# Solutions for Homework Assignment #3

## Answer to Question 1.

**a.** The scheduler $\mathcal{S}$ can be implemented by an augmented AVL tree $D$. Each node $u$ of $D$ contains information about a thread $t$ of $\mathcal{S}$. A node $u$ contains the following fields:

- $id(u)$: the integer id of the thread.
- $status(u)$: the status of the thread, namely $A$, $R$ or $S$.
- $ready(u)$: a boolean variable that is TRUE if and only if the subtree rooted at $u$ (including $u$) has a node $v$ such that $status(v) = R$. Note that the *ready* field of each node $u$ of $D$ satisfies the identity:

$$ready(u) = ready(lchild(u)) \vee (status(u) = R) \vee ready(rchild(u)) \qquad (*)$$

  where $ready(\text{NIL}) = \text{FALSE}$.[1]

- $lchild(u)$, $rchild(u)$ and $parent(u)$: pointers to the left child, right child, and the parent of $u$.
- $BF(u)$: the balance factor of $u$.

The *id* field is the *key* of the node in the AVL tree $D$ — thus, an in-order traversal of $D$ visits all the threads *in non-decreasing id order*.

**b.** The scheduler's operations are implemented as follows:

- NEWTHREAD($t$): *Given a thread $t = (i, status)$, where $status \in \{A, R, S\}$, inserts a node representing $t$ into $D$; assumes that $i$ is different from the id of every other thread in $D$.*

  (1) Prepare a new node $u$ with $id(u) = i$ and $status(u) = A$ to represent thread $t$, and use the ordinary BST insertion algorithm to insert node $u$ into $D$ with $i$ as the insertion key. After this insertion, the node $u$ that represents $t$ is a leaf of $D$.

  (2) If the *status* of $t$ is $R$ then traverse the path from the new leaf $u$ to the root of the tree $D$ to update the *ready* fields: for each node $v$ along this path set:

$$ready(v) := ready(lchild(v)) \vee (status(v) = R) \vee ready(rchild(v))$$

  (3) Traverse the path from the new leaf $u$ to the root of the tree $D$ again, performing rotations and updating the balance factors as required by the ordinary AVL insertion algorithm. In addition, update the *ready* field of every node $v$ involved in a rotation, according to the identity (*). Stop the rotations and updates to the balance factors as in the ordinary AVL insertion algorithm.
  *Remark:* The two traversals described above can be done in single pass where we first update the *ready* field of the node under consideration and then we do the balancing/rotations part (which may also require a few additional *ready* field updates).

  Suppose that $D$ contains $n$ threads. Since $D$ is an AVL tree, the height of any subtree of $D$ is $O(\log n)$. The time required by NEWTHREAD($t$) is that required by the ordinary AVL insertion algorithm, i.e., $O(\log n)$, plus the time required to update the *ready* fields of the new node's ancestors (if the new node's *status* is $R$), and the *ready* fields of the nodes involved in a rotation (a few nodes for each rotation). Since there are $O(\log n)$ such nodes and each *ready* update takes $O(1)$ time, the additional time required to update all the *ready* fields is also $O(\log n)$. Thus, the worst-case time complexity of NEWTHREAD($t$) is $O(\log n)$.

[1] NIL denotes an empty pointer.

- FIND$(i,x)$: *Given a thread id $i$ and a pointer to a node $x$ of $D$, returns a pointer to the thread $t=(i,-)$ in the subtree of $D$ rooted at $x$; if no such thread exists, then it returns $-1$.*

  This is a straightforward BST search algorithm:

  ```
  Find(i, x)
    if x = NIL then
        return −1
    else if id(x) < i then
        return Find(i, lchild(x))
    else if id(x) = i then
        return x
    else
        return Find(i, rchild(x))
    end if
  ```

  Note that FIND$(i)$ is simply FIND$(i,r)$ where $r$ points to the root of the tree $D$.[2]

  The worst-case time complexity of FIND$(i,x)$ is $O(\log n)$. To see why, note that: (i) each call at a node $u$ results in *at most one* recursive call, at the left or the right subtree of $u$; and (ii) each call involves a constant amount of work. So the time to execute FIND$(i,x)$ is proportional to the height the subtree of $D$ rooted at $x$, i.e., it is $O(\log n)$.

- COMPLETED$(i)$: *If $\mathcal{S}$ has a thread $t=(i,-)$ then remove $t$ from $\mathcal{S}$, else return $-1$.*

  First call *Find$(i)$* to determine whether $\mathcal{S}$ contains a thread $t=(i,-)$. If it does not, i.e., *Find$(i)$* returns $-1$, then return $-1$. If it does, i.e., *Find$(i)$* returns a pointer $x$ to the node of $D$ that contains $t=(i,-)$, proceed as follows:

  (1) Use the ordinary BST deletion algorithm to delete the node pointed to by $x$ from $D$. Since $D$ is an AVL tree, this deletion ultimately results in the removal of a leaf from the tree. *Let $z$ be the parent of that leaf.* The rest of the algorithm is very similar to parts (2) and (3) of NEWTHREAD$(t)$, except that the traversals described below start from node $z$ rather than the newly added leaf in NEWTHREAD$(t)$.

  (2) Traverse the path from $z$ to the root of the tree $D$ to update the *ready* fields: for each node $v$ along this path, set $ready(v) := ready(lchild(v)) \vee (status(v)=R) \vee ready(rchild(v))$.

  (3) Traverse the path from $z$ to the root of the tree $D$ again, performing rotations and updating the balance factors as required by the ordinary AVL deletion algorithm. In addition, update the *ready* field of every node $v$ involved in a rotation, according to the identity (*).

  Stop the rotations and updates to the balance factors as in the ordinary AVL deletion algorithm.

  *Remark:* The two traversals described above can be done in single pass where we first update the *ready* field of the node under consideration and then we do the balancing/rotations part (which may also require a few additional *ready* field updates).

  The time required by COMPLETED$(i)$ is that required by the ordinary AVL deletion algorithm, i.e., $O(\log n)$, plus the time required to update the *ready* fields of $z$'s ancestors, and the *ready* fields of the nodes involved in a rotation (a few nodes for each rotation). Since there are $O(\log n)$ such nodes and each *ready* update takes $O(1)$ time, the additional time required to update the *ready* fields is also $O(\log n)$. Thus, the worst-case time complexity of COMPLETED$(i)$ is $O(\log n)$.

[2] More precisely, if the FIND$(i)$ search is successful, it returns a *pointer* to a node $u$ in $D$ that represents some thread $t=(i, status)$; given this pointer, it is trivial to return the pair $t=(i, status)$ contained in $u$.

- $\textsc{ChangeStat}(x, stat)$: *Given a pointer $x$ to a node containing thread $t$ in $D$, sets the* status *of $t$ to stat.*

  To do so, we first set *status*$(x)$ to *stat*. If *stat* is not $R$, then we are done and just return. Otherwise, we update the *ready* field of every node in the path from $x$ to the root of the tree $D$, according to the identity (*). Note that if the status of any node along this path does not change, this update process can stop (this obvious optimization is not shown in the basic pseudocode below).

```
ChangeStat(x, stat)
    status(x) := stat
    if stat ≠ R return
    y := x
    repeat
        ready(y) = ready(lchild(y)) ∨ (status(y) = R) ∨ ready(rchild(y))
        y := parent(y)
    until y = NIL
    return
```

  It is clear that the time to execute $\textsc{ChangeStat}(x, stat)$ is is proportional to length of the path from $x$ to the root of $D$. So the worst-case time complexity of $\textsc{ChangeStat}(x, stat)$ is $O(\log n)$.

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

```
ChangeStatus(i, stat)
    x := Find(i)
    if x = −1 then return −1
    else return ChangeStat(x, stat)
```

  Since the worst-case time complexity of $\textsc{Find}(i)$ and $\textsc{ChangeStat}(x, stat)$ is $O(\log n)$, the worst-case time complexity of $\textsc{ChangeStatus}(i, stat)$ is also $O(\log n)$.

  pointer is fine

- $\textsc{ScheduleNext}(x)$: *Given a pointer to a node $x$ of $D$, returns a pointer to the thread $t$ that has the smallest id among all the threads with status $= R$ in the subtree of $D$ rooted at $x$, and sets the status of $t$ to $A$; if the subtree rooted at $x$ has no thread with status $= R$, then the procedure returns $-1$.*

```
ScheduleNext(x)
    if x = NIL ∨ ready(x) = False then
        return −1
    else if ready(lchild(x)) = True then
        return ScheduleNext(lchild(x))
    else if status(x) = R then
        ChangeStat(x, A)
        return x
    else
        return ScheduleNext(rchild(x))
    end if
```

  Note that $\textsc{ScheduleNext}$ is simply $\textsc{ScheduleNext}(r)$ where $r$ points to the root of the tree $D$.

  The worst-case time complexity of $\textsc{ScheduleNext}(x)$ is $O(\log n)$. To see why, first note that $\textsc{ScheduleNext}(x)$ calls $\textsc{ChangeStat}(x, A)$ *at most once*, and, as we have argued earlier, the time complexity of $\textsc{ChangeStat}(x, A)$ is $O(\log n)$. Now consider the time required to execute $\textsc{ScheduleNext}(x)$ *excluding the time to execute* $\textsc{ChangeStat}(x, A)$. Note that: (i) each call at a node $u$ results in *at most one* recursive call, at the left or the right subtree of $u$; and (ii) each call involves a constant amount of work other than the recursive call that it makes and the call to $\textsc{ChangeStat}(x, A)$. So the time to execute $\textsc{ScheduleNext}(x)$ *excluding the time taken by* $\textsc{ChangeStat}(x, A)$ is proportional to the height the subtree of $D$ rooted at $x$, i.e., it is $O(\log n)$. Since the time complexity of $\textsc{ChangeStat}(x, A)$ is $O(\log n)$, the *total* time to execute $\textsc{ScheduleNext}(x)$ is also $O(\log n)$ in the worst-case.

**Answer to Question 2.**

**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?

Let $E = \{i_1, i_2, \ldots, i_k\}$ be the set of *empty* slots of $T$ before inserting $x$.

The probability that $h_1$ hashes $x$ into an *empty* slot of $T$ is:

$$\begin{aligned} Prob[h_1(x) \in E] &= Prob[h_1(x) = i_1 \vee h_1(x) = i_2 \vee \ldots \vee h_1(x) = i_k] \\ &= \sum_{j=1}^{j=k} Prob[h_1(x) = i_j] \end{aligned}$$

Since $T$ has $m$ slots and we assume that the hash function $h_1$ satisfies the *Simple Uniform Hashing Assumption* (SUHA), we have $Prob[h_1(x) = i_j] = \frac{1}{m}$ for every $j$, $1 \leq j \leq k$. Therefore:

$$Prob[h_1(x) \in E] = \sum_{j=1}^{j=k} \frac{1}{m} = \frac{k}{m}$$

So the probability that $h_1$ hashes $x$ into a *non-empty* slot of $T$ is:

$$Prob[h_1(x) \notin E] = 1 - \frac{k}{m}$$

Since $h_2$ also satisfies SUHA, the probability that $h_2$ hashes $x$ into a *non-empty* slot of $T$ is also:

$$Prob[h_2(x) \notin E] = 1 - \frac{k}{m}$$

Note that $x$ is inserted in a *non-empty* slot of $T$ if and only if *both* $h_1$ and $h_2$ hash into a *non-empty* slot of $T$. Therefore:

$$Prob[x \text{ is inserted in a } \textit{non-empty} \text{ slot of } T] = Prob[h_1(x) \notin E \wedge h_2(x) \notin E]$$

Thus, since the hash functions $h_1$ and $h_2$ are independent:

$$\begin{aligned} Prob[x \text{ is inserted in a } \textit{non-empty} \text{ slot of } T] &= Prob[h_1(x) \notin E] \cdot Prob[h_2(x) \notin E] \\ &= (1 - \frac{k}{m})^2 \end{aligned}$$

Therefore:

$$\begin{aligned} Prob[x \text{ is inserted in an } \textit{empty} \text{ slot of } T] &= 1 - Prob[x \text{ is inserted in a } \textit{non-empty} \text{ slot of } T] \\ &= 1 - (1 - \frac{k}{m})^2 \\ &= \frac{k(2m-k)}{m^2} \end{aligned}$$

**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?

Since the table has 4 slots, when we insert $x$ using the two independent hash functions $h_1$ and $h_2$ there are 16 possible hashing outcomes:

$$S = \{(i, j) \mid \text{where } 0 \leq i = h_1(x) \leq 3 \text{ and } 0 \leq j = h_2(x) \leq 3\} \qquad (S \text{ is our sample space})$$

Note that:

$$
\begin{aligned}
Prob[(i,j)] &= Prob[h_1(x) = i \wedge h_2(x) = j] && \text{(by definition)} \\
&= Prob[h_1(x) = i] \cdot Prob[h_2(x) = j] && \text{(because } h_1 \text{ and } h_2 \text{ are independent)} \\
&= \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16} && \text{(because } h_1 \text{ and } h_2 \text{ satisfy SUHA)}
\end{aligned}
$$

Let $X$ be the random variable denoting the length of the chain where $x$ is inserted. That is, for each possible outcome $o = (i,j) \in S$, $X(o)$ is the length of the chain where $x$ is inserted. We are seeking $E[X]$. Note that:

another way:
straight calculate expected value of length.
by summing up length of slot of all 16 outcomes
then divide by 16

$$
\begin{aligned}
E[X] &= \sum_{o \in S} X(o) \cdot Prob[o] \\
&= \sum_{(i,j) \in S} X[(i,j)] \cdot Prob[(i,j)] \\
&= \frac{1}{16} \sum_{(i,j) \in S} X[(i,j)] \\
&= \frac{1}{16} \big( X[(0,0)] + X[(0,1)] + X[(0,2)] + X[(0,3)] \\
&+ X[(1,0)] + X[(1,1)] + X[(1,2)] + X[(1,3)] \\
&+ X[(2,0)] + X[(2,1)] + X[(2,2)] + X[(2,3)] \\
&+ X[(3,0)] + X[(3,1)] + X[(3,2)] + X[(3,3)] \big)
\end{aligned}
$$

Recall that $T[0]$ contains 6 elements, $T[1]$ contains 3 elements, and $T[2]$, $T[3]$ contain 9 elements each. From our hashing scheme, when the hashing outcome is $(i,j)$, the length $X[(i,j)]$ of the chain that $x$ enters is as follows: if $i = 1$ or $j = 1$ then $X[(i,j)] = 3$, else if $i = 0$ or $j = 0$ then $X[(i,j)] = 6$ else $X[(i,j)] = 9$. Therefore:

$$
\begin{aligned}
E[X] &= \frac{1}{16} \big( 6 + 3 + 6 + 6 \\
&+ 3 + 3 + 3 + 3 \\
&+ 6 + 3 + 9 + 9 \\
&+ 6 + 3 + 9 + 9 \big) \\
&= \frac{87}{16} \\
&= 5.4375
\end{aligned}
$$

Another way to compute $E[X]$ is as follows. Note that the chain where $x$ is inserted has length 3, 6 or 9. So the only possible value $v$ of the random variable $X$ is 3, 6 and 9. An alternative formula for $E[X]$ is:

$$
E[X] = \sum_{v \in \{3,6,9\}} v \cdot Prob[X = v]
$$

It now suffices to compute $Prob[X = 3]$, $Prob[X = 6]$, and $Prob[X = 9]$.
Note that $X = 3$ iff the outcome of the hashing is an $(i,j)$ with $i = 1$ or $j = 1$. Therefore:

$$
\begin{aligned}
Prob[X = 3] &= Prob[\{(0,1), (1,0), (1,1), (1,2), (1,3), (2,1), (3,1)\}] \\
&= Prob[(0,1)] + Prob[(1,0)] + Prob[(1,1)] + Prob[(1,2)] + Prob[(1,3)] + Prob[(2,1)] + Prob[(3,1)]\}] \\
&= 7 \cdot \frac{1}{16} = \frac{7}{16}
\end{aligned}
$$

Similarly:

$$
\begin{aligned}
Prob[X=6] &= Prob[\{(0,0),(0,2),(0,3),(2,0),(3,0)\}] \\
&= Prob[(0,0)] + Prob[(0,2)] + Prob[(0,3)] + Prob[(2,0)] + Prob[(3,0)]\}] \\
&= 5 \cdot \frac{1}{16} = \frac{5}{16} \\
Prob[X=9] &= Prob[\{(2,2),(2,3),(3,2),(3,3)\}] \\
&= Prob[(2,2)] + Prob[(2,3)] + Prob[(3,2)] + Prob[(3,3)]\}] \\
&= 4 \cdot \frac{1}{16} = \frac{4}{16}
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
E[X] &= \sum_{v \in \{3,6,9\}} v \cdot Prob[X=v] \\
&= 3 \cdot \frac{7}{16} + 6 \cdot \frac{5}{16} + 9 \cdot \frac{4}{16} \\
&= \frac{87}{16}
\end{aligned}
$$

**Answer to Question 3.**

direct addressing table is better than array can just search with A[index]

(1) We will use two data structures for our algorithm: a direct access table $A$ with 26 indices (corresponding to each letter in the English alphabet) and a hash table $T$.

We assume that the size $t$ of the hash table $T$ is within a constant factor of $l$; more precisely, we assume that $t = \kappa \cdot l$, for some constant $\kappa > 0$. We will also assume our hashing function satisfies the Simple Uniform Hashing Assumption (SUHA).

(2) The idea for our algorithm is as follows. Initially, we set every value in our direct access table $A$ to `null`; $A$ will store the current most frequent word starting with each letter. When we receive an input *word*, we insert $(word, 1)$ into our hash table $T$ using *word* as our key if it does not already exist in $H$; otherwise (if it does), we increment its frequency:

$$(word, freq) \to (word, freq + 1)$$

Then, let the first letter of *word* be some letter corresponding to value $x$ (e.g. "b" corresponds to 2); if *word*'s frequency is greater than the frequency of the word at $A[x]$ (or if $A[x] ==$ `null`), then set $A[x]$ to *word*.

To perform a *query* operation, we simply read our direct access table from indices 1 to 26.

(3)
```
CHAINED-HASH-SEARCH(T, k):
  return pointer to node with key k in list T[h(k)], null if doesn't exist

CHAINED-HASH-INSERT(T, x):
  insert x at the head of list T[h(x.key)]

INSERT(T, A, word):
  // update hash table storing word frequencies
  u := CHAINED-HASH-SEARCH(T, word)
  if u = null:
    create a new node u s.t. u.key = word and u.value = 1
    CHAINED-HASH-INSERT(T, u)
  else:
```

```
    increment u.value by 1
  // check if there is a new most frequent word
  l = word[0] // first letter of word
  x = l - 'a' // convert to number
  current_max = CHAINED-HASH-SEARCH(T, A[x]).value
  if u.value > current_max or (u.value = current_max and word < A[x]):
    A[x] := word

QUERY(A):
  for i in 1 ... 26:
   print T[i]
```

Note: it is also possible to store $(word, freq)$ in the direct access table so that you don't have to look up the frequency of the current max. This slightly increases efficiency in exchange for a slight increase in storage.

(4) Under the simple uniform hashing assumption (SUHA), i.e., that each distinct value is equally likely to be hashed into any one of T's slots, the expected length of a linked list in $T$ is $\alpha$, where $\alpha = l/t$. Since $t$ is $t = \kappa \cdot l$, $\alpha$ is $\Theta(1)$, and so the expected length of a linked list in $T$ is $\Theta(1)$. Therefore the expected time for an `INSERT` operation is $\Theta(1)$ — a linked list in $T$ is accessed twice; all other operations are clearly constant (in terms of the current length of the sequence). The `QUERY` operation runs in constant time since accessing every element in $A$ is a constant time operation and $|A| = 26$; that is, $|A|$ is not dependant on the length of the sequence.