Hashing-Hard Version

林家丰 郭泽均 余瑞璟

Date: 2018-12-28

Chapter 1: Introduction

1. 1. Problem Description

In this project, we are asked to implement the inverse process of Hashing. With a given status of the hash table which is created by linear probing, we must reconstruct the input sequence. What’s more, when faced with multiple choices in printing the keys, the one with smallest value is taken.

1.2. Detailed Requirements of Input and Output

Input:

1. Input a positive integer N(N<=1000) as the size of the hash table.

2. Input N integers according to the given hash table, separated by a space, where a negative integer represents an empty cell in the hash table. Every non-negative integer is distinct.

Output:

1. Produce a program that solves the problem and then output the input sequence in a line, with numbers separated by a space.

1.3. Algorithm Background

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. The mapping function is called a Hash Function.

Linear probing is a scheme in programming for resolving collisions in hash tables, characterized by a linear function of i, typically F (i)=i. Also, stated in requirements of the specific problem of Hashing-Hard Version, linear probing is applied.

Using relative methods, we can easily obtain the status of the hash table with a given sequence of input numbers.

Chapter 2: Algorithm Specification

2.1. Sketch of Main Program

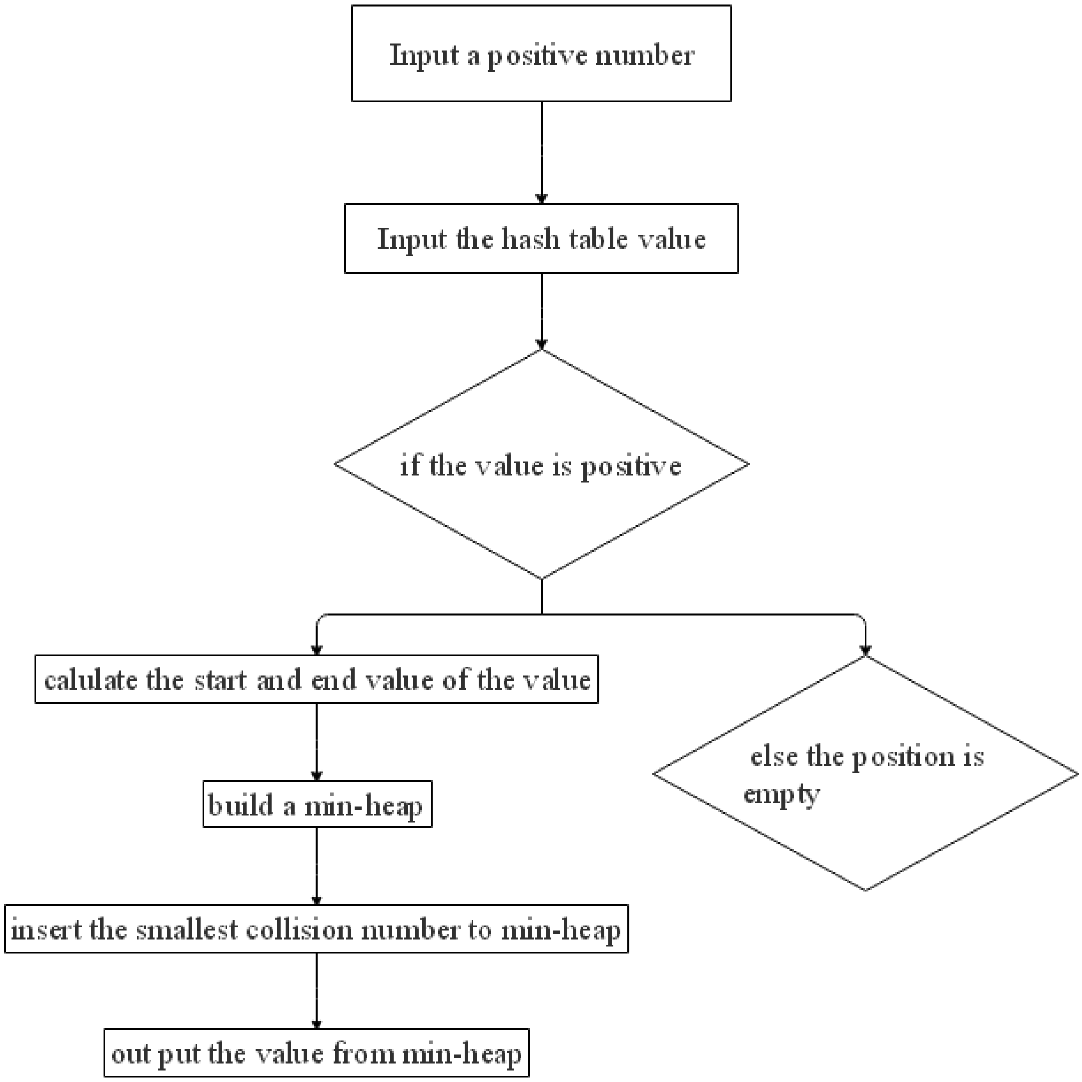
We solve the problem in three steps.

1. Read the input file from input stream and put them into a struct array called slot. And make a brief processing of the input data.

2. We build a heap using the method of percolate up to build a min-heap.

3. If collision happens in the insert sequence, we simulate the insert order and each time we output the smallest of the current situation (each time we insert the numbers which have different collision times). Else, we out put the numbers directly.

2.2. Flow Chart



2.3. Pseudo Code

2.3.1. Input

```
procedure READ_INPUT_STREAM
num <- read from input data
initial the input data to see whether they have collision when put into
the hash table, if the start position is the same with the end then
insert it into the min-heap
loop num times to read input data
    slot[index].value <- from input data
    if slot[index].value < 0 then the space is empty
    else
        slot[index].start <- slot[index].value % num
        slot[index].end <- index
        slot[index].count <- 0
        if slot[index].start == slot[index].end
            slot[index].flag <- 1
            heap.array[heap.size]=slot[index]
            heap.size++
            count++
        else
            slot[index].flag = 0
```

2.3.2. Build min-heap

```
procedure BUILD_MIN_HEAP
loop pos = heap->size / 2 - 1 when pos > 1 each time -1
    percolate down heap position pos
    use standard percolate down way to build a min-heap
```

2.3.3. Out put

```
procedure OUT_PUT
loop until all collision are solved in the hash table
    get the smallest number from min-heap
    loop all numbers in slot
        insert the present times of collisions of numbers into the min-
heap
    out put
```

2.4. Data structure

2.4.1. min-heap ADT

When we insert numbers into the min-heap we use a way call percolate down to make sure that the list satisfies ki <= k2i, ki <= k2i+1 for any ki in the min-heap. It can help us to always get the smallest number in the current situation. Because we insert every kind of collision situation which can make sure we won't traverse the hash table in a wrong order.

2.4.2. list ADT

The list ADT is mainly used to store the hash table and the min-heap. We combined list with structure which give us a lot of flag data to define the data situation. The technique is quite convenient and efficient.

2.5. Algorithm

2.5.1. The algorithm used to process the input

We use cycle and branch to process the hash table to give a brief status of its structure to make further process easier.

1. We calculate the hash value of each positive input number
2. Store the position of the number in the hash table.
3. Initial the count member of the number to define the collision times

2.5.2. The algorithm used to store current data

We use min-heap to store the number structure. And percolate down them to build a min-heap. Every time we read a number, percolate down the list again to make sure the

2.6. Correctness of algorithm

1. Every time we read a number from heap, we percolate down it again, which ensure us to get correct number each time we output.

2. When the first time we output a number. It must be the smallest and have no collision one, so it must be the first input number.

3. every time we call the insert function to insert a number. It must have just one more collision time than the last time. So the next time when we get a number from the min-heap it must be the smallest and won't be printed before the number which is smaller than it and have less collision time than it.

In all, these algorithms make sure the program can output number in current order.

Chapter 3: Testing Results

3.1. Environment

Processor: intel i7-7500U

System: Windows 10.0.17134

Memory: 8 Gigabyte

3.2. Test Case

Test case	Input N	Input Hash Table	Output	Current status
PTA case	11	33 1 13 12 34 38 27 22 32 -1 21	1 13 12 21 33 34 38 27 22 32	pass
Empty case	1	-1		pass
Smallest case	1	2	2	pass
Elements conflict in one cell	7	8 50 29 22 36 15 64	50 29 22 36 15 64 8	pass
No element collisions	5	5 16 47 38 9	5 9 16 38 47	pass
Complex case	31	806 742 370 868 310 93 772 496 90 744 620 434 217 526 403 309 617 682 401 589 -1 734 -1 333 458 427 -1 678 834 308 712	308 333 458 427 678 712 734 806 742 370 834 868 310 93 772 496 90 744 620 434 217 526 403 309 617 682 401 589	pass
Max case	997	Because the number is too large, please refer to the document “max-size test Input”	Because the number is too large, please refer to the document “max-size test Output”	pass

3.3. Test purpose

Test case	Purpose
PTA case	It is the PTA case, a comprehensive case which is to test whether our algorithm is right.
Empty case	The empty case is tested to check whether our algorithm is right. This is the test of how well the program can resolve empty cell.
Smallest case	We want to test whether a hash table with the smallest case can output as expected.
Elements conflict in one cell	We want to test whether a hash table where all elements conflict in one cell can output as expected. This is a test of how well the program can resolve completely conflicting situations.
No element collisions	We want to test whether a hash table with no element collisions can output as expected. This is a test how well the program can handle situations where there is no conflict at all. (Since the smallest will be taken when there are multiple choices, the output is arranged from the smallest to the largest.)
Complex case	We want to test whether a hash table with a complex case can output as expected. This is a test how well the program can handle situations where there are some collisions and some empty cells. (a comprehensive case)
Max case	We want to test whether a hash table with a complex case with max-size(N=997) can output as expected. This is a test how well the program can handle situations where there are lots of data. (The second comprehensive case. We use a program to generate some rand-input, and save the result in “max-size test Output”.)

Chapter 4: Analysis and Comments

4.1. Analysis

4.1.1. Analysis for Time Complexity

Firstly, we use a loop to read the elements in each slot of hash table and the time complexity is O(N).

Secondly, we judge whether the slot is empty, whose time complexity is O(N). Then we calculate the initial insert position, assign the final insert position and judge whether it has the collision. The time complexity is O(N).

Thirdly, we build a heap whose time complexity is O(N). Then we use a loop to insert elements for different situation. The time complexity is O(NlogN).

Finally, we use a loop to print our result. The time complexity is O(N).

In conclusion, the total complexity is O (N + N + N + N + NlogN+N), so the time complexity of the program is O(NlogN).

4.1.2. Analysis for Space Complexity

In the algorithm, hash table, heap is used. The space complexity of hash table is O(N);The space complexity of building a heap is O(1);And the space complexity of heap-sort is O(N).

In conclusion, the space complexity of the algorithm is O(N).

4.2. Comment

In this project, we use heap and hash table to solve this problem. We also can solve this problem by graph. Firstly, traverse every element. If the in-degree is zero, then output. Otherwise, start from the residue position, the position is reached by linear detection method. For all the non-empty element positions passed, an edge to the element position is generated, and the location is added to the degree of 1 in the topological sorting. With priority queues, priority is given to output elements with smaller values.

Chapter 5 Appendix

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1005

/* Element structure, record the information of every element in the
slot */
typedef struct {
    int start; /* initial insert position */
    int end;   /* finial  insert position  */
    int value; /* element value */
    int count; /* the number of other elements that is already
                 in the hashtable with position between start and end */
    int flag;  /* if it has inserted to heap or it is a empty element,
flag==1, otherwise flag==0 */
} Element;

/* min-heap structure */
typedef struct{
    Element array[MAX];  /* heap array */
    int size;           /* heap size */
} Heap;


/* name:     percDown
   input:    Heap structure pointer, a specified starting position
   output:   none
   function: adjust the subheap with specified position as top satisfy
the properties of min-heap
*/
void percDown(Heap *heap,int pos)
{
    int parent,child;
    Element temp=heap->array[pos];

    for(parent=pos;parent*2+1<heap->size;parent=child)
    {
        child=parent*2+1; /* assume the minimum child is left child */
        if(child+1<heap->size&&heap->array[child].value>heap->array[chi
ld+1].value)
```

```
            child++; /* if right child exists and its value less than
left child's, adjust minimum child */
        if(heap->array[child].value>temp.value)
            break;  /* if the child with minimum value is larger than
parent's value */
        else
            heap->array[parent]=heap->array[child]; /* move downward
and continue finding position */
    }

    heap->array[parent]=temp; /* if the position is found, insert the
initial top element */
}


/* name:     initial
   input:    Heap structure pointer
   output:   none
   function: initial the heap
*/
void initial(Heap *heap)
{
    heap->size=0;
}


/* name:     build
   input:    Heap structure pointer
   output:   none
   function: adjust a mixed array, rebuild it as a min-heap
*/
void build(Heap *heap)
{
    int pos;
    for(pos=heap->size/2-1;pos>=0;pos--)
        percDown(heap,pos); /* adjust every subheap form first interior
node to top*/
}


/* name:     insert
   input:    Heap structure pointer, an elements to be inserted
   output:   none
   function: insert an element to the min-heap, adjust itself to keep
the properties
```

```
*/
void insert(Heap *heap,Element element)
{
    int child;
    Element temp=element;

    for(child=heap->size;child>0;child=(child-1)>>1)  /* find position
start from buttom */
    {
        if(temp.value>heap->array[(child-1)>>1].value) /* if the
element is less than its parents, then position is found */
            break;
        else
            heap->array[child]=heap->array[(child-1)>>1]; /* if not,
move upward and continue finding position */
    }

    heap->array[child]=temp; /* insert the elemens to found position */
    heap->size++;
}


/* name:     remove
   input:    Heap structure pointer
   output:   the least element of the initial min-heap
   function: remove the least element of initial min-heap and adjust it
to a min-heap
*/
Element removal(Heap *heap)
{
    Element temp=heap->array[0]; /* get the least element of min-heap*/

    heap->array[0]=heap->array[heap->size-1]; /* move the last elements
to top*/
    heap->size--;

    percDown(heap,0);  /* adjust the heap */
    return temp;
}


int main()
{
    Heap heap;
    Element slot[MAX],temp;
```

```
    int seq[MAX],num,index;
    int scount=0,dcount=0;  /* dcount: number of currently processed slots (empty slot or slot that has inserted to heap) */
                            /* scount: number of current sequence elements */

    initial(&heap); /* initial the heap */

    scanf("%d",&num);
    if(num<=0||num>1000) /* illegal size number */
    {
        fprintf(stderr,"Illegal size number\n");
        system("pause");
        exit(1);
    }

    for(index=0;index<num;index++) /* read the elements in each slot of hashtable */
    {
        scanf("%d",&(slot[index].value));
        if(slot[index].value<0)  /* if it's a empty slot */
        {
            slot[index].flag=1;
            dcount++;
        }
        else
        {
            slot[index].start=slot[index].value%num; /* calculate the initial insert position */
            slot[index].end=index;  /* assign the finial insert position */

            slot[index].count=0;
            if(slot[index].start==slot[index].end) /* if it has no collision when it first insert to slot */
            {
                slot[index].flag=1;
                heap.array[heap.size]=slot[index]; /* insert it to heap, which means that it can insert to slot */
                heap.size++;
                dcount++;
            }
            else
                slot[index].flag=0;
        }
```

```
    }

    if(heap.size==0&&dcount!=num) /* no element can be firstly inserted
to slot and the sequence is not empty, which represents an illgel
input. */
    {
        fprintf(stderr,"No Solution\n");
        system("pause");
        exit(1);
    }
    else
        build(&heap); /* build the min-heap */

    while(dcount!=num) /* simulate the insertion process until all
elements are inserted to heap */
    {
        if(heap.size!=0)
            temp=removal(&heap);  /* get the least elements in heap,
insert it to slot */
        else /* if there is no element in the heap but remains
untreated elements, it means none of rest elements can insert to slot.
No solution */
        {
            fprintf(stderr,"No Solution\n");
            system("pause");
            exit(1);
        }
        for(index=0;index<num;index++) /* adjust other elements */
        {
            if(slot[index].flag==0) /* if the slot is not empty and the
element hasn't inserted to heap */
            {
                if(slot[index].start<slot[index].end)
                {
                    if(temp.end>=slot[index].start&&temp.end<slot[index
].end) /* if the insert position is in the collision sequence of
current element */
                        slot[index].count++;
                }
                else /* the collision sequence go through the tail */
                {
                    if(temp.end>=slot[index].start||temp.end<slot[index
].end) /* if the insert position is in the collision sequence of
current element */
```

```
                        slot[index].count++;
                }
                if(slot[index].count==(slot[index].end-
slot[index].start+num)%num) /* if the slot form initial position to end
position is all full */
                {
                    slot[index].flag=1; /* insert current element to
heap, which means that it can insert to slot */
                    insert(&heap,slot[index]);
                    dcount++;
                }
            }
        }
        seq[scount++]=temp.value; /* add the current element into
insertion sequence */
    }

    while(heap.size!=0) /* if all elements is in the heap, then only
insert elements to slot in order from small to large */
    {
        temp=removal(&heap); /* get the least element and simulated
insert it to slot */
        seq[scount++]=temp.value; /* add the current element into
insertion sequence */
    }

    for(index=0;index<scount;index++) /* print the element in insertion
sequence */
    {
        if(index==0)
            printf("%d",seq[index]);
        else
            printf(" %d",seq[index]);
    }

    system("pause"); /* pause the program to see the result */
    return 0;
}
```

Chapter 6 Declaration

We hereby declare that all the work done in this project titled "Hashing-hard version" is of our independent effort as a group.

Chapter 7 Duty Assignments

Programmer: 余瑞璟

Tester: 林家丰

Report Writer: 郭泽均