S20190010034_A4

C Bhavesh Kumar

October 2020

1 Maintaining a Multi-set data

To implement the given problem a customized **red black tree** can be used. The structure of the red black tree is defined below,i.e, each node in rbt is of type:

```
struct node{
    int data;
    int color;
    struct node *left,*right,*parent;
    int frequency;//used to maintain frequencies of each data
    int size;//used for select operation.
};
```

The variable frequency is used to maintain the count of unique data's duplicates.

(a) Implementation description:

(1) *Init*(M): The data structure can be initialized in $O(1)$ time as it is just allocating memory for the root node which can be done in $O(1)$ runtime.

(2) *Insert*(M,i): Insertion in a Red black tree takes time $O(log(n))$ time where n is the number of nodes.
For our implementation insertion takes $O(log||M||)$ time.
Note that searching in Rbt takes $O(log(||M||))$ time.
$||M||$ represents number of distinct elements in the tree.
First we search for the position to add the new node or search for the node to add its duplicate in $O(log||M||)$ time.And then we insert in constant time $O(1)$,and we fix violation in $O(log(||M||))$ if any.
If the node we are looking to add is already present then we increase the frequency variable of the node.
Therefore time taken to insert is $O(log||M||)$.

(3) *Remove*(M,i): To remove a number from the tree we search for it.And this can be done in $O(log||M||)$ time.
After finding the node we will decrease its frequency by one.If frequency reaches zero we will delete the node and fix the tree if any violation occurs and this can be done in $O(log(||M||))$.
Therefore the time taken to remove a number is $O(log(||M||))$.

(4) *Frequency*(M,i):
To find the frequency of a number i,all we need to do is search for the number and get the value of frequency variable of that node.
Search takes $O(log(||M||))$ time.And accessing frequency variable takes $O(1)$ time and hence the running time of $O(log(||M||))$.

(5) *Select*(M,i): The Red black tree gives the elements in a sorted order if we traverse the tree using inorder traversal.
Using this we can find the k'th smallest element.
But note that this takes O($||M||$) time.
To implement select opeeration in $O(log(||M||))$ time we augment tree with extra information.
This extra information in our case is the size of the subtrees of the node which we store in the variable size.
$Sizeofnumber = size(x) = size(leftsubtree)+size(rightsubtree)$+frequency of the node.
We can calculate this during insertion operation and update it during deletion operation,this takes constant time and does not affect time complexity of insertion and deletion operations.
After successfully obtaining the information that we need, we can implement the select operation in $O(log(||M||))$ time.
The following pseudo code implements it.

```
Select(M, i):
    if M.left!=NILL and i<=M.left.size then
     return select(M.left,i)
    elif M.right!=NILL and i>M.size-M.right.size then
     return select(M.right,i-(M.size-M.right.size))
    else
     return M.data
  endif

```

(b) Algorithm for sorting.
We know that inorder traversal of red black tree gives the sorted order of elements but doing it this way gives us time complexity of $O(|L| * |L|)$,where $|L|$ is the number of nodes in tree.
To sort efficiently, we make use of select operation.
The following code sorts the list in time = $O(|L|log(||L||))$ .where $|L|$ is the number of elements in the list and $||L||$ is the number of distinct elements in the list.

```
init(M)
for i = 1 to |L|
    insert(M,L[i])
for i= 1 to |L|
    L[i]=select(M,i)

```

This algorithm successfully sorts the elements in the list.As select(M,i) gives i'th smallest element in $O(log(||L||))$ time,and insert operation also

takes $O(log(||L||))$ time, the time complexity of the algorithm is $O(|L|log(||L||)+ |L|log(||L||))$ which is nothing but $O(|L|log(||L||))$.

2 The Count Min sketch

(i) The final array $A = \{3, 2, 4, 4, 3\}$.
That is, A[0]=3, A[1]=2, A[2]=4, A[3]=4, A[4]=3.

(ii) $V_i = \sum_{j=1}^{n} f_j * C_{i,j}$

(iii) From previous question we know that $V_i = \sum_{j=1}^{n} f_j * C_{i,j}$
$=> V_i = f_i + \sum_{j \neq i} f_j * C_{i,j}$
$=> V_i = f_i + \sum_{j \neq i} f_j * C_{i,j} \geq f_i$
$=> V_i \geq f_i$
Hence proved.

(iv) Show that $E[Vi] \leq fi + N/w.$, where $N = \sum_{i=1}^{n} fi$
We know that $V_i = f_i + \sum_{j \neq i} f_j * C_{i,j}$
$=> E[V_i] = E[f_i + \sum_{j \neq i} f_j * C_{i,j}]$
$=> E[V_i] = E[f_i] + E[\sum_{j \neq i} f_j * C_{i,j}]$
$=> E[V_i] = f_i + E[\sum_{j \neq i} f_j * C_{i,j}]$, since E[Constant]=Constant.
$=> E[V_i] \leq f_i + \sum_{j \neq i} f_j * E[C_{i,j}]$
$=> E[V_i] \leq f_i + N * E[C_{i,j}]$
We know that $E[C_{i,j}]$ =probability of $[C_{1,j}]$ = probability of $[h[x_i] = h[x_j]]$
Since h is taken randomly from the universal set of hash functions, we know
$P[h[x_i] = h[x_j]] = 1/w$
Therefore,
$E[V_i] \leq f_i + N/w$

(v) Prove that $P(V_i \geq f_i + eN/w) \leq 1/e$
Let $V_i^{'} = V_i - f_i$
$=> P(V_i \geq f_i + eN/w) = P(V_i^{'} \geq eN/w)$
Using markov's inequality,
$=> P(V_i^{'} \geq eN/w) \leq E[V_i^{'}]/(eN/w) \leq (N/w)/(eN/w) \leq 1/e$
where $E[V_i^{'}] = E[V_i] - f_i = N/w$
Therefore $P(V_i \geq f_i + eN/w) \leq 1/e$.

(vi) Prove that $P(U_i \geq f_i + eN/w) \leq (1/e)^d$
From previous question we know that $P(V_i \geq f_i + eN/w) \leq 1/e$
Now,$P(U_i \geq f_i + eN/w) = P$(for all i in [1,d] $V_i \geq f_i + eN/w) \leq (1/e) * (1/e) * .....d$ times $\leq (1/e)^d$
Therefore $P(U_i \geq f_i + eN/w) \leq (1/e)^d$