计算机应用编程实验报告

计算机应用编程实验2：树结构字符串检索

2020110656 高垚 2020111562 杨恩

# 实验目标与意义

# 随着网络技术的普及，互联网的信息逐渐庞大，如何在海量且冗余的信息中找到用户所需的结果是互联网每天都要应对的难题。信息的基数与增长速度，用户查找内容的多样性，查询结果的实时动态变化都使精确搜索成为互联网面临的挑战。

# 本实验的目的是设计一套能够在庞大的词库中寻找特定目标的算法。需要通过四种树形结构算法，在给定的具有127万多个词语的dict.txt词典中，查找有1.7万个词的string.txt文本中匹配的词。

# 系统设计与实现

根据实验目标要求，综合考虑时间复杂度和空间复杂度，在实验中，利用m阶B+树，原始256叉树，m节点分支数和2叉压缩树，四种结构体，完成对dict.txt文件中127万个字符串的建立，并通过各自的算法，搜索在string.txt文件中查找一样的字符，并写入result.txt文件中。

## M阶B+树

M阶B+树算法是一种数据库系统中广泛使用的索引结构，它具有查询速度快，效率高，查询性能高，所有叶子节点形成有序链表，适合范围查询等优势。

1 2

3 5

6 8

9 11

13 15

11 15

8 15

2 5 8

图 1 B+树

### 实验思路

本节概括了实验的流程。

- 为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 设置合适的参数m，m决定了B+树的结构。（本实验设定m为64）

2. 初始化一棵B+树，把根的多个值赋值为0。

3. 开始进行插入操作，每读取dict一个关键词，计算Hash值，给B+树插入一个节点。

4. 读取完dict文本中的每一行关键词时，为叶子节点添加指针。

5. 读取string文本中的每一行关键词，并计算Hash值，然后查找。

6. 如果成功查找到，则写入result文本中。

其中涉及到的函数有：

|  |
| --- |
| /\*本实验中用到的函数\*/  void bplus\_init\_tree();//初始化一个B+树。  int bplus\_insert\_recoder(char \*str);// 插入一个节点。  FileHandler\* open\_file(char\* file\_name, char\* open\_type) //创建指向文本的指针和内存缓冲区。  void bplus\_destroy\_tree()；//删除一个B+树。  int bplus\_query\_recoder(char \*str) //查询节点。  int write\_line(FileHandler\* file\_handler, char\* str)//把内存中的值写入创建的文本中。  int close\_file(FileHandler\* file\_handler)//关闭文件，释放空间。 |

### 2.1.2 B+树的建立

利用Hash算法查找的核心是建立合理的Hash列表。 Hash列表中的每一个节点包含了，内部的关键词和指向下一个节点的指针。在创建新的Hash列表时，首先要给节点设置空值。然后再根据给定的内容，计算其Hash值，赋值到Hash节点中，然后返回新的节点。其中包含了两种情况，一种是不发生冲突时，节点可以直接设置在原先处；若发生了冲突，节点需要找到这个位置的下一个节点，然后再进行赋值操作。其数据结构设计如下：

|  |
| --- |
| /\* 构造B+树的结构 \*/  typedef struct BPlusTreeNode  {  int isRoot, isLeaf;  int key\_num;  int key[MAX\_CHILD\_NUMBER];  struct BPlusTreeNode \*child[MAX\_CHILD\_NUMBER];  struct BPlusTreeNode \*father;  struct BPlusTreeNode \*next;  struct BPlusTreeNode \*last;  } BPlusTreeNode;  /\* 初始化一个B+树的根 \*/  void bplus\_init\_tree()  {  BPlusTree\_Destroy();  Root = New\_BPlusTreeNode();  Root->isRoot = true;  Root->isLeaf = true;  TotalNodes = 0;  }  /\*创建一个B+树节点 \*/  BPlusTreeNode \*New\_BPlusTreeNode()  {  struct BPlusTreeNode \*p = (struct BPlusTreeNode\*)malloc(sizeof(struct BPlusTreeNode));  return p;  }; |

### B+树节点的插入

B+树的插入操作极其复杂，因为它包含了判断树是否平衡，对树的结构做分类，重新构造树的结构，判断值与根的大小。这些要求导致插入函数复杂，因此我们设计了多个不同功能的函数，一起完成节点的插入步骤。其数据结构如下。

|  |
| --- |
| /\* 插入节点操作 \*/  int bplus\_insert\_recoder(char \*str)  {  return BPlusTree\_Insert(get\_hash(str), str);  }  /\* 根据Hash值，插入\*/  int BPlusTree\_Insert(int key, void \*value)  {  return true;  }  /\*插入函数\*/  void Insert(BPlusTreeNode \*Cur, int key, void \*value){}  /\*如有必要，重新修改B+树的构造\*/  void Split(BPlusTreeNode \*Cur){}  /\*计算Hash值，并插入节点\*/  BPlusTreeNode \*Find(int key){  return Cur;  } |

### B+树节点的查询

B+树的查询操作，需要通过读取string文本中每一行的词，然后计算其Hash值，和当前节点的值进行比较，然后根据比较的结果进入下一个子节点。其数据结构如下。

|  |
| --- |
| /\* 插入节点操作 \*/  int bplus\_query\_recoder(char \*str)  {  return 1;//查找成功  return 0;//查找失败  }  /\*计算Hash值，并寻找节点\*/  BPlusTreeNode \*Find(int key){  return Cur;  } |

### B+树节点的删除

B+树的删除操作，是把根节点释放，整棵树占用的空间就释放了。

|  |
| --- |
| /\* 删除节点操作 \*/  void bplus\_destroy\_tree()  {  //BPlusTree\_Destroy();  printf("destroy b\n");  }  void Destroy(BPlusTreeNode \*Cur)  {  free(Cur);  }  void BPlusTree\_Destroy()  {  if (Root == NULL)  return;  Destroy(Root);  Root = NULL;  } |

### 2.1.6 文件的操作

本实验涉及到了三个文件，其中两个为输入文件dict.txt和string.txt，和一个输出文件result.txt。dict.txt文件是包含了127万个词的文件，而string.txt包含了1.7万个词。result.tx文件中所包含的词包含了输入两个文件的交集。

本实验中，利用的文件操作涉及到了指针。其数据结构设计如下：

|  |
| --- |
| /\* 文件指针的建立 \*/  typedef struct FileHandler {  FILE\* file;//  char file\_name[99];  int open\_status;  int point;  unsigned int buffer\_size;  char\* buffer;  }FileHandler;  /\*创建文件 \*/  FileHandler\* open\_file(char\* file\_name, char\* open\_type) {  return file\_handler;  }; |

**(1) 文件的读取**

本实验利用缓存区读取文件。首先通过全局变量为缓存区设定大小。当缓存区内部填充满时，再进行读取。这样读取的次数就大大减少了。文件读取的数据结构如下：

|  |
| --- |
| /\* 文件的读取 \*/  int read\_line(FileHandler\* file\_handler, char\*\* str) {  return 1；//读取成功  } |

**(2) 文件的写入**

本实验中利用的写入操作有一个特点，即直接通过对缓存区进行写入，当缓存区内部填充满时，再进行写入，直到完全把结果输出到文本中。文件写入的数据结构如下：

|  |
| --- |
| /\* 文件的写入 \*/  int write\_line(FileHandler\* file\_handler, char\* str) {  char target[20];//we have 6306 numbers  static int num = 1;//numbers will change  char\* t = target;  sprintf(target,"%d",num);  return 0；//写入成功  }; |

## 原始256叉树

原始256叉树，又称单词字典树。根节点不存储任何字符，但是其他节点同时包含一个关键字符与指向其子节点的指针。

每次添加节点时，都需要从根节点遍历，直到当前指针指向的节点不存在对应的关键字时，添加子节点，并把对应的关键字赋给该节点，同时为该节点新建其子节点。当关键字已经添加完毕，需要标记该关键字的最后一个字符，表示该关键词的结束点。

同理，每次检索，都从树的根节点开始。当关键字的指针前进一位时，节点就要移动到其对应的子节点。若不存在对应的子节点，则查找失败，跳过，进入到下一个关键词。若查找到了最后一位关键词，仍然需要判断对应的子节点是否有被标记，否则也是查找失败。例如，原始256叉树中的关键词是abd，而我们要查找的是ab关键词，如果b没有被标记，那么我们认为查找失败。查找成功后，需把该关键词写入result.txt文件中。

1、创建带有256个子节点的根，根的值置为0，并且其指针指向第一个孩子

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

2、当传入关键词的第一个字符str[0]，则指向第str[0]个子节点。然后再读取第二个字符str[1],指针指向第str[1]个子节点。

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

3、当关键词全部传入完成，则把最后一个节点的count++，标记为最后一个节点。

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

0

0

0

0

0

0

0

0

0

0

0

… …

… …

0

0

0

0

1

0

0

0

0

0

0

0

… …

… …

图 2 原始256叉树的操作

### 实验思路

为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 初始化一棵原始256叉树，把根的多个值赋值为0。

2. 开始进行插入操作，每读取dict一个关键词，给原始256叉树插入一个节点。

3. 读取完dict文本中的每一行关键词时，为叶子节点添加指针。

4. 读取string文本中的每一行关键词，然后查找。

5. 如果成功查找到，则写入result文本中。

其中涉及到的函数有：

|  |
| --- |
| /\*本实验中用到的函数\*/  void rawtrie\_init\_tree();//初始化一个原始256叉树。  int rawtrie\_insert\_recoder(char \*str);// 插入一个节点。  FileHandler\* open\_file(char\* file\_name, char\* open\_type) //创建指向文本的指针和内存缓冲区。  void rawtrie\_destroy\_tree()；//删除一个原始256叉树。  int rawtrie\_query\_recoder(char \*str) //查询节点。  int write\_line(FileHandler\* file\_handler, char\* str)//把内存中的值写入创建的文本中。  int close\_file(FileHandler\* file\_handler)//关闭文件，释放空间。 |

### 2.2.2 原始256叉树的初始化

创建一个原始的256叉树需要先构造它的结构体，除了根节点，其他每个节点都包含有关键字和子节点。其数据结构如下。

|  |
| --- |
| /\* 原始256叉树的建立\*/  typedef struct TrieNode {  struct TrieNode \*child [256];  int count; //判断是否为关键词的最后一个字符  char value;//该节点的字符  } TrieNode;  /\* 建立根\*/  TrieNode\* rawtrie\_init\_tree(){  return node;  }  /\* 建立节点\*/  TrieNode\* rawtrie\_new\_tree(){  return node;  } |

### 2.2.3 原始256叉树的操作

根据实验要求，需要根据dict.txt文件中的127万个字符生成原始256叉树，并利用string.txt文件查询其中的字符串。所以对原始256叉树的结构包含遍历树的节点，添加新的节点，以及查询和删除。其数据结构如下。

|  |
| --- |
| /\* 插入 \*/  int rawtrie\_insert\_recoder(char \*str){  return 1；//如果插入成功则返回1  return 0；//如果插入失败则返回0  }  /\* 查询 \*/  int rawtrie\_query\_recoder(char \*str){  return 1；//如果查询成功则返回1  return 0；//如果查询失败则返回0  }  /\* 删除 \*/  void rawtrie\_destroy\_tree(){  } |

## Mtrie查找树

Mtrie查找树是对原始256叉树的优化，因为从中文角度来说，分支下的节点数不平衡，每个分支下的节点数差距可以很大。而Mtrie查找树是根据关键码每个二进制位的编码来划分，是对整个关键码大小范围的划分。每个内部结点都代表一个位的比较，必然产生两个子结点，所以它是一个满二叉树，进行一次检索，最多只需要关键码位数次的比较即可。

### 实验思路

本节概括了实验的流程。

- 为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 设置合适的参数M，M决定了mtrie树的结构。（本实验设定M为8）

2. 初始化一棵mtrie树，把根的M个子节点赋为空。

3. 开始进行插入操作，每读取dict一个关键词，给mtrie树插入一个节点。

4. 读取完dict文本中的每一行关键词时，为节点插入标签，记为完成。

5. 读取string文本中的每一行关键词，查找。

6. 如果成功查找到，则写入result文本中。

其中涉及到的函数有：

|  |
| --- |
| /\*本实验中用到的函数\*/  void mtrie\_init\_tree();//初始化一个Mtrie树。  int mtrie\_insert\_recoder(char \*str);// 插入一个节点。  FileHandler\* open\_file(char\* file\_name, char\* open\_type) //创建指向文本的指针和内存缓冲区。  void mtrie\_destroy\_tree()；//删除一个Mtrie树。  int mtrie\_query\_recoder(char \*str) //查询节点。  int write\_line(FileHandler\* file\_handler, char\* str)//把内存中的值写入创建的文本中。  int close\_file(FileHandler\* file\_handler)//关闭文件，释放空间。 |

* + 1. Mtrie查找树的初始化

创建一个原始的M叉查找树需要先构造它的结构体，除了根节点，其他每个节点都包含有关键字和子节点。其数据结构如下。

|  |
| --- |
| /\* 原始256叉树的建立\*/  typedef struct m\_tree\_node {  struct m\_tree\_node \*child(1<<M);  char is\_end;//判断是否为最后一个节点  }MTNode;  /\* 原始256叉树的建立\*/  typedef struct ex\_str{  struct ex\_str \*next;  char \*str;//该节点的字符  }EXStr;  /\* 建立根\*/  void mtrie\_init\_tree(){  root = create\_new\_node();  }  /\* 建立节点\*/  MTNode \*create\_new\_node(){  return node;  } |

### 2.3.3 Mtrie查找树的操作

根据实验要求，需要根据dict.txt文件中的127万个字符生成原始256叉树，并利用string.txt文件查询其中的字符串。所以对原始256叉树的结构包含遍历树的节点，添加新的节点，以及查询和删除。其数据结构如下。

|  |
| --- |
| /\* 插入 \*/  int mtrie\_insert\_recoder(char \*str){  return 1；//如果插入成功则返回1  return 0；//如果插入失败则返回0  }  /\* 查询 \*/  int mtrie\_query\_recoder(char \*str){  return 1；//如果查询成功则返回1  return 0；//如果查询失败则返回0  }  /\* 删除 \*/  void mtrie\_destroy\_tree(){  } |

## Radix查找树

Radix树是基于二进制构建的一个二叉树，在每个节点中都存储有在进行下一次比特测试之前需要跳过的比特数目，以此来避免单路分支。

### 实验思路

本节概括了实验的流程。

- 为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 首先通过dict文本中的第一行和第二行，分别插入第一个和第二个二进制串。

2. 得到他们的公共前缀并把它们的公共前缀独立为新节点，并重置两个结点的内容。

3. 按照节点的最前面的k个比特内容，作为节点的序号。

4. 插入第三个节点，并且得到公共前缀。

5. 将公共前缀独立为单独的节点，同样将其插入到新节点对应的位置即可。

6. 读取string文本中的每一行关键词，然后查找

7. 如果成功查找到，则写入result文本中。

其中涉及到的函数有：

|  |
| --- |
| /\*本实验中用到的函数\*/  void radix\_init\_tree();//初始化一个Radix树。  int radix\_insert\_recoder(char \*str);// 插入一个节点。  FileHandler\* open\_file(char\* file\_name, char\* open\_type) //创建指向文本的指针和内存缓冲区。  void radix\_destroy\_tree()；//删除一个Radix树。  int radix\_query\_recoder(char \*str) //查询节点。  int write\_line(FileHandler\* file\_handler, char\* str)//把内存中的值写入创建的文本中。  int close\_file(FileHandler\* file\_handler)//关闭文件，释放空间。 |

### 2.4.2 Radix查找树的构建

|  |
| --- |
| /\* 新建Radix树 \*/  typedef struct radix\_node  {  char \*str; //节点内的二进制序列  unsigned short len; //节点内的二进制序列有效长度  unsigned char is\_end; //以此节点为结尾的字符串是否出现过  struct radix\_node \*child; //节点的子节点  struct radix\_node \*brother; //节点的兄弟节点  } Radix\_Node;  /\* 创建节点 \*/  Radix\_Node \*create\_radix\_node(char \*str){  return node;  }  void radix\_init\_tree(){  } |

### 2.4.3 Radix查找树的操作

|  |
| --- |
| /\* 插入 \*/  int radix\_insert\_recoder(char \*str){  return 1；//如果插入成功则返回1  return 0；//如果插入失败则返回0  }  /\* 查询 \*/  int radix\_query\_recoder(char \*str){  return 1；//如果查询成功则返回1  return 0；//如果查询失败则返回0  }  /\* 删除 \*/  void radix\_destroy\_tree(){  } |

# 实验分工

按照上述的系统设计，实验分工如下：

高垚实现了M叉树查找，B+树查找和Radix树的查找。

杨恩实现了原始256叉树查找的程序。

# 实验结果与分析

### 4.1 B+树查找树

经过实验，程序共爬取了1270574行字符，总时间为1.054047s，运行内存峰值31M。最终得到6306行字符，程序运行结果如下图3。



图 3 1.1 实验的输出

### 4.2 原始256叉树

经过实验，程序共爬取了1270574行字符，总时间为7.181202s。内存峰值为4324M，程序运行结果如下图4。

### 

图 4 1.2实验的输出

### 4.3 M叉树查找树

经过实验，程序共爬取了1270574行字符，总时间为1.260061s。内存峰值为754M，程序运行结果如下图5。

### 

图 5 1.3 实验的输出

### 4.4 Radix树

经过实验，程序共爬取了1270574行字符，总时间为1.186937s。内存峰值为157M，程序运行结果如下图6。



# 实验问题

图 6 1.4 实验的输出

### 5.1 B+树查找树

**（1）Hash值K发生冲突**

解决方法：在叶子节点之间创建链表，利用指针形式，遍历链表中的关键字（或数据）。

**（2）报错提醒conflicting types for “”**

解决方法：头文件（.h）中与.c程序中的函数的声明不一致，需要改用成一致的声明。

5.2 原始256叉树

**（1）每个节点存在无限个子节点**

解决方法：本算法通过创建一个带有指针的字符，使得该有指针的字符能把每个关键字都安排在0～255的列表中。但是因为编译器默认char类型字符是有正负，因此把它转为整形字符串时，不是我们原以为的0～255，而是-127～128，因此需要把char类型改成unsigned char类型的字符。

5.3 M叉树查找树

**（1）新的str太短，而当前节点还没有达到叶子节点**

解决方法：为每一个节点添加标记，如果当前节点被标记，则表示当前节点及其祖先节点共同保存的数据出现过。

5.4 Radix树

**（1）在进行插入操作时，输入的新str太长时，而当前节点已经遍历完成**

解决方法：分为两种情况，一种是当前节点存在子节点，则在子节点中遍历；另一种是当前节点不存在子节点，则创建新的子节点，把str剩余的位存储在新的子节点中。

# 实验结论

综上所述，实验程序能够对海量数据进行筛选和搜索，并能在较快时间和满足需求的存储空间内完成计算。实验可以通过对数据文本的遍历，建立多种树形数据结构，达到查询的目的，顺利完成实验目标，符合实验预期结果。