**CSCI-GA.1170-001/002 Fundamental Algorithms** October 29, 2015

# Solutions to Problem 1 of Homework 5 (15 points)

*Name: GOWTHAM GOLI (N17656180)* *Due: Tuesday, October 13*

Assume you are given a data structure $D$ which supports the following two operations:

- INSERT$(D, value)$. Inserts a value $value$ into $D$. If $D$ has $n$ elements, assume this procedure takes $I(n)$ time.
- SEARCH$(D, value)$. It $D$ contains at least one element equal to $value$, return the pointer to this element (else returns $nil$). Assume this procedure takes $S(n)$ time.
- INORDERWALK$(D)$. Outputs all $n$ elements of $D$ in sorted order. Assume this procedure takes linear time $O(n)$.

Using $D$, you would like to build a new data structure $R$, which can deal with many repeated elements more efficiently, by supporting the following operations.

- ADD$(R, value)$. Inserts a value $value$ into $R$.
- FREQUENCY$(R, value)$. Returns the number of elements of $R$ equal to $value$ (i.e., how many times was ADD$(R, value)$ called before).
- FASTINORDERWALK$(R)$. Outputs all *distinct* elements of $D$ in sorted order, together with their frequency values.

(a) (5 pts) Using $D$, show how to implement $R$, so that the following is true. If $R$ contains $n$ records, but only $t$ of them are distinct, where $t$ could be much less than $n$, then

- ADD$(R, key)$ should run in time $A(n, t) \approx I(t) + S(t)$;
- FREQUENCY$(R, key)$ should run in time $F(n, t) \approx S(t)$;
- FASTINORDERWALK$(R)$ should run in time $O(t)$.

Namely, all run times are *independent of* $n$. For example, if ADD has been called 4 times on $(R, 7)$ and 5 times on $(R, 6)$ then FREQUENCY$(R, 3)$ returns 0 but FREQUENCY$(R, 7)$ returns 4, and both calls take time $F(9, 2) \approx S(2)$, where $t = 2$ because only two distinct values were inserted so far (despite $n = 4+5 = 9$). Also, FASTINORDERWALK$(R)$ will output $(6, 5), (7, 4)$ in time $O(2)$.
(**Hint**: Add a field $v.num$ in addition to $v.key$, which counts how many elements are equal to $v.key$.)

**Solution:**

## Implementing $R$ using $D$

Maintain a field called $num$ for each node in $D$ which maintains the frequency of that node and eliminate all the other duplicates from $D$. This new data structure formed is $R$ (i.e $R$ is similar to $D$, the only difference is $R$ has all distinct elements in it and maintains an additional frequency field). Therefore if $D$ has $t$ distinct elements in it then number of elements in $R$ is $t$.

INSERT$(R, value)$ takes $I(t)$ time
SEARCH$(R, value)$ takes $S(t)$ time
INORDERWALK$(R)$ takes $O(t)$ time

ADD$(R, key)$

- Search if the $key$ is present in $R$. Time taken in this step is $S(t)$
- If there is a node $v$ such that $v.key = key$ then $v.num = v.num + 1$. Time taken in this step is $O(1)$
- If there is no such node $v$ then insert the $key$ into $R$. Time taken in this step is $I(t)$

$\therefore A(n, t) \approx I(t) + S(t)$

FREQUENCY$(R, key)$

- Search if the key is present in $R$. Time taken in this step is $S(t)$
- If there is a node $v$ such that $v.key = key$ then return $v.num$
- Else return 0

$\therefore F(n, t) = S(t)$

FASTINORDERWALK$(R, key)$

- Just calls INORDERWALK$(R)$ and also return the frequency of each node along with it's key. Therefore, the procedure returns the list of tuples $(v.key, v.num)$ sorted by $v.key$

$\therefore$ Time taken is $O(t)$ □

(b) (5 pts) For each of the following implementations of $D$, compute the running times $A(n, t)$ and $F(n, t)$ of ADD and FREQUENCY that you get by using your solution from part (a). Which data structure is the best? Make sure to justify your answers.

- Implement $D$ as a linked list.
- Implement $D$ as a sorted array.
- Implement $D$ as a 2-3-tree.

**Solution:**

### $D$ as a linked list

$S(t) = O(t)$ and $I(t) = O(1)$

$\therefore A(n,t) = O(t+1) = O(t)$, $F(n,t) = O(t)$ and FastInOrderWalk takes $O(t)$ time

### $D$ as a sorted array

$S(t) = O(\log t)$ and $I(t) = O(t)$

$\therefore A(n,t) = O(t + \log t) = O(t)$, $F(n,t) = O(\log t)$ and FastInOrderWalk takes $O(t)$ time

### $D$ as a 2-3 tree

$S(t) = O(\log t)$ and $I(t) = O(\log t)$ $\therefore A(n,t) = O(\log t + \log t) = O(\log t)$, $F(n,t) = O(\log t)$ and FastInOrderWalk takes $O(t)$ time

Therefore the best data structure to use would be a 2-3 Tree □

(c) (5 pts) Using the best data structure developed in part (b), give an algorithm for sorting $n$ integers with at most $t$ distinct values in time $O(n \log t)$. Make sure you justify your running time bound.

**Solution:**

$R$ in this case is a 2-3 Tree. Therefore, each leaf node stores two fields, *key* and *num*. Also, in 2-3 Trees, the leaf nodes when seen from left to right are in a sorted order. So starting from the leftmost node keep calling it's successor until we reach the rightmost node. (Computing successor of a node takes $O(\log t)$ time which is proved in Problem-4)

```
1  Algorithm: Sort(T)
2  A ← NewArray(n)
3  c ← 0
4  v ← Left-most leaf node in T
5  while v is not NULL do
6  |   for i ← 1 to v.num do
7  |   |   A[c] ← v.key
8  |   |   c ← c + 1
9  |   end
10 |   v ← Successor(v)
11 end
12 Return A
```

**Algorithm 1:** Sorting $n$ integers with $t$ distinct values in $O(n \log t)$ time

## Time Complexity

Time taken to find the left most leaf node in the first step is $O(\log t)$. The while loop runs for $t$ times and in each iteration we call SUCCESSOR which takes $O(\log t)$ (proof in Problem-4), the *for* loop iterates for number of repetitions of that element. Hence the overall run time of *for* loop is $O(n)$ and SUCCESSOR is $O(t \log t)$. Therefore, the total running time is $O(n + t \log t) = O(n \log t)$ □

**CSCI-GA.1170-001/002 Fundamental Algorithms** October 29, 2015

# Solutions to Problem 2 of Homework 5 (10 (+5) points)

*Name: GOWTHAM GOLI (N17656180)* *Due: Tuesday, October 13*

Assume you are given a binary search tree $T$ of height $h$ and with $n$ elements in it. For simplicity, assume all the elements are distinct.

(a) (5 pts) Use a slight modification of the PostOrder-Tree-Walk procedure to argue that in time $\Theta(n)$ you can compute, for every node $v$, the number of even nodes (call it $even(v)$) in $v$'s sub-tree.
(**Hint**: In addition to $even(v)$, also compute the total number of nodes in $v$'s subtree.)

**Solution:**

Let $T$ be the given tree and $x$ be $T.root$, then number of even nodes in $x$, $even(x)$ will be

- If $x.value$ is even, then $even(x.left) + even(x.right) + 1$
- If $x.value$ is odd, then $even(x.left) + even(x.right)$

```
1  Algorithm: NumEvenNodes(T)
2  x ← T.root
3  if x is NULL then
4  |  Return 0
5  even_l ← NumEvenNodes(x.left)
6  even_r ← NumEvenNodes(x.right)
7  if x.value is divisible by 2 then
8  |  even(x) ← even_l + even_r + 1
9  else
10 |  even(x) ← even_l + even_r
11 Return even(x)
```

**Algorithm 2:** Algorithm to calculate number of even nodes in a Tree $T$

□

(b) (5 pts) Now that each node $v$ contains the value $even(v)$, show how to keep maintaining this value for each successive $Insert$ operation. Namely, show how to perform an $Insert$ operation in time $O(h)$, while correctly maintaining all the $even(v)$ values.

**Solution:**

Let $z$ be the node being inserted

- If $z.key$ is odd then just insert $z$ at the appropriate position. This step takes $O(h)$ time
- If $z.key$ is even, then for every node $v$, $z$ visits while being inserted add 1 to $even(v)$. This step takes $O(h)$ time

Therefore, the updated Insert algorithm takes $O(h)$ time

```
1  Algorithm: INSERT(T, z)
2  y ← NULL
3  x ← T.root
4  if z.key is divisible by 2 then
5  |  z.even = 1
6  else
7  |  z.even = 0
8  while x is not NULL do
9  |  if z.key is divisible by 2 then
10 |  |  x.even = x.even + 1
11 |  y = x
12 |  if z.key < x.key then
13 |  |  x = x.left
14 |  else
15 |  |  x = x.right
16 end
17 z.p = y
18 if y is NULL then
19 |  T.root ← z
20 else if z.key < y.key then
21 |  y.left ← z
22 else
23 |  y.right ← z
```

**Algorithm 3:** Insert procedure in $O(h)$ while maintaining all the $even(v)$ values

□

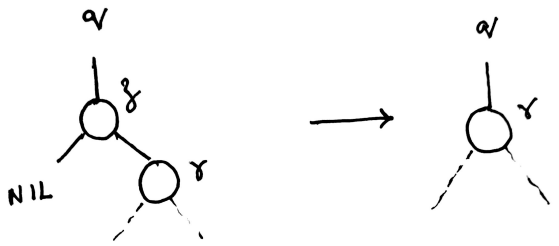
(c)* (5 pts) (**Extra Credit:**) Similar to part (b), but do it for the *Delete* operation. Namely, show how to perform a *Delete* operation in time $O(h)$, while correctly maintaining all the $even(v)$ values.

**Solution:**

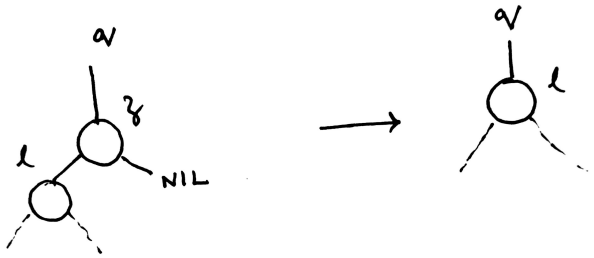
Let $z$ be the node to be deleted. Then we have the following 4 cases and in all the cases first we do the following operation

If $z.key$ is even then starting from $z$ go up until the root and for each node $v$ in the path, decrease $even(v)$ by 1. Therefore, this takes $O(h)$ time

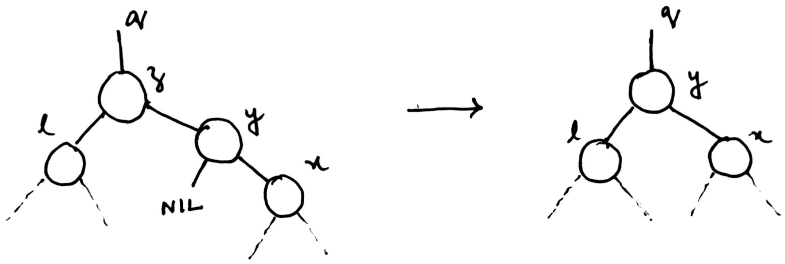
- Node $z$ has no left child. We replace $z$ by its right child $r$ which may or may not be NIL. This takes only $O(1)$ time




- Node $z$ has a left child $l$ but no right child. We replace $z$ by $l$. This takes only $O(1)$ time

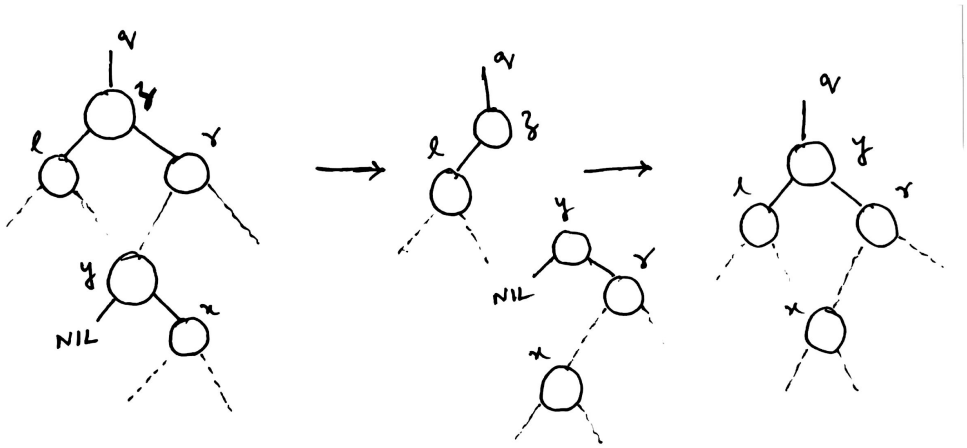


– Node $z$ has two children, its left child is node $l$, its right child is its successor $y$ and $y$'s right child is node $x$. We replace $z$ by $y$, updating $y$'s left child to become $l$, but leaving $x$ as $y$'s right child and update $even(y) = even(y) + even(l)$. This takes only $O(1)$ time




update $even(y) = even(y) + even(l)$. Therefore, this takes $O(h)$ time

– Node $z$ has two children (left child $l$ and right child $r$), and its successor $y \neq r$ lies within the subtree rooted at $r$. If $y$ is even then for each node $u$ starting from $y.p$ to $r$ decrease $even(u)$ by 1 and replace $y$ by it's own right child $x$, set $y$ to be $r$'s parent and update $even(y) = 1 + even(r)$.




Then set $y$ to be $q$'s child and again update $even(y) = even(y) + even(l)$. This, takes $O(h)$ time

Therefore by adding some extra computations which take $O(h)$ time we can still perform the delete operation while correctly maintaining the $even(v)$ values in $O(h)$ time $\square$

**CSCI-GA.1170-001/002 Fundamental Algorithms** October 29, 2015

# Solutions to Problem 3 of Homework 5 (8 points)

*Name: GOWTHAM GOLI (N17656180)* *Due: Tuesday, October 13*

We want to build a data structure for maintaining a (potentially infinite) matrix $M$ and support the following operations.

- INITIALIZE($M$): Create an empty matrix $M$ with all zero entries.
- FIND($M, i, j$): Return the value at index $i, j$.
- UPDATE($M, i, j, e$): Change the value at index $i, j$ to $e$.
- TRANSPOSE($M$): Transpose the matrix $M$.
- ADD($M$): Return the sum of all entries of $M$.

Assume that the matrix is of arbitrary dimensions. Use 2-3 trees appropriately to obtain a data structure such that INITIALIZE, TRANSPOSE, and ADD run in $O(1)$ time, and FIND and UPDATE run in $O(\log k)$ time, where $k$ is the number of non-zero entries in the matrix.

**Solution:**

## Building $M$ using 2-3 Tree

Let $H$ be a hash function such that $H[(i, j)]$ produces a unique key for each pair $(i, j)$. Assume that applying $H$ on any index $(i, j)$ takes only $O(1)$ time.

Let $M$ be an empty 2-3 Tree. Every node $v \in M$ has the following fields

- $v.left, v.mid, v.right$
- If $v$ is a leaf node then $v.key$ stores the hashed value of $(i, j)$. If $v$ is a non-leaf node then $v.key$ stores the maximum key of it's children
- $v.sum$ stores the sum of all the sums of it's children. If $v$ is a leaf node then $v.sum = v.key$
- If $v$ is a root node then $v$ has one additional field $init$ such that $v.init = k$ initializes all the matrix entries to $k$
- If $v$ is a leaf node then $v$ has one additional field $value$ where $v.value$ stores the value at $M(i, j)$

Now compute $H[(i, j)]$ for every $(i, j)$ and insert them into $M$ while maintaining the above fields at every node.

INITIALIZE($M$)

Let $x$ be the root of $M$. Update $x.init$ to 0. Therefore, INITIALIZE takes $O(1)$ time

FIND($M, i, j$)

Compute $H(i, j)$. Now perform the usual Search operation and return $v.value$ where $v$ is the leaf node returned by the Search operation. Therefore, FIND takes $O(\log k)$ time

UPDATE($M, i, j, e$)

Compute $H(i, j)$. Now perform the usual Search operation and update $v.value$ to $e$ where $v$ is the leaf node returned by the Search operation. Therefore, UPDATE takes $O(\log k)$ time

TRANSPOSE($M$)

When a matrix is transposed, the indices are all flipped i.e $(i, j) = (j, i)$. Therefore to return any element at $(i, j)$ position return the element at $(j, i)$ position. Therefore, TRANSPOSE takes $O(1)$ time

ADD($M$)

If $x$ is the root of $M$ then return $x.sum$. Therefore, ADD takes $O(1)$ time □

**CSCI-GA.1170-001/002 Fundamental Algorithms** October 29, 2015

# Solutions to Problem 4 of Homework 5 (14 points)

*Name: GOWTHAM GOLI (N17656180)* *Due: Tuesday, October 13*

Assume that you are given a 2-3 tree $T$ containing $n$ distinct elements.

(a) (4 points) Show how to find the successor of a given element $x \in T$ in time $O(\log n)$.

**Solution:**

The leaves of 2-3 Trees are in a sorted order when seen from left to the right. Therefore the successor of any given node is the adjacent leaf node to it's right. In order to find the successor of any leaf node $x$, we proceed as follows.

- Go up the tree to the first ancestor of $x$ that has a child to the right. (i.e we find the first ancestor of $x$, such that the sub tree in which $x$ is located has a right sibling)
- Find the leftmost leaf node in this sibling sub tree by following the left children all way until the leaf. This leaf we find is the successor of $x$

In the worst case, we start from $x$ and go up the tree until the root and then till the bottom to the minimal leaf node in root's right subtree. Therefore, the worst case running time is $O(\log n)$

□

(b) (4 points) Show that if the input element $x$ is chosen *uniformly at random* from $T$, then your procedure from part (a) runs in *expected* time $O(1)$.

**Solution:**

Consider a node $v$ which has three children $a, b$ and $c$. Therefore, $succ(a) = b$ and $succ(b) = c$. Finding the successors of these two nodes takes only $O(1)$ time and to find $succ(c)$ we need to go up the tree until we find an ancestor such that it has a right child. This might take $O(\log n)$ time. Therefore, the probability that the successor operation takes only $O(1)$ time is $2/3$ and $O(\log n)$ is $1/3$.

If the node $v$ only two children $a$ and $b$ then it is clear that the probability that the successor operation takes only $O(1)$ time is $1/2$ and $O(\log n)$ is $1/2$. Therefore, in the worst-case scenario, every node of the tree has only 2 children i.e a perfectly balanced binary tree

Let $n$ be the total number of leaf nodes and $h$ be the height of the 2-3 Tree i.e $h = \log n$

The number of leaf nodes $x$ such that ancestor of $x$ has a right child at height 1 from $x$ is $n/2$. Therefore, finding successor of these $n/2$ nodes takes time $n/2.O(1)$

The number of leaf nodes $x$ such that ancestor of $x$ has a right child at height 2 from $x$ is $n/4$. Therefore, finding successor of these $n/4$ nodes takes time $n/4.O(2)$

The number of leaf nodes $x$ such that ancestor of $x$ has a right child at height 3 from $x$ is $n/8$. Therefore, finding successor of these $n/4$ nodes takes time $n/8.O(3)$

$\vdots$

The number of leaf nodes $x$ such that ancestor of $x$ has a right child at height $\log n$ from $x$ is 1. Therefore, finding successor of this 1 node takes time $O(\log n)$

Let $T(n)$ be the expected running time, then

$$\begin{aligned} T(n) &= \frac{1}{n}[\frac{n}{2}O(1) + \frac{n}{4}O(2) + \frac{n}{8}O(3) + \ldots + 1.O(\log n)] \\ &= \frac{1}{n}\sum_{i=1}^{\log n}\frac{n}{2^i}O(i) \\ &= \sum_{i=1}^{\log n}\frac{O(i)}{2^i} \end{aligned}$$

Consider the summation, $\sum_{i=0}^{\infty} x^j = \frac{1}{1-x}$

Take derivative on both sides and multiply by $x$ we get $\sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}$

Put $x = 1/2$ in the above equation, we get $\sum_{i=0}^{\infty}\frac{i}{2^i} = 2 \implies \sum_{i=0}^{\infty}\frac{O(i)}{2^i} = O(1)$

$$\begin{aligned} T(n) &= \sum_{i=1}^{\log n}\frac{O(i)}{2^i} \leq \sum_{i=0}^{\infty}\frac{O(i)}{2^i} \\ \therefore T(n) &= O(1) \end{aligned}$$

□

Assume that we wish to augment our 2-3 tree data structure so that that each node $v$ maintains a pointer $v.succ$ to the successor of $v$, so that queries for the successor of an element can be answered in $O(1)$ time *worst-case*.

(c) (6 points) Show that the 2-3 trees can be augmented while maintaining $v.succ$, such that the INSERT and DELETE operations can still be performed in $O(\log n)$ time. (**Hint**: Think of a linked list.)

**Solution:**

Given that leaf node $v$ has a pointer $v.succ$ to the successor of $v$, so the leaf nodes of the 2-3 Tree form a sorted liked list.

INSERT

Let $y$ be the node to be inserted into the tree. Perform the usual Insert operation in $O(\log n)$ time.

- If $y$ is the leftmost leaf node in the tree. Let $z$ be the adjacent leaf node of $y$ to it's right, then $y.succ = z$. This takes $O(1)$ time
- If $y$ is the rightmost leaf node in the tree. Let $z$ be the adjacent lead node to $y$ to it's left, then $z.succ = y$ and $y.succ = null$. This takes $O(1)$ time
- Else, Let $x$ be the predecessor of $y$ and let $z$ be $x.succ$ then $x.succ = y$ and $y.succ = z$. This step takes $O(\log n)$ time

Therefore, the Insert operation still takes $O(\log n)$ time while maintaining the *succ* field

DELETE

Let $y$ be the node to be deleted in the tree

- If $y$ is the leftmost or the rightmost leaf node to be deleted in the tree, then perform the usual Delete operation. This step takes $O(\log n)$ time
- Else, Let $x$ be the predecessor of $y$ and $y.succ$ be $z$, then perform the usual Delete operation and update $x.succ$ to $z$. This step takes $O(\log n)$ time

Therefore, the Delete operation still takes $O(\log n)$ time while maintaining the *succ* field $\square$