Data Structure and Algorithm

Homework #3 Solution

TA email: dsa1@csie.ntu.edu.tw

Problem 1. Hash (25%)

1. (8%)

   Open addressing with linear probing

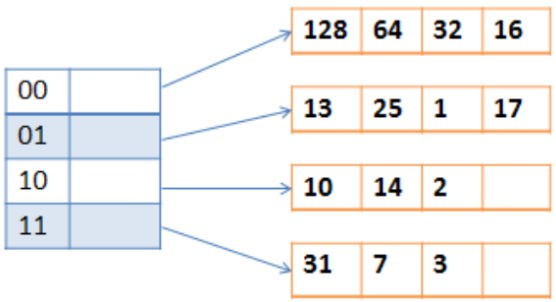
initial	0	1	2	3	4	5	6	7	8	9	10
18								18			
34		34						18			
9		34						18		9	
37		34			37			18		9	
40		34			37			18	40	9	
32		34			37			18	40	9	32
89		34	89		37			18	40	9	32

   Open addressing with double hashing

initial	0	1	2	3	4	5	6	7	8	9	10
18								18			
34		34						18			
9		34						18		9	
37		34			37			18		9	
40		34			37			18	40	9	
32		34			37			18	40	9	32
89	89	34			37			18	40	9	32

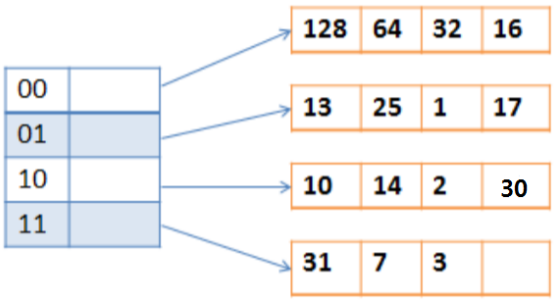
2. (6%)

   (a) (3%)

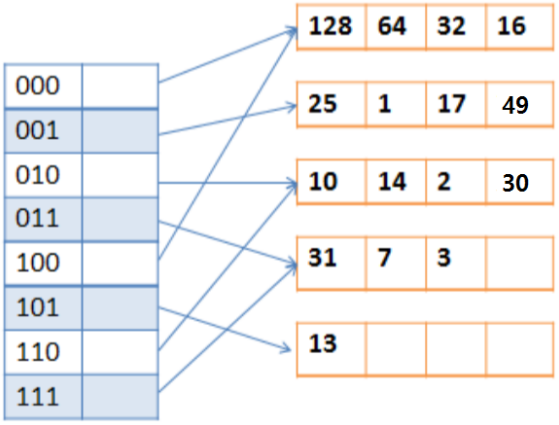


(b) (3%)

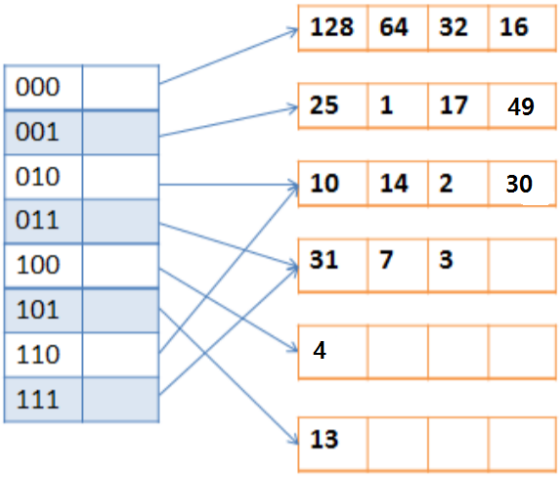
insert 30



insert 49



insert 4



3. (5%) In the beginning, the two TAs must decide which of them transfers the message first. (Assume TA_1). In the end of the game, there is a verification to judge if TA_1 tells a lie. If TA_2 finds that TA_1 is a liar, then TA_1 loses the game.
   FLOW :

   (1) TA_1 chooses which one to throw (e.g. *rock*) and a random string (e.g. 94879487), concatenates them to a string (*rock*94879487), and takes the string as input to SHA256.

   (2) TA_1 transfers the output of SHA256 to TA_2.

   (3) When TA_2 receives the message from TA_1, TA_2 decides which one to throw (e.g. paper) and transfers to TA_1.

   (4) TA_1 transfers the plaintext (*rock*94879487) to TA_2, and TA_2 can use the plaintext and the hashed message received to verify whether TA_1 has told a lie.

   Now both of them know who wins the game. (In the example above, TA_1 wins.)
   There are two nice properties in hash function like SHA256. One is that there is almost no collision in SHA256. (Take any two different strings as inputs to the hash function, you will get two different outputs.) Besides, SHA256 is a one-way function. That is, when you see the output of hash, you are not able to know what the input is. (The random string in the flow is to enlarge the input space so that TA_2 cannot know TA_1s raw input by brute-force search.) Now lets consider the flow, TA_2 cannot know what TA_1 throws due to the one-way property before FLOW 3. All TA_2 can do is randomly choose one. And TA_1 cannot change his previous choice after receiving the message sent from TA_2 because TA_1 is not able to generate another input string such that its hash value matches the first-sent message. If TA_1 tells a lie, TA_2 will know by the verification. This is an easy example of how hash can be used in the real word.

4. (6%)

   (a) $h_1(k) = k \mod 7$
   $h_2(k) = \lfloor \frac{k}{7} \rfloor \mod 7$

k	h1(k)	h2(k)
6	6	0
31	3	4
2	2	0
41	6	5
30	2	4
45	3	6
44	2	6

The following are 2 hash tables over time. Columns show which element is inserted.

Table 1: table for $h_1(k)$

	6	31	2	41	30	45	44
0							
1							
2			2	2	30	30	44
3		31	31	31	31	45	31
4							
5							
6	6	6	6	41	6	6	6

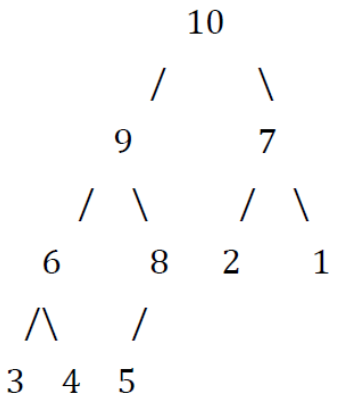
Table 2: table for $h_2(k)$

	6	31	2	41	30	45	44
0				6	2	2	2
1							
2							
3							
4						31	30
5					41	41	41
6							45

(b) Any element such that the insertion leads to an infinite sequence of displacements is a valid answer. For instance, we insert element 3. And all the elements included in the sequence of displacements are $[3, 31, 30, 44, 45, 2]$.

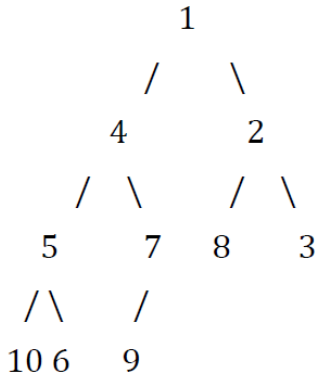
Problem 2. Heap (15%)

1. (3%)



in-order number sequence : 3, 6, 4, 9, 5, 8, 10, 2, 7, 1

2. (3%)



in-order number sequence : 10, 5, 6, 4, 9, 7, 1, 8, 2, 3

3. (3%)

```
AtMost (int q, Node head){
    if (head != NULL && head->value <= q) {
        print head->value
        AtMost(head->right)
        AtMost(head->left)
    }
}
```

The algorithm traverses its two children only when the root is qualified. Thus, the function will be called for at most $3k$ times. The time-complexity is $O(k)$.

4. (6%) For each non-leaf node in $array[i]$, its left-child and right-child are stored in $array[2i]$ and $array[2i+1]$, respectively. For example, the root is stored in $array[1]$, and its left-child and right-child are stored in $array[2]$ and $array[3]$, respectively.

   For a heap with depth=H, it takes 0 action to modify the leaves; 1 action to modify nodes at depth (H-1); 2 actions for nodes at depth (H-2) .... and at most H actions to modify the root at depth 0.

   The total time $= H + 2 \times (H-1) + 4 \times (H-2) + \cdots + 2^{H-1} \times 1$

   Let $S = H + 2 \times (H-1) + 4 \times (H-2) + \cdots + 2^{H-1} \times 1$

   $2S = 2H + 4 \times (H-1) + 8 \times (H-2) + \cdots + 2^{H} \times 1$

   $2S - S = 2^{H+1} - 2 - H = S$

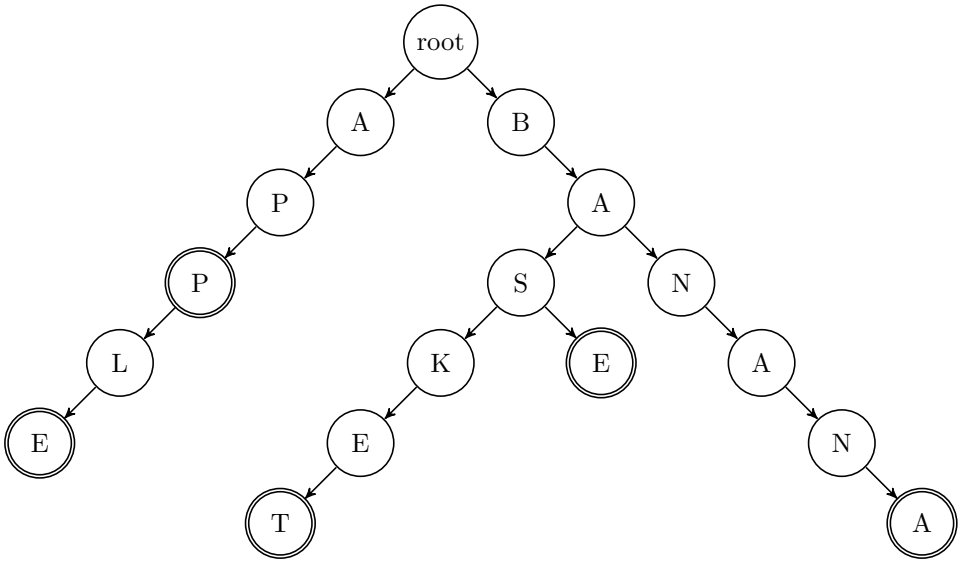
   and $H = \log(N+1) - 1$

   $S < 2^{\log(N+1)} - 1 - \log(N+1) \Rightarrow O(N)$

   You can also refer to lecture PPT for the proof.

Problem 3. Trie Trie See (20%)

1. (5%)



2. (5%)

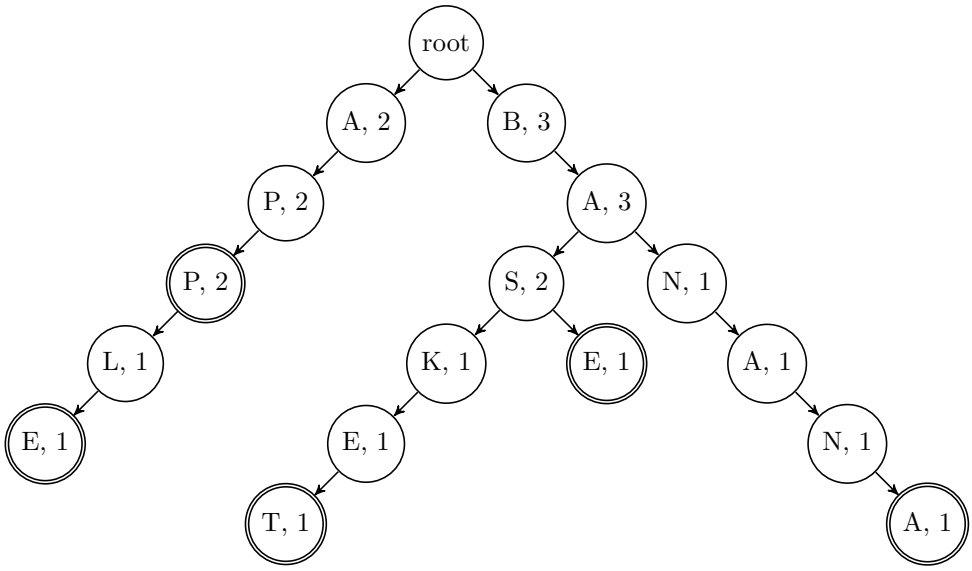
```
void insert(char word[], int N){
    struct Node *cur = ROOT;
    for(int i=0;i<N;i++){
        int no = word[i] - 'a';
        if (cur->children[no] == NULL){
            cur->children[no] = new_node();
        }
        cur = cur->children[no];
    }
    cur->is_word = 1;
}

void delete(char word[], int N) {
    struct Node *cur = ROOT;
    for(int i=0;i<N;i++){
        int no = word[i] - 'a';
        if (cur->children[no] == NULL){
            return;
        }
        cur = cur->children[no];
    }
    cur->is_word = 0;
}

int query(char word[], int N) {
```

```
26      struct Node *cur = ROOT;
27      for(int i=0;i<N;i++){
28          int no = word[i] - 'a';
29          if (cur->children[no] == NULL){
30              return 0;
31          }
32          cur = cur->children[no];
33      }
34      return cur->is_word;
35  }
```

For each function, there is only one for-loop which runs at most N times. So we will traverse N+1 node at most. In the *insert* function, we will malloc N nodes at most and each malloc will run in constant time.

3. (5%) First construct a trie T1 from the N words, and then another trie T2 for all the N reversed words. For example, if the words are as in Problem 4.1, then the second trie is constructed from elppa, ananab, ppa, esab and teksab.

   While building the two tries, one additional step is to accumulate a count value (can simply use the tag variable) whenever a node is traversed. In other words, the tag on each node indicated how many end-of-word nodes are in the subtree rooted at the node. For example, if the words are as in Problem 4.1, then the trie T1 would be as follows (where in each node the tag is added):



Obviously the above can be done in $O(\sum_{i=1}^{N} |W_i|)$ time. The rest is just to output the tag on the node of S in the trie according to the type of queries, which can be done in $O(|S|)$ time. Pseudo code is as follows:

```
construct tries T1, T2 for original words and reversed words
    , respectively, while maintaining tag as stated above
for each query q = (S, type) do
    if type is prefix:
        let T = T1
    else:
        let T = T2
    if S is not on the trie T, i.e. query(S) == 0: output 0
       and continue
    locate the node n on T for S
    output n.tag
```

4. (5%)

```
construct a trie T from the N words
for each leaf node L of T: mark L as lose
do
    for each non-marked node L of T whose children are
       marked:
        if any child of L is marked lose: mark L as win
        else: mark L as lose
until all nodes are marked

if T.root is marked win:
    output the first player has a winning strategy
else:
    output the second player has a winning strategy
```

Note that this algorithm, particularly the do-until part, can be implemented with recursion. The mark operation can be done with the tag variable.

Analysis: In each step, the current word has to be the prefix of at least one of the N words. This means a state of the game can be viewed as a node on the trie, and the string from the root to the node is the current word. In this way, each move of a player is just moving down the trie by one node.

Now we want to find out whether a player has a winning strategy on each node if it's his/her turn. Clearly on the leaf nodes the player is going to lose, since he can't move down anymore. So we mark each leaf node as lose. If any of the children of a node is marked lose, the player can simply go down to that node, so that the other player will definitely lose. In this case we mark the node to be win. Otherwise, we mark the node as lose. Finally, the mark on the root node indicates whether the first player has a winning strategy.

With a recursive approach, each node will be visited exactly once, so the time complexity is linear to the sum of length of the N words. (You don't need to show this.)

Problem 4. Machine Manager (Programming problem) (20%)

First of all, this problem has relatively large amount of requests, i.e., about 10^8 requests. Therefore, even just storing all the requests without doing anything will cause large usage of memory. Since each request consisting of $(L+1)$ characters, total memory usage will be $(L+1)*Q$ bytes (`char` takes 1 byte). Even for 40% points, it will take about `300 MB` which already exceeds the memory limit(`256 MB`). Therefore, we should handle the requests sequentially without memorizing it.

To deal with a request, we need to check or activate some given machine. Since we already know that we can't afford storing all the requests, we should at least store the state of machine. Then, how many states should we keep?

From the code generating requests, we can find that the character of machine's name is something ended with `63`. In C, and(`&`) is bit-wise and, `63` is $\texttt{111111}_2$. Thus, the character will have only first 6 least significant bits being `1`, i.e., only 2^6 choices. Thus, there are at most $(2^6)^L$ different machine names. Supposed we assign some memory space for each machine name, we can easily tag on its space and check whether it's active or not by the tagging.

The most easily way to assign a memory space to something and have $O(1)$ random access time is using array. In C, array index should be non-negative integer. But, now, the machine name is a string which cannot directly be an index of an array. Therefore, we should map(hash) each string to an integer. However, if the hashing function is too bad, some collisions may happen. Theoretically, we need a hash table with $((2^6)^L)^2$ slots to lower the expected collisions less than 1. However, if we choose hash function carefully, in this specific problem, we can only need a hash table with exactly $(2^6)^L$ slots without any collision. Since each character only takes 6 bits, we can concatanate L characters as $6 \times L$ bits as an integer. Code showed as follow:

```
int hash_value( char* s , int L ){
    int ret = 0;
    for( int i = 0 ; i < L ; i ++ )
        ret = ( ret << 6 ) | s[i];
    return ret;
}
```

For 40% points, we have $L \leq 3$. Thus, we need 2^{18} slots. We can just create an `int` array which will take $2^{18} \times 4 = 2^{20} = \texttt{2MB}$ which easily fit in memory limit.

For 100% points, if we still create an `int` array, the memory usage will be $2^{30} \times 4 = 2^{32} = \texttt{4GB}$ which exceeds memory limit much. Even you use a `char` array, it will take $2^{30} = \texttt{1GB}$ memory space which still can't fit in memory limit. Then, how much space could a slot take? we can just calculate it out as $\frac{\texttt{256MB}}{2^{30}} = \frac{2^{28}\texttt{B}}{2^{30}} = \frac{2^{31}\texttt{bits}}{2^{30}} = \texttt{2bits}$. Thus, for each slot, we can barely use up to 2 bits to store it. Actually, it's enough. Since we only need to set it as `active` and check it is `active`

or `inactive`. It's an 0/1 state. Thus, we just need to use 1 bit for each machine. The method to map each integer to some bit can be achived by bit operation. Code showed as follow:

```
#define SLOTS (1<<30)
#define SIZE 32
int hash_table[SLOTS/SIZE]
void activate( int id ){
  hash_table[id / SIZE] |= (1 << (id % SIZE))
}
bool check( int id ){
  return (hash_table[id / SIZE] >> (id % SIZE)) & 1;
}
```

Problem 5. Yea, I'm a router. (Programming problem) (20%)

Here we provide a solution with trie. The basic concept is to first (1) build a trie with all the given masks, and then (2) traverse the trie from root for each of the given IP addresses. If the traversing reachese an accept node, output "TRUE" for the corresponding IP address; otherwise, output "FALSE".

Pseudocode

In the following two sections are two pseudocode snippets for the previously mentioned 2 steps.

Building Trie with Masks

The following `build()` function takes a trie and a mask, and augments the trie to accept this mask. The `root` parameter is the root node of the trie, and a node supports these data members: `left`, `right`, and `accept`. The `accept` field stores a boolean indicating that the trie, *i.e.*, a compound subnet mask, accepts any IPs beyond this point.

```
function build(root, mask_ip, mask_bits):
    current = root
    for i in 0 to mask_bits-1:
        if mask_ip[i] == '0':
            # check whether we can go left
            if !current.left:
                current.left = create_node()
            # go left
            current = current.left
        else:
            # check whether we can go right
            if !current.right:
                current.right = create_node()
            # go right
            current = current.right
    current.accept = True
```

Time Complexity: If we look into this function `build()`, you'll find it runs in O(1) time since the only loop in the function would not run for more than 32 times, given that an IP is 32-bit long. Therefore, the total time complexity for building the trie with `M` masks is O(M).

Traversing Trie with IP addresses

The function `verify()` below checks if an IP is accepted by the trie. It returns `True` if the IP address is accepted, `False` otherwise. Note that the parameter `ip` is a bit string.

```
function verify(root, ip):
    current = root
    for i in 0 to 31:
        if ip[i] == '0':
            # check whether we can go left
            if !current.left:
                return False
            current = current.left
        else:
            # check whether we can go right
            if !current.right:
                return False
            current = current.right
        if current.accept:
            return True
    return False
```

Time Complexity: Similar to the `build()` function, each calling of `verify()` will be O(1) time complexity, and therefore, for verifying `N` IP addresses, the process takes O(N) time complexity.

Summary

With both `build()` and `verify()`, we can do the firewall checking in O(M+N) time complexity. However, please note that this is NOT the only solution to the problem.