
# 0.写在前面

在之前的内容中，我们已经学习了基于二叉树结构的多种树形结构的应用

这些应用绝大部分都使用来保证数据的有序性的，并且在此特性基础之上，还要保证树结构的平衡性

进而达到有效提升元素查找效率的目的

但是这些树形结构都是基于二叉树结构的，但是实际上，除了二叉树结构之外，在树形结构中还有很多其他品种的“树”

这些树结构，并不局限于只有两个子节点的二叉树形式，而是根据不同的应用场景，不断变换自身的结构，从而达到形成不同作用的目的

接下来，我们就来研究几种二叉树之外的树结构，并具体讨论他们的用途和使用场景

# 1.B树与B+树

**①回忆杀：2-3-4树结构**

在之前我们学习红黑树结构的时候，为了方便大家理解红黑树的结构特征，我们特意引入了2-3-4树结构

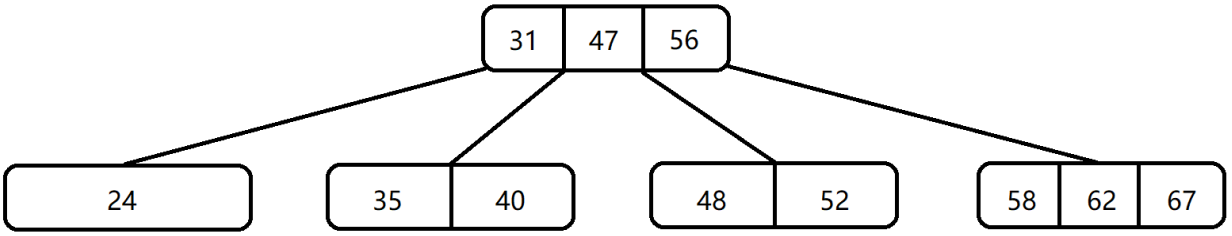
在2-3-4树结构中，每一个节点中最多能够存储3个数据，并且这3个按照数据取值的比较结果，按照从大到小（或者从小到大）

的顺序进行排列，在每一个节点的3个数据的间隔之中，我们可以引出其子节点，那么一个2-3-4树节点，就能够同时最多具有2个、3个或者4个叶子节点

每一个叶子节点中数据的取值，又是介于其父节点中，形成间隔的两个数据的取值之间

同时，2-3-4树是一种自底向上构建的树结构，这也保证了每一个节点中，不会出现空的数据间隔，进而形成了一种完美平衡的树结构

下面就是一张2-3-4树结构的示意图：




实际上，2-3-4树结构就是一种特殊的B树结构

下面就让我们一起来研究一下，B树的结构特征

**②B树结构的构建**

按照官方定义，B树是一种多路平衡查找树，也就是说，它的一个节点之下，可以挂载多个子节点，并且存在于B树中的任意一个节点，都是平衡的

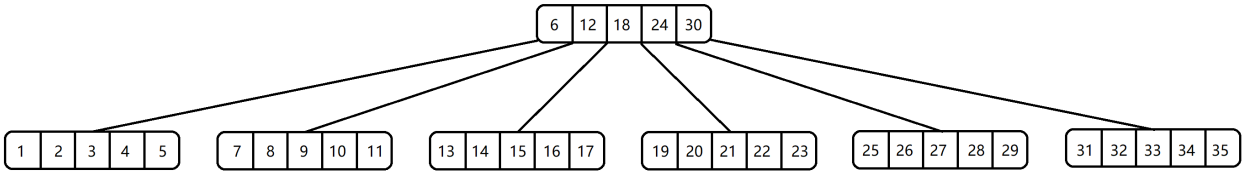
且存在于子节点中的数据取值，一定介于其父节点中对应的数据间隔之间

在B树中，有一个非常重要的概念：阶数。

阶数，描述的是一个B树节点，能够具有的子节点的最大值；例如：一个B树结构的阶数为6，就表示这个B树结构的节点，最多能够同时具有6个子节点

反过来说，就表示这个B树结构中的每一个节点，同时最多能够存储5个数据

下图演示的，就是一个阶数为6的B树结构：




从上图中我们不难发现如下规律：

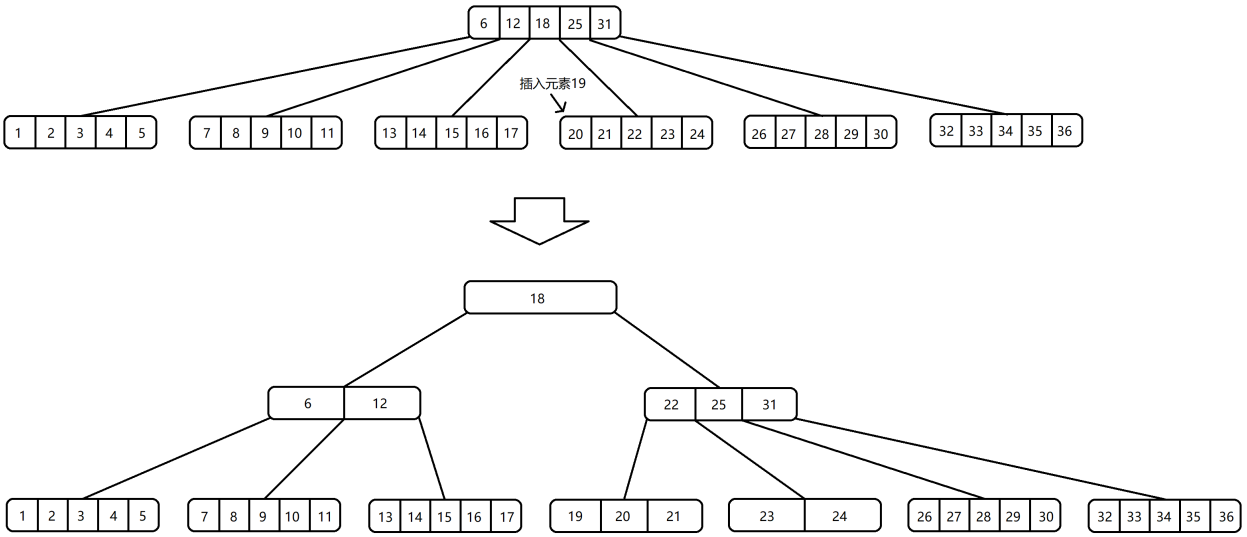
规律1：如果我们将一个B树结构的阶数定义为4，那么这个B树结构就变成了2-3-4树

规律2：如果我们将一个B树结构的结束定义为2，那么这个B树结构就变成了二叉排序树（BST）

对于B树插入和删除元素的过程，也就是节点分裂/提升和下降的过程，我们已经在上一章节的红黑树结构中，以2-3-4树结构为例进行了详细的讨论，所以在此就不再赘述

但是如果我们重新审视整个B树添加元素，从而导致节点分裂和提升的过程，我们会发现一些适用于B树结构的特殊规律

首先让我们重新来看一下，一个阶数为6的B树，在添加元素过程中产生的节点分裂和元素提升的过程：




之前我们曾经说过：在想2-3-4树结构中插入元素的时候，只能够向叶子节点插入元素，不能够向中间节点或者根节点插入元素，此时这个约束也同样适用于B树结构

这也就导致了，在B树结构中，所有的中间节点和整个B树的根节点，都是因为子节点的分裂与元素的提升而被“挤”出来的

那么这就导致了B树节点的如下特征：

特征1：B树的根节点中最少可以仅存在1个元素

特征2：在阶数为n的B树中，非根节点（包括叶子节点）中的元素个数，最少为(n / 2)向上取整 - 1个

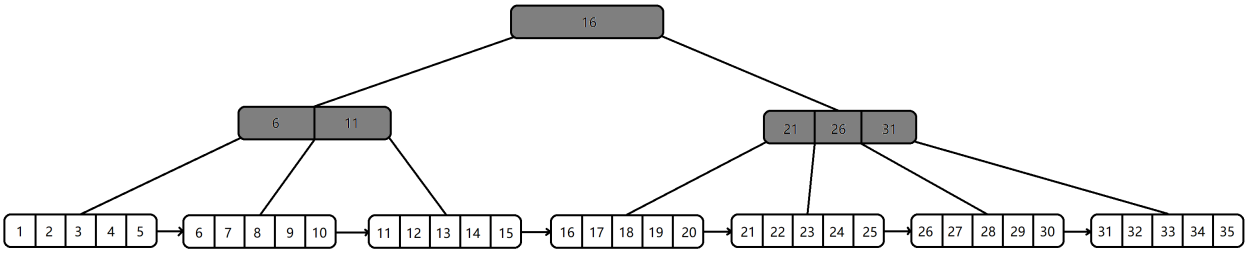
例如，在上面这个阶数为6的B树结构中，其每一个非根节点中，元素的数量都不会少于(6 / 2) - 1个

注意：在实际应用当中，为了更方便的保存数据，B树的阶数取值可能非常大，例如取值为100等，这样做是为了尽可能的发挥B树的检索效率

而不至于因为元素增删导致频繁的结构调整，从而频繁的调整B树的结构，从而拖慢B树结构整体的元素存取效率

**③B+树结构的构建**

首先我们来看一个B+树的结构图：




在看完上面的B+树的结构图之后，很多同学会说：这不就是一个B树结构吗？但是……貌似和B树结构又有一些不太一样的地方……

没错B+树本身就是B树的升级版，并且和B树又有一些不相同的地方：

不相同1：在B树中，不管是中间节点还是叶子节点中，是不存在重复数据的；但是在B+树结构中，非叶子节点中的数据都是在叶子结点中出现过的“重复数据”

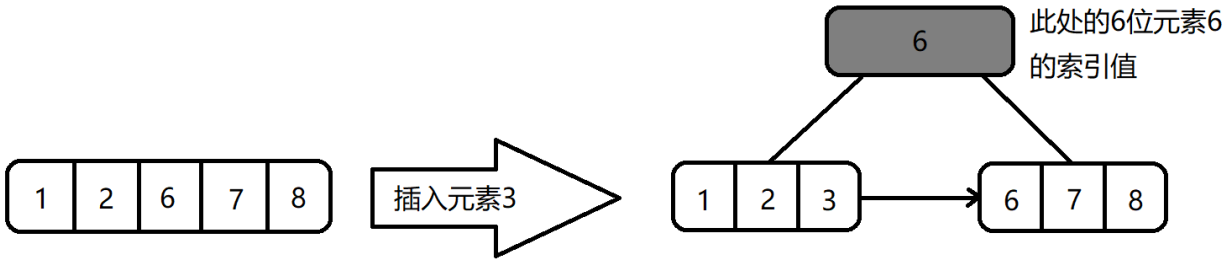
不相同2：在B树中，所有的叶子节点虽然都在同一层当中，但是互相之间没有相互关联关系；在B+树中，所有的叶子节点之间使用单链表结构进行串联，每一个叶子节点都是一个单链表的节点

导致上述两个特性的原因，就隐藏在B+树结构的构建过程当中。

B+树结构的构建过程，与B树结构的构建过程大致相似，但是在叶子结点分裂的时候，我们并不是将叶子结点的中间元素真正的向上提升

而是将叶子结点中间元素的索引（可以理解为中间元素的“影子”）向上提升，而将中间元素的“真身”保留在分裂之后的叶子结点之中，并在叶子结点之间构建链表关系

下图演示的就是B+树结构中，叶子结点的分裂过程：




那么，索引到底指的是什么？索引和元素本身之间的关系又是怎么样的？B+树结构使用中间节点保存索引，并且为叶子节点之间构建单链表又到底有什么目的？

如果仅仅从上面两幅图出发去解释B树和B+树之间的区别，是解释不清楚的，所以我们必须从B树和B+树在实际应用中的案例入手来进行比较和说明

**④B树与B+树的使用场景：MySQL数据库的索引**

学习过MySQL数据库的同学应该对MySQL数据库中的索引（Index）结构并不陌生：

索引结构是在表结构的基础之上，按照表结构中的某一字段，对数据进行快速分类的一种结构，并且索引本身并不占用额外的硬盘存储空间存储数据

我们可以将索引的概念想象成为字典的拼音查询目录，通过索引结构，能够加快数据的查询速度，使得在查询结构复杂的数据表数据的时候，提升查询效率

实际上，MySQL数据库的索引结构，其底层实现原理就是B+树结构。

那么有的同学可能会问：为什是B+树结构而不是B树结构呢？下面我们就用一组实际案例对这一问题进行说明，从而进一步比较B树结构和B+树结构在实际应用中的区别

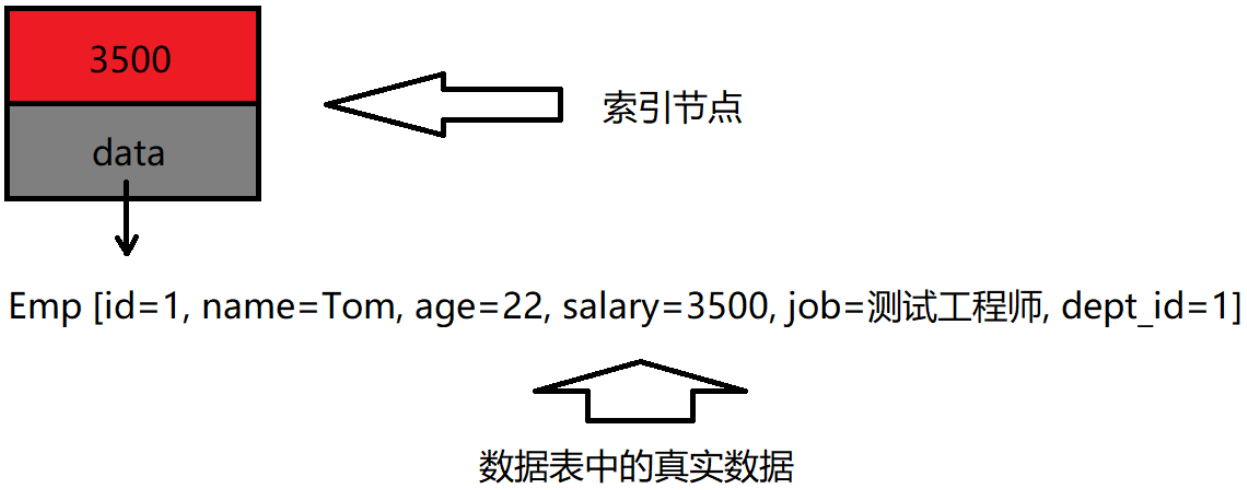
假设我们有如下的一组数据：

Emp表：

| ID | 姓名 | 年龄 | 薪资 | 工种 | 部门编号 |
|---|---|---|---|---|---|
| 1 | Tom | 22 | 3500 | 测试工程师 | 1 |
| 2 | Jerry | 21 | 3400 | 测试工程师 | 1 |
| 3 | Jack | 25 | 3600 | 运维工程师 | 1 |
| 4 | Rose | 24 | 4700 | 产品销售 | 2 |
| 5 | Bruce | 22 | 8500 | 销售经理 | 2 |
| 6 | Ben | 35 | 5600 | 产品销售 | 2 |
| 7 | Frank | 26 | 7500 | 开发工程师 | 1 |
| 8 | Abby | 37 | 5700 | 产品销售 | 2 |
| 9 | Aaron | 29 | 6600 | 开发工程师 | 1 |
| 10 | Adam | 34 | 3800 | 运维工程师 | 1 |
| 11 | Clark | 22 | 3700 | 运维工程师 | 1 |
| 12 | Edward | 25 | 7700 | 项目经理 | 1 |
| 13 | Gary | 30 | 8200 | 项目总监 | 1 |
| 14 | Harry | 28 | 6200 | 项目组长 | 1 |
| 15 | Jeff | 32 | 5800 | 产品销售 | 2 |

现在我们要对上面的Emp表中的15条数据按照薪资字段构建索引，那么在MySQL数据库中，一个索引节点可以理解为如下的结构：


如果我们将索引节点和数据表中的数据进行挂钩，那么就会得到这样的结构：




从上面的图中我们不难看出：就是因为索引节点只是保存了数据表中真实数据的一部分（比如本案例中的员工薪资），并没有存储整条真实数据

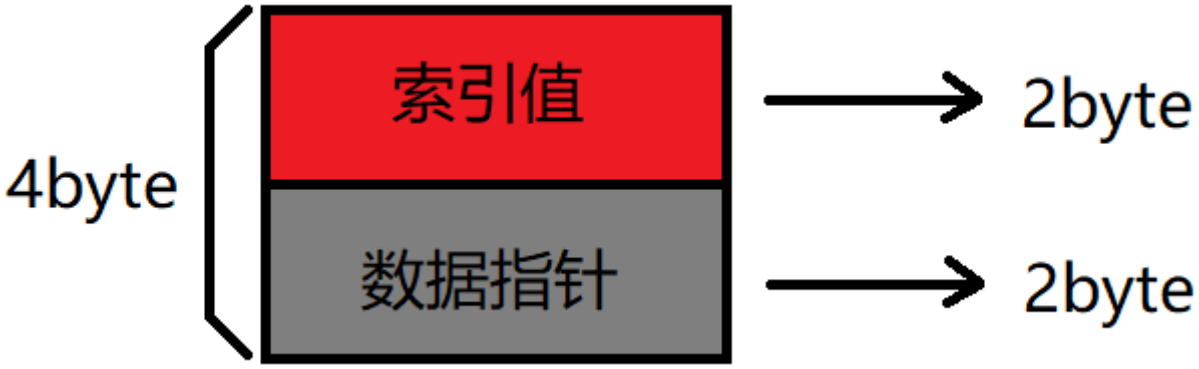
所以MySQL数据库的索引结构是不需要额外占用空间的

为了保证能够通过一个索引节点找到这个节点对应的数据表真实数据，我们会在索引节点中保存一个指向这条数据的指针，指向这条真实数据（当然指针的磁盘占用也是很小的）

同样的，我们也可以将上面的索引节点结构，看作是MySQL数据库中，实现索引的B+树结构的节点

并且我们还要对保存数据的硬盘结构进行一些定义：

假设现在我们的数据表和索引结构都存储在一块硬盘上，索引节点中的索引值和数据指针均为2byte大小，即：




那么，一个索引节点的大小就是4byte

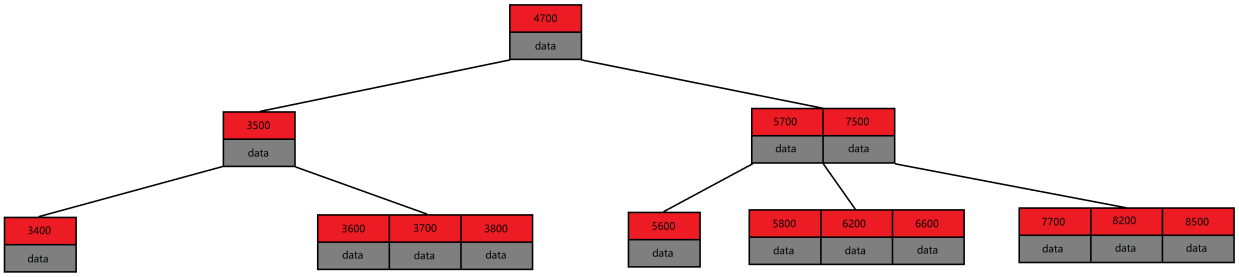
并且假设，一个磁盘块的大小是8byte，且不论是使用B树结构还是B+树结构实现索引，树结构的阶数都是4

接下来，就让我们一起比较一下，使用B树结构和B+树结构实现MySQL数据库的索引，具体有什么不同之处

*1.绘制上述数据对应的B树索引结构和B+树索引结构*

当我们使用B树结构实现索引的时候，B树中的每一个节点，不论是根节点、中间节点还是叶子节点，都要同时保存索引值和数据指针

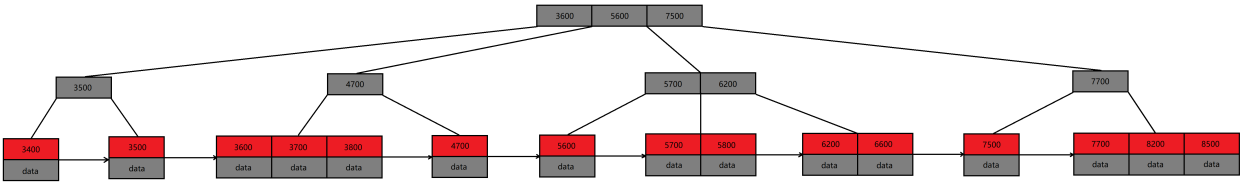
那么上面的数据就形成了这样的一个B树结构：




从图中不难看出，B树结构中的每一个节点都要存储两部分数据：索引值 + 数据指针

在B+树结构中，所有非叶子节点，仅仅保存一个索引值，并不需要保存具体数据指针，只有叶子结点才是真正同时保存索引和具体数据指针的

根据上述描述，我们可以得到如下的B+树结构：




从上图中我们不难看出这，为什么在B+树结构中，会存在“重复数据”的原因：这些重复数据并不是真正重复的节点，只是索引值重复了

重复的索引值用于指示子节点的位置：子节点之间依然保持左小右大的顺序

*2.查询数据执行磁盘读写次数的不同*

在B树当中，如果我们从树根开始，寻找索引值为3600对应的具体数据的话，需要经历3次节点查询，也就是说，此时我们要读取12byte的数据，才能够找到索引值为3600的节点

再根据前面的约定，一个磁盘块的大小为8byte，那么我们不难看出，此时需要执行2次磁盘数据读取，才能够完成这次查询操作

同样的，我们在B+树结构中查询索引值为3600对应的具体数据，同样需要经历3次节点查询，但是因为在查询B+树中间节点的时候，中间节点并不携带具体数据指针

所以，我们只要读取2 + 2 + 4 = 8byte的数据，就能够获得这个索引对应的具体数据，如果按照磁盘块大小来计算的话，就是读取1个磁盘块大小的数据内容

综上所述：在保证数据量相同的情况下，读取使用B+树实现的索引结构中的数据，比读取B树实现的索引结构中的数据，所经历的磁盘读写次数更少

*3.查询数据稳定性的不同*

我们还是比较上面两张图中的结构：

在B树中，节点的查询效率是不稳定的，最好情况下，我们只要读取根节点，就能够得到想要的数据；在最坏情况下，必须要读取到叶子结点，才能够得到所需的数据

但是在B+树结构中，因为只有叶子结点代真正存储数据指针，所以每一次查询的效率是相同的

但是这并不意味着B+树的查询效率就比B树的查询效率更低，因为前面我们已经说过，真正的磁盘查找，看的是对磁盘块的读取次数

而在数据量相同的情况下，B+树的磁盘块读取次数更少

综上所述：从整体上来讲，B+树的查询效率不仅不比B树更低，反而B+树的查询比B树的查询更加稳定

*4.范围查询性能的不同*

假设现在我们要查询工资在取值范围[3500, 6600]之间所有员工的信息

那么在B树结构中，我们必须每查询一个员工的信息，都必须从树根开始重新查询；

但是在B+树结构中，因为所有的叶子结点之间构成了一个有序单链表结构，所以只要我们查询到取值为3500的索引，就可以通过遍历单链表的方式，逐个找出后序所有的索引

这无疑也就降低了磁盘读取的次数，提升了范围查询的效率

综上所述：B+树的范围查询效率，比B树更高

经过上面的比较，最终我们可以得出一个结论：

在真实的文件系统当中，使用B+树结构构建的索引结构，不论是在磁盘读写的效率、稳定性方面，还是在范围查询的效率方面，都优于使用B树实现的索引结构

所以：B+树结构更适合于实现文件系统中的索引结构

# 2.Trie树（字典树）

在开发过程中，我们经常会遇见这样的场景：很多英语单词的开头是相同或者相近的，只有结尾的几个字符之间有区别

例如：Comparator和Comparable、Absolute和Abstract、Action和Activity等等……

假如说现在需要实现一个功能，用来保存这些单词，那么你会如何进行实现呢?

将这些单词一个一个的使用数组或者链表存储起来？容易实现，但是增删和查询的效率低得惊人，且十分浪费存储空间……所以这是最笨的办法

所以，我们一定要利用这些单词“开头相似”的特性，来存储这些单词

在数据结构的树形结构当中，就存在着这样一种结构，专门用来解决上述问题，这就是Trie树

Trie树结构也称之为前缀树或者字典树，是一种利用单词的相同前缀，简化存储结构，降低存储空间成本，并提高增删查询效率的树形结构

Trie树具有如下结构特征：

1.根节点不包含字符，除根节点意外每个节点只包含一个字符。

2.从根节点到某一个节点，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串不相同。

因为这些特征，Trie树广泛应用于大批量的字符串存储以及词频统计、字符串排序等场景之下

下面就让我们一起来研究Trie树的构建过程和实际的应用场景

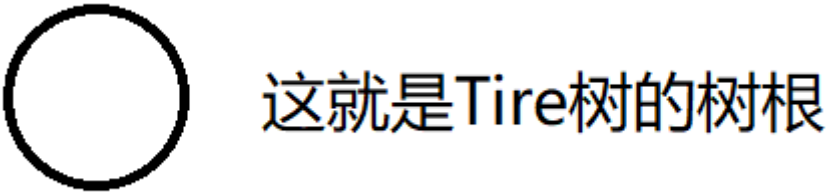
**①Trie树的构建**

假设我们有如下的20个单词：

| | | | | |
|---|---|---|---|---|
| pen | note | padding | pencil | absolute |
| action | java | abstract | and | run |
| activity | node | android | nodded | runnable |
| jack | parent | panda | rainbow | join |

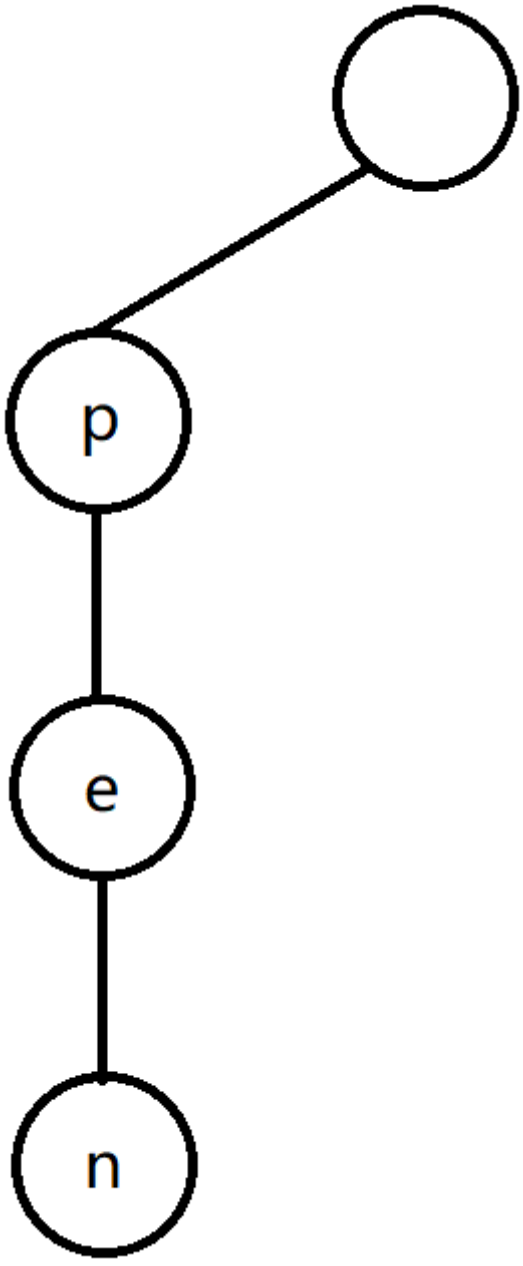
接下来我们就来演示如何创建一个Trie树，用来保存上面的单词：

步骤1：Trie树的根节点是不保存任何信息的，就是一个空节点



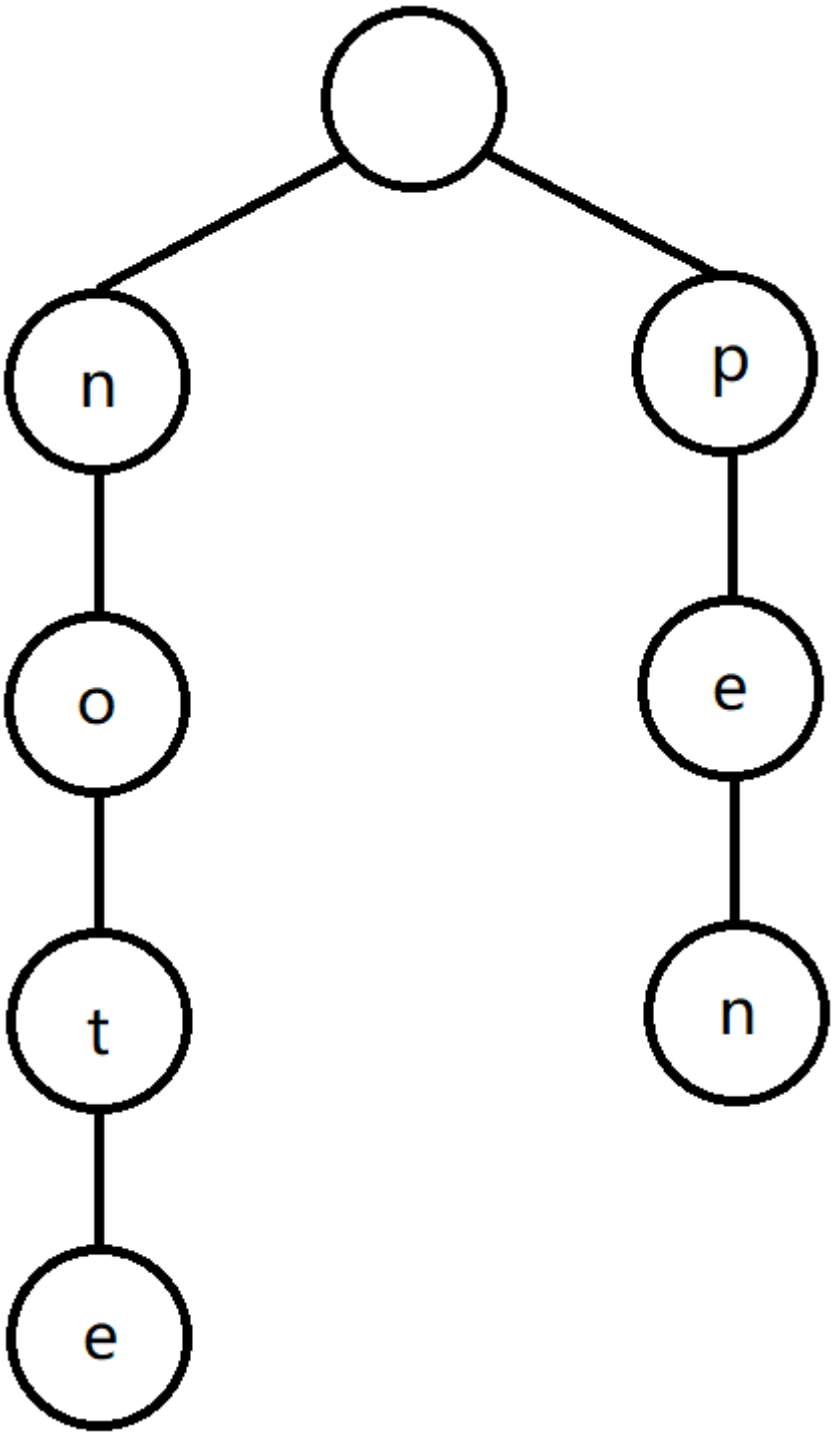

步骤2：找到表中的一个单词，将这个单词拆分成为单个字符，单词的首字母就是树根的孩子节点

每一个字符之间的关系，就是树结构中的父子节点关系，也就是：靠后的字符是靠前字符的孩子节点



步骤3：如果遇见前缀完全不相同的单词，我们可以选择按照单词首字母的关系，对根节点的所有孩子节点进行排序

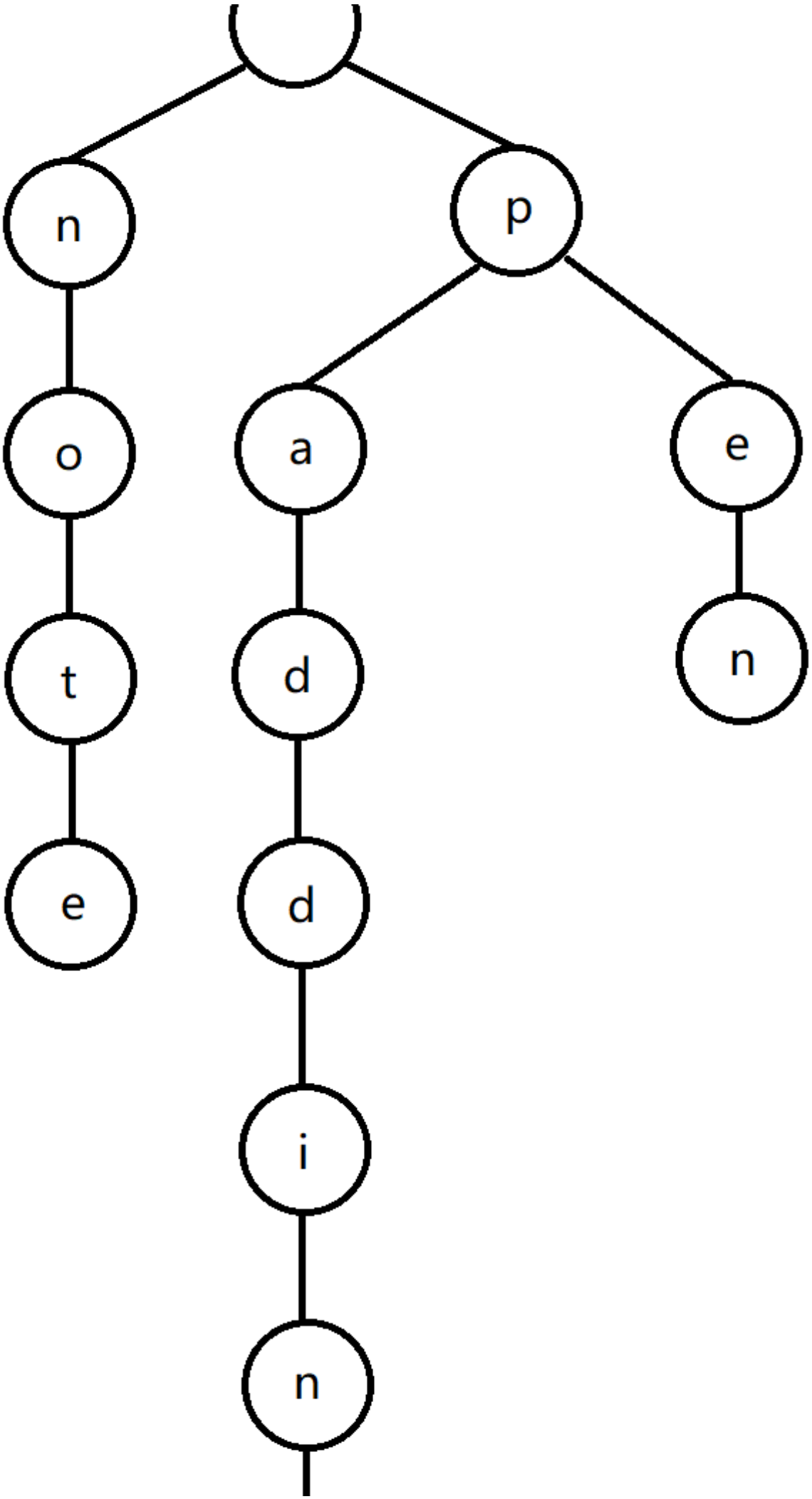
这样的操作有利于后序的所有单词进行排序

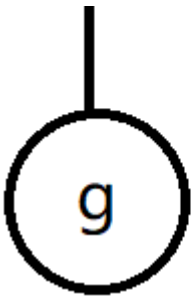



步骤4：如果遇见前缀相同的单词，则根据共同前缀，在分化出不同后序字符串的位置，为父节点添加其他子节点

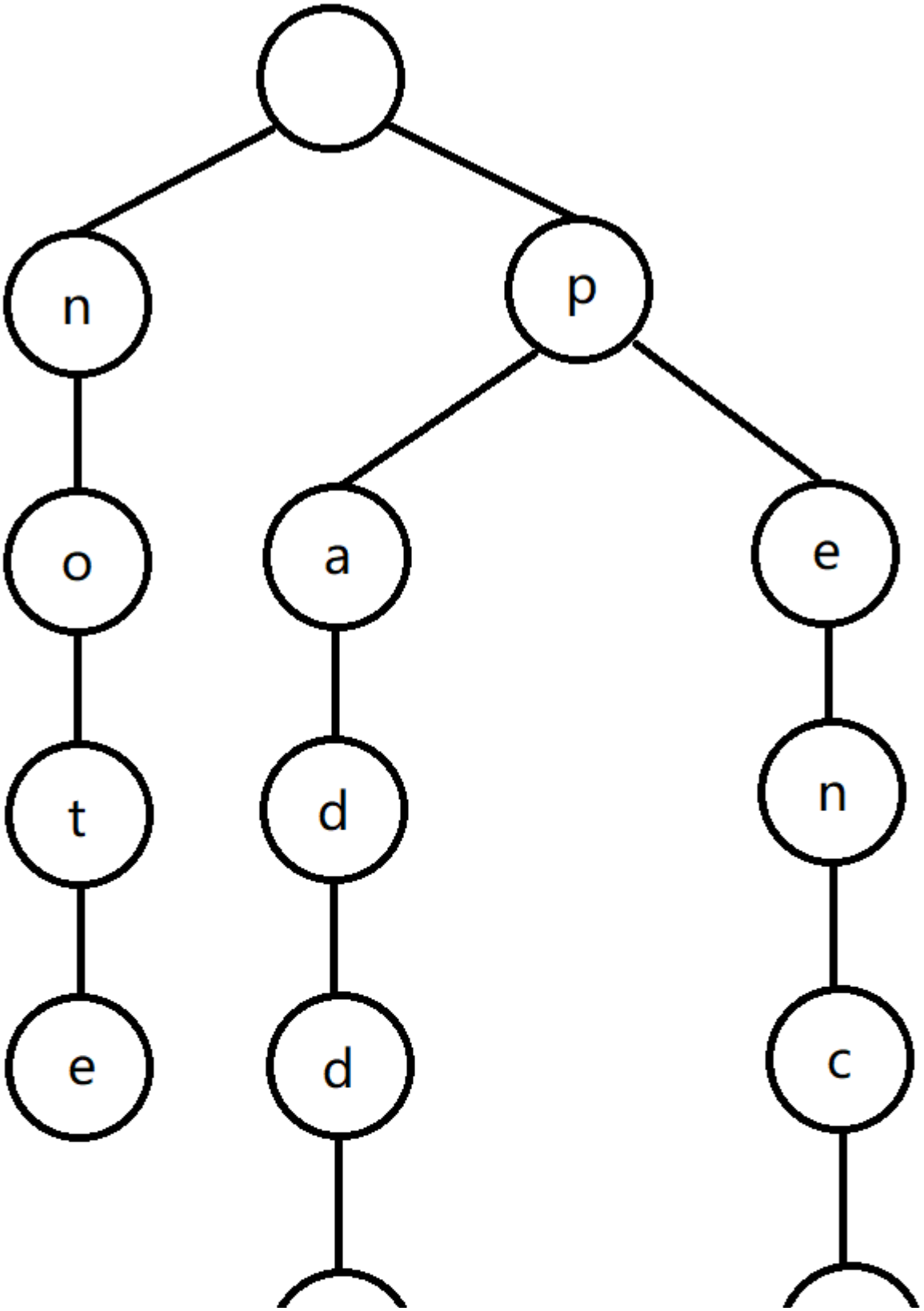
例如：padding和pen具有共同前缀p，所以在节点p之下有两个子节点：a和e

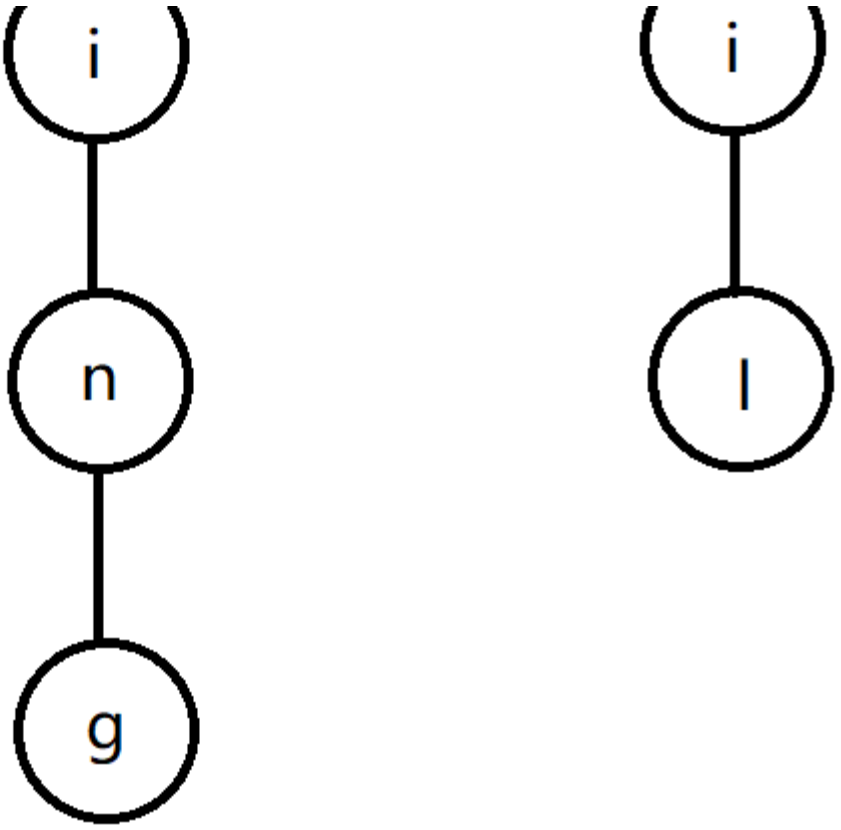
此处请注意：如果想要构建字符串有序的Trie树结构的话，在为任何中间节点添加子节点的过程中，都要保证同一父节点的所有子节点之间的有序性


n
p
o
a
e
t
d
n
e
d
i
n

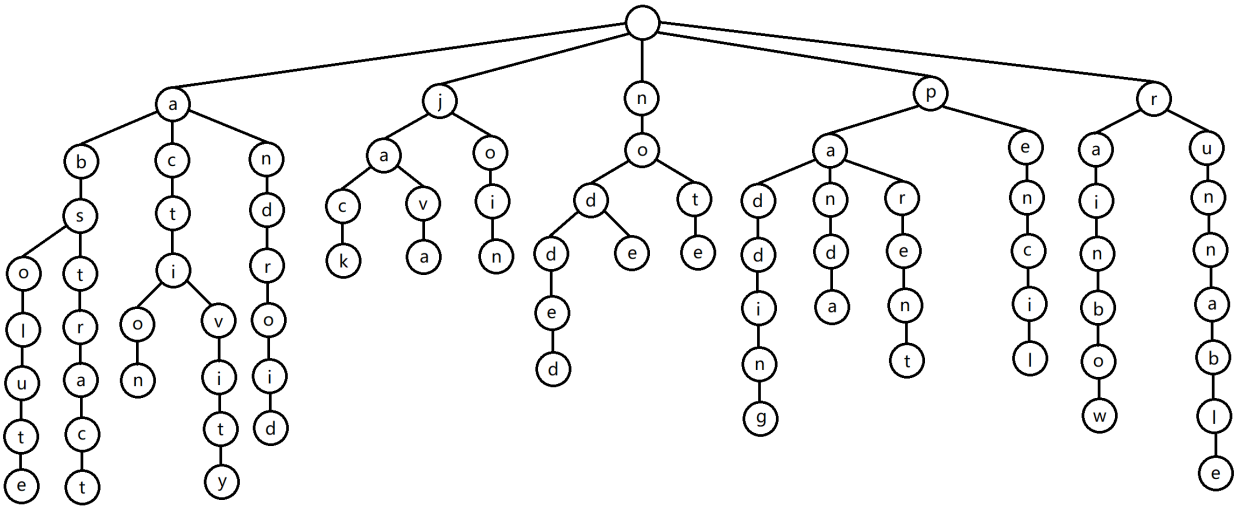



步骤5：如果遇见某一单词是另一个单词的前缀，例如pen是pencil的前缀，则直接在原有单词的基础上，继续延伸树结构即可






在明白了上面的步骤之后，最终我们能够将上面表格中的20个单词，整理成为如下的一个Trie树结构：



在上述Trie树结构中我们不难发现，如果对Trie树进行先序遍历，那么最终就能够对表格中的所有单词进行排序了

**②Trie树的使用场景**

Trie树除了能够用来保存大量字符串、进行字符串排序和统计词频之外，在其他很多方面也有应用

例如进行网络访问路径的匹配方面

下面我们看这样一道笔试题：

*在对域名进行分析中，常常会碰到"主域归属"问题。首先，我们有一个主域列表，如下所示：*

**.sports.sina.com.cn*

**.sina.com.cn*

**.baidu.com*

**.tencent.com*

**.com*

**.cn*

*等等，这个列表可能会包含百万级别的配置。*

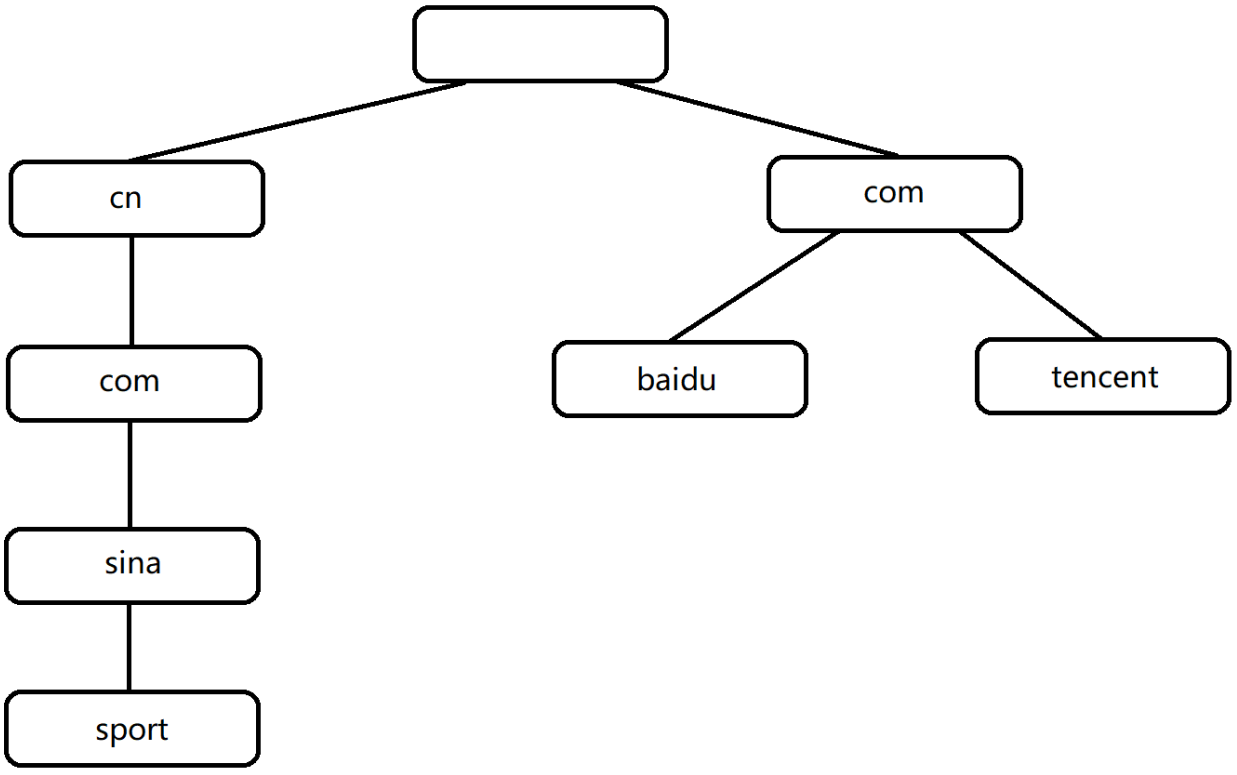
*在有这个配置的前提下，给定一个域名，比如roll.sports.sina.com.cn，希望能够找到它所匹配的最长的"主域"，比如，对于上面这个域名，应该匹配到*.sports.sina.com.cn这个主域。*

*请实现一个程序，从配置文件domain.txt读取主域列表，每行一个；从标准输入读取需要匹配的域名，每行一个；向标准输出打印：需要匹配的域名\t它匹配到的最长主域。*

*注意：请尽可能高效，使用正则匹配会非常慢。*

这道题乍一看上去貌似和Trie树结构没有任何关系，但是我们经过仔细分析发现：

如果将每一个主域名从后向前看，并且以.作为分隔符，那么就可以构成这样一个Trie树结构：



其余主域名同理，可以纳入这个Trie树结构中

然后我们再对任何一个域名进行主域名归属解析的时候，只要从后向前的，在这个Trie树结构中进行查找就行了

# 3.树状数组

**①拉哥藏钱的故事**

在正式开始学习树状数组这种结构之前，我们首先来看一个“拉哥藏钱”的故事：

> *拉哥婚后生活家教甚严（然而现实中的拉哥是一条单身Doge……），导致拉哥的私房钱总能被成功缴获……*
>
> *逆境之中，拉哥斗智斗勇，一口气开了8张银行卡用于藏匿私房钱。*
>
> *拉哥在藏钱的过程当中，需要不断的向其中任意一张银行卡进行存款操作（一般都是整数存款），而且还可能不定期统计其中多张银行卡的存款总数。*
>
> *那么，拉哥应该如何处理这8张银行卡，才能够快速有效的实现上述两种操作呢?*

在我们看完上述感人至深，催人泪（niao）下的拉哥藏钱的故事之后，让我们一起来分析一下：

首先，我们可以将拉哥用来藏钱的8张银行卡，看做一个整数数组，并且按照下标1-8进行编号的话，可以得到如下结构：

| a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8 |
|---|---|---|---|---|---|---|---|

而上面问题中，对银行卡进行存款的操作，可以看做是对数组元素的单点修改操作和对数组中元素的区间查询操作。

在我们学习数组的时候我们就知道，因为针对数组元素，我们可以使用快速随机访问进行定位并修改，所以我们对数组的单点修改操作的时间复杂度是O(1)的；

但是在对数组中的元素进行区间查询的时候，最坏的可能性就是统计数组中全部元素的总和，也就相当于将数组中的全部元素进行遍历和相加，其时间复杂度为O(n)。

那么，在对上述数组进行多次单点修改操作和区间查询操作的情况下，我们可以得出如下结论：

一维数组多次单点修改操作时间复杂度：O(n)

一维数组多次区间查询操作时间复杂度：$O(n^2)$

乍一看上去，对数组执行多次单点修改操作的时间复杂度尚可接受，但是对一维数组进行多次区间查询的时间复杂度就太高了！

尤其是在大数据领域当中，如果对一组数据序列进行区间查询操作的事件复杂度是$O(n^2)$，那么真正的执行时间将是无法接受的。

那么，是否有一种结构，能够提升对一维数组结构区间查询操作的效率呢？答案就是：树状数组结构。

**②前序知识：lowbit(n)操作**

工欲善其事，必先利其器！首先让我们来学习一个在后面要用到的非常重要的概念：非负整数的lowbit操作。

我们都知道：任意十进制非负整数，都可以转换成为2进制的表现形式，有0和1构成。

现有如下需求：要求计算出一个非负整数n的2进制表现形式当中，最后一个1以及后续所有0构成的整数取值。

而这一操作，我们即称之为求非负整数n的lowbit操作，可以表示为lowbit(n)。

举个例子：

非负整数n=38，n的二进制表示=0010 0110，则：lowbit(38) = lowbit(0010 0110) = 2

非负整数n=17，n的二进制表示=0001 0001，则：lowbit(17) = lowbit(0001 0001) = 1

非负整数n=24，n的二进制表示=0001 1000，则：lowbit(24) = lowbit(0001 1000) = 8

通过上述案例，我们应该能够很明白的观察出，对于非负整数n的lowbit(n)操作的含义。

那么落实到代码上，lowbit(n)操作应该如何实现呢？

以n=24为例：首先对n=24的二进制结构进行取反操作：取反结果=1110 0111，然后在末位加1：加1操作结果=1110 1000；

由此我们可以看出：原始的2进制取值与取反加1之后的2进制取值，从最后一个1开始到最后完全相同，只有前面的部分是不同的。

那么再利用与运算有0则0，同1为1的特性，对n=24原始的2进制取值与取反加1之后的2进制取值进行与运算，即可得到：

lowbit(24) = 0001 1000 & 1110 1000 = 1000 = 8

那么我们根据上式可得lowbit(n)的公式为：

lowbit(n) = n & (~n + 1)

又因为在计算机当中，存储整数使用的补码机制中，求非负整数相反数的操作，就是将其2进制位按位取反，末位加1，所以我们可以得到简化后的lowbit(n)公式：

lowbit(n) = n & -n

落实到代码当中即为：

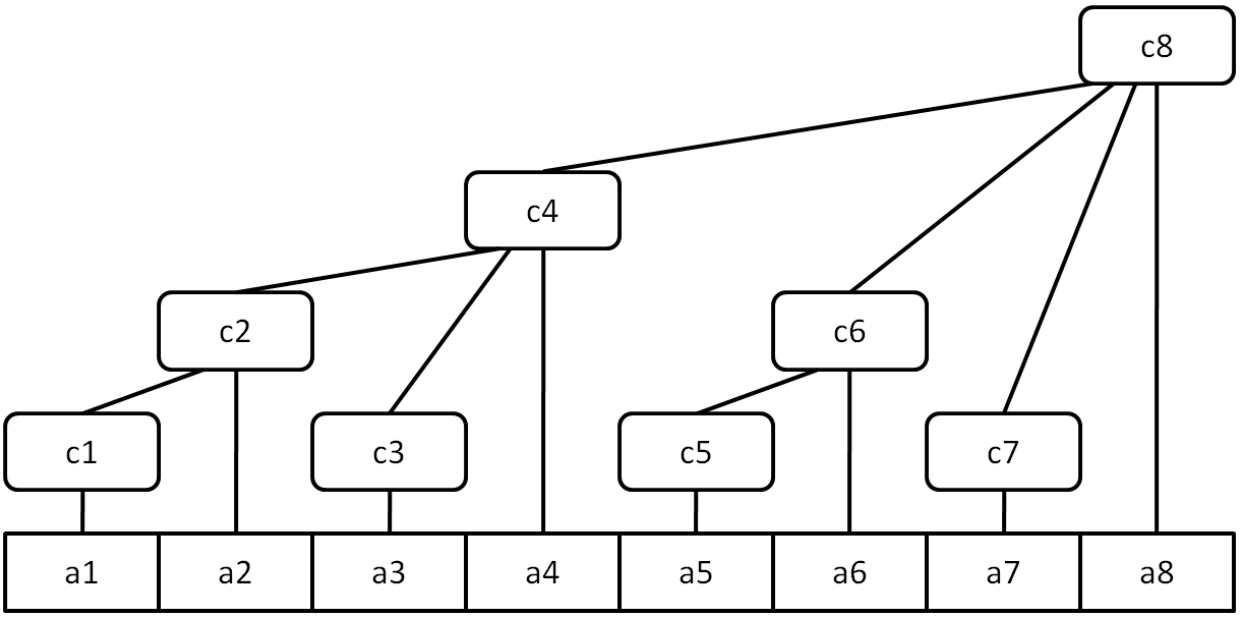
```java
/**
 * 求取非负整数n的2进制位中，最后一位1以及向后所有的0构成的整数值
 * @param n 一个非负整数
 * @return 非负整数n的2进制表示中，最后一位1余后续所有0共同构成的整数值
 */
public int lowbit(int n) {
    //return n & (~n + 1);
    return n & -n;
}
```

在了解了lowbit(n)操作之后，就让我们一起在学习树状数组的过程当中对其进行应用。

**③树状数组的构建**

在前文中我们提到，单纯的对数组进行区间查询操作，在大批量的访问情况下，时间复杂度较高，为O($n^2$)，

现在我们在原始数组的基础上，建立一个树状数组结构C：




而在这个树状数组结构C当中，具有如下特点：

特点1：若当前数组A的长度为n，则树状数组结构同样具有n个节点；

特点2：以上图为例，上面的树状结构C中各个节点c[i]与数组A中各个节点a[i]之间的关系是：

c[1] = a[1]

c[2] = a[1] + a[2]

c[3] = a[3]

c[4] = a[1] + a[2] + a[3] + a[4]

c[5] = a[5]

c[6] = a[5] + a[6]

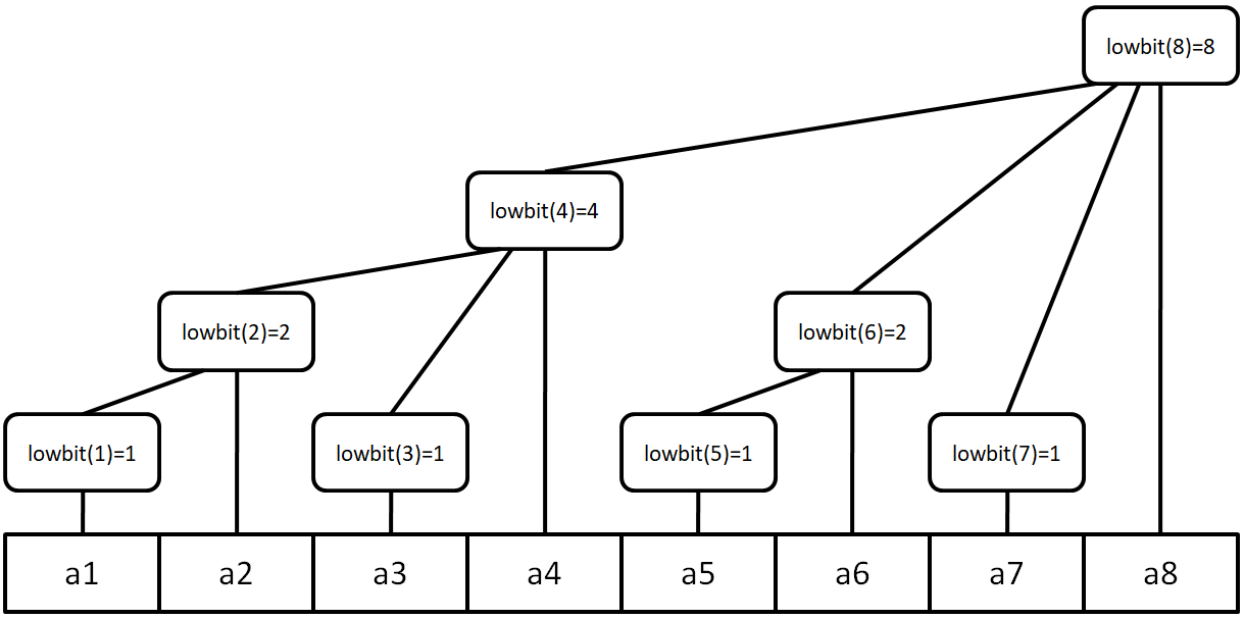
c[7] = a[7]

c[8] = a[1] + a[2] + a[3] + a[4] + a[5] + a[6] + a[7] + a[8]

特点3：整棵树的深度与数组长度n之间的关系为：树的深度 = $\log_2 n + 1$

看到这里，很多同学应该已经晕了：树状数组结构的每一个树节点与数组中每一个节点的关系，是如何推断出来的呢?

下面，让我们将树状数组结构中，每一个树节点的下标取值都进行lowbit运算，我们会发现：




通过仔（yi）细（dun）观（xia）察（meng）我们能够发现如下神奇的规律：

树状数组结构中每一个树节点维护的数组元素数量，就是这个树节点下标的lowbit运算的取值。

简单来说就是：

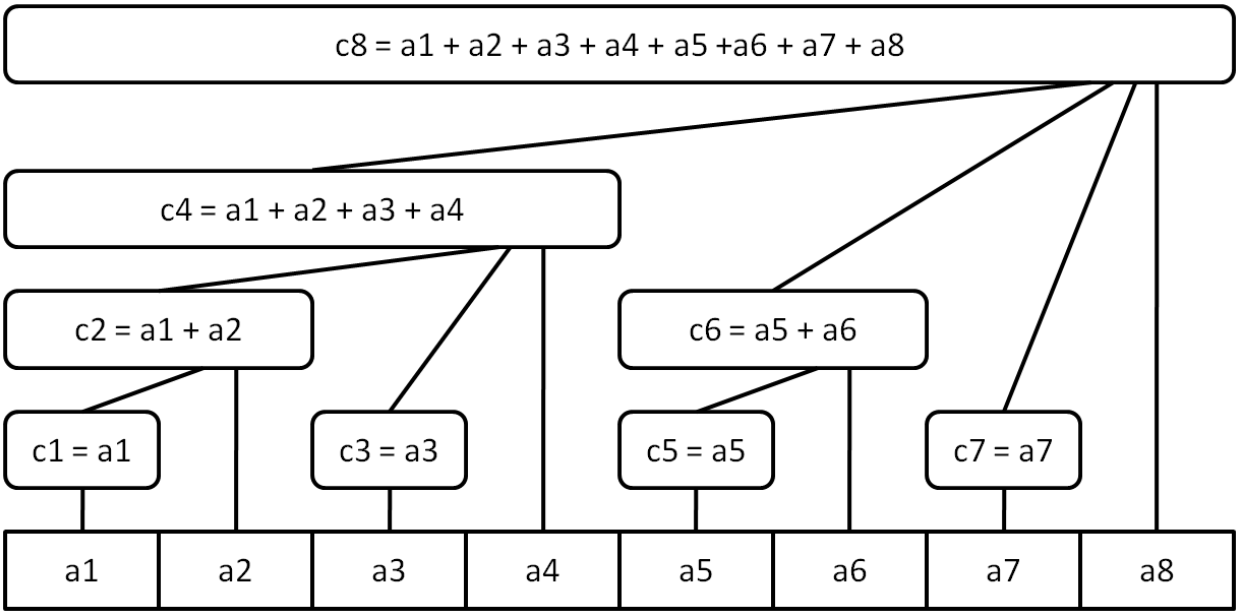
c1节点的下标lowbit运算结果为lowbit(1)=1，所以c1节点仅维护数组中a1节点的取值；

c2节点的下标lowbit运算结果为lowbit(2)=2，所以c2节点维护数组中a1 + a2节点取值的加和；

……

c8节点的下标lowbit运算结果为lowbit(8)=8，所以c8节点维护数组中a1 + a2 + ... + a8节点取值的加和。

如果将上述规律与树状数组结构节点的长度相结合，就能够得到如下图示：




到此为止，我们已经基本掌握了构建一个树状数组的操作方式，以及树状数组结构中，各个树节点与数组元素之间的对应覆盖关系。

接下来，我们将以上图为基础，一起讨论树状数组结构的一些基本操作以及这些操作的效率问题。

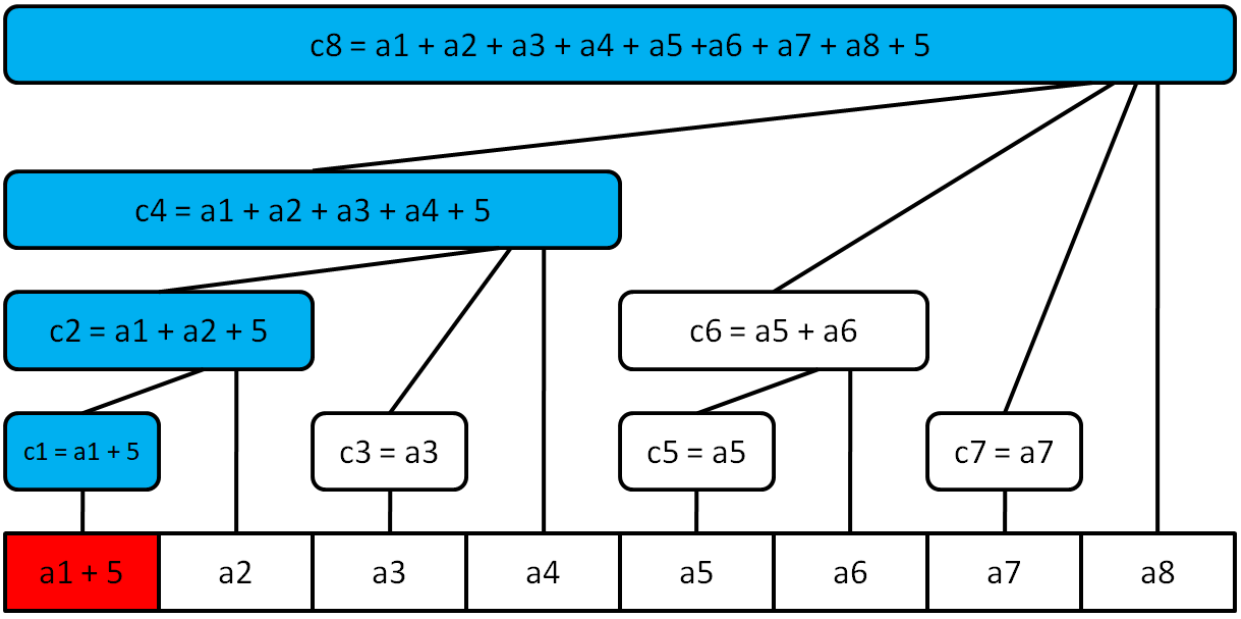
**④树状数组的基本操作**

在这里，我们首先对树状数组的两个基本操作进行讨论：单点修改操作和前序和查询操作。

*1.单点修改操作*

首先，我们假设对数组中元素a1进行加法操作，如：在元素a1原值的基础上+5，那么，除了要在数组元素a1上进行+5操作外，

我们还要考虑到需要对树节点中所有计算范围覆盖到a1节点的值进行+5操作，即：



从图中我们可以看出：树节点c1、c2、c4和c8之间，呈现父子节点的关系。而在这些节点的下标之间，还隐藏着如下规律：

c2节点为c1节点的父节点，则：2 = 1 + lowbit(1);

c4节点为c2节点的父节点，则：4 = 2 + lowbit(2);

c8节点为c4节点的父节点，则：8 = 4 + lowbit(4);

是的，上述规律总结出来就是：在树状数组的树结构当中，任意两个树节点之间如果存在父子关系，则两个树节点之间的下标满足如下关系：

父节点下标 = 子节点下标 + lowbit(子节点下标)

并且，这个规律是普适于同一树状数组中，任意两个具有父子关系的树节点之间的。

根据上述规律，我们可以给出树状数组中，对数组节点进行单点修改操作，并维护树节点取值的相关代码：

```
/**
 * 向树状数组中的数组结构下标为index的位置上，加上value的值
 * @param index 数组下标
 * @param value 加上的值
 */
public void add(int index, int value) {
    array[index] += value;
    //每一次循环，树节点下标增量为lowbit(i)，即找到自己的父节点
    for(int i = index; i < treeNodes.length; i+= lowbit(i)) {
        treeNodes[i] += value;
    }
}
```

从图上来看，我们不难发现：这一操作及其类似于沿着二叉树的一条路径，得到路径上所有节点操作的逆过程，也就是不断二分操作的逆过程！

而在树状数组中，虽然并不是每一次寻找父节点都是标准的“二分”操作，但是从整体上来看，并不影响我们对时间复杂度的计算。故而：

树状数组执行一次单点修改操作的时间复杂度是：$O(\log_2 n)$

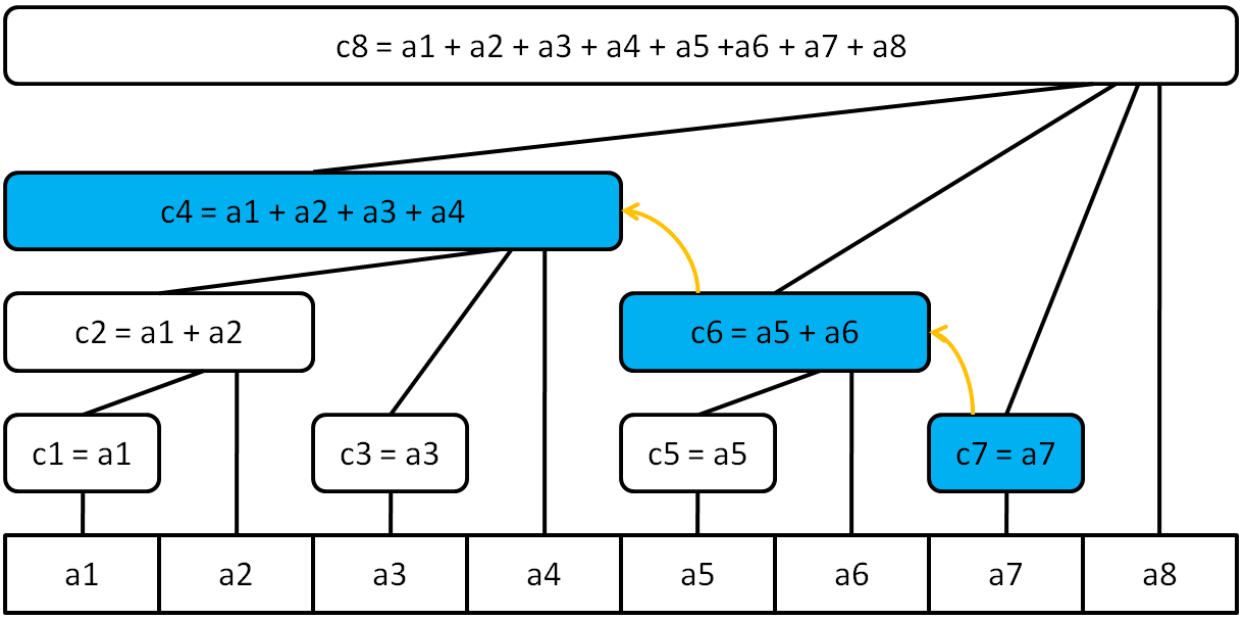
树状数组执行多次单点修改操作的时间复杂度是：$O(n\log_2 n)$

从最后的时间复杂度结果来看，树状数组执行单点修改操作的效率，确实低于单纯对数组结构执行单点修改操作的效率。

*2.前序和查询操作*

所谓下标n的前序和，指的就是从下标1开始计算，将下标从1（包含）至n（包含）的所有数组元素取值相加所得到的累加和取值。

同样以上图为例，假设需要计算n=7的前序和操作，即为计算数组中a[1] + a[2] + a[3] + ... + a[7]的元素累加和，那么在树状数组中，可以得到如下计算结构：




即：下标为7的元素的前序和 = c[7] + c[6] + c[4]的取值，而树节点c7、c6与c4之间又存在如下规律：

c7为计算起点；

c6为c7后续需要相加的节点，则：6 = 7 - lowbit(7)；

c4为c6后续需要相加的节点，则：4 = 6 - lowbit(6)；

从这一规律，我们不难给出计算数组前序和操作的代码：

```
/**
 * 计算下标为n元素的数组前序和的操作，
 * 即计算从a[1]（包含）至a[n]（包含）之间所有元素累加和的操作
 * @param n 前序和终点下标
 * @return 从a[1]至a[n]所有元素的累加和
 */
public int preOrderSum(int n) {

    int sum = 0;

    for(int i = n; i > 0; i -= lowbit(i)) {
        sum += treeNodes[i];
    }

    return sum;
```

```
16
17 }
```

同样的，这一过程也和二叉树结构中，延一条路径获得其中所有节点的逆过程相似，所以：

树状数组执行一次前序和查询操作的时间复杂度是：$O(\log_2 n)$

树状数组执行多次前序和查询操作的时间复杂度是：$O(n\log_2 n)$

从这一点看来，利用树状数组结构查询前序和操作的效率，远高于单纯使用数组结构进行前序和查询操作的效率！

*3.总结*

从上面我们对树状数组结构进行单点修改操作，以及前序和查询操作的总结看来：

树状数组结构牺牲了一部分单点修改操作的效率，换来了前序和查询操作效率的大幅提升。

所以从这一点上来说，树状数组适合于一些元素不经常发生变化，而会经常进行各种区间求和、计算最大（小）值、计算平均值等统计工作的数据。

在树状数组上述基本操作的基础上，还能够衍生出更多相对复杂的操作。在下面的章节当中，我们要做的就是将这些高级操作进行整理和说明。

**⑤树状数组的高级操作**

在上面的内容当中，我们给出了树状数组元素单点修改的方法add(int index, int value)和前序和查询的方法preOrderSum(int n)的方法，

那么从这两个方法出发，我们还能够总结出更多关于树状数组元素的高级操作方法。

*1.单点查询操作*

方法定义：

```
1 /**
2  * 树状数组单点查询方法，可以查询树状数组中，数组结构中下标为index的元素取值
3  * @param index 树状数组中，数组部分元素下标
4  * @return 树状数组中，数组部分下标为index的元素取值
5  */
6 public int get(int index);
```

方法作用：

查询树状数组中，数组结构下标为index的元素取值。

方法实现：

```
/**
 * 树状数组单点查询方法，可以查询树状数组中，数组结构中下标为index的元素取值
 * @param index 树状数组中，数组部分元素下标
 * @return 树状数组中，数组部分下标为index的元素取值
 */
public int get(int index) {
    return preOrderSum(index) - preOrderSum(index - 1);
}
```

注意：本例中给出的单点查询方法的实现是通过前序和查询方法实现的。但是通过直接查询数组下标的方式也是可以的。

*2.区间查询操作*

方法定义：

```
/**
 * 树状数组区间查询方法，
 * 可以查询数组结构从a[start]（包含）至a[end]（包含）范围内所有元素的加和取值
 * @param start 区间查询起点下标
 * @param end 区间查询终点下标
 * @return 返回树状数组结构中，数组结构指定连续范围内元素的加和
 */
public int getRange(int start, int end);
```

方法作用：

返回树状数组结构中，数组结构中指定下标范围内所有元素的连续加和，即：返回从a[start]（包含）至a[end]（包含）范围内的所有元素加和。

方法实现：

```
/**
 * 树状数组区间查询方法，
 * 可以查询数组结构从a[start]（包含）至a[end]（包含）范围内所有元素的加和取值
 * @param start 区间查询起点下标
 * @param end 区间查询终点下标
 * @return 返回树状数组结构中，数组结构指定连续范围内元素的加和
```

```
 */
public int getRange(int start, int end) {
    return preOrderSum(end) - preOrderSum(start - 1);
}
```

### *3.差分数组的引入

在前面我们使用的树状数组中，其数组部分结构存储的数据就是一般的整数值，而这一类树状数组结构对单点修改操作的时间复杂度为$O(\log_2 n)$，

但是如果我们要求对一段范围内的全部数组元素都进行修改的话，如果仅仅使用暴力的循环方式执行，那么其区间修改操作的时间复杂度将降至$O(n^2)$，

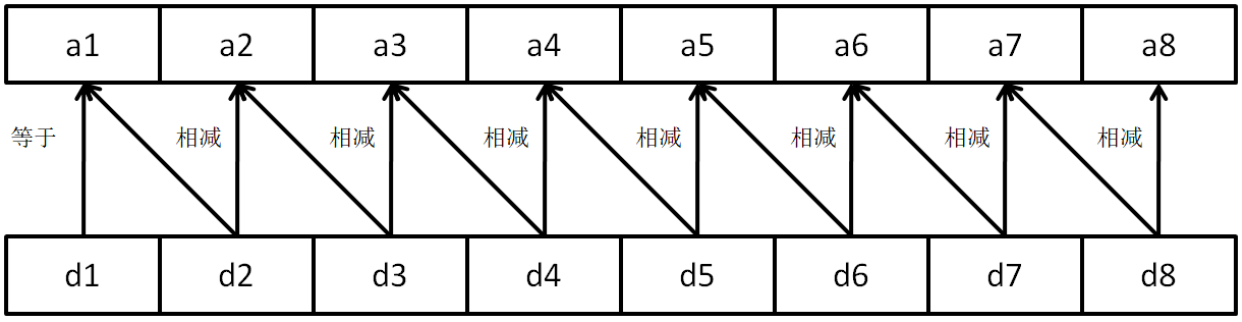
这显然不是我们希望看到的。所以为了解决这个问题，我们引入了差分数组的概念，并在对差分数组构建树状数组结构的基础上，提升区间修改和区间查询的效率。

首先给出差分数组的概念：

假设有原始数组a，长度为n，下标从1开始计算，那么与之对应的差分数组d可以定义为：

$$d[i] = \begin{cases} a[i] & i = 1 \\ a[i] - a[i-1] & i > 1 \end{cases}$$

也就是说，从下标为2的元素开始，差分数组的每一个元素取值，都等于原始数组对应下标位置元素与前一位元素的差值，用图形表示即为：




而差分数组的特性就是：在对原始数组元素进行区间修改的时候，差分数组可以将这一段原始数组元素的修改化作仅对两个差分数组元素的修改。

什么意思？举个例子：有如下的原始数组和差分数组取值：

| 下标 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

| 原始数组a | 1 | 5 | 2 | 4 | 8 | 2 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|
| 差分数组d | 1 | 4 | -3 | 2 | 4 | -6 | -1 | 6 |

现要求对原始数组中，下标从3到7的所有元素加2进行操作，对比两组原始数组和差分数组的取值：

| 下标 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 原始数组a | 1 | 5 | 4 | 6 | 10 | 4 | 3 | 7 |
| 差分数组d | 1 | 4 | -1 | 2 | 4 | -6 | -1 | 4 |

很明显的，在原始数组对5个元素进行修改的时候，差分数组仅需要修改其中的两个元素即可，而这两个元素之间的所有差分值均未发生变化。

即：对原始数组a的[i, j]区间元素加n，等价于修改差分数组中元素：d[i] + n且d[j+1] - n，而d[i+1]到d[j]之间的元素取值不变。

所以，对于区间修改操作而言，差分数组的操作效率远高于单纯的原始数组，所以，使用差分数组构建的树状数组结构，更适合于进行区间修改等操作。

同学们看到这里之后，可能会产生这样一个疑问：如果我仅仅使用一个数组记录差分取值的话，我们如何进行数组原始元素的单点查询操作呢？

实际上在差分数组中，因为d[1] = a[1]，所以我们可以计算得到：a[i] = d[1] + d[2] + ... + d[i]，也就是说：差分数组第i位的前序和取值，就是原始数组a[i]的取值。

*4.区间修改，单点查询*

在前面的内容中我们已经介绍过关于使用差分数组的树状结构的如下两个方面重点内容：

重点1：为了方便实现树状数组结构的区间修改，我们将树状数组中的一般数组改为了差分数组，使得一般数组的区间修改变成了差分数组中修改区间两端两个元素的修改；

重点2：差分数组中，下标为1的元素取值等于原始数组下标为1的元素取值，而原始数组中后续所有元素均等于差分数组对应位置元素的前序和。

根据上述两条内容，我们可以直接给出使用差分数组实现树状数组结构的差分数组元素修改、差分数组前序和查询、原始数组区间修改和原始数组单点查询等方法的代码：

差分数组元素修改：

```java
/**
 * 对diff差分数组元素进行添加操作的方法
 * 注意：该方法修改的是差分数组的元素取值，而不是原始数组的取值
 * @param index 差分数组下标
 * @param value 添加的值
 */
private void addDiff(int index, int value) {
    diff[index] += value;
    for(int i = index; i < diffNodes.length; i+= lowbit(i)) {
        diffNodes[i] += value;
    }
}
```

差分数组前序和查询：

```java
/**
 * 对差分数组下标为index位元素求取前序和的方法
 * 注意：该方法求取的是差分数组的前序和，而不是原始数组的前序和
 * @param index 差分数组下标
 * @return diff[index]位前序和
 */
private int diffPreOrderSum(int index) {
    int sum = 0;
    for(int i = index; i > 0; i -= lowbit(i)) {
        sum += diffNodes[i];
    }
    return sum;
}
```

基于差分数组元素修改和差分数组前序和查询的方法，我们有可以得到针对原始数组元素进行区间修改和单点查询的方法。

原始数组区间修改：

```java
/**
 * 对原始数组元素进行区间修改的方法
 * 对原始数组的区间修改，直接体现在对差分数组对应修改区间起点和终点元素的修改上
 * 即：对原始数组array中，
 * 从array[start]（包含）开始到array[end]（包含）结束的元素全部加value的操作
```

```
 * 等价于对差分数组元素diff[start]和diff[end+1]加上value的操作
 * @param start 原始数组区间修改的起点下标
 * @param end 原始数组区间修改的终点下标
 * @param value 原始数组进行区间修改的取值
 */
public void addRange(int start, int end, int value) {
    addDiff(start, value);
    if(end + 1 < diff.length) {
        addDiff(end + 1, -value);
    }
}
```

原始数组单点查询：

```
/**
 * 针对原始数组取值进行单点查询的方法
 * 原始数组array的元素取值，等于与之对应下标的差分数组元素的前序和取值
 * @param index 单点查询的下标取值
 * @return 原始数组array[index]取值
 */
public int get(int index) {
    return diffPreOrderSum(index);
}
```

*5.区间修改，区间查询*

在了解了上面对于使用差分数组所执行的区间修改和单点查询操作之后，我们再来考虑一下关于使用差分数组的树状数组结构的区间查询操作。

之前我们说过：对使用差分数组实现的树状数组结构的单点查询，就等于是对差分数组进行同等下标的前序和查询操作，即：a[i] = d[1] + d[2] + ... + d[i]。

那么如果我们需要对这个树状数组进行一次原始数组取值的区间和查询操作，就会得到如下的结果：

假设：查询原始数组中，下标从1到i的区间元素和，操作为：

a[1] + a[2] + a[3] + ... + a[i] = (d[1]) + (d[1] + d[2]) + (d[1] + d[2] + d[3]) + ... + (d[1] + d[2] + d[3] + ... + d[i])

很明显的，如果我们仅仅是进行上述操作，那么这个树状数组的区间和查询的事件复杂度还是会退化为O($n^2$)，这也不是我们想要看到的。

但是我们可以将上述公式进行整理，得到如下的新公式：

a[1] + a[2] + a[3] + ... + a[i] = (d[1] * i) + (d[2] * (i-1)) + (d[3] * (i-3)) + ... + (d[i] * 1)

但是，在上面的公式当中，i的取值是变化的，所以我们对上述公式进行如下变形：

(d[1] * i) + (d[2] * (i-1)) + (d[3] * (i-3)) + ... + (d[i] * 1) = i * (d[1] + d[2] + d[3] + ... + d[i]) - (0 * d[1] + 1 * d[2] + 2 * d[3] + ... + (i-1) * d[i])

将上述公式使用数学公式连续起来进行表达就是：

$$\sum_{i=1}^{n} a[i] = i * \sum_{i=1}^{n} d[i] - \sum_{i=1}^{n} (d[i] * (i-1))$$

公式中n表示原始数组的长度。

上述公式看上去复杂，但是从公式本身出发的话我们能够发现：d[i]就是差分数组中的每一位元素，是已知的，所以我们只要再创建一个树状数组，用来维护d[i] * (i-1)的值就行了。

我们将这个用来维护d[i] * (i-1)的数组，称之为差分和数组。

将上一章节中给出的针对单独差分数组操作的代码进行修改，得到如下代码：

差分和数组元素修改：

```
/**
 * 对diffSum差分和数组进行添加操作的方法
 * @param index 差分和数组下标
 * @param value 添加的值
 */
private void addDiffSum(int index, int value) {
    diffSum[index] += value;
    for(int i = index; i < diffSumNodes.length; i += lowbit(i)) {
        diffSumNodes[i] += value;
    }
}
```

差分和数组前序和查询：

```
/**
 * 对差分和数组下标为index位元素求取前序和的方法
 * @param index 差分和数组下标
 * @return diffSum[index]位前序和
 */
private int diffSumPreOrderSum(int index) {
```

```
    int sum = 0;
    for(int i = index; i > 0; i -= lowbit(i)) {
        sum += diffSumNodes[i];
    }
    return sum;
}
```

修改后的差分数组元素修改：

```
/**
 * 对diff差分数组元素进行添加操作的方法
 * 并且同时维护diffSum差分和数组及其对应树状结构
 * 注意：该方法修改的是差分数组的元素取值，而不是原始数组的取值
 * @param index 差分数组下标
 * @param value 添加的值
 */
private void addDiff(int index, int value) {
    diff[index] += value;  //维护差分数组
    //维护差分数组树结构节点
    for(int i = index; i < diffNodes.length; i+= lowbit(i)) {
        diffNodes[i] += value;
    }
    //维护差分和数组及其对应树结构节点
    addDiffSum(index, value * (index - 1));
}
```

根据上述代码，可以得到使用差分数组和差分和数组共同维护的树状数组结构中，对原始数组的如下操作：

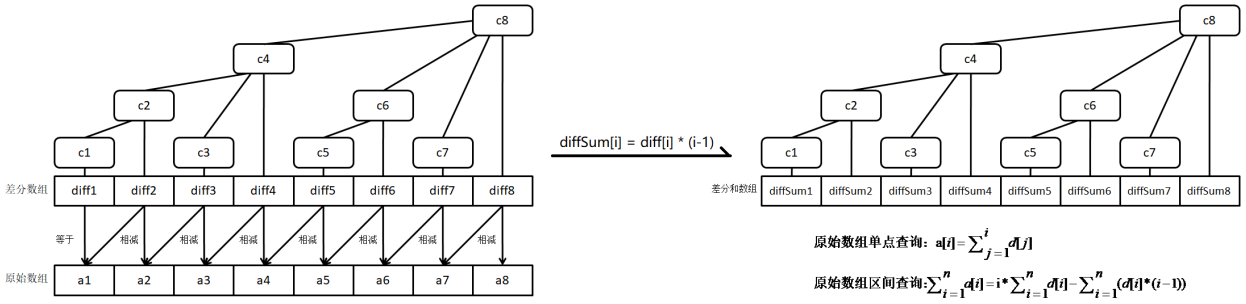
原始数组前序和查询：

```
/**
 * 针对原始数组的前序和查询操作
 * @param index 原始数组前序和查询终点下标
 * @return 原始数组前序和
 */
public int preOrderSum(int index) {
    return index * diffPreOrderSum(index) - diffSumPreOrderSum(index);
}
```

原始数组区间查询：

```
/**
 * 针对原始数组的区间和查询方法
 * @param start 原始数组区间起点下标（包含）
 * @param end 原始数组区间终点下标（包含）
 * @return 原始数组从array[start]（包含）到array[end]（包含）之间元素的区间和
 */
public int getRange(int start, int end) {
    return preOrderSum(end) - preOrderSum(start-1);
}
```

上述使用差分数组与差分和数组共同维护的树状数组结构使用图示表示即为：

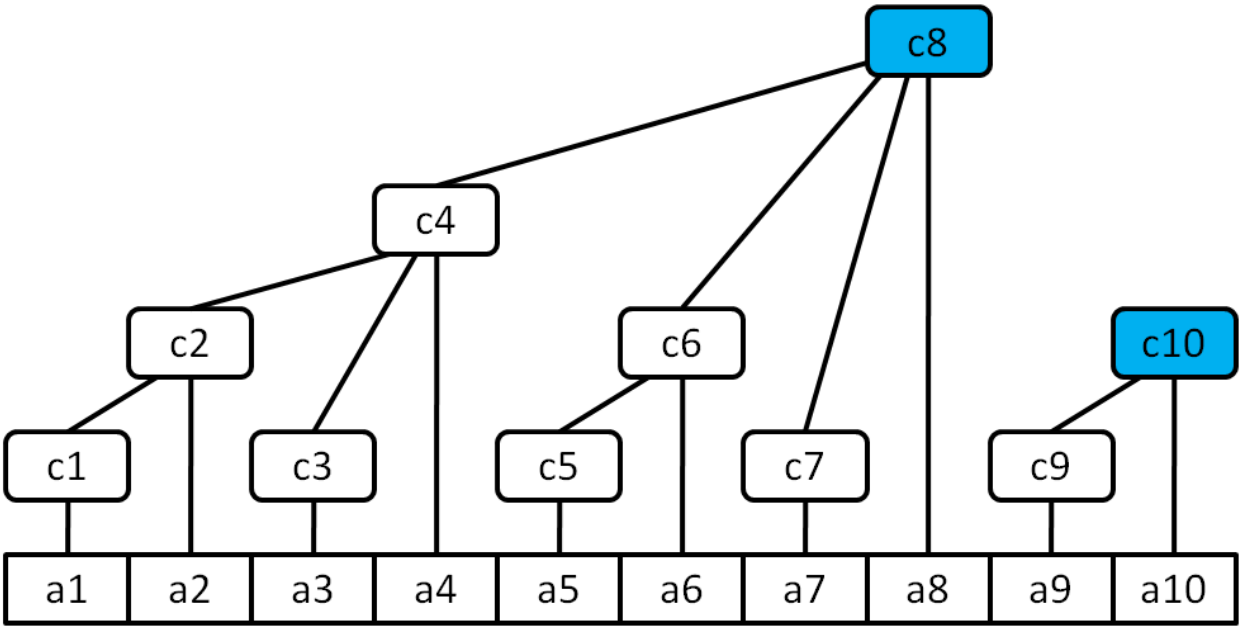



## *6.其他注意事项*

在前面我们介绍的树状数组结构当中，不管是一般的树状数组，还是使用差分数组的树状数组，为了理解方便，我们实际上使用的是一个极为特殊的例子。

之所以说他特殊，是因为在这个案例当中，树状数组的树结构，是一个完整的树结构，而在一般化的树状数组当中，数组上面的，实际上应该是一个森林结构。

比如：当数组长度为10，下标从1开始计算的时候：



其中树结构节点c8与c10就分别隶属于不同的两个树结构当中，而以c8为根节点的树结构，与以c10为根节点的树结构之间，构成了森林结构。

虽然上图中的树状结构分各个节点之间，分属于不同的独立树结构，但是我们在之前篇章中总结的，针对树状数组节点的各种规律，依然是成立的。