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:** ********** INSERT YOUR SOLUTION HERE **********

(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.

********** INSERT YOUR NAME HERE *****, Homework 6, Problem 1, Page 1
– Implement $D$ as a linked list.
– Implement $D$ as a sorted array.
– Implement $D$ as a 2-3-tree.

**Solution:** ***************** INSERT YOUR SOLUTION HERE *****************  

(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:** ***************** INSERT YOUR SOLUTION HERE *****************
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: ****************** INSERT YOUR SOLUTION HERE ******************

(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: ****************** INSERT YOUR SOLUTION HERE ******************

(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: ****************** INSERT YOUR SOLUTION HERE ******************
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:** ***************** INSERT YOUR SOLUTION HERE ************

**** INSERT YOUR NAME HERE ****, Homework 6, Problem 3, Page 1
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: ***************** INSERT YOUR SOLUTION HERE *****************  

(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: ***************** INSERT YOUR SOLUTION HERE *****************  

Assume that we wish to augment our 2-3 tree data structure so 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 $\text{INSERT}$ and $\text{DELETE}$ operations can still be performed in $O(\log n)$ time. (Hint: Think of a linked list.)

Solution: ***************** INSERT YOUR SOLUTION HERE *****************  

**** INSERT YOUR NAME HERE ****, Homework 6, Problem 4, Page 1