COMP20230: Data Structures & Algorithms
Lecture 19: Trees (2)

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Searching Trees

Outline

Trade offs: Searching, Inserting, Deleting, Updating
Using different tree data structures can facilitate algorithms that prioritise different operations

Balanced Search Trees trees (not on exam)

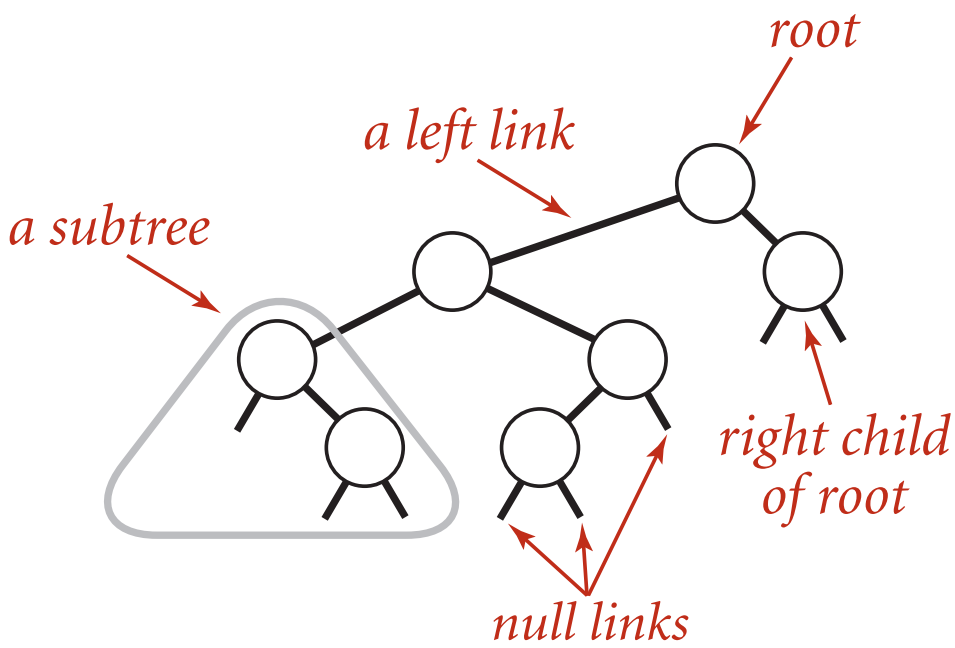
We will introduce them for context

Binary Trees

Binary Tree

A tree where each node has up to two leaves

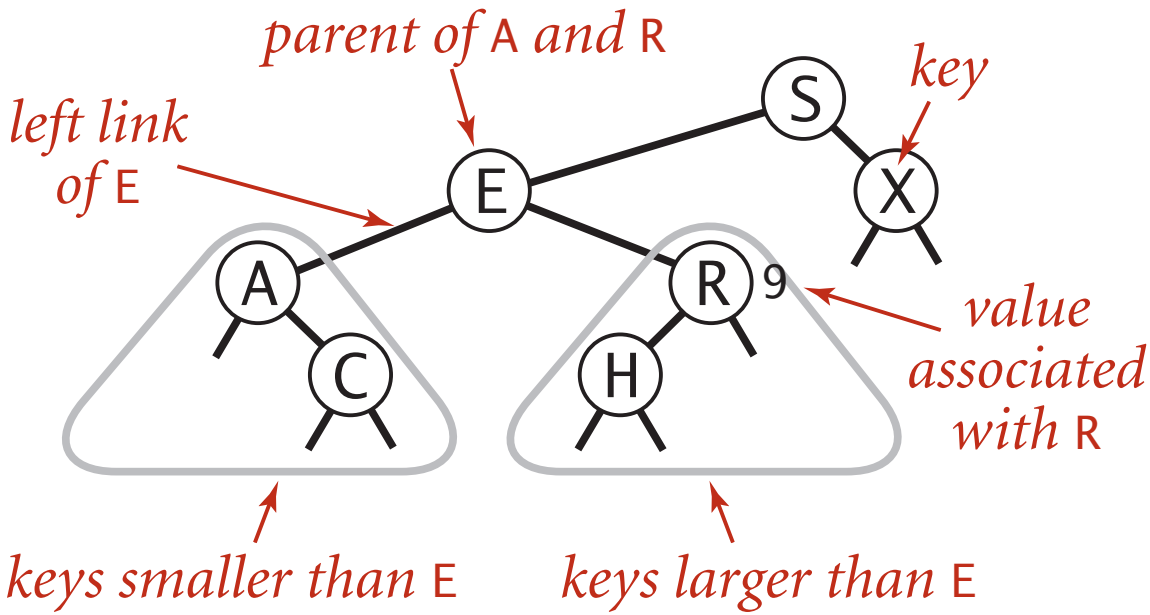
A binary tree node is either **empty** or has **two disjoint** binary sub-trees (left and right)



Binary Search Trees

A binary trees used for searching

Left child contains only nodes with values less than the parent
Right child only contains nodes with values greater than or equal to the parent

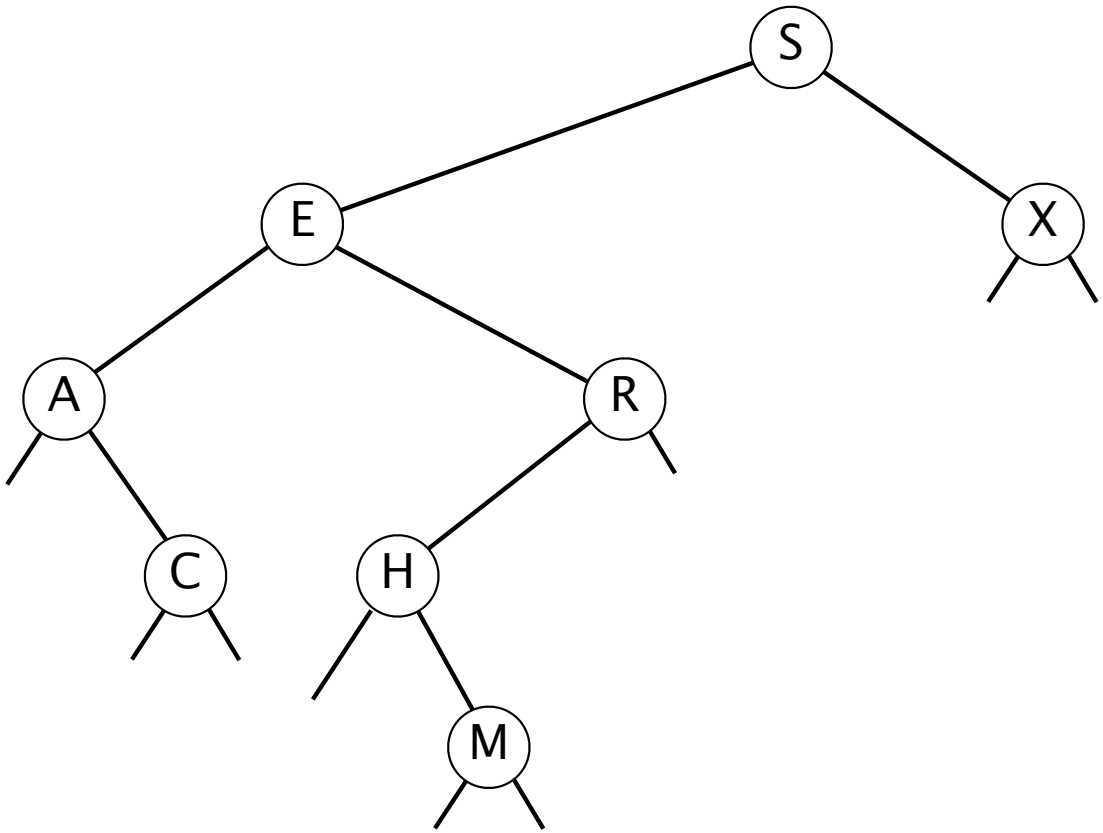


Binary Search Trees

Searching in a BST

If less, go left; if greater, go right; if equal, search hit.

search for H

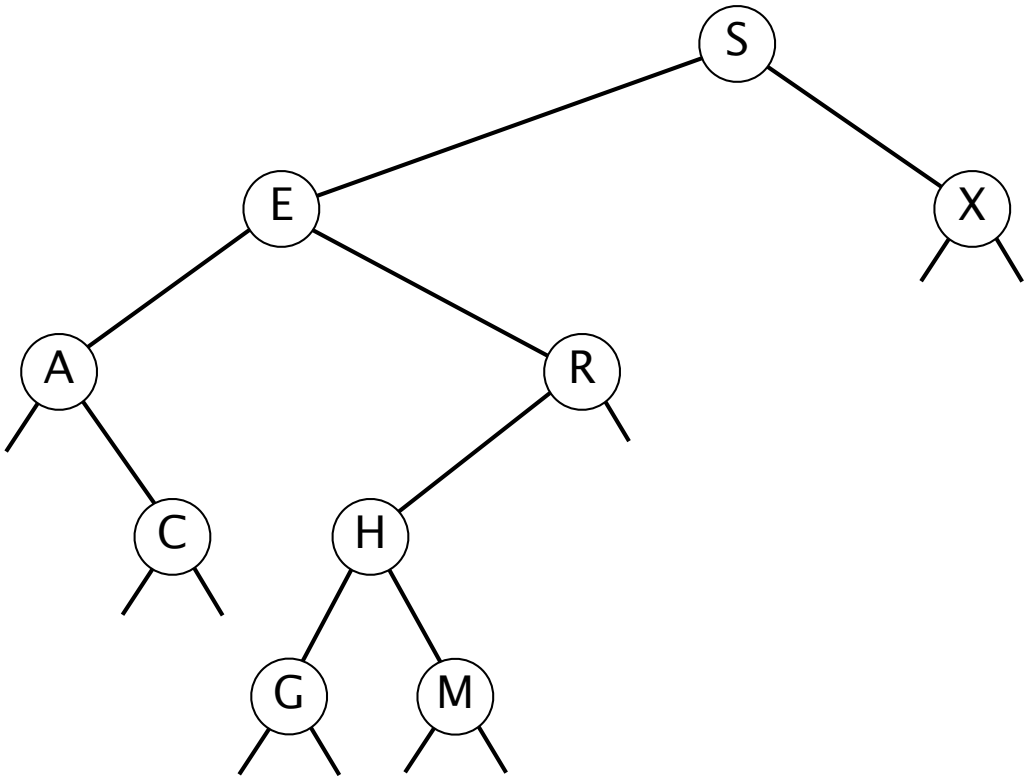


Binary Search Trees

Insert in a BST

If less, go left; if greater, go right; if null, insert.

insert G



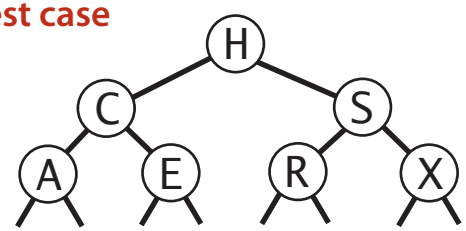
Binary Search Trees

Shape

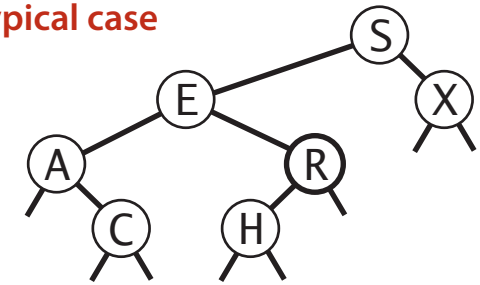
Many BSTs correspond to the same set of keys

# of comparisons for search/insert $= 1 +$ *node_depth*.

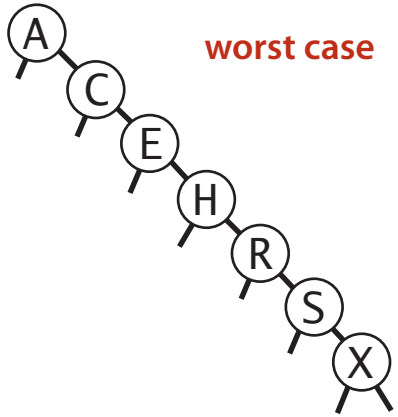
best case



typical case

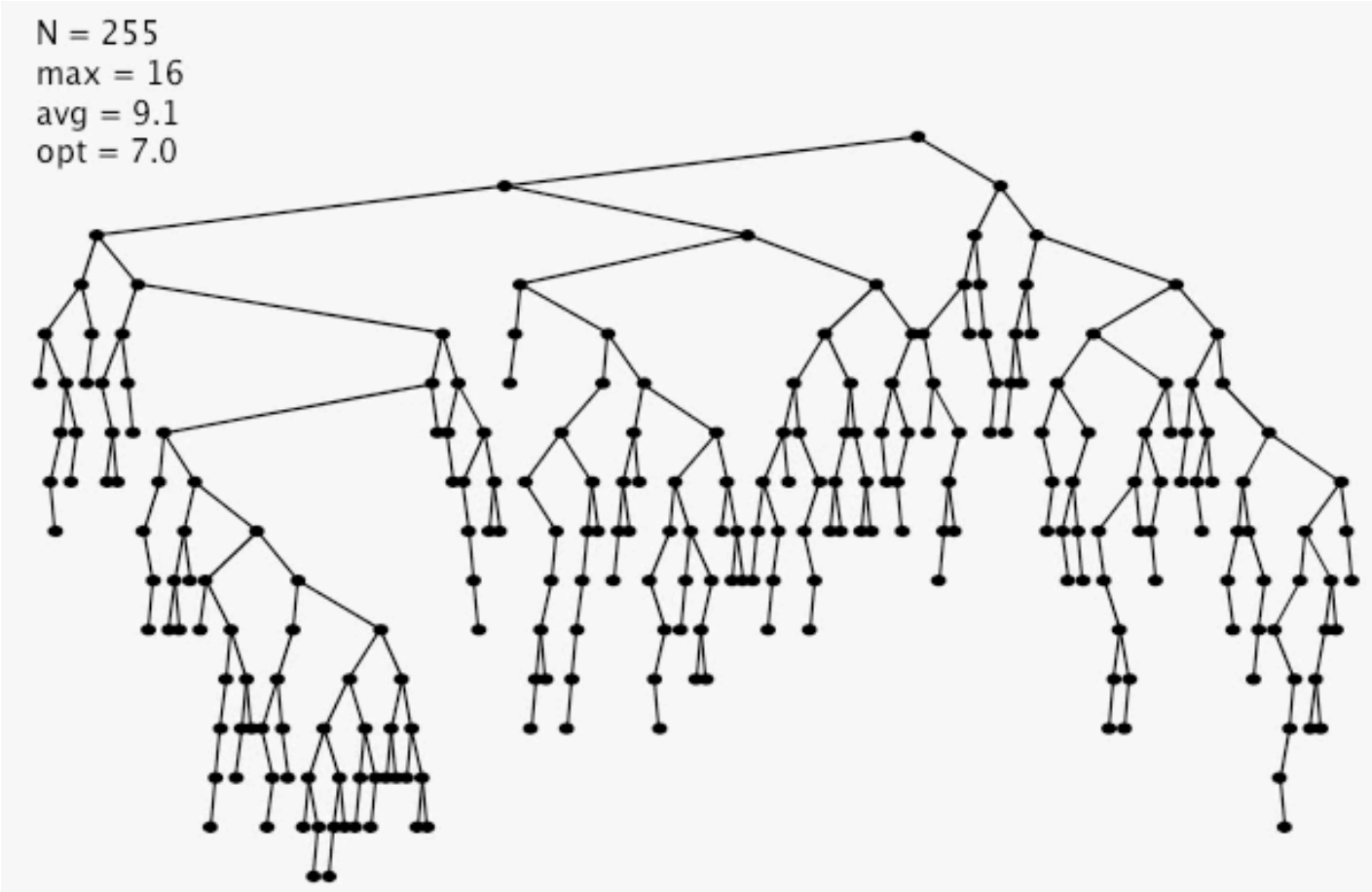


worst case



BST Insertion Visualisation

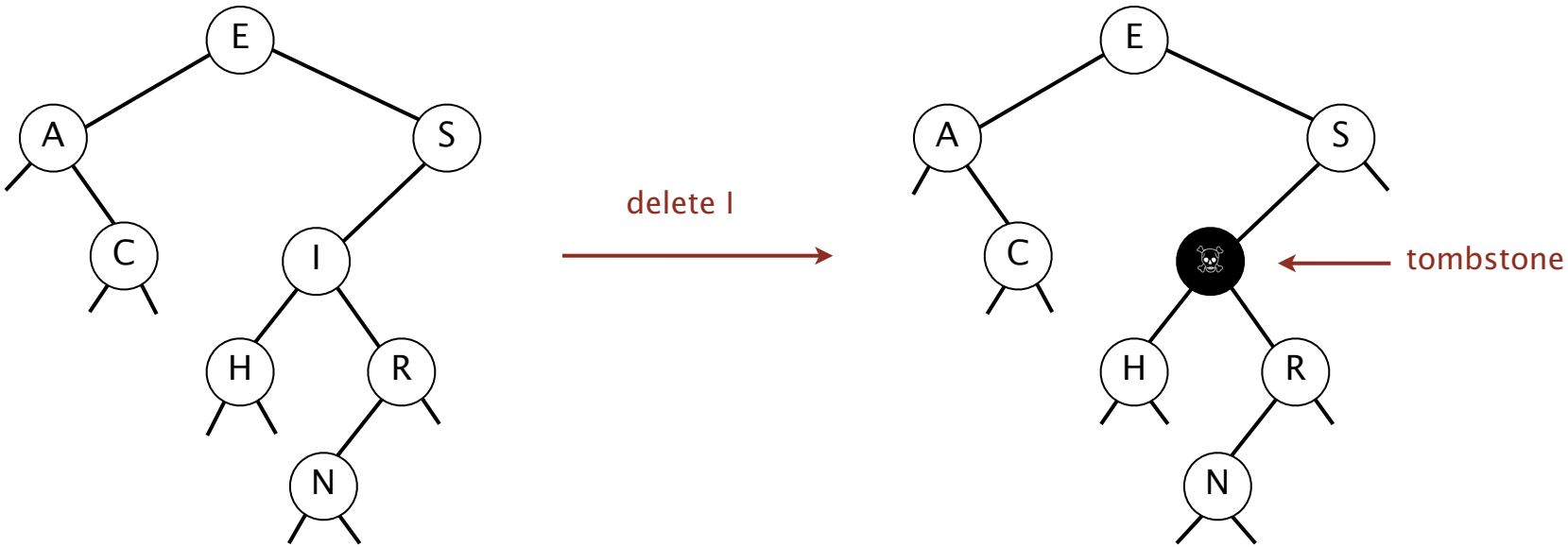
`https://www.cs.usfca.edu/~galles/visualization/BST.html`



Binary Search Trees

("Lazy Approach" to) Delete

Remove a node with a given key by **setting value to null**
Leave the key in place to guide searches (but don't consider it equal in a search)

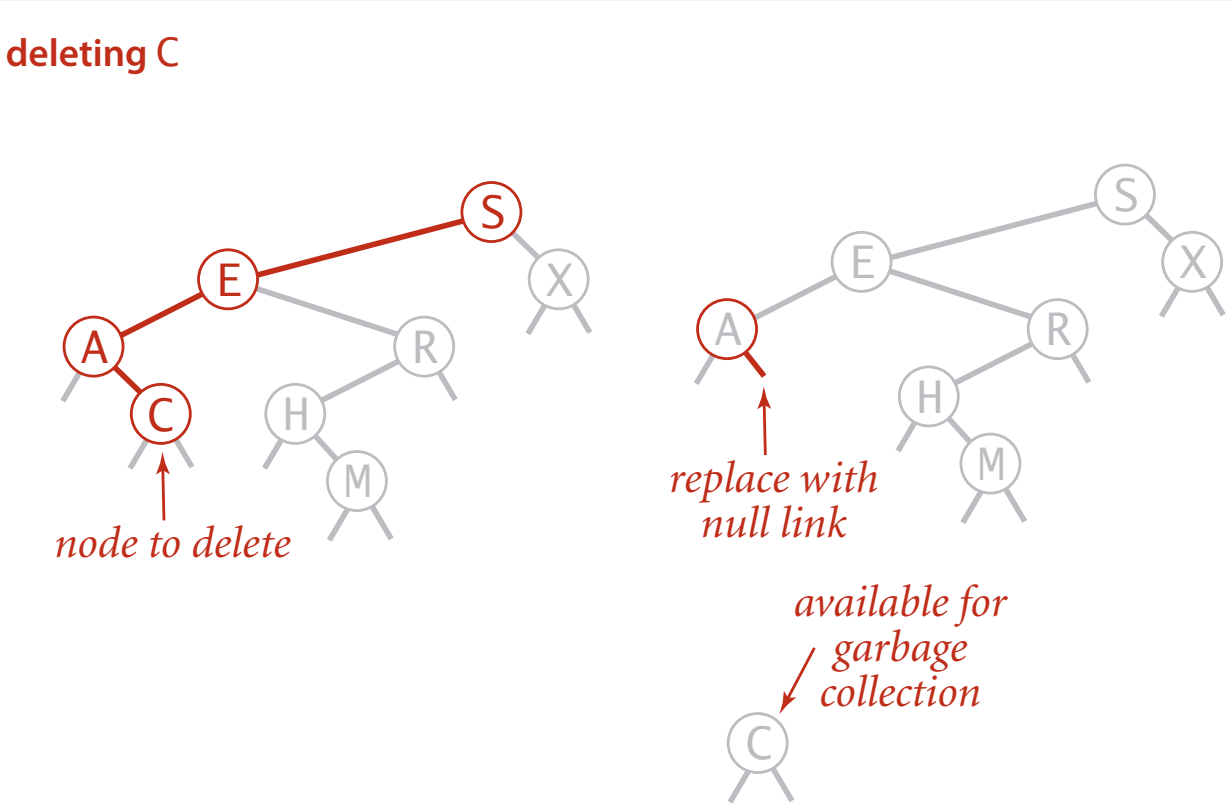


Hibbard Deletion

To delete a node with key k: search for node t containing key k

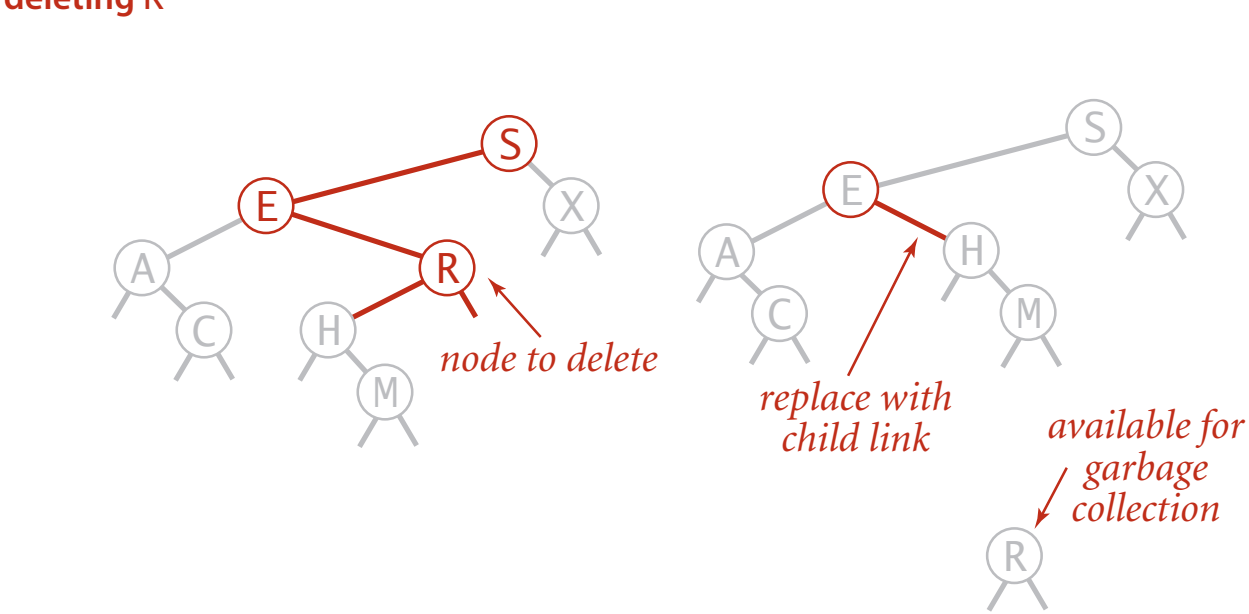
Case 0 [0 children]

Delete t by setting parent link to null.



Case 1 [1 child]

Delete t by replacing parent

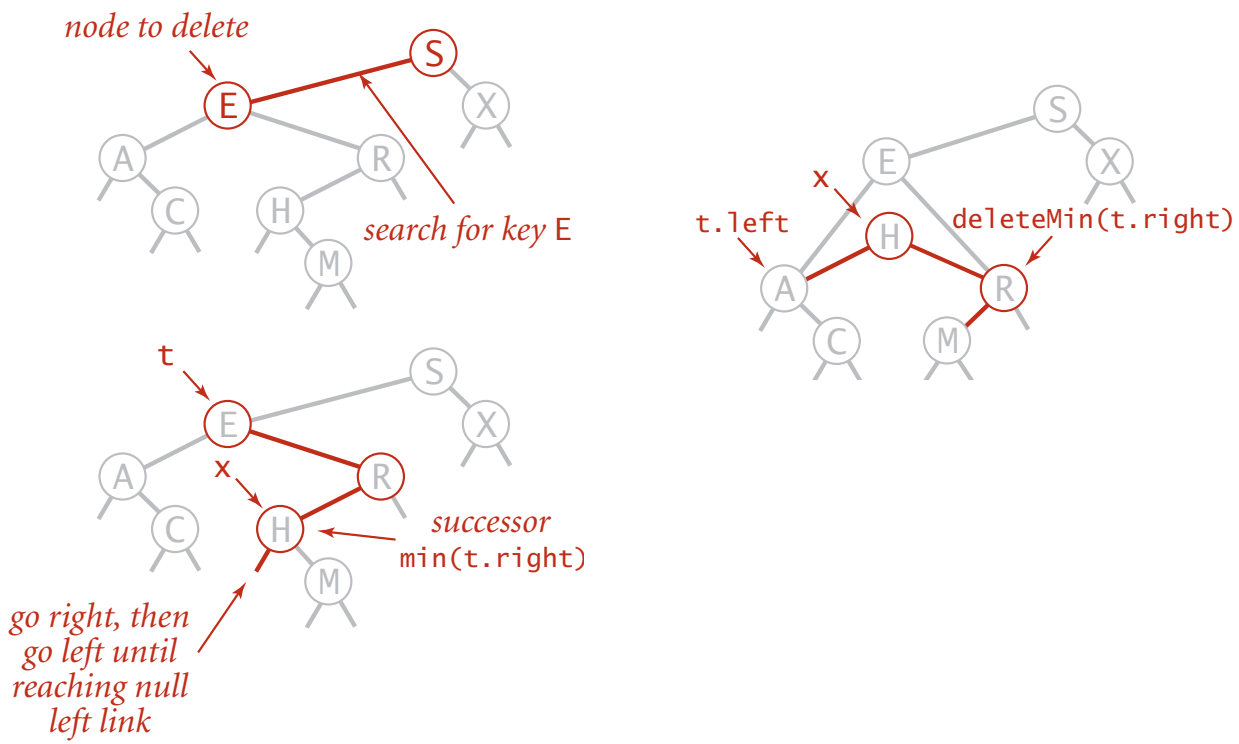


Case 2: 2 children

Find successor `x` of `t`

Delete the minimum in right subtree of `t`

Substitute `x` in for `t`



Binary Search Trees

Balance

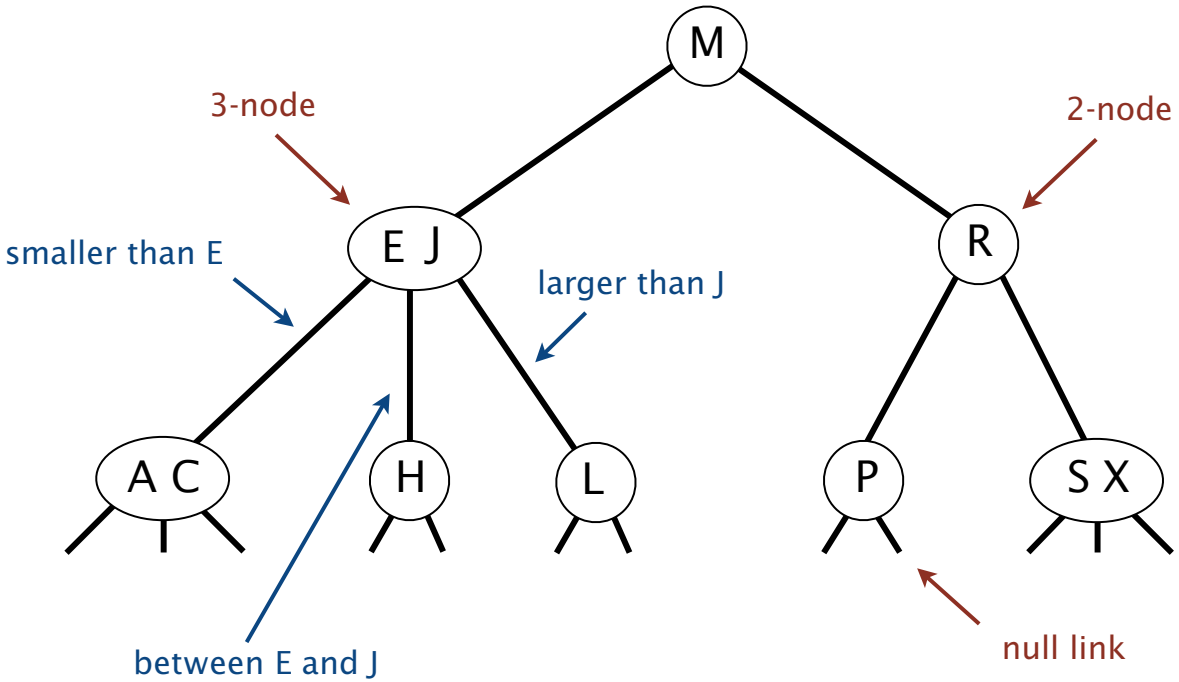
Aim for balanced trees as the shape is important to performance

Balanced Search Trees: 2–3 Trees

Nodes with 1 *or* 2 keys

2-node: one key, two children.

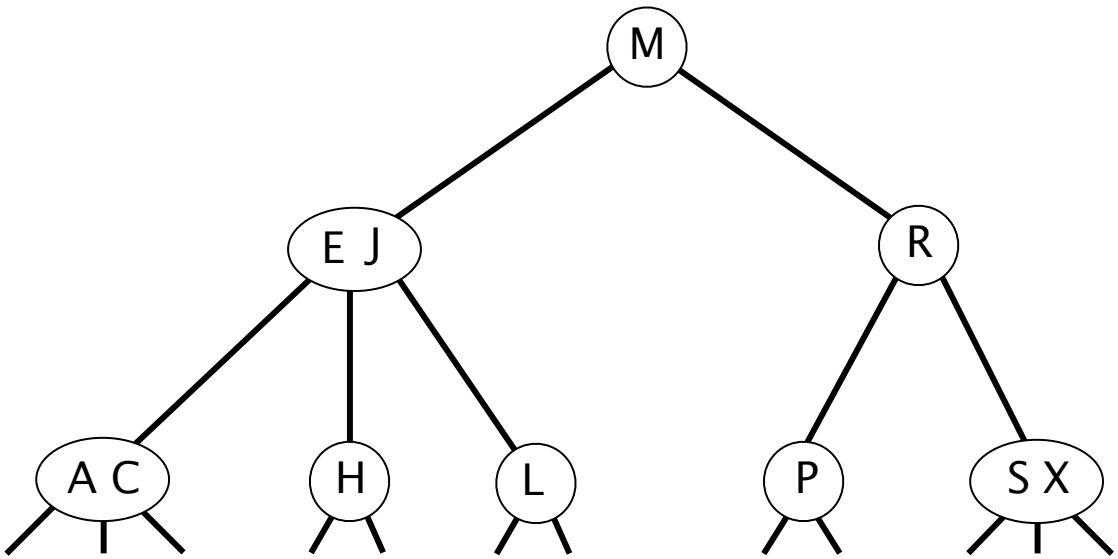
3-node: two keys, three children.



2–3 Trees

Search

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

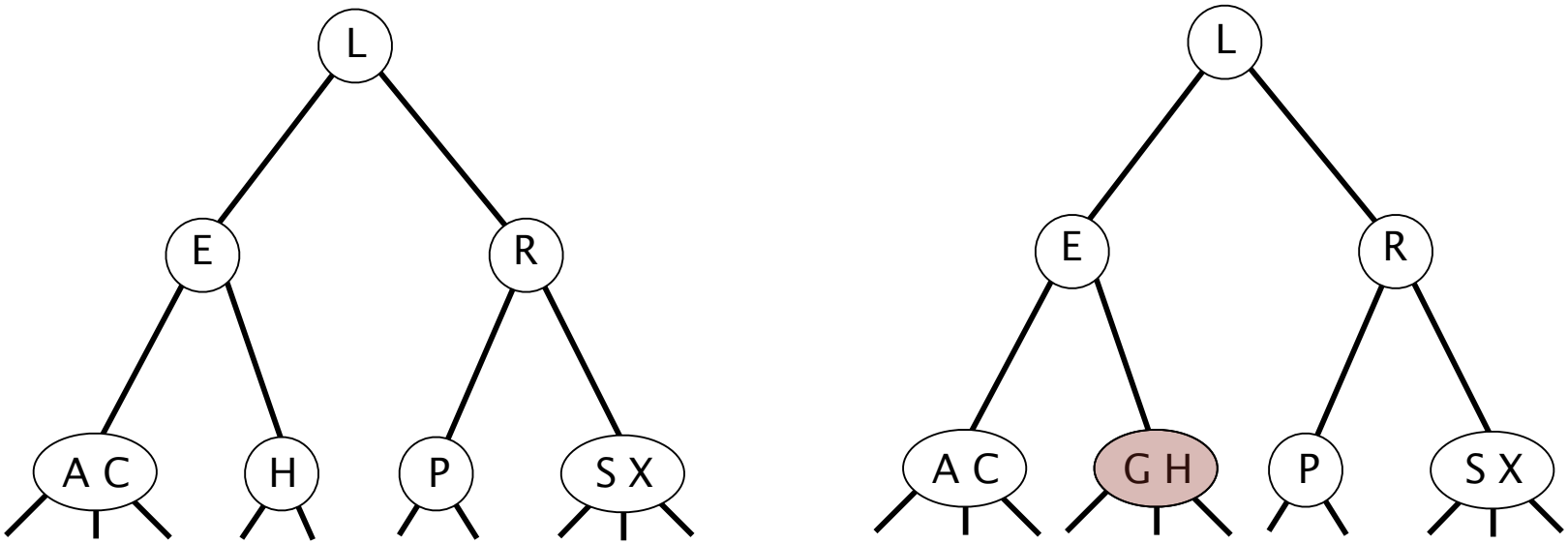


Insert

Insertion into a 2-node at bottom.

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

insert G

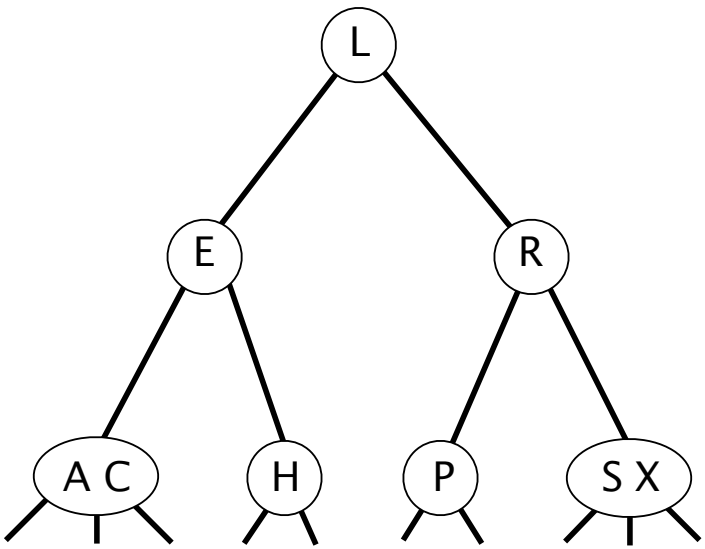


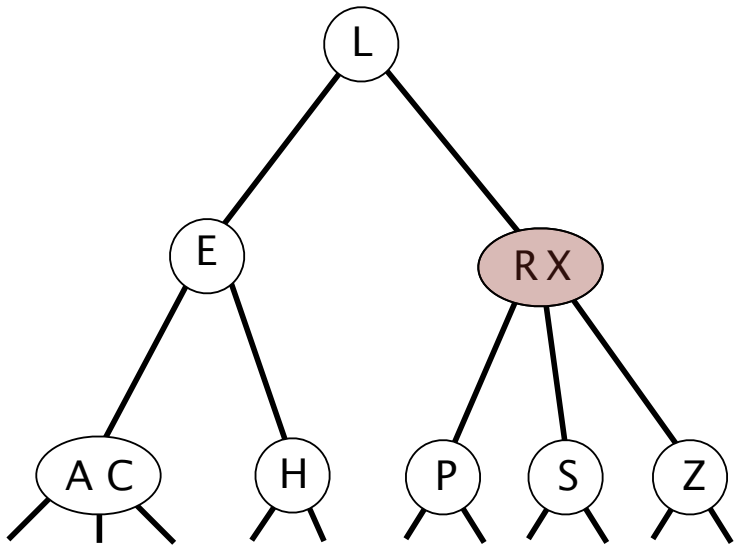
2–3 Trees

Insert into a 3-node at bottom

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

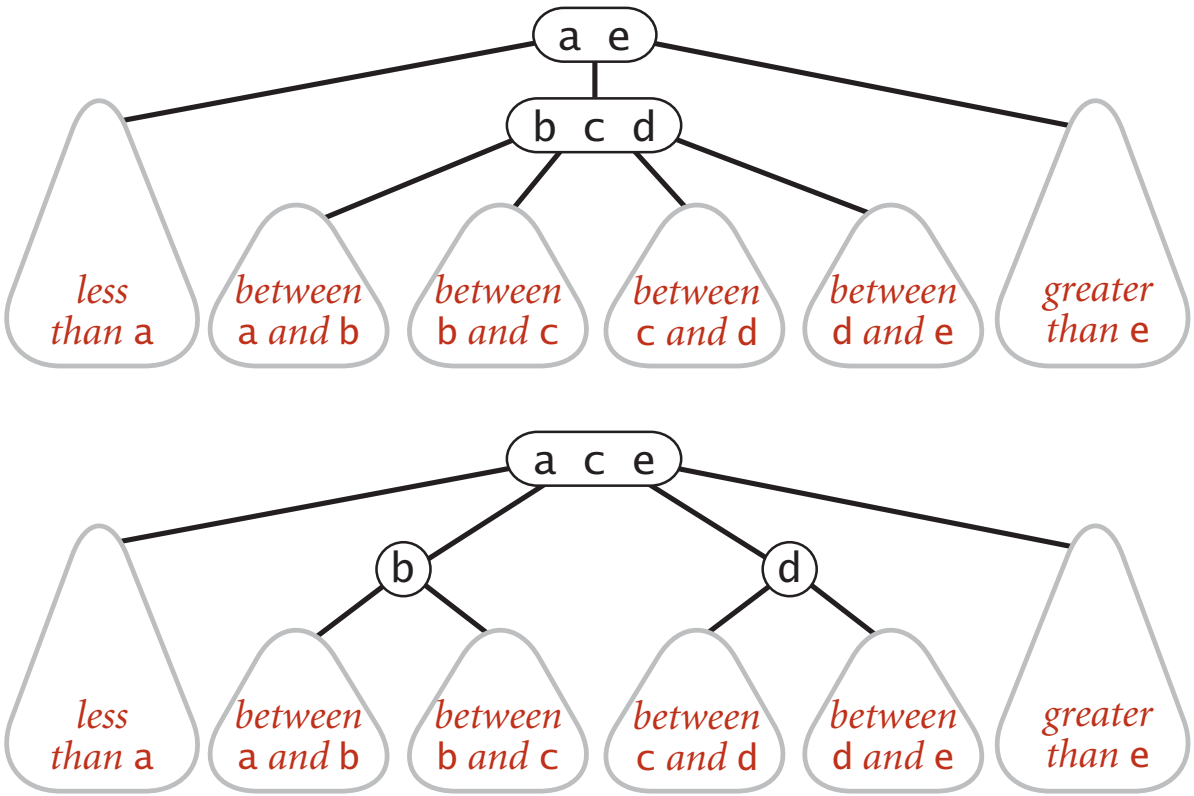
insert Z





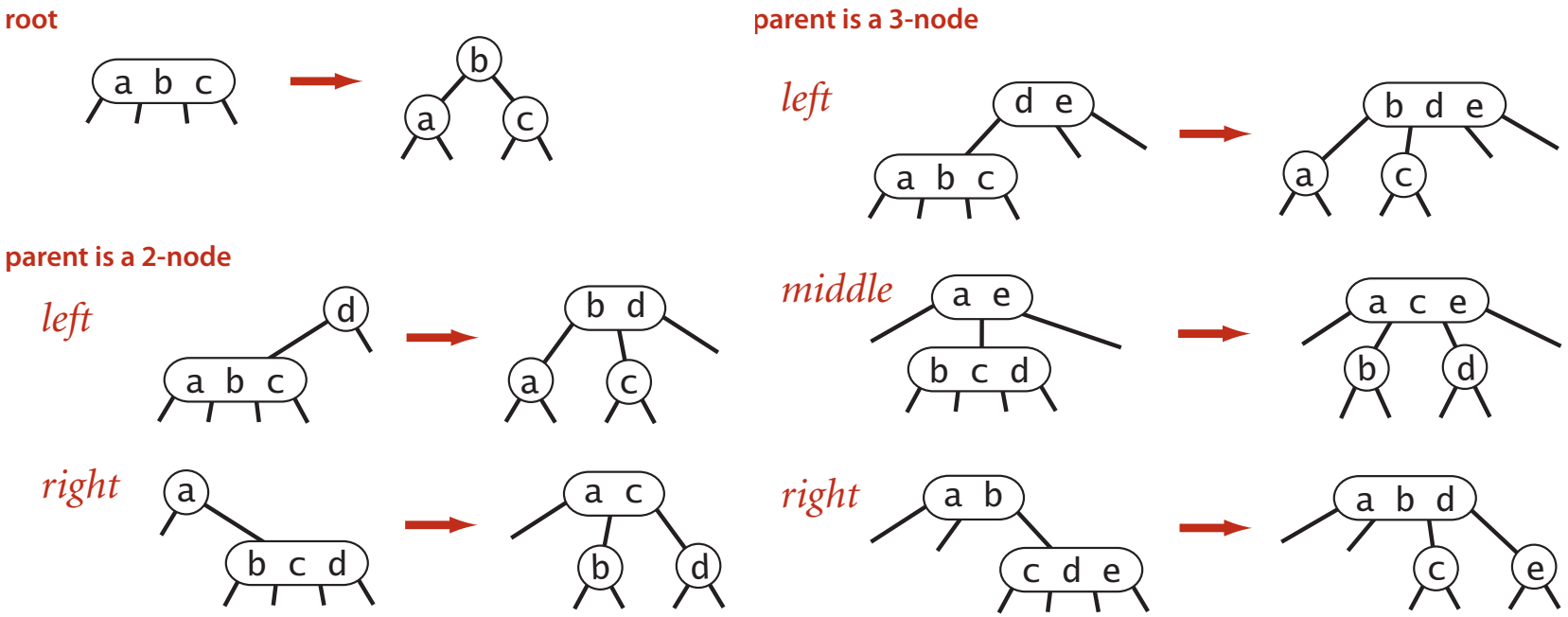
Local Transformations in a 2–3 Tree

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



Global Properties

Each transformation maintains symmetric order and perfect balance



2–3 Trees

Performance

Every Path from root to leaf has same length

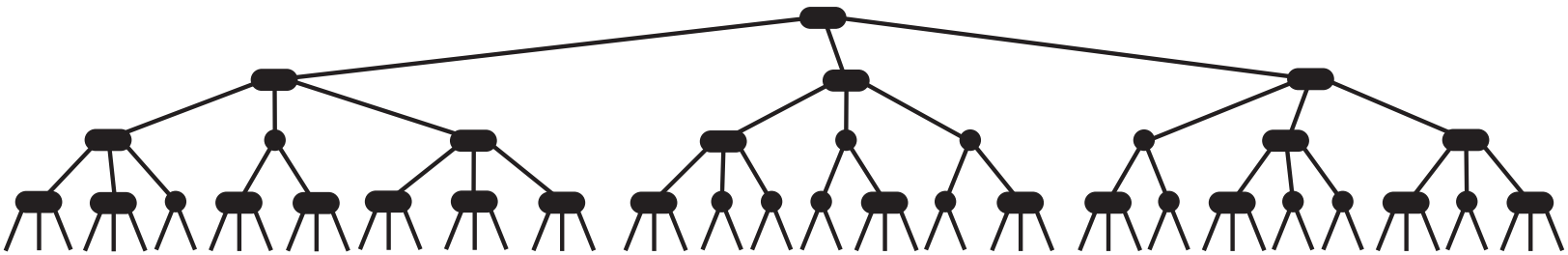
worst case: $\log_2(n)(all2 - nodes)$

best case: $\log_3(n)(all3 - nodes)$

Path length

between 12 and 20 for 1M nodes

between 18 and 30 for 1B nodes



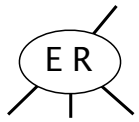
2–3 Trees

Direct implementation is complicated

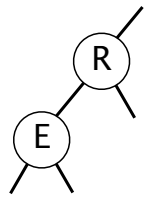
- Maintaining multiple node types is cumbersome
- Need multiple compares to move down tree
- Need to move back up the tree to split 4-nodes
- Large number of cases for splitting

Implementing with Binary Trees

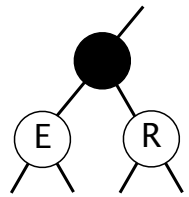
- **Challenge:** How to represent a 3 node?



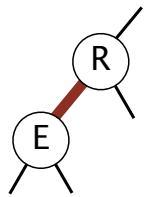
- **Approach 1:** regular BST
  - No way to tell a 3-node from a 2-node
  - Cannot map from BST back to 2-3 tree



- **Approach 2:** regular BST with **"glue" nodes**
  - Wastes space, wasted link
  - Difficult to follow implementation code



- **Approach 3:** Regular BST with **red "glue" links**
  - Widely used in practice
  - Arbitrary restriction: red links lean left

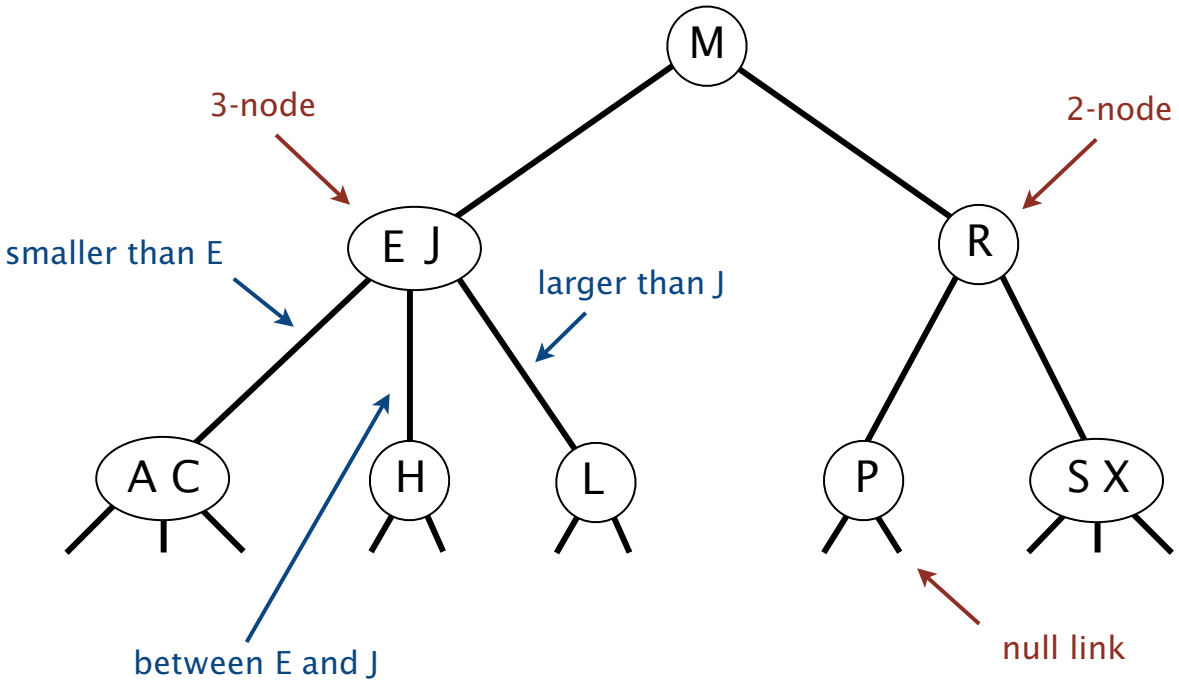


Balanced Search Trees: 2–3 Trees

Nodes with 1 *or* 2 keys

2-node: one key, two children.

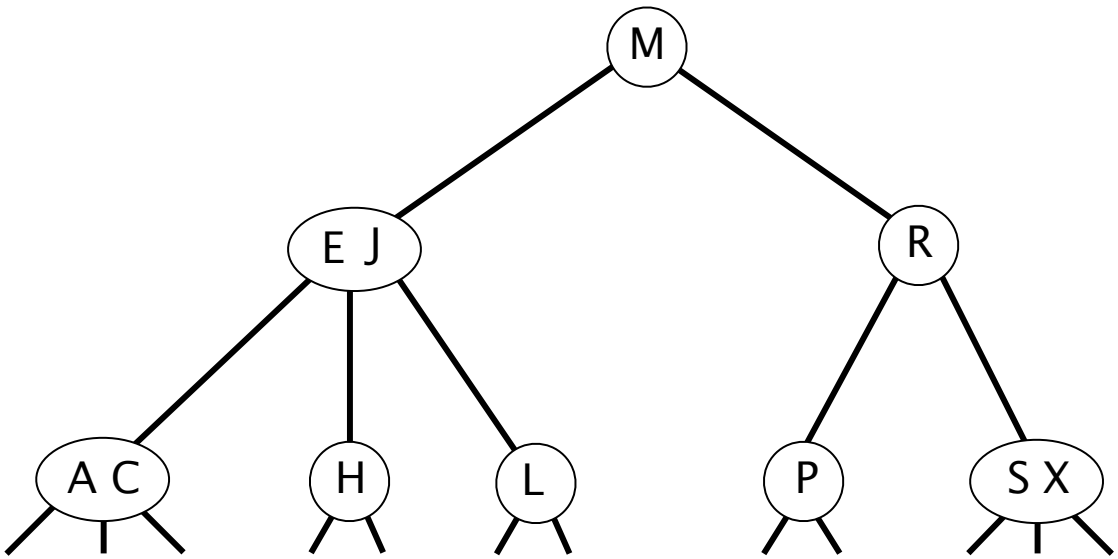
3-node: two keys, three children.



2–3 Trees

Search

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

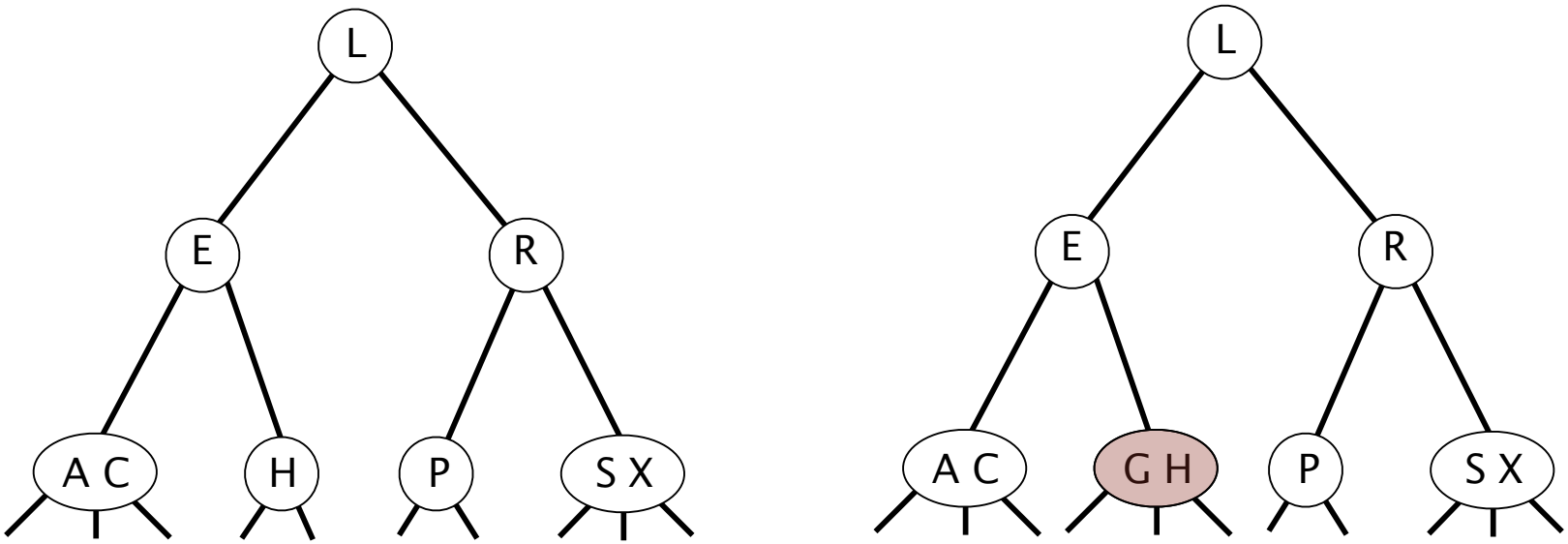


Insert

Insertion into a 2-node at bottom.

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

insert G

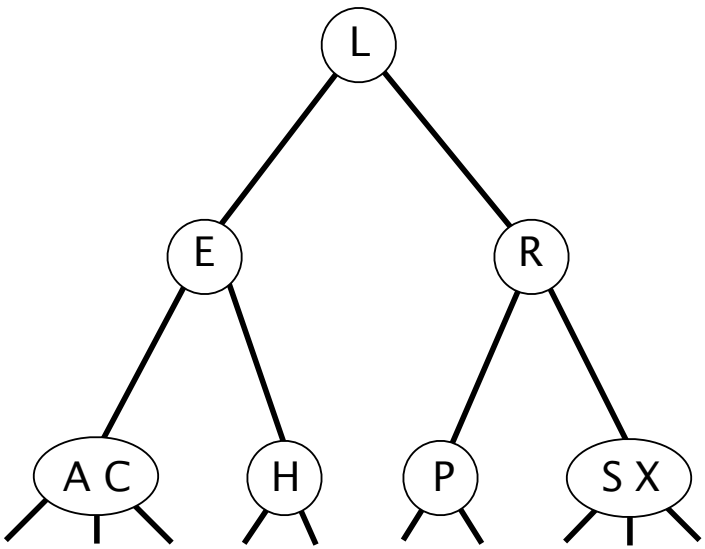


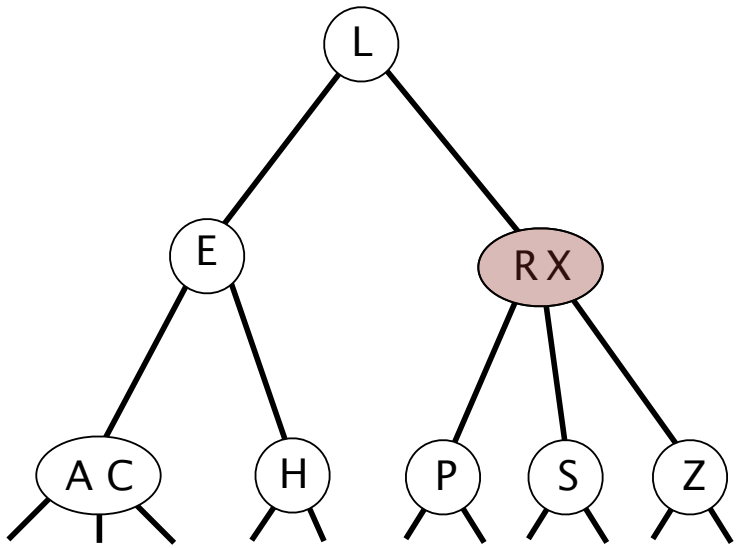
2–3 Trees

Insert into a 3-node at bottom

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

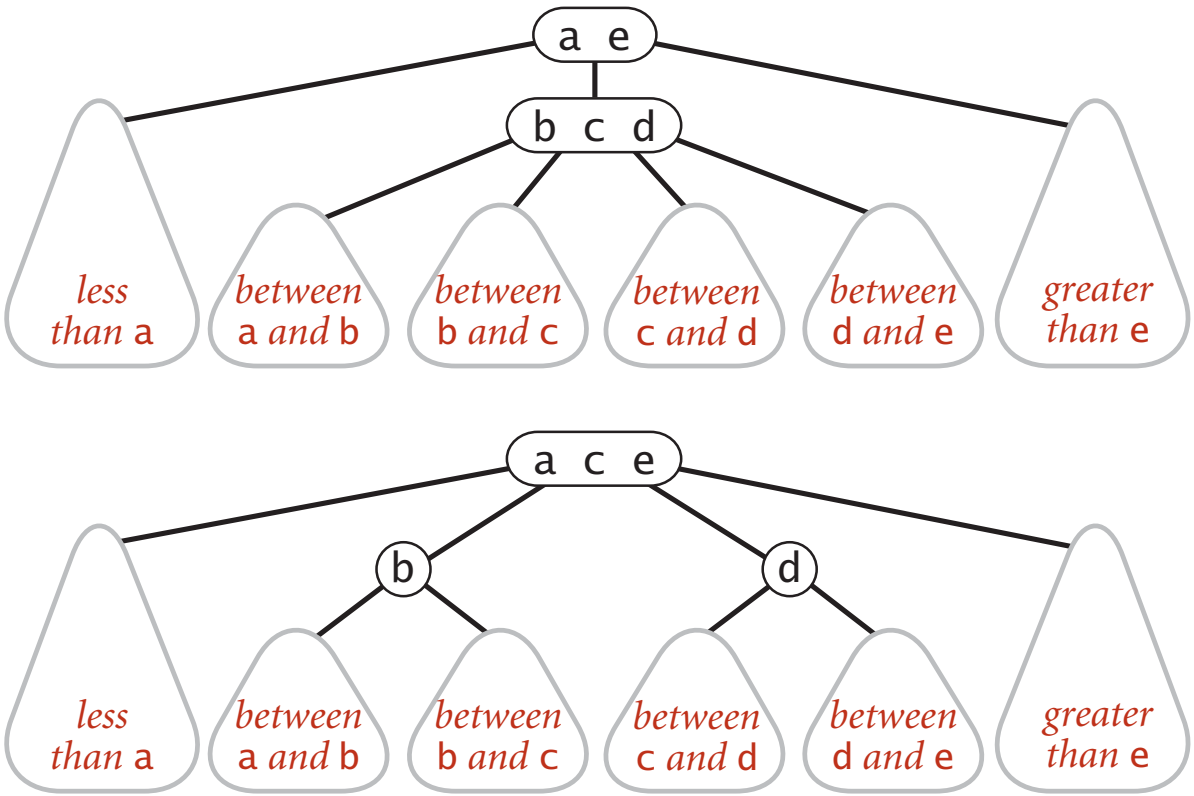
insert Z





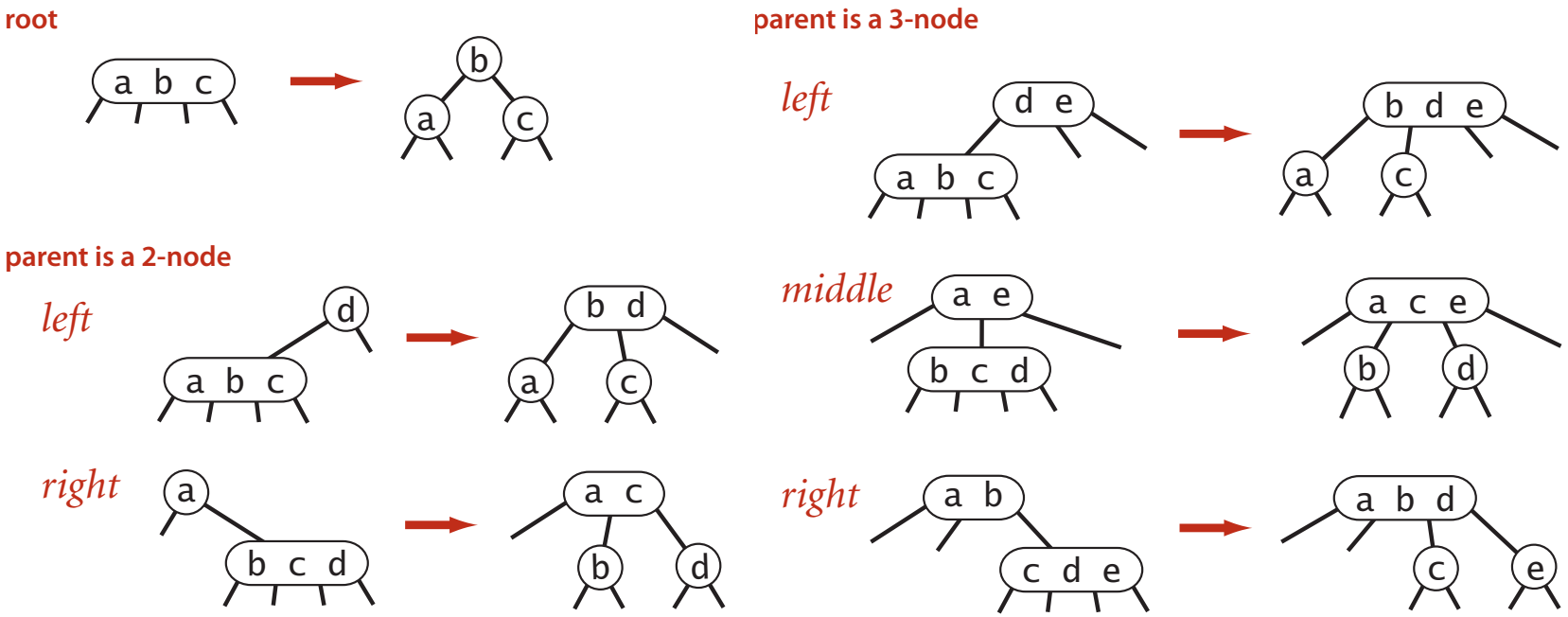
Local Transformations in a 2–3 Tree

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



Global Properties

Each transformation maintains symmetric order and perfect balance



2–3 Trees

Performance

Every Path from root to leaf has same length

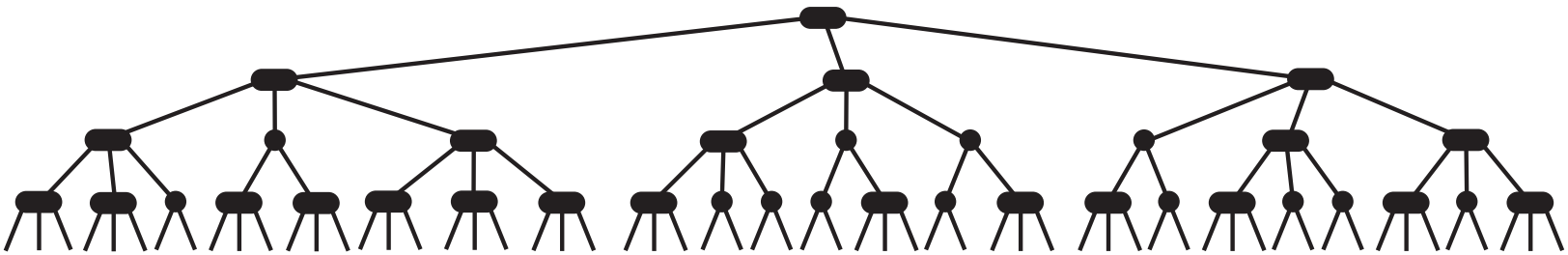
worst case: $log_2(n)$ (all 2-nodes)

best case: $log_3(n)$ (all 3-nodes)

Path length

between 12 and 20 for 1M nodes

between 18 and 30 for 1B nodes



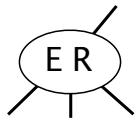
2–3 Trees

Direct implementation is complicated

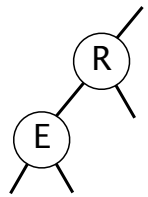
- Maintaining multiple node types is cumbersome
- Need multiple compares to move down tree
- Need to move back up the tree to split 4-nodes
- Large number of cases for splitting

Implementing with Binary Trees

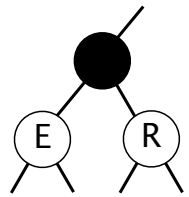
- **Challenge:** How to represent a 3 node?



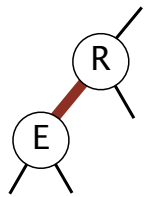
- **Approach 1:** regular BST
  - No way to tell a 3-node from a 2-node
  - Cannot map from BST back to 2-3 tree



- **Approach 2:** regular BST with **"glue" nodes**
  - Wastes space, wasted link
  - Difficult to follow implementation code



- **Approach 3:** Regular BST with **red "glue" links**
  - Widely used in practice
  - Arbitrary restriction: red links lean left



2-3 Trees, 2-3-4 trees, Red-black trees

Red-black trees are based on implementing 2-3 or (2,4) trees within a binary tree

Red-black trees

simple code unifies the algorithms and they differ in the position of one line of code:

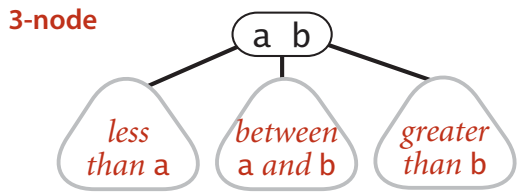
- left-leaning versions of 2-3 trees
- top-down (2,4) trees (a.k.a. 2,3,4 trees)

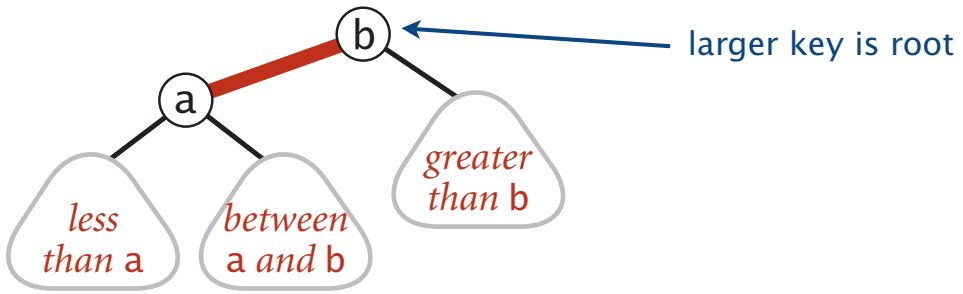
Implementing 2-3 trees

Red-black BSTs

Red-black BSTs are a representation that shows how tim implement 2-3 balanced search trees.

- Red links **lean left**
- No node has two red links connected to it
- **The tree has perfect black balance:** every path from the root to a null link has the same number of black links

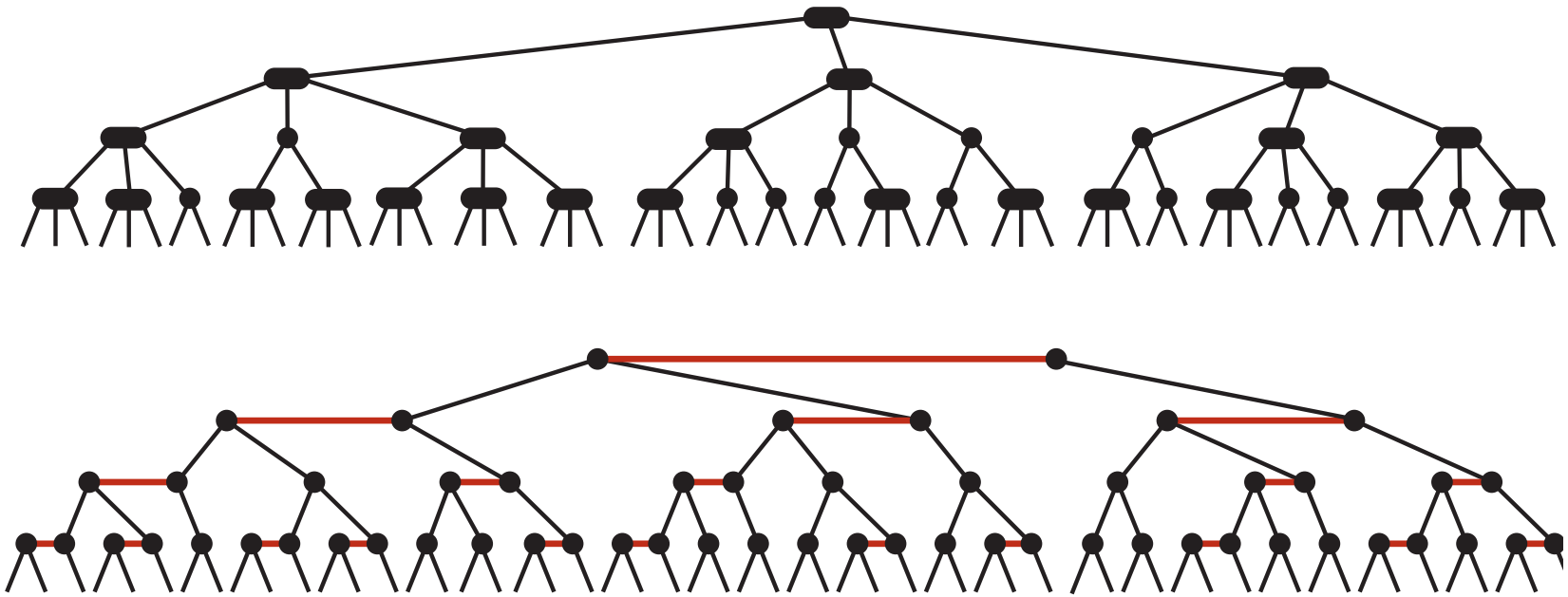




Left-leaning red-black BSTs

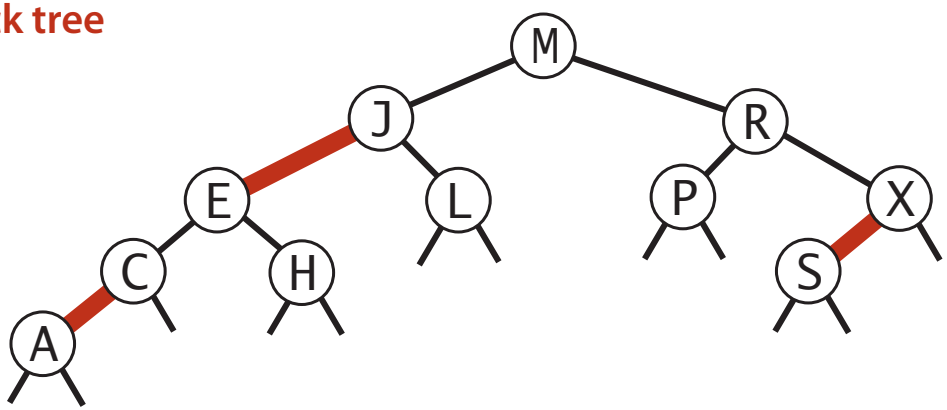
Why red-black?

Each node is only pointed to one other node (its parent)
Encode the colour using a boolean (`red==true`, `black==false`)
When we refer to the colour of a node we refer to the colour of the link pointing to it.
Allows us to turn a binary tree into a 2–3 tree by making **red links horizontal**

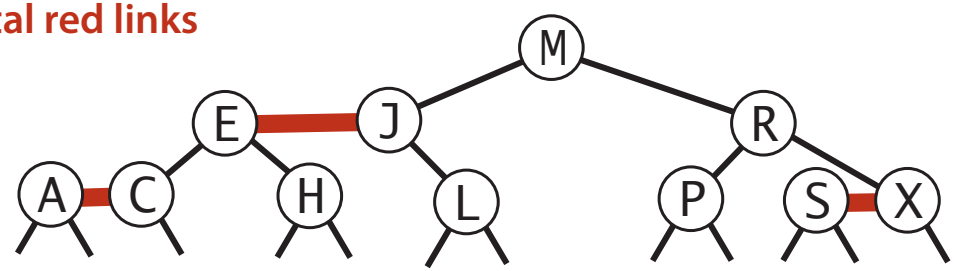


Corresponding 2-3 tree and RB tree

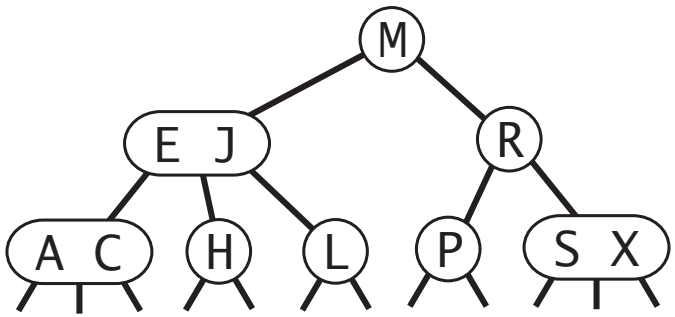
red–black tree



horizontal red links



2-3 tree

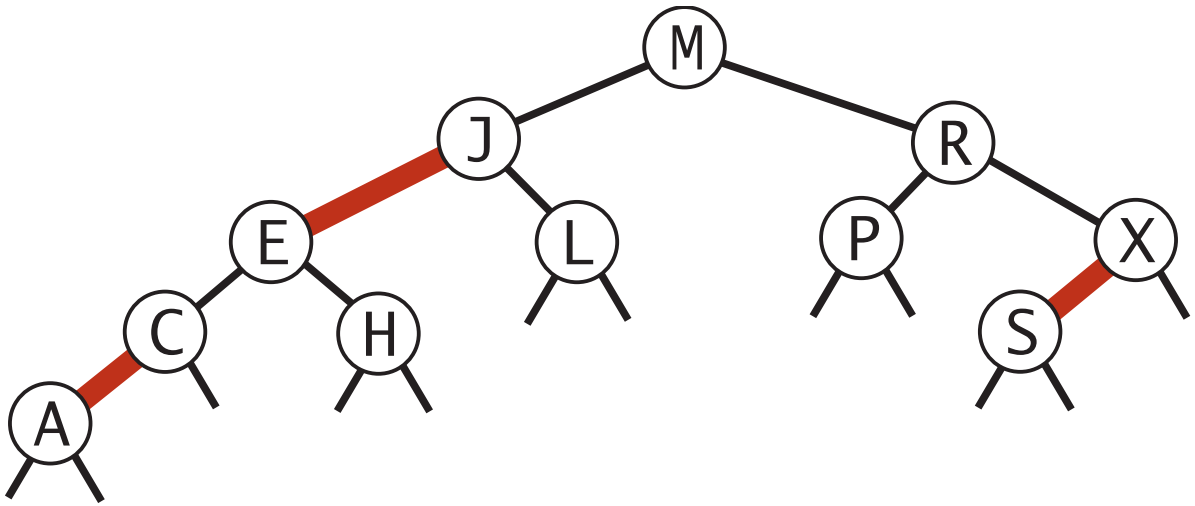


Search in RB tree

Search ignores colour

Same as for elementary BSTs

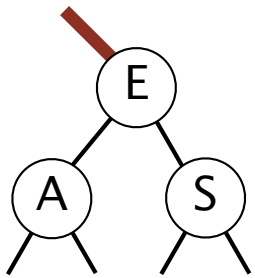
Faster because of better balance



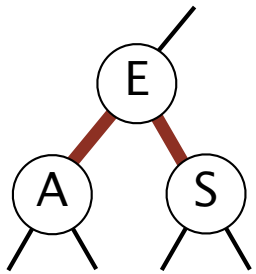
LLRB Tree: Insertion

Objective

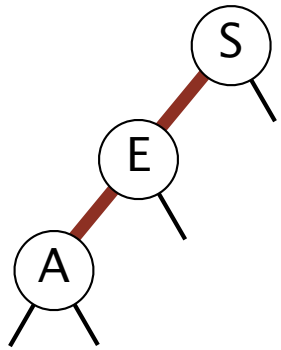
Maintain correspondence with 2-3 trees



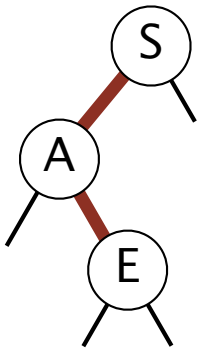
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)

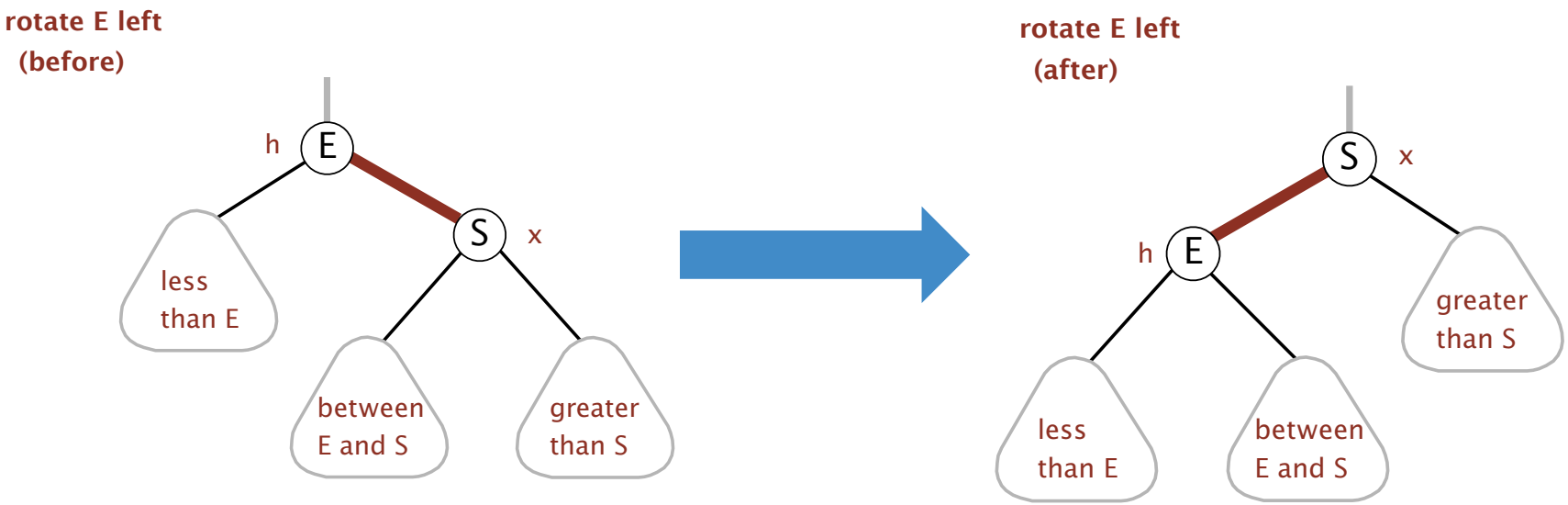


left-right red (a temporary 4-node)

Left Rotation

Rotation Concept

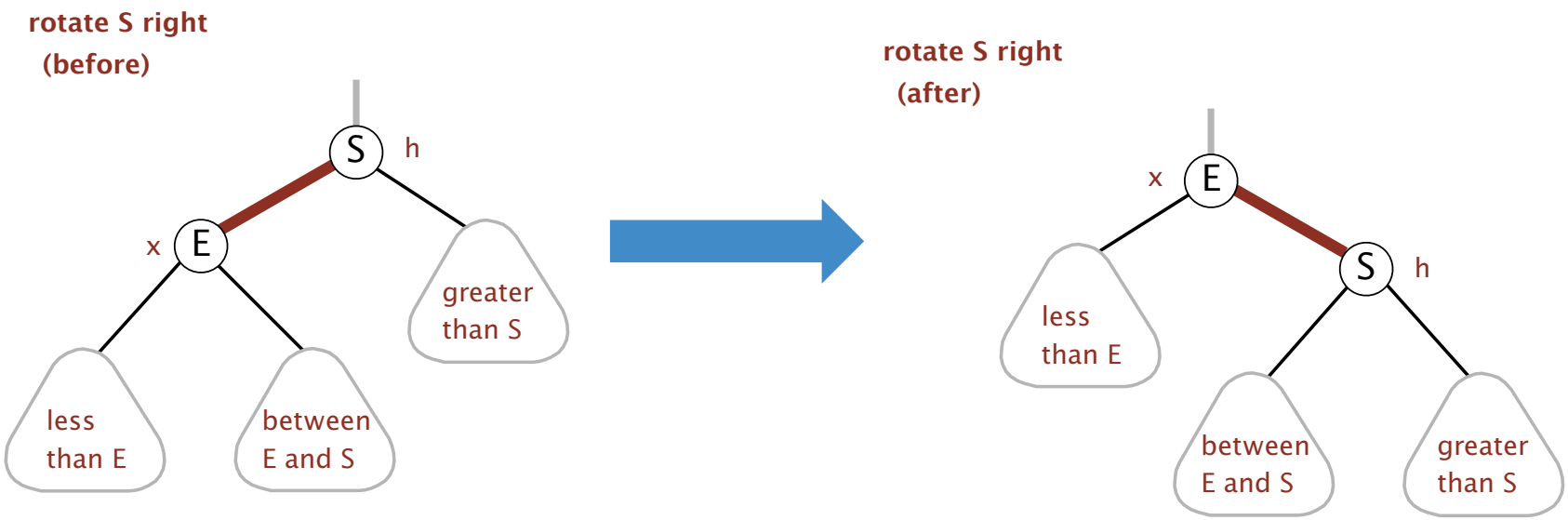
Re-orient a (temporarily) right-leaning red link to lean left



Right Rotation

Rotation Concept

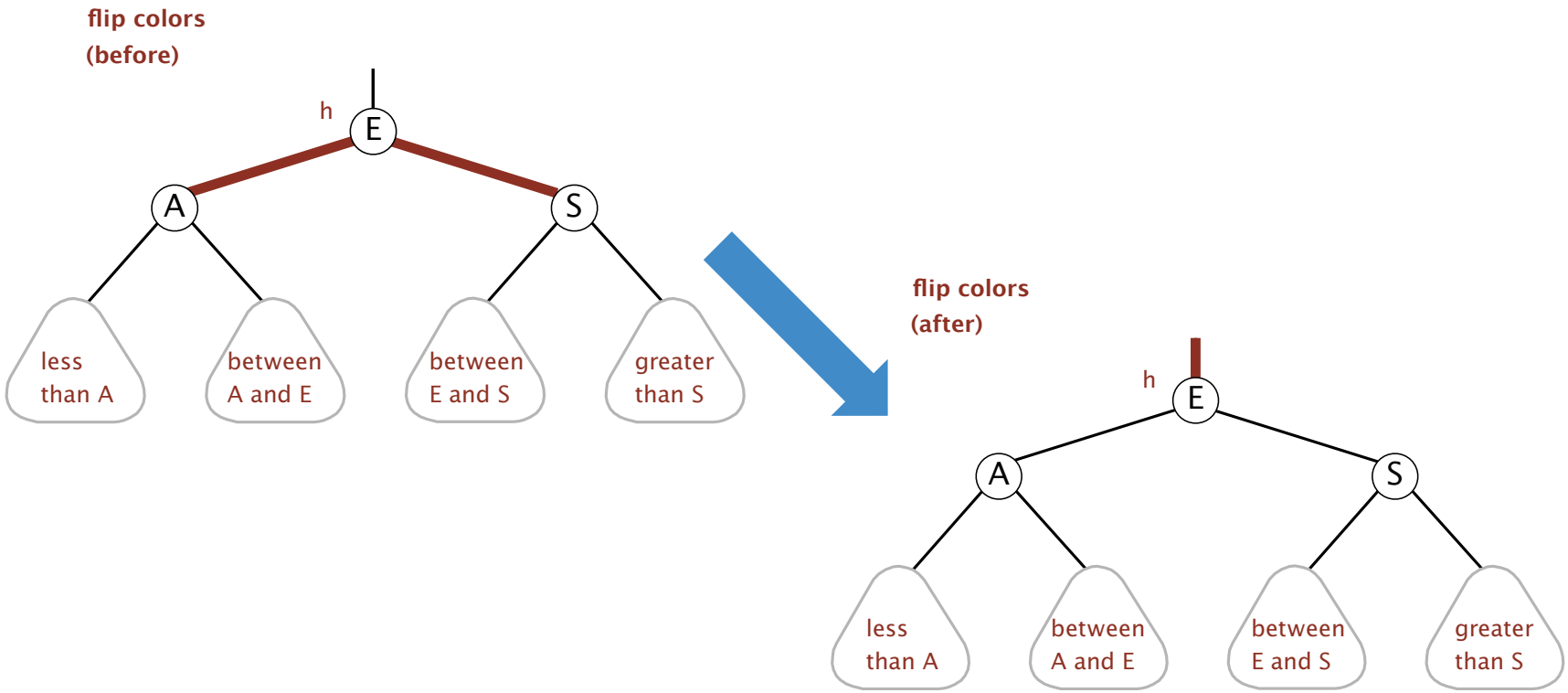
Re-orient a left-leaning red link to (temporarily) lean right



Colour Flip

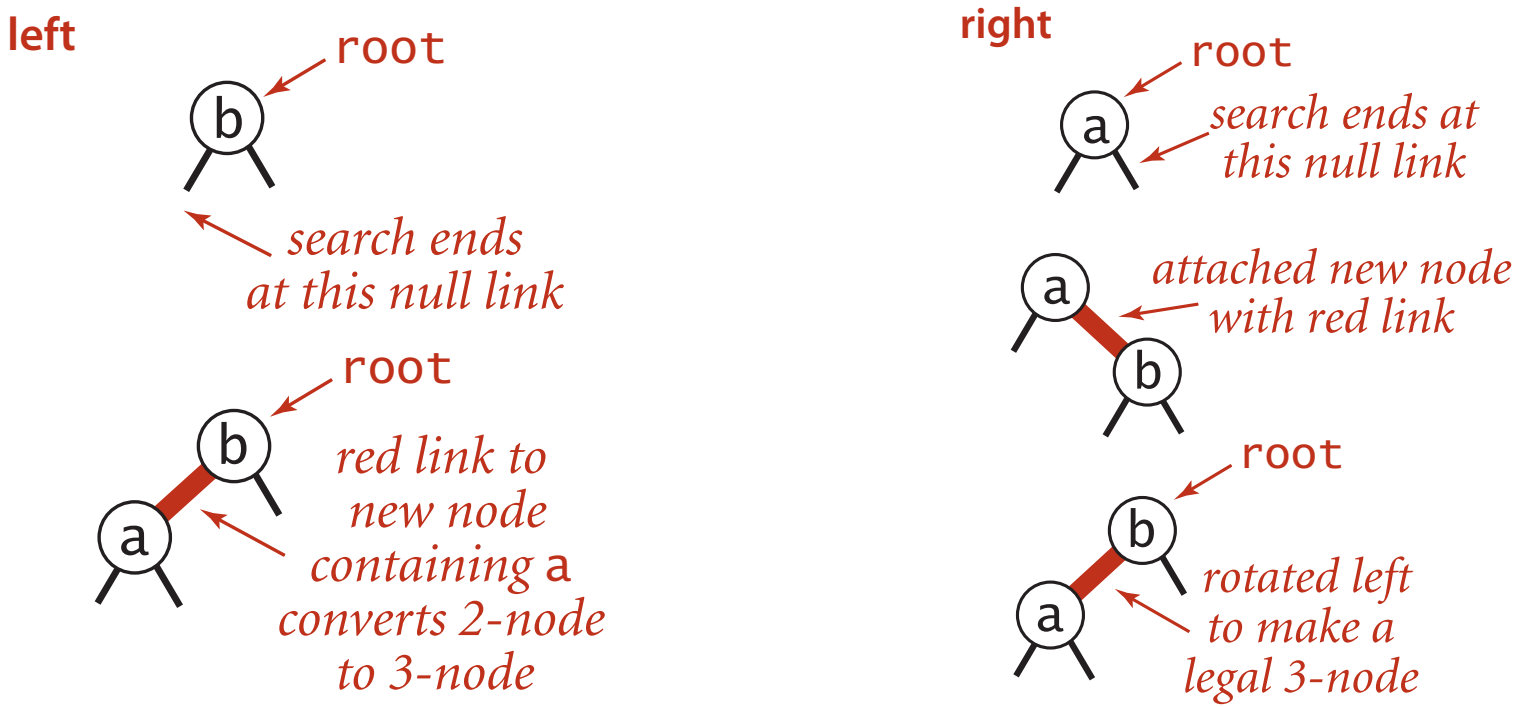
Recolour

Recolor to split a (temporary) 4-node



LLRB Tree: Insertion

1-node Insertion

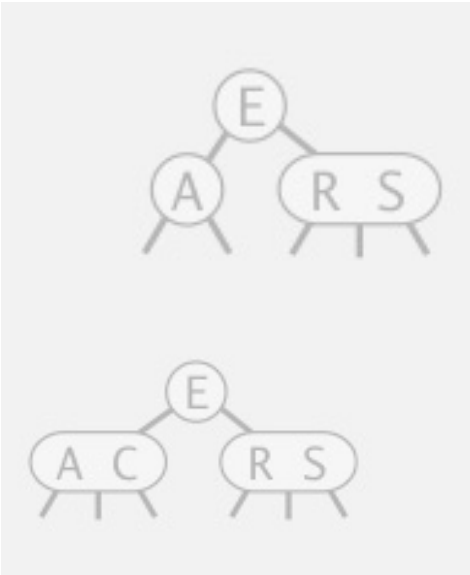


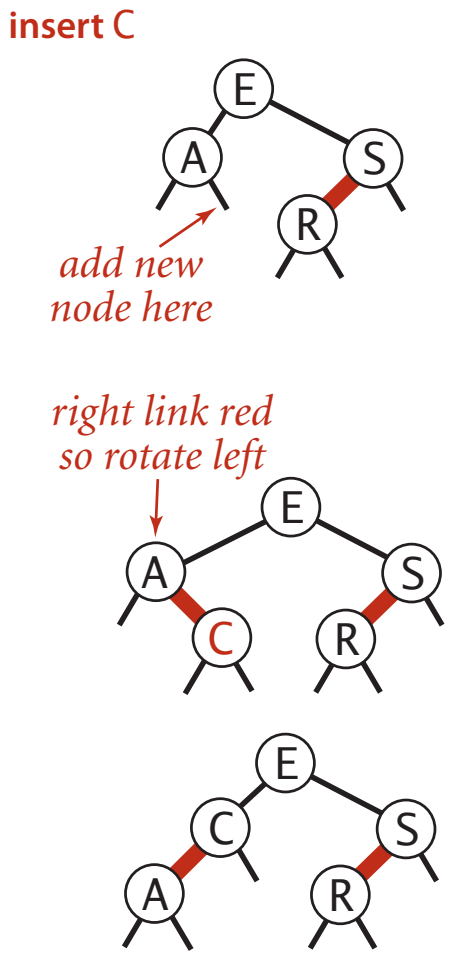
Maintain 3-node search termination after insert

LLRB Tree: Insertion

2-node insert at bottom

Standard BST insert (colouring new link `RED`)
If new red link leans right, rotate left.



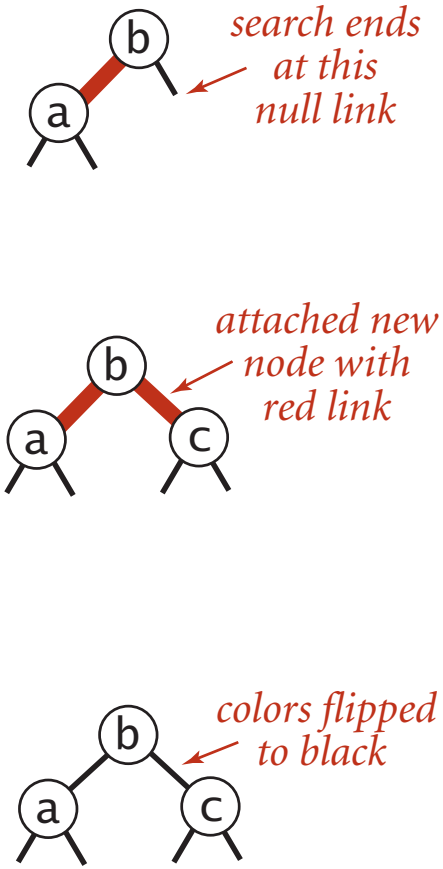


Insert into a 2-node at the bottom

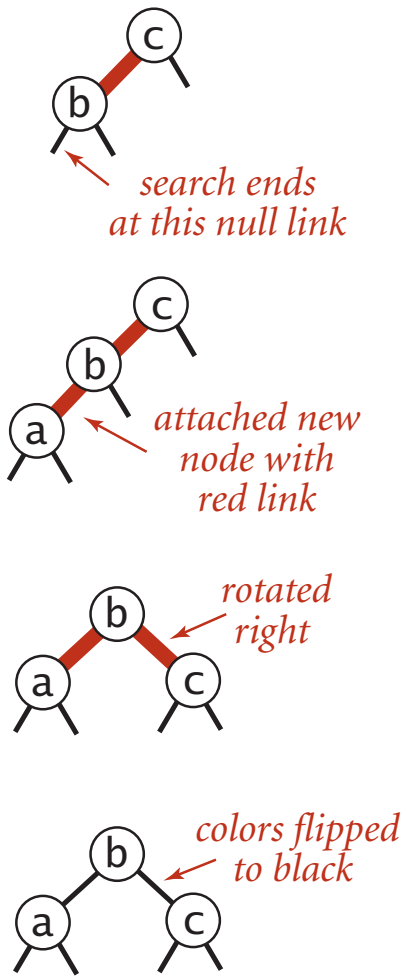
LLRB Tree: Insertion

Insert into a RB trees with exactly 2 nodes

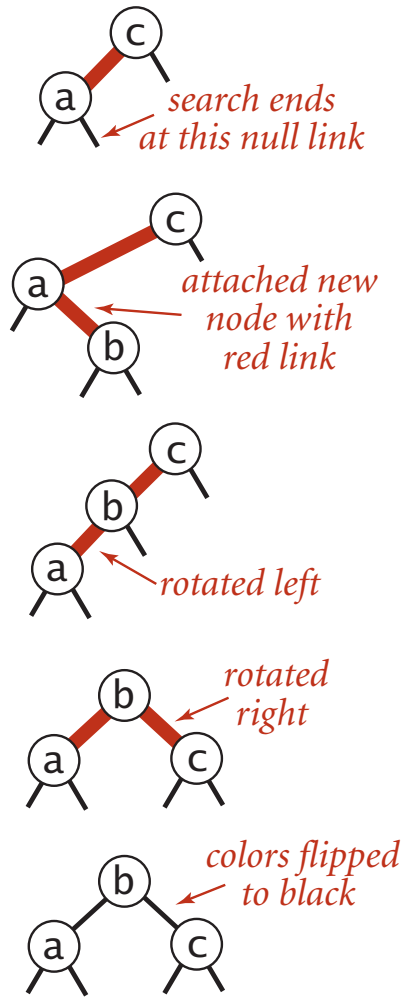
larger



smaller



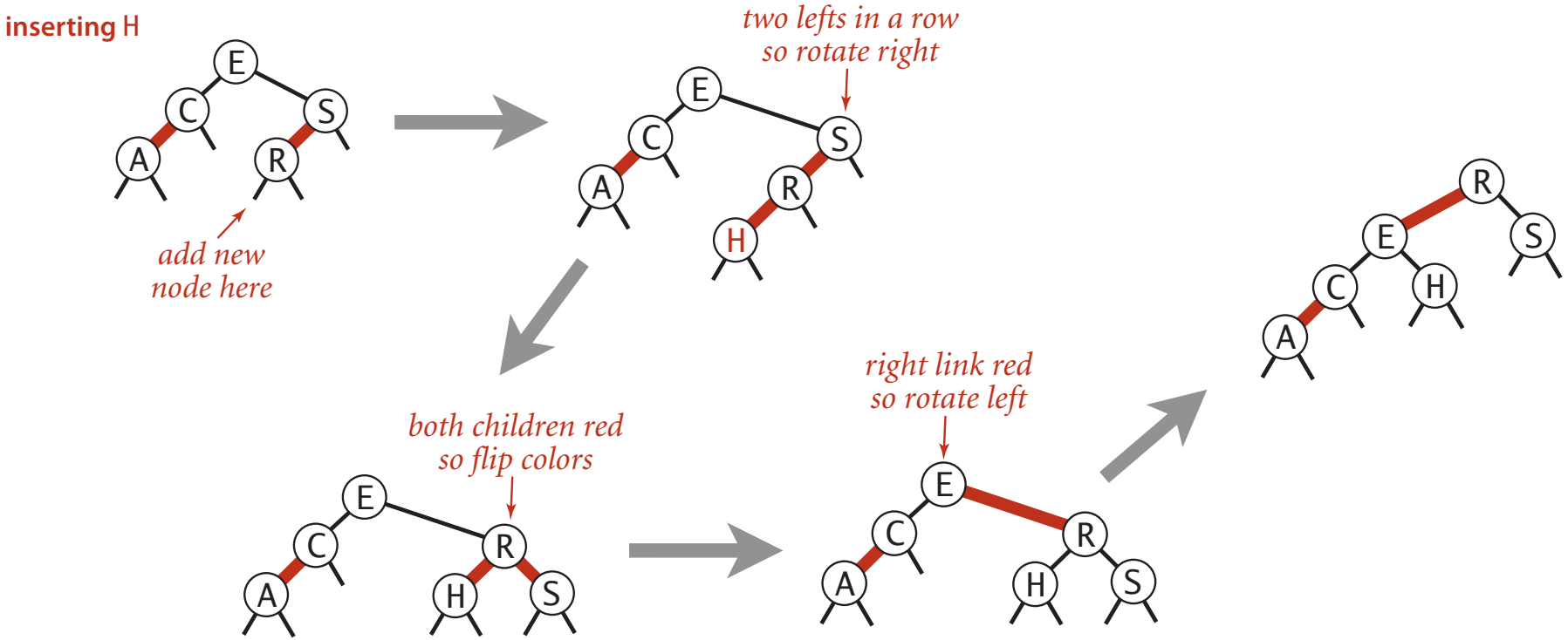
between



LLRB Tree: Insertion

Insert into a RB trees with a 3-node bottom

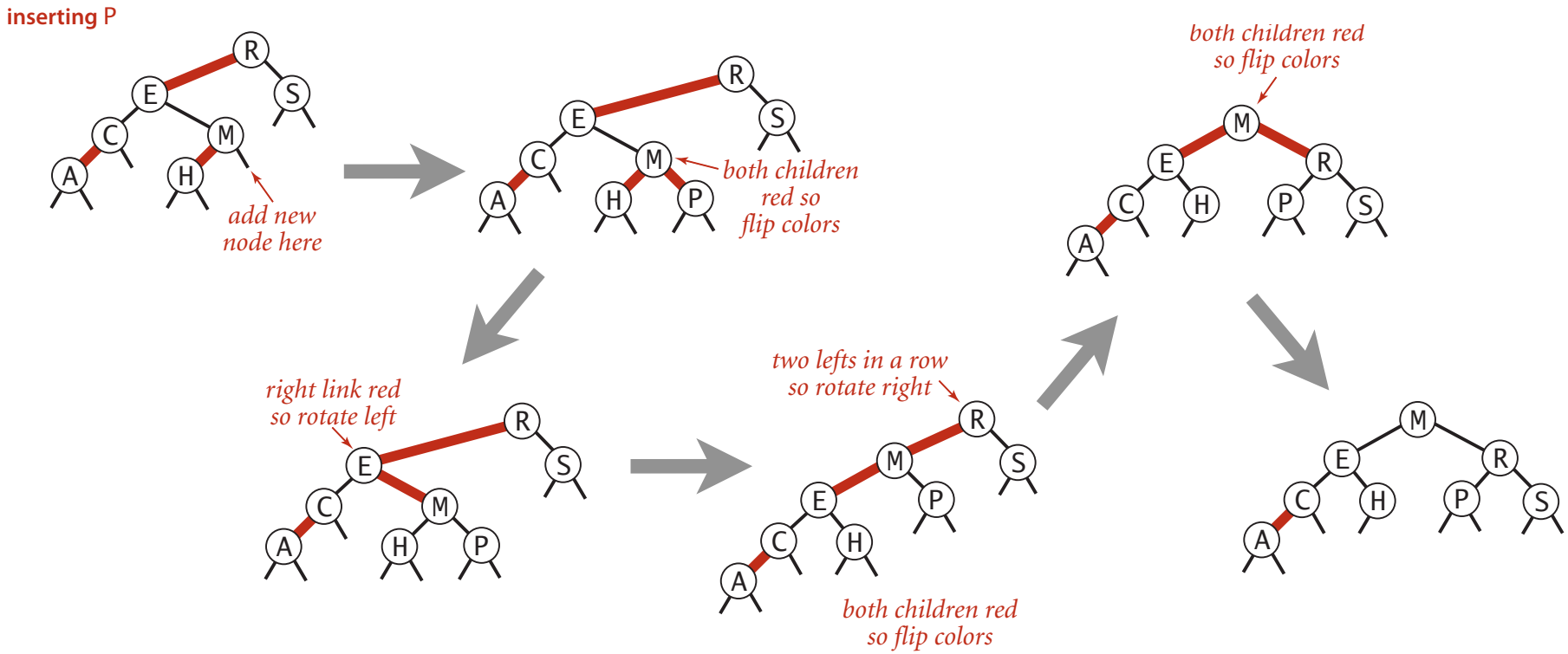
Standard BST insert (colouring new link `RED`)
If needed, **rotate** to balance 4-node
Flip colours to pass red link up one level
If new red link **leans right, rotate left**.



LLRB Tree: Insertion

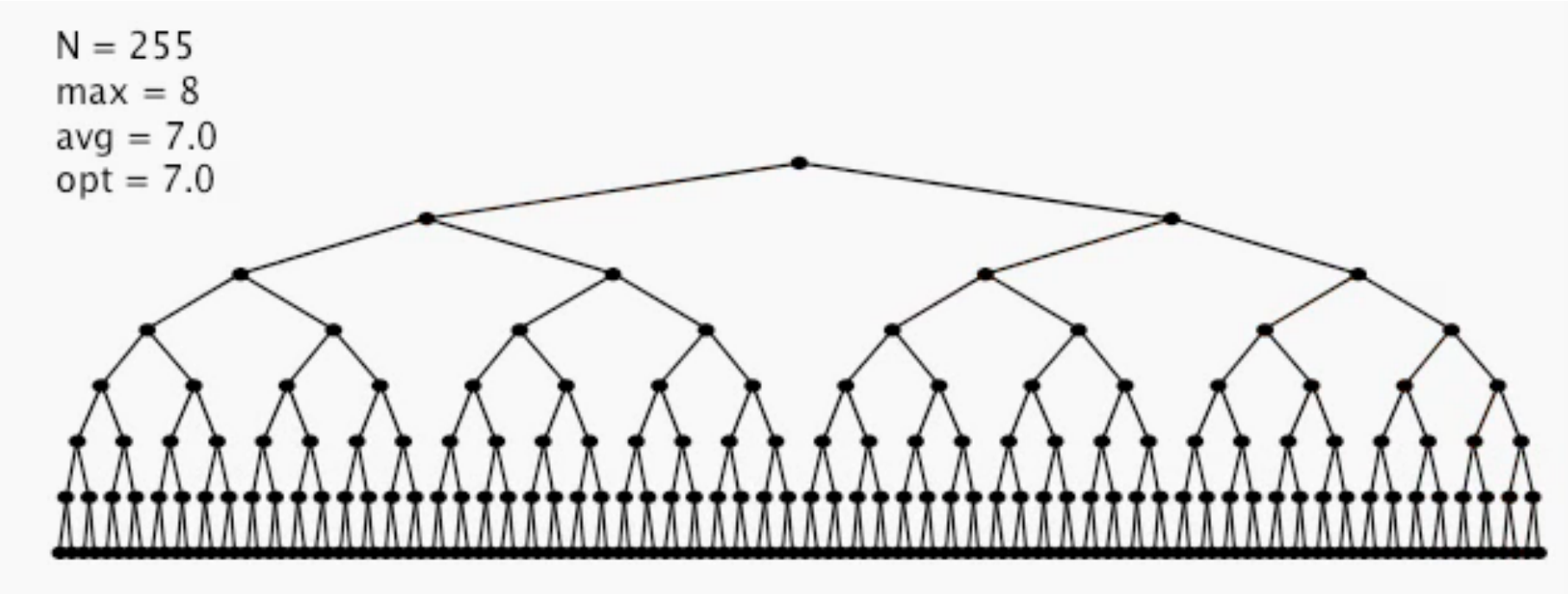
Insert into a RB trees with a 3-node bottom

and **Repeat**, if needed



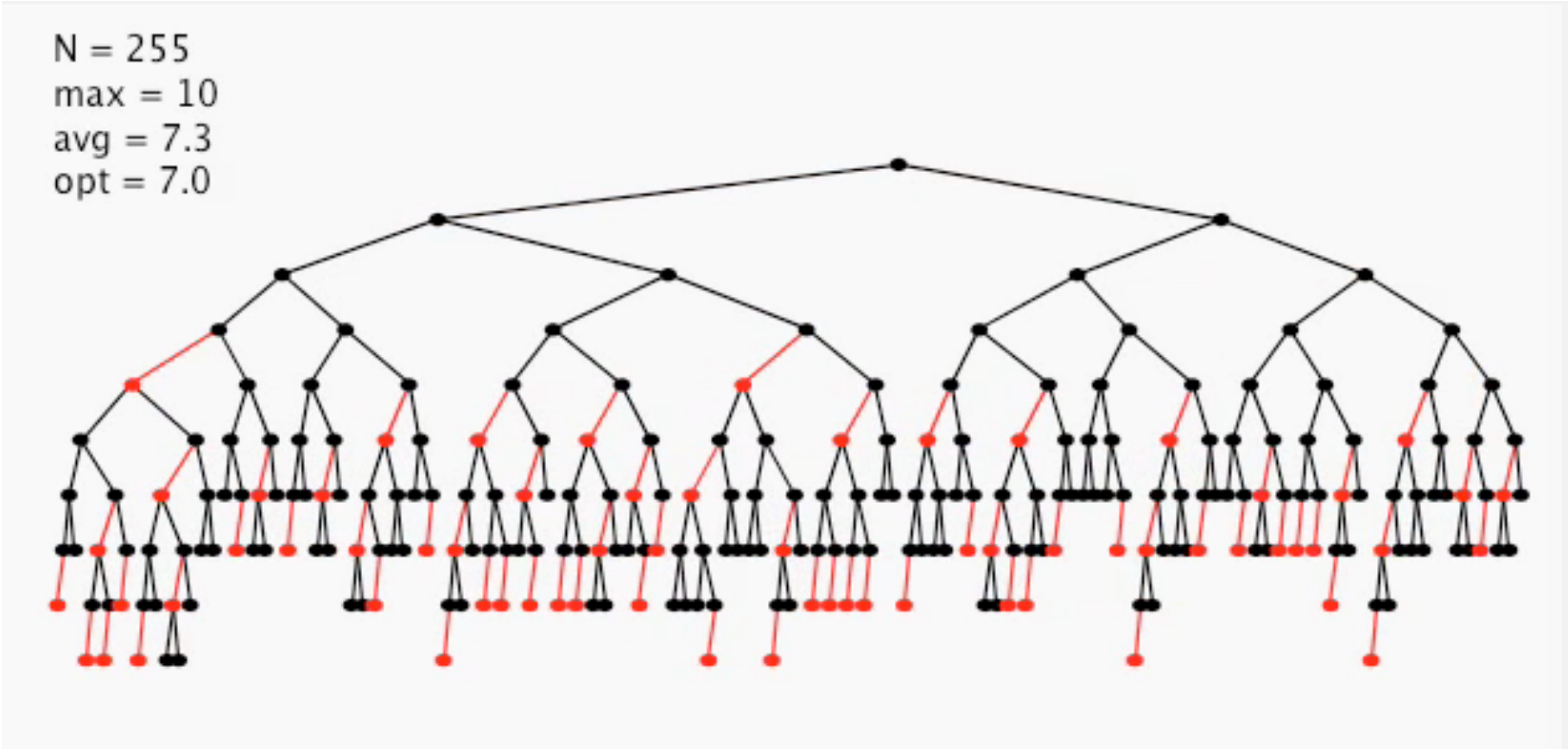
Visualisations

https://algs4.cs.princeton.edu/33balanced/

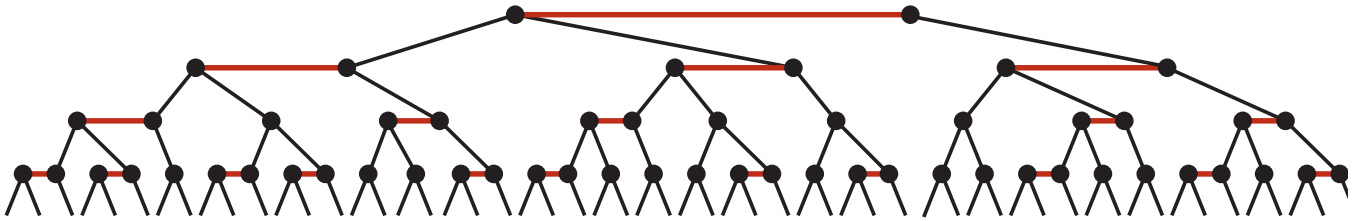


Visualisations

https://algs4.cs.princeton.edu/33balanced/



Summary



Hash tables, Binary and Balanced Sort Trees

Allow rapid data access through search algorithms combined with clever data structures.

Trade-offs

between speed and efficiency for insert, delete, search and update operations.