# 3.3 Balanced Search Trees


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

# Symbol table review

| implementation | guarantee | | | average case | | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | N/2 | N | N/2 | no | `equals()` |
| binary search (ordered array) | lg N | N | N | lg N | N/2 | N/2 | yes | `compareTo()` |
| BST | N | N | N | 1.39 lg N | 1.39 lg N | ? | yes | `compareTo()` |
| goal | log N | log N | log N | log N | log N | log N | yes | `compareTo()` |

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.

introduced to the world
in COS 226, Fall 2007

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

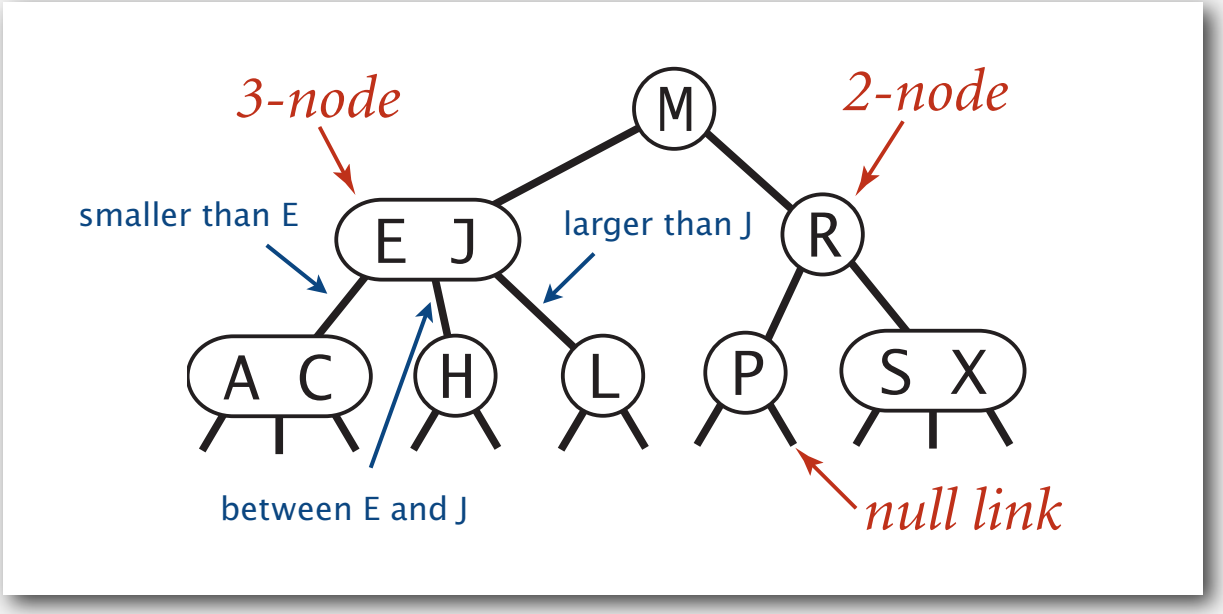
# 2-3 tree

Allow 1 or 2 keys per node.

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

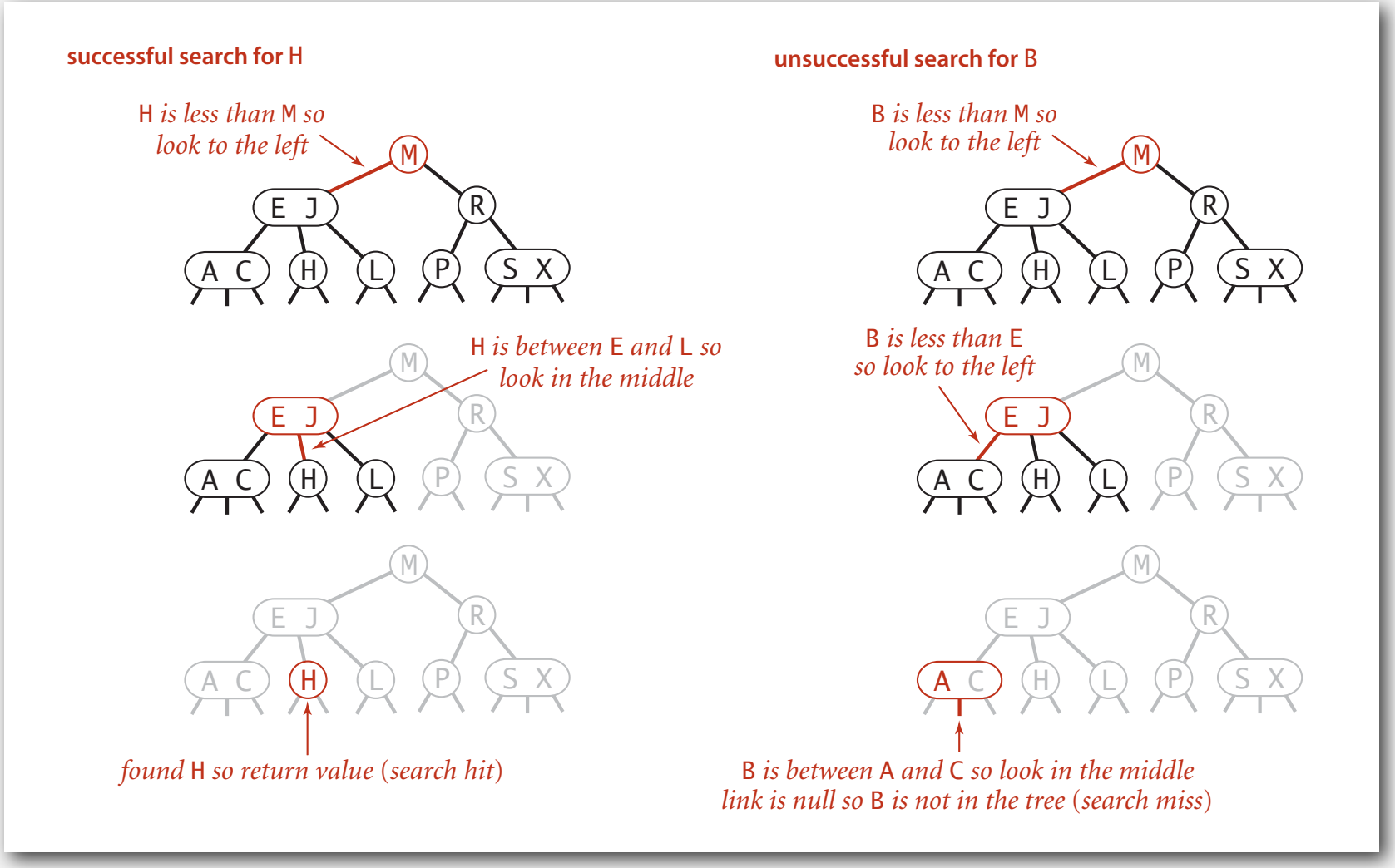
Symmetric order. Inorder traversal yields keys in ascending order.

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



# Search in a 2-3 tree

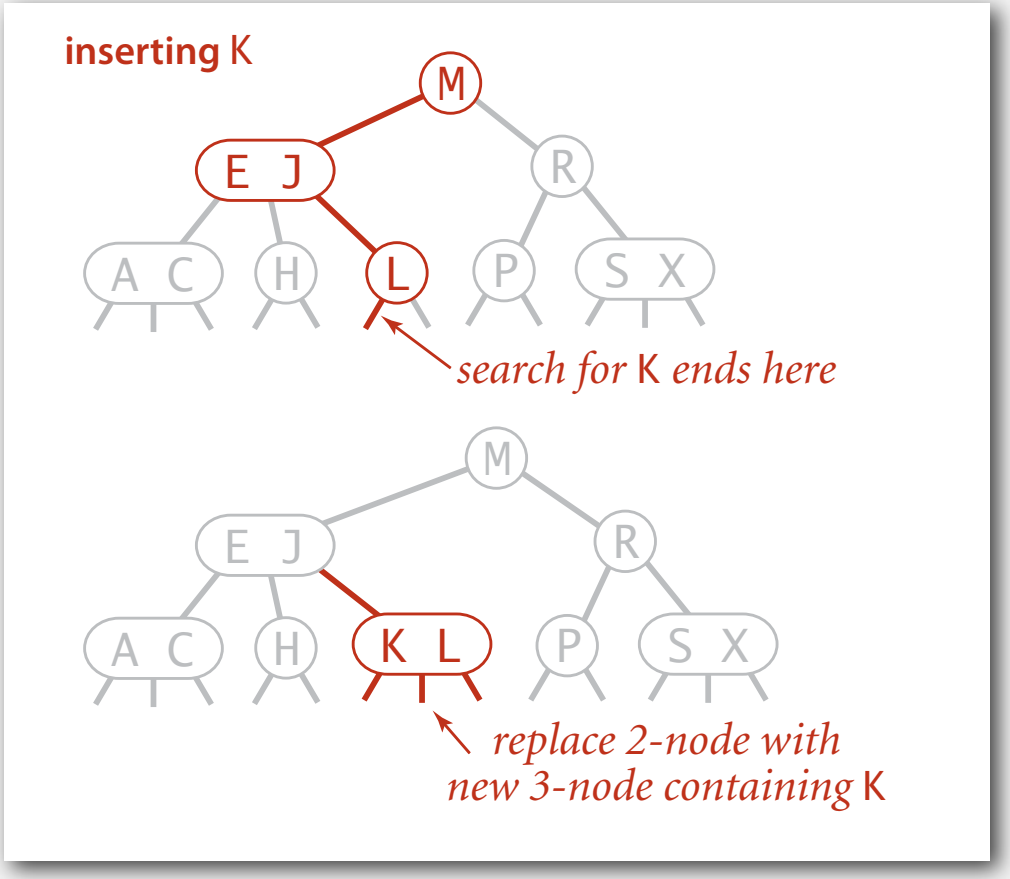
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



# Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

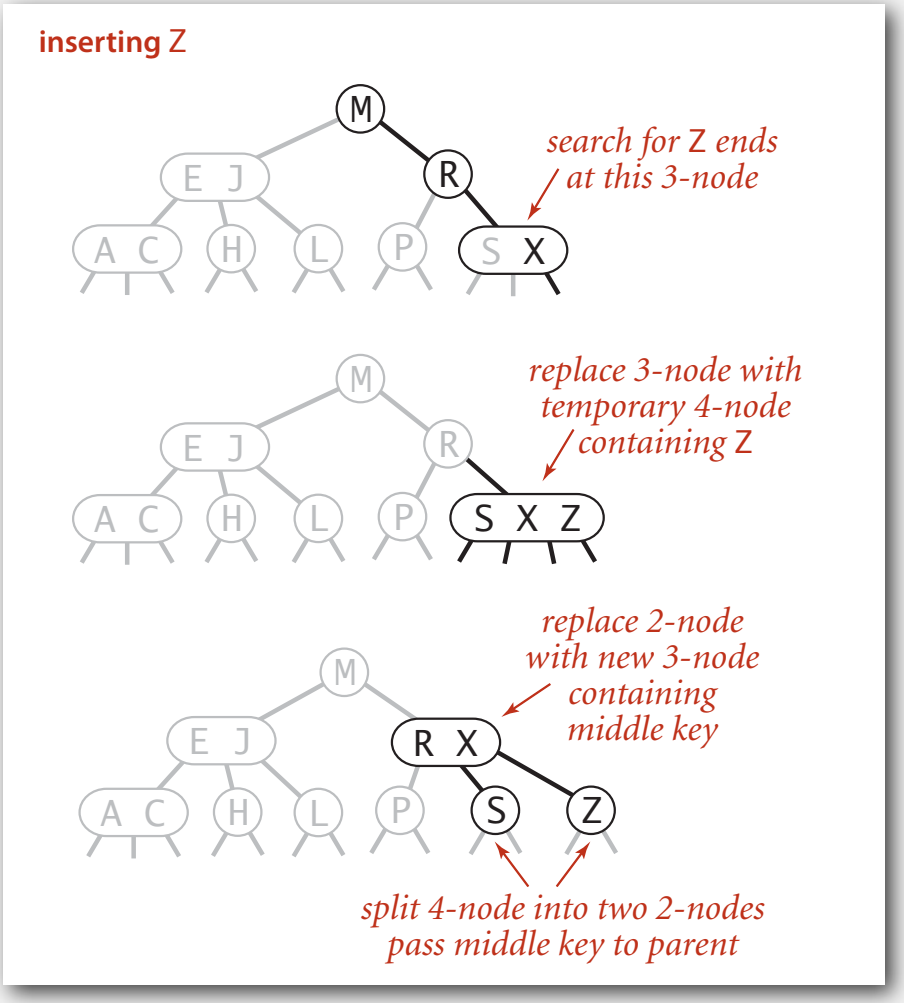


# Insertion in a 2-3 tree

Case 2. Insert 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.

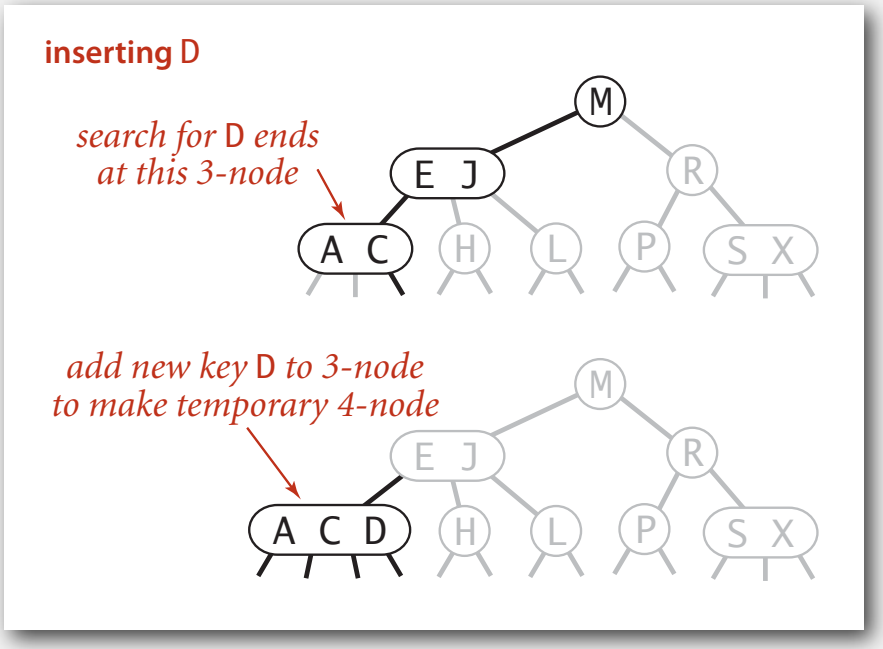
why middle key?

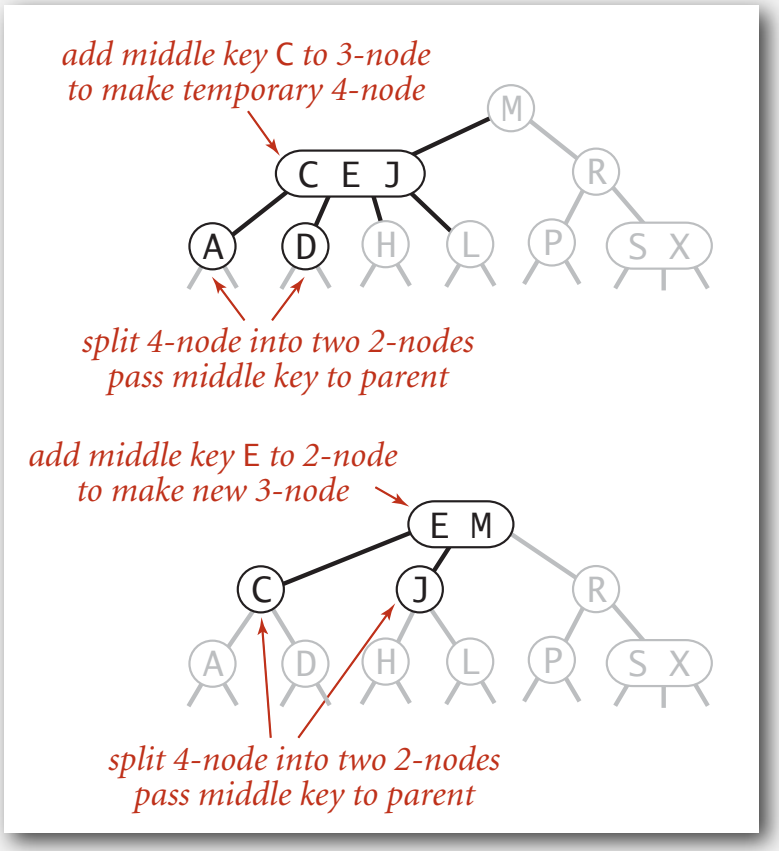


# Insertion in a 2-3 tree

Case 2. Insert 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.

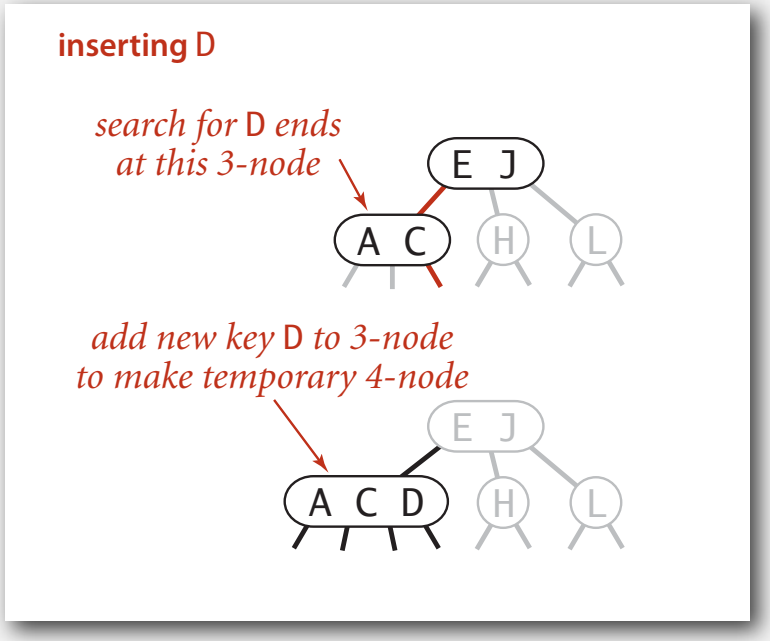


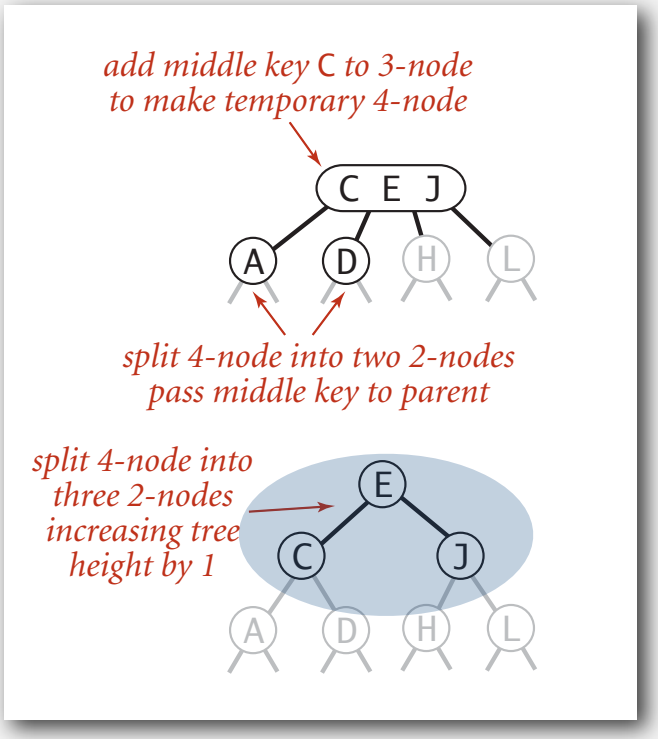



# Insertion in a 2-3 tree

Case 2. Insert 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.

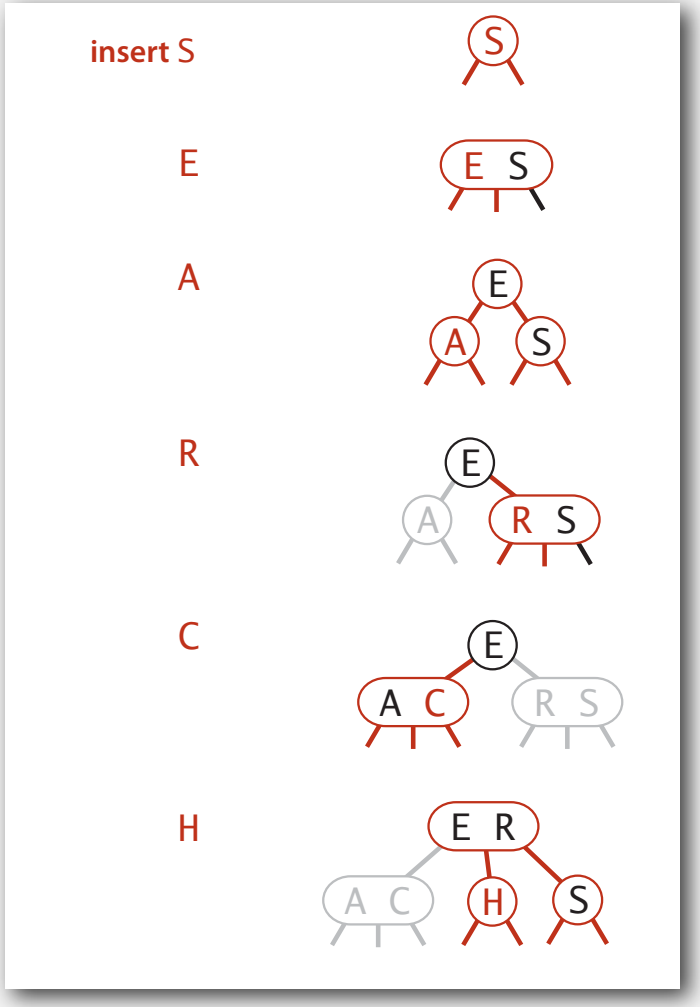


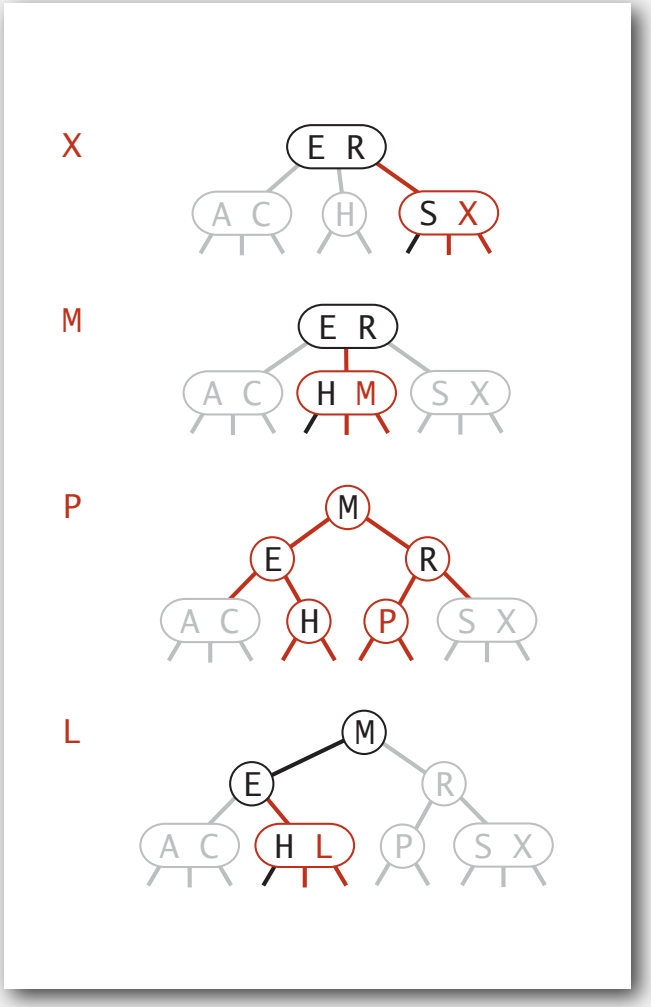




Remark. Splitting the root increases height by 1.

# 2-3 tree construction trace

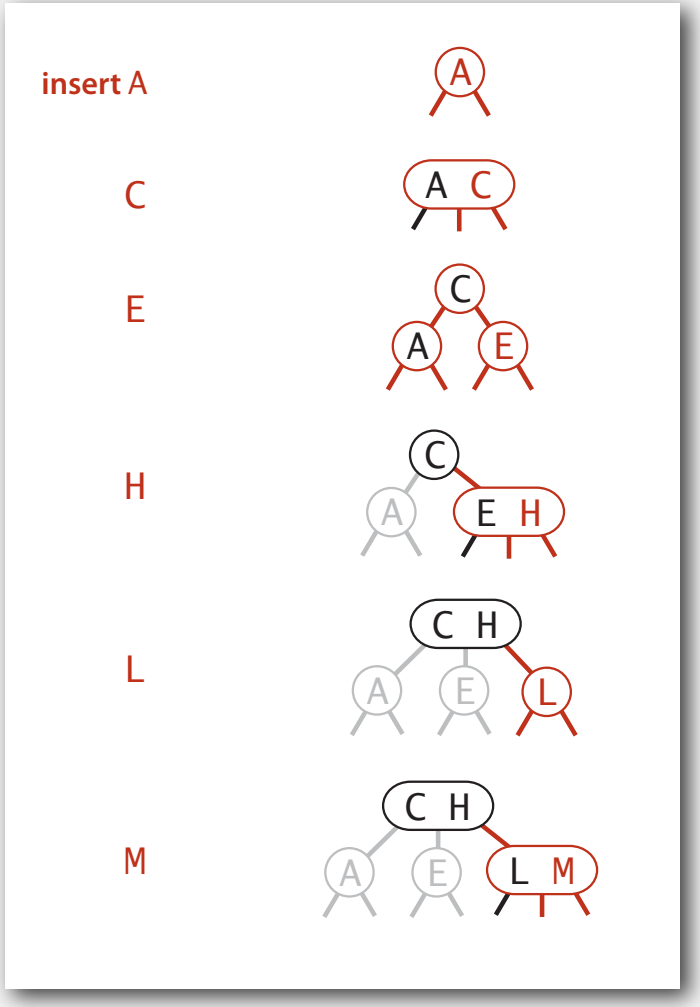
Standard indexing client.

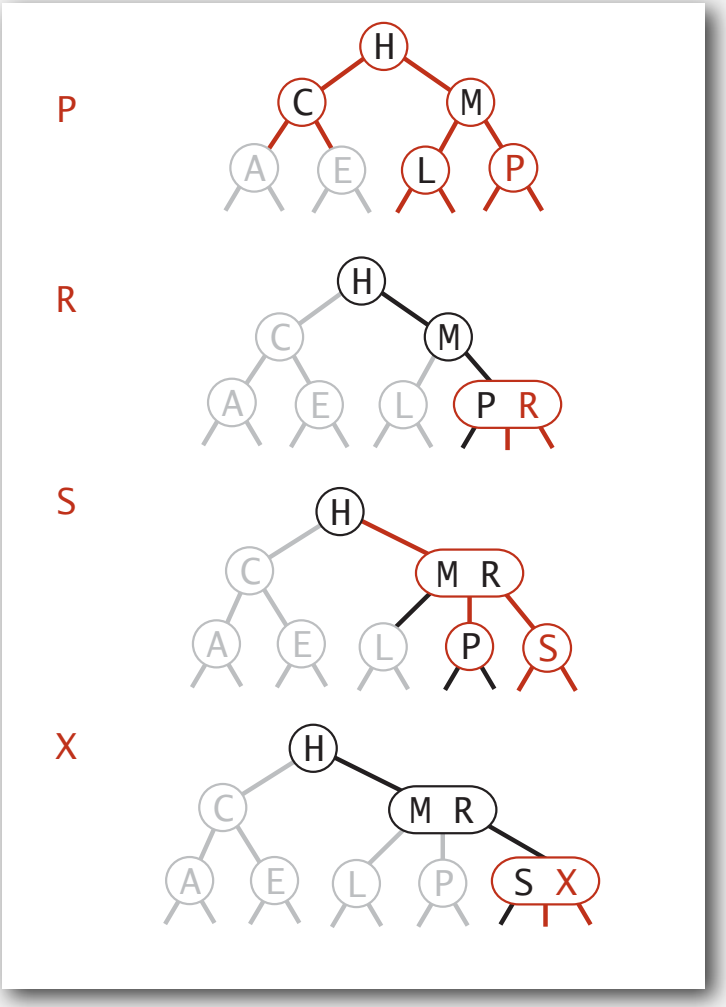


# 2-3 tree construction trace

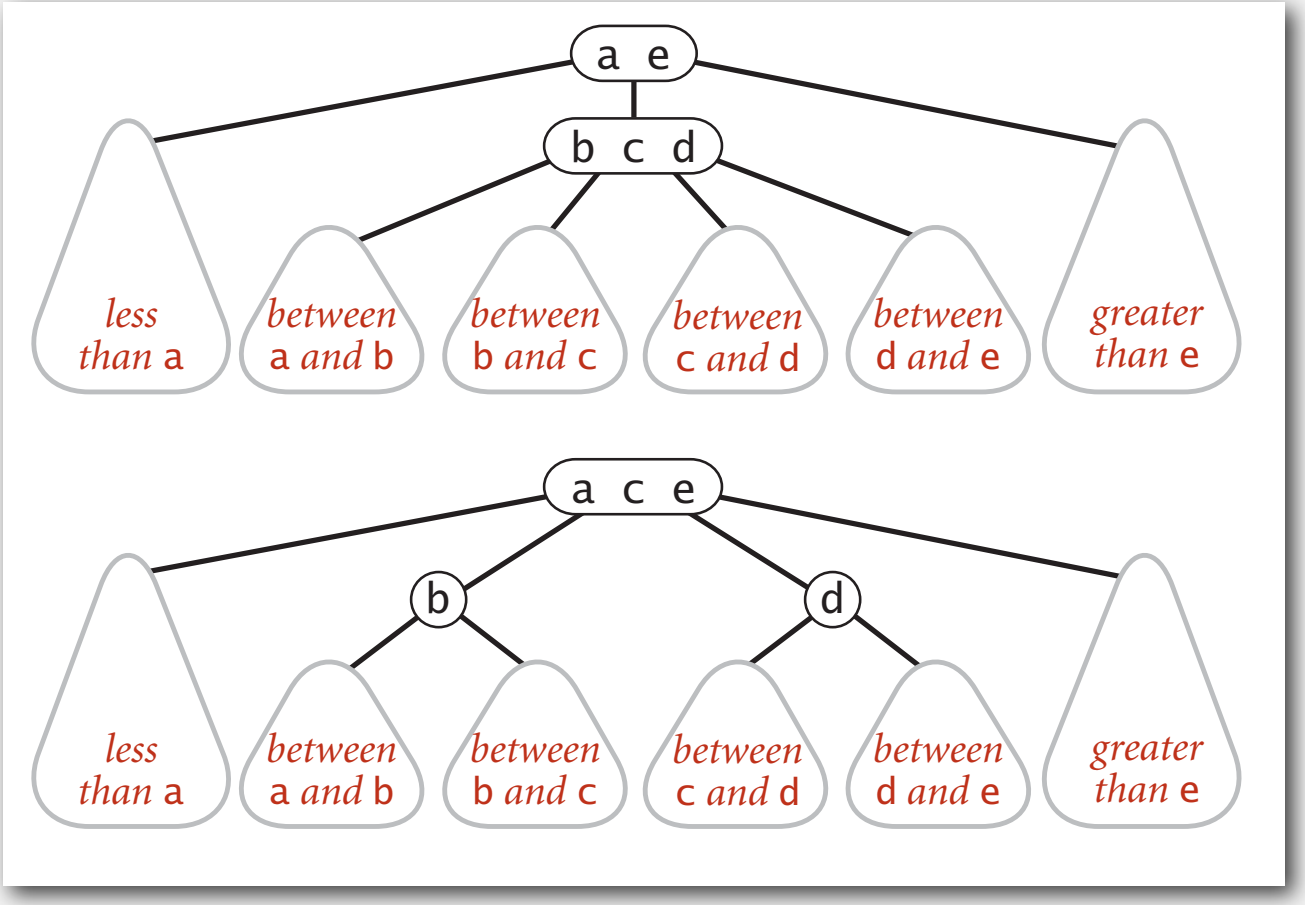
The same keys inserted in ascending order.



# Local transformations in a 2-3 tree

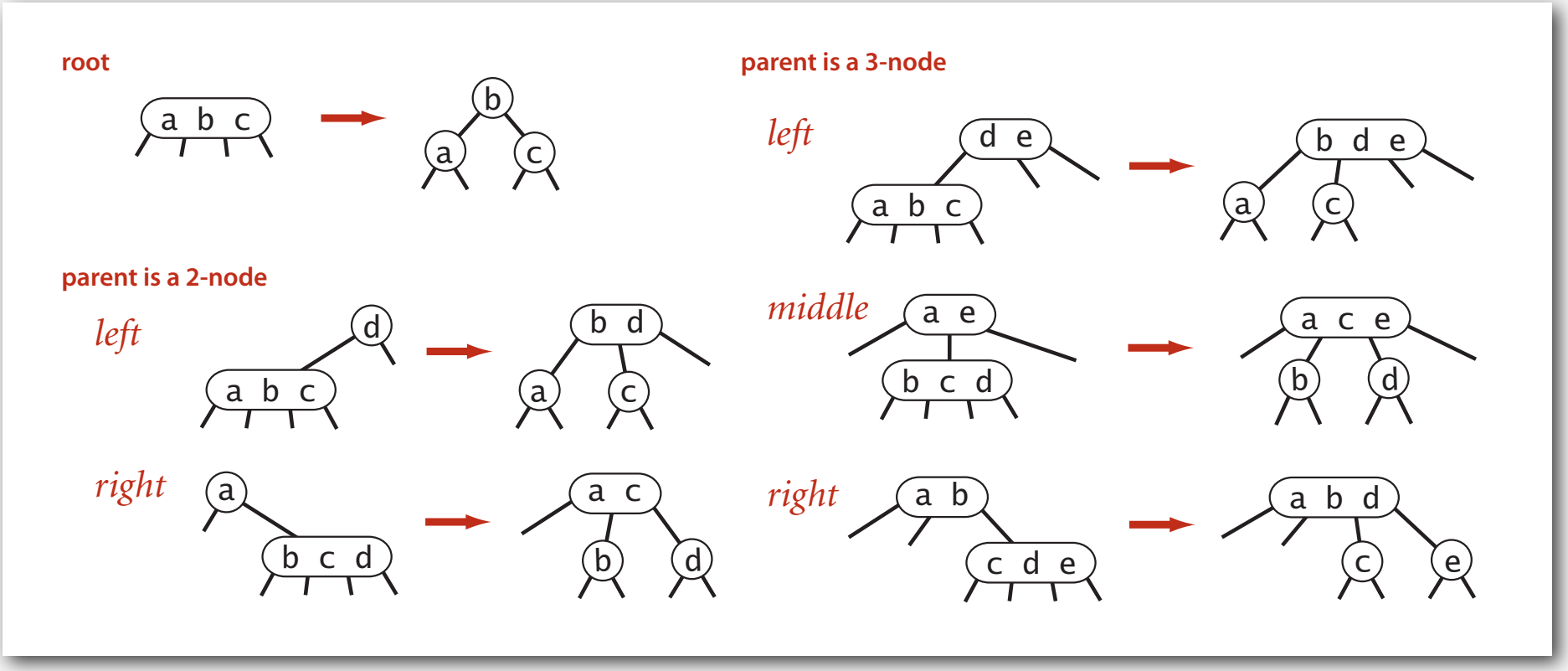
Splitting a 4-node is a local transformation: constant number of operations.



# Global properties in a 2-3 tree

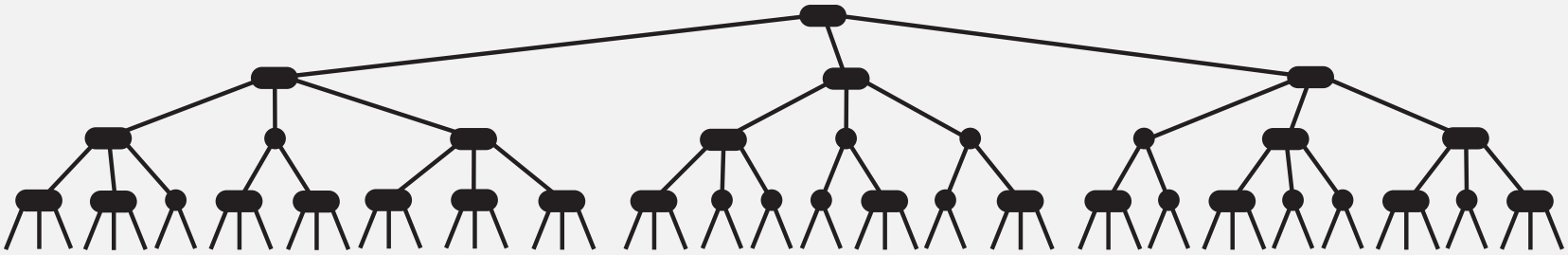
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



## 2-3 tree: performance

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



Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

# ST implementations: summary

| implementation | guarantee | | | average case | | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | N/2 | N | N/2 | no | `equals()` |
| binary search (ordered array) | lg N | N | N | lg N | N/2 | N/2 | yes | `compareTo()` |
| BST | N | N | N | 1.39 lg N | 1.39 lg N | ? | yes | `compareTo()` |
| 2-3 tree | c lg N | c lg N | c lg N | c lg N | c lg N | c lg N | yes | `compareTo()` |

constants depend upon implementation

## 2-3 tree: implementation?

Direct implementation is complicated, because:

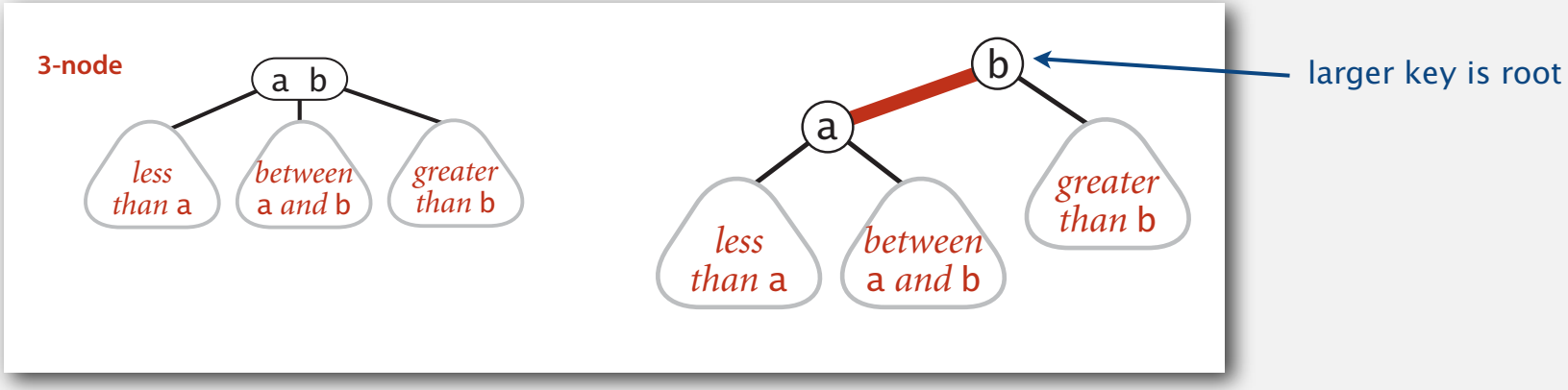
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

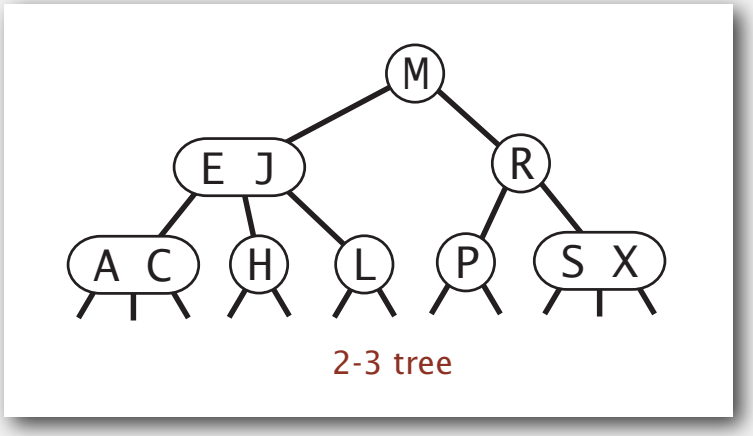
Bottom line. Could do it, but there's a better way.

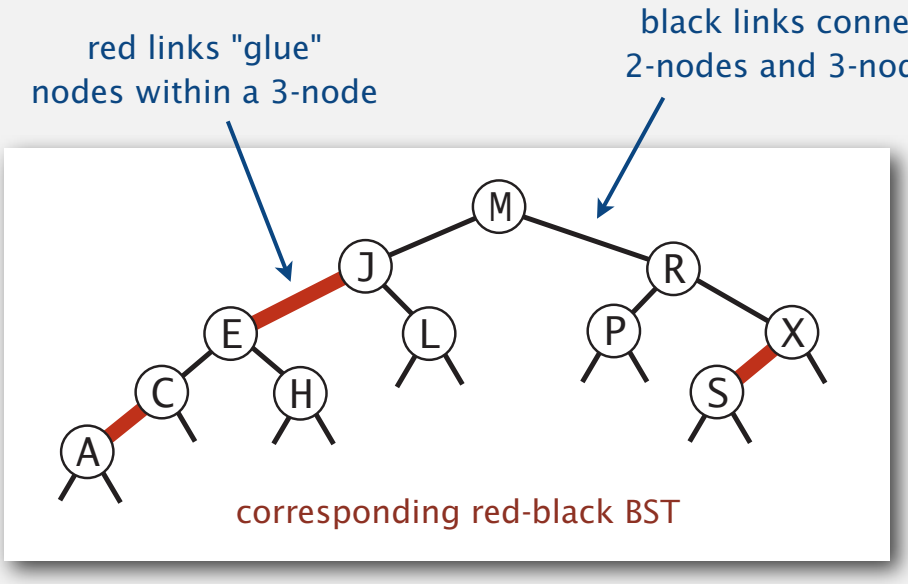
- 2-3 search trees
- **red-black BSTs**
- B-trees

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

1. Represent 2-3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.



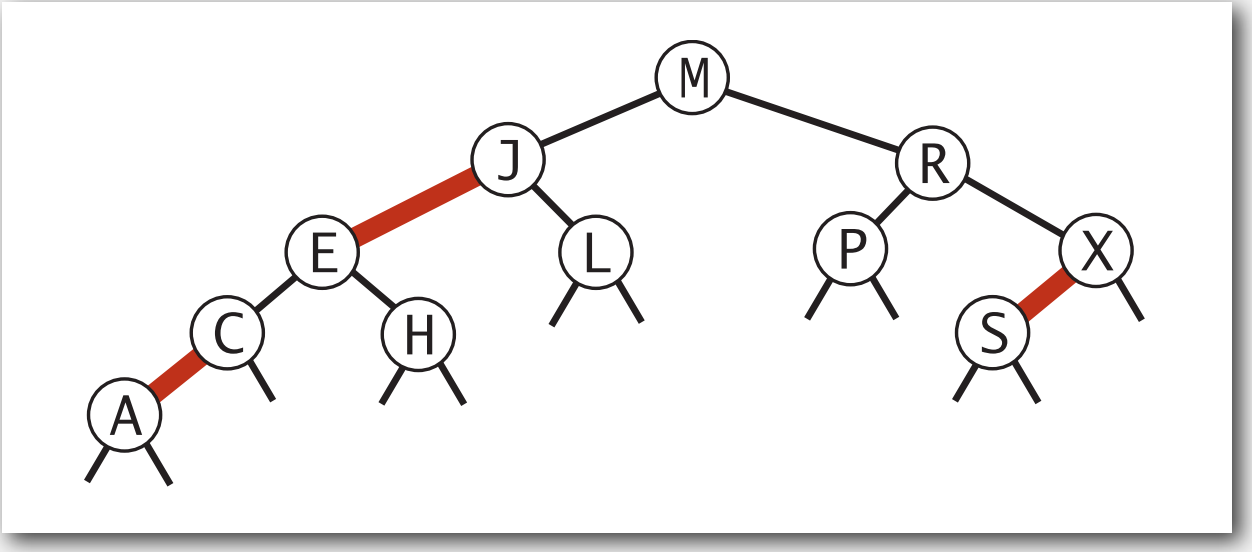



# An equivalent definition

A BST such that:

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

"perfect black balance"



# Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.