CSC 212
Data Structures & Algorithms

Fall 2022 | Jonathan Schrader

Left-Leaning Red-Black Trees

Housekeeping

Lab 9: Balancing Act

- In-person labs are canceled
- Assignment 4 Due

Term Project

- Claim your topic…projects without an approved topic will not be graded…

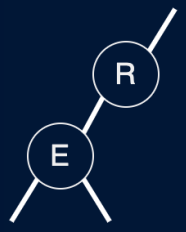
LEFT-LEANING RED-BLACK TREES

Implementing 2-3 trees with binary trees

Challenge: How to represent a 3-node?

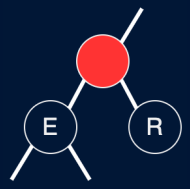
Approach 1: Regular BST

- No way to tell a 3-node from a 2-node
- Can't (uniquely) map from BST back to 2-3 tree



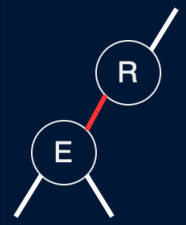
Approach 2: Regular BST with red "glue" nodes

- Wastes space for extra node
- Messy code



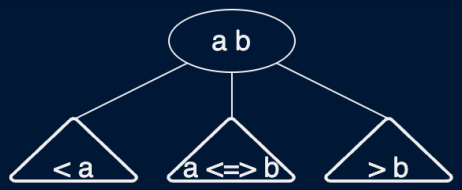
Approach 3: Regular BST with red "glue" links

- Widely used in practice
- Arbitrary restriction: red links lean left

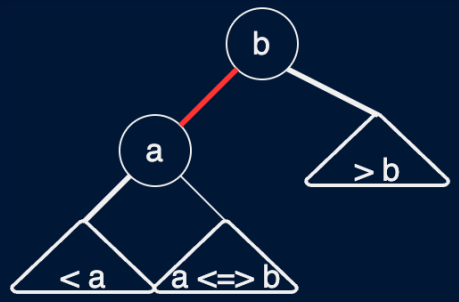


Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2-3 trees as a BST
2. Use “internal” left-leaning links as “glue” for 3-nodes



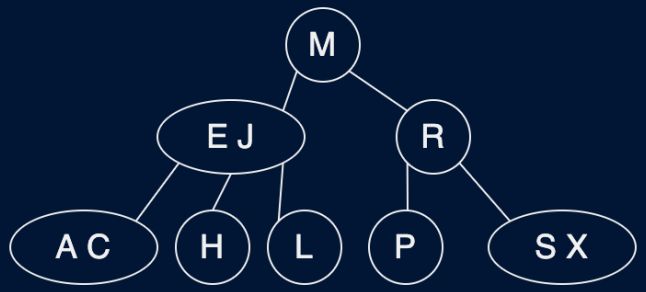
$3-node$



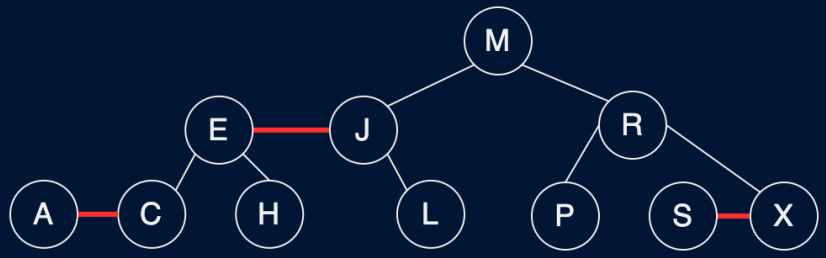
$b\ is\ the\ larger\ root\ key$

1-1 correspondence with 2-3 trees

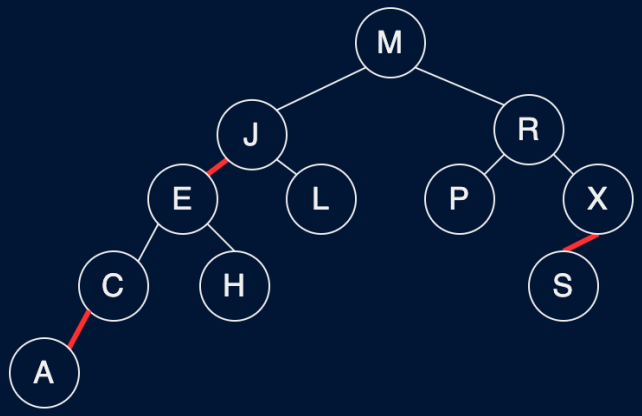
Key property:



$2-3\ tree$



$horizontal\ red\ links$

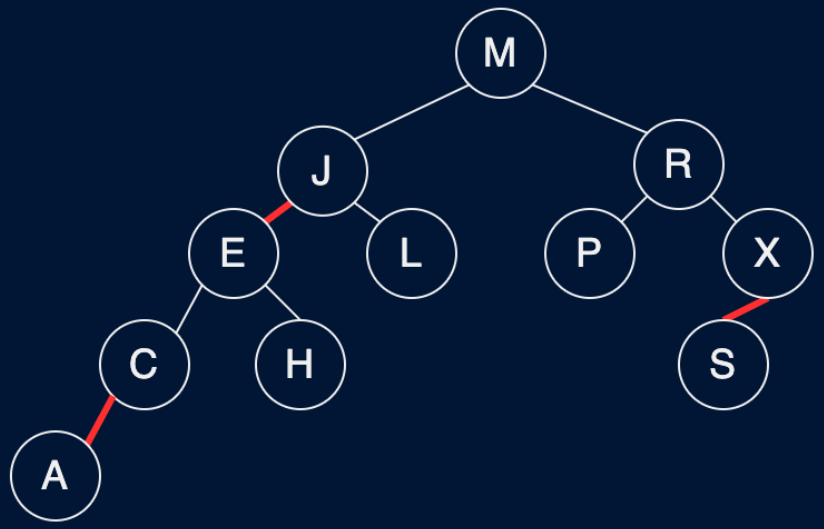


$red-black\ tree$

Definition of LLRB trees (without reference to 2-3)

A BST such that:

- No node has two red links connected to it
- Red links lean left
- Every path from root to null has the same number of black links



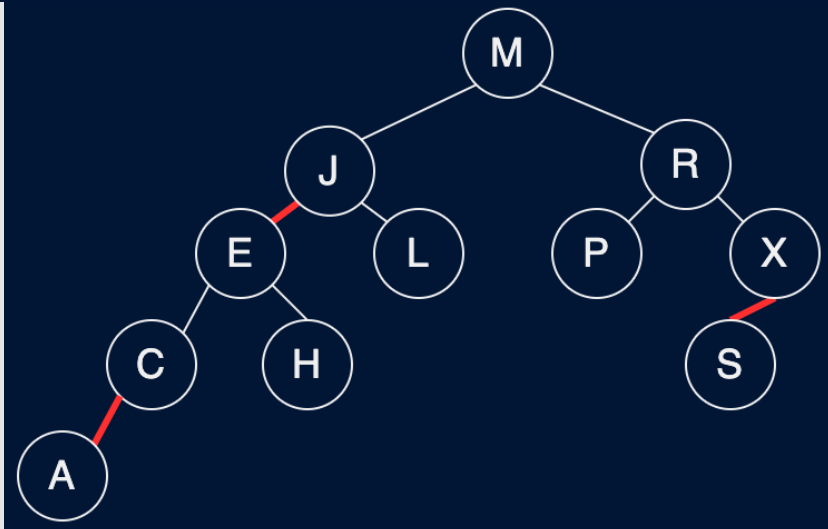
$$red - black\ tree$$

SEARCH

Observation

Search is the same as for BST (ignore color)

```
public Value get(Key key) {
  Node x = root;
  while (x != null) {
    int cmp = key.compareTo(x.key);
    if      (cmp <  0)  x = x.left;
    else if (cmp >  0)  x = x.right;
    else if (cmp == 0)  return x.val;
  }
  return null;
}
```



$red - black\ tree$

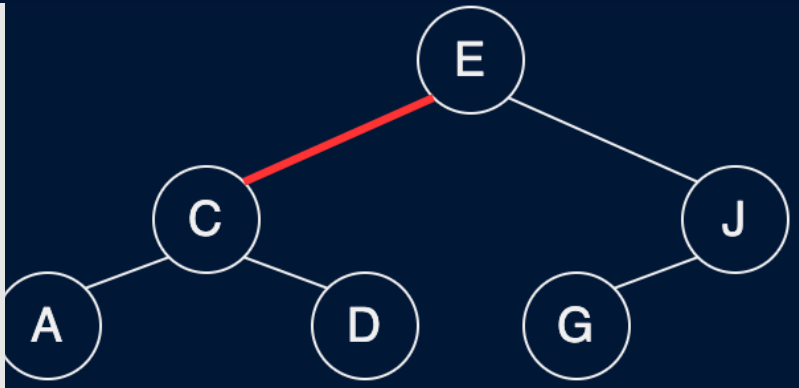
Remark: Many other ops (floor, iteration, rank, slection) are also identical

Representation

Each node is pointed to by precisely one link (from its parent)

$\Rightarrow$ can encode color of links in nodes

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node {
  Key     key;
  Value   val;
  Node    left;
  Node    right;
  Boolean color;                  // color of parent link
}

private boolean isRed(Node x) {
  if (x == null) return false;    // null links are black
  return x.color == red;
}
```

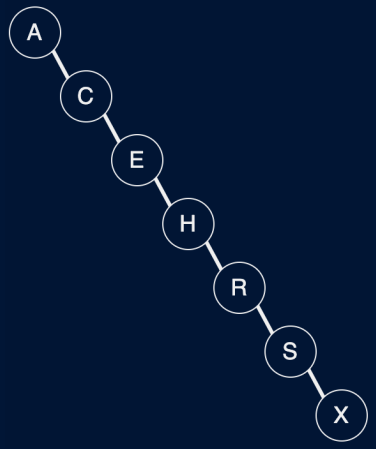


$h = root$

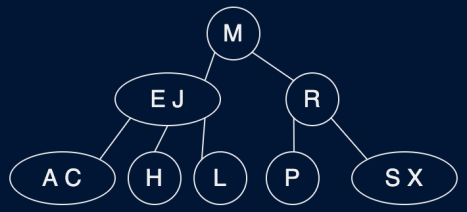
$h.left.color\ is\ RED$

$h.right.color\ is\ BLACK$

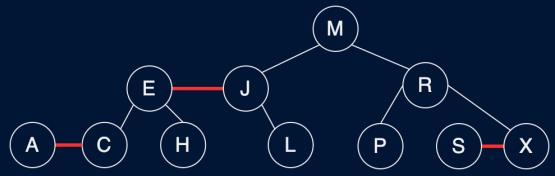
The road to LLRB trees



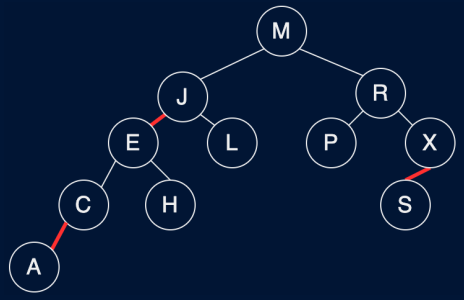
BSTs (can get imbalanced)



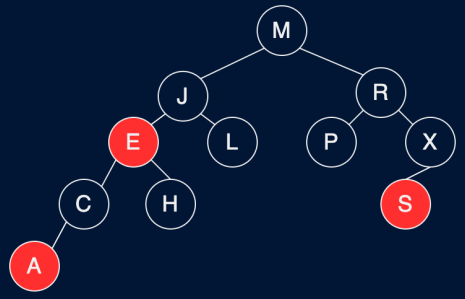
2-3 trees (balanced but awkward to implement)



3-nodes "glued" together with red links



LLRB trees (color in links)



implementing LLRB trees (color in nodes)

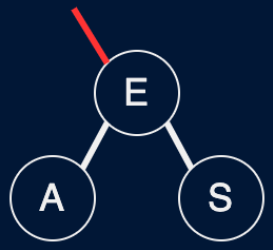
R-BTree OPERATIONS

Overview

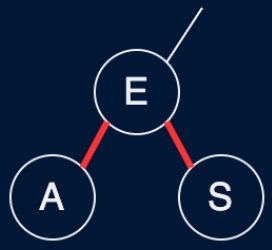
Basic strategy: Maintain 1-1 correspondence with 2-3 trees
During internal operations, maintain

- Symmetric order.
- Perfect black balance.
- [but not necessarily color invariants]

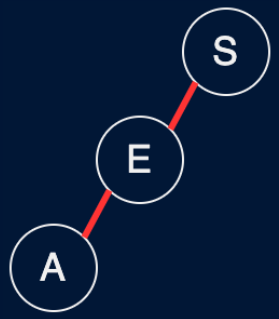
Example violations of color invariants



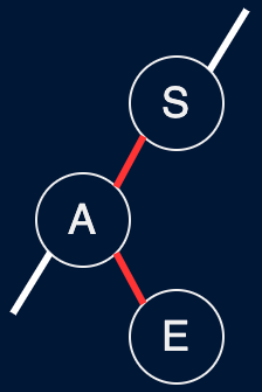
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)



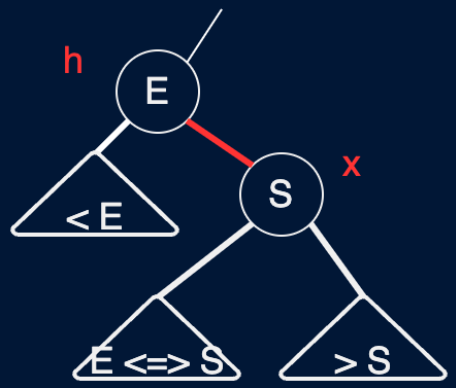
left-right red (a temporary 4-node)

To restore color invariants: perform rotations and color flips

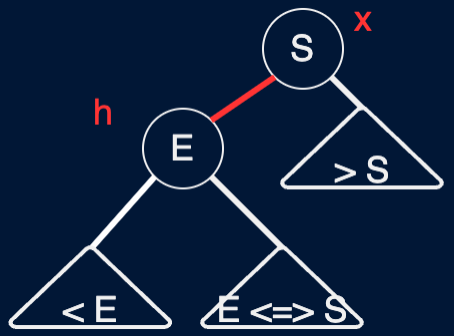
Operation: Left rotation

Orient a (temporarily) right-leaning red link to lean left

```
private Node rotateLeft(Node h) {
  assert isRed(h.right);
  Node x  = h.right;
  h.right = x.left;
  x.left  = h
  x.color = h.color;
  h.color = RED;
  return x;
}
```



rotate E left (before)



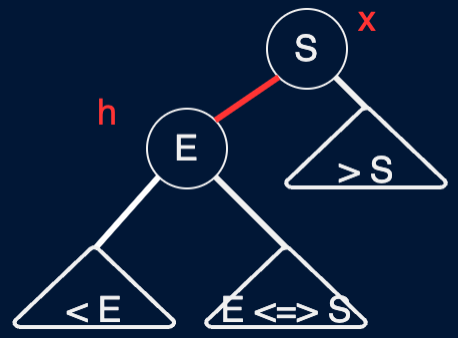
rotate E left (after)

Invariants: Maintains symmetric order and perfect balance

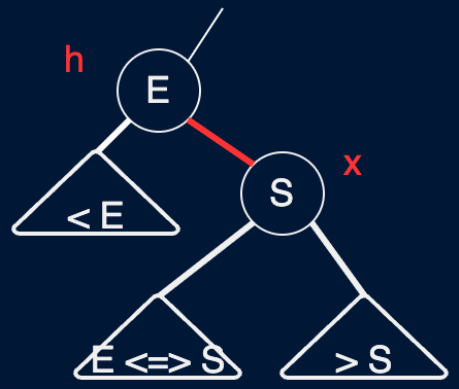
Operation: Right rotation

Orient a left-leaning red link to (temporarily) lean right

```
private Node rotateLeft(Node h) {
  assert isRed(h.right);
  Node x  = h.left;
  h.left  = x.right;
  x.right = h
  x.color = h.color;
  h.color = RED;
  return x;
}
```



rotate E right (before)



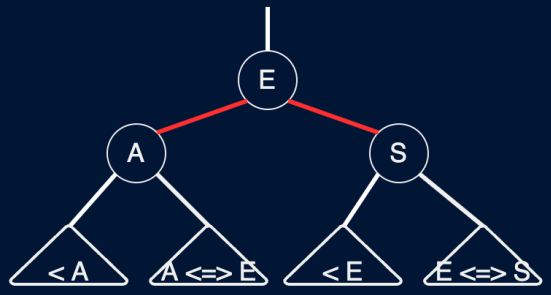
rotate E right (after)

Invariants: Maintains symmetric order and perfect balance

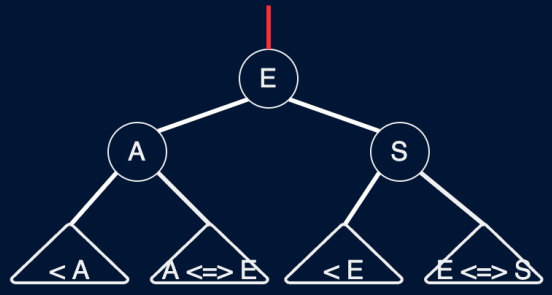
Operation: Color flip

Recolor to split a (temporary) 4-node

```
private Node rotateLeft(Node h) {
  assert isRed(h);
  assert isRed(h.left);
  assert isRed(h.right);
  h.color       = RED;
  h.left.color  = BLACK;
  h.right.color = BLACK;
}
```



flip colors (before)



flip colors (after)

Invariants: Maintains symmetric order and perfect balance

LLRB INSERTIONS

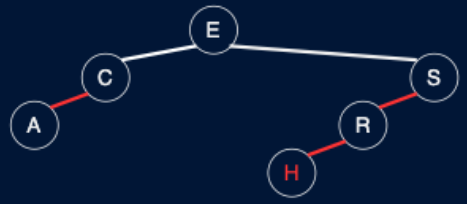
Implementation

Do standard BST insert ← preserves symmetric order
Color new link red ← preserve perfect black balance
Repeat up the tree until color invariants restored

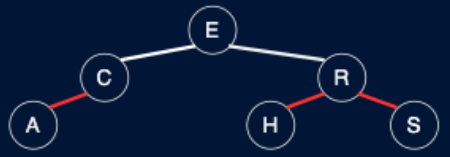
- two left red links in a row? ⇒ rotate right
- left and right links both red? ⇒ color flip
- right link only red? ⇒ rotate left

Inserting H

[R]: add new node here



[S]: two lefts in a row, rotate right



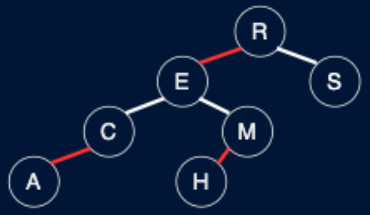
[R]: both chidren red, flip colors

Implementation

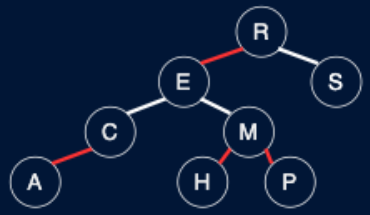
Do standard BST insert ← preserves symmetric order
Color new link red ← preserve perfect black balance
Repeat up the tree until color invariants restored

- two left red links in a row? ⇒ rotate right
- left and right links both red? ⇒ color flip
- right link only red? ⇒ rotate left

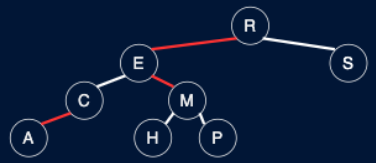
Inserting P



[M]: add new node here

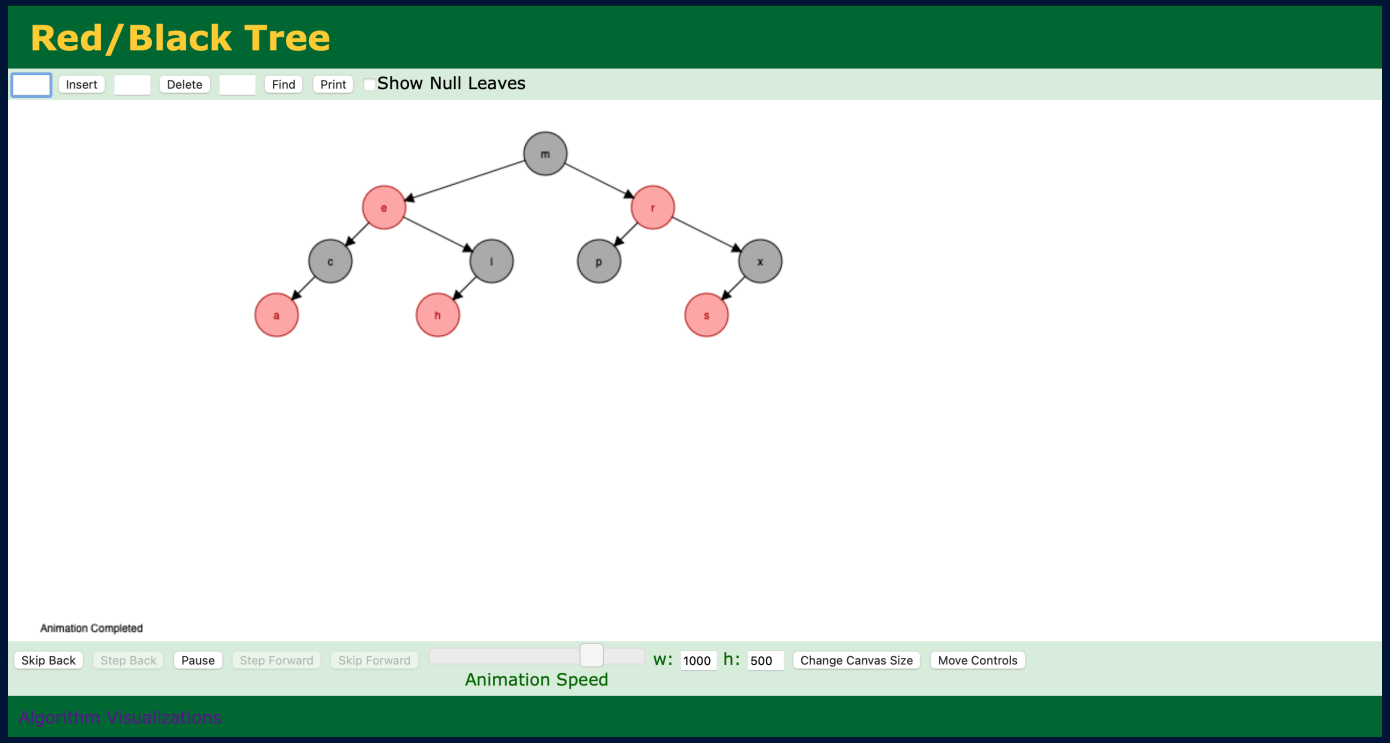


[M]: both chidren red, flip colors



right link red, rotate left

rbt



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

Implementation

Do standard BST insert and color new link red
Repeat up the tree until color invariants restored

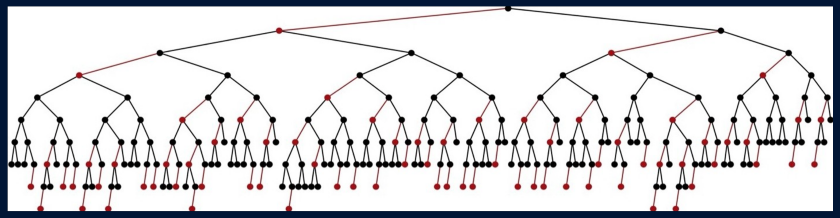
- two left red links in a row? ⇒ rotate right
- left and right links both red? ⇒ color flip
- right link only red? ⇒ rotate left

```
private Node put(Node h, Key, k, Value v) {
  if (h == null) return new Node(key, val, RED;          // insert at bottom and color red

  int cmp = key.compareTo(h.key)
  if      (cmp <  0) h.left = put(h.left, key, val);
  else if (cmp >  0) h.left = put(h.right, key, val);
  else if (cmp == 0) h.val  = val);

  // following lines restore color invariants
  if (isRed(h.right) && !isRed(h.left))        h = rotateLeft(h);
  if (isRed(h.left ) && !isRed(h.left.left))   h = rotateRight(h);
  // only a few extra lines of code provides near-perfect balance
  if (isRed(h.left ) && isRed(h.right))        h = flipColors(h);

  return h;
}
```

Visualizations

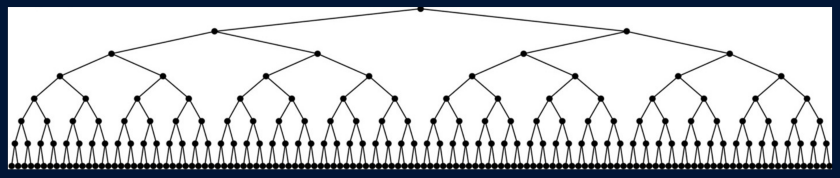
$n = 255$

Random order
$height = 9$
$average\ depth = 6.3$



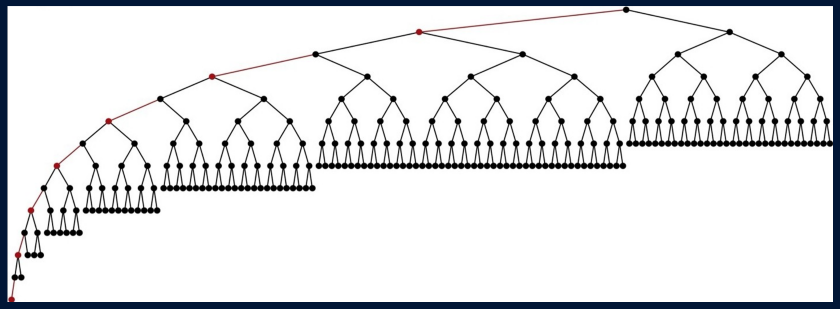
random

Ascending order
$height = 7$
$average\ depth = 6.0$



ascending

Descending order
$height = 13$
$average\ depth = 6.5$



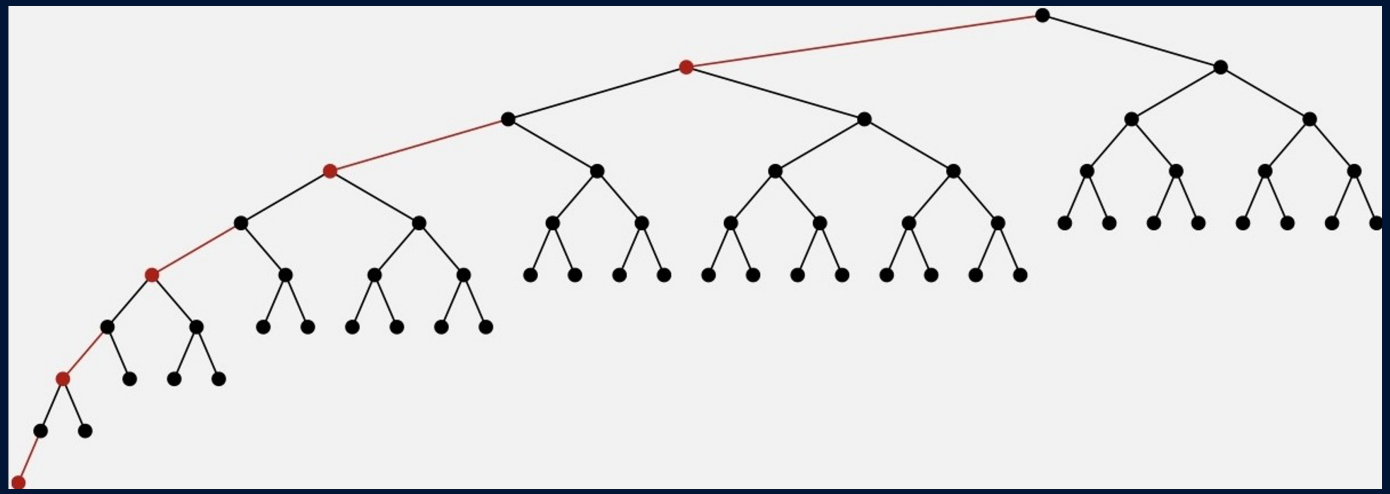
descending

Balance

Proposition: Height of corresponding 2-3 tree is $\leq log_2\ n$

Proof:

- Black height = height of corresponding 2-3 tree $\leq log_2\ n$
- Never two red links in a row
  $\Rightarrow$ height of LLRB tree $\leq (2 \times black\ height) + 1 \leq 2\ log_2\ n$
- [A slightly more refined argument show $height\ \leq 2\ log\ n$]



$$height\ \leq 2\ log_2\ n$$

Summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search	insert	delete		
sequential search (unordered list)	n	n	n	n	n	n		`equals()`
binary search (ordered array)	$\log n$	n	n	$\log n$	n	n	✔	`compareTo()`
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$	✔	`compareTo()`
2-3 tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✔	`compareTo()`
red-black BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✔	`compareTo()`

hidden constant c is small for r-b BST
(at most $2 \log_2 n$ compares)