# 常见数据结构

本文介绍几种常见的数据结构:栈、队列、堆、哈希表，等等。

写在前面

* 本文所有图片均截图自coursera上普林斯顿的课程[《Algorithms, Part I》](https://class.coursera.org/algs4partI-010/)中的Slides
* 相关命题的证明可参考[《算法（第4版）》](https://book.douban.com/subject/19952400/)
* 源码可在[官网](http://algs4.cs.princeton.edu/home/)下载,也可以在我的github仓库 [algorithms-learning](https://github.com/brianway/algorithms-learning)下载，已经使用maven构建
* 仓库下载：git clone git@github.com:brianway/algorithms-learning.git

Stacks(栈)

LIFO(后进先出):last in first out.

* 使用linked-list实现

保存指向第一个节点的指针，每次从前面插入／删除节点。

以字符串栈为例，示例代码：

public class LinkedStackOfStrings {

private Node first = null;

private class Node {

String item;

Node next;

}

public boolean isEmpty() {

return first == null;

}

public void push(String item) {

Node oldfirst = first;

first = new Node();

first.item = item;

first.next = oldfirst;

}

public String pop() {

String item = first.item;

first = first.next;

return item;

}

}

* 使用数组实现

使用数组来存储栈中的项

public class FixedCapacityStackOfStrings {

private String[] s;

private int N = 0;

public FixedCapacityStackOfStrings(int capacity) {

s = new String[capacity];

}

public boolean isEmpty() {

return N == 0;

}

public void push(String item) {

s[N++] = item;

}

public String pop() {

String item = s[--N];

s[N] = null;

return item;

}

}

上面的实现会有几个问题：

1. 从空栈pop会抛出异常
2. 插入元素过多会超出数组上界

这里重点解决第二个问题，resizing arrays.一个可行的方案是: **当数组满的时候，数组大小加倍；当数组是1/4满的时候，数组大小减半。** 这里不是在数组半满时削减size,这样可以避免数组在将满未满的临界点多次push-pop-push-pop操作造成大量的数组拷贝操作。

插入N个元素，N + (2 + 4 + 8 + ... + N) ~ 3N。

* N:1 array access per push
* (2 + 4 + 8 + … + N):k array accesses to double to size k (ignoring cost to create new array)

由于resize操作不是经常发生，所以均摊下来，平均每次push/pop操作的还是常量时间(constant amortized time).

Queues(队列)

FIFO(先进先出):first in first out.

* 使用linked-list实现

保存指向首尾节点的指针，每次从链表尾插入，从链表头删除。

public class LinkedQueueOfStrings {

private Node first, last;

private class Node {

/\* same as in StackOfStrings \*/

}

public boolean isEmpty() {

return first == null;

}

public void enqueue(String item) {

Node oldlast = last;

last = new Node();

last.item = item;

last.next = null;

if (isEmpty()) {

first = last;

} else {

oldlast.next = last;

}

}

public String dequeue() {

String item = first.item;

first = first.next;

if (isEmpty()) last = null;

return item;

}

}

* 使用数组实现

・Use array q[] to store items in queue.

・enqueue(): add new item at q[tail].

・dequeue(): remove item from q[head].

・Update head and tail modulo the capacity.

・Add resizing array.

Priority Queues

Collections. Insert and delete items.

* Stack. Remove the item most recently added.
* Queue. Remove the item least recently added. Randomized queue. Remove a random item.
* **Priority queue**. Remove the largest (or smallest) item.

unordered array 实现

public class UnorderedMaxPQ<Key extends Comparable<Key>> {

private Key[] pq; // pq[i] = ith element on pq

private int N; // number of elements on pq

public UnorderedMaxPQ(int capacity) {

pq = (Key[]) new Comparable[capacity];

}

public boolean isEmpty() {

return N == 0;

}

public void insert(Key x) {

pq[N++] = x;

}

public Key delMax() {

int max = 0;

for (int i = 1; i < N; i++)

if (less(max, i)) max = i;

exch(max, N - 1);

return pq[--N];

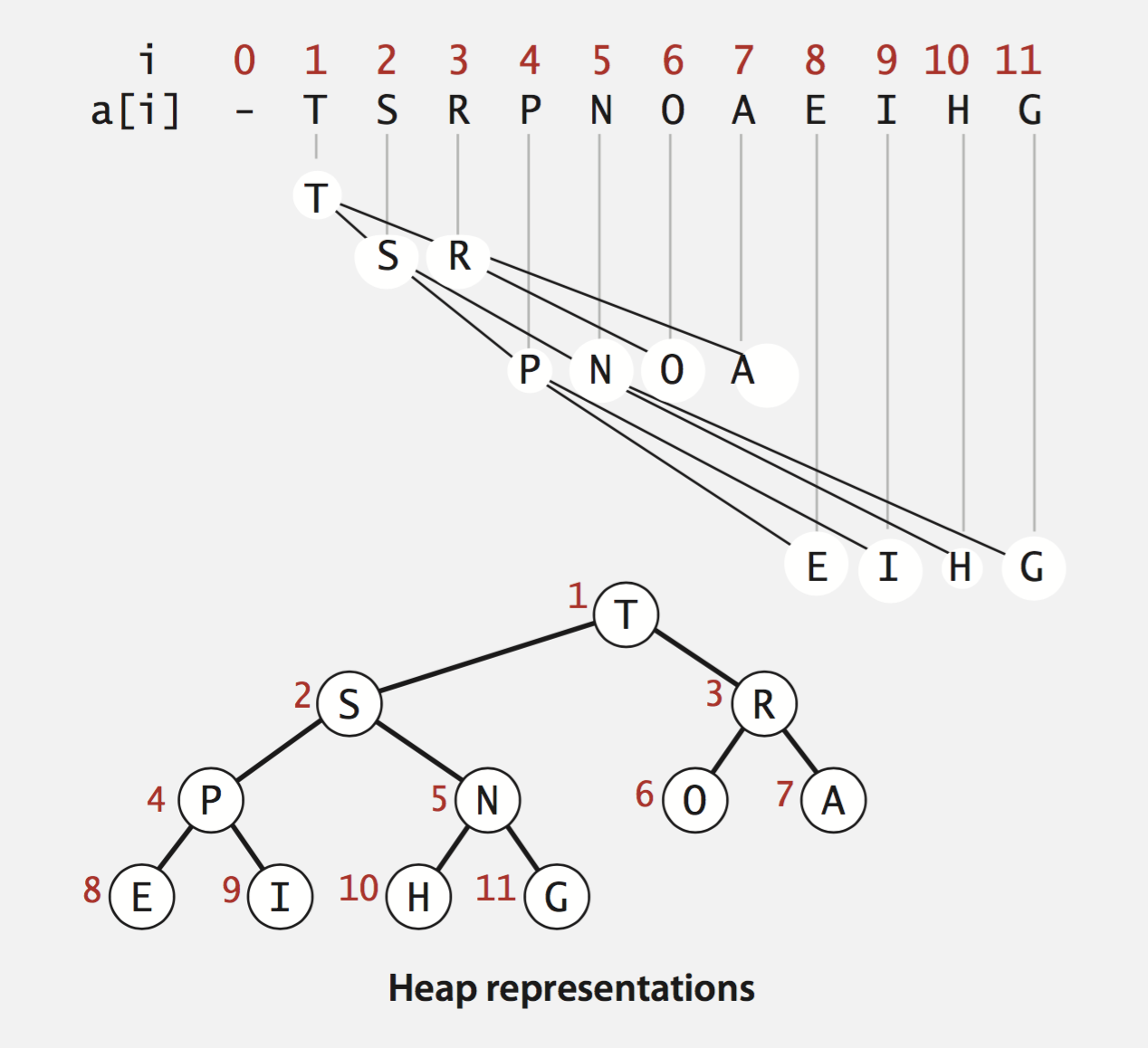
}

}

Binary Heaps(二叉堆)

使用数组来表示一个二叉堆。根节点索引从1开始。索引对应在树中的位置，最大的键值是a[1],同时也是二叉树的根节点。

* Parent’s key no smaller than children’s keys
* Indices start at 1.
* Parent of node at k is at k/2.
* Children of node at k are at 2k and 2k+1.



上浮和下沉

有两种情况会触发节点移动：

1. 子节点的键值变为比父节点大
2. 父节点的键值变为比子节点（一个或两个）小

而 **要消除这种违反最大堆定义的结构，就需要进行节点移动和交换， 使之满足父节点键值不小于两个子节点** 。对应的操作分别是 **上浮** 和 **下沉**

* 上浮：子节点key比父节点大
  + Exchange key in child with key in parent.
  + Repeat until heap order restored.
* 下沉：父节点key比子节点（one or both）小
  + Exchange key in parent with key in **larger** child.
  + Repeat until heap order restored

/\* 上浮 \*/

private void swim(int k) {

while (k > 1 && less(k / 2, k)) {

exch(k, k / 2);

k = k / 2;

}

}

/\* 下沉 \*/

private void sink(int k) {

while (2 \* k <= N) {

int j = 2 \* k;

if (j < N && less(j, j + 1)) j++;

if (!less(k, j)) break;

exch(k, j);

k = j;

}

}

插入和删除

所有操作（插入和删除）都保证在log N 时间内。

* 插入：二叉堆的插入操作比较简单，把节点加在数组尾部，然后上浮即可。
* 删除最大：二叉堆的删除则是把根节点和末尾的节点交换，然后下沉该节点即可。

/\* 插入 \*/

public void insert(Key x){

pq[++N] = x;

swim(N);

}

/\* 删除 \*/

public Key delMax(){

Key max = pq[1];

exch(1, N--);

sink(1);

pq[N+1]=null;

return max;

}

最后，堆中的键值是不能变的，即Immutable.不然就不能保证父节点不小于子节点。

Symbol Tables

键值对的抽象.其中键一般使用immutable的类型，值是任何普通类型。

关于比较，所有的java类都继承了equals()方法，要求对于引用x,y,z

* Reflexive: x.equals(x) is true.
* Symmetric: x.equals(y) iff y.equals(x).
* Transitive: if x.equals(y) and y.equals(z), then x.equals(z).
* Non-null: x.equals(null) is false.

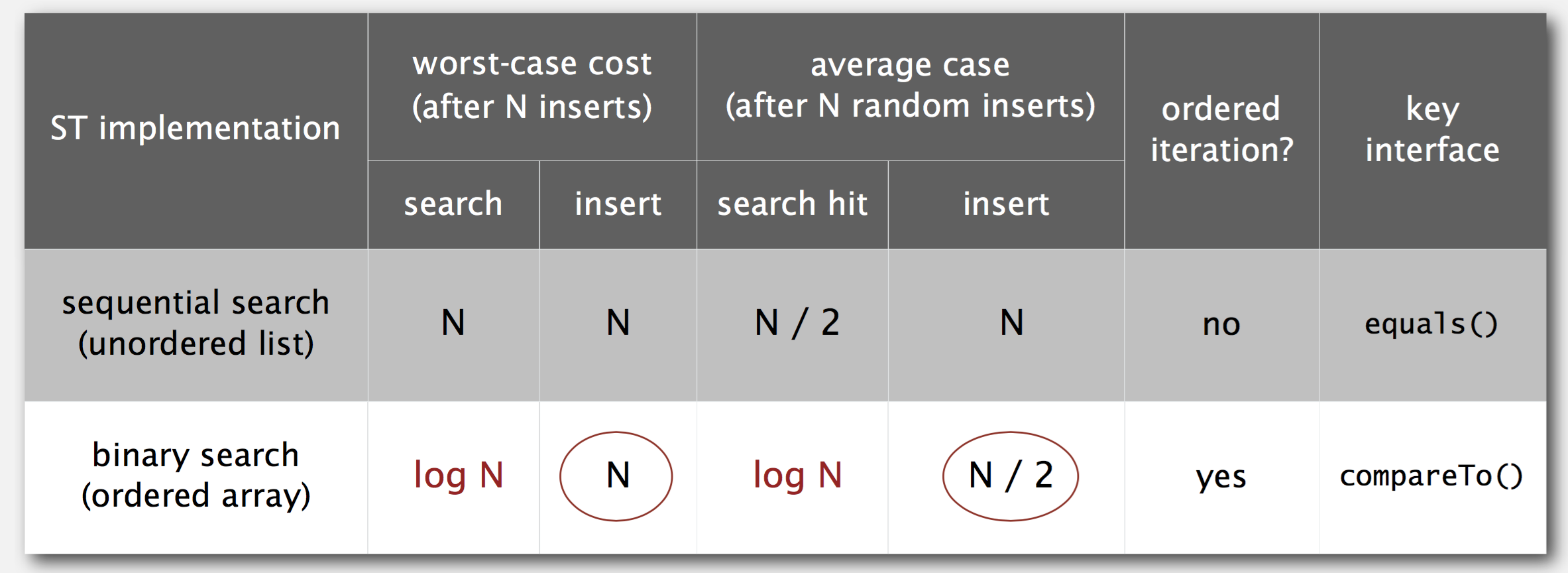
对于用户自定义的类型，一般按如下流程实现equals()方法:

* Optimization for reference equality.
* Check against null.
* Check that two objects are of the same type and cast.
* Compare each significant field:
  + if field is a primitive type, use ==
  + if field is an object, use equals().[apply rule recursively]
  + if field is an array, apply to each entry.[alternatively, use Arrays.equals(a, b) or Arrays.deepEquals(a, b),but not a.equals(b)]

两种实现的数据结构：

1. 无序链表：Maintain an (unordered) linked list of key-value pairs.
2. 有序数组：Maintain an ordered array of key-value pairs.

在有序数组进行查找时使用二分查找。两种方式的对比如下图：



Hash Tables(哈希表)

上面几种数据结构都是通过遍历或者二分查找去搜寻某个元素，而哈希表则是通过一个key-indexed table来存储其中的项，即“索引”是“键”的一个函数。换句话说，哈希是通过定义一种函数/计算方法，把键直接映射成一个哈希值（再通过取余操作换算成数组的下标索引），从而定位元素，而避免耗时的逐个比较和遍历的操作。

* Hash code:An int between -2^31 and 2^31 - 1.
* Hash function. An int between 0 and M - 1 (for use as array index).

//这里hashCode可能为负，且-2^31取绝对值会溢出，所以要“位与”

private int hash(Key key){

return (key.hashCode() & 0x7fffffff) % M;

}

所有的java类均继承了hashCode()方法来计算哈希值, 返回一个32-bit的int.默认实现是返回该对象的内存地址。对常用的类型有自己的实现，以java的String类为例子：

public int hashCode() {

int h = hash;

if (h == 0 && value.length > 0) {

char val[] = value;

for (int i = 0; i < value.length; i++) {

h = 31 \* h + val[i];

}

hash = h;

}

return h;

}

hash code design.”Standard” recipe for user-defined types：

* Combine each significant field using the 31x + y rule.
* If field is a primitive type, use wrapper type hashCode().
* If field is null, return 0.
* If field is a reference type, use hashCode().[applies rule recursively]
* If field is an array, apply to each entry.[or use Arrays.deepHashCode()]

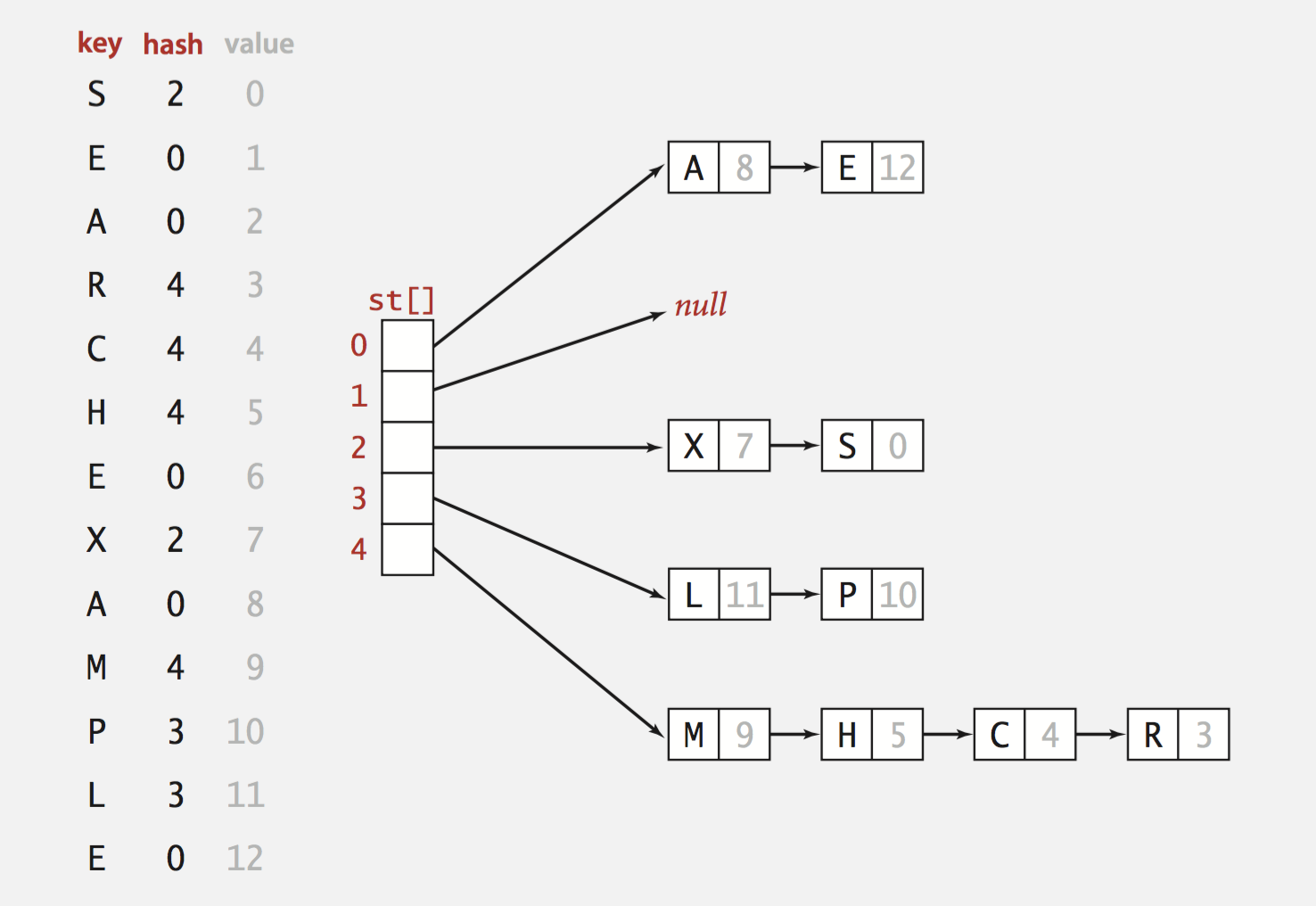
当然，这种映射并不能保证是一对一的，所以一定会出现多个键映射到同一个哈希值的尴尬情况（尤其是对数组的size取余操作后,映射到同一数组下标），即哈希冲突，这是就需要一些方法来解决。这里介绍两种常用的方法：

* separate chaining
* linear probing

separate chaining

Use an array of M < N linked lists.

* 哈希：将key映射到0 ~ M-1 之间的一个整数i
* 插入：将值插在第i个链的前端
* 查找：只需遍历第i个链



linear probing

开放地址：如果发生冲突，将值放入下一个空的位置.(数组尺寸 M 必须比键值对的数目 N 要多.)

* 哈希：将key映射到 0 ~ M-1 之间的一个整数i
* 插入：如果数组索引为 i 的位置为空，则把值放入，否则依次尝试 i+1,i+2等索引，直到有空的
* 查找：先找索引 i，如果被占用且没匹配，则依次尝试i+1, i+2,等等

本文介绍数据结构中几种常见的树:二分查找树，2-3树，红黑树，B树

写在前面

* 本文所有图片均截图自coursera上普林斯顿的课程[《Algorithms, Part I》](https://class.coursera.org/algs4partI-010/)中的Slides
* 相关命题的证明可参考[《算法（第4版）》](https://book.douban.com/subject/19952400/)
* 源码可在[官网](http://algs4.cs.princeton.edu/home/)下载,也可以在我的github仓库 [algorithms-learning](https://github.com/brianway/algorithms-learning)下载，已经使用maven构建
* 仓库下载：git clone git@github.com:brianway/algorithms-learning.git

Binary Search Tree(二分查找树)

定义：A BST is a **binary tree** in **symmetric order**.

A binary tree is either:

* Empty.
* Two disjoint binary trees (left and right).

Symmetric order.Each node has a key, and every node’s key is:

* Larger than all keys in its left subtree.
* Smaller than all keys in its right subtree.

在java的实现中，每个节点(Node)由四个域组成：**key,value,left,right**。即：键，值，左子树，右子树。

private class Node {

private Key key;

private Value val;

private Node left, right;

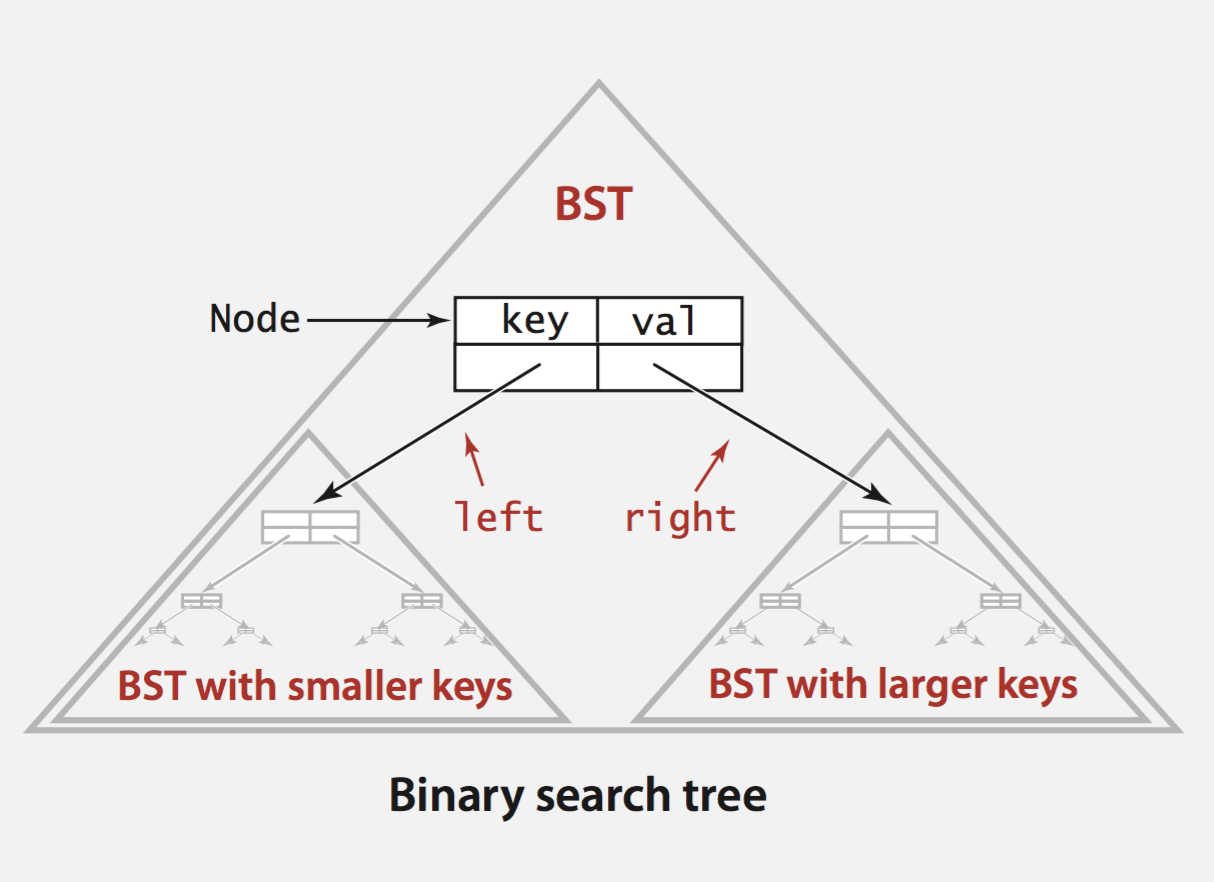
public Node(Key key, Value val) {

this.key = key;

this.val = val;

}

}



* 查找：得到相应键的值，若无此键则返回null.

/\* 查找 \*/

public Value 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;

}

* 插入：如果小，往左；如果大，往右；如果null，插入；如果存在，覆盖。

/\* 插入 \*/

public void put(Key key, Value val) {

root = put(root, key, val);

}

/\* 辅助函数，递归调用 \*/

private Node put(Node x, Key key, Value val) {

if (x == null) return new Node(key, val);

int cmp = key.compareTo(x.key);

if (cmp < 0) {

x.left = put(x.left, key, val);

} else if (cmp > 0) {

x.right = put(x.right, key, val);

} else { // if (cmp == 0)

x.val = val;

}

return x;

}

比较的次数为节点的深度+1,由于插入节点的顺序会有差异，所以树的高度不确定，最坏的情况是N个节点的树高度为N。

* 删除：列出下面几种处理方法
  + 将值置为null，在树中保留键
  + 删除最小值：一直向左找到左子树为null的节点，用它的右子节点代替它。
  + Hibbard deletion

下面重点讲一下Hibbard deletion,分为三种情况：

1. 没有子节点的节点，将其parent link置为null即可。
2. 有一个子节点的节点，删除该节点并以子节点代替即可。
3. 有两个子节点的节点，找到该节点t的下一个节点x（即右子树的最小节点），在右子树删除这个节点，并将该节点x放到t的位置。

/\* 删除 \*/

private Node delete(Node x, Key key) {

if (x == null) return null;

int cmp = key.compareTo(x.key);

if (cmp < 0) {

x.left = delete(x.left, key);

} else if (cmp > 0) {

x.right = delete(x.right, key);

} else {

if (x.right == null) return x.left; // no right child

if (x.left == null) return x.right; // no left child

Node t = x;

x = min(t.right); // replace with successor

x.right = deleteMin(t.right);

x.left = t.left;

}

x.count = size(x.left) + size(x.right) + 1;

return x;

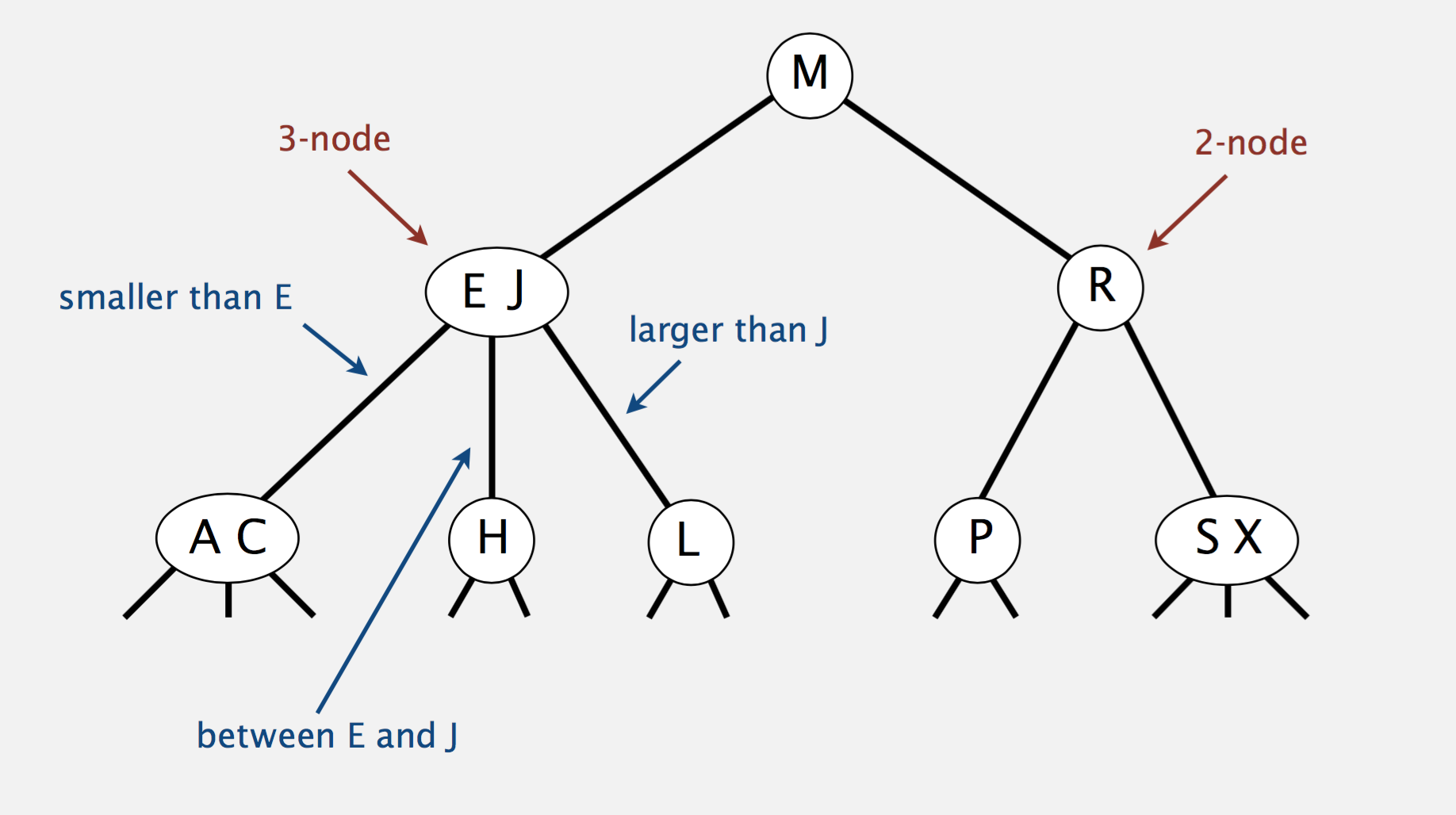
}

2-3 Search Trees(2-3树)

在介绍红黑树前，先介绍一下2-3树，便于后面理解红黑树。

2-3树是二分查找树的变形，每个节点是下面两种情况之一：

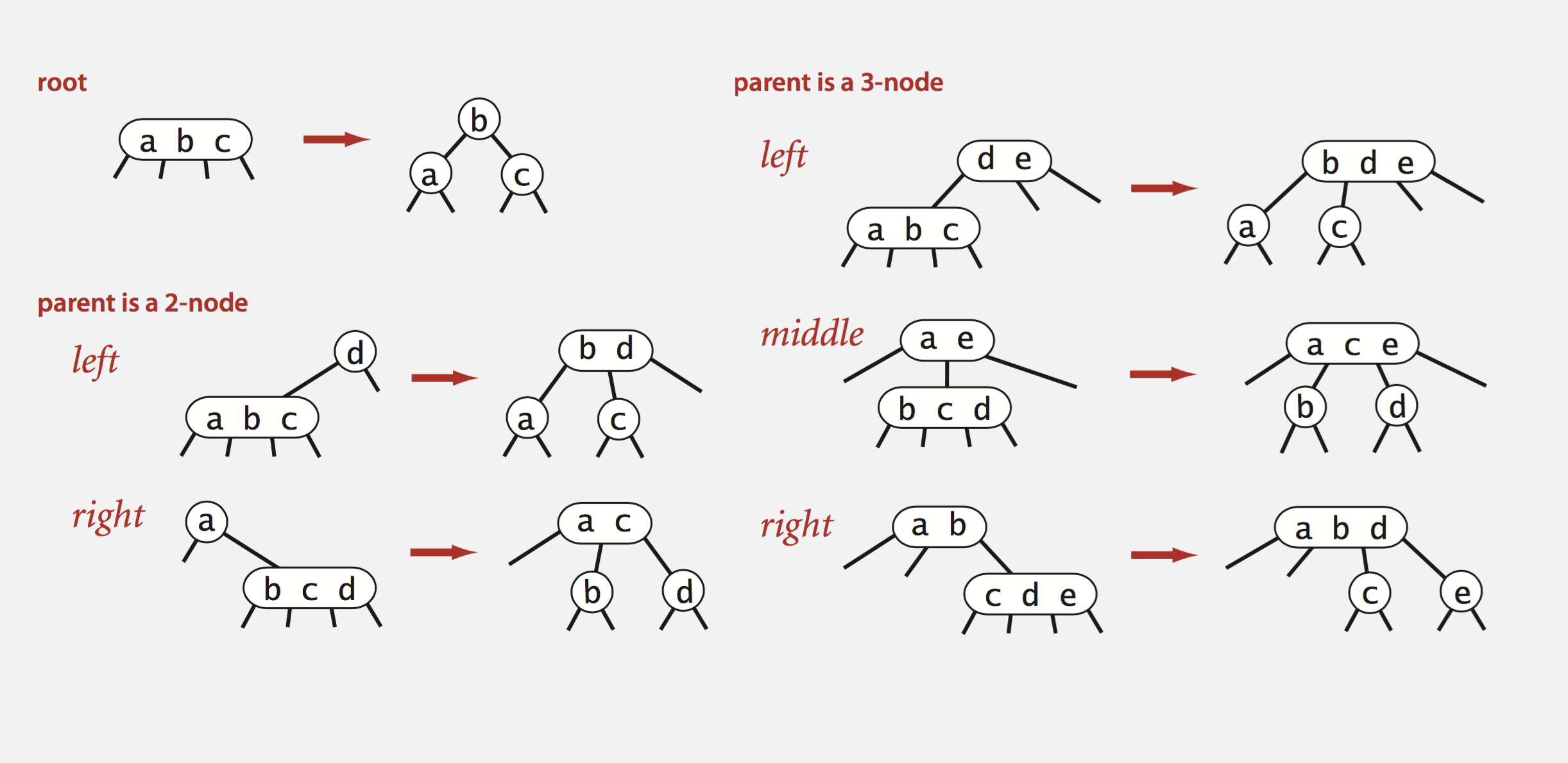
* 2-node:一个键，两个分叉（smaller,larger）
* 3-node:两个键，三个分叉（smaller,between,larger）



在底部向一个3-node插入。

* 向3-node插入一个键，临时成为一个4-node
* 将4-node中间的key移动到父节点
* 向上重复
* 如果到了顶端的根节点，且根节点是4-node,将其分成3个2-nodes.

总结起来就是：当插入的值导致节点变四叉时进行分裂，将中间的值传给上一个节点，并将另外两个值作为两个子节点分开，若上一节点也因此变成四叉，依次类推。分裂4-node是一个local transformation，只会进行常数次数的操作。**高度加一由且仅由顶节点分裂造成**

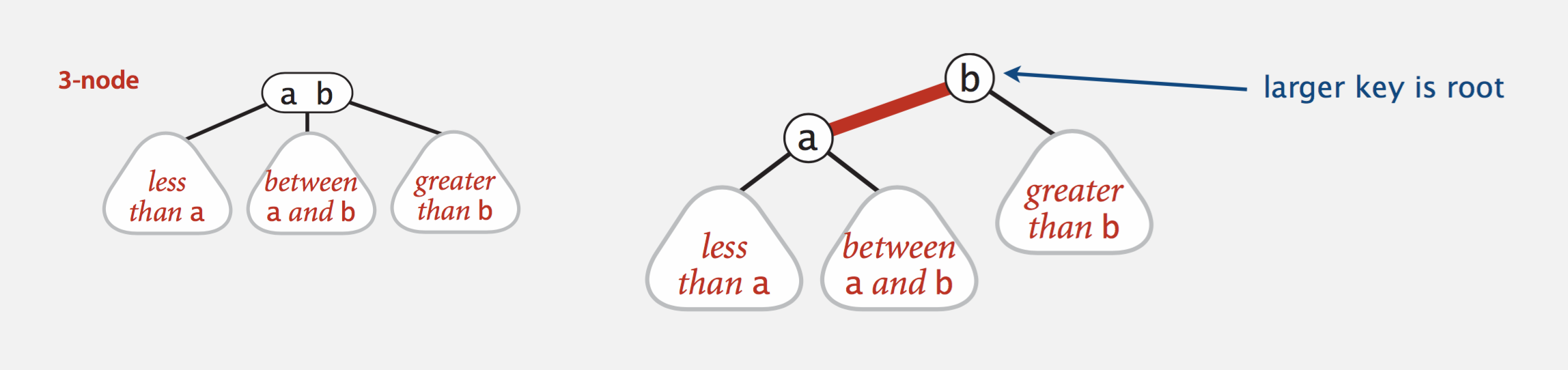


树的高度，在查找和插入时，保证了logarithmic的性能。

* Worst case: lg N. [all 2-nodes]
* Best case: log3 N ≈ 0.631 lg N. [all 3-nodes]

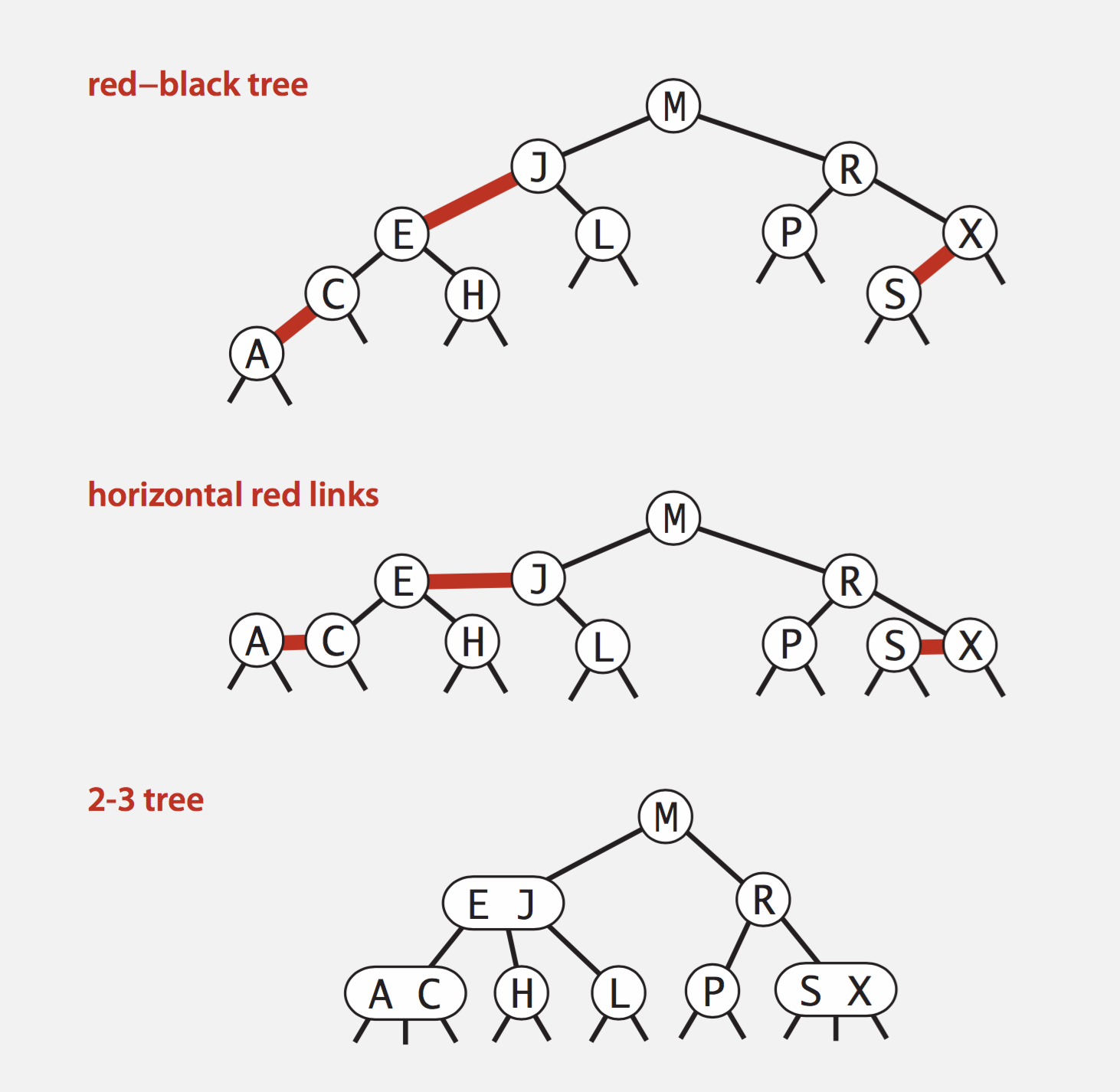
Red-Black BSTs(红黑树)

这里的红黑树均指Left-leaning red-black BSTs。主要是用二叉树的形式来表示2-3树，用一个“内部”的left-leaning连接来表示3-node。red link是2-3tree的三叉节点的连接两个key的内部link，大值作为根节点，小值作为左子节点，故名left leaning 红黑树。



一个等价的定义,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.



红黑树的java表示

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;

}

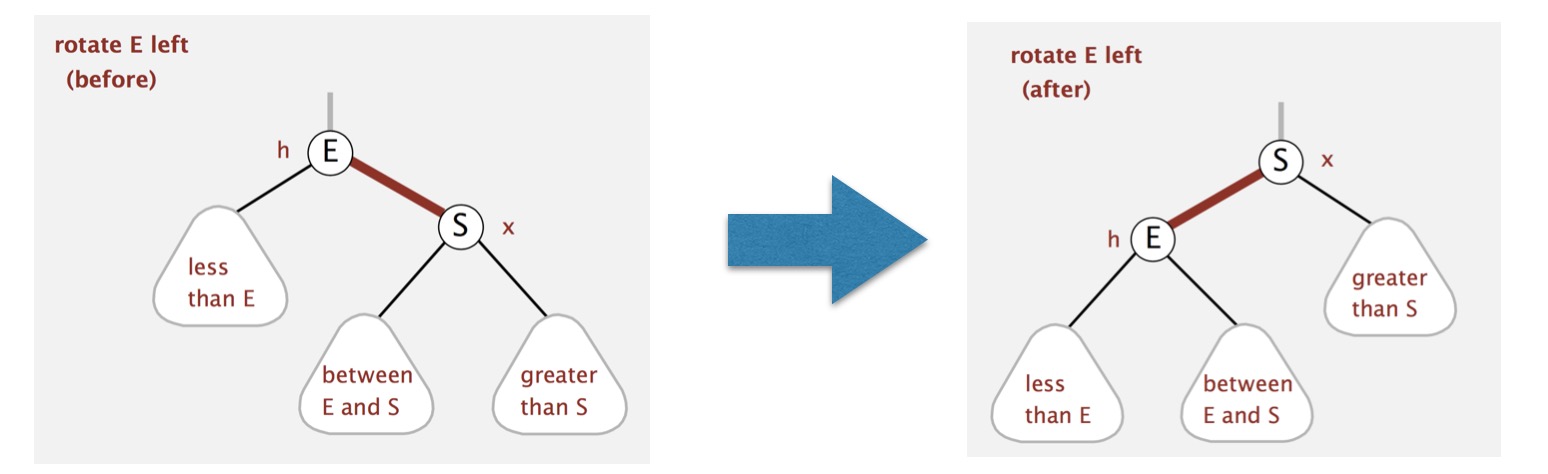
左转-右转-变色

红黑树插入过程中可能用到的三个基本操作（左转，右转，变色）：

* left rotate
* right rotate
* flip colors

下面依次介绍

* 左转



/\* left rotate \*/

private Node rotateLeft(Node h) {

assert isRed(h.right);

Node x = h.right;

h.right = x.left;

x.left = h;

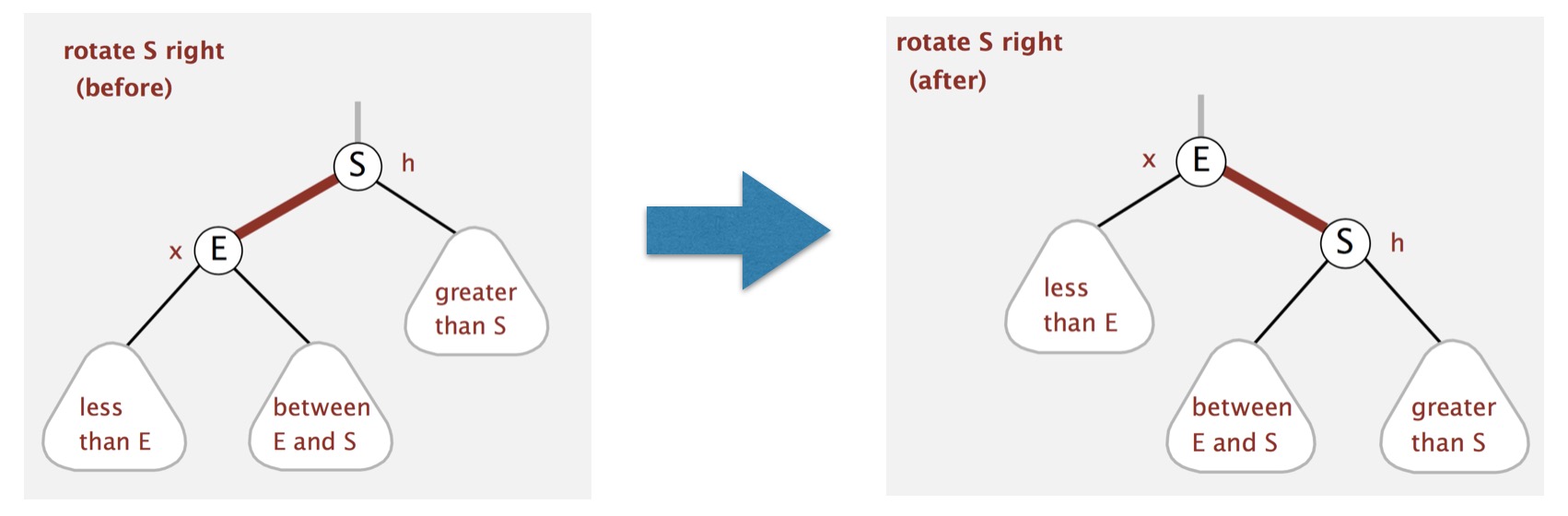
x.color = h.color;

h.color = RED;

return x;

}

* 右转



/\* right rotate \*/

private Node rotateRight(Node h) {

assert isRed(h.left);

Node x = h.left;

h.left = x.right;

x.right = h;

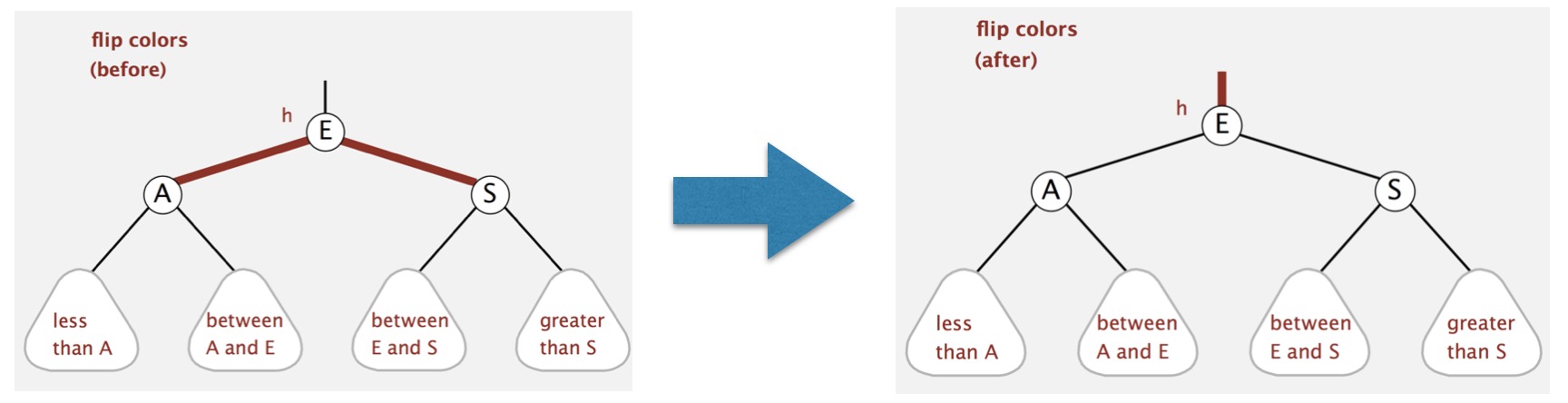
x.color = h.color;

h.color = RED;

return x;

}

* 变色



/\* flip colors \*/

private void flipColors(Node h) {

assert !isRed(h);

assert isRed(h.left);

assert isRed(h.right);

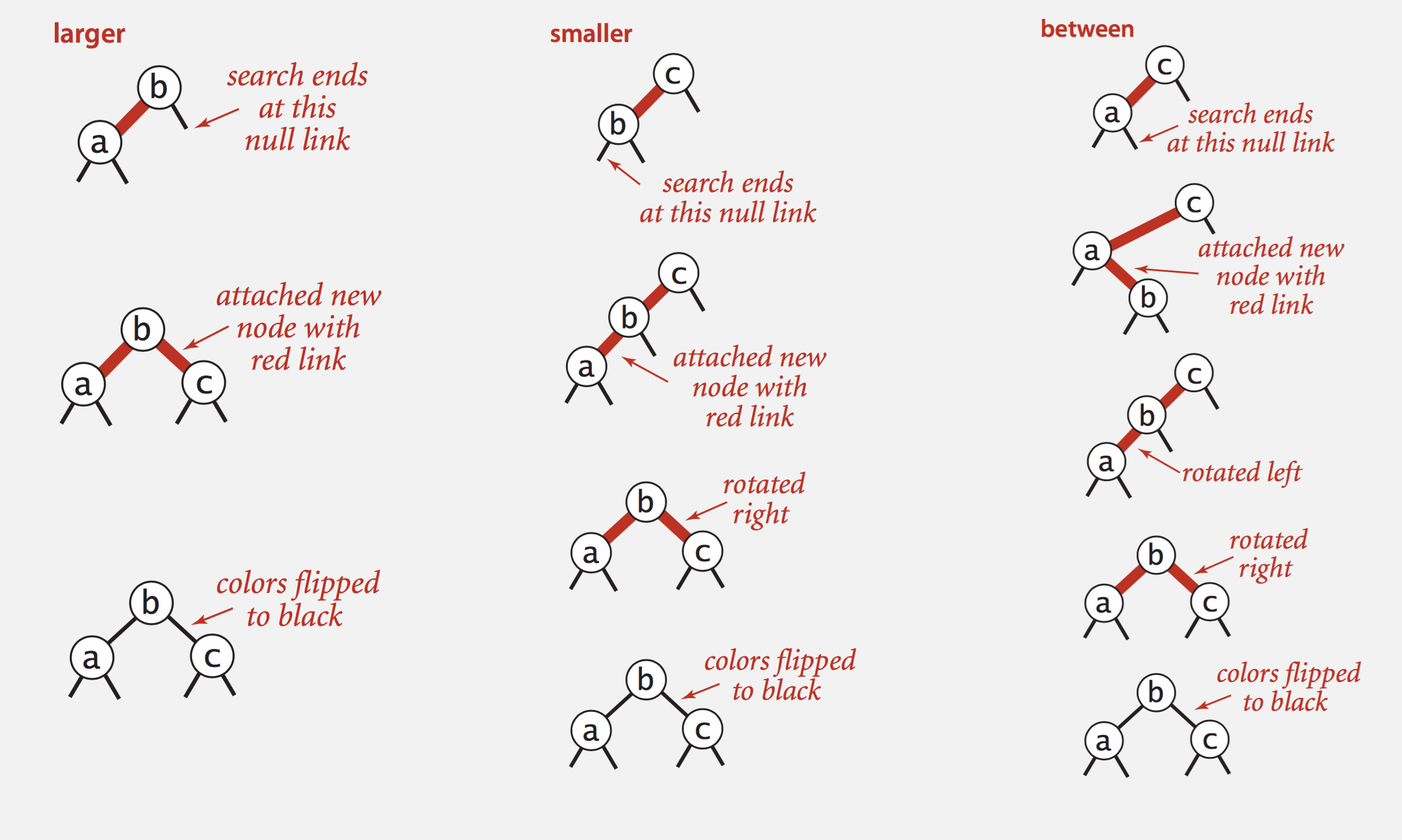
h.color = RED;

h.left.color = BLACK;

h.right.color = BLACK;

}

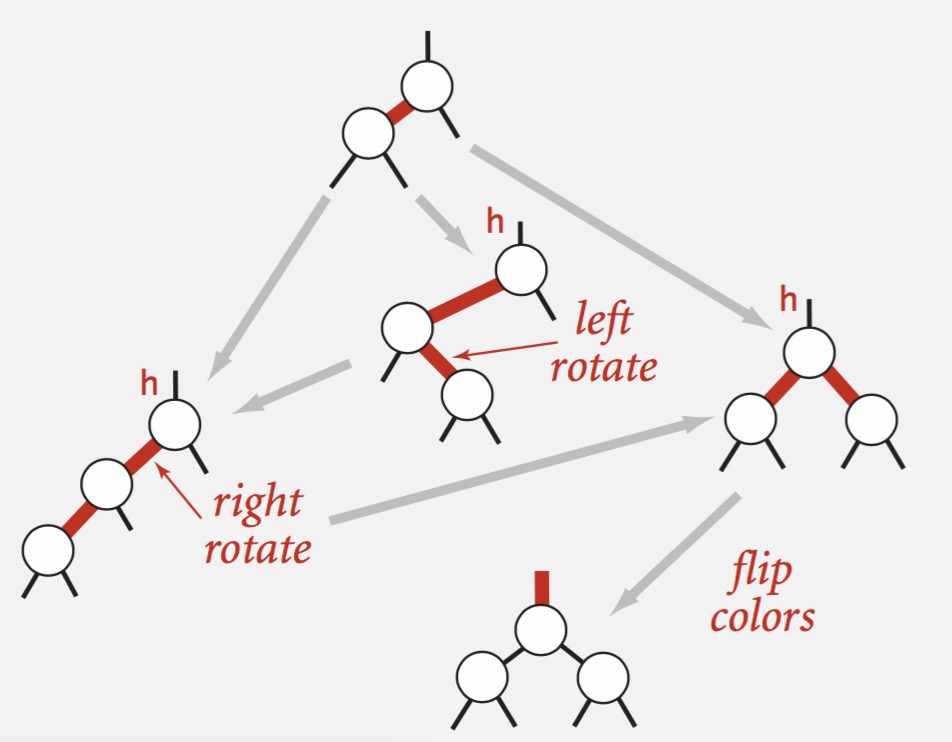
插入操作



从图中可以看出，插入的次序不同，需要转换的操作也不同，分三种情况（图中每一列是一种情况）：

1. 已有a和b时，c插入在b的右子节点，直接变色即可
2. 已有b和c时，a插入在b的左子节点，先右转把b滑上去，成1中的状态，再变色即可
3. 已有a和c时，b插入在a的右子节点，先左转把a滑下去，成2中的状态，再右转＋变色即可

从上面的分析可以看出，三种情况之间有转换关系，且逐步趋向简单，如下图所示：



**根本原因在于，2-3树中，是把3-node中处于中间的那个键传递给父节点，所以在红黑树中，当有一个节点连了两个 red link时，说明这三个点是一个3-node，但次序还需要调整，从而达到中间键在最上的状态，进而变色。而这个这个调整的趋势则是先让b处于a,c中间(即a的父，c的左子，成一条线)，再让b成为a,c的父节点，最后变色。记住这个顺序和原因，写代码就简单了，状态3->状态2->状态1**

private Node put(Node h, Key key, Value val) {

//insert at bottom (and color it red)

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;

}

if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);// lean left

if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);//balance 4-node

if (isRed(h.left) && isRed(h.right)) flipColors(h);//split 4-node

return h;

}

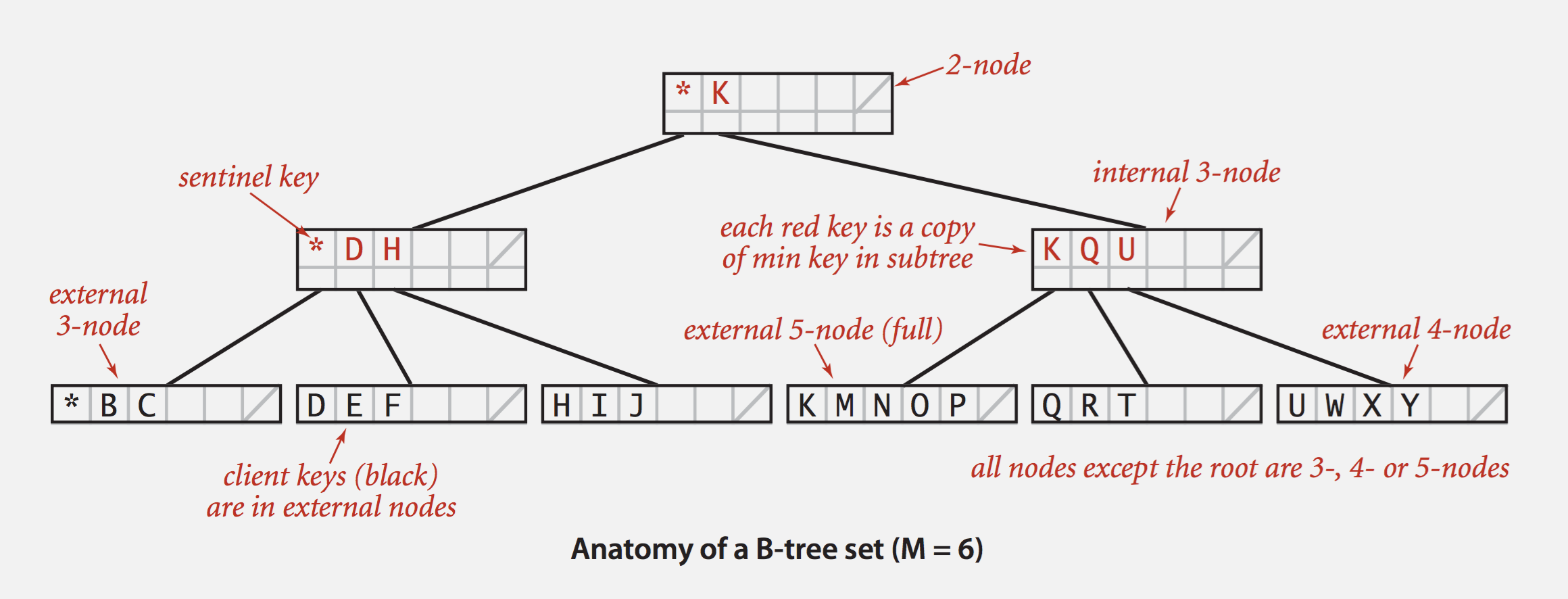
红黑树的高度 h <= 2 lg N，证明：

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

B-Trees(B树)

最后简单提一下B树，就是将2-3树一般化，将每个节点的key-link pairs增加到 M - 1

* 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.



在B树中查找

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

在B树中插入

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

命题：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