**Assignment 2**

**Name: Hongzhi Fu**

**Student Number: 1058170**

**1.(a)**

**function InsertNode**(x, u)

v ← **NewNode**(x, u, NULL)

**while CoinFlip**()

**while** u.up = NULL **and** u.left ≠ NULL **do**

u ← u.left

u ← u.up

v ← **NewNode**(x, u, v)

**if IsHead**(u) **then**

w ← **NewNode**(, NULL, u)

**NewNode**(, w, v.right)

**return** v

**(b)**

**function Merge**(A, m, B, n)

**while** A.down ≠ NULL **do**

A ← A.down

A ← A.right

**while** B.down ≠ NULL **do**

B ← B.down

B ← B.right

**while** B ≠ **do**

**if** A.key < B.key **then**

A ← A.right

**else**

**InsertNode**(B.key, A.left)

B ← B.right

**2.(a)**

// input two sets of X and Y

// output their uncommon components stored in an array

// algorithm: sort two sets at the nondecreasing order, and we iterate each element in set X, // check if it is in the set Y, append uncommon element in set X. After that, we reverse this // procedure. In other words, inspect each element in set Y and put the uncommon one into the // list S.

**function FindUncommon**(X, Y)

**HeapSort**(X), **HeapSort**(Y) // sort set X and Y

k ← 0, S[ ] ←NULL

**for** i ← 0 to n - 1 **do**

**if not BinarySearch**(X, Y[i]) **then**

S[k] = Y[i], k += 1 // if Y[i] is not in set X using binary search, append it into an array

**for** j ← 0 to n - 1 **do**

**if BinarySearch**(Y, X[i]) **then**

S[k] = X[j], k += 1 // if X[i] is not in set Y using binary search, append it into an array

**return** S

**(b)**

// input: two sets of X and Y

// output: their uncommon components stored in an array

// algorithm: create two hash tables with the elements in set X and set Y, respectively. We // iterate the process to check if the element of set X is in set Y. If not, we put the uncommon // element into a list S. Then we select the uncommon element in set Y by iterating the // elements in set X and finally append them into list S.

**function FindUncommon**(X, Y)

k ← 0, S[ ] ← NULL

tableX ← HashTable(), tableY ← HashTable() // create two empty hash tables

**for** i ← 0 to n - 1 **do**

tableX.insert(X[i]) // insert elements of set X into a hash table

tableY.insert(Y[i]) // insert elements of set Y into a hash table

**for** j ← 0 to n - 1 **do**

**if not** tableX.search(Y[j]) **then**

S[k] = Y[j], k += 1 // if searching Y[j] failed in tableX, append it into an array

**for** k ← 0 to n - 1 **do**

**if not** tableY.search(X[j]) **then**

S[k] = X[j], k += 1 // if searching X[k] failed in tableY, append it into an array

**return** S

**3.(a)**

To solve this problem, we need to split the original problem into two subproblems, that is finding the minimal awkwardness score of left subtree plus the minimal awkwardness score of right subtree. whether each node(staff member) is selected or not determines the awkwardness score between this node and its parent node and child node(s). Concretely, If **T(root)** is our goal we want to get and the left child of the root node is selected, we can reduce this problem to **T(root.L) + root.L.alpha**. However, if the **root.L** is not chosen, we need to consider the next level of the **root.L**, such that finding the minimal awkwardness score of **root.L.L** plus the minimal awkwardness score of **root.L.R**. In this case, the problem is reduced to **T(root.L.L) + T(root.L.R)**. To find the global minimal value of awkwardness, we choose the the smaller one between **T(root.L) + root.L.alpha** and **T(root.L.L) + T(root.L.R).** The procedure is done recursively until we encounter the leaf node which merely returns 0. Furthermore, some special cases are needed to taken into consideration and the recurrence relationship can be described as follows:

T(root) = 0 if root = NULL or (root.L = NULL and root.R = NULL)

T(root) = **Min**(root.L.alpha + **T**(root.L), **T**(root.L.L) + **T**(root.L.R))+**Min**(root.R.alpha + **T**(root.R), **T**(root.R.L) + **T**(root.R.R))

if root has two levels that has a complete tree

**(b)**

// input: the pointer of root node

// output: the minimal awkwardness score of the tree

// function **Min** means return the minimal number of two numbers

// root.L.alphaand root.R.alpha means the score between root node and its left(right) child

// special cases: for those cases where left child or right child is absent, we need to consider

// how to return the local minimal to the parent node. Specifically, for those absent left node or

// right node, we can replace them with 0, which means NULL pointer always returns 0 to the

// bottom, so we return the corresponding value based on each case.

**function FindMinimal**(root)

**if** root = NULL **or** (root.L = NULL **and** root.R = NULL) **then**

**return** 0 // return 0 if it is the NULL pointer or it is the leaf node

**if** root.L ≠ NULL **and** root.R = NULL **then**

**if** root.L.L = NULL **and** root.L.R = NULL **then**

**return Min**(root.L.alpha, 0)

**if** root.L.L ≠ NULL **and** root.L.R = NULL **then**

**return Min**(root.L.alpha + **FindMinimal**(root.L), **FindMinimal**(root.L.L), 0)

**if** root.L.L = NULL **and** root.L.R ≠ NULL **then**

**return Min**(root.L.alpha + **FindMinimal**(root.L), **FindMinimal**(root.L.R), 0)

**if** root.L = NULL **and** root.R ≠ NULL **then**

**if** root.R.L = NULL **and** root.R.R = NULL **then**

**return Min**(root.R.alpha, 0)

**if** root.R.L ≠ NULL **and** root.R.R = NULL **then**

**return Min**(root.R.alpha + **FindMinimal**(root.R), **FindMinimal**(root.R.L), 0)

**if** root.R.L = NULL **and** root.R.R ≠ NULL **then**

**return Min**(root.R.alpha + **FindMinimal**(root.R), **FindMinimal**(root.R.R), 0)

left ← **Min**(root.L.alpha + **FindMinimal**(root.L), **FindMinimal**(root.L.L) + **FindMinimal**(root.L.R))

right ← **Min**(root.R.alpha + **FindMinimal**(root.R), **FindMinimal**(root.R.L) + **FindMinimal**(root.R.R))

**return** left + right