
《实验三：用动态查找表实现集合》

实验报告

工作分工	姓名	学号
哈希编码实现	xxkky2	
键树编码、测试	AUV888	
报告撰写	xxkky2 AUV888	

2025 年 12 月 27 日

目录

题目：用动态查找表实现集合.....	1
(一) 问题描述：	1
(二) 设计描述.....	1
2.1 算法思路	1
2.2 数据存储结构类型的定义	1
2.3 hash 核心程序处理流程.....	2
2.4 Trie 核心程序处理流程.....	3
2.5 各模块调用关系	4
(三) 测试设计	4
3.1 测试	4
3.2 环境说明	5
(四) 心得体会	5
(五) 提交材料说明	7

题目：用动态查找表实现集合

(一) 问题描述：

本实验要求分别基于哈希表和 Trie 树这两种动态查找表结构，实现对字符串集合的基本操作。集合中的元素为长度不超过 20 个字符的字符串（字符限定为英文小写字母，实际实现中支持大小写自动转换）。需实现的七种集合操作包括：初始化集合（InitSet）、销毁集合（DestroySet）、插入元素（SetInsert，若元素已存在则不重复插入）、删除元素（SetErase，若元素不存在则无影响）、清空集合（SetClear）、查找元素（SetFind）以及获取集合大小（SetSize）。要求两种实现均能高效地完成这些操作，并在 50 万数据测试中统计相关性能指标。

输入为长度小于等于 20 的字符串，字符为英文小写字母（支持大小写自动转换）；返回值整型结果或元素数量。

(二) 设计描述

2.1 算法思路

哈希表能够利用哈希函数的映射能力，将字符串快速定位到固定大小的桶数组中，然而面对大规模数据，哈希冲突是难以避免的，我们采用链地址法处理哈希冲突，在每个桶位置维护一个链表，当多个元素哈希到同一位置时，将它们链接起来形成链表。这种方法实现简单，且能有效处理频繁冲突的情况。

Trie 树的实现则基于字符串前缀匹配的原理，构建一棵多叉树结构。树的每个节点代表一个字符，从根节点到叶子节点的路径构成一个完整的字符串。为了降低内存占用和加速查找，我们采用兄弟-孩子节点表示法，每个节点只需存储指向兄弟节点和子节点的指针。

2.2 数据存储结构类型的定义

我们对 hash 表的定义如下（在 hash.h 中）：

```
typedef struct HASH_NODE
{
    char *text;
```

```

    struct HASH_NODE *next;
} node;

typedef struct HSET
{
    node *bucket[TABLE_MAX_SIZE];
    unsigned long size;
} hset;

```

在这个结构体中，`bucket` 是一个含有 `TABLE_MAX_SIZE`（通过宏定义为 `0x80000`，即 `524288`）个元素的数组，每个元素指向节点。节点内有一个 `char*` 指针指向字符串和一个指向下一个节点的指针。

我们对 `trie` 的结构定义如下（定义在 `trie.h` 中）：

```

typedef struct TNODE
{
    char d;
    struct TNODE *sib;
    struct TNODE *next;
} tnode;

typedef tnode *tset;

```

`trie` 树的结构就是普通的孩子-兄弟树。其中，数据是倒序储存的，以方便模拟真实场景中的含通配符的 `DNS` 查询。例如 `www.google.com` 将被储存为 `m-o-c-.e-l-g-o-o-g-.w-w-w-\0'`

2.3 hash 核心程序处理流程

哈希函数的设计采用了多阶段处理：首先将字符串按 4 字节分组进行异或运算，然后将每个字节压缩为 5 位，再组合成 20 位的中间结果，最后通过三次位移和异或操作（雪崩效应）增加哈希值的随机性，旨在降低冲突概率并提高分布均匀性。

在初始化集合操作中，我们使用 `malloc` 为 `data` 分配相应的储存空间，并将 `size` 设为 0。

在销毁集合操作中，我们使用 `free` 将 `bucket` 的每个节点储存空间释放，接着使用 `free` 释放 `hset`，再将指向 `hset` 的指针置为空。

在增添元素的操作中，我们在排除了错误情况后，先在哈希表中查找该元素是否存在。如果不存在的话，我们创造一个新的节点，然后使用头插法将其连接到哈希表上并将 `size` 自增 1。

在删除元素的操作中，与增添元素类似，我们通过哈希函数找到其所在的桶，然后遍历所有同义词找到要删除的元素并将其从链上取下。

在清除所有元素的操作中，我们首先依次遍历元素并释放，使集合恢复到初始化状态。

在返回集合大小的过程中，我们直接返回之前已经维护好的 `size` 值。不过需要注意的是，由于 C 语言的 `by-value` 参数传递特性，如果使用 `hset` 作为参数类型，则会拷贝整个集合，经测试会导致 `Segmentation Fault`。因此我们采用了 `hset*` 指针类型作为参数，这样不会导致 `Segmentation Fault`。

在查找元素操作中，我们使用哈希函数算出元素所在的桶编号，并遍历同义词找到元素。查找函数的返回值是在查找过程中所遇到的冲突个数。

2.4 Trie 核心程序处理流程

在 Trie 树的集合设计中，字符串采用逆序插入方式进行处理：首先将字符串字符转为小写并自后向前逐层匹配，若当前层不存在对应字符节点则创建新节点并通过兄弟指针（`sib`）横向链接，通过孩子指针（`next`）纵向构建树形结构，同时在字符串末尾插入 `'\0'` 作为终止标记，以实现多字符串的高效存储与前缀区分。

在销毁集合操作中，我们采用广度优先遍历策略：使用队列依次存入各层节点，依次释放每个节点的兄弟子树和下一层子树，最终释放根节点并将集合指针置为空。

在插入元素操作中，我们沿字符串逆序逐字符匹配：若某层中未找到对应字符，则新建节点并插入当前层的兄弟链中；若整条路径均为新建节点，则更新根节点指针；函数返回值为本次插入是否实际增加了新节点（0 表示已存在，1 表示新插入）。

在删除元素操作中，我们同样逆序遍历字符串并匹配路径：仅当当前节点为 `'\0'` 时，改变当前节点为 `'0x1'`（一个不可能出现在域名中的控制字符）。我们“标记但不删除”的原因是，在以往的测试中，如果尝试 `free` 相关元素，必须判断是否为共享节点，且需要先进行 BFS 遍历后再遍历队列判断。这种方式不仅不精准，而且速度慢。故我们决定只标记而不删除。

在查找元素操作中，沿字符串逆序逐层匹配字符并统计遍历的节点数（包括兄弟节点和下一层节点），若完整匹配到终止符 `'\0'` 则返回查找路径长度，否则返回 0 表示未找到。

在统计集合大小操作中，采用广度优先遍历整棵 Trie 树，对所有没有下一层指针（`next`）的节点进行计数，每个此类节点代表一个存储字符串的终止位置，其总数即为集合中不同字符串的个数。

需要特别是，我们考虑到真实网址的特性——即大多是网址都是以 `.com`、`.cn` 等域名作为结尾的——采用了倒序储存，节省了大量空间。此外，我们发现本任务类似“DNS 查询”，因此为了实现真实世界中允许的通配符查询，我们也采用了倒序储存。不过因为通配符查询过于复杂，在调试过程中出现了许多问题，我们没有将其放入稳定的 `main` 分支中，而是发布在了我们仓库的 `dev` 分支当中，但我们的确实现了一定的效果。

2.5 各模块调用关系

- hash.h / trie.h: 定义了哈希表和键树这两大数据结构。
- hash_adt.h / trie_adt.h: 声明了集合操作的函数原型，作为模块间的调用接口。
- hash_adt.c / trie_adt.c: 负责实现所有集合操作功能。
- DST_Set.c: 主程序，调用两种实现进行测试。

(三) 测试设计

3.1 测试

我们首先新建了一个 Hash 集合（以下简称 H 集合）和一个 Trie 集合（以下简称 T 集合）。接下来，我们将位于 `./data/1M_Domain.txt` 的文件作为输入，分别插入进 H 和 T 集合。

在进行初始插入之后，我们关闭了文件，并测试了当前两个集合的大小。输出显示，两个集合大小都为 1000000，与我们直接打开文件后看到的行数相等，说明已经全部插入成功。

接下来，我们尝试在 H 和 T 集合中查找 `bupt.edu.cn`。查询结果显示，在 H 集合中 `bupt.edu.cn` 的同义词有 3 个，在 T 集合中查找的路径长度为 34。

在此之后，我们尝试在集合中删除 `tiktok.com`，并尝试在删除之后查找 `tiktok.com`。结果显示，我们无法找到 `tiktok.com`，说明其已经被删除了。此时打印集合大小，H 和 T 集合均为 999999。

最后，我们销毁了两个集合。值得一提的是，在 VSCode 以非调试模式运行销毁集合的时候，其运行速度特别缓慢。这是由于我们对 T 集合销毁操作是采用 BFS 而插入操作是 DFS 的，因此在销毁时缓存未命中现象严重。具体来说，当遍历释放节点时 CPU 缓存中预取的数据与实际访问模式不匹配，每次 `free()` 都需要从主内存重新加载数据，而内存碎片进一步加剧了这个问题。VSCode 的“非调试模式运行”仍然启动了 GDB，无形之中放大了这种缓存未命中的效应。在尝试了使用 Bash 直接运行这个程序后，我们发现 VSCode 上述特征使得程序运行速度降低到了原先的几分之一。

我们认为，本测试已经覆盖了常见使用场景，并成功验证了上述 adt 操作的接口合理性。

```

MINGW64/d/Semester_3/Data_Structure/Experiment 3/src
LI@DESKTOP-FOVNAKJ MINGW64 /d/Semester_3/Data_Structure/Experiment 3/src (main)
$ time ./a.exe
START:
INSERT HASH & TRIE OK
INITIAL SIZE: HASH 1000000 TRIE 1000000
bupt.edu.cn FIND: HASH 3 TRIE 34
ERASED tiktok.com IN HASH & TRIE
TRY TO FIND tiktok.com: HASH 0 TRIE 0
SIZE: HASH 999999 TRIE 999999
HASH DESTROY OK
Deleting: 6590000
TRIE DESTROY OK

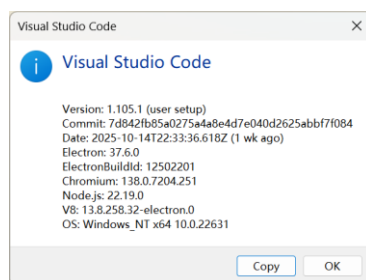
real    0m8.286s
user    0m0.000s
sys     0m0.000s

LI@DESKTOP-FOVNAKJ MINGW64 /d/Semester_3/Data_Structure/Experiment 3/src (main)
$
  
```

程序运行截图如上图所示。

3.2 环境说明

我们的程序是在 Windows 11 系统下，利用如下工具编写的：



```
命令提示符
gcc --version
gcc (x86_64-win32-seh-rev0, Built by MinGW-Builds project) 14.1.0
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

gdb --version
GNU gdb (GDB) 14.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

code --version
1.105.1
7d842fb85a0275a4a8e4d7e040d2625abbf7f084
x64
|
```

(四) 心得体会

(1)我们通过线下研讨，确定了对题目内容的理解，并在线上保持沟通，用 git 同步每个人的更改情况。我们在本次实验中第一次尝试了多文件编程，发现这样的方式使得每个文件都只负责一个具体的功能，结构清晰，方便我们快速定位错误和需要更改的地方。但美中不足的是，我们在编译时为了方便只是使用了 VSCode 的 tasks.json，将多文件参数加入其中，没有使用 Makefile 等工具。

(2)我们使用了最简单的处理冲突方法来实现哈希表。在哈希函数的选择上，我们考虑到一般的除留余数法过于基础，在网上查询了之后，发现异或操作比一般的加法要更有效。因此我们使用字符的 ASCII 码进行异或运算，并将字符按照顺序分为 4 组。我们又进行了压缩和雪崩，才写出了这个基于除留余数法的哈希函数。

(3)在键树的设计上，我们曾经考虑过用 37 叉（a-z, dot, 0-9）指针的树来实现，但发现这样每个节点至少要 $37 \times 8 + 1 = 297$ bytes（在 x86-64 系统中），过于浪费空间。因此我们选择使用孩子-兄弟树来表示键树，这样每个节点只需要 $2 \times 8 + 1 = 17$ bytes。且考虑到 .com 等后缀在网址中大量出现，我们采用了倒序储存的方式储存数据。

(4)调试过程中，我们遇到了多种问题，主要集中在 T 集合中。我们在 T 集合中出错的主要原因是：T 集合涉及了大量的指针引用和解引用，容易出错。且 T 集合的树结构用递归方法写程序十分简洁和优雅，但 50 万量级的输入规模一定会导致 stack overflow，故我们不得不放弃递归。在遍历的选择上，我们可以从 BFS 和 DFS 中选择。非递归 DFS 的核心难点在于除了要维护指针的栈（模拟系统的 runtime stack）之外，还需要考虑一个状态机模型，来确定访问本节点是来自父母、兄弟姐妹还是子女的回溯。综合考虑，我们采用了 BFS，并手动维护了一个链队列。

(5)算法的复杂度分析和改进设想：时间复杂度如下表所示

	Hash	Trie	备注
Init	$O(m)$	-	m 为 TABLE_MAX_SIZE
Destroy	$O(n + m)$	$O(n)$	n 为节点/元素总数
Insert	$O(L)$	$O(L \cdot b)$	b 为兄弟个数
Erase	$O(L)$	$O(L \cdot b)$	-
Clear	$O(n + m)$	-	-
Find	$O(L)$	$O(L \cdot b)$	-
Size	$O(1)$	$O(n)$	-
Utilities	$O(L)$	$O(1)$	L 为字符串长度

经过时间复杂度分析，我们发现对于网址的信息，哈希表数据结构优于键

树。键树的构造十分复杂，且操作的时间复杂度高，因此本场景下不适合使用键树。

(6)小组成员自评：

AUV888: 实验中我设计了主要的数据结构与算法，并进行了代码测试与后续的调试，在这个过程中我体会到设计和改进算法的奥妙。我深知本程序并非完全健壮，但囿于时间所限，不得不进行妥协。我将在以后的学习中不断提高思维能力，争取快速想出高健壮性的程序并减少调试次数。

xxkky2: 这次实验让我对数据结构的实现细节有了前所未有的深入理解，在哈希表实现中，我最大的收获是理解了链地址法的具体实现。不仅要考虑插入、查找、删除的基本逻辑，还要处理各种边界情况：空表插入、重复插入、删除不存在的元素等。Trie 树的实现则是让我对指针操作有了新的认识。通过亲手实现函数，我意识到编写健壮的代码需要充分考虑各种异常情况，进行充分的测试。

(五) 提交材料说明

提交材料	是否提交	文件名称
1、实验报告	<input checked="" type="checkbox"/>	/doc/数据结构 第三次实验 实验报告 AUV888 xxkky2.pdf
2、可执行程序	<input checked="" type="checkbox"/>	/src/DST_Set.exe
3、源程序，如果是多个文件要压缩到一个文件	<input checked="" type="checkbox"/>	/src/hash.h /src/hash_adt.h /src/trie.h /src/trie_adt.h /src/hash.c /src/hash_adt.c /src/trie.c /src/trie_adt.c /src/DST_Set.c

注：上述三个内容已经打包至“AUV888-xxkky2-数据结构-第三次实验.rar”