Data Structure and Algorithm, Spring 2019

Homework 3

Release: Saturday, May 4, 2019

Due: 23:59:59, Sunday, May 19, 2019

TA E-mail: `dsa1@csie.ntu.edu.tw`

Rules and Instructions

- In homework 3, the problem set contains 5 problems and is divided into two parts, non-programming part (problem 1, 2, 3) and programming part (problem 4, 5).
- Please go to *DSA Judge* (`https://dsa2019.csie.org`) and complete the first problem to familiarize yourself with the usage of the judge system.
- For problems in the non-programming part, you should combine your solutions in ONE PDF file, and then submit it via the judge system. Your file should be as clean as possible, and the use white text on black background is prohibited. Your solution must be as simple as possible. At the TAs' discretion, solutions which are too complicated will be regarded as incorrect. Moreover, if you would like to use any theorem which is not mentioned in the classes, please include its proof in your solution.
- For problems in the programming part, you should write your code in C programming language, and then submit it via the judge system. You can submit up to 5 times per day for each problem. The compilation command of the judge system is `gcc main.c -static -std=c11 -O2 -lm`.
- Discussions with others are encouraged. However, you should write down your solutions in your own words. In addition, for `each problem`, you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution of that problem in your solution.
- The score of the part that is submitted after the deadline will get some penalties according to the following rule:

$$LateScore = (\frac{86400 - DelayTime(sec.)}{86400})OriginalScore$$

Problem 1. Heaps! More Heaps! (18 *points*)

You have learned the data structure *heap* in the lecture, and should know that it supports the functions `Insert(x)`, `Extract-Max()`, and `Max()` (assuming that it is a max heap). Heaps in practice can be used to dynamically maintain a set of data, and help us to find some statistics about the set of data efficiently. However, the operations we have introduced now are not enough, so we will now explore a few new operations with heaps.

In the following questions, please answer with C code or pseudo code, and provide a brief explanation. You can assume that all the values in the heap are unique, and you can make use of the functions `Insert(x)`, `Extract-Max/Min()`, and `Max/Min()` on a max/min heap freely in your functions. Also, please use the array representation of heap, as introduced in the lecture, in your code.

1. (4 *points*) First, we want to support the operation `Find-Greater(v)` on a max heap. The function should return the number of elements in the heap that is greater than v. The function should run in $O(k)$-time, where k is the number of elements in the heap that is greater than v

2. (6 *points*) Second, we want to support the operation `delete(id)` on a max heap. Heaps are usually stored in an array, where each element in the array represents a node in the heap. The function should delete the node in the heap corresponding to the id^{th} element in the array, and it should run in $O(\log n)$ time, where n is the number of elements in the heap.

3. (8 *points*) At last, if today we are not interested in the maximum nor minimum value of a set of data, but the median of it, then we will need a slightly different data structure. Please design a data structure that supports the operations `Insert(x)`, `Extract-Median()`, and `Median()`. Here we define the median as the $(\frac{n}{2})^{th}$ smallest element if n is an even number. Like a normal heap, `Insert(x)` and `Extract-Median()` should run in $O(\log n)$-time, and `Median()` should run in $O(1)$-time. (Hint: One ordinary heap may not be able to solve this problem, but a few may.)

Problem 2. Algorithmic Complexity Attack (2) - Hash Table and Function (21 *points*)

As mentioned in Homework 2, some algorithms, including Quick Sort, are vulnerable to Algorithmic Complexity Attack. Here we would like to discuss another algorithm which might also be threatened by such attack.

We know that with hash table, the expected time complexity for insert and search are both $O(1)$. However, things are not always that ideal.

1. (6 *points*) Python has a built-in data structure called `dictionary`. `Dictionary` stores key-value pairs (k, v), and support expected $O(1)$-time complexity to find v given the key k. Sounds like magic, but it is nothing but a hash table. To simplify this problem, we assume Python dictionary uses `hash(k)`, a python built-in function, as the bucket index of key k, and the hash table size is initially set as the same size as the output range of `hash()`. We also assume Python dictionary resolves hash collision with `Open Addressing`. The program runs with `Python 3.7.3`, which is available on CSIE workstations.

   (a) (3 *points*) It is known that when x is an integer, $hash(x) = x \, mod \, k$, where k is a large integer. Please find out the value of k of the built-in function `hash` of `Python 3.7.3`. Briefly explain your procedure and approaches to get the clues. Note that directly looking for the answer on the Internet is not allowed in this sub-problem. If you are not familiar with Python or workstation, you can access the web service at `http://linux10.csie.org:17707`. With this service, you can enter an arbitrary integer x, and the service will return the value of $hash(x)$ to you.

   (b) (3 *points*) As an attacker, please provide 10^6 valid key-value pairs to be inserted into the `dictionary` to implement Algorithmic Complexity Attack to the Python dictionary. Briefly explain why your strategy works, and derive the time complexity of the search operation based on your input. Note that the keys should be integers, and they should also be unique.

2. (6 *points*) Such attack might not be feasible if the hash function is unknown to the adversaries. In addition, if the inputs are benign, which implies the inputs are uniform-randomly generated, the time complexity for the search operation might still be within the expected complexity. In fact, the time complexity has much to do with load factor α. Consider an open-addressing hash table with a uniformly-distributed hash function. Based on the additional information in the sub-problems, please calculate the expected number of probes for searching.

   (a) (3 *points*) $\alpha = 0.8$, and collisions are resolved by chaining.

   (b) (3 *points*) $\alpha = 0.6$, and collisions are resolved by linear probing.

3. (9 *points*) Hash functions can be used in lots of ways, in addition to hash tables. For example, `MD5` is a hash function commonly used to check whether two files are identical or not. If the two files are not identical, then their `MD5` hash values should be different. Therefore, it is used as a checksum mechanism to verify the integrity of a downloaded file. For example, when you download a file from NTUCC Download Center, MD5 checksum of the original file is also provided. Once finished downloading, the user can easily verify whether the downloaded file is authentic and not corrupted by transmission error. If the received checksum and calculated checksum do not match, then the downloaded file is either corrupted or tampered.

   Another example is the judge system you use to upload your hand-written homework solutions. After uploading, you will get a `SHA1` (another hash function.) checksum. You can use the checksum to verify if the file on the server side is same as the file on your side. A checksum hash function should satisfy the following two properties: 1) Collision Resistance, 2) Fast Computation. If it is easy to find two different files with the same hash value, the integrity of the files can no longer be verified because even when two files have the same checksum, they can still be different. On the other hand, if computation takes a long time (e.g., linear to the file size), directly performing byte-wise comparison of the two files might be more efficient.

   Yanger has developed his own checksum hash function YAG. YAG takes a file a as input, and generates a 32-bit unsigned integer as the output, which serves as the checksum of file a. YAG will first separate the file into 4 even parts, and then for each part, YAG will bitwise-xor all the bytes in the parts. Finally, YAG concatenates the bitwise-xor results of the 4 parts to form a 32-bit integer, which is the output. For example, let x be an 8-byte-long file with byte-wise content `3B4B5A621000101E` in hex format. $YAG(x) = (7038100e)_{16}$. For simplicity, assume for all files, the number of bytes in a single file is always a multiple of 4.

   (a) (2 *points*) Let y be a 16-byte-long file, please calculate $YAG(y)$, the checksum of file y. The file content of y is `11BE3AEF951CA6B444006FC0478D9205`. Please answer in hex format.

   (b) (2 *points*) Let z be a randomly generated 16-byte-long file. What is the probability that $YAG(z) = YAG(y)$?

   (c) (3 *points*) Let a and b be two randomly generated 16-byte-long files. What is the probability that $YAG(a) = YAG(b)$?

   (d) (2 *points*) Based on the results in previous sub-problems, is YAG a good checksum hash function? Please briefly explain your answer.

Problem 3. Disjoint Set (21 *points*)

As shown in the lecture, a disjoint set data structure supports the operations of `MAKE_SET`, `FIND_SET` and `UNION`.

For any sequence of m operations, the overall time complexity is given by $O(m\alpha(m))$, where α is a function that grows very slowly. It can be shown that α(The number of atoms in this universe) < 4, and $\alpha(km) = O(\alpha(m))$ for any positive integer k.

The implementation of functions are shown as follows.

```
MAKE_SET(x)
    x.parent = x
    x.rank = 0

UNION(x,y)
    LINK(FIND_SET(x), FIND_SET(y))

LINK(x,y)
    if(x.rank>y.rank)
        y.parent = x
    else
        x.parent = y
        if(x.rank == y.rank)
            y.rank = y.rank + 1

FIND_SET(x)
    if(x.parent != x)
        x.parent = FIND_SET(x.parent)
    return x.parent
```

1. **Minimum** (4 *points*)

   In this subproblem, you are asked to implement an additional operation, `MIN_ELEMENT`. The return value of `MIN_ELEMENT(k)` should be the minimal element in the set containing `k`.

   The time complexity should remain the same after adding this new operation. That is, for any sequence of m operations consisting of `MAKE_SET`, `FIND_SET`, `UNION`, and `MIN_ELEMENT`, the overall time complexity should still be $O(m\alpha(m))$. You can modify the implementation of those operations if you want.

2. **Isolation** (6 *points*)

   In this subproblem, you are asked to implement an additional operation, `ISOLATE`. When `ISOLATE(k)` is called, `k` will be isolated from the set it belonged to, i.e, removed from that set and form a set by itself.

   Since we may isolate the representative of a set, you don't need to support `FIND_SET` in this scenario. Instead, you need to support `SAME_SET`. If `x` and `y` are in the same set, then `SAME_SET(x,y)` should return `True`, otherwise, it should return `False`.

   For example, after the following operations,

```
MAKE_SET(1)
MAKE_SET(2)
MAKE_SET(3)
UNION(1,2)
UNION(2,3)
ISOLATE(2)
```

   There will be two sets, `{1,3}` and `{2}`.

   The time complexity should remain the same. That is, for any sequence of m operations consisting of `MAKE_SET`, `SAME_SET`, `UNION`, and `ISOLATE`, the overall time complexity should still be $O(m\alpha(m))$. You can modify the implementation of those operations if you want.

3. **Decoration** (11 *points*)

   It is still seven months from Christmas, but you decide to start preparing the decoration for the holiday. You bought n colorful balls, indexed from `1` to `n`, and each ball is attached to a piece of wire 1 inch in length. You can fasten the other end of the wire to either the ceiling or another ball.

   At the beginning, all balls are attached to the ceiling directly by one piece of wire. However, sometimes you may want to attach a ball to another ball via the piece of wire, and sometimes you may want to find out the distance from a ball to the ceiling. In this part, we are going to solve this problem efficiently using *disjoint set*.

   To be more precise, we are going to complete two functions, `FIND_DISTANCE` and `ATTACH`.

- `FIND_DISTANCE(x)`: Return the distance from ball `x` to the ceiling, in inches.
- `ATTACH(x,y)`: Attach ball `x` to ball `y` using the piece of wire on ball `x`. It is guaranteed that before this operation, ball `x` and ball `y` were not connected via wires either directly or indirectly, and ball `x` was attached to the ceiling directly. However, there's no such guarantee for ball `y`.

You can assume that the balls are very small, which means that their radius is negligible.

For example, consider the following sequence of operations with `n=4`.

```
ATTACH(2,4)
ATTACH(1,4)
print FIND_DISTANCE(4)
ATTACH(4,3)
print FIND_DISTANCE(2)
```

The following graphs show how your decoration looks like after the three `ATTACH` operations are executed, respectively.

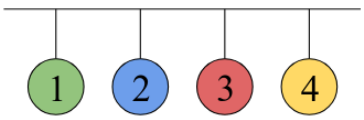

Figure 1

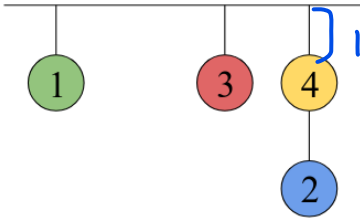

Figure 2

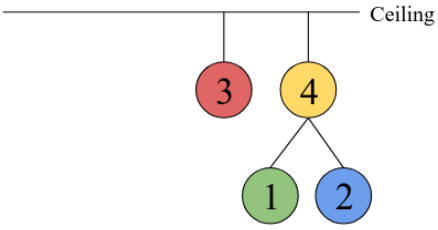

Figure 3

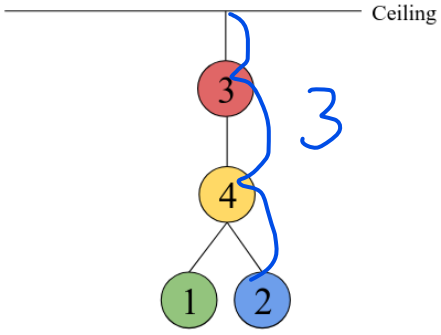

Figure 4

Since the balls are very small, the distance from a ball to the ceiling is actually the number of wires from the ball to the ceiling.

Thus, the first output should be `1` because the distance from ball `4` to the ceiling is `1` inch in figure 3. Similarly, the second output should be `3`, as illustrated in figure 4.

(a) (4 *points*) Assume that we don't care about the time complexity. The most intuitive way to implement `ATTACH(x,y)` is setting `x.parent` to be `y`. When `FIND_DISTANCE(x)` is called, we need to iterate through the path from `x` to the root of `x` and count the number of edges. Note that we can't use either *path compression* or *union by rank* in such case.

```
for i from 1 to n
    i.parent = i
ATTACH(x,y)
    x.parent = y
FIND_DISTANCE(x)
    current_distance = 1
    while(x.parent != x)
        x = x.parent
        current_distance = current_distance + 1
    return current_distance
```

Analyze the time complexity for executing m operation with this naive implementation in Θ notation. You need to show prove upper bound and lower bound.

(b) (7 *points*) The implementation in (a) is apparently too slow. Now, we want to adapt *path compression* when calling `FIND_DISTANCE`. Since *path compression* would destroy the original structure of the tree, we need to keep track of more information.

For each node `x`, we store another property `x.dist`, which is the real distance from `x` to `x.parent` on the decoration.

Now, the code of `ATTACH` may be like this.

```
ATTACH(x,y)
    x.parent = y
    x.dist = 1
```

We set `x.dist` to be `1` because `x.parent` is `y` now, and since ball `x` is attached to ball `y` directly, their distance on the decoration is `1`.

Let's look at the example at the beginning again. After we execute `FIND_DISTANCE(2)`, `2.parent` would become `3` because of path compression. Thus, `2.dist` should be updated to `2` because the distance from ball `2` to ball `3` is `2` on the decoration.

Please implement a new version of `FIND_DISTANCE(x)` to perform *path compression*, and update `x.dist` correctly.

Programming Part

Problem 4. Arvin and the Pancake House (20 *points*)

Marvelous Arvin is up for new ideas again. He has decided to open a pancake house to share his love of pancakes! Everyday, Arvin makes and sells pancakes. The pancakes which are not sold at the end of each day are shared with his friends on the pancake party.

Arvin would like to save the biggest pancakes for his friends, but has trouble finding an effective way to do so. His friend, piepie, suggested that maybe using a queue and a stack would be a good way to manage his pancakes. The rules go as follows:

- Every time a new pancake is made, Arvin can choose to place(push or enqueue) it into the stack or the queue.
- Every time a customer orders a pancake, it should be removed(pop or dequeue) from the stack or queue immediately (a good vendor should not keep his customer waiting).

Now, given the sequence of pancakes made and sold, can you help figuring out which pancakes Arvin can save for his friends?

Input Format

The first line contains an integer N, indicating the total number of pancakes made and sold.

In each of the following N lines, there is an instruction (0 or 1). Instruction 0 means that a pancake is made. Instruction 1 means that a pancake is sold. If the instruction is 0, another integer follows, representing the pancake size.

Output Format

A single number representing the sum of the sizes of the pancakes saved

Input Constraint

20% of the test cases: $0 < N \leq 25$

80% of the test cases: $0 < N \leq 500000$

100% of the test cases: $0 < pancake\,size \leq 2 \times 10^6$, $|pancakes\,made| >= |pancakes\,ordered|$ at any time.

Sample Input

```
6
0 2
0 1
1
0 4
0 3
1
```

Sample Output

7

Sample Testdata Explanation

push 2 into queue

push 1 into queue

pop 2 from the queue

push 4 into queue

push 3 into queue

pop 1 from queue

–> sizes of pancakes saved = $[4, 3]$

–> sum = $4 + 3 = 7$

Problem 5. Dark Hot Pot Party (20 *points*)

> "Welcome to the hot pot party! Do you want to take part in the hot pot party? As long as you bring your own ingredients, the organizer will provide you with an empty hot pot for you to enjoy this party by yourself!"

You, the host of the party, will record any events immediately. Events that may happen at this party are as follows:

- "New": A new participant has joined this hot pot party. You know the name of this participant and what ingredient(s) are brought to the party.
- "Merge": Because you want to make the party come alive and avoid "hot-pot-themselves", you may command two specific participant to mix their hot pots into one during the party. Once the two hot pots are mixed, the owners of these two hot pots are automatically merged into a new group and share the mixed hot pot together. For example, if the current party status is as follows:

Team Member	Hot pot Ingredients
Bucciarati, Abbacchio	Zip, Abbaccha
Giovanna	Ladybug
Mista	Bullets, Bullets

  And the party status after the event 'Merge Abbacchio Giovanna':

Team Member	Hot pot Ingredients
Bucciarati, Abbacchio,Giovanna	Zip, Abbaccha,Ladybug
Mista	Bullets, Bullets

- "Hot Pot Resonance": If the ingredients of the two hot pots are exactly the same, then the two hot pots attract each other and are merged automatically because of the magic power of hot pot resonance. Note that if the respective numbers of each ingredients of the two hot pots are the same, then these two hot pots are regarded as the same.

Because "Hot Pot Resonance" is an extremely interesting phenomenon, we would like you to write a program to track if any "Hot Pot Resonance" happens, given the "New" and "Merge" events in chronological order.

Input Format

The first line contains an integer N. The following N lines are the "New" and "Merge" events in chronological order. Next, we describe the format of these events in detail.

- `New` S_{name} M $ingredients_1$ $ingredients_2$ ... $ingredients_M$
  There's a newcomer S_{name}, who brings M ingredient(s), and the number of these ingredients are given in the following $ingredients_1$ $ingredients_2$ ... $ingredients_M$.

- `Merge` S_{name1} S_{name2}
  You ask S_{name1} and S_{name2} to merge their hot pots. All the ingredients in these two hot pots are merged into the new mixed one.

Output Format

Any two hot pots possessing exactly the same ingredients merge automatically by the magic power of "Hot Pot Resonance". If such phenomenon happens, please output a single line of "S_{name1} S_{name2}", where S_{name1}, S_{name2} identifies the person who came to the party first in each hot pot group, respectively . It is also required that the name of the person who came earlier should be placed in front.

Note that the event record we give is in chronological order, so the name appears in the record earlier came to the party earlier.

For example, if Qmao came earlier than Arvin, Arvin came earlier than Mama, Mama came earlier than Piepie, the party status now can be illustrated as follows.

Team Member	Hot pot Ingredients
Qmao	A
Arvin	B
Mama	A,B
Piepie	A,A,B,B

The party status after the new event 'Merge Qmao Arvin':

Team Member	Hot pot Ingredients
Qmao,Arvin	A,B
Mama	A,B
Piepie	A,A,B,B

{Qmao, Arvin} and {Mama} automatically merge because the ingredients of these two hot pots are the identical. At that time, you should output "Qmao Mama". Note that you can't output "Mama Qmao" as your answer because Qmao came earlier.

Team Member	Team Ingredients
Qmao,Arvin,Mama	A,A,B,B
Piepie	A,A,B,B

Then, {Qmao, Arvin, Mama} and {Piepie} automatically merge because the ingredients of these two hot pots are the identical. At that time, you should output "Qmao Piepie". Again, you can't output "Piepie Qmao" as your answer because Qmao came earlier.

Input Constraint

All the name of participants and ingredients consist of upper and lower case English alphabets with length $\leq 10^2$.

The participants' names are unique.

In a merge event, S_{name1}'s hot pot $\neq S_{name2}$'s hot pot.

There's no invalid event or operation.

20% of test cases: $0 <$ Number of total ingredients $\leq 10^2$, $0 < N \leq 10^2$

50% of test cases: $0 <$ Number of total ingredients $\leq 3 \times 10^3$, $0 < N \leq 3 \times 10^3$

100% of test cases: $0 <$ Number of total ingredients $\leq 2 \times 10^6$, $0 < N \leq 2 \times 10^6$

Sample Input

```
9
New Qmao 1 A
New Howard 1 B
New Mama 2 A B
New Arvin 4 A A B B
New Piepie 2 C C
New Yanger 1 D
Merge Qmao Howard
Merge Piepie Yanger
New Handsome 3 C C D
```

Sample Output

```
Qmao Mama
Qmao Arvin
Piepie Handsome
```