**Lab5 哈希表**

**唐凯成 PB16001695**

**一.实验要求**

1、掌握哈希函数的构造方法和解决冲突的技术，实现哈希造表

·通过除留余数法构建哈希函数

·学习使用线性探测法实现解决冲突

·学习使用拉链法实现解决冲突

1. 掌握哈希表的查找技术

·创建哈希表的查找成功与查找失败次数表

·掌握平均查找长度ASL的计算方法

**二.实验平台**

Apple LLVM version 9.1.0 (clang-902.0.39.2)

Visual Studio Code v1.29.0

**三.实验内容**

1、总体设计

创建两个哈希表，分别表示两种实现冲突的方式，两种表均使用对长度取模的哈希函数，在冲突实现方面有一定区别。在哈希表外维护和哈希表相同结构的查找成功次数表及一维线性的查找失败次数表。

2、线性探测法解决Hash表冲突

对于使用线性探测法解决冲突的处理较为简单，即当通过取模发现表内对应位置已有关键字时，直接探测关键字后一位置是否为空，如有即存入，若没有即继续寻找下一位置，直至寻找到空白位置。（或寻找到自己第一次到的位置，即说明hash表已满，停止寻找并报错。）

对于线性探测法的查找成功及失败次数查询，成功次数可与生成hash表同步计算，即每往后移一位，对应查找成功次数+1。失败次数则需独立计算，即从每个位置向后寻找空白的位置，寻找次数+1即为查找失败次数，另外如果此时hash表已满，则每个位置的失败次数都应是hash表的表长。

3、拉链法解决Hash表冲突

对于拉链法解决Hash表冲突，实现起来复杂一些，但会使得数据结构更清晰，更方便对Hash表内数据查找。实现时使用了一位的指针数组作为存储结构，即数组中的每个位置都存放一指向该hash链表节点的指针，该指针后即构成该余数对应的链表。每当出现冲突时，只需新建一个链表节点并将其放在当前该余数的链表最后即可。

对于链表法的成功查找次数表，由于其结构与hash表一样，故直接在原hash表节点中加入标签searchtime，用以记录该链表节点的深度，即为查找成功所需的次数；对于查找失败的次数，即为每个链表最深节点的深度+1即可。

**四.实验代码**

1、hash.cpp

#include<iostream>

#include <fstream>

#include <sstream>

using namespace std;

int hashkey[1000];

// 线性探测法实现hash表

void hash1(int \*hash,int n,int key)

{

int i,val,tmp;

int \*stime=new int[key];

int \*failtime=new int[key];

hash=new int[key];

// init hash

for(i=0;i<key;i++)

{

hash[i]=-1;

stime[i]=0;

}

// create hash

int suctime;

for(i=0;i<n;i++)

{

val=hashkey[i]%key;

suctime=1;

if(hash[val]==-1)

{

hash[val]=hashkey[i];

stime[val]=suctime;

}

else

{

tmp=val;

val++;

while(val!=tmp)

{

if(val>=key) val=val-key;

suctime++;

if(hash[val]==-1)

{

hash[val]=hashkey[i];

stime[val]=suctime;

break;

}

val++;

}

if(val==tmp)

{

printf("The hash is full!\n");

printf("Insert %d failed and quited.\n",hashkey[i]);

exit(0);

}

printf("Insert %d success!\n",hashkey[i]);

}

}

// fail search time

for(i=0;i<key;i++)

{

failtime[i]=1;

if(hash[i]!=-1)

{

tmp=i;

while(hash[tmp]!=-1)

{

failtime[i]++;

tmp++;

if(tmp>=key)

tmp=tmp-key;

if(tmp==i-1) break;

}

}

}

// output hash

float suc\_total=0,fail\_total=0;

printf("哈希表的地址：\t");

for(i=0;i<key;i++)

{

printf("%d\t",i);

}

printf("\n表中的关键字：\t");

for(i=0;i<key;i++)

{

if(hash[i]!=-1)

printf("%d\t",hash[i]);

else printf("-\t");

}

printf("\n成功查找次数：\t");

for(i=0;i<key;i++)

{

suc\_total+=stime[i];

printf("%d\t",stime[i]);

}

printf("\n失败查找次数：\t");

for(i=0;i<key;i++)

{

fail\_total+=failtime[i];

printf("%d\t",failtime[i]);

}

printf("\n查找成功的平均查找长度：%.2f\n",suc\_total/n);

printf("查找失败的平均查找长度：%.2f\n",fail\_total/key);

}

// 拉链法节点定义

typedef struct hashnode{

int num; // key

int seatime; // success search time

hashnode \*next; // next pointer

} hashnode;

// 拉链法实现hash表

void hash2(hashnode \*hash,int n,int key)

{

int i,val,tmp;

int \*failtime=new int[key];

hash=new hashnode[key];

// init hash

for(i=0;i<key;i++)

{

hash[i].num=-1;

hash[i].seatime=0;

hash[i].next=NULL;

}

// create hash

int suctime;

for(i=0;i<n;i++)

{

val=hashkey[i]%key;

suctime=1;

if(hash[val].num==-1)

{

hash[val].num=hashkey[i];

hash[val].seatime=suctime;

}

else

{

// malloc new node

hashnode \*p=new hashnode;

p->num=hashkey[i];

p->next=NULL;

// set the node to the tail

hashnode \*q=(hash+val);

suctime++;

while(q->next!=NULL)

{

suctime++;

q=q->next;

}

q->next=p;

p->seatime=suctime;

printf("Insert %d success!Deep: %d!\n",hashkey[i],suctime);

}

}

// fail search time

for(i=0;i<key;i++)

{

hashnode \*q=(hash+i);

while(q->next!=NULL)

{

q=q->next;

}

failtime[i]=q->seatime+1;

}

// output hash

float suc\_total=0,fail\_total=0;

printf("Address： Key：\n");

for(i=0;i<key;i++)

{

printf("%d\t",i);

hashnode \*q=(hash+i);

while(q!=NULL)

{

printf("%d\t",q->num);

q=q->next;

}

printf("\n");

}

printf("Address： Successtime：\n");

for(i=0;i<key;i++)

{

printf("%d\t",i);

hashnode \*q=(hash+i);

while(q!=NULL)

{

printf("%d\t",q->seatime);

suc\_total+=q->seatime;

q=q->next;

}

printf("\n");

}

printf("Address：\t");

for(i=0;i<key;i++)

{

printf("%d\t",i);

}

printf("\nFailtime：\t");

for(i=0;i<key;i++)

{

fail\_total+=failtime[i];

printf("%d\t",failtime[i]);

}

printf("\nSearching successfully average length：%.2f\n",suc\_total/n);

printf("Searching fail average length：%.2f\n",fail\_total/key);

}

int main(int argc,char \*argv[])

{

int n,i,key;

int \*hash;

hashnode \*hashla;

ifstream in(argv[1]);

in>>n;

for(i=0;i<n;i++)

{

in>>hashkey[i];

}

in>>key;

hash1(hash,n,key);

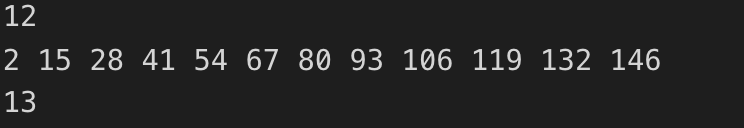
hash2(hashla,n,key);

}

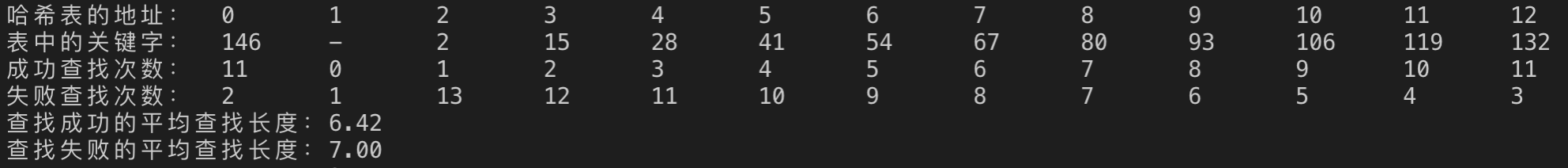
**五.实验结果**

1、线性探测法实现的hash表：

（1）输入：采用in2.txt输入



（2）输出



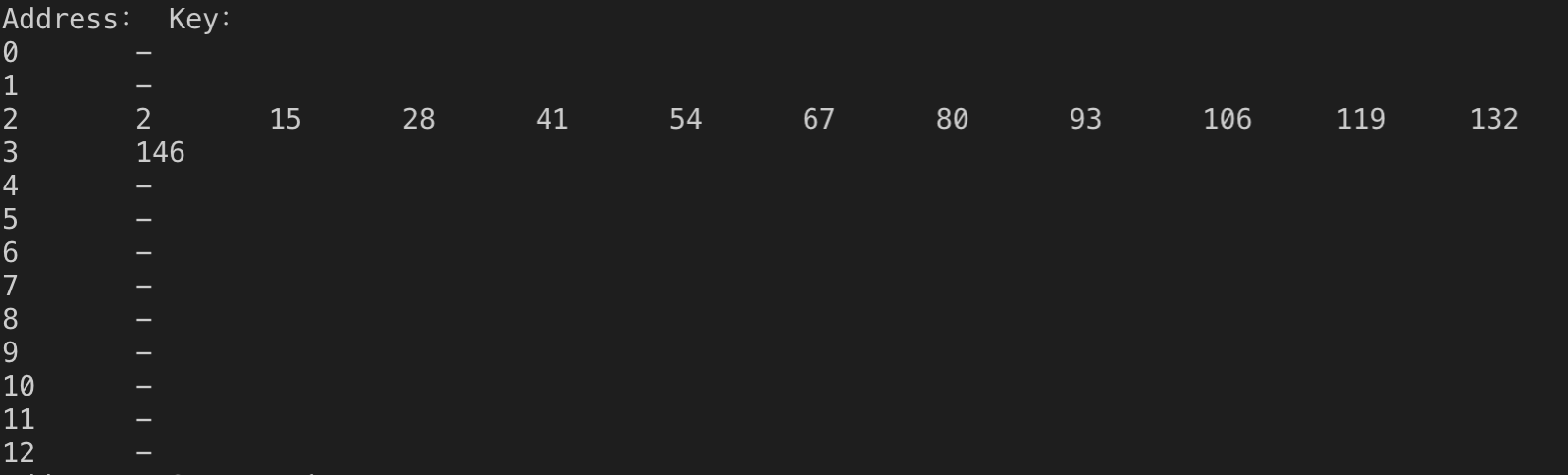
（3）分析

由以上输出可知结果正确，该组测试数据除最后一个146外均是被13模等于2的数字，故一直冲突到132，而146只能探测一圈后存放在第一个位置。故结果与预期完全吻合，可见当冲突较多时，线性探测的方法冲突较多，平均成功/失败的查找长度均较长。

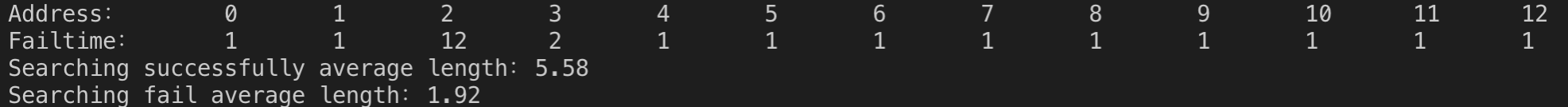
2、拉链法实现的hash表：

（1）输入：同样使用上述的in2.txt作为输入

（2）输出：







（3）分析

由结果验证可知正确，可见对比线性探测法，拉链法对于冲突较多的数据的平均查找次数会有一定幅度的减少，特别是查找失败的次数，相比线性探测法有大幅度地降低。

**五.实验总结**

本次实验实现了hash表的数据结构，尝试了两种基础的处理数据冲突的方式，更好地掌握了hash表的结构，也深深体会到O(1)查找速度上的优越。