First Name: Zhicheng

Last Name: Yan

Student ID: 1002643142

First Name: Zhihong

Last Name: Wang

Student ID: 1002095207

First Name: Jialiang

Last Name: Yi

Student ID: 1002286929

This submission has been prepared using LaTeX.

Writer: Jialiang Yi

Reader: Zhihong Wang, Zhicheng Yan

Question 1
A *Scheduler* S consists of a set of *threads*; each thread is a tuple $t = (id, status)$ where id is a distinct positive integer and $status \in \{A, R, S\}$; intuitively, A means active, R means ready to be scheduled, and S means stalled. The operations that Scheduler S supports are:

$NewThread(t)$: Given a thread $t = (id, status)$, add thread t to S. You can assume that the id of t is different from the id of any thread currently in S (so that all the threads in S have distinct ids).

$Find(i)$: If S has a thread $t = (i,)$ then return t, else return -1.

$Completed(i)$: If S has a thread $t = (i,)$ then remove t from S, else return -1.

$ChangeStatus(i, stat)$: If S has a thread $t = (i,)$ then set the $status$ of t to $stat$, else return -1.

$ScheduleNext$: Find the thread t with smallest id among all the threads whose $status$ is R in S, set the $status$ of t to A and return t; if S does not have a thread whose $status$ is R then return -1.

In this question, you must describe how to implement the Scheduler S described above using a *single augmented AVL tree* such that each operation takes $O(logn)$ time in the worst-case, where n is the number of threads in S. Since this implementation is based on a data structure and algorithms described in class and in the AVL handout, you should focus on describing the extensions and modifications needed here.

a. Give a precise and full description of your data structure. In particular, specify what data is associated with each node, specify what the key is at each node, and specify what the auxiliary information is at each node. Illustrate this data structure by giving an example of it (with a small set of threads of your own choice).

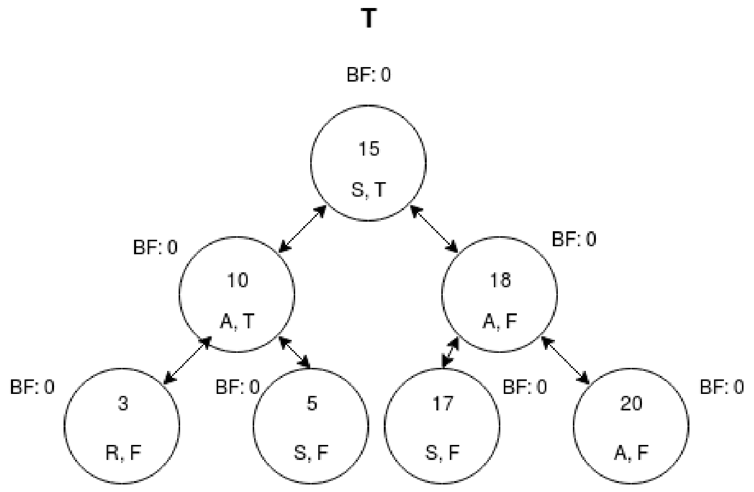
Solution:

According to the functionalities and requirements of Scheduler, we are using a single augmented AVL tree to implement the Scheduler; an AVL tree augmented a boolean to show whether its subtree have a thread with *status R*.

Each node has a $thread - id$ as the key where id is a distinct positive integer.

Each node also has following fields: $lchild$, $rchild$ and $parent$ which are pointers to the leaf and right children and parent of the node; and integer $balance\ factor$ to represent the balance status of the current tress node; a string $status$ showing corresponding status where $status \in \{A, R, S\}$; a boolean r_exist indicating whether a thread or threads with status R exist under the subtree of such node and whether itself is a node with status R.

For example: Consider the following single-augmented AVL Tree T, representing a Scheduler.



using the root node to demonstrate; the root node has id 15 as its key. Its status is S and there exist a node with status R in its subtree, which is node 3 as the graph show.

For the new field argumented for this AVL tree, a boolean *r_exist* is indicating whether a thread or threads with status R exist under the subtree of such node and whether itself is a node with status R. In some operations, such as $INSERT$, $DELETE$ or any other operations that effects the struture of the tree, we have to rebalance the tree to maintain AVL properties. Thus, updating nodes ancestors is necessary. As *r_exist* is simply a boolean, when we need to update its information, we can simply change it in constant time, which is $O(1)$. Next, we need to consider the total cost of updating ancestors operation. If a node's status change from A to S or from S to A, we don't have to update anything.

However, update is required when changes are related to R. If a node with status R is changed to A or S, check its sibiling tree. If sibiling has R, then its parent reamin the same; otherwise, when its sibiling tree does not have R, update it's parent's *r_exist* to F. In opposite, if a node's status changes from A or S to R, check its sibiling tree again; if sibiling has R, remain parent's status. Otherwise, change parent's *r_exist* to T. In conclusion, for every update, we first check the sibling tree and then update parent's field accordingly. We perform the same operation to other ancestors of the node. Therefore, only a single path of node will be update in worst case. When there are n nodes in the tree, there will be maximum logn updates in total. Summing up, each update takes constant time and there are maximum logn updates in total.

b. Describe the algorithm that implements each one of the five operations above, and explain why each one takes $O(logn)$ time in the worst-case. *Your description and explanation should be in clear and concise English*. For the operations $ChangeStatus(i, stat)$ and $ScheduleNext$, you should also give the algorithms high-level pseudocode.

Solution:

$NewThread(t)$: Given a thread $t = (id, status)$, add thread t to S.

Algorithm: S is the AVL tree representing a Scheduler. Create a new tree node, say x, let $x.key = id$ and $x.status = status$ and Perform $INSERT$, method from API library on node x into tree S.

Runtime: according to the algorithm of *NewThread*, a new tree node can be created in constant time which is $O(1)$. Next, *INSERT* method from API library of an AVL tree depends on the height of the tree. When n is the number of threads in tree S, in worst case, it takes $O(logn)$ time to complete insert operation (According to lecture, tutorial, CLRS and properties of AVL tree). As *INSERT* in API include the step of balancing the tree in order to maintain AVL properties, the sturcture of the tree will be mutate. Thus, updating fields of ancestors is required, *r_exist* in this case. As we have described in part a, each update takes constant time and there are maximum logn updates in total, which gives us $O(logn)$ and will not effect the overall performance. Therefore, *NewThread* operation takes $O(logn)$ time in the worst-case.

$Find(i)$: If S has a thread $t = (i, -)$ then return t, else return -1.

Algorithm: simply treat the single augmented AVL tree S which implementing a Scheduler as an ordinary binary search tree. Perform *Search* method from API library; compare the key of the node and the input thread's id. If the two matches, return the node which is a *thread* of a tuple (i, $i.status$). Otherwise, return -1.

Runtime: according to the algorithm of *Find*, we simply used the *Search* method from API library. According to the properties of AVL tree and Binary Search tree, search operation depends on the height of the tree. When n is the number of threads in tree S, in worst case, it takes $O(logn)$ time to complete comparison and return searching result.(According to lecture, tutorial, CLRS). Therefore, *Find* operation takes $O(logn)$ time in the worst-case.

$Completed(i)$: If S has a thread $t = (i, -)$ then remove t from S, else return -1.

Algorithm: S is the AVL tree representing s Scheduler. Use the *Search* operation in API; $Search(i)$. Then, if t exist,perform the *Delete* method from API library; Delete(i). If t does not exist in S, return -1.

Runtime: according to the algorithm of *Complete*, *Search* operations is performed in the beginning, which has runtime $O(logn)$ as described. In worst case, the target thread is found in tree S, *Delete* operation is performed. Referring to description in API, runtime of deleting operation is related to the height of the tree. After deleting the node, which completes in constant time, the ancestors of the deleted node need to be updated, r_exist in this case. As we have described in part a, each update takes constant time and there are maximum logn updates in total, which gives us $O(logn)$ and will not effect the overall performance.Thus, *Delete* is $O(logn)$. In conclusion, it takes $O(2logn)$ to complete the algorithm in worst case. Therefore, *Complete* operation takes $O(logn)$ time in worst-case.

$ChangeStatus(i, stat)$: If S has a thread $t = (i, -)$ then set the *status* of t to *stat*, else return -1.

Algorithm: perform *Find* operations as described above to find whether a node corresponds to the input thread exist in tree S. If such node does not exist in tree S, simply return -1. If such node do exists in tree S, say node x, change x's status to the input thread status (Ex: $x.status = stat$). Lastly, update ancestors of node x accordingly (r_exist and BF in this case where r_exist is a boolean representing whether a node with status R exists in subtree and BF is the balance factor).

Pseudocode:

```
global variable: Single augmented AVL tree representing a Scheduler.
Precondition: input i is a distinct positive integer and
              stat in {A, R, S}
Postcondition: If S has a thread t = (i, -) then set status of t
              to stat, else return -1.

   ChangeStatus(i, stat){
     if (Search(i)){
         node = Find(i);
         node.status = stat;
         Update node's ancestors;
        }else{
         return -1;
7           }
9         return;
8       }
```

Prove correctness of algorithm:
There is no loop or recursion in *ChangeStatus*. It performs actions base on the condition of Search(i); if i exists, from $line2$, by calling $Find(i)$ operation, *node* is the thread we are modifying. Then change *node*'s *status* to *stat* and update its ancestors; Otherwise, return -1. Therefore, *node* will not be null and *ChangeStatus* must terminate.

Runtime: according to the algorithm of *ChangeStatus*, *Find* operation is performed at the beginning, which takes $O(logn)$ as described above. As we are considering worst case, thread t is found in tree S. Changing the status of the target node completes in constant time, $O(1)$. Lastly, update ancestors. Runtime of updating ancestors is related to the height of the tree S. Only the ancestors of currently processing node will be updated; only a single path of nodes are updating. In worst case, which is when the target node is a leaf, and all of its ancestors needs to be update, assuming S has n nodes, maximum logn nodes will be updated. Each update will be updating the augmented field r_exist, which can be done in constant time, O(1). Thus, maximum logn updates and each update takes constant time gives us runtime $O(logn)$, which will not effect the overall runtime of the algorithm, maintaining as $O(logn)$ after all. Summing up, it takes $O(2logn + 1)$ to complete the algorithm in worst case. Therefore, *ChangeStatus* operation takes $O(logn)$ time to complete in worst-case.

ScheduleNext: Find the thread t with smallest *id* among all the threads whose *status* is R in S, set the *status* of t to A and return t; if S does not have a thread whose *status* is R then return -1.

.

Algorithm: According to the properties of AVL tree, the smallest node must locate at the bottom left of the tree. Therefore, to find the smallest node, we must trace down the path from the left until it reaches a leaf.

Base on this character, *ScheduleNext* takes a root of an AVL tree, say *root* as input. Base cases are: 1) when $root.left$ and *root.right* are NULL, if root has status R, change *root*'s status to A, then update root's ancestors (r_exist and BF in this case where r_exist is a boolean representing whether itself's status is *R* and whether a node with status *R* exists in subtree; *BF* is the balance factor); else, return -1. 2) when *root* have both left and right child, but no node with status *R* exist in their subtrees and their status is not *R*; if root has status *R*, change *root*'s status to *A*, then update root's ancestors; else, return -1.

Otherwise, Case 1, if *root* has left child and subtree of *root*'s left child have a node with *status* as *R*, ((root.left).r_exist), then root = root.left, call $ScheduleNext(root)$, trace down from the left recursively. Case 2, if *root* has no left child or *root*'s left child does not have a node with *status* as *R*, if *root*'s right child is not NULL and root's right child has status *R* or a node with status *R* under its subtree, root = root.right, call $ScheduleNext(root)$, trace down recursively;

Pseudocode:

```
global variable: Single augmented AVL tree representing a Scheduler.
Precondition: input root is the root of the AVL tree.
Postcondition: Find the thread $t$ with smallest $id$ among all the
        threads whose $status$ is $R$ in $S$, set the $status$ of
        $t$ to $A$ and return $t$; if $S$ does not have a thread
        whose $status$ is R then return −1.\\

    ScheduleNext(root){
      if ((root.left and root.right are NULL)) {
        if (root.status == "R") {
                root.status == "A";
                Update root's ancestors;
                return root;
        }else {
                return −1;
          }

      } else if ((root.left.r_exist == F and root.right.r_exist == F))
        if (root.status == "R") {
                root.status == "A";
                Update root's ancestors;
                return root;
        } else {
```

```
16.                 return -1;
17.         }
18.
19.     } else {
20.       if ((root.left.r_exist == T)) {
21.                 root = root.left;
22.                 ScheduleNext(root);
23.       } else {
24.             if ((root.right.r_exist == T)) {
25.                 root = root.right;
26.                 ScheduleNext(root);
27.             }
28.         }
29.     }
30. }
```

Runtime: according to the algorithm, base case of $ScheduleNext$, $line 1 - 17$ change the status of the node, update the node's ancestors and return the node. Changing the status is done in constant time; $O(1)$.Next is to Update ancestors. Runtime of updating ancestors is related to the height of the tree S. Only the ancestors of currently processing node will be updated; only a single path of nodes are updating. In worst case, which is when the target node is a leaf, and all of its ancestors needs to be update, assuming S has n nodes, maximum logn nodes will be updated. Each update will be updating the augmented field *r_exist*, which can be done in constant time, O(1). Thus, maximum logn updates and each update takes constant time gives us runtime $O(logn)$, which will not effect the overall runtime of the algorithm, maintaining as $O(logn)$ after all.

Recursion in this functions depends on the status of the currently processing node. In worst case, if its left child is not NULL and exists a node with status R in left child's sub tree or its left child is NULL, right child is not NULL and its status is not R, we trace one level down and recursively call $ScheduleNext$. Clearly, the number of time of recursive calls is related to the height of the tree. Say there are n nodes in tree S, the height of S is $logn$. In worst case, the smallest node with R is a leaf, tracing from root to leaf is required; logn recursive calls will happened.

Summing up, in worst case, there will be logn recursion calls, which is $O(logn)$ and when algorithm reaches base case in final recursion loop, the runtime is also $O(logn)$; O(2logn) in total. Therefore, in worst case, $ScheduleNext$ operation takes $O(logn)$ time to complete.

Writer: Zhicheng Yan

Reader: Jialiang Yi, Zhihong Wang

Question 2
The *PowerofTwoChoices* hashing scheme is a method of hashing that reduces the number of collisions when searching in a hash table. Suppose we have a hash table T with m slots and two *independent* hash functions $h1$ and $h2$. Each hash function satisfies the *Simple Uniform Hashing Assumption* (SUHA): every key in the universe U hashes with equal probability to any hash slot of T, independently from the hash values of all other keys in U. When we insert a new element x, we add it to the chain in one of the two hash slots $T[h1(x)]$ or $T[h2(x)]$; we pick the slot in which the chain is shorter. If the chains are same length, we add x to the chain in $T[h1(x)]$.

a. If the number of empty slots in T is k before inserting X, what is the probability that x is inserted into an empty slot? Justify your answer.

Solution:
The case that x is inserted into an empty slot is either of h_1 or h_2 hashes to the empty slot(i.e. Case that is not both h_1 and h_2 hashes to non-empty slot).
Let e be the case that x is inserted into empty slot, $\bar{e}$ is cast that it is not.
The probability that both hashes to non-empty slot:
$P(\bar{e}) = (\frac{m-k}{m})^2 = \frac{(m-k)^2}{m^2}$
So, the probability that x inserted to the empty slot(at least one of the hash function hashed to the empty slot):
$P(e) = 1 - P(\bar{e}) = 1 - \frac{(m-k)^2}{m^2}$

b. Suppose that m = 4, and $T[0]$ contains 6 elements, $T[1]$ contains 3 elements, and $T[2]$, $T[3]$ contain 9 elements each. What is the *expected* length of the chain x is inserted into, *not counting* x itself? Justify your answer.

Solution:
There is totally 16 cases, the probability of each case is $\frac{1}{16}$
The case that x is inserted into $T[0]$:
h_1 hashes to $T[0]$ and h_2 hashes to $T[0]$

h_1 hashes to $T[0]$ and h_2 hashes to $T[2]$
h_1 hashes to $T[0]$ and h_2 hashes to $T[3]$
h_1 hashes to $T[2]$ and h_2 hashes to $T[0]$
h_1 hashes to $T[3]$ and h_2 hashes to $T[0]$
So, the probability that x inserted into $T[0]$ is
$P(0) = \frac{1}{16} \times 5 = \frac{5}{16}$
The case that x is inserted into $T[1]$:
h_1 hashes to $T[0]$ and h_2 hashes to $T[1]$
h_1 hashes to $T[1]$ and h_2 hashes to $T[0]$
h_1 hashes to $T[1]$ and h_2 hashes to $T[1]$
h_1 hashes to $T[1]$ and h_2 hashes to $T[2]$
h_1 hashes to $T[1]$ and h_2 hashes to $T[3]$
h_1 hashes to $T[2]$ and h_2 hashes to $T[1]$
h_1 hashes to $T[3]$ and h_2 hashes to $T[1]$
So, the probability that x inserted into $T[1]$ is
$P(1) = \frac{1}{16} \times 7 = \frac{7}{16}$
The case that x is inserted into $T[2]$:
h_1 hashes to $T[2]$ and h_2 hashes to $T[2]$
h_1 hashes to $T[2]$ and h_2 hashes to $T[3]$
So, the probability that x inserted into $T[2]$ is
$P(2) = \frac{1}{16} \times 2 = \frac{1}{8}$
The case that x is inserted into $T[3]$:
h_1 hashes to $T[3]$ and h_2 hashes to $T[2]$
h_1 hashes to $T[3]$ and h_2 hashes to $T[3]$
So, the probability that x inserted into $T[2]$ is
$P(3) = \frac{1}{16} \times 2 = \frac{1}{8}$
Let $l(i)$ be the length of the chain, where i is the index of the table
Then, the expect length is:

$$\begin{aligned}
E(l) &= \Sigma_{i=0}^{3}(l(i)P(i)) \\
&= l(0)P(0) + l(1)P(1) + l(2)P(2) + l(3)P(3) \\
&= 6 \times \frac{5}{16} + 3 \times \frac{7}{16} + 9 \times \frac{1}{8} + 9 \times \frac{1}{8} \\
&= \frac{87}{16} \\
&= 5.4375
\end{aligned}$$

Therefore, the expected length of the chain x is insert into is 5.4375

Writer: Zhihong Wang

Reader: Jialiang Yi, Zhicheng Yan

Question 3
You are to write an efficient algorithm for the following problem. Your algorithm is given a sequence (of finite, unknown length) of English words, one at a time. Since we cant account for misspellings, slang, colloquialisms, etc., the universe of words is infinite but assume that the number of distinct words in our sequence is bounded above by a known constant l. A *query* operation can occur at any point between any two inputs in the sequence. When a *query* occurs, the algorithm must return, in sorted order, the most frequent word beginning with each letter. That is, it must return the most frequent word beginning with a, the most frequent word beginning with b, etc. Ties in frequency can be resolved by taking the word occurring first in alphabetical order. If no words beginning with a particular letter have been input, simply return null for that letter. Assume words are all lowercase.

Describe a simple algorithm that solves the above problem with the following time complexity:

• $\Theta(1)$ *expected time* to process each input, under some **reasonable assumptions** that you should state

• $\Theta(1)$ $worst - case$ to perform each *query* operation.

To answer this question, you must:
1. State which data structure(s) you are using, the items contained within, and any assumptions you are making.
2. Explain your algorithm clearly and concisely, in English.
3. Give the algorithms pseudo-code, including the code to process an input word and the code for query.
4. Explain why your algorithm achieves the required time complexity described above, given the assumptions in 1.

Solution:

Step 1. State which data structure(s) you are using, the items contained within, and any assumptions you are making.

We will use Hash table.

Note: all hash functions appeared in our algorithm are all by SUHA.

Let $T1$ represents the first Hash table which is large enough to contain all inputs without collision (i.e. the size of $T1$ is larger than l, the bound of the number of distinct words in our sequence); $T2$ represents the second Hash table which has 26 empty buckets in order to contain most frequent words beginning with each letter. Two hash functions $f1$ and $f2$ (by SUHA), $f1$ (for $T1$) can give an unique $key1$ for any English word input, so the element in $T1$ can be represented by $T1[key1]$ (i.e. $key1$ can be the same as the input word, just like python's dictionary, which let we are able to search the input in $T1$ directly); $f2$ (for $T2$) can give 26 different keys (i.e. 1,2,3,...,26) for most frequent words beginning with 26 letters (i.e. 1 for a, 2 for b,...., 26 for z).

Let W represents the node which will stored in any $T1$ and $T2$. Every node W of T has following fields: an English word *word*; an integer *size* (initial it by 1) which shows the number of same words.

Step 2. Explain your algorithm clearly and concisely, in English.

Note: Every input is one English word. i.e. Input one English word one time.

The algorithm can be develop base on Hash table.

First, we will initial two empty Hash table $T1$ and $T2$ as global variables. $T1$ represents the first Hash table which is large enough to contain all inputs without collision; $T2$ represents the second Hash table which has 26 empty buckets in order to contain most frequent words beginning with each letter.

We also has 2 hash functions $f1$ and $f2$ (by SUHA). $f1$ (for $T1$) can give an unique *key* for any English word input, so the element in $T1$ can be represented by $T1[key]$; $f2$ (for $T2$) can give 26 different keys (i.e. 1,2,3,...,26) for most frequent words beginning with 26 letters (i.e. 1 for a, 2 for b,...., 26 for z)

Assume the input is a English word x, our algorithm will construct a node *New* with x in *word* and $size = 1$.

Then using the *word* and it's beginning letter as inputs of $f1$ and $f2$ (by SUHA) to generate an unique $key1$ (i.e. $key1$ can be the same as the input word, just like python's dictionary, which let we are able to search the input in $T1$ directly) and $key2$ (i.e. 1,2,3,...,26) .

Then we will apply search method on *New*. If x is first occurred, we will insert *New* into a empty bucket $T1[key1]$; if x occurred before, we will find the node *Old* which contains the same word and assign it's *size* to be $size + 1$.

Then we will use *New* (if first occurred) or *Old* (if occurred before) to compare with the corresponding node (same beginning letter) in bucket $T2[key2]$. If the bucket is empty, we simply insert *New* or *Old* into this bucket. If the *size* of *New* or *Old* is greater than the node in bucket, then the corresponding node's fields will be replaced by *New*'s or *Old*'s fields. If the *size* of *New* or *Old* is equal to the node in bucket and the *word* of *New* or *Old* is smaller than that node(i.e. if "apple" inputted 5 times, "ace" inputted 5 times and there are no other words begin with "a", "ace" is smaller because the "c" in "ace" is smaller than "p" in "apple), then then the corresponding node's fields will also be replaced by *New*'s or *Old*'s fields. Else, we do nothing.

When *query* occur, we will return *word*s form $T2$ in key's order (i.e. 1 to 26).

Step 3. Give the algorithms pseudo-code, including the code to process an input word and the code for query.

```
global variable: Hash table T1 (large enough)
global variable: Hash table T2 (26 buckets)
global variable: a list of keys (contains 26 sorted keys in T2)

Precondition: The input is a English word.
Postcondition: return a unique key for T1.
key f1(a English word)

Precondition: The input is a English letter (a to z).
Postcondition: return a key for T2 (totally 26 sorted keys).
key f2(a English letter)

Precondition: The input is a English word.
Postcondition: return nothing.
processing_word(a English word)
1.  k2 = f2(this English word's first letter);
2.  build a node New for this word;
3.  if T2[k2] is NULL
4.      T2[k2] = New;
5.  search this word in T1;
6.  if cannot find this English word
7.      k1 = f1(this English word);
8.      T1[k1] = New;
9.      if New.size > T2[k2].size
10.         replace T2[k2]'s fields by New's fields;
11.     if New.size == T2[k2].size and New is smaller than T2[k2]
12.         replace T2[k2]'s fields by New's fields;
13.     else
14.         return;
15. else
16.     k3 = the key to this word in T1
17.     T1[k3].size++;
18.     Old = T1[k3];
19.     if Old.size > T2[k2].size
20.         replace T2[k2]'s fields by Old's fields;
21.     if Old.size == T2[k2].size and Old is smaller than T2[k2]
22.         replace T2[k2]'s fields by Old's fields;
23.     else
24.         return;
```

```
25. return;

\\Precondition: The input is T2.
\\Postcondition: return the most frequent words
                 beginning with 26 letters in sorted order
list_of_words  query(T2)
1.  initial a list called result;
2.  add T2[keys[0].word] into result;
3.  add T2[keys[1].word] into result;
4.  add T2[keys[2].word] into result;
    ...
27. add T2[keys[25].word] into result;
28. return result;
```

Step 4. Explain why your algorithm achieves the required time complexity described above, given the assumptions in Step 1.

Prove correctness of our algorithm:

1. For function $f1$ and $f2$:

These function must return one correct key one time by assumption.

2. For function *query*:

The input value for *query* is a global Hash table $T2$, and the return value is a list of words.

There is no loop or recursion in this function.

Because $T2$ must exist and has 26 buckets by assumption, *keys* must exist and holds all keys of $T2$ by assumption, then $T2[keys[0-25]]$ must exist and the *result* of *query* must exist, so *query* must return.

3. For function *processingword*:

The input value for *processingword* is a English word, and the return value is nothing.

There is no loop or recursion in this function.

Because this function aim to change (or do not change) one of nodes in buckets of $T1$ and $T2$, then by line 3,6,9,11,13,15,19,21,23, all situations are handled, so by line 25, *processingword* must return.

(a),(b) will explain why our algorithm achieves the required time complexity described above, given the assumptions in Step 1:

(a). $\Theta(1)$ *expected time* to process each input, under some **reasonable assumptions** that you should state.

Our algorithm based on Hash table.

There is no loop or recursion in processing each input.

Let the time complexity to be $T(x)$.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

To prove the running time of my algorithm is $\Theta(1)$, we have to prove my algorithm is $O(1)$ and $\Omega(1)$.

Proof of $O(1)$:

To prove $T(x)$ is $O(1)$, we need to show for all input x, $T(x) \leq c$ for constant $c > 0$.

By line 5, search method in Hash table only require constant time.

By line 11 and 21, because the length of a word is a reasonable length in this question (by piazza's answer), so these code lines of words comparing are all in constant time.

By line 4,8,10,12,20,22, insertion and node(in Hash table bucket) modification on Hash table are only require constant time.

Then, all lines in *processingword* only require constant time, which is 1.

Then, for all input x, $T(x) \leq 1$.

So, there exists $c > 0$ such that $T(x) \leq c$.

Therefore, $T(x)$ is $O(1)$.

Proof of $\Omega(1)$:

To prove $T(x)$ is $\Omega(1)$, we need to show there exist an input x, $T(x) \geq c$ for constant $c > 0$.

Because, all lines in *processingword* only require constant time, which is 1.

Then, there exist an input x, $T(x) \geq 1$.

So, there exists $c > 0$ such that $T(x) \geq c$.

Therefore, $T(x)$ is $\Omega(1)$.

Therefore, $T(x)$ is $\Theta(1)$.

Therefore, $\Theta(1)$ *expected time* to process each input, under some **reasonable assumptions** that we stated.

(b). $\Theta(1)$ $worst - case$ to perform each *query* operation.

Our algorithm based on Hash table.

There is no loop or recursion in *query*.

Let the $worst - case$ time complexity to be $T(x)$.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

To prove the $worst - case$ running time of my algorithm is Θ(1), we have to prove my algorithm is $O(1)$ and $\Omega(1)$.

Proof of $O(1)$:

To prove $T(x)$ is $O(1)$, we need to show for all input x, $T(x) \leq c$ for constant $c > 0$.

By line 1, initialize a list as output cost constant time.

By line 2 to 27, adding words form nodes of $T2$ to the list only require constant time.

Then, all lines in $query$ only require constant time, which is 1.

Then, for all input x, $T(x) \leq 1$.

So, there exists $c > 0$ such that $T(x) \leq c$.

Therefore, $T(x)$ is $O(1)$.

Proof of $\Omega(1)$:

To prove $T(x)$ is $\Omega(1)$, we need to show there exist an input x, $T(x) \geq c$ for constant $c > 0$.

Because, all lines in $query$ only require constant time, which is 1.

Then, there exist an input x, $T(x) \geq 1$.

So, there exists $c > 0$ such that $T(x) \geq c$.

Therefore, $T(x)$ is $\Omega(1)$.

Therefore, $T(x)$ is Θ(1).

Therefore, Θ(1) $worst - case$ to perform each $query$ operation.