# 数据结构大纲

<https://blog.csdn.net/weixin_50651363/article/details/119784676>

# 复杂度分析

**时间复杂度：**估算该算法的运行时间（代码执行时间随数据规模增大而增大）；

**空间复杂度：**估算算法运行时需要开辟的额外空间（函数中变量的个数，注意容器的大小）

PS:量级都是估算的，并不是一个准确的数值；

**大O表示法：**

时间复杂度公式：T(n) = O（f(n)）；

n： 数据规模（循环的次数）；

f(n)： 每行代码执行次数的总和；

T(n)： 代表代码执行时间；

O： 代表代码的执行时间T(n)与f(n)表达式成正比；

大O时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，所以，也叫作渐进时间复杂度(asymptotic time complexity)，简称时间复杂度。

**复杂度分析规则：**

单段代码看高频：比如循环；

多段代码取最大：一段代码中有单循环和多重循环，那么取多重循环复杂度；

嵌套代码求乘积：比如递归、多重循环；

多个规模求加法：比如两个参数控制两个循环次数，那么复杂度两者相加；

**时间复杂度分析：**

**只关注循环执行次数最多的一段代码；**

**加法法则：总复杂度 = 量级最大的那段代码的复杂度（代码并行，没有嵌套）；**

**乘法法则：嵌套代码的复杂度 = 嵌套内外代码复杂度的乘积；**

**常见复杂度量级（按数量级递增）：**

**常量阶：O(1) ------代码的执行时间不随n的增大而增加；**

**对数阶：O(logn) ------（二分查找法）**

**线性阶：O(n) ------代码的执行时间随n的增大线性增加；**

**线性对数阶：O(nlogn) ------ 将O(logn)的代码，在循环n遍；**

**次方阶：O(nn) ------把O(n)的代码在嵌套循环n遍；（**嵌套代码求乘积**）**

**指数阶：O(2n) ------**

**阶层阶：O(n!) ------**

**排序算法：**

# 散列表

<https://blog.csdn.net/weixin_47251999/article/details/113037111>

<https://www.jianshu.com/p/0e6f2a195168>

散列函数：将key值转换成数组下标的算法；

散列冲突解决：

1.开放寻址法；

2.链表法 --- 将链表替换为红黑树；

动态扩容：装载因子过大时，需要动态扩容；

**装载因子 = 填入数据的元素个数 / 数组长度；**

**装载因子过大，数组空闲位置越少，冲突频率越高，散列表性能降低；**

**一般规则：装载因子 > 0.75 ，就可以调整列表的长度了；**

动态扩容原则：将一次性扩容的代价，均摊到多个插入操作上；

**位图**

# 树

<https://blog.csdn.net/qq_43563538/article/details/116017187>

<https://blog.csdn.net/Real_Fool_/article/details/113930623>

## 树相关术语

1. 节点的度：一个节点含有的子树（就是它的子节点）的个数称为该节点的度；
2. 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；

从1开始，也有定义是从0开始的；

1. 节点的高度：以整个树为基准（，从上往下，依次递减）；
2. 节点的深度：以整个树为基准；

1. 树的度：一棵树中，最大的节点的度称为树的度；
2. 树的高度或深度：树中节点的最大层次；
3. 空树：度为0的树；
4. 叶节点：度为0的节点称为叶节点；
5. 分支节点：度不为0的节点；
6. 父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；
7. 子节点：一个节点含有的子树的根节点称为该节点的子节点；
8. 兄弟节点：具有相同父节点的节点互称为兄弟节点；
9. 堂兄弟节点：双亲在同一层的节点互为堂兄弟；
10. 子孙节点：以某节点为根的子树中任一节点都称为该节点的子孙。
11. 节点的祖先：根节点；
12. 森林：由m（m>=0）棵互不相交的树的集合称为森林；

## 树的存储结构

双亲表示法（不常用）

孩子表示法（不常用）

孩子兄弟表示法（最常用，将一颗复杂的树构建成一颗“二叉树”）



typedef struct CSNode{

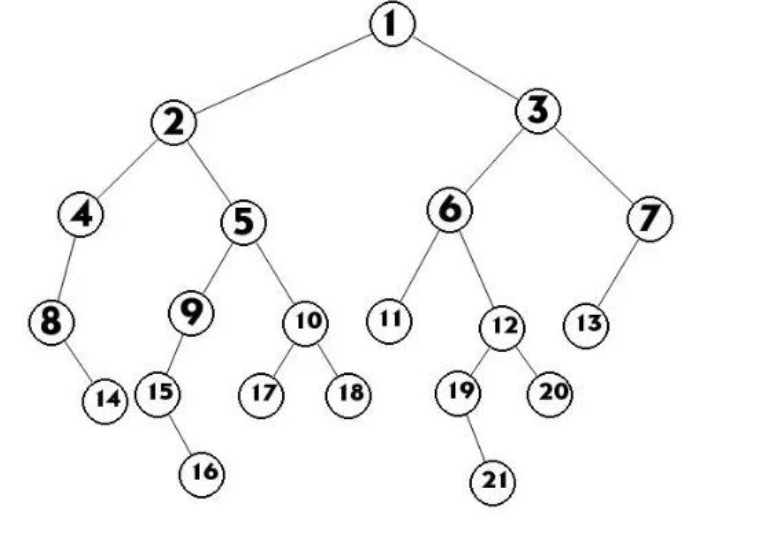
TElemtype data;

struct CSNode \*firstchild, \*rightsib;

} CSNode, \*CSTree;

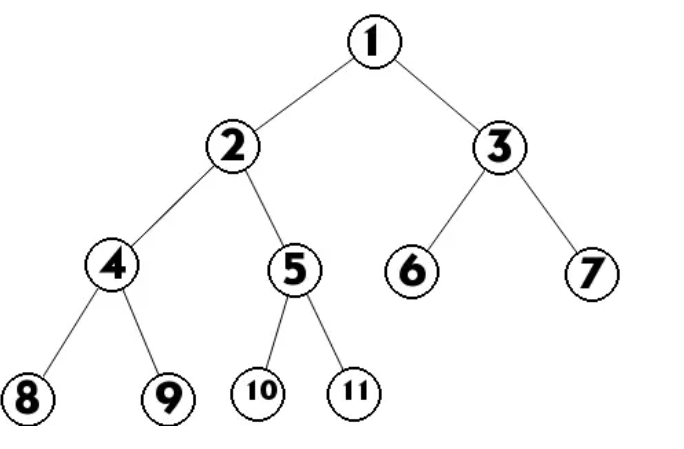
## 普通二叉树

普通二叉树：某一个节点缺少“左节点”或缺少“右节点”；



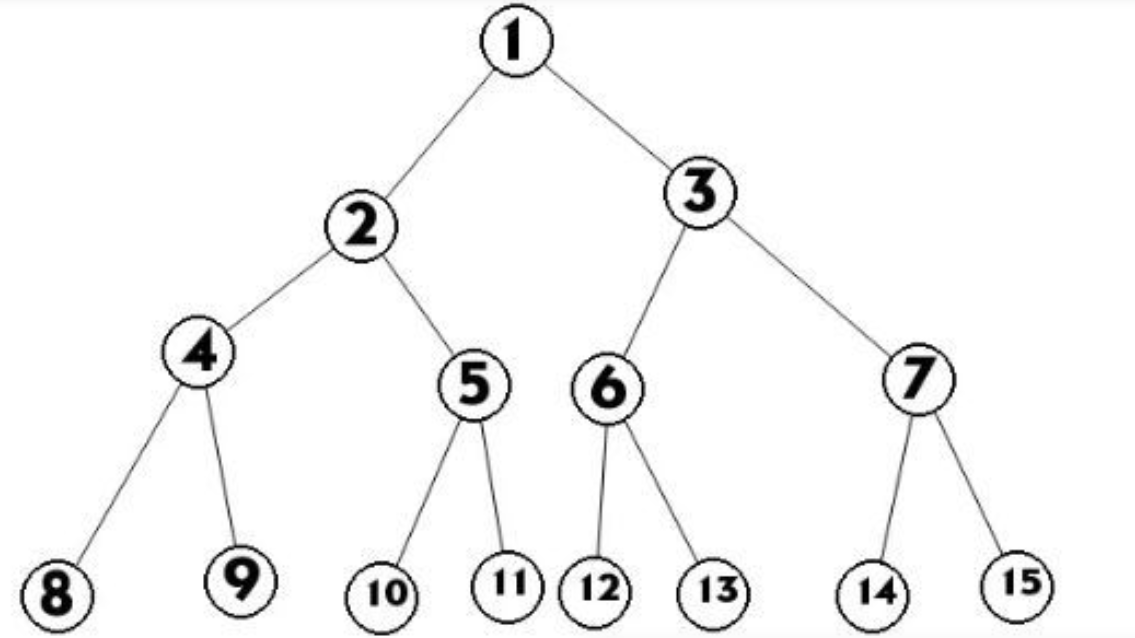
## 特殊二叉树（完全二叉树、满二叉树）

完全二叉树：每一个节点都有左右两个子节点（叶子节点除外）；



满二叉树：每一个节点都有左右两个子节点（叶子节点除外）并且所有叶子节点的深度相同；

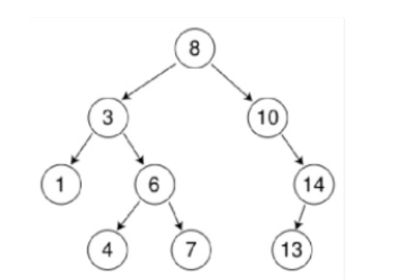
满二叉树属于完全二叉树的一种；



## 二叉树遍历

三种遍历方式：前序遍历（中左右）、中序遍历（左中右）、后序遍历（左右中）；

PS：三种遍历的方式区别：子节点的父节点何时被访问（即上述的“中”何时被访问，以“节点”为单位）；



前序：8 3 1 6 4 7 10 14 13

中序：1 3 4 6 7 8 10 13 14

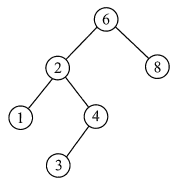
后序：1 4 7 6 3 13 14 10 8

## 二叉查找树

二叉查找树（也称二叉排序树、二叉搜索树）：是一个空树或者具有三个特性的二叉树：

* 若左子树非空，则左子树上所有结点的值均小于根结点的值。
* 若右子树非空，则右子树上所有结点的值均大于根结点的值。
* 左、右子树也分别是一棵二叉排序树。

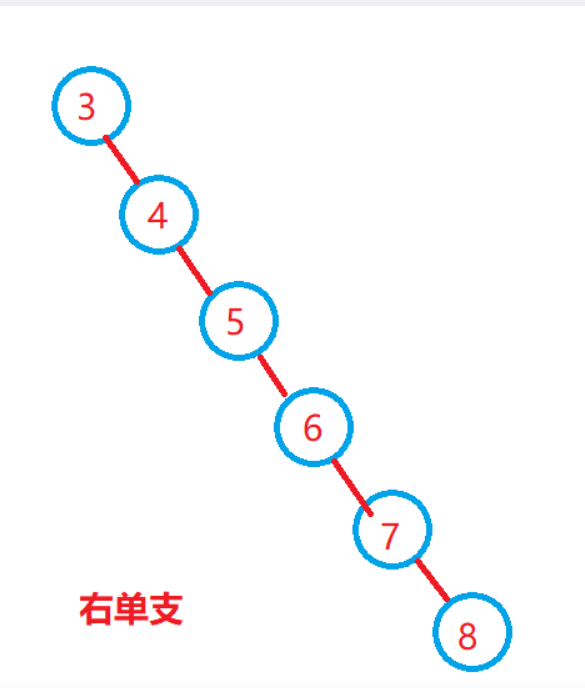
通过中序遍历，得到是一个有序数的集合；



中序遍历：1 2 3 4 6 8

时间复杂度为:O(logn)；

如果二叉搜索树退化成链时，时间复杂度为O(n)；



## 平衡二叉树（AVL）

<https://blog.csdn.net/jarvan5/article/details/112428036>

### 平衡二叉树历史

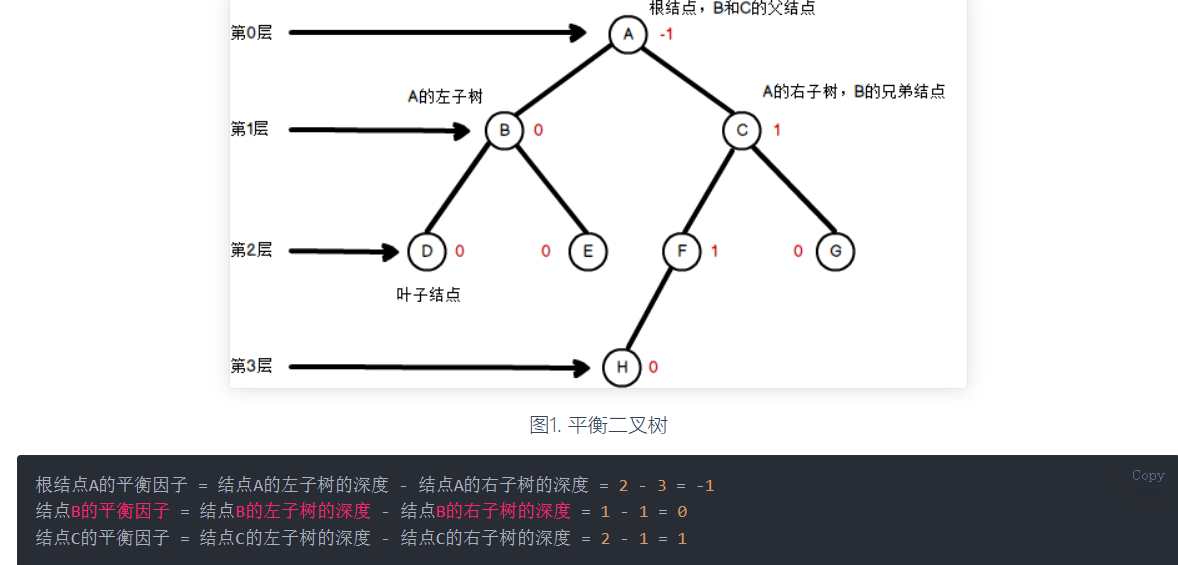
为了提高二叉搜索树的查找效率而提出的；（二叉搜索树可能会退化成链，此时查找时间复杂度为O(n)）

平衡二叉树是一种二叉排序树，其中每一个节点左节点和右节点的高度差的绝对值<=1；

### 平衡因子（BF）

将二叉树节点的“左子树的深度（高度）” 减去 右子树的深度（高度） 的值；（将左右子树当作一个整体，深度从1开始）；

Ps:平衡因子的绝对值必须<=1;



### 最小不平衡子树

距离插入节点最近的，并且BF的绝对值大于1的节点为根节点的子树；

### 旋转解决最小不平衡子树

只需要纠正最小平衡子树即可；

旋转方式：

* 左旋：

旧根节点为新根节点的左子树

新根节点的左子树（如果存在）为旧根节点的右子树；

* 右旋：

旧根节点为新根节点的右子树

新根节点的右子树（如果存在）为旧根节点的左子树

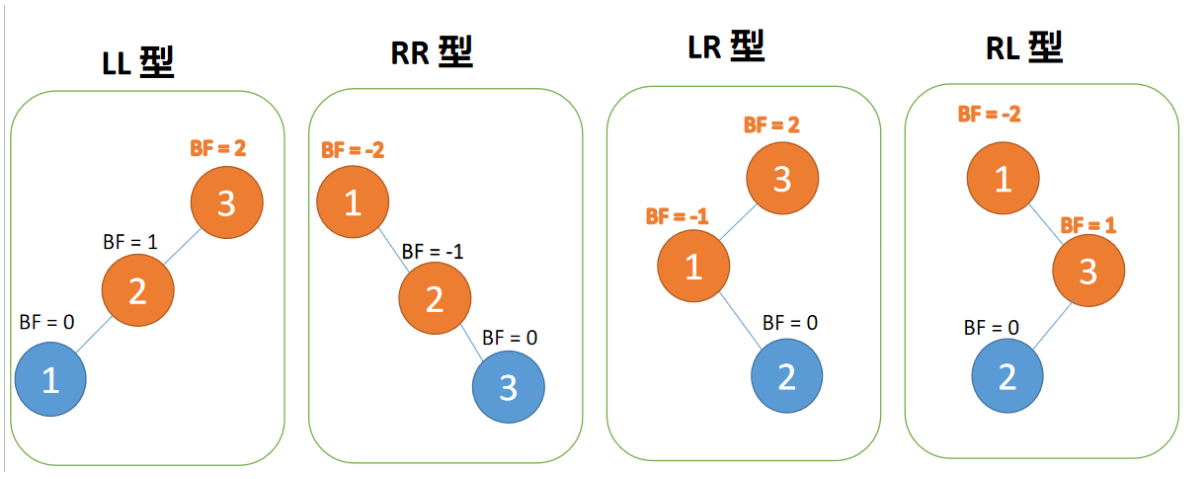
* 4 种「旋转」纠正类型：

LL 型：插入左孩子的左子树，右旋

RR 型：插入右孩子的右子树，左旋

LR 型：插入左孩子的右子树，先左旋，再右旋

RL 型：插入右孩子的左子树，先右旋，再左旋



### 代码实现

* 定义节点;
* 计算节点高度;
* 计算平衡因子;
* 旋转及刷新高度;

**AVL树节点定义：**

class AVLNode {

public:

int data; /\*\* 数据 \*\*/

int height; /\*\* 相对高度 \*\*/

AVLNode parent; /\*\* 父节点 \*\*/

AVLNode left; /\*\* 左子树 \*\*/

AVLNode right; /\*\* 右子树 \*\*/

AVLNode(int data) {

this.data = data;

this.height = 1;

}

}

**计算节点高度：**

节点高度 = 左子树高度和右子树的最大高度+1;

/\*\* 通过子树高度 计算高度 \*\*/

int calcHeight(AVLNode root) {

if (root.left == null && root.right == null) {

return 1;

}

else if (root.right == null) {

return root.left.height + 1;

} else if (root.left == null) {

return root.right.height + 1;

}else {

return root.left.height > root.right.height ? (root.left.height + 1) : (root.right.height + 1);

}

}

**计算平衡因子（BF）:**

BF = 左子树高度 – 右子树高度；

int calcBF(AVLNode root) {

if (root == null){

return 0;

}else if (root.left == null && root.right == null) {

return 0;

}else if (root.right == null) {

return root.left.height ;

} else if (root.left == null) {

return - root.right.height;

}else {

return root.left.height - root.right.height;

}

}

**旋转及刷新高度：**

思想：左旋、右旋；

重点理解：

旋转之后通过需要刷新高度（高度变化只有： oldRoot 和 newRoot）；

但是它们子树的高度是不变的（这很关键）；我们可以通过它们 子树的高度计算他们的高度；（使用不变的因数计算变化的因素是一个很好的思维；）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*左旋\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

AVLNode leftRotate(AVLNode root) {

AVLNode oldRoot = root;

AVLNode newRoot = root.right;

AVLNode parent = root.parent;

//1.newRoot 替换 oldRoot 位置（newRoot的节点是在父节点的左还是右）

if (null != parent ) {

if (oldRoot.parent.data > oldRoot.data) {

parent.left = newRoot;

}else {

parent.right = newRoot;

}

}

newRoot.parent = parent;

//2.重新组装 oldRoot (将 newRoot 的左子树 给 oldRoot 的右子树)

oldRoot.right = newRoot.left;

if (newRoot.left != null) {

newRoot.left.parent = oldRoot;

}

//3. oldRoot 为 newRoot 的左子树

newRoot.left = oldRoot;

oldRoot.parent = newRoot;

//刷新高度

oldRoot.height = calcHeight(oldRoot);

newRoot.height = calcHeight(newRoot);

return newRoot;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*右旋\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

AVLNode rightRotate(AVLNode root) {

AVLNode oldRoot = root;

AVLNode newRoot = root.left;

AVLNode parent = root.parent;

//1.newRoot 替换 oldRoot 位置

if (null != parent ) {

if (oldRoot.parent.data > oldRoot.data) {

parent.left = newRoot;

}else{

parent.right = newRoot;

}

}

newRoot.parent = parent;

//2.重新组装 oldRoot (将 newRoot 的右子树 给 oldRoot 的左子树)

oldRoot.left = newRoot.right;

if (newRoot.right != null) {

newRoot.right.parent = oldRoot;

}

//3. oldRoot 为 newRoot 的左子树

newRoot.right = oldRoot;

oldRoot.parent = newRoot;

//刷新高度

oldRoot.height = calcHeight(oldRoot);

newRoot.height = calcHeight(newRoot);

return newRoot;

}

**插入（总代码）：**

class ALVTree {

AVLNode root;

public void insert(int data) {

if (null == this.root) {

this.root = new AVLNode(data);

return;

}

this.root = insert(this.root, data);

}

AVLNode insert(AVLNode root, int data) {

//插入左子树

if (data < root.data) {

if (null == root.left) {

root.left = new AVLNode(data);

root.left.parent = root;

}else {

insert(root.left,data);

}

}

//插入右子树

else if (data > root.data) {

if (null == root.right) {

root.right = new AVLNode(data);

root.right.parent = root;

} else {

insert(root.right,data);

}

}

//刷新高度

root.height = calcHeight(root);

//旋转

//1. LL 型 右旋转

if (calcBF(root) == 2){

//2. LR 型 先左旋转

if (calcBF(root.left) == -1) {

root.left = leftRotate(root.left);

}

root = rightRotate(root);

}

//3. RR型 左旋转

if (calcBF(root) == -2){

//4. RL 型 先右旋转

if (calcBF(root.right)== 1) {

root.right = rightRotate(root.right);

}

root = leftRotate(root);

}

return root;

}

AVLNode leftRotate(AVLNode root) {

AVLNode oldRoot = root;

AVLNode newRoot = root.right;

AVLNode parent = root.parent;

//1.newRoot 替换 oldRoot 位置

if (null != parent ) {

if (oldRoot.parent.data > oldRoot.data) {

parent.left = newRoot;

}else {

parent.right = newRoot;

}

}

newRoot.parent = parent;

//2.重新组装 oldRoot (将 newRoot 的左子树 给 oldRoot 的右子树)

oldRoot.right = newRoot.left;

if (newRoot.left != null) {

newRoot.left.parent = oldRoot;

}

//3. oldRoot 为 newRoot 的左子树

newRoot.left = oldRoot;

oldRoot.parent = newRoot;

//刷新高度

oldRoot.height = calcHeight(oldRoot);

newRoot.height = calcHeight(newRoot);

return newRoot;

}

AVLNode rightRotate(AVLNode root) {

AVLNode oldRoot = root;

AVLNode newRoot = root.left;

AVLNode parent = root.parent;

//1.newRoot 替换 oldRoot 位置

if (null != parent ) {

if (oldRoot.parent.data > oldRoot.data) {

parent.left = newRoot;

}else {

parent.right = newRoot;

}

}

newRoot.parent = parent;

//2.重新组装 oldRoot (将 newRoot 的右子树 给 oldRoot 的左子树)

oldRoot.left = newRoot.right;

if (newRoot.right != null) {

newRoot.right.parent = oldRoot;

}

//3. oldRoot 为 newRoot 的左子树

newRoot.right = oldRoot;

oldRoot.parent = newRoot;

//刷新高度

oldRoot.height = calcHeight(oldRoot);

newRoot.height = calcHeight(newRoot);

return newRoot;

}

/\*\* 通过子树高度 计算高度 \*\*/

int calcHeight(AVLNode root) {

if (root.left == null && root.right == null) {

return 1;

}

else if (root.right == null) {

return root.left.height + 1;

} else if (root.left == null) {

return root.right.height + 1;

}else {

return root.left.height > root.right.height ? root.left.height + 1 : root.right.height + 1;

}

}

int calcBF(AVLNode root) {

if (root == null){

return 0;

}

else if (root.left == null && root.right == null) {

return 0;

}

else if (root.right == null) {

return root.left.height ;

} else if (root.left == null) {

return - root.right.height;

}else {

return root.left.height - root.right.height;

}

}

}

## 红黑树（R-B Tree）

<https://blog.csdn.net/weixin_45799086/article/details/126745366>

<https://blog.csdn.net/m0_37707561/article/details/122967286>

**红黑树历史：**

平衡二叉树优点搜索效率高，但是插入和删除效率不高，插入和删除会导致最小失衡子树的出现，需要不断消除最小失衡树；

如果插入删除频繁的话，平衡二叉树的效率就体现不出来了；

在此基础上，又发展出了红黑树的这种数据结构，红黑树相对于平衡二叉树来说，插入和删除效率较高，插入最多只需要2次旋转，删除最多需要3次旋转；

**红黑树简介：**

红黑树是对树结构的一种高度综合运用，涉及到多叉树，树平衡调整，节点旋转等等，这些是对数据结构基本功的最佳历练。

**红黑树定义：**

红黑树也是一种自平衡二叉树，它与AVL树类似，都在添加和删除时通过自旋转操作保持二叉树平衡，以求更高效的查询性能；

与AVL树相比，红黑树牺牲了部分平衡性，以换取插入/删除操作时较少的旋转操作，整体来说性能要优于AVL树。它可以在O(log n)时间内做查找,插入和删除,这里的n 是树中元素的数目；

**应用场景：**

红黑树是一种性能非常稳定的二叉查找树，所以，在工程中，但凡是用到动态插入、删除、查找数据的场景，都可以用到它。

例如：

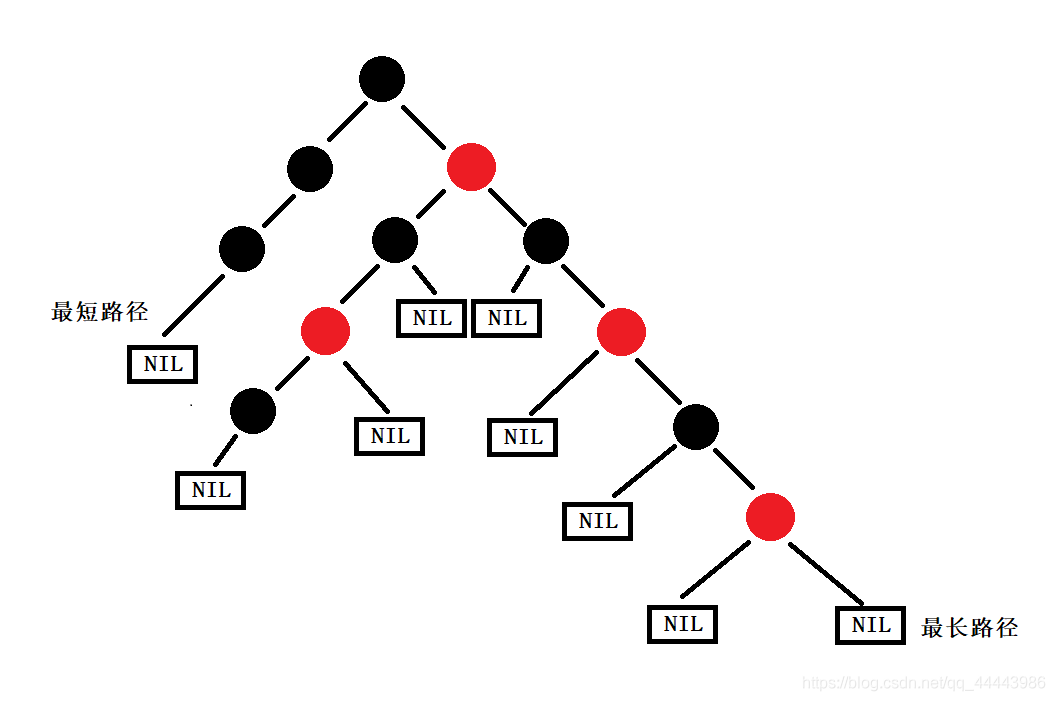
* Java的HashMap的底层实现，在JDK1.8中为了解决过度哈希冲突带来的长链表，将链表转为红黑树；
* Linux底层的CFS进程调度算法中，vruntime利用红黑树来进行存储；
* 多路复用技术的epoll的核心结构也是红黑树 + 双向链表；

**红黑树五大规则：**

* 1. 节点非黑即红；
  2. null或nil节点为黑；
  3. 根节点为黑；
  4. 红节点的子节点都为黑；(**不能存在连续的红色节点**)
  5. 从任一节点到该节点的每个叶子节点的所有路径，都包含相同数目的黑色节点；

由五大规则构建的红黑树遵循一个真理：通过任意一条**从根结点到叶子结点的路径上颜色的限制，从而确保红黑树的最长路径不会超过最短路径的两倍**，因此就可以达到近似平衡的效果；

如下图：



最短路径的路径上的结点全为黑色结点，最长路径则是黑红相间，而要保证黑色结点数目相同且不能出现连续的两个红结点。从而确定了最长路径不会超过最短路径的两倍；

**红黑树构建：**

以二叉树为基础，在二叉树的属性中加入一个颜色属性（RED或BLACK）来表示2-3-4-….层树中不同的节点；

