SEARCHING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

20: Balanced Binary Search Trees

Alexandra Papoutsaki
she/her/hers

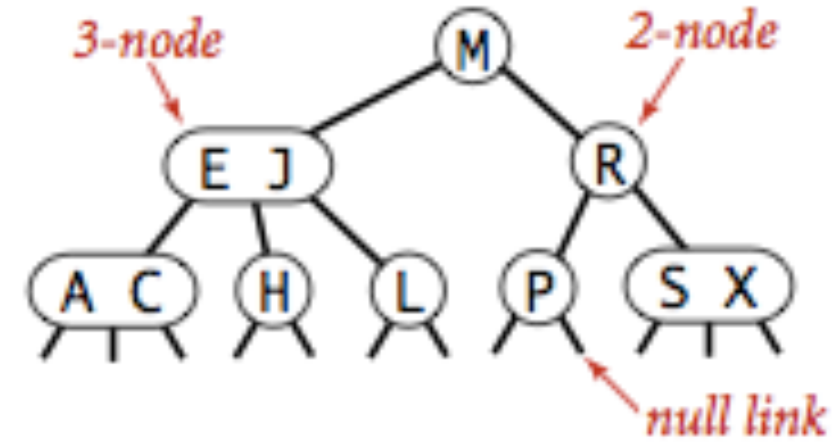
Lecture 20: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

Visualization of insertion into a binary search tree

- 255 insertions in random order.

2-3 tree

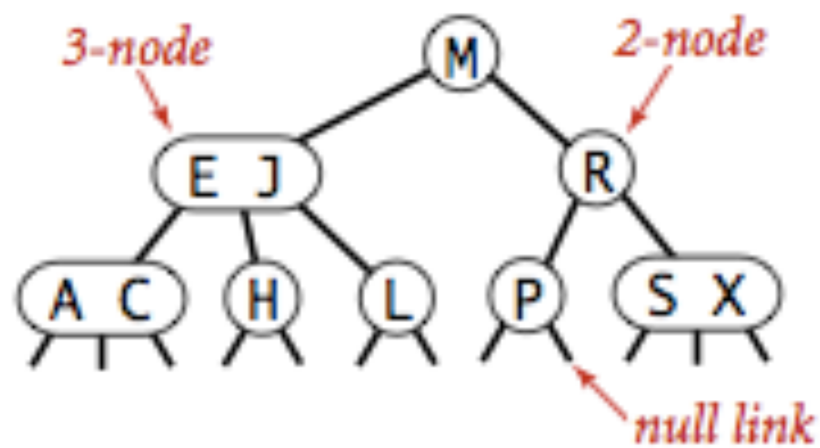


Anatomy of a 2-3 search tree

- Definition: A 2-3 tree is either empty or a
  - 2-node: one key (and associated value) and two links, a left to a 2-3 search tree with smaller keys, and a right to a 2-3 search tree with larger keys (similarly to standard BSTs), or a
  - 3-node: two keys (and associated values) and three links, a left to a 2-3 search tree with smaller keys, a middle to a 2-3 search tree with keys between the node's keys, and a right to a 2-3 search tree with larger keys.
- Symmetric order: In-order traversal yields keys in ascending order.
- Perfect balance: Every path from root to null link (empty tree) has the same length.

Example of a 2-3 tree

- 2-node, business as usual with BSTs.
  - (e.g., EJ are smaller than M and R is larger than M).
- In 3-node,
  - left link points to 2-3 search tree with smaller keys than first key,
    - (e.g., AC are smaller than E.)
  - middle link points to 2-3 search tree with keys between first and second key,
    - (e.g. H is between E and J.)
  - right link points to 2-3 search tree with keys larger than second key.
    - (e.g, L is larger than J).



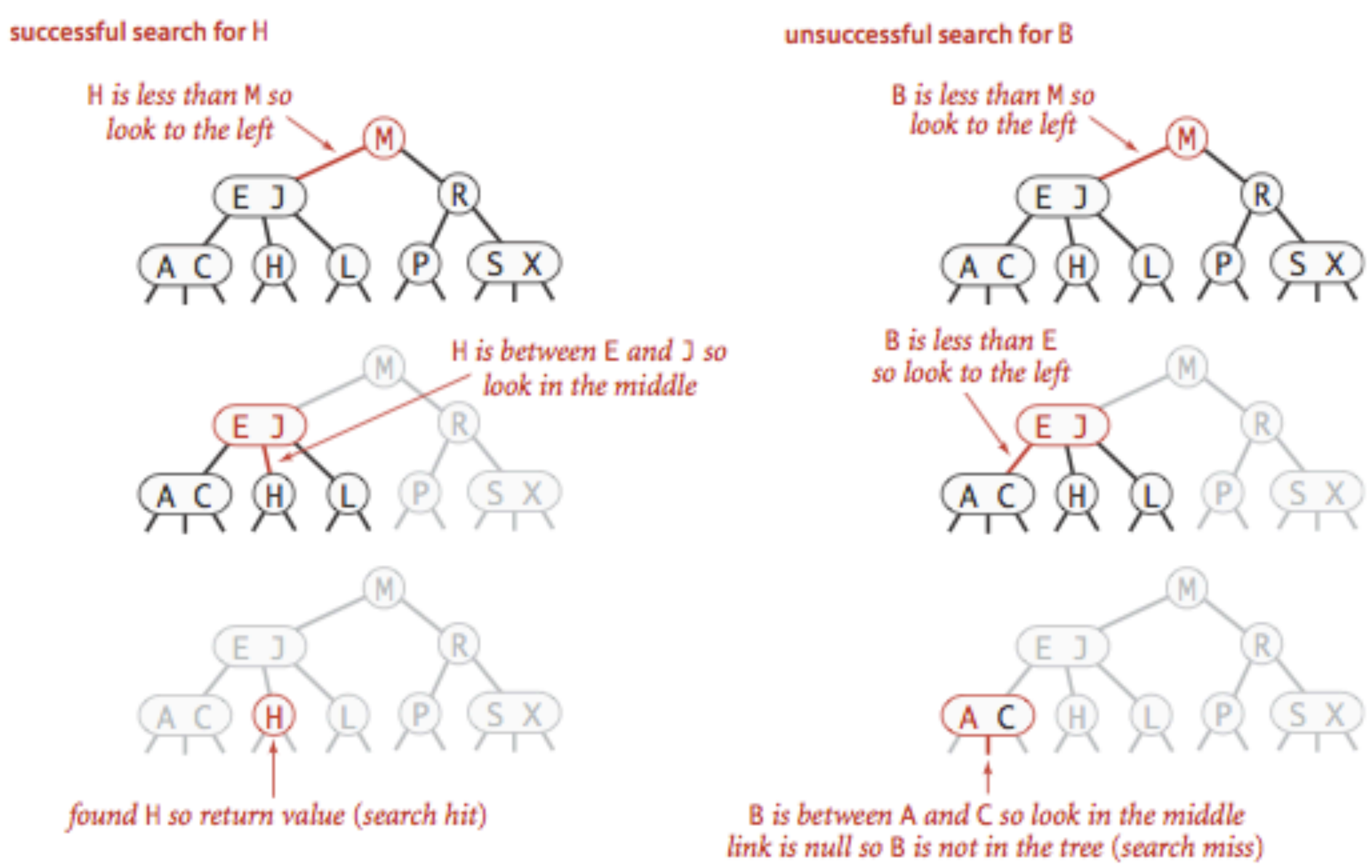
Anatomy of a 2-3 search tree

Lecture 20: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

How to search for a key

- Compare search key against (every) key in node.
- Find interval containing search key (left, potentially middle, or right).
- Follow associated link, recursively.



Search hit (left) and search miss (right) in a 2-3 tree

http://algs4.cs.princeton.edu

3.3 2–3 Tree Demo

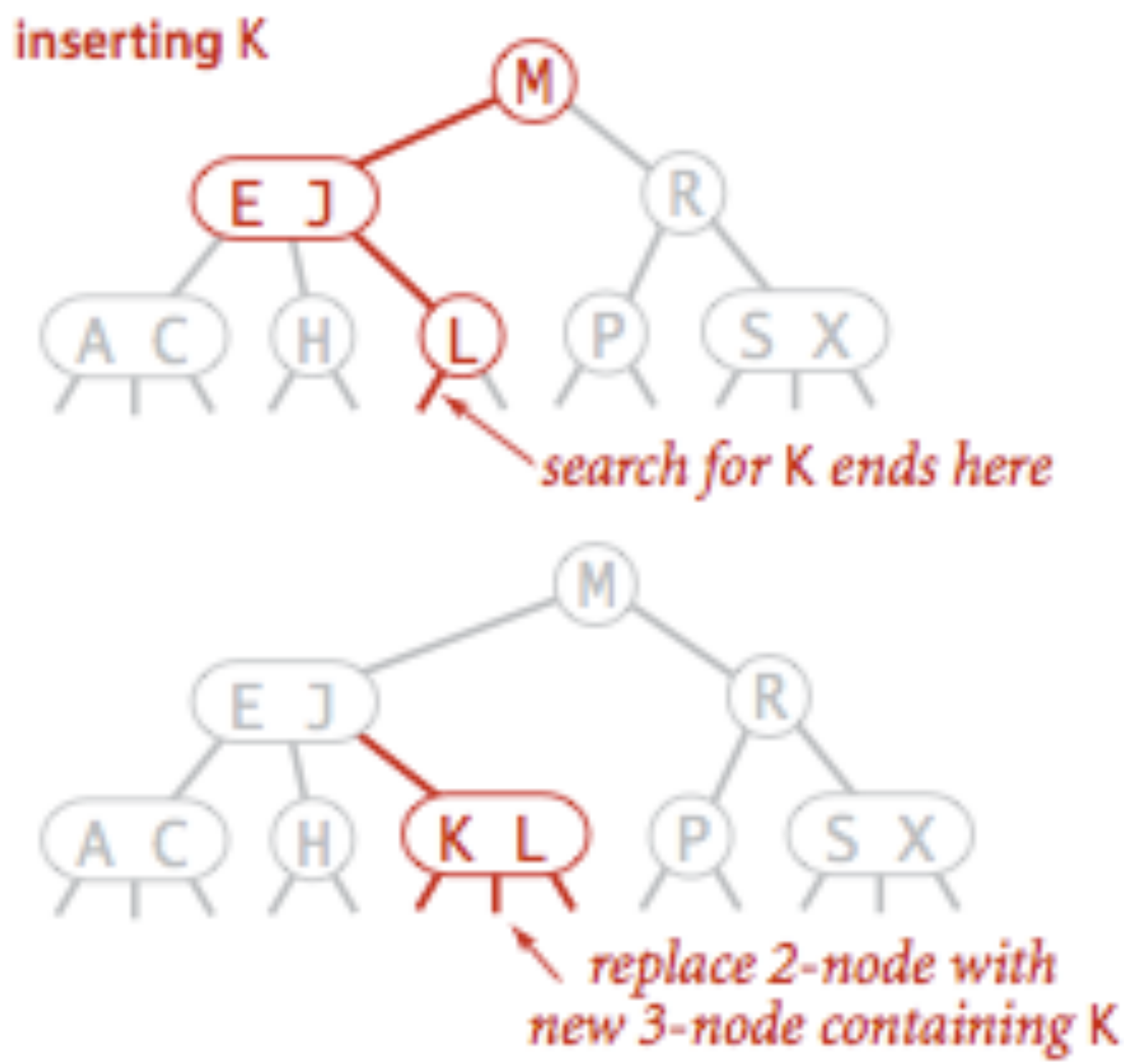
- *search*
- *insertion*
- *construction*

Lecture 20: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

How to insert into a 2-node

- Search for key and add new key to 2-node to create a 3-node.



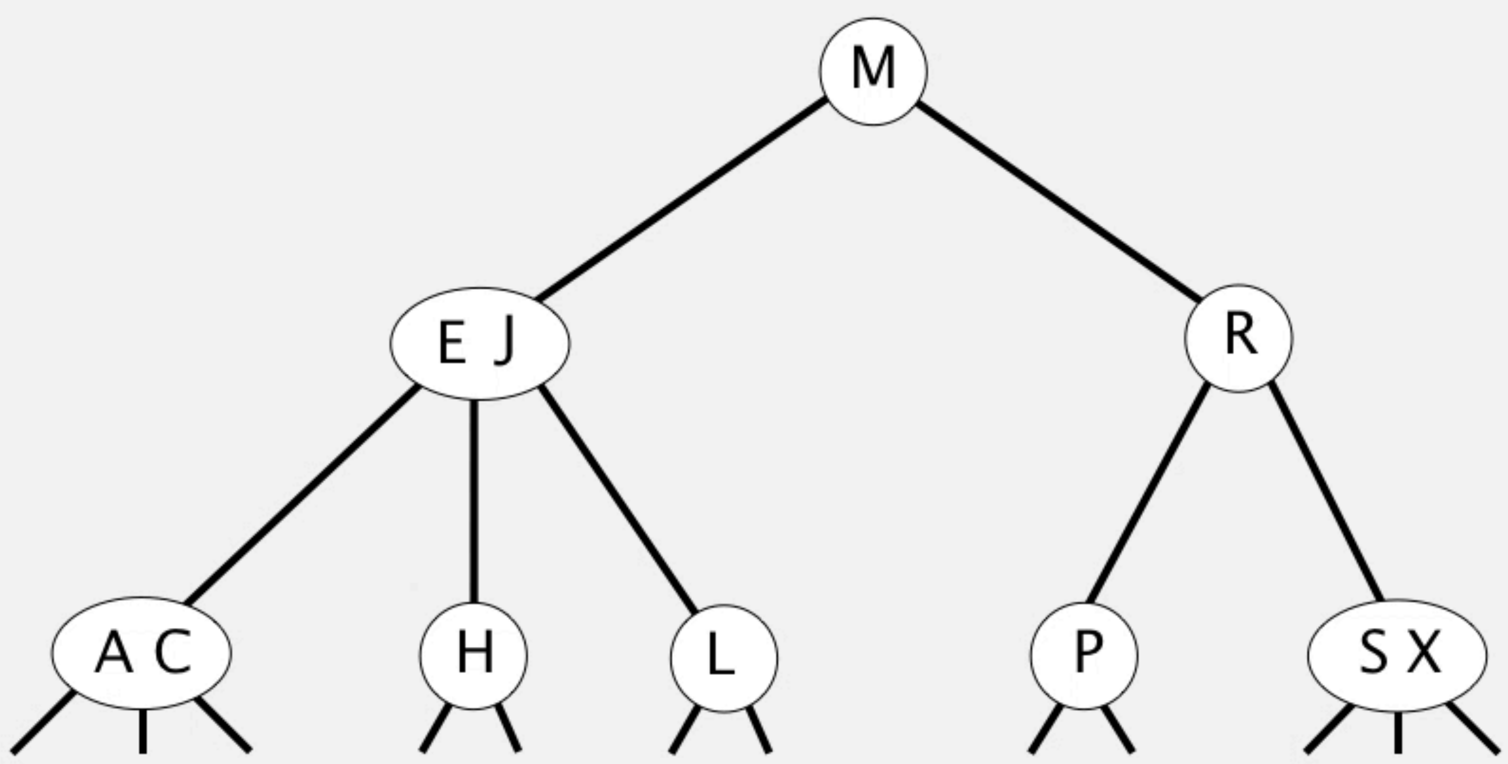
Insert into a 2-node

2–3 tree demo: insertion

Insert into a 2-node at bottom.

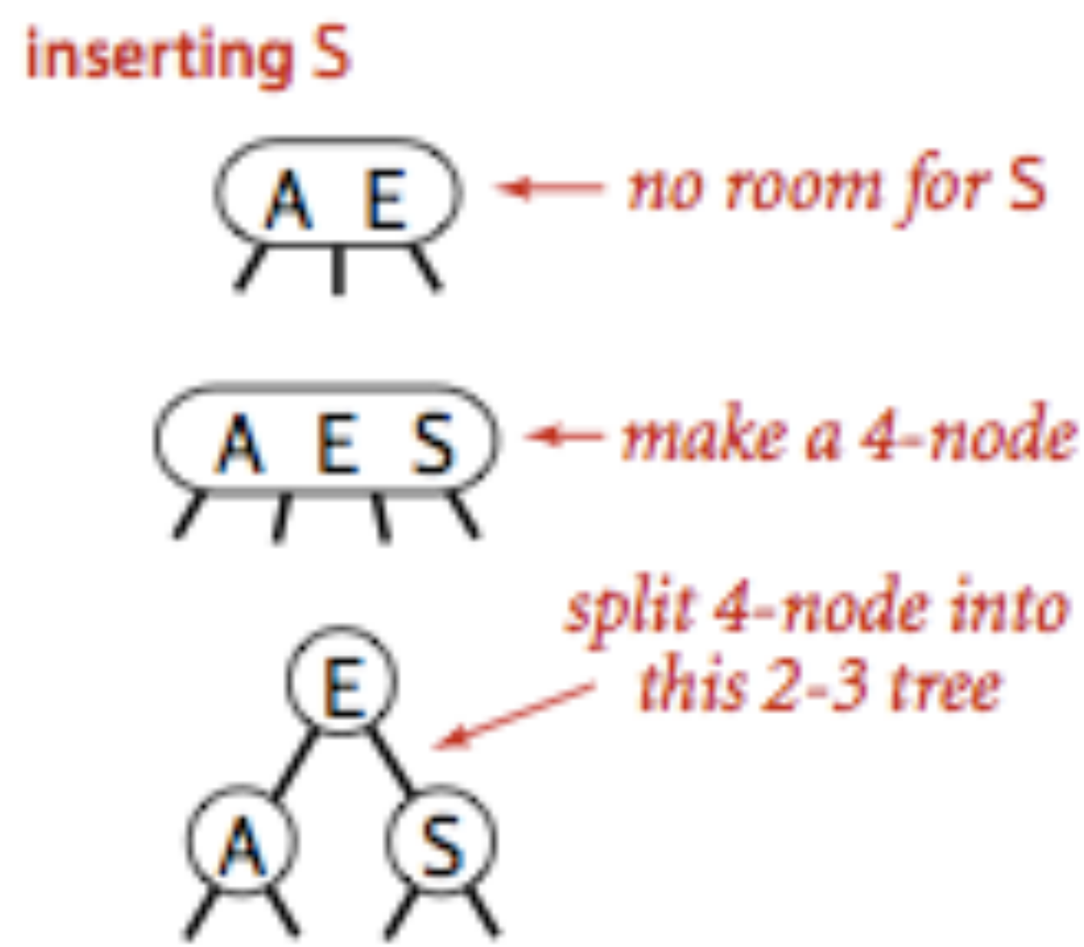
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K



How to insert into a tree consisting of a single 3-node

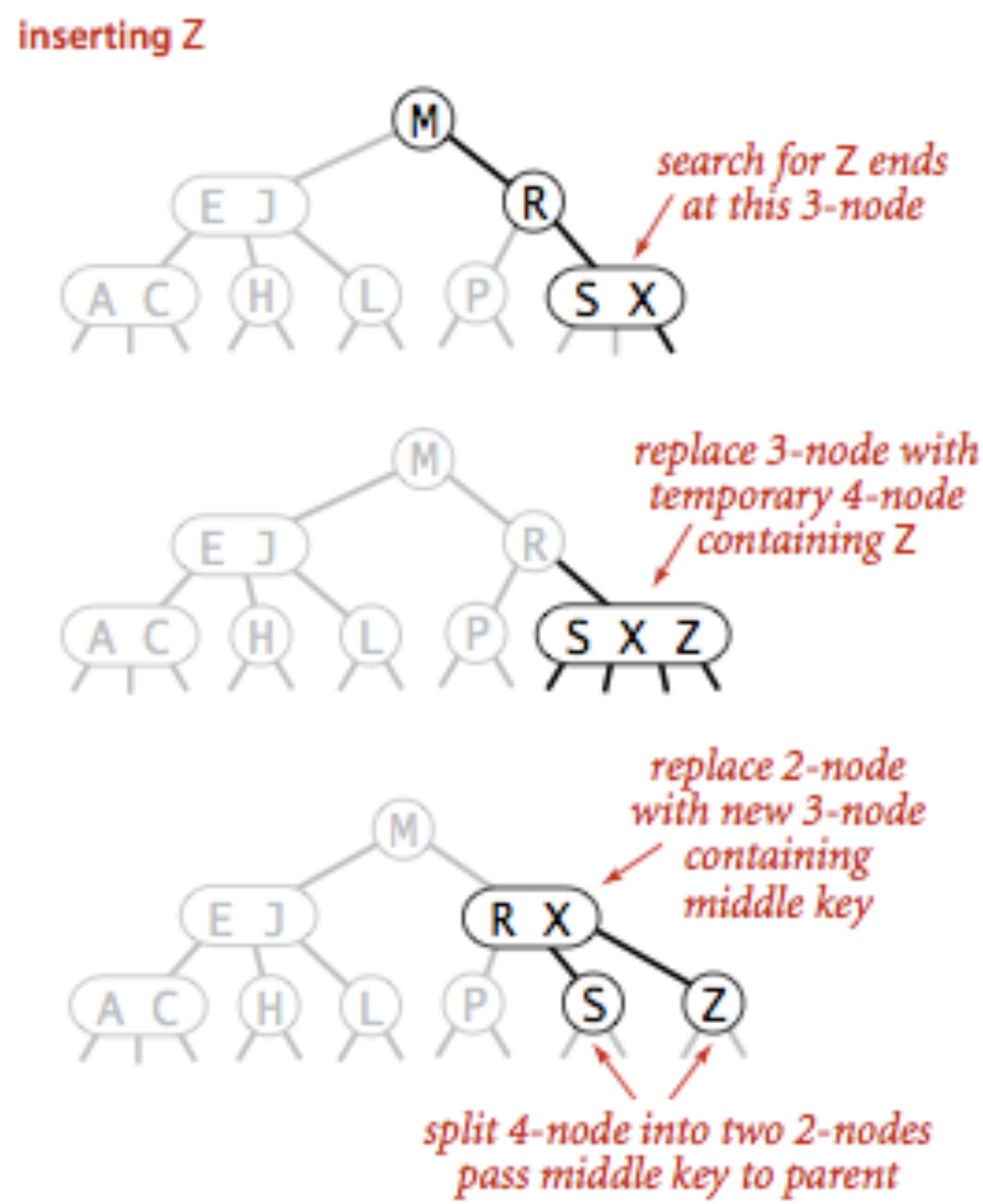
- Add new key to 3-node to create a temporary 4-node.
- Move middle key in 4-node into parent.
- Split 4-node into two 2-nodes.
- Height went up by 1.



Insert into a single 3-node

How to insert into a 3-node whose parent is a 2-node

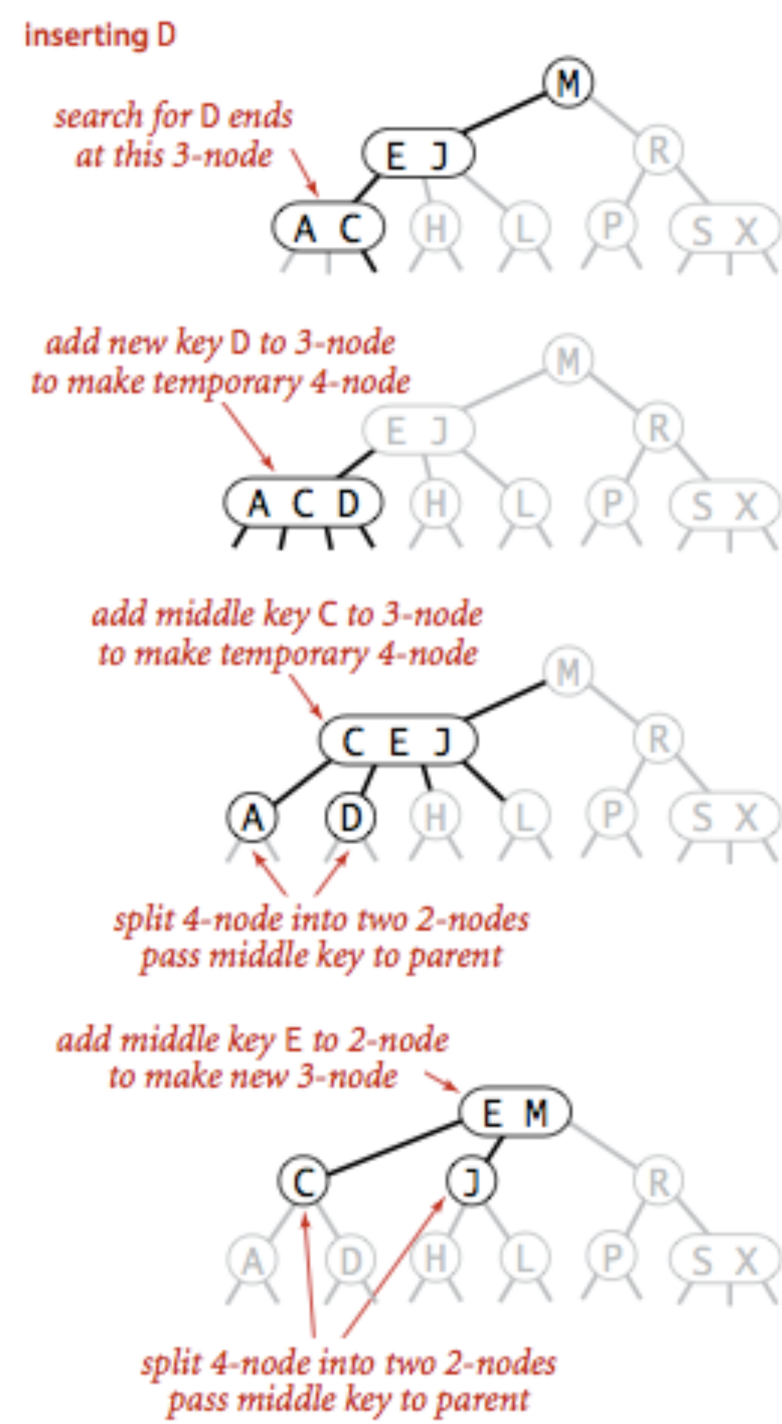
- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Replace 2-node parent with 3-node.



Insert into a 3-node whose parent is a 2-node

How to insert into a 3-node whose parent is a 3-node

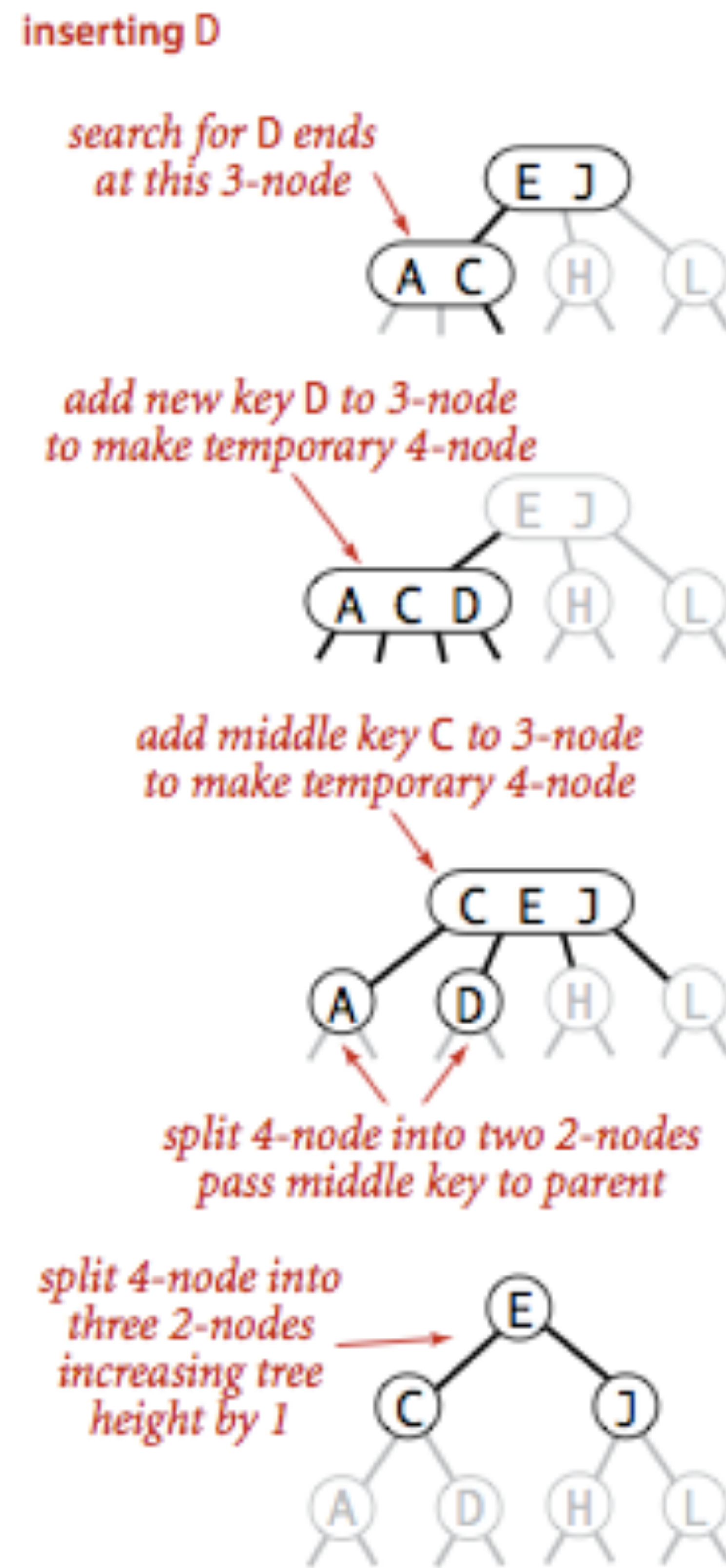
- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent creating a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Repeat up the tree, as necessary.



Insert into a 3-node whose parent is a 3-node

Splitting the root

- If end up with a temporary 4-node root, split into three 2-nodes.
- Increases height by 1 but perfect balance is preserved.



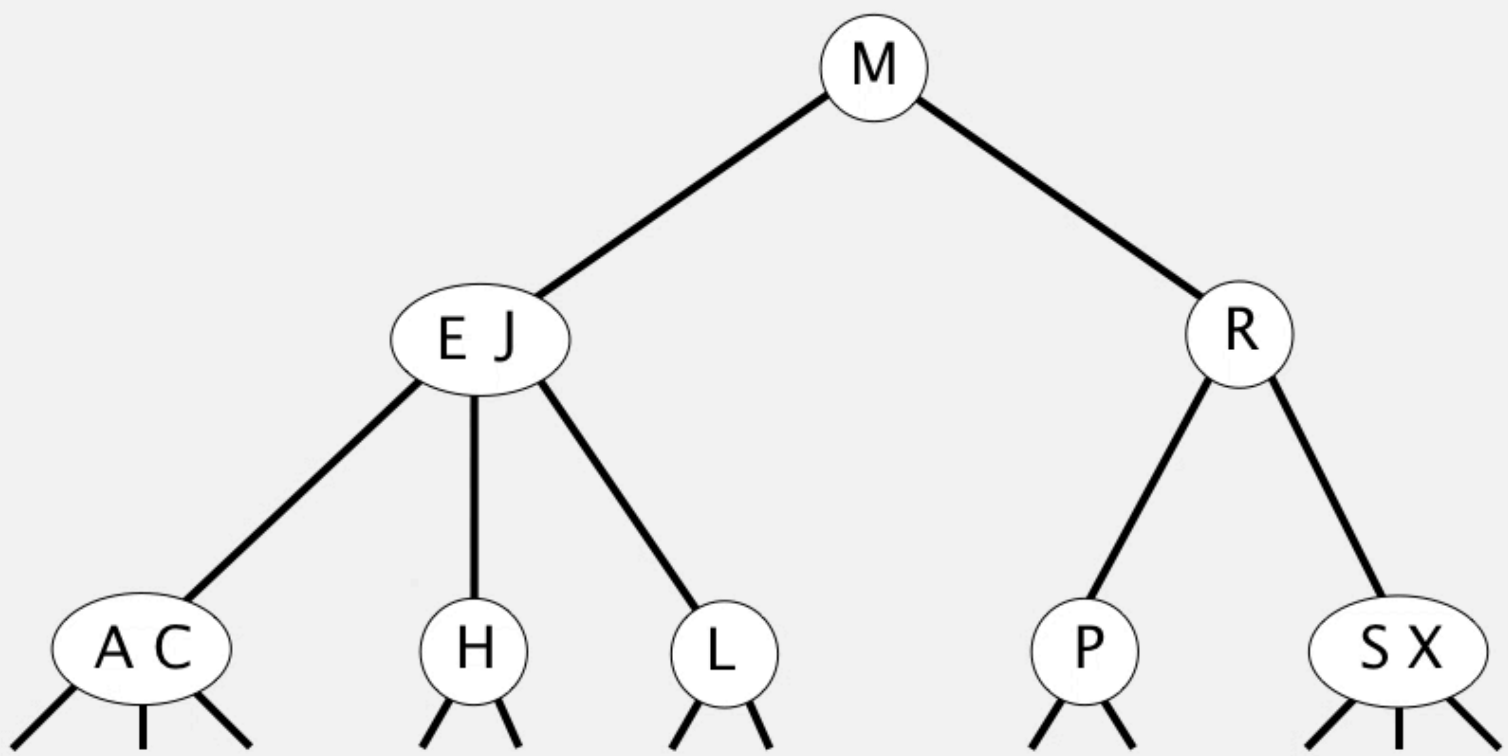
Splitting the root

2–3 tree demo: insertion

Insert into a 2-node at bottom.

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

insert K

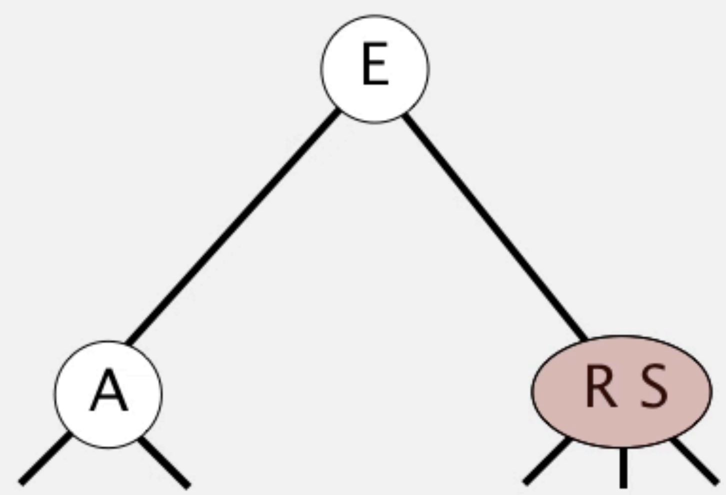


Lecture 20: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

2-3 tree demo: construction

insert R

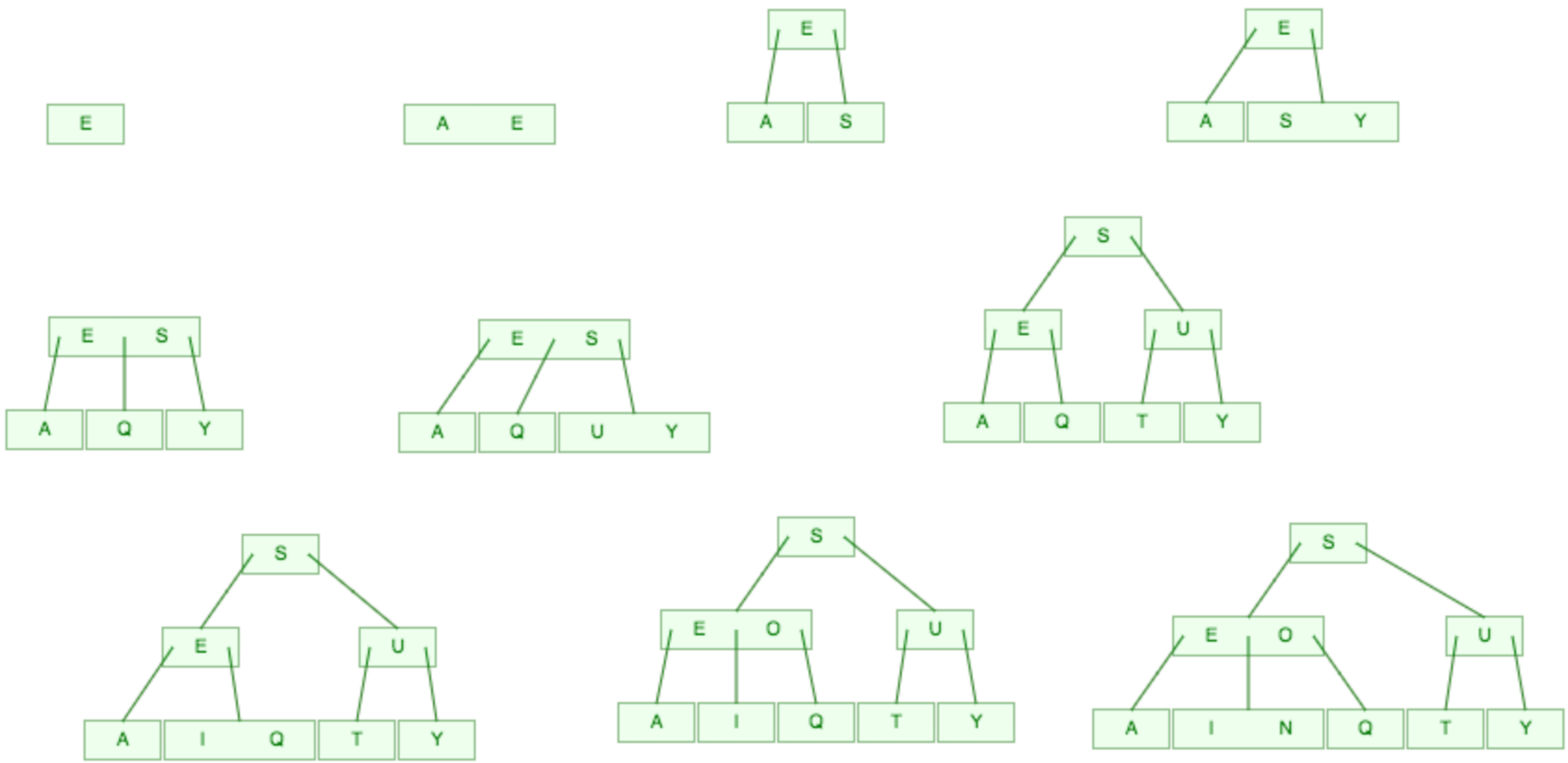


Practice Time

- Draw the 2-3 tree that results when you insert the keys: E A S Y Q U T I O N in that order in an initially empty tree.

Answer

- E A S Y Q U T I O N



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

Lecture 20: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

Height of 2-3 search trees

- Worst case: $\log n$ (all 2-nodes).
- Best case: $\log_3 n = 0.631 \log n$ (all 3-nodes)
  - That means that storing a million nodes will lead to a tree with height between 12 and 20, and storing a billion nodes to a tree with height between 19 and 30 (not bad!).
- Search and insert are $O(\log n)$!
- But implementation is a pain and the overhead incurred could make the algorithms slower than standard BST search and insert.
- We did provide insurance against a worst case but we would prefer the overhead cost for that insurance to be low. Stay tuned!

Summary for symbol table/dictionary operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
2-3 search trees	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Lecture 20: Left-leaning Red-Black Trees

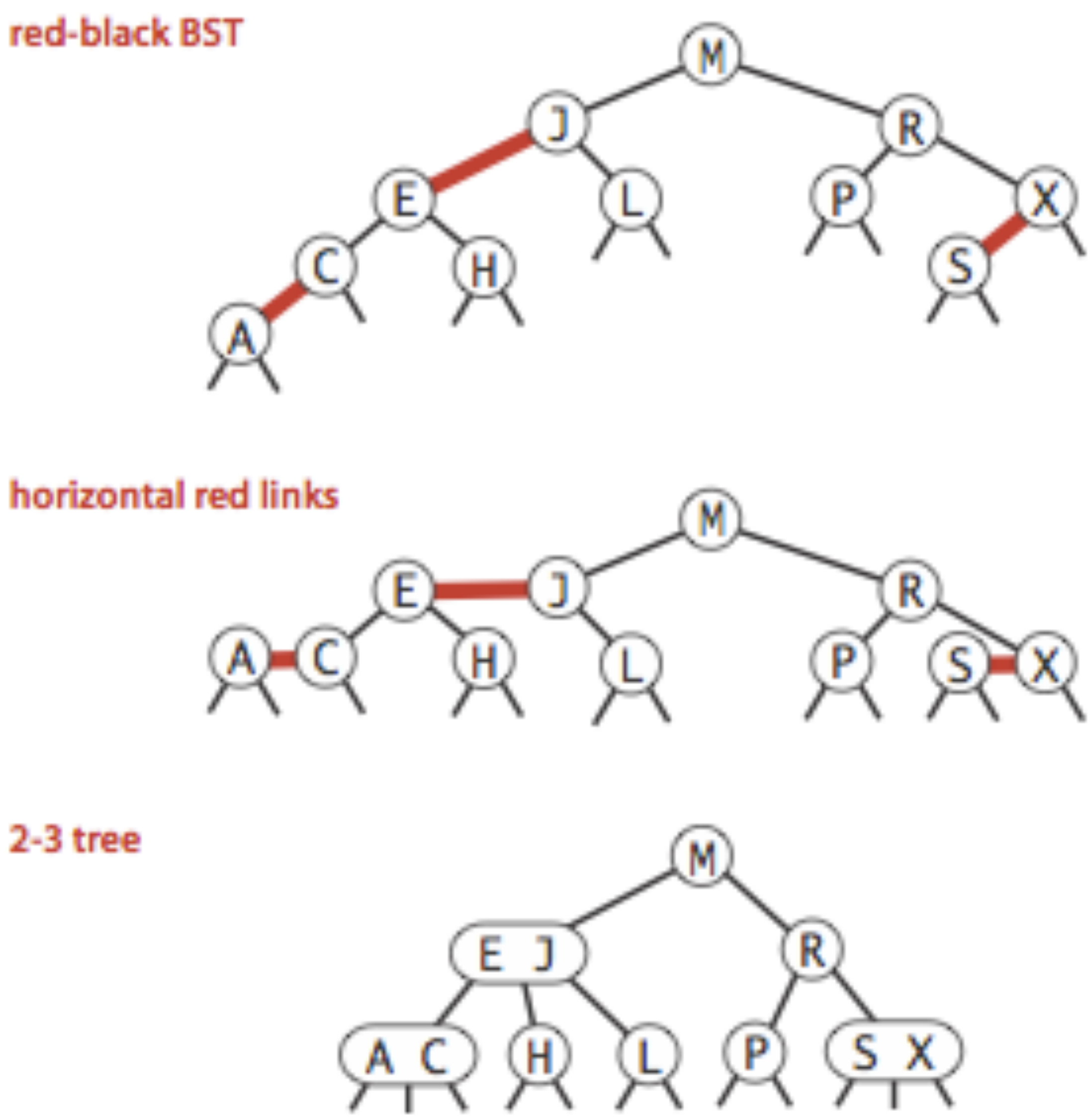
- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Some slides adopted from Algorithms 4th Edition or COS226

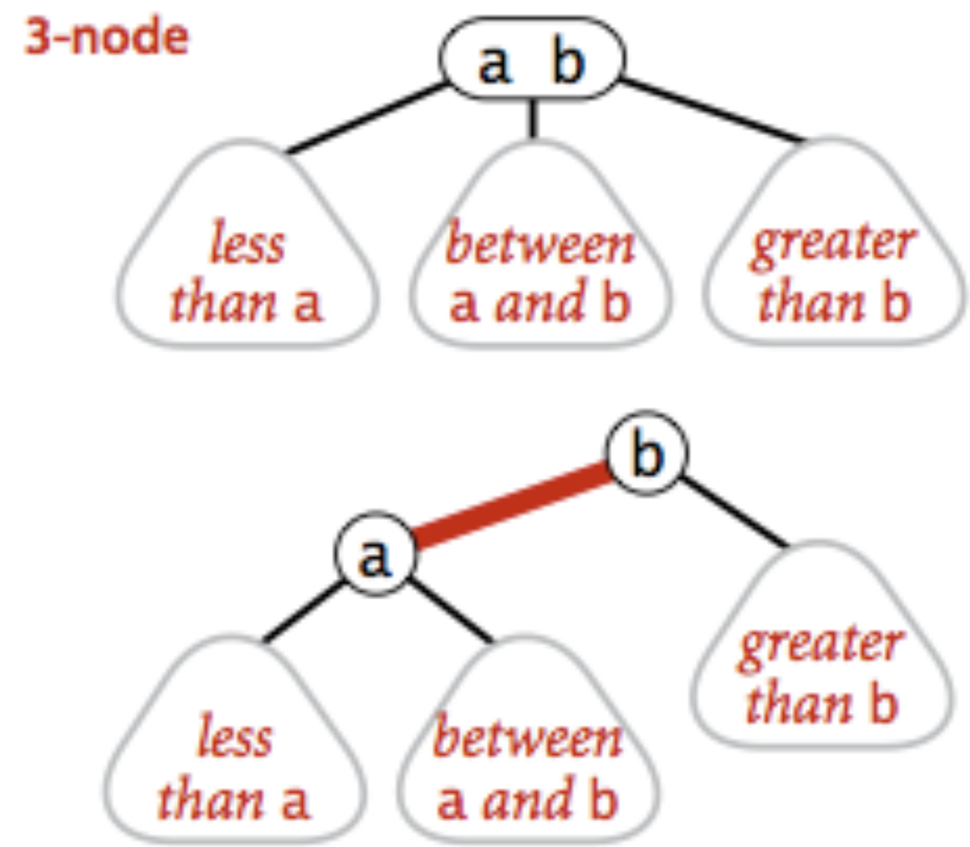
Left-leaning red-black BSTs correspond 1-1 with 2-3 trees

- Start with standard BSTs which are made up of 2-nodes.
- Add extra information to encode 3-nodes. We will introduce two types of links.
- Red links: bind together two 2-nodes to represent a 3-node.
  - Specifically, 3-nodes are represented as two 2-nodes connected by a single red link that leans left (one of the 2-nodes is the left child of the other).
- Black links: bind together the 2-3 tree.
- Advantage: Can use BST code with minimal modification.

Left-leaning red-black BSTs correspond 1-1 with 2-3 trees

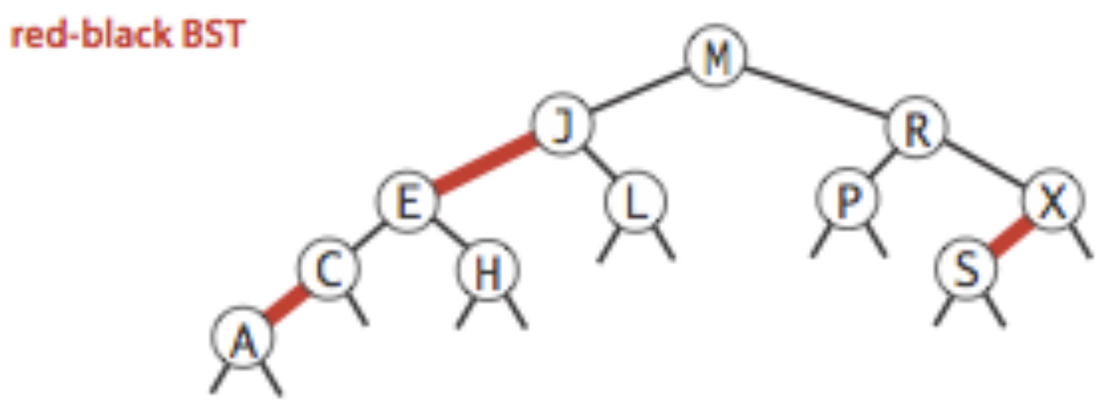


1-1 correspondence between red-black BSTs and 2-3 trees



Definition

- A left-leaning red-black tree is a BST such that:
  - No node has two red links connected to it.
  - Red link leans left.
  - Every path from root to leaves has the same number of black links (perfect black balance).



Search

- Exactly the same as for elementary BSTs (we ignore the color).
  - But runs faster because of better balance.

```
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    return get(root, key);
}

// value associated with the given key in subtree rooted at x; null if no such key
private Value get(Node x, Key key) {
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else              return x.val;
    }
    return null;
}
```

- Operations such as floor, iteration, rank, selection are also identical.

Representation

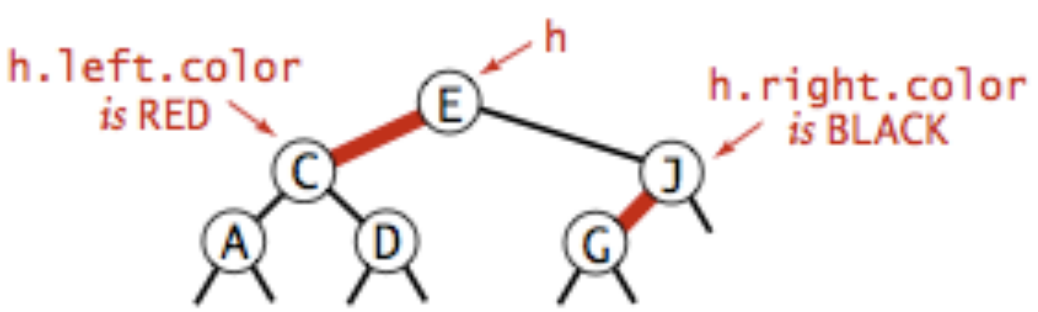
- Each node is pointed to by one node, its parent. We can use this to encode the color of the links in nodes.
- True if the link from the parent is red and false if it is black. Null links are black.

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

private Node root;     // root of the BST

// BST helper node data type
private class Node {
    private Key key;           // key
    private Value val;         // associated data
    private Node left, right;  // links to left and right subtrees
    private boolean color;     // color of parent link
    private int size;          // subtree count

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

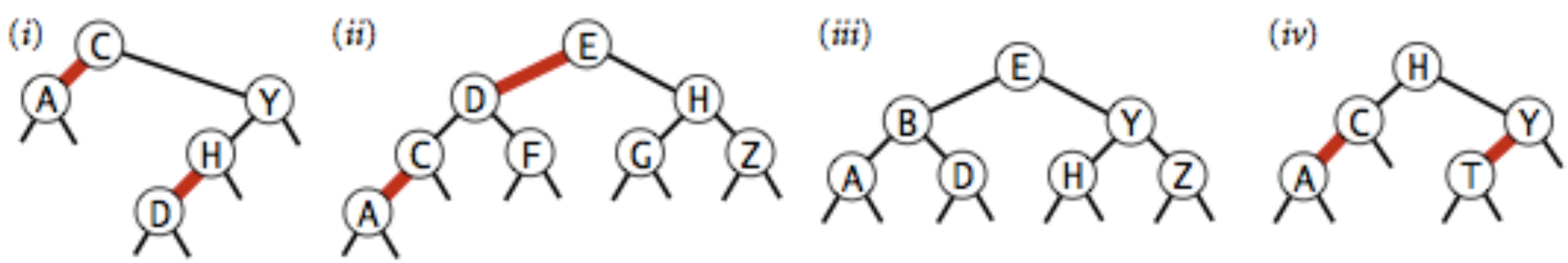


Story so far

- BSTs can get imbalanced and long.
- 2-3 trees are balanced but cumbersome to code.
- Imagine 3-nodes held together by internal glue links shown in red.
- Draw links by giving them red or black color.
- Represent them in memory by storing the color of the link coming from the parent as the color of the child node.

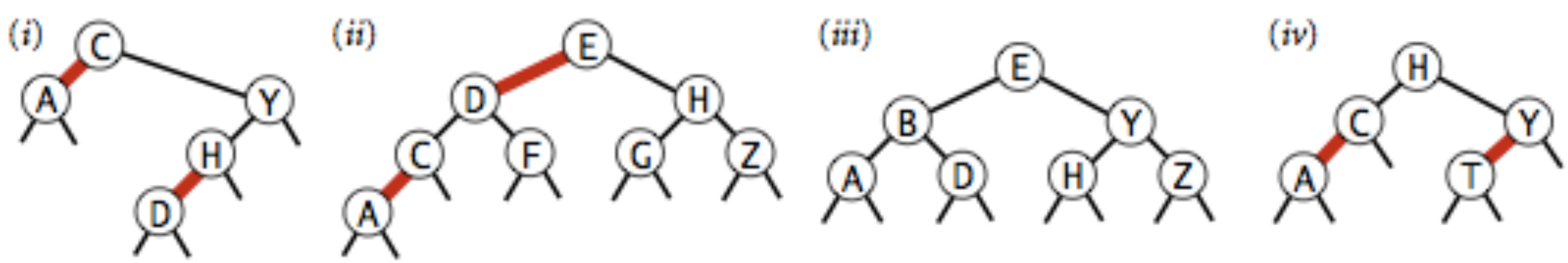
Practice Time

- Which of the following are legal LLRB trees?



Answer

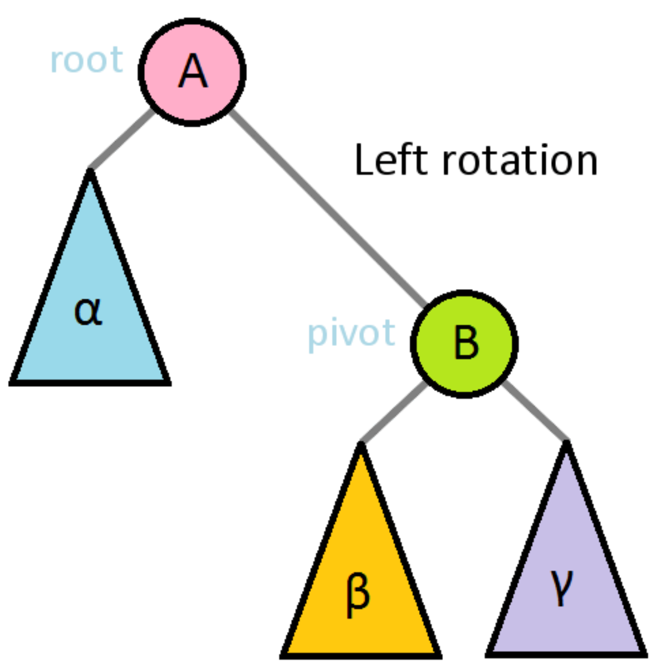
- Which of the following are legal LLRB trees?
- iii and iv
    - i is not balanced and ii is also not in symmetrical order

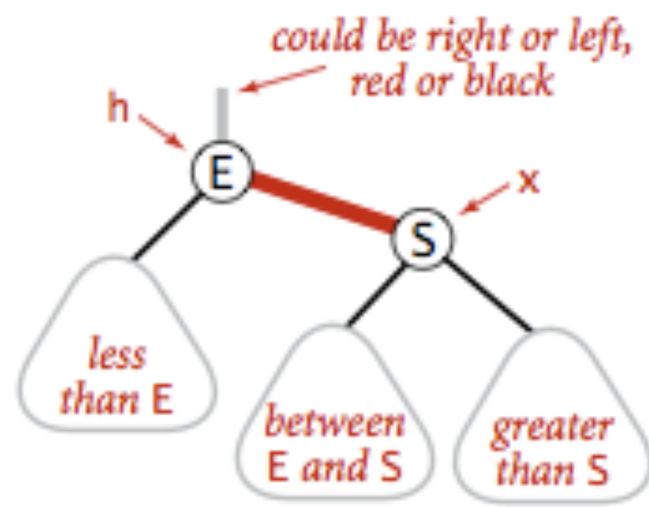


Lecture 20: Left-leaning Red-Black Trees

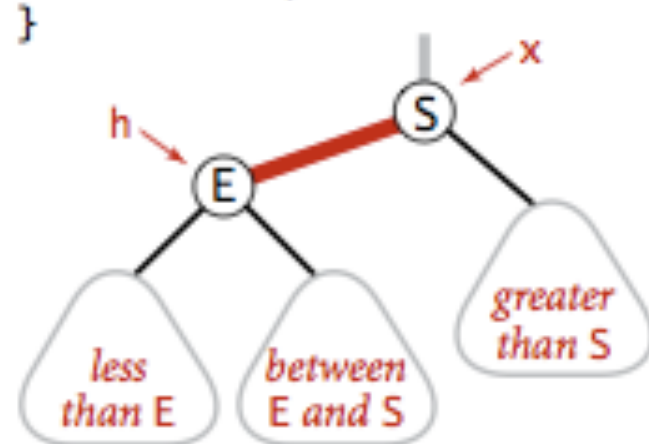
- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

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



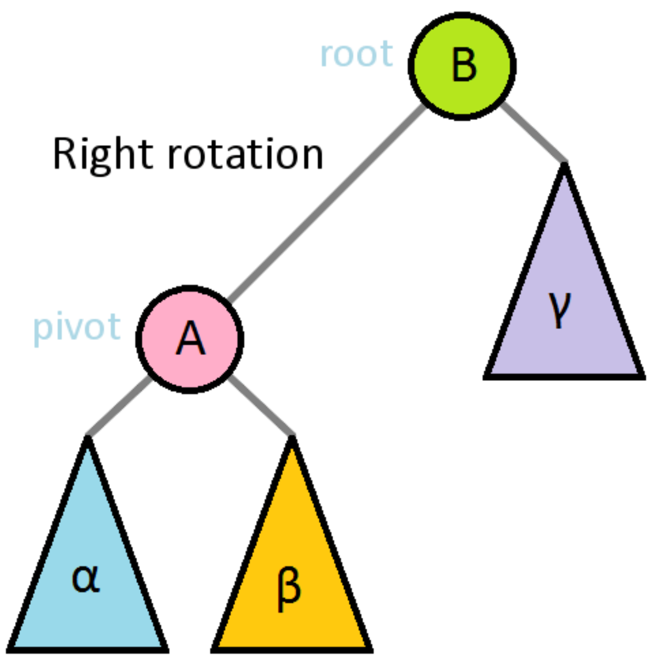


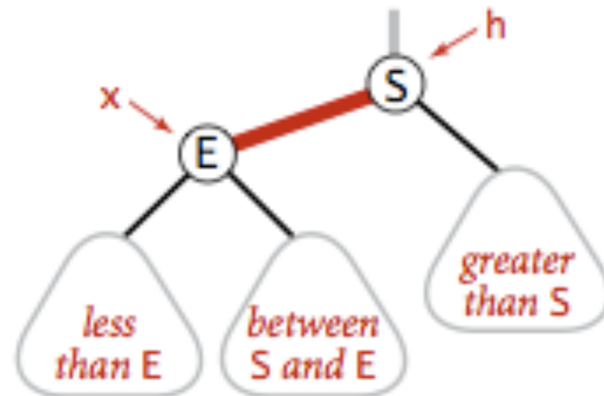

```
Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
            + size(h.right);
    return x;
}
```



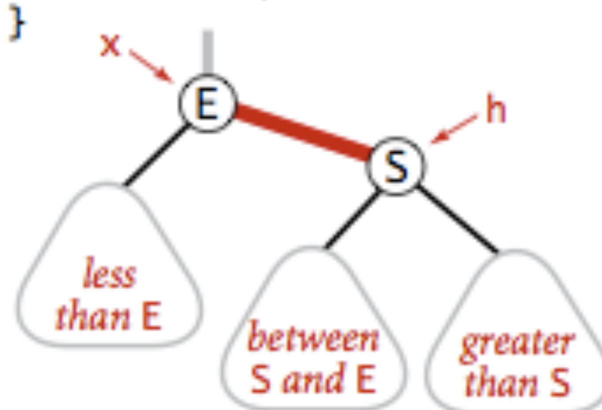
Left rotate (right link of h)

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



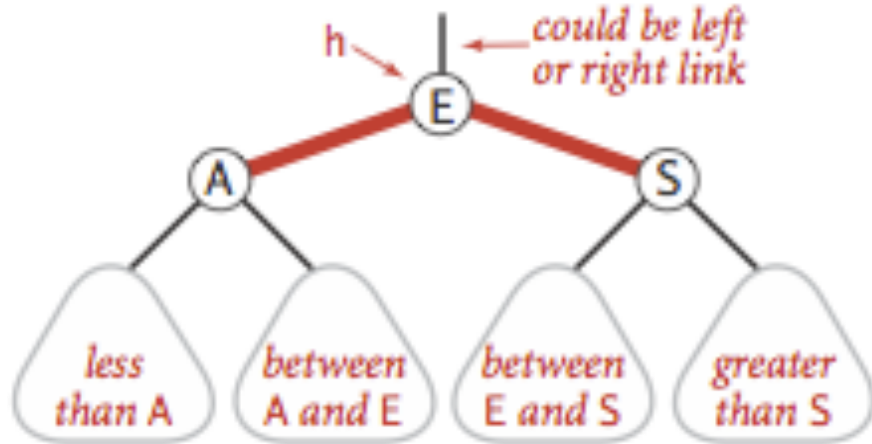



```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

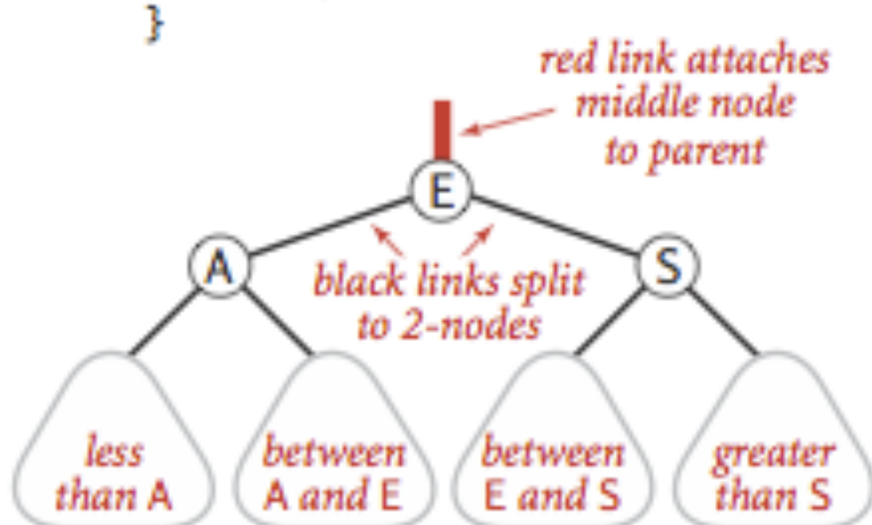


Right rotate (left link of h)

Color flip: Recolor to split a (temporary) 4-node




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



Flipping colors to split a 4-node

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

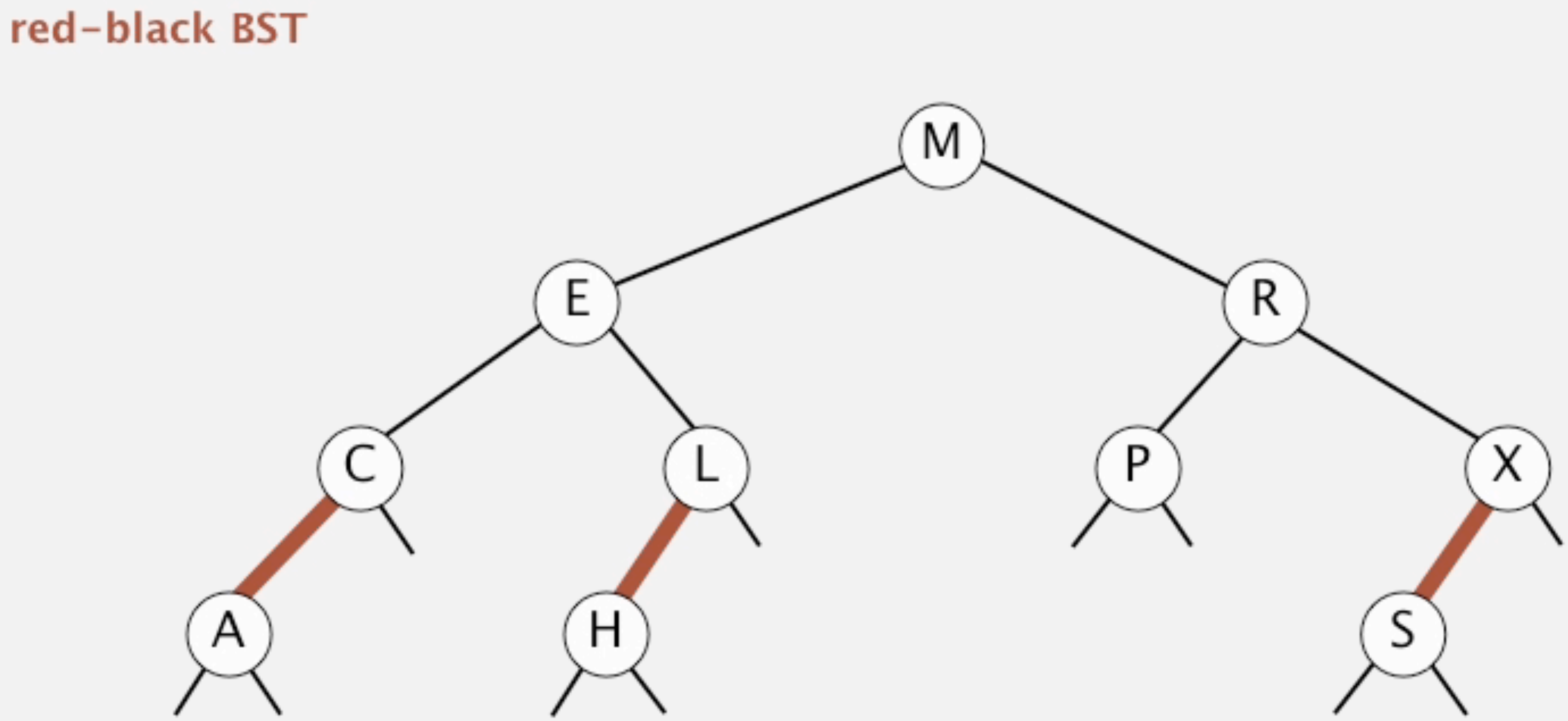
Basic strategy: Maintain 1-1 correspondence with 2-3 trees

- During internal operations, maintain:
  - symmetric order.
  - perfect black balance.
- But we might violate color invariants. For example:
  - Right-leaning red link.
  - Two red children (temporary 4-node).
  - Left-left red (temporary 4-node).
  - Left-right red (temporary 4-node).
- To restore color invariant we will be performing rotations and color flips.

Insertion into a LLRB

- Do standard BST insertion and color the new link red.
- Repeat until color invariants restored:
  - Both children red? Flip colors.
  - Right link red? Rotate left.
  - Two left reds in a row? Rotate right.

Red-black BST construction demo



Implementation

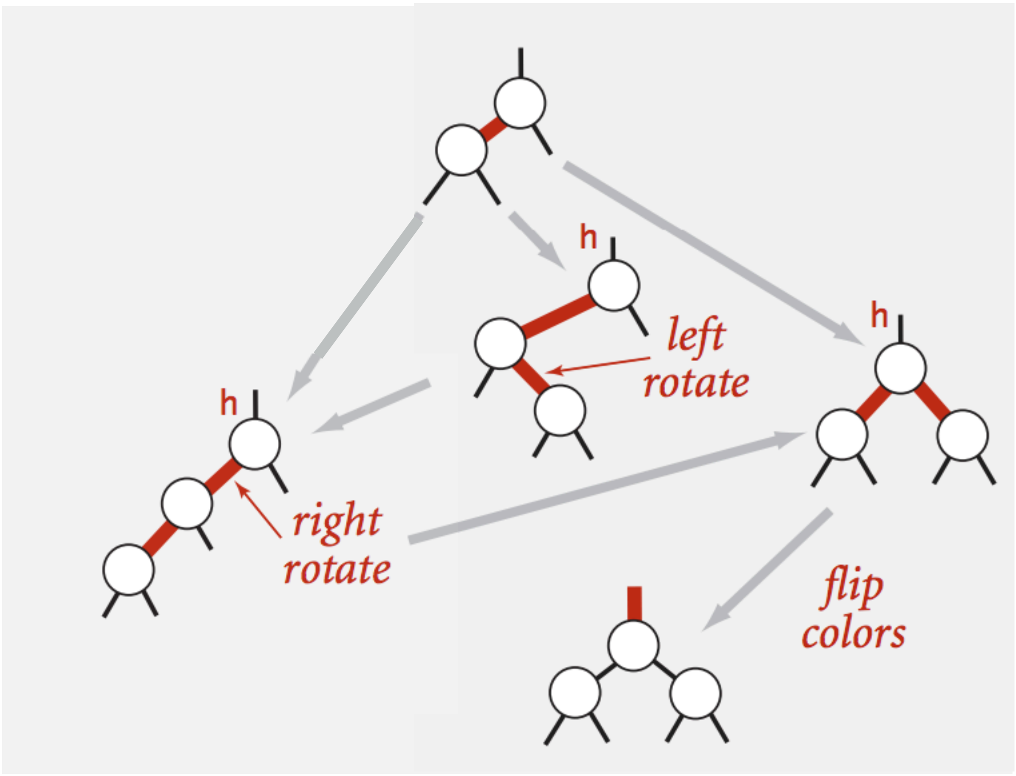
- Only three cases:
  - Right child red; left child black: rotate left.
  - Left child red; left-left grandchild red: rotate right.
  - Both children red: flip colors.

```
// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED, 1);

    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              h.val   = val;

    // fix-up any right-leaning links
    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);
    h.size = size(h.left) + size(h.right) + 1;

    return h;
}
```



Visualization of insertion into a LLRB tree

- 255 insertions in ascending order.

Visualization of insertion into a LLRB tree

- 255 insertions in descending order.

Visualization of insertion into a LLRB tree

- 255 insertions in random order.

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Balance in LLRB trees

- Height of LLRB trees is $\leq 2 \log n$ in the worst case.
- Worst case is a 2-3 tree that is all 2-nodes except that the left-most path is made up of 3-nodes.
- All ordered operations (min, max, floor, ceiling) etc. are also $O(\log n)$.

Summary for dictionary/symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
Sequential search (unordered	n	n	n	$n/2$	n	$n/2$
Binary search (ordered array)	$\log n$	n	n	$\log n$	$n/2$	$n/2$
BST	n	n	n	$1.39 \log n$	$1.39 \log n$	$\sqrt{n}$
2-3 search tree	$c \log n$	$c \log n$	$c \log n$	$c \log n$	$c \log n$	$c \log n$
Red-black BSTs	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Red-black trees

- A dichromatic framework for balanced trees. [Guibas and Sedgewick, 1978].
- Why red-black? Xerox PARC had a laser printer and red and black had the best contrast…
- Left-leaning red-black trees [Sedgewick, 2008]
  - Inspired by difficulties in proper implementation of RB BSTs.
- RB BSTs have been involved in lawsuit because of improper implementation.

Balanced trees in the wild

- Red-black trees are widely used as system dictionaries.
  - e.g., Java: `java.util.TreeMap` and `java.util.TreeSet`.
- Other balanced BSTs: AVL, splay, randomized.
- 2-3 search trees are a subset of b-trees.
  - See recommended textbook for more.
  - B-trees are widely used for file systems and databases.

Readings:

- Recommended Textbook: Chapter 3.3 (Pages 424-447)
- Website:
  - https://algs4.cs.princeton.edu/33balanced/
- Visualization:
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html (for 2-3 trees)
  - https://algs4.cs.princeton.edu/GrowingTree/ (for LLRB trees)

Practice Problems:

- 3.2.1-3.2.13, 3.3.2-3.3.5, 3.3.9-3.3.22
- In-class worksheet