# 定义

树是节点的有限集合。

相关概念：

孩子、双亲、度、叶子（终端节点）、根（非终端节点）、有序树、无序树、祖先、子孙

深度、节点深度、树深度

# 特点

每个节点有若干个或0个子节点；

根节点没有父节点；

每一个非根节点有且只有一个父节点；

每个子节点可以分为多个不相交的子树；

# 分类

## 二叉树

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2i-1个结点；深度为k的二叉树至多有2k-1个结点；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。



遍历：前序遍历、中序遍历、后序遍历

### 完全二叉树

满二叉树：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。节点数达到最大值，所有叶子结点必须在同一层上。

　　满二叉树的性质：

　　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2h;

　　3) 第k层的结点数是：2k-1;

　　4) 总结点数是：2k-1，且总节点数一定是奇数。



### 满二叉树

完全二叉树：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。

**注：**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。

## 二叉查找树/BST

二叉查找树又称为是二叉排序树（Binary Sort Tree，简写BST）或二叉搜索树。

树的每个节点都包含了健值 key、数据值 data、左子节点指针、右子节点指针。其中健值 key 是最核心的部分，它的值决定了树的组织形状；数据值 data 是该节点对应的数据，有些场景可以忽略。

### 特点

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

　　1) 若左子树不空，则左子树上所有结点的key值均小于它的根结点的key值；

　　2) 若右子树不空，则右子树上所有结点的key值均大于或等于它的根结点key的值；

　　3) 左、右子树也分别为二叉排序树；

4) 没有键值相等的节点。

### 性质

二叉查找树的性质：对二叉查找树进行中序遍历，即可得到有序的数列。

### 插入

　　二叉查找树的高度决定了二叉查找树的查找效率。

　　二叉查找树的插入过程如下：

　　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。

### 查询

### 删除

　　二叉查找树的删除，分三种情况进行处理：

　　1) p为叶子节点，直接删除该节点，再修改其父节点的指针（注意分是根节点和不是根节点），如图a;

　　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），如图b;

　　3) p的左子树和右子树均不空。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；或者方法二是找到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y的左子树的父亲节点。如图c。







### 复杂度

二叉查找树的时间复杂度：它和二分查找一样，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（比如，我们查找上图（b）中的“93”，我们需要进行n次查找操作）。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。

## 平衡二叉树

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为log2n，其各操作的时间复杂度O(log2n)同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)。我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。于是就有了我们下边介绍的平衡二叉树。

　　平衡二叉树（Balanced Binary Tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在O(log2n)，大大降低了操作的时间复杂度。

　　最小二叉平衡树的节点的公式如下：

　　F(n)=F(n-1)+F(n-2)+1

　　这个类似于一个递归的数列，可以参考Fibonacci数列，1是根节点，F(n-1)是左子树的节点数量，F(n-2)是右子树的节点数量。

### AVL树

AVL树是最先发明的自平衡二叉查找树。AVL树得名于它的发明者 G.M. Adelson-Velsky 和 E.M. Landis，他们在 1962 年的论文 "An algorithm for the organization of information" 中发表了它。

在AVL中任何节点的两个儿子子树的高度最大差别为1，所以它也被称为高度平衡树，n个结点的AVL树最大深度约1.44log2n。查找、插入和删除在平均和最坏情况下都是O(logn)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。**这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。**

**AVL树的自平衡操作——旋转：**

　　AVL树最关键的也是最难的一步操作就是**旋转**。旋转主要是为了实现AVL树在实施了插入和删除操作以后，树重新回到平衡的方法。下面我们重点研究一下AVL树的旋转。

　　对于一个平衡的节点，由于任意节点最多有两个儿子，因此高度不平衡时，此节点的两颗子树的高度差2.容易看出，这种不平衡出现在下面四种情况：



　　1) 6节点的左子树3节点高度比右子树7节点大2，左子树3节点的左子树1节点高度大于右子树4节点，这种情况成为左左。

　　2) 6节点的左子树2节点高度比右子树7节点大2，左子树2节点的左子树1节点高度小于右子树4节点，这种情况成为左右。

　　3) 2节点的左子树1节点高度比右子树5节点小2，右子树5节点的左子树3节点高度大于右子树6节点，这种情况成为右左。

　　4) 2节点的左子树1节点高度比右子树4节点小2，右子树4节点的左子树3节点高度小于右子树6节点，这种情况成为右右。

　　从图2中可以可以看出，1和4两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，我们称之为单旋转。2和3两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，我们称之为双旋转。

**单旋转**

　　单旋转是针对于左左和右右这两种情况的解决方案，这两种情况是对称的，只要解决了左左这种情况，右右就很好办了。图3是左左情况的解决方案，节点k2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的左子树X子树，所以属于左左情况。



为使树恢复平衡，我们把k2变成这棵树的根节点，因为k2大于k1，把k2置于k1的右子树上，而原本在k1右子树的Y大于k1，小于k2，就把Y置于k2的左子树上，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

　　这样的操作只需要一部分指针改变，结果我们得到另外一颗二叉查找树，它是一棵AVL树，因为X向上一移动了一层，Y还停留在原来的层面上，Z向下移动了一层。整棵树的新高度和之前没有在左子树上插入的高度相同，插入操作使得X高度长高了。因此，由于这颗子树高度没有变化，所以通往根节点的路径就不需要继续旋转了。

**双旋转**

对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。双旋转是针对于这两种情况的解决方案，同样的，这样两种情况也是对称的，只要解决了左右这种情况，右左就很好办了。图4是左右情况的解决方案，节点k3不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的右子树k2子树，所以属于左右情况。



  为使树恢复平衡，我们需要进行两步，第一步，把k1作为根，进行一次右右旋转，旋转之后就变成了左左情况，所以第二步再进行一次左左旋转，最后得到了一棵以k2为根的平衡二叉树。

### 红黑树

红黑树的定义：红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在1972年由鲁道夫·贝尔发明的，称之为"对称二叉B树"，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于1978年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(logn)时间内做查找，插入和删除，这里的n是树中元素的数目。

　　红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用如实时应用（real time application）中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为建造板块的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树。此外，红黑树还是2-3-4树的一种等同，它们的思想是一样的，只不过红黑树是2-3-4树用二叉树的形式表示的。

　　红黑树的性质：

　　红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制的一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

　　性质1. 节点是红色或黑色。

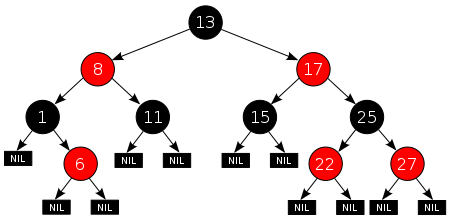
　　性质2. 根是黑色。

　　性质3. 所有叶子都是黑色（叶子是NIL节点）。

　　性质4. 每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)

　　性质5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

　　下面是一个具体的红黑树的图例：



　　这些约束确保了红黑树的关键特性: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

　　要知道为什么这些性质确保了这个结果，注意到性质4导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

　　以下内容整理自[wiki百科之红黑树](https://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91)。

　　红黑树的自平衡操作：

　　因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。恢复红黑树的性质需要少量(O(logn))的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)。虽然插入和删除很复杂，但操作时间仍可以保持为O(logn) 次。

　　我们首先以二叉查找树的方法增加节点并标记它为红色。如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的（违背性质5）。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意:

* 性质1和性质3总是保持着。
* 性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。
* 性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

　　插入操作：

　　假设，将要插入的节点标为N，N的父节点标为P，N的祖父节点标为G，N的叔父节点标为U。在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含的。

　　情形1: 该树为空树，直接插入根结点的位置，违反性质1，把节点颜色有红改为黑即可。

　　情形2: 插入节点N的父节点P为黑色，不违反任何性质，无需做任何修改。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

　　注： 情形1很简单，情形2中P为黑色，一切安然无事，但P为红就不一样了，下边是P为红的各种情况，也是真正难懂的地方。

　　情形3: 如果父节点P和叔父节点U二者都是红色，(此时新插入节点N做为P的左子节点或右子节点都属于情形3,这里右图仅显示N做为P左子的情形)则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色(用来保持性质4)。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G的父节点也有可能是红色的，这就违反了性质4。为了解决这个问题，我们在祖父节点G上递归地进行上述情形的整个过程（把G当成是新加入的节点进行各种情形的检查）。比如，G为根节点，那我们就直接将G变为黑色（情形1）；如果G不是根节点，而它的父节点为黑色，那符合所有的性质，直接插入即可（情形2）；如果G不是根节点，而它的父节点为红色，则递归上述过程（情形3）。



情形4: 父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转; 在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色(如果P和G都是红色就违反了性质4，所以G必须是黑色)。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。



情形5: 父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色; 接着，我们按情形4处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。



　　注: 插入实际上是原地算法，因为上述所有调用都使用了尾部递归。

　　删除操作：

　　如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们找到要么在它的左子树中的最大元素、要么在它的右子树中的最小元素，并把它的值转移到要删除的节点中。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值，不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

　　我们只需要讨论删除只有一个儿子的节点(如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子)。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质3和性质4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质5。

　　需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为N(在新的位置上)，称呼它的兄弟(它父亲的另一个儿子)为S。在下面的示意图中，我们还是使用P称呼N的父亲，SL称呼S的左儿子，SR称呼S的右儿子。

　　如果N和它初始的父亲是黑色，则删除它的父亲导致通过N的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质5，树需要被重新平衡。有几种情形需要考虑:

　　情形1: N是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

　　注意: 在情形2、5和6下，我们假定N是它父亲的左儿子。如果它是右儿子，则在这些情形下的左和右应当对调。

　　情形2: S是红色。在这种情形下我们在N的父亲上做左旋转，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按情形4、情形5或情形6来处理。



　情形3: N的父亲、S和S的儿子都是黑色的。在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前不通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从情形1开始，在P上做重新平衡处理。



　　情形4: S和S的儿子都是黑色，但是N的父亲是红色。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。



　　情形5: S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。在这种情形下我们在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形6。N和它的父亲都不受这个变换的影响。



　情形6: S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先: 要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。

　　此时，如果一个路径不通过N，则有两种可能性:

* 它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。
* 它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

　　在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。



## 线索二叉树

## B-树/2-3树

2-3树，是最简单的B-树，其中2、3主要体现在每个非叶子节点都有2个或3个子节点，B-树即是平衡树，平衡树是为了解决不平衡树查询效率问题，常见的二叉平衡书有AVL树，它虽然提高了查询效率，但是插入操作效率不高，因为它需要再每次插入节点后维护树的平衡，而为了解决查询效率同时有兼顾插入效率，于是提出了2-3树。

### 特点

2-3树是一棵平衡树，但不是二叉平衡树。

对于高度相同的2-3树和二叉树，2-3树的节点数要大于满二叉树，因为有些节点可能有三个子节点。

2-3树可以是一棵空树。

对于2节点来说，该节点保存了一个key及对应的value，除此之外还保存了指向左右两边的子节点，子节点也是一个2-3节点，左子节点所有值小于key，右子节点所有值大于key。

对于3节点来说，该节点保存了两个key及对应的value，除此之外还保存了指向左中右三个方向的子节点，子节点也是一个2-3节点，左子节点的所有值小于两个key中较小的那个，中节点的所有值在两个key值之间，右子节点大于两个key中较大的那个。

对2-3树进行中序遍历能得到一个排好序的序列。

### 插入

### 查找

## B树

B树也是一种用于查找的平衡树，但是它不是二叉树。

B树即平衡查找树，一般理解为平衡多路查找树，也称为B-树、B\_树。是一种自平衡树状数据结构，能对存储的数据进行O(log n)的时间复杂度进行查找、插入和删除。B树一般较多用在存储系统上，比如数据库或文件系统。

B树的定义：B树（B-tree）是一种树状数据结构，能够用来存储排序后的数据。这种数据结构能够让查找数据、循序存取、插入数据及删除的动作，都在对数时间内完成。B树，概括来说是一个一般化的二叉查找树，可以拥有多于2个子节点。与自平衡二叉查找树不同，B-树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结构常被应用在数据库和文件系统的实作上。

### 特点

　　B树作为一种多路搜索树（并不是二叉的）：

　　1) 定义任意非叶子结点最多只有M个儿子；且M>2；

　　2) 根结点的儿子数为[2, M]；

　　3) 除根结点以外的非叶子结点的儿子数为[M/2, M]；

　　4) 每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

　　5) 非叶子结点的关键字个数=指向儿子的指针个数-1；

　　6) 非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

　　7) 非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

　　8) 所有叶子结点位于同一层；

### 插入

### 查找

　　在B树中查找给定关键字的方法是，首先把根结点取来，在根结点所包含的关键字K1,…,Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取指针Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。

## B+树

B+树是B树的一种变体，也属于平衡多路查找树，大体结构与B树相同，包含根节点、内部节点和叶子节点。多用于数据库和操作系统的文件系统中，由于B+树内部节点不保存数据，所以能在内存中存放更多索引，增加缓存命中率。另外因为叶子节点相连遍历操作很方便，而且数据也具有顺序性，便于区间查找。

### 特点

B+树是B树的变体，也是一种多路搜索树：

　　1) 其定义基本与B-树相同，除了：

　　2) 非叶子结点的子树指针与关键字个数相同；

　　3) 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

　　4) 为所有叶子结点增加一个链指针；

　　5) 所有关键字都在叶子结点出现；

B+树的搜索与B树也基本相同，区别是B+树只有达到叶子结点才命中（B树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

　　B+的性质：

　　1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

　　2.不可能在非叶子结点命中；

　　3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.更适合文件索引系统。

### 插入

### 查询

## B\*树

B\*树是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针，将结点的最低利用率从1/2提高到2/3。

　　B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）；

　　B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

　　B\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

　　所以，B\*树分配新结点的概率比B+树要低，空间使用率更高。

## Trie树

Trie树称为字典树，又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

　　Tire树的三个基本性质：

　　1) 根节点不包含字符，除根节点外每一个节点都只包含一个字符；

　　2) 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；

　　3) 每个节点的所有子节点包含的字符都不相同。

　　Tire树的应用：

　　1) 串的快速检索

　　给出N个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

在这道题中，我们可以用数组枚举，用哈希，用字典树，先把熟词建一棵树，然后读入文章进行比较，这种方法效率是比较高的。

　　2) “串”排序

　　给定N个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出。用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可。

　　3) 最长公共前缀

对所有串建立字典树，对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数，于是，问题就转化为求公共祖先的问题。

## Radix树

Radix树，即基数树，也称压缩前缀树，是一种提供key-value存储查找的数据结构。与Trie不同的是，它对Trie树进行了空间优化，只有一个子节点的中间节点将被压缩。同样的，Radix树的插入、查询、删除操作的时间复杂度都为O(k)。

### 特点

一般由根节点、中间节点和叶子节点组成。

每个节点可以包含一个或多个字符。

树的叶子结点数即是数据条目数。

从根节点到某一节点经过路径的字符连起来即为该节点对应的字符串。

每个节点的所有子节点字符串都不相同。

### 插入

### 查询

# 存储

## 双亲表示法

## 孩子表示法

## 孩子兄弟表示法

# 代码

## 数组实现

## 链表实现

# 遍历

#include <iostream>

#include <stack>

#include <vector>

#include <deque>

using namespace std;

/\*

二叉搜索树的创建

二叉树的递归遍历

二叉树的非递归遍历

\*/

typedef struct Bin\_Tree BinTree;

struct Bin\_Tree

{

int value;

BinTree\* right;

BinTree\* left;

};

/\*

构造二叉搜索树

\*/

BinTree\* InsertNode(BinTree\* root,int value)

{

BinTree\* newnode = new BinTree;

newnode->value = value;

newnode->right = NULL;

newnode->left = NULL;

//创建完一个新的节点，需要将此节点插入到合适的位置

if(root == NULL)

{

root = newnode;

}

else

{

//找合适的位置

BinTree\* parent = root;

while(parent != NULL)

{

if(parent->value < value)

{

if(parent->right == NULL)

break;

else

parent = parent->right;

}

else

{

if(parent->left == NULL)

break;

else

parent = parent->left;

}

}

if(parent->value < value)

parent->right = newnode;

else

parent->left = newnode;

}

return root;

}

/\*

递归的方法前序遍历二叉树

\*/

void Preorder(BinTree\* root)

{

if(root == NULL)

return ;

cout<<root->value<<endl;

Preorder(root->left);

Preorder(root->right);

}

/\*

递归的方法中序遍历二叉树

\*/

void Inorder(BinTree\* root)

{

if(root == NULL)

return ;

Inorder(root->left);

cout<<root->value<<endl;

Inorder(root->right);

}

/\*

递归的方法后序遍历二叉树

\*/

void Postorder(BinTree\* root)

{

if(root == NULL)

return ;

Postorder(root->left);

Postorder(root->right);

cout<<root->value<<endl;

}

/\*非递归的遍历二叉树\*/

/\*

非递归的前序遍历

\*/

void Preorder\_Nonrecursive(BinTree\* root)

{

if(root == NULL)

return ;

stack<BinTree\*> s;

s.push(root);

while(!s.empty())

{

BinTree\* temp = s.top();

cout<<temp->value<<" ";

s.pop();

if(temp->right)

s.push(temp->right);

if(temp->left)

s.push(temp->left);

}

}

/\*第二种方法\*/

void Preorder\_Nonrecursive1(BinTree\* root)

{

if(root == NULL)

return ;

stack<BinTree\*> s;

BinTree\* cur = root;

while(cur != NULL || !s.empty())

{

while(cur != NULL)

{

cout<<cur->value<<" ";

s.push(cur);

cur = cur->left;

}

if(!s.empty())

{

cur = s.top();

s.pop();

cur = cur->right;

}

}

}

/\*非递归中序遍历\*/

void Inorder\_Nonre(BinTree\* root)

{

if(root == NULL)

return ;

BinTree\* cur = root;

stack<BinTree\*> s;

while(cur != NULL || !s.empty())

{

while(cur != NULL)

{

s.push(cur);

cur = cur->left;

}

if(!s.empty())

{

cur = s.top();

s.pop();

cout<<cur->value<<" ";

cur = cur->right;

}

}

}

/\*

第二种方法

\*/

void Inorder\_Nonre2(BinTree\* root)

{

if(root == NULL)

return ;

stack<BinTree\*> st;

BinTree\* p = root;

while(p!= NULL || !st.empty())

{

if(p != NULL)

{

st.push(p);

p = p->left;

}

else

{

p=st.top();

st.pop();

cout<<p->value<<endl;

p = p->right;

}

}

}

/\*

非递归的后序遍历

\*/

void Postorder\_Nonre(BinTree\* root)

{

stack<BinTree\*> s;

BinTree\* cur = root;

BinTree\* pre = NULL;//指向前一个被访问的节点

while(cur != NULL || !s.empty())

{

while( cur != NULL) //一直向左走直到为空

{

s.push(cur);

cur = cur->left;

}

cur = s.top();

//当前节点的右孩子如果为空或者已经被访问，则访问当前节点

if(cur->right == NULL || cur->right == pre)

{

cout<<cur->value<<" ";

pre = cur;

s.pop();

cur = NULL;

}

else

cur = cur->right;//否则访问右孩子

}

}

/\*

第二种方法

\*/

void Postorder\_Nonre2(BinTree\* root)

{

//使用双栈法

stack<BinTree\*> s1,s2;

BinTree\* cur;//指向当前要检查的节点

s1.push(root);

//栈为空时结束

while(!s1.empty())

{

cur = s1.top();

s1.pop();

s2.push(cur);

if(cur->left)

s1.push(cur->left);

if(cur->right)

s1.push(cur->right);

}

while(!s2.empty())

{

cout<<s2.top()->value<<" ";

s2.pop();

}

}

int main()

{

BinTree\* root =NULL;

int array[]={10,6,14,4,8,12,16};

for(int i=0;i<sizeof(array)/sizeof(int);i++)

root = InsertNode(root,array[i]);

Inorder\_Nonre2(root);

return 0;

}

# 应用

压缩软件——赫夫曼树

搜索——人机对战（不断做树的搜索，找到最优的路径）