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:

1. 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).

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

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

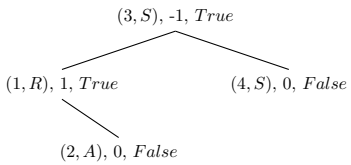
4. ChangeStatus($i, stat$): If S has a thread $t = (i, -)$ then set the $status$ of t to $stat$, else return -1.

5. 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 $O(\log n)$ 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.

1. 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. The data structure is an AVL tree T of size n, which means that the usual parameters for an arbitrary node x in AVL tree persists, specifically, left and right child and parent $x.left$, $x.right$, $x.parent$, balancing factor $x.bf$, and key $x.key$. We store the thread to every node as auxillary information, i.e. $x.t$. The key at each node $x.key$ is the unique id from the thread, i.e. $x.key = x.t.id$. We add an additional member $x.ready$, which holds a boolean value of whether the subtree rooted at x contains any node y where $y.t.status == R$.
   As an example, suppose that threads $t_1 = (1, R)$, $t_2 = (2, A)$, $t_3 = (3, S)$, and $t_4 = (4, S)$ are in T, then we have the following tree. Note that in each node, we display $thread$, $balancingfactor$, and $ready$ field, in this order.



□

2. Describe the algorithm that implements each one of the five operations above, and explain why each one takes $O(\log n)$ 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.

   (a) NewThread(t): The algorithm to insert a new thread $t = (id, status)$ to S is pretty much the same as AVL-Insert($T, t.id$). When we find the correct position by comparing keys, we add the new node as described previously with t as its auxillary information and $x.key$ pointing to $x.t.id$. During the bubble up traversal from the newly added node to the root, in addition to possible rotation and balancing factor update, we also update the $x.ready$ field. Specifically, if $x.t.status == R$ then we assign $v.ready$ to $True$ for all ancestors of x including x iteself; Otherwise we do not update $v.ready$ for ancestors of x and we set $x.ready$ to $False$. The algorithm just added a constant time during the traversal step where we update the $v.ready$. The worst case time complexity is therefore same as that of AVL-Insert, i.e. $O(\log n)$.

   (b) Find(i): The algorithm is identical to AVL-Search(T, i) where we search for the corresponding thread's id over the tree using binary search algorithm. Instead of returning pointer to the matching node we return $x.t$ instead. If no such node exists, then we return -1. The algorithm has the same worst case time complexity as AVL-Search of $O(\log n)$

   (c) Completed(i): We call AVL-Search(T, i) and evaluate the return value. If there is no thread with id of i we return -1; If the return value is a pointer p to a node whose thread has id of i, then we call AVL-Delete(T, p). When we are traversing up from the deleted node to the root, in addition to possible tree rotation and balancing factor update, we also update $v.ready$ field for all ancestor v of x, the deleted leaf node. During the traversal, we assign $v.ready$ to $True$ if any of $v.left.ready$, $v.right.ready$, and $v.ready$ is $True$. Otherwise we assign $v.ready$ to $False$. Find(i) has a worst case time complexity of $O(\log n)$. The algorithm when removing the thread from the tree adds a constant time operation during the tree traversal step and therefore has the same worst time complexity as AVL-Delete. Therefore, The algorithm has a worst time complexity of $O(\log n) + O(\log n) = O(\log n)$

(d) ChangeStatus$(i, stat)$: We call AVL-Search(T, i) and evaluate the return value. If there is no thread with id of i we return -1; If the return value is a pointer p to a node x whose thread has id of i, we modify the node by assigning $stat$ to $x.t.status$. Now we set $x.ready$ to $True$ if $stat$ is R; otherwise set ift to $False$. Now we traverse up ancestors v of x. During the traversal, we assign $v.ready$ to $True$ if any of $v.left.ready$, $v.right.ready$, and $v.ready$ is $True$. Otherwise we assign $v.ready$ to $False$.

Algorithm 1:

```
1  Function ChangeStatus (i, stat)
2      x = AVL-Search (T, i)        #T is the AVL tree
3      if x == NIL then
4          return −1
5      else
6          x.t.status ← stat
7          if stat is R then
8              x.ready = True
9          else
10             x.ready = False
11         for v ← x to T.root do
12             if Any of x.left.ready, x.right.ready, x.ready is True then
13                 x.ready = True
14             else
15                 x.ready = False
```

The worst case time complexity consists of the AVL-Search(S, i) step which takes $O(\log n)$ as well as the upward traversal step for updating $ready$ data member. Since a single node at each level is processed exactly once in the for loop at line 11, each taking a constant time. The worst case time complexity of the update step is therefore $O(h)$ where h is the height of the tree. Since AVL tree is height balanced, the time complexity for the update is therefore $O(\log n)$. Altogether, the worst case time complexity for ChangeStatus is hence $O(\log n)$

(e) ScheduleNext: We traverse the AVL tree T downward. At each level, we decide the potential position of node n whose thread has the smallest id and a status of R. First note that inorder traversal gives a list of nodes whose thread has ascending id. So if $x.left.ready$ is $True$, it means that there is a node with a smaller thread id and status of R, so we move down next level by advancing to $x.left$. Otherwise, if in addition x has a thread status of R, then x itself is holds the smallest thread id so we return $x.t$. If $x.t.status$ is not R and $x.left.ready$ is not $True$, it means that the thread with the status R is in the right subtree, so we move down next level by advancing to $x.right$. This loops over until either the correct node is found or reached a leaf node. In the former case, we simply call ChangeStatus$(x.i, A)$ to change the status of the thread to A. In the latter

case, we simply return -1.

Algorithm 2:

```
1  Function ScheduleNext
2      x ← T.root
3      while x is not NIL do
4          if x.left.ready == True then
               /* t with smallest id and R status in left subtree          */
5              x ← x.left
6          else if x.t.status == R then
               /* x itself has the smallest thread id with R status        */
7              ChangeStatus (x.t.id, A)
8              return x.t
9          else
               /* Otherwise, t with status R is in right subtree           */
10             x ← x.right
11     return -1
```

Since a single node at each level is processed exactly once in the while loop at line 3, each taking a constant time. The worst case time complexity of the algorithm is therefore $O(h)$ where h is the height of the tree. Since AVL tree is height balanced, the time complexity is therefore $O(\log n)$. If a node is found, ChangeStatus takes $O(\log n)$. So the time complexity of the alogrithm is still $O(\log n)$.

Question 2

The Power of Two Choices 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 h_1 and h_2. 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 $T[h_1(x)]$ or $T[h_2(x)]$; we pick the slot in which the chain is shorter. If the chains are same length, we add x to the chain in $T[h_1(x)]$.

1. 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. Let S be a set of slot that is empty. By the simple uniform hashing assumption we have that
   $$\mathbb{P}(h(x) \in S) = k\mathbb{P}(h(x)) = \frac{k}{m}$$

The probability that x is inserted in to an empty slot is given by

$$\begin{aligned}
\mathbb{P}(h_1(x) \in S \cup h_2(x) \in S) &= \mathbb{P}(h_1(x) \in S) + \mathbb{P}(h_2(x) \in S) - \mathbb{P}(h_1(x) \in S \cap h_2(x) \in S) \\
&= \mathbb{P}(h_1(x) \in S) + \mathbb{P}(h_2(x) \in S) - \mathbb{P}(h_1(x) \in S)\mathbb{P}(h_2(x) \in S) \\
&\qquad \text{(independence of } h_1 \text{ and } h_2\text{)} \\
&= \frac{k}{m} + \frac{k}{m} - (\frac{k}{m})^2 \\
&= \frac{2km - k^2}{m^2}
\end{aligned}$$

□

2. 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. By simple uniform hashing assumption, for any $h \in \{h_1, h_2\}$ given key x, we have

$$\mathbb{P}(h(x) = i) = \frac{1}{4}$$

where $i \in \{1, 2, 3, 4\}$ represents the index of hash table slots, i.e. $T[i]$. Let L be a random variable denoting the length of chain x is inserted into. First we find probability distribution that x is inserted into any of the 4 slots. As an example, by the rule specified, x is inserted into the the $T[0]$ if whenever one of the hash function yields a hash value of 0 with the other hash function yielding a hash value of any of $\{0, 2, 3\}$, and vice versa. This is because x will always be inserted to the slot with the lesser number of elements. We have the probability that x will be inserted to slot indexed by 0,

$$\begin{aligned}
&\mathbb{P}\big((h_1(x) = 0 \wedge h_2(x) \in \{0, 2, 3\}) \vee (h_1(x) = \{0, 2, 3\} \wedge h_2(x) = 0)\big) \\
&\quad = \mathbb{P}(h_1(x) = 0 \wedge h_2(x) \in \{0, 2, 3\}) + \mathbb{P}(h_1(x) = \{0, 2, 3\} \wedge h_2(x) = 0) - \mathbb{P}(h_1(x) = 0 \wedge h_2(x) = 0) \\
&\quad = \mathbb{P}(h_1(x) = 0)\mathbb{P}(h_2(x) \in \{0, 2, 3\}) + \mathbb{P}(h_1(x) = \{0, 2, 3\})\mathbb{P}(h_2(x) = 0) - \mathbb{P}(h_1(x) = 0)\mathbb{P}(h_2(x) = 0) \\
&\qquad\qquad \text{(by independence of } h_1 \text{ and } h_2\text{)} \\
&\quad = \frac{1}{4} \times \frac{3}{4} + \frac{1}{4} \times \frac{3}{4} - \frac{1}{4} \times \frac{1}{4} \\
&\quad = \frac{5}{16}
\end{aligned}$$

Using a similar approach, we compute the distribution of L to yield the probability

mass function,

$$f_L(i) = \begin{cases} \frac{5}{16} & x \text{ inserted to index } 0 \\ \frac{7}{16} & x \text{ inserted to index } 1 \\ \frac{2}{16} & x \text{ inserted to index } 2 \\ \frac{2}{16} & x \text{ inserted to index } 3 \end{cases}$$

To compute the expected value of L, we define l_i be the length of the chain at slot $T[i]$, so then

$$\begin{aligned} \mathbb{E}[L] &= \sum_{i=0}^{3} l_i f_L(i) \\ &= 6 \times \frac{5}{16} + 3 \times \frac{7}{16} + 9 \times \frac{2}{16} + 9 \times \frac{2}{16} \\ &= \frac{87}{16} \end{aligned}$$

□

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:

1. $O(1)$ expected time to process each input, under some reasonable assumptions that you should state
2. $O(1)$ 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.

   Solution. We will be using a chained (w/e linked list) hash table data structure T. we will store the tuple $t = (word, count)$ in the hash table, where $word$ is the input word and $count$ is the frequency of the word encountered so far. We will use the dot notation to access the members of the tuple, i.e. $t.word$. We are using the simple uniform hashing assumption. Given a proper hash function h, for any two keys $t_1.word$ and $t_2.word$ we have

   $$\mathbb{P}(h(t_1.word) == h(t_2.word)) = \frac{1}{m}$$

   where m is the size of the hash table. We also assume that the size of the hash table is proportional to l, i.e. $m = O(l)$ and that hash computation is $O(1)$. We will also require an array A of size 26, each defaults to $NULL$. This array holds pointers to tuple with the most frequent words starting with 'a', 'b', and so on in lexicographical order. Since there exists a direct mapping between index of array and the alphabet, it is trivially simple to get the tuple pointed to given the first letter, hence we define A, GetTuple($char$) to access the tuple containing the most frequent word given a character ranging from 'a' to 'z'. It is equally trivial to assign pointers to a tuple to the index for a letter, which we denote by AssignTuple($A, char, t$) which assigns a pointer to tuple t at the index corresponding to letter $char$. We also assume that given two tuple, $t1$ and $t2$ we will be able to get the tuple whose word comes first in alphabetical order by calling the function CompareOrdering($t1, t2$) in $O(1)$. □

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

   (a) ProcessWord($T, A, word$). We first create a tuple $t = (word, count)$ and initialize $count$ to 1. Then we call HashTable-Search($T, word$) to find, if exists, the tuple u stored such that $u.word$ is the same as $word$. If there is no such tuple, we call HashTable-Insert($T, word$), where we store the tuple t to the linked list at the corresponding slot. We then assign pointers to the newly added tuple to array A at the index corresponding to character $word[0]$, which is the first character of $word$. We do this by calling AssignTuple($A, word[0], t$) described previously. If tuple u is returned, we increment $u.count$ by one and then call GetTuple($A, t.word[0]$) to find the previous tuple p, if any, with the largest $count$ given the first character of $word$ in array A. If $u.count$ is larger than $p.count$, we call AssignTuple($A, word[0], u$) to update the pointers such that it still points to tuple with the largest word count. If $u.count$ is equal to $p.count$, we resolve the conflict by comparing their words' alphabetical order and assign the tuple whose word comes first in alphabetical order to the array A at the corresponding index, AssignTuple($A, word[0]$, CompareOrdering(u, p)).

(b) QUERY(A) The algorithm loops over the array A and prints $t.word$, if exists, for every tuple t pointed to in the order of array index, which we defined to be in alphabetical order. If the word begining with the letter has not been inputted, we simply return $NULL$.

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

Algorithm 3:

```
1  Function ProcessWord (T, A, word)
2      firstLetter ← word[0]
3      t.word ← word
4      t.count ← 1
5      u ← HashTable-Search(T, word)
6      if u is NULL then
7          HashTable-Insert (T, word) /* Note tuple t is inserted here      */
8          AssignTuple (A, firstLetter, t)
9      else
10         u.count + +
11         p = GetTuple(A, firstLetter)
12         if u.count > p.count then
13             AssignTuple(A, firstLetter, u)
14         else if u.count == p.count then
15             comeFirst ← CompareOrdering(u, p)
16             AssignTuple(A, firstLetter, comeFirst)
```

Algorithm 4:

```
1  Function Query (A)
2      wordList[A.length]
3      for i = A[0] to A[A.length − 1] do
4          wordList[i] ← A[i].word
5      return wordList
```

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

(a) PROCESSWORD$(T, A, word)$ In the algorithm, there is no loop or recursion. Note that procedures like increment, assigning pointers with ASSIGNTUPLE$(A, char, t)$, returning pointers with GETTUPLE$(A, char)$, comparing words based on alphabetical order with COMPAREORDERING(t_1, t_2), and inserting a tuple to a chained hash table HASHTABLE-INSERT all takes $O(1)$ time. We also know that HASHTABLE-SEARCH$(T, word)$ takes expected $O(1)$ time proved in the textbook

with the given assumptions in part 1. So altogether this algorithm takes $O(1)$ expected running time.

(b) QUERY(A) The loop is executed at most constant number, 26, times, with each iteration doing a constant time operation. Hence the algorithm has a $O(1)$ worst case running time.