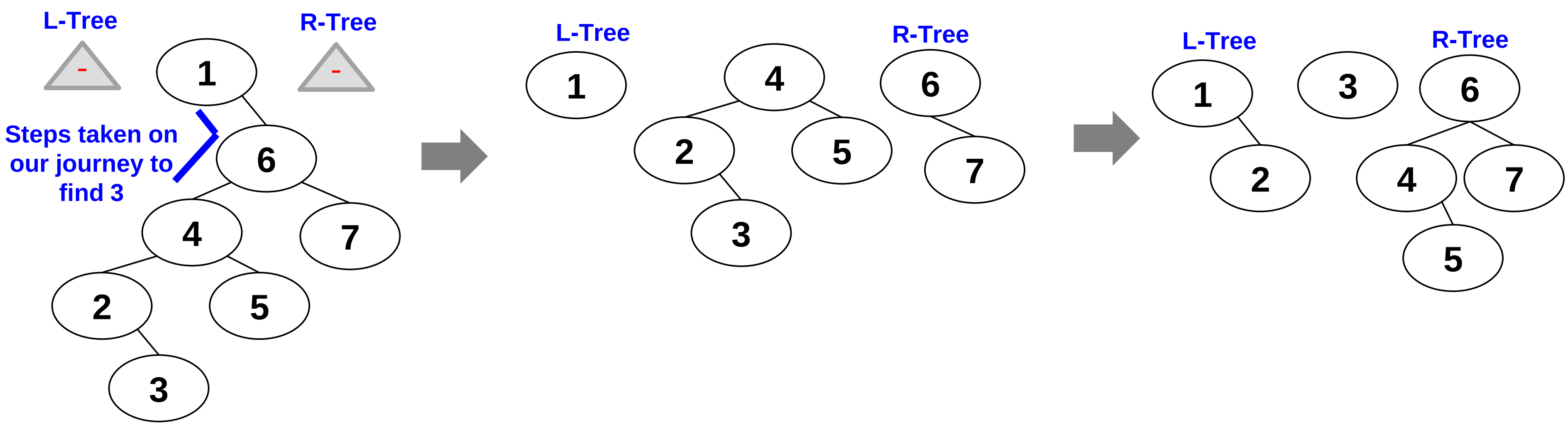


4. Zig-Zag

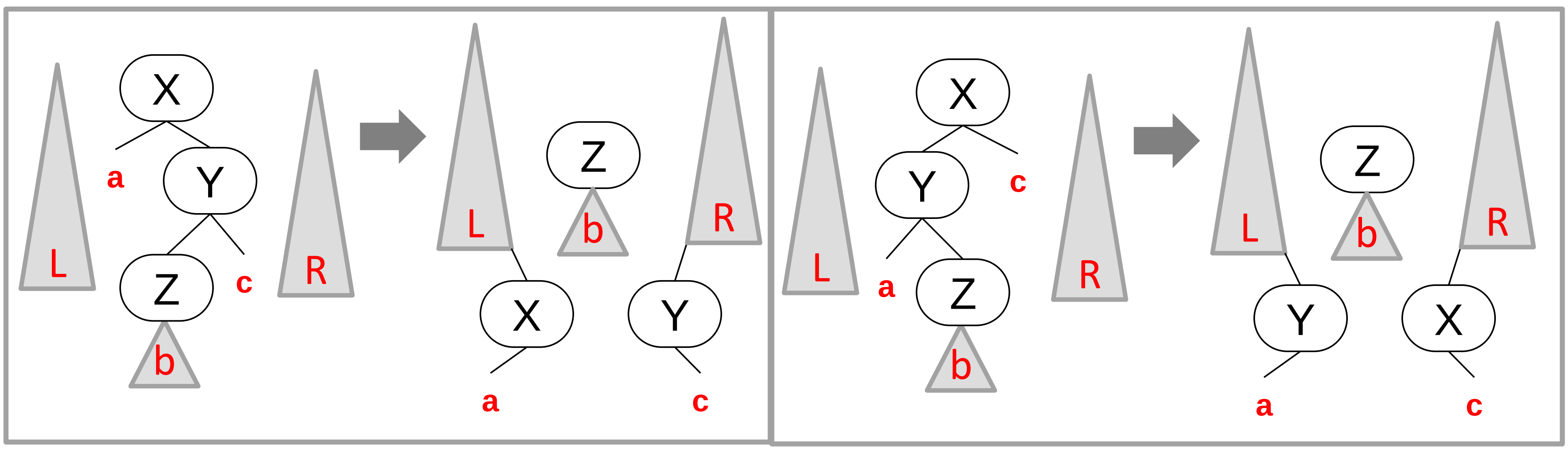




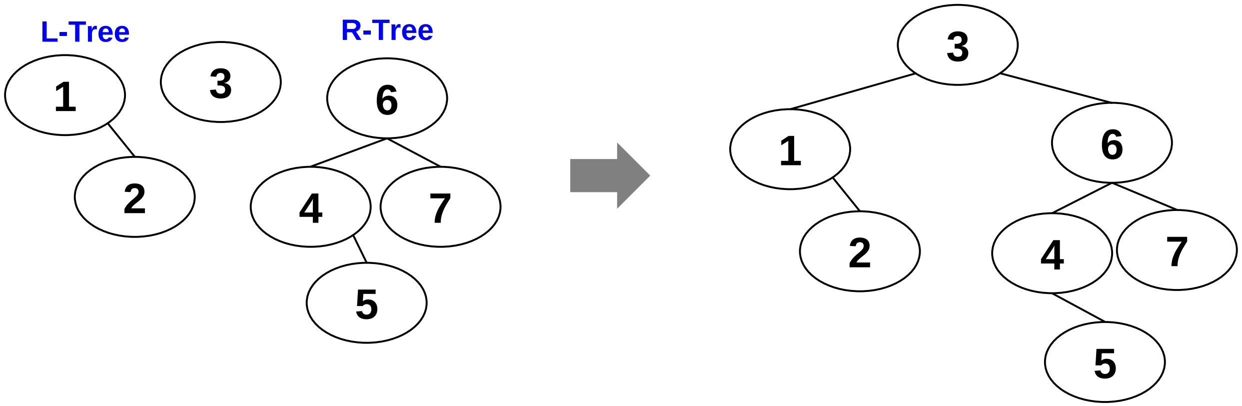
Find(3)



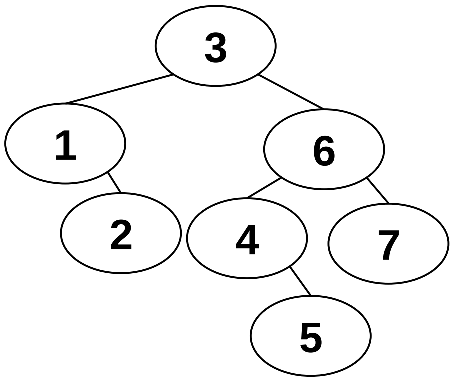
Zig-Zag



Find(3)

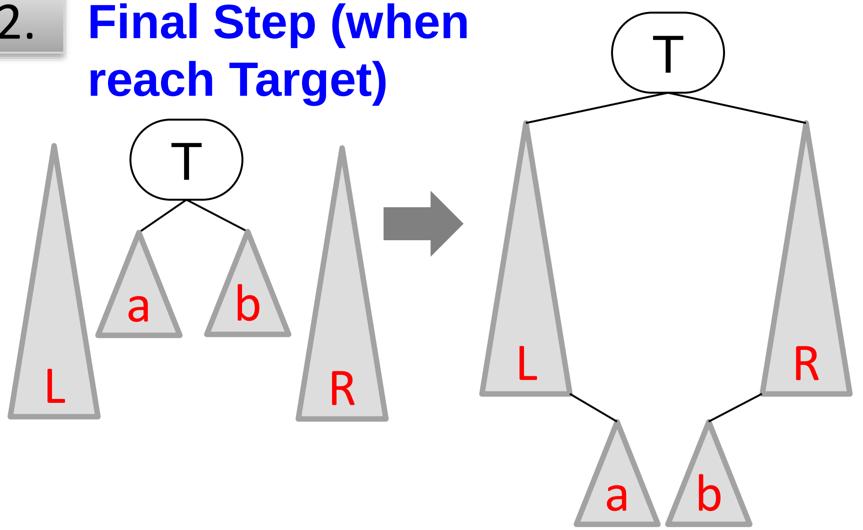


Resulting tree after find

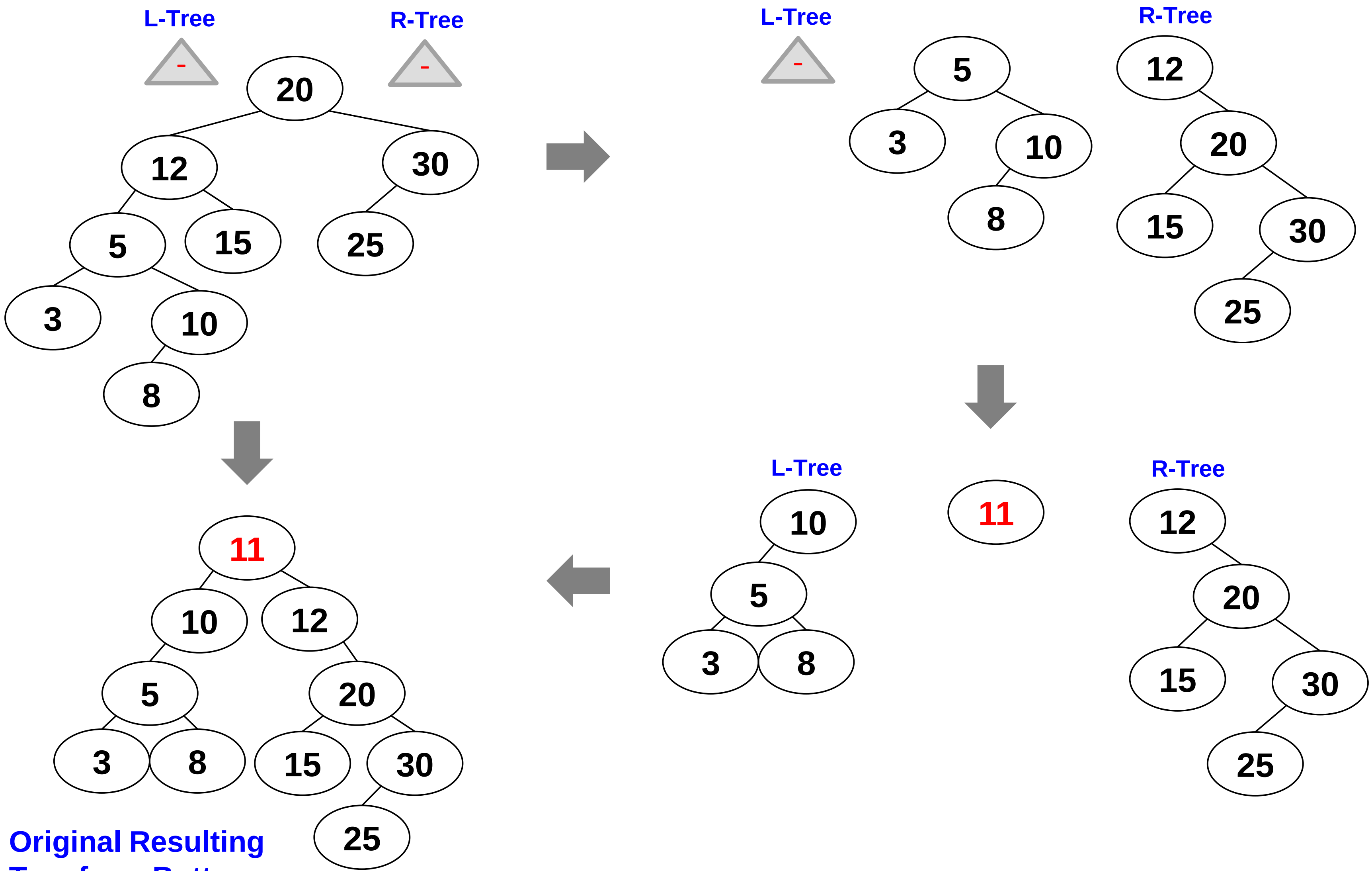


Resulting tree from bottom-up approach

2. **Final Step (when reach Target)**



Insert(11)



Original Resulting Tree from Bottom-up approach

Summary

- Splay trees don't enforce balance but are self-adjusting to yield a balanced tree
- Splay trees provide efficient amortized time operations
  - A single operation may take O(n)
  - m operations on tree with n elements => O(m(log n))
- Uses rotations to attempt balance
- Provides fast access to recently used keys