Hashing
Hard Version

Fundamental of Data Structure

Project 3

Jie Feng Yizhou Chen Jinze Wu

2018-12-28

Chapter 1 Introduction

1.1 Background Information

A hash function is any function that can be used to map data. The hash function will give a value to each key in range of 0 to *TableSize* to place the key into the hash table. Of course, hash functions have to deal with collision, and linear probing, a strategy that if a collision happened just move to the nearest available cell, is a common method to solve it.

1.2 Problem Description

General Description

The problem is that we have already obtained the hash function and the hash table with a sequence of input, which use linear probing as the method to solve collision. The job of us is to get the original sequence before the rearrangement operated by hash function out of the given status. If a place has several options, it should take the smallest one.

INPUT / OUTPUT Specification

- **INPUT**
  - a positive integer N (N<=1000) to define the size of the hash table.
  - N integers as the keys, a negative integer represents an empty cell, all the non-negative integers are distinct.
- **OUTPUT**
  - The original input sequence. The numbers should be separated by a space and no extra space at the end of each line.

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
```

1.3 Overview of our job

For this problem, we use topological sort to find the original input sequence, while operating topological sort min heap sort will be used to make sure smallest one to be taken first. The test part will generate a random input sequence in range of 0 to tablesize and use this to generate result of hashing, then use this result to get the original input by code. Because the random input can’t make sure that the smallest one is taken first, we hash the input(get by project) again and get the hashtable, compare two hashtables, if they are identical, our coding works. The Time complexity and the space complexity are both O(N^2).

Chapter 2 Algorithm Specification

2.1 Structure Description

Two structures are used in the program, one is the AdjListNode, which is used to build a AOV network to operate topological sort, another one is used to operate heap sort.

```
Struct AdjListNode
1:  int vertex
2:  Pointer next
```

```
Struct Heapsort
1:  int size, capacity
2:  Pointer elements
```

2.2 A Sketch of Main Program

2.2.1 Specification

This whole project consists of a main function, three functions to operate heap sort and a test.

The main idea is that we storage the hash table first, and then calculate the in_degree of keys (just the number of collisions). Regard every element as a vertex and build a directed graph to operate topological sort (the direction are determined by the in-degree), because when a multiple choice exist we have to take the smallest one, we put the vertexes with in_degree equals to zero into a min priority queue. Finally, we output the min from the heap and decrease the in_degree of all the next adjacency vertexes, if the in_degree equals to zero too, enqueue, do this until all the keys are output.

Data structure used in the project:

Priority queue; hash table; adjacency list.

2.2.2 Pseudo code

Main Function

```
//specification:   value[] contain the values in the hash table( given by the problem)
//hashkey[] contain the key of the elements in the hashtable(the place in the table)
//input[] is the original input sequence got by this program.
//list[] is the adjacency list.
1:   Function main( )
2:   int n, m, i, j, x;
3:   int value[1001], hashkey[1000], in_degree[1001], input[1001];
4:   AdjList temp;                /*point to AdjListNode */
5:   AdjList list[1001];
6:   PriorityQueue Heap;     /*point to Heapsort*/
7:   Input n;
8:   For( i = 0, j = 0; i < n; i++)                          /*input*/
9:        Input x;
10:       If( not empty )
11:            Store the value and place;
12:            j++;
13:  m = j;          /*the number of vertexes*/
14:  For(i = 0; i < m; i++)                                  /*build list*/
15:       in_degree[i] = 0;
16:       list[i] = NULL;
17:  For(i = 0; i < m; i++)
18:        j = i;                                            /*calculate the in_degree and build*/
19:       While( have predecessor)                           /*the adjList*/
20:            j--;
21:            If( j==-1)   j=m-1;
22:            in_degree[ i ]++;
23:            Build vertex with this key;
24:            Add vertex to adjList;
25:  Call initialize_heap;
26:  For( i = 0; i < m; i++ )
27:       If(in_degree[i] == 0)
28:            Call insert(i,value,Heap);
```

```
For( i = 0; i < m; i++ )                    /*heap sort*/
    If ( Heap is empty)
        Exit(error);
    input [ i ] = value[ heap->elements[1] ];
    Call delete(value,Heap);
    While ( the adjList is not empty )
        in_degree--;
        If( in_degree == 0 )
            Call insert( temp->vertex,value,Heap );
        Move to next;
For( i =0; i < m-1; i++)
    Output( input[ i ]);
End function
```

2.2.3 The correctness

As we learned in the lecture that the topological sort can be used to get the precedence relation. Obviously, this question is a typical question for precedence relation. In the code, we program in a standard manner to operate this algorithm, so it’ s effective. To meet the demand of minimum value taken first , we use min heap to get the minimum, in the min heap, the key at the root must be minimum, the correctness is obvious.

The rest functions are standard initialize function, insert function and delete function, they have been proven working in our classes.

2.3 Function initialize_heap

2.3.1 Specification:

This function is used to build the heap and initialize it, this heap will be used to insert all the vertexes with in_degree equals to zero and output in a certain order.

2.3.2 Pseudo code:

Function : initialize_heap

```
function   initialize_heap( int m, PriorityQueue H )
```

```
2:  H = malloc( sizeof ( HeapStruct) );
3:  H->capacity = m+1;
4:  H->size = 0;
5:  H->elements = malloc( sizeof( int )*( m+1 ));
6:  H->elements[0] =-1;
7:  return   H;
8:   end function
```

2.4 Function insert

2.4.1 Specification:

This function is used to insert the the vertexes with in_degree equals to zero.

2.4.2 Pseudo code:

```
Function :      insert
1:  function   insert(int x,int value[],PriorityQueue H)
2:  H->size++;
3:  Percolate up;
4:  Insert the vertex;
5:   end function
```

2.5 Function delete

2.5.1 Specification:

This function is used to delete the min element in this heap, which has been output .

2.5.2 Pseudo code:

```
Function :      delete
1:  function   delete(int value[ ], PriorityQueue H)
2:  H-size--;
3:  Percolate down;
4:  place the last vertex in proper place;
8:   end function
```

Chapter 3 Testing Results

3.1 Purpose

Our purpose is to test the correctness of the algorithm.

3.2 Specification

3.2.1 Correctness Test

Special Data Test

Firstly, we test the algorithm with common sample and extreme data , including a common sample, the hash table is empty, the hash table is full, the hash table is very large, the input data is error.

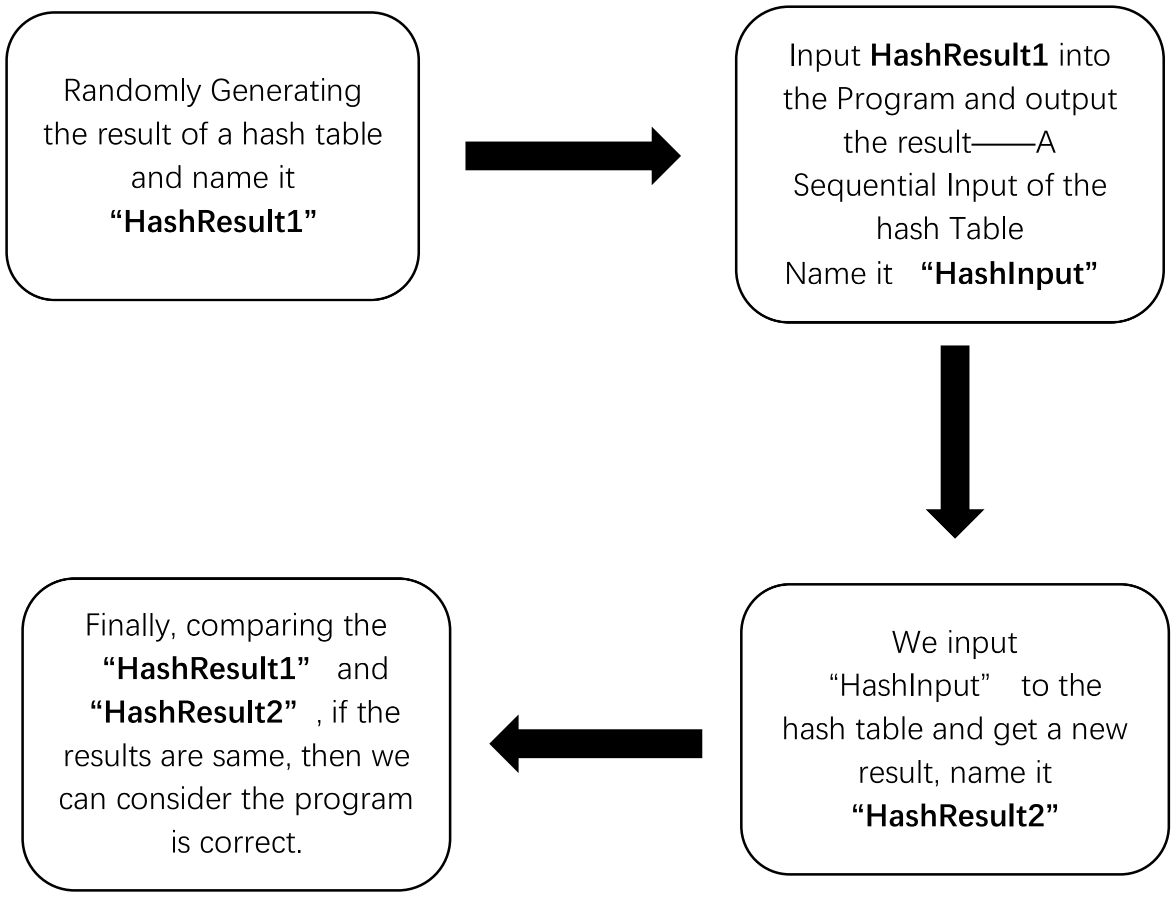
Massive Data Test

Secondly, we test the algorithm with massive random data, to further ensure its correctness. We implement **1-GenerateRandomHashResult.c** to randomly generate a result of hash table with the required input data(Input data can be edited in **input.in**). We output the result of hash table into a file **HashResult_1.out**. Then we treat the output as input to another program named **2-GenerateNewHashInput.c** to generate the input of hash table with ascending order, the output is named **HashInput.out**. Next, we input the **HashInput.out** into a new program named **3-GenerateNewHashResult.c**, which will output the file named **HashResult_2.out** including the result of hash table. Finally, we implement a program **compare.c**, to compare **HashResult_1.out** and **HashResult_2.out** to see if

3.3 Test Data

3.3.1 Special Data Test

We first randomly generate the result of a hash table as the input of the program, then put the program’ s results into the hash table again, and compare the results of the two hash tables to check the correctness of the program.


Randomly Generating the result of a hash table and name it **“HashResult1”**
Input **HashResult1** into the Program and output the result——A Sequential Input of the hash Table Name it **“HashInput”**
We input “HashInput” to the hash table and get a new result, name it **“HashResult2”**
Finally, comparing the **“HashResult1”** and **“HashResult2”**, if the results are same, then we can consider the program is correct.

In order to get reliable results, we manually calculate the program’ s result, input the result to the hash table to check the correctness.

Sample1 (sample)

purpose: To test the common case of small-scale comprehensive test.

input

```
11
44 -1 13 -1 4 49 -1 18 30 86 21
```

output

```
4 13 18 21 30 44 49 86
```

Manual inspection

	0	1	2	3	4	5	6	7	8	9	10
Insert 4					4						
Insert 13			13								
Insert 18								18			
Insert 21											21
Insert 30									30		
Insert 44	44										
Insert 49						49					
Insert 86										86	
Input Data	44	-1	13	-1	4	49	-1	18	30	86	21

Sample2 (full Table with collision)

purpose : To test the special condition when hash table is fill up and have collision .

input

```
11
32 1 24 3 25 38 17 28 8 30 20
```

output

```
1 3 8 17 24 25 28 30 20 32 38
```

Manual inspection

	0	1	2	3	4	5	6	7	8	9	10
Insert 1		1									
Insert 3				3							
Insert 8									8		
Insert 17							17				
Insert 24			24								
Insert 25					25						
Insert 28								28			
Insert 30										30	
Insert 20											20
Insert 32	32										
Insert 38						38					
Input Data	32	1	24	3	25	38	17	28	8	30	20

Sample3 (full table without collision)

purpose : To test the common case of small-scale comprehensive test.

input

```
8
16 9 34 11 20 13 30 63
```

output

```
9 11 13 16 20 30 34 63
```

	0	1	2	3	4	5	6	7
Insert 9		9						
Insert 11				11				
Insert 13						13		
Insert 16	16							
Insert 20					20			
Insert 30							30	
Insert 34			34					
Insert 63								63
Input Data	16	9	34	11	20	13	30	63

Sample4 (error table)

purpose : To test the special condition when input data are not the result of the hash table.

input

```
4
3 4 5 6
```

output

```
Error
```

Sample5 (empty table)

purpose : To test the special condition when table is empty.

input

```
1
-1
```

output

```
The table is empty.
```

Sample6 (size of table = 100 & number of digits = 95)

purpose: This is a comprehensive test, an example of our random tests.

input

```
100

3284 191 1702 3 1202 1602 1806 268 2408 8 878 6311 512 813 7514 215 4013 6417 5317 119 2318

3219 6222 2518 4614 925 1826 26 8080 29 230 309 77 -1 -1 8135 2335 1836 737 -1 2840 2740 1842

9343 3444 745 245 1647 148 249 2548 3945 2152 3152 4244 441 456 157 2958 2157 60 -1 762 563

-1 65 1966 1867 2966 267 4270 870 1872 3772 74 4570 1976 1968 3578 3467 2365 81 1682 882 484

883 3786 6587 1988 2589 7690 890 4590 2589 6194 395 1995 7597 4597 2799
```

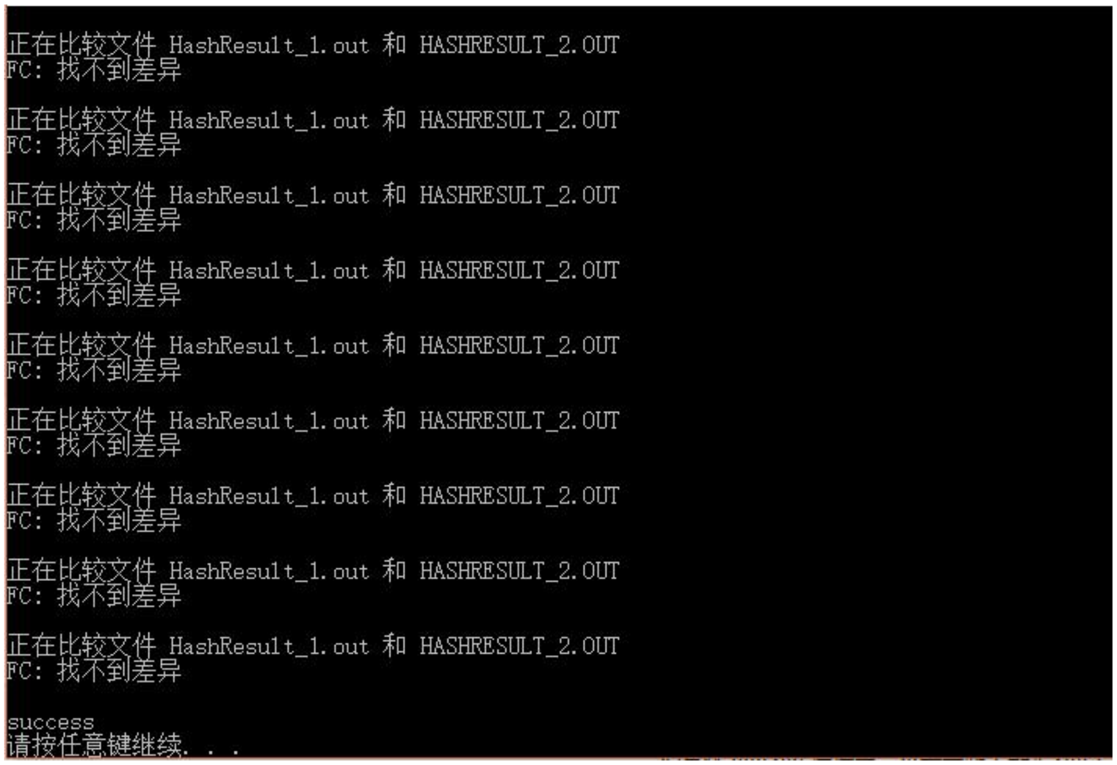
output

```
3 29 60 65 74 81 119 148 157 215 230 249 395 456 484 512 563 745 245 762 813 925 1647 1682

882 883 1702 1202 1602 1806 1826 26 1842 1867 1872 1966 1976 1988 1995 2152 2408 8 2548

2589 2799 2840 2740 2958 2157 2966 267 3152 3444 3578 3772 3786 3945 4244 4270 870 4570

1968 3467 2365 6194 6222 6311 6417 5317 2318 3219 2518 6587 7514 4013 4614 7597 4597 7690

890 4590 2589 3284 191 268 878 8080 309 77 8135 2335 1836 737 9343 441
```

3.3.2 Massive Data Test

Batch test (TableSize=1000 Number of Digits=950 cases=1000)

purpose: To confirm the correctness through a great number of test cases.



```
正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

正在比较文件 HashResult_1.out 和 HASHRESULT_2.OUT
FC: 找不到差异

success
请按任意键继续. . .
```

we tested 1000 cases of random hash input of hash result whose TableSize=1000 and NumberOfDights=950, and there existed no error. All the outputs matched the correct answer.

Chapter 4 Analysis and Comments

4.1 Analysis

4.1.1 Correctness

In the first part, we choose some special cases for testing, and we can see that all the outputs are correct. In the second part, we generate a great amount of random data, and all the outputs are the same to the standard answer. So, we can basically confirm that our program is correct.

4.1.2 Time Complexity

The program can be mainly divided into three parts: storing elements in the hash table, building the graph using adjacency list, and topological sorting to output the input sequence.

We temporarily use N stands for table size, V stands for number of nodes, E stands for number of edges.

- Storing elements in the hash table: only need 1 loop to store -> O(N)
- Building the list -> best=O(V) & worst=O(V^2)
  - Considering a worst case (for example):
    Table size=1000 Hash table: 0 1000 2000 3000 4000 4 5 ... 998
    The number of edges to build is V*(V-1)/2 -> O(V^2)
  - Considering a best case (for example):
    Table size=1000 Hash table: 0 1 2 ... 999
    We simply traverse each vertex -> O(V)
- Topological sorting -> O (VlogV+E)
  - Decrease the degree -> O(E)
    In topological sorting, we must decrease all the degree of vertices to zero, so we must traverse each edge for one time. The time complexity for decreasing-degree operation is O(E).
  - Insert and delete each vertex into heap -> O(VlogV)
    As for each vertex, it will be put in the heap for only once and delete from the heap for once. Insert one vertex into the heap and delete one vertex from the heap needs O(logV). So, the complexity for total insert and delete operation is O(VlogV).

Considering the best case: the graph is a chain. E=N and each insert operation takes only one move, then the time complexity will be O(N).

Considering the worst case: the table is full and there are edges between each pair of nodes. E=V(V-1)/2=N(N-1)/2, then the time complexity will be O(N^2).

4.1.3 Space Complexity

Several arrays to store the information for eachvertex->O(V).
A priority queue -> O(V)
Each node has an adjacency nodes list -> O(V^2)
As in the worst case, V = N. So, the Space Complexity is O(N^2).

4.2 Comments

(1) Use “malloc” function to allocate space for arrays may be better.
(2) We simply consider the error case that there exists a loop in the graph. If we go deeper, we can consider more condition that the input is illegal and can’t be earned by the hash function.

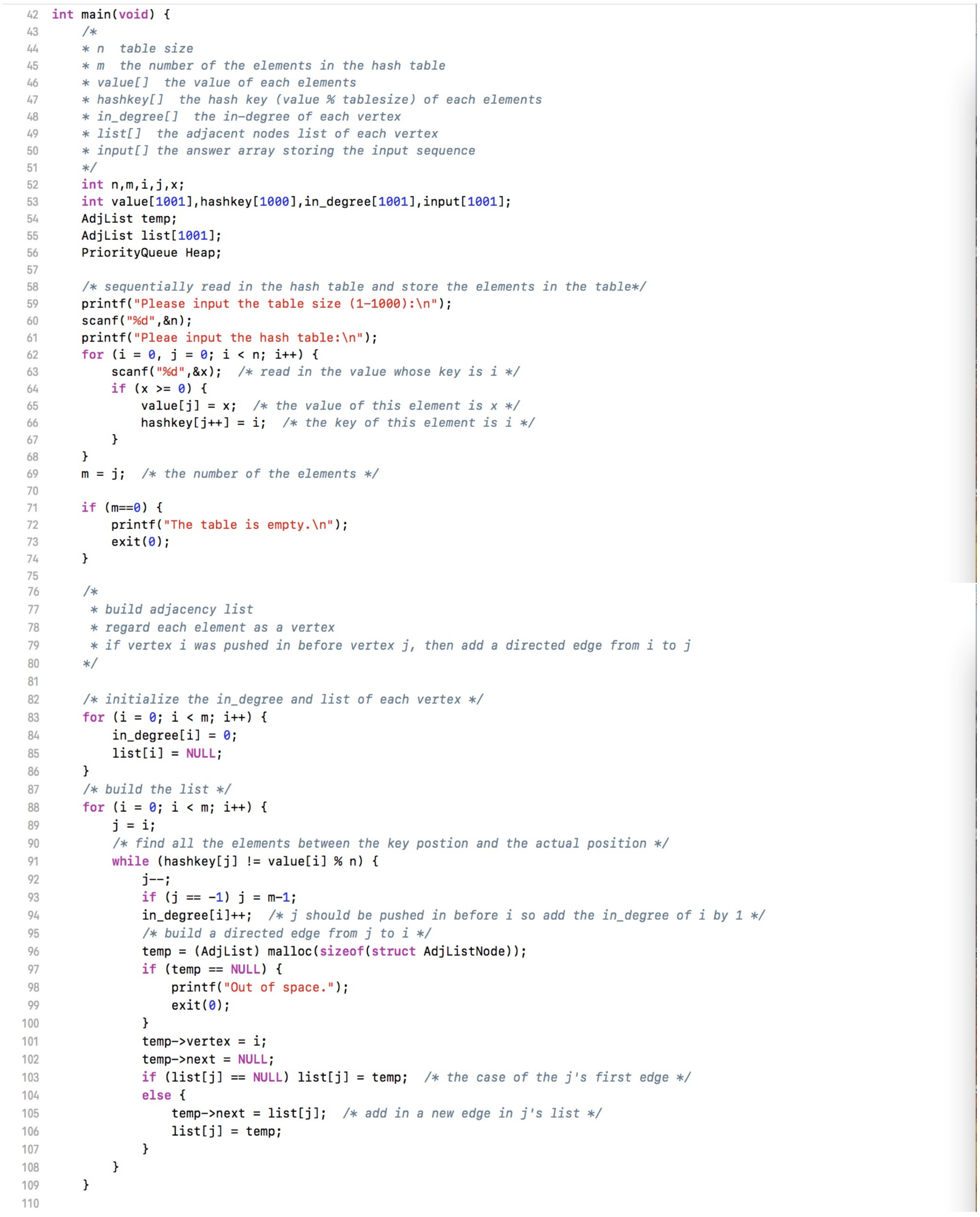
Appendix

Appendix A Source code

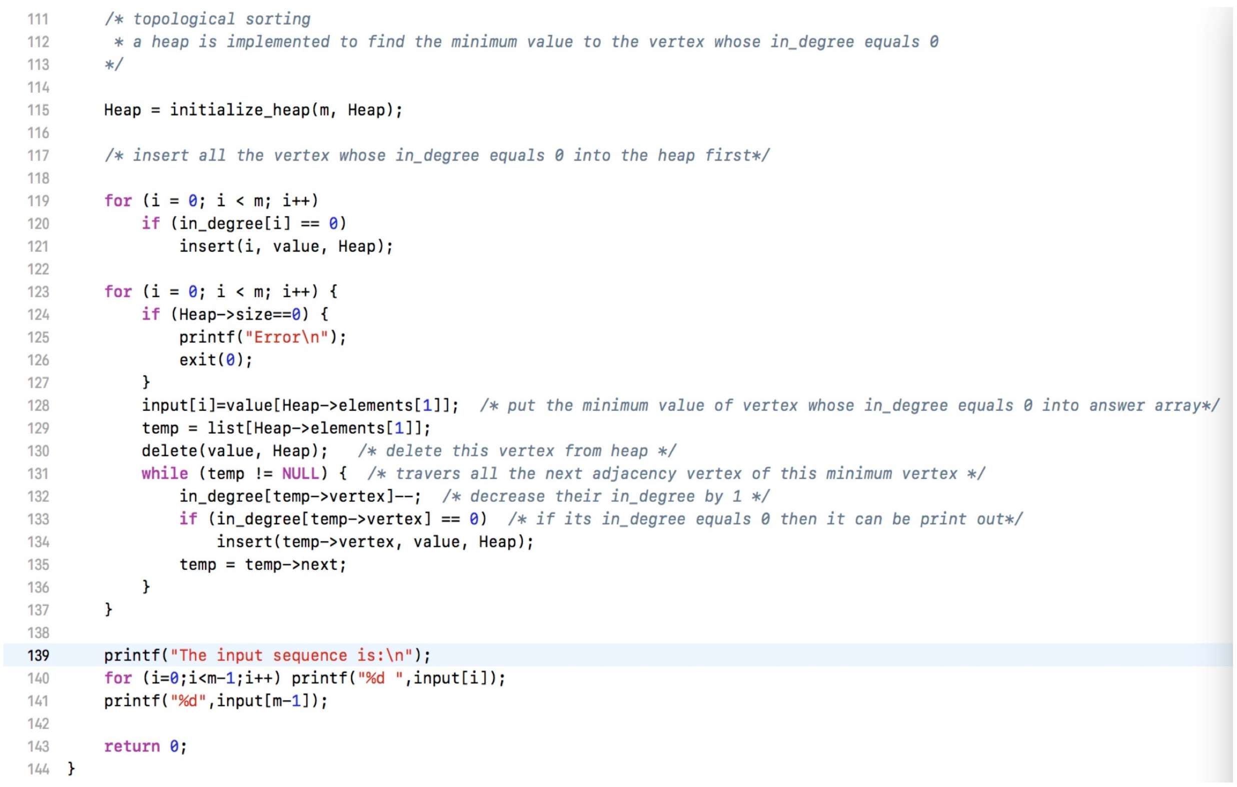
A.1 header

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

/* the struct of each node int the adjacency list*/
typedef struct AdjListNode {
    int vertex;
    struct AdjListNode* next;
} *AdjList;

/* the struct of binary heap used as a priority queue*/
typedef struct HeapStuct {
    int size,capacity;
    int *elements;
} *PriorityQueue;

/*
 * Description: This function is used to initialize the binary heap
 * Parameter[in] m  the number of vertices may be pushed in the heap
 * Parameter[in] H  the biranay heap without initialization
 * return H  the initialized heap
*/
PriorityQueue initialize_heap(int m,PriorityQueue H);


/*
 * Description: This function is used to insert a new element into the heap
 * Parameter[in] x  the vertex
 * Parameter[in] value[]  the value of vertices
 * Parameter[in] H  the biranay heap without initialization
*/
void insert(int x,int value[], PriorityQueue H);

/*
 * Description: This function is used delete the minimum value in the heap
 * Parameter[in] value[]  the value of vertices
 * Parameter[in] H  the biranay heap without initialization
*/
void delete(int value[],PriorityQueue H);

```

A.2 main function

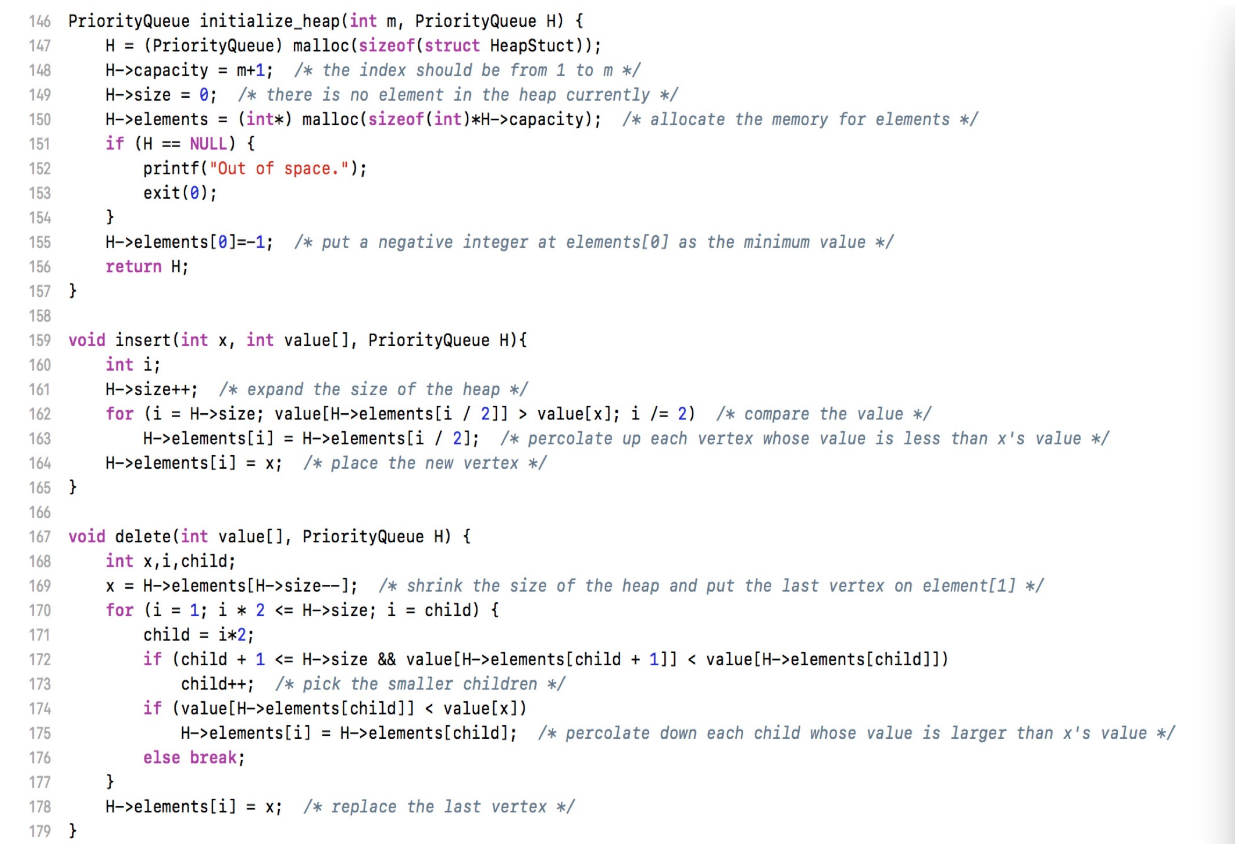


```c
int main(void) {
    /*
    * n  table size
    * m  the number of the elements in the hash table
    * value[]  the value of each elements
    * hashkey[]  the hash key (value % tablesize) of each elements
    * in_degree[]  the in-degree of each vertex
    * list[]  the adjacent nodes list of each vertex
    * input[] the answer array storing the input sequence
    */
    int n,m,i,j,x;
    int value[1001],hashkey[1000],in_degree[1001],input[1001];
    AdjList temp;
    AdjList list[1001];
    PriorityQueue Heap;

    /* sequentially read in the hash table and store the elements in the table*/
    printf("Please input the table size (1-1000):\n");
    scanf("%d",&n);
    printf("Pleae input the hash table:\n");
    for (i = 0, j = 0; i < n; i++) {
        scanf("%d",&x);  /* read in the value whose key is i */
        if (x >= 0) {
            value[j] = x;  /* the value of this element is x */
            hashkey[j++] = i;  /* the key of this element is i */
        }
    }
    m = j;  /* the number of the elements */

    if (m==0) {
        printf("The table is empty.\n");
        exit(0);
    }

    /*
     * build adjacency list
     * regard each element as a vertex
     * if vertex i was pushed in before vertex j, then add a directed edge from i to j
    */

    /* initialize the in_degree and list of each vertex */
    for (i = 0; i < m; i++) {
        in_degree[i] = 0;
        list[i] = NULL;
    }
    /* build the list */
    for (i = 0; i < m; i++) {
        j = i;
        /* find all the elements between the key postion and the actual position */
        while (hashkey[j] != value[i] % n) {
            j--;
            if (j == -1) j = m-1;
            in_degree[i]++;  /* j should be pushed in before i so add the in_degree of i by 1 */
            /* build a directed edge from j to i */
            temp = (AdjList) malloc(sizeof(struct AdjListNode));
            if (temp == NULL) {
                printf("Out of space.");
                exit(0);
            }
            temp->vertex = i;
            temp->next = NULL;
            if (list[j] == NULL) list[j] = temp;  /* the case of the j's first edge */
            else {
                temp->next = list[j];  /* add in a new edge in j's list */
                list[j] = temp;
            }
        }
    }

```



```c
    /* topological sorting
     * a heap is implemented to find the minimum value to the vertex whose in_degree equals 0
    */

    Heap = initialize_heap(m, Heap);

    /* insert all the vertex whose in_degree equals 0 into the heap first*/

    for (i = 0; i < m; i++)
        if (in_degree[i] == 0)
            insert(i, value, Heap);

    for (i = 0; i < m; i++) {
        if (Heap->size==0) {
            printf("Error\n");
            exit(0);
        }
        input[i]=value[Heap->elements[1]];  /* put the minimum value of vertex whose in_degree equals 0 into answer array*/
        temp = list[Heap->elements[1]];
        delete(value, Heap);   /* delete this vertex from heap */
        while (temp != NULL) {  /* travers all the next adjacency vertex of this minimum vertex */
            in_degree[temp->vertex]--;  /* decrease their in_degree by 1 */
            if (in_degree[temp->vertex] == 0)  /* if its in_degree equals 0 then it can be print out*/
                insert(temp->vertex, value, Heap);
            temp = temp->next;
        }
    }

    printf("The input sequence is:\n");
    for (i=0;i<m-1;i++) printf("%d ",input[i]);
    printf("%d",input[m-1]);

    return 0;
}
```

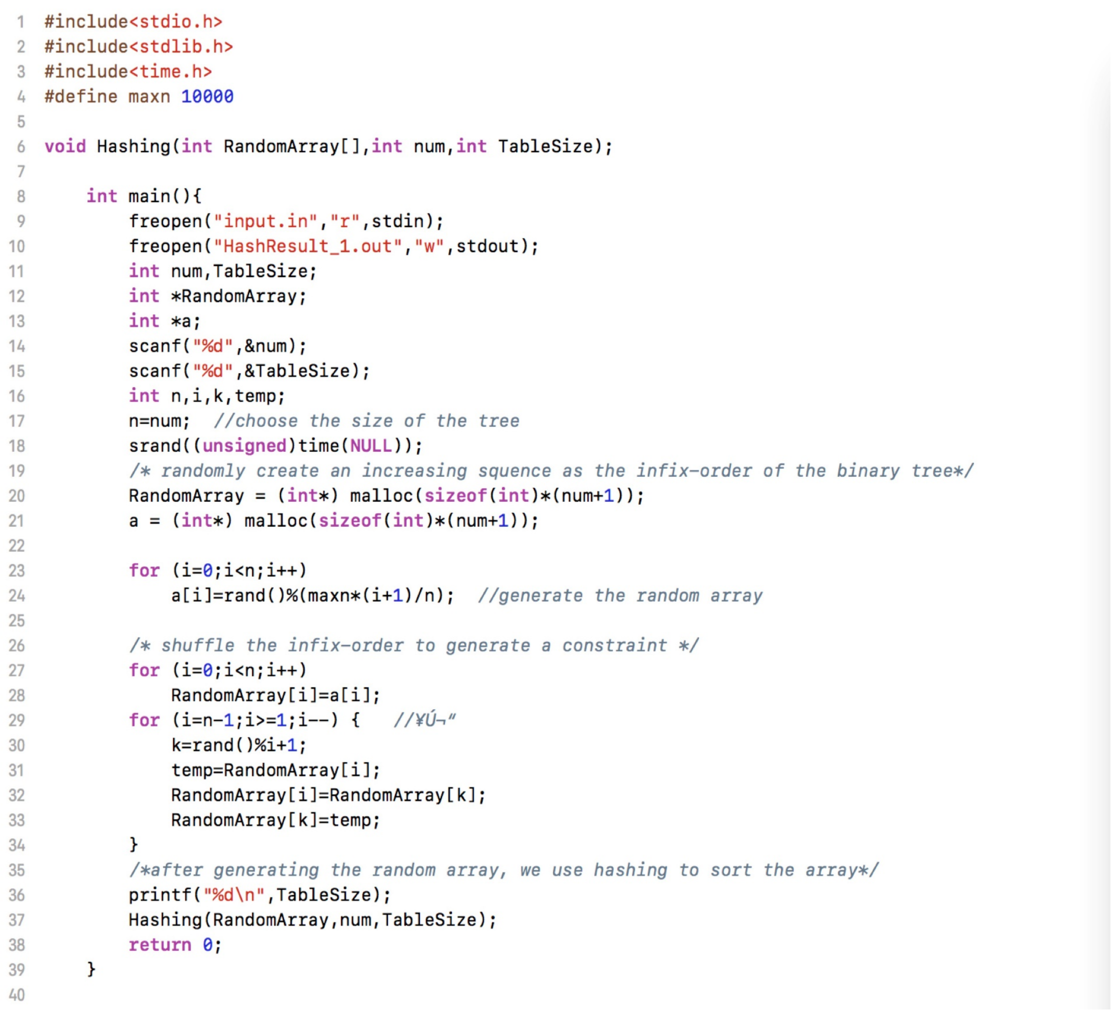
A.3 functions of heap



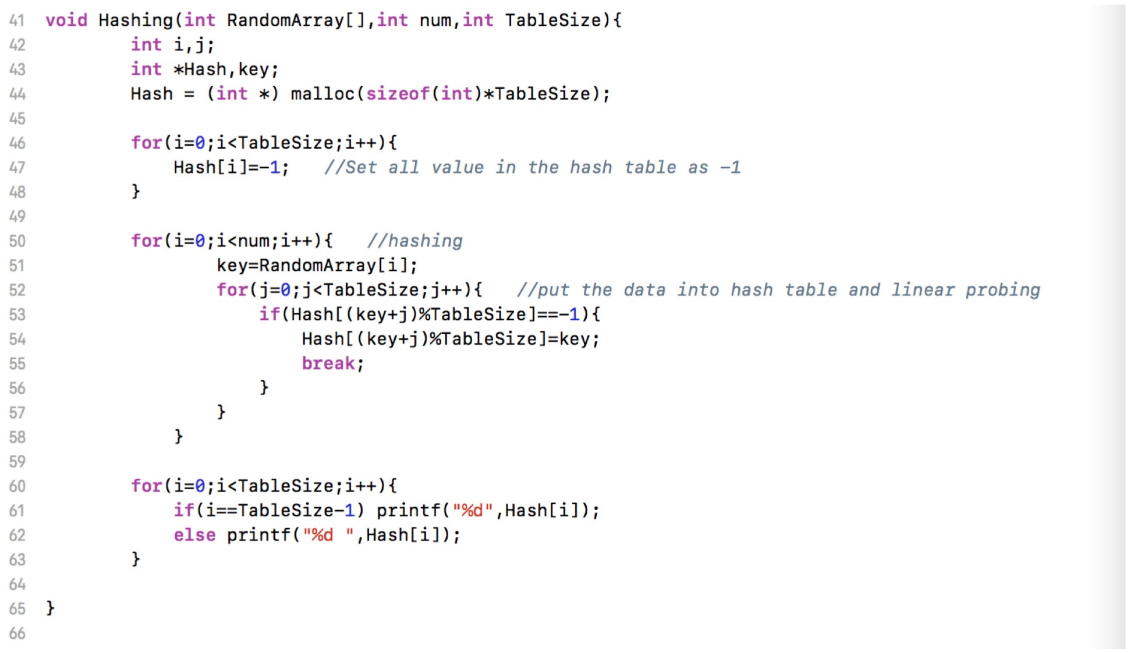
```c
PriorityQueue initialize_heap(int m, PriorityQueue H) {
    H = (PriorityQueue) malloc(sizeof(struct HeapStuct));
    H->capacity = m+1;  /* the index should be from 1 to m */
    H->size = 0;  /* there is no element in the heap currently */
    H->elements = (int*) malloc(sizeof(int)*H->capacity);  /* allocate the memory for elements */
    if (H == NULL) {
        printf("Out of space.");
        exit(0);
    }
    H->elements[0]=-1;  /* put a negative integer at elements[0] as the minimum value */
    return H;
}

void insert(int x, int value[], PriorityQueue H){
    int i;
    H->size++;  /* expand the size of the heap */
    for (i = H->size; value[H->elements[i / 2]] > value[x]; i /= 2)  /* compare the value */
        H->elements[i] = H->elements[i / 2];  /* percolate up each vertex whose value is less than x's value */
    H->elements[i] = x;  /* place the new vertex */
}

void delete(int value[], PriorityQueue H) {
    int x,i,child;
    x = H->elements[H->size--];  /* shrink the size of the heap and put the last vertex on element[1] */
    for (i = 1; i * 2 <= H->size; i = child) {
        child = i*2;
        if (child + 1 <= H->size && value[H->elements[child + 1]] < value[H->elements[child]])
            child++;  /* pick the smaller children */
        if (value[H->elements[child]] < value[x])
            H->elements[i] = H->elements[child];  /* percolate down each child whose value is larger than x's value */
        else break;
    }
    H->elements[i] = x;  /* replace the last vertex */
}
```

A.4 test function

A.4.1 Generate Random HashResult

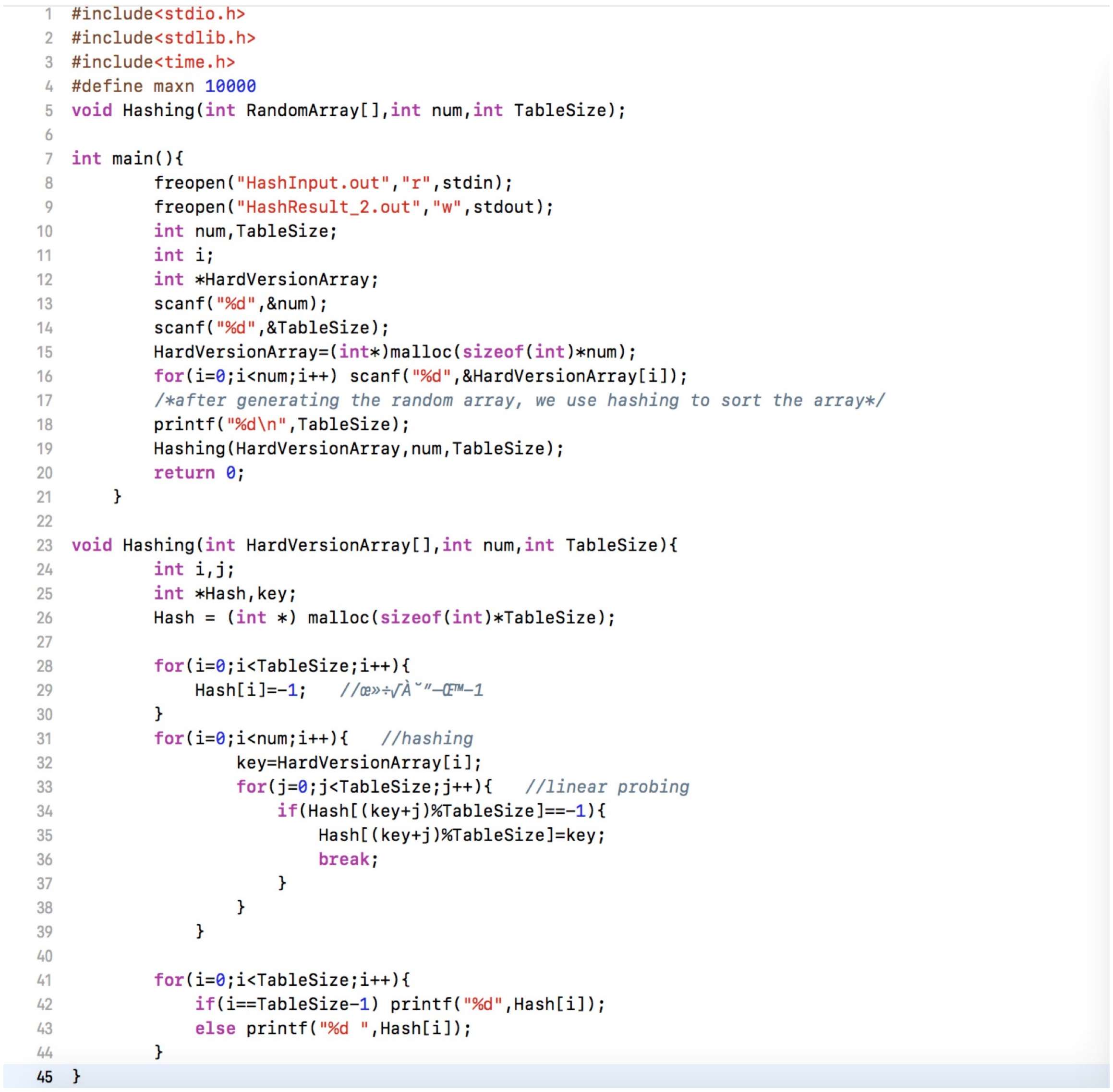


```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define maxn 10000

void Hashing(int RandomArray[],int num,int TableSize);

    int main(){
        freopen("input.in","r",stdin);
        freopen("HashResult_1.out","w",stdout);
        int num,TableSize;
        int *RandomArray;
        int *a;
        scanf("%d",&num);
        scanf("%d",&TableSize);
        int n,i,k,temp;
        n=num;  //choose the size of the tree
        srand((unsigned)time(NULL));
        /* randomly create an increasing squence as the infix-order of the binary tree*/
        RandomArray = (int*) malloc(sizeof(int)*(num+1));
        a = (int*) malloc(sizeof(int)*(num+1));

        for (i=0;i<n;i++)
            a[i]=rand()%(maxn*(i+1)/n);  //generate the random array

        /* shuffle the infix-order to generate a constraint */
        for (i=0;i<n;i++)
            RandomArray[i]=a[i];
        for (i=n-1;i>=1;i--) {   //¥Ú¬“
            k=rand()%i+1;
            temp=RandomArray[i];
            RandomArray[i]=RandomArray[k];
            RandomArray[k]=temp;
        }
        /*after generating the random array, we use hashing to sort the array*/
        printf("%d\n",TableSize);
        Hashing(RandomArray,num,TableSize);
        return 0;
    }
```



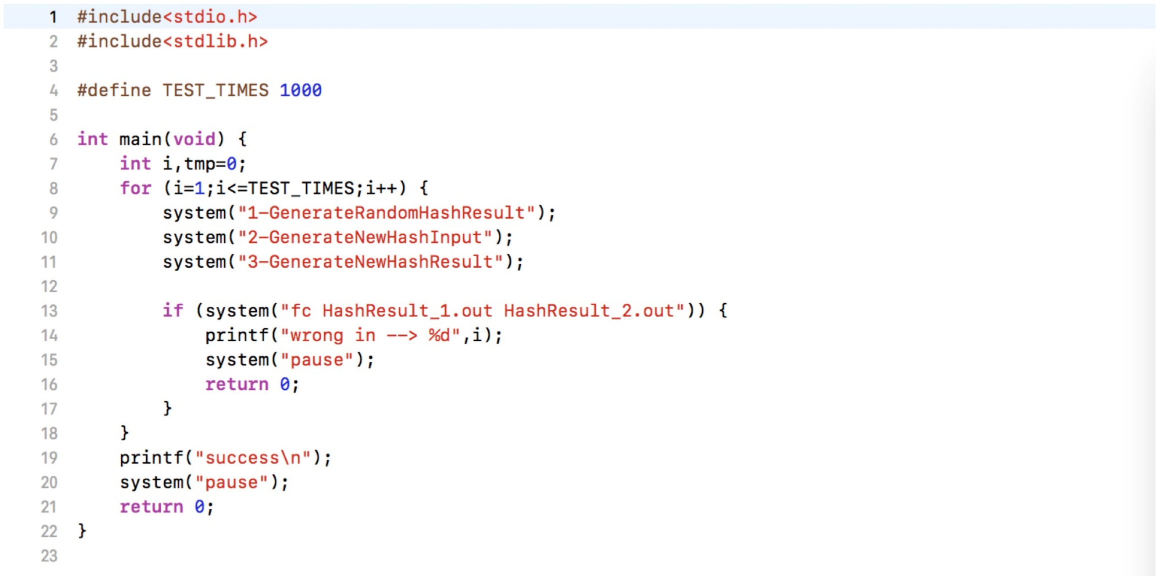
```c
void Hashing(int RandomArray[],int num,int TableSize){
        int i,j;
        int *Hash,key;
        Hash = (int *) malloc(sizeof(int)*TableSize);

        for(i=0;i<TableSize;i++){
            Hash[i]=-1;   //Set all value in the hash table as -1
        }

        for(i=0;i<num;i++){   //hashing
                key=RandomArray[i];
                for(j=0;j<TableSize;j++){   //put the data into hash table and linear probing
                    if(Hash[(key+j)%TableSize]==-1){
                        Hash[(key+j)%TableSize]=key;
                        break;
                    }
                }
            }

        for(i=0;i<TableSize;i++){
            if(i==TableSize-1) printf("%d",Hash[i]);
            else printf("%d ",Hash[i]);
        }

}
```

A.4.2 Generate New HashResult



```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define maxn 10000
void Hashing(int RandomArray[],int num,int TableSize);

int main(){
        freopen("HashInput.out","r",stdin);
        freopen("HashResult_2.out","w",stdout);
        int num,TableSize;
        int i;
        int *HardVersionArray;
        scanf("%d",&num);
        scanf("%d",&TableSize);
        HardVersionArray=(int*)malloc(sizeof(int)*num);
        for(i=0;i<num;i++) scanf("%d",&HardVersionArray[i]);
        /*after generating the random array, we use hashing to sort the array*/
        printf("%d\n",TableSize);
        Hashing(HardVersionArray,num,TableSize);
        return 0;
    }

void Hashing(int HardVersionArray[],int num,int TableSize){
        int i,j;
        int *Hash,key;
        Hash = (int *) malloc(sizeof(int)*TableSize);

        for(i=0;i<TableSize;i++){
            Hash[i]=-1;    //œ»÷√À˘”–Œ™–1
        }
        for(i=0;i<num;i++){    //hashing
                key=HardVersionArray[i];
                for(j=0;j<TableSize;j++){    //linear probing
                    if(Hash[(key+j)%TableSize]==-1){
                        Hash[(key+j)%TableSize]=key;
                        break;
                    }
                }
            }

        for(i=0;i<TableSize;i++){
            if(i==TableSize-1) printf("%d",Hash[i]);
            else printf("%d ",Hash[i]);
        }
}
```

A.4.3 compare



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

#define TEST_TIMES 1000

int main(void) {
    int i,tmp=0;
    for (i=1;i<=TEST_TIMES;i++) {
        system("1-GenerateRandomHashResult");
        system("2-GenerateNewHashInput");
        system("3-GenerateNewHashResult");

        if (system("fc HashResult_1.out HashResult_2.out")) {
            printf("wrong in --> %d",i);
            system("pause");
            return 0;
        }
    }
    printf("success\n");
    system("pause");
    return 0;
}

```

Appendix B Declaration

We hereby declare that all the work done in this project titled “hashing, hard version” is of our independent effort as a group.

Duty Assignments

Programmer： Yizhou Chen

Tester： Jinze Wu

Reporter： Jie Feng