



<https://algs4.cs.princeton.edu>

3.3 BALANCED SEARCH TREES

- ▶ *2–3 search trees*
- ▶ *red–black BSTs*
- ▶ *B-trees (see book or videos)*

Symbol table review

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()
goal	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✓	compareTo()

Challenge. Guarantee performance.

optimized for teaching and coding;
introduced to the world in COS 226!

This lecture. 2–3 trees and left-leaning red–black BSTs.

co-invented by Bob Sedgwick



<https://algs4.cs.princeton.edu>

3.3 BALANCED SEARCH TREES

- ▶ *2-3 search trees*
- ▶ *red-black BSTs*
- ▶ *B-trees*

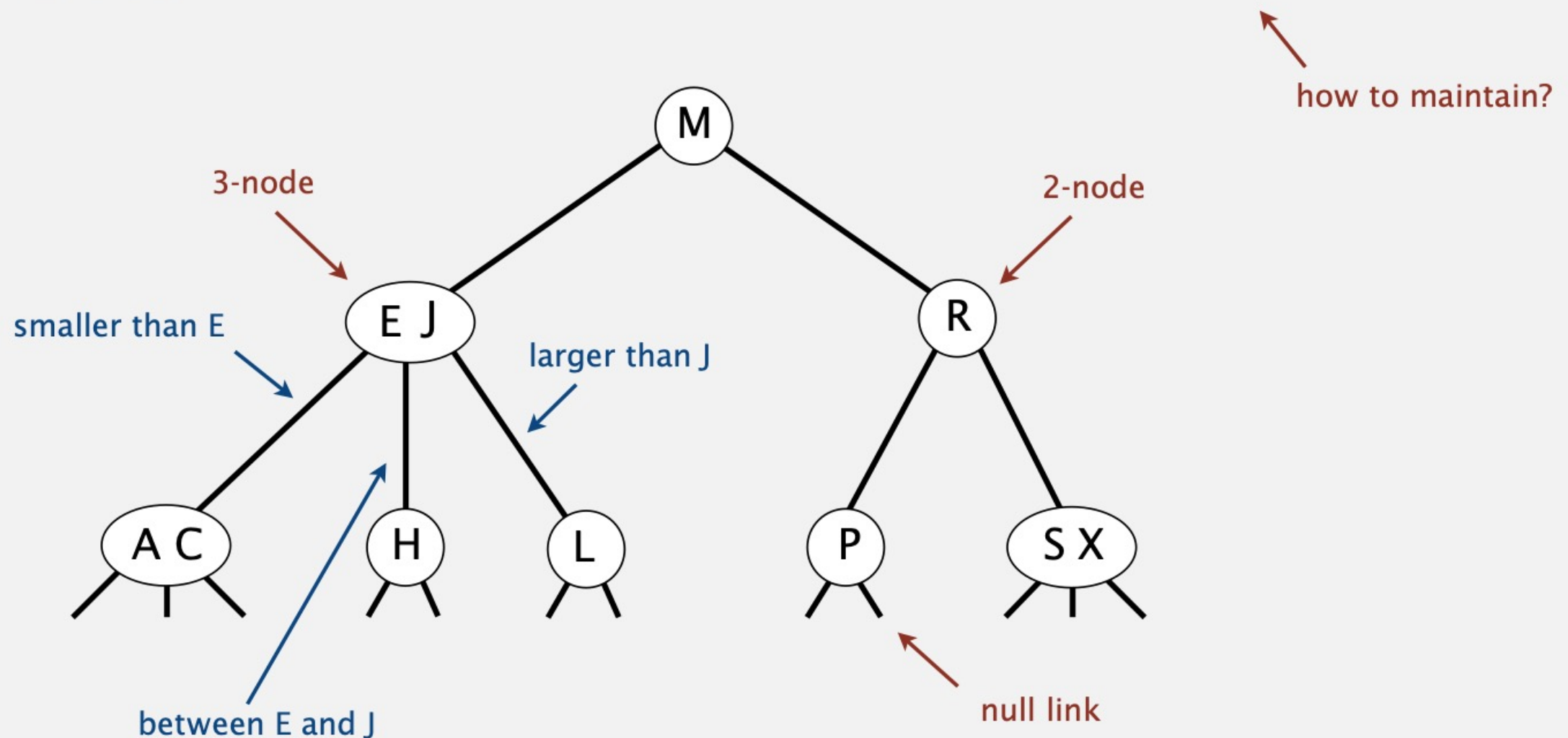
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



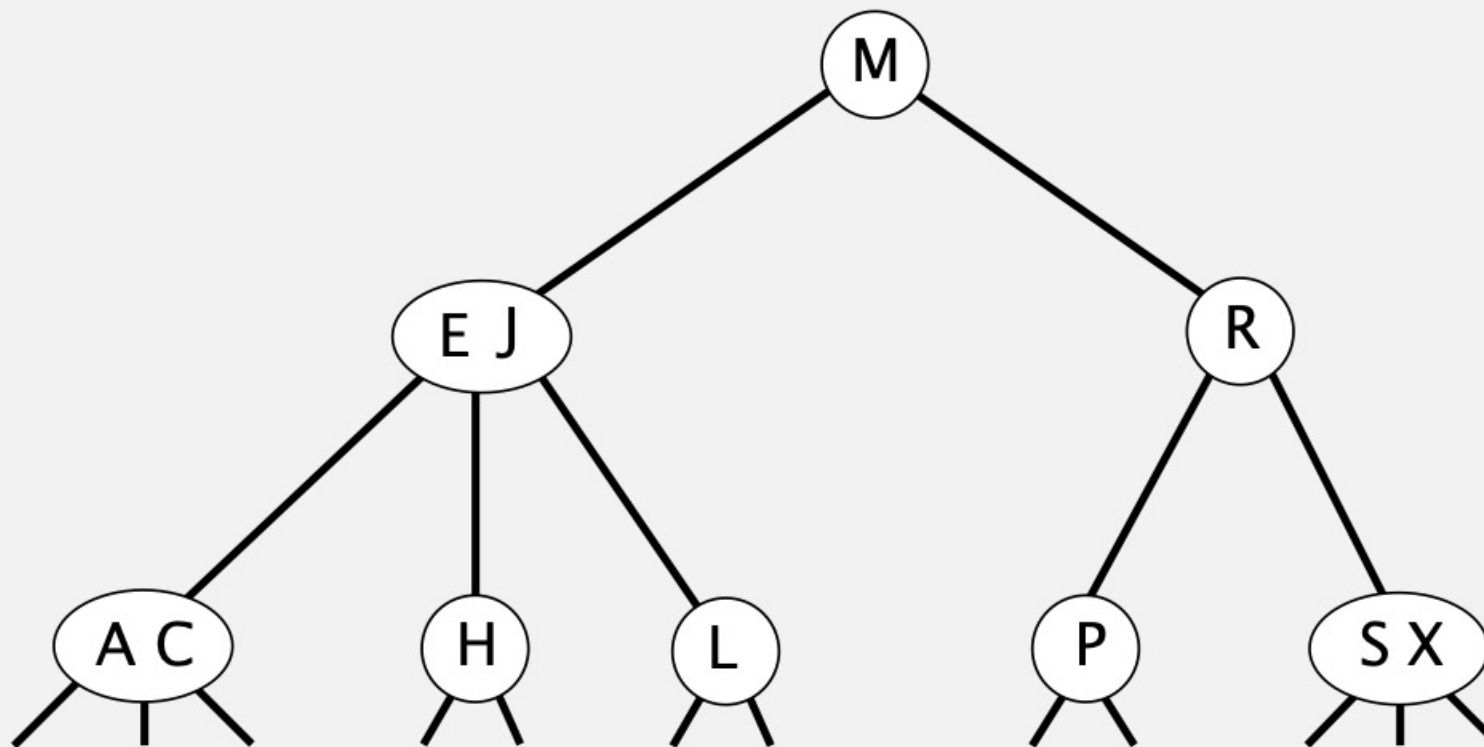
2-3 tree demo

Search.

- Compare search key against key(s) in node.
- Find interval containing search key.
- Follow associated link (recursively).



search for H

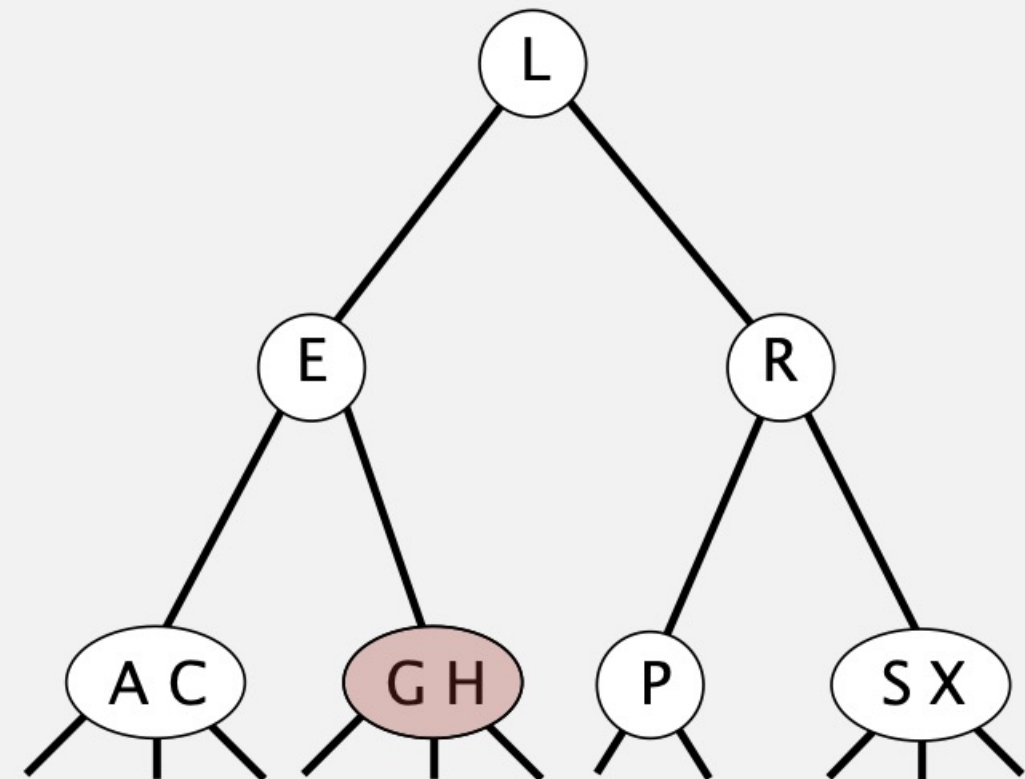
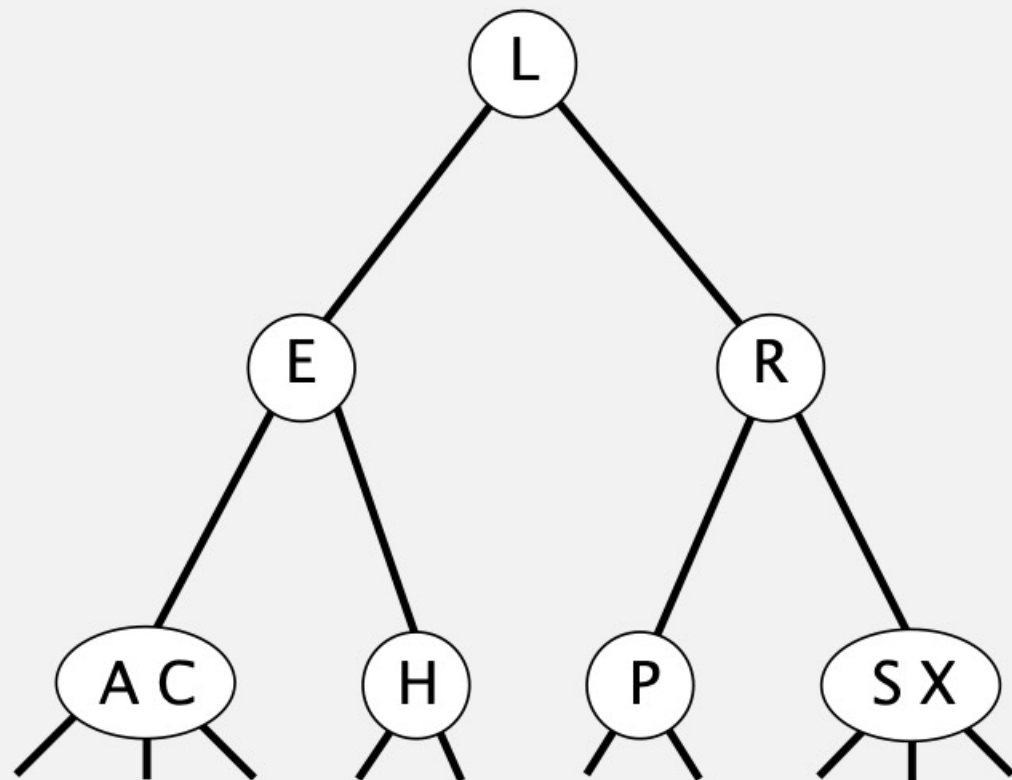


2-3 tree: insertion

Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

insert G

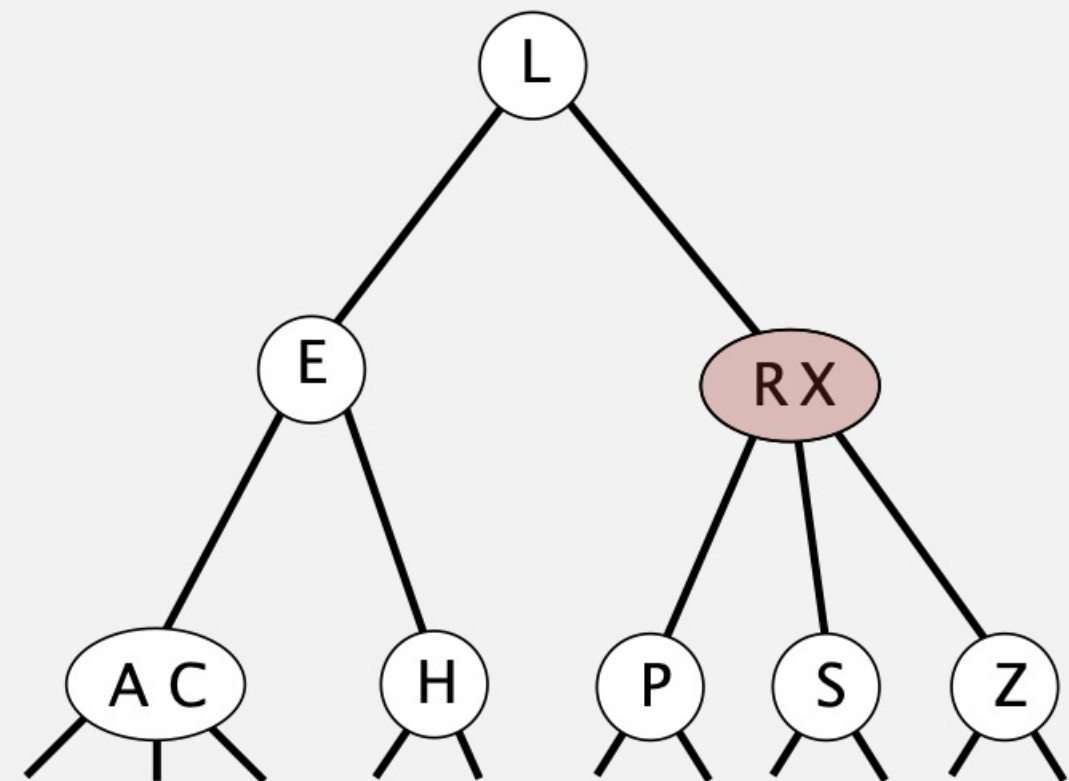
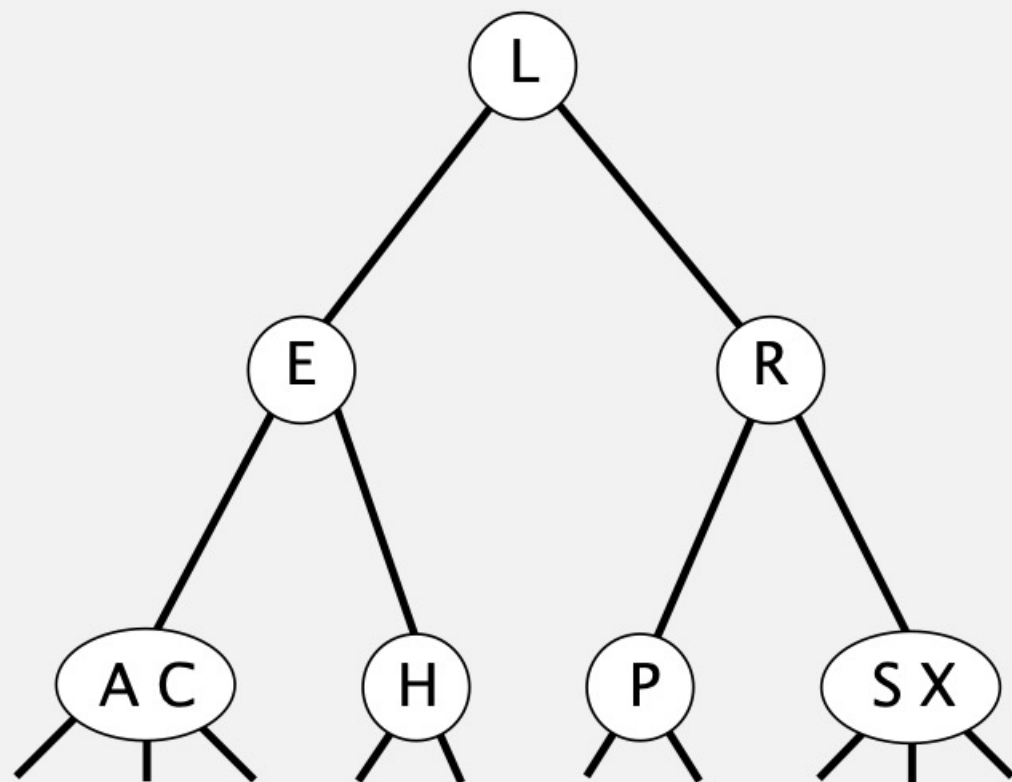


2-3 tree: insertion

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert Z



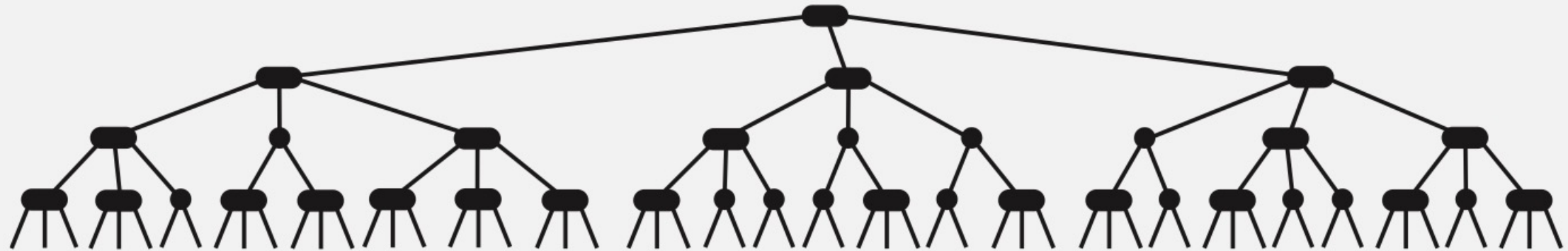


What is the maximum height of a 2–3 tree with n keys?

- A. $\sim \log_3 n$
- B. $\sim \log_2 n$
- C. $\sim 2 \log_2 n$
- D. $\sim n$

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



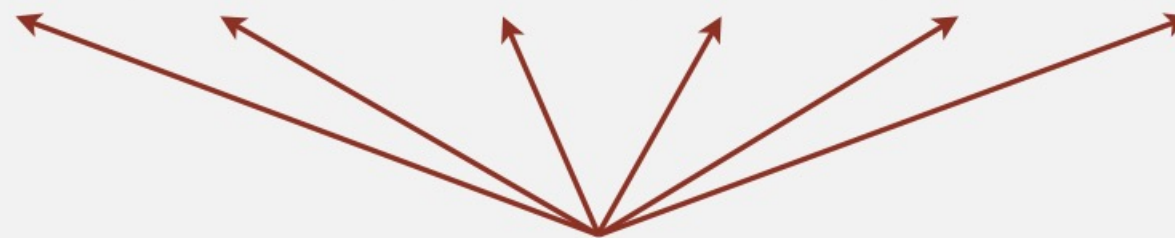
Tree height.

- Min: $\log_3 n \approx 0.631 \log_2 n.$ [all 3-nodes]
- Max: $\log_2 n.$ [all 2-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed **logarithmic** performance for search and insert.

ST implementations: 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()



but hidden constant c is large
(depends upon implementation)

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

fantasy code

```
public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.