CSC 226

Algorithms and Data Structures: II

More Red-Black Trees

Tianming Wei
twei@uvic.ca
ECS 466

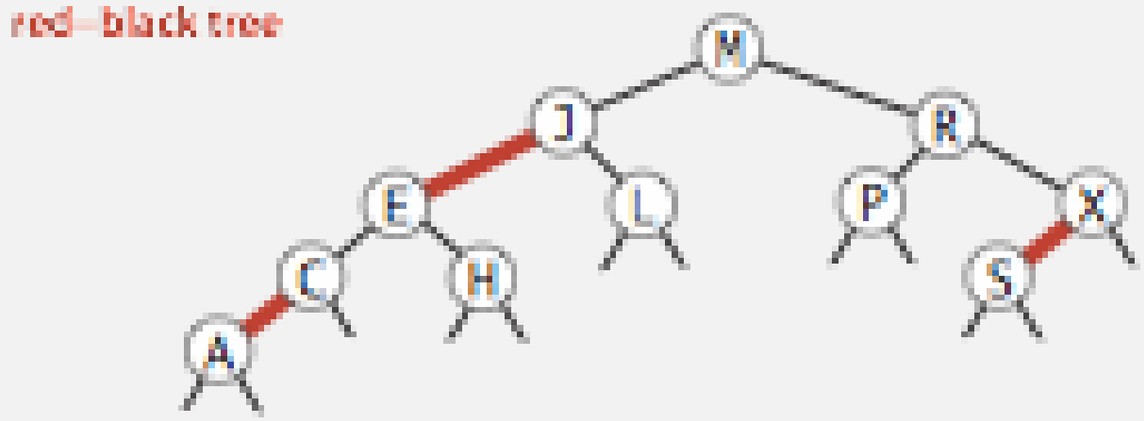
Red-black trees and 2-3 trees

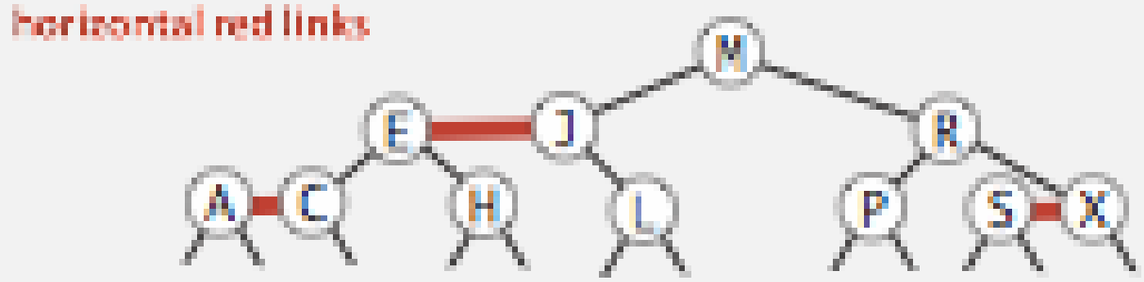
- There is a 1-1 correspondence between red-black BSTs and 2-3 trees.

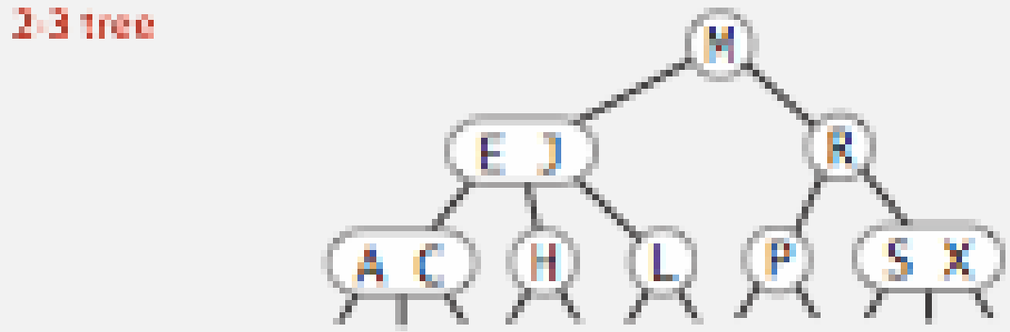
- To see this, imagine that red links are collapsed: the collapsed nodes correspond to 3-nodes, all others to 2-nodes.

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

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







We show: For every red-black tree there is a 2-3 tree

- Given a red-black tree T, we build a 2-3 tree T'
- The nodes of the 2-3 tree T' are obtained as follows.
  - for every node v_{rb} in the red-black tree with key k that is incident to black edges only create a 2-node v_{23} for the 2-3 tree with key k. That is: $23(v_{rb}) = v_{23}$
  - for every red edge and its incident two nodes v_{rb} and w_{rb}—where w_{rb} is the parent of v_{rb}—containing keys k_1 and k_2, respectively, create a 3-node vw_{23} containing keys k_1 and k_2 (with k_1 being the left entry). That is: $23(v_{rb}) = 23(w_{rb}) = vw_{23}$

We show: For every red-black tree there is a 2-3 tree

- The edges of the 2-3 tree T' are obtained as follows

  - Let $u_{rb}v_{rb}$ be a **black** edge in the red-black tree. Then $23(u_{rb})23(v_{rb})$ is an edge in the 2-3 tree. Further,

    - if u_{rb} is the left child of v_{rb} then $23(u_{rb})$ is the left child of $23(v_{rb})$

    - else if u_{rb} is the right child of v_{rb}, and the edge between v_{rb} and it's parent is red, then $23(u_{rb})$ is $23(v_{rb})$'s middle child

    - else $23(u_{rb})$ is the right child of $23(v_{rb})$

We show: For every 2-3 tree there is a red-black tree

- Given a 2-3 tree T, we build a red-black tree T'

- The nodes of the red-black tree T' are obtained as follows.

  - for every 2-node v_{23} in the red-black tree with key k create node v_{rb} with key k.

  - for every 3-node vw_{23} with keys k_1 and k_2, $k_1 \leq k_2$, create nodes v_{rb} and w_{rb} containing keys k_1 and k_2, respectively.

We show: For every red-black tree there is a 2-3 tree

- The edges of the red-black tree are obtained as follows
  - for every remaining edge between 2-node w_{23} and child v_{23} the corresponding red-black tree nodes are connected by a black edge
  - for every 3-node vw_{23} with keys k_1 and k_2, $k_1 \leq k_2$, and its corresponding red-black tree nodes v_{rb} with key k_1 and w_{rb} with key k_2 we define recursively:
    - add **red** edge $v_{rb}w_{rb}$ in red-black tree T' with v_{rb} being w_{rb}'s left child
    - the red-black tree of vw_{23}'s left subtree is v_{rb}'s left subtree, the red-black tree of vw_{23}'s middle subtree is v_{rb}'s right subtree, and the red-black tree of vw_{23}'s right subtree is w_{rb}'s right subtree

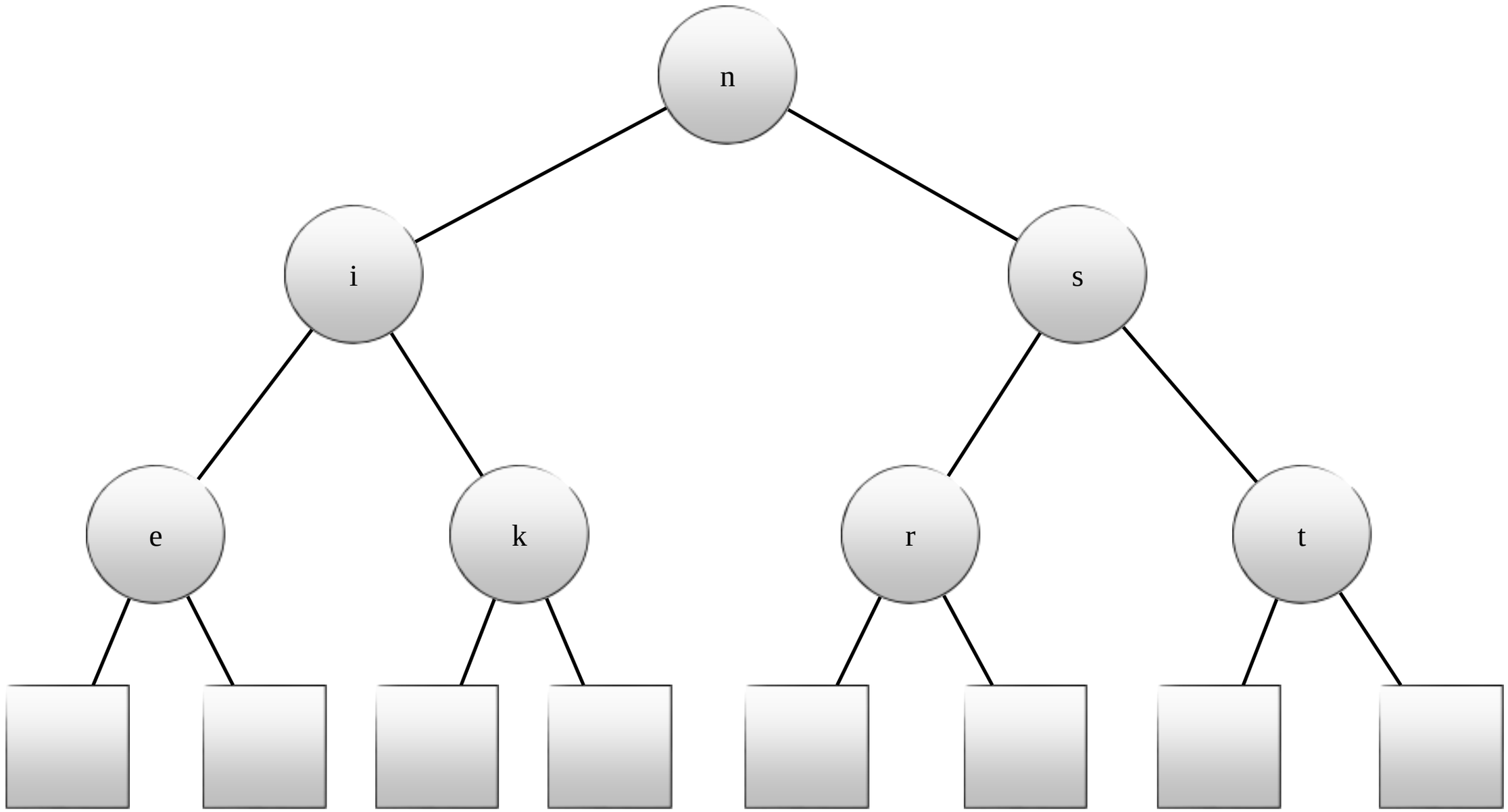
The height of a red-black tree with n keys is $\mathrm{O}(\log(n))$

- *Proof Idea.* The (black) path length from leaf to root is the same as the height of its corresponding 2-3 tree, that is $\mathrm{O}(\log(n))$. The height of a red-black tree can be twice the height of the 2-3 tree, namely in the case that every second edge in a path from leaf to root is red. The height remains $\mathrm{O}(\log(n))$.

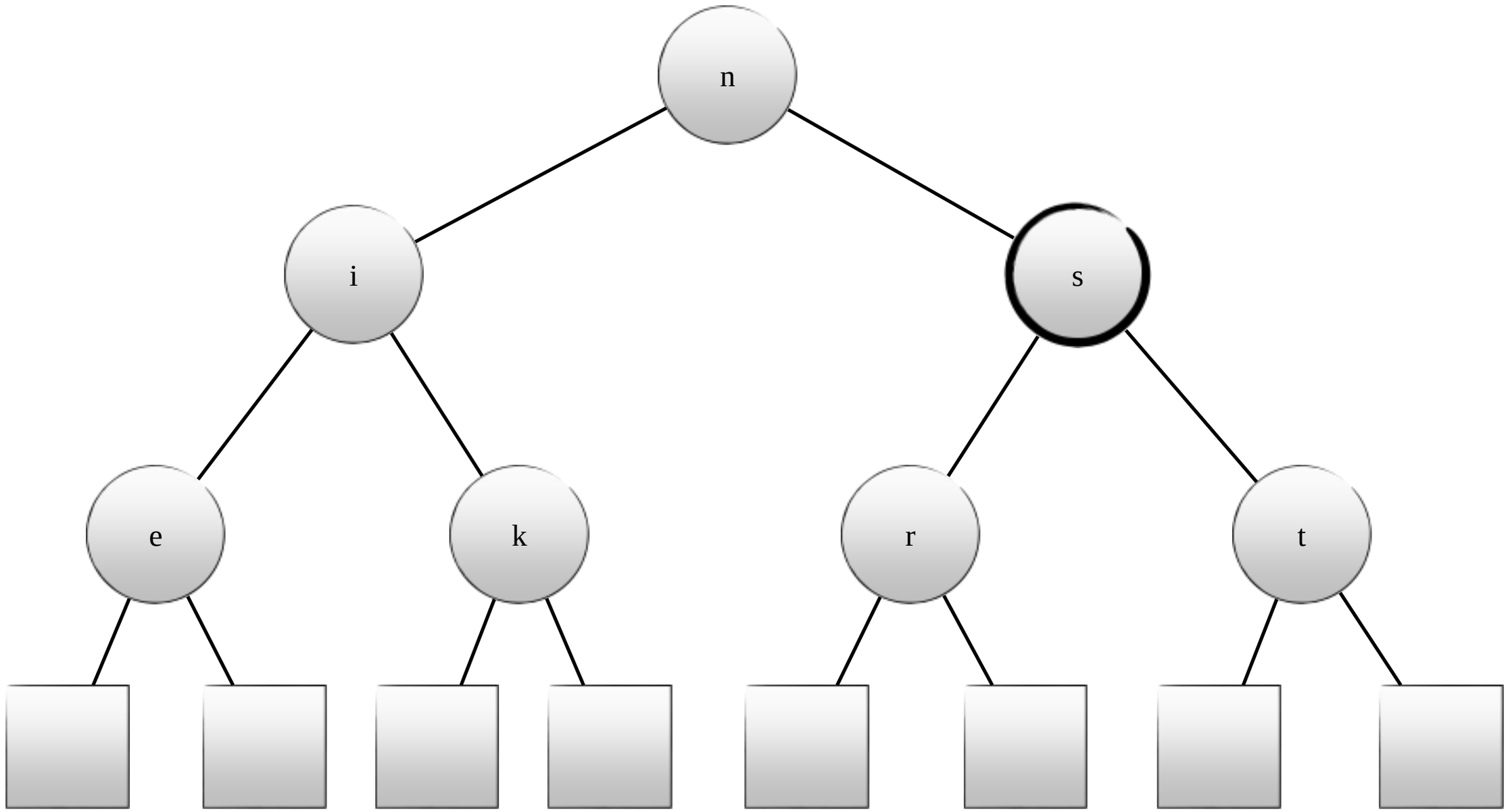
Deletion from 2-3/red-black trees

- Deletion is more problematic than insertion. Why?
- During insertion we can only cause problems with the colouring of the nodes
  - i.e. violate the left red and/or one red edge properties.
- During deletion we can cause black-depth imbalance as well.

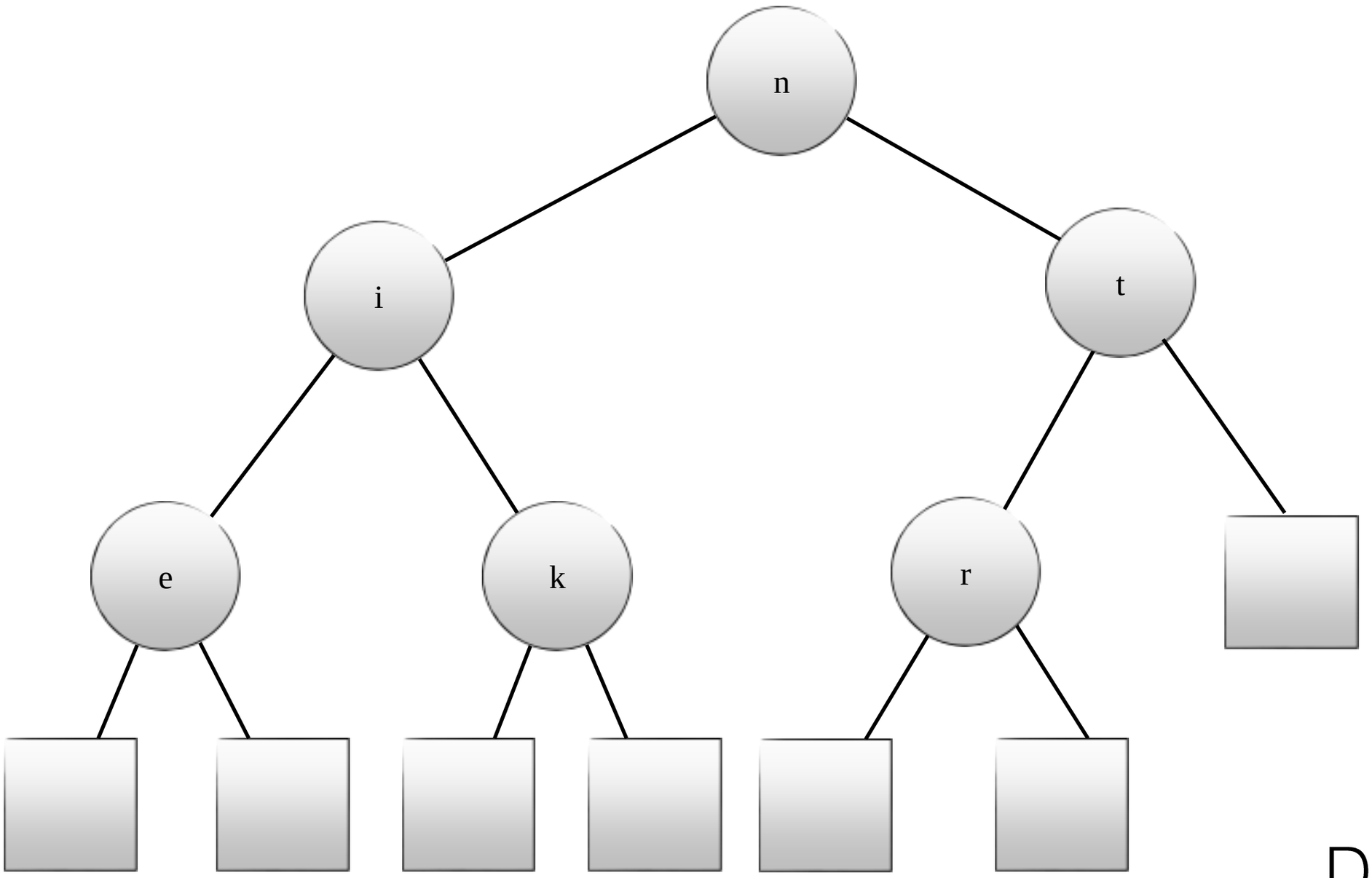
Delete key *s*



Delete key s

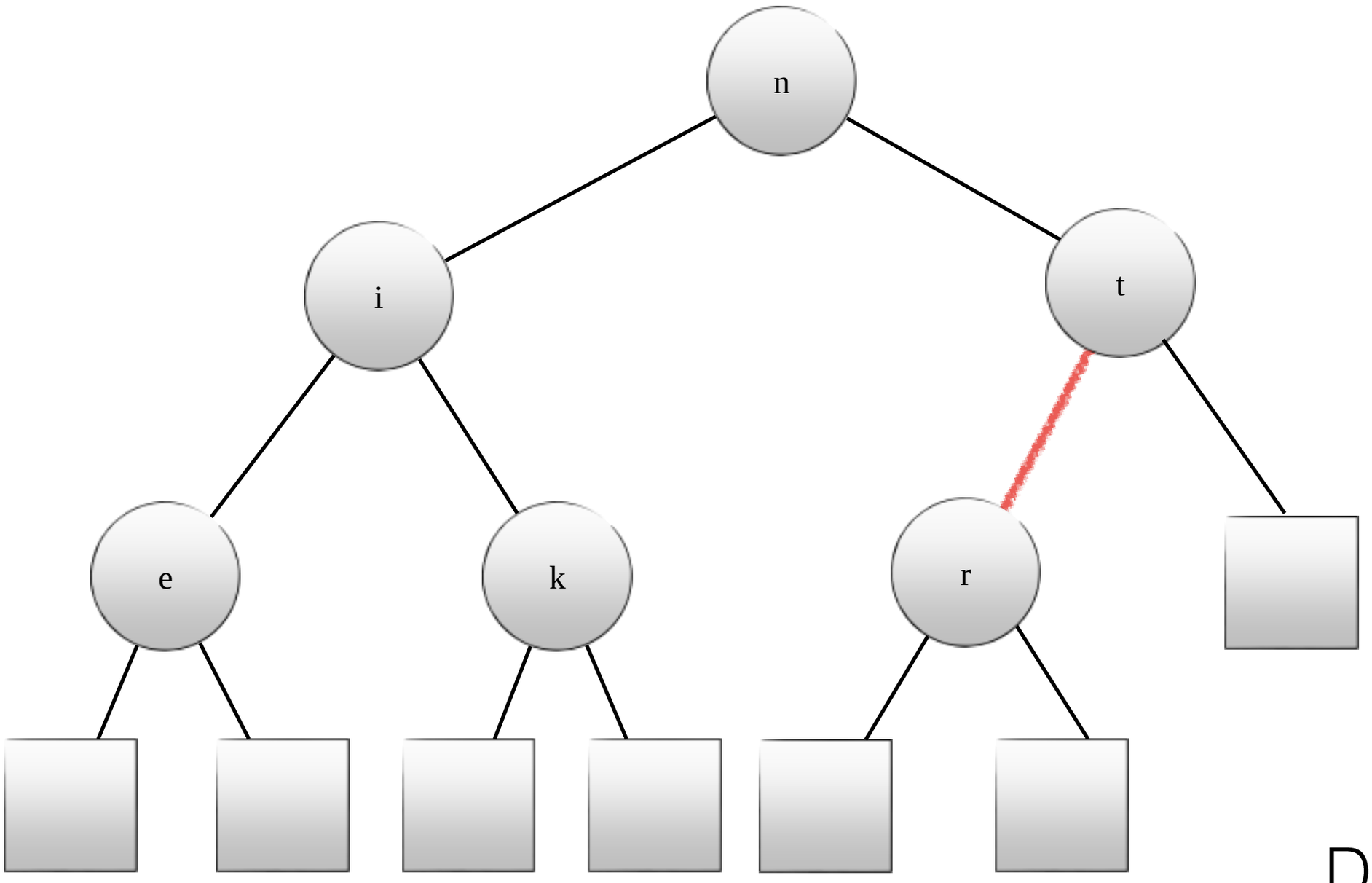


Delete key s



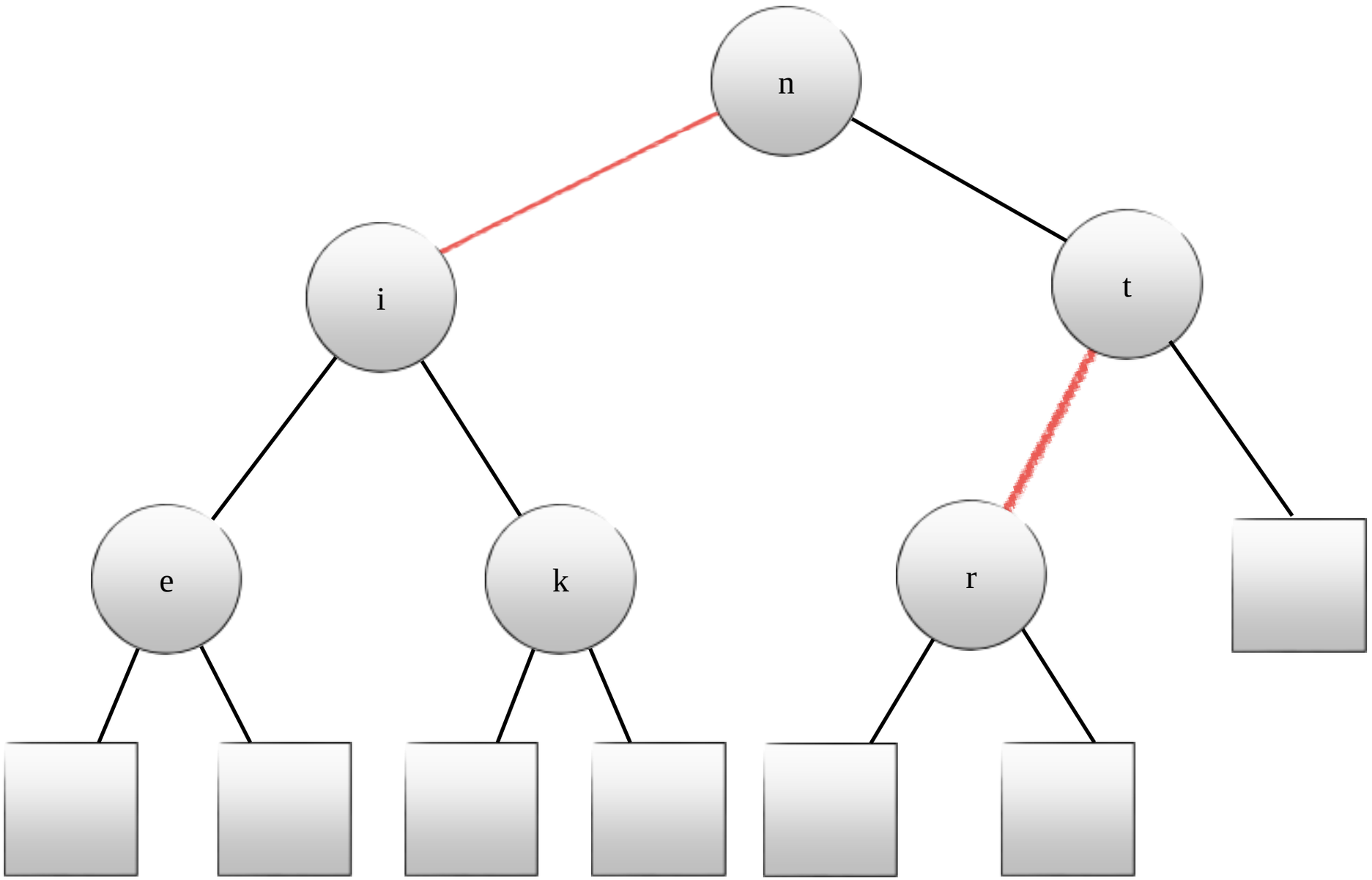
Doesn’t work

Delete key s



Doesn’t work

Delete key s



We need!

Deletion from 2-3/red-black trees

- Deletion is slightly more complex than insertion as you need to account for a potential imbalance on the way “down”

- So, we rotate and/or color flip both on the way down the recursion as well as on the way up.

- Generally, we make 2-nodes into 3-nodes or 4-nodes on the way down to accommodate a removal then correct the red edges on the way back up if needed.

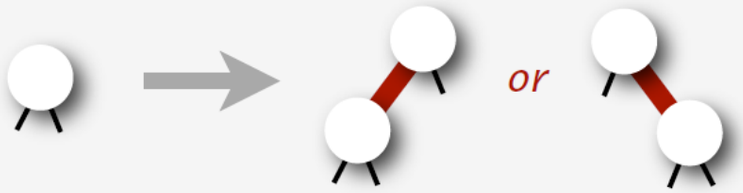
https://www.cs.princeton.edu/~rs/talks/LLRB/08Dagstuhl/RedBlack.pdf

Insert code for LLRB trees

is based on three simple operations.

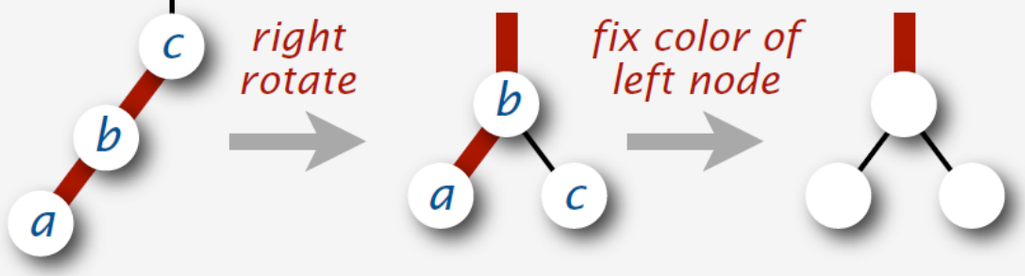
1. Insert a new node at the bottom.

```
if (h == null)
      return new Node(key, value, RED);
```



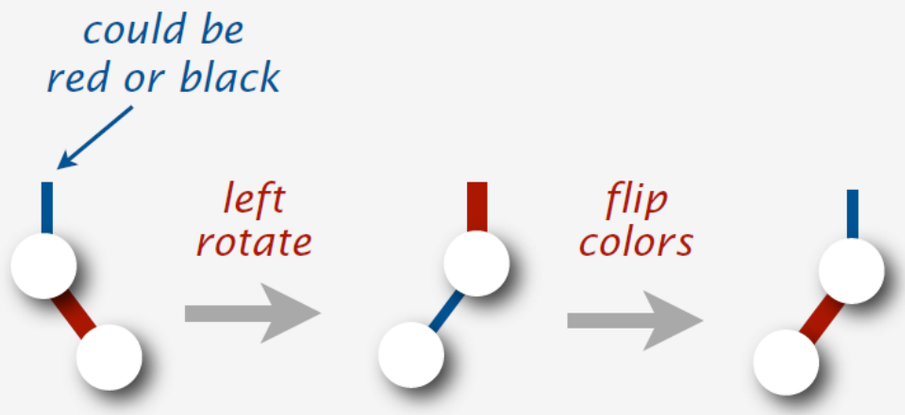
2. Split a 4-node.

```
private Node splitFourNode(Node h)
{
   x = rotR(h);
   x.left.color  = BLACK;
   return x;
}
```



3. Enforce left-leaning condition.

```
private Node leanLeft(Node h)
{
   x = rotL(h);
   x.color      = x.left.color;
   x.left.color = RED;
   return x;
}
```



Insert implementation for LLRB trees

is a few lines of code added to elementary BST insert


```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);

   if (isRed(h.left))
      if (isRed(h.left.left))
         h = splitFourNode(h);

   int cmp = key.compareTo(h.key);
   if (cmp == 0) h.val = val;
   else if (cmp < 0)
      h.left = insert(h.left, key, val);
   else
      h.right = insert(h.right, key, val);

   if (isRed(h.right))
      h = leanLeft(h);

   return h;
}
```

insert at the bottom

split 4-nodes on the way down

standard BST insert code

fix right-leaning reds on the way up

Why revisit red-black trees?

Take your pick:

```
private Node insert(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp == 0) x.val = val;
   else if (cmp < 0))
   {
     x.left = insert(x.left, key, val, false);
     if (isRed(x) && isRed(x.left) && sw)
        x = rotR(x);
     if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotR(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else // if (cmp > 0)
   {
      x.right = insert(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotL(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotL(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   return x;
}
```

very tricky


```
private Node insert(Node h, Key key, Value val)
{
    int cmp = key.compareTo(h.key);
    if (h == null)
      return new Node(key, val, RED);
    if (isRed(h.left))
      if (isRed(h.left.left))
      {
         h = rotR(h);
         h.left.color  = BLACK;
      }
   if (cmp == 0) x.val = val;
   else if (cmp < 0)
      h.left = insert(h.left, key, val);
   else
      h.right = insert(h.right, key, val);
   if (isRed(h.right))
   {
      h = rotL(h);
      h.color      = h.left.color;
      h.left.color = RED;
   }
   return h;
}
```

straightforward

Left-Leaning Red-Black Trees

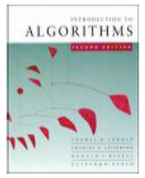
Robert Sedgewick
Princeton University

Why revisit red-black trees?

Take your pick:

TreeMap.java

Adapted from CLR by experienced professional programmers (2004)

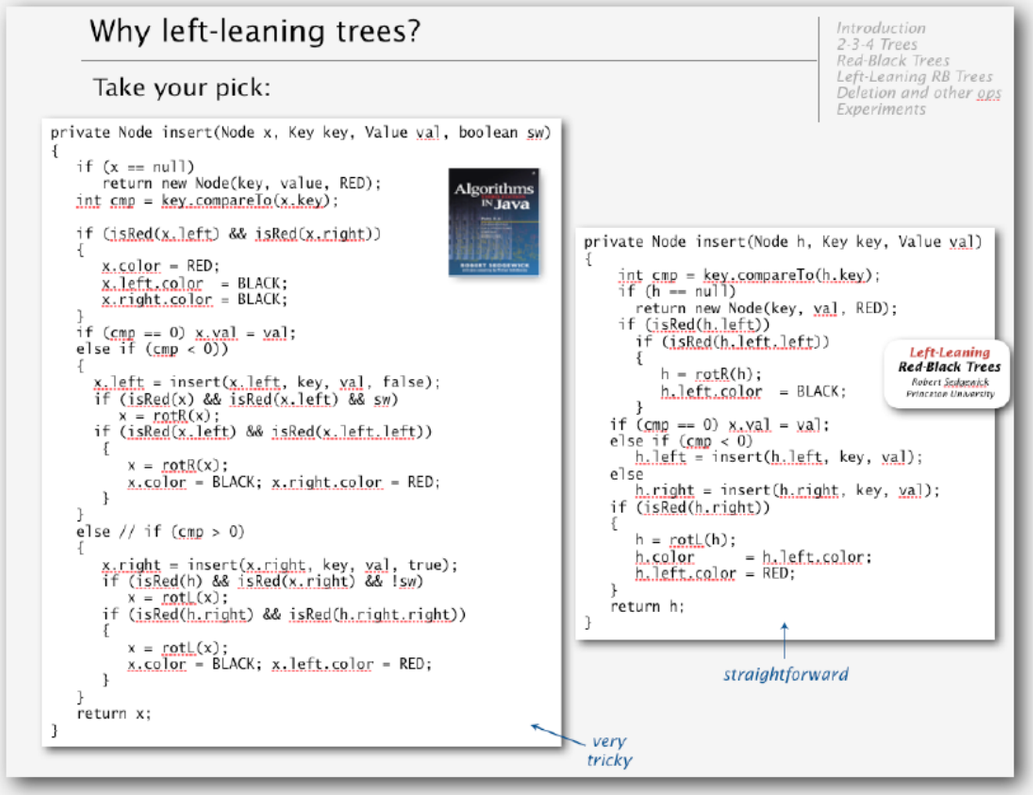


150

40

30

wrong scale!

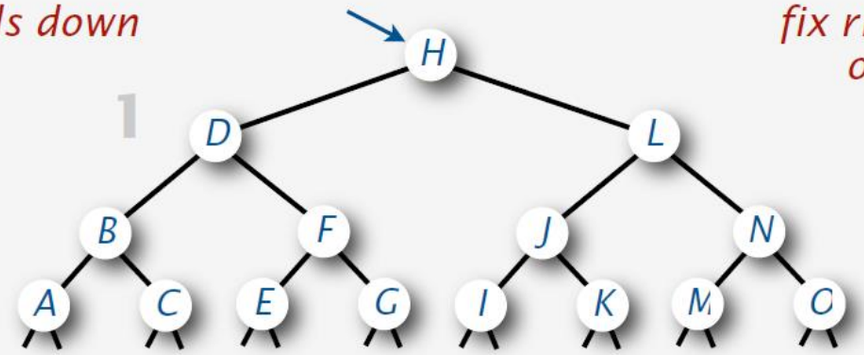


← *lines of code for insert (lower is better!)*

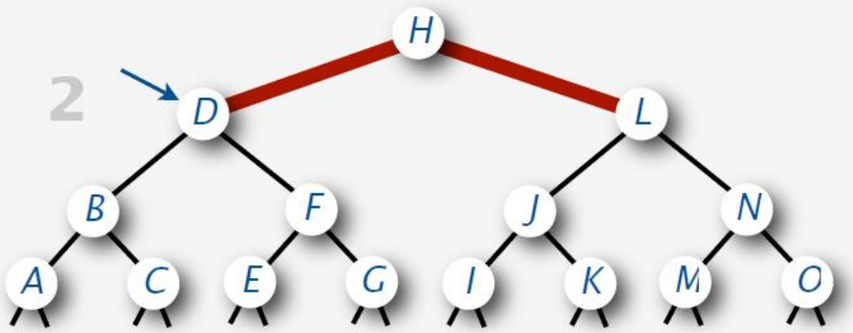
deleteMin() example

push reds down

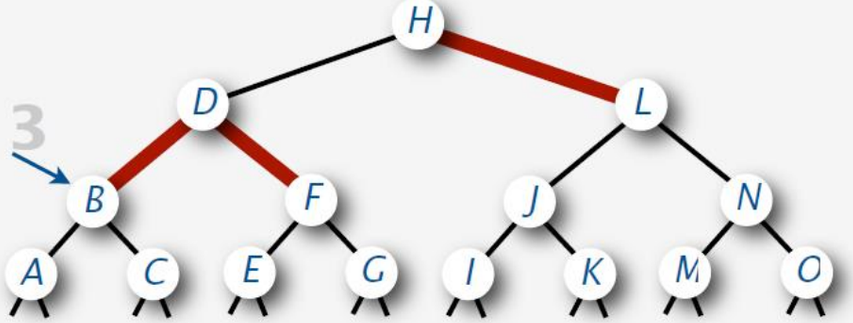
1



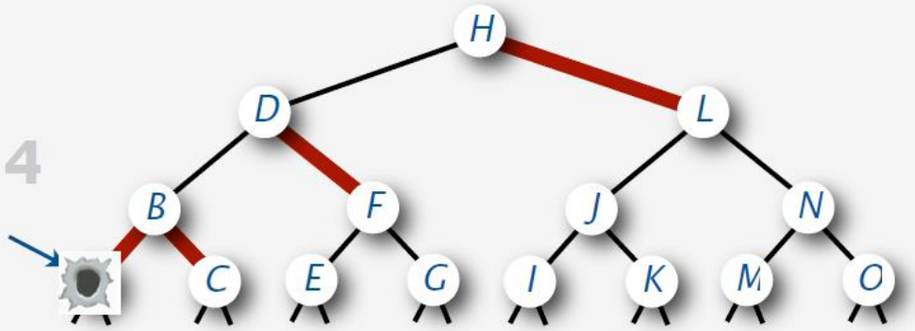
2



3

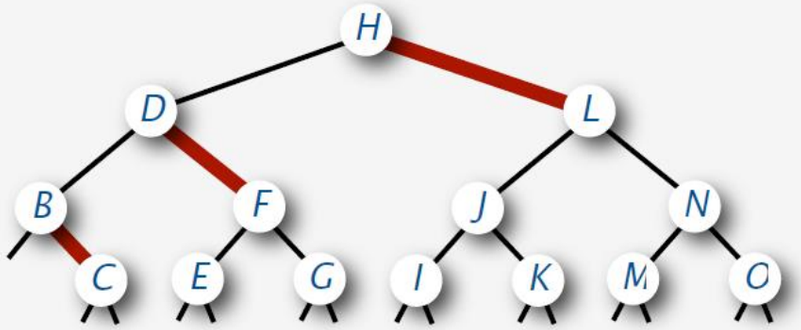


4



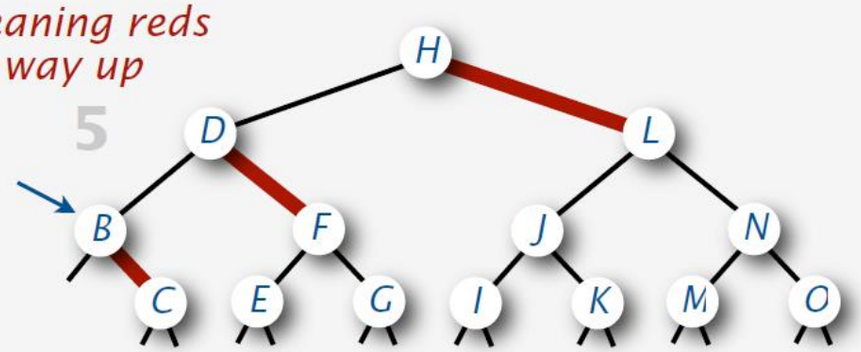
remove minimum

5

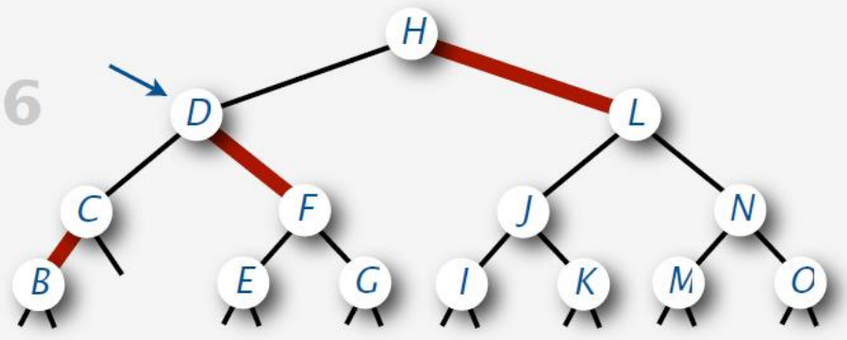


fix right-leaning reds on the way up

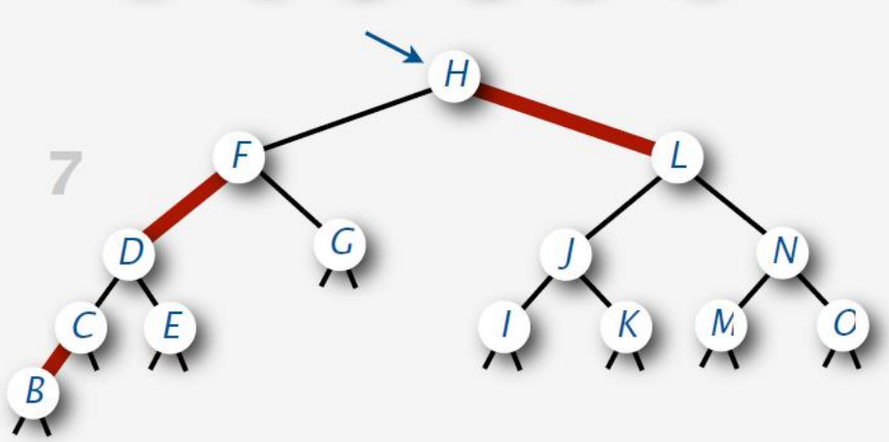
5



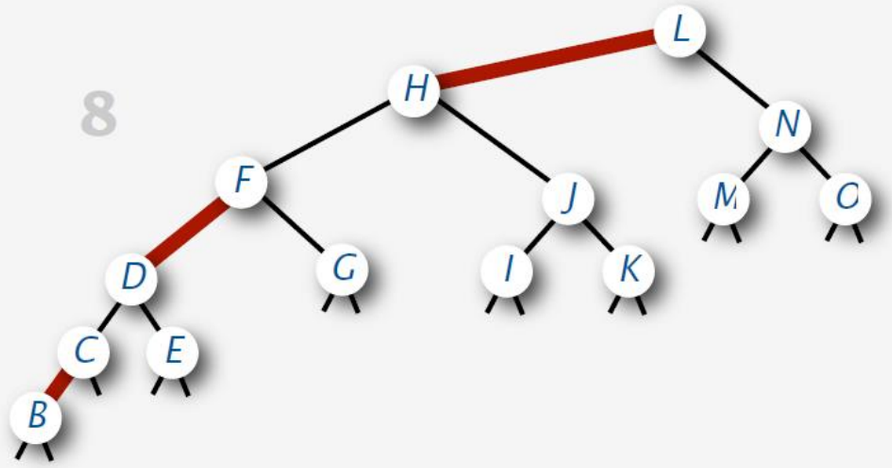
6



7



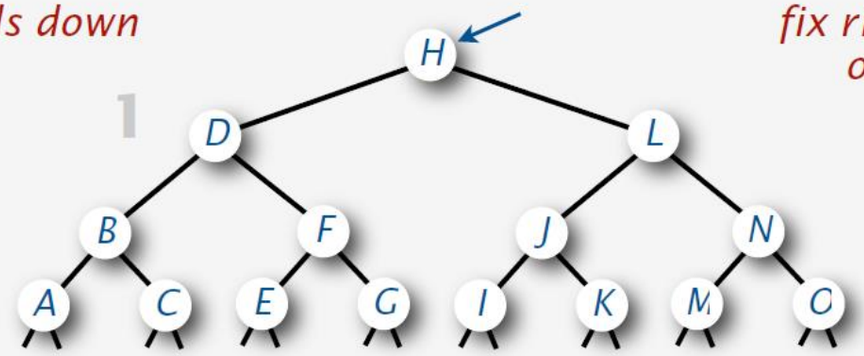
8



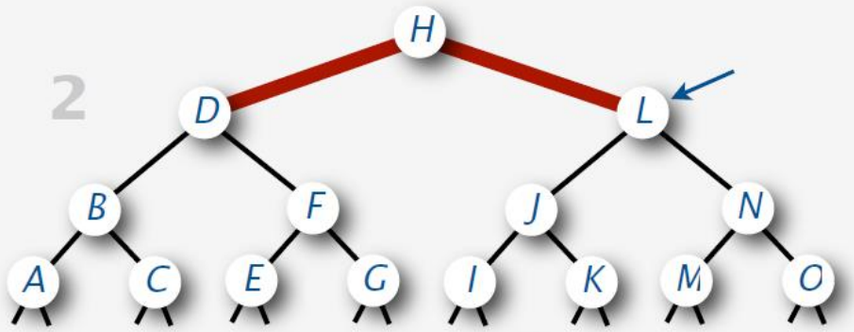
deleteMax() example

push reds down

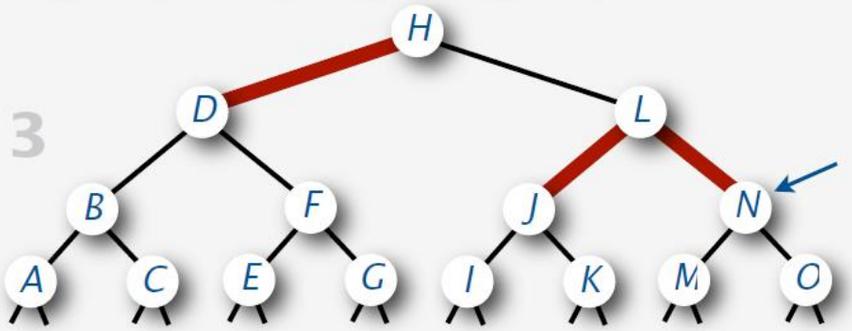
1



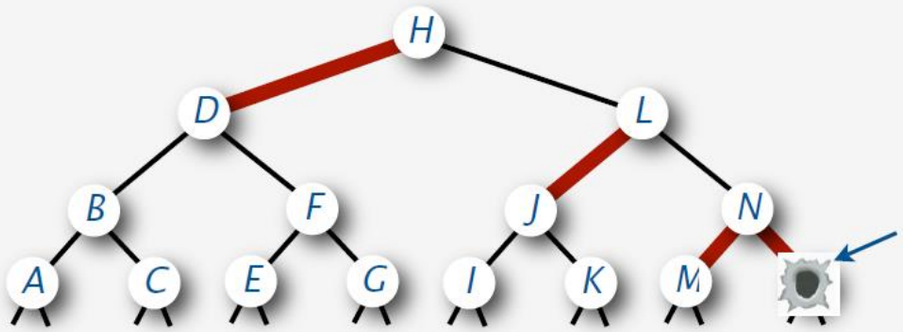
2



3

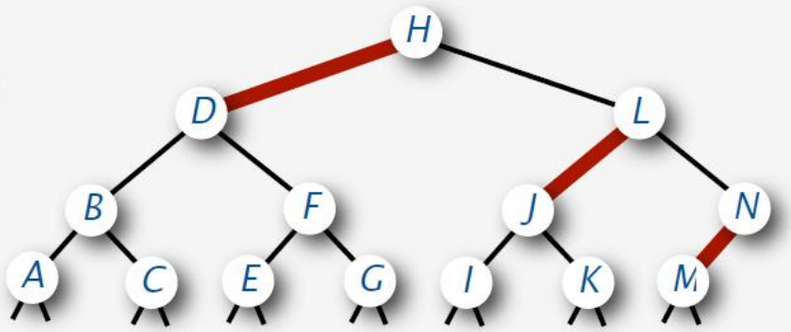


4



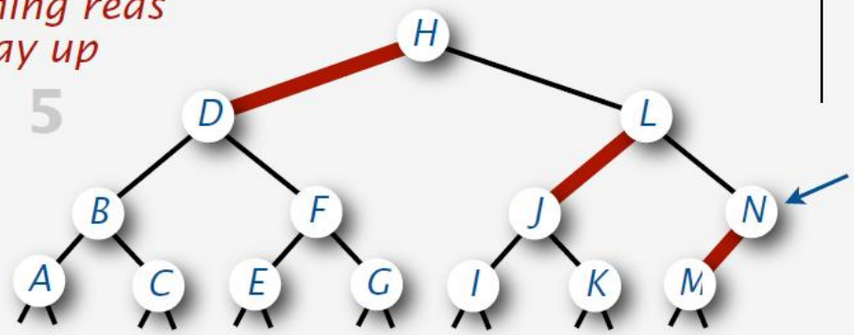
remove maximum

5

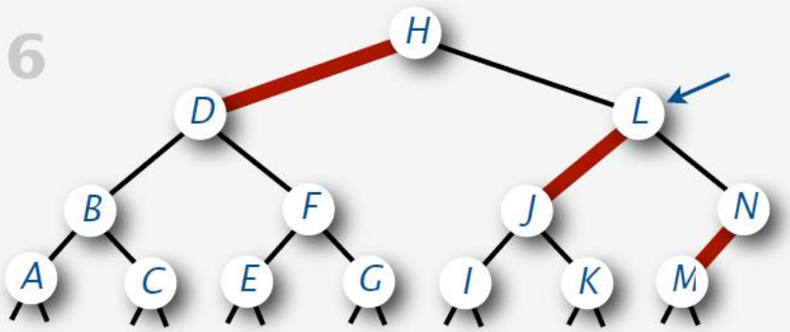


fix right-leaning reds on the way up

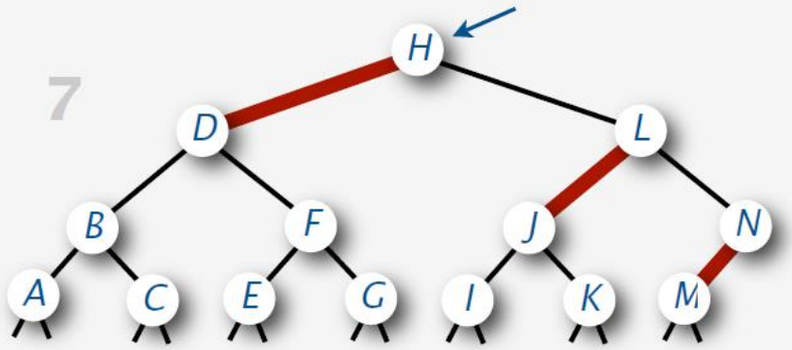
5



6



7

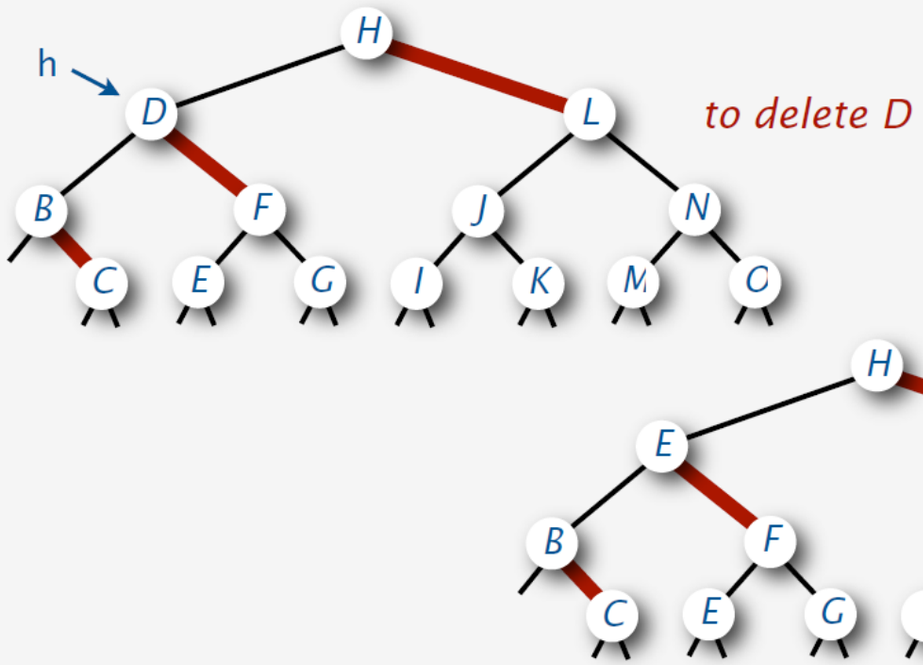


(nothing to fix!)

Deleting an arbitrary node

reduces to `deleteMin()`

A standard trick:

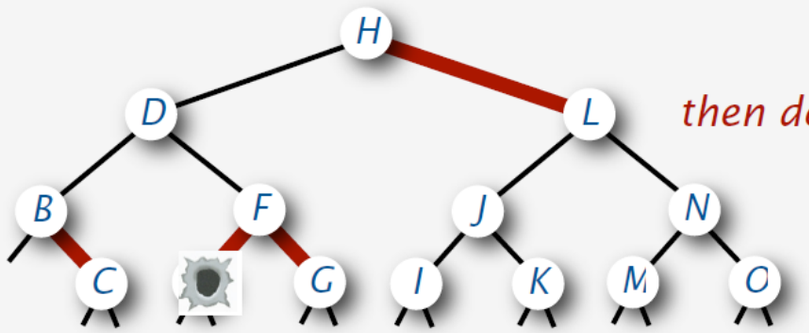


to delete D

```
h.key   = min(h.right);
h.value = get(h.right, h.key);
h.right = deleteMin(h.right);
```


replace its key, value with those of its successor

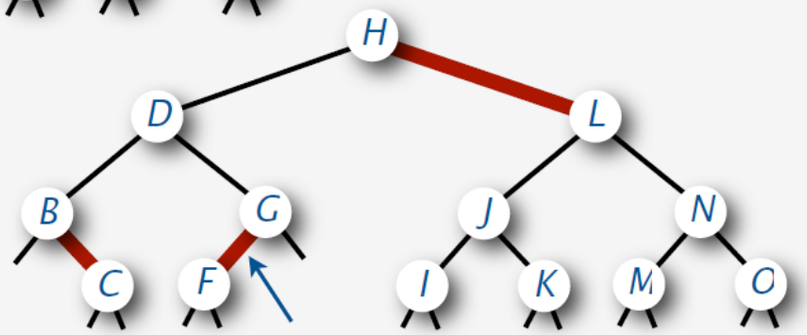
deleteMin(right child of D)



then delete the successor

flip colors, delete node

fix right-leaning red link



Need MORE Demos?

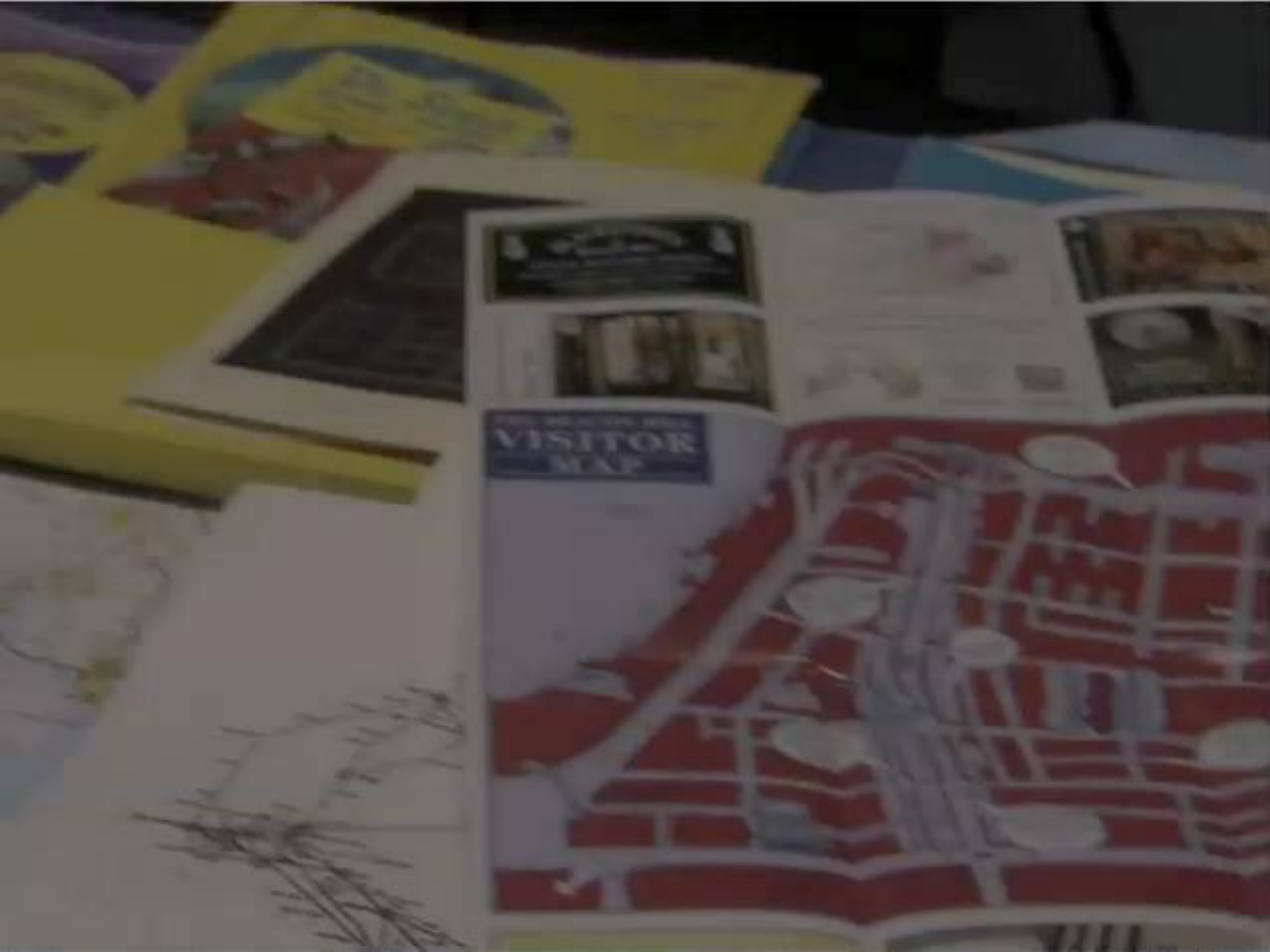
- http://inst.eecs.berkeley.edu/~cs61b/fa17/materials/demos/ll-red-black-demo.html

Red-Black trees in the wild

Red-black trees are widely used as system symbol tables.

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.
- Emacs: conservative stack scanning.

Common sense. Sixth sense.
Together they're the
FBI's newest team.


VISITOR
MAP

Red-black BSTs in the wild

ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

B-trees (Bayer-McCreight, 1972)

B-Tree. Generalizes 2-3-4 trees by allowing up to M links per node.

Main application: file systems.

- Reading a page into memory from disk is expensive.
- Accessing info on a page in memory is free.
- Goal: minimize # page accesses.
- Node size M = page size.

Space-time tradeoff.

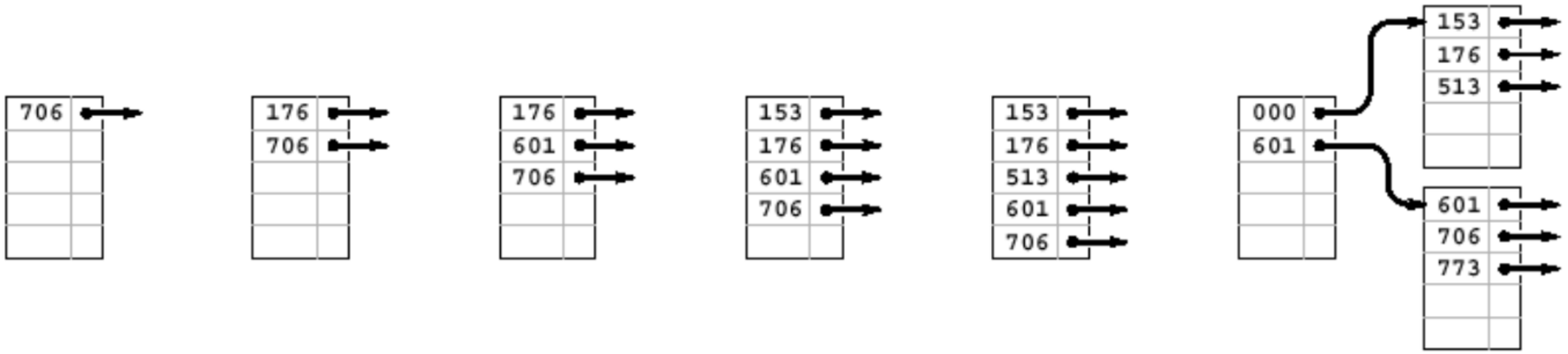
- M large ⇒ only a few levels in tree.
- M small ⇒ less wasted space.
- Typical M = 1000, N < 1 trillion.

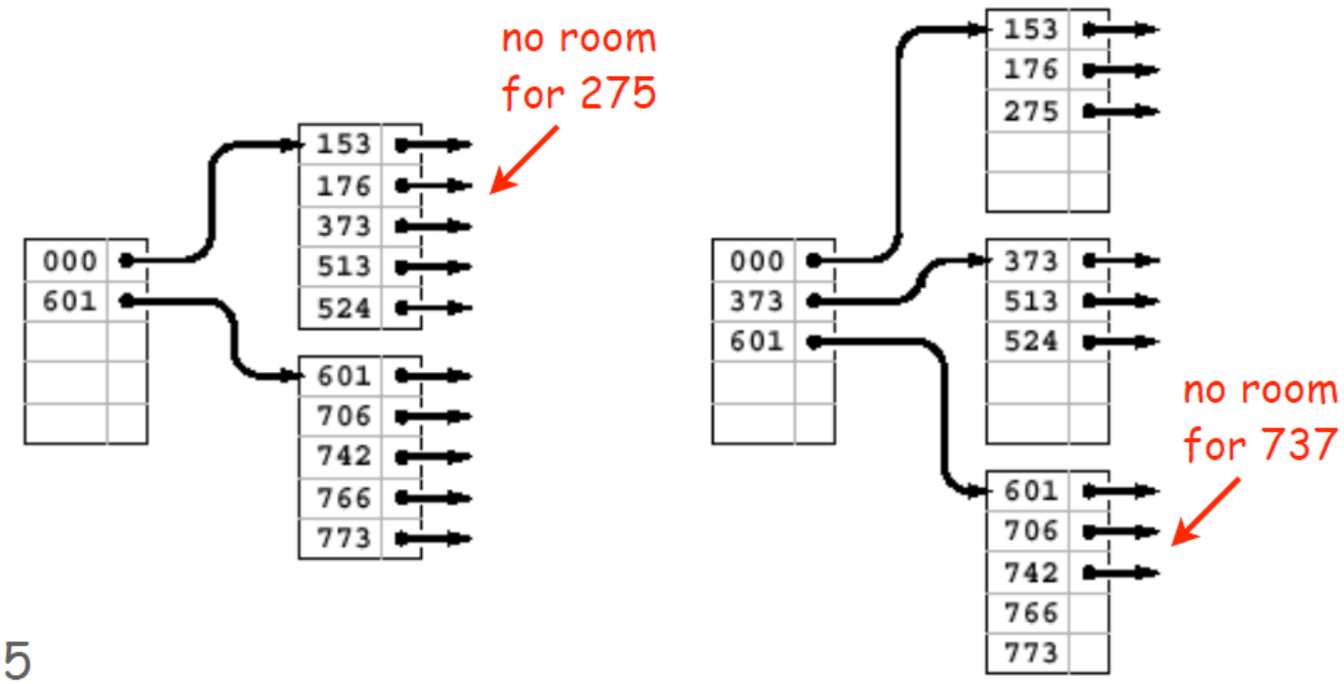
Bottom line. Number of page accesses is $\log_M N$ per op.

in practice: 3 or 4 (!)

https://www.cs.princeton.edu/~rs/AlgsDS07/09BalancedTrees.pdf

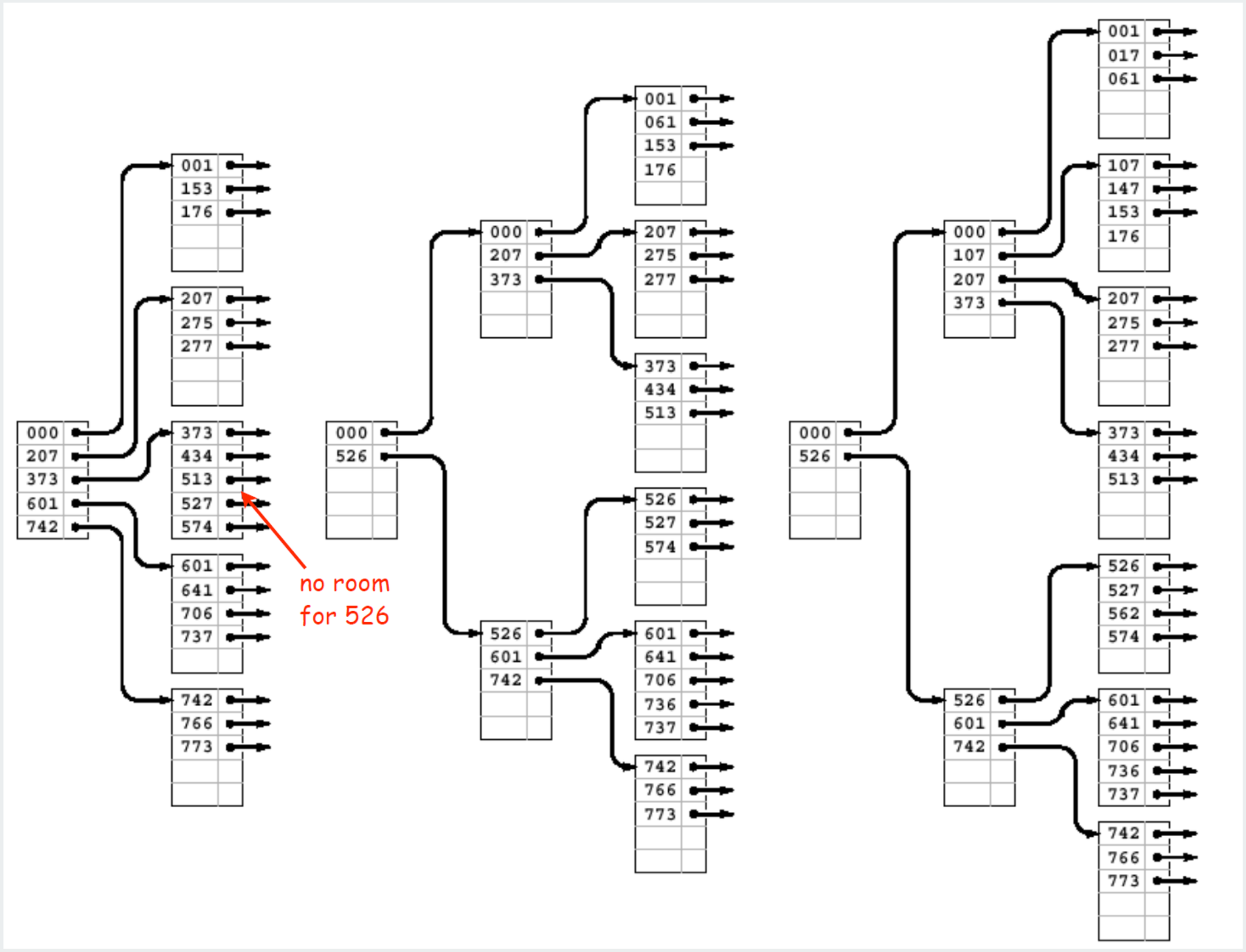
B-Tree Example





M = 5

B-Tree Example (cont)



Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.44 lg N	1.44 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.44 lg N	1.44 lg N	1.44 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes
B-tree	1	1	1	1	1	1	yes

B-Tree. Number of page accesses is $\log_M N$ per op.

Balanced trees in the wild

Red-black trees: widely used as system symbol tables

- Java: `java.util.TreeMap, java.util.TreeSet.`
- C++ STL: map, multimap, multiset.
- Linux kernel: `linux/rbtree.h.`

B-Trees: widely used for file systems and databases

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL

Bottom line: ST implementation with lg N guarantee for all ops.

- Algorithms are variations on a theme: rotations when inserting.
- Easiest to implement, optimal, fastest in practice: LLRB trees
- Abstraction extends to give search algorithms for huge files: B-trees