

**课 程 设 计 报 告**

**题目： 基于查找表的单词检索软件**

**课程名称： 数据结构课程设计**

**专业班级： CS1404**

**学 号： U201413730**

**姓 名： 张辉**

**指导教师： 袁凌**

**报告日期： 2017-2-11**

**计算机科学与技术学院**

**任务书**

* **设计内容**

设计与实现静态查找表、基于二叉排序树的动态查找表及HASH表等三种中的任意两种查找表结构及其抽象数据类型；以一本英文电子书（如英文小说，科普读物或圣经之类的社会书籍，书的篇幅不少于2万次单词）作为单词文本数据来源，使用上述查找表ADT，通过读取电子书而建立对应的两种查找表，以单词作为关键字，单词在书籍中出现的次数及每次出现的页码，行号等信息作为查找表数据元素属性；通过理论与实际测试结果对比分析两种查找表性能。

* **设计要求**

1. 静态查找表ADT要求实现Create、Destroy、Search、Traverse等操作，另外静态查找表同时要求采用某种排序算法Sort对其排序，形成无序存储与有序表存储两种物理存储，并同时实现在有序表上的二分查找Search\_Bin，且作为性能对比分析的一种情形。
2. 动态查找表ADT要求实现InitDSTable、DestroyTable、SearchDSTable、InsertDSTable、DeleteDSTable、TraverseDSTable等操作，以二叉链表为物理存储结构。
3. HASH表要求实现InitHash、DestroyHash、SearchHash、InsertHash、DeleteHash、TraverseHash等操作，设计合理的HASH函数与冲突解决办法，并在报告中分析说明选择理由。
4. 一个单词的多次出现可以由链表表示，形成类似于倒排索引的结构。查找表数据以文件形式保存，如果在程序重启动时能够从查找表文件恢复查找表，或界面友好，或增加了有意义的功能等，具有一定特色，则给予鼓励，酌情加分。但是，如果只实现了其中一种查找表，则综合成绩不超过75分。

**目录：**

[1问题描述 3](#_Toc475172155)

[1.1课程设计的意义 3](#_Toc475172156)

[1.2课程设计的主要工作 3](#_Toc475172157)

[2系统需求分析与总体设计 4](#_Toc475172158)

[2.1系统需求分析 4](#_Toc475172159)

[2.2系统总体设计 4](#_Toc475172160)

[3系统详细设计 6](#_Toc475172161)

[3.1有关数据结构的定义 6](#_Toc475172162)

[3.2主要算法设计与时间复杂度分析 8](#_Toc475172163)

[4系统实现与测试 15](#_Toc475172164)

[4.1系统实现 15](#_Toc475172165)

[4.2系统测试 21](#_Toc475172166)

[5课程设计总结 22](#_Toc475172167)

[5.1全文总结 22](#_Toc475172168)

[5.2优点与不足 22](#_Toc475172169)

[5.3体会 22](#_Toc475172170)

[参考文献 23](#_Toc475172171)

[附录 24](#_Toc475172172)

1问题描述

1.1课程设计的意义

在日常生活、程序设计中经常使用到查找操作。本次课程设计通过各种基本的数据结构实现了基于链表、数组、二叉查找树、HASH表的查找表，并在此基础上解决了单词文本中单词出现位置检索的问题。

通过设计与实现基于链表、数组、二叉查找树、HASH表的数据结构的单词检索软件，加深对相关数据结构知识的理解和应用，并在实际问题中灵活的根据数据结构设计相应的算法，增强和提高综合分析问题与解决问题的能力。

1.2课程设计的主要工作

* 基于单链表的单词检索
* 基于静态查找表(数组)的单词检索
* 基于二分查找树的单词检索
* 基于HASH表的单词检索

2系统需求分析与总体设计

2.1系统需求分析

* 单链表实现以下功能:

初始化链表、添加单词节点、添加单词节点位置信息、遍历链表、销毁链表、查找单词位置信息

* 静态查找表(数组)实现以下功能:

创建静态查找表、对静态查找表进行扩容、向静态查找表中添加单词节点、向静态查找表中添加单词节点的位置信息、遍历查找表、基于单词的字典序对查找表进行快速排序、对没有进行排序的静态查找表进行暴力搜索、对进行过排序的静态查找表进行二分查找、销毁静态查找表、保存数据到索引文件中、从索引文件中读取数据重建静态查找表

* 二分查找树实现以下功能:

初始化二分查找树、向二分查找树添加单词节点、中序遍历二分查找树、在二分查找树中搜索单词、销毁单词节点、销毁二分查找树、将二分查找树保存到文件中

* HASH表实现以下功能:

初始化Hash表、bkdrHash函数、向HASH表中插入单词节点、在HASH表中查找单词、删除单词节点、销毁HASH表、遍历HASH表，将HASH表数据保存到索引文件中，从索引文件中恢复HASH表

四种方式独立开来进行设计，最后每种数据结构设计的单词检索软件需要根据运行程序提供的参数进行操作，如使用HASH表，运行./main –s bible读入数据文件并创建HashTable,然后响应查找输入，当用户结束查找时，返回本次查找过程平均耗时， 随后将数据保存到倒排索引文件中方便下次直接从该文件中输入(./main –i index.dat[直接从index.dat恢复数据])，最后销毁掉查找表。

2.2系统总体设计

1. 命令行参数的设计

运行程序时传入参数，参数规则如下(表2-1)

表2-1命令行参数

|  |  |
| --- | --- |
| -s file | 从原始文件中读取数据 |
| -i file | 从索引文件中读取数据 |
| -u | 不进行排序,使用暴力搜索(在静态查找表中) |
| -h | 查看帮助 |

使用如: ./main –s bible –u (从原始文件bible中读取数据，并使用无序存储)

1. 总体结构

各种方式设计的单词查找软件功能相似，总体设计以静态查找表为例。如图2-2

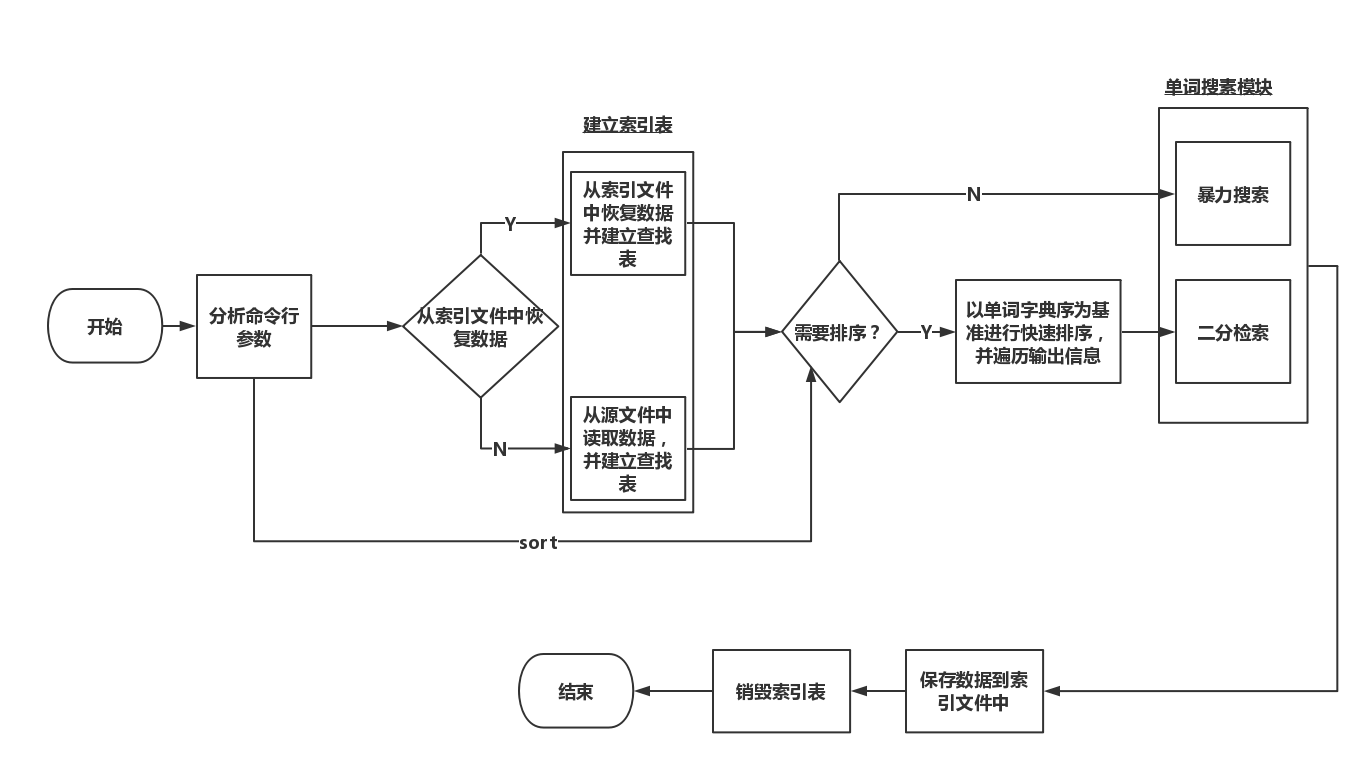


图2-1 程序总体结构图

3系统详细设计

3.1有关数据结构的定义

程序中处理的数据为: 英文文本数据中单词字符串、单词的位置(行号和行内偏移量)。其中单词字符串可以使用C++中的string数据类型表示，单词出现的行号和出现的位置使用unsigned int类型表示。

1. 链式查找表

单词的位置信息组成一个位置链表，同时为每种单词创建一个单词节点(包括单词本身字符串，单词出现位置的链表，和指向下一个单词节点的指针)，通过单链表的形式将这些节点组织起来，形成了一个链式查找表。

typedef struct Position{

unsigned int line; //行号

unsigned int offset;

struct Position \*next; //下一个位置

}Position;

typedef struct Node{

char word[32]; //单词

Position pos; //位置链表

struct Node \*next; //指向下一个表项

}Node, \*List;

1. 静态查找表(数组)

单词的位置信息组成一个链表，同时为每种单词创建一个单词节点(包括单词本身的字符串，单词出现位置的链表)，最后通过一个动态数组将这些单词节点组织起来，形成一个静态查找表。

//位置链表

typedef struct Position{

int line;

int offset;

struct Position \*next;

}Position;

typedef struct Node{

string word;

Position pos;

}Node;

//静态查找表

typedef struct Table{

Node \*data;

int capacity;

int cnt;

}Table;

1. 基于二分查找树

单词的位置信息组成一个单链表，为每种单词构造一个结点(其中包括单词本身字符串，位置链表、左孩子节点指针、右孩子节点指针、父亲节点指针)，最后通过指针之间的相互链接形成了一棵二分查找树

typedef struct Position{

unsigned int line;

unsigned int offset;

Position \*next;

}Position;

typedef struct Node{

string word;

Position pos;

struct Node \*lchild;

struct Node \*rchild;

struct Node \*parent;

}Node, \*Bstree;

1. 基于HASH表

单词的位置信息组成一个单链表，为每种单词构造一个结点(其中包括单词本身字符串，位置链表，指向下一个节点指针[用于解决hash地址冲突])，使用一个节点指针数组组织单词节点，通过单词HASH函数值与存储位置对应从而形成一个HASH表。

const int TABLE\_SIZE = 19997;

/\*位置链\*/

typedef struct Position{

unsigned int line;

unsigned int offset;

struct Position \*next;

}Position;

/\*单词结点\*/

typedef struct Node{

string word;

Position pos;

struct Node \*next;//链地址解决冲突

}Node, \*pNode;

/\*hash表\*/

typedef struct HashTable{

int size;

pNode \*data;

}HashTable;

3.2主要算法设计与时间复杂度分析

分析程序中主要的函数模块使用的算法和相应的时间复杂度，包括从源文件中建立相应数据结构、查找单词模块、快速排序算法、保存数据到索引文件中、从索引文件中恢复数据结构、销毁相应数据结构等。算法伪代码表示。具体细节详见附录

1. **链式查找表**
   1. 将单词数据插入到链表中，伪代码如下

**addList 将单词信息加入到单链表中**

**@in: 链表L, 单词word, 行号line, 行内偏移offset**

**@out: 更新单词结点信息之后的链表L**

结点指针p指向结点的头结点

while p不是最后一个结点 and p指向结点的单词不是word

p指向链表下一个结点

if p指向最后一个结点 then

if p指向结点单词数据不是word then call addword(p, word, line, offset);

else call addpos(p, line, offset)//更新p结点中位置链表内容

else

call addpos(p, line, offset)//更新p节点中位置链表内容

时间复杂度：程序在每次添加节点的时候都要进行一个单链表的遍历，最坏情况下，每次添加节点或者更新位置的位置都在链表的尾部，程序输入为n个单词数据，addpos、addword操作的时间复杂度为，因此addList最坏情况下时间复杂度

* 1. 从源文件中读取数据，伪代码如下

**build\_dictionary**

**@in: 链表L(已经初始化了), 源文件文件名 srcfile**

**@out: 填充单词节点数据的链表L**

使用读模式打开源文件

line = 1

while 没有读取完 do

读取一行数据到buf中， line = line + 1

call word\_token(L, buf, line)//分离出单词并建立相应数据结构

关闭源文件

end

**word\_token 从一整行数据中分离出单词并建立相应数据结构**

**@in: 链表L, 字符串str，行号line**

**@out: 向链表中填充相应单词节点信息**

call strtok(str, " \":,.?!-\\(\\)';1234567890")除掉非单词符号

使用字符串指针p指向返回的第一个单词

offset = 0

while p 不是空字符串指针

向屏幕上打印当前p指向的单词

将p指向的单词改为小写

call addList(L, p, line, offset)更新链表数据

offset = offset + 1

call strtok(NULL, " \":,.?!-\\(\\)';1234567890")

p指向下一个单词

end

时间复杂度：程序将读取文本文件的每一个单词并调用addList将单词信息加入到链表中，其中addList的时间复杂度为，因此build\_dictionary时间复杂度

1. **静态查找表**
   1. 将单词节点数据添加到静态查找表中，伪代码如下

**addSqList(T， word, line, offset)//向静态查找表中添加一个单词节点**

**@in: 静态查找表T， 单词word, 单词出现的行号line, 行内偏移offset**

**@out: 更新单词信息的静态查找表T**

if 查找表长度 < 单词节点个数 then call expand(T)进行扩容

for i = 0 to T.cnt step 1 do

if word == T.data[i].word break

if i = T.cnt then

新建一个单词节点

else 更新T.data[i]的位置信息

时间复杂度：与链式查找表类似，时间复杂度为

* 1. 保存数据到索引文件，伪代码如下

**saveIndexFile(T, filename)//保存数据信息加入到索引文件中**

**@in: 静态查找表T, 文件名filename**

**@out: 填充信息到filename命名的文件中去**

以写模式打开文件filename

for i = 0 to 查找表长度 step 1 do

写入T中第i个数据项的word, 换行

p指向节点位置链首元结点的下一个结点

while p do

写入p指向数据项的line和offset，格式为 "line\_offset"

写入+

p指向下一个结点

写入位置链首元结点数据项的line和offset，格式为"line\_offset"

关闭文件

时间复杂度：程序遍历查找表中每一个单词结点，时间复杂度为

* 1. 从索引文件中恢复数据结构，伪代码如下

**getTableFromIndex(T, f\_index)//从索引文件中恢复数据结构**

**@in: 静态查找表T, 倒排索引文件名f\_index**

**@out: 填充数据的查找表 T**

读模式打开索引文件f\_index

while 没有读取到文件尾 do

读取一行数据到buf中

call addSqList(T, buf)添加单词节点【注意这里是重载的addSqList与前面不同】区别在于这里直接确定一个单词节点，但是没有添加任何位置信息

将buf数据打印到屏幕上显示

call addPosition(T.data[T.cnt] -1, buf)//从buf中分离位置信息，更新节点位置信息

关闭文件

时间复杂度分析：程度从文件中读取一整行数据速度较快，时间主要耗在调用函数addSqList和addPosition上，且addPosition需要分析buf的每一位数据将字符串数据转换为数字信息，耗时较大。

* 1. 单词查找主模块，伪代码如下

**searchDictionary(T, sort)**

**@in: 静态查找表T, 查找表有序标志sort[1表示已经排序，0表示未排序]**

**@out: 查找平均耗时**

cnt = 0//查找次数

sumTime = 0//查找总耗时

begin = 0, end = 0//用于记录时间

提示用户输入查找的单词(空行结束输入),记录到search\_word中

获取一行输入数据

while 输入的不是空行 do

cnt = cnt + 1//记录查找次数

begin = clock()//记录查找开始时间

if 有序 then

search\_result = call bin\_search(T, search\_word)//二分查找

else

search\_result = call search\_unsort(T, search\_word)//暴力查找

end = clock()//记录查找结束时间

sumTime += end - begin//记录总时间

if search\_result 为空 then 打印没有找到

else call display\_result(search\_word) 展示查找结果

if cnt = 0 then return -1//一次也没有查找

else return sumTime / cnt//返回平均耗时

时间性能分析：主要是一些与用户交互的代码，主要时间消耗在bin\_search或search\_unsort上

* 1. 暴力查找算法

**search\_unsort(T, word)//在静态查找表中使用暴露搜索单词**

**@in: 静态查找表T， 需要查找的单词**

**@out: 查找到返回对应的单词节点指针，没有找到返回NULL**

for i = 1 to T.cnt step 1 do

if 查找表中单词节点=word then 返回对应的节点指针

返回NULL

时间复杂度分析：暴力查找需要进行一次遍历，最坏时间复杂度为

* 1. 快速排序算法，伪代码如下：

**quicksort(data, lo, hi)**

**@in: 指向节点数组的指针data[Node \\*], 排序元素范围[lo,hi)**

**@out: 按照单词的字典序排号序的数组 data[lo,hi)**

if hi - lo < 2 then return

else

pivot = call partiction(data, lo, hi)

quicksort(data, lo, pivot)

quicksort(data, pivot + 1, hi)

//以节点数组第一个元素为划分元素，将数组元素data[lo, hi)分为左右两部分，

//左边小于划分元素,右边大于划分元素

**partiction(data, lo, hi)**

**@in: 节点数组data[lo, hi)**

**@out: 划分元素位置pivot**

i = lo, j = hi - 1

key = data[lo]

while i < j do

while i < j && key.word <= data[j].word do i = i + 1

if key.word > data[j].word then data[i] = data[j]

while i < j && data[i].word <= key.word do i = i + 1

if key.word < data[i].word then data[j] = data[i]

data[i] = key

return i

时间复杂度分析：划分算法需要移动前后指针，将每一个单词节点word数据项与基准元素比较，时间复杂度为。对于快速排序，最好情况下元素划分的比较均匀，用表示所需时间，递归表达式为，由中心定理知时间复杂度为，对于一般情况假设pivot在第k个元素位置上，有，平均时间复杂度，解递归方程可知，平均时间复杂度。当然算法的最坏情况是数据已经是有序的时候，此时pivot=0，递归表达式退化为，最坏情况时间复杂度为。考虑到数据来源为英文文本文件，算法以单词的字典序为基准，其有序性很差，因此该排序算法可以接受。

* 1. 二分查找算法，伪代码如下

**bin\_search(T, word)//二分查找算法**

**@in: 静态查找表T， 需要查找的单词word**

**@out: 找到则返回对应的节点指针，否则返回NULL**

lo = 0, hi = T.cnt

while lo < hi

mid = lo + ((hi - lo) >> 1)

if mid中单词项对应word return &T.data[mid]//找到

else if mid中单词项 < word then lo = mid + 1

else hi = mid

return NULL //没有找到

时间复杂度分析：每次循环都可以减少一半的搜索空间，时间复杂度为

1. **二分查找树**
   1. 添加一个结点信息到二分查找树中

**insertBST(T, word, line, offset)//将一个数据项加入到二分查找树**

**@in: 二分查找树树根指针T, 单词word, 单词位置信息行号line, 行内偏移offset**

**@out: 更新数据之后的二分查找树**

if 不是空树 then

if T指向节点word数据项 < word then

if T的右孩子不为空 then 递归调用insertBST(T->rchild, line, offset)

else 建立一个结点插入到T的右孩子位置上

else if word < T 指向的节点word数据项 then

if T的左孩子不为空 then 递归调用insertBST(T->lchild, line, offset)

else 建立一个结点插入到T的左孩子位置上

else 将line\_offset加入到T指向节点的位置链上

else //空树

新建立一个结点，并让T指向之

时间复杂度分析：对于文本中的n个单词，最坏情况出现在输入单词有序，此时二分查找树退化为链表，时间复杂度为。但在一般情况下，输入单词有序性很差，二分查找树比较均匀，可以理想为一颗完全二叉树，对于深度为i的节点来说，其进行字符串比较操作为i次，时间复杂度可以表示为，相对于链式查找表和静态查找表建表时间有明显的提高。

* 1. 单词节点的删除操作

**deleteNode(pNode)//删除特定节点**

**@in: 指向需要删除的节点的指针**

**@out:**

if pNode有左孩子 then

if pNode有右孩子 then

//寻找pNode中序遍历对应的直接后续节点,即pNode右子树左边链最底的那个节点

p = call findMinInTree(pNode->rchild)

将p的数据项覆盖掉pNode指向的节点数据项

call deleteNode(p)//调用调用自身删除节点p

else then//左孩子不空，右孩子空

将pNode->lchild中数据项覆盖掉pNode指向的节点的数据项

delete pNode->lchild, pNode->lchild = 0

else

if pNode 有右孩子 then//左孩子空，右孩子不空

将pNode->rchild中数据项覆盖掉pNode指向的节点的数据项

delete pNode->rchild, pNode->rchild = 0

else//左右孩子都是空的

更新pNode父节点的指针域

delete pNode, pNode = 0

else

时间复杂度分析：当pNode的左右孩子节点有一个为空，或者都为空的时候，只需要的时间完成删除，当pNode为内节点的时候，需要遍历右子树的左部链，遍历时间对应为，表示pNode的高度，最坏情况为删除根节点，时间复杂度, 这比链式查找表删除有了很大提高。

* 1. 单词查找算法

**searchBST(T, word)**

**@in: 二分查找树， 待查找单词word**

**@out: 找到返回单词节点指针，否则返回0**

if T不为空 then

if T指向节点word项 == word then 返回 T

else if T指向节点word项 < word return call searchBST(T->rchild, word)

else return call searchBST(T->lchild, word)

else

return 0

时间复杂度分析：当二叉查找树比较均匀的时候，类似与二分查找，时间复杂度为

1. **HASH表**
   1. bkdrHash函数, [by Kernighan and Dennis]，伪代码如下

**bkdrHash(word)**

**@in: 字符串word**

**@out: hash函数值**

if word.size() == 0 then return 0//空字符串直接返回

seed = 131

i = 0

while i < word.length do

hash = hash \* seed + word[i], i++

return hash % TABLE\_SIZE

性能分析: 该hash函数的数学计算表达式为，其中has6h表示散列的地址，string[i]表示输入字符串的第i位，seed表示基数，类似16进制中的16，length为该字符串的长度。该hash函数使字符串中每个字符都参与形成了哈希值计算，从而当改变字符串中的一个字符，对最终的哈希值也会造成很大影响。减少了非常相像的字符串聚集的可能性，从而减少了冲突的概率。同时seed的选取为一个质数，同时hash表的大小TABLE\_SIZE为一个素数19997，使用其他方法已经测出单词节点总共为13687，从而该hash表的装填因子为

* 1. 向hash表中插入数据

**insertHashTable(T, word, line, offset)**

**@in: 哈希查找表T, 插入单词word, 单词位置line, offset**

**@out: 更新单词信息的哈希查找表**

hash = bkdrHash(word)//得到可能存储位置

if T.data[hash] 为空槽

T.data[hash] = createNode(word, line, offset)

else

遍历这个槽解决冲突使用的链

if 找到word同名的单词节点 then 更新这个节点的位置链

else call createNode创建一个结点，加入到冲突槽

时间复杂度分析：插入第n个元素可以类比为在表中装有n-1个记录的查找，因此时间复杂度和查找一致。

* 1. **查找**

**searchHashTable(T, word)**

**@in: 哈希查找表T, 待查找单词word**

**@out: 如果找到返回对应单词节点的指针，否则返回空**

hash = bkdrHash(word)

if T.data[hash]为空槽 then return 0

else

遍历T.data[word]的冲突链

if 找到待查找单词节点word then 返回其地址p

else return 0

时间复杂度分析：使用链地址解决冲突，查找成功的时候，平均查找长度，由于表示一个常数，因此平均时间复杂度可以表示为

4系统实现与测试

4.1系统实现

1. 开发环境，见下表4-1

表4-1开发环境

|  |  |
| --- | --- |
| 文本编辑器 | Vim7.4 |
| 编译器 | g++ 5.4.0 |
| 调试器 | gdb 7.11.1 |
| 操作系统 | Ubuntu 16.04.1 LTS、Linux4.4.0 |
| 编译链接 | 见附录Makefile文件 |

1. 函数说明

* 基于链式查找表的单词检索软件，函数说明见表4-2

表4-2 链式查找表涉及到的函数清单

|  |  |
| --- | --- |
| 初始化链式查找表 | void initList(List \*T); |
| 在单词节点P后面添加一个单词节点 | void addword(Node \*p, char word, unsigned int line, unsigned int offset); |
| 向单词节点P中添加位置信息 | void addpos(Node \*p, unsigned int line, unsigned int offset); |
| 将单词信息写入链式查找表 | void addList(List T, char \*word, unsigned int line, unsigned offset); |
| 遍历链式查找表 | void traverse(List T, char \*filename, void (\*visit)(Node \*, char \*)); |
| 在链式查找表中查找单词并显示结果 | void search\_word(List T, void(\*display)(Node \*)); |
| 将字符串全部改成小写 | char \* change\_to\_lower(char \*str); |
| 从一行字符串中提取出单词，并写入到链式查找表 | void word\_token(List L, char \*str, unsigned line); |
| 从文件建立查找表 | void build\_dictionary(List L, char \*srcfile); |

* 基于静态查找表的单词检索软件，函数说明见表4-3

表4-3 静态查找表涉及到的函数清单

|  |  |
| --- | --- |
| 在静态查找表中循环搜索单词，并返回平均查找时间 | double searchDictionary(Table &T, int sort)  【空行结束】【sort表示查找表是否有序，有序使用二分查找，否则为暴力查找】 |
| 从源文件建表 | void getTableFromSrc(Table &T, char \*f\_src); |
| 从索引文件建表 | void getTableFromIndex(Table &T, char \*f\_index); |
| 从buf中提取出位置信息，并写入到节点node位置链 | void addPosition(Node &node, string buf); |
| 分离出单词信息，并写入到查找表中 | void word\_token(Table &T, char \*str, unsigned int line); |
| 转换字符串为小写 | char \* change\_to\_lower(char \*str); |
| 建立静态查找表 | void createSqList(Table &T); |
| 查找表扩容 | void expand(Table &T); |
| 添加单词信息到静态查找表 | void addSqList(Table &T, string word, unsigned int line, unsigned int offset); |
| 添加单词到静态查找表 | void addSqList(Table &T, string word); |
| 遍历查找表 | void traverse(Table &T); |
| 暴力查找 | Node \* search\_unsort(Table &T, string word); |
| 快速排序 | int partiction(Node \*data, int lo, int hi);  void quickSort(Node \*data, int lo, int hi); |
| 二分查找 | Node \* bin\_search(Table &T, string word); |
| 保存数据到索引文件 | void saveIndexFile(Table &T, char \*filename); |
| 显示查找结果 | void display\_result(Node \*node); |
| 销毁链表 | void destorySqList(Table &T); |

* 基于二叉查找树的单词检索软件，函数说明见表4-4

表4-4 二叉查找树涉及到的函数清单

|  |  |
| --- | --- |
| 从源文件中建表 | void getTableFromSrc(Bstree &T, char \*f\_src); |
| 从索引文件建表 | void getTableFromIndex(Bstree &T, char \*f\_index); |
| 在BST中循环搜索单词，并返回平均查找时间 | double searchDictionary(Bstree &T); |
| 分离出单词信息，加入到查找表中 | void word\_token(Bstree &T, char \*str, unsigned int line); |
| 将字符串转换为小写并返回 | char \* change\_to\_lower(char \*str); |
| 初始化BST | void initBST(Bstree &T); |
| 将单词信息加入BST | void insertBST(Bstree &T, string word, unsigned int line, unsigned int offset); |
| 中序遍历BST | void inOrderBST(Bstree &T);/\*中序遍历binary search tree\*/ |
| 在BST中查找单词 | Node \* searchBST(Bstree &T, string word); |
| 新建一个单词节点 | Node \* createNode(string word, unsigned int line, unsigned int offset); |
| 添加位置信息到节点pNode上 | void addPosition(Node \*pNode, unsigned int line, unsigned int offset); |
| 找节点的直接后继节点 | void addPosition(Node \*pNode, unsigned int line, unsigned int offset); |
| 在屏幕上展示单词位置信息 | void display\_result(Node \*node); |
| 根据word,在BST中查找出单词节点并删除 | void deleteBST(Bstree &T, string word); |
| 销毁BST | void destoryBST(Bstree &T); |
| 将BST保存到索引 | void saveIndexFile(Bstree &T); |
| 删除节点和位置链 | void destoryPosition(Position &pos);  void destoryNode(Node \*pNode); |

基于哈希查找表的单词检索软件，函数说明见表4-5

表4-5 哈希查找表涉及到的函数清单

|  |  |
| --- | --- |
| 初始化HASH表 | void initHashTable(HashTable &T); |
| 生成HASH地址 | unsigned int bkdrHash(string word); |
| 生成一个单词节点 | pNode createNode(string word, unsigned int line, unsigned int offset); |
| 添加位置信息到节点的位置链 | void addPosition(pNode node, unsigned int line, unsigned int offset); |
| 将单词信息加入到HASH表 | void insertHashTable(HashTable &T, string word, unsigned int line, unsigned int offset); |
| 在HASH表中查找单词word | pNode searchHashTable(HashTable &T, string word); |
| 显示查找结果 | void displayResult(Node \*node); |
| 在HASH表中删除单词word | void deleteHash(HashTable &T, string word) |
| 销毁HASH表 | void destoryHashTable(HashTable &T); |
| 将HASH表数据保存到索引文件中 | void saveIndexFile(HashTable &T, char \*filename); |
| 在HASH表中循环搜索单词，并返回平均查找时间 | double searchDictionary(HashTable &T); |
| 从源文件中建表 | void getTableFromSrc(HashTable &T, char \*f\_src); |
| 从索引文件中建表 | void getTableFromIndex(HashTable &T, char \*f\_index); |
| 从源文件一行数据中分离单词，并插入到HASH表中 | void word\_token(HashTable &T, char \*str, unsigned int line);  char \* change\_to\_lower(char \*str); |
| 从索引文件一行字符串中提取位置信息，并加入到单词节点中 | void addAllPosition(HashTable &T, int hash, string word, string buf); |

1. 函数之间的调用关系

* 基于链式查找表的单词检索软件，函数调用关系见图4-1

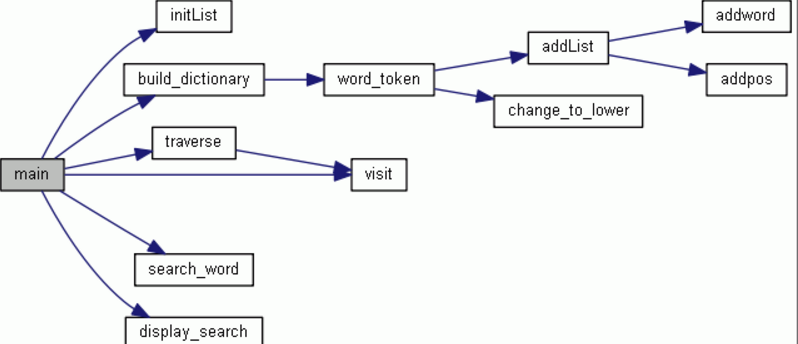


图4-1 基于链式查找表的单词检索软件函数调用关系图

* 基于静态查找表的单词检索软件，函数调用关系见图4-2

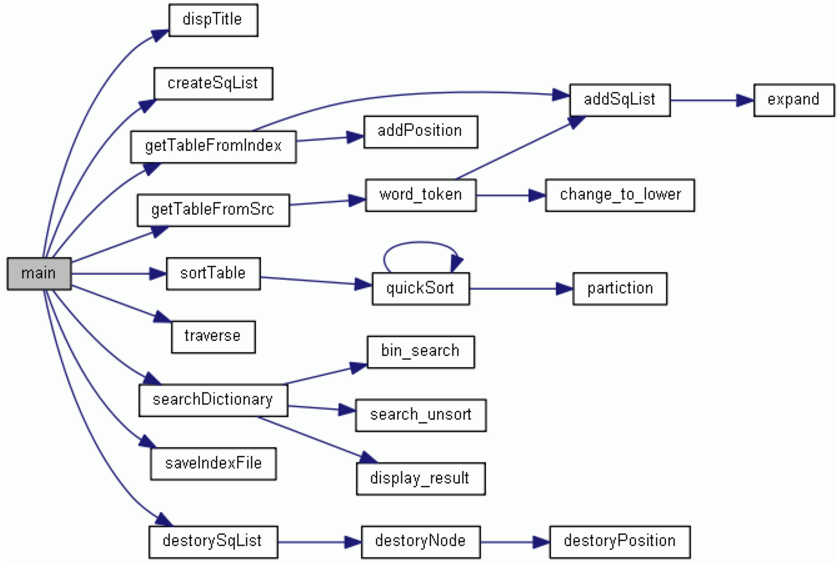


图4-2 基于静态查找表的单词检索软件函数调用关系图

* 基于BST的单词检索软件，函数调用关系见图4-3

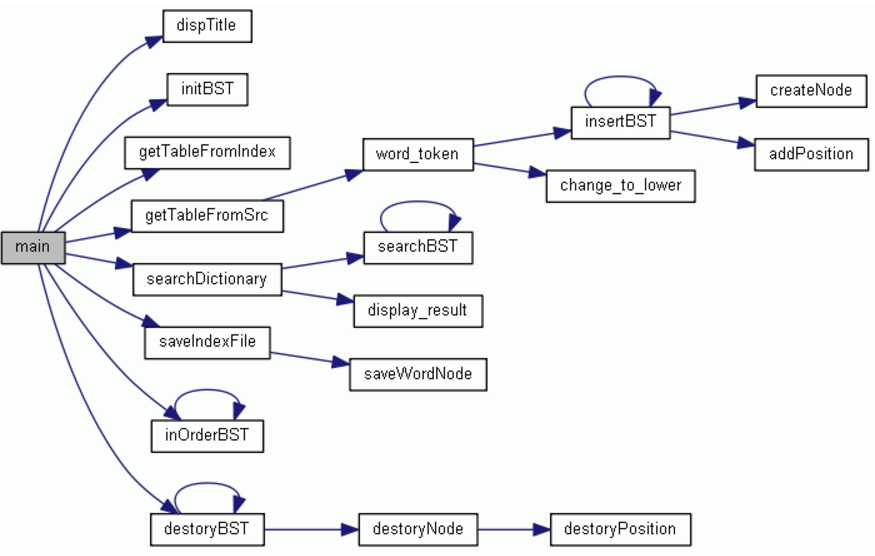


图4-3 基于BST的单词检索软件函数调用关系图

* 基于HASH表的单词检索软件，函数调用关系见图4-5

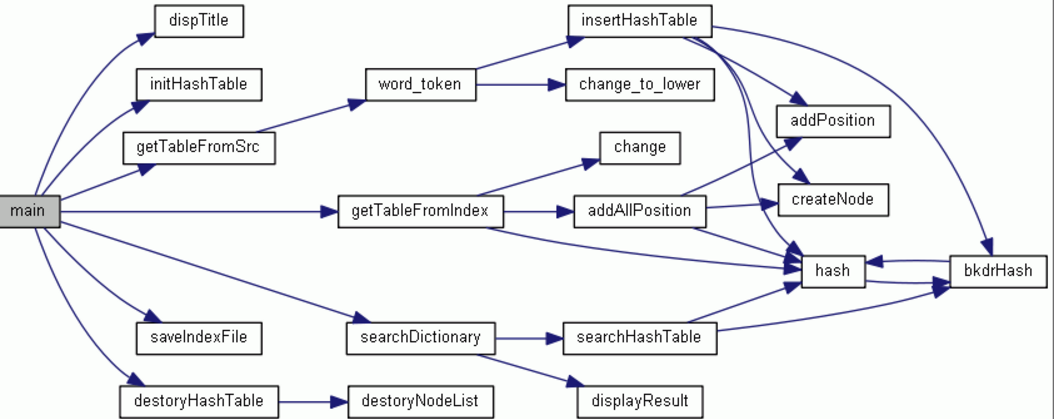


图4-3 基于HASH表的单词检索软件函数调用关系图

4.2系统测试

测试了各种方式查找性能，并和理论分析进行了对比，测试数据为：book、amos、friend、life、god、bible、mom、things、lord、computer、math、city、zombie、power、death、emace、houce、idea、jack、kit共20个随机的单词数据，使用这些数据查找的平均时间和算法的平均时间复杂度进行对比，见表4-6

表4-6 查找性能对比分析表

|  |  |  |  |
| --- | --- | --- | --- |
| **查找算法** | **实际平均耗时** | **平均时间复杂度** | **截图** |
| 暴力查找无序的静态查找表 | **232.95ms** |  | **H:\无序.png** |
| 二分查找有序的静态查找表 | **11.2ms** |  | **H:\有序.png** |
| 在BST中进行查找 | **13.1ms** |  | **H:\BST.png** |
| 在HASH表中进行查找 | **8.8ms** |  | **H:\hash.png** |

5课程设计总结

5.1全文总结

1. 通过四种不同的存储方式实现了单词检索软件，分别为链式查找表、静态查找表（有序和无序两种方式的存储形式）、二分查找树、哈希表
2. 实现了从源文件中读取数据和倒排索引文件中恢复数据
3. 支持向程序中传递参数

5.3体会

这次课程设计，通过实际检测了各种查找方式的时间性能，使我对书本上查找一部分有了更加深刻的认识。

参考文献

[1]严蔚敏,吴伟民等.数据结构(C语言版).清华大学出版社

附录

源代码清单：