CSC263 A3

Zikun Chen, Yue Li, Yueen Ma

September 7, 2018

Question 1

(Written by Yue Li, read by Yueen Ma)

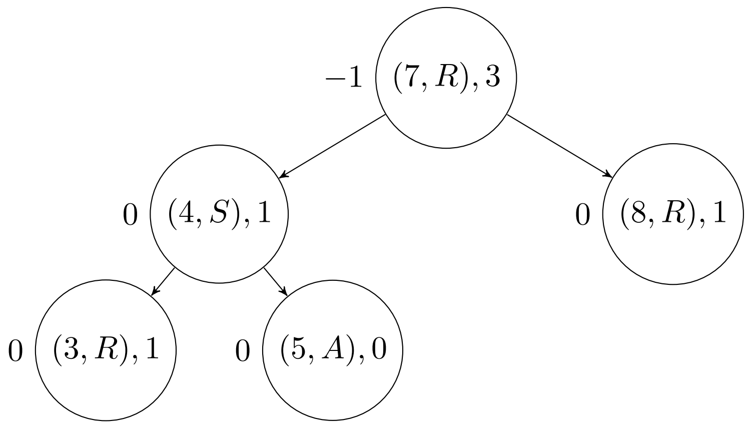
(a) **Data Structure:** augmented binary search AVL tree.
-Each node contains id and status of thread t, balance factor and an auxiliary attribute.

-Auxiliary information of each node:
R_number: the number of thread t with status R contained in tree rooted at current node (**include itself**)

-The key of each node is id

- Note: the order of nodes in tree in sorted by key(attribute id) in each node.

Example:
input: (7, R) (4, S) (3, R) (5, A) (8, A)
result:



1. (b)

- **NewThread(t):** (Similar to AVL Tree Insert)

***Description*:**
Insert a tuple t to tree T, trace a path from root downward and insert a new node with key t.id and status t.status, so as to preserve the binary search tree property, then rebalance tree and updating balance factor of each node involved in rotation(balancing). To update extra info R_number, simply find the new node's parent, and new_node.parent.R_number = new_node.parent.left.R_number + new_node.parent.right.R_number + r, if new node's parent itself is R status, $r = 1$; otherwise, $r = 0$. keep doing this until the pointer reaches the root of the tree.
(Note: If pointer reaches null, return 0)

Time complexity:
-Time to find the proper position to insert new node t in worst case depends on the height of tree T, which take at most $O(logn)$ time;

-Rebalancing and updating balance factors takes constant time, which takes at most $O(1)$ time;

-A change to an *R_number* attribute in a node t propagates only to ancestors of t in the tree, in worst case, All ancestors of t need to be updated, which is a path from new node t to root, then it takes at most $O(logn) time$

Therefore T(n) $\in O(logn) + O(1) + O(logn) = O(logn)$

- **Find(i):** (Similar to AVL Tree Search)

Description:
By using binary search tree property, updating pointer to node by comparing i with attribute id in each node, if find a node such that $i = pointer.id$, return (pointer.id, pointer.status) as a tuple, else finally pointer points to NULL, which means tree T doesn't contain a thread whose id is i, then return -1 as required.

Time complexity:
In worst case i is not the id of any thread in the tree T, and we need to trace the path from root to the deepest leaf, which is the height of tree T, then it takes at most $O(logn)$ time

Therefore worst case T(n) is $O(logn)$

- **Completed(i)**:(Similar to AVL Tree Delete)

Description:
Find the node in tree such that node.id = i by using binary search tree property, if such node doesn't exist, return -1; else, denote that node as n and delete n. During the rebalancing process, continuing updating extra info *R_number*. To update extra info R_number, simply find the removed

node's parent, and removed_node.parent.R_number = removed_node.parent.left.R_number + removed_node.parent.right.R_number + r, if new node's parent itself is R status, $r = 1$; otherwise, $r = 0$. keep doing this until the pointer reaches the root of the tree.

(Note: If pointer reaches null, return 0)

Time Complexity:
In worst case, the thread whose status is R is stored in the deepest leaf node n in the tree, time to find node n depends on the height of the tree, then it takes at most $O(logn)$;

During the process of deleting n, in worst case, we need to rebalance the tree, updating balance factor and *R_number* for all ancestors of node n, rotation and update attributes take constant time for each node, and the number of nodes need to update depends on the height of the tree(the longest path from node n to root), then in total it has to do at most $O(logn)$ times.

Therefore, worst case time complexity T(n) $\in O(logn) + O(1)O(logn) = O(logn)$

- **ChangeStatus(i, stat):**

pseudocode:

```
# HelpChangeStatus is a helper function to get the target AVL tree:
# It returns -1 if AVL tree rooted at root does not include a node with id i.
# Else, set the status of node with id i to stat.
# Note: root is the pointer to the root of the target AVL tree
def HelpChangeStatus(i, stat, root):
    if root == NIL:
            return -1
    else if (root.id == i):
        if (root.stat != R and stat == R):
            root.R_number += 1
            # Update extra info R_number
            while (root.parent != NIL):
                root.parent.R_number += 1
                root = root.parent
        if (root.stat == R and stat != R):
            root.R_number -= 1
            # Update extra info R_number
            while (root.parent != NIL):
                root.parent.R_number -= 1
                root = root.parent
        root.stat = stat
        return root
    else if (root.id > i):
            return HelpChangeStatus(i, stat, root.left)
    else if (root.id < i):
            return HelpChangeStatus(i, stat, root.right)

def ChangeStatus(i, stat)
    return HelpChangeStatus(i, stat, root);
```

Description:
-First, we find the node whose id = i by applying binary search tree property, if that node doesn't exist, return -1.

-If there exist such a node, denote it as n, we set the status of n to be stat.

-Extra info *R_number* needs to update in two cases as follows:

Case 1: Status of n change from A or S to R, then we need to add 1 to *R_number* of n and all ancestors of n.

Case 2: Status of n change from R to A or S , then we need to subtract 1 from *R_number* of n and all ancestors of n.

Time Complexity:
In worst case, node n that contains thread whose id is i is the deepest leaf of tree, then time to find node n takes at most $O(logn)$ times since the height of an AVL tree is $O(logn)$.

Changing the status of the thread takes $O(1)$.
Also in worst case we need to update extra info *R_number* for all ancestors of node n, which is a path from n to root of tree, the time it takes depends on the height of AVL tree, then it takes at most $O(logn)$ time.

Therefore, worst case time complexity T(n) $\in O(logn)+O(1)+O(logn) = O(logn)$

- **ScheduleNext:**

pseudocode:

```
# HelpScheduleNext is a helper function to get the target AVL tree:
# It returns -1 if AVL tree rooted at root does not include a node with status R.
# Else, find node n with smallest id among all the node whose status is R in tree,
# set the status of n to A and return (n.id, n.status).
# Note: root is the pointer to the root of the target AVL tree
def HelpScheduleNext(root):
    if root.R_number == 0:
        return -1
    else if (root.left.R_number != 0):
```

```
10            return HelpScheduleNext(root.left)
11        else if (root.status == R):
12            root.status = A
13            root.R_number -= 1
14            node_pt = root
15            # Update extra info R_number
16            while (node_pt.parent != NIL):
17                node_pt.parent.R_number -= 1
18                node_pt = node_pt.parent
19            return (root.id, root.status)
20        else if (root.status != R):
21            return HelpScheduleNext(root.right)
22
23    def ScheduleNext()
24        return HelpScheduleNext(root);
```

Description:
- If *root.R_number* is 0, which means tree doesn't have any node whose status is R, return -1.

- If $root.left, R\_number$ is 1, then, by binary search tree property, its left child contains a node whose status is R and has smaller id than root.id no matter if root.status is R or not, then we return the result of scheduling root's left child.

- If line 9 didn't pass, we know there is no node with status R in left child or left child is NIL, in this case, if root.status is R: root must have smallest id among all the nodes whose status is R in tree, then we set root.status to A, then update extra info R_number and return (root.id, root.status).

- If line 9 didn't pass and root.status is not R, we know:
1: nodes with status R can never appear in root's left child

2: Tree rooted at root must contain at least one node with status R.

3: Root itself is not in status R.

Then the node having smallest id among all nodes with status R must appear in the right child of root, therefore we return the result of scheduling root's right child.

Time Complexity:
For the worst case, by applying binary search tree property, we need to search along the path from root to the left-most leaf to find the node with smallest id among all the nodes whose status is R, since the longest path in AVL tree is $O(logn)$, then it takes at most $O(logn)$ time.

Updating the ancestor's R_number takes $O(logn)$
Therefore, the worst case time complexity T(n) $\in O(logn) + O(logn) = O(logn)$

Question 2

(Written by Zikun Chen, Read by Yue Li, Yueen Ma)

(a) If the number of empty slots in T is k before inserting x, and there are m slots in total.
Then there are $m - k$ occupied slots in T before inserting x.
Then by SUHA, for any hash function h and an new element x, satisfies $P(h(x) \text{ is occupied}) = \frac{m-k}{m}$

$P(x \text{ is inserted into an empty slot})$

by Power of Two Choices, it happens when at least one of the hash function sends x to an empty slot.

$= P(T[h_1(x)] \text{ is empty or } T[h_2(x)] \text{ is empty})$

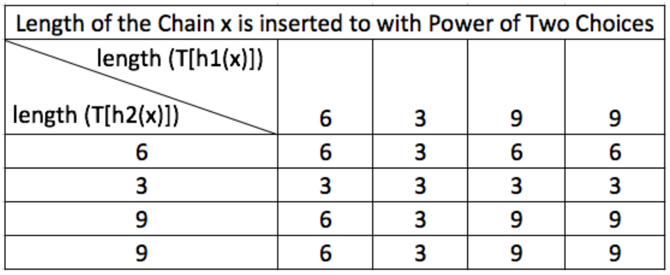
$= 1 - P(T[h_1(x)] \text{ is occupied and } T[h_2(x)] \text{ is occupied})$

$= 1 - P(T[h_1(x)] \text{ is occupied})P(T[h_2(x)] \text{ is occupied})$

since the two hash functions are independent

$$= 1 - (\frac{m-k}{m})^2 = \frac{m^2 - (m^2 - 2mk + k^2)}{m^2} = \frac{2mk - k^2}{m^2}$$

(b)



Length of the Chain x is inserted to with Power of Two Choices

length (T[h1(x)]) / length (T[h2(x)])	6	3	9	9
6	6	3	6	6
3	3	3	3	3
9	6	3	9	9
9	6	3	9	9

Since there are in total 16 possible combinations from the two independent hash functions so by SUHA each cell has probability of $\frac{1}{16}$.

If we count the number of occurrence for each of the 6, 3, 9 in the table, we get:

$P(x \text{ is inserted to } T[0]) = \frac{5}{16}$

$P(x \text{ is inserted to } T[1]) = \frac{7}{16}$

$P(x \text{ is inserted to } T[2] \text{ or } T[3]) = P(x \text{ is inserted to } T[2]) + P(x \text{ is inserted to } T[3]) = \frac{4}{16}$

since the two events are mutually exclusive

As we can see from the table

E(length of the chain x is inserted to)

$= \sum_{i=1}^{4} P(x \text{ is inserted to } T[j]) \times \text{length}(T[j])$

by definition of expectation

$= 6P(x \text{ is inserted to } T[0]) + 3P(x \text{ is inserted to } T[1]) + 9P(x \text{ is inserted to } T[2] \text{ or } T[3])$

$= 6\frac{5}{16} + 3\frac{7}{16} + 9\frac{4}{16} = \frac{87}{16} = 5.4375$

Question 3

(Written by Yueen Ma, Read by)

(1)
- We use a Hash Table with 26 slots, each of which is corresponding to the rank of each letter in the alphabet. i.e. the hash function is :
  $a \to 0, b \to 1, c \to 2, ..., z \to 25$
  Each slot of the Hash Table contains a pointer pointing to the root of AVL tree for storing the words with the corresponding initial letter. Therefore, we have 26 AVL trees in total.
  The keys of the nodes in the AVL tree are the words, and the tree is sorted in alphabetical order. Each node also contains the frequency of its word.

- We assume SUHA

(2)
- PROCESS
  1) Every time a word is to be processed, we hash the word according to the first letter of the word, and assign it a number as mentioned above, which corresponds to a position in the hash table :
  $a \to 0, b \to 1, c \to 2, ..., z \to 25$
  2) Then we access the AVL tree rooted at the position in the hash table indicated by the number assigned above.
  3) Then we search that tree for the word. If the word is added before(i.e., it is already in the tree), then we just increment the frequency by 1 and jump to step 5. If the word is not found, go to step 4.
  4) Then we insert a node containing that word into the tree, using the alphabetical order of the word(e.g. "air" has higher order than "apple" does). The freq of the word is set to 1.
  5) END

- QUERY
  1) We create a list with 26 elements to store the result.

2) For each tree pointed by the slots of the hash table, we do step 3.
3) We traverse through the tree to find the node with highest freq(frequency). If words tie in frequency, we take the word occurring first in alphabetical order. Add the node to the list mentioned in step 1.
4) return the list in step 1.

(3)

```
# Hash Function:
def hash_func(word):
    """
    return the corresponding number of the first letter of the word
    """
    first_letter = word[0]
    letter_ascii = (int)first_letter
    a_ascii = (int)'a'
    return letter_ascii - a_ascii

def Process(word):
    """Process an input word"""
    hash_idx = hash_func(word)
    # We are assuming that the hash table can be accessed anywhere
    # in this program, like a global variable.
    root = HashTable[hash_idx]

    is_found = process_one_tree(root, word)
    if not is_found:
        node = new Node()
        node.word = word
        node.freq = 1
        # This add_to_tree the the same as the Insert() operation
        # of AVL tree
        insert_to_tree(root, node)

def Query():
    """The query operation"""
    most_freq_words = []
    for i from 0 to 25:
        cur_tree = HashTable[i]
        word = query_one_tree(cur_tree)
        most_freq_words.append(word)
    return most_freq_words

# Helper Functions:
def process_one_tree(node, word):
    """
    Search the tree, if the word is added before, then increment
    its freq by 1 and return True, indicating the word is found,
    otherwise, return False
    """
    if node == null:
        return False
    else:
        if word == node.word:
            # word is found
            node.freq += 1
            return True
        else:
            left_result = process_one_tree(node.left, word)
            right_result = process_one_tree(node.right, word)

        # If neither of the return values is True, the return False
        return left_result or right_result

def query_one_tree(node):
    """
    Helper function for finding the node with highest frequency
    and being first in alphabetical order when there is a tie
    """
    if node is null:
        return null
    else:
        most_freq = node
        # left_result has higher alphabetical order
        left_result = query_one_tree(node.left)
        # right_result has lower alphabetical order
        right_result = query_one_tree(node.right)

        if left_result.freq >= most_freq.freq:
            # left_result has higher freq than node,
            # or it has higher alphabetical order
            # but the same freq as node does
            most_freq = left_result

        if right_result.freq > most_freq.freq:
            # right_result has higher freq than
            # both node and left_result
            most_freq = right_result

        return most_freq
```

(4)
- PROCESS
  1) Accessing the corresponding tree via the hash table takes constant time, $\Theta(1)$
  2) When searching the tree, the nodes will be visited at most once. And based on SUHA, the number of nodes in the tree(=the number of words, because we don't have duplicate word in trees) has expected value of $\frac{l}{26}$. Therefore, when every node is accessed, it takes time

$\Theta(\frac{l}{26})$. Because l is a constant, then the expected time it actually takes is $\Theta(1)$.
3) In the worst case, we need to add the node (i.e., inserting the node, this is the AVL tree's Insert operation) takes expected time $O(\frac{l}{26})$, Because l is a constant, then it is $O(1)$. Because no algorithm can take less than constant time, then adding the node takes $\Theta(1)$.
4) In conclusion, the expected time PROCESS takes is $\Theta(1)+\Theta(1)+\Theta(1) = \Theta(1)$

- QUERY Because there are in total l distinct words, meaning that there are in total l nodes in the 26 AVL Trees in our hash table. And every node in the 26 trees are all visited exactly once, and we don't have duplicate words in these trees. So the number of visits is at most the size of the vocabulary, which is l. And l is a constant so the time complexity is $\Theta(1)$