Set and Map ADTs: Left-Leaning Red-Black Trees

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements

- Case-vs-Asymptotic Analysis handout released!
  - https://courses.cs.washington.edu/courses/cse373/20wi/files/clarity_case_asymp.pdf
- Workshop Survey released (see Piazza)
  - Workshop Friday @ 11:30am, CSE 203

Lecture Outline

- **Review: 2-3 Trees and BSTs**

- Left-Leaning Red-Black Trees
  - Insertion

- Other Balanced BSTs

Review: BSTs and B-Trees

- **Search Trees** have great runtimes most of the time
  - But they struggle with sorted (or mostly-sorted) input
  - Must bound the height if we need runtime guarantees

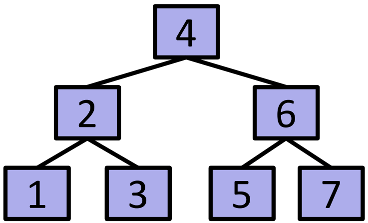
- **Plain BSTs**: simple to reason about/implement. A good starting point

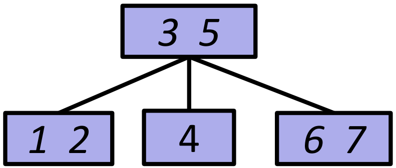
- **B-Trees** are a *Search Tree variant* that binds the height to Θ(log N) by only allowing the tree to grow from its root
  - A good choice for a Map and/or Set implementation

	LinkedList Map, Worst Case	BST Map, Worst Case	B-Tree Map, Worst Case
Find	Θ(N)	h = Θ(N)	Θ(log N)
Add	Θ(N)	h = Θ(N)	Θ(log N)
Remove	Θ(N)	h = Θ(N)	Θ(log N)

Review: 2-3 Trees

- 2-3 Trees are a specific type of B-Tree (with L=3)
- Its invariants are the same as a B-Tree's:
  1. All leaves must be the same depth from the root
  2. A non-leaf node with k keys must have exactly k + 1 non-null children
- Example 2-3 trees:

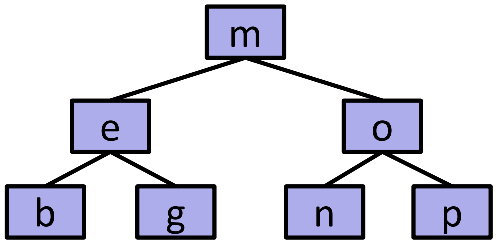


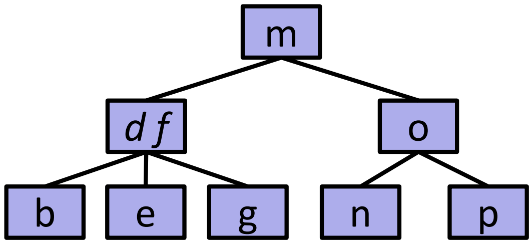


Improving Search Trees

- **Binary Search Trees (BST)**
  - Can balance a BST with rotation, but we have no fast algorithm to do so

- **2-3 Trees**
  - Balanced by construction: no rotations required
  - Tree will split nodes as needed, but the algorithm is complicated

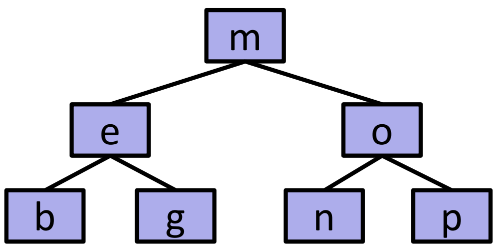




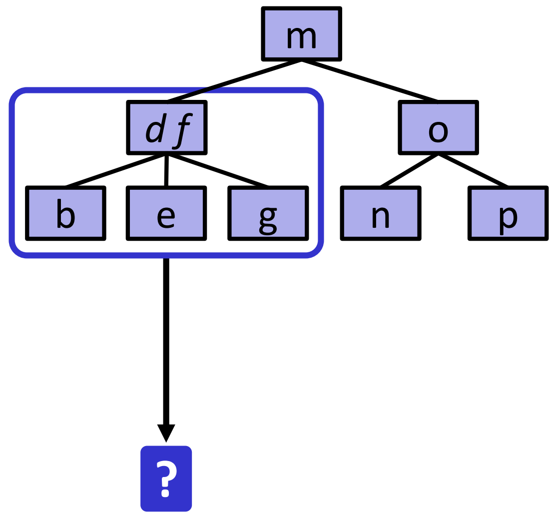
Can we get the best of both worlds: a BST with the functionality of a 2-3 tree?

Converting 2-3 Tree to BST

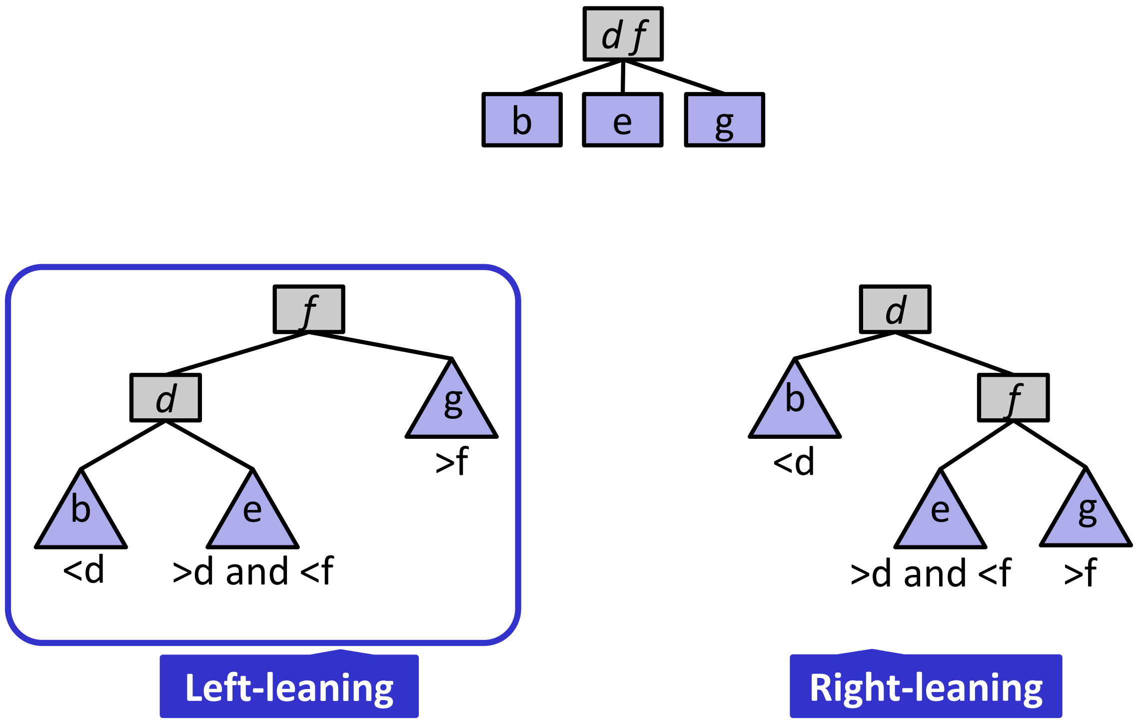
- 2-3 trees with only **2-nodes** (2 children) are already regular binary search trees



- How can we represent **3-nodes** as a BST?

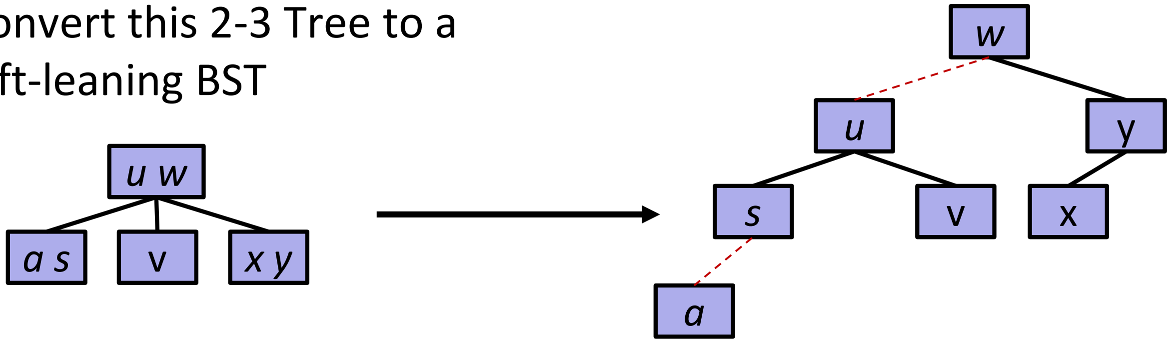


Splitting 3-nodes

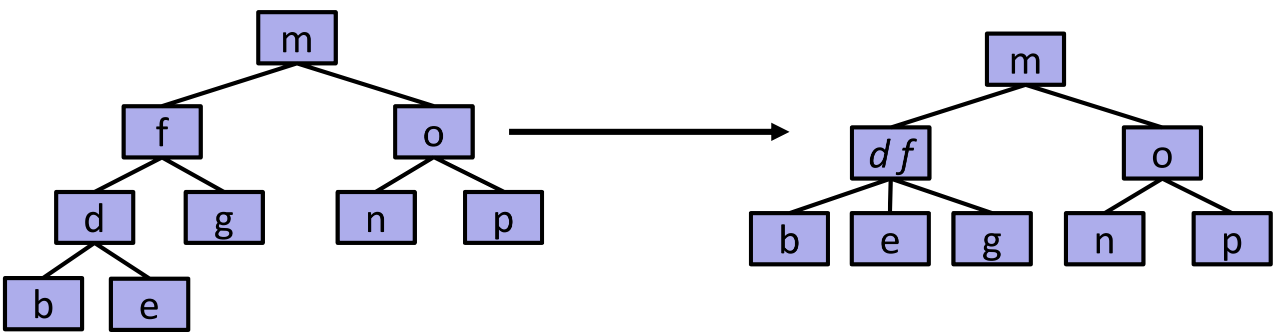


Practice:

- Convert this 2-3 Tree to a left-leaning BST



- Convert this left-leaning BST to a 2-3 Tree

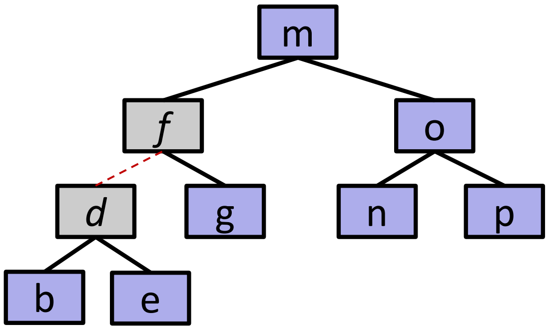


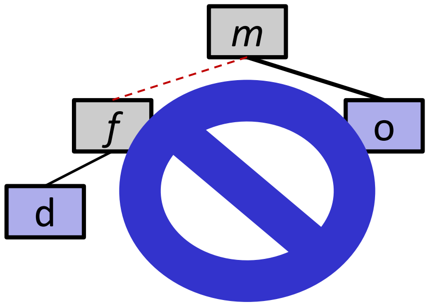
Lecture Outline

- Review: 2-3 Trees and BSTs
- **Left-Leaning Red-Black Trees**
  - Insertion
- Other Balanced BSTs

Left-Leaning Red-Black Tree

- **Left-Leaning Red-Black (LLRB) Tree** is a BST variant with the following additional invariants:
  1. Every root-to-bottom* path has the same number of black edges
  2. Red edges must lean left
  3. No node has two red edges connected to it, either above/below or left/right

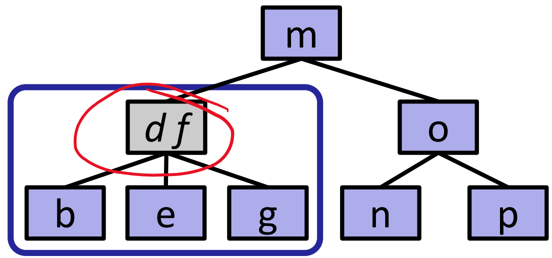




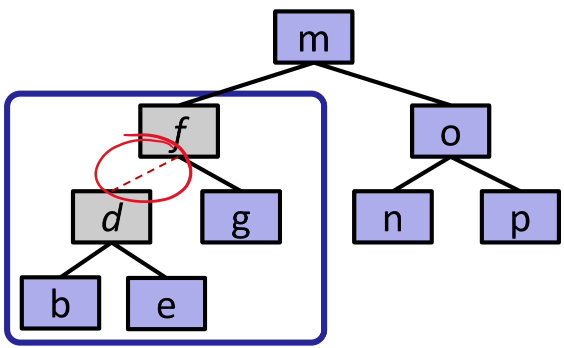
** "bottom" refers to single-children nodes and leaf nodes (which have no children)*

Left-Leaning Red-Black Tree == 2-3 Tree

- There is a *1-1 correspondence (bijection)* between 2-3 trees and Left-Leaning Red-Black trees
  - 2-nodes are the same in both trees
  - 3-nodes are connected by a red link



- Left-Leaning Red-Black (LLRB) Tree
  - Identify the link connecting the left-items in a 3-node and color it **red**



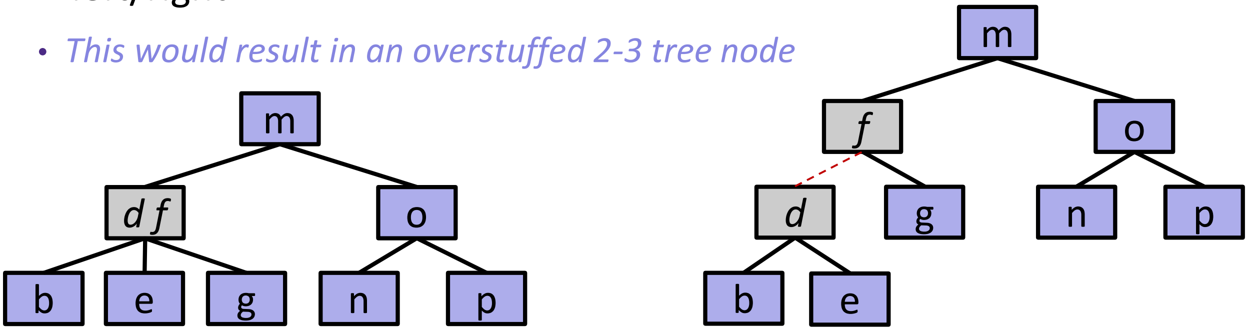
Left-Leaning Red-Black Tree == 2-3 Tree

- 2-3 Trees (more generally: B-Trees) are *balanced search trees:*
  - height is in Θ(log N)
  - find, insert, and remove are also in Θ(log N)

- Since any LLRB Tree can be a 2-3 Tree:
  - height is in Θ(log N)
  - find, insert, and remove are also in Θ(log N)

Left-Leaning Red-Black Tree

- **Left-Leaning Red-Black (LLRB) Tree** is a BST variant with the following additional invariants:
  1. Every root-to-bottom* path has the same number of black edges
     - *All 2-3 tree leaf nodes are the same depth from the root*
  2. Red edges lean left
     - *We arbitrarily choose left-leaning, so we need to stick with it*
  3. No node has two red edges connected to it, either above/below or left/right
     - *This would result in an overstuffed 2-3 tree node*

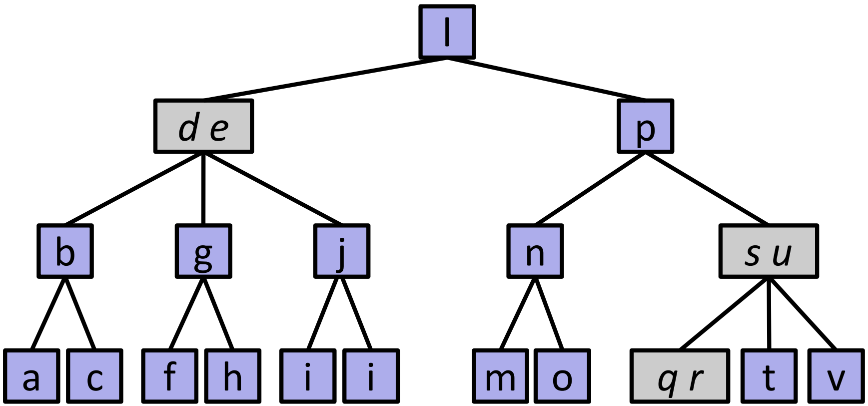


Poll Everywhere

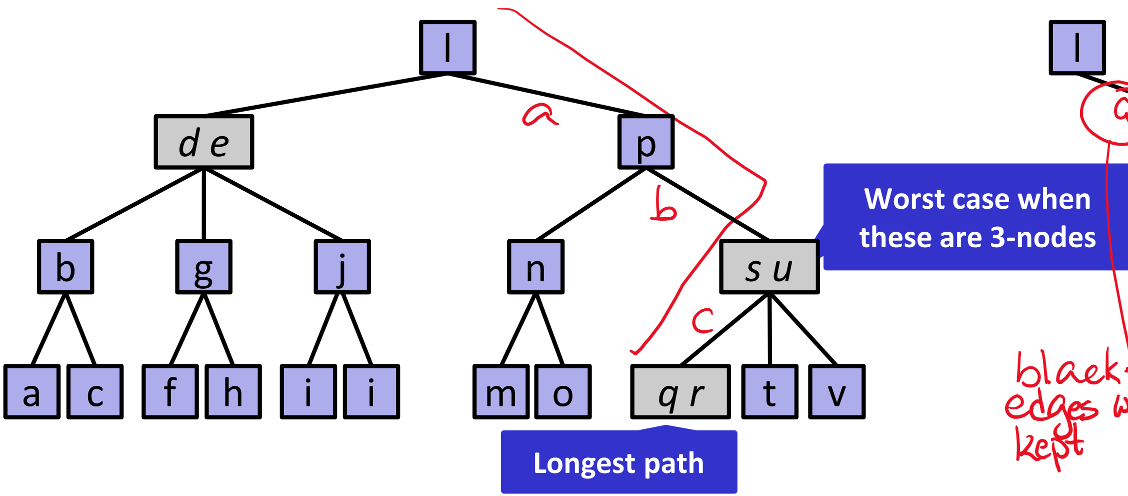
pollev.com/uwcse373

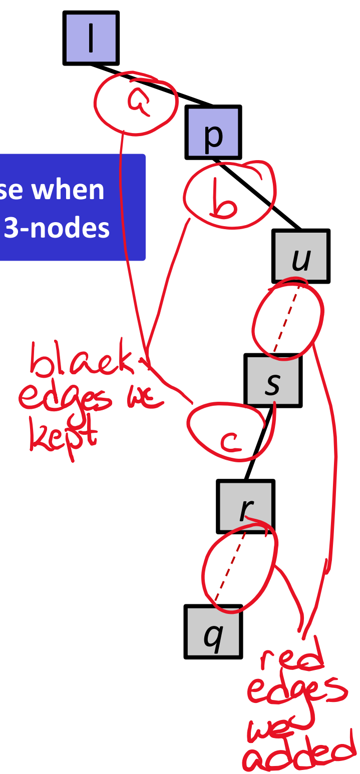
- What's the height of the corresponding Left-Leaning Red-Black tree?

A. 3
B. 4
C. 5
D. 6
E. 7
F. I'm not sure …



Height of a Left-Leaning Red-Black Tree





Given a 2-3 tree of height H, the corresponding LLRB tree has height: H + max(3-Nodes) = (H) + (H+1) ∈ Θ(log N)

Black Edges **Red Edges**

Lecture Outline

- Review: 2-3 Trees and BSTs
- Left-Leaning Red-Black Trees
  - **Insertion**

Pretend it's a 2-3 tree

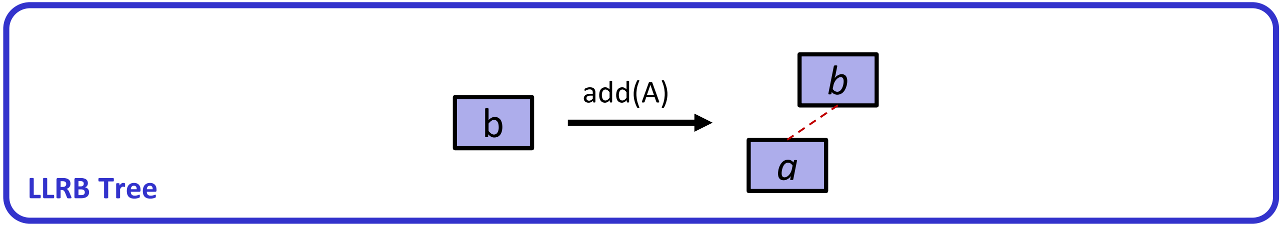
- Other Balanced BSTs

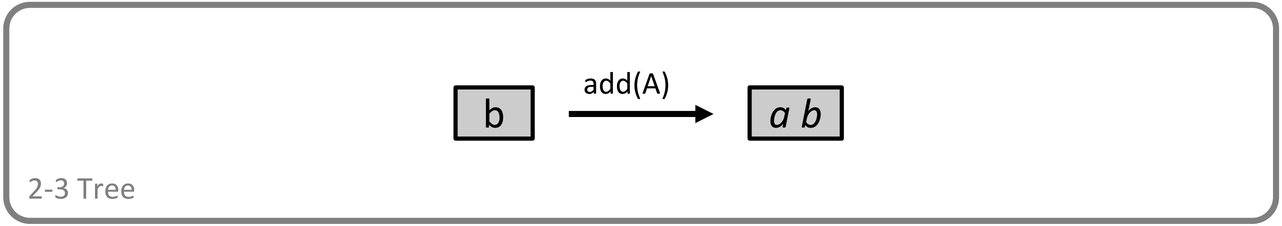
LLRB Tree Insertion: Overall Approach

- Insert nodes using the "Plain BST" algorithm, but join the new node to its parent with a red edge
  - This is analogous to how a 2-3 tree insertion always overstuffs a leaf
- If this results in an invalid Left-Leaning Red-Black Tree, repair
  - This is analogous to repairing a 2-3 tree after a leaf is too full and a key needs to be promoted

Insert: Overstuffing a Node (Left-Side)

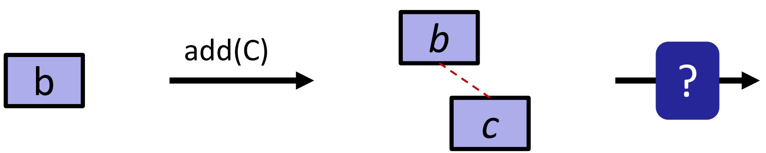
- Use a red link to mimic the corresponding 2-3 tree.

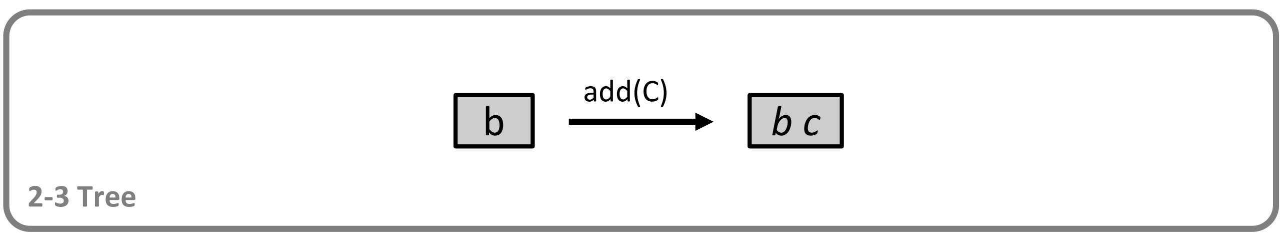




Insert: Overstuffing a Node (Right-Side)

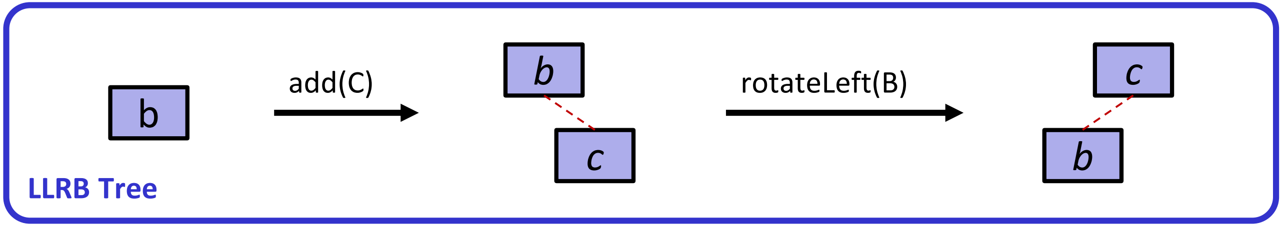
- What is the problem with inserting a red link to the right child? What should we do to fix it?

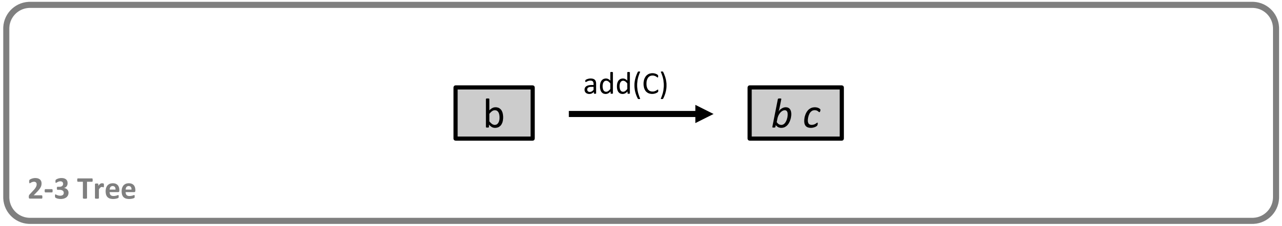




Insert: Overstuffing a Node (Right-Side)

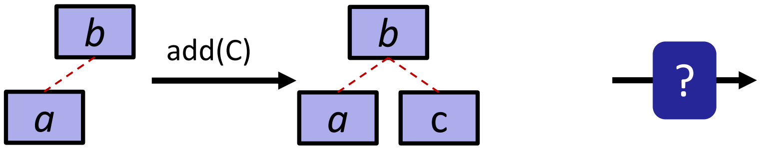
- Rotate left around B

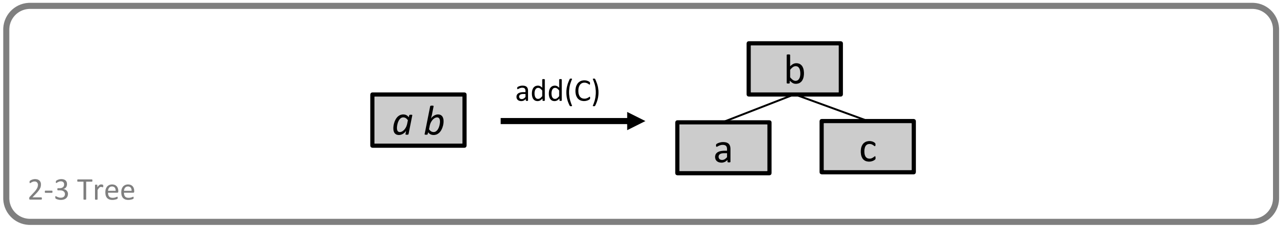




Insert: Inserting to the Right Side

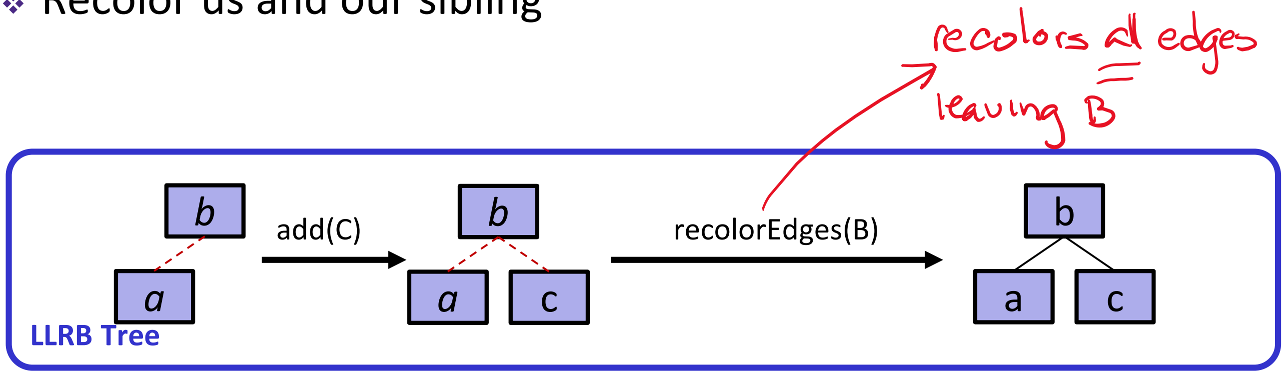
- How do we add to the right side?

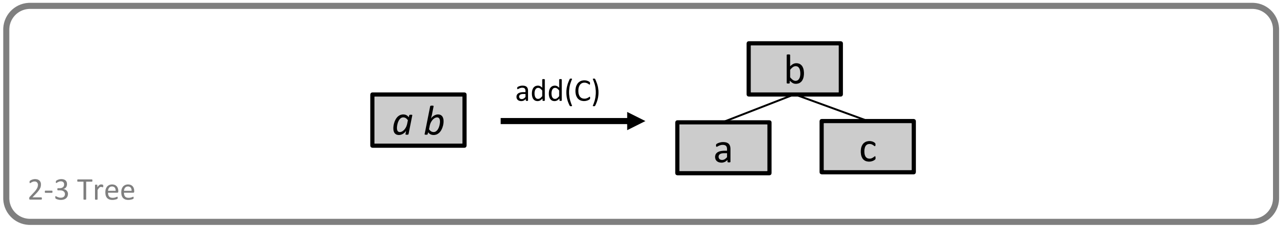




Insert: Inserting to the Right Side

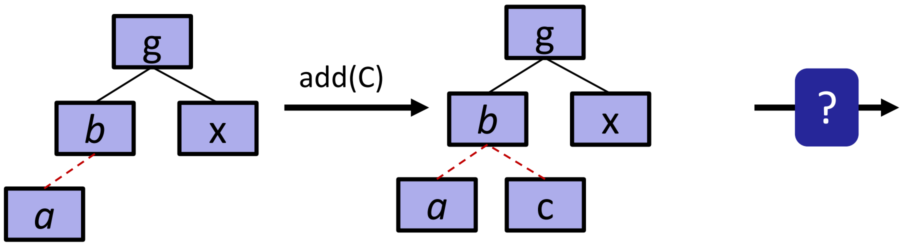
- Recolor us and our sibling

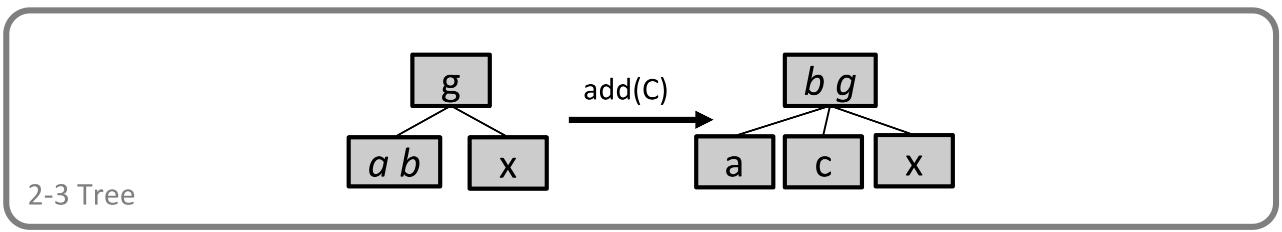




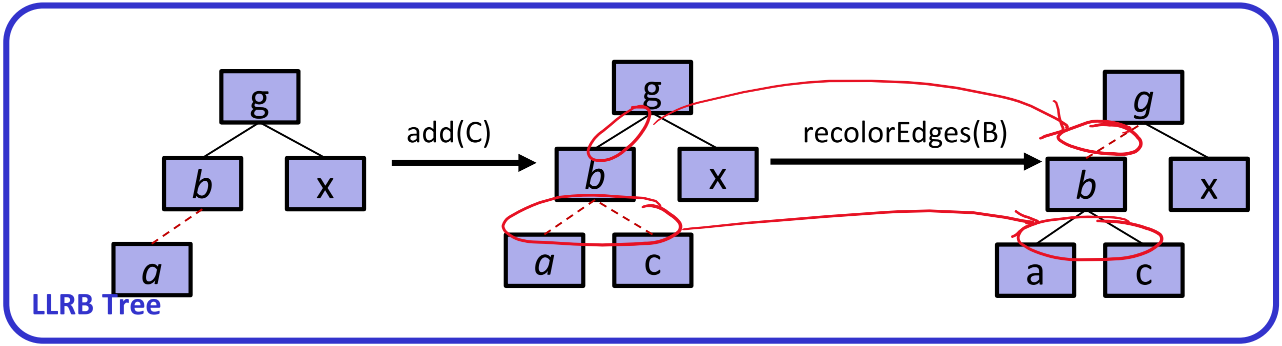
Insert: Splitting a Node

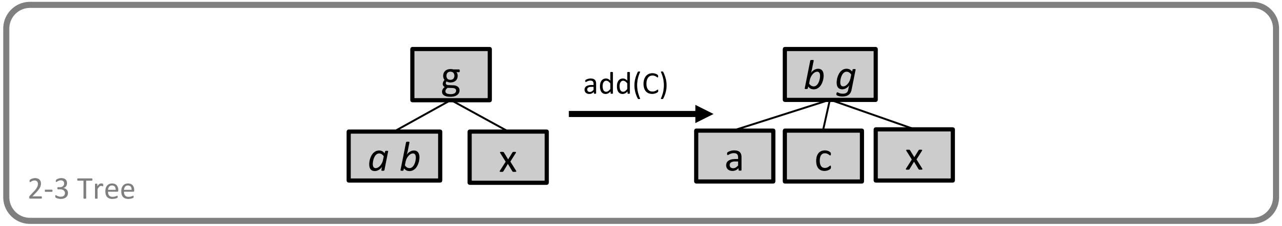
- How do we split a node?



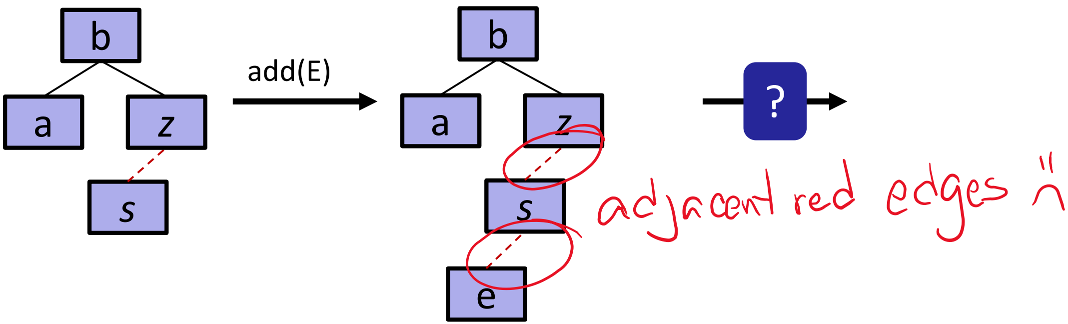


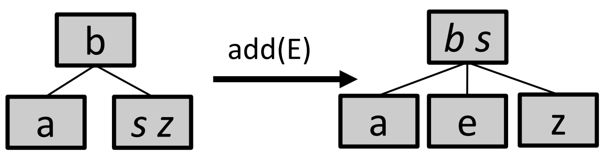
Insert: Splitting a Node





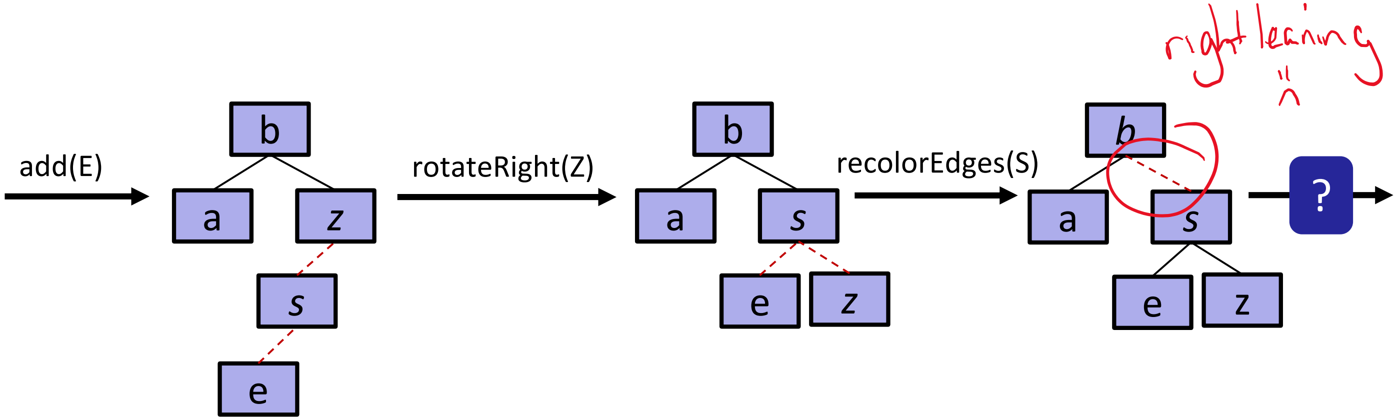
Insert: Practice

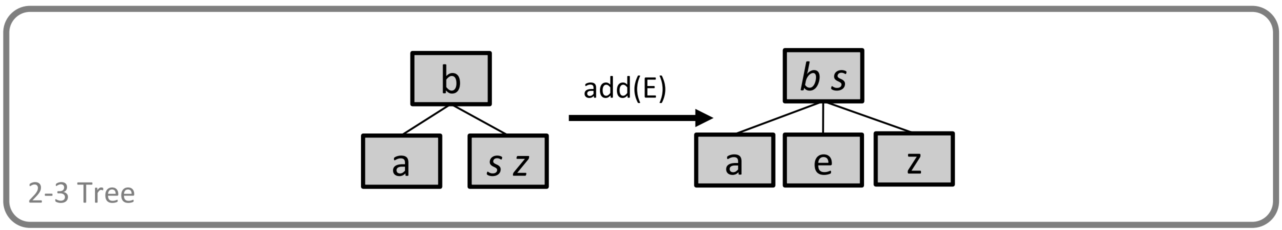




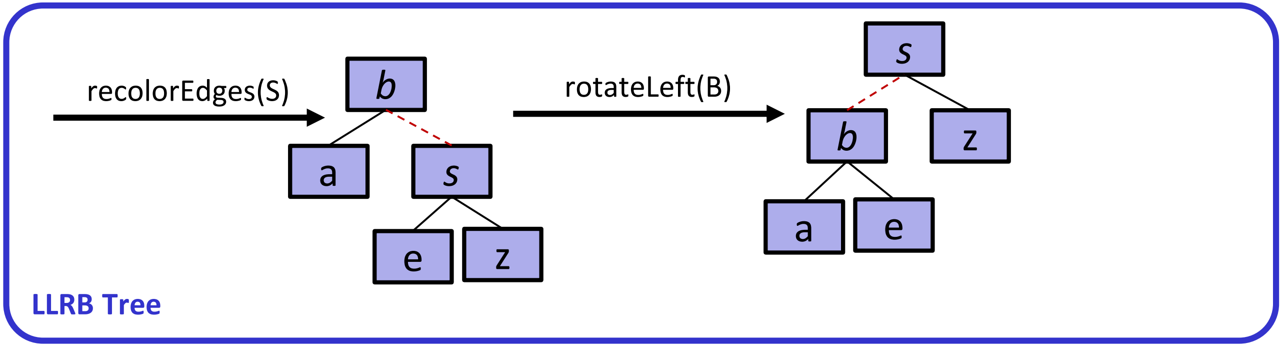
2-3 Tree

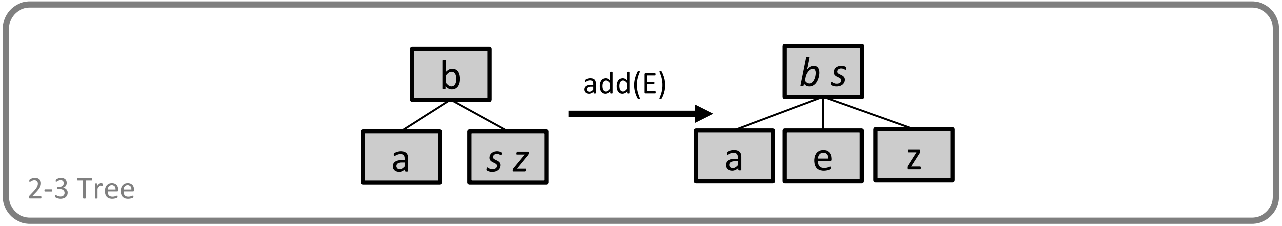
Insert: Practice





Insert: Practice





Left-Leaning Red-Black Tree Invariants

- **Left-Leaning Red-Black (LLRB) Tree** is a BST variant with the following additional invariants:
  1. Every root-to-bottom path has the same number of black edges
  2. Red edges lean left
  3. No node has two red edges connected to it, either above/below or left/right

- When repairing an LLRB Tree, use the following recipes:
  - Right link red? **Rotate left**
  - Two left reds in a row? **Rotate right**
  - Both children red? **Recolor all edges leaving the node**

Insert: Java Implementation

```
private Node insert(Node h, Key key, Value value) {
  if (h == null) { return new Node(key, value, RED); }

  int cmp = key.compareTo(h.key);
  if (cmp < 0)      { h.left  = insert(h.left,  key, val); }
  else if (cmp > 0) { h.right = insert(h.right, key, val); }
  else              { h.value = value;                     }

  if (isRed(h.right) && !isRed(h.left))      { h = rotateLeft(h);  }
  if (isRed(h.left)  &&  isRed(h.left.left)) { h = rotateRight(h); }
  if (isRed(h.left)  &&  isRed(h.right))     { recolorEdges (h);   }

  return h;
}
```

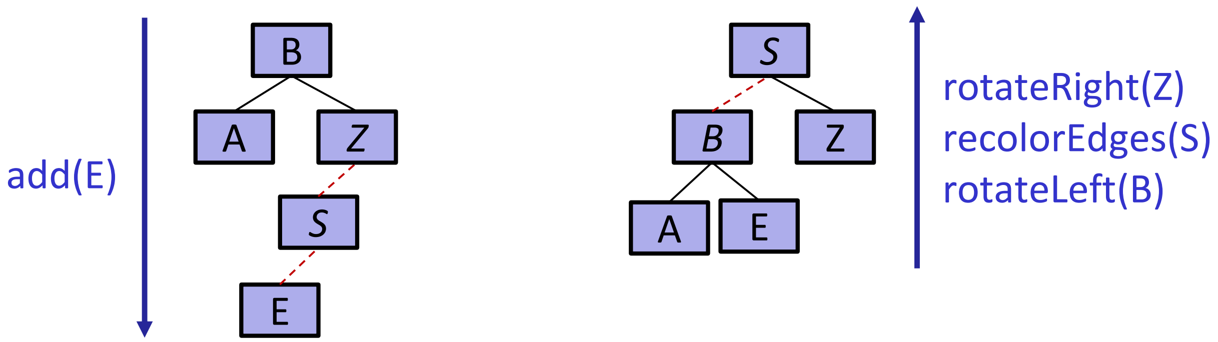
Right link red? **Rotate left**

Two left reds in a row? **Rotate right**

Both children red? **Recolor all edges leaving node**

Left-Leaning Red-Black Trees Runtime

- Searching for a key is the same as a BST
- Tree height is guaranteed in Θ(log N)
- Inserting a key is a recursive process
  - Θ(log N) to add(E)
  - Θ(log N) to **maintain invariants**



Lecture Outline

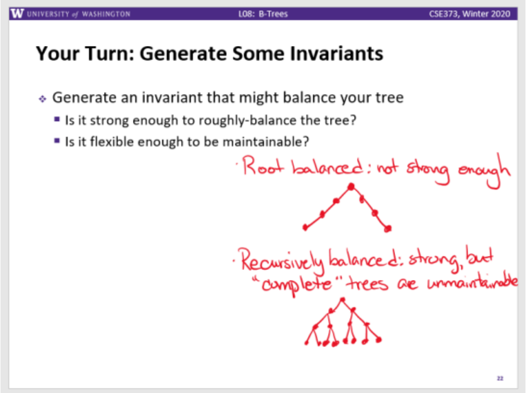
- Review: 2-3 Trees and BSTs
- Left-Leaning Red-Black Trees
  - Insertion
- **Other Balanced BSTs**

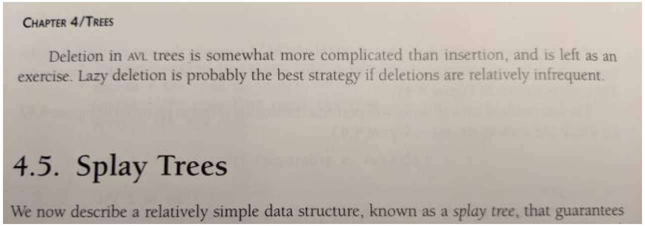
Red-Black Trees

- Left-leaning Red-Black trees:
  - Invented 2008 as a "simpler-to-implement" Red-Black tree
- Red-black trees:
  - Invented 1972 (!!) and handles the "right-leaning" case
  - Nodes, not edges, are colored red/black
  - Used millions (billions?) of times as a second: Java TreeMap, C++ Map, Linux scheduler and epoll, …
  - You will get to use (but not implement) in HW4!

AVL Trees

- Recursively balanced with equal heights = not flexible enough
  - Can only represent inputs of size 2^n - 1
- Recursively balanced with heights differing <=1
  - AVL tree!
- Insertions: add to leaf, then log N rotations until tree is rebalanced
- Deletions: lol





CHAPTER 4/TREES

Deletion in AVL trees is somewhat more complicated than insertion, and is left as an exercise. Lazy deletion is probably the best strategy if deletions are relatively infrequent.

4.5. Splay Trees

We now describe a relatively simple data structure, known as a *splay tree*, that guarantees

… and Still More

- Order Statistic Tree
- Interval Tree
- Splay Tree
- Dancing Tree
- And so much more!

tl;dr (1 of 2)

- **Search Trees** have great runtimes most of the time
  - But they struggle with sorted (or mostly-sorted) input
  - Must bound the height if we need runtime guarantees
- **Plain BSTs**: simple to reason about/implement. A good starting point
- **Left-leaning Red-Black Trees**: A *BST variant* with a Θ(log N) bound on the height
  - Invariants quite tricky, but implementation isn't bad!
  - Correctness and runtimes guaranteed by 1-1 mapping with 2-3 trees

tl;dr (2 of 2)

- **B-Trees** are a *Search Tree variant* with a $\Theta(\log_2 N)$ bound on the height
  - Only allows the tree to grow from its root
  - Added two simple invariants, but implementation quite tricky
    - All leaves must be the same depth from the root
    - A non-leaf node with k keys must have exactly k+1 non-null children
- Possible data structures for a Map and/or Set ADT:

	LinkedList Map, Worst Case	BST Map, Worst Case	B-Tree Map, Worst Case	LLRBT Map, Worst Case
Find	$\Theta(N)$	$h = \Theta(N)$	$\Theta(\log N)$	$\Theta(\log N)$
Add	$\Theta(N)$	$h = \Theta(N)$	$\Theta(\log N)$	$\Theta(\log N)$
Remove	$\Theta(N)$	$h = \Theta(N)$	$\Theta(\log N)$	$\Theta(\log N)$