INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 8

Instructor: James Pope

As Lectured by: Arda Gumusalan

Email: jpope8@gmu.edu

Today

- Questions from last class?
  - Binary Search Trees
  - Self Balancing Trees
  - AVL Rotations
- Schedule
  - 2-3 Trees
  - Red-Black Trees
  - B Trees

Binary Search Tree

- A type of ________
- Relationship maintained between...
  -
- Rules
  -
  -
  -
  -

Binary Search Tree

- A type of binary tree!
- Relationship maintained between...
  - parent and both children
- Rules
  - parent > elements in left sub tree
  - parent < elements in right sub tree
  - both children are binary search trees
  - no duplicates (how do we handle this?)
- http://people.ksp.sk/~kuko/bak/

Binary Search Tree: Big-O

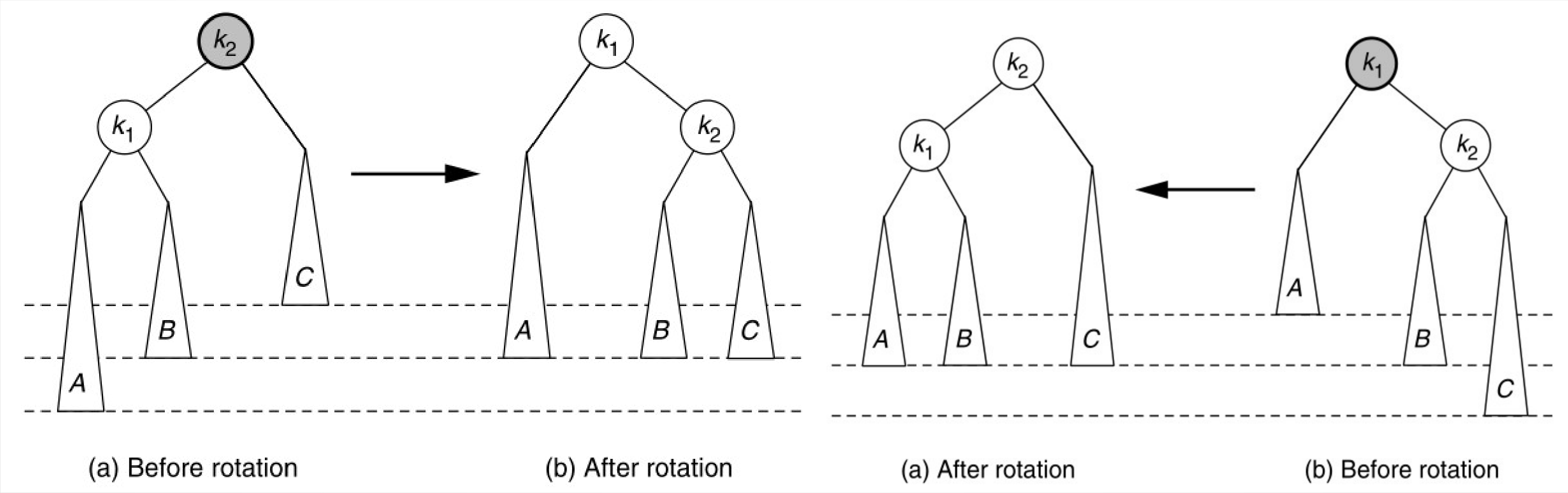
- So... binary search trees
  - What is the height?
  - What is worst/best/average of:
    - finding an element
    - inserting an element
    - deleting an element
      - What if we want to delete the root?
      - Recall:
        - Predecessor: the largest value that is smaller than X.
        - Successor: the smallest values that is greater than X.

How do we improve performance?

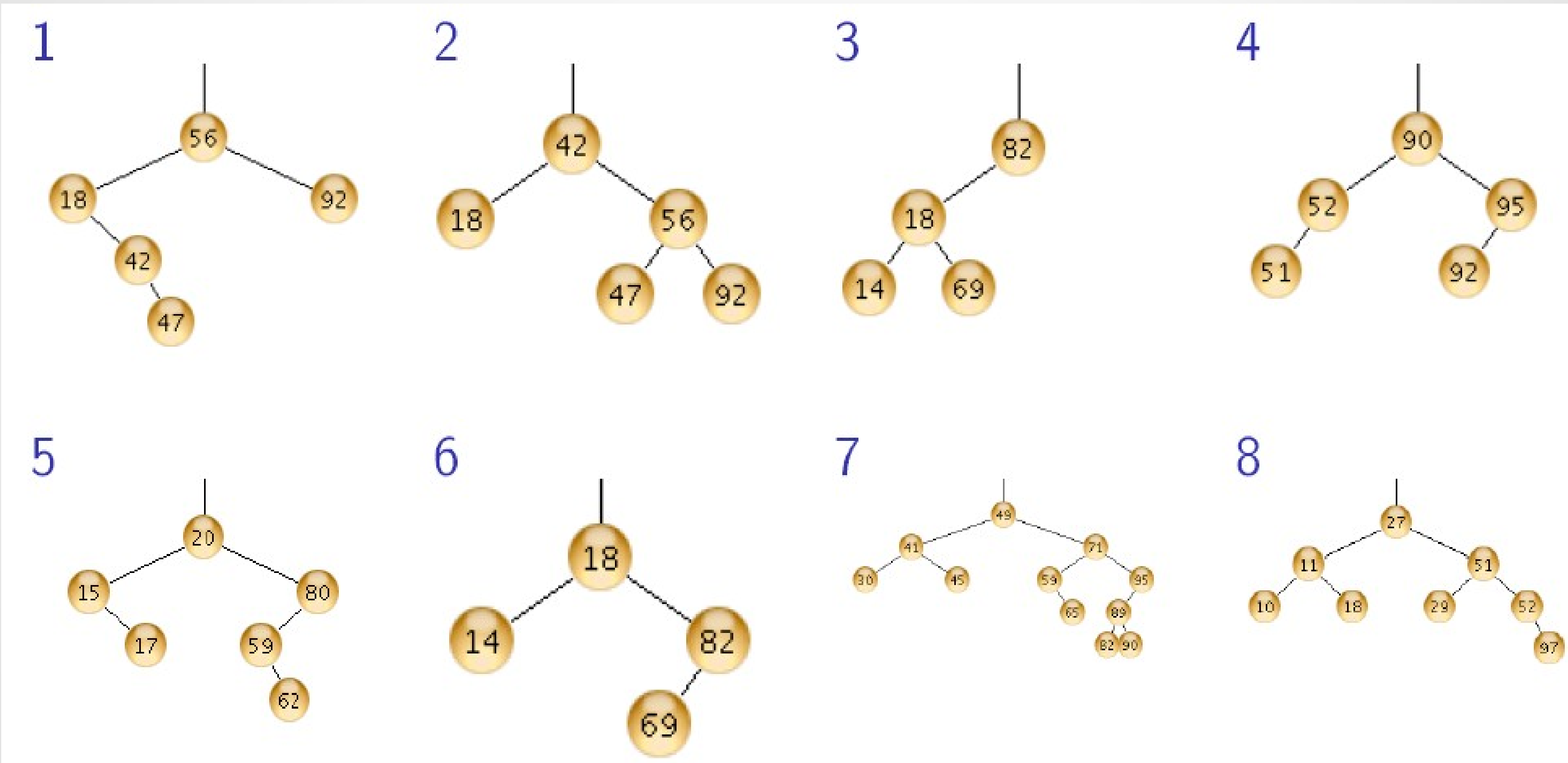
- What is the thing we need?
  - What causes the gap between worst case and average case?
- How do we get it?

How do we improve performance?

- What do we need? Balance!
  - preferably self-balancing! (balance as you add/remove/search the tree)
- How? Rotate!

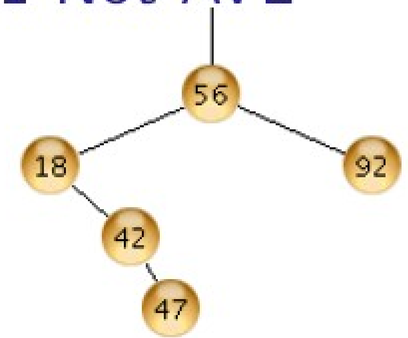


Are These AVL Trees?

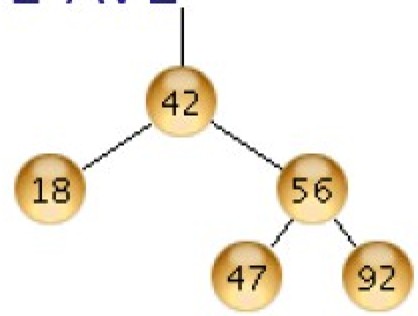


Answers

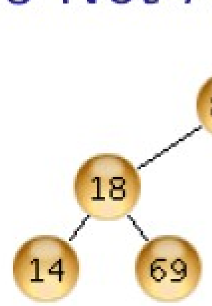
1 Not AVL



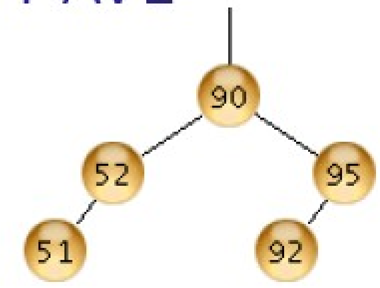
2 AVL



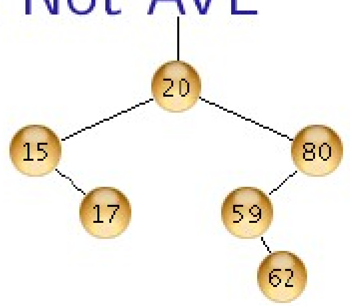
3 Not AVL



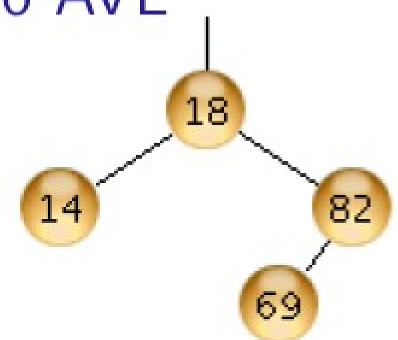
4 AVL



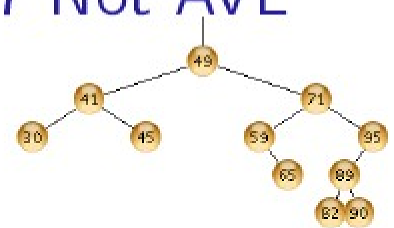
5 Not AVL



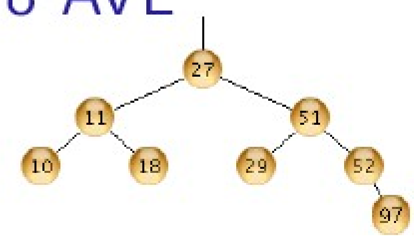
6 AVL



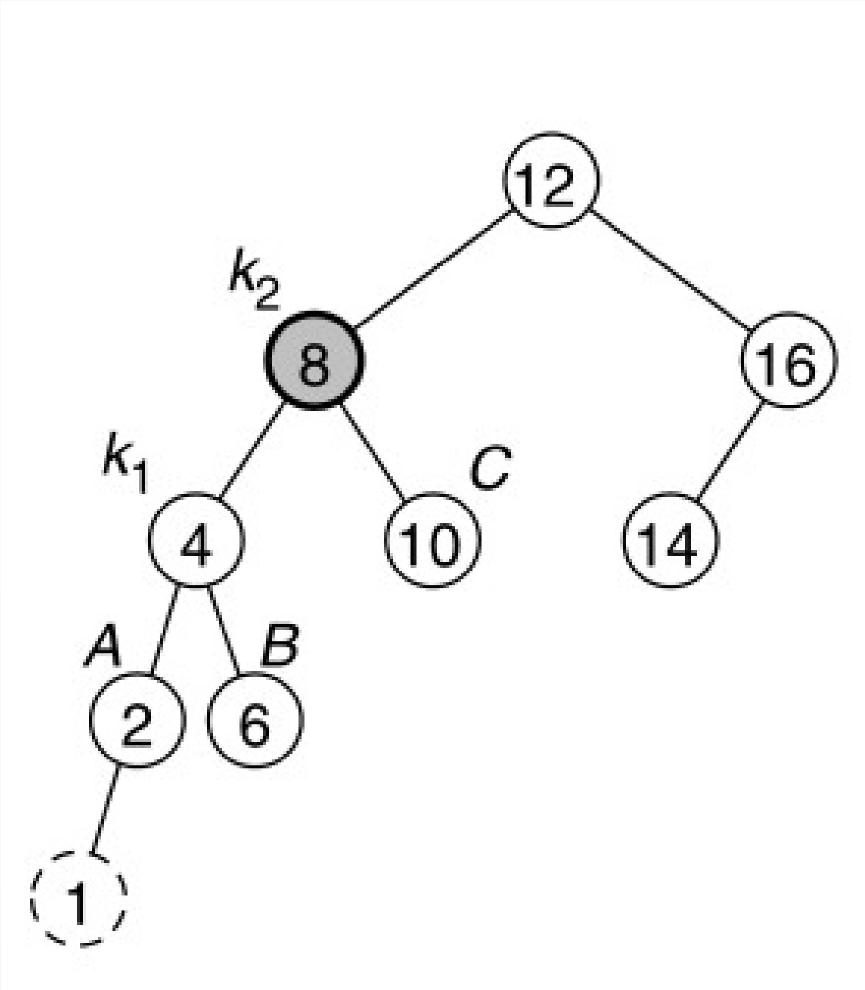
7 Not AVL



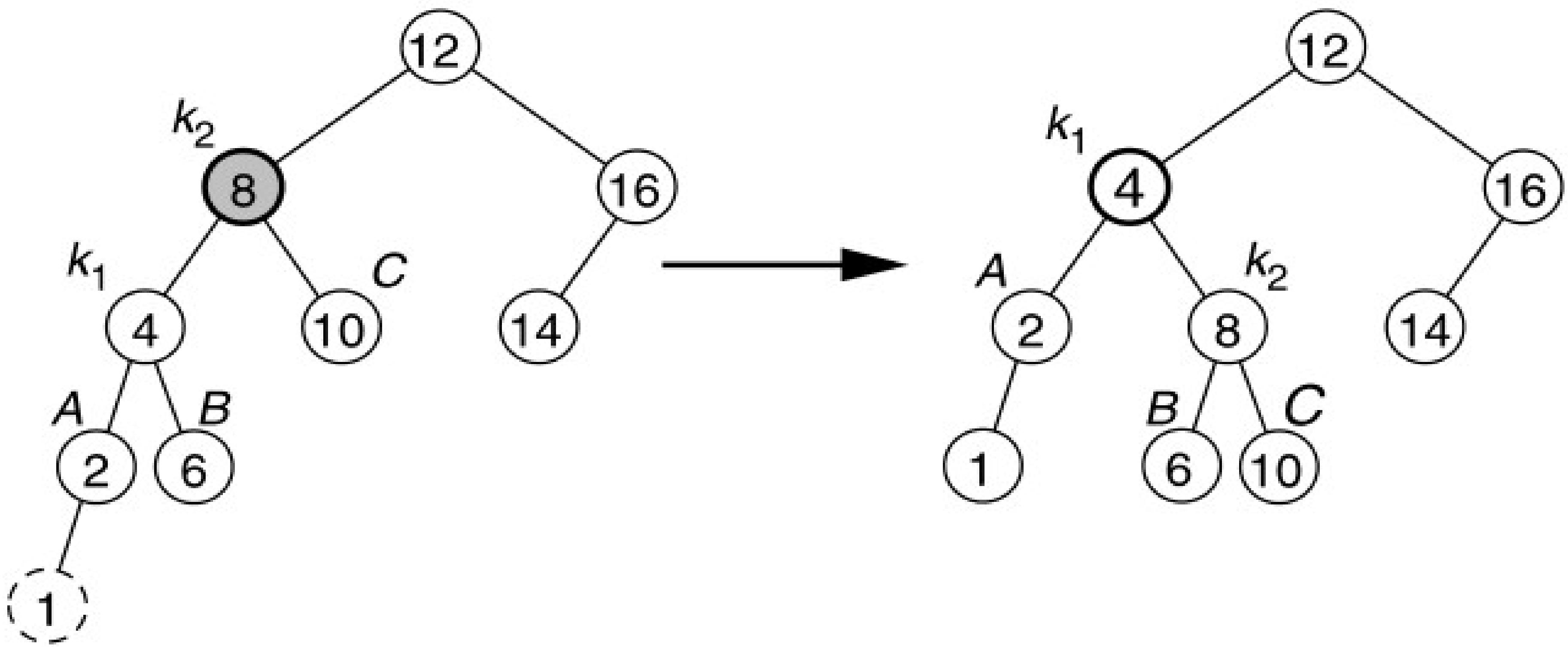
8 AVL



How do we fix this?



That's better!



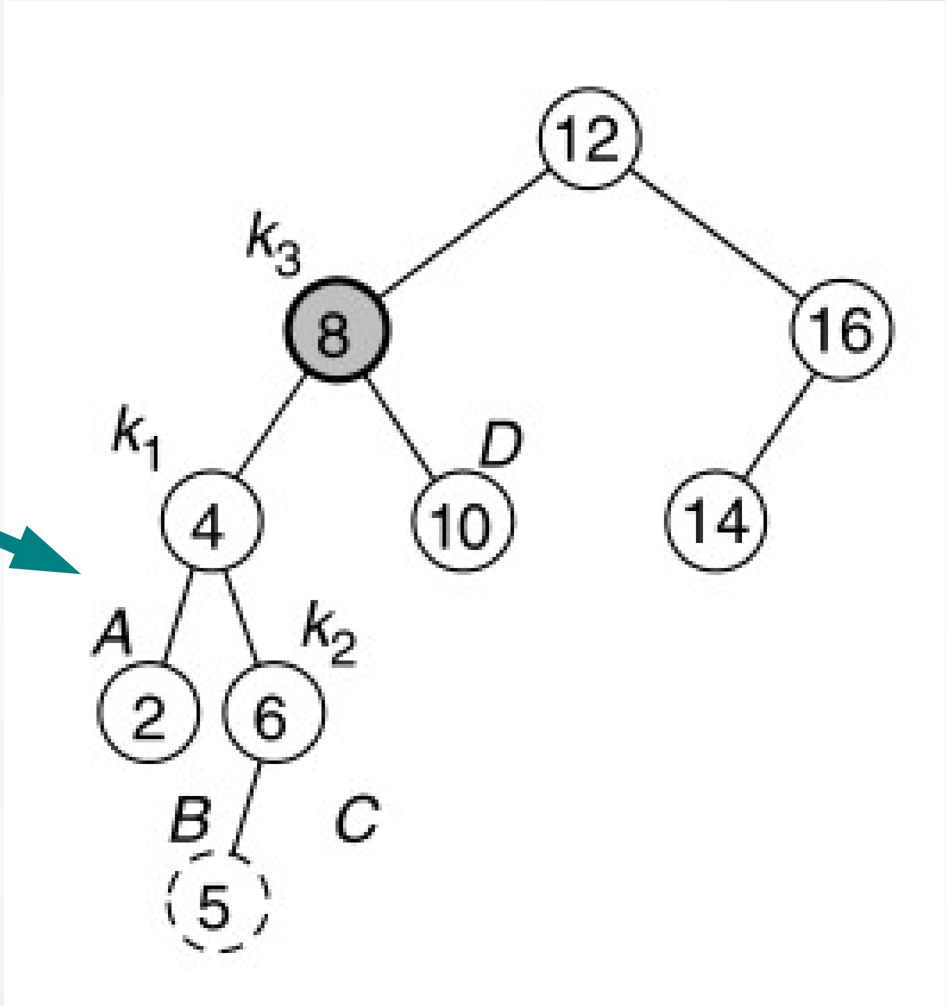
(a) Before rotation

(b) After rotation

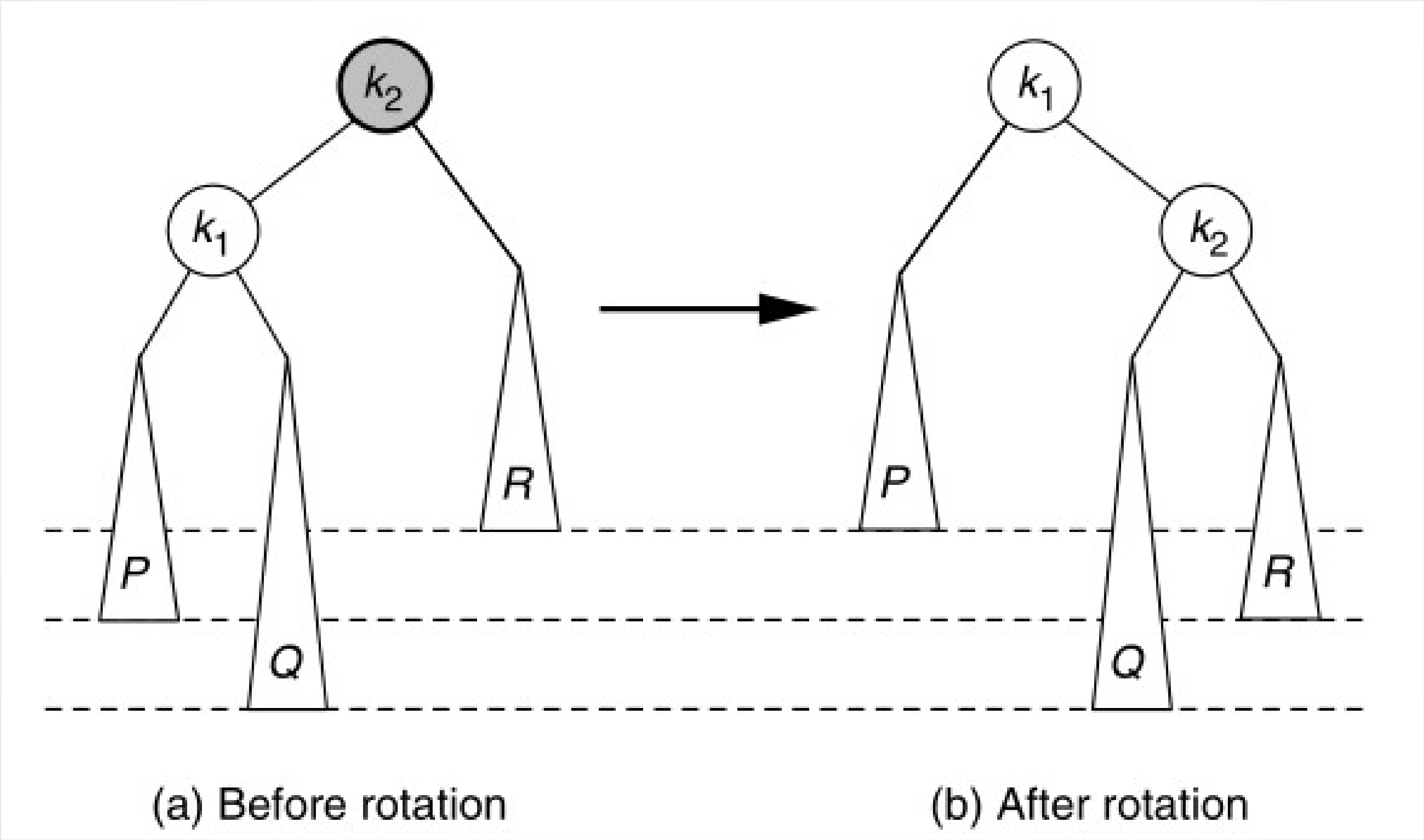
How do we fix this?

Will single rotation fix the following cases?

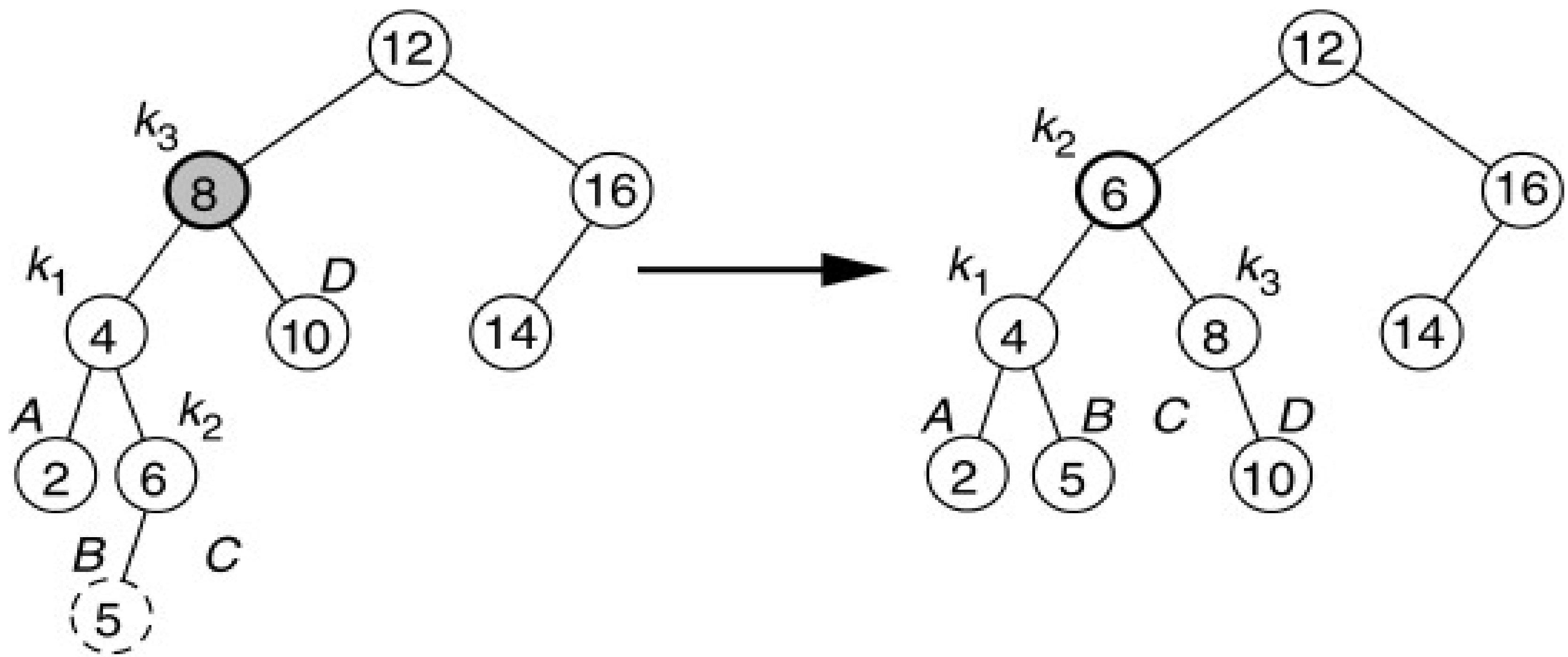
- An insertion in the right subtree of the left child of ***X***
- An insertion in the left subtree of the right child of ***X***



Single Rotation Won't Fix!



Double Rotation



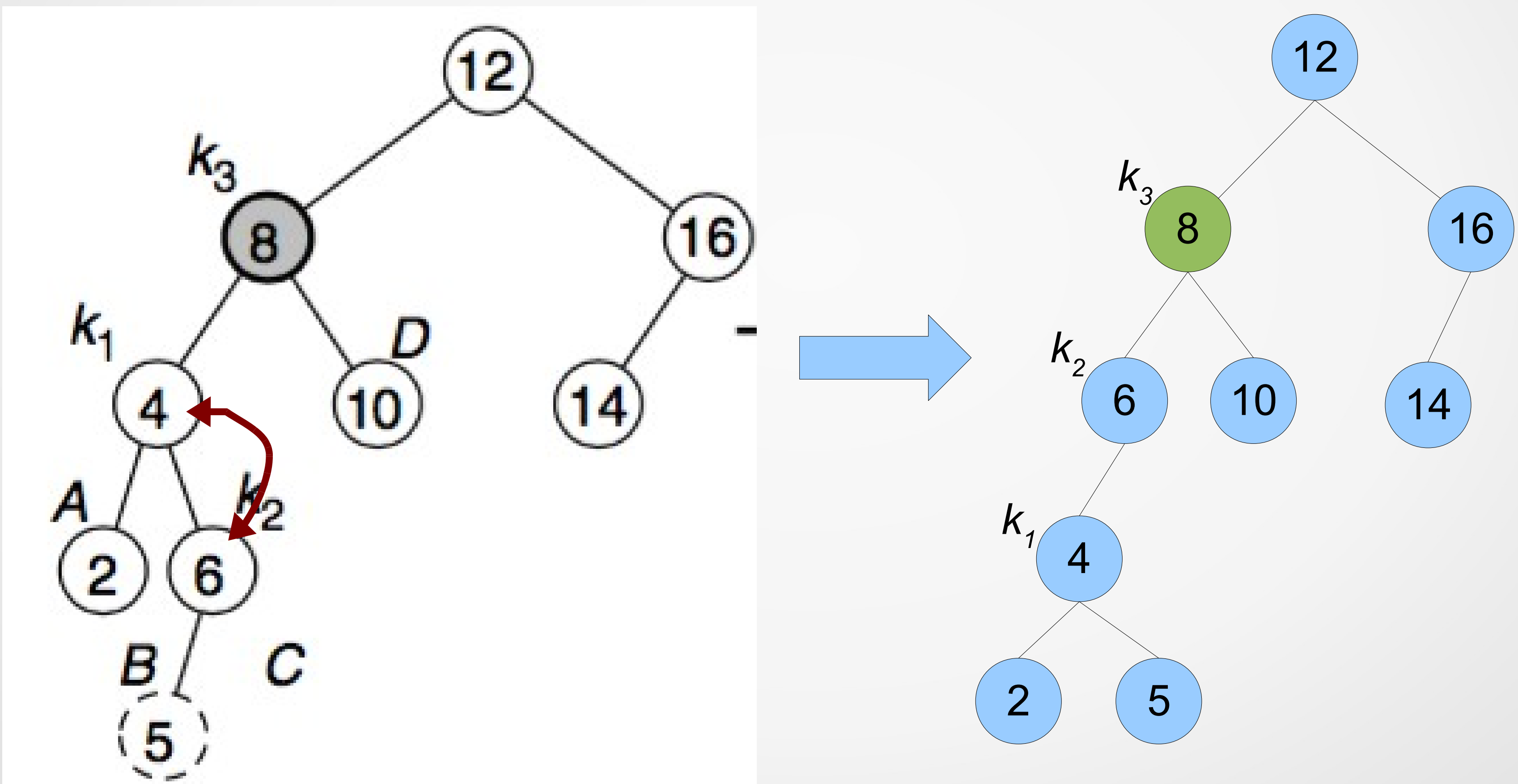
(a) Before rotation

(b) After rotation

- A rotation between X's child and grandchild.
- A rotation between X and its new child.

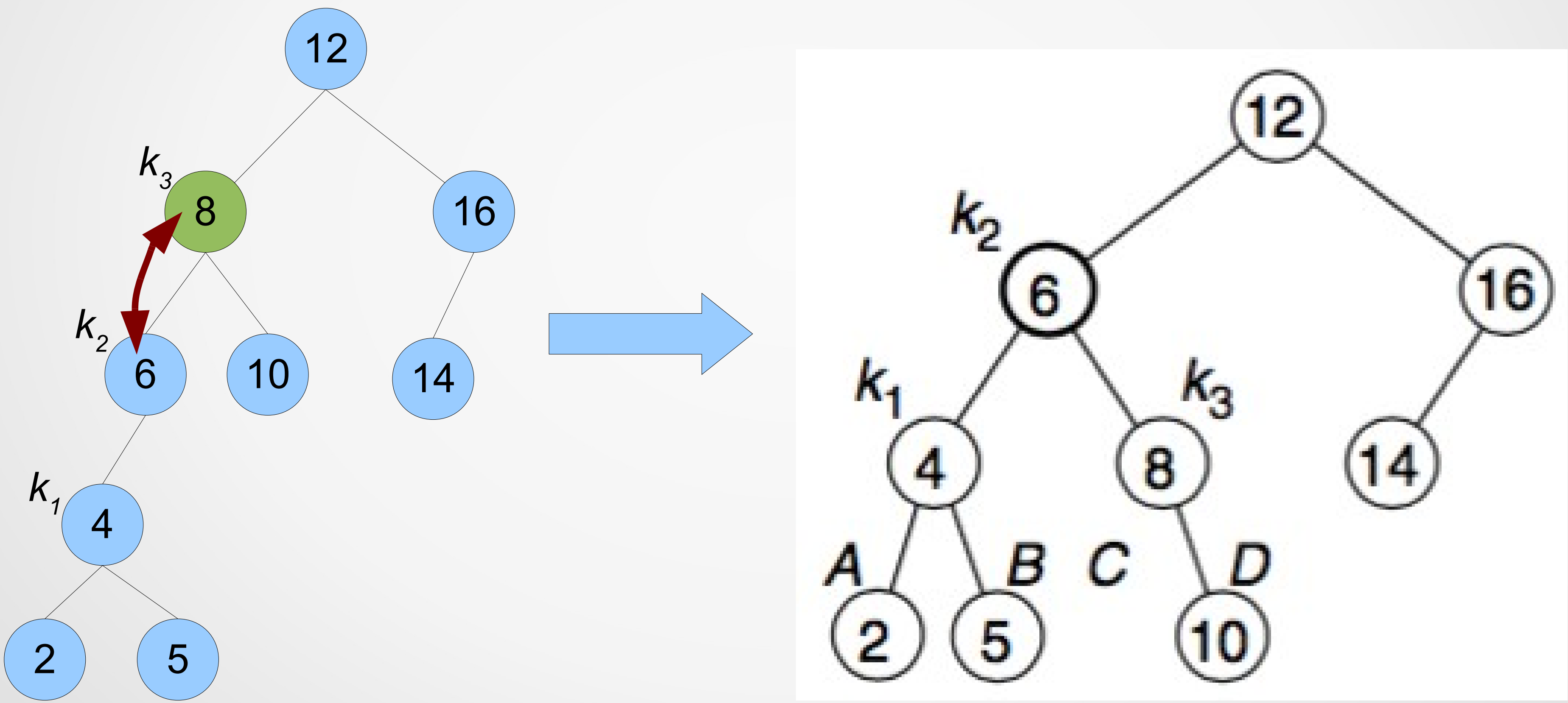
Double Rotation-First Rotation

- A rotation between X's child and grandchild.



Double Rotation-Second Rotation

- A rotation between X and its new child.



Questions?

2-3 Trees

Sedgewick/Wayne 3.3

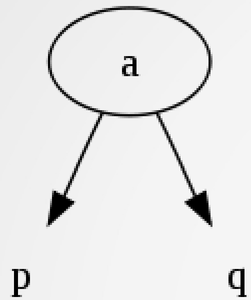
- Invented by John Hopcroft
- In general, nodes are described by the number of children (i.e. number of links)
  - a 2-node has 2 children
  - a 3-node has 3 children
- Every node of BST is a 2-node
  - two links and one key

2-3 Tree Properties

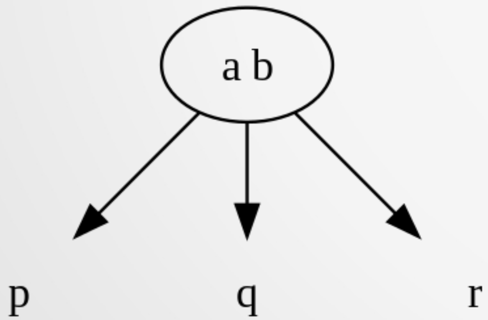
Sedgewick/Wayne 3.3

- Every node of 2-3 tree is a 2 or 3 node
  - every node has between 1 and 2 keys
  - values are stored in sorted order
  - between 2 and 3 children
  - including null links (leaves only)
- 2-3 Trees are perfectly balanced – all null links (those of the leaves) are equal distance to the root

2-3 Trees: Order Property



- 2-node
  - 1 value (a), 2 subtrees (p, q)
  - p < a < q



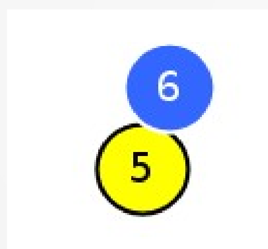
- 3-node
  - 2 values (a, b), 3 subtrees (p, q, r)
  - p < a < q < b < r

images: http://en.wikipedia.org/wiki/2%E2%80%933_tree

2-3 Tree Insert (Put) at Root

Sedgewick/Wayne 3.3

- Case1: Insert into a 2-node (no parent)
  - Simply add key to make it a 3-node

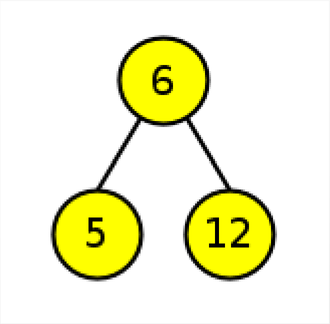


- Big clue: if your search ends at a 2-node, you always make it 3-node.

2-3 Tree Insert (Put) at Root

Sedgewick/Wayne 3.3

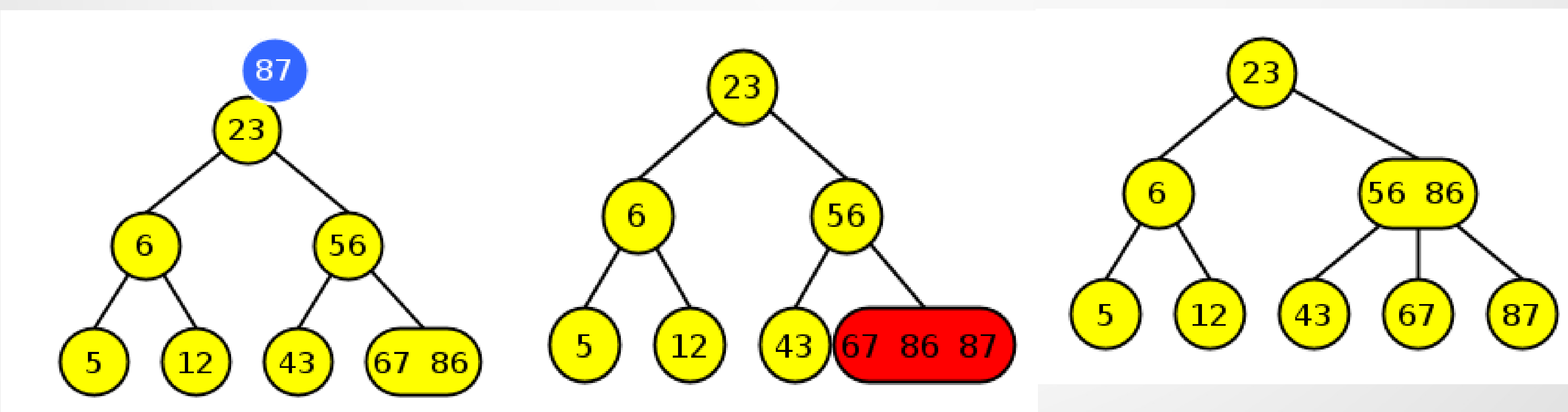
- Case2: Insert into a 3-node (no parent)
  - Temporarily add the key (in order) to make a 4-node
  - Take middle value, create the higher key node
  - Create two new nodes, one with the left key and one with the right key
  - Point the left child of the higher node to left key node and right child to right key node



2-3 Tree Insert (Put) with Parent

Sedgewick/Wayne 3.3

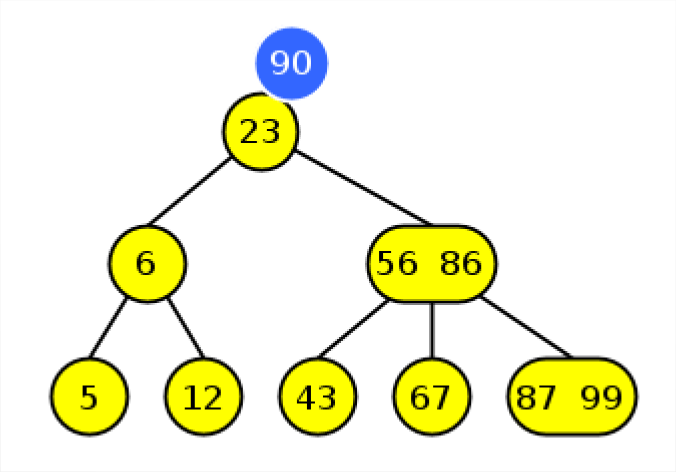
- Case3 Insert into a 3-node with 2-node parent
  - Similar to Case2, push middle key to parent



2-3 Tree Insert (Put) with Parent

Sedgewick/Wayne 3.3

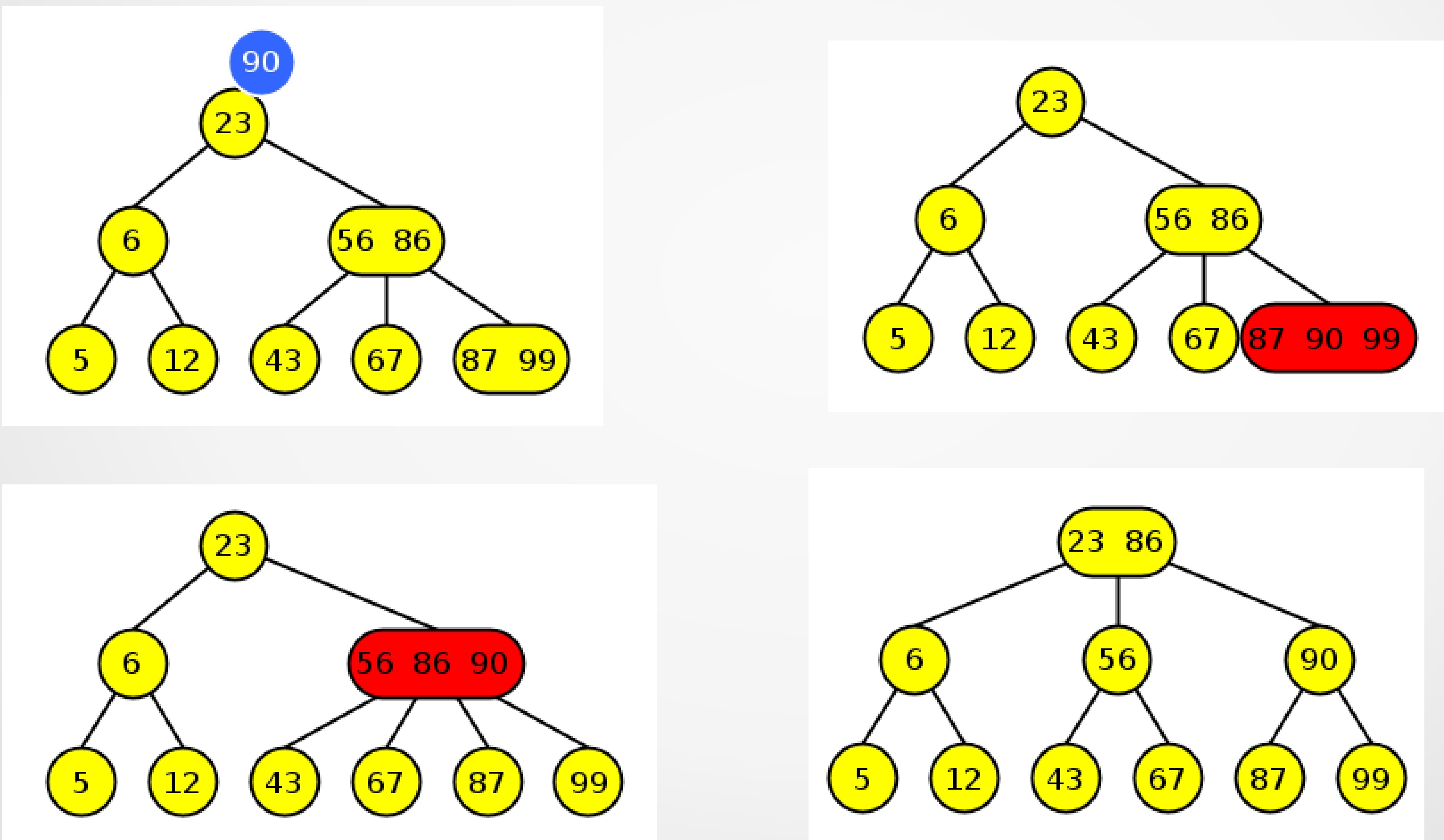
- Case4 Insert into a 3-node with 3-node parent
  - Temporarily create 4-node, split as in Case2
  - Push middle up to parent, parent now 4-node
  - Push middle of parent up, split (harder)
  - Repeat (recursion to root if necessary)



2-3 Trees: Case 4

Sedgewick/Wayne 3.3

- Case4:



2-3 Tree Proposition

Sedgewick/Wayne 3.3

- Proposition: Insert and search operations into a 2-3 tree with N keys is guaranteed to visit at most log(N) nodes.
- Proof: Consider two extremes for N keys (other scenarios fall between), when all nodes are 2-nodes and when all nodes are 3-nodes. Height of a 2-3 tree is between:
  - Lower bound (3-nodes): floor($\log_3(N)$)
  - Upper bound (2-nodes): floor($\log_2(N)$)

Demo

- Gnarley trees.

Red-Black Tree (RBT) Motivation

- Story so far
  - BST cannot guarantee performance for symbol table operations, specifically put, get, delete
  - AVL is balanced and relatively easy with casual implementation, but not as efficient as a 2-3 tree
  - 2-3 tree is (perfectly) balanced and can guarantee performance, more efficient
- So problem solved?
  - Recall one other desirable attribute for an algorithm, it should be easy to implement

Red-Black Tree (RBT) Intuition

- BST easy to implement, 2-3 tree is balanced
  - Can we combine to get best of both?
  - "Yes", store 2-3 tree in a BST structure
- Keep 1-to-1 correspondence between the implemented BST and the logically represented 2-3 tree
  - Put operation involves several different cases
  - Remove/delete also maintain invariant
- Debatable whether RBT is easy, though it is easier than 2-3 tree and performs well

Red-Black Tree as a 2-3-4 Tree

Weiss 19.5

- 2-3-4 trees are also balanced
- Weiss presents RBT using 2-3-4 approach
- Invariants using 2-3-4 approach

  1) Every node colored either red or black

  2) Root is black

  3) If node is red, its children must be black

  4) Every path from a node to a null link must contain the same number of black nodes
- Involves several rotation cases

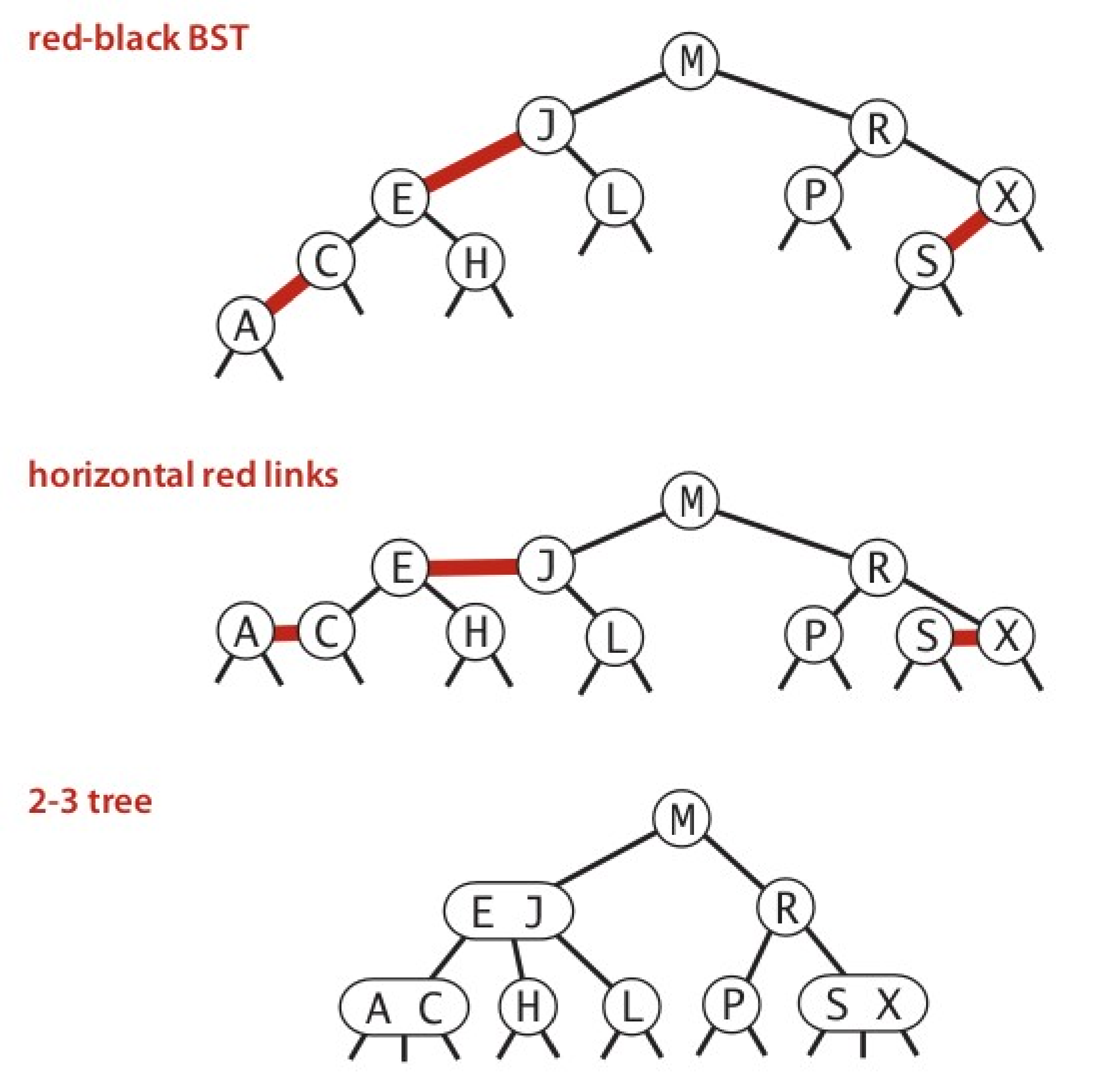
Red-Black Tree as a 2-3 Tree

Sedgewick 3.3

- Sedgewick presents RBT using 2-3 approach
- Invariants using 2-3 approach
    1) Red links lean left ("left leaning RBT")
    2) Root is black
    3) No node has two red links connected to it
    4) Tree has perfect black balance: every path from the root to a null link has the same number of black links
- Also involves several rotation cases. We will use this approach for RBT.

Red-Black as 2-3 Tree and BST

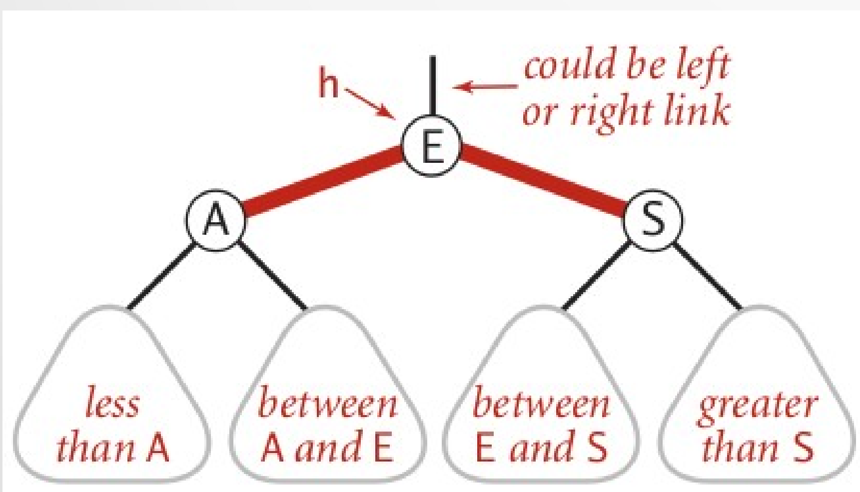
Sedgewick/Wayne 3.3

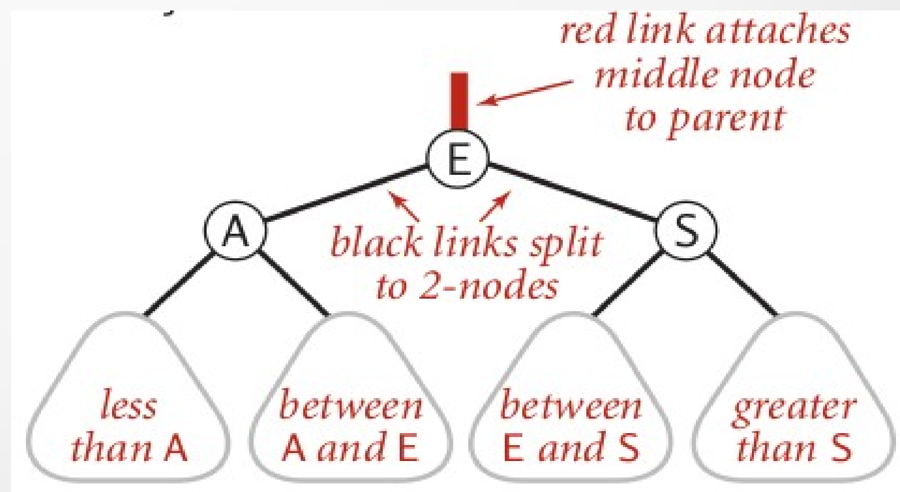


1-1 correspondence between red-black BSTs and 2-3 trees

RBT: Maintaining Search Order and Perfect Black Balance

- Insert nodes at bottom with red link
- Left rotations and right rotation operations
- Flip color operation
  - Flip children (i.e. red to black)
  - Flip parent from black to red

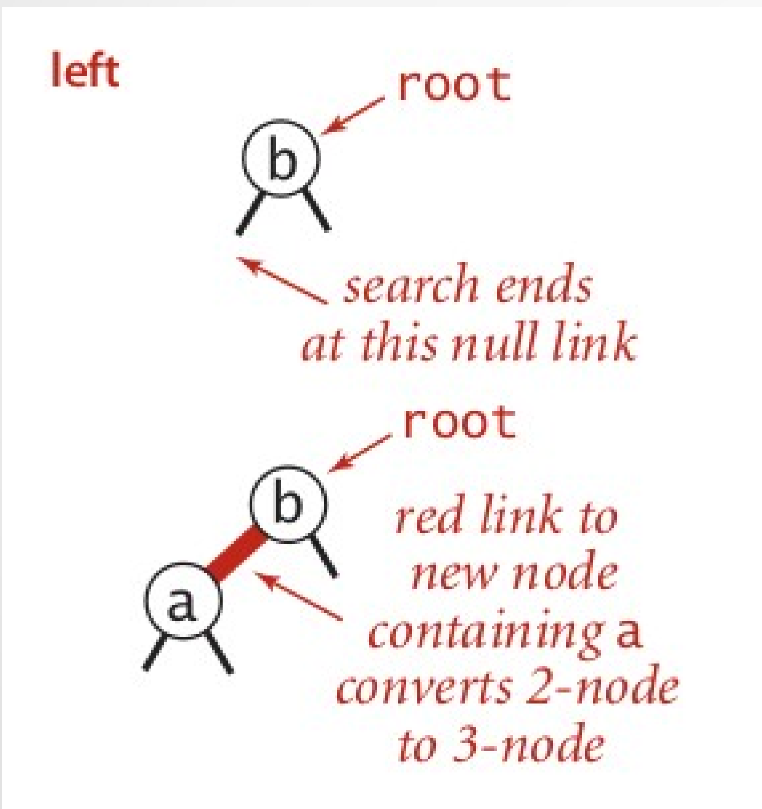




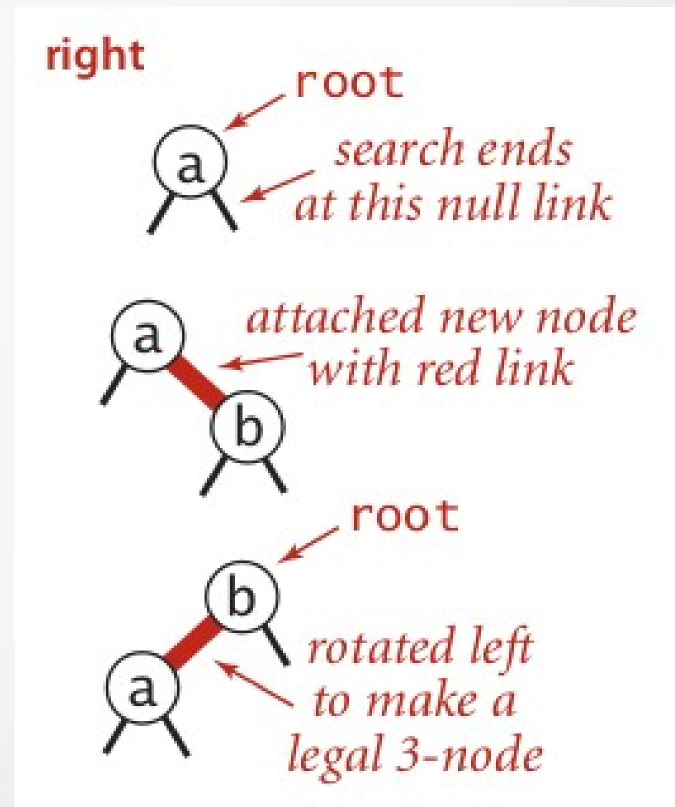
RBT: Insert into 2-node

- Insert into a single 2-node → root

Left insert



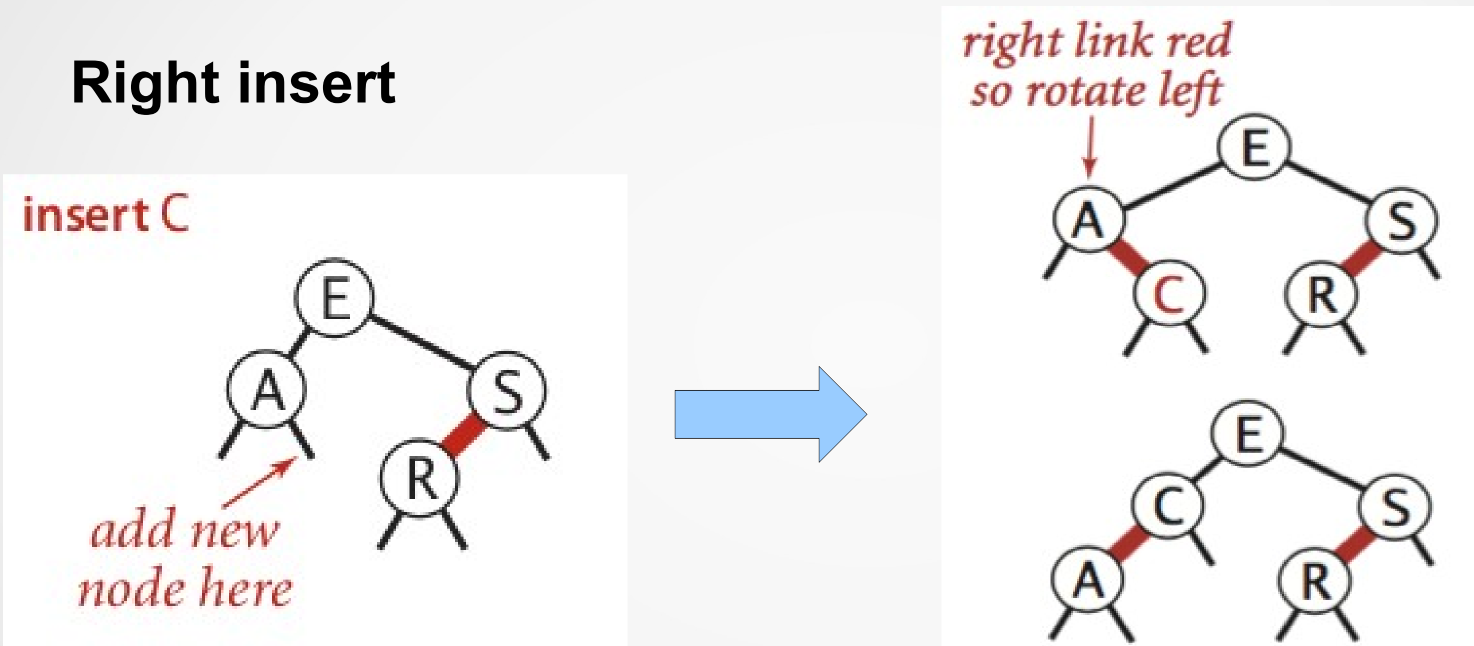
Right insert



RBT: Insert into 2-node

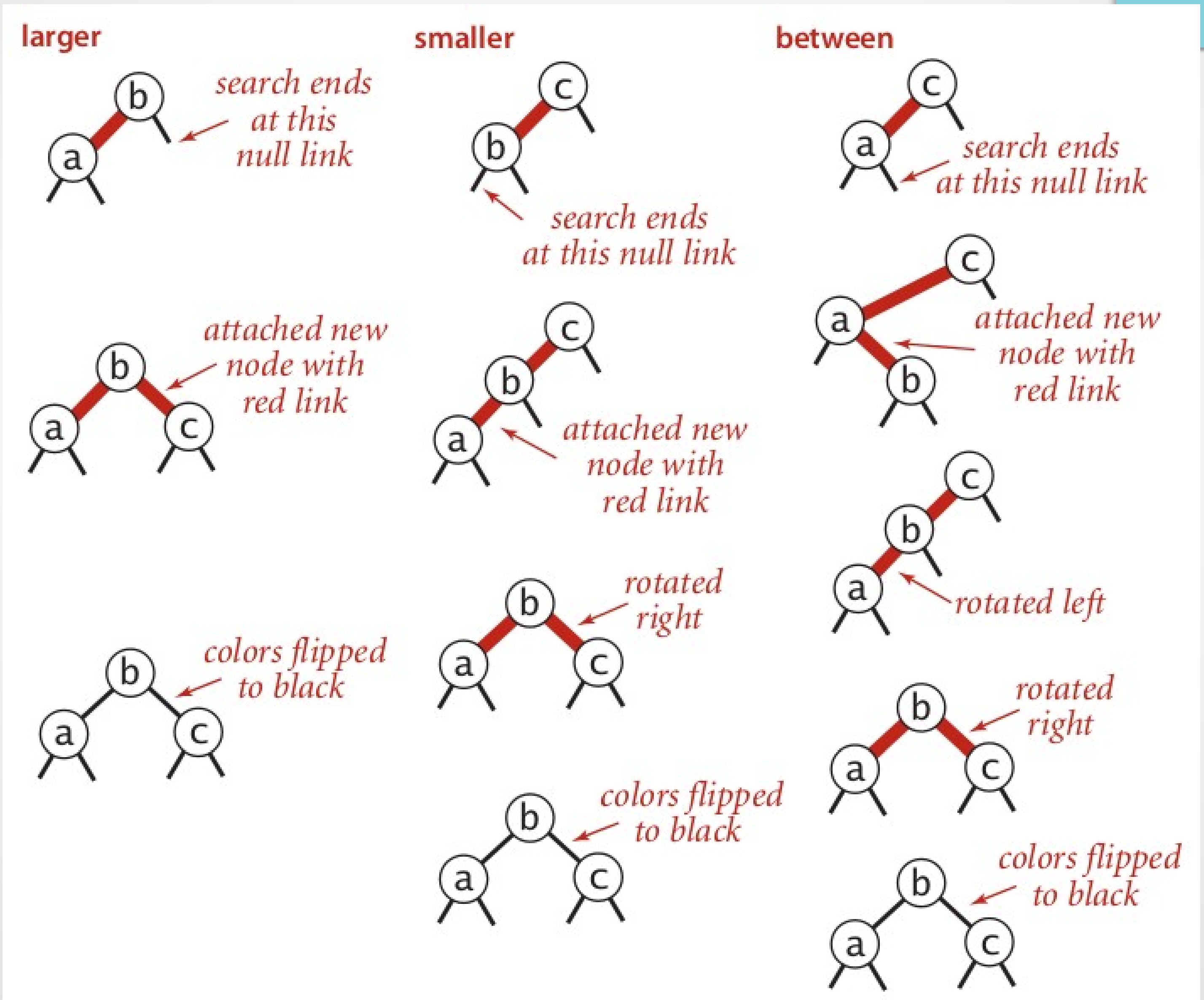
- Insert into 2-node at bottom.

Right insert



- Left insert does not require a rotation.

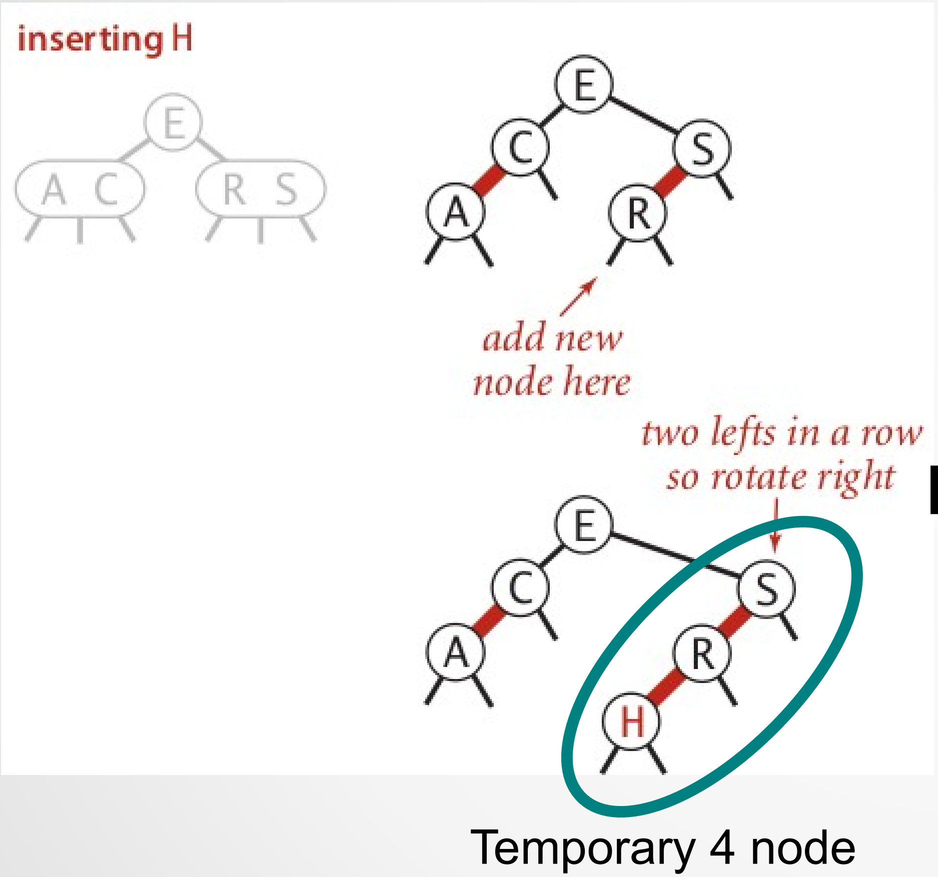
RBT: Insert into a single 3-node

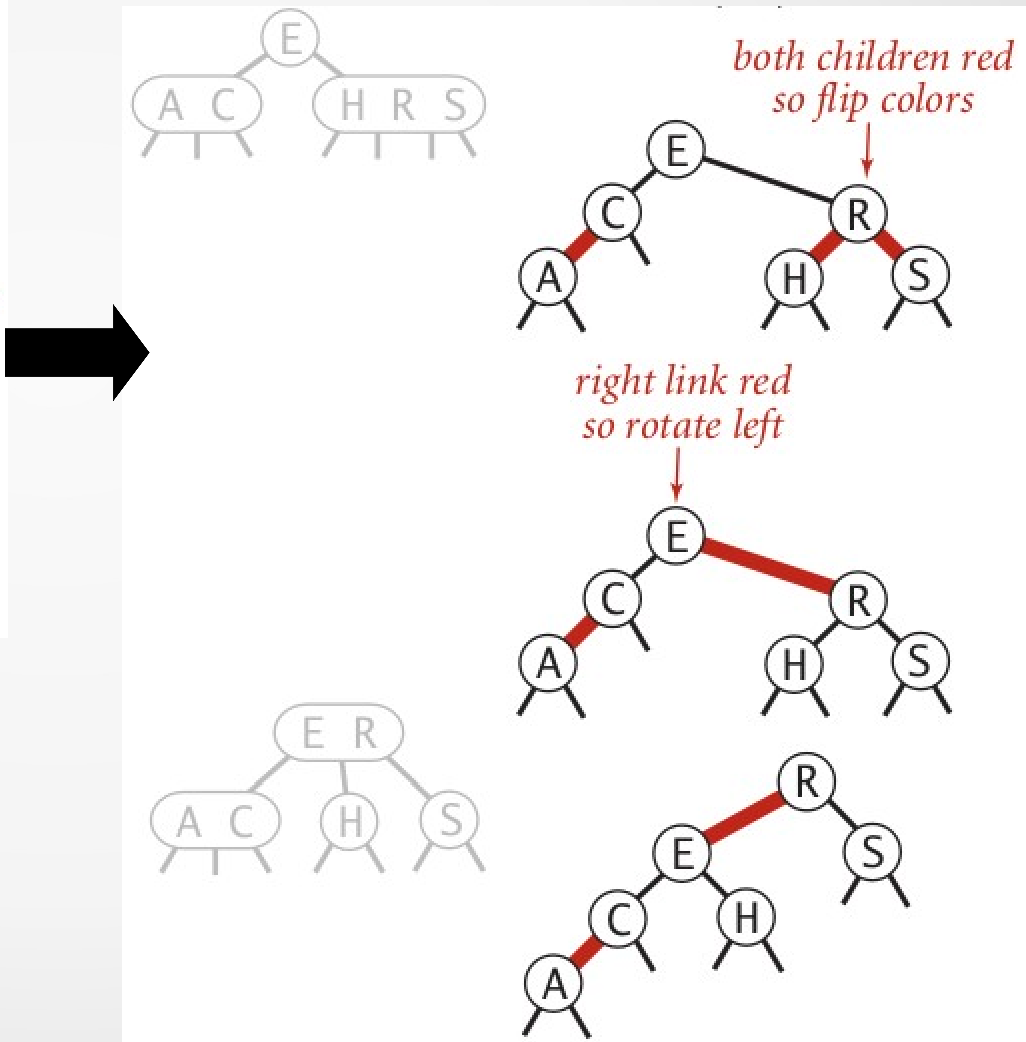


RBT: Root Color

- Root should always be black. If red, means it is part of 3 node (i.e. not the root).
- Color flips can change the root's color to be red. In this exceptional case, the root should be changed to black.
  - When the root is changed from red to black, it means that the black height has increased by one

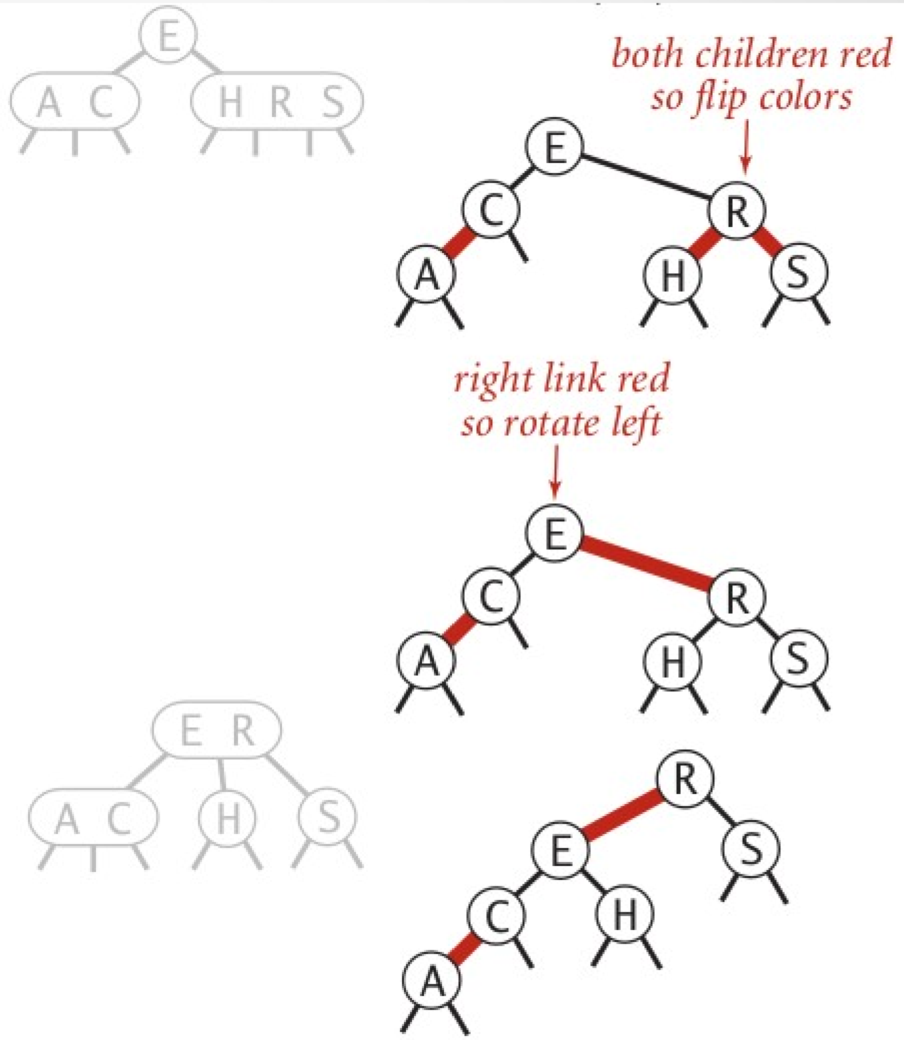
RBT: Insert into 3-node at bottom





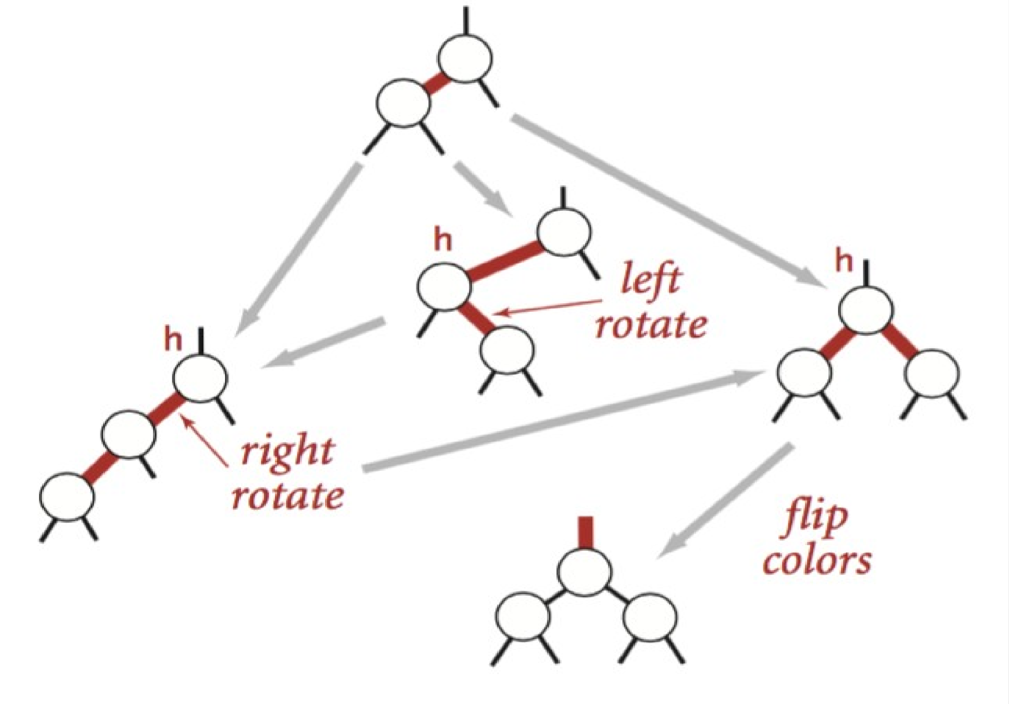
RBT: Passing Red Link Up Tree

- Rotation transformations are local but may result in a red link being passed up to its parent
- Can treat recursively
  - Think of it as adding a new node/red link.
- Repeat until parent is a 2 node or the root



Left Leaning RBT: Summary

- Can maintain RBT as 2-3/BST recursively by the following rules:
    1) If right child is red and left child is black, then rotate left
    2) If left child is red and its left child is red, then rotate right
    3) If both left and right child are red, flip colors
- RBT a 2-3 Tree and BST. The get operation is same as BST.
    - One of the main reasons for all this trouble...



Symbol Table Summary

- Generally use hash table unless guaranteed performance or need ordered operations

Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	N	N	lg N	lg N	Yes	Easy
AVL	lg N	lg N	lg N	lg N	Yes	Easy
Red-Black	**lg N**	**lg N**	**lg N**	**lg N**	**Yes**	**Often Used**
HT Chaining	N	N	N / M	N / M	No	Often Used
HT Probing	N	N	1	1	No	

* Depending on variant, will assume O(lg(N)) ~ O(1)

Questions?

Huge Data Sets

- Up to now entire data structure fits in memory
  - If it's too big it won't fit into memory...
  - ... so use a tree to break it up, store on disk
- Now, we have to perform in-memory instructions intermixed with disk accesses
  - Can ~25 million instructions in one second
  - Can ~ 6 disk accesses in one second
    - Creates a big bottleneck.
- Given 1 million records, assuming disk access required, balanced BST
  - Requires 20 accesses, about 3.5 seconds

Partition Keys / Values

- Only need to keep keys (or subset of keys) in memory
- Data stored on disk and accessed when needed
  - Accessing disk performs best when reading/writing disk blocks (sometimes referred to as pages) at a time.

B Tree Motivation

- Problem of data too big for memory was solved by breaking data into tree and storing much of the data on disk.
  - Created new problem. Now have to access disk
  - Asymptotically nothing has changed but the constants, even for lg(N) operations is objectionable
- How can we reduce the constant (said another way, how can we reduce the height of the tree)?

B Tree Motivation

- Organize data within the tree to correspond to pages on disk.
  - One tree node represents one disk page.
  - Read one disk page at a time.
  - Maximum degree chosen depends on disk characteristics and problem description.
- If we read 1000 keys, but look at 10, is that good?
  - Yes, the biggest latency is before we get the first bit, consecutive reads are relatively faster.
  - We end up making only 1/10 as many disk reads.

As taken from David Scot Taylor’s lecture slides.

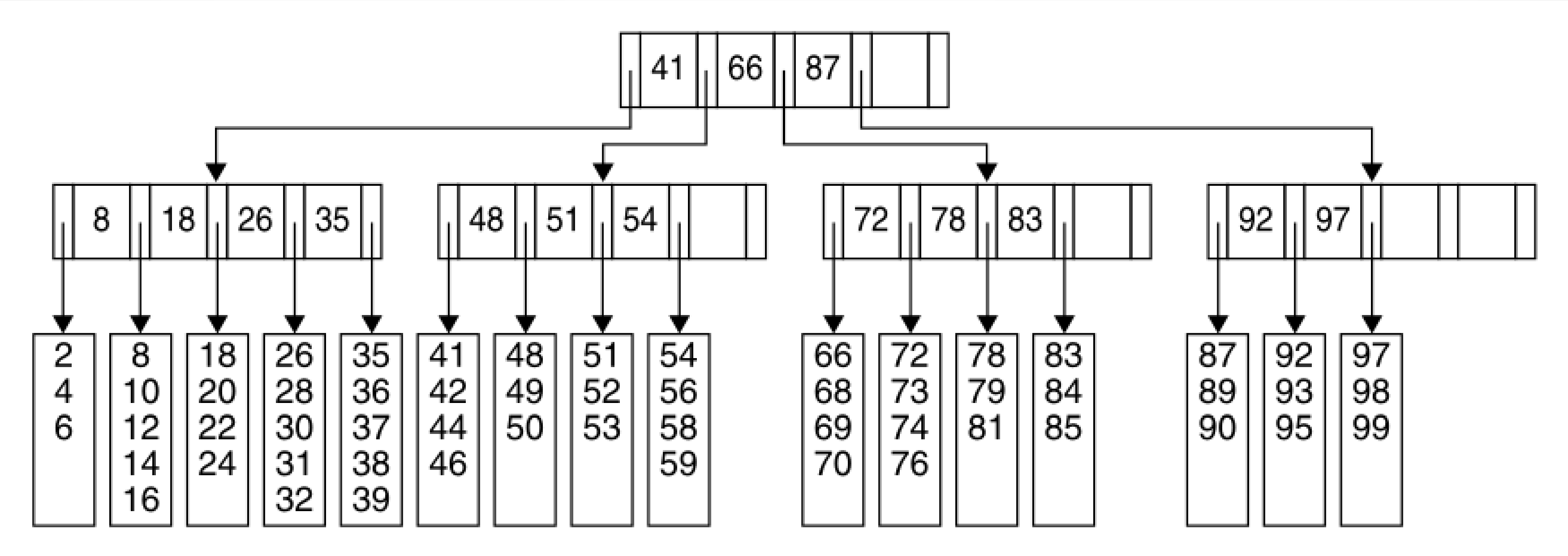
B Tree versus B+ Trees

- B Trees
  - internal nodes store keys and values
  - Think of it as 2-3-....-many nodes
- B+ same as B Trees, except
  - internal nodes do not store values
    - only store keys, i.e. stores index like book
  - sibling leaves may have a pointer to link them in order
- Weiss 19.8 presents “B Tree” but is technically a B+ Tree. Will use term “B Tree” to refer to “B+ Tree” and only distinguish if necessary.

B Trees

- Commonly taught, commonly used, easy to implement
- Basic idea:
  - Tree + List = B-Tree
  - We want a really big list...
    - but if it's too big it won't fit into memory...
    - ... so use a tree to break it up
- When to fix balance?
  - inserting/deleting

2 for 1: It's a list! In a tree!

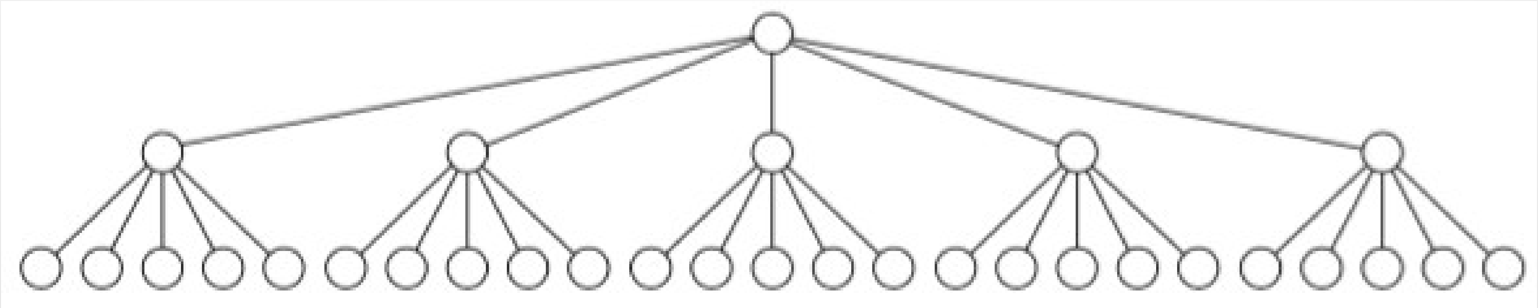


Learning B Trees

- 2-3 Trees
  - Restricted, simpler version of B Trees
- B Trees
  - General form of 2-3 trees
- B+ Trees
  - Same idea, different internal nodes

B Tree Parameters

- Usually referred to by the M value
  - so an order 5 B-tree, has M=5
- Additional Note: a 2-3 tree is a B Tree with:
  - M = 3
  - L = 2 → number of data items

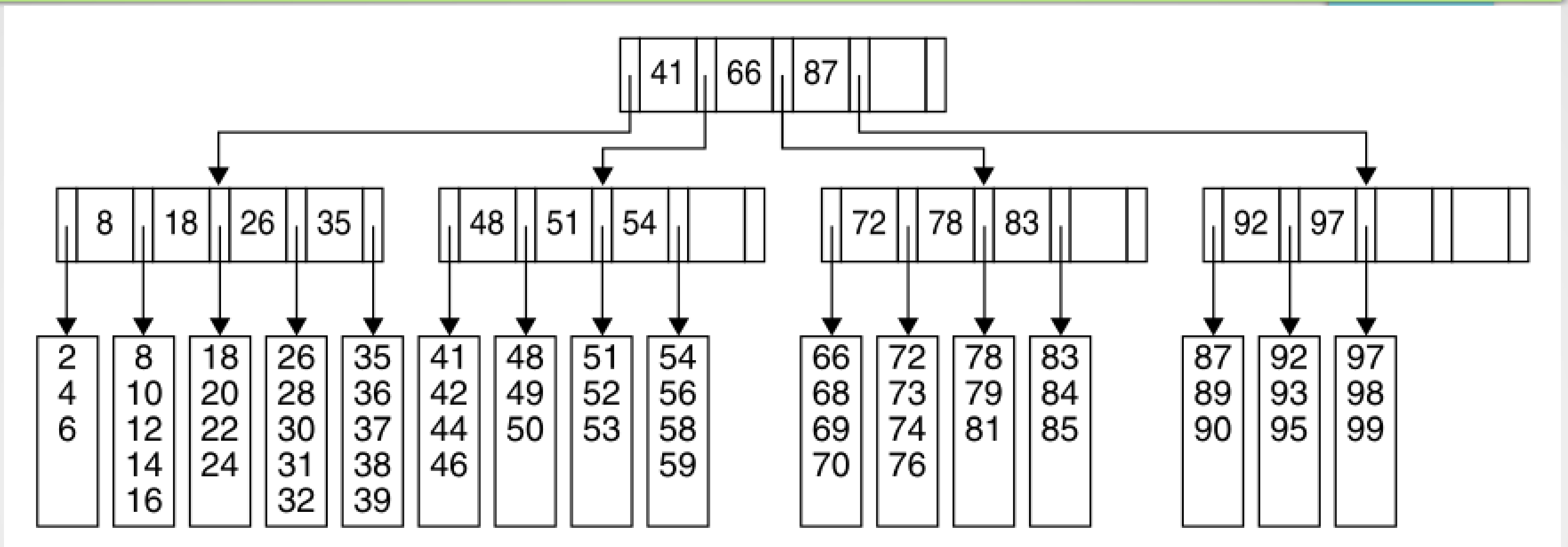


M=5

B Tree Properties

- B tree of order M is an M-ary tree, invariants:
  1) Data Items are stored as leaves
  2) Non-leaf node store M-1 keys to guide search
  3) Root is leaf or between 2 and M children
  4) Non-leaf nodes (other than root) have between ceil(M/2) and M children
  5) All leaves are at same depth and have ceil(L/2) and L data items
- Note: Invariants keep tree from becoming degenerate

B Tree Properties

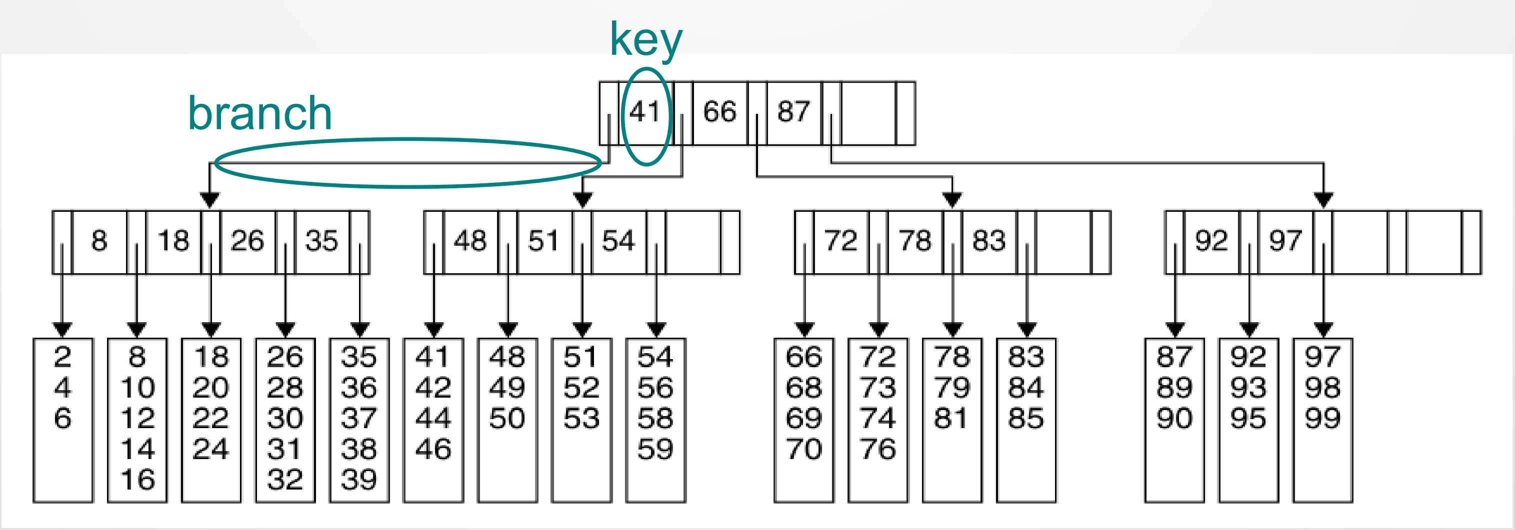


- A tree of order 5
- All nonleaf nodes have between 3 and 5 children
  - Which also means between 2 and 4 keys
- Each leaf node has between 3 and 5 data items
  - ceil(L/2) <= # of data items <= L

B Tree Example Calculations 1/2

Weiss, Page 758

- Choose the maximum M and L that allow internal and external nodes to fit in one disk block. Example, store 500,000,000 records:
  - Assume block size is 4096 bytes, each key uses 8 bytes, each branch uses 4 bytes, each data record uses 32 bytes.



B Tree Example Calculations 1/2

Weiss, Page 758

- Block size=4096 bytes, Key=8 bytes, Brach=4 bytes, Data record=32 bytes
- M=?
  - M is for non-leaf nodes
    - M-1 keys → 8(M-1)
    - M branches → 4M
    - Adds up to 12M-8 bytes
      - 12M-8=4096 → M=342
  - L is for leaf nodes
    - Will store only data records.
      - 4096/32 → L=128

B Tree Example Calculations 2/2

Weiss, Page 758

- Internal nodes branch at least ceil(342/2)=172
- Log_{172}(500 million) = 3.89, max tree height 4
- At most 500 million / 64 = 7,812,500 leaves
- In general:
  - Worse case number of accesses ~ $\log_{M/2}(N)$
  - At most leaves is = N / ceil(L / 2)

Profound Implication

https://en.wikipedia.org/wiki/Observable_universe

- The number of atoms on the observable universe
  - Approximately 10^{80} (roughly 2^{266})
- B Tree with M=2048 nodes
  - Max height ~ $\log_{1024}(N) = \log_{1024}(2^{266})$

    $\log_2(2^{266}) / \log_2(1024) \sim 26.6$
- Assuming you had the disk storage, did you anticipate that in two dozen steps you could search for any atom in the universe?

B Tree vs other Balanced Trees

- B Tree reduces height of tree and therefore potentially number of disk accesses
- However, if all data can fit in-memory, then other balanced trees (e.g. RBT) should be used
  - Generally constants to traverse/insert into a RBT are better than B Tree

Questions?

PA8

- Extend your binary search tree into a balanced AVL tree

Free Question Time!