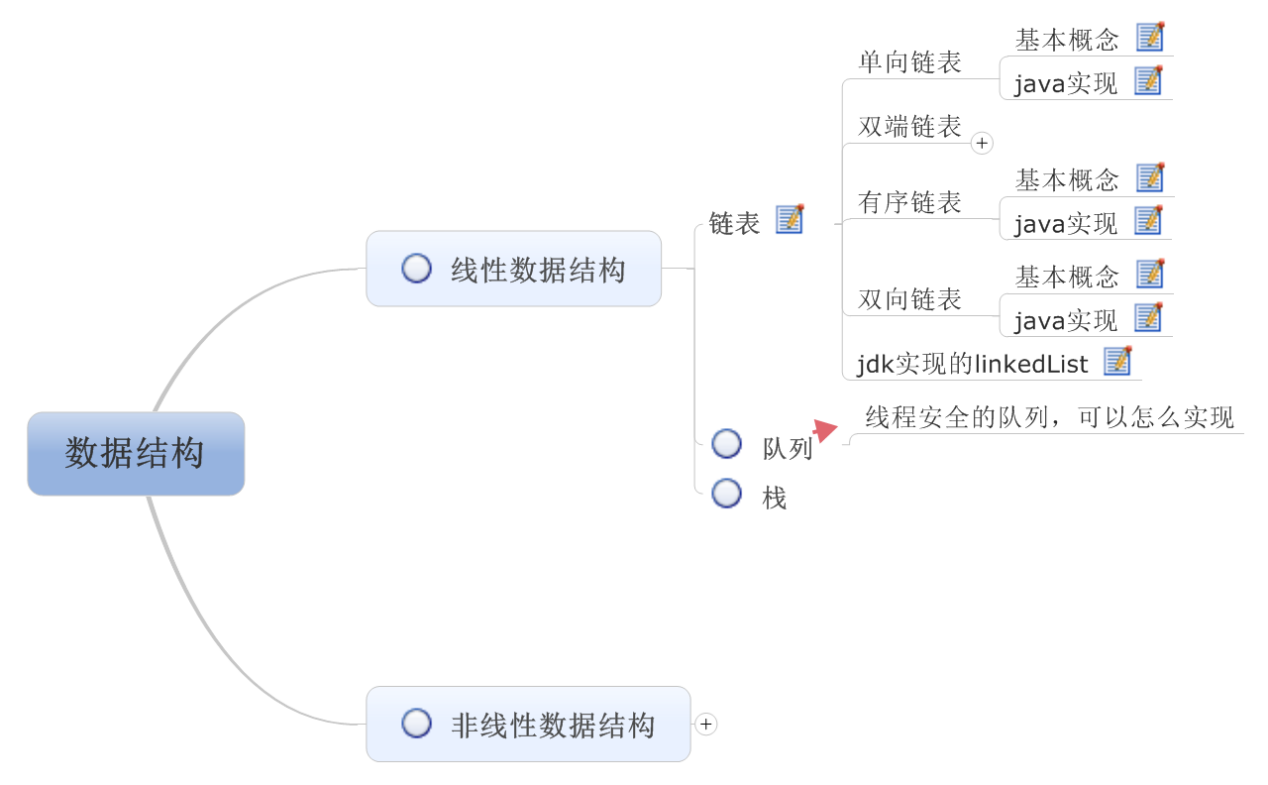
数据结构



数据结构 1

1  线性数据结构 8

1.1 链表 8

1.1.1 单向链表 8

1.1.1.1 基本概念 8

1.1.1.2 java实现 9

1.1.2 双端链表 9

1.1.2.1 基本概念 9

1.1.2.2 java实现 11

1.1.3 有序链表 11

1.1.3.1 基本概念 11

1.1.3.2 java实现 12

1.1.4 双向链表 13

1.1.4.1 基本概念 13

1.1.4.2 java实现 15

1.1.5 jdk实现的linkedList 16

1.2  队列 19

1.2.1 线程安全的队列，可以怎么实现 19

1.3  栈 19

2  非线性数据结构 19

2.1  树 19

2.1.1  树 19

2.1.1.1  定义 19

2.1.1.2  概念 20

2.1.1.2.1  结点的度（Degree）：结点拥有的子树数。注意，结点的子树，是指其直接后代。 20

2.1.1.2.2  结点的层次(Level)：从根开始定义起，根为第一层，根的孩子为第二层 21

2.1.1.2.3  堂兄弟：双亲在同一层的结点互为堂兄弟。 22

2.1.1.2.4  树的深度（Depth）或高度：树中结点的最大层次。 23

2.1.1.2.5  有序树：如果将树中各结点的子树看成是从左至右有次序的，不能互换的，那么这个树 就是有序树。否则，是无序树。 24

2.1.1.2.6  森林（Forest）：m(m>=0)棵互不相交的树的集合。对树中结点而言，其子树的集合，就是森林 24

2.1.1.2.7  叶结点（Leaf）或终端结点：度为0的结点。 24

2.1.1.2.8  非终端结点(分支结点)：度不为0的结点。 24

2.1.1.2.9  内部结点：除根结点之外的分支结点 24

2.1.1.2.10  树的度：树内各结点的度的最大值。 24

2.1.1.3  树的存储结构 24

2.1.1.3.1  双亲表示法： 以一组连续空间存储树的结点，每个结点除存储数据之外，还会存储一个指向该结点双亲 的指示器(双亲节点在连续空间中的位置)。 24

2.1.1.3.2  多重链表表示法：每个结点有多个指针域，其中每个指针指向一棵子树的根结点。 26

2.1.1.3.3  孩子表示法:由一个线性表和一组单链表组成 27

2.1.1.3.4  孩子表示法的改进--双亲孩子表示法 28

2.1.1.3.5  孩子兄弟表示法:一个结点由三部分组成：数据域（data），第一个孩子（firstChild）和第一个 右兄弟（rightsib）。 29

2.1.1.4  应用实例 30

2.1.2  二叉树 31

2.1.2.1  二叉树的概念 31

2.1.2.2  特殊二叉树 32

2.1.2.2.1  左斜树 32

2.1.2.2.2  右斜树 32

2.1.2.2.3  满二叉树 32

2.1.2.2.4  完全二叉树 33

2.1.2.3  性质 34

2.1.2.4  二叉树的存储结构 35

2.1.2.4.1  顺序存储结构 35

2.1.2.4.2  二叉链表 36

2.1.2.4.3  三叉链表 37

2.1.2.5  遍历二叉树 38

2.1.2.5.1  前序遍历 38

2.1.2.5.2  中序遍历 40

2.1.2.5.3  后序遍历 40

2.1.2.5.4  层序遍历 41

2.1.3  二叉查找树 42

2.1.3.1  定义 42

2.1.3.2  性质 42

2.1.3.3  时间复杂度 43

2.1.3.4  插入过程 43

2.1.3.4.1  若当前的二叉查找树为空，则插入的元素为根节点; 46

2.1.3.4.2  若插入的元素值小于根节点值，则将元素插入到左子树中 46

2.1.3.4.3  若插入的元素值不小于根节点值，则将元素插入到右子树中 46

2.1.3.5  删除操作 46

2.1.3.5.1  p为叶子节点，直接删除该节点， 再修改其父节点的指针（注意分被删除的节点是根节点和不是根节点），如图 50

2.1.3.5.2  p为单支节点（即只有左子树或右子树）。 让p的子树与p的父亲节点相连，删除p即可（注意分是根节点和不是根节点） 51

2.1.3.5.3  p的左子树和右子树均不空 52

 方法一： 找到p的后继y，因为y一定没有左子树，所以可以删除y， 并让y的父亲节点成为y的右子树的父亲节点，并用y的值代替p的值； 52

 方法二： 找到p的前驱x，x一定没有右子树，所以可以删除x， 并让x的父亲节点成为x的左子树的父亲节点。 52

2.1.3.6  查找操作 52

2.1.4  红黑树 53

2.1.4.1  红黑树的特性 54

2.1.4.2 红黑树的应用 54

2.1.4.3  复杂度 55

2.1.4.4 红黑树的基本操作 57

2.1.4.4.1  左旋和右旋 57

 左旋 58

 右旋 59

 区分左旋和右旋 60

2.1.4.4.2  添加节点 62

 将红黑树当作一颗二叉查找树，将节点插入 62

 将插入的节点着色为"红色" 63

 通过一系列的旋转或着色等操作， 使之重新成为一颗红黑树 64

 被插入的节点是根节点 直接把此节点涂为黑色 66

 被插入的节点的父节点是黑色 什么也不需要做。节点被插入后，仍然是红黑树 66

 被插入的节点的父节点是红色 66

 当前节点的祖父节点的另一个子节点（叔叔节点）也是红色 66

 叔叔节点是黑色的 68

 当前节点（被插入的节点）是其父节点的右孩子 68

 当前节点是其父节点的左孩子 69

2.1.4.4.3  删除节点 70

将红黑树当作一颗二叉查找树，将节点删除 70

通过"旋转和重新着色"等一系列来修正该树 使之重新成为一棵红黑树 72

伪代码实现 73

x是“红+黑”节点 直接把x设为黑色，结束。此时红黑树性质全部恢复 75

x是“黑+黑”节点，且x是根 什么都不做，结束。此时红黑树性质全部恢复 75

x是“黑+黑”节点，且x不是根 75

x的兄弟节点是红色 此时x的父节点和x的兄弟节点的子节点都是黑节点 75

x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色 77

2.1.5 B树 77

2.1.6 B+树 77

2.1.7 B\*树 77

2.1.8 Trie-Tree 77

2.1.8.1 原理 77

2.1.8.2 特点 77

2.1.8.3 应用 77

2.2  图 77

1.  线性数据结构
   1. 链表

链表是一种递归的数据结构，它要么为空（null），要么指向一个结点（node）的引用，该节点含有一个**泛型元素**（该泛型元素可以是任意数据类型），和一个指向另一个节点的引用。链表有很多种，下面主要介绍**单向链表**、**双端链表**、**有序链表**、**双向链表**

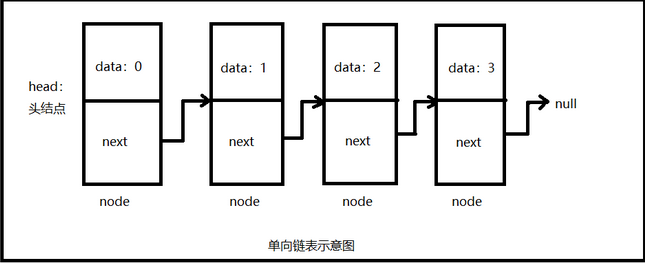
* + 1. 单向链表
       1. 基本概念

单向链表是最简单、最基础的链表，它的一个结点（node）分两部分，第一部分存储结点的数据信息（data），第二部分存储指向下一结点的地址（next）信息。最后一个结点（链尾）指向一个空地址（null）。

单向链表一般只在链表表头（链头）结点的位置插入元素，这样每次新加入的元素都会在链头位置，而最先加入的元素会在链尾位置。

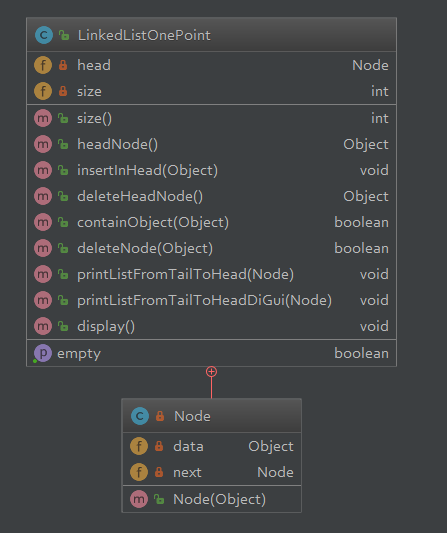
删除操作时，如果在链头位置删除，只需要把头结点指向其下一个结点即可；如果是在中间位置删除，只需要将其前一个结点指向其下一个结点即可。

单向链表示意图如下图所示:



可以用来实现栈；

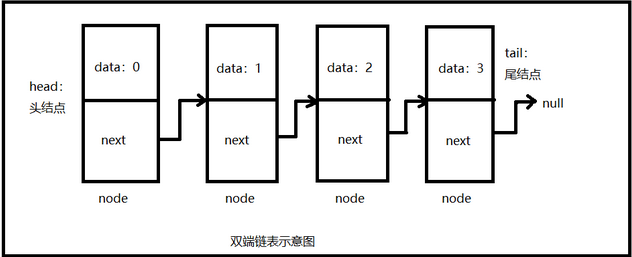
* + - 1. java实现



* + 1. 双端链表
       1. 基本概念

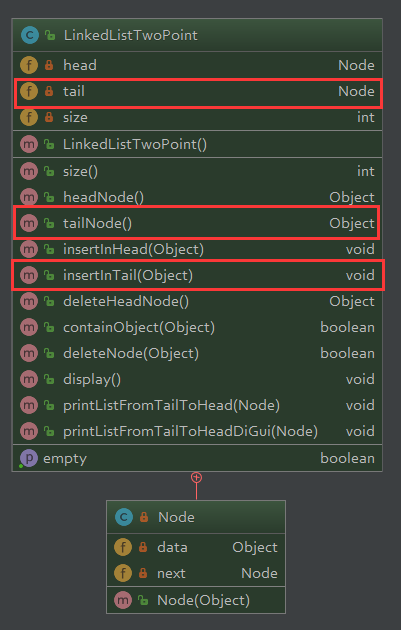
See also: [队列](#队列)

双端链表和单向链表大体上是一样的，不同的是，单向链表在表尾部分插入元素时，需要从头结点一直遍历到尾结点才能进行插入操作，这样难免有些繁琐。因此如果加入一个对尾结点的引用，这样就可以很方便地在尾结点进行插入操作，这就是双端链表。**除了有一个头结点（head），还有一个尾结点（tail）**。**注意它和后面双向链表的区别！**



可以用来实现队列；

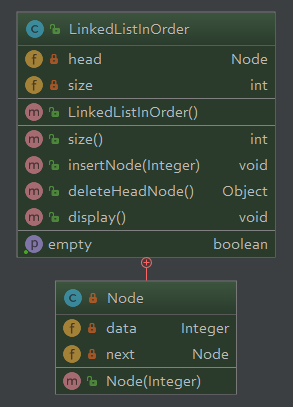
* + - 1. java实现

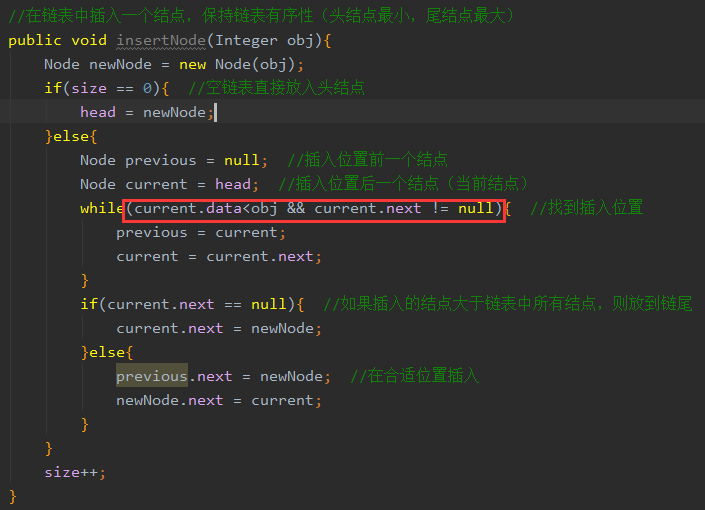


* + 1. 有序链表
       1. 基本概念

链表本身是一种无序的数据结构，元素的插入和删除不能保证顺序性，但是有没有有序的链表呢？答案是肯定的，我们在单向链表中插入元素时，只需要将插入的元素与头结点及其后面的结点比较，从而找到合适的位置插入即可。一般在大多数需要使用有序数组的场合也可以使用有序链表，**有序链表在插入时因为不需要移动元素，因此插入速度比数组快很多，另外链表可以扩展到全部有效的使用内存，而数组只能局限于一个固定的大小中**。

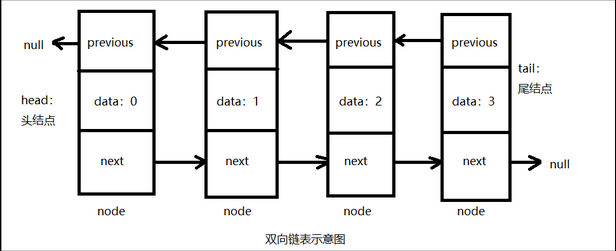
* + - 1. java实现



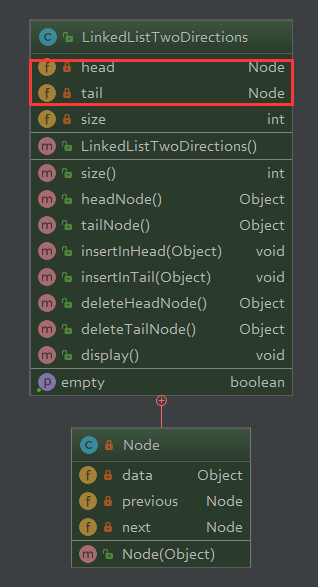


* + 1. 双向链表
       1. 基本概念

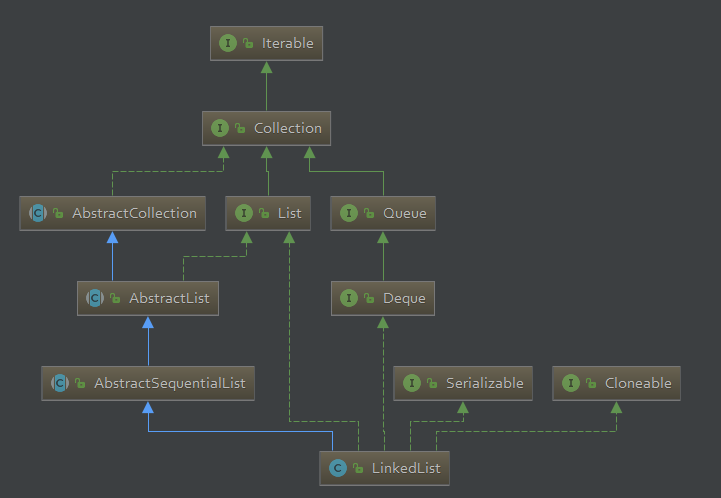
双向链表的每个结点既能指向下一个结点，又能指向前一个结点，双向链表既能从头结点向尾结点遍历，又能从尾结点向头结点遍历，既有一个头结点，又有一个尾结点。

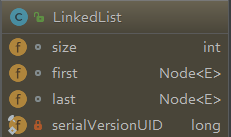
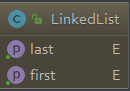


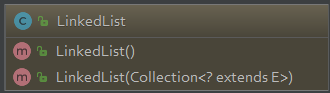
* + - 1. java实现

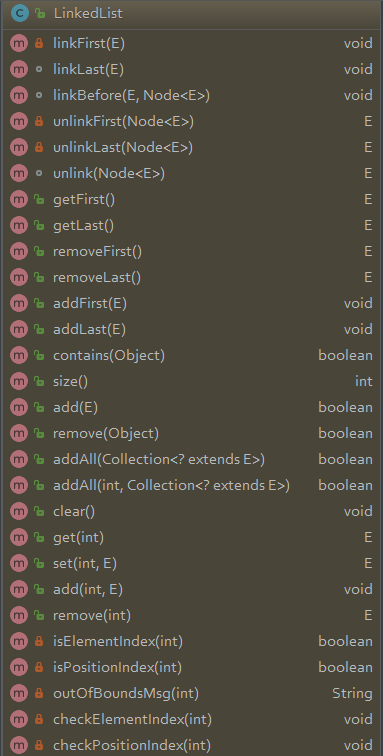


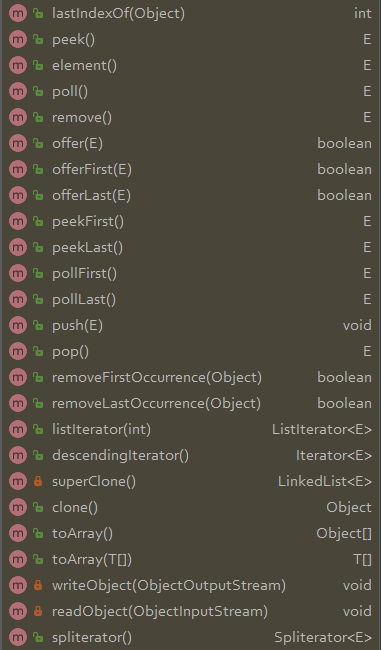
* + 1. jdk实现的linkedList









* 1.  队列
     1. 线程安全的队列，可以怎么实现
  2.  栈

1.  非线性数据结构
   1.  树
      1.  树
         1.  定义

1、树是一种数据结构，可以用来表示层次关系，因表示的样子很像一颗倒立的树而得名。

在数据结构中的特点，是**一对多**（链表是一对一，图是多对多）;

2、树(Tree)是n(n>=0)个结点的有限集：

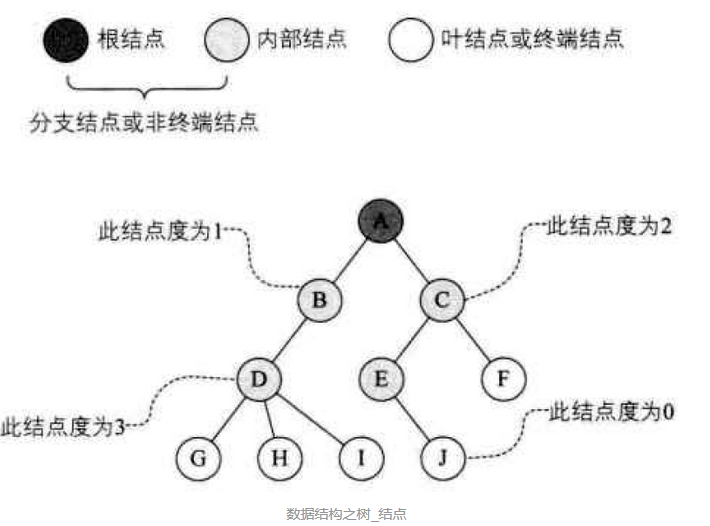
（1）、n = 0的树是空树。

（2）、在任意一棵非空树中有且仅有一个称为根(Root)的结点。

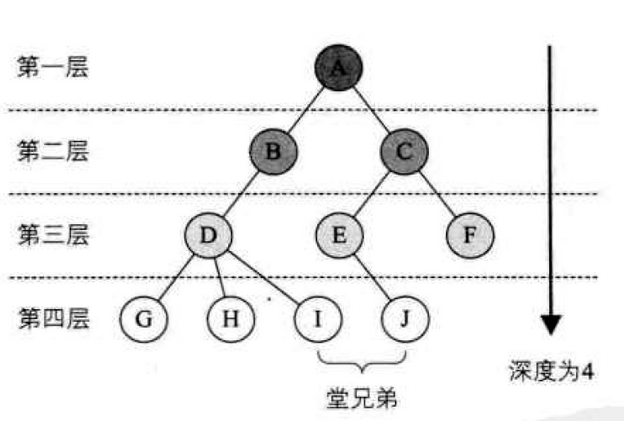
(3）、当n>1时，其余结点可分为m(m>0)个互不相交的有限集T1，T2，..., Tm。其中每个结点本身又是一棵树，并且是根的子树(SubTree):

（4）、**子树**，是指父母结点的**直接后代**，不包含间接的.

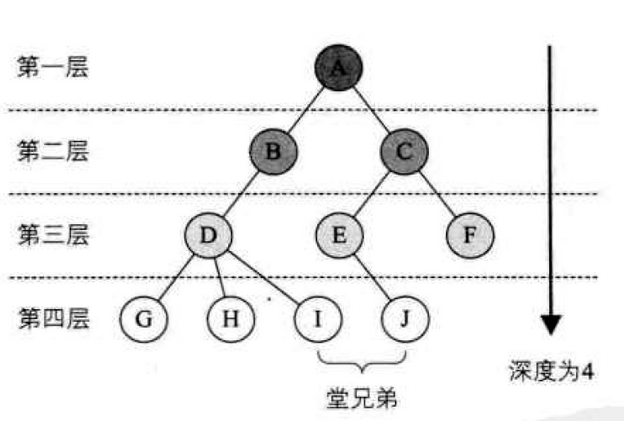
* + - 1.  概念
         1.  结点的度（Degree）：结点拥有的子树数。注意，结点的子树，是指其直接后代。



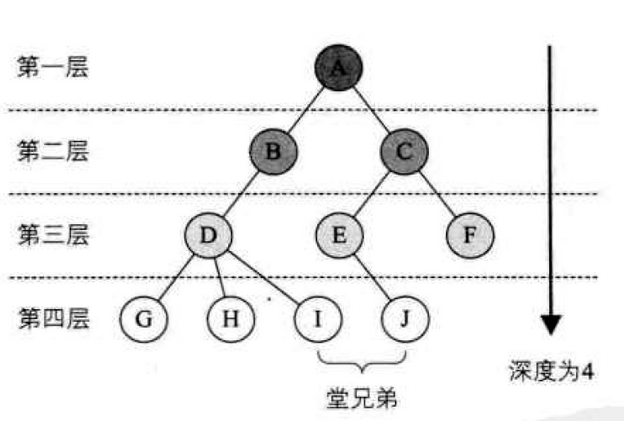
* + - * 1.  结点的层次(Level)：从根开始定义起，根为第一层，根的孩子为第二层



* + - * 1.  堂兄弟：双亲在同一层的结点互为堂兄弟。

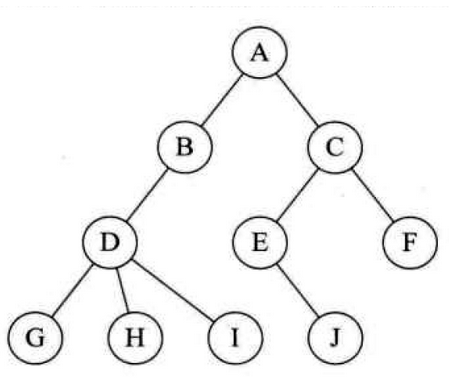


* + - * 1.  树的深度（Depth）或高度：树中结点的最大层次。



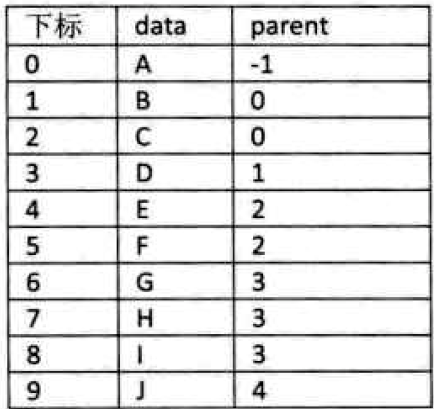
* + - * 1.  有序树：如果将树中各结点的子树看成是从左至右有次序的，不能互换的，那么这个树 就是有序树。否则，是无序树。
        2.  森林（Forest）：m(m>=0)棵互不相交的树的集合。对树中结点而言，其子树的集合，就是森林
        3.  叶结点（Leaf）或终端结点：度为0的结点。
        4.  非终端结点(分支结点)：度不为0的结点。
        5.  内部结点：除根结点之外的分支结点
        6.  树的度：树内各结点的度的最大值。
      1.  树的存储结构
         1.  双亲表示法： 以一组连续空间存储树的结点，每个结点除存储数据之外，还会存储一个指向该结点双亲 的指示器(双亲节点在连续空间中的位置)。

**树：**



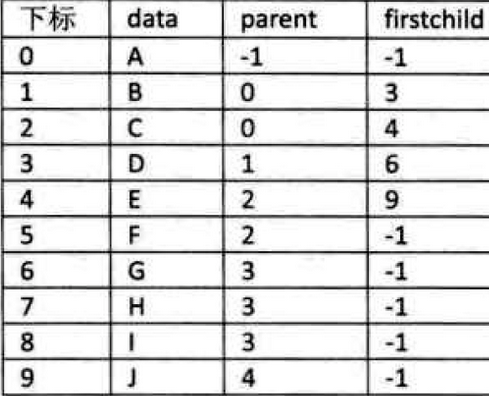
**存储结构：**

**（1）只保存双亲的双亲表示法：**

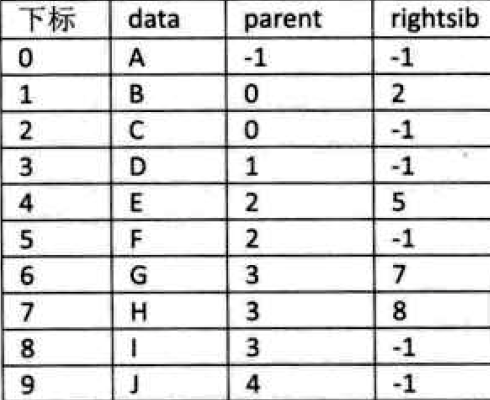


**（2）保存双亲、及其长子域**

**长子域是指结点的最左边的孩子结点。**

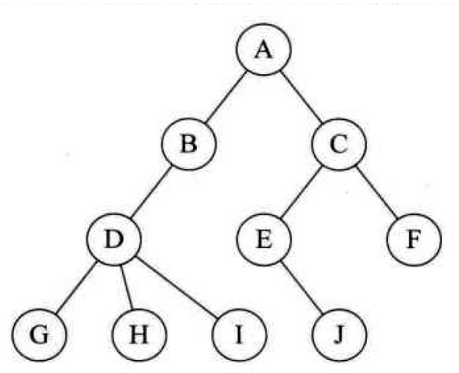


**（3）、保存双亲、右兄弟域的双亲表示法**

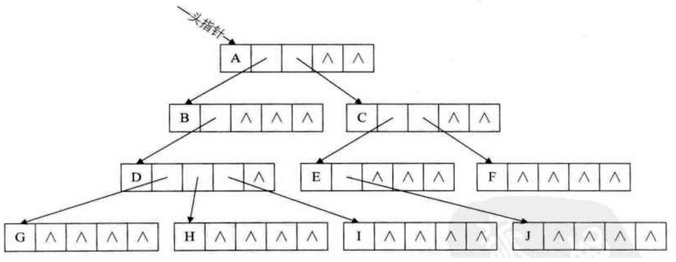


* + - * 1.  多重链表表示法：每个结点有多个指针域，其中每个指针指向一棵子树的根结点。

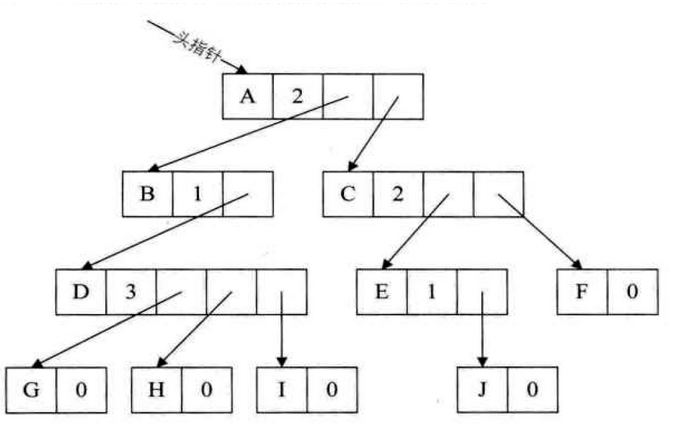
树的每个结点的度不同(有几个直接子节点就有几个指针域)，因此有两种解决方案：



**1、每个结点的指针域的数量等于树的度（会有浪费，因为树的度是所有节点中度最大的度）：**



**2、每个结点的指针域的数量等于该结点的度。但是，每个结点需要专门设置一个位置来存储该结点的度（比较复杂）:**



* + - * 1.  孩子表示法:由一个线性表和一组单链表组成

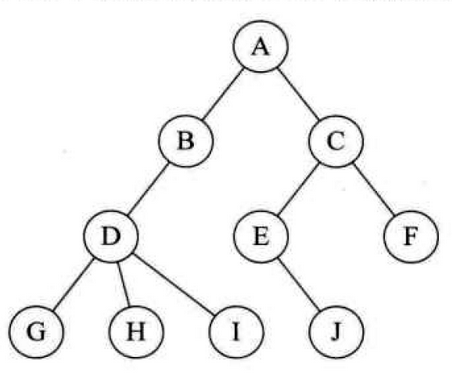
1、线性表存储树的n个结点：

线性表的每个单元，由数据域和指向**第一个**孩子的指针构成。

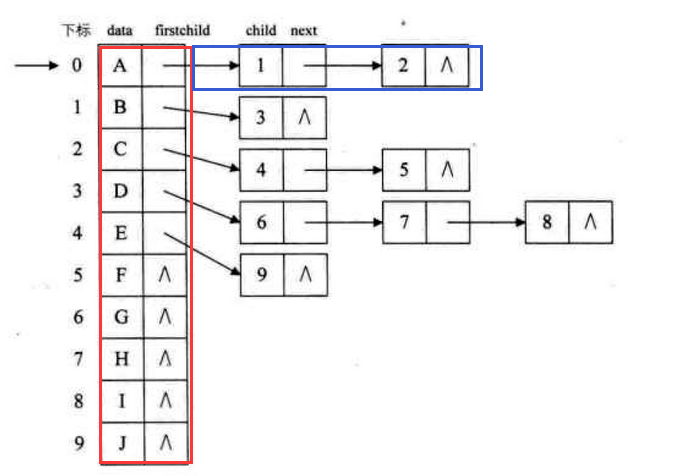
2、每个结点的孩子结点按顺序排列，存储在单链表中，那么n个结点，就有n个单链表。叶结点的单链表为空：

单链表的每个单元，由数据域和指向它的兄弟结点的指针域构成。

**树：**



**结构：**

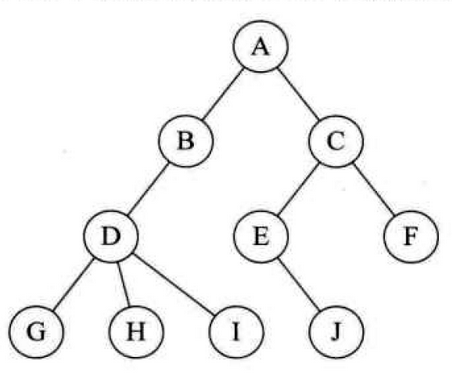


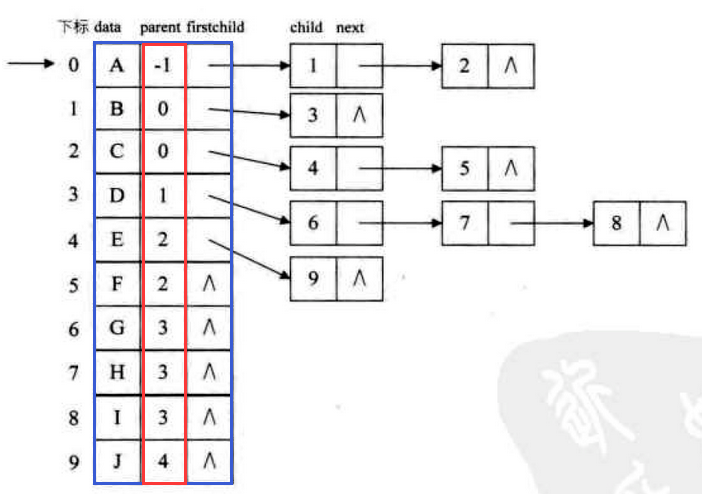
**缺点：**

查找某个节点的双亲节点比较复杂；

其中红色的 框表示使用线性表表示的节点；蓝色的表示使用单链表表示的子节点；

* + - * 1.  孩子表示法的改进--双亲孩子表示法

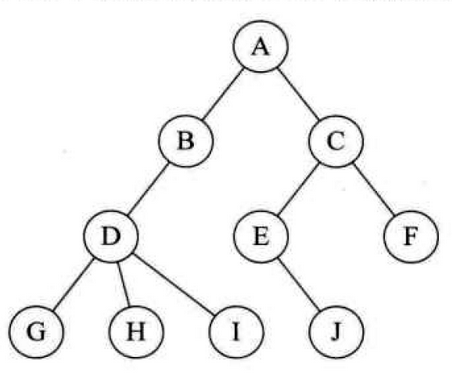


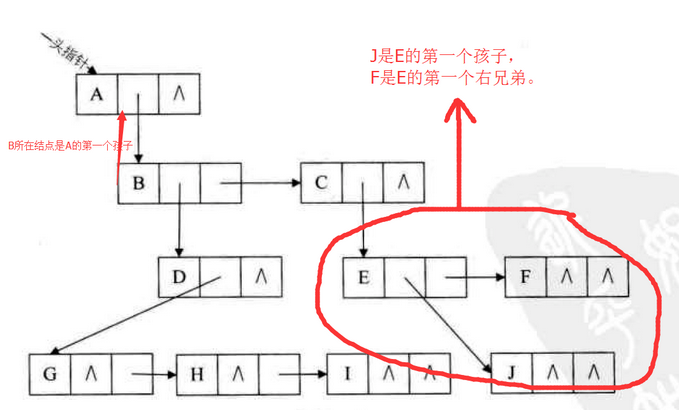


孩子双亲表示法解决了孩子表示法查找某个节点的双亲节点较复杂的问题：

蓝色的表示存储所有节点的线性表；添加了红色的部分，表示每一个节点的双亲节点的指针域；

* + - * 1.  孩子兄弟表示法:一个结点由三部分组成：数据域（data），第一个孩子（firstChild）和第一个 右兄弟（rightsib）。





* + - 1.  应用实例

See document(s): [45e056315c62](http://www.jianshu.com/p/45e056315c62)

**文章评论数据结构的设计**

**需求**

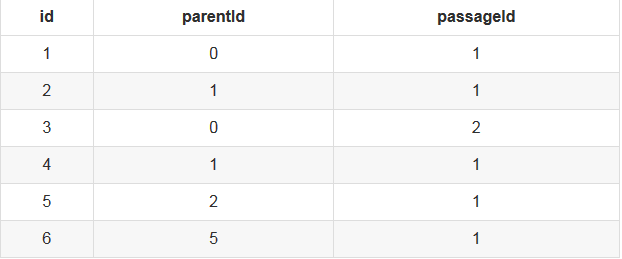
 1、求一篇文章的所有评论数量

 2、无限级评论（不限制每一条评论的层级）

 3、求一篇文章下的所有评论数据，包含对该文章的直接评论的评论。

**思考**

**先看“双亲表示法”的实现：**



**id表示评论记录的主键，parentId表示当前评论的双亲评论，passageId：表示文章编号；**

**第1条评论记录是文章编号1的第一级评论，**

**第2条评论记录是文章编号1的评论，且他是对第1条评论的评论，所以是第二级评论；**

* + 1.  二叉树
       1.  二叉树的概念

二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)；

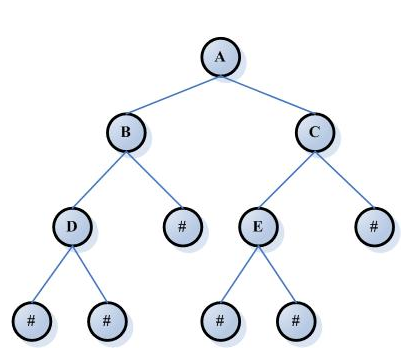
二叉树的子树有左右之分，次序不能颠倒；

二叉树的第i层至多有2^(i-1)个结点；

深度为k的二叉树至多有2^K - 1个结点； **如下图最多有15个节点**

对任何一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则n0=n2+1：

**如下图，终端节点数：6，度为2的节点数：5**



* + - 1.  特殊二叉树

：

* + - * 1.  左斜树

左斜树是所有的结点都只有左子树的二叉树

* + - * 1.  右斜树

右斜树是所有的结点都只有右子树的二叉树

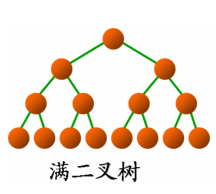
* + - * 1.  满二叉树

**概念：**

除最后一层无任何子节点外，每一层上的所有结点都有两个子结点。

也可以这样理解，除叶子结点外的所有结点均有两个子结点。

树的节点数达到最大值，所有叶子结点必须在同一层上。



**性质：**

1) 一颗树深度为h，最大层数为k，深度与最大层数相同，k=h;

2) 叶子数为2^h;

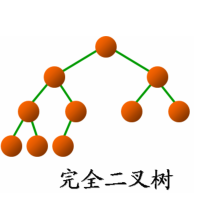
3) 第k层的结点数是：2^(k-1);

4) 总结点数是：2^k-1，且**总节点数一定是奇数**。

* + - * 1.  完全二叉树

**定义：**

 若设二叉树的深度为h，除第 h 层外，其它各层 (1～(h-1)层) 的结点数都达到最大个数，第h层所有的结点都连续集中在最左边，这就是完全二叉树。



完全二叉树是效率很高的数据结构，堆是一种完全二叉树或者近似完全二叉树，所以效率极高，像十分常用的排序算法、Dijkstra算法、Prim算法等都要用堆才能优化，二叉排序树的效率也要借助平衡性来提高，而**平衡性基于完全二叉树**

* + - 1.  性质

**1) 在非空二叉树中，第 i 层的结点总数不超过2^(i-1), i>=1;**

**2) 深度为 h 的二叉树最多有2^h-1个结点(h>=1)，最少有h个结点（斜树）;**

**3) 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1;**

**4) 具有n个结点的完全二叉树的深度为log2(n+1);**

**5)有N个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：**

**若I为结点编号则 :**

**如果I>1，则其父结点的编号为I/2；**

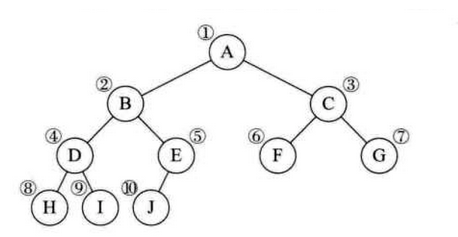
**如果2I<=N，则其左儿子（即左子树的根结点）的编号为2I；若2I>N，则无左儿子；**

**如果2I+1<=N，则其右儿子的结点编号为2I+1；若2I+1>N，则无右儿子。**

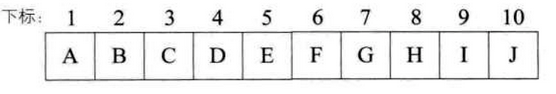
**6)给定N个节点，能构成h(N)种不同的二叉树，其中h(N)为卡特兰数的第N项，h(n)=C(2\*n, n)/(n+1)。**

**7)设有i个枝点，I为所有枝点的道路长度总和，J为叶的道路长度总和J=I+2i。**

* + - 1.  二叉树的存储结构
         1.  顺序存储结构



将上边的二叉树使用顺序的存储方式存入数组中，相应的坐标对应其同样的位置：

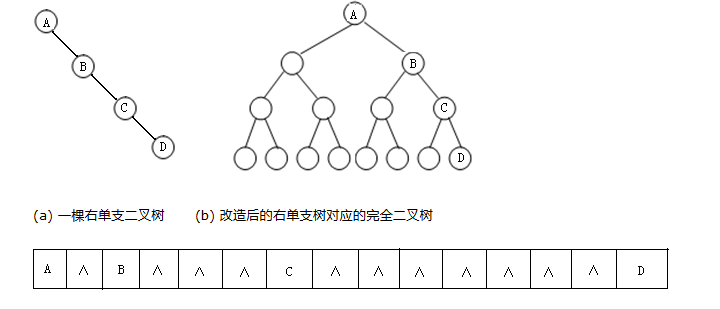


优点：

根据二叉树的性质五，能很快查出一个结点的双亲、左孩子和右孩子；

缺点：

有些二叉树，比如右斜树，采用顺序存储结构，非常浪费空间：



* + - * 1.  二叉链表

**概念：**

二叉树的链式存储结构（简称二叉链表）是指**用一个链表来存储一棵二叉树**，二叉树中**每一个结点用链表中的一个链结点**来存储。在二叉树中，标准存储方式的结点结构如图所示：

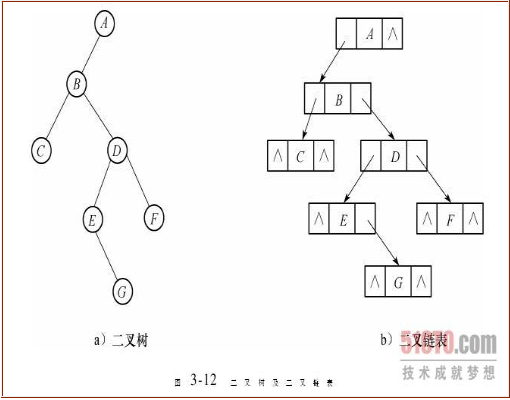


其中：

data表示值域，用于存储对应的数据元素；

lchild和rchild分别表示左指针域和右指针域，分别用于存储左子女结点和右子女结点（即左、右子树的根结点）的**存储位**置（即指针）。

**示例：**



**缺点：**

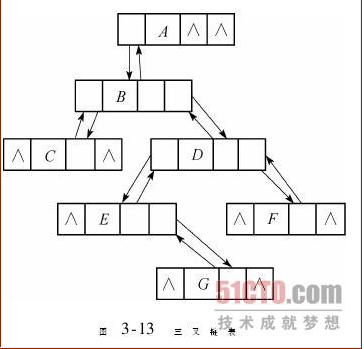
**查找某个节点的双亲节点不方便；**

* + - * 1.  三叉链表

**概念：**

为了便于查找任一结点的双亲结点，可以在结点中再增加一个指针域parent，它称为三叉链表；

**示例：**



* + - 1.  遍历二叉树

二叉树的遍历（traversing binary tree），是指从**根结点**出发，按照某种访问次序，访问二叉

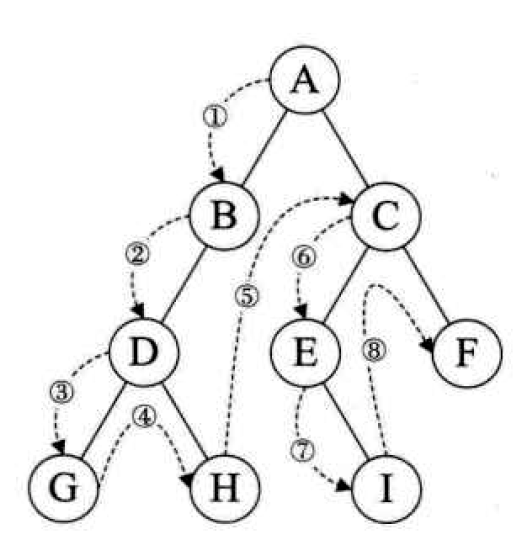
树的所有结点。对所有的结点，都必须**访问且只访问一次**

* + - * 1.  前序遍历

**规则是：**

 若二叉树为空，则空操作返回，否则先访问根结点，然后前序遍历左子树，再前序遍历右子树：

**示例：**



 ABDGHCEIF

**算法：**

void PreOrderTraverse(BiTree T)

{

   if(T==NULL){

       return;

   }

   printf("%c", T->data);

   PreOrderTraverse(T->lchild);    // 前序遍历左子树

   PreOrderTraverse(T->rchild);    // 前序遍历右子树

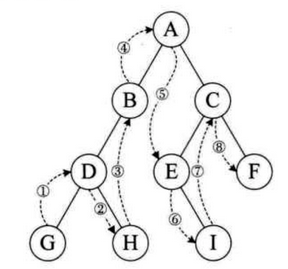
}

* + - * 1.  中序遍历

**规则：**

 若树为空，则空操作返回；否则从根节点开始（并不是先访问根节点），中序遍历根节点的左子树，然后访问根节点，最后中序遍历右子树;

**示例：**



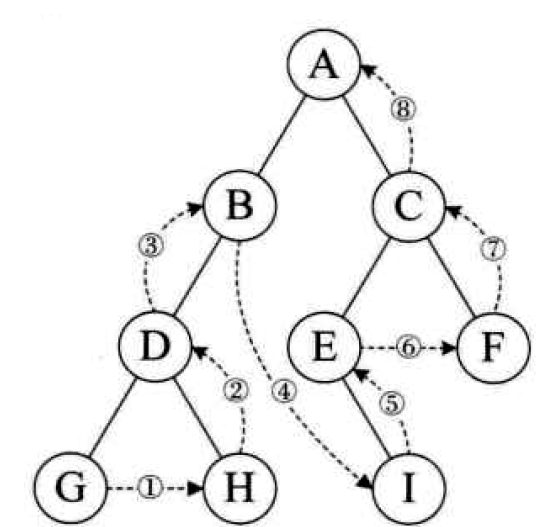
**结果： GDHBAEICF**

* + - * 1.  后序遍历

**规则：**

若树为空，空操作返回；否则，从左到右先叶子；最后访问根节点；

**示例：**



**结果：**

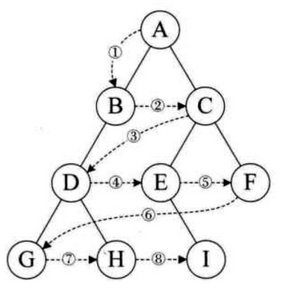
GHDBIEFCA

* + - * 1.  层序遍历

**规则：**

从树的第一层，也就是根节点开始访问，从上而下逐层遍历，在同一层按照从左到右的顺序遍历：

**示例：**



**结果：**

ABCDEFGHI

* + 1.  二叉查找树

又称为是二叉排序树（Binary Sort Tree）或二叉搜索树

* + - 1.  定义

有如下性质的**二叉树**就是二叉排序树：

1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；

2) 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；

3) 左、右子树也分别为二叉排序树；

4) 没有值相等的节点。

* + - 1.  性质

1、**对二叉查找树进行中序遍历，即可得到有序的数列**

**2、二叉查找树的高度决定了二叉查找树的查找效率**

* + - 1.  时间复杂度

二叉查找树的查找和二分查找一样：

插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。

原因在于插入和删除元素的时候，**树没有保持平衡,**我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是**平衡查找树**设计的初衷;

* + - 1.  插入过程

源码：

struct node

{

   int val;

   pnode lchild;

   pnode rchild;

};

pnode BT = NULL;

**//递归方法插入节点**

pnode **insert**(pnode root, int x)     **//从根节点开始插入**

{

   pnode p = (pnode)malloc(LEN);

   p->val = x;

   p->lchild = NULL;

   p->rchild = NULL;     //分配一块足够大的内存给需要插入的节点，并且设置节点的值及初始化其左右子树的指针；

   if(root == NULL){

       root = p;   //表示当前二叉树的root节点为空，则插入的元素就是根节点

   }

   else if(x < root->val){   //如果被插入元素的值小于根元素的值，则递归调用insert方法,将被插入的节点插入根节点的左边

       root->lchild = **insert**(root->lchild, x);

   }

   else{

       root->rchild = **insert**(root->rchild, x);

   }

   return root;

}

**//非递归方法插入节点**

void insert\_BST(pnode q, int x)

{

   pnode p = (pnode)malloc(LEN);

   p->val = x;

   p->lchild = NULL;

   p->rchild = NULL;

   if(q == NULL){

       BT = p;

       return ;

   }

   while(q->lchild != p && q->rchild != p){

       if(x < q->val){

           if(q->lchild){

               q = q->lchild;

           }

           else{

               q->lchild = p;

           }

       }

       else{

           if(q->rchild){

               q = q->rchild;

           }

           else{

               q->rchild = p;

           }

       }

   }

   return;

}

* + - * 1.  若当前的二叉查找树为空，则插入的元素为根节点;
        2.  若插入的元素值小于根节点值，则将元素插入到左子树中
        3.  若插入的元素值不小于根节点值，则将元素插入到右子树中
      1.  删除操作

bool delete\_BST(pnode p, int x) //返回一个标志，表示是否找到被删元素

{

  bool find = false;

  pnode q;  **//q是被删除节点的双亲节点**

  p = BT;

  while(p && !find){  **//寻找被删元素**

      if(x == p->val){ //找到被删元素

          find = true;

      }

      else if(x < p->val){ //沿左子树找

          q = p;

          p = p->lchild;

      }

      else{ //沿右子树找

          q = p;

          p = p->rchild;

      }

  }

  if(p == NULL){ //没找到

      cout << "没有找到" << x << endl;

  }

  if(p->lchild == NULL && p->rchild == NULL){ //p为叶子节点

      if(p == BT){ //p为根节点

          BT = NULL;

      }

      else if(q->lchild == p){

          q->lchild = NULL;

      }

      else{

          q->rchild = NULL;

      }

      free(p); //释放节点p

  }

  else if(p->lchild == NULL || p->rchild == NULL){ //p为单支子树

      if(p == BT){ //p为根节点

          if(p->lchild == NULL){

              BT = p->rchild;

          }

          else{

              BT = p->lchild;

          }

      }

      else{

          if(q->lchild == p && p->lchild){ //p是q的左子树且p有左子树

              q->lchild = p->lchild;   //将p的左子树链接到q的左指针上

          }

          else if(q->lchild == p && p->rchild){

              q->lchild = p->rchild;

          }

          else if(q->rchild == p && p->lchild){

              q->rchild = p->lchild;

          }

          else{

              q->rchild = p->rchild;

          }

      }

      free(p);

  }

  else{ //p的左右子树均不为空

      pnode t = p;

      pnode s = p->lchild; //从p的左子节点开始

      while(s->rchild){ //找到p的前驱，即p左子树中值最大的节点

          t = s;

          s = s->rchild;

      }

      p->val = s->val; //把节点s的值赋给p

      if(t == p){

          p->lchild = s->lchild;

      }

      else{

          t->rchild = s->lchild;

      }

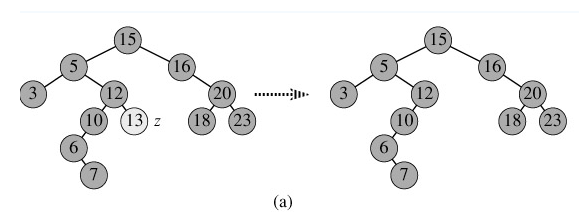
      free(s);

  }

  return find;

}

* + - * 1.  p为叶子节点，直接删除该节点， 再修改其父节点的指针（注意分被删除的节点是根节点和不是根节点），如图



if(p->lchild == NULL && p->rchild == NULL){  **//p为叶子节点**

       if(p == BT){ //p为根节点

           BT = NULL;

       }

       else if(q->lchild == p){

           q->lchild = NULL;

       }

       else{

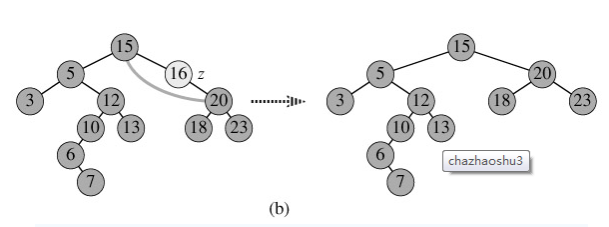
           q->rchild = NULL;

       }

       free(p); //释放节点p

   }

* + - * 1.  p为单支节点（即只有左子树或右子树）。 让p的子树与p的父亲节点相连，删除p即可（注意分是根节点和不是根节点）

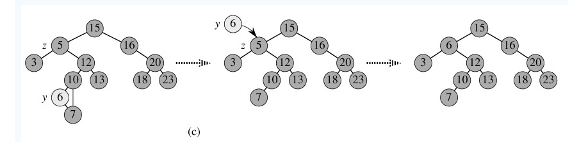


如图，被删除的节点16 是单支节点，只有右子树。

只要将16的双亲节点指向16的孩子节点，然后删除16节点即可

* + - * 1.  p的左子树和右子树均不空

 方法一： 找到p的后继y，因为y一定没有左子树，所以可以删除y， 并让y的父亲节点成为y的右子树的父亲节点，并用y的值代替p的值；



注意：

**后继是指比**当前节点值大的第一个节点：

 如本例中要删除数值是5的节点，他的后继就是值为6的节点，很显然6一定没有左子节点；

 前驱是指比当前节点值小的第一个节点：

 方法二： 找到p的前驱x，x一定没有右子树，所以可以删除x， 并让x的父亲节点成为x的左子树的父亲节点。

* + - 1.  查找操作

pnode search\_BST(pnode p, int x)

{

   bool solve = false;

   while(p && !solve){

       if(x == p->val){

           solve = true;

       }

       else if(x < p->val){

           p = p->lchild;

       }

       else{

           p = p->rchild;

       }

   }

   if(p == NULL){

       cout << "没有找到" << x << endl;

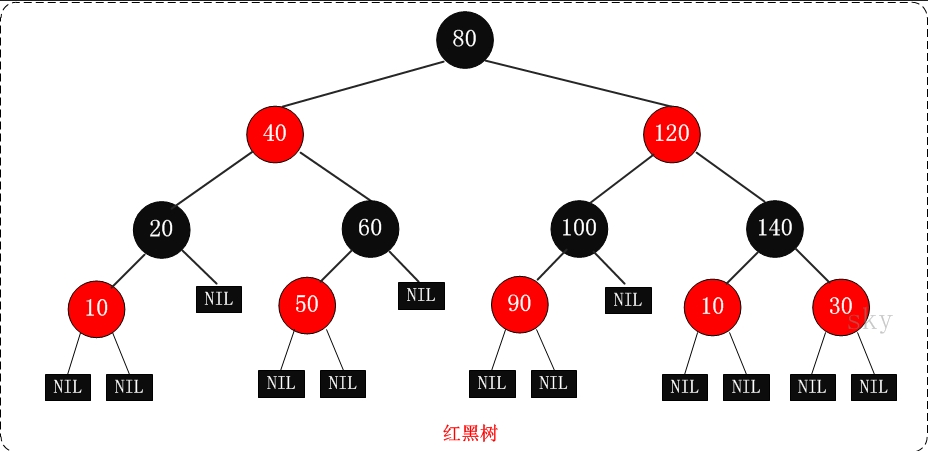
   }

   return p;

}

* + 1.  红黑树

**R-B Tree**，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。



* + - 1.  红黑树的特性

1. 每个节点或者是黑色，或者是红色；
2. 根节点是黑色；
3. 每个叶子节点（NIL）是黑色： [注意：这里叶子节点，**是指为空(NIL或NULL)的叶子节点！**]；
4. 如果一个节点是红色的，则它的子节点必须是黑色的；
5. 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点（确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树）；

* + - 1. 红黑树的应用

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高：

* java中的 TreeSet；
* java中的 TreeMap
* c++ stl中的set/map
* Linux虚拟内存的管理

* + - 1.  复杂度

红黑树的时间复杂度为: **O(lgn)：**

**定理：一棵含有n个节点的红黑树的高度至多为2log(n+1).**

证明：

   "一棵含有n个节点的红黑树的高度至多为2log(n+1)" 的逆否命题是 "高度为h的红黑树，它所包含的内节点个数至少为 2^(h/2)-1个"。

   我们只需要证明逆否命题，即可证明原命题为真；即只需证明 "高度为h的红黑树，它的包含的内节点个数至少为 2h/2-1个"。

   从某个节点x出发（不包括该节点）到达一个叶节点的任意一条路径上，黑色节点的个数称为该节点的黑高度(x's black height)，记为bh(x)。关于bh(x)有两点需要说明：

   第1点：根据红黑树的"特性(5) ，即从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点"可知，从节点x出发到达的所有的叶节点具有相同数目的黑节点。这也就意味着，bh(x)的值是唯一的！

   第2点：根据红黑色的"特性(4)，即如果一个节点是红色的，则它的子节点必须是黑色的"可知，从节点x出发达到叶节点"所经历的黑节点数目">= "所经历的红节点的数目"。假设x是根节点，则可以得出结论"bh(x) >= h/2"。进而，我们只需证明 "高度为h的红黑树，它的包含的黑节点个数至少为 2bh(x)-1个"即可。

   到这里，我们将需要证明的定理已经由

"一棵含有n个节点的红黑树的高度至多为2log(n+1)"

   转变成只需要证明

"高度为h的红黑树，它的包含的内节点个数至少为 2bh(x)-1个"。

下面通过"数学归纳法"开始论证高度为h的红黑树，它的包含的内节点个数至少为 2bh(x)-1个"。

(01) 当树的高度h=0时，

   内节点个数是0，bh(x) 为0，2bh(x)-1 也为 0。显然，原命题成立。

(02) 当h>0，且树的高度为 h-1 时，它包含的节点个数至少为 2bh(x)-1-1。这个是根据(01)推断出来的！

   下面，由树的高度为 h-1 的已知条件推出“树的高度为 h 时，它所包含的节点树为 2bh(x)-1”。

   当树的高度为 h 时，

   对于节点x(x为根节点)，其黑高度为bh(x)。

   对于节点x的左右子树，它们黑高度为 bh(x) 或者 bh(x)-1。

   根据(02)的已知条件，我们已知 "x的左右子树，即高度为 h-1 的节点，它包含的节点至少为 2bh(x)-1-1 个"；

   所以，节点x所包含的节点至少为 ( 2bh(x)-1-1 ) + ( 2bh(x)-1-1 ) + 1 = 2^bh(x)-1。即节点x所包含的节点至少为 2bh(x)-1。

   因此，原命题成立。

   由(01)、(02)得出，"高度为h的红黑树，它的包含的内节点个数至少为 2^bh(x)-1个"。

   因此，“一棵含有n个节点的红黑树的高度至多为2log(n+1)”。

* + - 1. 红黑树的基本操作
         1.  左旋和右旋

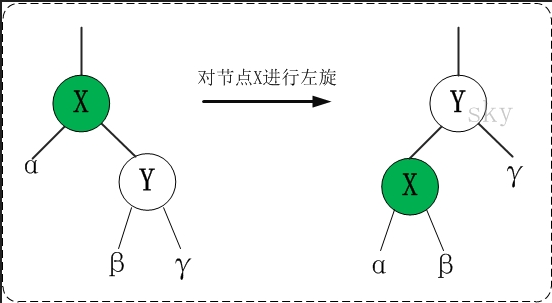
红黑树的基本操作是添加、删除。

在对红黑树进行添加或删除之后，都会用到旋转方法。为什么呢？

道理很简单，添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。**而通过旋转，可以使这颗树重新成为红黑树**。简单点说，旋转的目的是让树保持红黑树的特性。

旋转包括两种：左旋 和 右旋。

 左旋



**对x进行左旋，意味着"将x变成一个左节点"**

**伪代码：**

**LEFT-ROTATE(T, x)**

**01 y ← right[x]            // 前提：这里假设x的右孩子为y。下面开始正式操作**

**02 right[x] ← left[y]      // 将 “y的左孩子” 设为 “x的右孩子”，即 将β设为x的右孩子**

**03 p[left[y]] ← x          // 将 “x” 设为 “y的左孩子的父亲”，即 将β的父亲设为x**

**04 p[y] ← p[x]             // 将 “x的父亲” 设为 “y的父亲”**

**05 if p[x] = nil[T]**

**06   then root[T] ← y      // 情况1：如果 “x的父亲” 是空节点，则将y设为根节点**

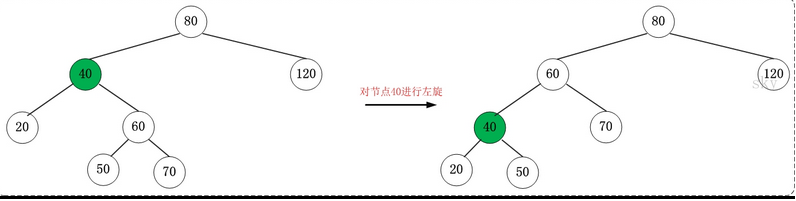
**07 else if x = left[p[x]]**

**08     then left[p[x]] ← y   // 情况2：如果 x是它父节点的左孩子，则将y设为“x的父节点的左孩子”**

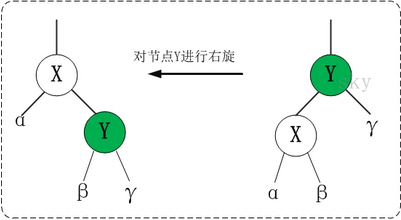
**09 else right[p[x]] ← y     // 情况3：(x是它父节点的右孩子) 将y设为“x的父节点的右孩子”**

**10 left[y] ← x              // 将 “x” 设为 “y的左孩子”**

**11 p[x] ← y                 // 将 “x的父节点” 设为 “y”**



 右旋



**对x进行左旋，意味着"将x变成一个左节点"**

RIGHT-ROTATE(T, y)

x ← left[y]             **// 前提：这里假设y的左孩子为x。下面开始正式操作**

left[y] ← right[x]      **// 将 “x的右孩子” 设为 “y的左孩子”，即 将β设为y的左孩子**

p[right[x]] ← y         **// 将 “y” 设为 “x的右孩子的父亲”，即 将β的父亲设为y**

p[x] ← p[y]             **// 将 “y的父亲” 设为 “x的父亲”**

if p[y] = nil[T]

   then root[T] ← x       **// 情况1：如果 “y的父亲” 是空节点，则将x设为根节点**

else if y = right[p[y]]

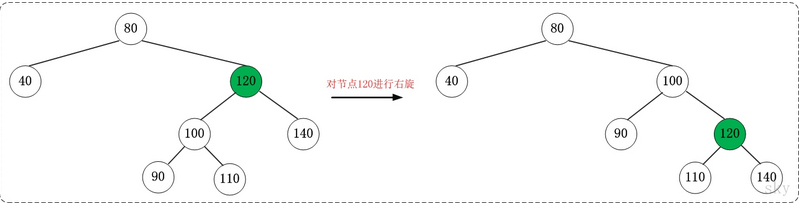
   then right[p[y]] ← x   **// 情况2：如果 y是它父节点的右孩子，则将x设为“y的父节点的右孩子”**

else

      left[p[y]] ← x         **// 情况3：(y是它父节点的左孩子) 将x设为“y的父节点的左孩子”**

right[x] ← y           // 将 “y” 设为 “x的右孩子”

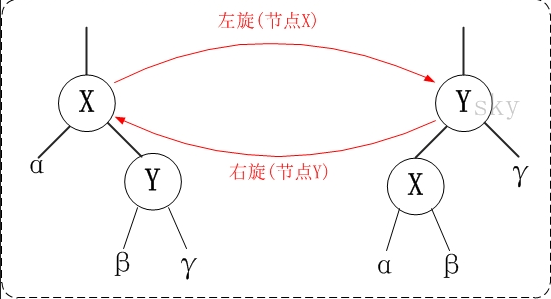
p[y] ← x               // 将 “y的父节点” 设为 “x”



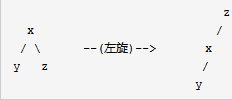
 区分左旋和右旋

仔细观察"左旋"和"右旋"的示意图。我们能清晰的发现，它们是对称的。

无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树；



**左旋示例图(以x为节点进行左旋)：**

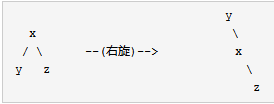


对x进行左旋，意味着，将“x的右孩子”设为“x的父亲节点”；

即，将 x变成了一个左节点(x成了为z的左孩子)！。

**因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”**

**右旋示例图(以x为节点进行右旋)：**



对x进行右旋，意味着，将“x的左孩子”设为“x的父亲节点”；

即，将 x变成了一个右节点(x成了为y的右孩子)！

**因此，右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。**

* + - * 1.  添加节点

将一个节点插入到红黑树中，需要执行哪些步骤呢？

**首先，**将红黑树当作一颗二叉查找树，将节点插入；

**然后，**将节点着色为红色；

**最后，**通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。

 将红黑树当作一颗二叉查找树，将节点插入

See also: [插入过程](#插入过程)

1、红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。

也就意味着，**树的键值仍然是有序的**。

2、无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。

这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

**伪代码实现：将节点Z插入红黑树（二叉查找树T中）**

RB-INSERT(T, z)

y ← nil[T]                        // 新建节点“y”，将y设为空节点。

x ← root[T]                       // 设“红黑树T”的根节点为“x”

while x ≠ nil[T]                  // 找出要插入的节点“z”在二叉树T中的位置“y”

    do y ← x

       if key[z] < key[x]

          then x ← left[x]

          else x ← right[x]

p[z] ← y                          // 设置 “z的父亲” 为 “y”

if y = nil[T]

   then root[T] ← z               // 情况1：若y是空节点，则将z设为根

else if key[z] < key[y]

   then left[y] ← z       // 情况2：若“z所包含的值” < “y所包含的值”，则将z设为“y的左孩子”

else

    right[y] ← z         // 情况3：(“z所包含的值” >= “y所包含的值”)将z设为“y的右孩子”

left[z] ← nil[T]                  // z的左孩子设为空

right[z] ← nil[T]                 // z的右孩子设为空。至此，已经完成将“节点z插入到二叉树”中了。

 将插入的节点着色为"红色"

为什么着色成红色，不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：Ò

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]

(4) 如果一个节点是红色的，则它的子节点必须是黑色的。

(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

将插入的节点着色为红色，**不会违背"特性(5)"**！少违背一条特性，就意味着我们需要处理的情况越少

**伪代码：**

color[z] ← RED                    // 将z着色为“红色”

 通过一系列的旋转或着色等操作， 使之重新成为一颗红黑树

第二步中，将插鍌点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？

**对于"特性(1)"**，显然不会违背了。因为我们已经将它涂成红色了。

**对于"特性(2)"**，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。

**对于"特性(3)"**，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。

**对于"特性(4)"**，是有可能违背的！

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

**伪代码实现：**

RB-INSERT-FIXUP(T, z)

while color[p[z]] = RED      // 若“当前节点(z)的父节点是红色”，则进行以下处理。

   do if p[z] = left[p[p[z]]]    **// 若“z的父节点”是“z的祖父节点的左孩子”，则进行以下处理。**

         then y ← right[p[p[z]]]   // 将y设置为“z的叔叔节点(z的祖父节点的右孩子)”

              if color[y] = RED                **// Case 1条件：叔叔是红色**

                 then color[p[z]] ← BLACK     ▹ Case 1   // (01) 将“父节点”设为黑色。

                      color[y] ← BLACK        ▹ Case 1   // (02) 将“叔叔节点”设为黑色。

                      color[p[p[z]]] ← RED    ▹ Case 1   // (03) 将“祖父节点”设为“红色”。

                      z ← p[p[z]]             ▹ Case 1   // (04) 将“祖父节点”设为“当前节点”(红色节点)

               else if z = right[p[z]]         **// Case 2条件：叔叔是黑色，且当前节点是右孩子**

                         then z ← p[z]           ▹ Case 2   // (01) 将“父节点”作为“新的当前节点”。

                              LEFT-ROTATE(T, z)   ▹ Case 2   // (02) 以“新的当前节点”为支点进行左旋。

               else

                    color[p[z]] ← BLACK ▹ Case 3   // Case 3条件：叔叔是黑色，且当前节点是左孩子。(01) 将“父节点”设为“黑色”。

                         color[p[p[z]]] ← RED               ▹ Case 3   // (02) 将“祖父节点”设为“红色”。

                         RIGHT-ROTATE(T, p[p[z]])            ▹ Case 3   // (03) 以“祖父节点”为支点进行右旋。

    else (same as then clause with "right" and "left" exchanged)      **// 若“z的父节点”是“z的祖父节点的右孩子”，将上面的操作中“right”和“left”交换位置，然后依次执行。**

color[root[T]] ← BLACK

 被插入的节点是根节点 直接把此节点涂为黑色

 被插入的节点的父节点是黑色 什么也不需要做。节点被插入后，仍然是红黑树

 被插入的节点的父节点是红色

该情况与红黑树的“特性(5)”相冲突。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为3种情况(Case)

 当前节点的祖父节点的另一个子节点（叔叔节点）也是红色

**策略：**

(01) 将“父节点”设为黑色。

(02) 将“叔叔节点”设为黑色。

(03) 将“祖父节点”设为“红色”。

(04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。

**伪代码实现**

if color[y] = RED                **// Case 1条件：叔叔是红色**

then color[p[z]] ← BLACK    ▹ Case 1 // (01) 将“父节点”设为黑色。

    color[y] ← BLACK           ▹ Case 1 // (02) 将“叔叔节点”设为黑色。

    color[p[p[z]]] ← RED      ▹ Case 1 // (03) 将“祖父节点”设为“红色”。

    z ← p[p[z]]                  ▹ Case 1 // (04) 将“祖父节点”设为“当前节点”(红色节点)

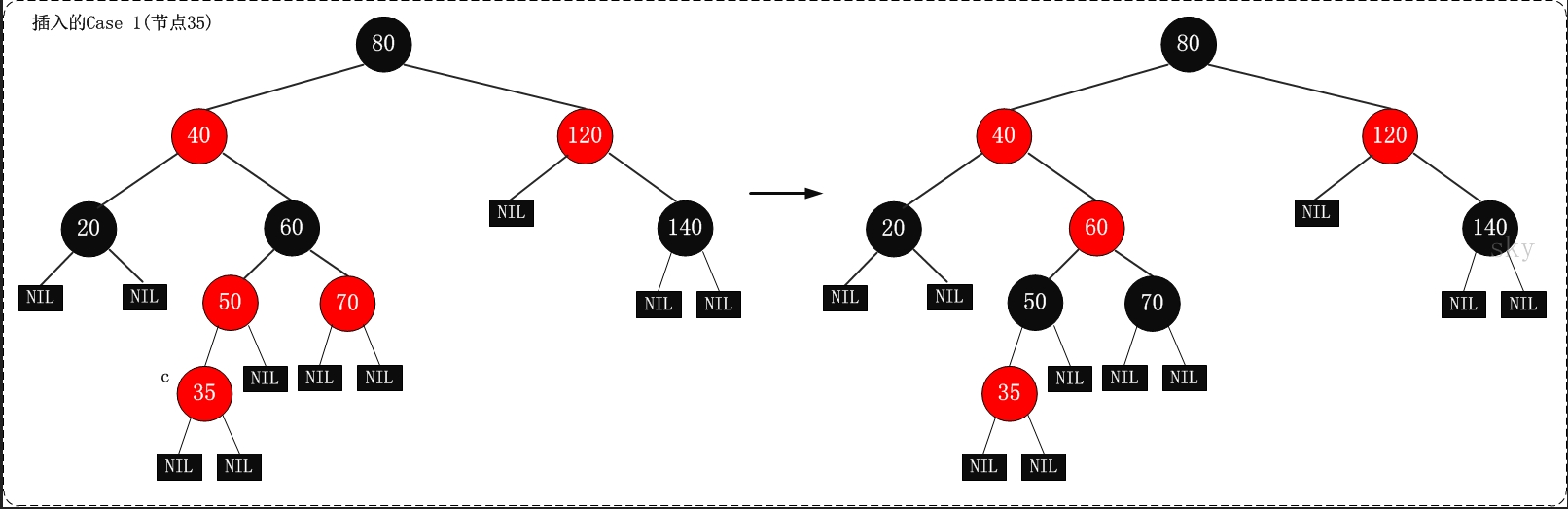
**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)

“当前节点”和“父节点”都是红色，违背“特性(4)”。所以，将“父节点”设置“黑色”以解决这个问题。

但是，将“父节点”由“红色”变成“黑色”之后，违背了“特性(5)”：

 因为，包含“父节点”的分支的黑色节点的总数增加了1。解决这个问题的办法是：将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”。关于这里，说明几点：第一，为什么“祖父节点”之前是黑色？这个应该很容易想明白，因为在变换操作之前，该树是红黑树，“父节点”是红色，那么“祖父节点”一定是黑色。 第二，为什么将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；能解决“包含‘父节点’的分支的黑色节点的总数增加了1”的问题。这个道理也很简单。“包含‘父节点’的分支的黑色节点的总数增加了1” 同时也意味着 “包含‘祖父节点’的分支的黑色节点的总数增加了1”，既然这样，我们通过将“祖父节点”由“黑色”变成“红色”以解决“包含‘祖父节点’的分支的黑色节点的总数增加了1”的问题； 但是，这样处理之后又会引起另一个问题“包含‘叔叔’节点的分支的黑色节点的总数减少了1”，现在我们已知“叔叔节点”是“红色”，将“叔叔节点”设为“黑色”就能解决这个问题。 所以，将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；就解决了该问题。

   按照上面的步骤处理之后：当前节点、父节点、叔叔节点之间都不会违背红黑树特性，但祖父节点却不一定。若此时，祖父节点是根节点，直接将祖父节点设为“黑色”，那就完全解决这个问题了；若祖父节点不是根节点，那我们需要将“祖父节点”设为“新的当前节点”，接着对“新的当前节点”进行分析。



 叔叔节点是黑色的

 当前节点（被插入的节点）是其父节点的右孩子

**策略：**

(01) 将“父节点”作为“新的当前节点”。

(02) 以“新的当前节点”为支点进行左旋。

**伪代码实现**

else if z = right[p[z]]         **// Case 2条件：叔叔是黑色，且当前节点是右孩子**

 then z ← p[z]          ▹ Case 2 // (01) 将“父节点”作为“新的当前节点”。

  LEFT-ROTATE(T, z) ▹ Case 2 // (02) 以“新的当前节点”**为支点进行左旋**。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)

 为了便于理解，我们先说明第(02)步，再说明第(01)步；为了便于说明，我们设置“父节点”的代号为F(Father)，“当前节点”的代号为S(Son)。

 1、为什么要“以F为支点进行左旋”呢？

 根据已知条件可知：S是F的右孩子。而之前我们说过，我们处理红黑树的核心思想：将红色的节点移到根节点；然后，将根节点设为黑色。既然是“将红色的节点移到根节点”，**那就是说要不断的将破坏红黑树特性的红色节点上移**(即向根方向移动)。 而S又是一个右孩子，因此，我们可以通过“左旋”来将S上移！

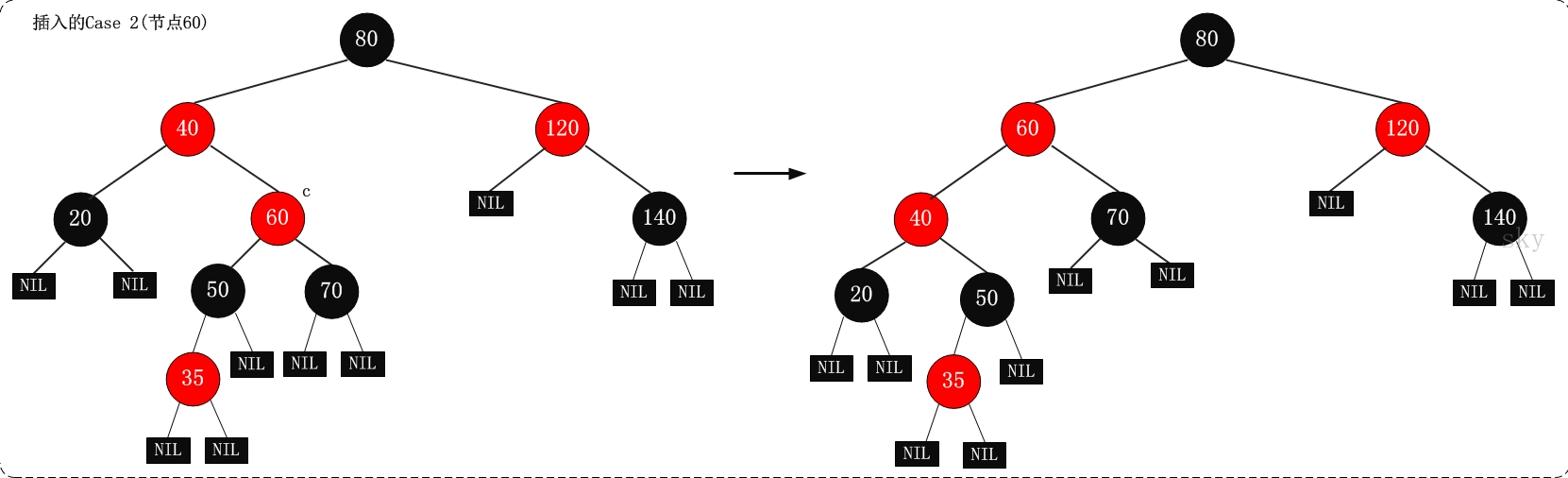
 2、按照上面的步骤(以F为支点进行左旋)处理之后：

    （1）、若S变成了根节点，那么直接将其设为“黑色”，就完全解决问题了；

    （2）、若S不是根节点，那我们需要执行步骤(01)，即“将F设为‘新的当前节点’”。

      那为什么不继续以S为新的当前节点继续处理，而需要以F为新的当前节点来进行处理呢？

      这是因为“左旋”之后，F变成了S的“子节点”，即S变成了F的父节点；而我们处理问题的时候，需要从下至上(由叶到根)方向进行处理；也就是说，必须先解决“孩子”的问题，再解决“父亲”的问题；所以，我们执行步骤(01)：将“父节点”作为“新的当前节点”



 当前节点是其父节点的左孩子

**策略：**

(01) 将“父节点”设为“黑色”。

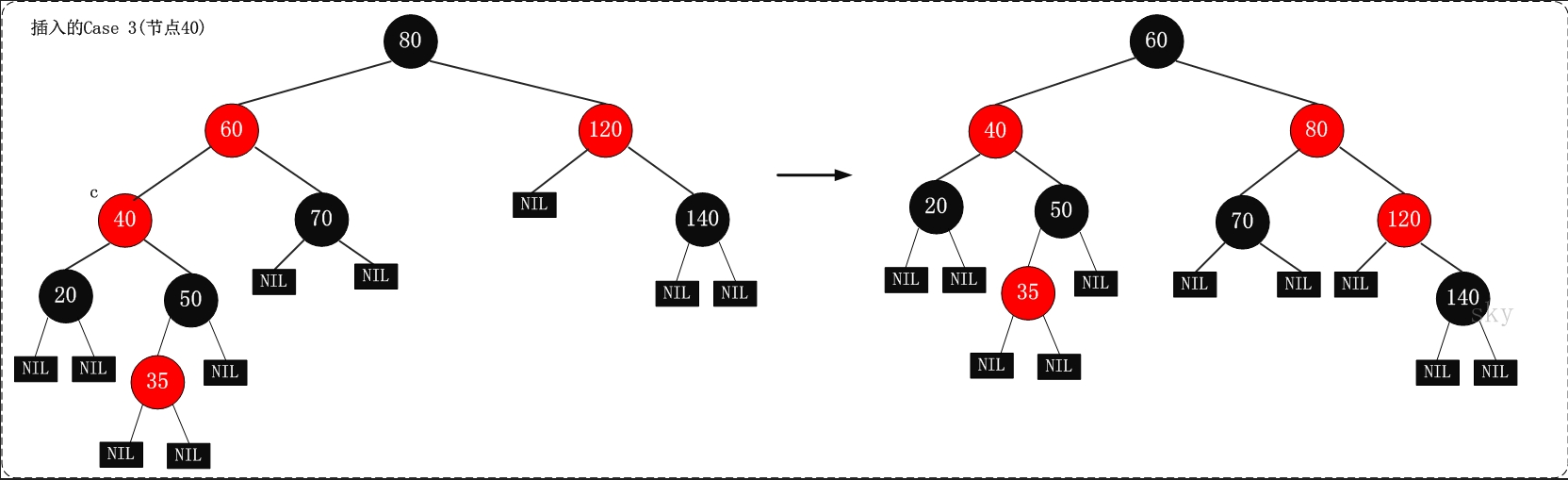
(02) 将“祖父节点”设为“红色”。

(03) 以“祖父节点”为支点进行右旋

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)

为了便于说明，我们设置“当前节点”为S(Original Son)，“兄弟节点”为B(Brother)，“叔叔节点”为U(Uncle)，“父节点”为F(Father)，祖父节点为G(Grand-Father)。

   S和F都是红色，违背了红黑树的“特性(4)”，我们可以将F由“红色”变为“黑色”，就解决了“违背‘特性(4)’”的问题；但却引起了其它问题：违背特性(5)，因为将F由红色改为黑色之后，所有经过F的分支的黑色节点的个数增加了1。那我们如何解决“所有经过F的分支的黑色节点的个数增加了1”的问题呢？ 我们可以通过“将G由黑色变成红色”，同时“以G为支点进行右旋”来解决。



* + - * 1.  删除节点

将红黑树内的某一个节点删除。需要执行的操作依次是：

首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；

然后，通过"**旋转和重新着色**"等一系列来修正该树，使之重新成为一棵红黑树；

将红黑树当作一颗二叉查找树，将节点删除

See also: [删除操作](#删除操作)

这和"删除常规二叉查找树中删除节点的方法是一样的"。

分3种情况：

    ① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就OK了。

    ② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

    ③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。

    在这里，后继节点相当于替身，在将后继节点的内容复制给"被删除节点"之后，再将后继节点删除。这样就巧妙的将问题转换为"删除后继节点"的情况了，下面就考虑后继节点。 在"被删除节点"有两个非空子节点的情况下，它的后继节点不可能是双子非空。既然"的后继节点"不可能双子都非空，就意味着"该节点的后继节点"要么没有儿子，要么只有一个儿子。若没有儿子，则按"情况① "进行处理；若只有一个儿子，则按"情况② "进行处理

**删除节点的伪代码实现：**

RB-DELETE(T, z)   //从红黑树（二叉查找树）T中删除节点z

if left[z] = nil[T] or right[z] = nil[T]

then y ← z                        // 若“z的左孩子” 或 “z的右孩子”为空，则将“z”赋值给 “y”；

else y ← TREE-SUCCESSOR(z)        // 否则（被删除的节点有两个孩子节点），将“z的后继节点”赋值给 “y”。

if left[y] ≠ nil[T]

then x ← left[y]                  // 若“y的左孩子” 不为空，则将“y的左孩子” 赋值给 “x”；

else x ← right[y]                 // 否则，“y的右孩子” 赋值给 “x”。

p[x] ← p[y]                          // 将“y的父节点” 设置为 “x的父节点”

if p[y] = nil[T]

then root[T] ← x                  // 情况1：若“y的父节点” 为空，则设置“x” 为 “根节点”。

else if y = left[p[y]]

         then left[p[y]] ← x       // 情况2：若“y是它父节点的左孩子”，则设置“x” 为 “y的父节点的左孩子”

         else right[p[y]] ← x      // 情况3：若“y是它父节点的右孩子”，则设置“x” 为 “y的父节点的右孩子”

if y ≠ z

then key[z] ← key[y]              // 若“y的值” 赋值给 “z”。注意：这里只拷贝z的值给y，而没有拷贝z的颜色！！！

      copy y's satellite data into z

if color[y] = BLACK

then RB-DELETE-FIXUP(T, x)         // 若“y为黑节点”，则调用

return y

通过"旋转和重新着色"等一系列来修正该树 使之重新成为一棵红黑树

"第一步"中删除节点之后，可能会违背红黑树的特性。所以需要通过"旋转和重新着色"来修正该树，使之重新成为一棵红黑树

前面我们将"删除红黑树中的节点"大致分为两步，在第一步中"将红黑树当作一颗二叉查找树，将节点删除"后，可能违反"特性(2)、(4)、(5)"三个特性。第二步需要解决上面的三个问题，进而保持红黑树的全部特性。

为了便于分析，我们假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。为什么呢？

通过RB-DELETE算法，我们知道：**删除节点y之后，x占据了原来节点y的位置**。 既然删除y(**y是黑色,如果Y不是黑色不需要进行这一步**)，意味着减少一个黑色节点；那么，再在该位置上增加一个黑色即可。这样，当我们假设"x包含一个额外的黑色"，就正好弥补了"删除y所丢失的黑色节点"，也就不会违反"特性(5)"。 因此，假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。

现在，x不仅包含它原本的颜色属性，x还包含一个额外的黑色。即x的颜色属性是"红+黑"或"黑+黑"，它违反了"特性(1)"。

现在，我们面临的问题，由解决"违反了特性(2)、(4)、(5)三个特性"转换成了"解决违反特性(1)、(2)、(4)三个特性"。

RB-DELETE-FIXUP需要做的就是通过算法恢复红黑树的特性(1)、(2)、(4)。RB-DELETE-FIXUP的思想是：**将x所包含的额外的黑色不断沿树上移(向根方向移动)，直到出现下面的姿态：**

a) x指向一个"红+黑"节点。此时，将x设为一个"黑"节点即可。

b) x指向根。此时，将x设为一个"黑"节点即可。

c) 非前面两种姿态。

② 情况说明：x是“黑+黑”节点，且x是根。

处理方法：什么都不做，结束。此时红黑树性质全部恢复。

③ 情况说明：x是“黑+黑”节点，且x不是根。

处理方法：这种情况又可以划分为4种子情况。

伪代码实现

伪代码实现：

RB-DELETE-FIXUP(T, x)

while x ≠ root[T] and color[x] = BLACK

   do if x = left[p[x]]

         then w ← right[p[x]]                                             // 若 “x”是“它父节点的左孩子”，则设置 “w”为“x的叔叔”(即x为它父节点的右孩子)

              if color[w] = RED                                           // Case 1: x是“黑+黑”节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。

                 then color[w] ← BLACK                        ▹ Case 1   //   (01) 将x的兄弟节点设为“黑色”。

                      color[p[x]] ← RED                       ▹ Case 1   //   (02) 将x的父节点设为“红色”。

                      LEFT-ROTATE(T, p[x])                    ▹ Case 1   //   (03) 对x的父节点进行左旋。

                      w ← right[p[x]]                         ▹ Case 1   //   (04) 左旋后，重新设置x的兄弟节点。

              if color[left[w]] = BLACK and color[right[w]] = BLACK       // Case 2: x是“黑+黑”节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。

                 then color[w] ← RED                          ▹ Case 2   //   (01) 将x的兄弟节点设为“红色”。

                      x ← p[x]                               ▹ Case 2   //   (02) 设置“x的父节点”为“新的x节点”。

                 else if color[right[w]] = BLACK                          // Case 3: x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。

                         then color[left[w]] ← BLACK          ▹ Case 3   //   (01) 将x兄弟节点的左孩子设为“黑色”。

                              color[w] ← RED                  ▹ Case 3   //   (02) 将x兄弟节点设为“红色”。

                              RIGHT-ROTATE(T, w)              ▹ Case 3   //   (03) 对x的兄弟节点进行右旋。

                              w ← right[p[x]]                 ▹ Case 3   //   (04) 右旋后，重新设置x的兄弟节点。

                       color[w] ← color[p[x]]                 ▹ Case 4   // Case 4: x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的。(01) 将x父节点颜色 赋值给 x的兄弟节点。

                       color[p[x]] ← BLACK                    ▹ Case 4   //   (02) 将x父节点设为“黑色”。

                       color[right[w]] ← BLACK                ▹ Case 4   //   (03) 将x兄弟节点的右子节设为“黑色”。

                       LEFT-ROTATE(T, p[x])                   ▹ Case 4   //   (04) 对x的父节点进行左旋。

                       x ← root[T]                            ▹ Case 4   //   (05) 设置“x”为“根节点”。

      else (same as then clause with "right" and "left" exchanged)        // 若 “x”是“它父节点的右孩子”，将上面的操作中“right”和“left”交换位置，然后依次执行。

color[x] ← BLACK

x是“红+黑”节点 直接把x设为黑色，结束。此时红黑树性质全部恢复

x是“黑+黑”节点，且x是根 什么都不做，结束。此时红黑树性质全部恢复

x是“黑+黑”节点，且x不是根

x的兄弟节点是红色 此时x的父节点和x的兄弟节点的子节点都是黑节点

(01) 将x的兄弟节点设为“黑色”。

(02) 将x的父节点设为“红色”。

(03) 对x的父节点进行左旋。

(04) 左旋后，重新设置x的兄弟节点

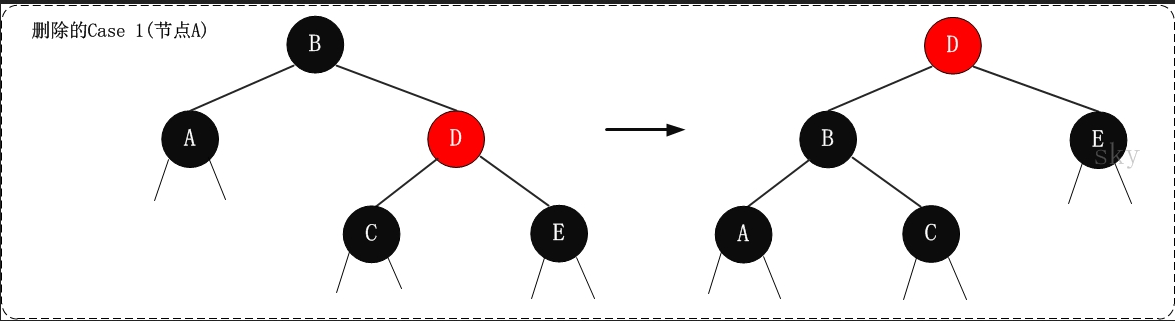
**这样做的目的是:**

将“Case 1”转换为“Case 2”、“Case 3”或“Case 4”，从而进行进一步的处理。

对x的父节点进行左旋；

 左旋后，为了保持红黑树特性，就需要在左旋前“将x的兄弟节点设为黑色”，同时“将x的父节点设为红色”；

左旋后，由于x的兄弟节点发生了变化，需要更新x的兄弟节点，从而进行后续处理。



x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色

* + 1. B树
    2. B+树
    3. B\*树
    4. Trie-Tree
       1. 原理
       2. 特点
       3. 应用

字符串检索

词频统计

* 1.  图