CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

LOG STRUCTURED MERGE TREES

Series Summation Review

- Let n = 1 + 2 + 4 + … + 2^k = $\sum_{i=0}^{k} 2^i$. What is n?
  - n = 2^{k+1}-1

- What is $\log_2(1) + \log_2(2) + \log_2(4) + \log_2(8) + \ldots + \log_2(2^k)$ = 0 + 1 + 2 + 3+… + k = $\sum_{i=0}^{k} i$
  - O(k^2)

Arithmetic series:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$$

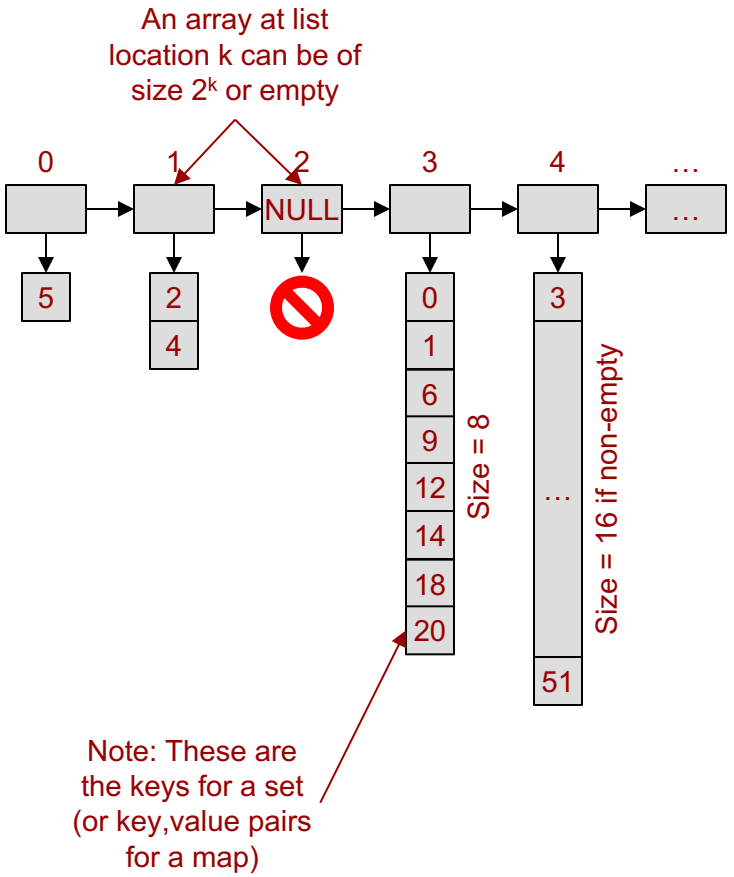
Geometric series

$$\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1} = \theta(c^n)$$

- So then what if k = log(n) as in:
  $\log_2(1) + \log_2(2) + \log_2(4) + \log_2(8) + \ldots + \log_2(2^{\log(n)})$
  - O($\log^2 n$)

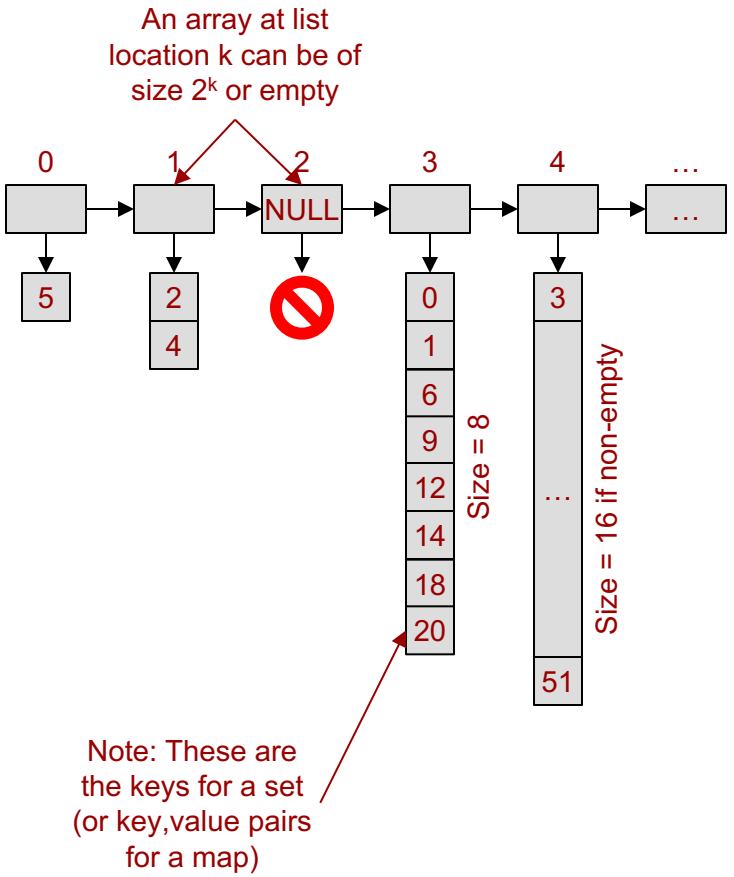
Merge Trees Overview

- Consider a list of (pointers to) arrays with the following constraints
  - Each array is sorted *though no ordering constraints exist between arrays*
  - The array at list index k is of exactly size 2^k or empty



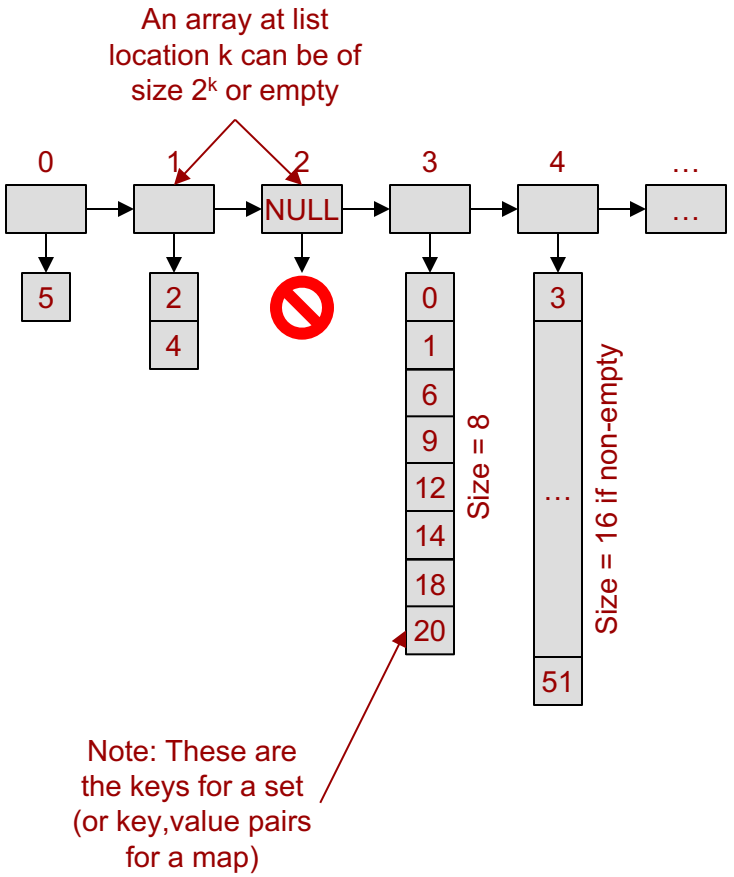
Merge Trees Size

- Define…
  - n as the # of keys in the entire structure
  - k as the size of the list (i.e. positions in the list)
- Given k, what is n?
  - Let n = 1 + 2 + 4 + … + 2^k = $\sum_{i=0}^{k} 2^i$. What is n?
- $n=2^k-1$



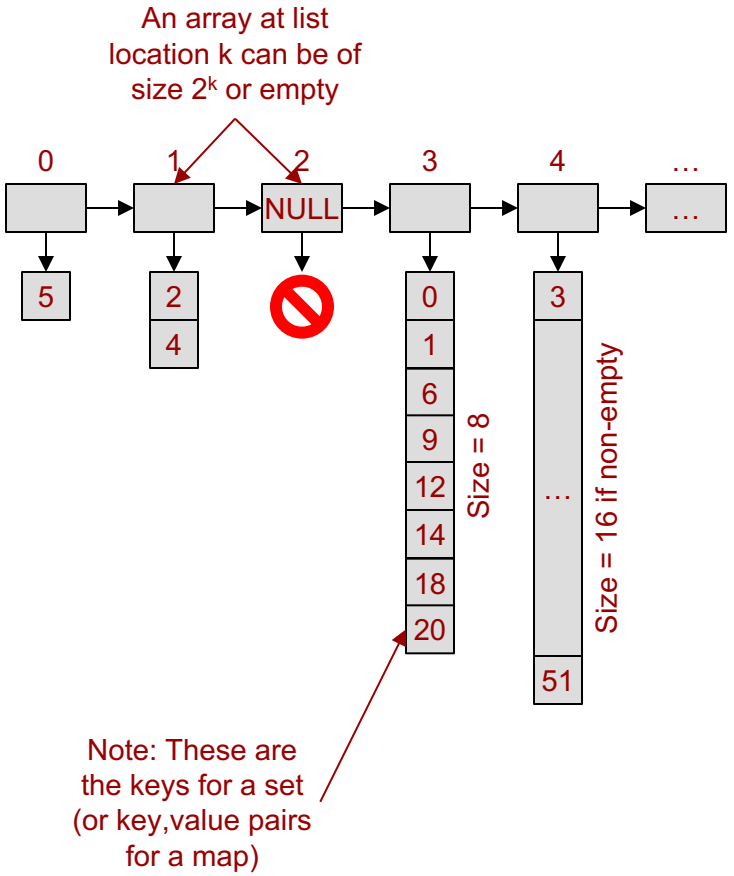
Merge Trees Find Operation

- To find an element (or check if it exists)
- Iterate through the arrays in order (i.e. start with array at list position 0, then the array at list position 1, etc.)
  - In each array perform a binary search
- If you reach the end of the list of arrays without finding the value it does not exist in the set/map



Find Runtime

- What is the worst case runtime of find?
  - When the item is not present which requires, a binary search is performed on each list
- $T(n) = \log_2(1) + \log_2(2) + \ldots \log_2(2^k)$
- $= 0 + 1 + 2 + \ldots + k = \sum_{i=0}^{k} i$
  $= O(k^2)$
- But let's put that in terms of the number of elements in the structure (i.e. n)
  - Recall $k = \log_2(n)+1$
- So find is $O(\log_2(n)^2)$

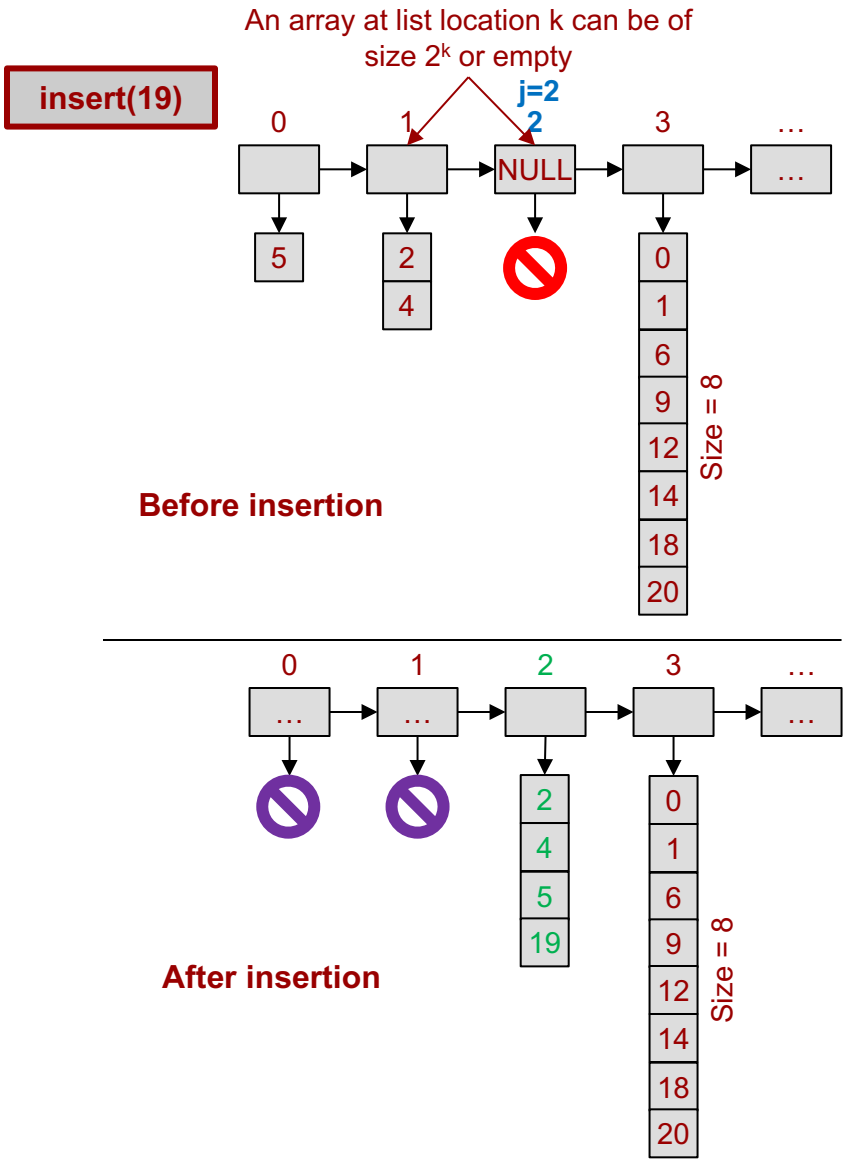


Improving Find's Runtime

- While we might be okay with $[\log(n)]^2$, how might we improve the find runtime in the general case?
  - Hint: I would be willing to pay O(1) to know if a key is not in a particular array without having to perform find
- A Bloom filter could be maintained alongside each array and allow us to skip performing a binary search in an array

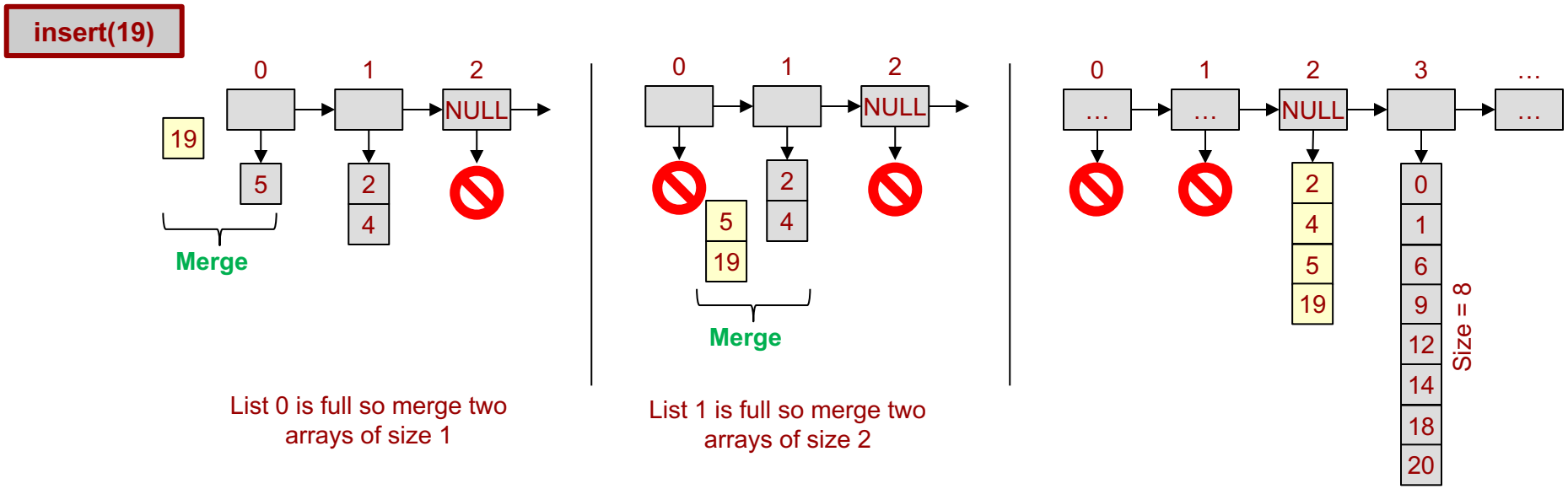
Insertion Algorithm

- Let j be the smallest integer such that array j is empty (first empty slot in the list of arrays)
- An insertion will cause
  - Location j's array to become filled
  - Locations 0 through j-1 to become empty

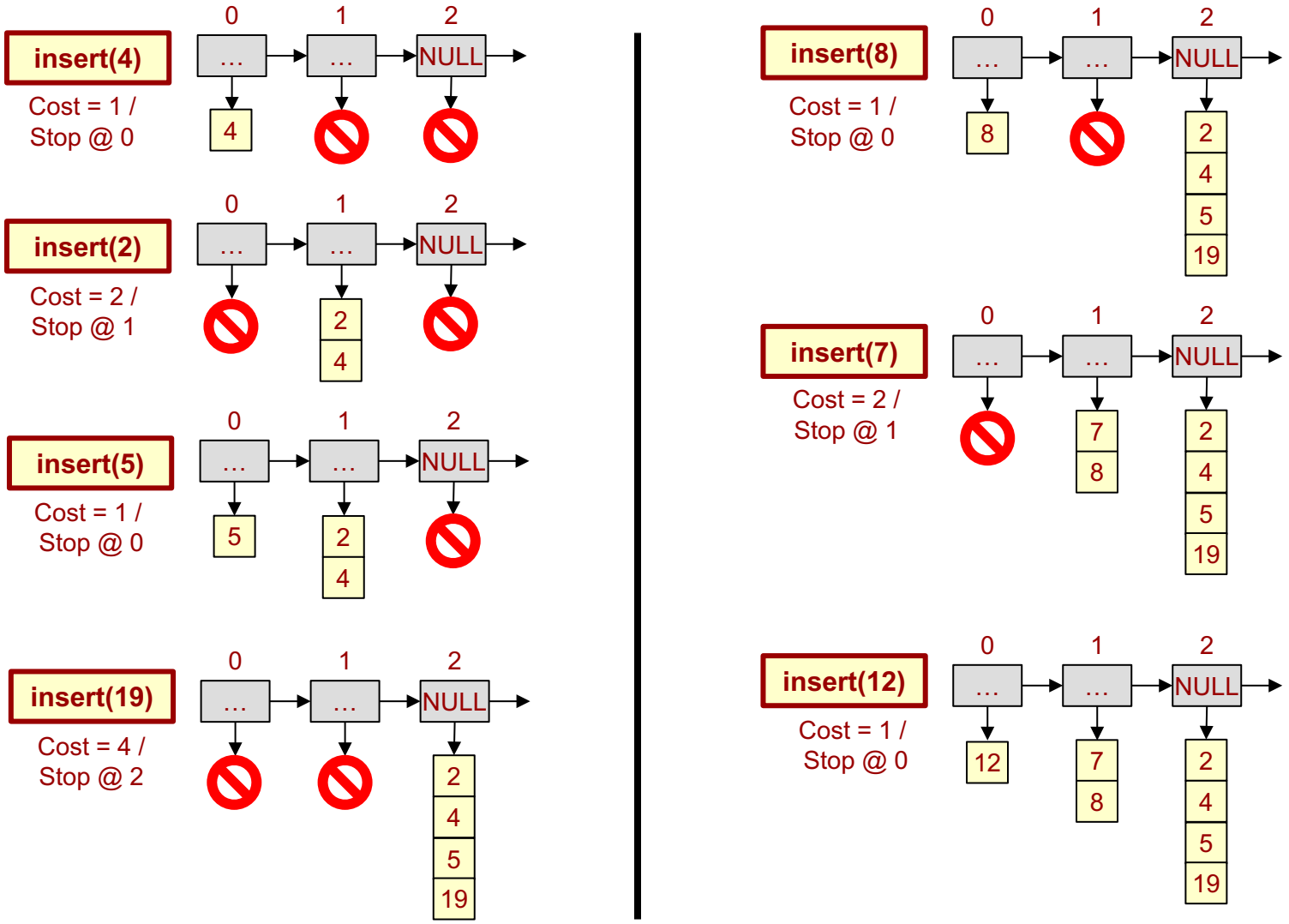


Insertion Algorithm

- Starting at array 0, iteratively merge the previously merged array with the next, stopping when an empty location is encountered

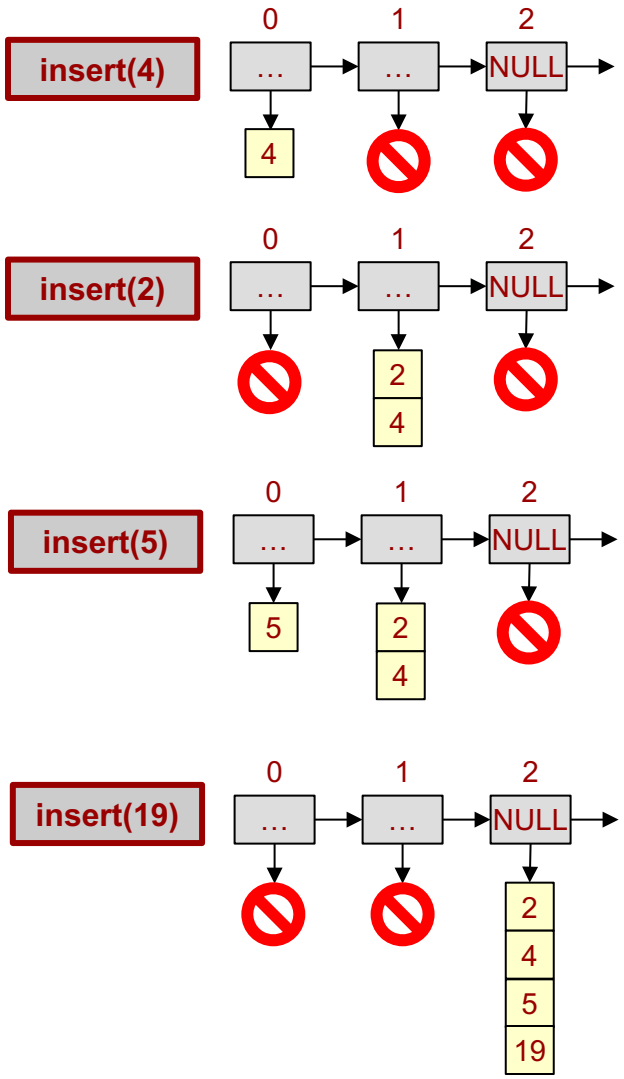


Insert Examples



Insertion Runtime: First Look

- Best case?
  - First list is empty and allows direct insertion in O(1)
- Worst case?
  - All list entries (arrays) are full so we have to merge at each location
  - In this case we will end with an array of size $n=2^k$ in position k
  - Also recall merging two arrays of size m is Θ(m)
  - So the total cost of all the merges is $1 + 2 + 4 + 8 + \ldots + n = 2*n-1 = \Theta(n) = \Theta(2^k)$
- But if the worst case occurs how soon can it occur again?
  - It seems the costs vary from one insert to the next
  - This is a good place to use amortized analysis

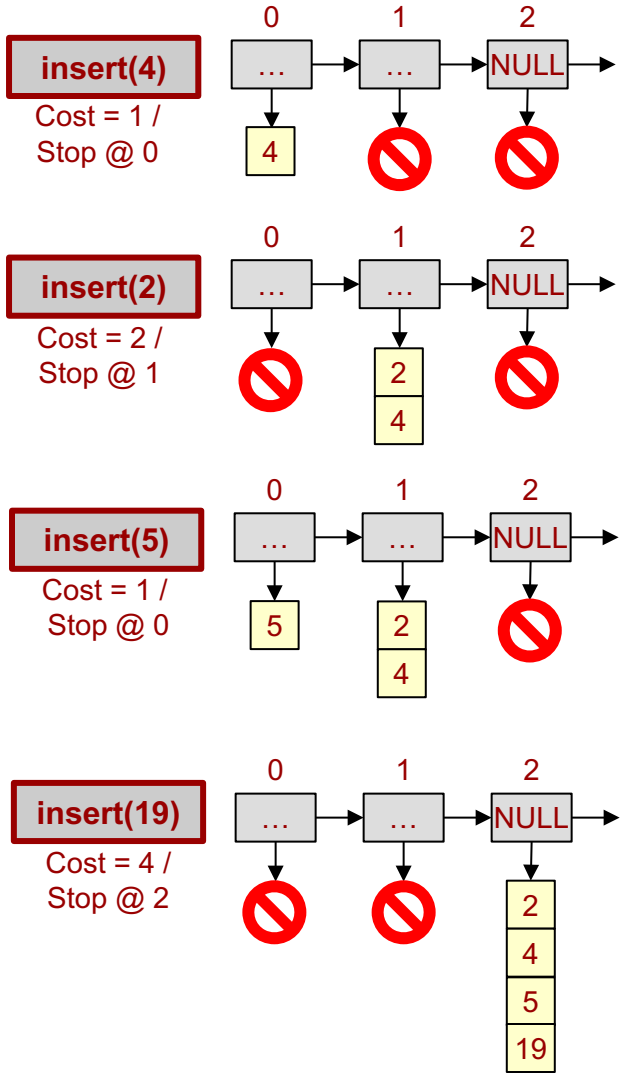


Total Cost for N insertions

- Total cost of n=16 insertions:
  - 1+2+1+4+1+2+1+8+1+2+1+4+1+2+1+16
- =1*n/2 + 2*n/4 + 4*n/8 + 8*n/16 + n
- =n/2 + n/2 + n/2 + n/2 + n
- =n/2*$\log_2(n)$ + n
- Amortized cost = Total cost / n operations
  - $\log_2(n)/2 + 1 = O(\log_2(n))$

Amortized Analysis of Insert

- We have said when you end (place an array) in position k you have to do $O(2^{k+1})$ work for all the merges
- How often do we end in position k
  - The 0^{th} position will be free with probability ½ (p=0.5)
  - We will stop at the 1^{st} position with probability ¼ (p=0.25)
  - We will stop at the 2^{nd} position with probability 1/8 (p=0.125)
  - We will stop at the k^{th} position with probability $1/2^k = 2^{-k}$
- So we pay 2^{k+1} with probability $2^{-(k+1)}$
- Suppose we have n items in the structure (i.e. max k is $\log_2 n$) what is the expected cost of inserting a new element
  - $\sum_{k=0}^{\log(n)} 2^{k+1} 2^{-(k+1)} = \sum_{k=0}^{\log(n)} 1 = \log(n)$



Summary

- Variants of log structured merge trees have found popular usage in industry
  - Starting array size might be fairly large (size of memory of a single server)
  - Large arrays (from merging) are stored on disk
- Pros:
  - Ease of implementation
  - Sequential access of arrays helps lower its constant factors
- Operations:
  - Find = $\log(n)^2$
  - Insert = Amortized log(n)
  - Remove = often not considered/supported

SPLAY TREES

Sources / Reading

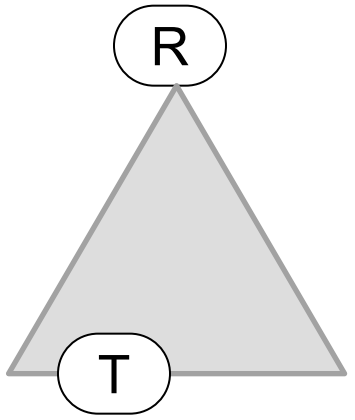
- Material for these slides was derived from the following sources
  - https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf
  - http://digital.cs.usu.edu/~allan/DS/Notes/Ch22.pdf
  - http://www.cs.umd.edu/~meesh/420/Notes/MountNotes/lecture10-splay.pdf
- Nice Visualization Tool
  - https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Splay Tree Intro

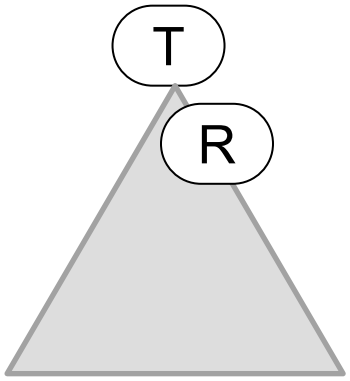
- Another map/set implementation (storing keys or key/value pairs)
  - Insert, Remove, Find
- Recall…To do **m** inserts/finds/removes on an RBTree w/ **n** elements would cost?
  - O(**m***log(**n**))
- Splay trees have a worst case find, insert, delete time of…
  - O(**n**)
- However, they guarantee that if you do **m** operations on a splay tree with **n** elements that the total ("amortized"…uh-oh) time is
  - O(**m***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…



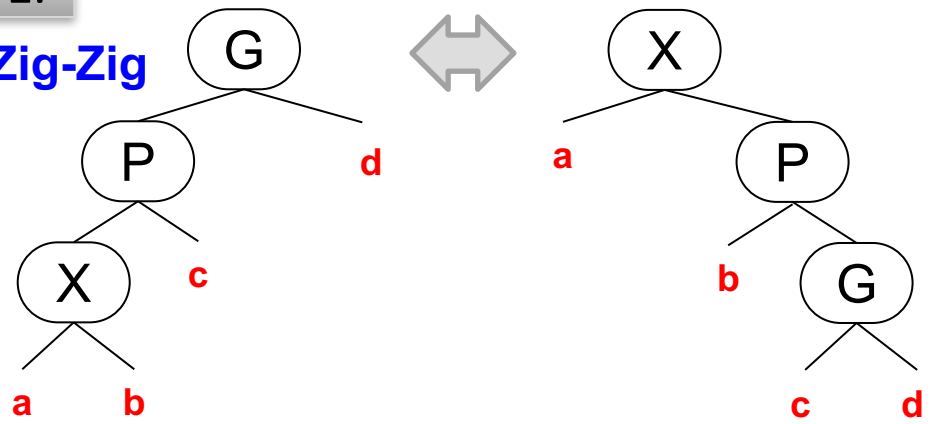
…T will end up as the root node with the old root in the top level or two

Principle of Locality

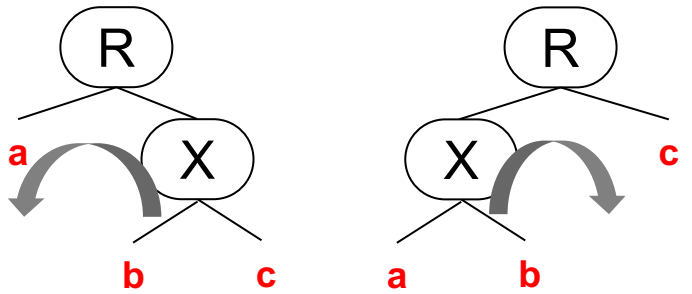
- 2 dimensions of this principle: space & time
- Spatial Locality – Future accesses will likely cluster near current accesses
  - Instructions and data arrays are sequential (they are all one after the next)
- Temporal Locality – Future accesses will likely be to recently accessed items
  - Same code 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

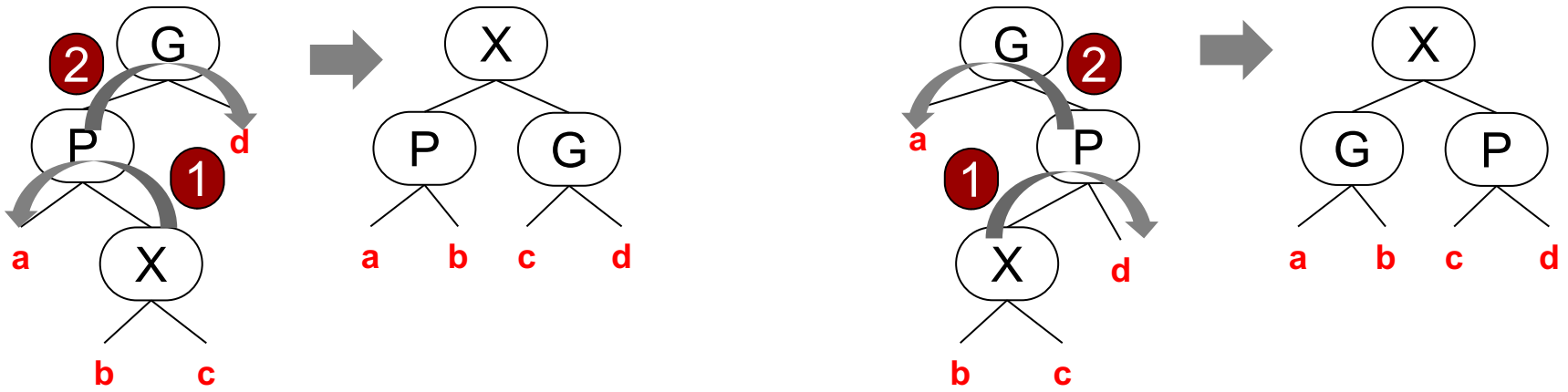
1. **Zig-Zig**



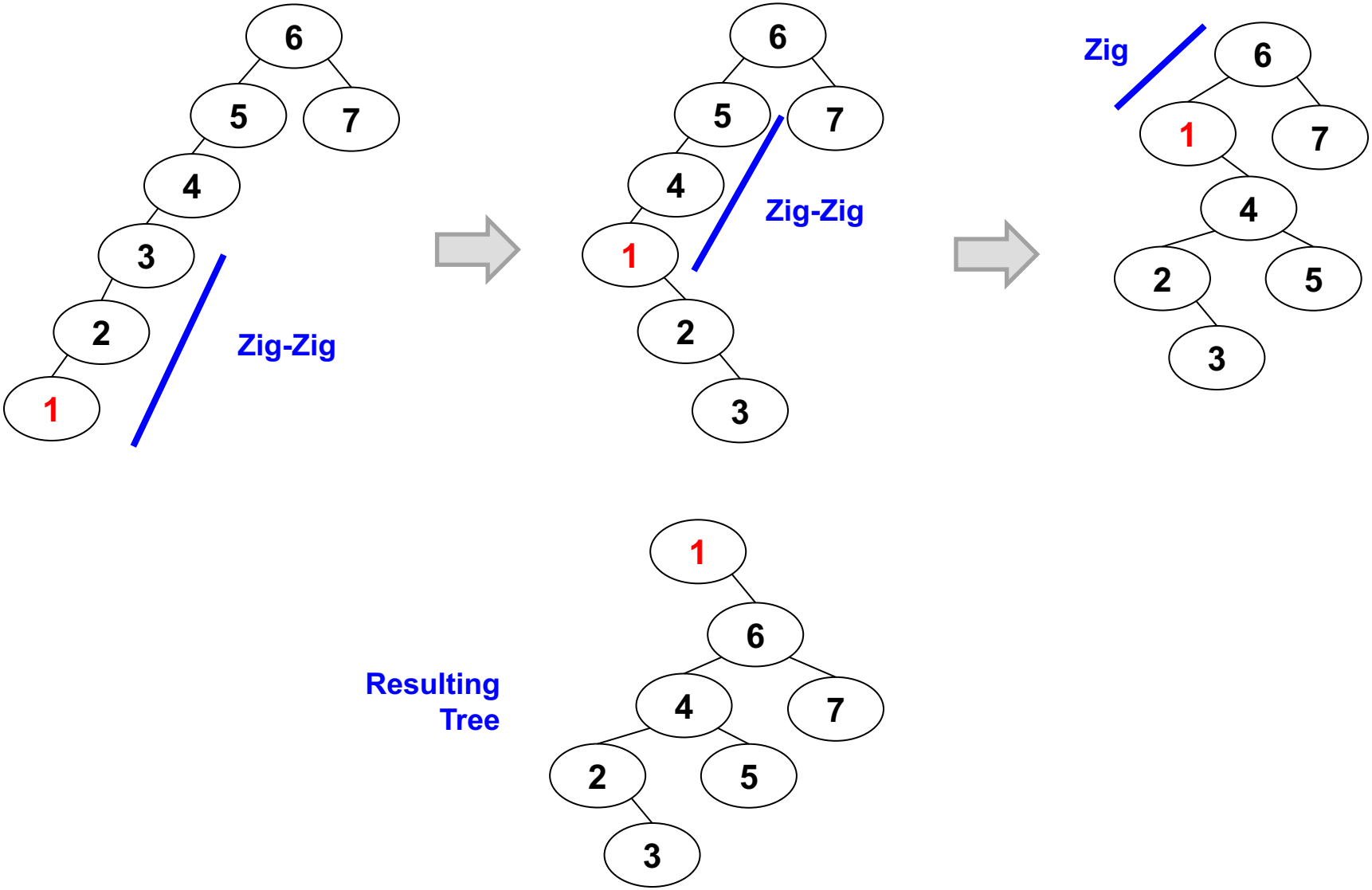
3. **Root/Zig Case (Single Rotation)**



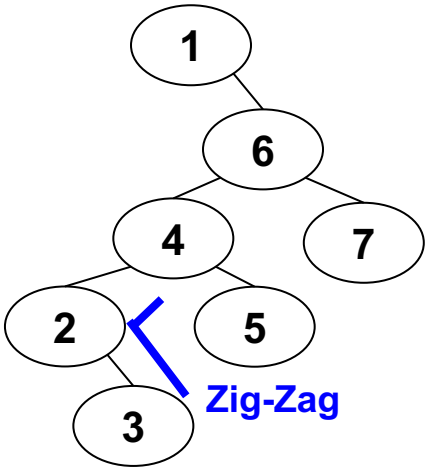
2. **Zig-Zag**

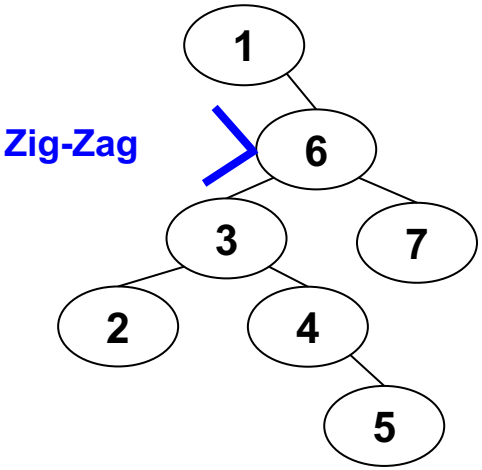


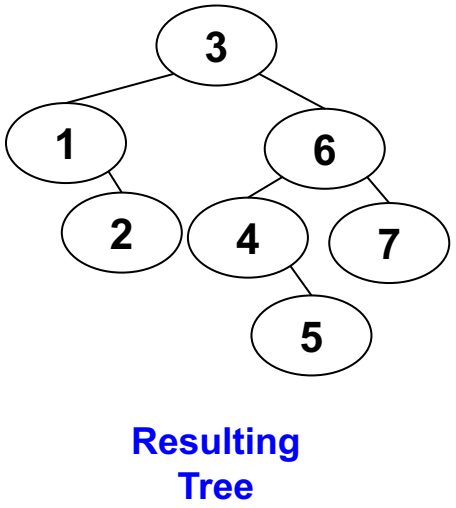
Find(1)


6
5
7
4
3
2
1
Zig-Zig
6
5
7
4
1
2
3
Zig-Zig
Zig
6
1
7
4
2
5
3
Resulting Tree
1
6
4
7
2
5
3

Find(3)





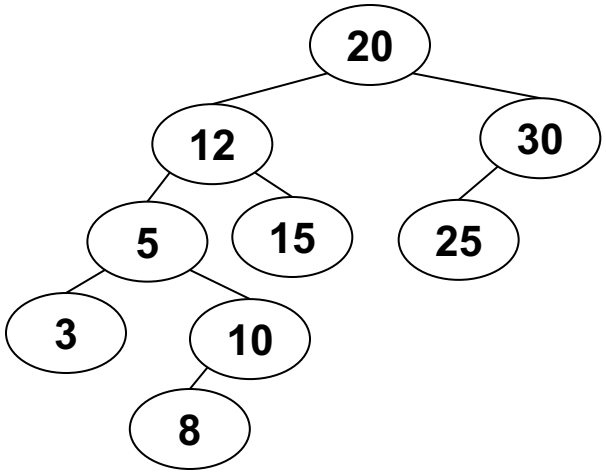


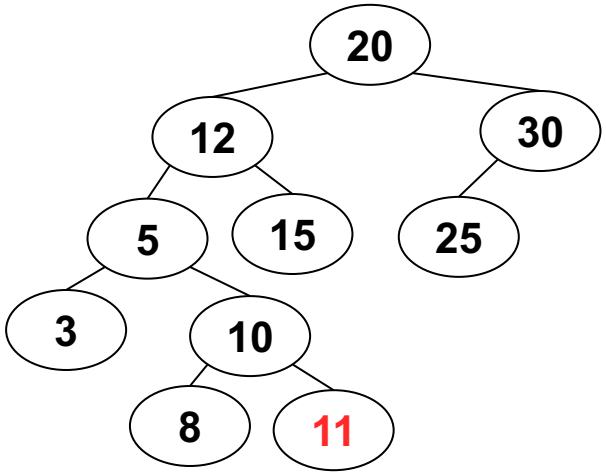
- Notice the tree is starting to look at lot more balanced

Worst Case

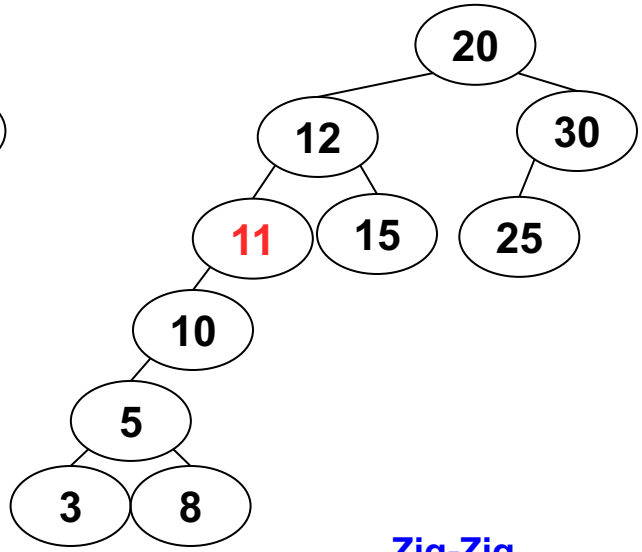
- Suppose you want to make the amortized time (averaged time over multiple calls to find/insert/remove) look bad, you might try to always access the _____ node in the tree
  - Deepest
- But splay trees have a property that as we keep accessing deep nodes the tree starts to balance and thus access to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

Insert(11)

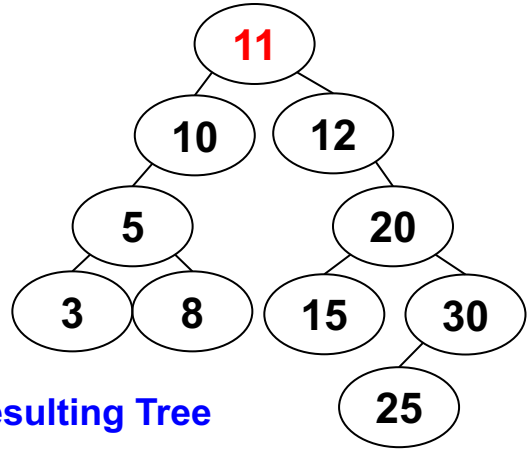




Zig-Zig

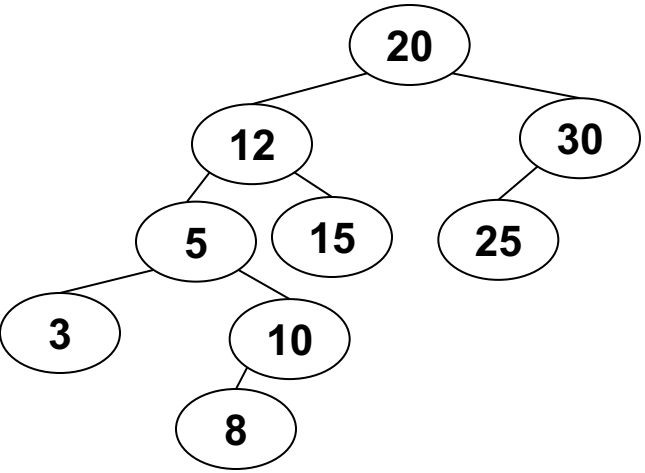


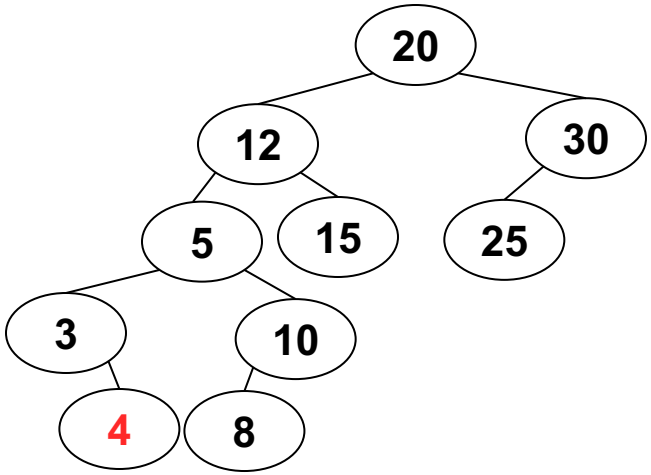
Zig-Zig



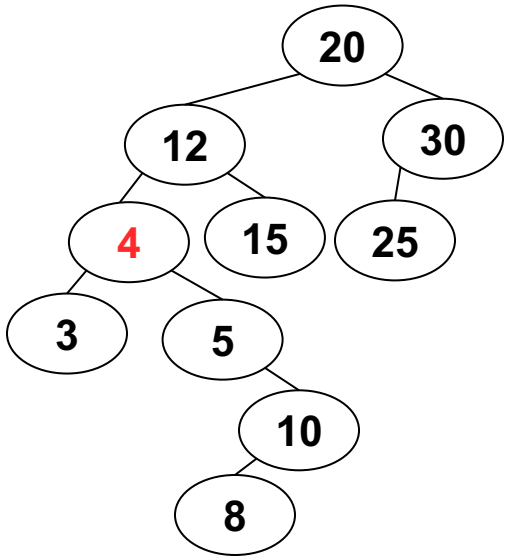
Resulting Tree

Insert(4)



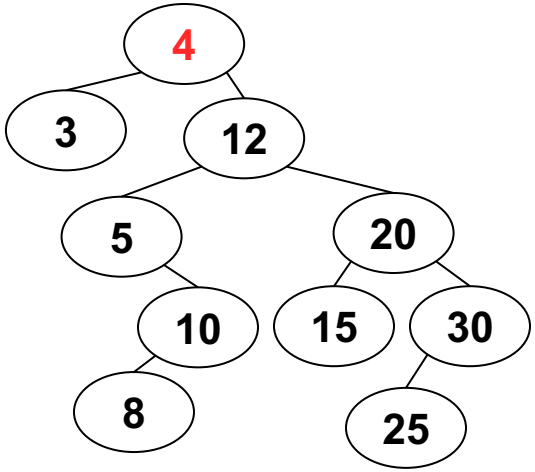


Zig-Zag



Zig-Zig

Resulting Tree



Activity

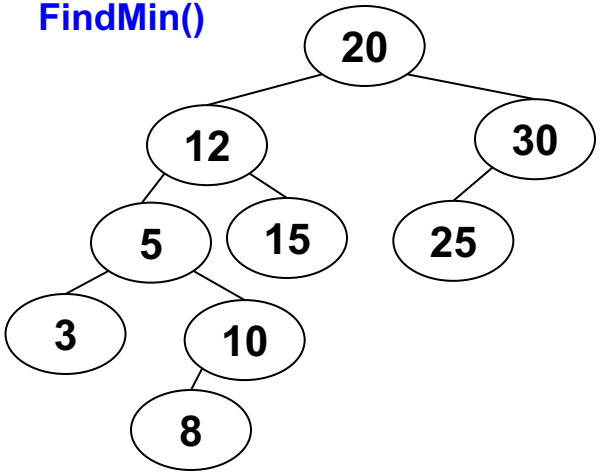
- Go to
  - https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
  - Try to be an adversary by inserting and finding elements that would cause O(n) each time

Splay Tree Supported Operations

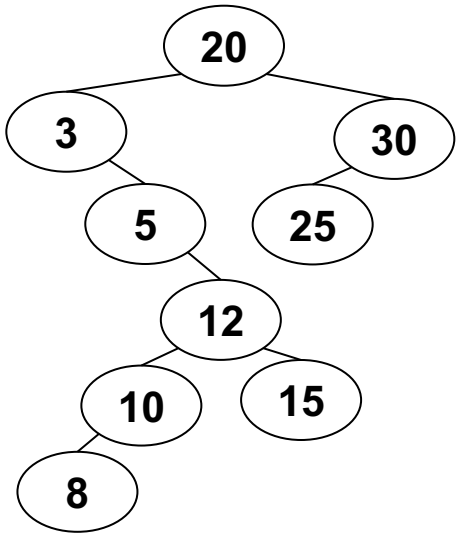
- Insert(x)
  - Normal BST insert, then splay x
- Find(x)
  - Attempt normal BST find(x) and splay last node visited
    - If x is in the tree, then we splay x
    - If x is not in the tree we splay the leaf node where our search ended
- FindMin(), FindMax()
  - Walk to far left or right of tree, return 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()

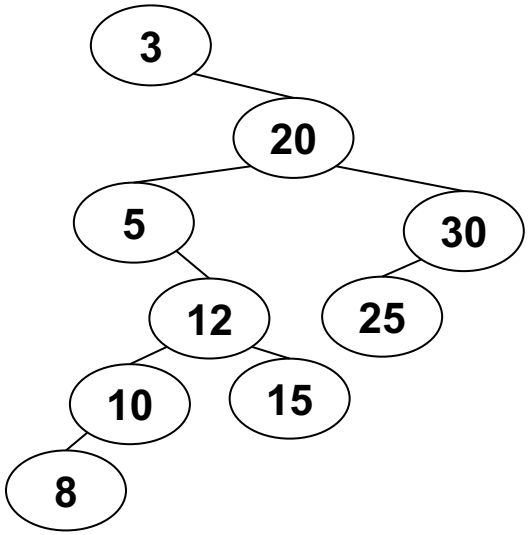
FindMin()



Zig-Zig

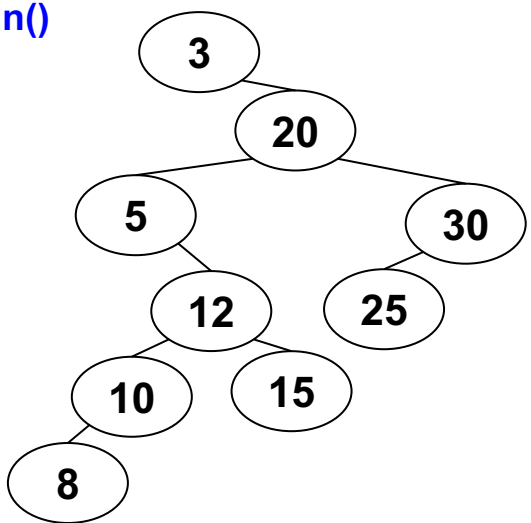


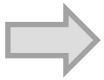
Zig

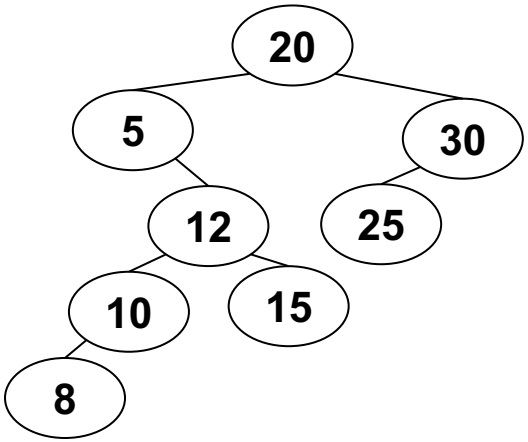


Resulting Tree

DeleteMin()

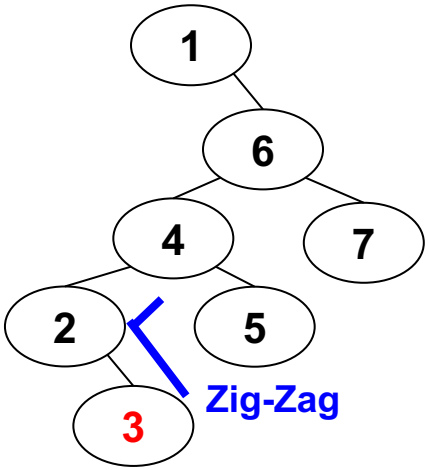


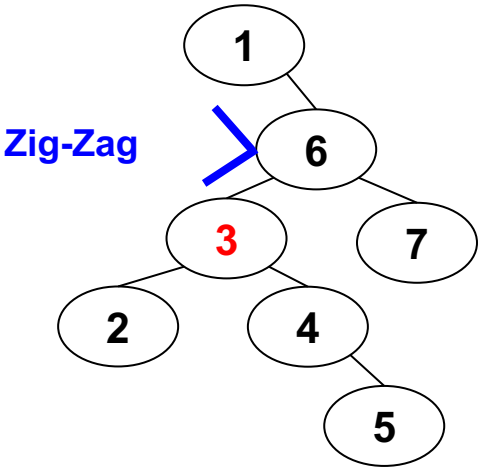


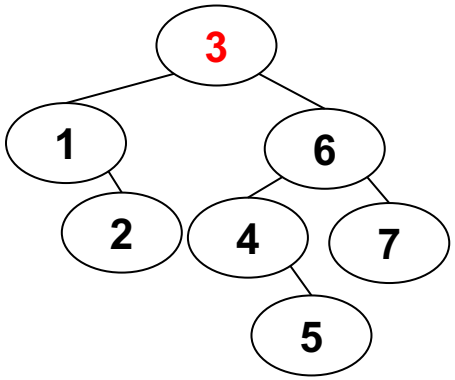


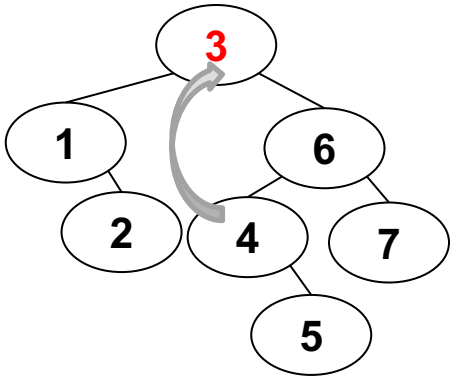
Resulting Tree

Remove(3)

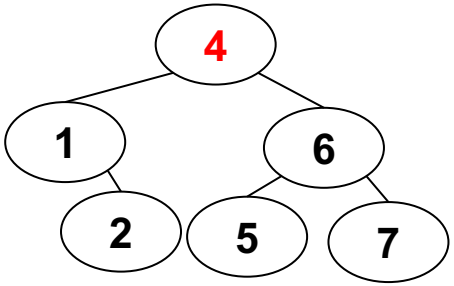








Copy successor or predecessor to root

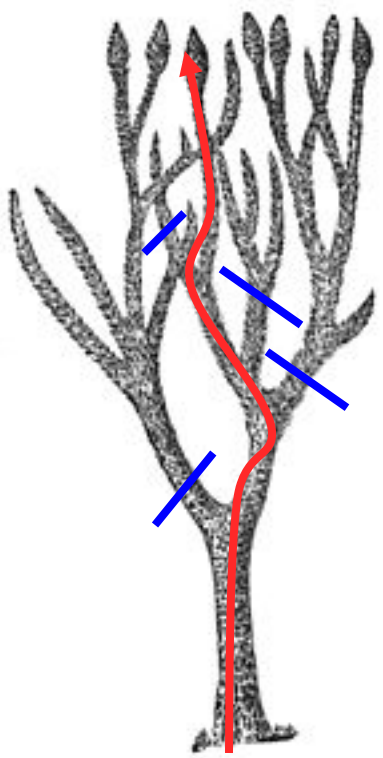


Delete successor (Remove node or reattach single child)

Resulting Tree

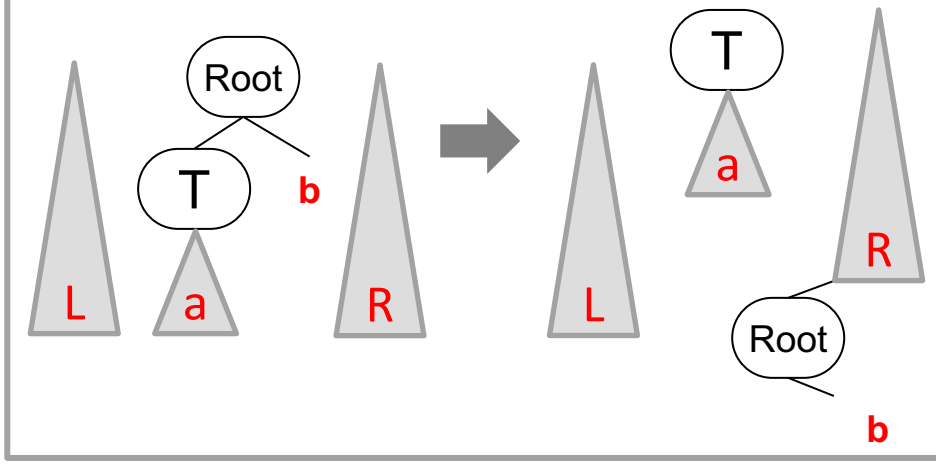
Top Down Splaying

- 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

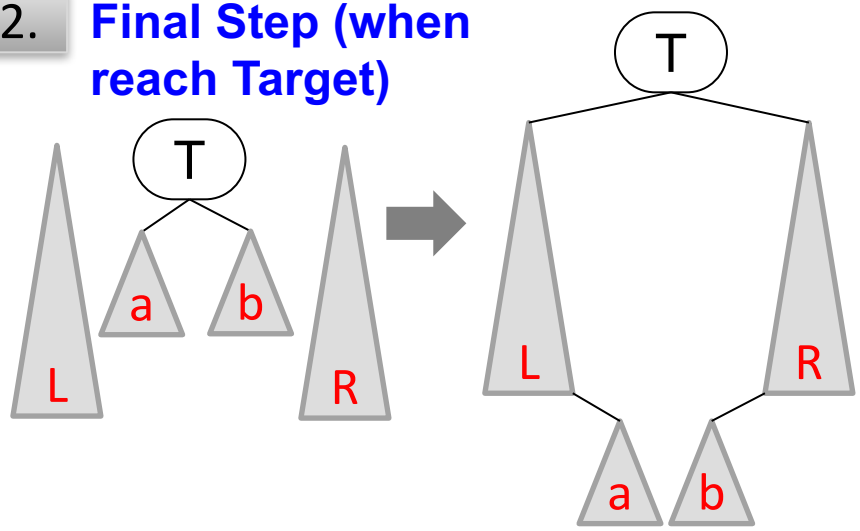


Top-Down Splaying

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

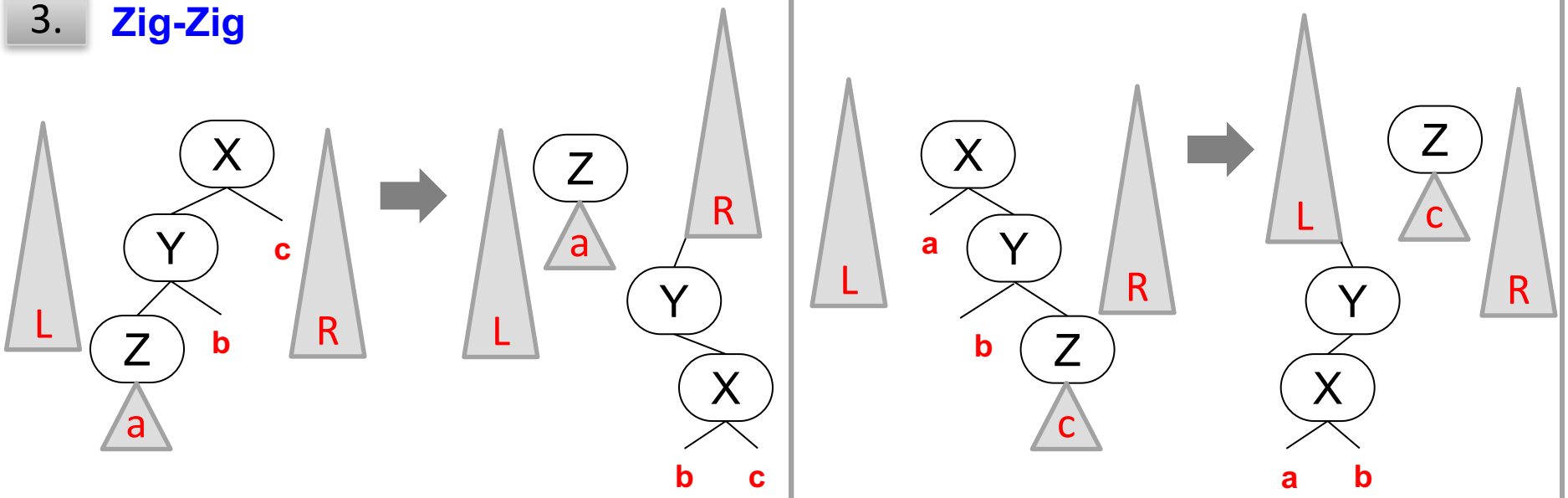


2. **Final Step (when reach Target)**

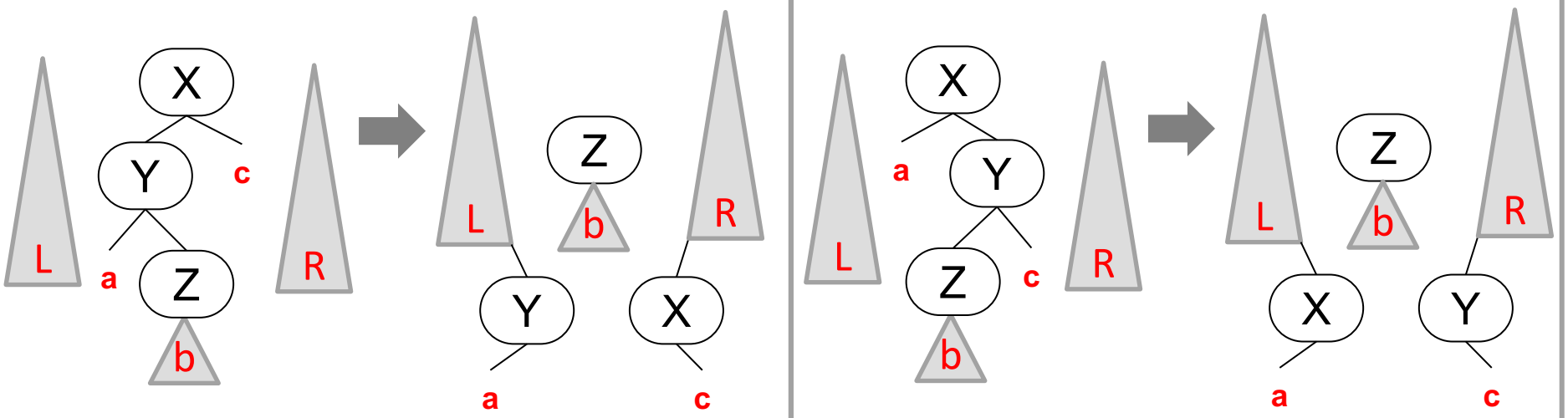


Top-Down Splaying

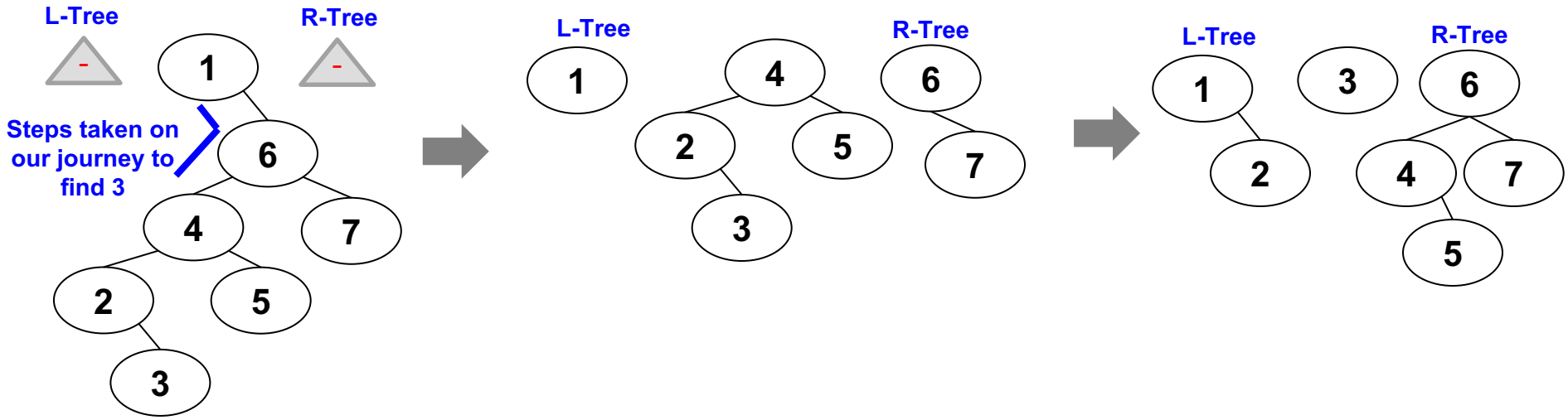
3. Zig-Zig



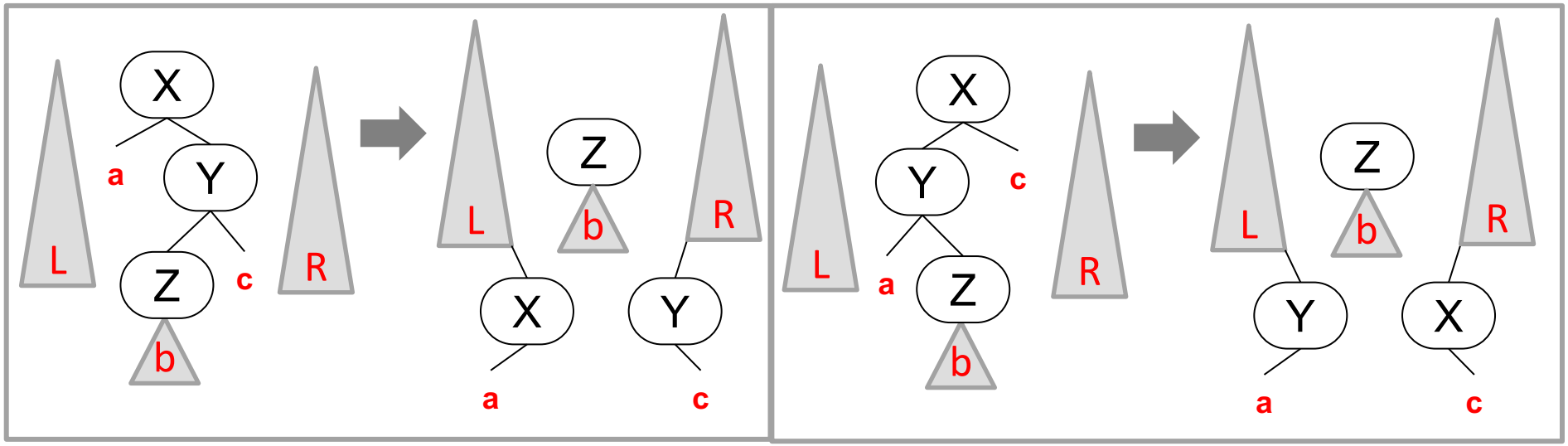
4. Zig-Zag



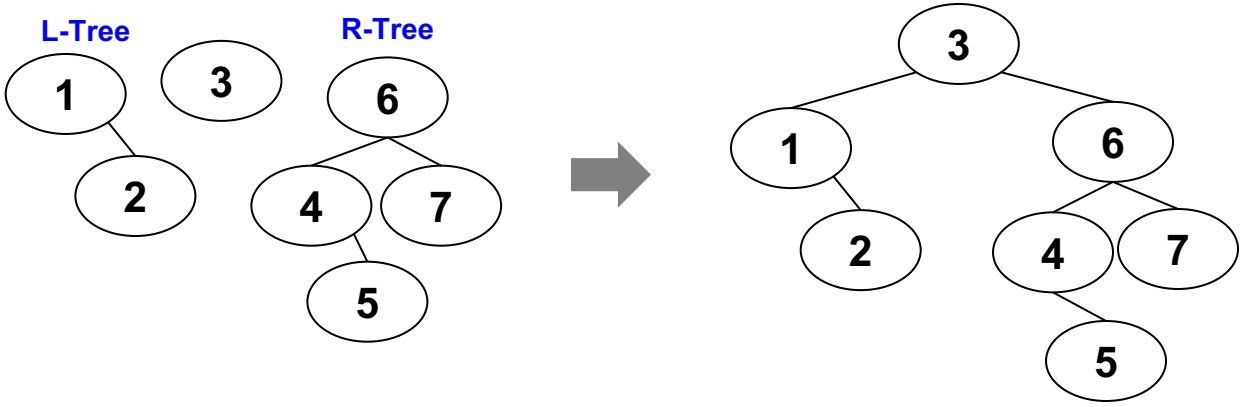
Find(3)



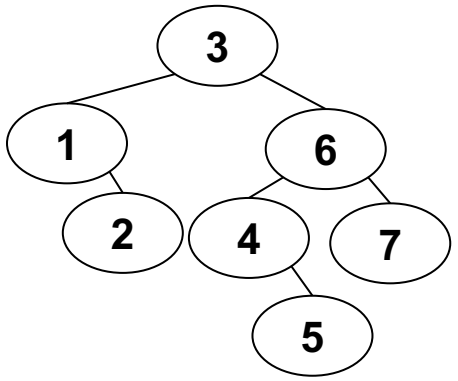
Zig-Zag



Find(3)

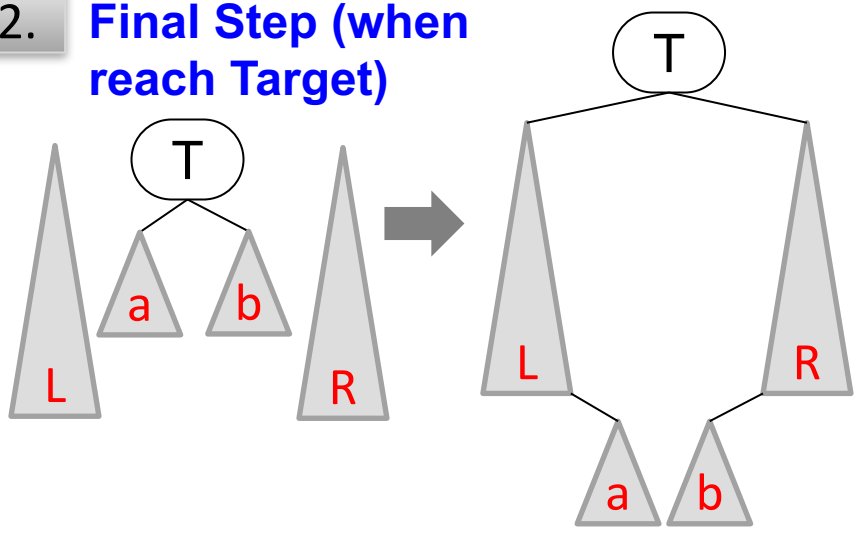


Resulting tree after find

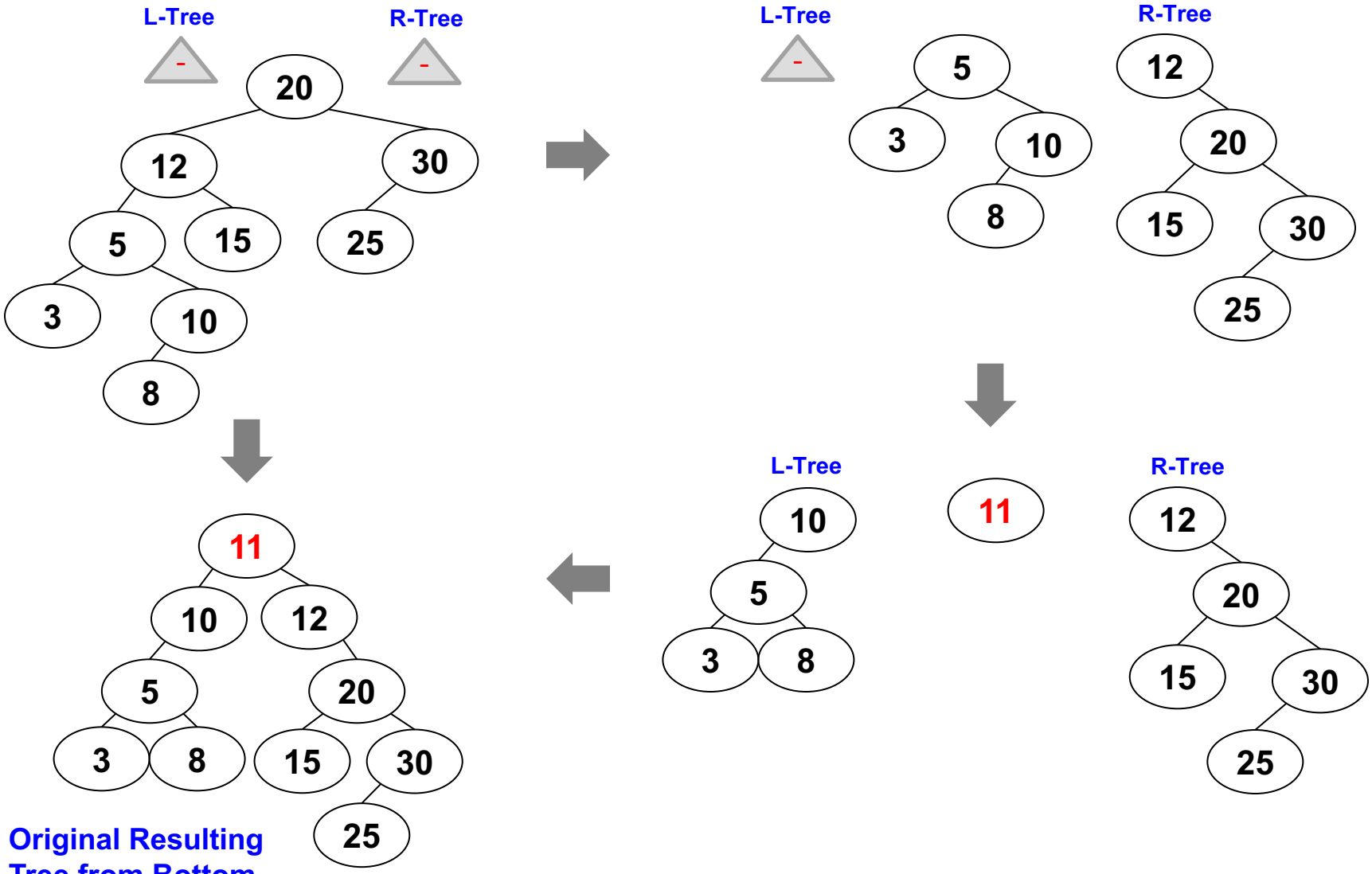


Resulting tree from bottom-up approach

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 attempt 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