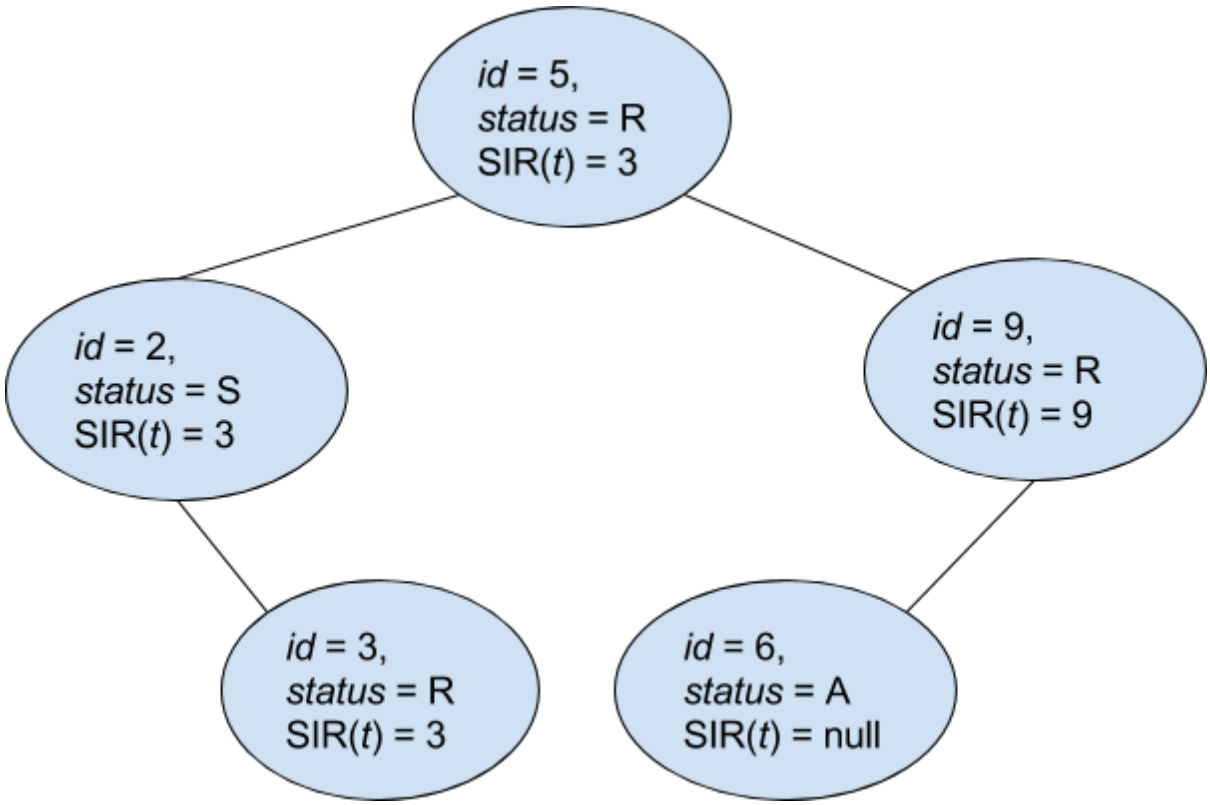
Question 1(Zhenjie Xiong):

//Verified by Xiaoxue Xing

a)

It would be an iterative process. The *id* is the key of each node. The info stored at each node is the *status*. The function which we need to compute is the smallest *id* for all the threads with the *status* R. We store the computing result at each node *t*, caling it SIR(*t*), which is the smallest *id* for all threads in the sub-tree rooted at *t*.
An example of this data structure would look like this:



b)

NEWTHREAD(*t*):
We do a regular AVL-insert(***S***, <*t.id*, *t.status*, SIR=null>). However, as you traverse up the tree and adjust balance factors, we also adjust SIR(*t*) for each node on the path to the root. After every rotation, we adjust the SIR(*t*) for every node involved in a rotation by using this formula:
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.

Why O(log n):
Because rotation as well as updating SIR(*t*) traverse at most the AVL tree height, which is O(log n).

FIND(*i*):
We would use the down-phase searching approach in our augmented AVL tree. We only care about the key, which is *id*, in FIND. Starting from the root, if *i*(the key we are looking for) is less than the key in the current node, we move to the left child of that node. Otherwise, we move to the right child.
As soon as we find the *i* in S, we return the thread *t* that has the key value of *i.* If not, keep moving down until the current node is null(i.e. the previous node is a leaf). Then we return -1 because *i* cannot be found in S.

Why O(log n):
Because recursive call on a root's child until it finally finds the thread takes at most O(tree height) which is O(log n).

COMPLETED(*i*):
This approach would be down-phase only if the thread with key *i* is not found in S, in which case would return -1. But if it is indeed found, we would also do an up-phase as well, since we then do a regular AVL-delete of that thread.
However, as you traverse up the tree and adjust balance factors, we also adjust SIR(*t*) for each node on the path to the root. After every rotation, we adjust the SIR(*t*) for every node involved in a rotation by using this formula:
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.

Why O(log n):
Because recursive call on a root's child, as well as rotation and updating SIR(*t*) if it's found traverse at most 2 times the AVL tree height, which is O(log n).

CHANGESTATUS(*i, stat*):
Similar to COMPLETED(*i*), if a down-phase search cannot find the thread with key *i,* it returns -1. However, if the thread is found, instead of removing it, we replace the *t.status* with *stat*. We don't need to adjust the balance factor because no thread is inserted or removed, but we do need to adjust the SIR on its path all the way to the root

Pseudocode
Update_SIR(*t*):

if u is root:
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.
else: //if u is not root
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.
t = parent(*t*)
Update_SIR(*t*)

Find_update(*i*, *t*, updated_*status*):
if *t* is null then
return -1
else if *i* = *t.id* then
t.status = updated_status
Update_SIR(*t*)
else if *i* < *t.id* then
Find_update(*i*, left(*t*), updated_*status*):
else if *i* > *t.id* then
Find_update(*i*, right(*t*), updated_*status*):

CHANGESTATUS(*i*, *stat*):
Find_update(*i*, root(***S***), stat)

Why O(log n):
Similar to COMPLETED(*i*). Because recursive call on a root's child, as well as updating SIR(*t*) if it's found traverse at most 2 times the AVL tree height, which is O(log n).

SCHEDULENEXT:
We already know the SIR info in the root, which shows the node with smallest *id* that contains status R. If initially the SIR of the root is null we return -1. Otherwise, we first down-phase traverse from the root to find that thread *t*. Once we reach that *t* we change the *status* to A and return *t*. Before returning t though, we also need to update the SIR of that *t*'s predecessor accordingly.

Pseudocode
Update_SIR(*t*):
if *t* is root:
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.
else: //if u is not root
SIR(*t*) = min{*t.id* iff *t.status*=R else null, SIR(t_{left}), SIR(t_{right})}.
t = parent(*t*)
Update_SIR(*t*)

Find_update_return(*i*, *t*, updated_*status*):

```
if i = t.id then
        t.status = updated_status
        Update_SIR(t)
        return t
else if i < t.id then
        Find_update_return(i, left(t), updated_status):
else if i > t.id then
        Find_update_return(i, right(t), updated_status):
```

SCHEDULENEXT:

```
if SIR(root(S)) is null then
        return -1
else
        Find_update_return(SIR(root(S)), root(S), A)
```

Why O(log n):

Similar to CHANGESTATUS(*i*, *stat*). Because recursive call on a root's child, as well as updating SIR(*t*) if it's found traverse at most 2 times the AVL tree height, which is O(log n).

Question 2(Xiaoxue Xing):

//Verified by Zhenjie Xiong

2.(a)

Since each hash function satisfies the Simple Uniform Hashing Assumption, so for each function, the probability that x is inserted to the empty slot is (k/m), m is the total slot.

P(T[h_1(x)] is empty) = k/m

P(T[h_2(x)] is empty) = k/m

P(T[h_1(x)] is not empty) =(m- k)/m

P(T[h_2(x)] is not empty) = (m-k)/m

Because we use the power of two choices, we will put x in the shorter slot between T[h_1(x)] and T[h_2(x)]. So, if one of T[h_1(x)] and T[h_2(x)] is the empty slot, then x will add to the empty slot. If T[h_1(x)] and T[h_2(x)] are both empty, then x will add to the empty slot T[h_1(x)]. If T[h_1(x)] and T[h_2(x)] are both not empty, x can only add to the slot with shorter chain, but can't add to the empty slot.

h1 and h2 are two independent functions.

P(T[h_1(x)] is empty and T[h_2(x)] is empty) = (k/m)^2

P(T[h_1(x)] is empty and T[h_2(x)] is not empty) = (k/m)*((m-k)/m)

P(T[h_1(x)] is not empty and T[h_2(x)] is empty) = (k/m)*((m-k)/m)

P(T[h_1(x)] is not empty and T[h_2(x)] is not empty) = ((m-k)/m)^2

P(x add to the empty slot) = P(T[h_1(x)] is empty and T[h_2(x)] is empty)
+ P(T[h_1(x)] is empty and T[h_2(x)] is not empty)
+ P(T[h_1(x)] is not empty and T[h_2(x)] is empty)
= (k/m)^2 + (k/m)*((m-k)/m) + (k/m)*((m-k)/m)
= (2*k*m -k^2)/(m^2)

2(b) since m = 4, h1(x) = 0 or 1 or 2 or 3, with each probability ¼. Same as h2(x). h2(x) = 1 or 1 or 2 or 3, with each probability ¼. T[h_1(x)] = 6 or 3 or 9 or 9, with each probability ¼, and T[h_2(x)] = 6 or 3 or 9 or 9, with each probability ¼.

Because h1 and h2 are independent.

	h1(x)	0	1	2	3
h2(x)	T[h(x)]	6	3	9	9
0	6	6	3	6	6
1	3	3	3	3	3
2	9	6	3	9	9

4	9	6	3	9	9

The cell in the shadowed area is the length of the chain x inserted, each with probability 1/16. So the expected value of the length of the chain x inserted is (6+3+6+6+3+3+3+3+6+3+9+9+6+3+9+9)/16 = 5.4375

Question 3(Xiaoxue Xing):

//Verified by Zhenjie Xiong

3. (1)

The data structure we use is a map, key is the word and the value is the frequency. We only have 26 hash slots, and the hash function is the key's first letter. For each hash-table slot T[j], it is a double linked list beginning with the most frequent word (if there is a frequency tie, the word occurring first in alphabetical order rank first)

3.(2)

Algorithm for process each input

Every time we input a word, we first use hash function to decide which slot we will out this word.

Then we search that slot to see if that slot already has a key equal to the word.

If the slot doesn't have the same key with that word, we just create a new key to the slot.The key is the word and value is 1.

If the double linked list doesn't contain key. We put the key in the head of the double linked list.

If the slot contains more than one key, we just compare the first key in the slot with new key. Firstly, we compare their value, if the new key's value is greater than the first key's value, we enter the key to the first place in the double linked list. Else If the new key's value equals to the first key's value, we then compare the alphabetical order of the key. If the new key's alphabetical order is smaller,we enter the key to the first place in the double linked list. Else we enter the key to the end of the double linked list.

If the slot already has that key, we just renew the value with value +1.

If now the slot only contains that one key, we will do nothing.

If now the slot contains more than one key, we just compare the first key in the slot with the key we just changed. Firstly, we compare their value, if the changed key's value is greater than the first key's value, we add the key to the head of linked list and delete the key from the old position. If the changed key's value equals to the first key's value, we then compare the alphabetical order of the key. If the changed key's alphabetical order is smaller, we add the key to the head of linked list and delete the key from the old position. Else we enter the key to the end of the double linked list.

Algorithm for query
We will return the first key in each slot's linked list. If the slot empty, we return null

3(3)

```
Function insertBeginning(List list, Node newNode)
 If list.firstNode ==null
        list.firstNode  := newNode
        list.lastNode   := newNode
        newNode.prev  := null
        newNode.next  := null
else
        insertBefore(list, list.firstNode, newNode)
```

```
Function insertEnd(List list, Node newNode)
If  list.lastNode == null
        insertBeginning(list, newNode)
else
        insertAfter(list, list.lastNode, newNode)
```

```
Function SearchKey(list,key k):          // list is the duble linked list in the specific harsh slot
a = null
node  := list.firstNode
While node ≠ null
        If node = k:
                a = node
                End loop
        Else node  := node.next
Return a
```

```
Function remove(List list, Node node)
If  node.prev ==null
        list.firstNode  := node.next
else
        node.prev.next  := node.next
If node.next ==null
        list.lastNode  := node.prev
else
        node.next.prev  := node.prev
```

```
Function Compare(List L, Key k):
If K.value > L.firstNode.value:
        Return true
Elseif K.value == L.firstNode.value:
        If alphabeticalorder(K) < alphabeticalorder( L.firstNode):
                Return T
        Else: return false
Else: return false

Function Process-an-input(H,k):                 //H is the hash map and the k is the input
L = H(h(k))                                     // h is the hash function,  L is the double linked list
a  = Search (L,k)
If a = null:
        Key = (k,1)
        b = Compare(L,key)
        if b is false:
                insertEnd(L,key)
        If b is true:
                insertFirst(L,key)
If a ≠ null:
        Key = (a.key,.avalue+1)
        b = Compare(L,key)
        if b is false:
                insertEnd(L,key)
        If b is true:
                remove(L,a)
                insertFirst(L,key)


Function query(H):
For linkedlist L in H:
        Return L.firstNode.key
```

3(4)
For process input,
It firstly takes time 1 for each input to decide which slot the input should be.
Then, we search the linked list in that slot, which need time h (h is the length of the list).
Then, we insert the key to the right place, which need constant time c
So, for all input we need $\square h + n*(1+c)$, the expected time is $(\square h + n*(1+c))/n$.
Since the $\square h = n$, so $(\square h + n*(1+c))/n = 1 + (1+c) = (2+c)$. So the expected time to process each input, is $\Theta(1)$.
For query, there is only 26 slot in the map. And we just need to return the first key from the linked list in each slot. So the run time is 26, which is $\Theta(1)$.