CS 3110

Balanced Trees

Prof. Clarkson

Fall 2017

Today's music: Get the Balance Right by Depeche Mode

Prelim

- See Piazza post @422
- Makeup: Thursday night 5:30 pm
- Wed noon: All requests for other makeup/conflict accommodations due to cs3110-mgmt-L@cs.cornell.edu

Review

Previously in 3110:

- Advanced data structure: streams (and laziness)

Today:

- Binary search trees
- Balanced search trees
- Running example: sets
- (Balanced trees also useful for maps)

Set interface

```
module type Set = sig
  type 'a t
  val we      : 'a t
  val are     : 'a -> 'a t -> 'a t
  val groot   : 'a -> 'a t -> bool
end
```

Set interface

```
module type Set = sig
  type 'a t
  val empty  : 'a t
  val insert : 'a -> 'a t -> 'a t
  val mem    : 'a -> 'a t -> bool
end
```

Set implementations

```
module ListSet : Set = struct
  ...
end

module BstSet : Set = struct
  ...
end

module RbSet: Set = struct
  ...
end
```

Set implementations: performance

	Workload 1	
	`insert`	`mem`
`ListSet`	35s	106s

MacBook, 1.3 GHz Intel Core m7, 8 GB RAM, OCaml 4.05.0, median of three runs

Set implementations: performance

	Workload 1	
	`insert`	`mem`
`ListSet`	35s	106s
`BstSet`	130s	149s

MacBook, 1.3 GHz Intel Core m7, 8 GB RAM, OCaml 4.05.0, median of three runs

Set implementations: performance

	Workload 1		Workload 2	
	`insert`	`mem`	`insert`	`mem`
`ListSet`	35s	106s	35s	106s
`BstSet`	130s	149s	0.07s	0.07s

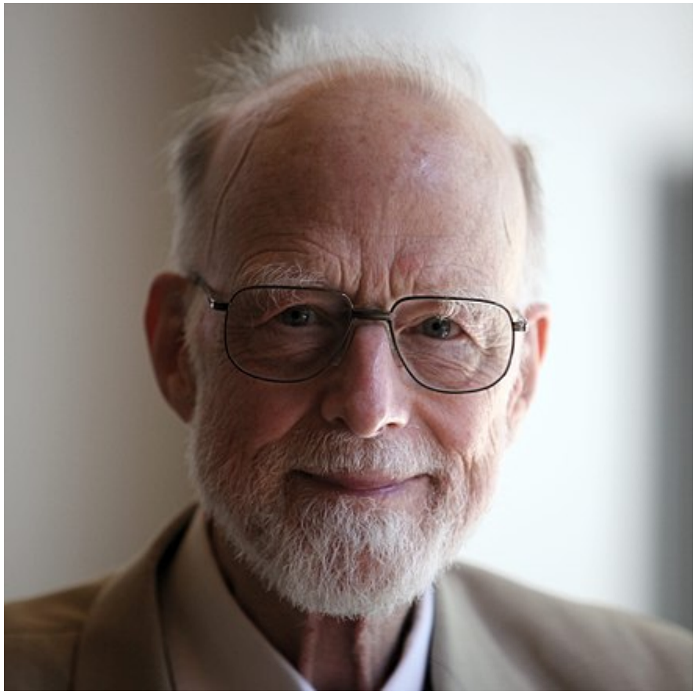
MacBook, 1.3 GHz Intel Core m7, 8 GB RAM, OCaml 4.05.0, median of three runs

Set implementations: performance

	Workload 1		Workload 2	
	`insert`	`mem`	`insert`	`mem`
`ListSet`	35s	106s	35s	106s
`BstSet`	130s	149s	0.07s	0.07s
`RbSet`	0.12s	0.07s	0.15s	0.08s

MacBook, 1.3 GHz Intel Core m7, 8 GB RAM, OCaml 4.05.0, median of three runs

Sir Tony Hoare



b. 1934

Turing Award Winner 1980

For his fundamental contributions to the definition and design of programming languages.

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

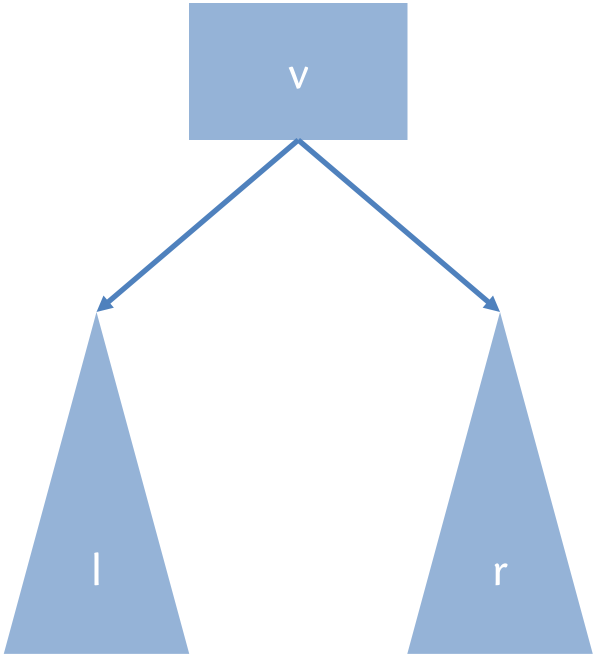
LIST VS. BST

List

```
module ListSet = struct
  (* AF: [x1; ...; xn] represents
   *   the set {x1, ..., xn}.
   * RI: no duplicates. *)
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let insert x s =
    if mem x s then s else x::s
end
```

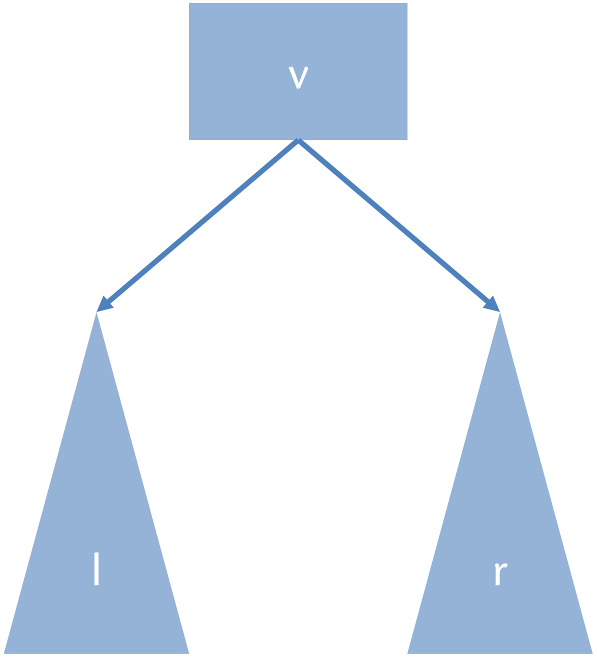
Binary search tree (BST)

- Binary tree: we are groot
- BST invariant:
  - all values in l are less than v
  - all values in r are greater than v



Binary search tree (BST)

- Binary tree: every node has two subtrees
- BST invariant:
  - all values in l are less than v
  - all values in r are greater than v



Question

You might remember from 2110 that finding element in list is *O(n).* How efficient is finding an element in a BST?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. O(n^2)

Question

You might remember from 2110 that finding element in list is $O(n)$. How efficient is finding an element in a BST?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. $O(n^2)$

BST

```
module BstSet = struct
  (* AF:  [Leaf] represents the empty set.
   *    [Node (l, v, r)] represents
   *    the set $AF(l) \union {v} \union
   *    AF(r)$.
   * RI:  for every [Node (l, v, r)],
   *    all the values in [l] are strictly less than
   *    [v], and all the values in [r] are strictly
   *    greater than [v]. *)

  type 'a t =
    | Leaf
    | Node of 'a t * 'a * 'a t
```

BST

```
module BstSet = struct
  ...
  let rec mem x = function
    | Leaf -> false
    | Node (l, v, r) ->
      if      x < v then mem x l
      else if x > v then mem x r
      else      true
```

BST

```
module BstSet = struct
  ...
  let rec insert x = function
    | Leaf -> Node (Leaf, x, Leaf)
    | Node (l, v, r) ->
      if       x < v then Node(insert x l, v, r         )
      else if x > v then Node(l,          v, insert x r)
      else               Node(l,          x, r         )
end
```

Back to performance

	Workload 1		Workload 2	
	`insert`	`mem`	`insert`	`mem`
`ListSet`	35s	106s	35s	106s
`BstSet`	130s	149s	0.07s	0.07s

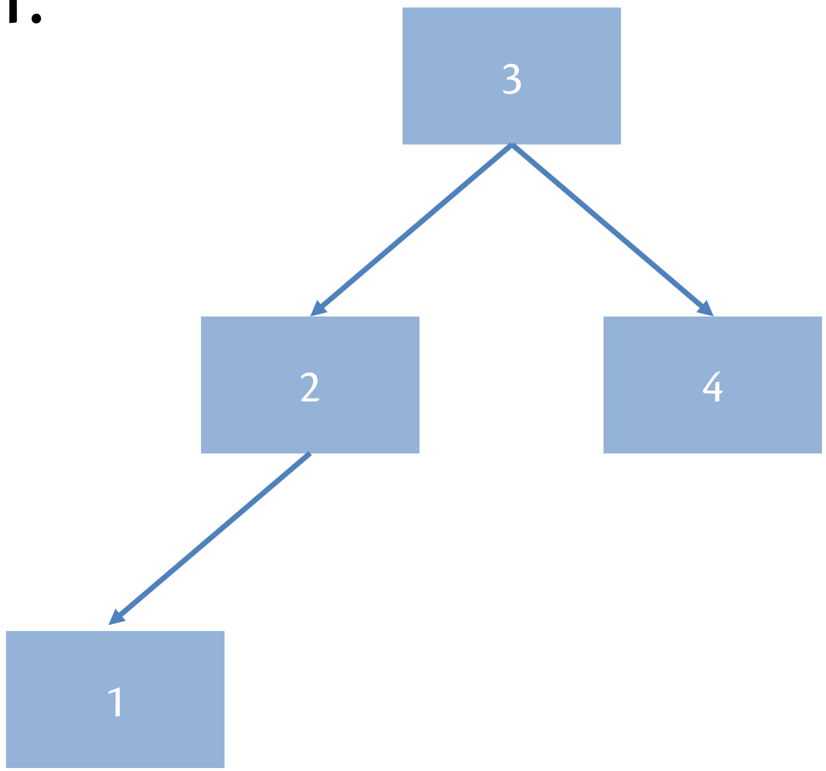
Workloads

- Workload 1:
  - `insert`: 50,000 elements in ascending order
  - `mem`: 100,000 elements, half of which not in set

- Workload 2:
  - `insert`: 50,000 elements in random order
  - `mem`: 100,000 elements, half of which not in set

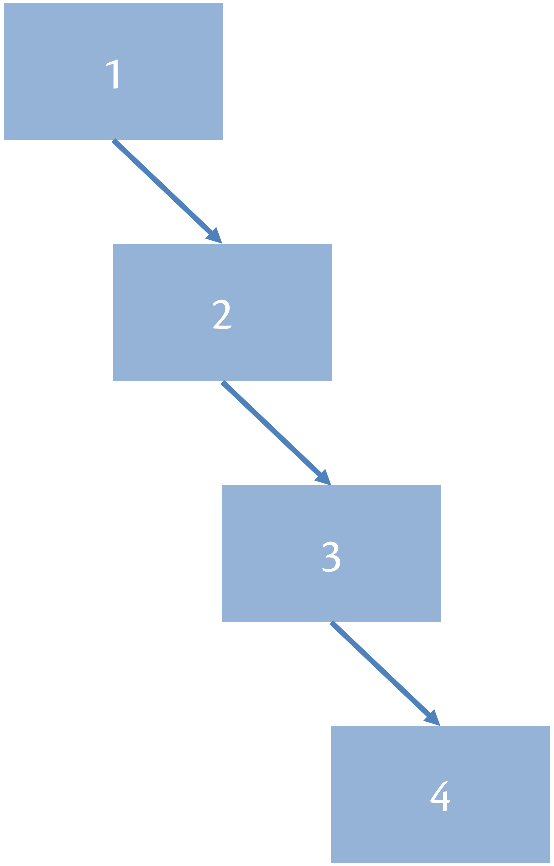
Insert in random order

- Resulting tree depends on exact order
- One possibility for inserting 1..4 in random order 3, 2, 4, 1:



Insert in linear order

Only one possibility for inserting 1..4 in linear order 1, 2, 3, 4:

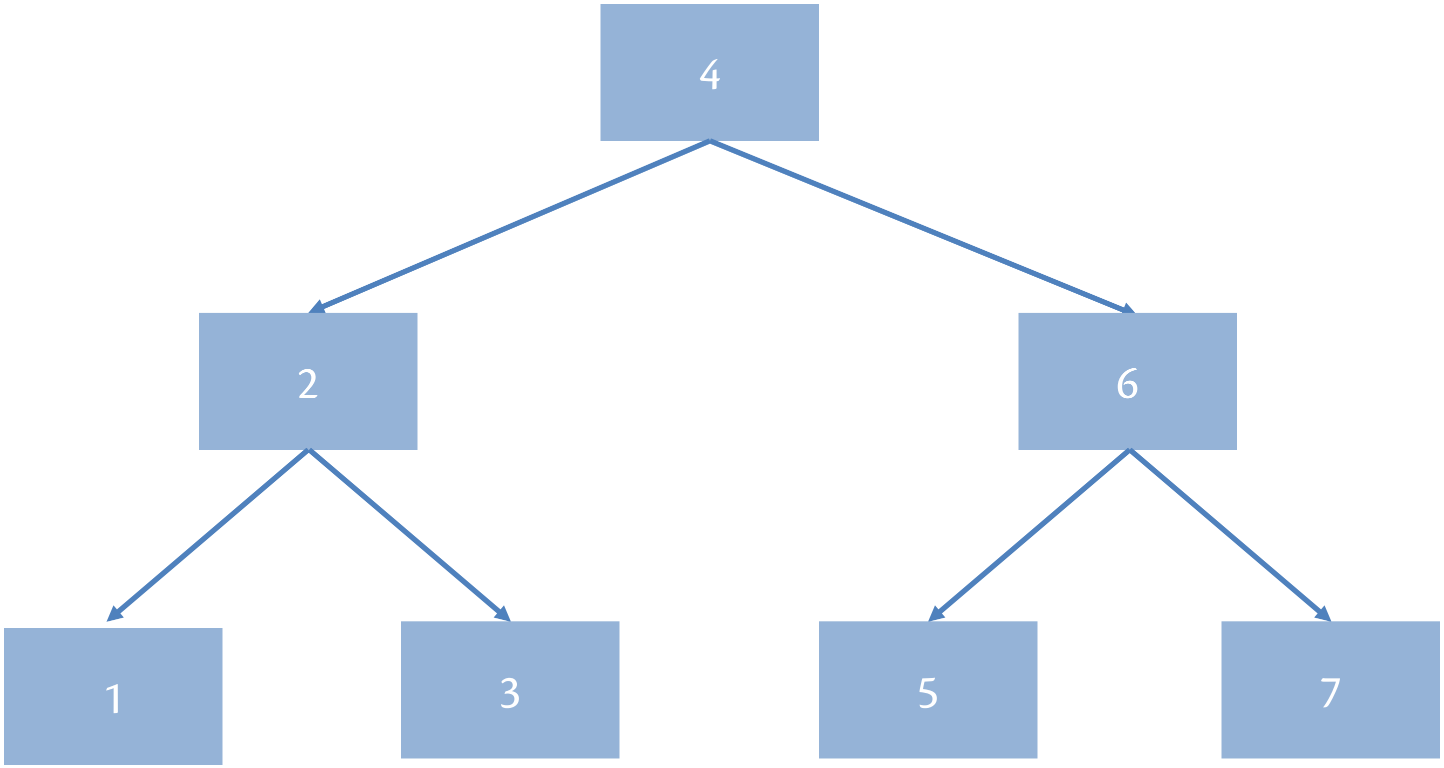


unbalanced: leaning toward the right

When trees get big

- Check out:
  - `linear_bst 100`
  - `rand_bst 100`
- Inserting next element in linear tree always takes *n* operations where *n* is number of elements in tree already
- Inserting next element in randomly-built tree might take far fewer…

Best case tree



all paths through *perfect binary tree* have same length: $log_2\ (n{+}1)$,
where n is the number of nodes,
recalling there are implicitly leafs below each node at bottom level

Performance of BST

- `insert` and `mem` are both *O(n)*
  - recall, big-O means worst case execution time
- But if trees always had short paths instead of long paths, could be better: *O(log n)*
- How could we ensure short paths?
  i.e., *balance* trees so they don't lean

Strategies for achieving balance

- In general:
  - Strengthen the RI to require balance
  - And modify insert to guarantee that RI
- Well known data structures:
  - 2-3 trees: all paths have same length
  - AVL trees: length of shortest and longest path from any node differ at most by one
  - Red-black trees: length of shortest and longest path from any node differ at most by factor of two
- All of these achieve *O(log(n))* `insert` and `mem`

CS 3110

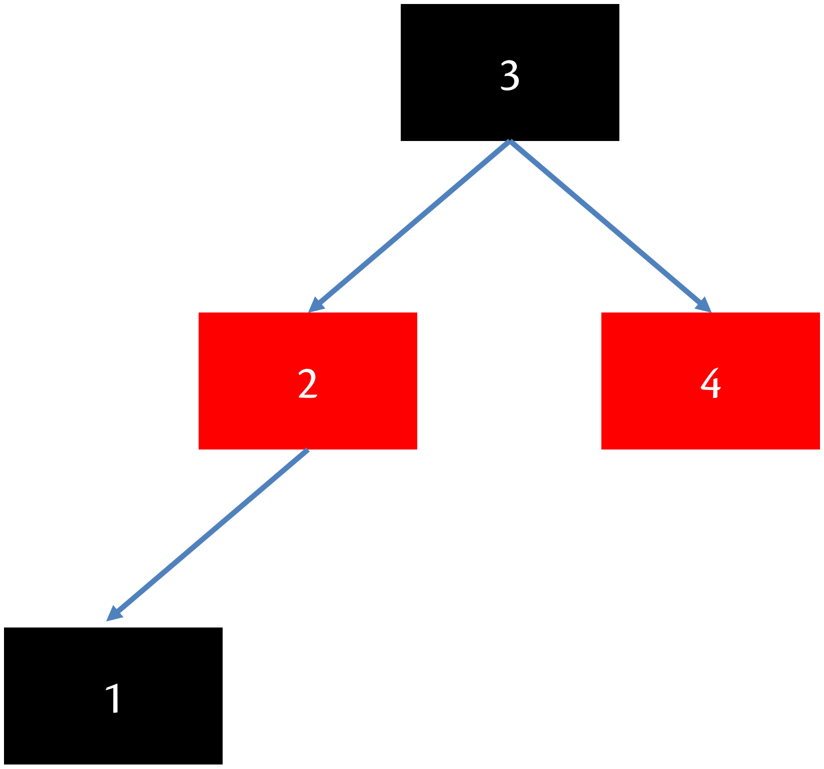
RED-BLACK TREES

Red-black trees

- [Guibas and Sedgewick 1978], [Okasaki 1998]
- Binary search tree with each node *colored* red or black
- Conventions:
  - Root is always black
  - Empty leafs are considered to be black
- RI: BST +
  - No red node has a red child
  - Every path from the root to a leaf has the same number of black nodes

Question

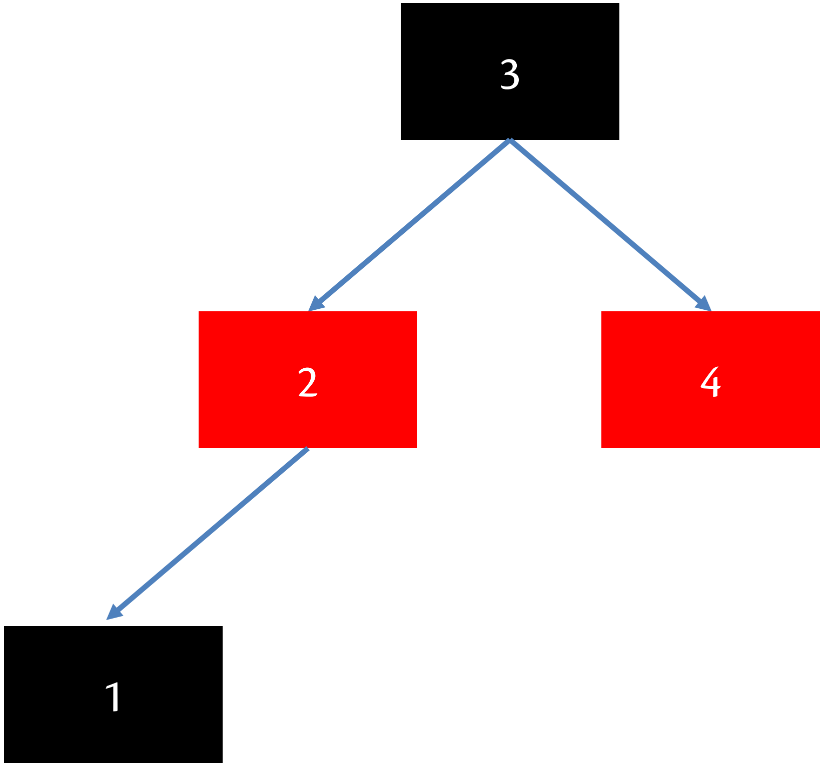
Is this a valid rep?



A. Yes
B. No

Question

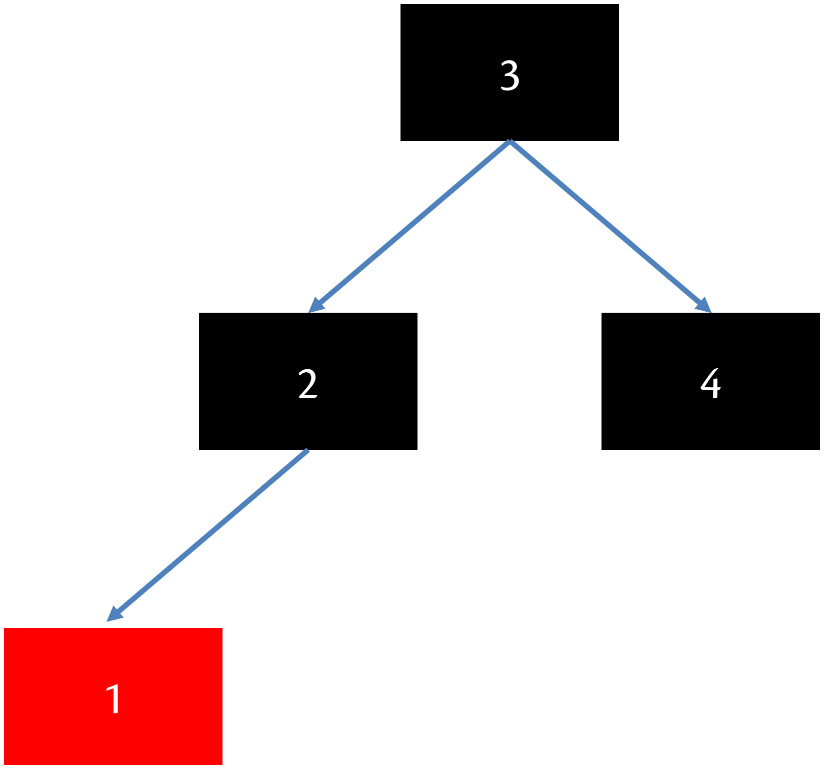
Is this a valid rep?



A. Yes
B. No

Question

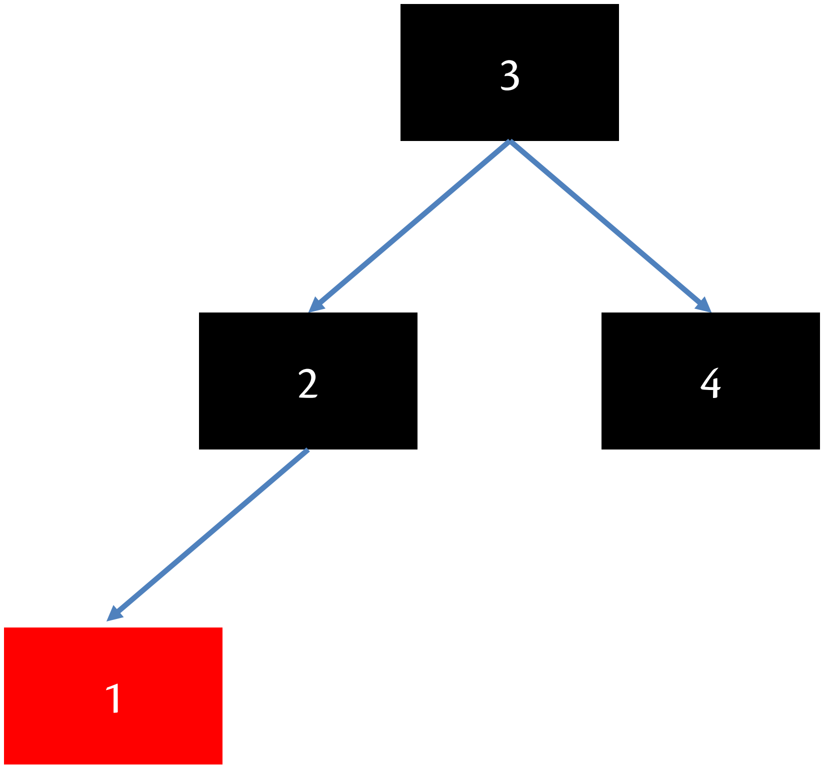
Is this a valid rep?



A. Yes
B. No
C. We are groot

Question

Is this a valid rep?



A. Yes

B. No

Path length

- Recall invariants:
  - No red node has a red child
  - Every path from the root to a leaf has the same number of black nodes
- Together imply: length of longest path is at most twice length of shortest path
  - e.g., B-R-B-R-B-R-B vs. B-B-B-B

Red-black implementation

```
module RbSet = struct
  type color = Red | Blk
  type 'a t =
    | Leaf
    | Node of (color * 'a t * 'a * 'a t)
  let empty = Leaf
  let rec mem x = function
    | Leaf -> false
    | Node (_, l, v, r) ->
      if      x < v then mem x l
      else if x > v then mem x r
      else    true
```

Same as BST except for color

Same as BST except for color

Red-black insert algorithm

- Use same algorithm as BST to find place to insert
- Color inserted node Red
- Now RB invariant might be violated (Red-Red)
- Recurse back up through tree, restoring invariant at each level with a *rotation* that balances subtree
  - 4 possible rotations
  - corresponding to 4 ways a black node could have a red child with red grandchild
- Finally color root Black

Red-black insert

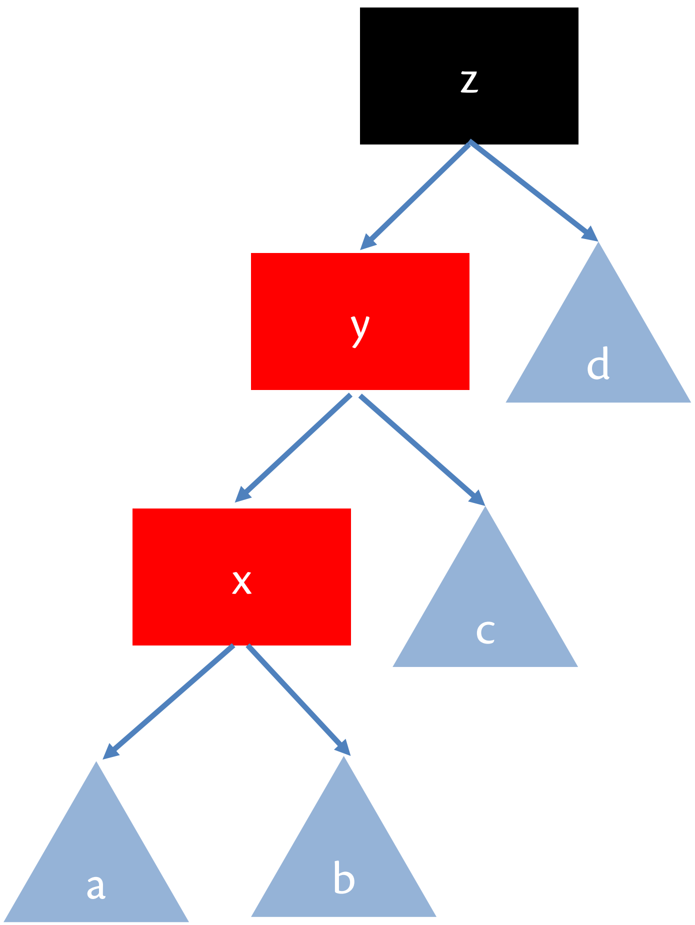
```
let rec insert' x = function
  | Leaf -> Node(Red, Leaf, x, Leaf)
  | Node (col, l, v, r) ->
    if x < v
    then balance (col, (insert' x l), v, r)
    else if x > v
    then balance (col, l, v, (insert' x r))
    else Node (col, l, x, r)

let insert x s =
  match insert' x s with
  | Node (_, l, v, r) -> Node(Blk, l, v, r)
  | Leaf -> failwith "impossible"
```

color new node Red

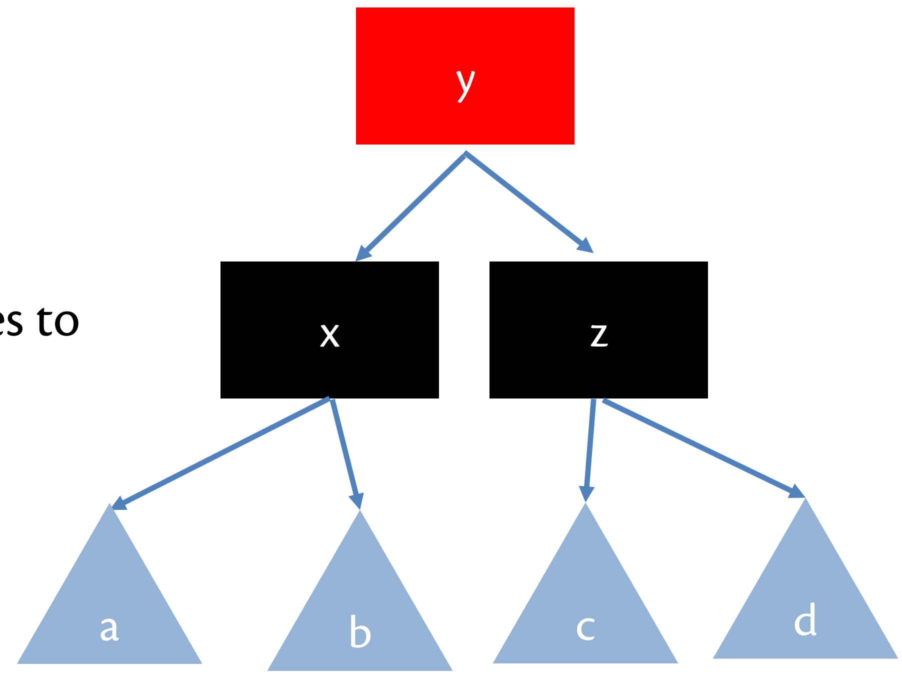
like BST insert except balance each subtree on way back up

color root Black

RB rotate (1 of 4)

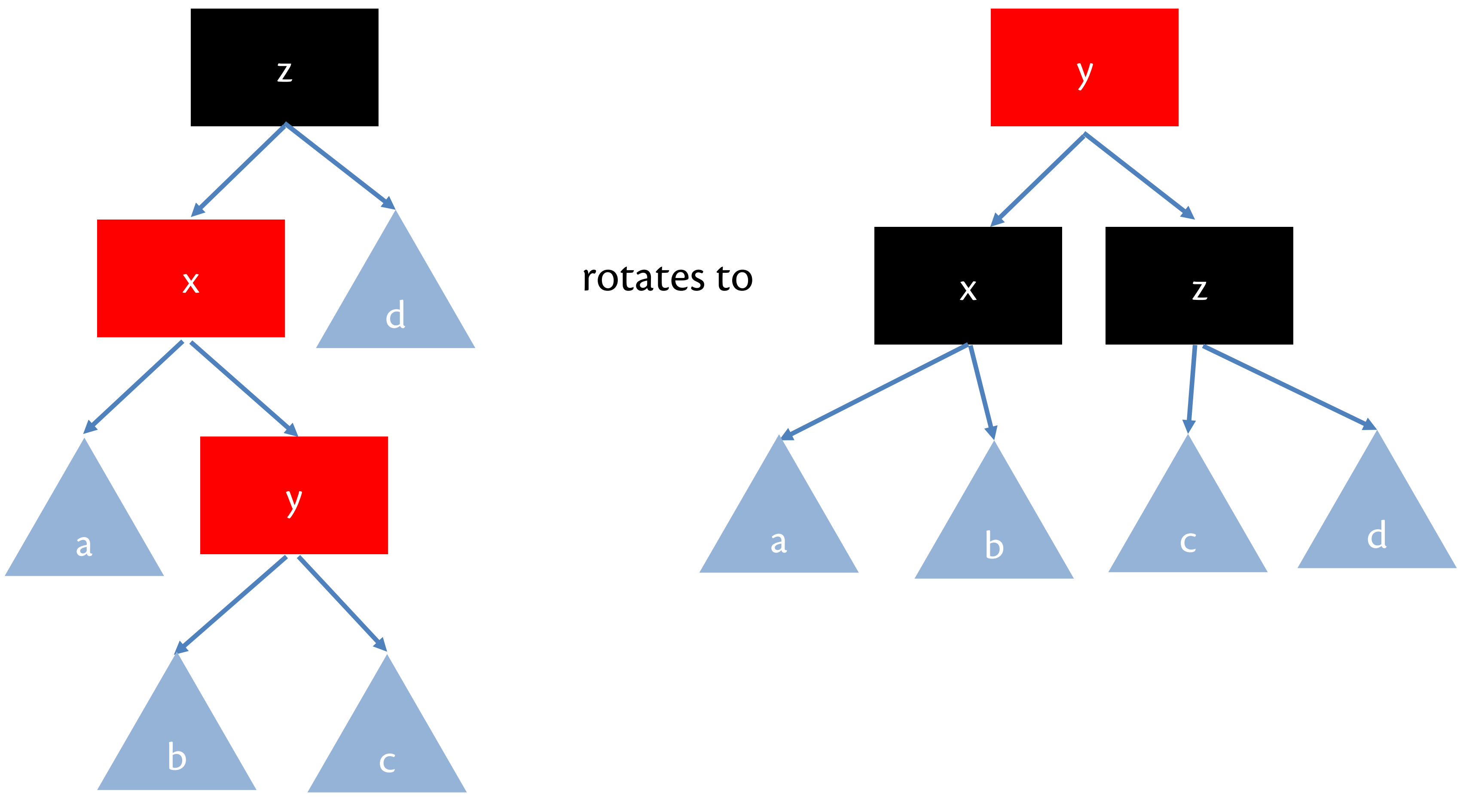


rotates to



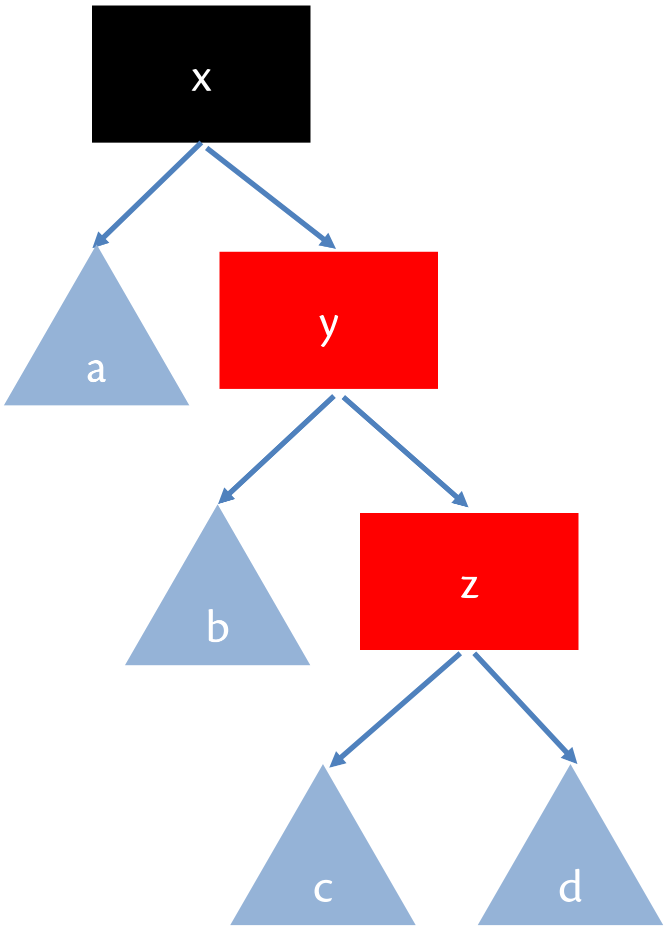
eliminates y-x violation
but maybe y has a red parent: new violation
keep recursing up tree

RB rotate (2 of 4)

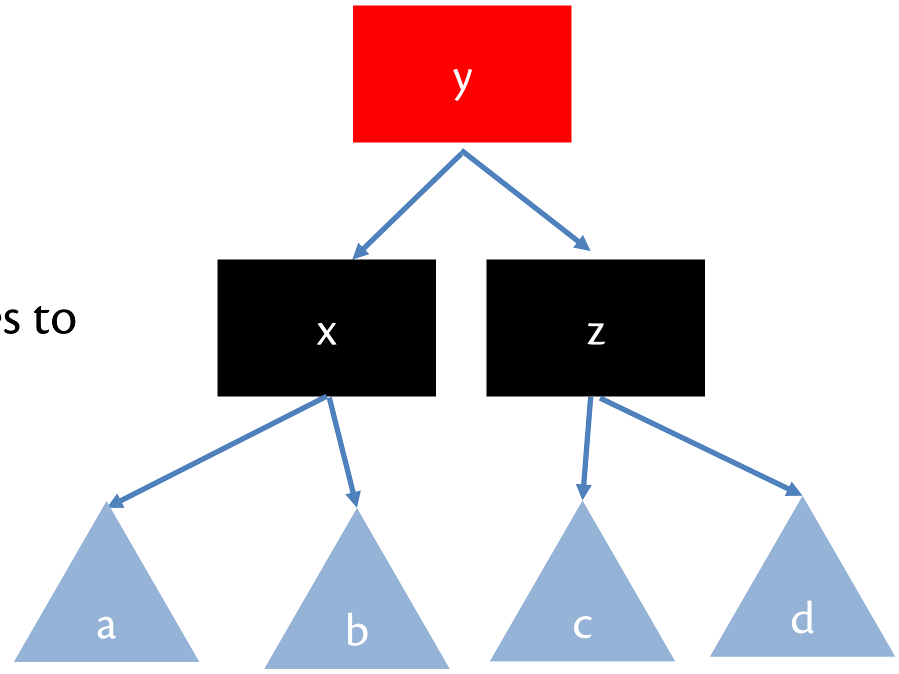


rotates to

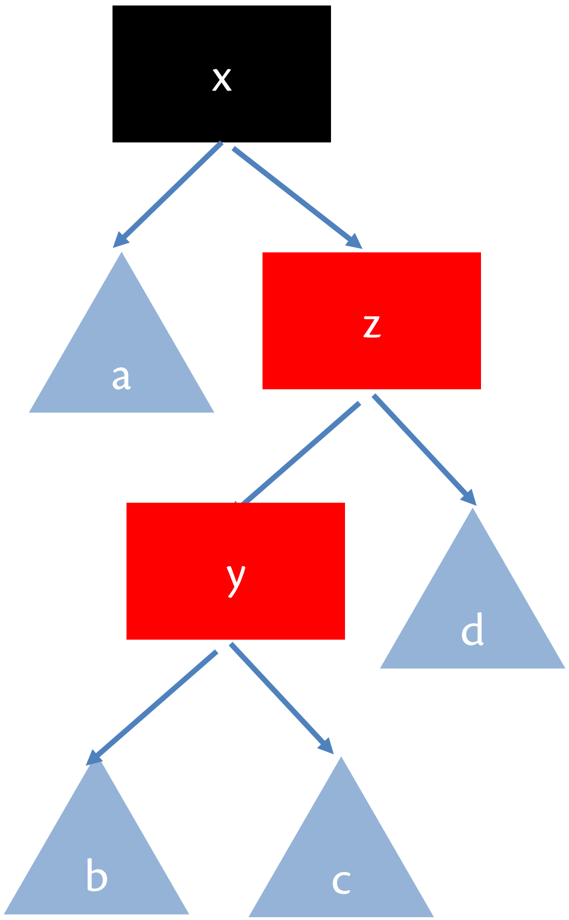
RB rotate (3 of 4)



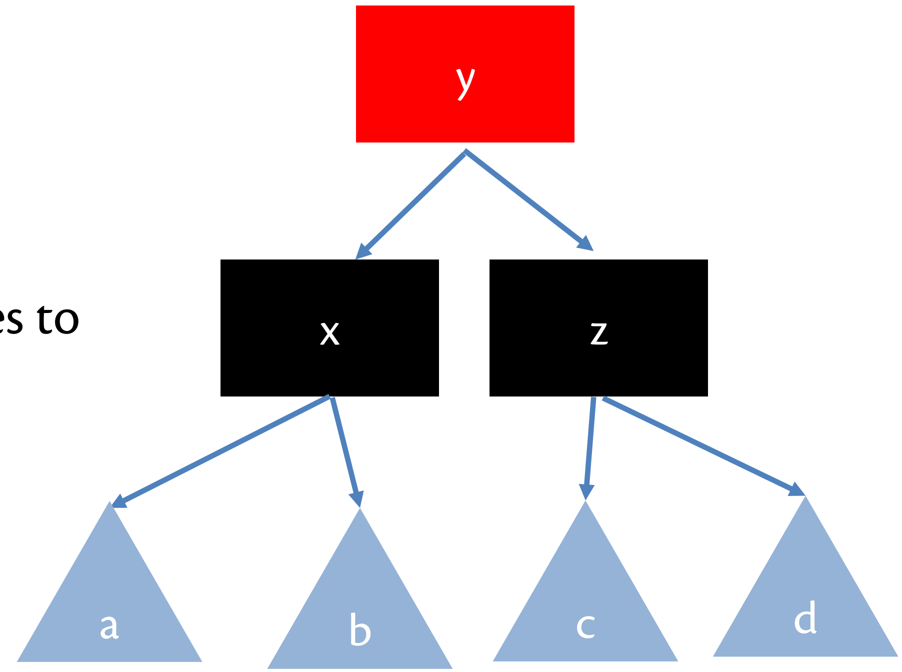
rotates to



RB rotate (4 of 4)



rotates to



RB balance

```
let balance = function
  | (Blk, Node (Red, Node (Red, a, x, b), y, c), z, d) (* 1 *)
  | (Blk, Node (Red, a, x, Node (Red, b, y, c)), z, d) (* 2 *)
  | (Blk, a, x, Node (Red, Node (Red, b, y, c), z, d)) (* 4 *)
  | (Blk, a, x, Node (Red, b, y, Node (Red, c, z, d))) (* 3 *)
    -> Node (Red, Node (Blk, a, x, b), y, Node (Blk, c, z, d))
  | t -> Node t
```

Upcoming events

- [Wed] A2 due
- [10/12] **Prelim**

This is blissfully balanced.

WE ARE GROOT

Upcoming events

- [Wed] A2 due
- [10/12] **Prelim**

This is blissfully balanced.

WE ARE 3110

Upcoming events

- [Wed] A2 due
- [10/12] **Prelim**

This is blissfully balanced.

THIS IS 3110

2-3 TREES

2-3 trees

- [Hopcroft 1970]

John Hopcroft [Gates 426]

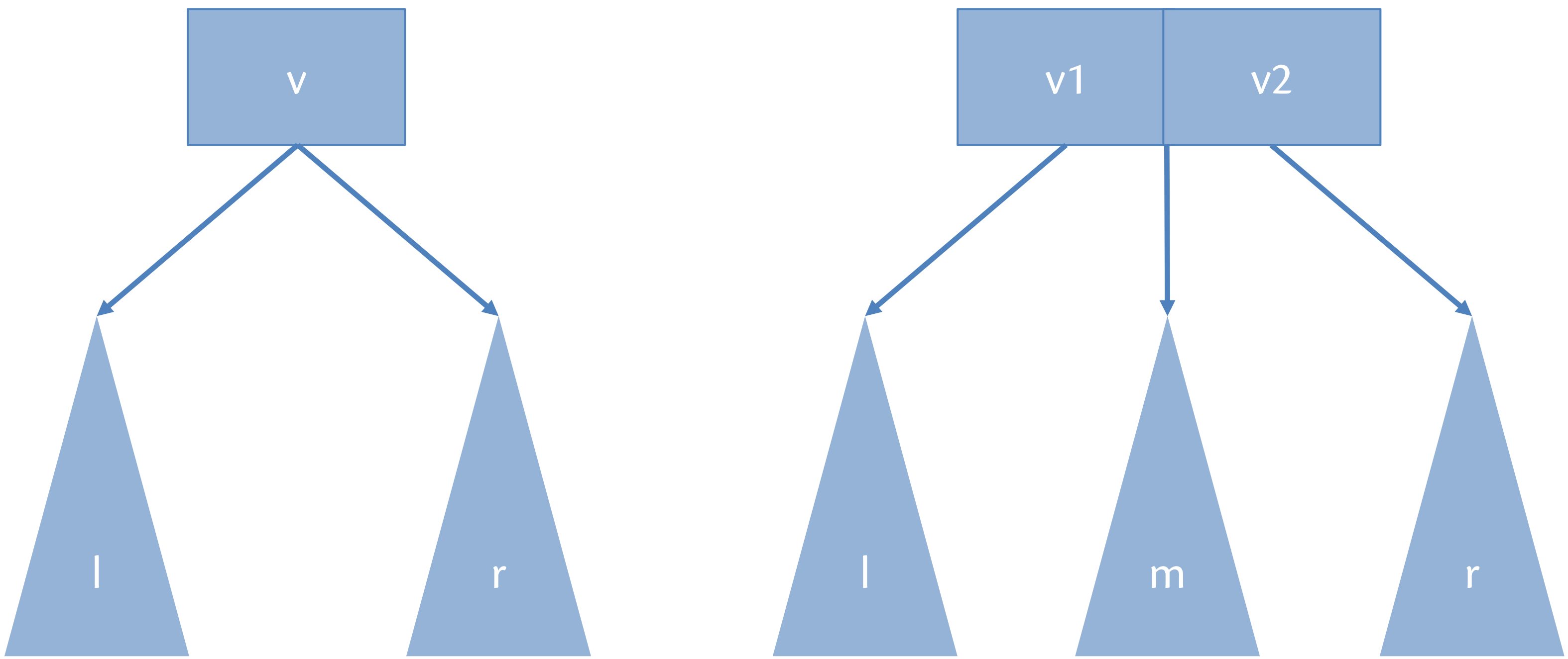
b. 1939

Turing Award Winner 1986

For fundamental achievements in the design and analysis of algorithms and data structures

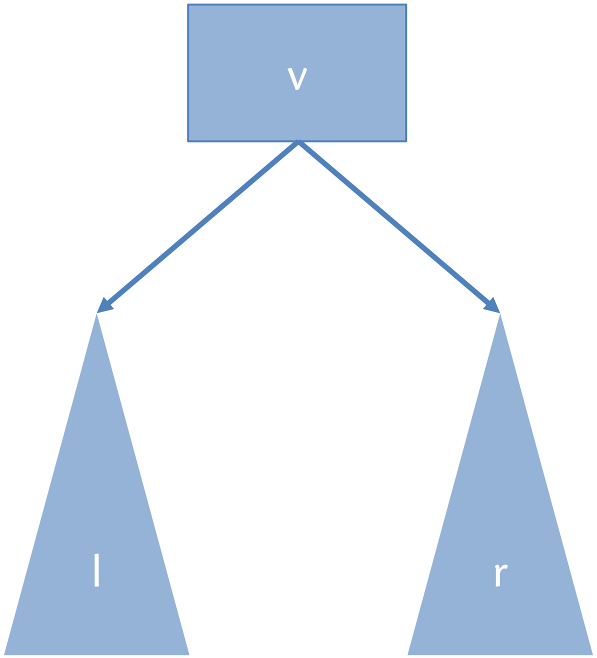
2-3 trees

- [Hopcroft 1970]
- Two kinds of nodes:



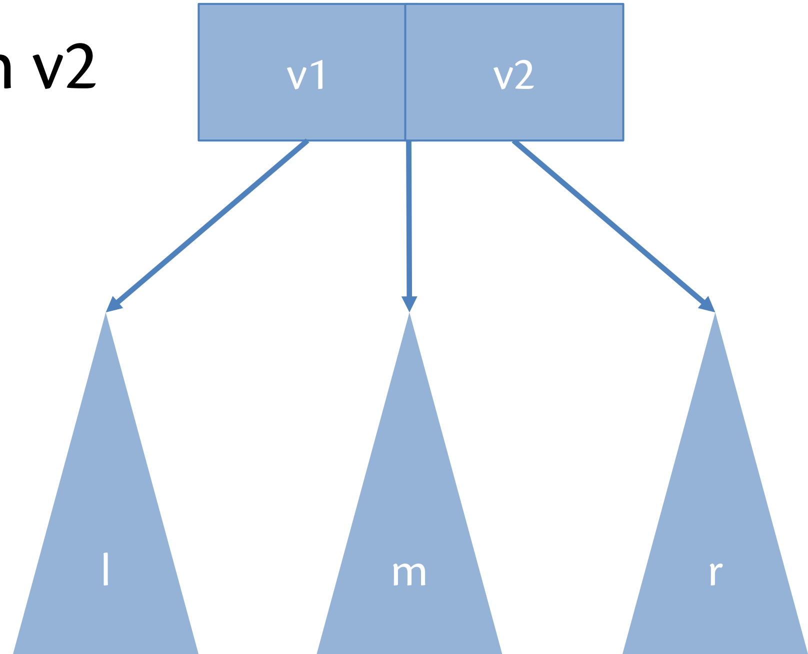
2-node

- Node contains one value and has two subtrees
- Obeys the BST invariant:
  - All values in l are less than v
  - All values in r are greater than v



3-node

- Node contains two values and has three subtrees
- Obeys something like the BST invariant:
  - All values in l are less than v1
  - All values in m are between v1 and v2
  - All values in r are greater than v2

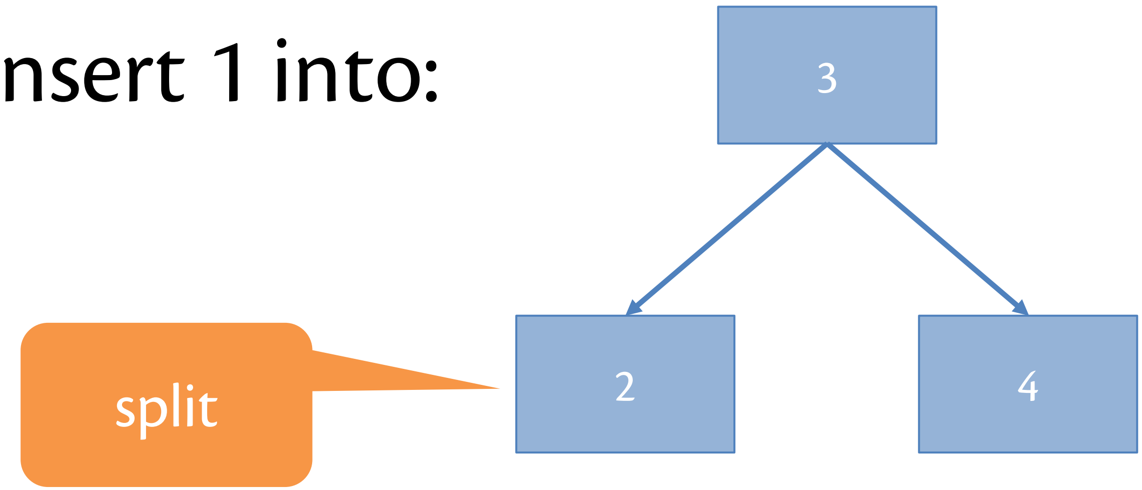


Inserting into 2-3 tree

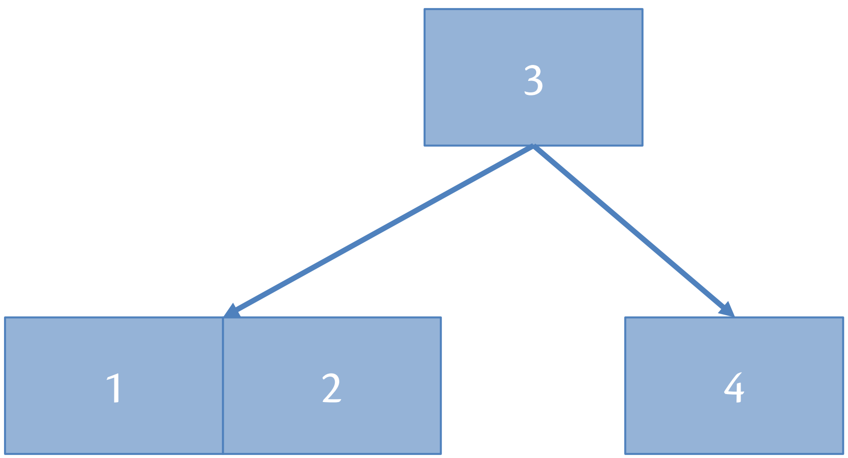
- Strategy: split nodes as necessary
- Complete algorithm too long to give in slides
- But you will implement it in A3!

Example of inserting into 2-3 tree

Insert 1 into:

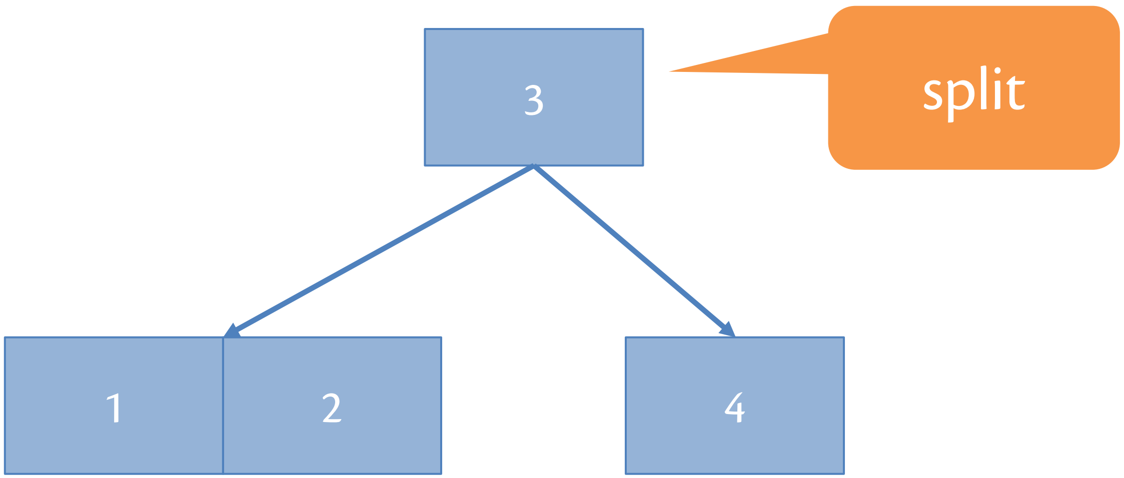


Result:



Example of inserting into 2-3 tree

Insert 0 into:



Result: