**树结构字符串检索实验报告**

**二叉搜索树（Binary Search Tree）：**

对于使用二叉搜索树来进行字符串匹配，其实现思路与一般的二叉搜索树建立于查找相似，但为了方便树的建立与查找时的效率，故先将各字符串hash为一unsigned int型整数再使用树的构造函数进行插入。

所使用数据结构为：

|  |
| --- |
| typedef struct btnode {  unsigned int val; //将字符串转发为数字后再比较  int bf; //平衡因子balance flag  struct btnode\* left, \* right;  }btnode, \*btree; |

首先将字符串转换为unsigned int后，使用InsertAVL(&root, result, &taller);将其插入，之后再对树进行平衡操作，保持其仍为一平衡二叉树。这样在之后的查找时只需使用

O(logN)的时间，最终结果为：

|  |
| --- |
| 1269614 19837 19944 986004 516816 |

可以看出每一个数据都建立的一个树节点，内存消耗较大。

**B+树实现:**

B+树是在数据系统中广泛使用的索引结构，其使用叶子节点来存储记录，非叶子节点则作为索引，本例数据结构定义为：

|  |
| --- |
| typedef struct bplus\_node {  int keynum; //该节点的关键字的个数，可以看作keys[]的逻辑边界  KEY\* keys; //主健，最大max=m个,最小min=m/2个, 空间max+1，以备分裂操作时的中间步骤使用  /\*int\* data; //存放数据的数组，  本例中keys就代表了要存放的数据，故不需要data字段\*/  struct bplus\_node\*\* child; //字节的指针数组  struct bplus\_node\* parent;  struct bplus\_node\* next; //兄弟节点，仅在叶子节点内有意义  }bplus\_node, \* bplus\_tree; |

相较于其他实现方式，B+树的实现较为复杂，在节点分裂时还需考虑叶子节点与非叶子节点两种情况进行分裂操作。而判断其是否为叶子节点则依靠child == NULL 来进行判断。而在实现父节点关键字更新时还需考虑父与子两个节点之间的关系与操作，可以说实现难度为本次实验最高，本例使用#define BranchM 5 //5阶B+树，结果为：

|  |
| --- |
| 957372 23624 23279 986004 516816 |

由于许多索引节点的存在，在使用了5阶B+树的前提上节点树仍有95万，但可以看出内存的消耗十分小。

**Radix Trie（基数树）实现：**

Trie树依据字符串的前缀进行查找，通过共用相同前缀减少了许多无意义的存储消耗与查找比较，效率很高。具体数据结构为：

|  |
| --- |
| typedef struct trienode {  struct trienode\* chidren[branch]; //本例为4叉树  char end; //是否能终结  }trienode, \* trietree; |

本例实现较为简单，读取各字符串的关键字，并依据两个bit的01四种情况作出四种分支子树存储，查找与插入类似，读取关键字判断是四种分支中的哪个子树往下查找，并在遍历完关键字后查看该节点的end，查看是否有字符串在此处结束。结果为：

|  |
| --- |
| 11296948 220643 23761 986004 516816 |

由于本例关键字是使用两个bit判断，节点会比前例要多得多达到了千万级，内存消耗也十分巨大。

**2基数的Radix Trie：Patricia Trie实现：**

实现基本与基数数实现一致，只是关键字仅使用一个比特，其余代码基本一致。结果为：

|  |
| --- |
| 21931421 514017 47298 986004 516816 |

可以看出，仅使用一个bit作关键字，一个char就需要8个节点，总节点数达到了2千万，内存消耗更是达到了500MB级别，十分巨大。

**总结：**

在多种树结构实现字符串匹配的实现中，B+数对内存的消耗十分小，在大型数据库中使用十分占优，而平衡二叉数在查找时相当于进行二分查找，时间消耗为O(logN)，十分快速。