Lecture 9
Balanced Search Trees

Jianchen Shan
Department of Computer Science
Hofstra University

Lecture Goals

- Develop balanced search trees with guaranteed logarithmic performance for search and insert (and many other operations).
- We begin with 2-3 trees, which are easy to analyze but hard to implement.
- Next, we consider red-black binary search trees, which we view as a novel way to implement 2-3 trees as binary search trees.
- Finally, we introduce B-trees, a generalization of 2-3 trees that are widely used to implement file systems.

2–3 Search Trees

The 2-3 tree is a way to generalize BSTs to provide the flexibility that we need to guarantee fast performance

Allow 1 or 2 keys per node

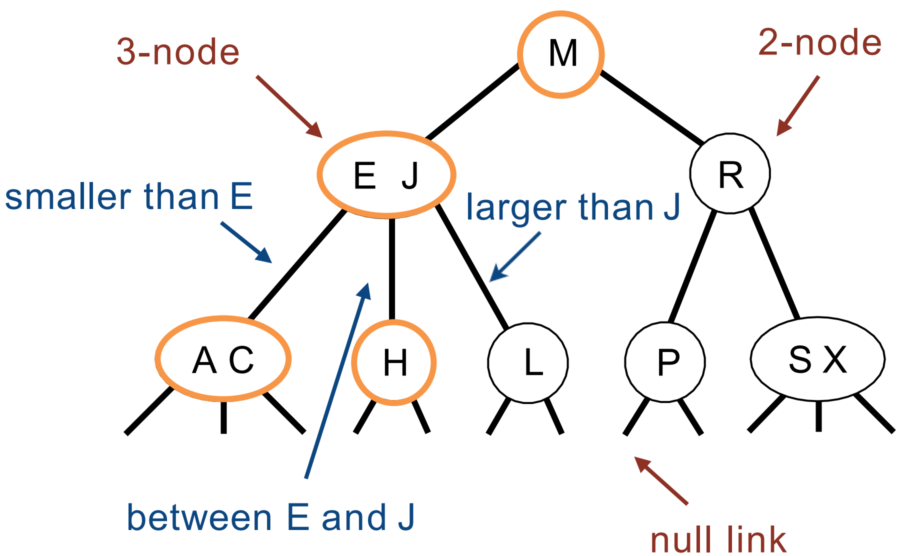
- 2-node: one key, two children.
- 3-node: two keys, three children.

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

how to maintain?

Symmetric order: Inorder traversal yields keys in ascending order.

So the search is a generalization of the search in BSTs



Search

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

Search for H

H is less than M (go left)

H is between E and J (go middle)

Search hit

Search for B

B is less than M (go left)

B is less than E (go left)

B is between A and C (go middle)

Link is null

Search miss

Insert in 2–3 Search Trees

Insertion into a 2-node at bottom.

- Search for key, as usual
- Add new key to 2-node to create a 3-node.

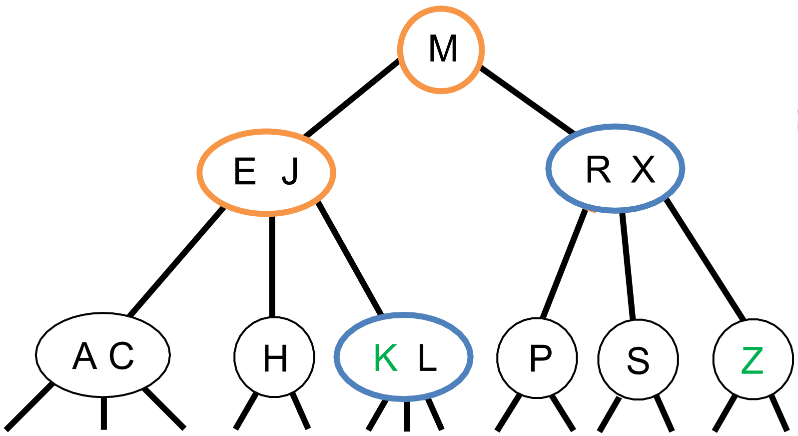
Insert K

K is less than M (go left)

K is larger than J (go right)

K is less than L

Search ends here and replace 2-node with 3-node containing K



Insert Z

Z is larger than M (go right)

Z is larger than R (go right)

Z is larger than X

Search ends here

Replace 3-node with temporary 4-node containing Z

Split 4-node into two 2-nodes (pass middle key to parent)

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.

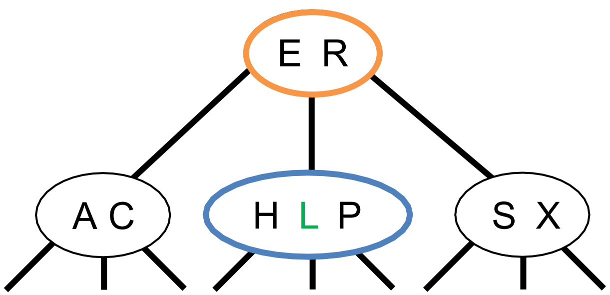
Insert in 2–3 Search Trees (Contd.)

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 L`

L is between E and R (go middle)

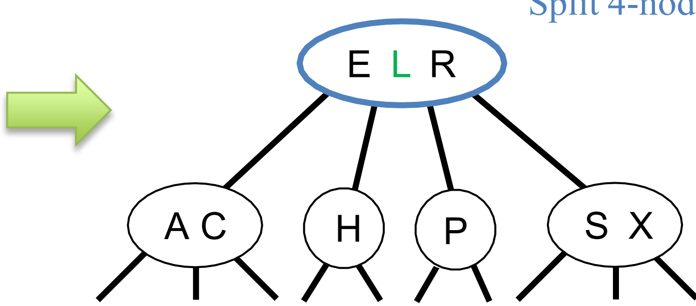


Search ends here

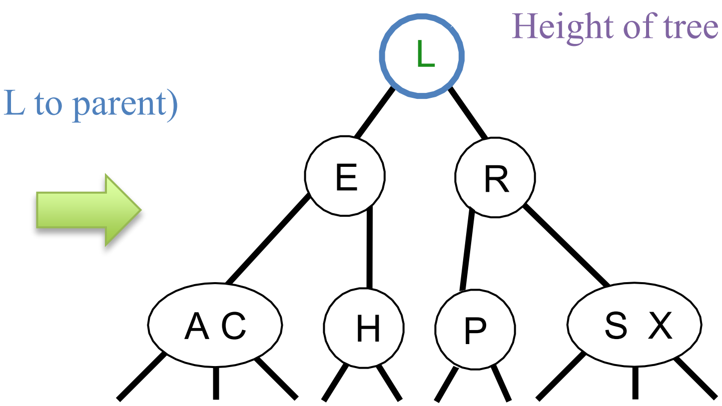
Replace 3-node with temporary 4-node containing L

Split 4-node (move L to parent)

Split 4-node (move L to parent)



Height of tree increases by 1



2–3 Search Trees Construction

`Insert S` Create 2-node in the empty tree

`Insert E` Convert 2-node into 3-node

`Insert A` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move E to parent)

`Insert R` Convert 2-node into 3-node

`Insert C` Convert 2-node into 3-node

`Insert H` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move R to parent)

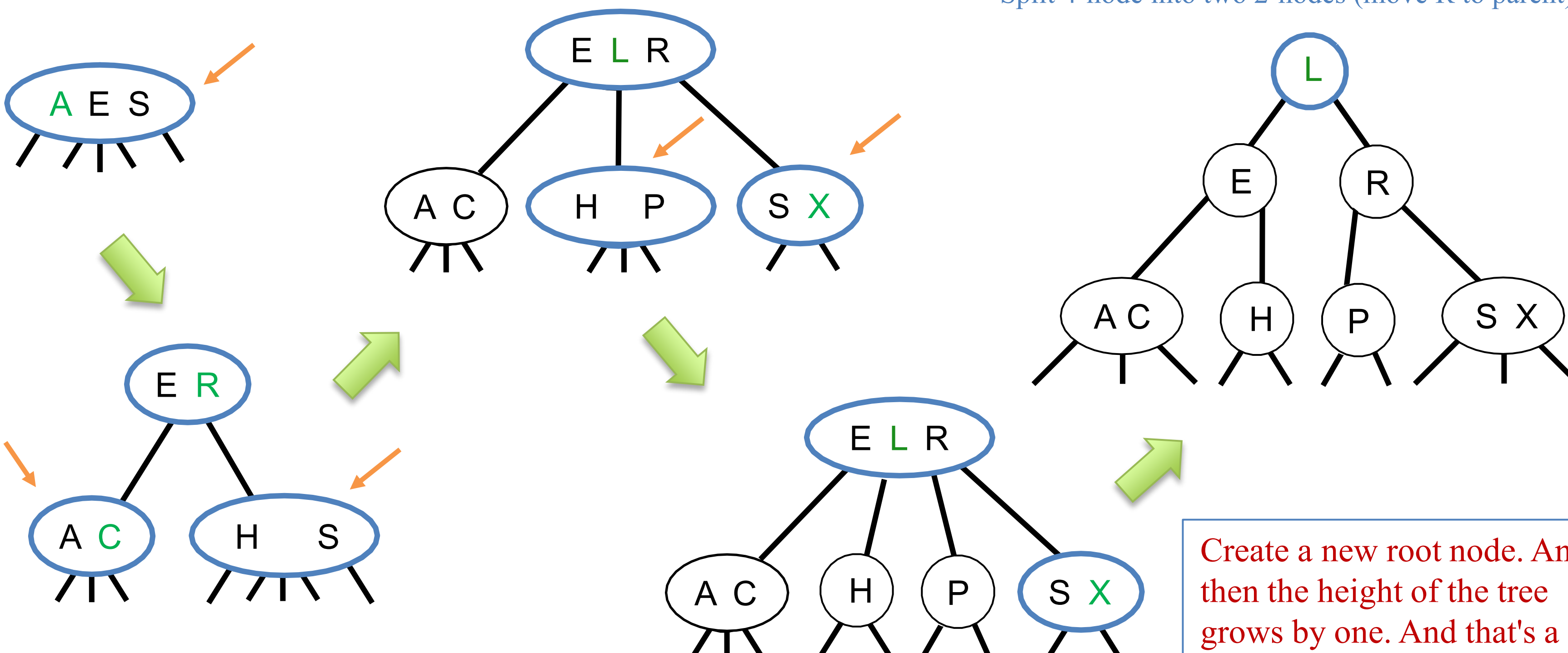
`Insert X` Convert 2-node into 3-node

`Insert P` Convert 2-node into 3-node

`Insert L` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move R to parent)

Split 4-node into two 2-nodes (move R to parent)

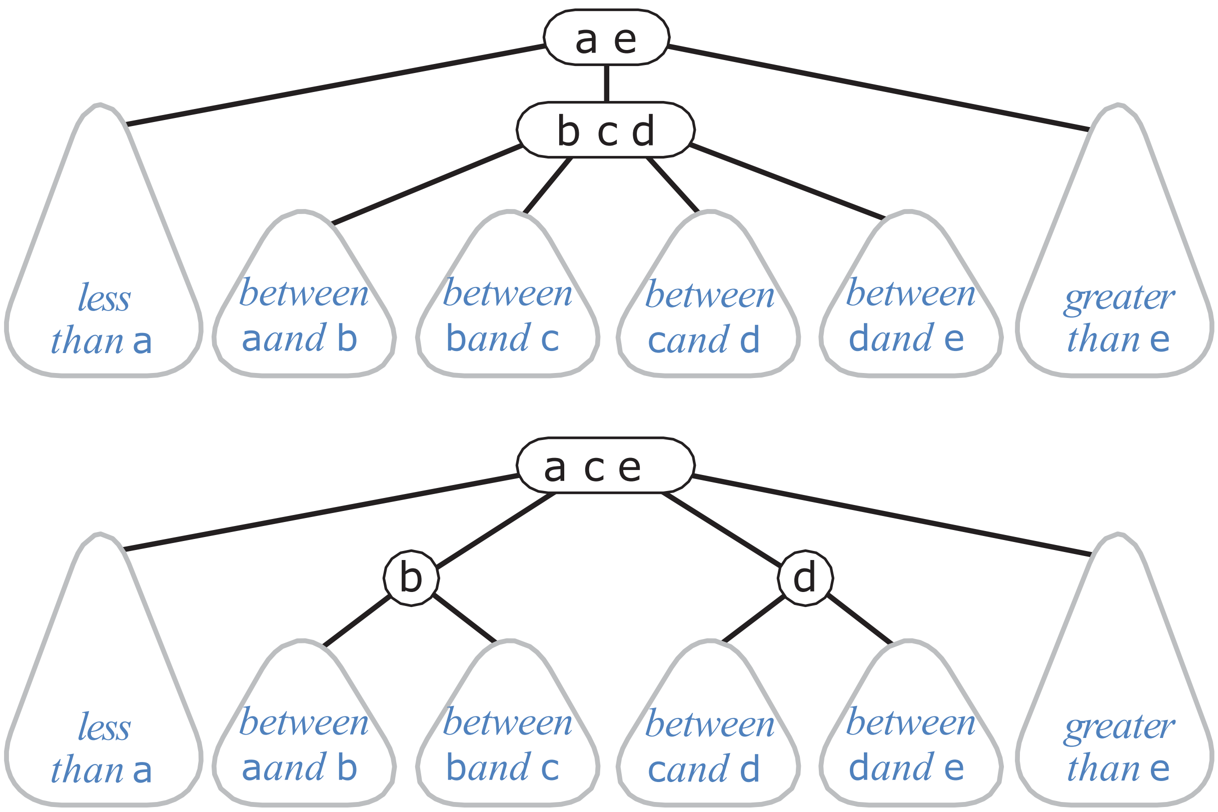


Create a new root node. And then the height of the tree grows by one. And that's a legal 2-3 tree, so we stop.

Local Transformations in a 2-3 Tree

Converting a 2-node to a 3-node

Converting a three to a four, and then splitting and passing a node up



only operations we need to consider to get balance.

local transformation: constant number of operations.

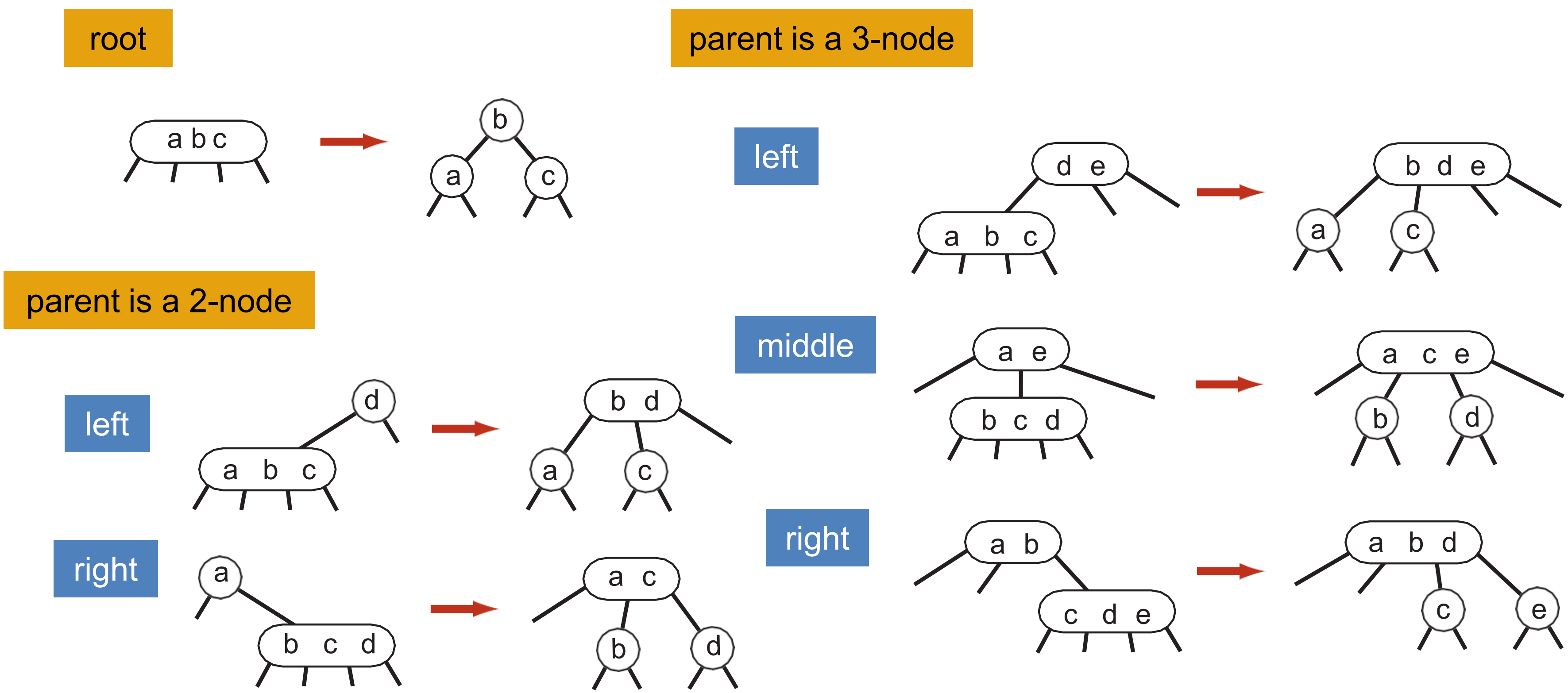
Only involves changing a constant number of links, and is independent of the tree size.

Global Properties in a 2-3 Tree

Invariants. Maintains symmetric order and perfect balance.

Proof. Each transformation maintains symmetric order and perfect balance.

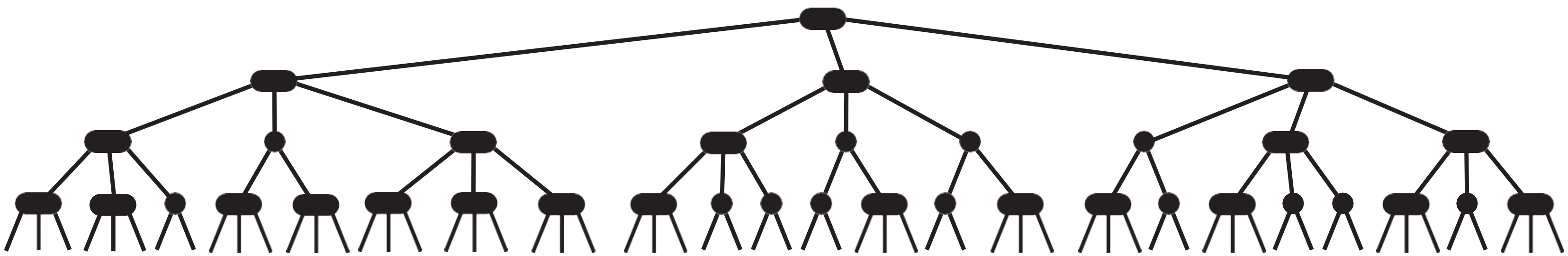
Splitting a 4-node



The height doesn’t change. If it is perfect balance before, it is perfect balance afterwards

Performance of 2-3 Tree

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



So, that's going to give us, a very easy way to describe a performance.

Operations have costs that's proportional to the path link from root to the bottom.

Tree height

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

	Best case	Average case	Worst case
BST	O(1)	O(log n)	O(n)
2-3 Tree	O(1)	O(log n)	O(log n)

2-3 tree model guaranteed logarithmic performance for search and insert.

Lecture Goals

- Develop balanced search trees with guaranteed logarithmic performance for search and insert (and many other operations).
- We begin with 2-3 trees, which are easy to analyze but hard to implement.
- Next, we consider red-black binary search trees, which we view as a novel way to implement 2-3 trees as binary search trees.
- Finally, we introduce B-trees, a generalization of 2-3 trees that are widely used to implement file systems.

Red-Black BSTs

Motivation

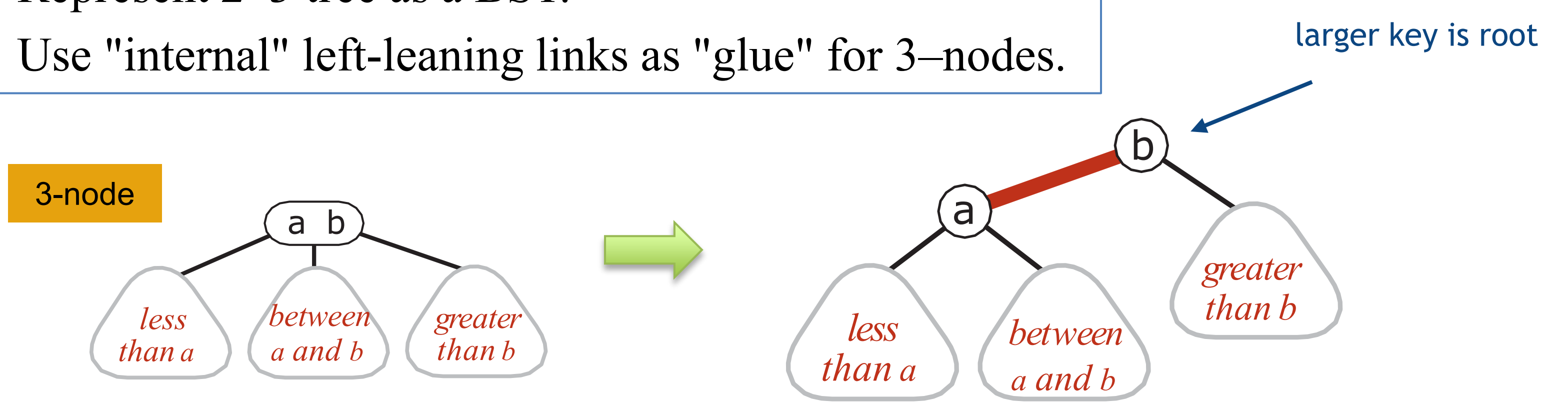
Direct implementation of 2-3 trees can be done but it is complicated, because:

- 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.

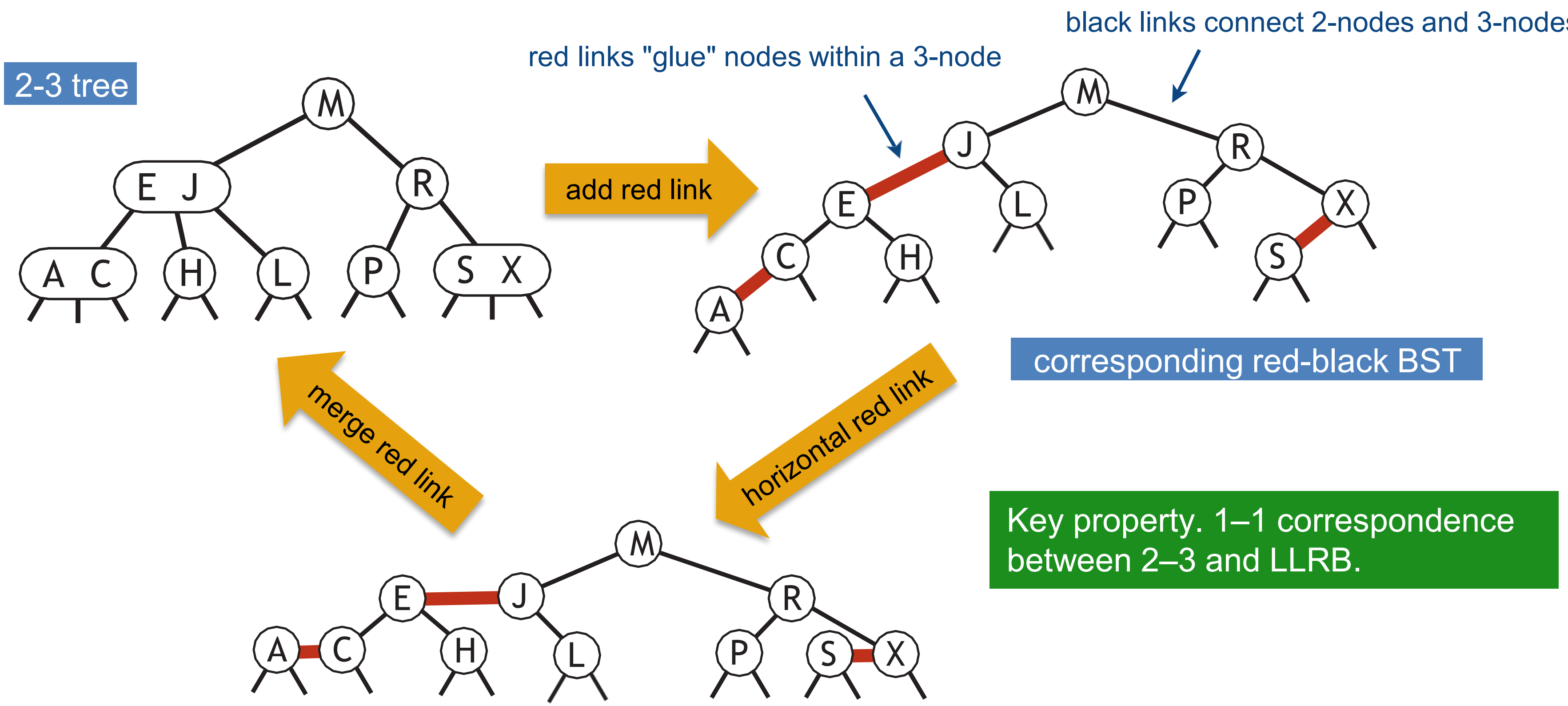
Red-Black BSTs (Sedgewick 1979): a simple data structure that allows us to implement 2-3 tree with very little extra code beyond the basic BST code.

Left-leaning red-black BSTs (Sedgewick 2007) : the simplest to implement and at least as efficient

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



Left-Leaning Red-Black BSTs



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

Because the only red links are internal to three nodes.

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.

follows directly from the corresponding property for 2-3 trees

"perfect black balance"

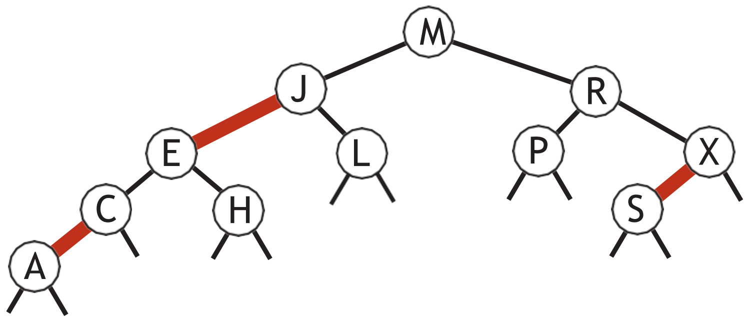
Search Implementation for Red-Black BSTs

Observation. Search is the same as for elementary BST *(ignore color).*

but runs faster because of better balance

```
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if(cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

We don't have to change the code at all. Our regular search code doesn't examine the color of a link and so we can just use it exactly as is.



Remark. Most other ops (e.g., iteration) are also identical.

We just make sure that we maintain the balance properties during the insertion

Red-Black BST Representation

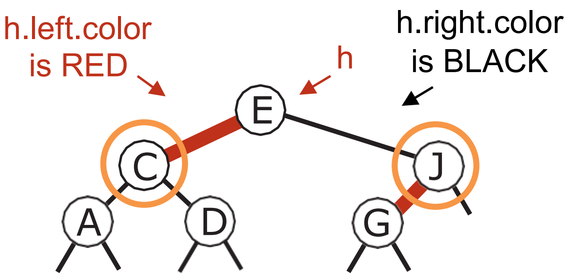
Each node is pointed to by precisely one link *(from its parent)*
→ can encode color of links in nodes.

```
private  static  final  boolean  RED     = true;
private  static  final  boolean  BLACK   = false;

private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean color;         // color of parent link
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

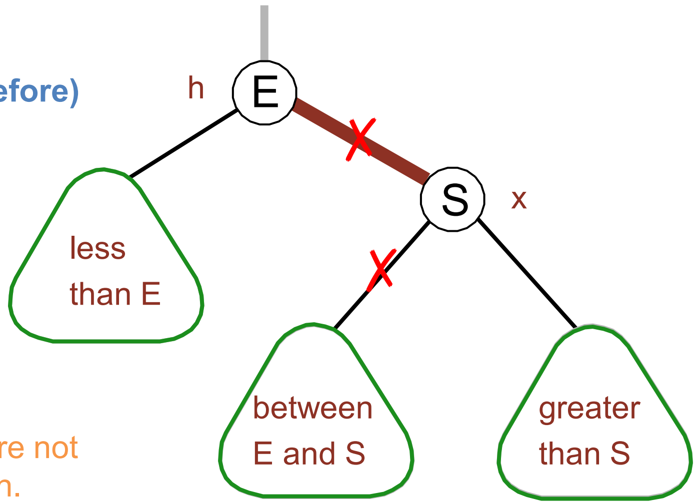
null links are black



Elementary Red-Black BST Operations

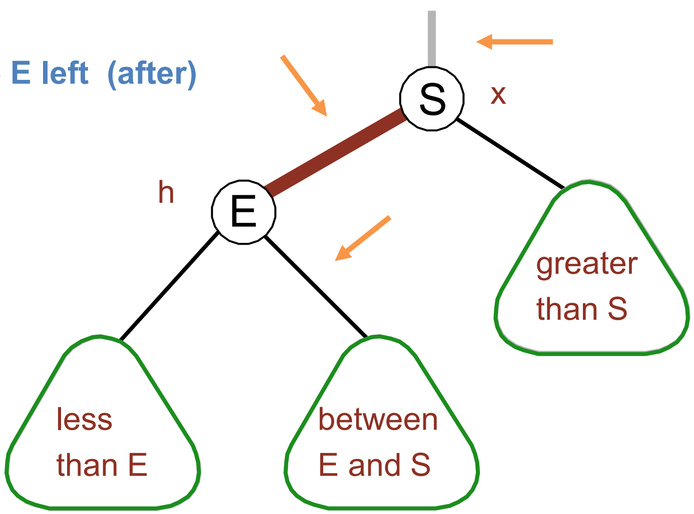
Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left (before)



Their black heights are not changed after rotation.

rotate E left (after)




```
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;      ← new root to connect
                     the node above
}
```

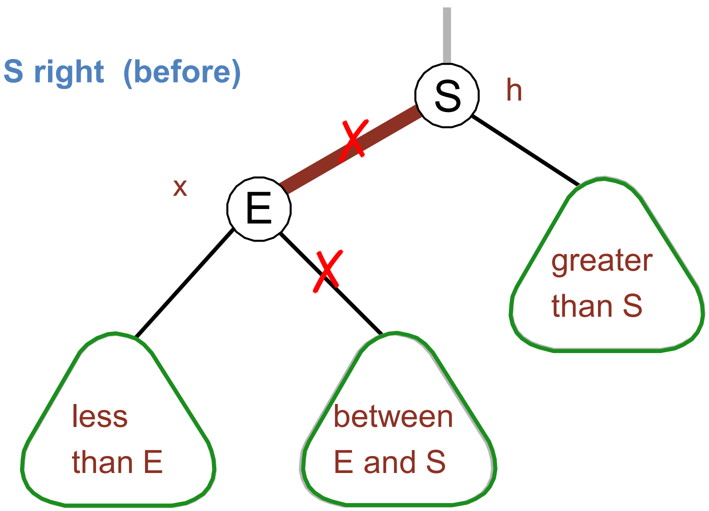
local operation that only changes a few links

Invariants. Maintains symmetric order and perfect black balance.

Elementary Red-Black BST Operations (Contd.)

Right rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate S right (before)

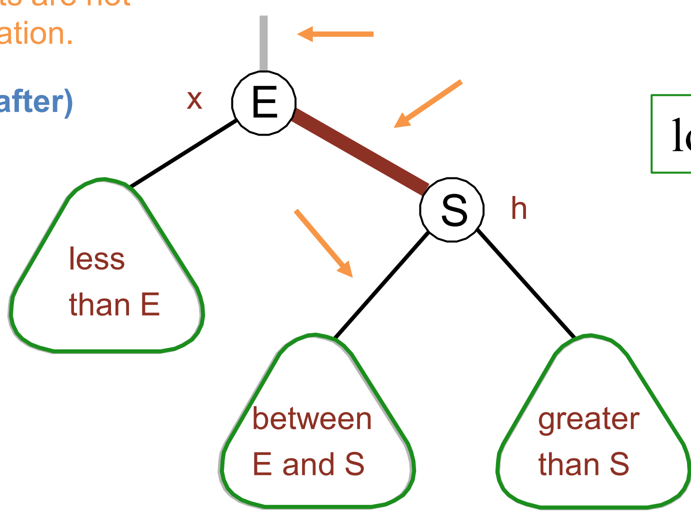


symmetric code of left rotation

```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;    ← new root to connect the node above
}
```

Their black heights are not changed after rotation.

rotate S right (after)



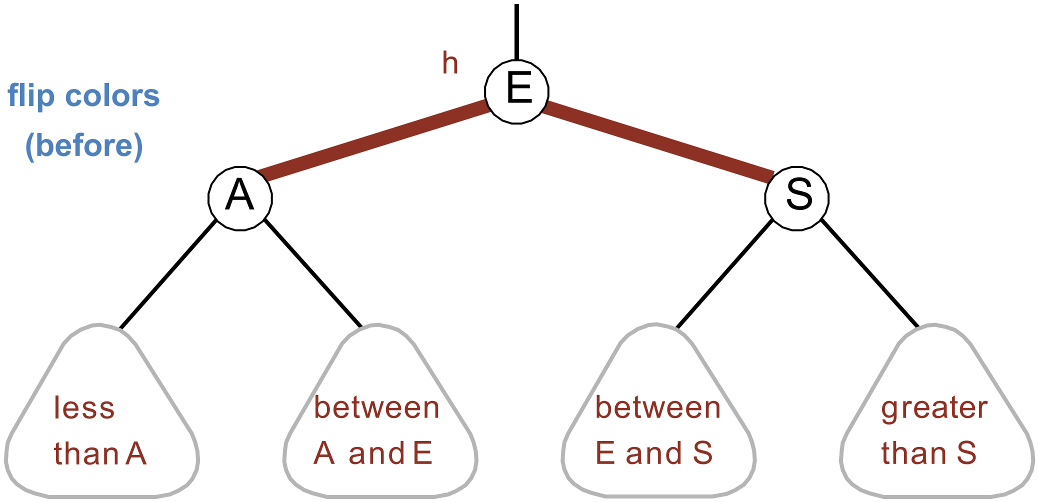
local operation that only changes a few links

Invariants. Maintains symmetric order and perfect black balance.

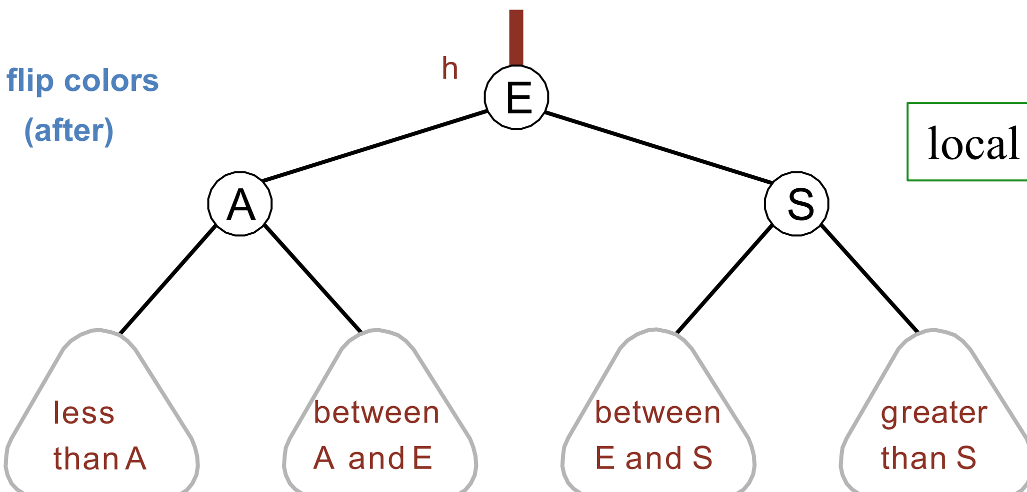
Elementary Red-Black BST Operations (Contd.)

Color flip. Recolor to split a (temporary) 4-node and pass up the center node.

flip colors (before)



flip colors (after)



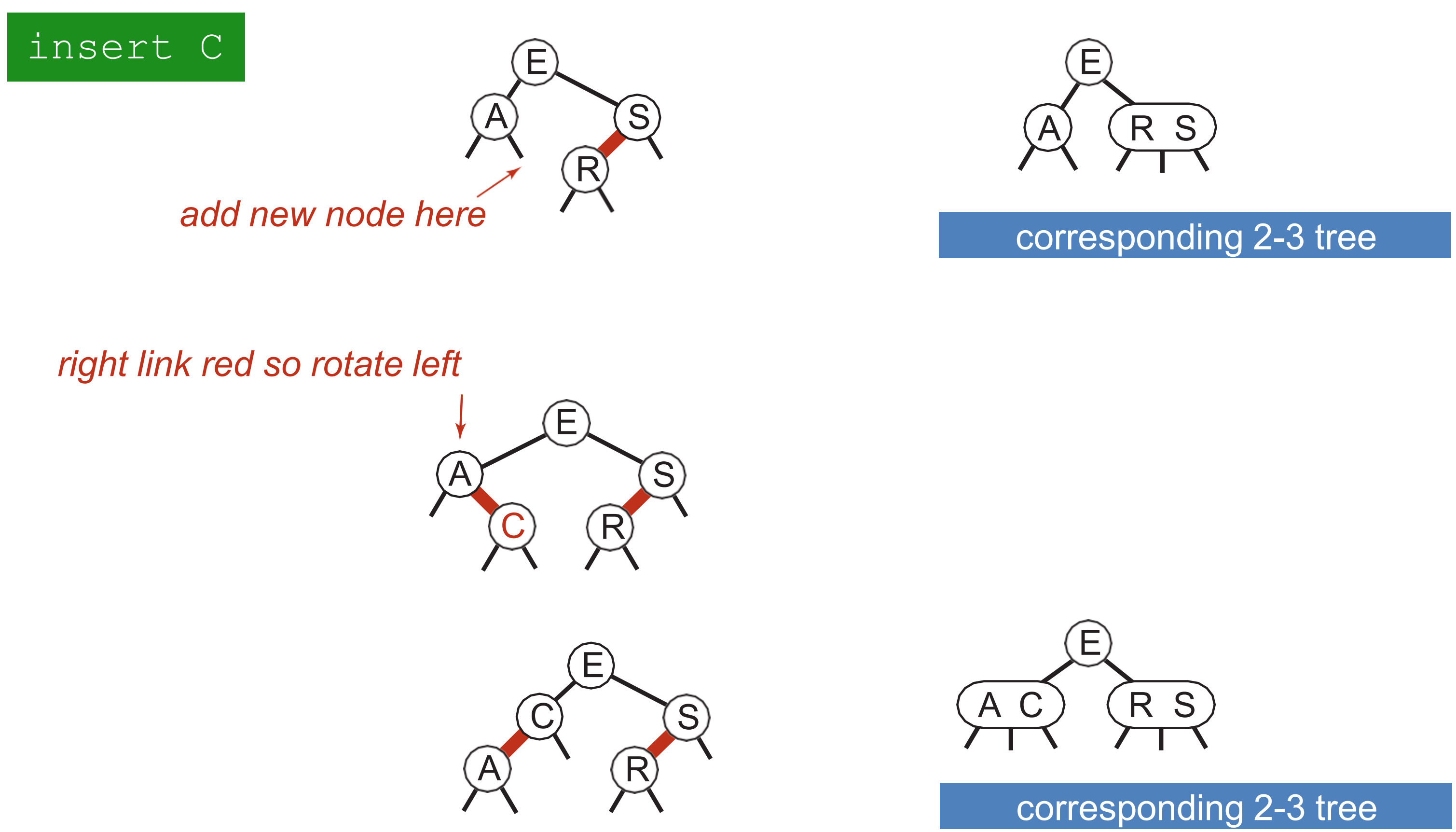

```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

local operation that only changes a few links

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB Tree: Overview

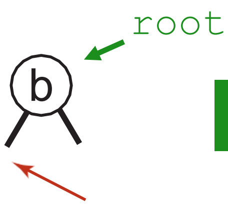
Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.



Insertion in a LLRB Tree

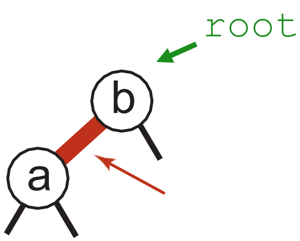
Warmup 1. **Insert into a tree with exactly 1 node.**

left



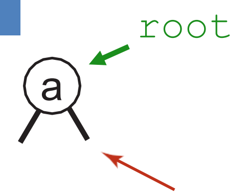
`insert a`

search ends at this null link



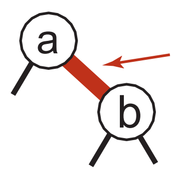
- *red link to new node containing a*
- *converts 2-node to 3-node*

right

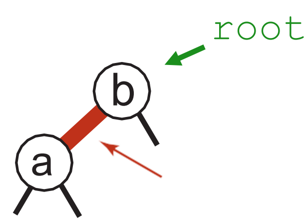


`insert b`

search ends at this null link



attached new node with red link



rotated left to make a legal 3-node

Insertion in a LLRB Tree

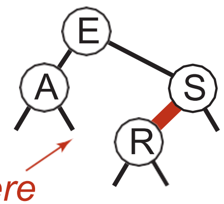
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red
- If new red link is a right link, rotate left.

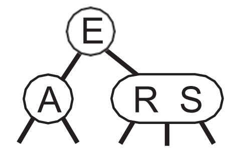
to maintain symmetric order and perfect black balance

to restore color invariants

insert C

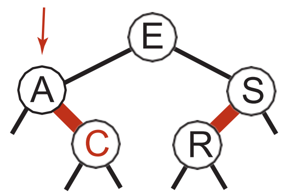


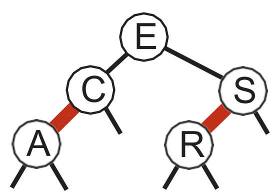
add new node here

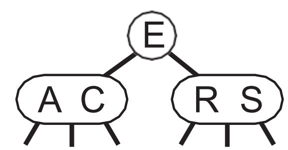


corresponding red-black BST

right link red so rotate left





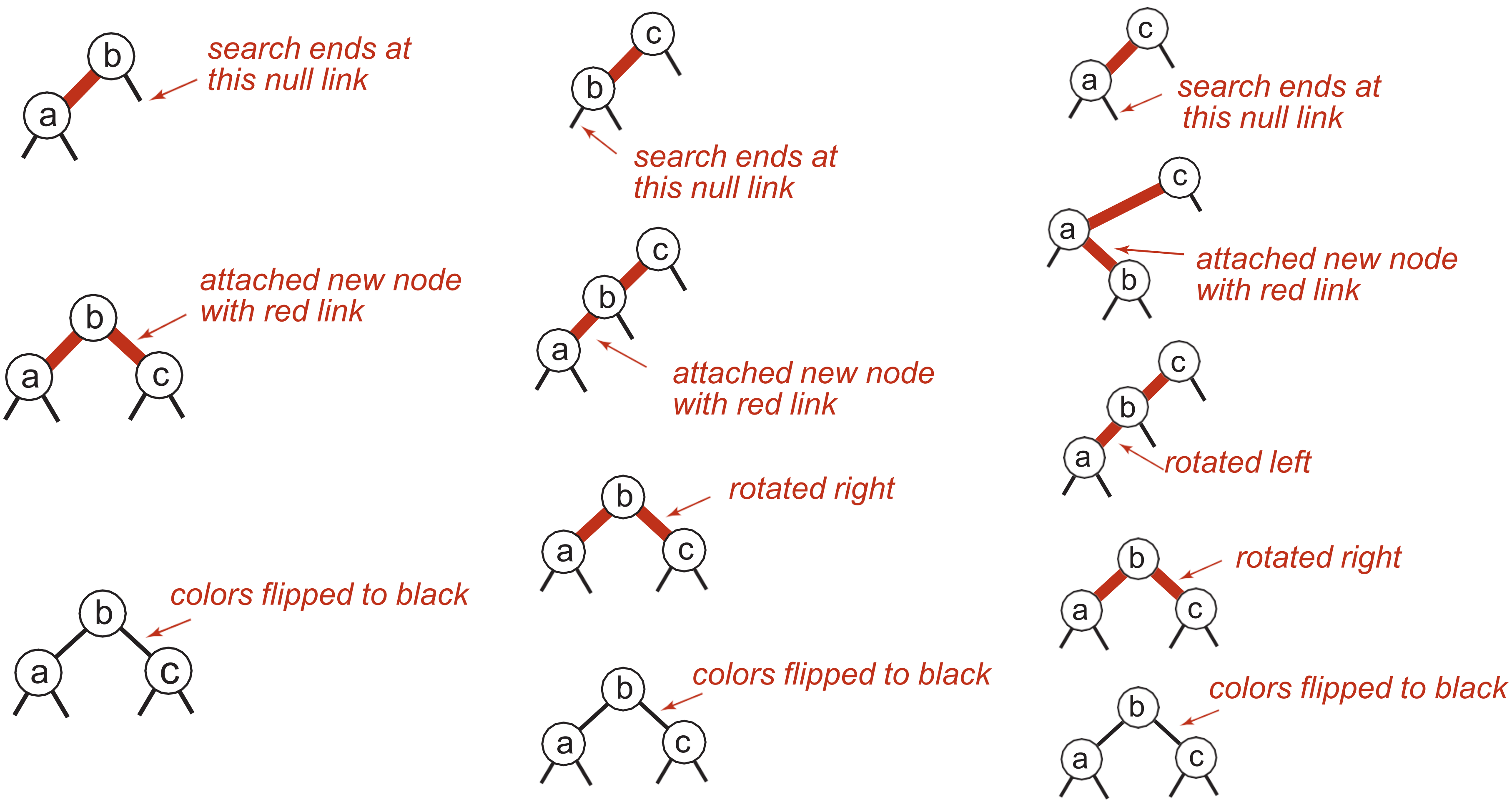


corresponding red-black BST

Insertion in a LLRB Tree

Warmup 2. Insert into a tree with exactly 2 nodes.

larger `insert c` | smaller `insert a` | between `insert b`



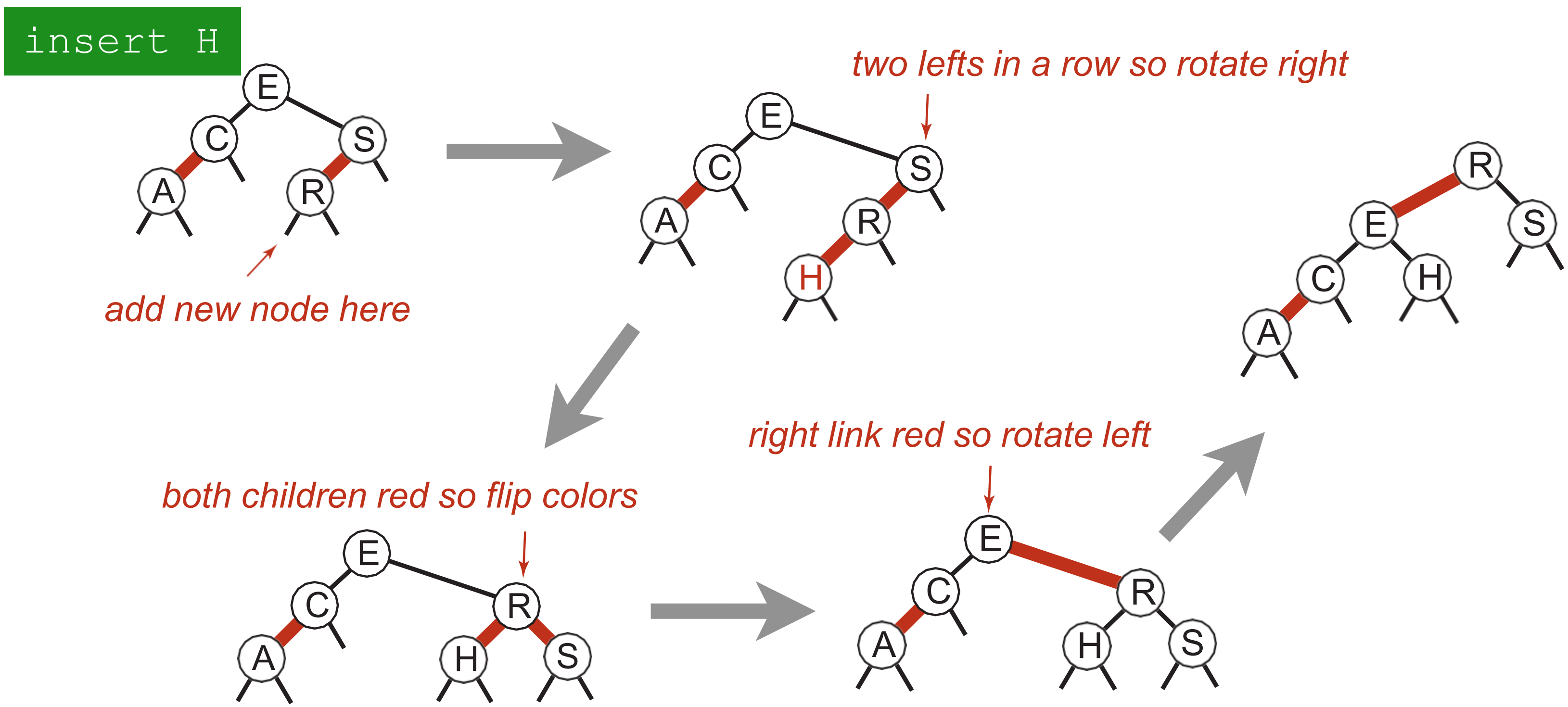
Insertion in a LLRB Tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

to maintain symmetric order and perfect black balance

to restore color invariants

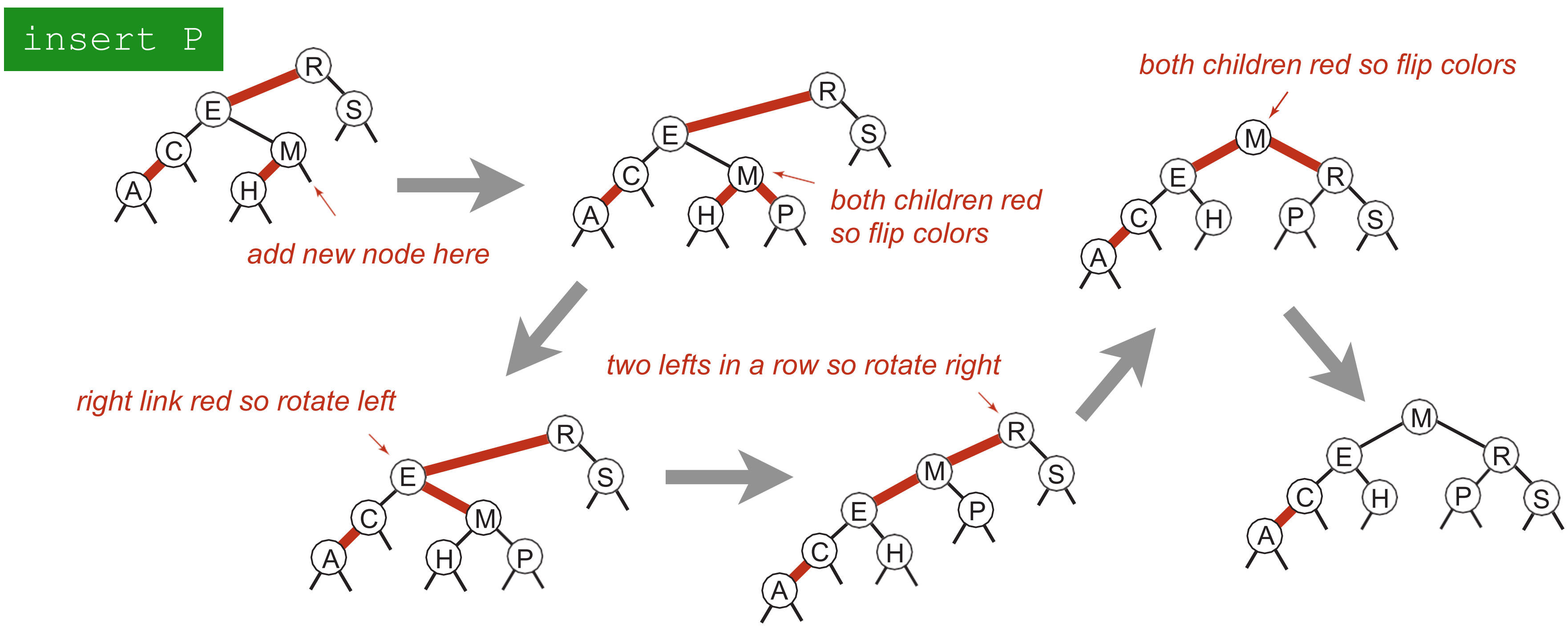


Insertion in a LLRB Tree: Passing Red Links up the Tree

Case 2. Insert into a 3-node at the bottom.

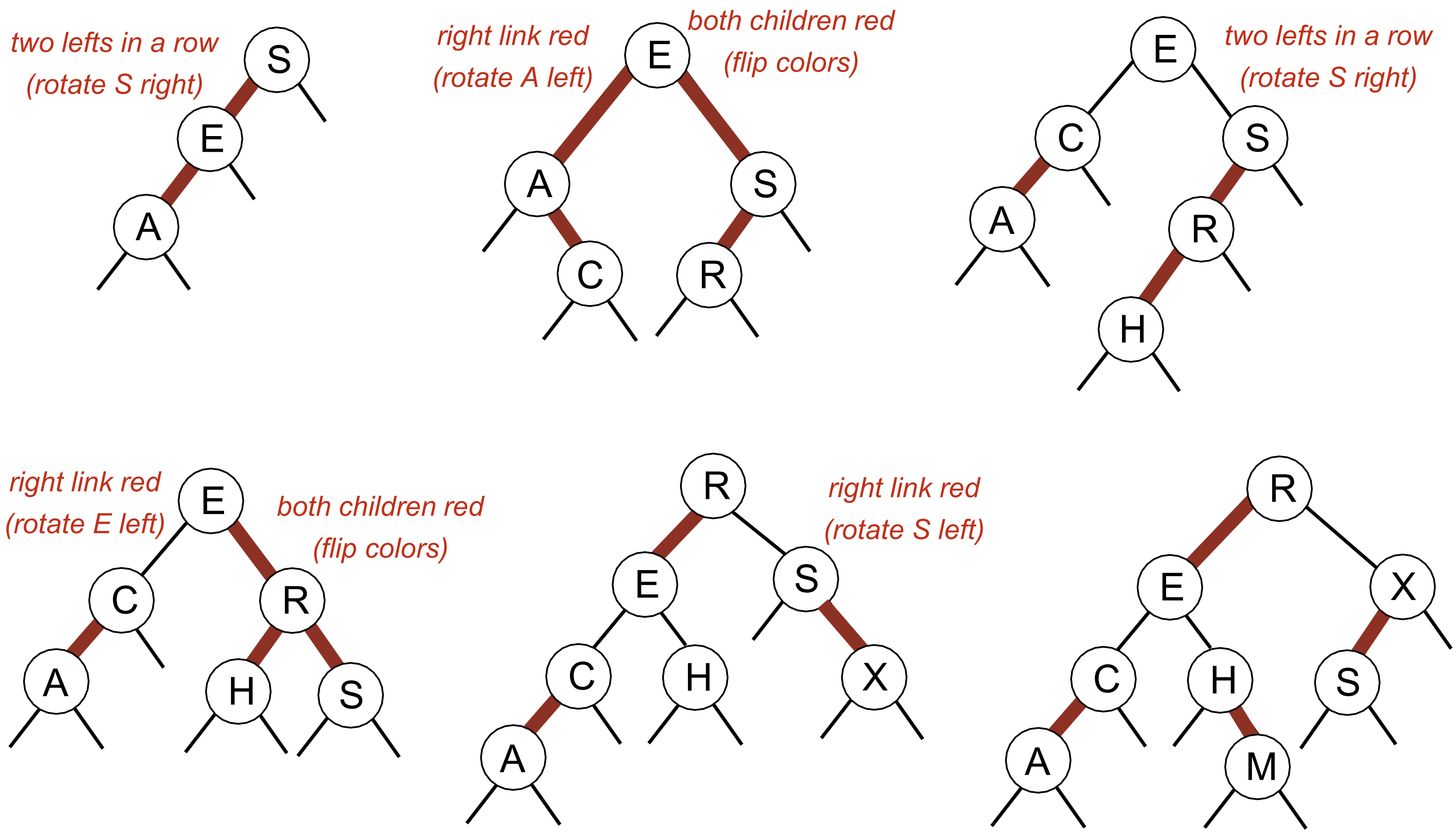
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).

If the red linke is passed up to a 3-node, we treate it the same way we do at the bottom



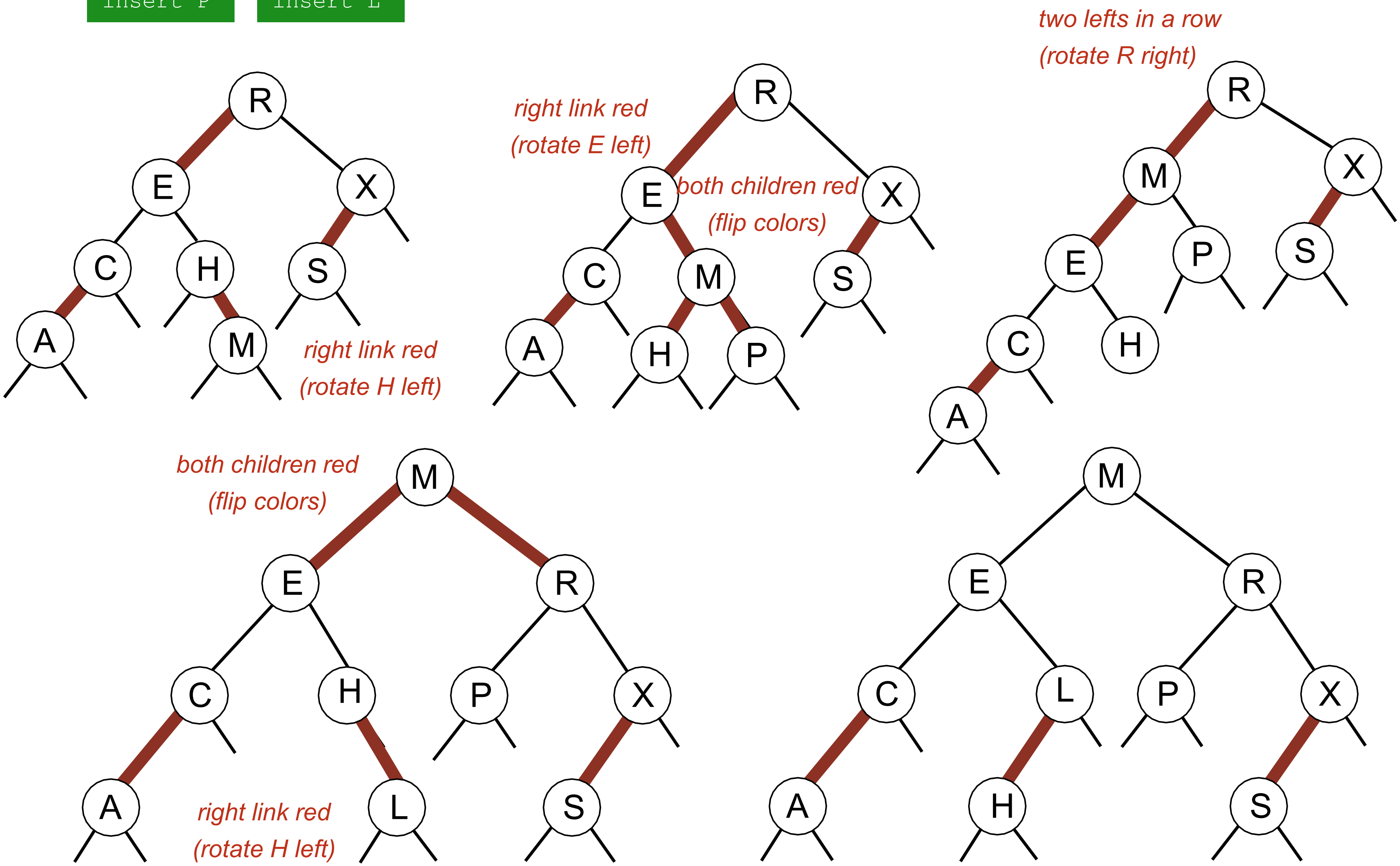
LLRB Tree Construction Demo

insert S insert E insert A insert R insert C insert H insert X insert M



LLRB Tree Construction Demo (Contd.)

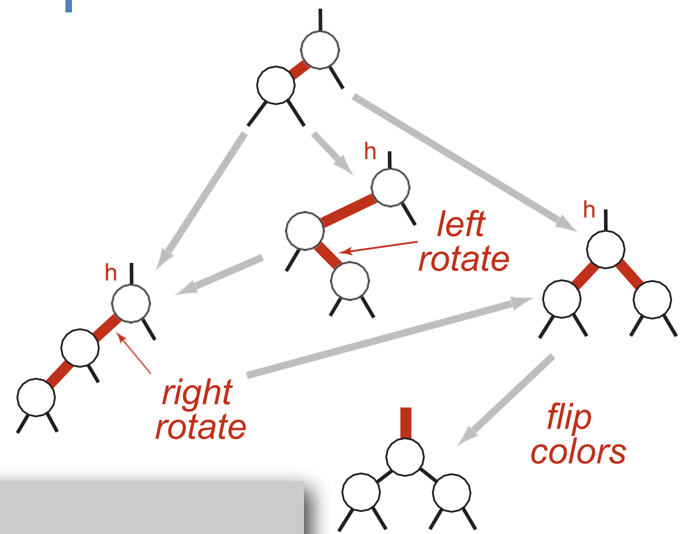
`insert P` `insert L`



Insertion in a LLRB tree: Java implementation

Same code for all cases.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.



The standard BST code for insertion is in grey

```
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)

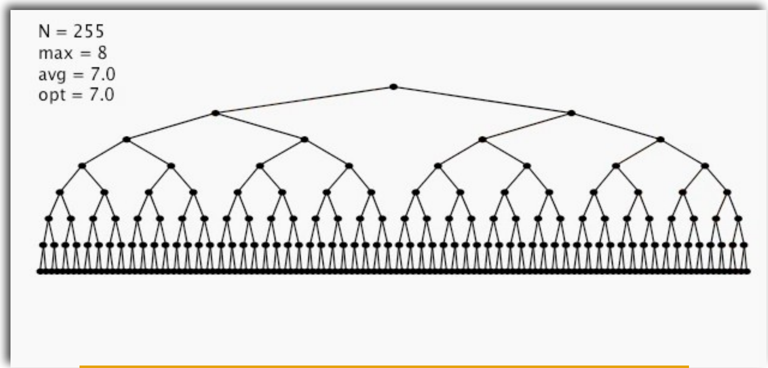
lean left

balance 4-node

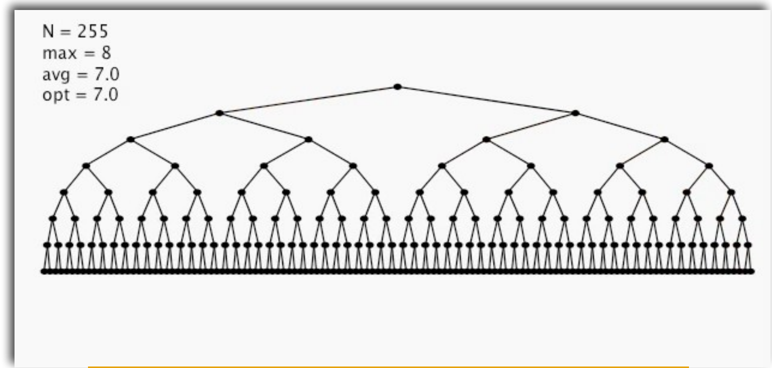
split 4-node

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

Insertion in a LLRB Tree: Visualization

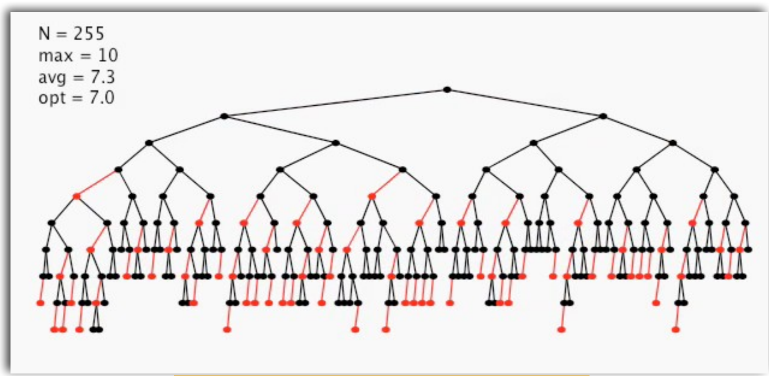


255 insertions in descending order



255 insertions in ascending order

Height of tree is $\leq$ 2 lg N in the worst case.



255 random insertions

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.

Height of tree is ~ 1.00 lg N in typical applications.

- N → number of nodes
- max → max number of nodes in a path
- avg → average length of a path
- opt → minimal length of a path

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	O(N)			O(1.39 lg N)		
2-3 tree	O(c lg N)			O(c lg N)		
LLRB	O(2 lg N)			O(1.00 lg N)		

constants depend upon implementation

exact value of coefficient unknown but extremely close to 1

Additional Resources

- Left-leaning Red–Black Trees
  - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.139.282
- B-tree
  - https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/

Lecture Goals

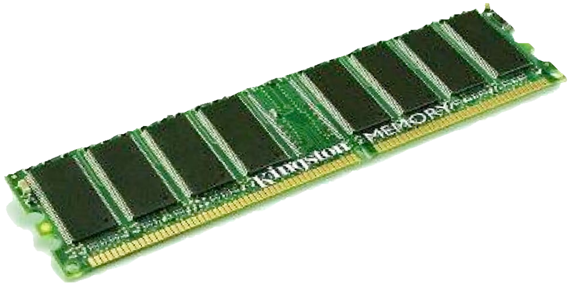
- Develop balanced search trees with guaranteed logarithmic performance for search and insert (and many other operations).
- We begin with 2-3 trees, which are easy to analyze but hard to implement.
- Next, we consider red-black binary search trees, which we view as a novel way to implement 2-3 trees as binary search trees.
- Finally, we introduce B-trees, a generalization of 2-3 trees that are widely used to implement file systems.

Motivation for B-tree: File system model

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.

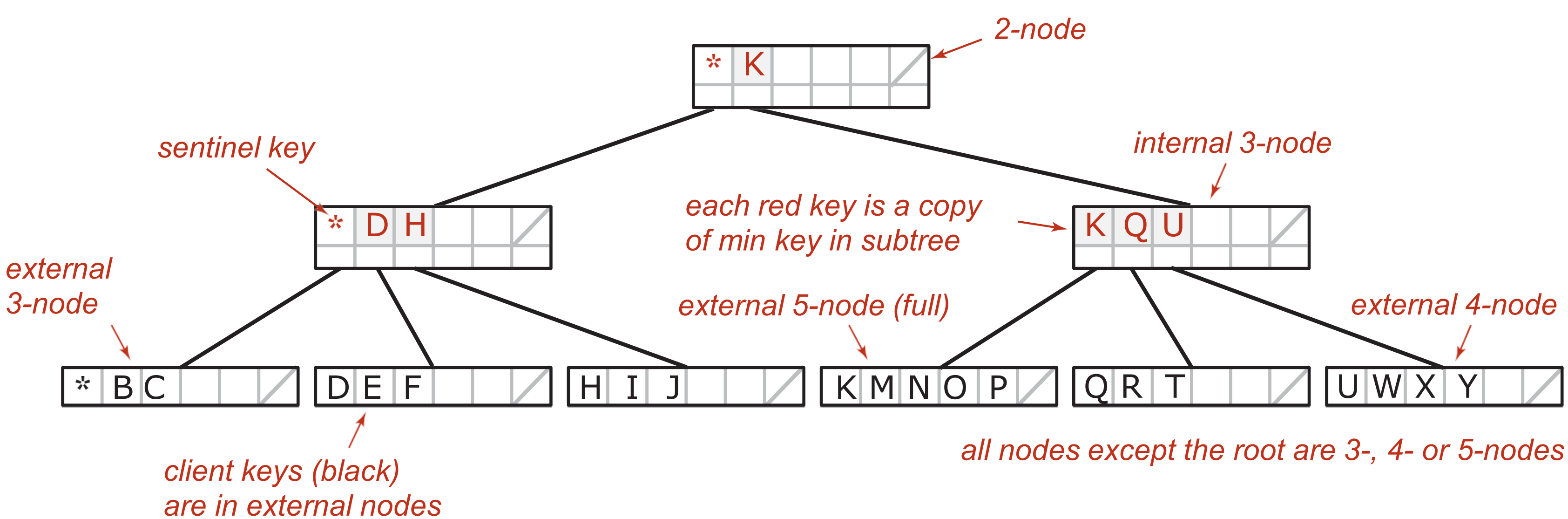
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

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.

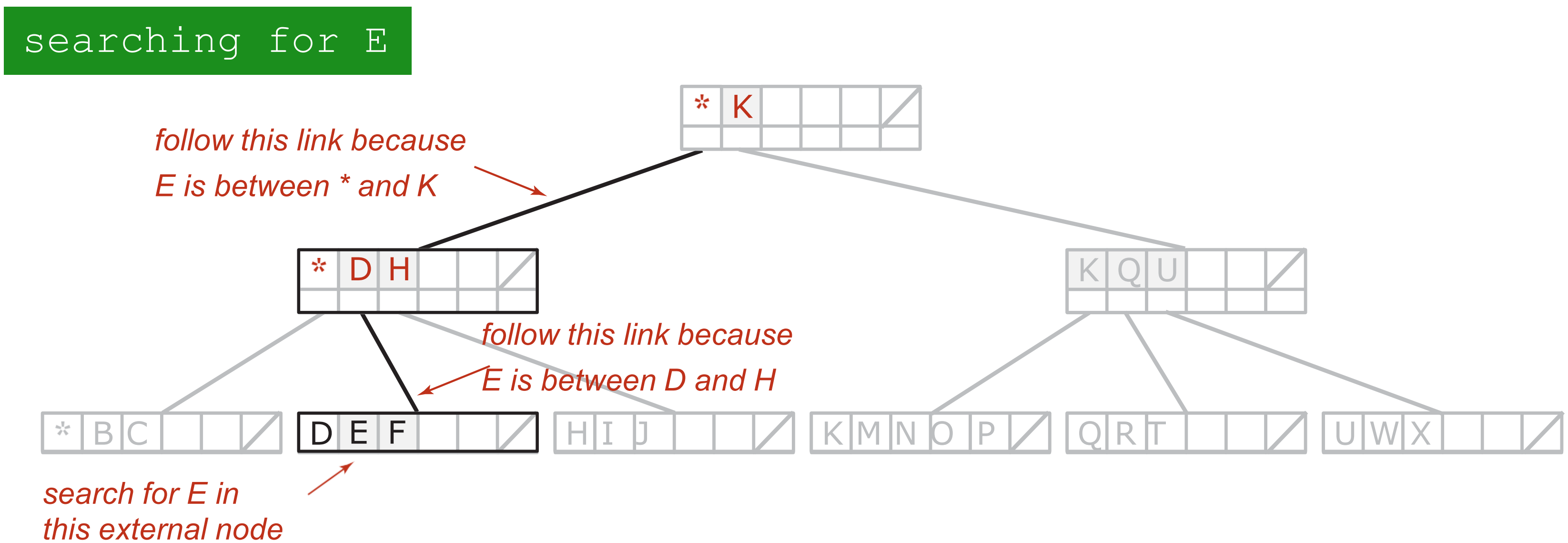
Choose M as large as possible so that M links fit in a page, e.g., M = 1024



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

Searching in a B-tree

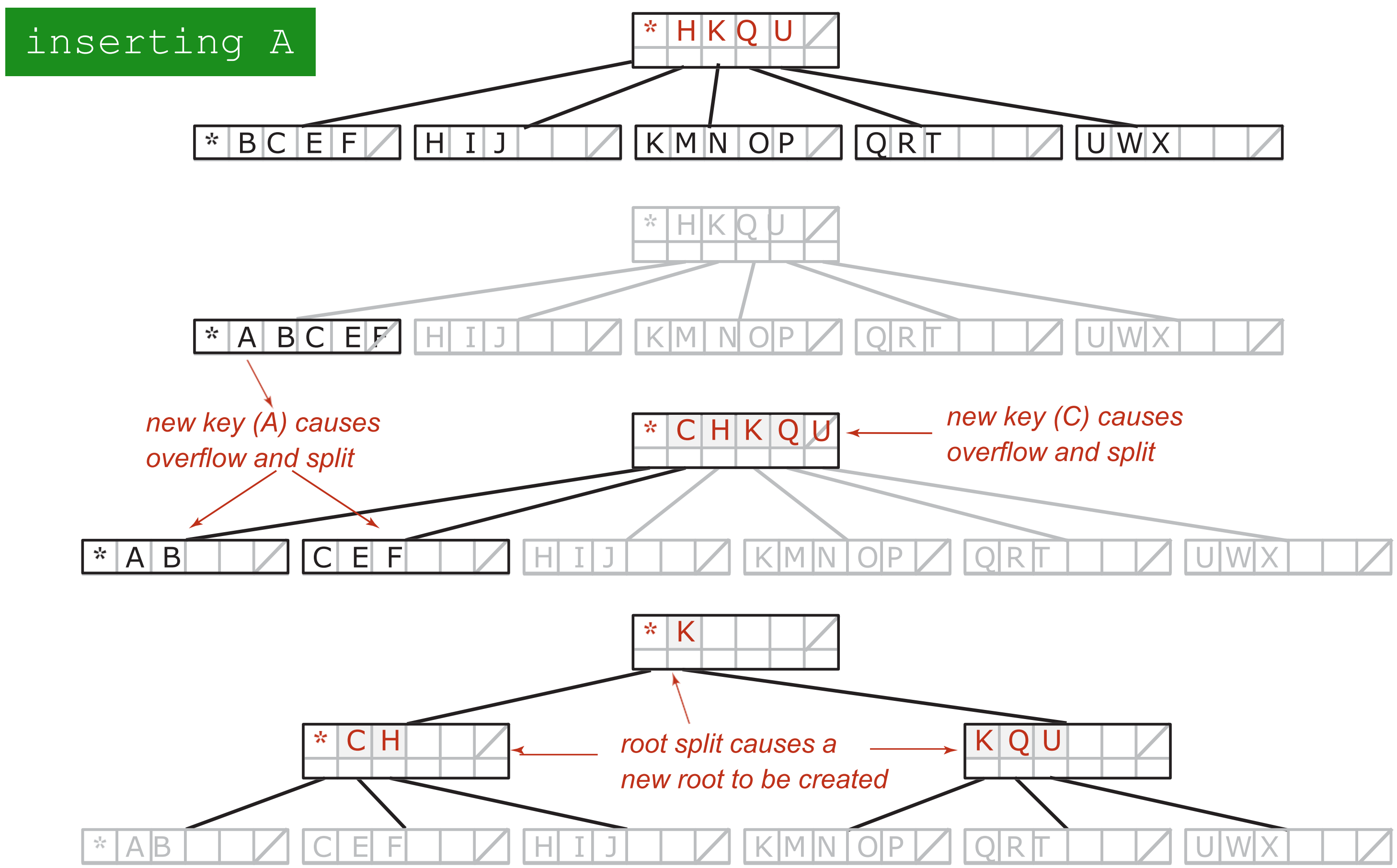
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between M / 2 and M - 1 links.

In practice. Number of probes is at most 4. ← M = 1024; N = 62 billion $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Balanced 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.
- B-tree variants. B+ tree, B*tree, B# tree, …
- B-trees (and variants) are widely used for file systems and databases.
  - Windows: NTFS.
  - Mac: HFS, HFS+.
  - Linux: ReiserFS, XFS, Ext3FS, JFS, BTRFS.
  - Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.