0玩转数据结构 从入门到进阶

C6 二分搜索树

6-2 二分搜索树基础

二分搜索树的性质：

- 二分搜索树是二叉树
- 二分搜索树的每个节点的值大于其左子树的所有节点的值，小于其右子树的所有节点的值

6-12 删除二分搜索树的任意元素

- 如果一个节点没有左子树，则可以删除以该节点为根的树的最小元素（即只要把该节点的右子树放在该节点位置即可）
- 如果一个节点没有右子树，则可以删除以该节点为根的树的最大元素（即只要把该节点的左子树放在该节点位置即可）
- 如果一个节点既有左子树也有右子树，Hibbard Deletion：可以把左子树的最大元素（或者右子树的最小元素）替换到该节点的位置。

```
public class BST<K extends Comparable<K>, V> {

    private class Node{
        public K key;
        public V value;
        public Node left, right;

        public Node(K key, V value){
            this.key = key;
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node root;
    private int size;

    public BST(){
        root = null;
        size = 0;
    }

    public int getSize(){
        return size;
    }


    public boolean isEmpty(){
```

```
        return size == 0;
    }

    // 向二分搜索树中添加新的元素(key, value)
    public void add(K key, V value){
        root = add(root, key, value);
    }

    // 向以node为根的二分搜索树中插入元素(key, value)，递归算法
    // 返回插入新节点后二分搜索树的根
    private Node add(Node node, K key, V value){

        if(node == null){
            size ++;
            return new Node(key, value);
        }

        if(key.compareTo(node.key) < 0)
            node.left = add(node.left, key, value);
        else if(key.compareTo(node.key) > 0)
            node.right = add(node.right, key, value);
        else // key.compareTo(node.key) == 0
            node.value = value;

        return node;
    }

    // 返回以node为根节点的二分搜索树中，key所在的节点
    private Node getNode(Node node, K key){

        if(node == null)
            return null;

        if(key.equals(node.key))
            return node;
        else if(key.compareTo(node.key) < 0)
            return getNode(node.left, key);
        else // if(key.compareTo(node.key) > 0)
            return getNode(node.right, key);
    }

    public boolean contains(K key){
        return getNode(root, key) != null;
    }

    public V get(K key){

        Node node = getNode(root, key);
        return node == null ? null : node.value;
    }

    public void set(K key, V newValue){

        Node node = getNode(root, key);
```

```
        if(node == null)
            throw new IllegalArgumentException(key + " doesn't exist!");

        node.value = newValue;
    }

    // 返回以node为根的二分搜索树的最小值所在的节点
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }

    // 删除掉以node为根的二分搜索树中的最小节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMin(Node node){

        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;
            return rightNode;
        }

        node.left = removeMin(node.left);
        return node;
    }

    // 从二分搜索树中删除键为key的节点
    public V remove(K key){

        Node node = getNode(root, key);
        if(node != null){
            root = remove(root, key);
            return node.value;
        }
        return null;
    }

    private Node remove(Node node, K key){

        if( node == null )
            return null;

        if( key.compareTo(node.key) < 0 ){
            node.left = remove(node.left , key);
            return node;
        }
        else if(key.compareTo(node.key) > 0 ){
            node.right = remove(node.right, key);
            return node;
        }

        else{   // key.compareTo(node.key) == 0
```

```
            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                return rightNode;
            }

            // 待删除节点右子树为空的情况
            if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                return leftNode;
            }

            // 待删除节点左右子树均不为空的情况

            // 找到比待删除节点大的最小节点，即待删除节点右子树的最小节点
            // 用这个节点顶替待删除节点的位置
            Node successor = minimum(node.right);
            successor.right = removeMin(node.right);
            successor.left = node.left;

            node.left = node.right = null;

            return successor;
        }
    }
}
```

C7 集合和映射

7-8 映射的复杂度分析和更多映射相关的问题

	LinkedListMap	BSTMap 平均	BSTMap 最差
增 add	O(n)	O(logn)	O(n)
删 delete	O(n)	O(logn)	O(n)
改 set	O(n)	O(logn)	O(n)
查 get	O(n)	O(logn)	O(n)
查 contains	O(n)	O(logn)	O(n)

LinkedListMap在做增删改查操作时，都需要先查询一个Map中是否含有key，所以时间复杂度也是O(n)。

因为二分搜索树最差情况下有可能沦为一个链表，这时二分搜索树时间复杂度也会很大，这是二分搜索树的一个最致命问题。所以后序会引入平衡二叉树。

映射：

- 有序映射：键具有顺序性　<== 基于搜索树的实现：TreeMap和TreeSet基于平衡二叉树（或者红黑树）实现
- 无序映射：键没有顺序性　<== 基于哈希表的实现：HashSet和HashMap基于哈希表实现

C8 优先队列和堆

8-1 什么是优先队列

- 普通队列：先进先出，后进后出
- 优先队列：出队顺序和入队顺序无关；和优先级相关

关键词：动态

	入队	出队（拿出最大元素）
普通线性结构：出队先扫描找出最大值，把最大值出队	O(1)	O(n)
顺序线性结构：入队按照大小顺序排列入队	O(n)	O(1)
堆	O(logn)	O(logn)

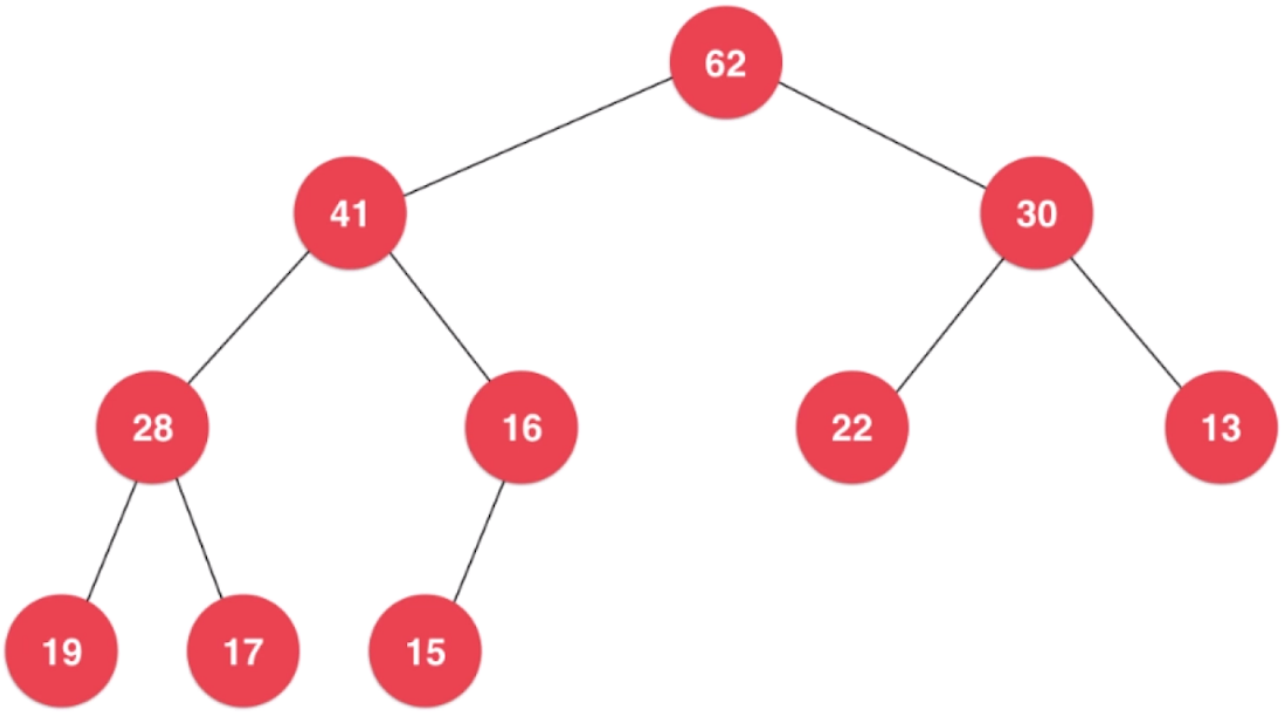
8-2 堆的基础表示

堆最为主流的一种实现方式：**二叉堆**

二叉堆的性质：

- 二叉堆是一颗完全二叉树
  - 完全二叉树：不一定是满二叉树（除了叶子节点，其他节点的左右子数都不为空），但是缺失的部分一定是在二叉树的右侧。
- 最大堆：二叉堆中某个节点的值总是不大于其父节点的值（根节点的元素是最大的）。
- 最小堆：二叉堆中某个节点的值总是不小于其父节点的值（根节点的元素是最小的）。

注意：最大堆的低层级的节点值不一定就大于高层级的节点值。



例如位于2层的16和3层的19。

因为完全二叉树可以看作一层一层地从左向右码上去，所以可以把完全二叉树用数组来存储。

0	1	2	3	4	5	6	7	8	9
62	41	30	28	16	22	13	19	17	15

此时对于节点 i（节点在数组中的索引是 i）：

$$parent(i) = (i-1)/2$$

$$leftchild = 2 * i + 1$$

$$rightchild = 2 * i + 2$$

8-3 向堆中添加元素和Sift Up

1. 先在index 10添加元素52
2. 因为52的父节点 16小于 52，所以把52和16换一下位置，即把索引4变为52，索引10变为16
3. 此时52依然大于父节点41，所以把52和41再交换位置
4. 此时满足堆的性质。

8-4 从堆中取出元素和Sift Down

1. 取出堆的根节点
2. 把堆数组的最后一个元素挪到根节点
3. 把根节点元素和子元素的最大值比较，如果小，则交换位置，依次进行下去，直到满足堆特性。

8-5 Heapify和Replace

- Replace：取出堆中最大元素，再向堆中添加一个元素

  可以直接用新添加的元素替换堆顶元素，再执行Sift Down即可

- heapify：将任意数组整理成堆的形状

  把数组看作一个完全二叉树，从最后一个非叶子节点i开始，从索引i到0不断进行Sift Down操作，直到满足堆性质。

计算最后一个非叶子节点的索引：直接根据最后一个节点的索引，计算其父节点，也就是最后一个非叶子节点。

- 将n个元素逐个插入到一个空堆中，算法复杂度是O(logn)，因为对每个元素都要执行O(logn)级别的插入操作，总共有n个元素。
- heapify的过程，算法复杂度是O(n)。

```
public class MaxHeap<E extends Comparable<E>> {

    private List<E> data;

    public MaxHeap(int capacity) {
        data = new ArrayList<>(capacity);
    }

    public MaxHeap() {
        data = new ArrayList<>();
    }

    /**
     * heapify
     * @param arr 任意元素
     */
    public MaxHeap(E[] arr) {
        data = new ArrayList<>(Arrays.asList(arr));
        for (int i = parent(arr.length - 1); i >= 0; i--) {
            siftDown(i);
        }
    }

    public int size() {
        return data.size();
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }

    // 返回完全二叉树的数组表示中，一个索引所表示的元素的父节点的索引
    private int parent(int index) {
        if (index == 0) {
            throw new IllegalArgumentException("index-0木有父节点");
        }
        return (index - 1) / 2;
    }

    // 返回完全二叉树的数组表示中，一个索引所表示的元素的左孩子节点的索引
```

```
    private int leftChild(int index) {
        return index * 2 + 1;
    }

    // 返回完全二叉树的数组表示中，一个索引所表示的元素的右孩子节点的索引
    private int rightChild(int index) {
        return index * 2 + 2;
    }

    // 向堆中添加元素
    public void add(E e) {
        data.add(e);
        siftUp(data.size() - 1);
    }

    //查看堆的最大值
    public E findMax() {
        if (data.size() == 0) throw new IllegalArgumentException("堆是空的");
        return data.get(0);
    }

    // 从堆中取出最大元素
    public E extractMax() {
        E result = findMax();
        data.set(0, data.get(data.size() - 1));
        data.remove(data.size() - 1);
        siftDown(0);
        return result;
    }

    // 取出堆中最大元素，并且替换成元素e
    public E replace(E e) {
        E ret = findMax();
        data.set(0, e);
        siftDown(0);
        return ret;
    }

    private void siftDown(int k) {
        while (leftChild(k) < data.size()) {
            int i = leftChild(k);
            if (i + 1 < data.size() && data.get(i + 1).compareTo(data.get(i)) > 0) {
                i = rightChild(k);
            }
            //此时data.get(i)是左右两个子节点的最大值
            if (data.get(i).compareTo(data.get(k)) <= 0)
                break;
            E tmp = data.get(i);
            data.set(i, data.get(k));
            data.set(k, tmp);
            k = i;
        }

    }
```

```
    private void siftUp(int index) {
        for (; index > 0 && data.get(parent(index)).compareTo(data.get(index)) < 0;) {
            E tmp = data.get(parent(index));
            data.set(parent(index), data.get(index));
            data.set(index, tmp);
            index = parent(index);
        }
    }
}
```

8-7 Leetcode上优先队列相关问题

在1,000,000个元素中选出前100名？

在N个元素中选出前M个元素：

- 排序 ==> O(NlogN)
- 优先队列 ==> O(NlogM)

使用优先队列，维护当前看到的前M个元素，遇见一个新的元素，把它和队列中最小的元素比，如果比最小的大，则把最小的元素替换成当前元素，并执行相应操作满足二叉堆的性质。

这里需要使用***最小堆***。或者使用最大堆，但是规定频率越小优先级越高。

> 注：最大堆和最小堆的大和小都是相对的，可以根据需要定义不同的大和小的概念。

8-8 Java中的PriorityQueue

Java中的PriorityQueue内部是最小堆。

8-9 和堆相关的更多话题和广义队列

这里实现的堆有一个明显的缺点：只能看见堆首的元素，不能看见堆中的元素。

索引堆：可以解决这个问题，还可以修改堆中的元素。

C9 线段树

9-1 什么是线段树

线段树、区间树、Segment Tree

1. 为什么要使用线段树？
    - 有一类问题，我们关心的是线段（或者区间）
    - 最经典的线段树问题：区间染色

      有一面墙，长度为n，每次选择一段墙进行染色（可以覆盖之前已经染过色的区间），m次操作后，我们可以在[i, j]区间内看见多少种颜色？

	使用数组实现
染色操作（更新区间）	O(n)
查询操作（查询区间）	O(n)

- 另一类经典问题：区间查询

  查询一个区间[i, j]的最大值，最小值，或者区间数字和

  2017年注册用户中消费最高的用户？消费最小的用户？学习时间最长的用户？数据是动态的，因为用户在不断地进行消费，2017年注册的用户2018年也在接着消费，数据在持续不停地更新的同时进行查询。

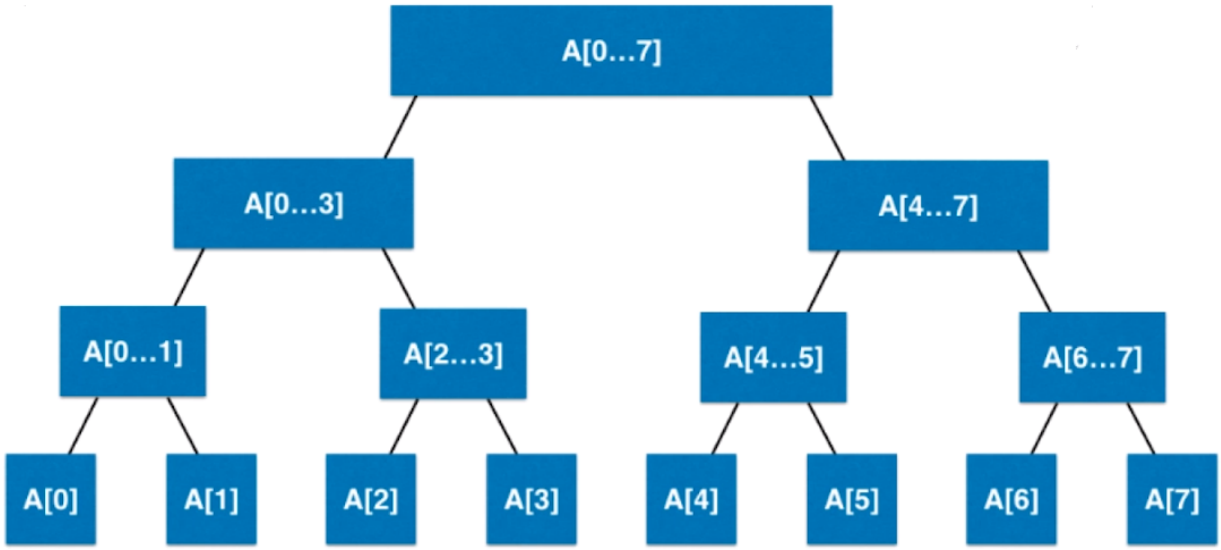
	使用数组实现	使用线段树
更新	O(n)	O(logn)
查询	O(n)	O(logn)

  对于给定区间

  更新：更新区间中一个元素或者一个区间的值

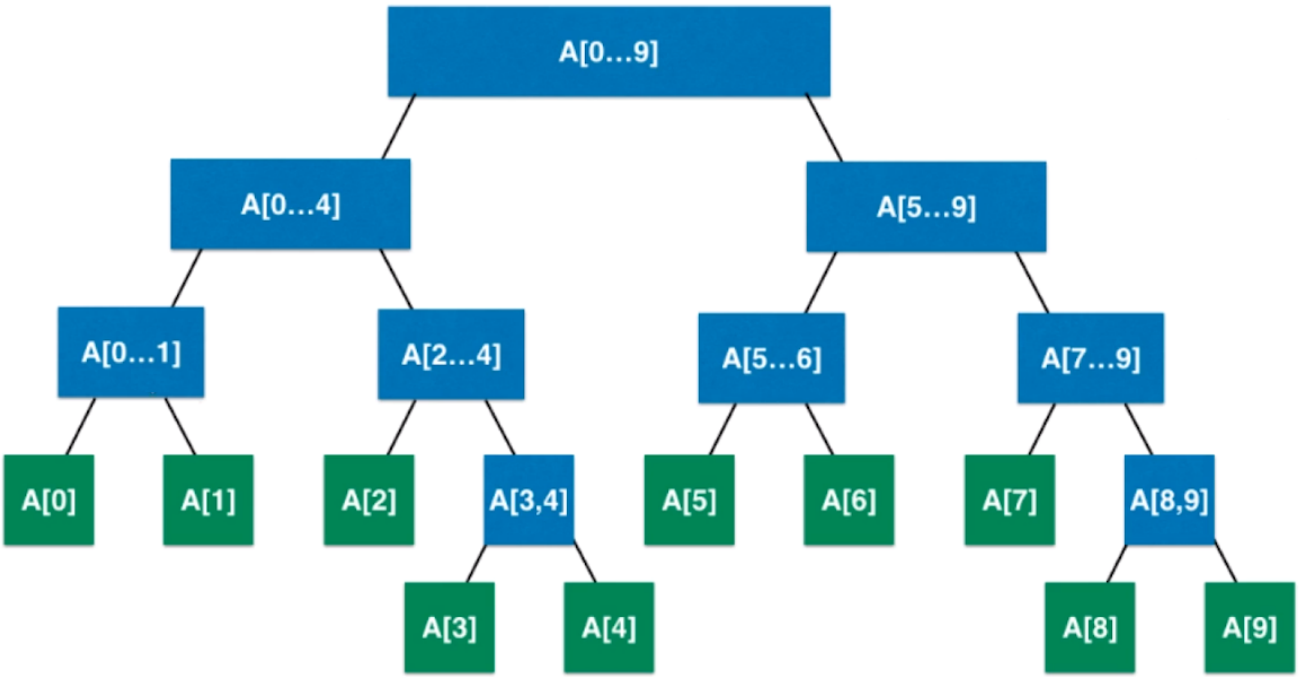
  查询：查询一个区间的最大值，最小值，或者区间数字和

2. 什么是线段树



以求和为例：根节点存储的是整个区间的和，A[0...3]存储的是0-3这个区间的和，A[4...7]存储的是4-7这个区间的和。如果要求3-5这个区间的和，那么只需把A[3]和A[4...5]这两个节点加起来就行。

9-2 线段树基础表示



线段树不一定是完全二叉树，线段树是平衡二叉树

平衡二叉树：对于整棵树来说，最大的深度和最小的深度相差不能超过1。

完全二叉树一定是平衡二叉树，所以堆也是平衡二叉树。

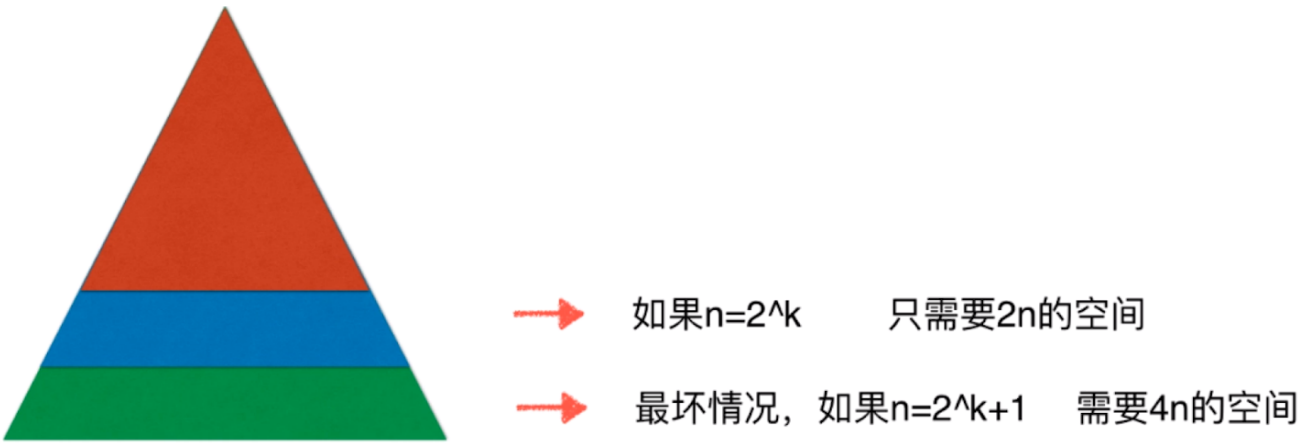
二分搜索树不一定是平衡二叉树。

线段树也可以用数组来表示，可以把它看作是完全二叉树，缺失的部分都是null。

如果区间有n个元素，数组表示需要有多少个节点？

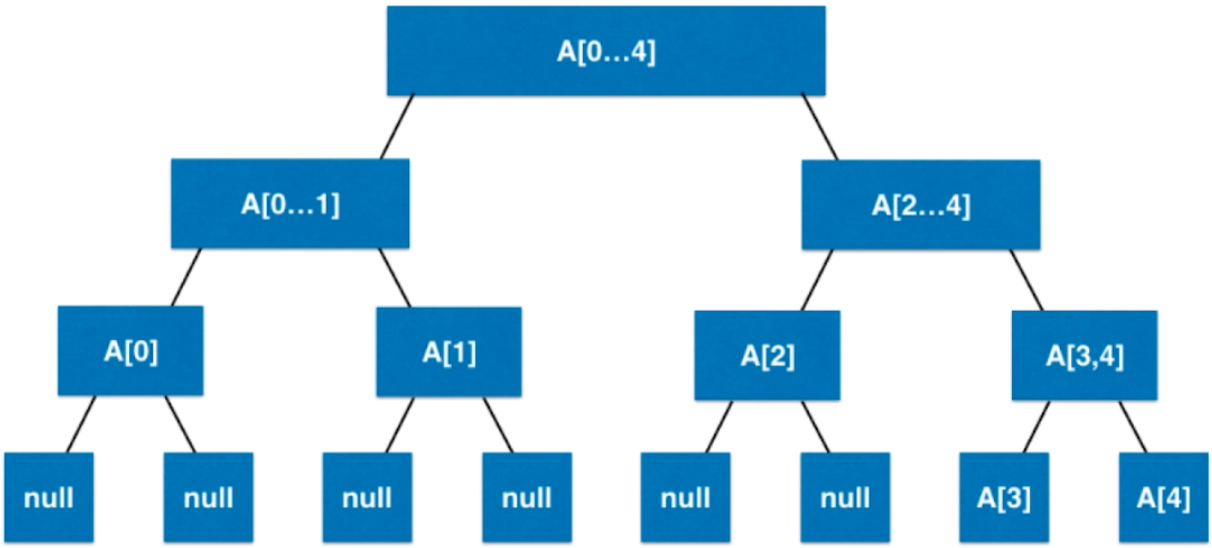
对满二叉树：h层，一共有2^h - 1个节点（约2 ^ h），最后一层（h-1层），有2 ^ (h-1)个节点，所以最后一层地节点数大致等于前面所有层节点之和。

如果区间有n个元素：



我们的线段树不考虑添加元素，即区间固定，使用4n的静态空间即可。

例如区间里有5个元素：




```
public class SegmentTree<E> {
    private E[] tree;
    private E[] data;
    private Merger<E> merger;

    public SegmentTree(E[] arr, Merger<E> merger) {
        data = (E[])new Object[arr.length];
        for (int i = 0; i < arr.length; i++) {
            data[i] = arr[i];
        }
        this.merger = merger;
        tree = (E[])new Object[arr.length * 4];
        buildSegmentTree(0, 0, data.length - 1);
    }

    /**
     * 在treeIndex的位置创建表示区间[l...r]的线段树
     * @param treeIndex 当前索引
     * @param s 区间起始位置
     * @param e 区间结束位置
     */
    private void buildSegmentTree(int treeIndex, int s, int e) {
        if (s == e) {
            tree[treeIndex] = data[s];
            return;
        }
        int leftTreeIndex = leftChild(treeIndex);
        int rightTreeIndex = rightChild(treeIndex);
        int mid = s + (e - s) / 2;          // 防止s + e太大溢出
        buildSegmentTree(leftTreeIndex, s, mid);
        buildSegmentTree(rightTreeIndex, mid + 1, e);

        tree[treeIndex] = merger.merge(tree[leftTreeIndex], tree[rightTreeIndex]);          //
这个是和业务逻辑相关的
    }
```

```
    public int getSize() {
        return data.length;
    }

    public E get(int index) {
        if (index < 0 || index >= data.length)
            throw new IllegalArgumentException("索引不合法");
        return data[index];
    }

    // 返回一个完全二叉树的数组表示中，一个索引所表示的元素的左孩子节点的索引
    private int leftChild(int index) {
        return index * 2 + 1;
    }

    // 返回一个完全二叉树的数组表示中，一个索引所表示的元素的右孩子节点的索引
    private int rightChild(int index) {
        return index * 2 + 2;
    }

    /**
     * 线段树的查询，返回区间[start, end]的值
     * @param queryS 起始索引
     * @param queryE 结束索引
     * @return 区间[start, end]的值
     */
    public E query(int queryS, int queryE) {
        if (queryS < 0 || queryE < 0 || queryS > data.length || queryE > data.length || queryS >
queryE)
            throw new IllegalArgumentException("参数不合法");
        return query(0, queryS, queryE, 0, data.length - 1);
    }

    /**
     * 把data的dataIndex位置的值更新为e
     * @param dataIndex
     * @param e
     */
    public void set(int dataIndex, E e) {
        if (dataIndex < 0 || dataIndex > data.length)
            throw new IllegalArgumentException("参数不合法");
        data[dataIndex] = e;
        set(0, 0, data.length - 1, dataIndex, e);
    }

    private void set(int treeIndex, int start, int end, int dataIndex, E e) {
        if (start == end) {
            tree[treeIndex] = e;
            return;
        }
        int leftChild = leftChild(treeIndex);
        int rightChild = rightChild(treeIndex);

        int mid = start + (end - start) / 2;
```

```
        if (dataIndex >= mid + 1)
            set(rightChild, mid + 1, end, dataIndex, e);            //更新右子树
        else
            set(leftChild, start, mid, dataIndex, e);                    //更新左子树
        tree[treeIndex] = merger.merge(tree[leftChild], tree[rightChild]);      //根据右子树和左子树来更新当前节点
    }


    private E query(int treeIndex, int queryS, int queryE, int start, int end) {
        if (queryS == start && queryE == end)
            return tree[treeIndex];
        int mid = start + (end - start) / 2;
        int leftTreeIndex = leftChild(treeIndex);
        int rightTreeIndex = rightChild(treeIndex);
        if (queryS >= mid + 1) {
            return query(rightTreeIndex, queryS, queryE, mid + 1, end);
        }else if (queryE <= mid) {
            return query(leftTreeIndex, queryS, queryE, start, mid);
        }else {
            E leftResult = query(leftTreeIndex, queryS, mid, start, mid);
            E rightResult = query(rightTreeIndex, mid + 1, queryE, mid + 1, end);
            return merger.merge(leftResult, rightResult);
        }
    }

    @Override
    public String toString() {
        StringBuilder res = new StringBuilder();
        res.append('[');
        for (int i = 0; i < tree.length; i++) {
            if (tree[i] != null)
                res.append(tree[i]);
            else
                res.append("null");
            if (i != tree.length - 1)
                res.append(", ");
        }
        res.append(']');
        return res.toString();
    }
}


interface Merger<E> {
    E merge(E a, E b);
}
```

9-7 更多线段树相关的话题

将区间中所有元素更新：O(n)的复杂度，为了解决这个问题，可以用懒惰更新。

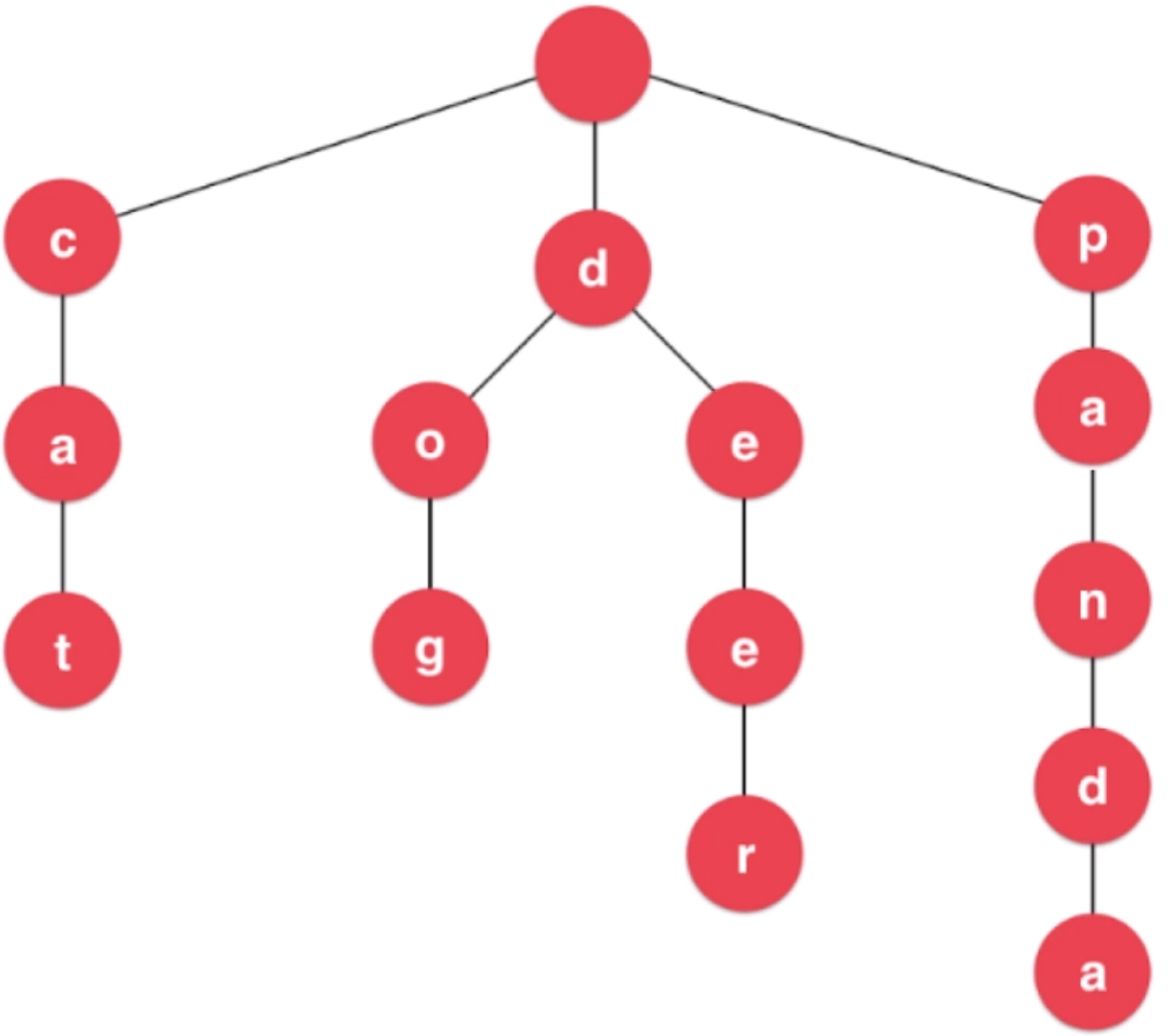
对于区间操作还有一个重要的数据结构：树状数组（Binary Index Tree）

C10 Trie 字典树

1. 什么是字典树（前缀树，Trie）

   Trie只用来处理字符串

字典	Trie
如果有n个条目，使用树结构，查询的时间复杂度是O(logn)，如果有100万个条目（2^20），logn大约为20	查询每个条目的时间复杂度和字典中一共有多少条目无关，时间复杂度为O(w)，w为查询单词的长度



上图中每一条都是一个单词，如图所示，查找一个单词只和单词的长度相关。

每个节点有若干个指向下个节点的指针。

```
class Node {
    boolean isWord;
    Map<Character, Node> next;
}
```

```
public class Trie {
```

```
    private class Node {
        public boolean isWord;          //当前节点是否是一个单词的结尾
        public TreeMap<Character, Node> next;
        public Node(boolean isWord) {
            this.isWord = isWord;
            next = new TreeMap<>();
        }
        public Node() {
            this(false);
        }
    }

    private Node root;
    private int size;   //存储了多少单词

    public Trie() {
        root = new Node();
        size = 0;
    }

    //获得Trie中存储的单词数量
    public int getSize() {
        return size;
    }

    public void add(String word) {
        Node cur = root;
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            if (cur.next.get(c) == null) {
                cur.next.put(c, new Node());
            }
            cur = cur.next.get(c);
        }
        if (!cur.isWord) {
            cur.isWord = true;
            size++;
        }
    }

    /**
     * 查询单词word是否在Trie中
     * @param word 要查询的单词
     * @return 单词是否在Trie中
     */
    public boolean contains(String word) {
        Node cur = root;
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            if (cur.next.get(c) == null) {
                return false;

            }
```

```
            cur = cur.next.get(c);
        }
        return cur.isWord;
    }

    /**
     * 查询是否有以prefix为前缀的单词
     * @param prefix 前缀
     * @return
     */
    public boolean startWith(String prefix) {
        Node cur = root;
        for (int i = 0; i < prefix.length(); i++) {
            char c = prefix.charAt(i);
            if (cur.next.get(c) == null) {
                return false;
            }
            cur = cur.next.get(c);
        }
        return true;
    }
}
```

10-7 更多和Trie字典树相关的话题

Trie最大的问题：空间

C11 并查集

一种由孩子指向父亲的树结构，可以解答图中的两点之间是否连同，非常快得判断网络中节点间的连接状态。也是数学中的集合类的实现。

对于一组数据，主要支持两个动作：

- union(p, q)
- isConnected(p, q)

11-2 Quick Find

节点编号	0	1	2	3	4	5	6	7	8	9
所属的集合编号	0	1	0	1	0	1	0	1	0	1

10个数据分为了两个集合，0和2属于同一个集合，所以它们之间是相连接的，0和1之间不连接。把这个步骤抽象一下，就是对于节点p和节点q，看看`find(p) == find(q)`是否成立，如果成立，则p和q是连接的。

Quick Find：

- union(p, q) ==> O(n)
- isConnected(p, q) ==> O(1)

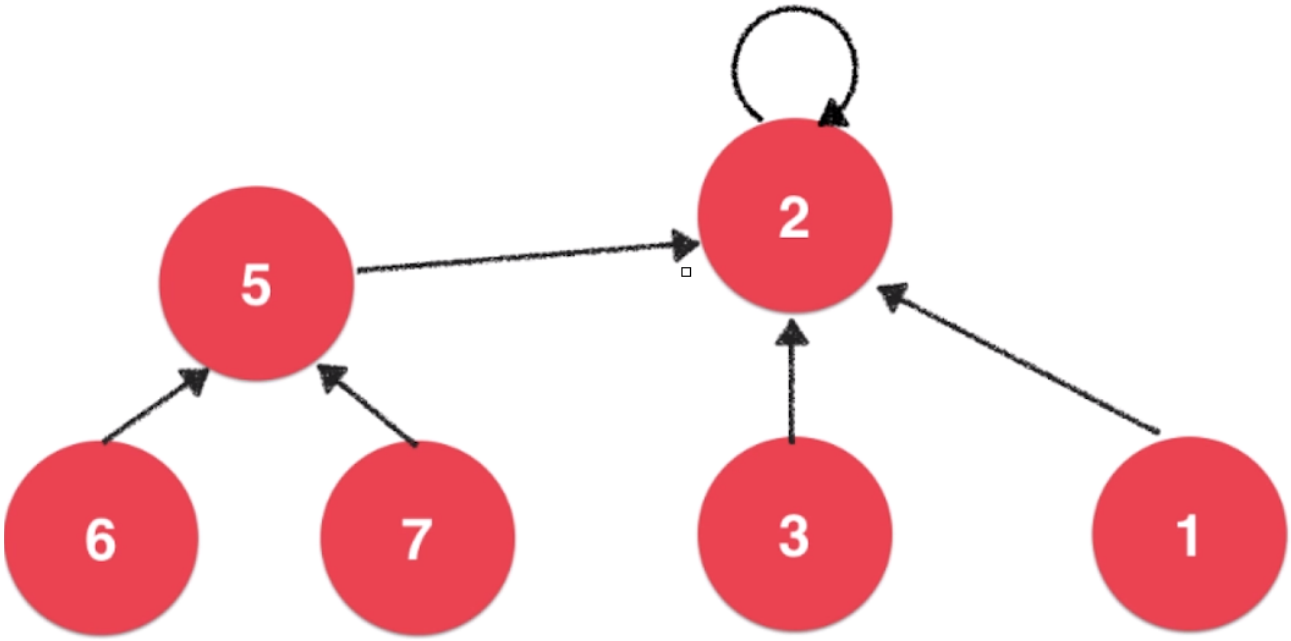
```
public class UnionFind1 implements UF {
```

```
    private int[] id;

    public UnionFind1(int size) {
        id = new int[size];
        for (int i = 0; i < size; i++)
            id[i] = i;          //初始时所有节点之间互不相连，每个元素属于不同集合
    }

    @Override
    public int getSize() { return id.length; }

    /**
     * 查找元素p对应的集合编号
     * @param p 节点
     * @return p所属的集合
     */
    private int find(int p) {
        if (p < 0 || p >= id.length)
            throw new IllegalArgumentException("参数不合法");
        return id[p];
    }

    /**
     * 查看元素p和元素q是否属于一个集合
     * @param p
     * @param q
     * @return
     */
    @Override
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }

    /**
     * 合并元素p和元素q所属的集合
     * @param p 元素p
     * @param q 元素q
     */
    @Override
    public void unioncElement(int p, int q) {
        int commP = find(p);
        int commQ = find(q);
        if (commP == commQ) return;
        for (int i = 0; i < id.length; i++) {
            if (find(i) == commQ)
                id[i] = commP;
        }
    }
}
```

11-3 Quick Union

将每一个元素，看做是一个节点

如果所示，5、6、7属于一个集合，1、2、3属于一个集合，如果要合并3和7所属的集合，只需将7所在的树的根节点，也就是5指向2即可。



Quick Unionc：

- union(p, q) ==> O(h)
- isConnected(p, q) ==> O(h)

h是树的高度

所以还可以用数组来存储，只不过数组中存储的是第i个元素所在的节点指向了哪个元素，所以在初始的时候，每个元素都指向自己。

```
public class UnionFind2 implements UF {

    private int[] parent;

    public UnionFind2(int size) {
        parent = new int[size];
        for (int i = 0; i < size; i++)
            parent[i] = i;          //初始的时候每个节点都指向自己，每个节点都是一颗独立的树
    }

    @Override
    public int getSize() {
        return parent.length;
    }

    private int find(int p) {
        if (p < 0 || p >= parent.length)
            throw new IllegalArgumentException("参数不合法");
        while (parent[p] != p) {
            p = parent[p];
```

```
        }
        return parent[p];
    }

    @Override
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }

    @Override
    public void unioncElement(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot) return;
        parent[pRoot] = qRoot;
    }
}
```

这个实现有一个问题，在极端情况下很有可能形成一个单链表，导致树的深度很大。

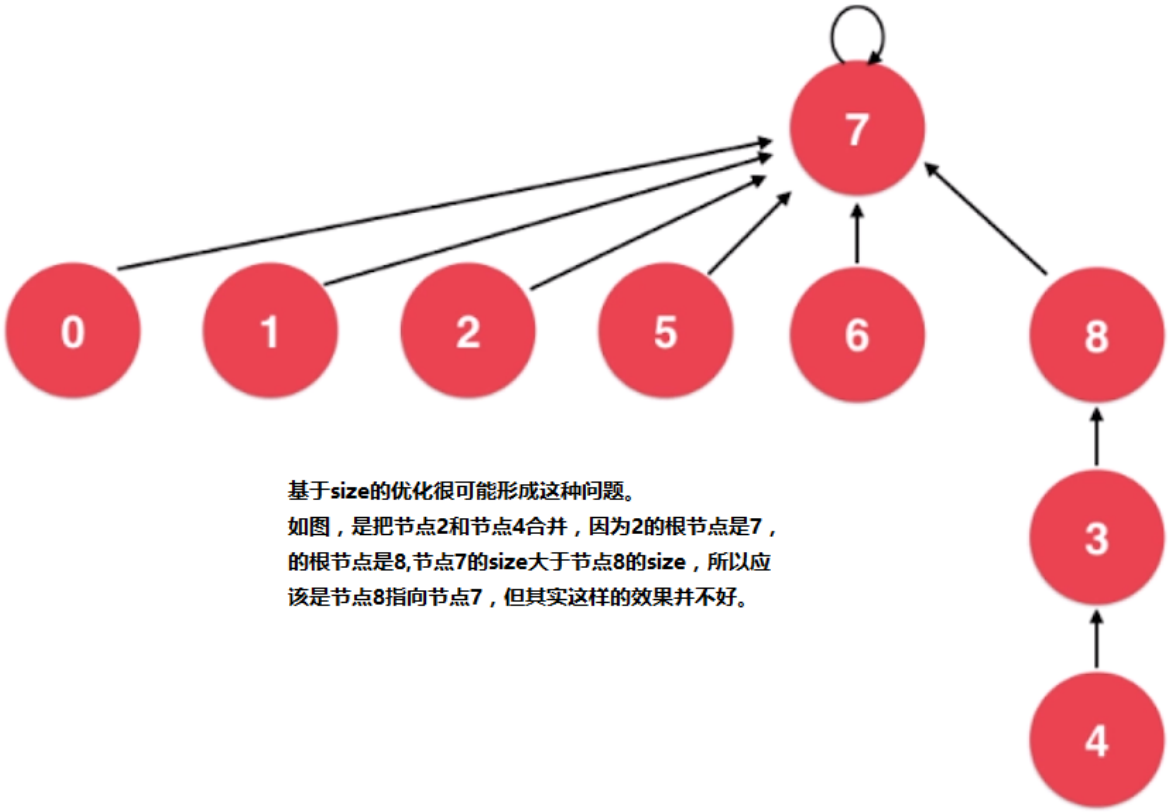
优化方式：

1. 基于size的优化

```
public class UnionFind3 implements UF {
    private int[] parent;
    private int[] sz;           //sz[i]表示以i为根的集合中元素个数

    public UnionFind3(int size) {
        parent = new int[size];
        sz = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;          //初始的时候每个节点都指向自己，每个节点都是一颗独立的树
            sz[i] = 1;
        }
    }

    // 其他操作和UnionFind2一样

    @Override
    public void unioncElement(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot) return;
        if (sz[pRoot] < sz[qRoot]) {
            parent[pRoot] = qRoot;
            sz[qRoot] += sz[pRoot];
        }else {
            parent[qRoot] = pRoot;
            sz[pRoot] += sz[qRoot];
        }
    }
}
```

2. 基于rank的优化

   rank就是指树的高度

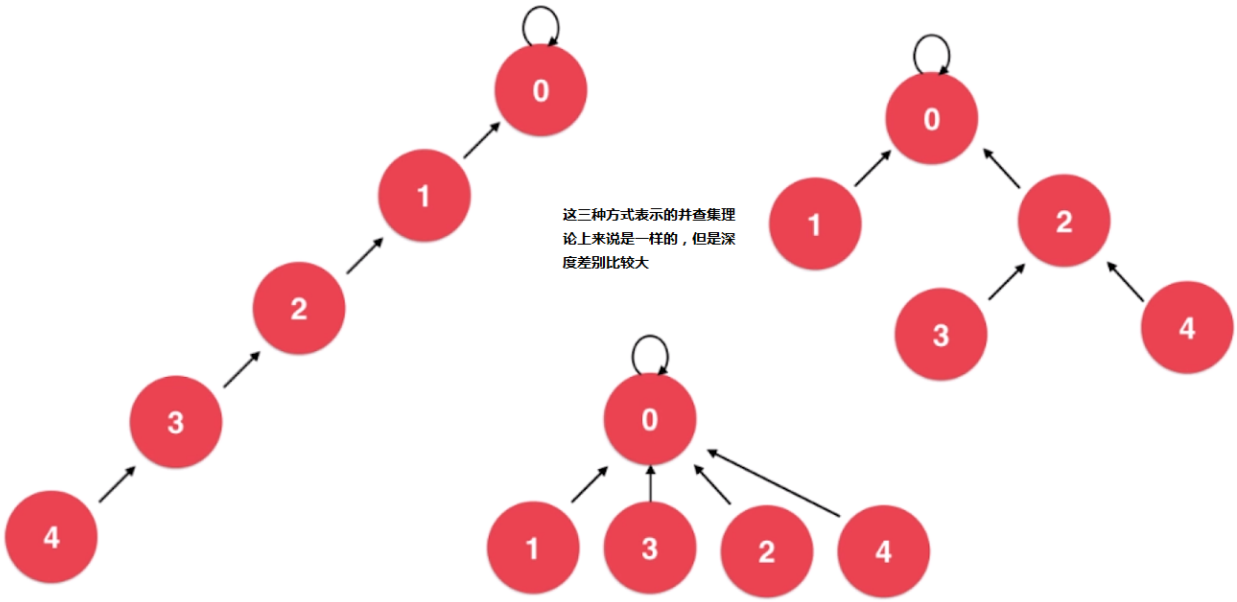


所以更加合理的方式是在每个节点上记录一下以这个节点为根的这棵树的最大深度，在合并时应该是深度比较浅的树指向深度比较深的树。

```
public class UnionFind4 implements UF {
    private int[] parent;
    private int[] rank;           //sz[i]表示以i为根的树的深度

    public UnionFind4(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;          //初始的时候每个节点都指向自己，每个节点都是一颗独立的树
            rank[i] = 1;
        }
    }

    // 其他操作和UnionFind2一样

    @Override
    public void unioncElement(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot) return;
        if (rank[pRoot] < rank[qRoot])
            parent[pRoot] = qRoot;
```

```
        else if (rank[qRoot] < rank[pRoot])
            parent[qRoot] = pRoot;
        else {
            parent[qRoot] = pRoot;
            rank[pRoot] += 1;
        }
    }
}
```

3. 路径压缩



路径压缩可以把一颗比较高的树压缩成比较矮的树。

路径压缩发生在`find()`中：`parent(p) = parent[parent[p]]`

```
public class UnionFind5 implements UF {
    private int[] parent;
    private int[] rank;           //sz[i]表示以i为根的树的深度

    //其他的和UnionFInd4一样

    private int find(int p) {
        if (p < 0 || p >= parent.length)
            throw new IllegalArgumentException("参数不合法");
        while (parent[p] != p) {
            parent[p] = parent[parent[p]];          //把当前节点指向父亲的父亲节点
            p = parent[p];
        }
        return parent[p];
    }
}
```

这个路径压缩只能相对减少树的高度，最理想的情况应该是所有树的高度就只有2，可以通过递归来实现

```
public class UnionFind6 implements UF {
```

```
    private int[] parent;
    private int[] rank;           //sz[i]表示以i为根的树的深度

    /**
     * 得到current这个节点的根节点，并把所有子节点全都直接指向根节点
     * @param p
     * @return p节点的根节点
     */
    private int find(int p) {
        if (p < 0 || p >= parent.length)
            throw new IllegalArgumentException("参数不合法");
        if (parent[p] != p)
            parent[p] = find(parent[p]);
        return parent[p];
    }
}
```

但实际上，UnionFind6的速度没有UnionFind5快，因为递归有时间开销。

并查集的时间复杂度：`O(log*n)`

$$log*n = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + log*(logn), & \text{if } n > 1 \end{cases}$$

比`O(logn)`还快，近乎于`O(1)`。

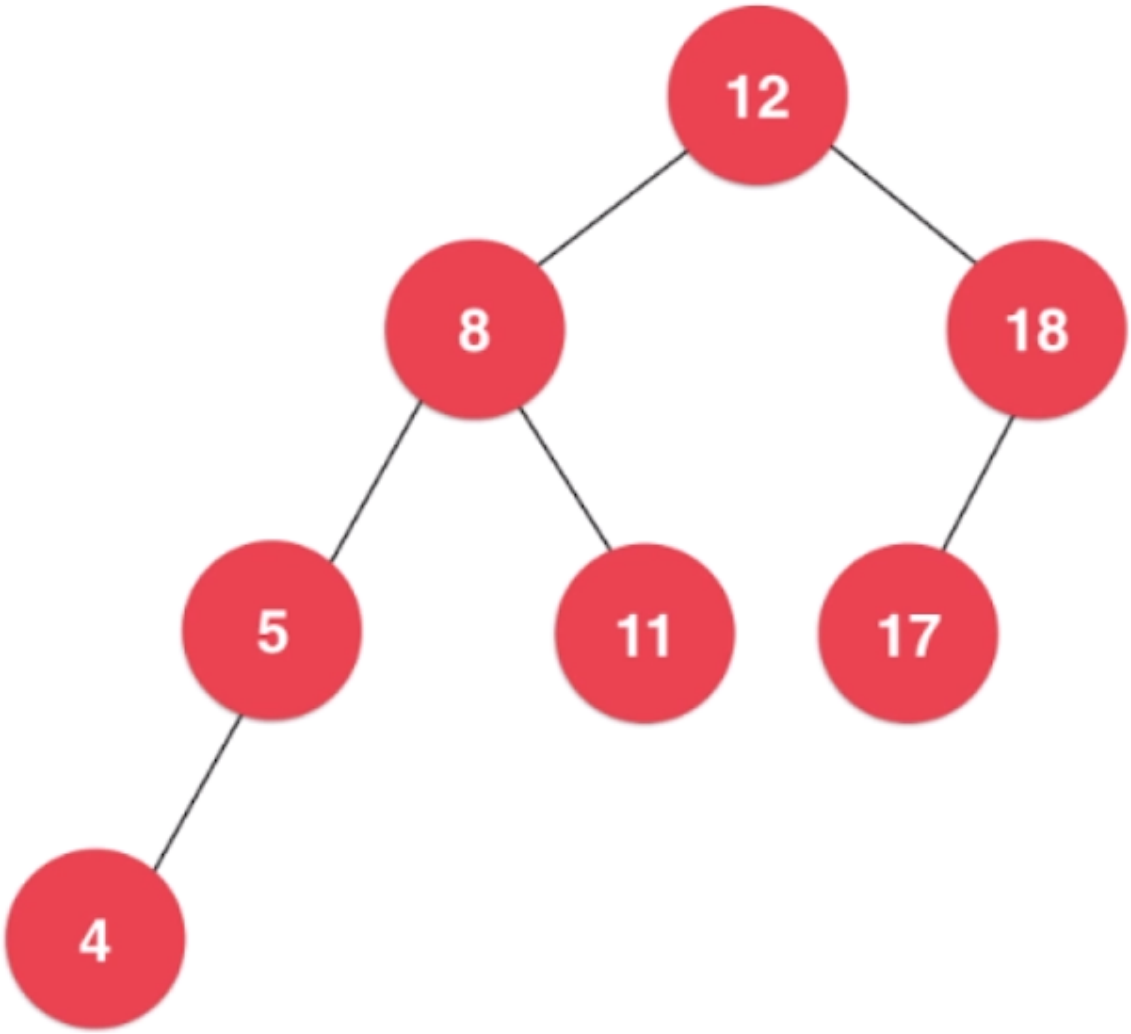
C12 AVL

二分搜索树的问题：如果是按从小到大或者从大到小的顺序添加二分搜索树的话，二分搜索树会退化为单链表。

AVL：最经典的平衡二叉树，最早的自平衡二分搜索树。

一棵满二叉树和完全二叉树一定是平衡二叉树。

- 平衡二叉树：二叉树中，对于任意一个叶子节点，高度差不能超过1。
- AVL中的平衡二叉树：对于任意一个节点，左子树和右子树的高度差不能超过1。



如图，这个满足AVL中的平衡二叉树，但这种结构不会出现在堆或者线段树中。这种平衡二叉树的高度和节点数量之间的关系也是`O(logn)`的。

AVL树既是二分搜索树，也是平衡二叉树。

12-4 旋转操作的基本原理

AVL树的左旋转和右旋转。

插入或者删除一个节点才有可能破坏平衡性，所以维护平衡的时机，**应该是加入节点后，沿着节点向上维护平衡性。**

```
public class AVLTree<K extends Comparable<K>, V> {
    private class Node{
        public K key;
        public V value;
        public Node left, right;
        public int height;          //这个节点当前所处的高度

        public Node(K key, V value){
            this.key = key;

            this.value = value;
```

```
            left = null;
            right = null;
            height = 1;
        }
    }

    private Node root;
    private int size;

    public AVLTree(){
        root = null;
        size = 0;
    }

    public int getSize(){
        return size;
    }

    public boolean isEmpty(){
        return size == 0;
    }

    private int getHeight(Node node) {
        if (node == null) return 0;
        return node.height;
    }

    /**
     * @param node 节点node
     * @return 节点node的平衡银子
     */
    private int getBalanceFactor(Node node) {
        if (node == null) return 0;
        return getHeight(node.left) - getHeight(node.right);
    }

    /**
     * 判断该二叉树是否是一颗二分搜索树
     * @return true:是二分搜索树；false:不是二分搜索树
     */
    public boolean isBST() {
        ArrayList<K> keys = new ArrayList<>();
        inOrder(root, keys);        //对一个二分搜索树，中序遍历后的列表是从小到大的
        for (int i = 1; i < keys.size(); i++) {
            if (keys.get(i - 1).compareTo(keys.get(i)) > 0)
                return false;
        }
        return true;
    }

    /**
     * 二分搜索树的中序遍历

     * @param root 二分搜索树的根节点
```

```
     * @param keys 遍历后存储节点的列表
     */
    private void inOrder(Node root, ArrayList<K> keys) {
        if (root == null)
            return;
        inOrder(root.left, keys);
        keys.add(root.key);
        inOrder(root.right, keys);
    }

    /**
     * 判断该二叉树是否是一颗平衡二叉树
     * @return true:是平衡二叉树；false:不是平衡二叉树
     */
    public boolean isBalanced() {
        return isBalanced(root);
    }

    private boolean isBalanced(Node node) {
        if (node == null)
            return true;
        int balanceFactor = getBalanceFactor(node);
        if (Math.abs(balanceFactor) > 1)
            return false;
        return isBalanced(node.left) && isBalanced(node.right);
    }

    // 向二分搜索树中添加新的元素(key, value)
    public void add(K key, V value){
        root = add(root, key, value);
    }

    // 向以node为根的二分搜索树中插入元素(key, value)，递归算法
    // 返回插入新节点后二分搜索树的根
    private Node add(Node node, K key, V value){

        if(node == null){
            size ++;
            return new Node(key, value);
        }

        if(key.compareTo(node.key) < 0)
            node.left = add(node.left, key, value);
        else if(key.compareTo(node.key) > 0)
            node.right = add(node.right, key, value);
        else // key.compareTo(node.key) == 0
            node.value = value;

        //更新height
        node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));

        //计算平衡因子

        int balanceFactor = getBalanceFactor(node);
```

```
        //插入的元素在不平衡的节点的左侧的左侧：LL
        if (balanceFactor > 1 && getBalanceFactor(node.left) >= 0)
            // 说明左子树比右子树高，而且左子树的左儿子的高度大于等于右儿子的高度
            return rightRotate(node);

        //插入的元素在不平衡的节点的右侧的右侧：RR
        if (balanceFactor < -1 && getBalanceFactor(node.right) <= 0)
            return leftRotate(node);

        /*
        *                 y                                                   y
        *         |               |                                   |               |
        *         x               T4                                  z               T4
        *     |       |                           ==>               |           |
        *     T1      z                                             x           T3
        *          |     |                                      |       |
        *          T2    T3                                     T1      T2
        *
        * 1. 先对x进行左旋转
        * 2. 转化为了LL的情况
        */
        //插入的元素在不平衡的节点的左侧的右侧：LR
        if (balanceFactor > 1 && getBalanceFactor(node.left) < 0) {
            node.left =  leftRotate(node.left);
            return rightRotate(node);
        }

        //插入的元素在不平衡的节点的右侧的左侧：RL
        if (balanceFactor < -1 && getBalanceFactor(node.right) > 0) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }
        return node;
    }

    //左旋转
    private Node leftRotate(Node y) {
        /*
        * 对节点y进行向左旋转操作，返回旋转后新的根节点x
        *    y                              x
        *  /  \                           /   \
        * T1   x      向左旋转 (y)       y     z
        *     / \   - - - - - - - ->    / \   / \
        *   T2  z                     T1 T2 T3 T4
        *      / \
        *     T3 T4
        */
        Node x = y.right;
        Node T2 = x.left;
        x.left = y;
        y.right = T2;
```

```
        //更新height
        y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;
        x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;
        return x;
    }

    // 右旋转
    private Node rightRotate(Node y) {
        /*
         * 如下图所示，y是不平衡节点，x的平衡因子是>=0的。T1 < z < T2 < x < T3 < y <T4
         *                  y                                               x
         *         |               |                                |               |
         *         x               T4                               z               y
         *      |      |                          ==>              |      |      |      |
         *      z      T3                                          T1     T2     T3     T4
         *   |     |
         *   T1    T2
         * y这个节点的左子树过高，希望通过一定步骤，保持y这个节点的平衡性：右旋转
         * 1. x.right = y
         * 2. y.left = T3
         * 3. return x
         * 这时得到的x这个节点既满足二分搜索树，也满足平衡二叉树
         *
         * 证明：
         *      左图中：
         *          因为y是第一个不平衡节点，所以z为根的树是平衡二叉树，说明T1和T2的高度差不会超过1，h = max(height(T1, T2)), height(z) = h + 1
         *          x也是平衡节点，0 <= factor(x) <= 1，所以height(T3) = h + 1 或者 h，所有height(x) = x + 2
         *          因为y本来是平衡的，但是添加了一个节点之后不平衡了，所以unbalance_factor(y) = 2，所以height(T4) = h
         *      右图中：
         *          height(z) = h + 1，
         *          因为height(T3) = h + 1 或者 h，height(T4) = h，所以height(y) = h + 2 或者 h + 1，此时y也是平衡的
         *          unbalance_factor = abs(height(z) - height(y)) = 0 或者 1，所以x也是平衡的
         */
        Node x = y.left;
        Node T3 = x.right;
        x.right = y;
        y.left = T3;

        //更新height
        y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;
        x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;
        return x;
    }

    // 返回以node为根节点的二分搜索树中，key所在的节点
    private Node getNode(Node node, K key){

        if(node == null)

            return null;
```

```
        if(key.equals(node.key))
            return node;
        else if(key.compareTo(node.key) < 0)
            return getNode(node.left, key);
        else // if(key.compareTo(node.key) > 0)
            return getNode(node.right, key);
    }

    public boolean contains(K key){
        return getNode(root, key) != null;
    }

    public V get(K key){

        Node node = getNode(root, key);
        return node == null ? null : node.value;
    }

    public void set(K key, V newValue){
        Node node = getNode(root, key);
        if(node == null)
            throw new IllegalArgumentException(key + " doesn't exist!");

        node.value = newValue;
    }

    // 返回以node为根的二分搜索树的最小值所在的节点
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }

    // 从二分搜索树中删除键为key的节点
    public V remove(K key){

        Node node = getNode(root, key);
        if(node != null){
            root = remove(root, key);
            return node.value;
        }
        return null;
    }

    private Node remove(Node node, K key){

        if(node == null)
            return null;
        Node retNode = null;
        if(key.compareTo(node.key) < 0){
            node.left = remove(node.left , key);

            retNode = node;
```

```
        }
        else if(key.compareTo(node.key) > 0 ){
            node.right = remove(node.right, key);
            retNode = node;
        }
        else{   // key.compareTo(node.key) == 0

            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                retNode = rightNode;
            } else if(node.right == null){      // 待删除节点右子树为空的情况
                Node leftNode = node.left;
                node.left = null;
                size --;
                retNode = leftNode;
            } else {
                // 待删除节点左右子树均不为空的情况

                // 找到比待删除节点大的最小节点，即待删除节点右子树的最小节点
                // 用这个节点顶替待删除节点的位置
                Node successor = minimum(node.right);
                successor.right = remove(node.right, successor.key);
                successor.left = node.left;
                node.left = node.right = null;
                retNode = successor;
            }
        }

        if (retNode == null)
            return null;

        //更新height
        retNode.height = 1 + Math.max(getHeight(retNode.left), getHeight(retNode.right));

        //计算平衡因子
        int balanceFactor = getBalanceFactor(retNode);

        //插入的元素在不平衡的节点的左侧的左侧：LL
        if (balanceFactor > 1 && getBalanceFactor(retNode.left) >= 0)
            // 说明左子树比右子树高，而且左子树的左儿子的高度大于等于右儿子的高度
            return rightRotate(retNode);

        //插入的元素在不平衡的节点的右侧的右侧：RR
        if (balanceFactor < -1 && getBalanceFactor(retNode.right) <= 0)
            return leftRotate(retNode);

        //插入的元素在不平衡的节点的左侧的右侧：LR
        if (balanceFactor > 1 && getBalanceFactor(retNode.left) < 0) {
            retNode.left =  leftRotate(retNode.left);

            return rightRotate(retNode);
```

```
        }

        //插入的元素在不平衡的节点的右侧的左侧：RL
        if (balanceFactor < -1 && getBalanceFactor(retNode.right) > 0) {
            retNode.right = rightRotate(retNode.right);
            return leftRotate(retNode);
        }
        return retNode;
    }

    public static void main(String[] args) {
        System.out.println("Pride and Prejudice");

        ArrayList<String> words = new ArrayList<>();
        if(FileOperation.readFile("pride-and-prejudice.txt", words)) {
            System.out.println("Total words: " + words.size());

            AVLTree<String, Integer> map = new AVLTree<>();
            for (String word : words) {
                if (map.contains(word))
                    map.set(word, map.get(word) + 1);
                else
                    map.add(word, 1);
            }

            System.out.println("Total different words: " + map.getSize());
            System.out.println("Frequency of PRIDE: " + map.get("pride"));
            System.out.println("Frequency of PREJUDICE: " + map.get("prejudice"));

            System.out.println("Is BST: " + map.isBST());
            System.out.println("Is Balanced: " + map.isBalanced());

            for (String word : words) {
                map.remove(word);
                if (!map.isBST() || !map.isBST())
                    throw new IllegalArgumentException("aoo,出错了");
            }
        }
    }
}
```

AVL的优化：

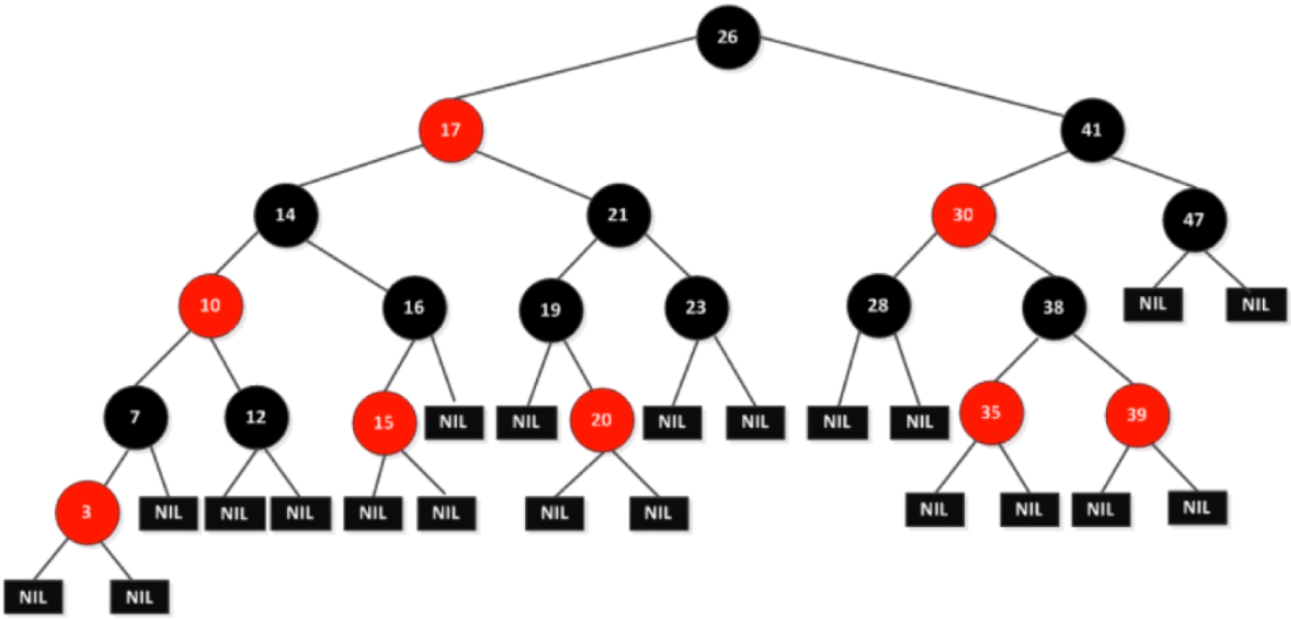
- 如果添加或者删除后的节点高度没有变化，那么该节点的祖宗节点就不需要维护平衡。

AVL树的局限：

- 平均来说，红黑树比AVL树的性能更好一些。

C13 红黑树

13-1 红黑树与2-3树

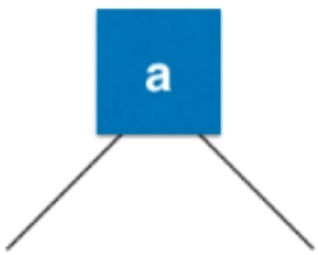


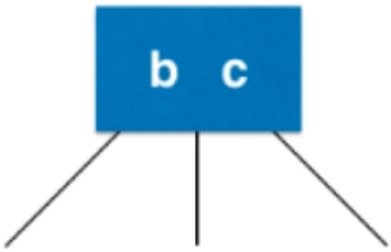
> 红黑树是二分搜索树和平衡二叉树。

> 红黑树与2-3树是等价的。

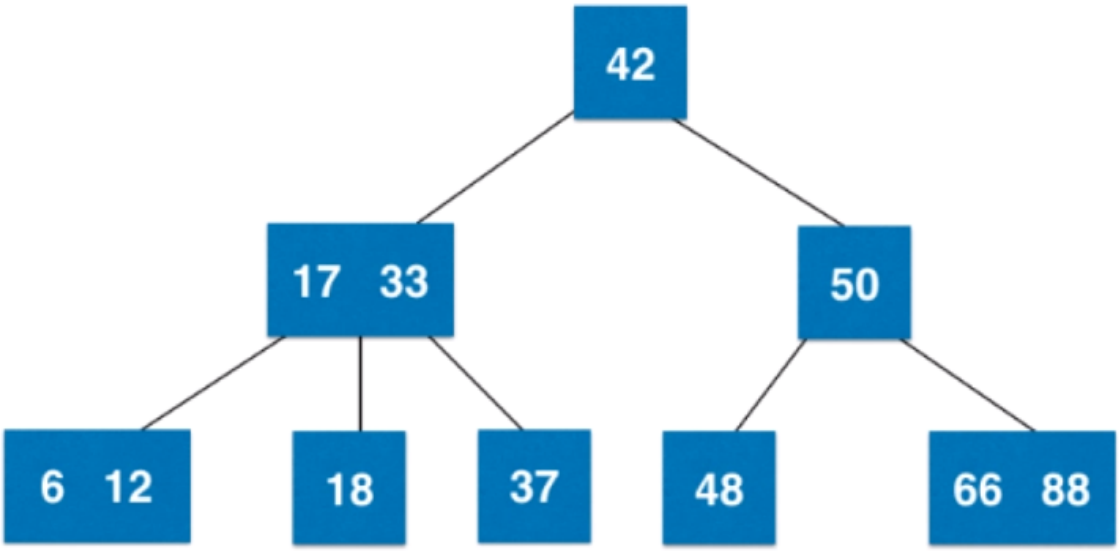
2-3树：

- 满足二分搜索树的基本性质，但不是二叉树
- 节点可以存放一个元素（2个孩子）或者两个元素（3个孩子）





  - 2节点：存放1个元素，和二分搜索树一样
  - 3节点：存放2个元素，左子树小于b，右子树大于c，中间的子数介于b和c之间。



2-3树的性质：

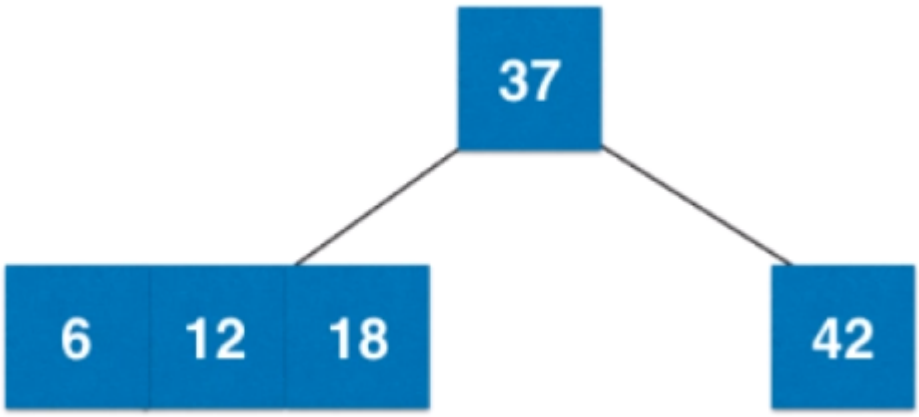
- 2-3树是一颗绝对平衡的树，AVL树虽然也是平衡二叉树，但是条件比这个绝对平衡宽松。对于2-3树来说，任意节点的左右子数的高度一定是相等的。
- 2-3树的绝对平衡性是由于它**添加节点的机制**。

13-2 2-3树的绝对平衡性

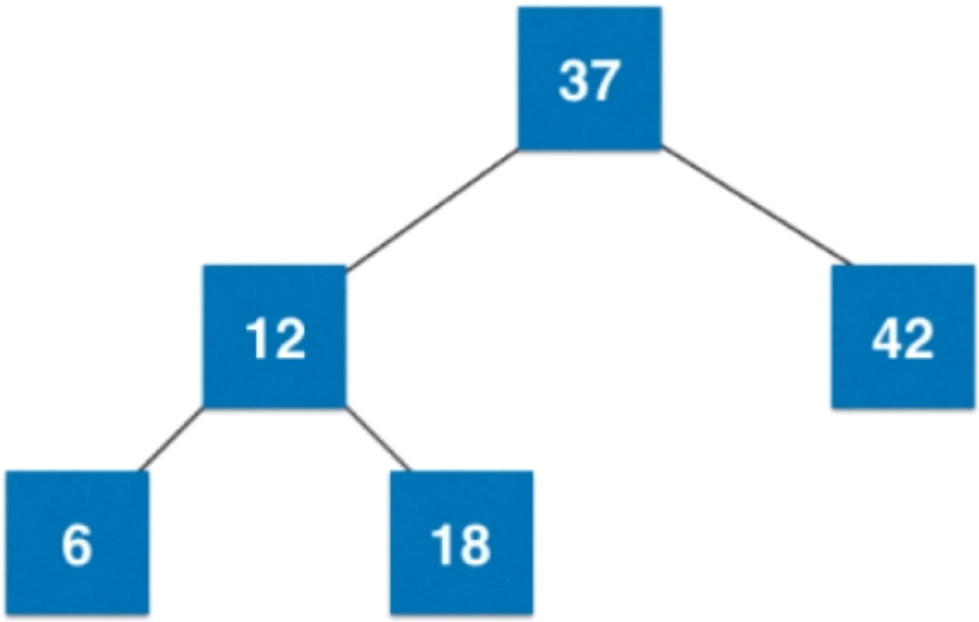
2-3树的添加机制。

2-3树添加节点不会添加到一个空的位置，而是添加到找到的最后一个叶子节点上。

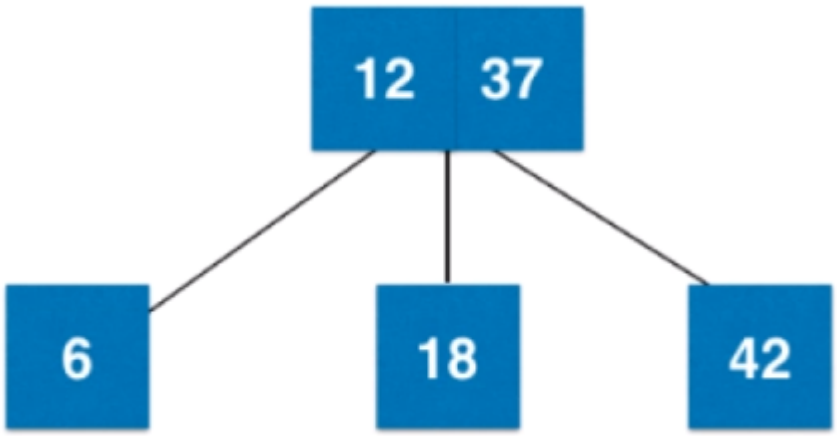
假设树为空，添加第一个节点42，此时42为根节点，再添加一个节点37，因为37小于42，42的左子树为空，所以把37和42融合一起变成一个3节点。此时向树中再添加一个元素12，先把12和37和42融合，变成一个4节点，再把它分裂成一个由3个2节点组成的树（根节点是37，左子树是12，右子树是42）。再来一个节点18，应该把它添加到12的右子树，但是2-3树不能添加到空位置，所以把12和18融合，组成一个3节点。此时再来一个节点6，和12和18融合成一个4节点，



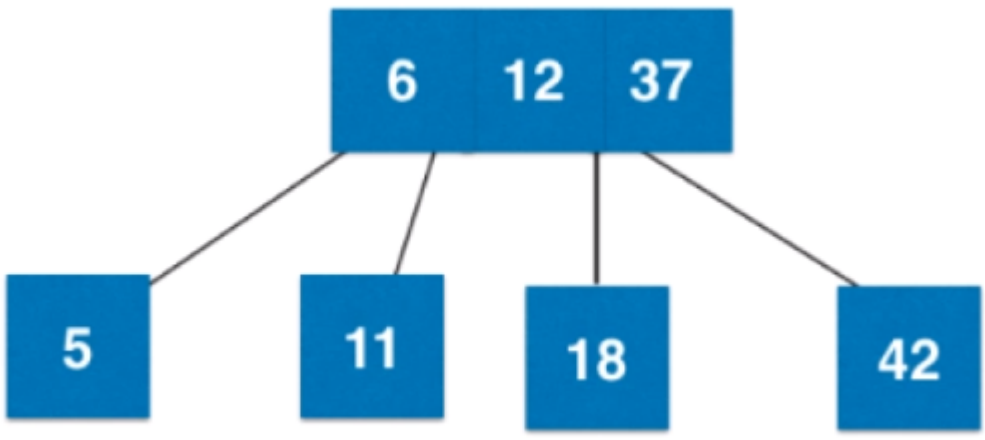
再把4节点拆解



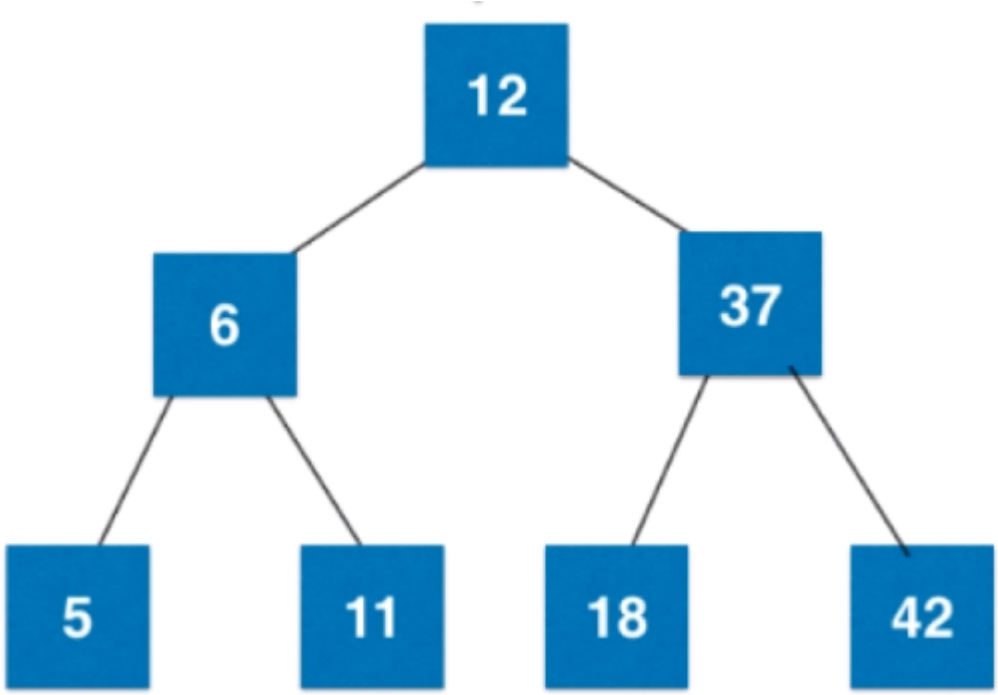
此时不是一颗平衡树，所以拆解后的子数的根节点12要向父节点融合。



再添加元素11，于是就和6融合形成6-11的3节点，再来一个节点5，融合成5-6-11的4节点，4节点拆分，节点6和12-37节点融合形成6-12-37的4节点，

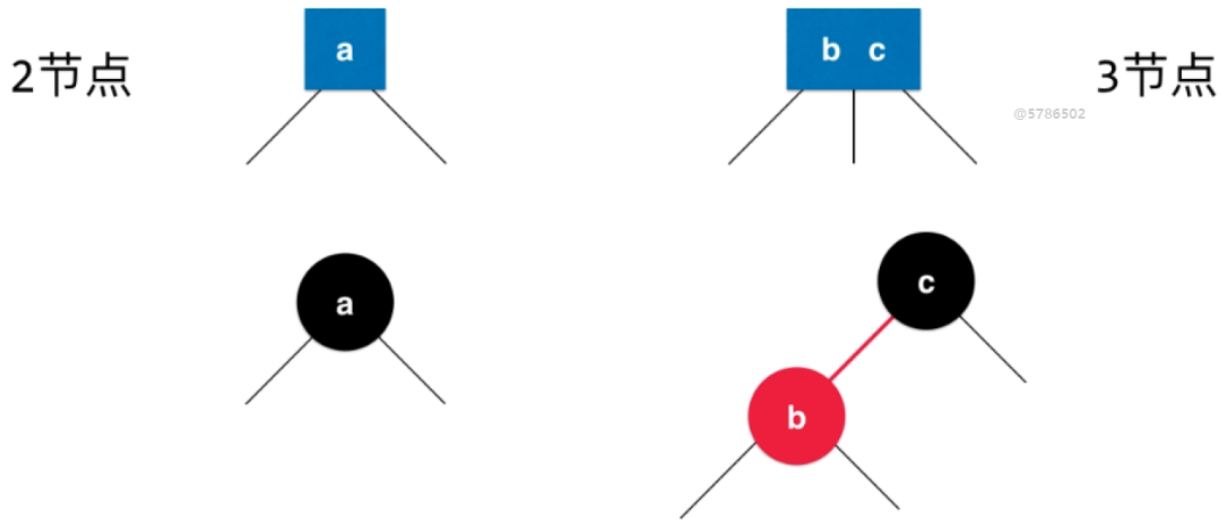


把4节点拆分，



13-3 红黑树与2-3树的等价性

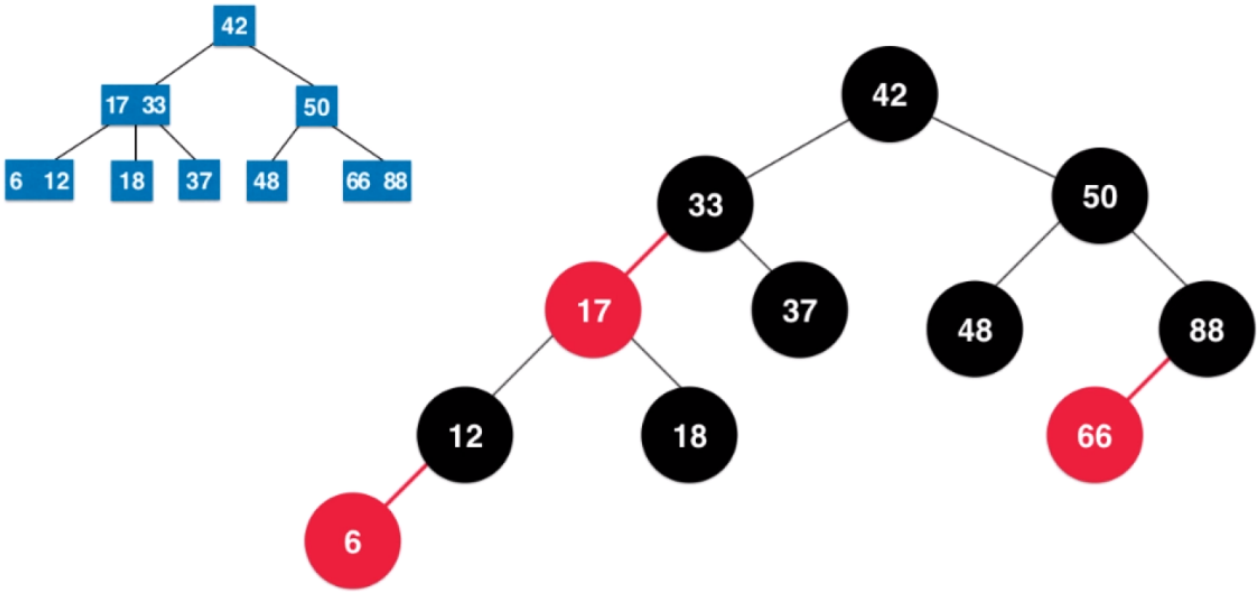
在红黑树上，用这样两种方式表示2-3树中的2节点和3节点



红色的节点表示和他的父节点一起表示2-3树中的3节点。

> 在红黑树中，所有的红色节点都是左倾斜的，因为b要小于c。

如下图，左边的2-3树和右边的红黑树是等价的。



对于任意的2-3树，都可以用这种方式来将它转化为红黑树。

13-4 红黑树的基本性质和复杂度分析

算法导论中红黑树满足的性质：

- 每个节点或者是红色的，或者是黑色的
- 根节点是黑色的

  由2-3树可知，根节点要么是2节点要么是3节点，不管是2节点还是3节点对应的红黑树的根节点但是黑色的。
- 每一个叶子节点（最后的空节点，不是左右子树都为空的节点）是黑色的

  空树本身也是红黑树，此时根节点本身为空，也是黑色的。
- 如果一个节点是红色，那么它的孩子节点都是黑色的

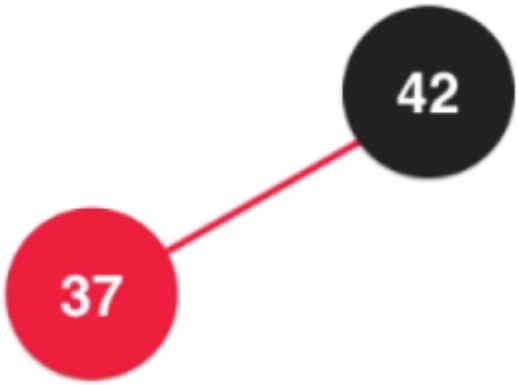
  红色节点的孩子节点可以看作是它孩子的根节点，由性质2可知，所有红色节点的孩子节点一定是黑色的。如果一个节点时黑色的，那么它的右孩子一定四黑色的。
- 从任意一个节点到叶子节点，经过的黑色节点是一样的

  2-3树是一个绝对平衡的树，意味着从2-3树一个节点出发到任意一个叶子节点经过的节点数是一样的，比如上图中从根节点42出发到所有叶子节点都经过2个节点。2-3中不管是2节点还是3节点，对应到红黑树中都会有一个黑色节点，所以就意味着从红黑树任意一个节点到叶子节点，经过的黑色节点数量是一样的。

红黑树时保存”黑平衡“的二叉树，严格意义上，不是平衡二叉树。最大高度：`2logn`，所以查询操作比AVL树稍微慢一些，但是增和删比AVL树高效。

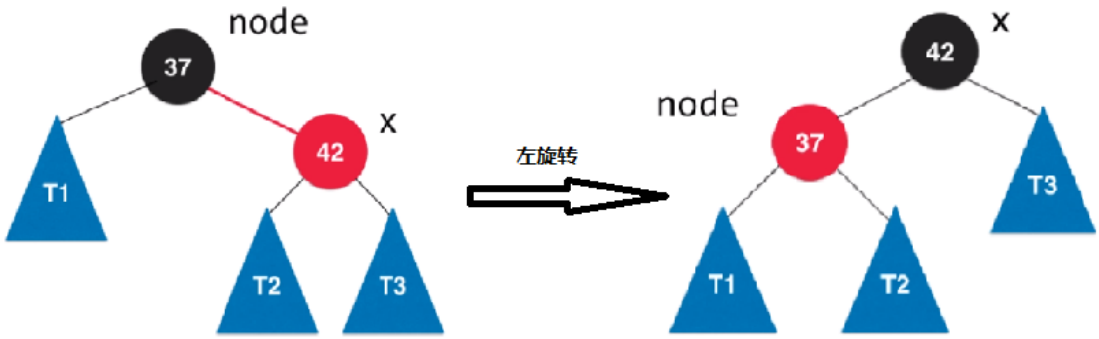
13-5 向红黑树中添加新元素

- 向红黑树中的”2-node“节点添加元素
  - 在红黑树中添加一个值37，要添加的值在红黑树的左侧，此时不需要做别的工作。

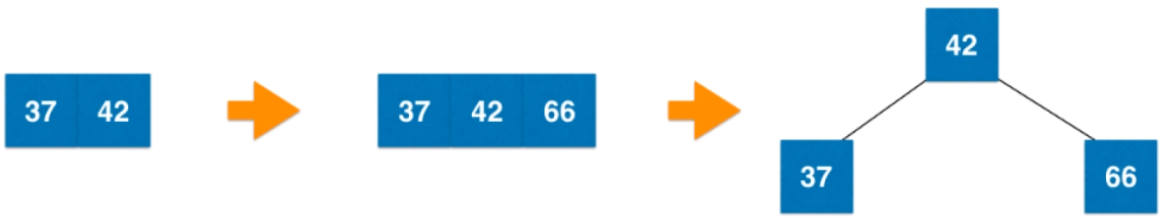


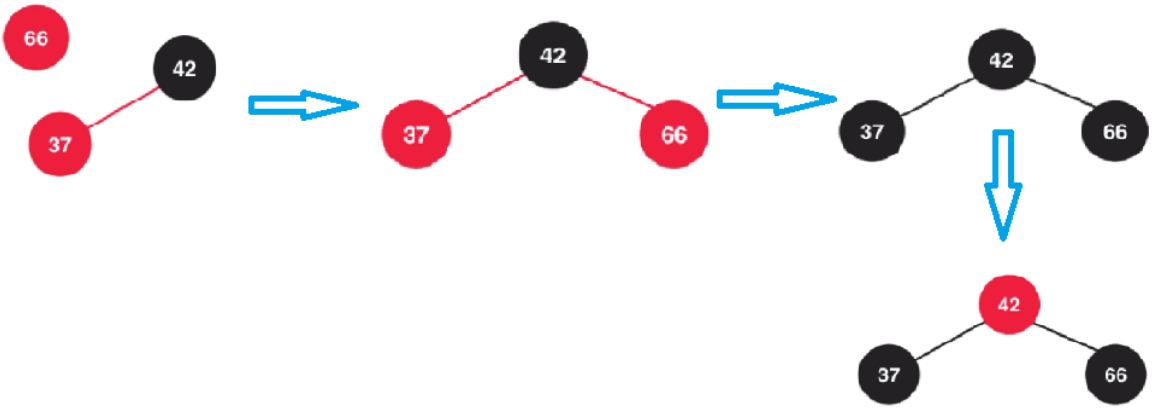
添加节点时默认颜色为红色（2-3树中添加一个节点时，添加的节点永远是和叶子节点融合，所以新添加的节点总是先融合，所以是默认是红色的）。

- 在红黑树中添加一个值42，要添加的值在红黑树的右侧，此时不满足红黑树的定义（红黑树的红色节点都是偏左的），需要进行左旋转。
  - `node.right = x.left;`
  - `x.left = node;`
  - `x.color = node.color;`
  - `node.color = RED; //表示要和父节点融合在一起`
  - `// 因为是2节点，所以node节点的起始颜色肯定是黑色的`

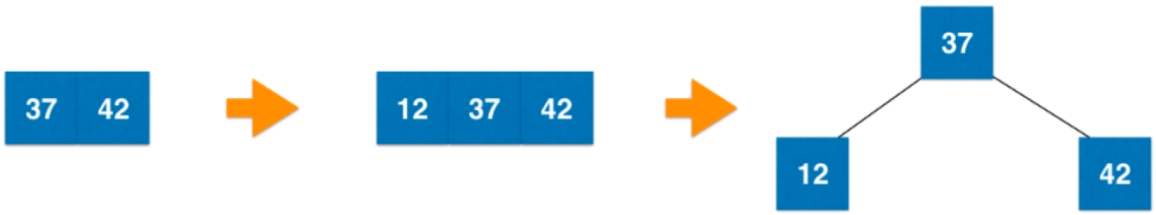


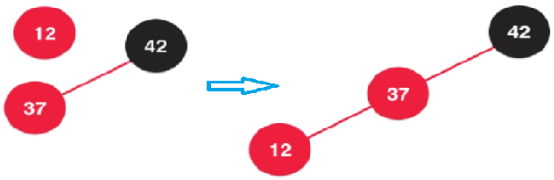
- 向红黑树中的”3-node“节点添加元素
  - 如下图，在37-42这个红黑树中添加元素66，在2-3树中会短暂地融合成一个4节点，然后拆分成3个2节点，对应在红黑树中，应该是3个黑色的节点。然后此时42节点要和它们之前的父节点进行融合，融合意味着42节点要变成红节点。

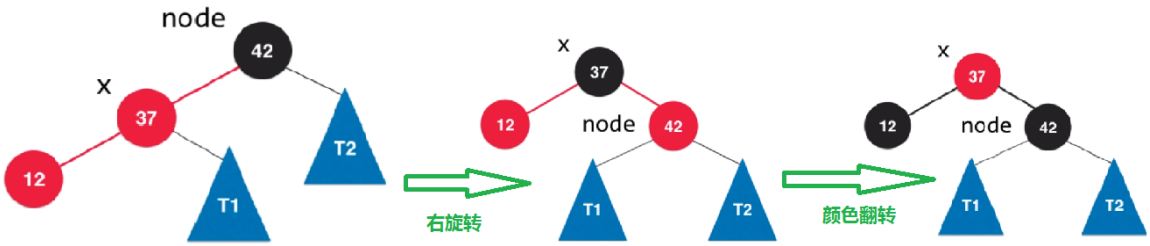




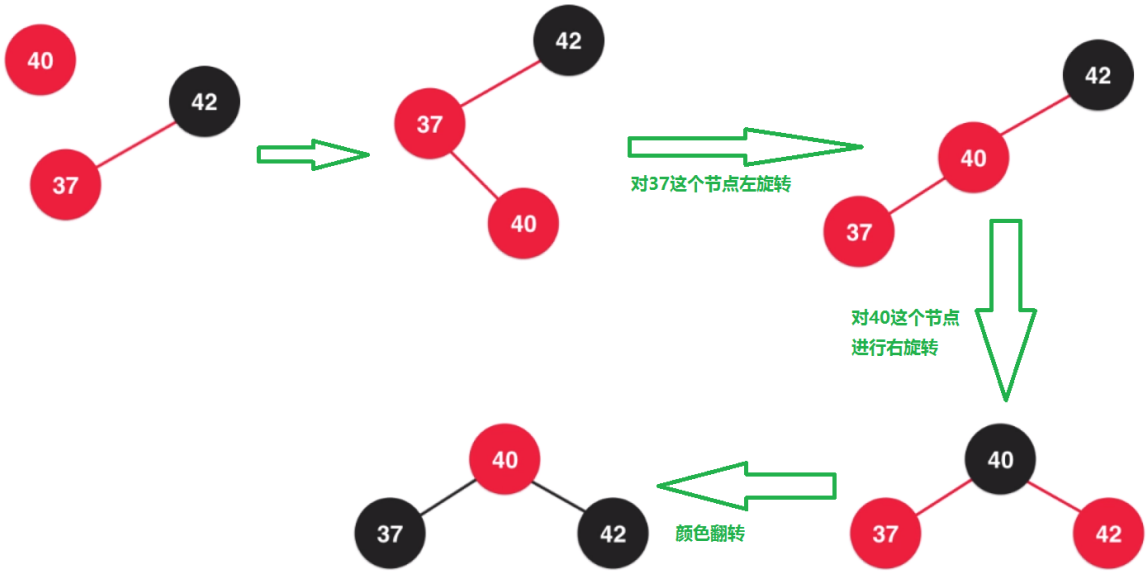
- 如下图，在37-42这个红黑树中添加新节点12， 此时会进行右旋转。
  - `node.left = T1;`
  - `x.right = node;`
  - `x.color = node.color;`
  - `node.color = RED;  //表示要和父节点融合在一起`



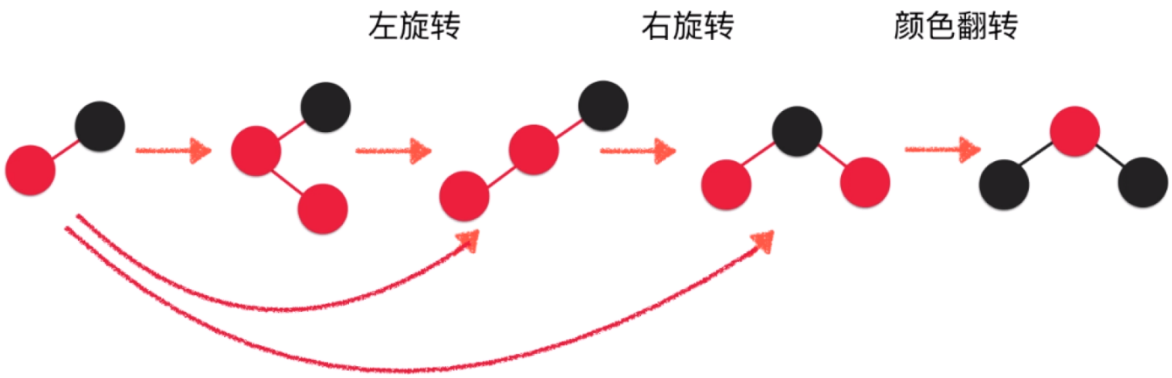




- 在37-42这个红黑树中添加元素37



- **这三种情况总结下来就是**：



不管向“2-node”还是向“3-node”中添加节点，如果一个节点的右孩子是红色的左孩子是黑色的（空节点也是黑色的），则对该节点进行左旋转。

> 对于完全随机的数据，普通的二分搜索树很好用，缺点：极端情况下退化成链表（或者高度不平衡）
>
> 对于查询较多的使用情况，AVL树很好用
>
> 红黑树牺牲了平衡性（ `2logn` 的高度）

红黑树的**统计性能更优**（综合增删改查所有的操作）

```
public class RBTree<K extends Comparable<K>, V> {

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

    private class Node{
        public K key;
        public V value;
        public Node left, right;
        public int height;          //这个节点当前所处的高度
        public boolean color;


        public Node(K key, V value){
```

```
            this.key = key;
            this.value = value;
            left = null;
            right = null;
            height = 1;
            color = RED; // 2-3树中添加一个节点时，添加的节点永远是和叶子节点融合，所以新添加的节点总是先融合，所以是默认是红色的。
        }
    }

    private Node root;
    private int size;

    public RBTree(){
        root = null;
        size = 0;
    }

    public int getSize(){
        return size;
    }

    public boolean isEmpty(){
        return size == 0;
    }

    private boolean isRed(Node node) {
        if (node == null)
            return BLACK;
        return node.color;
    }

    /*
    *   node                     x
    *  /   \     左旋转         /  \
    * T1   x   --------->   node   T3
    *     / \              /   \
    *    T2 T3            T1   T2
    */
    private Node leftRotate(Node node) {
        Node x = node.right;
        node.right = x.left;
        x.left = node;
        x.color = node.color;
        node.color = RED;
        return x;
    }

    /*     node                   x
    *    /   \     右旋转       /  \
    *   x    T2   ------->   y   node
    *  / \                       /  \

    * y  T1                     T1  T2
```

```
    */
    private Node rightRotate(Node node) {
        Node x = node.left;
        node.left = x.right;
        x.right = node;
        x.color = node.color;
        node.color = RED;

        return x;
    }

    /*
     *  调用这个方法时要保证node必须有左孩子和右孩子：
     *       node
     *      /    \
     *  left     right
     */
    // 颜色反转
    private void flipColors(Node node) {
        node.color = RED;
        node.left.color = BLACK;
        node.right.color = BLACK;
    }

    // 向二分搜索树中添加新的元素(key, value)
    public void add(K key, V value){
        root = add(root, key, value);
        root.color = BLACK;         //保持最终根节点为黑色节点
    }

    // 向以node为根的二分搜索树中插入元素(key, value)，递归算法
    // 返回插入新节点后二分搜索树的根
    private Node add(Node node, K key, V value){

        if(node == null){
            size ++;
            return new Node(key, value);
        }

        if(key.compareTo(node.key) < 0)
            node.left = add(node.left, key, value);
        else if(key.compareTo(node.key) > 0)
            node.right = add(node.right, key, value);
        else // key.compareTo(node.key) == 0
            node.value = value;

        if (isRed(node.right) && !isRed(node.left))
            node = leftRotate(node);

        if (isRed(node.left) && isRed(node.left.left))
            node = rightRotate(node);


        if (isRed(node.left) && isRed(node.right))
```

```
            flipColors(node);

        return node;
    }

    // 返回以node为根节点的二分搜索树中，key所在的节点
    private Node getNode(Node node, K key){

        if(node == null)
            return null;

        if(key.equals(node.key))
            return node;
        else if(key.compareTo(node.key) < 0)
            return getNode(node.left, key);
        else // if(key.compareTo(node.key) > 0)
            return getNode(node.right, key);
    }

    public boolean contains(K key){
        return getNode(root, key) != null;
    }

    public V get(K key){

        Node node = getNode(root, key);
        return node == null ? null : node.value;
    }

    public void set(K key, V newValue){
        Node node = getNode(root, key);
        if(node == null)
            throw new IllegalArgumentException(key + " doesn't exist!");

        node.value = newValue;
    }

    // 返回以node为根的二分搜索树的最小值所在的节点
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }

    // 删除掉以node为根的二分搜索树中的最小节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMin(Node node){

        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;

            return rightNode;
```

```
        }

        node.left = removeMin(node.left);
        return node;
    }

    // 从二分搜索树中删除键为key的节点
    public V remove(K key){
        // 红黑树的删除操作比添加操作更加复杂
        return null;
    }

    private Node remove(Node node, K key){

        if( node == null )
            return null;

        if( key.compareTo(node.key) < 0 ){
            node.left = remove(node.left , key);
            return node;
        }
        else if(key.compareTo(node.key) > 0 ){
            node.right = remove(node.right, key);
            return node;
        }
        else{   // key.compareTo(node.key) == 0

            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                return rightNode;
            }

            // 待删除节点右子树为空的情况
            if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                return leftNode;
            }

            // 待删除节点左右子树均不为空的情况

            // 找到比待删除节点大的最小节点，即待删除节点右子树的最小节点
            // 用这个节点顶替待删除节点的位置
            Node successor = minimum(node.right);
            successor.right = removeMin(node.right);
            successor.left = node.left;

            node.left = node.right = null;
```

```
                return successor;
            }
        }
    }
```

Java.util中的TreeMap和TreeSet都是基于红黑树。

实际上，这里实现的是左倾红黑树，但是算法导论中关于红黑树的定义并没有红节点一定左倾这一条，换句话说，左倾红黑树一定是红黑树，但是红黑树不一定是左倾的，根节点是黑色，左右孩子都是红色也是标准定义下的红黑树，但不满足这里的实现。

这里实现的左倾红黑树为了满足左倾的性质，做了额外的事情，消耗了性能，所以没有标准红黑树中“任何不平衡都可以在三次选择内解决”的性质。

C14 哈希表

14-1 哈希表基础

什么是哈希表？

一个关心的类型和一个索引出现一个映射关系。把关心的类型转换成索引的函数就是哈希函数。比如：

$$f(ch) = ch - 'a'，对于给定的一个字符 ch，把它转换成索引$$

这个就是把a-z映射为0-25的一个哈希函数。

大部分情况下，很难保证每一个“键”通过哈希函数的转换对应不同的“索引”，所以很多时候要处理**哈希冲突**（两个不同的“键”通过哈希函数后转换成了相同的“索引”）。

做哈希表的时候需要解决的两个关键问题：

- 如何设计哈希函数
- 如何解决哈希冲突

哈希表充分体现了算法设计的经典思想：空间换时间。

假设我们可以开辟很大空间的数组，足以容纳所有的情况，这种情况下可以用`O(1)`的事件完成各项操作；如果我们只有1的空间，所有的情况都需要存储到这一个空间中，我们只能用`O(n)`时间完成各项操作。**哈希表是时间和空间之间的平衡。**

14-2 哈希表的设计

“键”通过哈希函数得到的“索引”分布越均匀越好。

这个课程主要关注一般的哈希函数的设计原则

整型

- 小范围正整数直接使用
- 小范围负整数进行偏移：`-100 ~ 100 => 0 ~ 200`
- 大整数：例如身份证号
    - 通常做法：取模，比如取后四位，等同于`mod 10000`

      110108198512166666 ==> 6666

对于身份证号来说，如果取后六位，就会分布不均匀，因为后六位的前两位代表日期，所以范围只能是1~31，就会造成分布不均匀，而且也直接丢掉了前面的12位，没有利用所有信息

- 一个简单的解决办法：模一个素数，可以有效减少哈希冲突（分布更均匀）

浮点型

在计算机中都是32位或者54位的二进制表示，只不过计算机解析成了浮点数。可以把32或者64位空间当作一个大整数来处理

字符串

$$166 = 1 * 10^2 + 6 * 10^1 + 6 * 10^0$$

$$code = c * 26^3 + o * 26^2 + d * 26^1 + e * 26^0$$

这样就可以把字符串看做一个26进制的大的整型。

$$hash(code) = (c * B^3 + o * B^2 + d * B^1 + e * B^0)\%M = ((((c * B) + o) * B + d) * B + e)\%M$$

$$hash(code) = ((((c\%M) * B + o)\%M * B + d)\%M * B + e)\%M$$

这样可以防止整型溢出

```
int hash = 0;
for(int i = 0; i < s.length(); i++)
    hash = (hash * B + s.charAt(i)) % M;
```

复合类型

转成整型处理，只不过把字符串中的每个字符换成复合类型中的每个成员变量。

> 对于哈希函数的设计，这并不是唯一的方法

对于哈希函数的设计，应该遵循这三个原则：

1. **一致性**：如果a==b，则`hash(a) == hash(b)`，但反过来不一定成立。
2. **高效性**：计算高效简便
3. **均匀性**：哈希值均匀分布

14-3 Java中的hashCode

当使用HashSet或者HashMap的时候，会调用存储类型的`hashCode()`方法，默认的`hashCode()`方法把地址映射为整型。

当产生哈希冲突（对应的`hashCode()`值相等）的时候，会调用`equals()`方法来判断两个类是否真正相等。所以如果一个类要作为Hash表的键时，只覆盖一个`hashCode()`方法是不够的，还要覆盖`equals()`方法。

我们以“**类的用途**”来将“hashCode() 和 equals()的关系”分2种情况来说明。

1. 第一种 不会创建“类对应的散列表”

我们不会在HashSet, Hashtable, HashMap等等这些本质是散列表的数据结构中，用到该类。例如，不会创建该类的HashSet集合。

在这种情况下，该类的"hashCode() 和 equals() "**没有半毛钱关系的**！这种情况下，equals() 用来比较该类的两个对象是否相等。而hashCode() 则根本没有任何作用，所以，不用理会hashCode()。

2. 第二种 会创建"类对应的散列表"

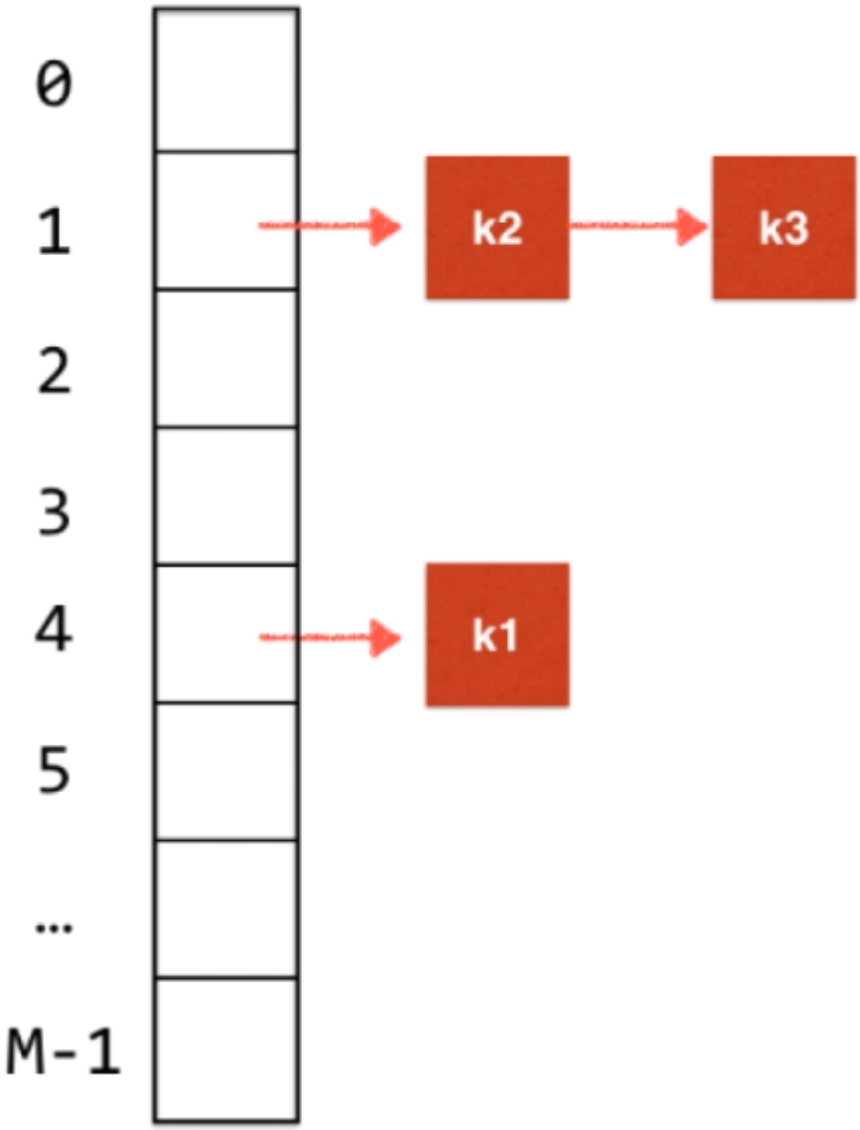
在这种情况下，该类的"hashCode() 和 equals() "是有关系的：

- 如果两个对象相等，那么它们的hashCode()值一定相同。这里的相等是指，通过equals()比较两个对象时返回true。
- 如果两个对象hashCode()相等，它们并不一定相等。因为在散列表中，hashCode()相等，即两个键值对的哈希值相等。然而哈希值相等，并不一定能得出键值对相等。补充说一句："两个不同的键值对，哈希值相等"，这就是哈希冲突。

14-4 哈希冲突的处理 链地址法

Seperate Chaining

java中的`hashCode()`有可能得到负数，为了去掉负数`(hashCode(x) & 0x7fffffff) % M`（整型和31个1进行按位与，整型的表示是32位，最高位是符号位，其实就是去掉符号位）



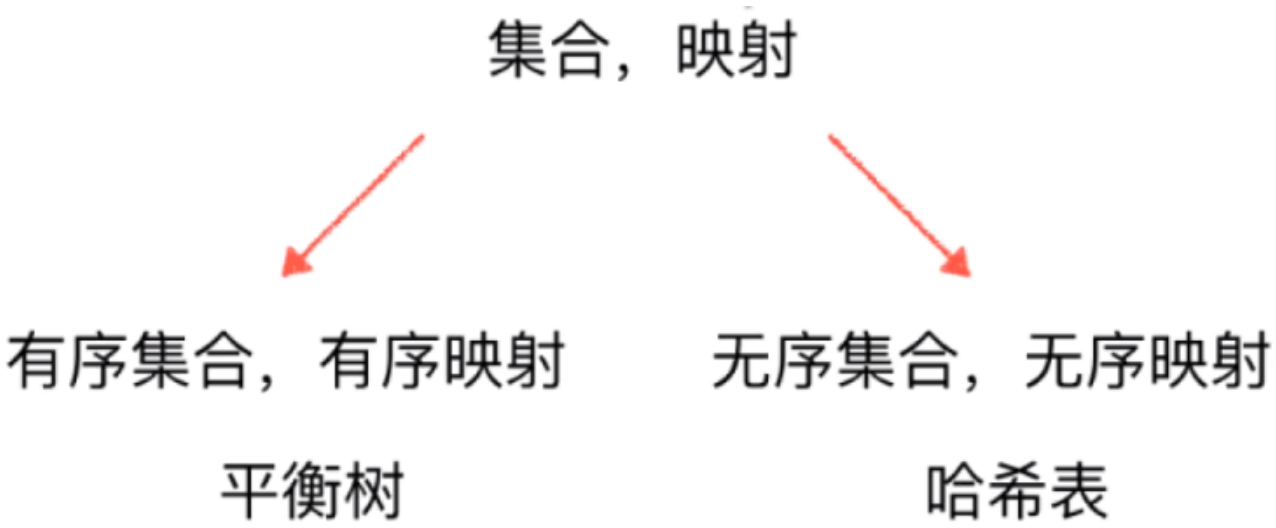
如图所示，k2和k3产生了哈希冲突。每一个chaining本质是一个查找表，不一定非得是链表，可以是树。

> 在java8之前，每个位置对应一个链表，java8开始，当哈希冲突达到一定程度，每一个位置从链表转成红黑树（当然要求key是可比较的，否则不会转成红黑树）。当数据规模很小的时候，链表是更快的。

14-6 哈希表的动态空间处理与复杂度分析

总共有M个地址，如果放入哈希表的元素为N，如果每个地址是链表：`O(N/M)`，如果每个地址是平衡树：`O(log(N/M))`，随着N增大，时间复杂度也会逐渐增大，所以哈希表应该是一个动态数组。当平均每个地址承载的元素多过一定程度，就扩容；如果平均每个地址承载的元素少过一定程度，就缩容。

对于哈希表来说，均摊复杂度为`O(1)`，***复杂度比树结构低，但是牺牲了顺序性。***



14-8 更多哈希冲突的处理方法

- 开放地址法：每一个地址对所有元素都是开放的，只要按照规则占就行。比如

$$hash(x) = x\%10$$

  - 线性探测法：11先进入，占据了1这个地址，31来了，哈希值和11冲突，1位置被占了，于是31就占1的后一个位置2，接着来81，占据位置3。这个方法性能比较低。
  - 平方探测法：遇到哈希冲突，先去+1的位置尝试，如果+1的位置被占，就去+4，如果还被占就去+9，然后+16等等。
  - 二次哈希法：遇到哈希冲突，就去`hash(hash(x))`的位置寻找。

  链地址法是封闭地址法
- 再哈希法：使用一个哈希函数产生了哈希冲突后，就使用另一个哈希函数产生索引。
- Coalesced Hashing：综合了Seperate Chaining和Open Address两种方法。

```
public class HashTable<K, V> {

    private final static int[] capacity
            = {53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593,
            49157, 98317, 196613, 393241, 786433, 1572869, 3145739, 6291469,
            12582917, 25165843, 50331653, 100663319, 201326611, 402653189, 805306457,
1610612741};

    private final static int UPPER_TOL = 10;
    private final static int LOWER_TOL = 2;
    private int capacityIndex = 0;

    private TreeMap<K, V>[] hashtable;
    private int M;
    private int size;

    public HashTable() {
        this.M = capacity[capacityIndex];

        size = 0;
```

```
        hashtable = new TreeMap[M];
        for (int i = 0; i < M; i++)
            hashtable[i] = new TreeMap<>();
    }

    private int hash(K key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }

    public int getSize() {
        return size;
    }

    public void add(K key, V value) {
        TreeMap<K, V> map = hashtable[hash(key)];
        if (map.containsKey(key))
            map.put(key, value);
        else {
            map.put(key, value);
            size++;

            if (size >= UPPER_TOL * M && capacityIndex + 1 < capacity.length) {
                capacityIndex++;
                resize(capacity[capacityIndex]);
            }
        }
    }

    public V remove(K key) {
        TreeMap<K, V> map = hashtable[hash(key)];
        V ret = null;
        if (map.containsKey(key)) {
            ret = map.remove(key);
            size--;

            if (size < LOWER_TOL * M && M / 2 >= capacity[0]) {
                capacityIndex--;
                resize(capacity[capacityIndex]);
            }
        }
        return ret;
    }

    public void set(K key, V value) {
        TreeMap<K, V> map = hashtable[hash(key)];
        if (!map.containsKey(key))
            throw new IllegalArgumentException("key不存在");
        map.put(key, value);
    }

    public boolean contains(K key) {
        return hashtable[hash(key)].containsKey(key);

    }
```

```
    public V get(K key) {
        return hashtable[hash(key)].get(key);
    }

    private void resize(int newM) {
        TreeMap<K, V>[] newHashTable = new TreeMap[newM];
        for (int i = 0; i < newM; i++)
            newHashTable[i] = new TreeMap<>();

        this.M = newM;

        for (TreeMap<K, V> map : hashtable) {
            for (K key : map.keySet()) {
                newHashTable[hash(key)].put(key, map.get(key));
            }
        }
        this.hashtable = newHashTable;
    }
}
```

数据结构的总结

共有三种基本类型的数据结构：

- 数组和列表
- 树
- 哈希表

这三种基本的数据结构分别对应三种类型：

- 数组和列表：保存插入顺序
- 树：排序
- 哈希表：无序