Fundamentals of Data structures

Laboratory Project 3

Hashing – Hard Version

Date: 2019-1-6

Chapter 1 Introduction

1.1 Description of the problem

Hash has been widely used in computer science. However, an intractable problem has always been troubling us: hash collision. To illustrate this problem, we may define a hash function from Z to N: y_{hash} = H(x), and a hash collision occurs whenx1 ≠ X2 but H(x1) = H(x2).

To solve this problem, we may use different methods, such as "Separate Chaining", "Open

Addressing" and "Rehashing". And the problem we'll solve now is the reverse problem of "Open Addressing": Given a hash table whose size is N, with a hash function H(x) = x % N, and we know all the elements' place in the hash table. Under the circumstance that "Linear Probing" method is used to solve the probable collisions, we need to get the input sequence in order.

We can use the given sample to comprehend this problem:

Table	***33***	***1***	***13***	***12***	***34***	***38***	***27***	***22***	***32***	***-1***	***21***
index	0	1	2	3	4	5	6	7	8	9	10

Given a sequence of "33 1 13 12 34 38 27 22 32 -1 21", we can obtain many input sequences to let this hash table be "33 1 13 12 34 38 27 22 32 -1 21", but these input sequences must meet certain condition, which is some numbers must be before some other numbers. However, one input also may have multiple choices. If there are many choices, we need to choose the smallest number. So in this sample we can get the input sequence, which is "1 13 12 21 33 34 38 27 22 32".

1.2 The way to solve the problem

Before we solve the problem in formal, we have considered a lot of algorithms, but most of them are abandoned because of the complexity of space and the complexity of time. Besides this, we also must consider the hardness of coding. Finally, we make the decision to use such algorithm.

By learning the hash table and the topological sort, we can combine these two parts of knowledge. Because we have concluded that some numbers must be before some other numbers, so we can easily associate the topological sort knowledge which we have just learnt. By using the topological sort and combine the topological sort and the hash table, we can easily solve the problem, so I will introduce our algorithm specification in details as follows.

Chapter 2 Algorithm Specification

In this problem, we can easily find out that some numbers have to be inserted before other numbers before other numbers due to the collision occurs and “the smallest number is always taken”, which can be regarded as a kind of partial order.

According what I said above, we can consider using topological sort to solve this problem.

In the main function, we will follow these steps:

INPUT: The size of the hash table and each element in the sequence of index.

OUTPUT: The initial input order of each element.

STEP 1: Create a hash table and initialize it.

STEP 2: Allocate space for array Graph and array Indegree and initialize them.

STEP 3: Create a corresponding graph and initialize it.

STEP 4: Solve the problem by topological sort.

2.1 Data Structure

1. hash table

As the codes shows, we define the hash table in this way, because in this problem the element type of the table is int, so define the element type is int.

```c
#define MAXIMUM 1e9
#define Elementtype int
typedef struct hash *HASH;//define a hash table structure
struct hash
{
    int TableSize;
    Elementtype* Table;
};
```

2. graph

Because of the fact that we have to solve this problem by topological sort, so we must construct a graph, which is stored in an adjacent matrix. And this adjacent matrix is stored in a two-dimensional array. In the meanwhile, we use an array to record the indegree of each vertex.

2.2 Description of Algorithm

2.2.1 create the hash table

```c
int Count = 0;
HASH HashTable = (HASH)malloc(sizeof(struct hash));
int i, j;
/create a hash table
scanf("%d", &HashTable->TableSize);
HashTable->Table = (Elementtype *)malloc(sizeof(Elementtype) *
HashTable->TableSize);
for (i = 0; i < HashTable->TableSize; i++)
{
    scanf("%d", &HashTable->Table[i]);
    if (HashTable->Table[i] >= 0)
    {
        Count++;
    }
}
```

Before reading the numbers we need to allocate the space to the hash table, and loops to write every numbers into the hash table.

2.2.2 create the graph

```c
//create a corresponding graph.
void MakeGraph(HASH HashTable, int** Graph, int* Indegree)
{
    int CurrentIndex = 0;
    int TempIndex;
//consider all slots in the hash table as vertexes
    while (CurrentIndex < HashTable->TableSize)
    {
//connect the collided numbers
        if (HashTable->Table[CurrentIndex] >= 0 && (HashTable
->Table[CurrentIndex] % HashTable->TableSize) != CurrentIndex)
        {
            TempIndex = CurrentIndex;
            while ((HashTable->Table[CurrentIndex] % HashTable
->TableSize) != TempIndex)
            {
                if (TempIndex == 0)
                {
                    TempIndex = HashTable->TableSize - 1;
                }
                else
                {
                    TempIndex--;
                }
                Graph[TempIndex][CurrentIndex] = 1;
                Indegree[CurrentIndex]++;
            }
        }
        CurrentIndex++;
    }
    return;
}
```

We create the graph by inspecting if this element is the exact position. One element is one vertex. If it is in the correct position, its in degree is 0. And if it is not in the exact position. From its correct position and its current position, these elements are before the element. So there are edges from these elements to this element. By doing this we can get a graph without circle, which can be topological sort.

2.2.3 topological sort

Before topological sort, we just have done some preparation work, Topological sort is the core key of this problem.

```c
void SolveReHash(HASH HashTable, int** Graph, int* Indegree, int Count)
{
    int repeat;
    int tempkey;
    int tempindex;
    int i;
//find the smallest number that could be incerted into hash table for now (whose indegree is one)
    for (repeat = 0; repeat < Count; repeat++)
    {
        tempkey = MAXIMUM;
        tempindex = 0;

        for (i = 0; i < HashTable->TableSize; i++)
        {
            if (HashTable->Table[i] >= 0 && HashTable->Table[i] < tempkey && Indegree[i] == 0)
            {
                tempindex = i;
                tempkey = HashTable->Table[i];
            }
        }
//delete that number and the indegree of which is connected to it minus 1
        HashTable->Table[tempindex] = -1;
        for (i = 0; i < HashTable->TableSize; i++)
        {
            if (Graph[tempindex][i] == 1)
            {
                Graph[tempindex][i] = 0;
                Indegree[i]--;
            }
        }
//output the results one by one
        if (repeat == Count - 1)
        {
            printf("%d", tempkey);
        }
        else
        {
            printf("%d ", tempkey);
        }
    }
    return;
}
```

First of all, our aim is to find out the remaining vertices whose number of indegree is 0. And then find the smallest number of all optional numbers. When we find the first smallest number of all optional elements, we accomplish the find of first input number. And then we should do some further processing. Firstly, we need delete the edges which is corresponding to the vertex. And the number of indegree of adjacent vertex should be subtracted 1. Finally, we can print this vertex.

When we have printed all the elements, this program will cease.

Chapter 3 Test

Test is a vital work, programmer, tester, and I are really care about it. So our tester wrote a program to generate the cases to test, besides he has already thought out some special test cases which will make the program error to test our program.

No.	*Test case*	*Correctness(Y or N)*
1	*Same as sample*	*Y*
2	*Min size(N=1)*	*Y*
3	*Max size(N=1000)(random)*	*Y*
4	*Input numbers including 0*	*Y*
5	*One than one non-negative integers*	*Y*
6	*Produce collision at N-1*	*Y*
7	*Random case*	*Y*
8	*All non-negative integers*	*Y*

Test cases and the results

1. Sample

Input :

```
11
33 1 13 12 34 38 27 22 32 -1 21
```

Output :

```
1 13 12 21 33 34 38 27 22 32
```

2. Min Size(N=1)

Input :

```
1
1
```

Output :

```
1
```

3. Max Size(N=1000)

Please see the test case and the result in the test1.txt and test1.png

4. **Input numbers including 0**

   Input ;
   10
   0 19 -1 3 2973 -10 -7 7 -14 9
   Output :
   0 3 7 9 19 2973

5. **More than one non-negative integers**

   Input :
   10
   9320 271 2 19212 -10 23565 32765 37 7 -1
   Output :
   2 37 7 271 9320 19212 23565 32765

6. **Produce collision at N-1**

   Input :
   10
   19 1 2 3 4 5 6 7 8 9
   Output :
   1 2 3 4 5 6 7 8 9 19

7. **Random case (using random function rand())**

   Input :
   10
   14300 7260 15842 31741 24593 21340 -1 -2 17888 -3
   Output :
   14300 7260 15842 17888 31741 24593 21340

8. **All empty**

   Input :
   8
   -1 -1 -1 -1 -1 -1 -1 -1
   Output :
   (Null)

Chapter 4 Analysis and Comments

4.1 The Analysis of Space Complexity

In the most space-consuming part, making digraph, we need a two-dimensional array to save vertices and edges of each element. Therefore, it requires complexity of **$O(N^2)$**. And we define a one-dimensional array to save indegree of each vertex, define a one-dimensional array to save hash table, whose complexity is **$O(N)$.**

Thus, above all, our space complexity of the program is **$O(N^2)$.**

4.2 The Analysis of Time Complexity

As for the complexity of time, all parts are linear except making digraph and topological sort it. Both functions involve time complexity of **$O(N^2)$** for the existence of a nested-loop. In many other parts of the program, we have traversing a one-dimensional array once. Its complexity is **$O(N)$.**

Thus, above all, our time complexity of the program is **$O(N^2)$.**

4.3 Comments

On the one hand, the program solves problem properly and perfectly meets the demand of the program, the form of grammar of this program is standard, and the structure of this program is clear. On the other hand, the program use much space, actually we can improve and optimize this program. And we can also not use the topological sort and use a simple algorithm.

Chapter 5 Declaration

We hereby declare that all the work done in this project titled "Hash – Hard Version" is of out independent effort as a group.

Chapter 6 Appendix

6.1 Program

```c
//  DS LAB 3

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXIMUM 1e9
#define Elementtype int
typedef struct hash *HASH;//define a hash table structure
struct hash
{
    int TableSize;
    Elementtype* Table;
};
void MakeGraph(HASH HashTable, int** Graph, int* Indegree);//declaration for function MakeGraph
void SolveReHash(HASH HashTable, int** Graph, int* Indegree, int Count);//declaration for function SolveReHash

int main()
{
    int CurrentIndex = 0;
    int Count = 0;
    HASH HashTable = (HASH)malloc(sizeof(struct hash));
    int i, j;
//create a hash table
    scanf("%d", &HashTable->TableSize);
    HashTable->Table = (Elementtype *)malloc(sizeof(Elementtype) * HashTable->TableSize);
    for (i = 0; i < HashTable->TableSize; i++)
    {
        scanf("%d", &HashTable->Table[i]);
        if (HashTable->Table[i] >= 0)
        {
            Count++;
        }
    }
//allocate space for array Graph and array Indegree and initialize them
```

```c
    int** Graph = (int **)malloc(sizeof(int *) * HashTable->TableSize);
    int* Indegree = (int *)malloc(sizeof(int) * HashTable->TableSize);
    for (i = 0; i < HashTable->TableSize; i++)
    {
        Graph[i] = (int *)malloc(sizeof(int) * HashTable->TableSize);
    }
    for (i = 0; i < HashTable->TableSize; i++)
    {
        Indegree[i] = 0;
        for (j = 0; j < HashTable->TableSize; j++)
        {
            Graph[i][j] = 0;
        }
    }
//create a corresponding graph.
    MakeGraph(HashTable, Graph, Indegree);
//solve the problem by topological sort.
    SolveReHash(HashTable, Graph, Indegree, Count);

    return 0;
}


//create a corresponding graph.
void MakeGraph(HASH HashTable, int** Graph, int* Indegree)
{
    int CurrentIndex = 0;
    int TempIndex;
//consider all slots in the hash table as vertexes
    while (CurrentIndex < HashTable->TableSize)
    {
//connect the collided numbers
        if (HashTable->Table[CurrentIndex] >= 0 && (HashTable->Table[CurrentIndex] % HashTable->TableSize) != CurrentIndex)
        {
            TempIndex = CurrentIndex;
            while ((HashTable->Table[CurrentIndex] % HashTable->TableSize) != TempIndex)
            {
                if (TempIndex == 0)
                {
                    TempIndex = HashTable->TableSize - 1;
                }
                else
                {
                    TempIndex--;
                }
                Graph[TempIndex][CurrentIndex] = 1;
                Indegree[CurrentIndex]++;
            }
        }
        CurrentIndex++;
    }
    return;
}


void SolveReHash(HASH HashTable, int** Graph, int* Indegree, int Count)
{
    int repeat;
    int tempkey;
    int tempindex;
    int i;
//find the smallest number that could be incerted into hash table for now (whose indegree is one)
    for (repeat = 0; repeat < Count; repeat++)
    {
        tempkey = MAXIMUM;
        tempindex = 0;

        for (i = 0; i < HashTable->TableSize; i++)
        {
            if (HashTable->Table[i] >= 0 && HashTable->Table[i] < tempkey && Indegree[i] == 0)
            {
                tempindex = i;
                tempkey = HashTable->Table[i];
            }
        }
//delete that number and the indegree of those which is connected to it minus 1
        HashTable->Table[tempindex] = -1;
        for (i = 0; i < HashTable->TableSize; i++)
        {
```

```c
            if (Graph[tempindex][i] == 1)
            {
                Graph[tempindex][i] = 0;
                Indegree[i]--;
            }
        }
//output the results one by one
        if (repeat == Count - 1)
        {
            printf("%d", tempkey);
        }
        else
        {
            printf("%d ", tempkey);
        }
    }
    return;
}
```

6.2 The program to generate test cases

```c
#include<stdio.h>
#include<time.h>
#include<memory.h>
#include<stdlib.h>

#define N 100000

int number[N];
int hash[1005];
int main(){
    FILE *fp;
    fp=fopen("test2.txt","w");
    if(fp==NULL){
        puts("ERROR");
        return -1;
    }
    int num=50,i,j,m,temp,sit,n;
    srand((unsigned)time(NULL));
    //num 控制测试总组数
    while(num--){
        memset(hash,-1,1005*sizeof(int));
        memset(number,0,N*sizeof(int));
        //m is to control the first number, I suggest it being 10 which is easy to debug.
        //m=rand()%1000+1;
        m=10;
        fprintf(fp,"%d\n",m);
        n=1;
        if(m!=1)    n=rand()%(m/2)+(m/2)+1;
        for(i=0;i<n;i++){
            temp=rand()%N;
            while(number[temp]==1){
                temp=rand()%N;
            }
            //printf("%d ",temp);
            number[temp]=1;
            sit=temp%m;
            while(hash[sit]!=-1){
                sit=(sit+1)%m;
            }
            hash[sit]=temp;
        }
        //puts("");
        for(i=0;i<m;i++){
            fprintf(fp,"%d ",hash[i]);
        }
        fprintf(fp,"\n");
    }
    return 0;
}
```

Duty Assignments:

Programmer: Chen Yuwei

Tester: Ren Qilan

Report Writer: Peng Zifan