Hashing – Hard Version

2016-1-2

Chapter 1: Introduction

The background of the problem

Given a hash table of size *N* , we can define a hash function *H(x) x%N* .Suppose that the linear probing is used to solve collisions, we can easily obtain the status of the hash table with a given sequence of input numbers.

The description of the problem

Now we are asked to solve the reversed problem: reconstruct the input sequence from the given status of the hash table. Whenever there are multiple choices, the smallest number is always taken.

The input and the output

Input Specification:

Each input file contains one test case. For each test case, the first line contains a positive integer N(<=1000), which is the size of the hash table. The next line contains N integers, separated by a space. A negative integer represents an empty cell in the hash table. It is guaranteed that all the non-negative integers are distinct in the table.

Output Specification:

For each test case, print a line that contains the input sequence, with the numbers separated by a space. Notice that there must be no extra space at the end of each line.

Chapter 2: Algorithm Specification

Description of the algorithm

As we can see, the number of collision times is equal to (index – key % N),   but consider the case that the index is smaller than key % N. We change the formula to

Number of collision times = (index – key % N + N) % N

We consider that collision is because the position we get from hash function is used up by other number. If we moved by linear probing and find the next position is used up too, then we meet another collision. So we can see that the collision is created because of influence of other numbers.

We then use a digraph, and just regard the collisions as edges. And reconstructing the input sequence is just topological sort.

The process of the algorithm is:

First, Input the hash table.

Second, Compute the in-degree(the number of collision times), and form the graph. We just regard the index as vertices. The vertices from key % N to index-1 all points to the current index just because these vertices influence current vertices.

Third, topological sort , and put the vertices whose in-degree equals to 0 into a min-heap.

Fourth, Find the vertices whose in-degree is 0 with minimal value(the root of the min-heap) and scan the vertices adjacent to it, decrease the in-degree with 1. If the in-degree is 0, put it into a min-heap.

Of course, when we find the root of the min-heap, we should print the key of the vertices.

Specifications of main data structures

1) The digraph

The first main data structure we used is the digraph. We use the adjacency list to represent the graph.

The way we definite the node is shown below:

```
struct node{
    int n_edges_in;
    int n_edges_out;
    int* nextnodes;
};
```

`n_edges_in` is in-degree, `n_edges_out` is out-degree, `*nextnodes` points to the next node.

We should the form the map first(how to form the map is shown in The process of the algorithm, third), Here is the code of linking two vertices:

```
void linknodeab_onlydealwitha(ptr* map,int a,int b){//link the node a&b which a precedes b
    int nedges=++map[a]->n_edges_out;
    map[a]->nextnodes=(int*)realloc(map[a]->nextnodes,sizeof(int)*(nedges+1));
    map[a]->nextnodes[nedges]=b;
}
```

We also need to delete the edges as the algorithm shows, the key code is shown below:

```
for(i=1;i<=map[deletefromheap]->n_edges_out;i++){
        next=map[deletefromheap]->nextnodes[i];
        map[next]->n_edges_in--;//don't worry, we don't need to delete the edge totally.
```

```
            if(map[next]->n_edges_in==0){//don't worry, if tab[x]<0,the in-degree of node x can't be 0
                heap_insert(myheap,next,tab);
            }
```

2) the min-heap

This is the definition of heap type:

```
struct heaptype{
   int n_elements;
   int* element;
};
```

We should do two things to the min-heap, one is to insert some elements to the min-heap(the code is shown below),

```
void heap_insert(struct heaptype* heap,int x,int* tab){//insert tab[x] into heap
    int now,upper;

    heap->n_elements++;

    heap->element=(int*)realloc(heap->element,sizeof(int)*(heap->n_elements+1));
    heap->element[heap->n_elements]=x;
    now=heap->n_elements;
    while(1){
        if(now==1) break;
        upper=now/2;
        if(tab[heap->element[now]]<tab[heap->element[upper]])
swap(&heap->element[now],&heap->element[upper]);
        now=upper;
    }
}
```

And another is to delete the root of the min-heap:

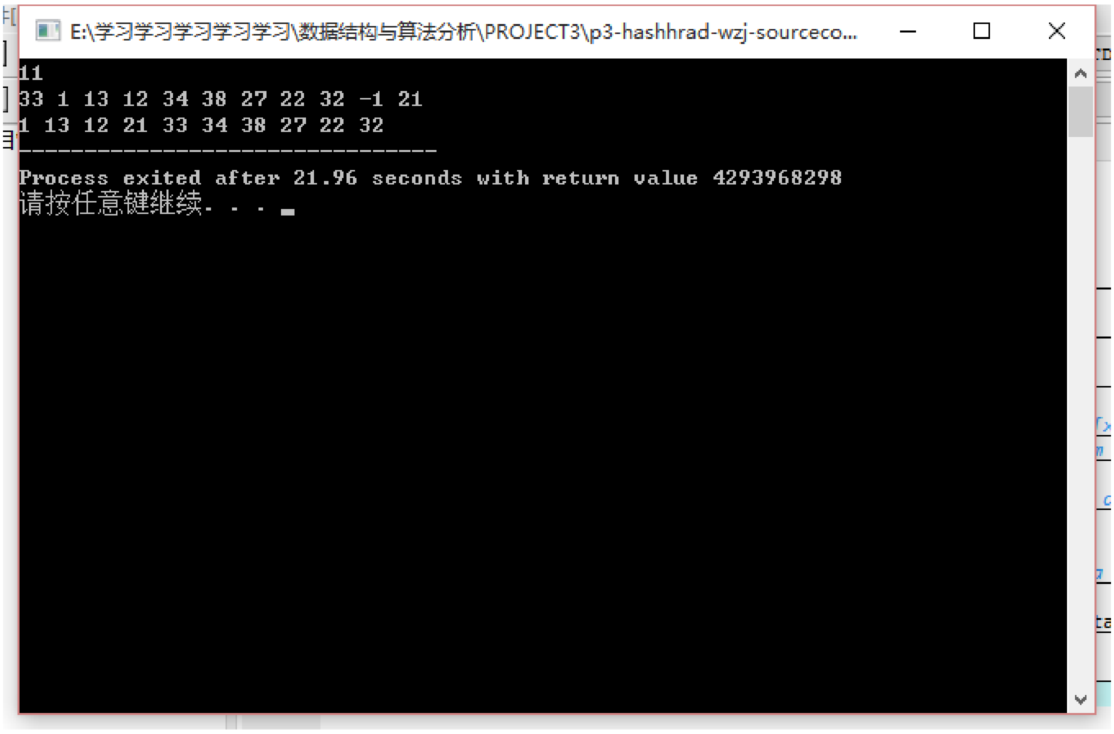
```
int heap_delete(struct heaptype* heap,int* tab){//delete the minimum element in the heap
    int ans;
    int now=1,l,r;

    if(heap->n_elements==0) return -998998;//means the heap is empty
    else ans=heap->element[1];

    swap(&heap->element[1],&heap->element[heap->n_elements]);
    heap->n_elements--;
```

```
    while(1){
        l=2*now;r=l+1;
        if(l<heap->n_elements){//has both left and right child
            if(tab[heap->element[l]]>tab[heap->element[r]]){
if(tab[heap->element[now]]>tab[heap->element[r]]){
                    swap(&heap->element[now],&heap->element[r]);
                }
                else break;
                now=r;
            }
            else{
                if(tab[heap->element[now]]>tab[heap->element[l]])
swap(&heap->element[now],&heap->element[l]);
                else break;
                now=l;
            }
        }
        else if(l==heap->n_elements){//only has left child
            if(tab[heap->element[now]]>tab[heap->element[l]])
swap(&heap->element[now],&heap->element[l]);
            else break;

        }
        else{//has no child
            break;
        }
    }

    return ans;
}
```

Chapter 3: Testing Result (Current Status: pass)

TEST CASES

a) INPUT
   11
   33 1 13 12 34 38 27 22 32 -1 21
   OUTPUT
   1 13 12 21 33 34 38 27 22 32
   RUNNING RESULT



b) INPUT

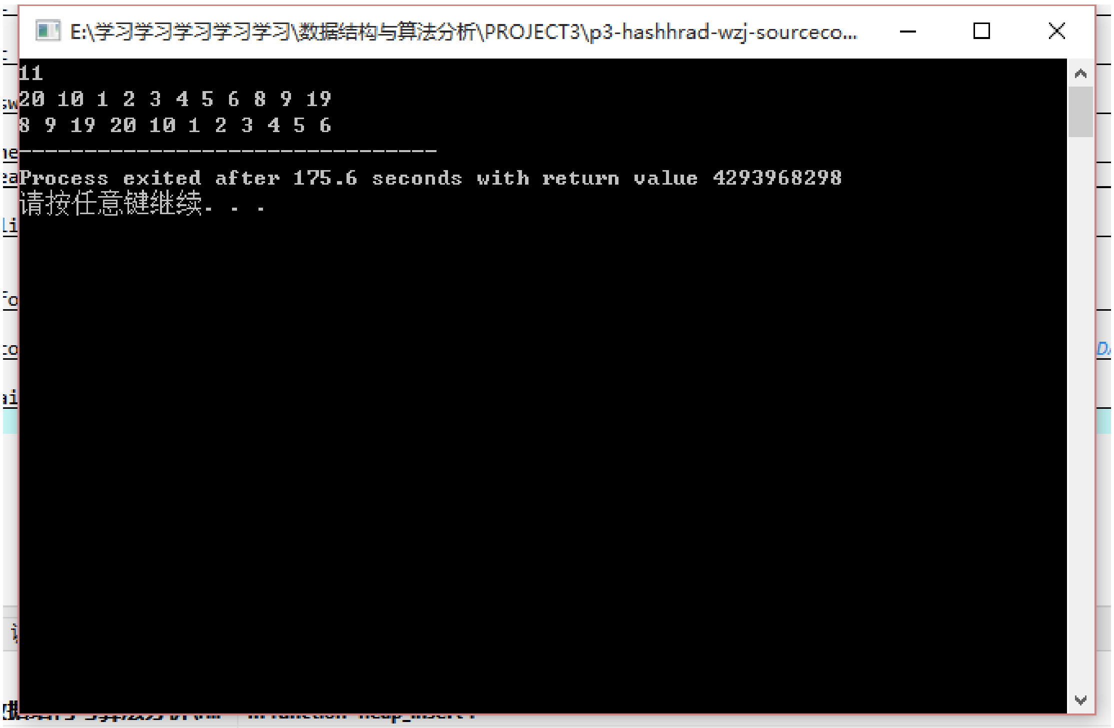
11

20 10 1 2 3 4 5 6 8 9 19

OUTPUT

8 9 19 20 10 1 2 3 4 5 6

RUNNING RESULT



c) INPUT

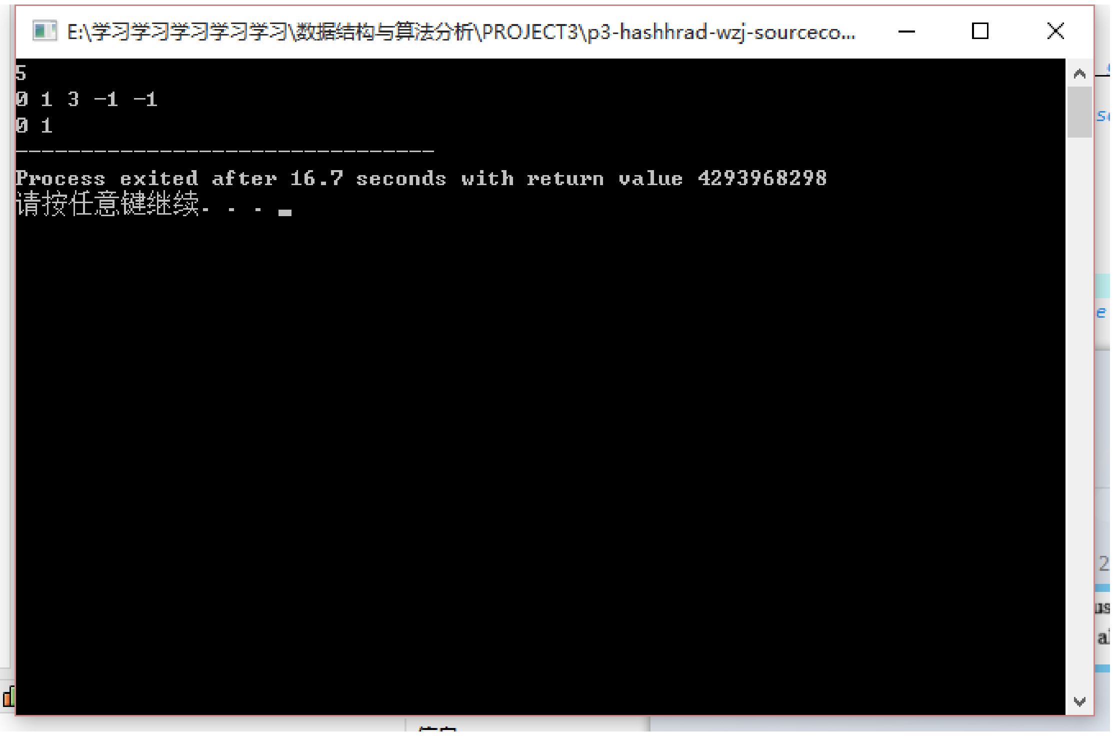
5

0 1 3 -1 -1

OUTPUT

0 1

RUNNING RESULT



Chapter 4: Analysis and Comments

TIME COMPLEXITY: The time complexity is mainly depend on the function *formmap* .

```
void formmap(ptr* map,int* tab,int n,struct heaptype* heap){//draw a DAG according the given
sequence
    int i,j;
    int hash;
    int nodetome;
    ptr newcell;

    for(i=0;i<n;i++) {//initiate all nodes of the map O(N)

        newcell=(ptr)malloc(1*sizeof(ptr));
        newcell->n_edges_in=0;newcell->n_edges_out=0;
        newcell->nextnodes=NULL;
        map[i]=newcell;
    }


    for(i=0;i<n;i++){ O(N)

        if(tab[i]<0){
```

```
			map[i]->n_edges_in=-998998;//don't worry, the in-degree -998998 won't change later
			continue;
		}

		hash=tab[i]%n;
		if(hash==i) {//if there is no collision
			heap_insert(heap,i,tab);
			continue;
		}
		//if there exist collision
		map[i]->n_edges_in=(i-hash+n)%n;//change the node i's in-degree in one step
		for(j=0;j<map[i]->n_edges_in;j++){ O(N)
			nodetome=(hash+j)%n;
			linknodeab_onlydealwitha(map,nodetome,i);
		}
	}
}
```

As we can see, the time complexity of this program is $O(N^2)$.

COMMENT:

The wonder of this problem is the topological sorting. The solution of this problem can inspire us that Graph Theory is always important in computer science.

Appendix: Source Code (in C)

```c
#include<stdio.h>
//#include<malloc.h>

typedef struct node* ptr;
struct node{
    int n_edges_in;
    int n_edges_out;
    int* nextnodes;
};

struct heaptype{
    int n_elements;
    int* element;
};

void swap(int* a,int* b){
    int k=*a;
```

```
    *a=*b;
    *b=k;
}

void heap_insert(struct heaptype* heap,int x,int* tab){//insert tab[x]
into heap
    int now,upper;

    heap->n_elements++;

heap->element=(int*)realloc(heap->element,sizeof(int)*(heap->n_elemen
ts+1));
    heap->element[heap->n_elements]=x;
    now=heap->n_elements;
    while(1){
        if(now==1) break;
        upper=now/2;
        if(tab[heap->element[now]]<tab[heap->element[upper]])
swap(&heap->element[now],&heap->element[upper]);
        now=upper;
    }
}
int heap_delete(struct heaptype* heap,int* tab){//delete the minimum
element in the heap
    int ans;
    int now=1,l,r;

    if(heap->n_elements==0) return -998998;//means the heap is empty
    else ans=heap->element[1];

    swap(&heap->element[1],&heap->element[heap->n_elements]);
    heap->n_elements--;
    while(1){
        l=2*now;r=l+1;
        if(l<heap->n_elements){//has both left and right child
            if(tab[heap->element[l]]>tab[heap->element[r]]){
                if(tab[heap->element[now]]>tab[heap->element[r]]){
                    swap(&heap->element[now],&heap->element[r]);
                }
                else break;
                now=r;
            }
            else{
```

```
                if(tab[heap->element[now]]>tab[heap->element[l]])
swap(&heap->element[now],&heap->element[l]);
                else break;
                now=l;
            }
        }
        else if(l==heap->n_elements){//only has left child
            if(tab[heap->element[now]]>tab[heap->element[l]])
swap(&heap->element[now],&heap->element[l]);
            else break;

        }
        else{//has no child
            break;
        }
    }

    return ans;
}

void linknodeab_onlydealwitha(ptr* map,int a,int b){//link the node a&b
which a precedes b
    int nedges=++map[a]->n_edges_out;

map[a]->nextnodes=(int*)realloc(map[a]->nextnodes,sizeof(int)*(nedges
+1));
    map[a]->nextnodes[nedges]=b;
}


void formmap(ptr* map,int* tab,int n,struct heaptype* heap){//draw a DAG
according the given sequence
    int i,j;
    int hash;
    int nodetome;
    ptr newcell;
    for(i=0;i<n;i++) {//initiate all nodes of the map
        newcell=(ptr)malloc(1*sizeof(ptr));
        newcell->n_edges_in=0;newcell->n_edges_out=0;
        newcell->nextnodes=NULL;
        map[i]=newcell;
    }

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

```
        if(tab[i]<0){
            map[i]->n_edges_in=-998998;//don't worry, the in-degree -998998 won't change later
            continue;
        }

        hash=tab[i]%n;
        if(hash==i) {//if there is no collision
            heap_insert(heap,i,tab);
            continue;
        }
        //if there exist collision
        map[i]->n_edges_in=(i-hash+n)%n;//change the node i's in-degree in one step
        for(j=0;j<map[i]->n_edges_in;j++){
            nodetome=(hash+j)%n;
            linknodeab_onlydealwitha(map,nodetome,i);
        }
    }
}

void topsort_onlyprint(ptr* map,int n,struct heaptype* myheap,int* tab){//topological-sort the above DAG
    int deletefromheap;
    int i;
    int next;
    int isthe1stoutput=1;

    while(1){
        deletefromheap=heap_delete(myheap,tab);
        if(deletefromheap==-998998) break;//-998998 means the heap is empty and we have finished
        if(isthe1stoutput){//to confirm that no extra space at the end of each line
            printf("%d",tab[deletefromheap]);
            isthe1stoutput=0;
        }
        else{
            printf(" %d",tab[deletefromheap]);
        }
        for(i=1;i<=map[deletefromheap]->n_edges_out;i++){
            next=map[deletefromheap]->nextnodes[i];
            map[next]->n_edges_in--;//don't worry, we don't need to delete the edge totally.
```

```
            if(map[next]->n_edges_in==0){//don't worry, if tab[x]<0,the in-degree of node x can't be 0
                heap_insert(myheap,next,tab);
            }
        }
    }
}

int main(){
    int a=1,b=2;
    ptr* map;

    int n,i,j;
    int tab[1005];

    struct heaptype* myheap=(struct heaptype*)malloc(1*sizeof(struct heaptype));
    myheap->n_elements=0;myheap->element=NULL;//create a heap

    scanf("%d",&n);
    for(i=0;i<n;i++){//input the hash table
        scanf("%d",&tab[i]);
    }

    map=(ptr*)malloc(n*sizeof(ptr));//give the map proper memory spaces
    formmap(map,tab,n,myheap);//form the map from the hash tab and insert all nodes whose in-degrees==0

    topsort_onlyprint(map,n,myheap,tab);//topsort while print the answer
}
```

Declaration:

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