Tree

1 Abstract Data Type(ADT)

- Stacks: Structures that support last-in first-out retrieval of elements
  - `push(int x)`: puts x on the top of the stack
  - `int pop()`: takes the element on the top of the stack
- Lists

  : an ordered set of elements

  - `add(int i)`: adds an element
  - `int get(int i)`: gets element at index i
- Sets

  : an unordered set of unique elements (no repeats)

  - `add(int i)`: adds an element
  - `contains(int i)`: returns a boolean for whether or not the set contains the value
- Maps

  : set of key/value pairs

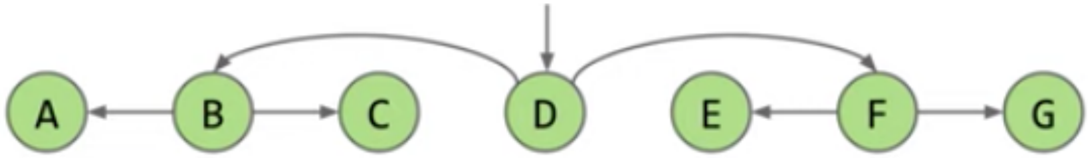
  - `put(K key, V value)`: puts a key value pair into the map
  - `V get(K key)`: gets the value corresponding to the key

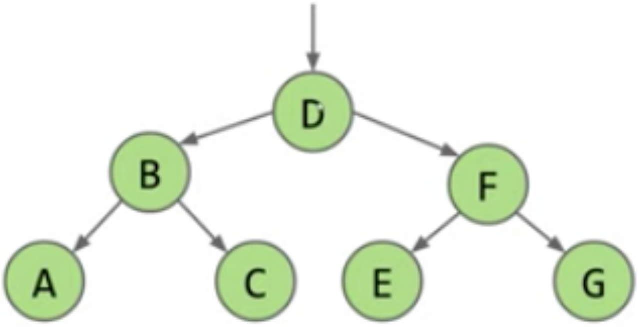
2. 自下而上与自上而下

- 自下而上：
  - 先递归再操作
  - 能划分成一个个小的subtree
    - root.left = xxxxx, root.right = xxxx 然后return root
    - 考虑函数的返回值
  - 类似DP
    - dfs(left)的结果，dfs(right)的结果，当前root.value --> 新的结果
  - 遍历方向是postorder
- 自上而下
  - 先操作再递归
  - 比如从root到leaf的路径
  - 遍历方向是preorder

3. Binary Search Tree知识点

3.1 BST定义





Tree

- 数的组成：node + edges
- root
- parent / child
- A node with no child is called a **leaf**

BST

- 左边的所有值 < node.value < 右边所有的值
- 没有重复的key
- 若p<q,则q<p
- 若p<q 且 q<r，则p<r

```
private class BST<Key> {
    private Key key;
    private BST left;
    private BST right;

    public BST(Key key, BST left, BST Right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    public BST(Key key) {
        this.key = key;
    }
}
```

3.2 BSTs as Sets and Maps

- Java provides Map, Set, List interfaces, along with several implementations.
- two ways to implement a Set (or Map):
  - ArraySet: Θ(*N*) operations in the worst case.
    - S.contains("Tokyo")) 判断是否存在
    - ArraySet是通过Array实现的
  - BST: Θ(log*N*) operations if tree is balanced.

4. BST操作

4.1 Search

- if searchKey equals T.key, return
  - if searchKey < T.key, search T.left
  - if searchKey > T.key, search T.right

- Time complexity log(n)
  - the height of the tree is log(n)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

//递归
public TreeNode searchBST(TreeNode root, int val) {
    if (root == null)
        return null;
    if (root.val > val) {
        return searchBST(root.left, val);
    } else if (root.val < val) {
        return searchBST(root.right, val);
    } else {
        return root;
```

```
    }
}

//迭代
public TreeNode searchBST(TreeNode root, int val) {
        if (root == null)
            return null;

        while (root != null) {
            if (root.val == val) {
                return root;
            } else if (root.val > val) {
                root = root.left;
            } else {
                root = root.right;
            }
        }
        // 易忘记
        return null;
 }
```

4.2 Insert

- always insert at a leaf node!!! 也就是说，每一次插入在最底层的node
- Steps
  - search in the tree for the node
  - if find it, do nothing
  - if not
    - be at a leaf node
    - add the new element to either the left or right of the leaf
- Time Complecity:
- return insert之后的new root

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
```

```
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */

// 递归
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }

        if (root.val > val) {
            root.left = insertIntoBST(root.left, val);
        } else {
            root.right = insertIntoBST(root.right, val);
        }

        return root;
    }
}

// 迭代
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }

        TreeNode res = root;
        while (root != null) {
            if (root.val > val) {
                if (root.left == null) {
                    root.left = new TreeNode(val);
                    return res;
                } else {
                    root = root.left;
                }

            } else {
                if (root.right == null) {
                    root.right = new TreeNode(val);
                    return res;
                } else {
                    root = root.right;
                }
            }
        }
```

```
        return res;
    }
}
```

```
```

```
// 老师给的hit
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.label()))
    T.left = insert(T.left, ik);
  else if (ik > T.label())
    T.right = insert(T.right, ik);  //Always set, even if nothing changes!!不要判断是否有左右
  return T;
}


// bad habit 老师说这是菜鸟的代码
// 下面的两个情况已经被包涵在了base case里
// Avoid "arms length base cases". Don't check if left or right is null!!!
if (root.left == null) {
  root = root.right;
}
else if (root.right == null) {
  root = root.left;
```

- 让新插入的node变成新的root，原本root上比他小的点放到左边，大的放到右边

??? 需要新的思考

4.3 Delet

3 Cases

- Deletion key has no children
  - 直接删
- Deletion key has one child
  - 被删点的左节点不存在，直接提升右节点
  - 被删点的右节点不存在，直接提升左节点
- Deletion key has two children
  - 需要找一个新的root，用被删节点的前驱点(predecessor)或者后驱点(successor)代替

        - 前驱点：left tree里面的最right的节点
        - 后驱点：right tree里面的最left的节点
    - predecessor 和successor永远不会有两个children
- 叫作Hibbard deletion
- 先删后遍历，先遍历后删都可以

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) {
            return null;
        }

        if (root.val > key) {
            root.left = deleteNode(root.left, key);
        }
        if (root.val < key) {
            root.right = deleteNode(root.right, key);
        }

        if (root.val == key) {
            if (root.left == null && root.right == null) {
                root = null;
            }
            else if (root.left == null) {
                root = root.right;
            }
            else if (root.right == null) {
                root = root.left;
            }
            else {
                root.val = findPre(root);
                root.left = deleteNode(root.left, root.val);
            }
```

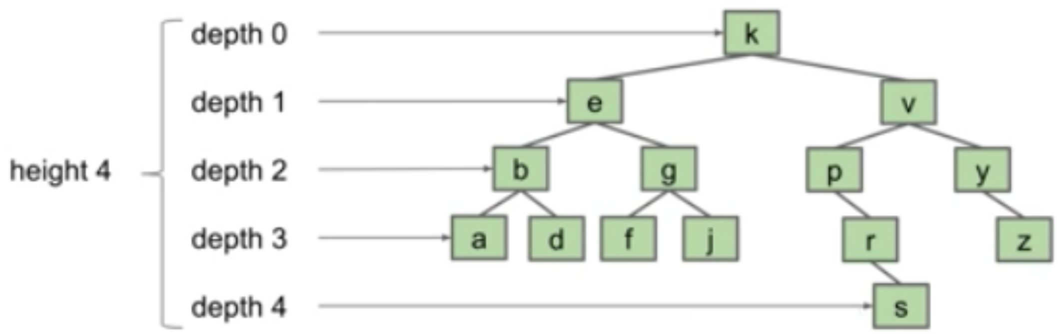
```
        }
        return root;
    }
    private int findPre(TreeNode root) {
        root = root.left;
        while (root.right != null) {
            root = root.right;
        }
        return root.val;
    }

    private int successor(TreeNode root){
        root = root.right;
        while(root.left != null){
            root = root.left;
        }
        return root.val;
    }
}
```

5. Balanced Search Trees

- Trees range from best-cae "**bushy**" to worst-case "**spindly**"(一层里面只有一个node)
    - **Worst case:** Θ(*N*)

      Best-case: Θ(log*N*) (where N*N* is number of nodes in the tree)

5.1 BST performance



- **depth**: the number of links between a node and the root.
    - 该node到root的距离
    - depth(g) = 2
- **height**: the lowest depth of a tree.

- **average depth**: average of the total depths in the tree.
  - (0 x 1 + 1 x 2 + 3 x 6 + 4 x 1) / (1 + 2 + 4 + 6 + 1) = 2.35
  - calculate this by taking$N\sum_{i=0}^{D} d_i n_i$ where di is depth and ni is number of nodes at that depth.
  - 可以计算出average-case runtime
    - Average case is 3.35 comparisons (average depth + 1 因为0-based to 1-based)
- 最合适的高度是balanced BST

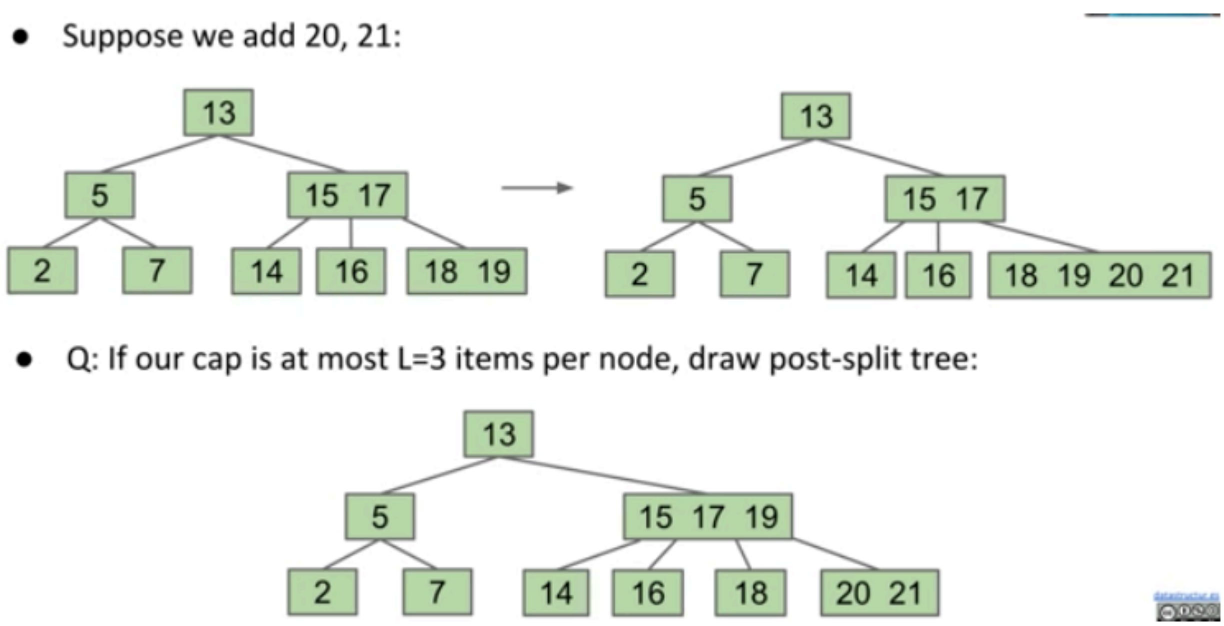
5.2 随机二分搜索树

- 随机二分搜索树，是利用随机数或者概率分布来到达平衡条件
- 构造树的时候，random insert，是**bushy**的，是很接近balanced BST的height（推导省略）
  - 此时的average depth和height都可以认为是$\Theta(\log N)$
- 没理解？？？
- 删除节点也可以如此

6. B-Trees

B-Trees are a modification of the binary search tree that avoids $\Theta(N)$ worst case.

- 在之前的BST操作里面，insert node的时候，都是加在leaf上，这样会导致height越来越大
- 通过Overstuffing来实现防止不平衡



- 每一个node里面可以放复数个树，16在15和17之间，所以放在下一层的15，17之间
- B-trees of order L = 3 -> called 2-3-4 tree or 2-4 tree（上图的树）3阶树
  - 该节点里面最多可以放三个数，然后该节点最多有四个孩子

- B-trees of order L = 2 -> called 2-3 tree 2阶树
    - 该节点里面最多可以放两个数，然后该节点最多有三个孩子
- B-tree一定会是平衡的，当该节点是一个数的时候，一定要有两个children，当该节点有3个数的时候，一定要有4个childre

不变量

- All leaves must be the same distance from the source.
    - 所有最下面一层的node到root的距离一定是相同的
- A non-leaf node with k items must have exactly k+1 children.
    - 除了leaf node，该节点上有k个items，就一定要有k+1个children

Time complexity

- Resulting tree has perfect balance. Runtime for operations is O(*log*N).

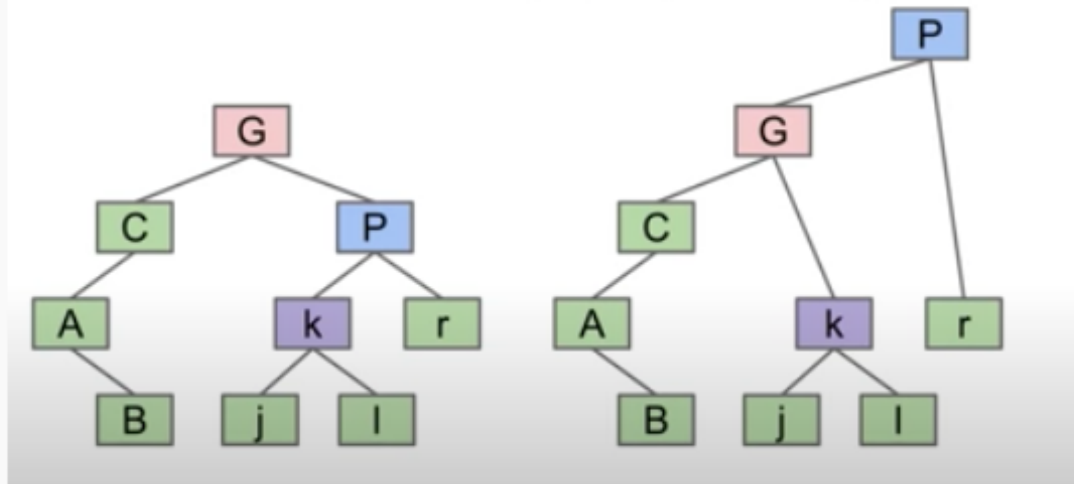
难以实现

```
public void put(Key key, Value val) {
   Node x = root;
   while (x.getTheCorrectChildKey(key) != null) {
      x = x.getTheCorrectChildKey();
      if (x.is4Node()) { x.split(); }
   }
   if (x.is2Node()) { x.make3Node(key, val); }
   if (x.is3Node()) { x.make4Node(key, val); }
}
```

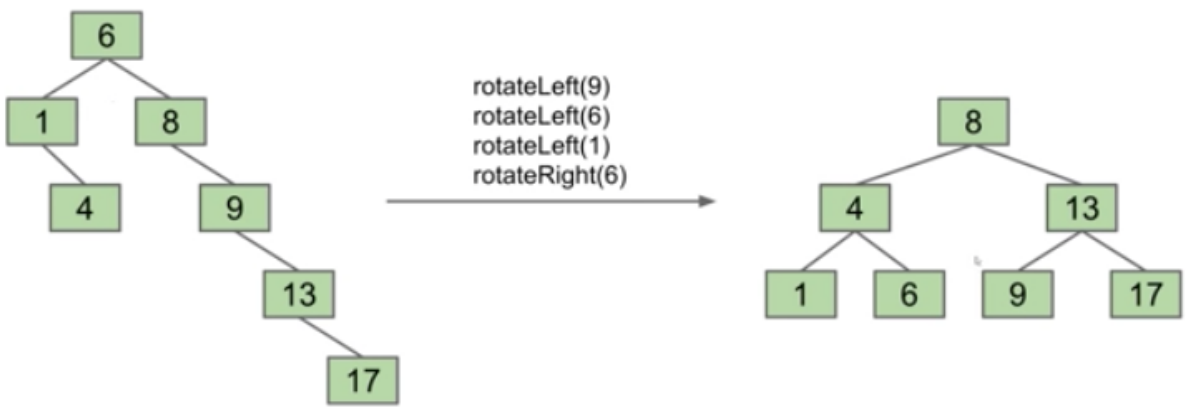
7. Rotating Trees

create a balanced tree

- 可构成BST的个数
    - 1，2，3 --> 5种情况
    - n个node可构成的BST种类是：卡塔兰数
        - 1, 1, 2, 5, 14, 42
    - 用rotation可以互相转换任意形状的BST
- rotation的定义

- G是任意一个node
- rotateLeft(G): Let x be the right child of G. Make G the new left child of x.
  - G -> left
  - G变成某一个node的左孩子，某一个node变成G的右孩子
  - 也就是说，原本G的右孩子变成root，原本G的右孩子的左孩子变成G的右孩子
  - 如下例，P变成新的root，G变成P的左孩子，k变成G的右孩子（不然的话，P就有3个孩子了）
  - 

- rotateRight(G): Let x be the left child of G. Make G the new right child of x.
  - G变成某一个node的右孩子，某一个node变成G的左孩子
  - 也就是说，原本G的左孩子变成root，原本G的左孩子的右孩子变成G的左孩子

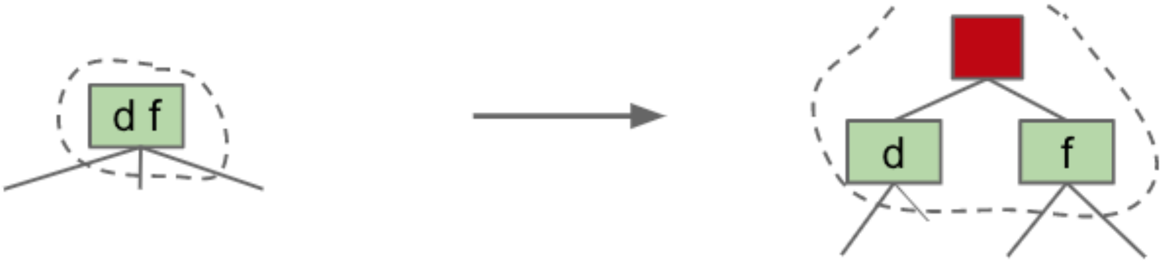



```
private Node rotateRight(Node h) {
    // assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    return x;
}

// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
    // assert (h != null) && isRed(h.right);
```

```java
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    return x;
}
```

8. Red-Black Trees

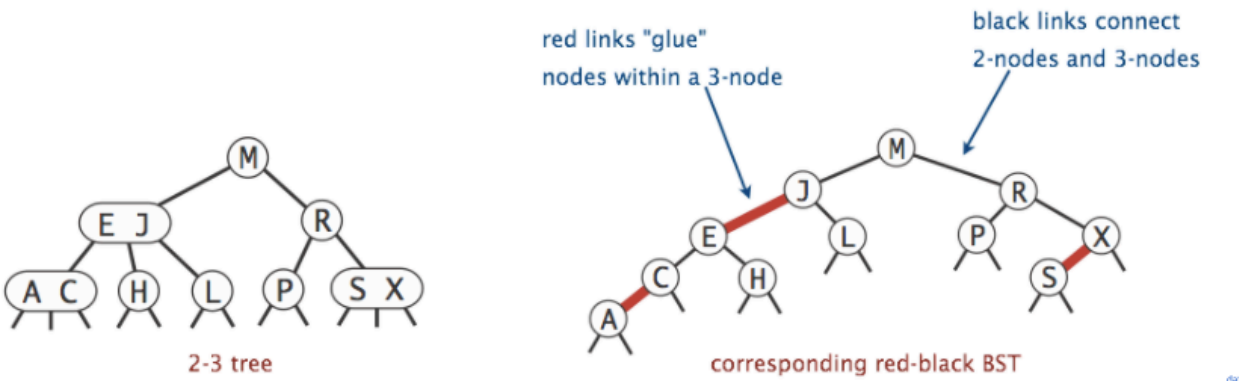
- 红黑树，一种自平衡二叉查找树，起源“对称二叉B树”，因为起源是2-3 tree
- 可以在O(logN)时间内完成查找，插入和删除，n是树种元素的数目
- java.util.TreeSet就是由此数据结构储存的， but not left learning
- 一个节点上有两个items的情况，利用一个dummy “glue” node来把它们拆分开



8.1 Left-Leaning Red Black Binary Search Tree(LLRB)

A BST with left glue links that represents a 2-3 tree is often called a “Left Leaning Red Black Binary Search Tree” or LLRB.

- LLRBs are normal BSTs!
- There is a 1-1 correspondence between an LLRB and an equivalent 2-3 tree.
- The red is just a convenient fiction. Red links don’t “do” anything special.



- 小的那个item放到右边下面

- black links connect 2-nodes and 3-nodes
- red links "glue" nodes within a 3-node

LLRB Height

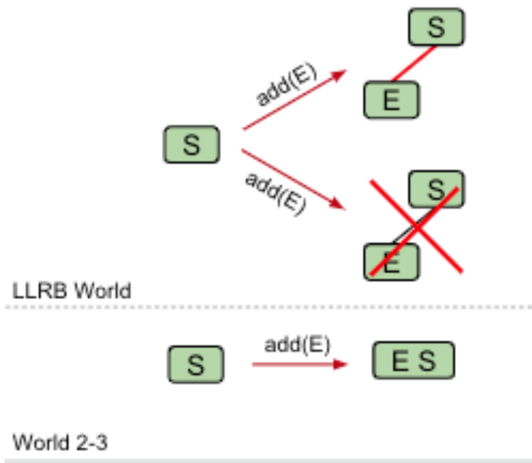
- 假设有一个2-3 tree of height H
- the maximum height of the corresponding LLRB: H(black) + H + 1(red) = 2H + 1

Properties

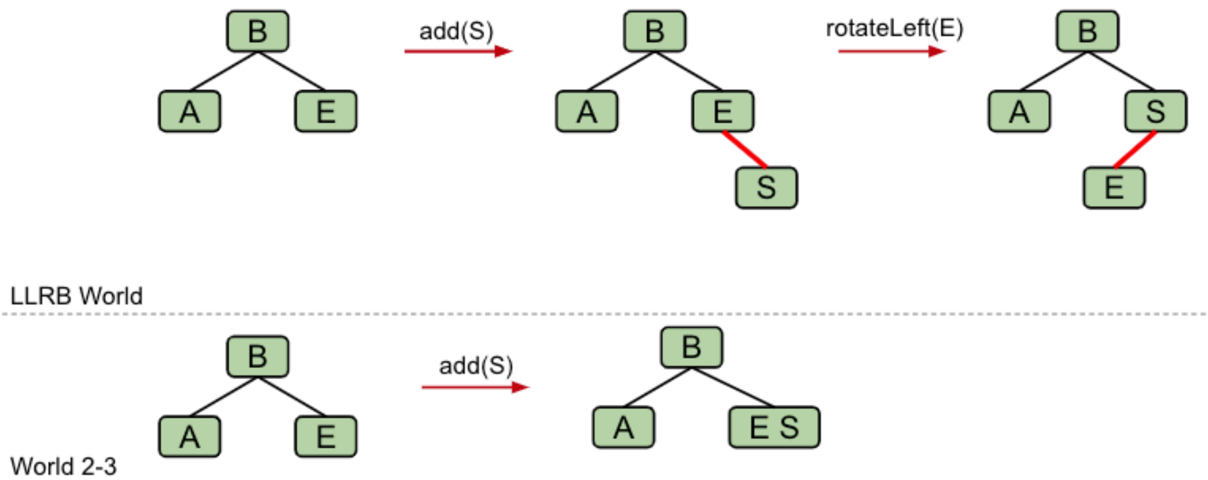
- 一个2-3tree对应一个LLRB, 1-1 mapping
- 2-3trees的关联LLRB，没有node有两个red links
- no red right-link
- 每一个path（从root到leaf）有相同数量的black links（因为2-3trees have the same number of links to every leaf)
- 高度不能超过2倍的对应2-3tree（LLRB：2H+1， 2-3 tree：H）

8.2 Insertion

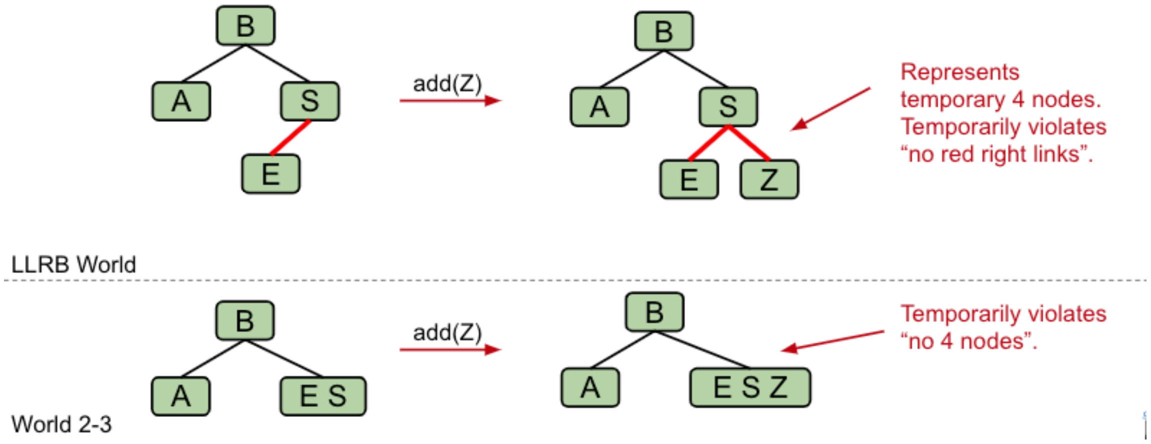
1. Should we use a red or black link when inserting?
    - Use red! 因为在2-3tree，new values always added to a leaf node(at first)



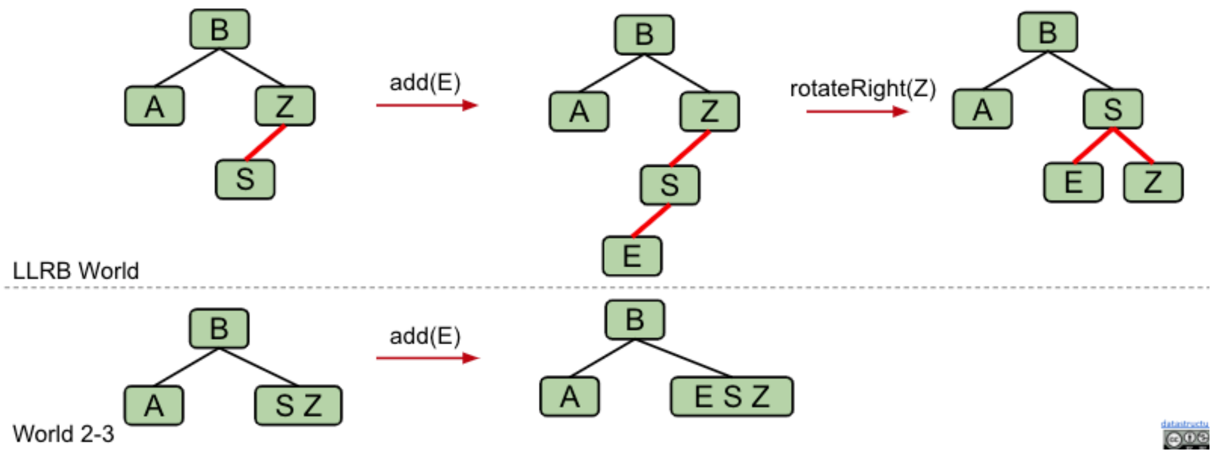
2. insertion on the right, 因为我们使用的是LLRB，不存在right red link，如果insert on the right, 则需要rotation



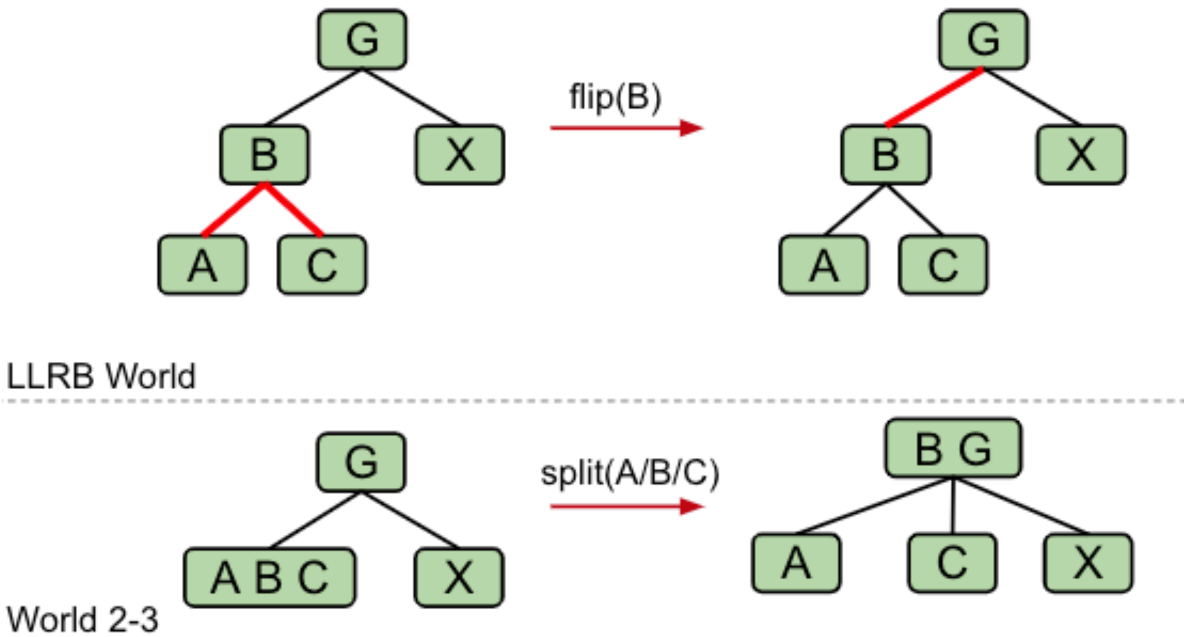
3. 原本该node上已经有两个items，再加一个就是三个的情况（暂时的）：
    - 会产生red right link



    - 如果是左边连续两个red link，则需要进行一次rotation，变成左右各一个red link



4. 把3个items进行split --> color flip



总结

1. 加到left，是red link，没有连续两个red link

2. 如果出现right link或者连续两个red link，通过rotation或者color flip来维护
    - right red link -> 加入进去的上一个节点需要rotate left
    - two consecutive left links --> 加入进去的上上一个节点需要rotate left
    - red left and red right --> color flip（red left 和red right所连接的node与该node的上一个node之间变成red link)

- 例子
    - 按 7 6 5 4 3 2 1的顺序插入
    - https://docs.google.com/presentation/d/1jgOgvx8tyu_LQ5Y21k4wYLffwp84putW8iD7_EerQmI/edit#slide=id.g463de7561_042

abstracted code for insertion into a LLRB

```
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              { h.val   = val;                    }

    // just 3 lines needed to balance
    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;
}
```

Runtime

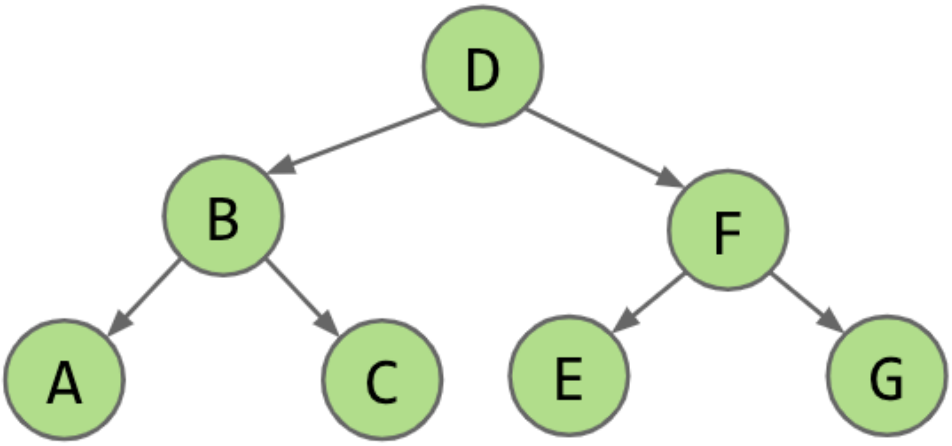
height是log2N +

无论什么操作都是都是logN

为什么time complexity是logN? ? ?

- insert is O(logN)
    - O(logN) to add the new node
    - O(logN) rotation and color flip operations per insert

9. Tree Traversals



Lever Order

- Visit top-to-bottom, left-to-right
- D-B-F-A-C-E-G

Depth First Traversals

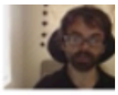
- 3 types: preorder, inorder, postorder
- preorder: key-left-right; D-B-A-C-F-E-G
  - 自上而下，先操作再遍历
  - 技巧：一侧一侧遍历，zigzag形
- inorder: left-key-right; A-B-C-D-E-F-G
  - 如果tree是BST的话，就是从小到大排序
  - 直通到最右边，然后root，再right
- postorder：left-right-key; A-C-B-E-G-F-D
  - 自下而上，先遍历后操作
  - 当作一个个的subtree来看，左边完了，右边，最后是root
  - 类似DP
    - dfs(left)的结果，dfs(right)的结果，当前root.value --> 新的结果

```
public class Traversal {
    private class BSTNode {
        int key;
        BSTNode left;
        BSTNode right;

        BSTNode() {}
        BSTNode(int key) { this.key = key; }
        BSTNode(int key, BSTNode left, BSTNode right) {
```

```java
            this.key = key;
            this.left = left;
            this.right = right;
        }

    }

    // preorder
    public void preOrder(BSTNode x) {
        if (x == null) return;
        System.out.println(x.key);
        preOrder(x.left);
        preOrder(x.right);
    }

    // inorder
    public void inOrder(BSTNode x) {
        if (x == null) return;
        inOrder(x.left);
        System.out.println(x.key);
        inOrder(x.right);
    }

    // postorder
    public void postOrder(BSTNode x) {
      if (x == null) return;
        postOrder(x.left);
        postOrder(x.right);
        System.out.println(x.key);
    }
}
```

Point

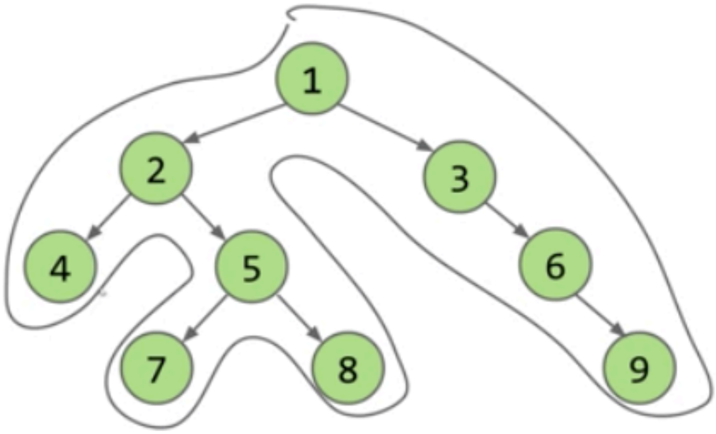
A Useful Visual Trick (for Humans, Not Algorithms)



- Preorder traversal: We trace a path around the graph, from the top going counter-clockwise. “Visit” every time we pass the LEFT of a node.
- Inorder traversal: “Visit” when you cross the bottom of a node.
- Postorder traversal: “Visit” when you cross the right a node.

Example: Post-Order Traversal

- 4 7 8 5 2 9 6 3 1



- preorder
  - 1-2-4-5-7-8-3-6-9
- inorder
  - 4-2-7-5-8-1-3-6-9
- postorder
  - 4-7-8-5-2-9-6-3-1