SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

20: Left-leaning Red Black Trees

Tom Yeh
he/him/his

Order of growth for symbol table operations

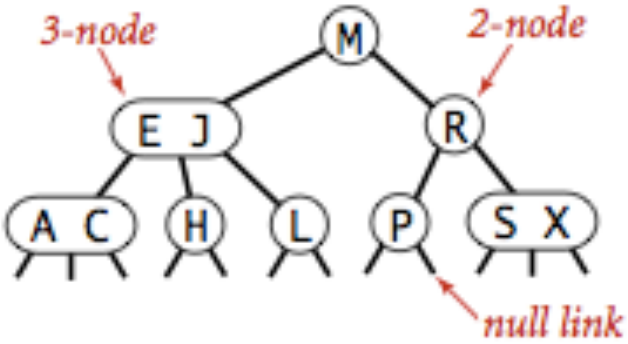
	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
Goal	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Lecture 20: Red-Black Search Trees

- 2-3 Search Trees
- Left-leaning Red-Black Trees
- Midterm Topics

Some slides adopted from Algorithms 4th Edition or COS226

2-3 tree

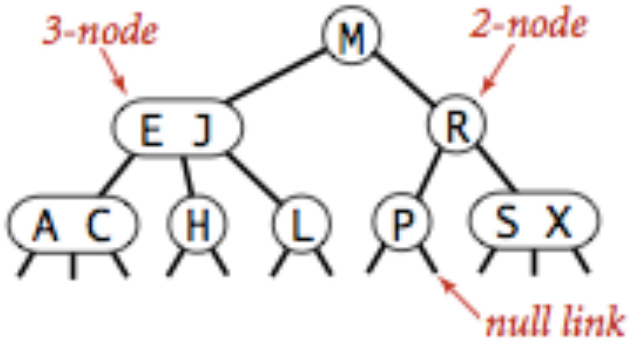


Anatomy of a 2-3 search tree

- 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.
  - (Node M: 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).



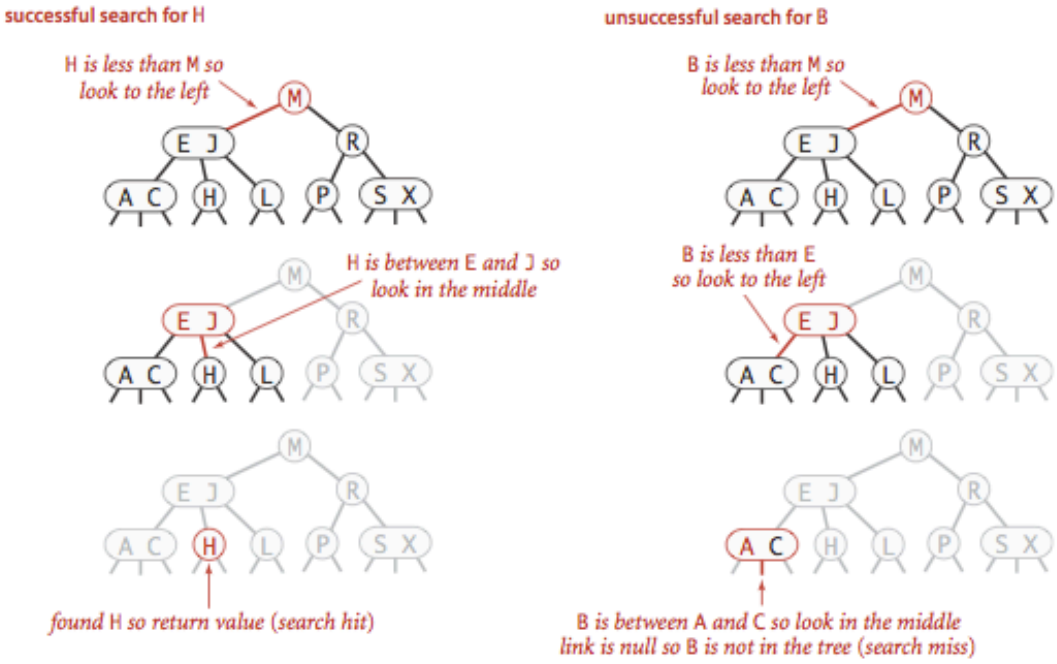
Anatomy of a 2-3 search tree

Lecture 19: 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

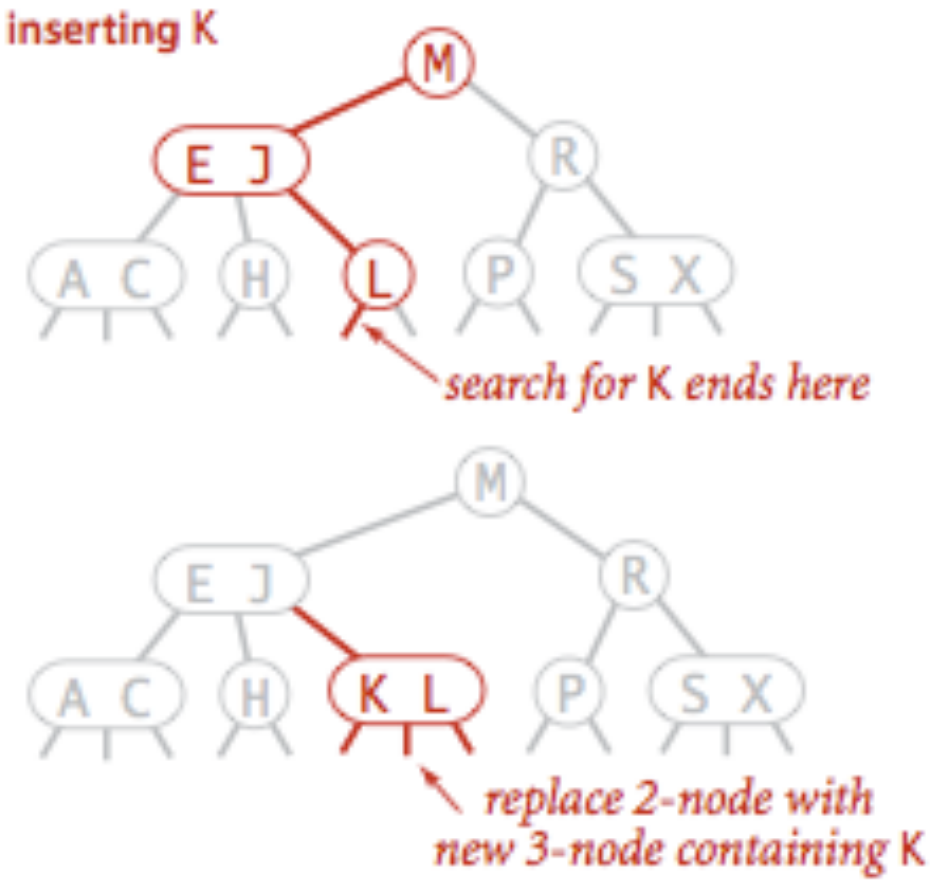
- *search*
- *insertion*
- *construction*

Lecture 24: 2-3 Search Trees

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

How to insert into a 2-node

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



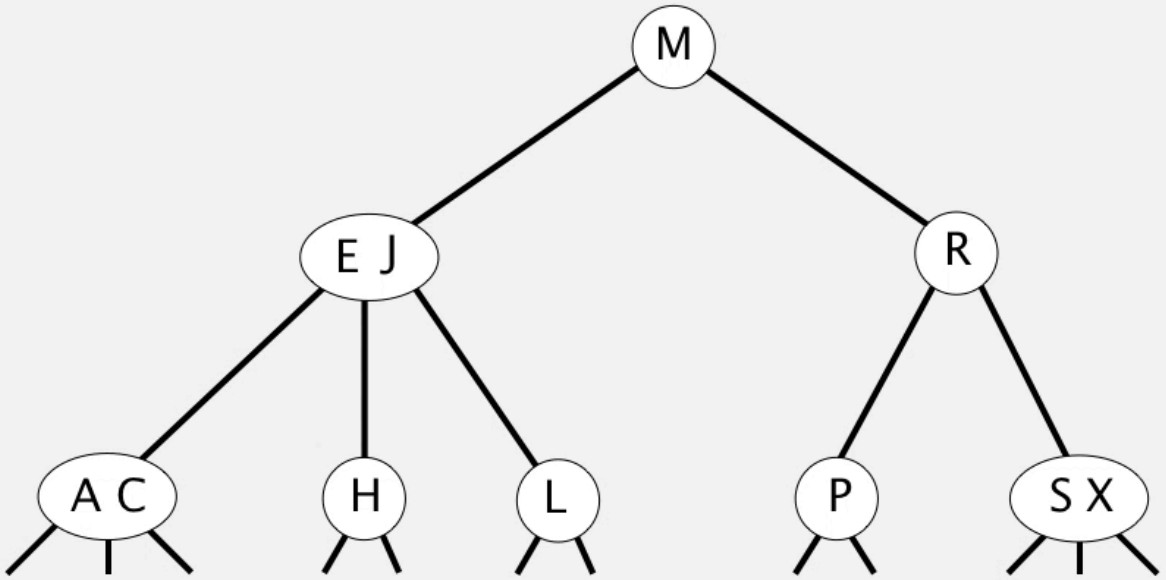
Insert into a 2-node

2-3 tree demo: insertion

Insert into a 2-node at bottom.

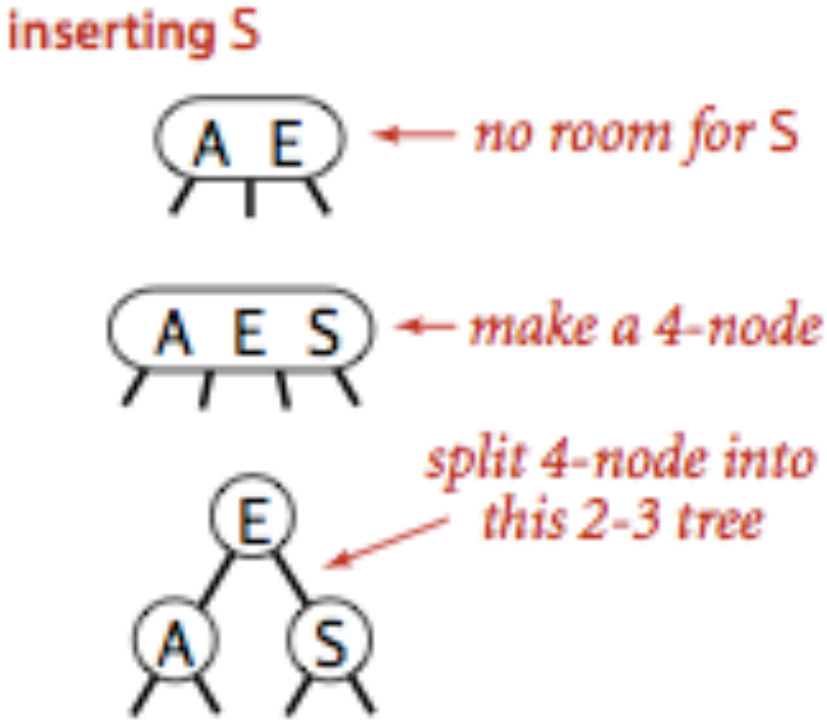
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K



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

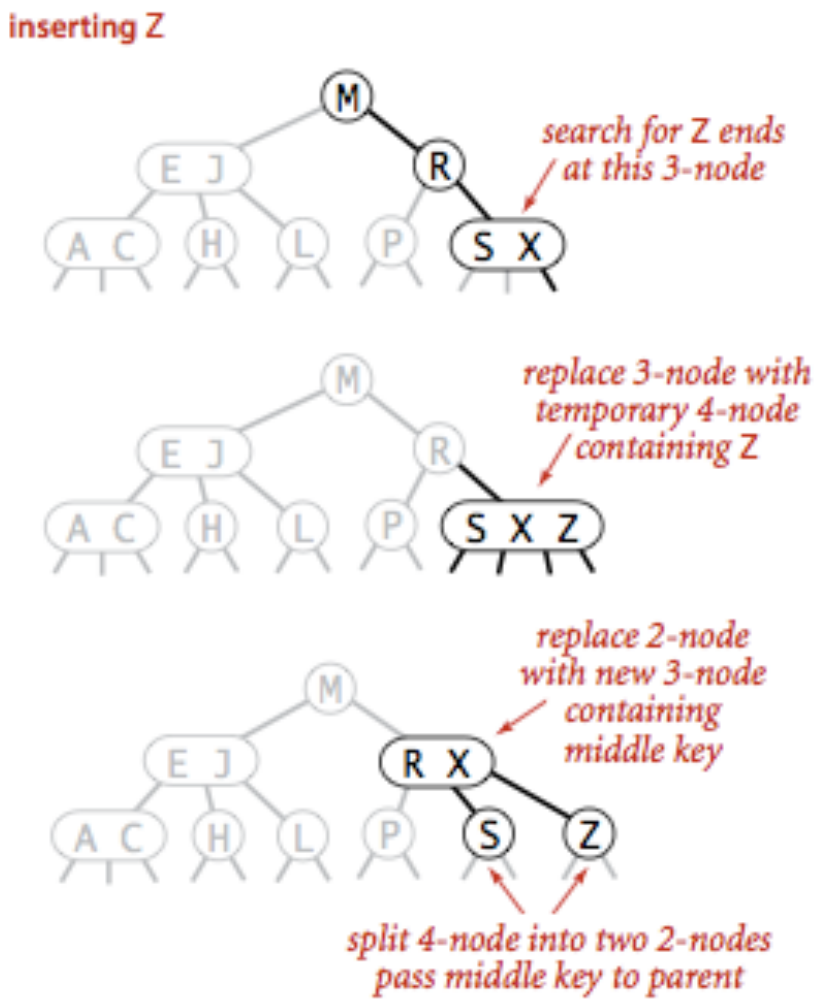
- Add new key to 3-node to create a temporary 4-node.
- Move middle key in 4-node into parent.
- Split 4-node into two 2-nodes. Middle key is elevated to parent.
- Height went up by 1.



Insert into a single 3-node

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

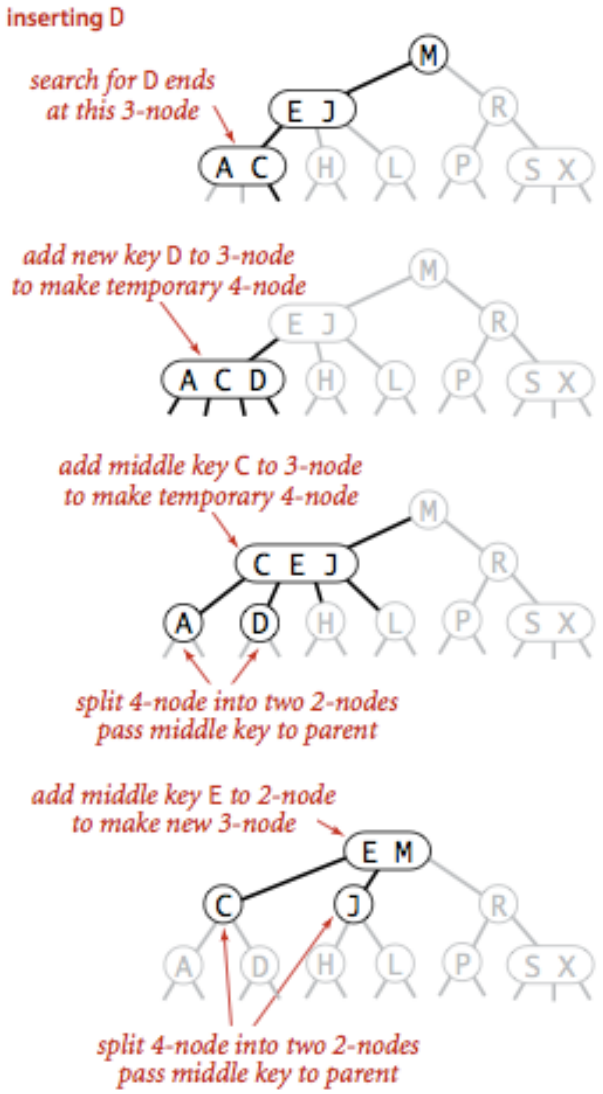
- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Replace 2-node parent with 3-node.



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

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

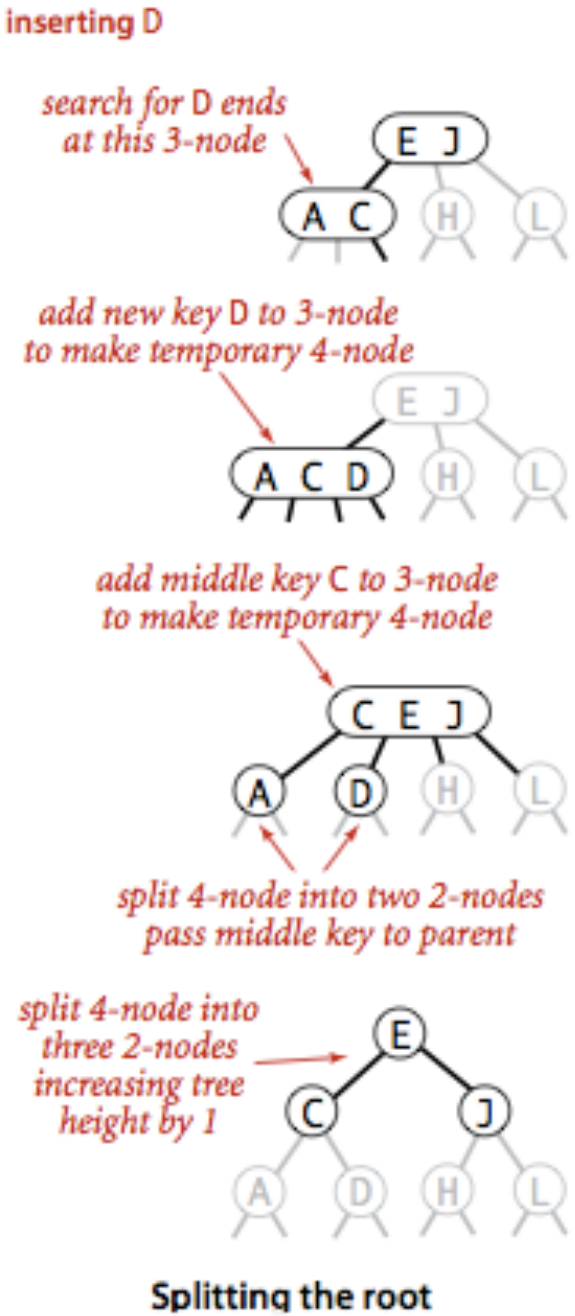
- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent creating a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Repeat up the tree, as necessary.



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.



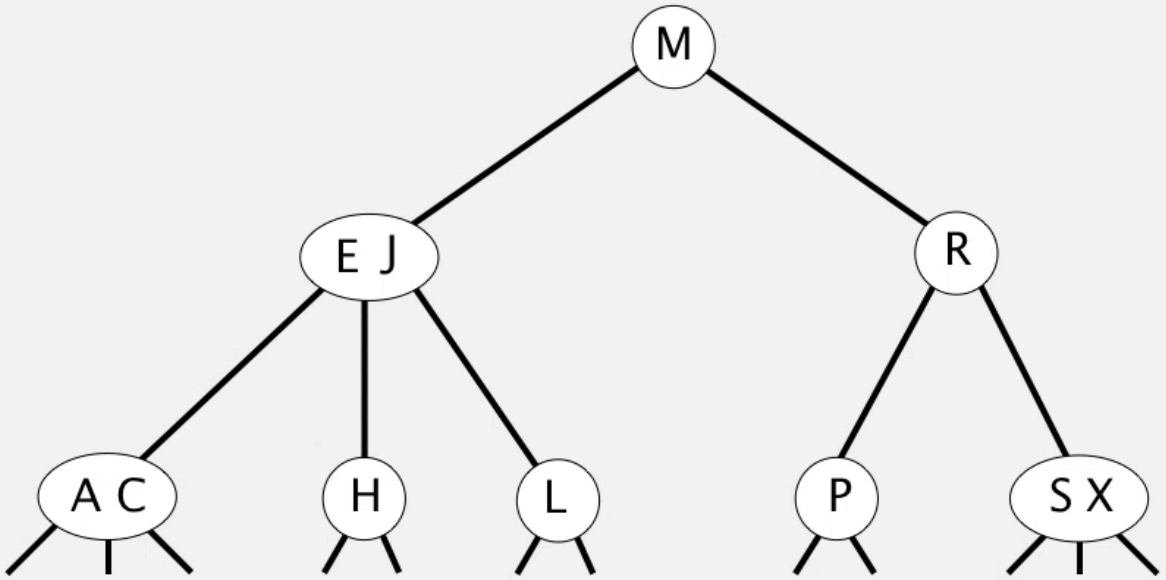
Splitting the root

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 24: 2-3 Search Trees

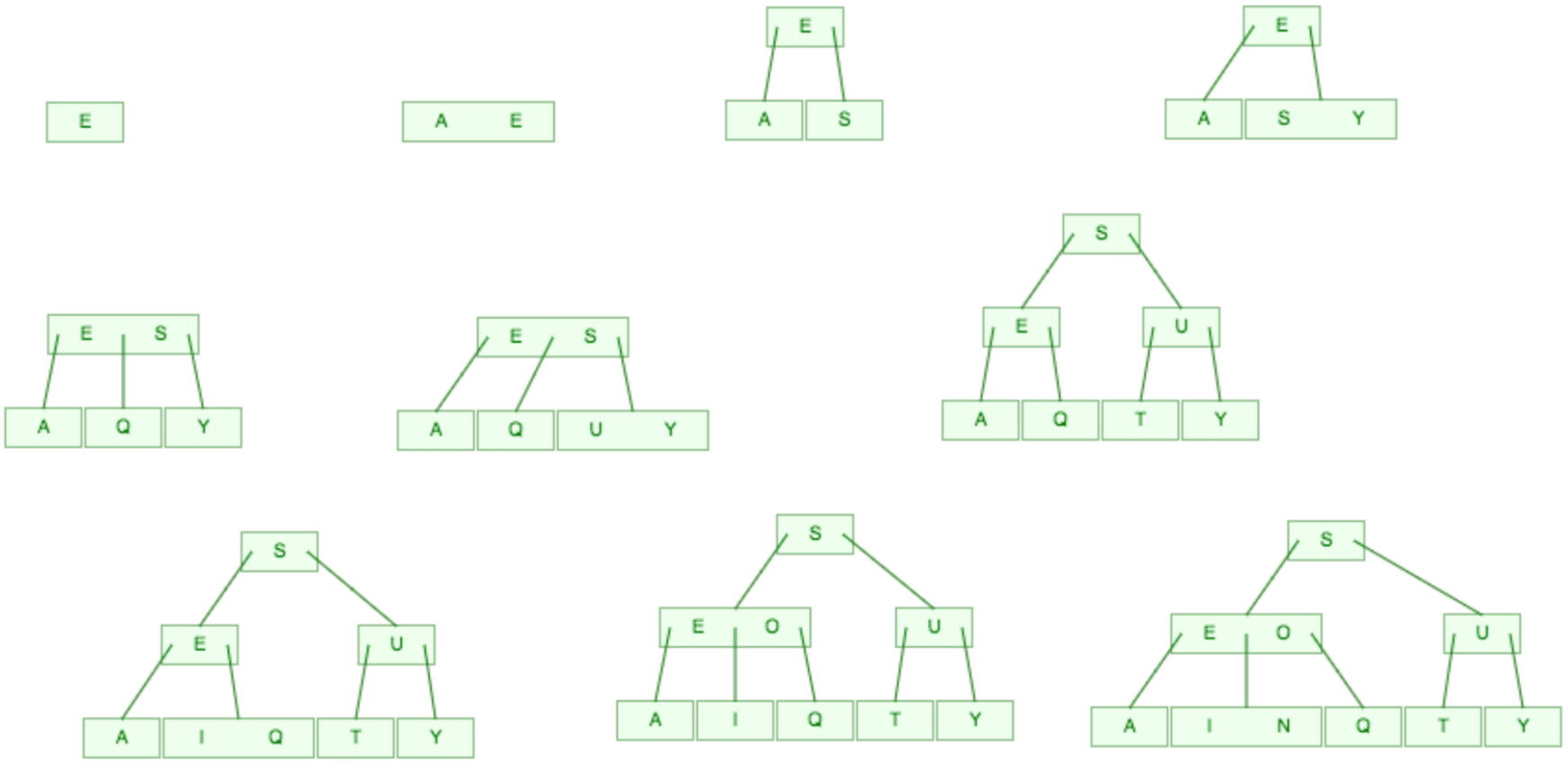
- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

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.

- E A S Y Q U T I O N

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.

- E A S Y Q U T I O N



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

Lecture 24: 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 18 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! We will see a much easier way.

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	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Readings:

- Textbook: Chapter 3.3 (Pages 424-431)
- Website:
  - https://algs4.cs.princeton.edu/33balanced/

Practice Problems:

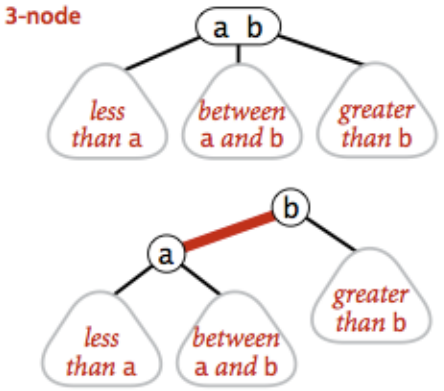
- 3.3.2-3.3.5

Lecture 20: Left-leaning Red-Black Trees

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

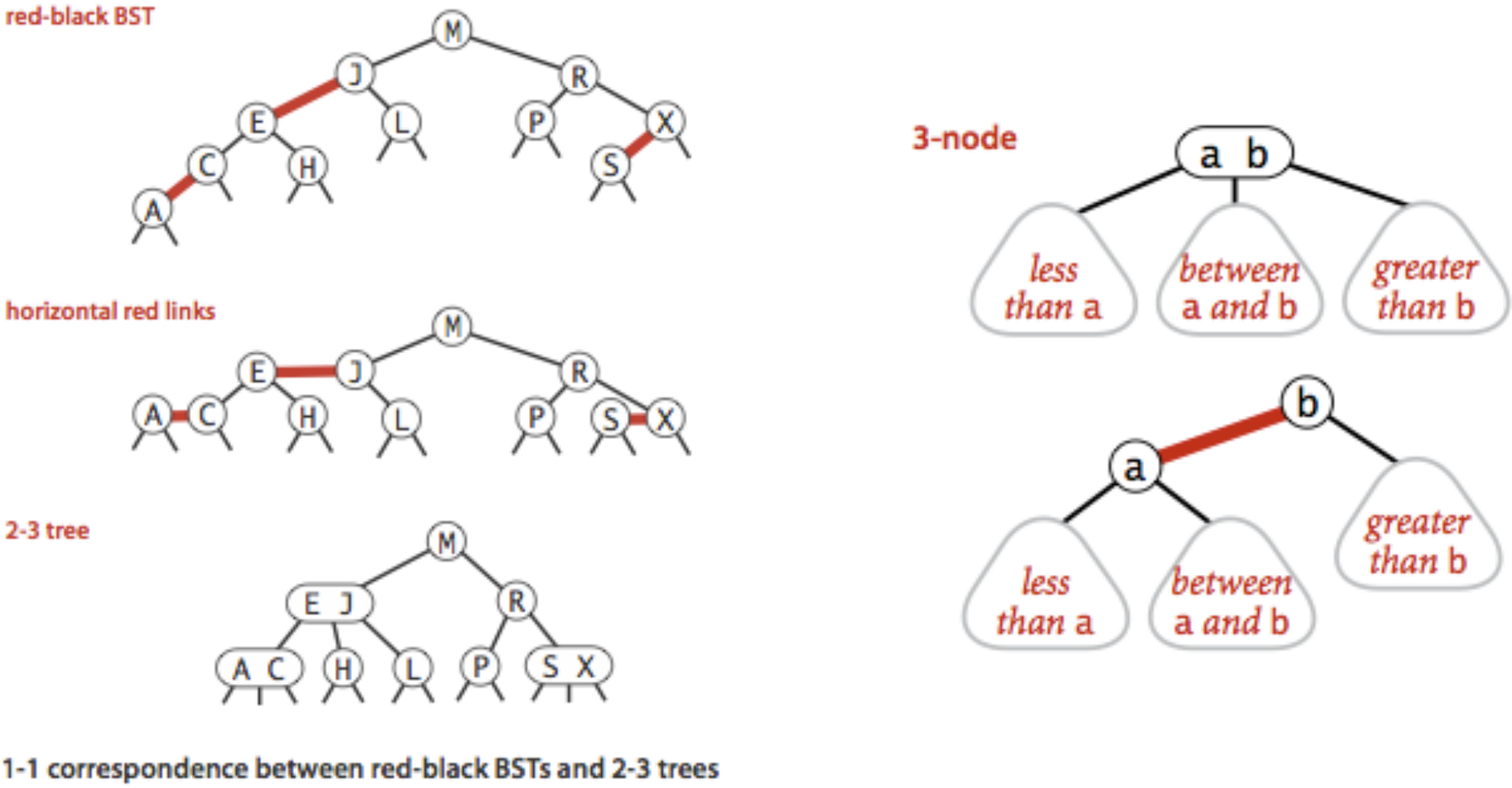
Some slides adopted from Algorithms 4th Edition or COS226

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



- Represent 2-3 tree as a BST
  - Start with standard BSTs which are made up of 2-nodes.
- Use "internal" left-leaning links as "glue" for 3-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.

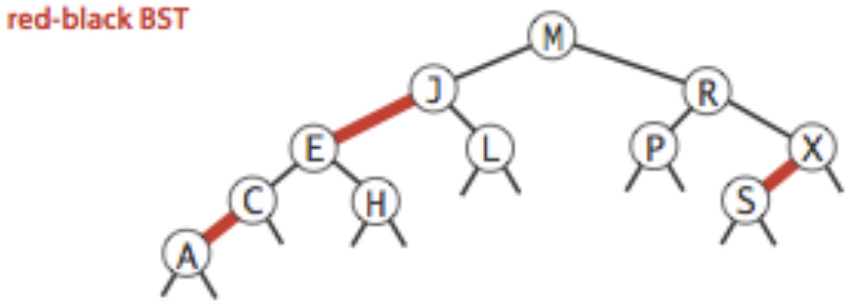
Left-leaning red-black BSTs correspond 1-1 with 2-3 trees



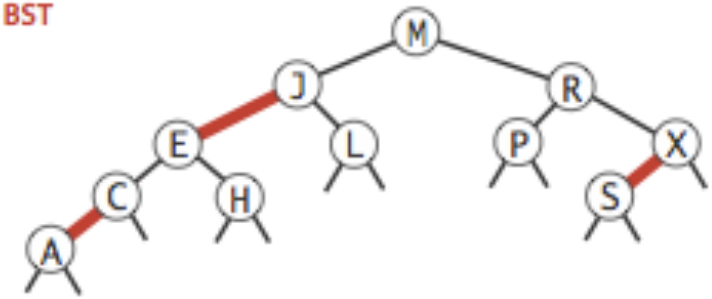
1-1 correspondence between red-black BSTs and 2-3 trees

Definition

- A left-leaning red-black tree is a BST such that:
  - No node has two red links connected to it.
    - Red links “glue” 2 2-nodes to make 1 3-node
  - Red link leans left.
  - Every path from root to leaves has the same number of black links (perfect black balance). Red links are “internal”.



red-black BST



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.
- Insertion needs to be updated

Representation

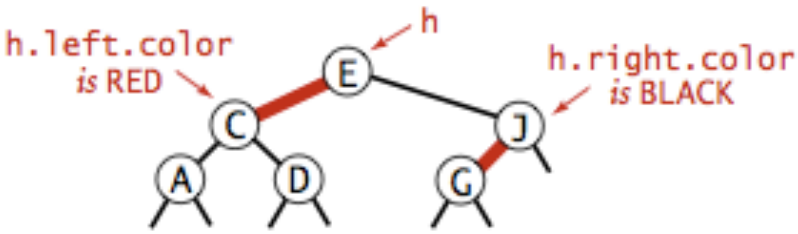
- There is no representation of the links in BST
  - Each node is pointed to by one node, its parent.
  - We can use this to encode the color of the links in nodes.
- True if the link from the parent is red and false if it is black. Null links are black.

```
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 righ
    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;
}
```

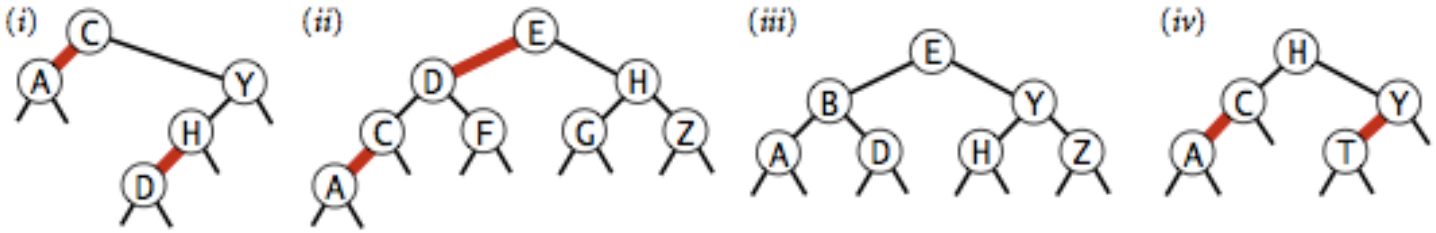


Story so far

- 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

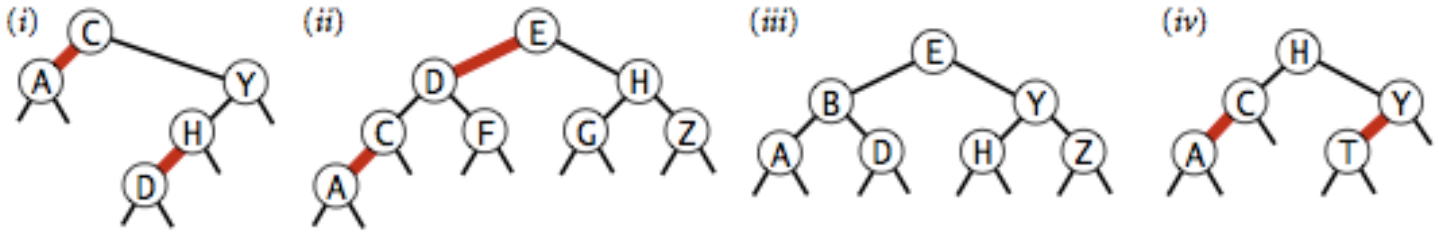
- Which of the following are legal LLRB trees?



- A left-leaning red-black tree is a BST such that:
  - No node has two red links connected to it.
    - Red links connect 2 2-nodes to make 1 3-node
  - Red link leans left.
  - Every path from root to leaves has the same number of black links (perfect black balance).

Answer

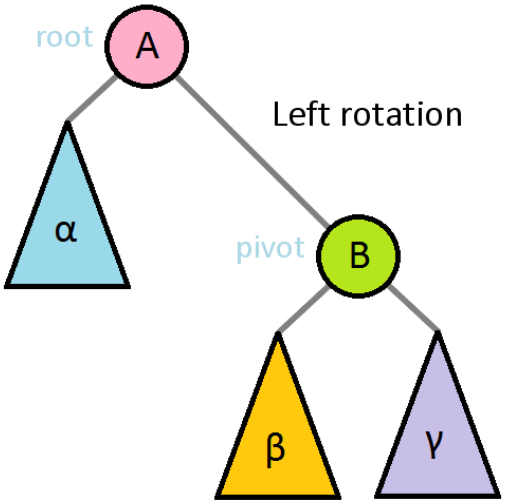
- Which of the following are legal LLRB trees?
- iii and iv
  - i is not balanced and ii is also not in symmetrical order

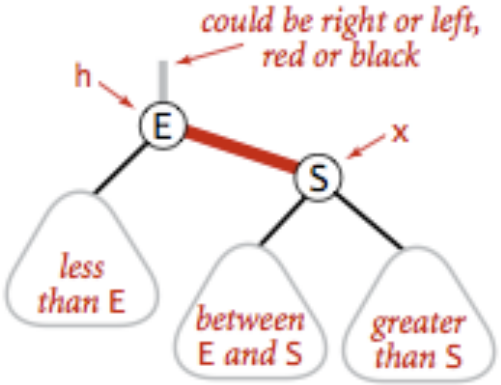


Lecture 28-29: Left-leaning Red-Black Trees

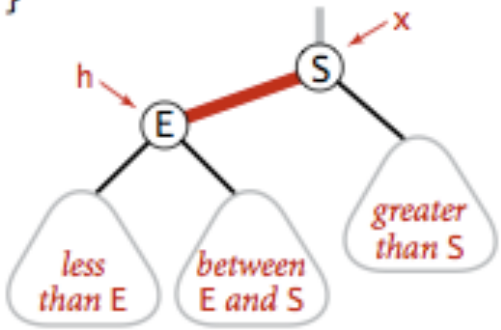
- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

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



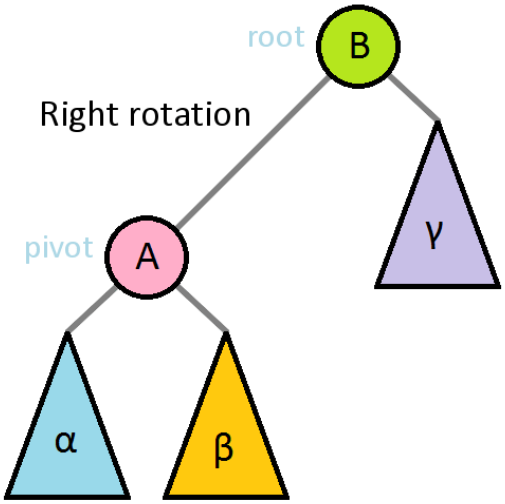


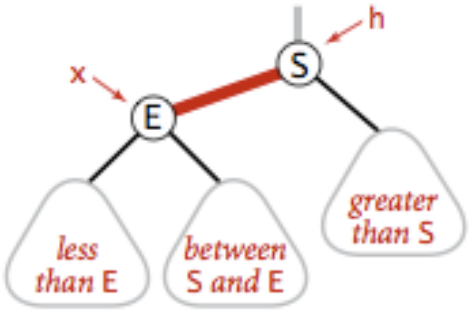

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



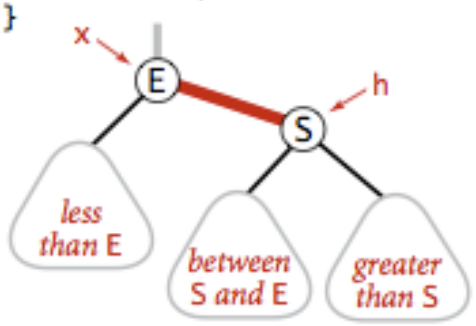
Left rotate (right link of h)

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



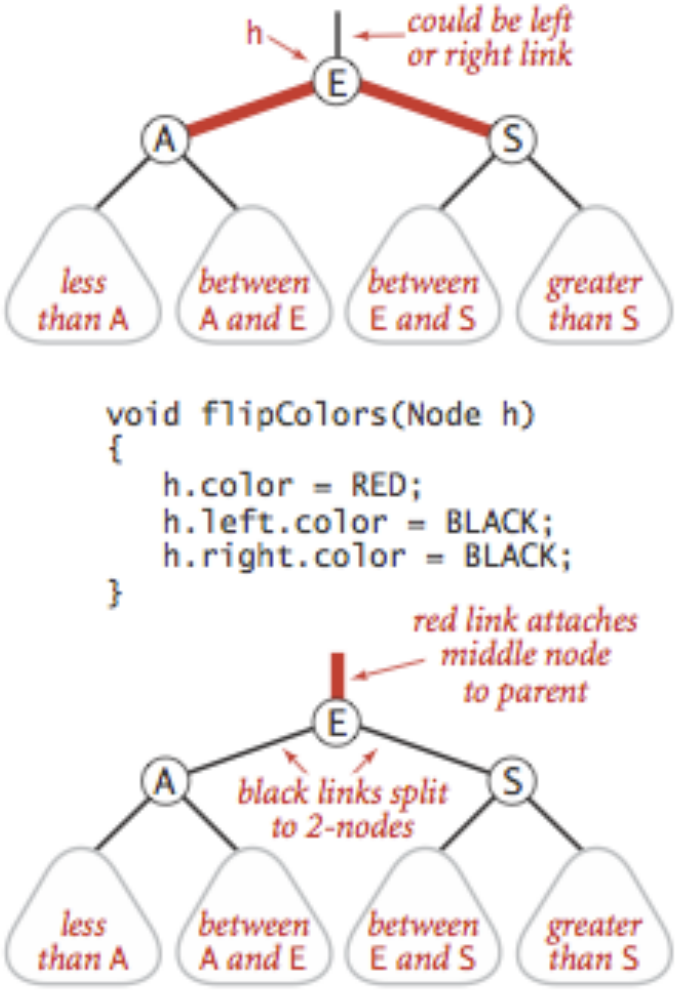



```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



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 28-29: 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

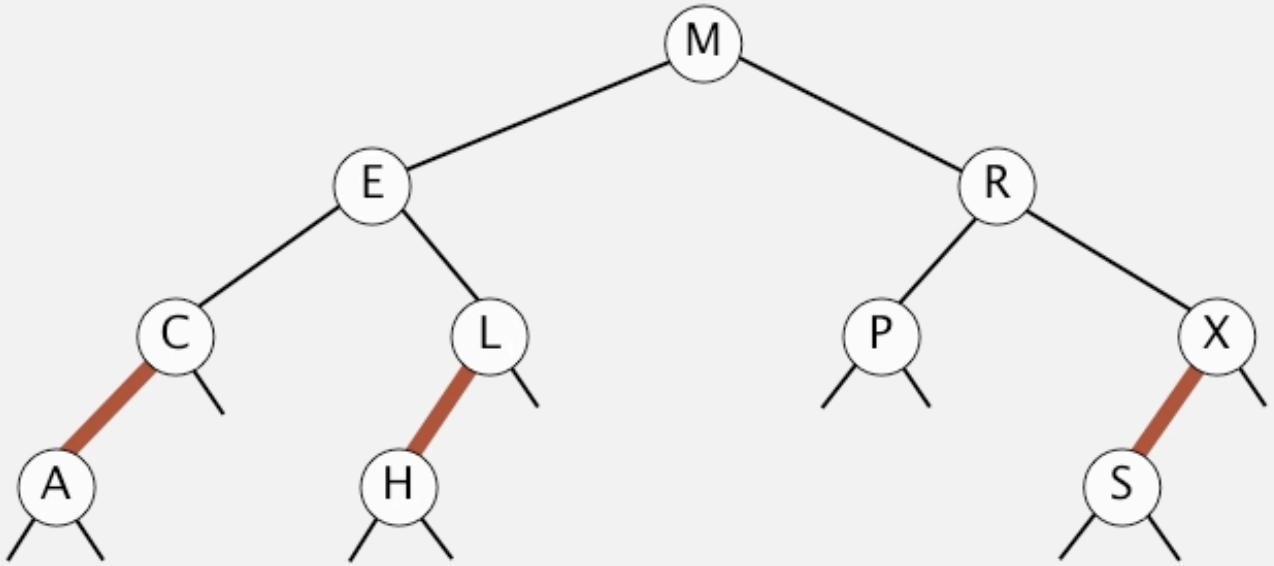
- During internal operations, maintain:
  - symmetric order.
  - perfect black balance.
- But we might violate color invariants. For example:
  - Right-leaning red link.
  - Two red children (temporary 4-node).
  - Left-left red (temporary 4-node).
  - Left-right red (temporary 4-node).
- To restore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

Insertion into a LLRB

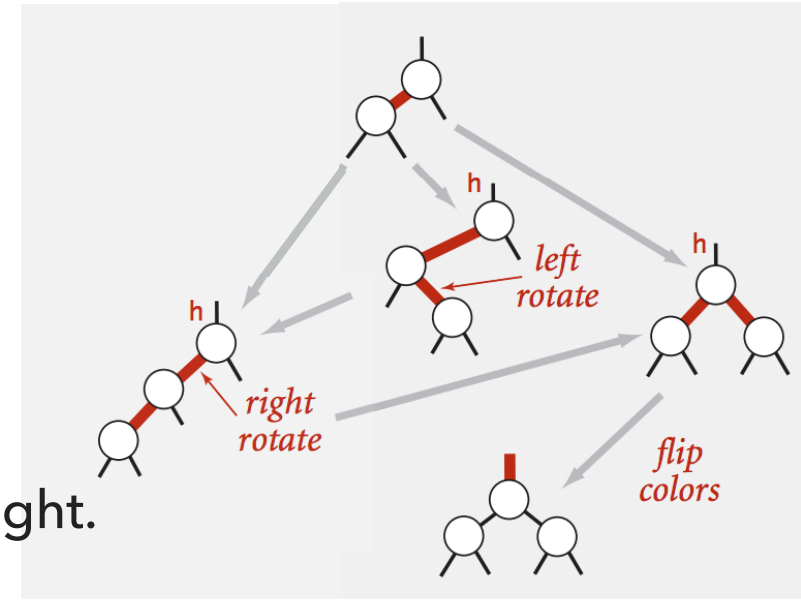
- Do standard BST insertion and **color the new link red.**
- Repeat until color invariants restored:
  - Both children red? Flip colors.
  - Right link red? Rotate left.
  - Two left reds in a row? Rotate right.

Red-black BST construction demo

red-black BST



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);  // Insert at bottom and color red

    int cmp = key.compareTo(h.key);                    // Compare as before to traverse tree
    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;

    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h); // Fix any right-leaning links
    if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);// 2 left red links
    if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);     // 4-node
    h.size = size(h.left) + size(h.right) + 1;

    return h;
}
```

Visualization of insertion into a LLRB tree

- 255 insertions in ascending order.

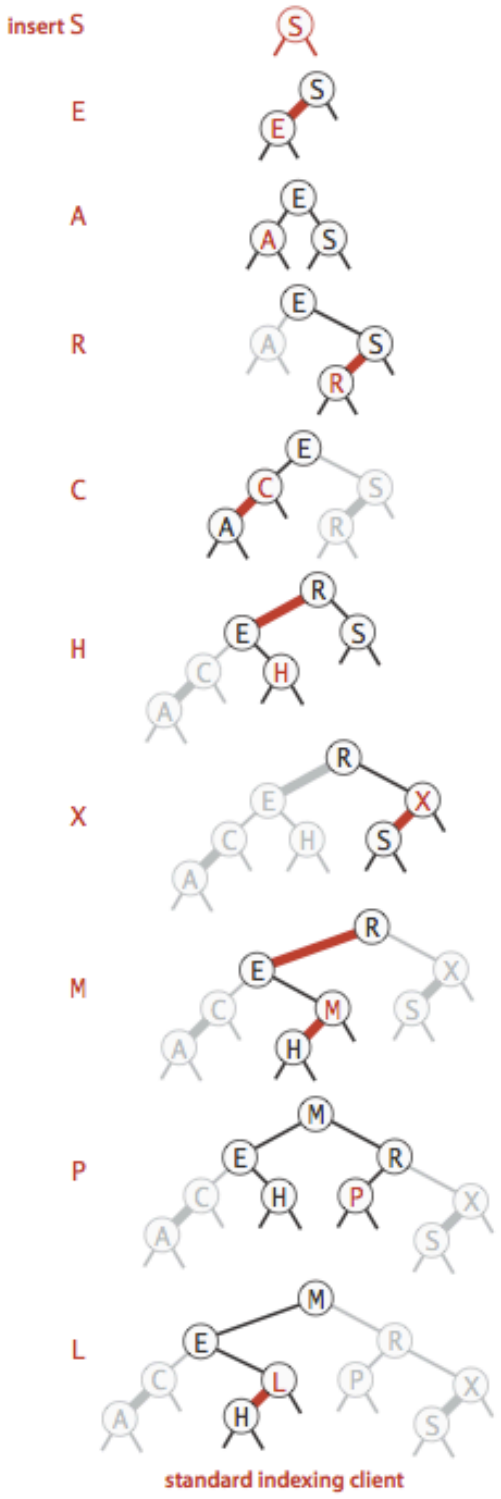
Visualization of insertion into a LLRB tree

- 255 insertions in descending order.

Visualization of insertion into a LLRB tree

- 255 insertions in random order.

Examples


insert S
E
A
R
C
H
X
M
P
L
standard indexing client
insert A
C
E
H
L
M
P
R
S
X
same keys in increasing order

Lecture 28-29: 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. Can you think of 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 symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
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	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

Summary for 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	$1.39 \log n$	$1.39 \log n$	?
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 28-29: Left-leaning Red-Black Trees

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

Red-black trees

- Why red-black? Invented at 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.
  - Telephone service outage due to exceeding height bound
  - Telephone company sues database provider

Balanced trees in the wild

- Red-black trees are widely used as system symbol tables.
  - 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 book for more.
  - B-trees are widely used for file systems and databases.

Lecture 28-29: Left-leaning Red-Black Trees

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

Readings:

- Textbook: Chapter 3.3 (Pages 432-447)
- Website:
  - https://algs4.cs.princeton.edu/33balanced/

Practice Problems:

- 3.3.9-3.3.22

Lecture 20: Midterm Topics

- Sorting
- Heaps/Priority Queues
- Dictionaries
- Misc
- Practice Problems
  - End of lecture slides
  - Go over in midterm review during lab

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Sorting

- Given an array of n items, sort them in non-descending order based on a comparable key.
- Cost model counts comparisons (calls to `less()`) and exchanges (calls to `exch()`) (or array accesses).
- Not in place: If linear extra memory is required.
- Stable: If duplicate elements stay in the same order that they appear in the input.
- Practice: https://visualgo.net/en/sorting (minus quick sort).

Dictionaries

- Binary search trees
  - Search
  - Insertion
  - Deletion
- 2-3 search trees
- Left-leaning Red Black search trees

Dictionaries

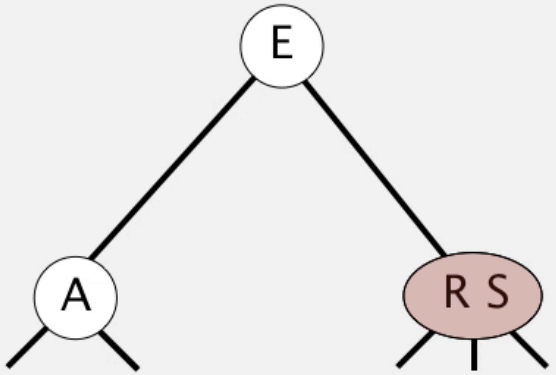
- Binary search trees
- 2-3 search trees

2-3 search trees

- Balanced (every path from root to leaf has same length) search tree that follow the symmetric order. Contain 2 nodes (one key and two children) or 3 nodes (two keys and three children).
- Search and insertion of keys (and values) is $O(\log n)$.
- A pain to implement.
- Practice: https://www.cs.usfca.edu/~galles/visualization/BTree.html (max-degree 3).

2-3 tree demo: construction

insert R



Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

Comparable Interface

- Interface with a single method that we need to implement:
  `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.
- Corresponds to natural ordering.

Comparator Interface

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate what kind of ordering we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that `compare(v, w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.

```
public static Comparator<ClassName> reverseComparator(){
   return (ClassName a, ClassName b)->{return -a.compareTo(b)};
 }
```

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

`Iterable<T>` Interface

- Interface with a single method that we need to implement: `Iterator<T> iterator()`
- Class becomes iterable, that is it can be traversed with a for-each loop.
-
```
for (String student: students){
        System.out.println(student);
}
```

`Iterator<T>` Interface

- Interface with two methods that we need to implement: `boolean hasNext()` and `T next()`.
- `hasNext()` checks whether there is any element we have not seen yet.
- `next()` returns the next available element.
- Always check if there are any available elements before returning the next one.
- Typically a comparable class, has an inner class that implements Iterator. Outer class's `iterator` method returns an instance of inner class.
- Can also be implemented in a standalone class where collection to iterate over is passed in the constructor.

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

BT traversals

- Pre-order: mark root visited, left subtree, right subtree.
- In-order: left subtree, mark root visited, right subtree.
- Post-order: left subtree, right subtree, mark root visited.
- Level-order: start at root, mark each node as visited level by level, from left to right.

Practice Problems

- Problem 1 - Sorting
- Problem 2 - Heaps
- Problem 3 - Tree traversals
- Problem 4 - Binary Trees
- Problem 5 - Binary Search Trees
- Problem 6 - Iterators

Problem 1 - Sorting

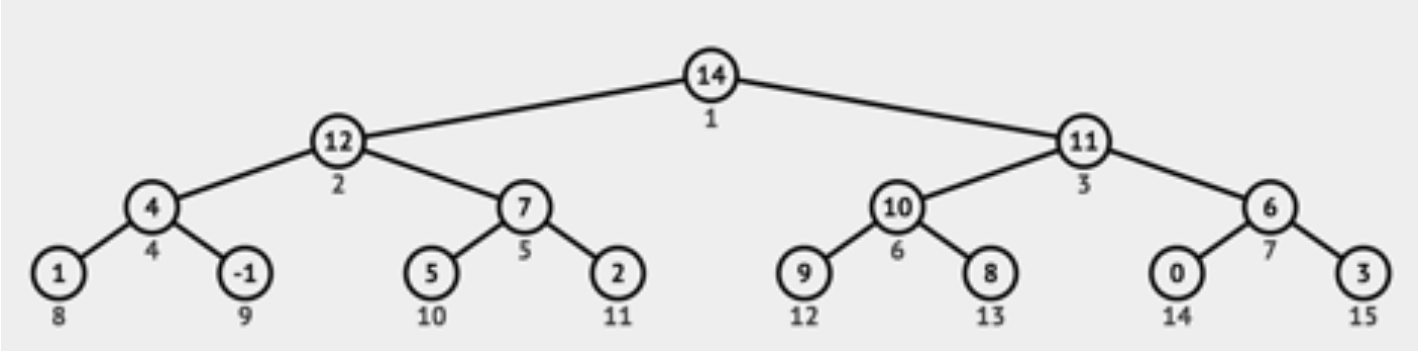
- In the next slide, you can find a table whose first row (last column 0) contains an array of 18 unsorted numbers between 1 and 50. The last row (last column 6) contains the numbers in sorted order. The other rows show the array in some intermediate state during one of these five sorting algorithms:
  - 1-Selection sort
  - 2-Insertion sort
  - 3-Mergesort
  - 4-Quicksort (no initial shuffling, one partition only)
  - 5-Heapsort
- Match each algorithm with the right row by writing its number (1-5) in the last column.

Problem 1 - Sorting

12	11	35	46	20	43	42	47	44	32	16	10	40	18	41	21	28	15	**0**
11	12	20	35	42	43	46	47	44	32	16	10	40	18	41	21	28	15	
10	11	12	46	20	43	42	47	44	32	16	35	40	18	41	21	28	15	
10	11	12	15	16	43	42	47	44	32	20	35	40	18	41	21	28	46	
43	32	42	28	20	40	41	21	15	11	16	10	35	18	12	44	46	47	
11	12	20	35	46	43	42	47	44	32	16	10	40	18	41	21	28	15	
10	11	12	15	16	18	20	21	28	32	35	40	41	42	43	44	46	47	**6**

Problem 2 - Heaps

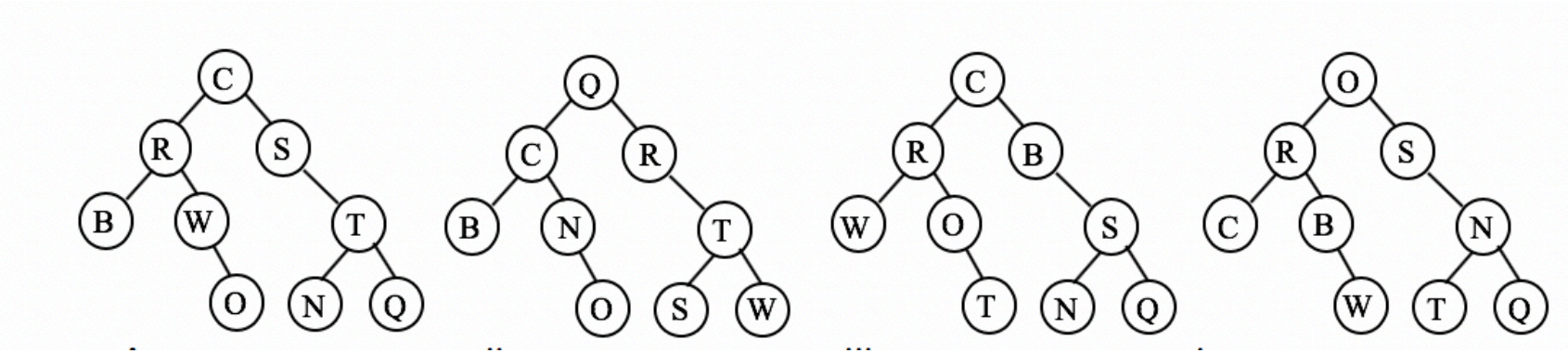
- Consider the following max-heap:



- Draw the heap after you insert key 13.
- Suppose you delete the maximum key from the original heap. Draw the heap after you delete 14.

Problem 3 - Tree Traversals

- Circle the correct binary tree(s) that would produce both of the following traversals:
  - Pre-order: C R B W O S T N Q
  - In-order: B R W O C S N T Q



Problem 4 - Binary Trees

- You are extending the functionality of the `BinaryTree` class that represents binary trees with the goal of counting the number of leaves. Remember that `BinaryTree` has a pointer to a `root` `Node` and the inner class `Node` has two pointers, `left` and `left` to the root nodes that correspond to its left and right subtrees.
- You are given the following public method:

```
public int sumLeafTree()
   return sumLeafTree(root);
}
```

- Please fill in the body of the following recursive method

```
private int sumLeafTree(Node x){…}
```

Problem 5 - Binary Search Trees

- You are extending the functionality of the BST class that represents binary search trees with the goal of counting the number of nodes whose keys fall within a given `[low, high]` range. That is you want to count how many nodes have keys that are equal or larger than `low` and equal or smaller than `high`. Remember that BST has a pointer to a `root` Node and the inner class Node has two pointers, `left` and `left` to the root nodes that correspond to its left and right subtrees and a `Comparable Key key` (please ignore the value).
- You are given the following public method:

```
public int countRange(Key low, Key high)
  return countRange(root, Key low, Key high);
}
```

- Please fill in the body of the following recursive method

```
private int countRange(Node x, Key low, Key high){…}
```

Problem 6 - Iterators

- A programmer discovers that they frequently need only the odd numbers in an arraylist of integers. As a result, they decided to write a class `OddIterator` that implements the `Iterator` interface. Please help them implement the constructor and the `hasNext()` and `next()` methods so that they can retrieve the odd values, one at a time. For example, if the arraylist contains the elements [7, 4, 1, 3, 0], the iterator should return the values 7, 1, and 3. You are given the following public class:

```
public class OddIterator implements Iterator<Integer> {

// The array whose odd values are to be enumerated
private ArrayList<Integer> myArrayList;

//any other instance variables you might need

//An iterator over the odd values of myArrayList
public OddIterator(ArrayList<Integer> myArrayList){…}

//runs in O(n) time
public boolean hasNext(){…}

//runs in O(1) time
public Integer next(){…}
}
```