中国科学院大学
University of Chinese Academy of Sciences

数据结构与算法分析实验报告

哈希表算法实现

姓名 徐昕妍

学号 2023K8009970008

班级 2311 班

2025 年 5 月 14 日

目录

1 实验内容

1.1 主要实验内容

1. 哈希表的算法实现，包括哈希表创建、哈希表插入和哈希表打印；

2. 哈希冲突解决方法的实现，包括链地址法、线性检测法、二次检测法和再哈希检测法；

3. 哈希表最邻近搜索的实现，以及与暴力搜索的对比。

1.2 实验目的

1. 掌握哈希表的基本概念，研究并实现不同的哈希函数；

2. 探索哈希冲突的解决方法：实现不同的哈希冲突解决方法，如链式法、线性探测法、二次探测法和再哈希探测法。通过对比不同方法的优缺点，理解如何有效减少冲突对哈希表性能的影响；

3. 实现并比较最近邻搜索：通过输入测试数据点，实现在哈希表中查找最近邻点，并与暴力搜索方法进行对比。比较哈希表查找和暴力搜索的搜索时间和搜索结果。

2 实验设计

2.1 哈希表的算法实现

2.1.1 哈希表的基本概念

哈希表（又称散列表）是一种根据关键码值（Key-Value）而直接进行访问的数据结构。它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数（哈希函数），存放记录的数组叫做散列表。

核心组成部分 [1]

一个完整的哈希表包含以下几个核心部分：

- 键 (Key)：要存储的数据的唯一标识，如学生的学号、单词的拼写等
- 值 (Value)：与键相关联的实际数据，如学生信息、单词的定义等
- 哈希函数 (Hash Function)：将键转换为数组索引的函数，是哈希表高效的关键
- 数组 (Buckets)：存储数据的底层结构，通常是一个固定大小的数组
- 冲突解决机制：处理多个键映射到同一索引的情况

2.1.2 代码实现

1. 存储结构

```
// 定义链表节点结构（用于处理哈希冲突的链地址法）
typedef struct node {
    int data;            // 存储的实际数据（键值或键值对中的值）
    struct node *next;   // 指向下一个节点的指针（解决冲突时链接到同一桶中的下一个元素）
} Node;

// 定义哈希表的主结构
typedef struct {
    Node *firstp;        // 指向链表头节点的指针（每个桶对应一个链表）
} HashTable;
```

2. 函数功能

表 1: 哈希表函数功能描述表

函数名称	功能描述	输入参数	返回值
insertHashTable	向哈希表中插入一个新元素，使用链式法解决哈希冲突	‘HashTable ha[]‘：哈希表 ‘int n‘：当前元素个数 ‘int p‘：哈希表的大小 ‘int k‘：要插入的元素	无返回值
creatHashTable	创建并初始化哈希表，将给定的键值数组插入哈希表	‘HashTable ha[]‘：哈希表 ‘int m‘：哈希表的大小 ‘int p‘：哈希表的地址范围 ‘int keys[]‘：键值数组 ‘int n1‘：数组长度	无返回值
printHashTable	打印哈希表中的所有元素及其地址	‘HashTable ha[]‘：哈希表 ‘int m‘：哈希表的大小	无返回值

(1) 哈希表的插入

首先通过哈希函数（以 k mod p 为例）计算出 key 在表中的存储位置，为 key 创建一个新的节点，如果桶位置 addr 的链表为空（即哈希表的该桶还没有存储元素），我们将新节点直接放入该位置，并将 firstp 指向新节点。如果该桶已经有元素（即链表不为空），我们将通过链地址法解决冲突：将新节点插入到链表的头部，新的节点指向原先的第一个节点，并更新 firstp 指向新节点。

```
void insertHashTable(HashTable ha[],int n,int p,int k){
    int addr;
    addr=k%p;//计算地址
    Node* q;
    q=(Node*)malloc(sizeof(Node));
```

```
    q->data=k;
    q->next=NULL;//创建新节点
    if (ha[addr].firstp==NULL)
    {
        ha[addr].firstp=q;//若addr链表为空直接插入
    }else
    {
        q->next=ha[addr].firstp;
        ha[addr].firstp=q;//采用链地址法解决冲突
    }
    n++;
}
```

(2) 哈希表的创建

首先初始化各个桶，再调用插入函数将 key 依次插入在各个桶中。

```
void creatHashTable(HashTable ha[],int m,int p,int keys[],int n1) {
    for (int i = 0; i < m; i++)
    {
        ha[i].firstp=NULL;
    }//初始化桶
    int n=0;
    for (int i = 0; i < n1; i++)
    {
        insertHashTable(ha,n,p,keys[i]);
    }   //插入key
}
```

(3) 哈希表打印以桶的形式打印哈希表。

```
void printHashTable(HashTable ha[], int m){
    for (int i = 0; i < m; i++) {
        printf("addr %d: ", i);
        Node* p = ha[i].firstp;
        while (p != NULL) {
            printf("%d -> ", p->data);
            p = p->next;
        }
        printf("NULL\n");
    }
}
```

输入：

keys[N]=[1,2,3,4,5,6,7,8,9,10,11,12,13];

p=3;

输出如下图所示：

```
addr 0: 12 -> 9 -> 6 -> 3 -> NULL
addr 1: 13 -> 10 -> 7 -> 4 -> 1 -> NULL
addr 2: 11 -> 8 -> 5 -> 2 -> NULL
```

图 1: 输出结果

整体代码流程图如下图所示：

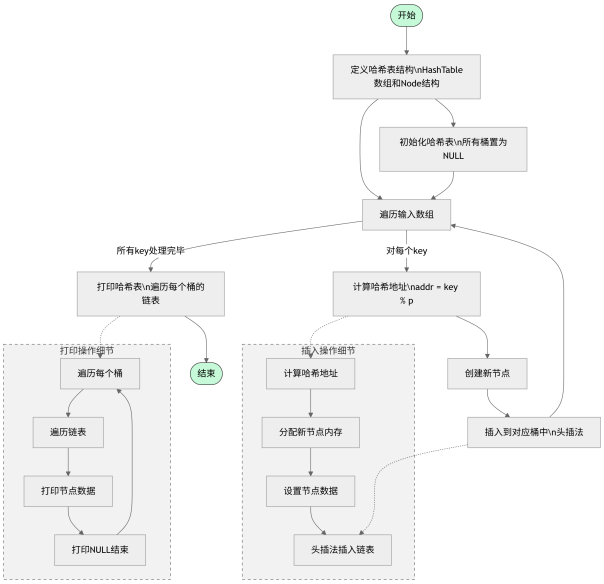

图 2: 哈希表实现流程图

2.2 哈希冲突解决方法的实现

2.2.1 具体解决方法

1. 链地址法

首先通过哈希函数（以 k mod p 为例）计算出 key 在表中的存储位置，为 key 创建一个新的节点，如果桶位置 addr 的链表为空（即哈希表的该桶还没有存储元素），我们将新节点直接放入该位置，并将 firstp 指向新节点。如果该桶已经有元素（即链表不为空），我们将通过链地址法解决冲突：将新节点插入到链表的头部，新的节点指向原先的第一个节点，并更新 firstp 指向新节点。

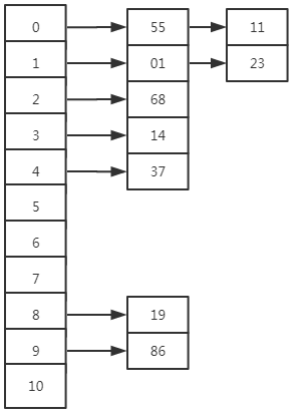

图 3: 链地址法示意图 [2]

具体代码见 2.1.2。

2. 线性探测法

在线性探测法中，如果发生冲突，算法会顺着哈希表的数组查找下一个空位置。如果当前位置已经被占用，则继续查找下一个位置，直到找到空位置。

不同于链地址法，线性探测法可以通过数组直接完成对哈希表的构建。存储结构如下所示：

```
typedef struct {
    int data;
    int isUsed; // 标记是否已使用
} HashEntry;
```

以下是具体代码：

```
void InsertHash(HashEntry ha[],int key,int p){
    int addr=key%p;
    int startaddr=addr;
    //处理冲突
    while (ha[addr].isUsed!=0)
    {
        addr=(addr+1)%p;//存储到addr的下一个位置上
        if (addr==startaddr)
        {
            printf("FULL!");\\当addr回到初始地址，说明哈希表已满
            return;
        }

    }
    ha[addr].data=key;
```

```
    ha[addr].isUsed=1;//标记为已存储
}
```

所得结果如下图所示：

```
addr 0: 47
addr 1: 93
addr 2: EMPTY
addr 3: EMPTY
addr 4: 4
addr 5: 52
addr 6: 37
addr 7: 86
addr 8: EMPTY
addr 9: 25
addr 10: 73
addr 11: 11
addr 12: 10
addr 13: 13
addr 14: 62
addr 15: 15
```

图 4: 线性探测法解决冲突结果

3. 二次探测法

二次探测法类似于线性探测法，只不过将每次探测的步长改为了 $i^2(i = 1, 2, 3...)$。它的优点是避免了线性探测的聚集问题，但可能会遇到某些位置永远无法访问的问题。

```
void InsertHash(HashEntry ha[],int key,int p){
    int addr=key%p;
    int startaddr=addr;
    int i=1;
    while (ha[addr].isUsed!=0)
    {
        addr=(key+i*i)%p;//以i*i为步长探测
        if (addr==startaddr)
        {
            printf("FULL!");
            return;
        }
        i++;
    }
    ha[addr].data=key;
    ha[addr].isUsed=1;
```

```
}
```

程序运行结果如下图所示：

```
addr 0: EMPTY
addr 1: 52
addr 2: 86
addr 3: 37
addr 4: 4
addr 5: 73
addr 6: EMPTY
addr 7: EMPTY
addr 8: 25
addr 9: 93
addr 10: 10
addr 11: 11
addr 12: 62
addr 13: 47
addr 14: 13
addr 15: 15
addr 16: EMPTY
```

图 5: 二次探测法解决冲突结果

4. 再哈希探测法

再哈希探测法使用了两个不同的哈希函数来确定冲突时的探测序列。当发生冲突是使用第二个哈希函数来计算步长，而不是简单地采用线性或者平方探测。

相比线性探测和平方探测，再哈希法能更有效地减少元素的聚集现象。同时，通过使用第二个哈希函数，使得探测序列更加随机化，分布更均匀。

```
//第一哈希函数
int hash1(int key,int p){
    return key%p;
}
//第二哈希函数
int hash2(int key,int i){
    return i*(key%7+1);
}
//哈希表插入（再哈希检测法）
void InsertHash(HashEntry ha[],int key,int p){
    int addr=hash1(key,p);
    int startaddr=addr;
    int i=1;
    \\处理冲突
    while (ha[addr].isUsed!=0)
    {
        addr=hash2(key,i);\\采用第二哈希函数确定探测步长
```

```
18          if (addr==startaddr)
19          {
20              printf("FULL!");
21              return;
22          }
23          i++;
24      }
25      ha[addr].data=key;
26      ha[addr].isUsed=1;
27  }
```

程序运行结果如下所示：

```
addr 0: EMPTY
addr 1: 52
addr 2: EMPTY
addr 3: 37
addr 4: 4
addr 5: 73
addr 6: 86
addr 7: 13
addr 8: 25
addr 9: 93
addr 10: 10
addr 11: 11
addr 12: EMPTY
addr 13: 47
addr 14: 62
addr 15: 15
addr 16: EMPTY
```

2.2.2 性能分析

表 2: 哈希冲突解决方法性能分析对比表

方法	时间复杂度	优点	缺点
链地址法	最优：O(1) 最差:O(n)	处理简单 负载因子可大于 1	指针额外消耗空间 缓存不友好
线性探测	最优：O(1) 最差:O(n)	空间连续 缓存友好	聚集现象严重 负载因子需小于 0.7
二次探测	最优：O(1) 平均:O(1)	缓解线性聚集	二次聚集问题 可能找不到空位
再哈希法	平均：O(1)	双重散列减少聚集	计算成本高 哈希函数设计复杂

应用场景

- 内存敏感场景：线性探测
- 高并发场景：再哈希法 + 原子操作
- 未知数据规模：链地址法 + 动态扩容
- SSD 存储系统：二次探测法

2.3 哈希表查找

本部分将在二次探测法处理冲突的基础上实现哈希表的查找。

2.3.1 哈希表查找的实现

基于二次探测法的哈希表查找主要是通过直接计算元素的地址来进行查找。

首先计算哈希地址，addr = key (mod p)；再检查 ha[addr] 位置上的元素是否与 key 相等，若不等则采用二次检测法继续计算哈希地址，直至查找到元素或遍历哈希表仍未查找到元素结束。

伪代码如下：

```
function SearchHash(ha[], key, p):
    addr = key % p                  // 计算哈希地址
    startaddr = addr                // 记录起始位置
    i = 1                           // 初始化探测次数
    steps = 1                       // 记录查找步数

```

```
    while ha[addr].isUsed != 0:    // 如果当前位置已使用
        if ha[addr].data == key:   // 找到目标元素
            return addr            // 返回地址

        addr = (key + i * i) % p   // 二次探测
        if addr == startaddr:      // 回到起始位置
            return -1              // 未找到目标元素

        i++                         // 增加探测次数
        steps++                     // 步数加1

    return -1                       // 未找到目标元素
```

2.3.2 与暴力搜索的性能对比

暴力搜索即从 ha[0] 开始遍历数组，直至找到元素。

通过计算含 100000 个元素的哈希表搜索时间及搜索步数，实现哈希表搜索与暴力搜索的性能对比。

程序运行结果如下图所示：

```
Single Search Comparison:
Key         Hash Found      Hash Steps      Brute Found     Brute Steps
11          11              1               27670           27671
37          37              1               10174           10175
93          95574           310             95112           95113
99          99              1               6531            6532
52          152             11              32491           32492
100         100             1               11292           11293

Performance Comparison (10000 searches):
Hash Table Search: 0.006000 seconds
Brute Force Search: 1.497000 seconds
Speed Ratio: 249.50:1 (Hash:Brute)

Average Steps per Search:
Hash Table: 54.17 steps
Brute Force: 30546.00 steps
```

图 6: 性能对比结果

从图中可知，哈希表查找的时间为 0.006000s，远小于暴力搜索查找时间 1.497000s，二者的速度之比为 249.5:1; 同时，哈希表的平均查找步数为 54.17 步，暴力搜索平均查找步数为 30546.00 步。由此可知，哈希表查找的效率远高于暴力搜索。

从算法角度分析，暴力搜索的时间复杂度为 O(n)，哈希表查找的平均时间复杂度接近 O(1)，所得结果也符合算法分析。

改变数据范围，测得多组查找时间和查找步数，绘成折线图。

取 N=10000,50000,100000,500000,1000000,SIZE=20000,100000,500000,1000000,2000000 时，平均步数、搜索时间、搜索速度对比如下图所示：

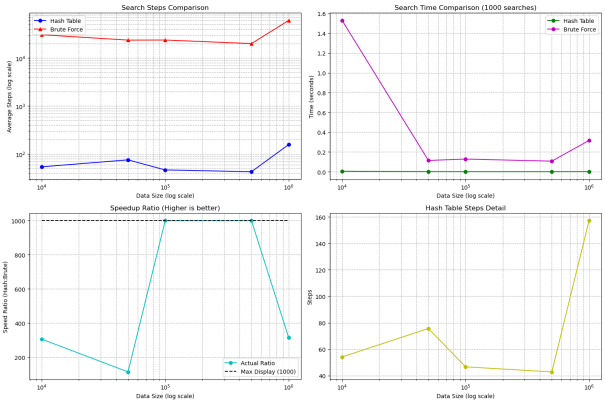

图 7: 结果对比图

由图表可以直观得到哈希表搜索速度和效率远高于暴力搜索。

3 实验总结

通过本次哈希表实验，我深刻理解了哈希表的核心原理与实现细节。从基础的哈希表构建、插入操作到复杂的冲突解决机制，我逐步实现了链地址法、线性探测、二次探测和再哈希四种方法，并在实践中对比了它们各自的性能特点。

实验中最有收获的部分是实现哈希表查找并与暴力搜索进行对比，这让我直观地认识到哈希表在时间效率上的巨大优势。同时在改变元素个数和哈希表桶数进行性能对比时，搜索时间和搜索步数的巨大变化也让我体会到参数调优对平衡查询速度与准确率的重要性。

参考文献

[1] https://blog.csdn.net/niuTyler/article/details/147797520. Accessed: 2025-5-12.

[2] https://blog.csdn.net/u011109881/article/details/80379505. Accessed: 2025-5-13.