ID1020: Balanced Search Trees

Dr. Per Brand
pbrand@kth.se

kap 3.3

Slides adapted from Algoritms 4th Edition, Sedgewick.

2-3 Trees

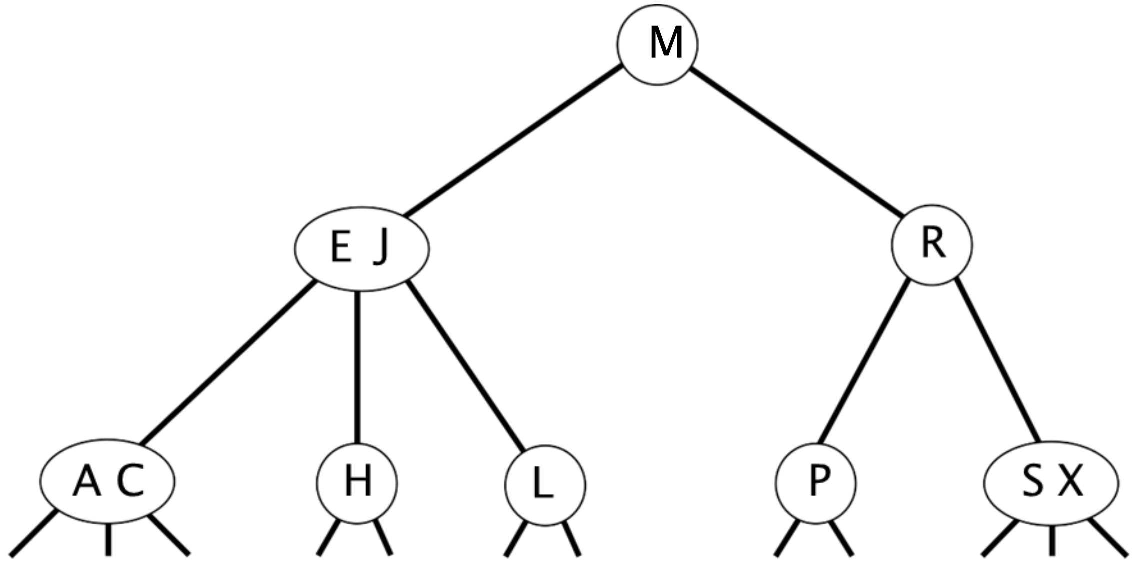
- Nodes either
  - 2-node: One key, two children (as before)
  - 3-node: Two keys, three children

- 3 –node
  - Left branch: keys less than left key
  - Right branch: keys more than right key
  - Middle branch: keys lie between left and right key

Example 2-3 tree

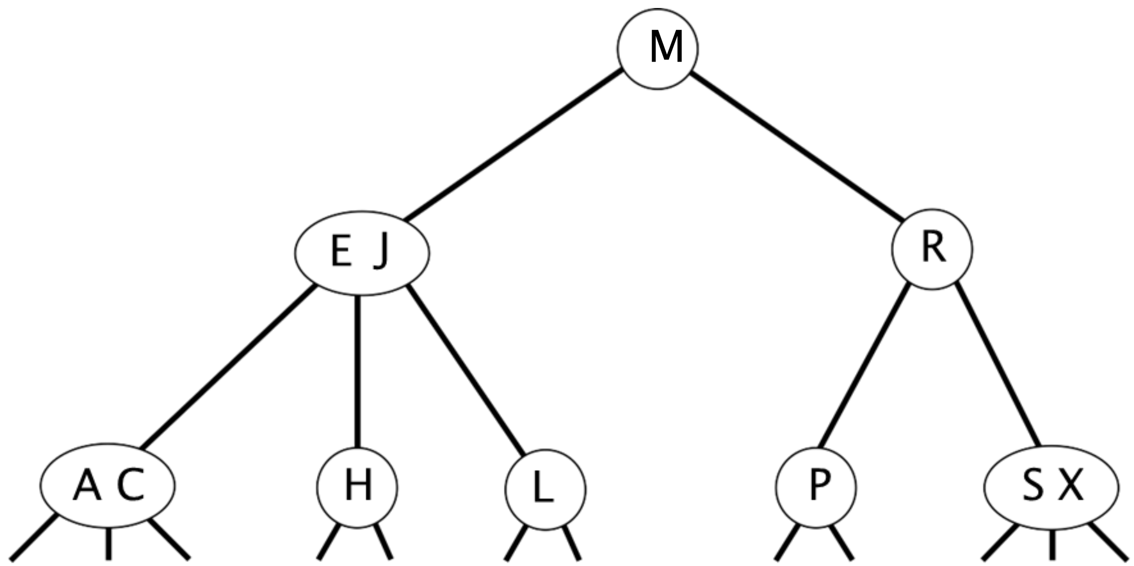
- Note: perfectly balanced

- Search simple
  - but maybe 2 comparisons needed



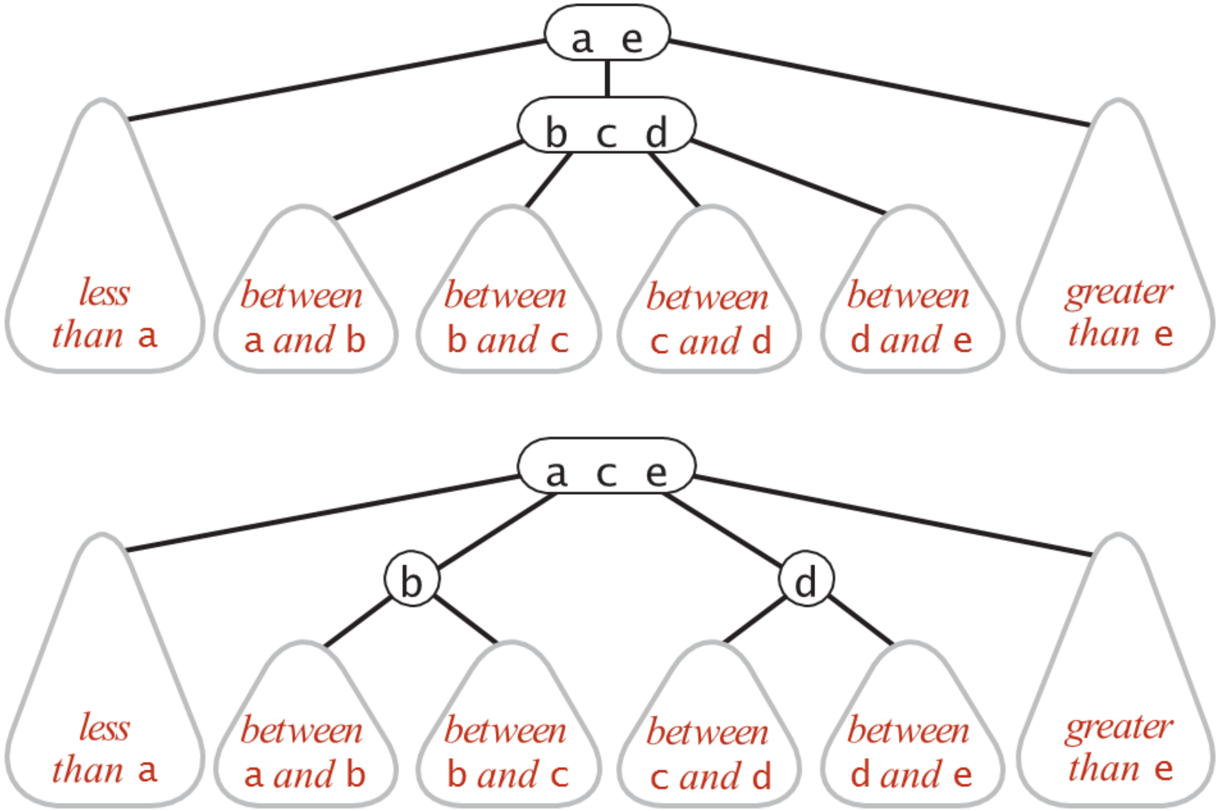
Insertion (maintaining balance)

- First case
  - search reaches a 2-node
  - e.g I
  - e.g. N



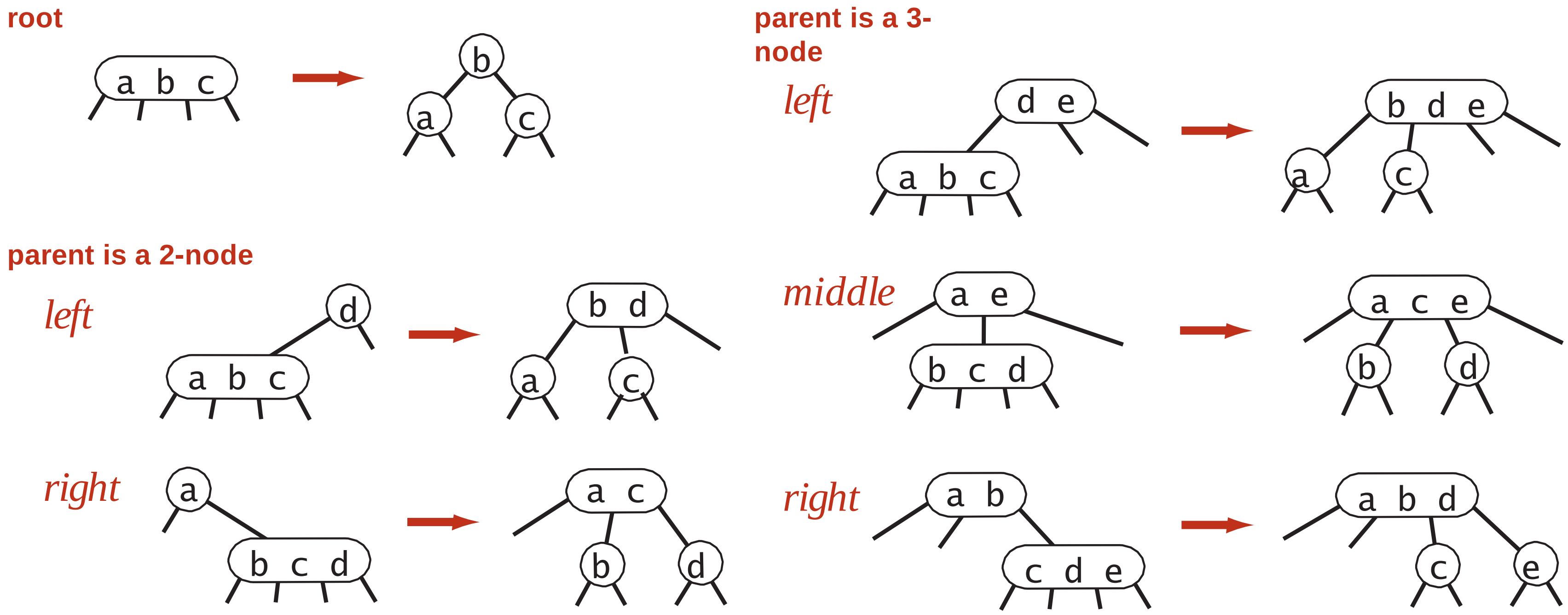
Creating a temporary 4-node

- Adding to 3-node
  - Create a temporary 4-node
  - Then split
- In the picture
  - Move 4-node one level up

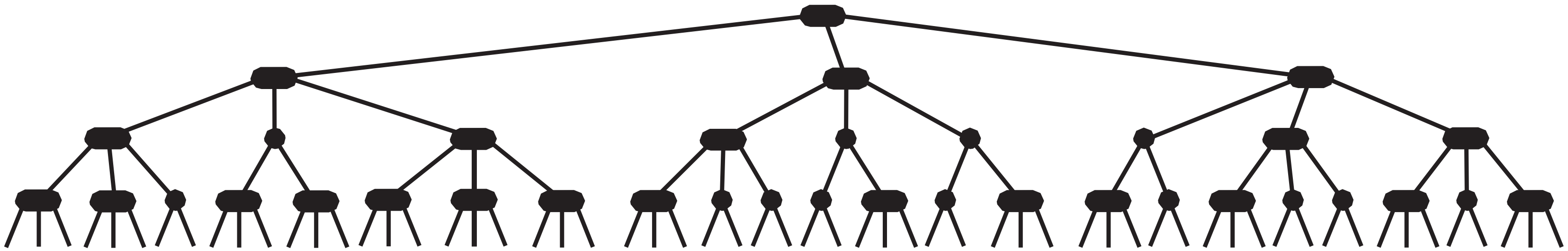


All cases

- All insertion cases shown below
  - Note splitting the root 4-node increases depth by 1 the only time depth increases



2-3 tree properties



- All paths same length
- Tree depth
  - Worst case lg N
  - Best case $\lg_3$ N
- Guaranteed logarithmic performance

Summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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	?	yes	`compareTo()`

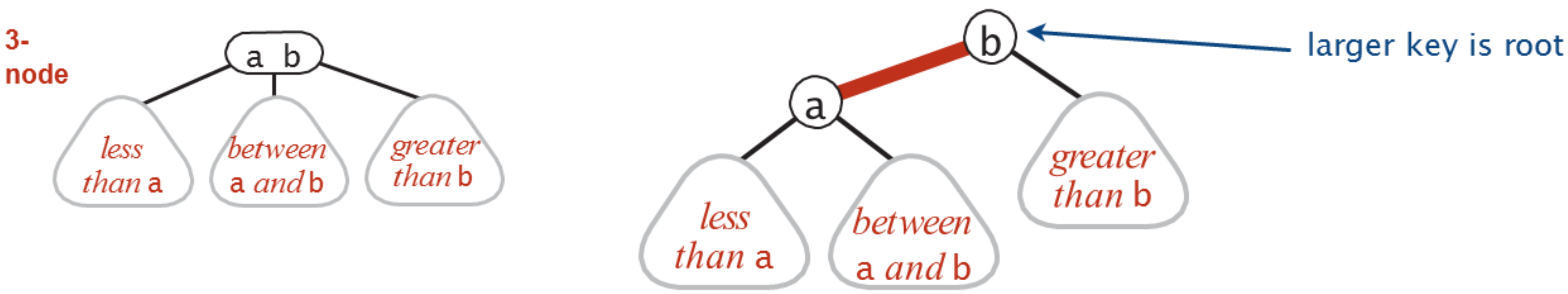
Implementation

- Could be done
- Bit complicated
- We won’t

- Instead present red-black trees
  - Which may be seen as encoding for 2-3 trees

Red-black BSTs

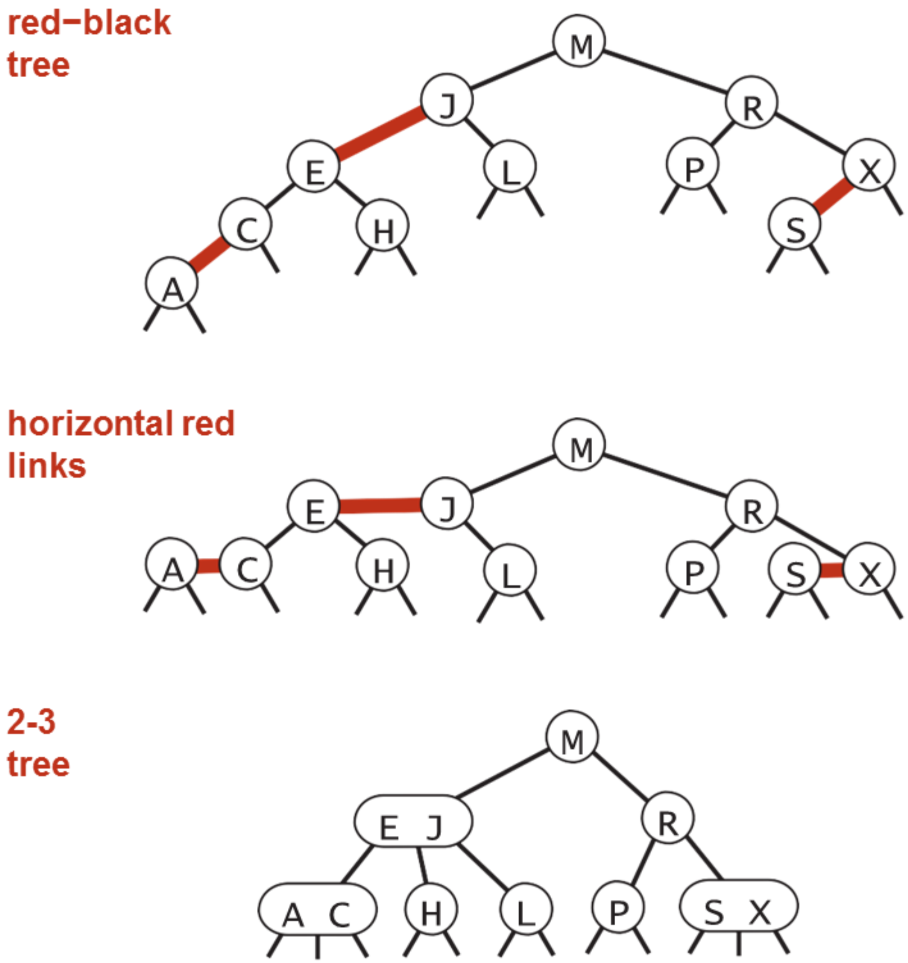
- The 3-node is encoded as a pair of 2-nodes



- Note:
  - Left-leaning
  - We distinguish between black and red links
  - At most one red link per node
  - Our goal perfect black balance

1:1 correspondence

- 1:1 correspondence between 2-3 and LLRB
  - LLRB: left-leaning red-black BST



Most operations

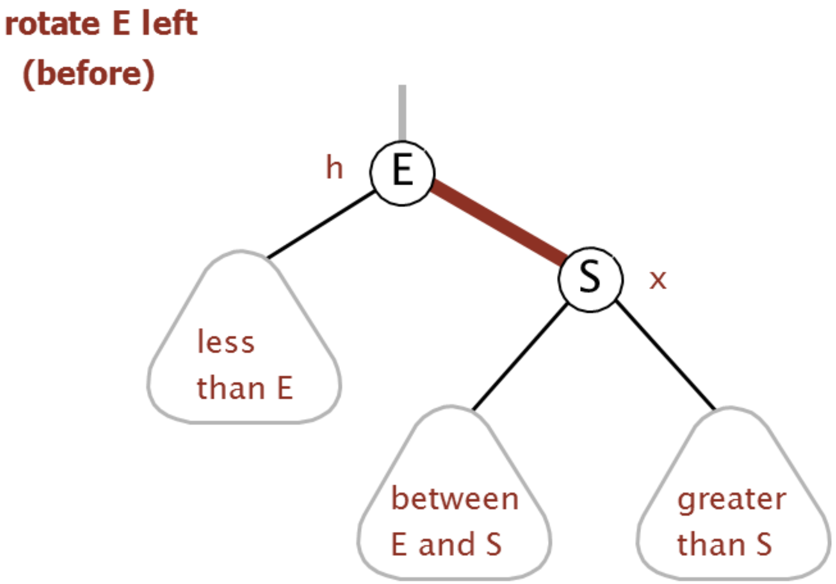
- Same as for other BSTs
  - Ignore color
-
- E.g., get, floor, rank, iteration, selection

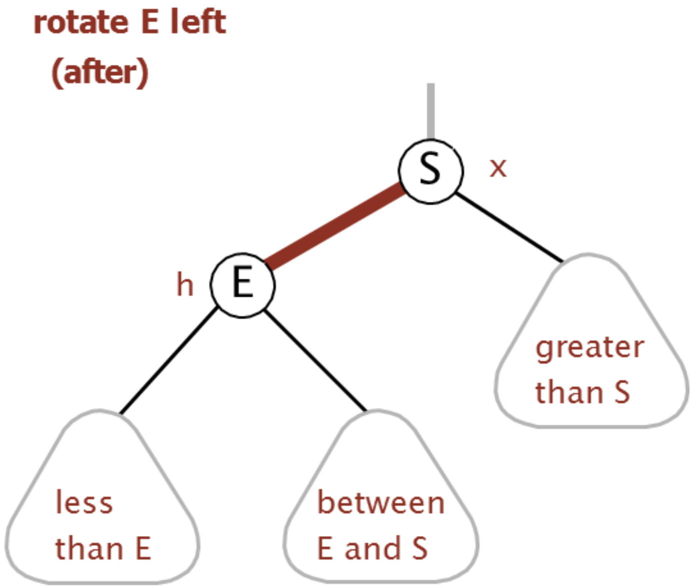
- Note: this would not be the case if we directly encode 2-3 trees.

- Differences in put, delete, etc.
- First we introduce 3 helper funtions

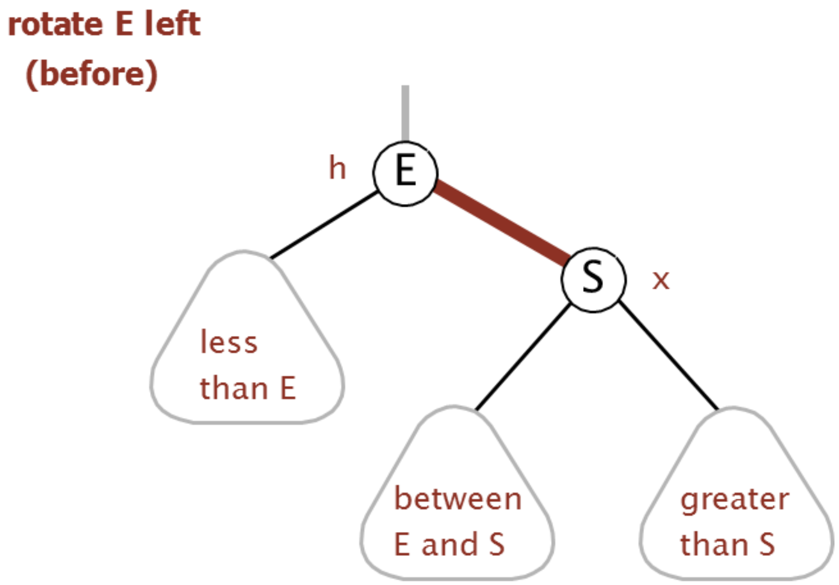
Helper function: Left rotation

- Left rotation
  - Fixes a (temporary) right-leaning red link





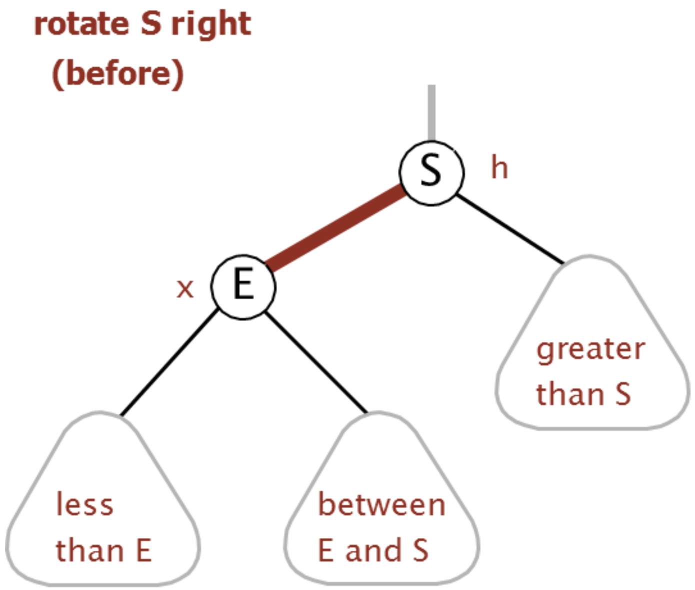
Left rotation

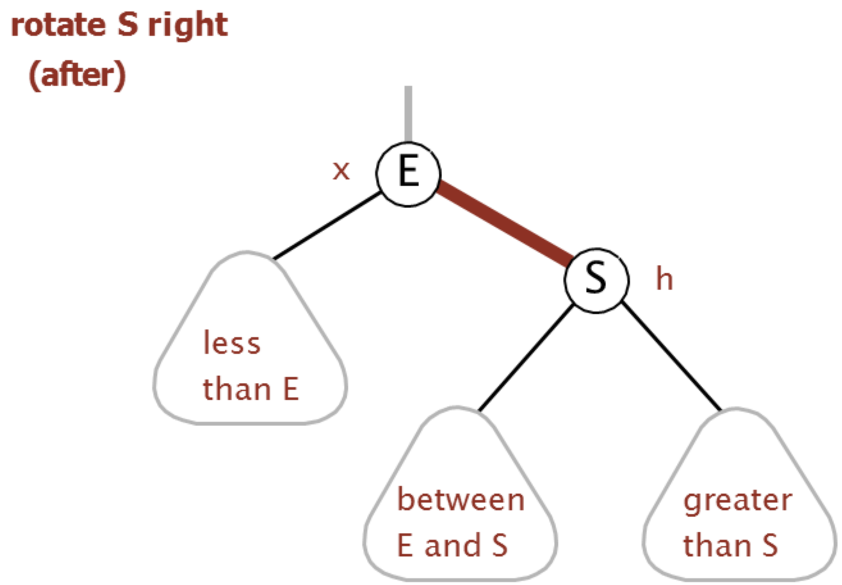



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

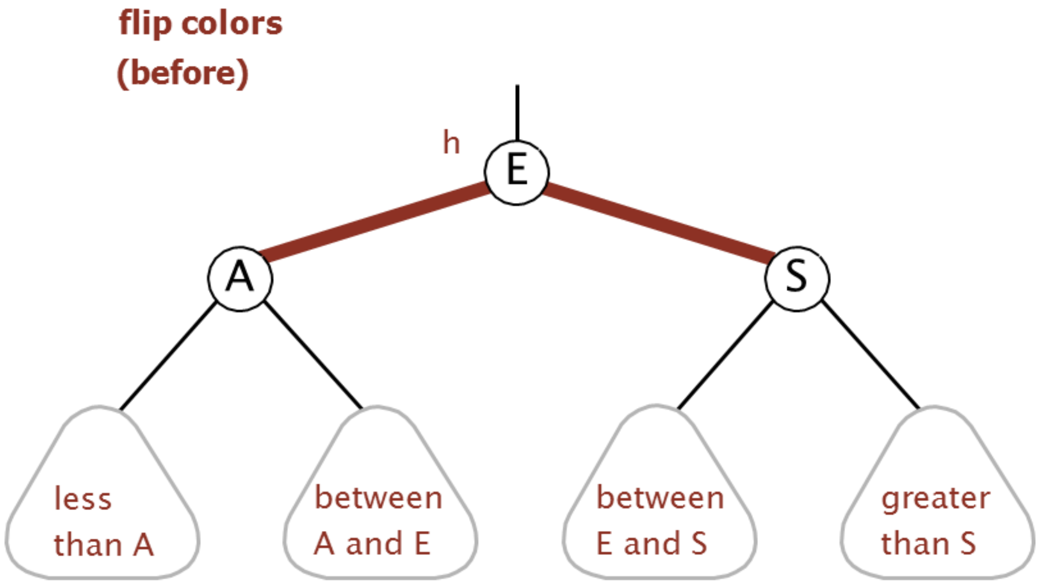
Right rotation

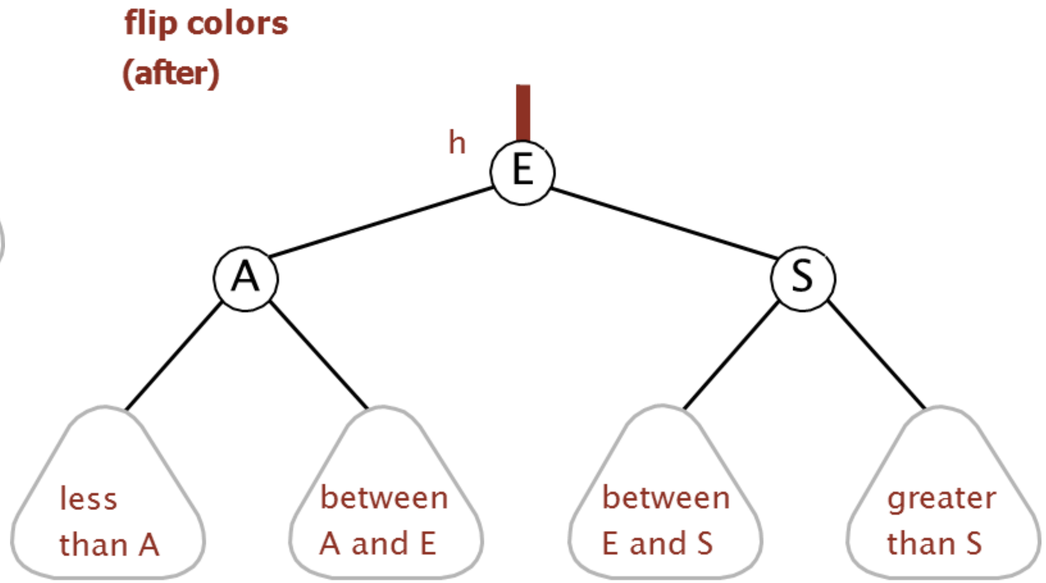
- Re-orient left leaning to (temporarily) lean right
- Will be made clear why we need this



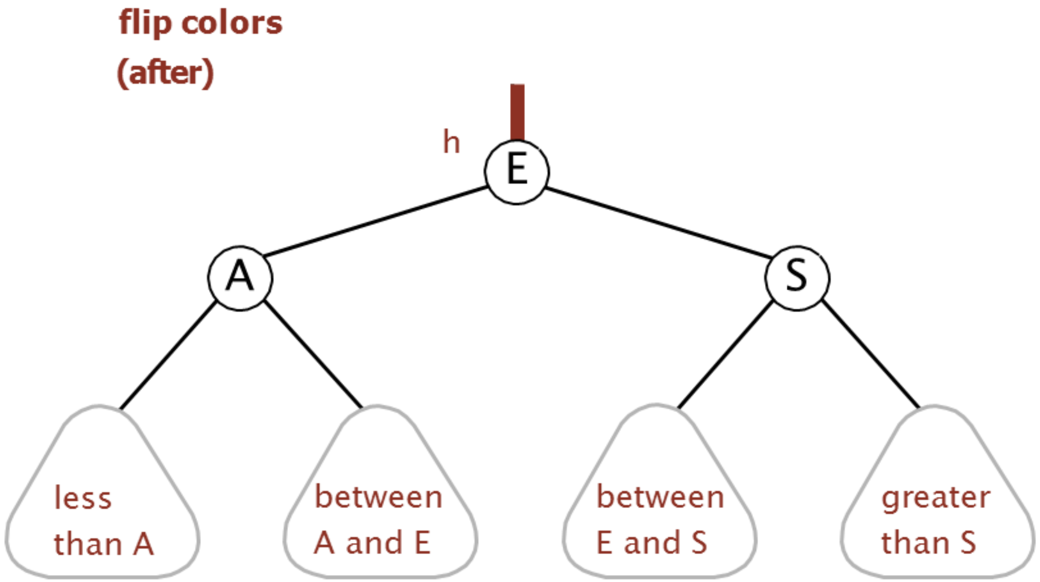


Color flip


flip colors
(before)
h
E
A
S
less
than A
between
A and E
between
E and S
greater
than S


flip colors
(after)
h
E
A
S
less
than A
between
A and E
between
E and S
greater
than S

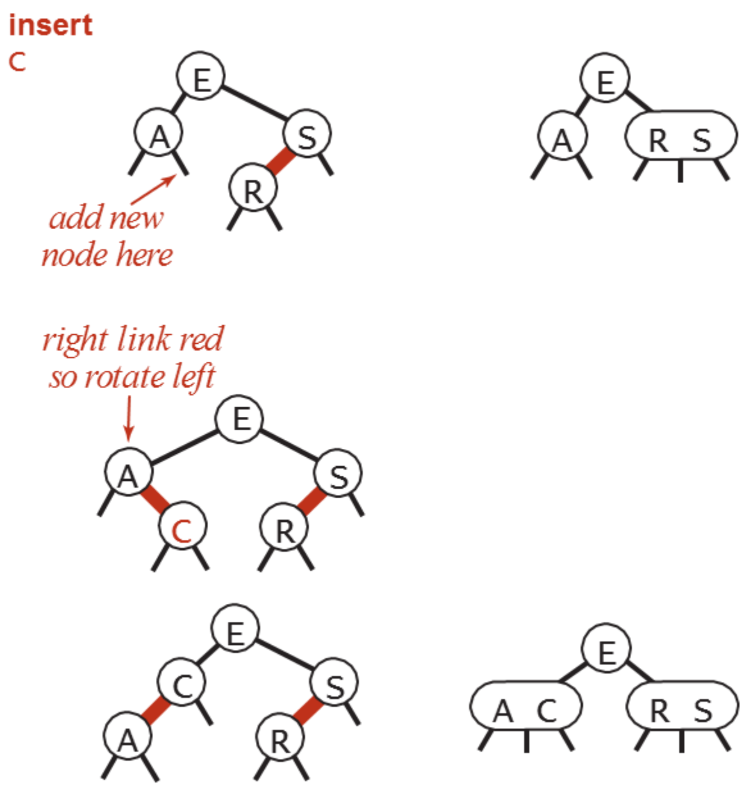
- Note that the parent link is made red




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

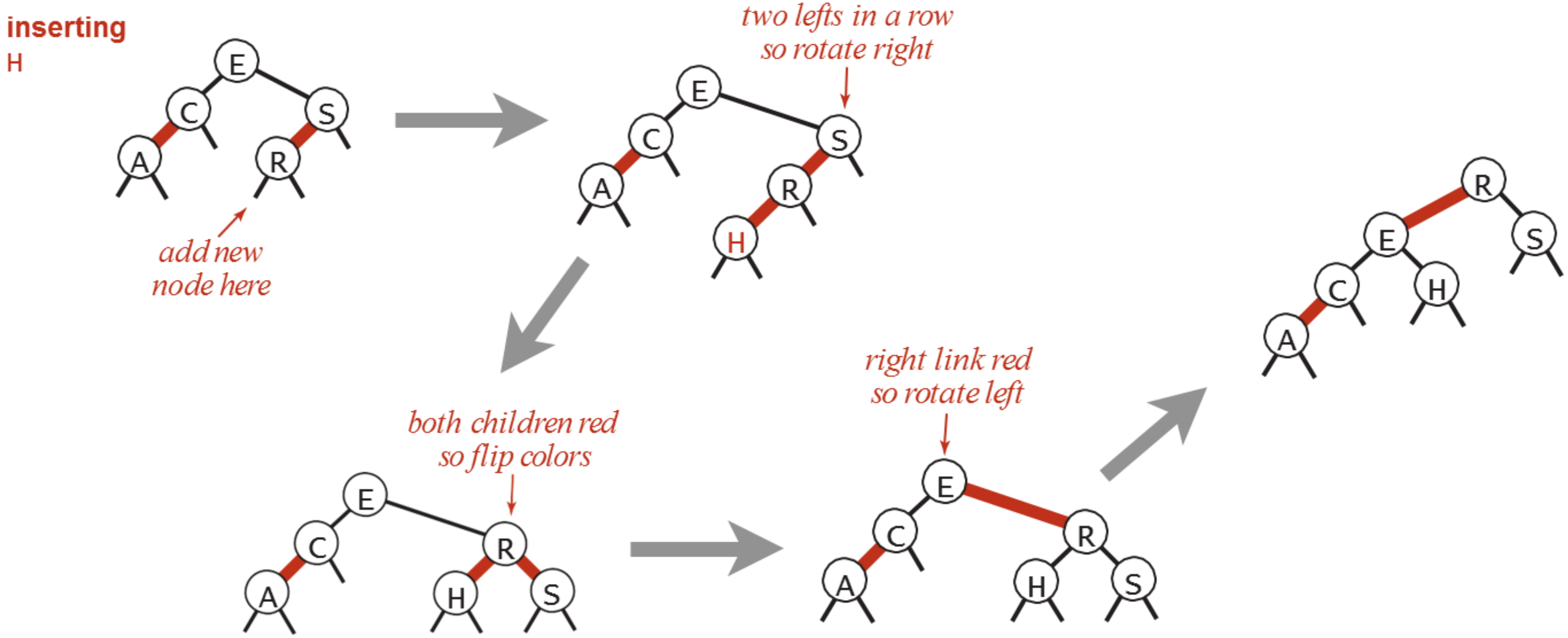
Insertion

- Case 1: Insert into 2-node at bottom
  - If new red link is right do a left rotation



Insertion (2)

- Case 2: Insert into 3-node at botoom
  - Do standard insert (color link red)
  - Rotate to balance 4-node(if needed)
  - Flip colors to pass red link up one level
  - Rotate to balance (if needed)
  - Repeat up the tree if needed



Insertion Implementation

- ,

```
private Node put(Node h, Key key, Value val)
{
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if      (cmp  < 0) h.left  = put(h.left,  key, val);
    else if (cmp  > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val = val;

    if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
    if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  && isRed(h.right))     flipColors(h);

    return h;
}
```

insert at bottom (and color it red)

lean left

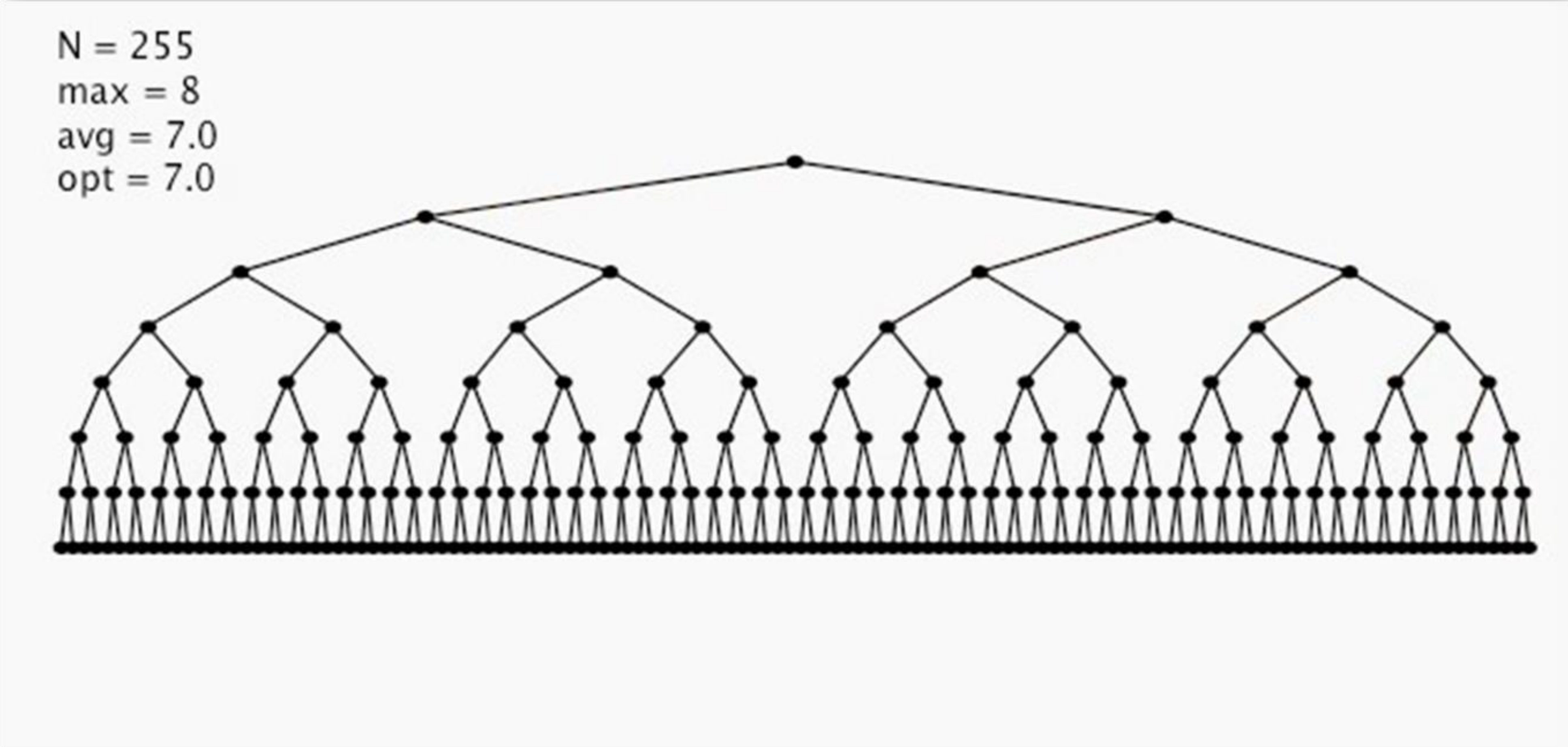
balance 4-node

split 4-node

only a few extra lines of code provides near-perfect balance

Visualization

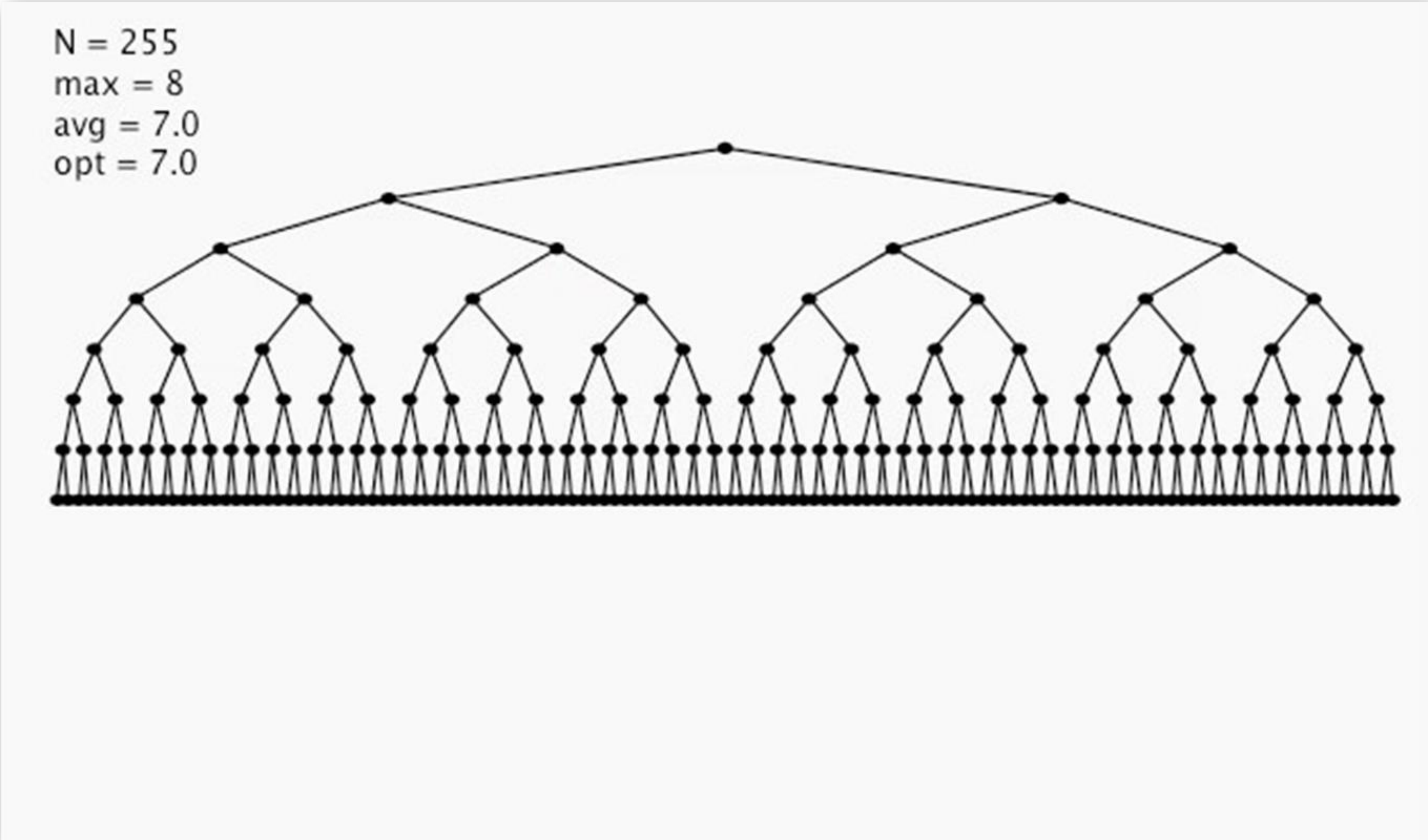
- Ascending order



255 insertions in ascending order

Visualization (2)

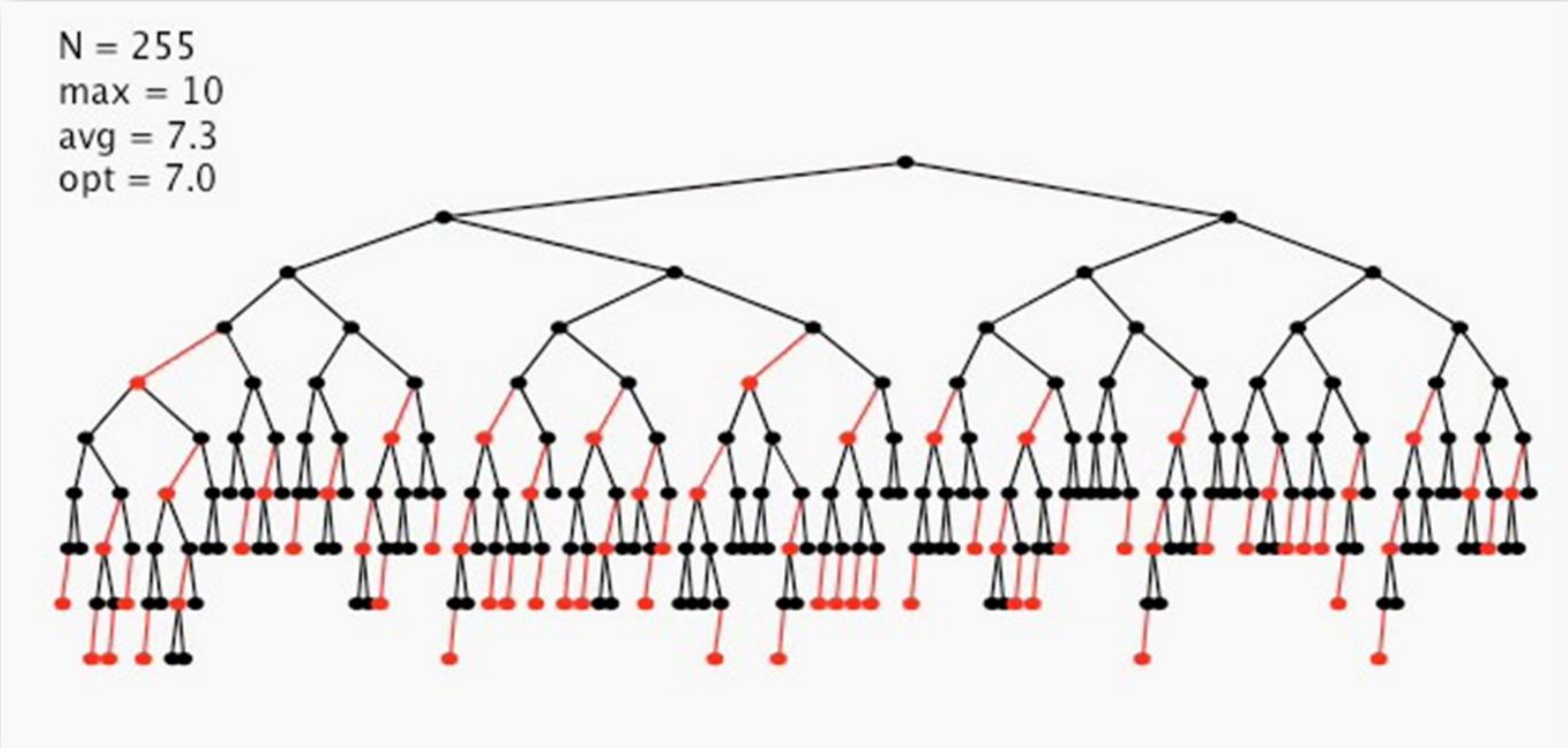
- Descending order



255 insertions in descending order

Visualization

- Random order



255 random insertions

Comparison

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered 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()
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	compareTo()

* exact value of coefficient unknown but extremely close to 1

True story

Telephone company contracted with database provider to build real-time database to store customer information.

Database implementation.

- Red-black BST search and insert; Hibbard deletion.
- Exceeding height limit of 80 triggered error-recovery process.

allows for up to 2^{40} keys

Extended telephone service outage.

Hibbard deletion was the problem

- Main cause = height bound exceeded!
- Telephone company sues database provider.
- Legal testimony:

> *" If implemented properly, the height of a red-black BST with N keys is at most 2 lg N. "* — *expert witness*


[redacted]

Other trees

- B-trees
  - Generalizes 2-3 trees
  - Each node has up to M-1 keys
  - Typically M is large so that M-1 keys just fits into a page

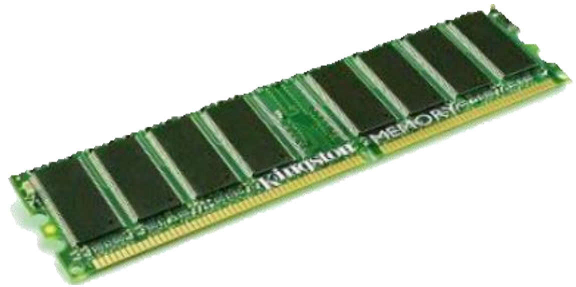
- AVL trees
  - Binary tree where depth varies by at most one
  - Rotation operations rebalance tree where needed

Hardware-dependent optimizations

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).

slow



fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

Goal. Access data using minimum number of probes.

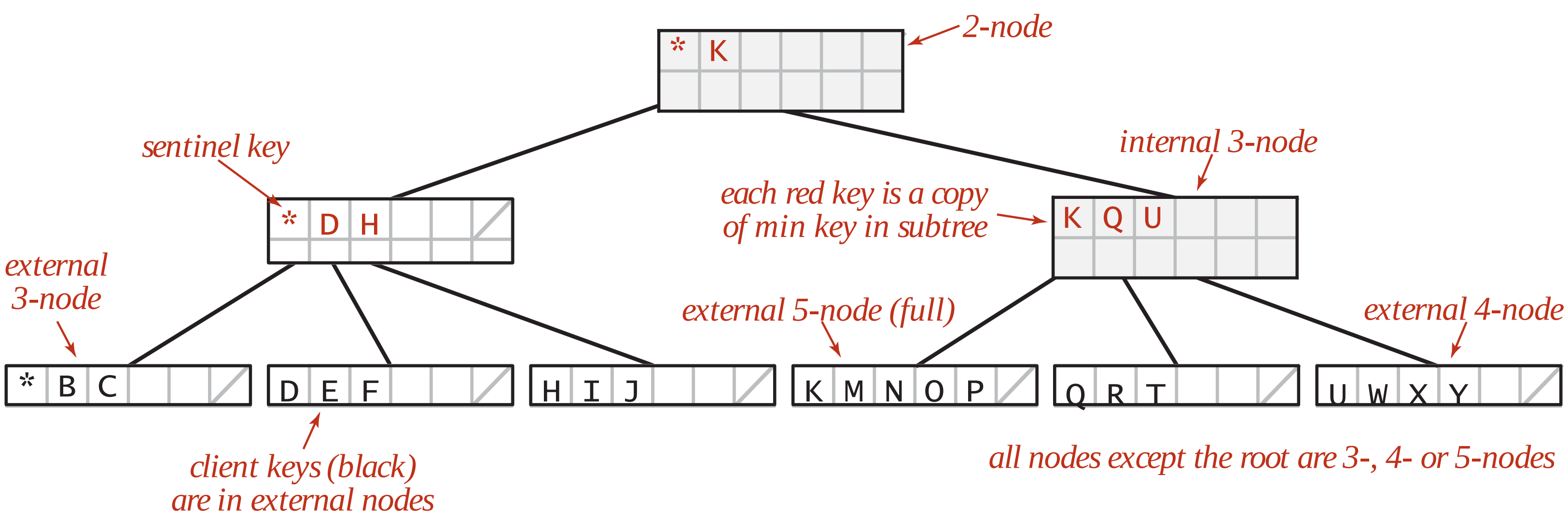
B-trees

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide

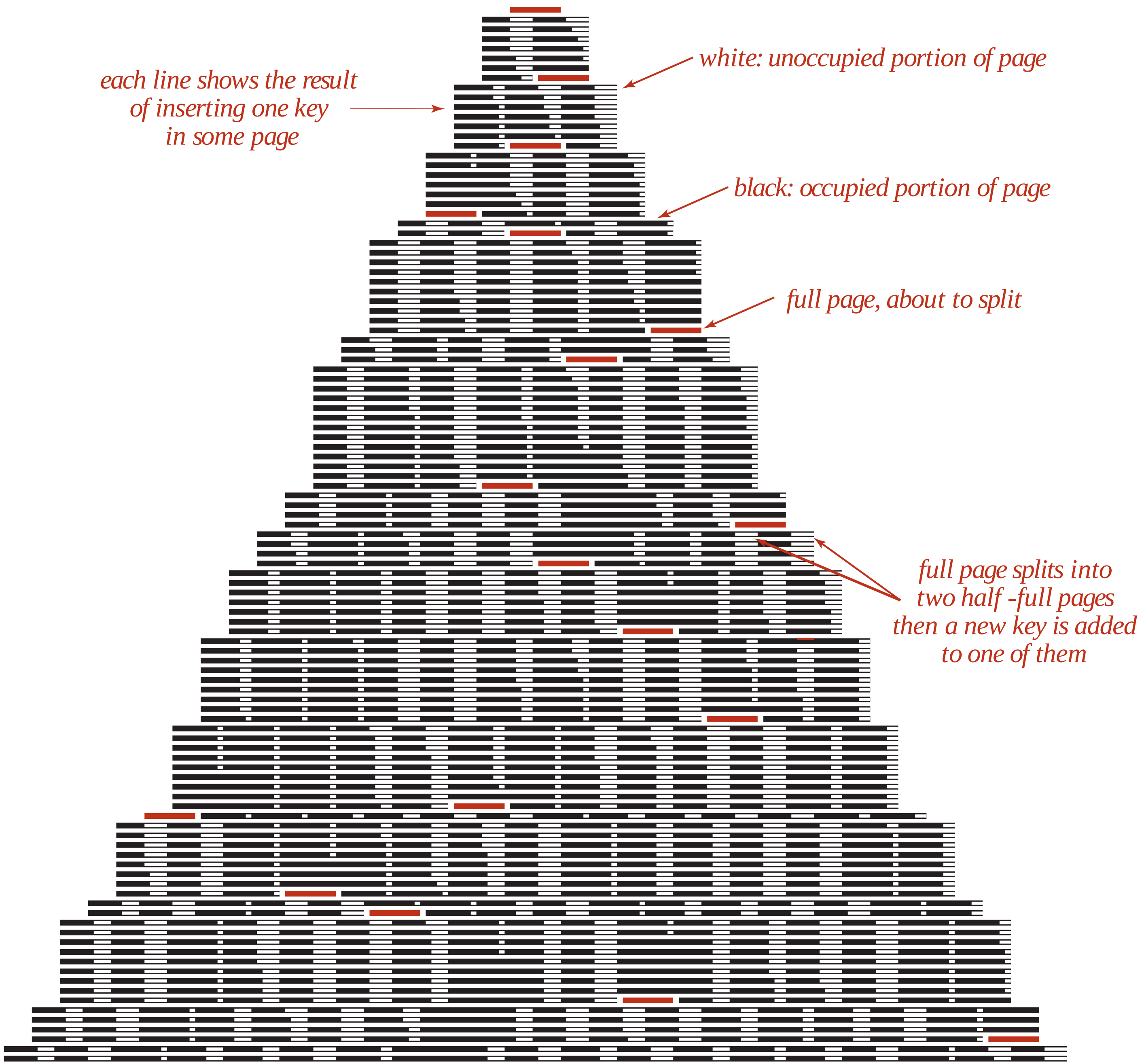
search.

choose M as large as possible so that M links fit in a page, e.g., M = 1024



Anatomy of a B-tree set (M = 6)

Large B-tree illustrated



Some applications

- Red-black trees widely used
  - Java
  - Linux kernel
  - Emacs
- B-trees and variants
  - Widely used for file systems and databases.
  - E.g., Windows, Mac, Linux
  - E.g, Oracle, DB2