Project3
Hashing – Hard Version

Author Names:
Group 2

Date: 2018-12-21

CONTENTS

Chapter 1: Introduction

1.1 Background description

A **hash table** is a data structure that can map keys to values. With a specific hash function, it computes an index into an array of slots, from which the desired value can be found. However, it might cause collisions where the hash function generates the same index for more than one key. To deal with that, we could use open addressing with linear probing, in which the interval between probes is 1.
Topological sort is a linear ordering of vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. A topological ordering is possible if and only if the graph has no directed cycles.

1.2 Problem description

We are given a hash table of size N. Suppose that the linear probing is used to solved collisions, we are supposed to reconstruct the input sequence from the given status of the hash table. If there are multiple choices, just print the smallest sequence.
After analysis, we found that the key of reconstructing is find collisions. When a collision occurs, it requires X to be inserted after Y. Reflecting the relationship in a directed graph, we build an edge from Y to X. Thus, we convert the problem to Topological sort.

Chapter 2: Algorithm Specification

In this chapter we'll show our main ideas and steps by pseudo-code.

TopSort(G)

Require: G is a directed acyclic graph (DAG)

```
1:  function TopSort(G)
2:      T ← empty list
3:      Z ← empty queue/stack/whatever
4:      in ← dictionary mapping all vertices to 0
5:      for each v ∈ V do
6:          for each u adjacent to v do
7:              increment in[v]
8:      for each v ∈ V do
9:          if in[v] = 0 then
10:             add v to Z
11:     while S is not empty do
12:         v ← Z.remove
13:         append v to T
14:         for each u adjacent to v do
15:             decrement in[u]
16:             if in[u] = 0 then
17:                 add u to Z
        return T
```

Chapter 3: Testing Results

In this chapter we will show the correctness of our algorithm and the robustness of our program by some representative and identifiable test cases.
To generate reliable test cases, we build a data maker which provides N-size valid input randomly. For the results our program has given, we construct a test program to check. The program performs well in the testing. We confirm it works out the accurate answers.
Here we will list 6 representative test cases which covers most conditions. Details and results of these test cases are as follows.

3.1 Details of test cases

Here we will list 6 representative test cases which covers most conditions. Details and results of these test cases are as follows.

3.1.1 Test Case 1

Test ID: 1

Test File: TestCase1.dat

Related Condition: Minimum N case

Input:

1

2

Output:

2

Testing Result:

PASS

3.1.2 Test Case 2

Test ID: 2

Test File: TestCase2.dat

Related Condition: An empty Hash table

Input:

5

-1 -1 -1 -1 -1

Output:

Testing Result:

PASS

3.1.3 Test Case 3

Test ID: 3

Test File: TestCase3.dat

Related Condition: Sample in PTA

Input:

11

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

Output:

1 13 12 21 33 34 38 27 22 32

Testing Result:

PASS

3.1.4 Test Case 4

Test ID: 4

Test File: TestCase4.dat

Related Condition: All the input numbers have collision except the first one

Input:

100

99 101 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

Output:

101 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

Testing Result:

PASS

3.1.5 Test Case 5

Test ID: 5

Test File: TestCase5.dat

Related Condition: Random case

Input:

Output:

Due to page limitation, we don't list the data here. You could find the files in the data folder.

Testing Result:

PASS

3.1.6 Test Case 6

Test ID: 6

Test File: TestCase6.dat

Related Condition: Maximum N case

Input:

Output:

Due to page limitation, we don't list the data here. You could find the files in the data folder.

Testing Result:

PASS

3.1.7 PTA Test Result

The correctness of our program was also verified by the PTA system

测试点	结果	耗时	内存
0	答案正确	1 ms	276KB
1	答案正确	1 ms	384KB
2	答案正确	2 ms	256KB
3	答案正确	1 ms	256KB
4	答案正确	3 ms	640KB

代码

```
268     HashTable H = malloc(sizeof(struct HashTbl));
269     H->TableSize = TableSize;
270     H->Elements = malloc(sizeof(int)*H->TableSize);
271     for( i = 0; i < H->TableSize; i++ )
272         scanf("%d", &H->Elements[i]);
273     return H;
274 }
275
276
277 int main(){
278
279     int i;
280     int N;
281     scanf("%d", &N);
282
283     HashTable H = InitHashTable( N );
284
285     AdjGraph G =  BuildAdjGraph(H);
286
287     TopSort( G );
288
289     return 0;
290 }
291
```

Chapter 4: Analysis and Comments

4.1 Time and space complexity analysis

4.1.1 Build a graph from the given hash table

Firstly, we scan N times to initialize the adjacency list. So the time complexity of this step is O(N). After that, every vertex in the graph represents an element in the hash table.

Then, we scan N times to calculate the indegree of every vertex *v*. Specifically, we run the hash function to get its position without collision (this operation costs constant time). Compare with the current position, we get its indegree. For every element *u* between the hash position and the insert position, we set an edge from the current vertex *v* to *u*. Since the collision times are constant, we could say the time complexity of building algorithm is O(N).

```
AdjGraph BuildAdjGraph( HashTable H ){
    int i, j;
    int N;
    /* Allocate the memory the adjacency list */
    AdjGraph G = malloc( sizeof( struct AdjGph ) );
    N = G->Nv = H->TableSize;
    G->List =  malloc( sizeof( struct VNode ) * N );
    /* Initialize the adjacency list */
    for( i = 0; i < N; i++ ){
        G->List[ i ].Element = H->Elements[ i ];
        if( G->List[ i ].Element < 0 )
            G->List[ i ].Indegree = EmptyCell;    /* Assign the Indegree the empty cell with -1 */
        else
            G->List[ i ].Indegree = 0;       /* Initialize the Indegree as 0 */
        G->List[ i ].FirstEdge = NULL;    /* Store the adjacent vertex using linked list with header
*/
    }


    for( i = 0; i < N; i++ ){
        if( G->List[ i ].Element < 0 )    /* Skip the empty cell */
            continue;
        int HashPos = Hash( G->List[ i ].Element, H->TableSize );   /* Calculate the position
without collision */
        G->List[ i ].Indegree = (i - HashPos + N) % N;      /* Calculate the conflict time and set
the indegree of the vertex */
        for( j = 0; j < G->List[ i ].Indegree; j++ ){      /* set the i-th vertex adjacent to the
conflict vertex */
            int InsertPos = (HashPos + j + N) % N; /* Calculate the position of conflict vertex */
            PtrToAdjVNode P, InsertNode;
            /* Initialize the node to be inserted */
            InsertNode = malloc( sizeof( struct AdjVNode ) );
```

```
            InsertNode->AdjV = i;
            InsertNode->Next = NULL;
            /* Insert the node into the adjacency list */
            P = G->List[ InsertPos ].FirstEdge;
            if( P == NULL )
                G->List[ InsertPos ].FirstEdge = InsertNode;
            else {
                while( P->Next != NULL )
                    P = P->Next;
                P->Next = InsertNode;
            }
        }
    }
    /* free the memory */
    free( H->Elements );
    free( H );

    return G;
}
```

According to the code, we build a adjacency list to store the graph, so the space complexity of this part is O(N+M), where M stands for the number of edges. Specifically in the algorithm, it depends on the collision times. If there are few collsions and the graph is sparse, the space complexity is close to O(N).

4.1.2 Topological sort

Firstly, we scan N times to find the 0-indegree vertexes and put them into a priority queue. So the time complexity of this step is O(N).

For the next step, we do the work until the priority queue is empty. It is obvious that each node will be visited only once. And the time cost of visit procedure only depends on the indegree of the vertex. We conclude that the time complexity of this step is O(N+K), K is a const number.

```
void TopSort( AdjGraph G )
{
    int i;
    int flag = 0;   /* flag to control the space output format */
    PriorityQueue H;
    Vertex V;
    PtrToAdjVNode  P;

    H = Initialize( G->Nv );    /* Initialize the Priority-queue for topological sort */

    for( i = 0; i < G->Nv; i++ )
```

```
    {
        if( G->List[ i ].Indegree == 0 )    /* If a vertex's indegree is 0, put it into the priority
queue */
            Insert( G->List[ i ].Element, i, H );
    }

    while( !IsEmpty( H ) )
    {
        V = DeleteMin( H ); /* Pop the current minimum element in the queue */
        /* Output in the required format */
        if( flag == 1 )
            printf(" ");
        printf("%d",G->List[V].Element);
        flag = 1;

        /* Traverse the adjacency list of the current node and decrements the node indegree by one.
         * If the node indegree becomes zero, insert it into the priority queue */
        P = G->List[ V ].FirstEdge;
        while( P != NULL )
        {
            if( --(G->List[ P->AdjV ].Indegree) == 0 )
                Insert( G->List[ P->AdjV ].Element, P->AdjV, H );
            P = P->Next;
        }
    }

    /* free the memory */
    Destroy( H );
    free( G->List );
    free( G );
}
```

According to the code, we initialize a priority queue in this part, which assign a N-size array. So the space complexity of this part is O(N).

4.1.3 Total program

In conclusion, the time complexity of the total program is O(N)
The space complexity of the total program is O(N+M)

4.2 Comments

4.2.1 Accuracy

The algorithm can deal with all the test cases provided and give correct answers.

4.2.2 Readability

The source codes have a lot of comments to help reading. Besides, the use of Macro substitution makes it easier to understand the whole program. Furthermore, the test code, algorithm code and the header file are all separate, which facilitates quick browsing and finding.

It is worth mentioning that the programmer used encapsulation to bundle the priority queue with the methods that operate on the data. This is quite standard and user-friendly.

4.2.3 Robustness

The program can deal with invalid inputs and some special cases. No bugs found yet.

In a word, the code is clear and concise. The program is reliable and effective. The algorithm is elegant and efficient. Of course, there are still some parts that can be improved. We will work hard to find and improve.

Appendix: Source Code (in C)

main.c

```c
#include "Reconstruct.h"


int main(){
    int N;

    scanf( "%d", &N );  /* Input the Size of the hash table */

    HashTable H = InitHashTable( N );   /* Construct the hash table from the input */

    AdjGraph G =  BuildAdjGraph( H );   /* Convert hash table into adjacency list */

    TopSort( G );   /* Topological sort and output */

    return 0;
}
```

Reconstruct.h

```c
#ifndef RECONSTRUCT_H
#define RECONSTRUCT_H
#include "BinHeap.h"


#define EmptyCell (-1)


typedef int Vertex;
typedef struct AdjVNode *PtrToAdjVNode;
typedef struct VNode *AdjList;
typedef struct AdjGph *AdjGraph;
typedef struct HashTbl *HashTable;


void TopSort( AdjGraph G );
int Hash( int Key, int TableSize );
AdjGraph BuildAdjGraph( HashTable H );
HashTable InitHashTable( int TableSize );


#endif
```

Reconstruct.c

```c
#include "Reconstruct.h"

/* Node of the adjacency list */
struct AdjVNode{
    Vertex AdjV;    /* Index of the adjacent Vertex */
    PtrToAdjVNode Next;
};
/* Header of the adjacency list */
struct VNode{
    int Element;    /* Value of the vertex */
    int Indegree;   /* Indegree of the vertex */
    PtrToAdjVNode FirstEdge;
};
/* Definition of the adjacency list*/
struct AdjGph{
    AdjList List;
    int Nv;     /* Number of vertex */
};
/* Definition of the hash table */
struct HashTbl{
    int *Elements;
    int TableSize;
};

/**
 * Construct the hash table from the input
 * @param TableSize: the size of the hash table
 * @return the hash table generated from input
 */
HashTable InitHashTable( int TableSize ){
    int i;
    /* Allocate the memory*/
    HashTable H = malloc( sizeof( struct HashTbl ) );
    H->TableSize = TableSize;
    H->Elements = malloc( sizeof(int) * H->TableSize );

    /* Assign the element from the input */
    for( i = 0; i < H->TableSize; i++ )
        scanf( "%d", &H->Elements[i] );
    return H;
```

```
}
/**
 * Convert hash table into adjacency list
 * @param H: hash table to be reconstructed
 * @return converted adjacency list
 */
AdjGraph BuildAdjGraph( HashTable H ){
    int i, j;
    int N;
    /* Allocate the memory the adjacency list */
    AdjGraph G = malloc( sizeof( struct AdjGph ) );
    N = G->Nv = H->TableSize;
    G->List =  malloc( sizeof( struct VNode ) * N );
    /* Initialize the adjacency list */
    for( i = 0; i < N; i++ ){
        G->List[ i ].Element = H->Elements[ i ];
        if( G->List[ i ].Element < 0 )
            G->List[ i ].Indegree = EmptyCell;    /* Assign the Indegree the empty cell with -1 */
        else
            G->List[ i ].Indegree = 0;       /* Initialize the Indegree as 0 */
        G->List[ i ].FirstEdge = NULL;    /* Store the adjacent vertex using linked list with header
*/
    }
    for( i = 0; i < N; i++ ){
        if( G->List[ i ].Element < 0 )    /* Skip the empty cell */
            continue;
        int HashPos = Hash( G->List[ i ].Element, H->TableSize );   /* Calculate the position
without collision */
        G->List[ i ].Indegree = (i - HashPos + N) % N;      /* Calculate the conflict time and set
the indegree of the vertex */
        for( j = 0; j < G->List[ i ].Indegree; j++ ){      /* set the i-th vertex adjacent to the
conflict vertex */
            int InsertPos = (HashPos + j + N) % N; /* Calculate the position of conflict vertex */
            PtrToAdjVNode P, InsertNode;
            /* Initialize the node to be inserted */
            InsertNode = malloc( sizeof( struct AdjVNode ) );
            InsertNode->AdjV = i;
            InsertNode->Next = NULL;
            /* Insert the node into the adjacency list */
            P = G->List[ InsertPos ].FirstEdge;
            if( P == NULL )
                G->List[ InsertPos ].FirstEdge = InsertNode;
```

```
            else {
                while( P->Next != NULL )
                    P = P->Next;
                P->Next = InsertNode;
            }
        }
    }

    /* free the memory */
    free( H->Elements );
    free( H );
    return G;
}

/* Hash function */
int Hash( int Key, int TableSize )
{
    return Key%TableSize;
}

/**
 * Topological sort and output
 * @param G Adjacency list to be topological-sorted
 */
void TopSort( AdjGraph G )
{
    int i;
    int flag = 0;   /* flag to control the space output format */
    PriorityQueue H;
    Vertex V;
    PtrToAdjVNode  P;
    H = Initialize( G->Nv );    /* Initialize the Priority-queue for topological sort */

    for( i = 0; i < G->Nv; i++ )
    {
        if( G->List[ i ].Indegree == 0 )    /* If a vertex's indegree is 0, put it into the priority
queue */
            Insert( G->List[ i ].Element, i, H );
    }

    while( !IsEmpty( H ) )
    {
```

```
        V = DeleteMin( H ); /* Pop the current minimum element in the queue */
        /* Output in the required format */
        if( flag == 1 )
            printf(" ");
        printf("%d",G->List[V].Element);
        flag = 1;

        /* Traverse the adjacency list of the current node and decrements the node indegree by one.
         * If the node indegree becomes zero, insert it into the priority queue */
        P = G->List[ V ].FirstEdge;
        while( P != NULL )
        {
            if( --(G->List[ P->AdjV ].Indegree) == 0 )
                Insert( G->List[ P->AdjV ].Element, P->AdjV, H );
            P = P->Next;
        }
    }

    /* free the memory */
    Destroy( H );
    free( G->List );
    free( G );
}
```

BinHeap.h

```
#ifndef BINHEAP_H
#define BINHEAP_H

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

#define MinData (-1)
#define MinPQSize (1)

struct HeapStruct;
typedef struct HeapStruct *PriorityQueue;

PriorityQueue Initialize( int MaxElements );
void Destroy( PriorityQueue H );
void MakeEmpty( PriorityQueue H );
void Insert( int X, int Index, PriorityQueue H );
```

```
int DeleteMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
int IsFull( PriorityQueue H );



#endif
```

BinHeap.c

```
#include "BinHeap.h"
typedef struct Element
{
    int Index;
    int Key;
}ElementType;
struct HeapStruct
{
    int Capacity;
    int Size;
    ElementType *Elements;
};

PriorityQueue Initialize( int MaxElemnets )
{
    PriorityQueue H;
    if( MaxElemnets < MinPQSize ){
        printf( "Priority queue size is too small" );
        exit( 1 );
    }

    H = malloc( sizeof( struct HeapStruct ) );
    if( H == NULL ){
        printf( "Out of Space!!!" );
        exit( 1 );
    }

    H->Elements = malloc( (MaxElemnets + 1) * sizeof(ElementType));

    if( H->Elements == NULL ){
        printf( "Out of Space!!!" );
        exit( 1 );
    }
```

```
    H->Capacity = MaxElemnets;
    H->Size = 0;
    H->Elements[ 0 ].Key = MinData;
    H->Elements[ 0 ].Index = -1;

    return H;
}

int DeleteMin( PriorityQueue H ){
    int i, Child;
    ElementType MinElement, LastElement;

    if( IsEmpty( H ) ){
        printf( "Priority queue is empty" );
        return H->Elements[ 0 ].Index;
    }

    MinElement = H->Elements[ 1 ];
    LastElement = H->Elements[ H->Size-- ];

    for( i = 1; i * 2 <= H->Size; i = Child ){
        /* Find Smaller child */
        Child = i * 2;
        if( Child != H->Size && H->Elements[ Child +1 ].Key < H->Elements[ Child ].Key )
            Child++;

        /* Percolate one level */
        if( LastElement.Key > H->Elements[ Child ].Key )
            H->Elements[ i ] = H->Elements[ Child ];
        else
            break;
    }
    H->Elements[i] = LastElement;
    return MinElement.Index;

}

void Insert( int X, int Index, PriorityQueue H )
{
    int i;
    if( IsFull( H ) ){
```

```
        printf( "Priority queue is full" );
        return;
    }

    for( i = ++H->Size; H->Elements[ i / 2 ].Key > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];

    H->Elements[i].Key = X;
    H->Elements[i].Index = Index;
}

int IsEmpty( PriorityQueue H )
{
    return H->Size == 0;
}

int IsFull( PriorityQueue H )
{
    return H->Size == H->Capacity;
}

void MakeEmpty( PriorityQueue H ){
    H->Size = 0;
}

int FindMin( PriorityQueue H ){
    return H->Elements[ 1 ].Index;
}

void Destroy( PriorityQueue H ){
    free( H->Elements );
    free( H );
}
```

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: 章含挺
Tester：柴子炜
Report writer: 章含挺/柴子炜