```
//instaAnalysis.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "avltree.h"
#include "queue.h"

AVLTree readData(char []);
AVLTree insertPost(AVLTree, char [], char [], int, int, char [], char []);
void displayPosts(AVLTree);
void mostReach(AVLTree);
void mostEngaged(AVLTree, int);

/* I used "side" element to connect the same total reach values to tree, as you said from the mail to me.
*/


int main(int argc, char *argv[]){
	printf("Welcome to Data Analysis @ Instagram\n");
	printf("====================================\n\n");

	AVLTree root;
	root = readData(argv[1]);// sending the fileName for reading.

	printf("Data Processed:\n");
	printf("===============\n");
	displayPosts(root);

	mostReach(root);
```

```
	mostEngaged(root, -1);

	return 0;
}

AVLTree readData(char nameFile[]){
	AVLTree root=NULL;

	FILE *inFile;

	inFile=fopen(nameFile,"r");
	if (inFile == NULL){
		printf("Error occured while reading the file!");
		exit(-1);
	}

	char *token;
	char line[1024];
	char postId[35], permaLink[75], type[10], posted[35];
	int totalReach, engagedUser;


	while((fscanf(inFile,"%[^\n]\n",line))!=EOF){

		token = strtok(line,";");
		strcpy(postId,token);
		token = strtok(NULL,";");
		strcpy(permaLink,token);
		totalReach = atoi(strtok(NULL,";"));
		engagedUser = atoi(strtok(NULL,";"));
		token = strtok(NULL,";");
```

```
            strcpy(type,token);
            token = strtok(NULL,"\n");
            strcpy(posted,token);

            root = insertPost(root, postId, permaLink, totalReach, engagedUser, type, posted);

        }

        fclose(inFile);
        return root;
}

AVLTree insertPost(AVLTree t, char postId[], char permaLink[], int totalReach, int engagedUser, char type[], char posted[]){
        if (t == NULL){
            t = (AVLTree)malloc(sizeof(struct AVLTreeNode));
            if (t == NULL){
                printf("Error occured while allocating the memory!");
                exit(-1);
            }

            strcpy(t->postId, postId);
            strcpy(t->permaLink, permaLink);
            t->totalReach = totalReach;
            t->engagedUser = engagedUser;
            strcpy(t->type, type);
            strcpy(t->posted, posted);

            t->height = 0;
```

```
                t->left = NULL;
                t->right = NULL;
                t->side = NULL; //initializing the side too
        }
        else{
                if (totalReach < t->totalReach){ // if smaller
                        t->left = insertPost(t->left, postId, permaLink, totalReach, engagedUser, type,
posted);

                        if (HeightAvlTree(t->left) - HeightAvlTree(t->right) == 2){
                                if (totalReach < t->left->totalReach)
                                        t = RightRotation(t);
                                else
                                        t = DoubleRightRotation(t);
                        }
                }

                else if (totalReach > t->totalReach){ // if bigger
                        t->right = insertPost(t->right, postId, permaLink, totalReach, engagedUser,
type, posted);

                        if (HeightAvlTree(t->right) - HeightAvlTree(t->left) == 2){
                                if (totalReach > t->right->totalReach)
                                        t = LeftRotation(t);
                                else
                                        t = DoubleLeftRotation(t);
                        }
                }

                else{ // if equal
                        AVLTree temp;
                        temp = (AVLTree)malloc(sizeof(struct AVLTreeNode));
```

```
                        if (temp == NULL){
                                printf("Error occured while allocating the memory!");
                                exit(-1);
                        }

                        strcpy(temp->postId, postId);
                        strcpy(temp->permaLink, permaLink);
                        temp->totalReach = totalReach;
                        temp->engagedUser = engagedUser;
                        strcpy(temp->type, type);
                        strcpy(temp->posted, posted);

                        AVLTree traversal = t;
                        while(traversal->side) // if there is existing side element, traverse to the last
side element
                                traversal = traversal->side;

                        traversal->side = temp;
                        temp->side = NULL;
                        temp->left = NULL;
                        temp->right = NULL;
                }

                t->height = MaxOfTwo(HeightAvlTree(t->left),HeightAvlTree(t->right))+ 1;
        }
        return t;
}

int HeightAvlTree(AVLTree t){
        if (t == NULL)
                return -1;
```

```
        else
                return t->height;
}

AVLTree RightRotation(AVLTree t){
        AVLTree temp;
        temp = t->left;
        t->left = temp->right;
        temp->right = t;

        t->height = MaxOfTwo(HeightAvlTree(t->left),HeightAvlTree(t->right))+1;
        temp->height = MaxOfTwo(HeightAvlTree(temp->left),HeightAvlTree(temp->right))+1;

        return temp;
}

AVLTree LeftRotation(AVLTree t){
        AVLTree temp;
        temp = t->right;
        t->right = temp->left;
        temp->left = t;

        t->height = MaxOfTwo(HeightAvlTree(t->left),HeightAvlTree(t->right))+1;
        temp->height = MaxOfTwo(HeightAvlTree(temp->left),HeightAvlTree(temp->right))+1;

        return temp;
}

AVLTree DoubleRightRotation(AVLTree t){
        t->left = LeftRotation(t->left);
        return RightRotation(t);
```

```
}

AVLTree DoubleLeftRotation(AVLTree t){
        t->right = RightRotation(t->right);
        return LeftRotation(t);
}

int MaxOfTwo(int a, int b){
        if (a > b)
                return a;
        else
                return b;
}



void displayPosts(AVLTree t){
   if(t != NULL){
      displayPosts(t->right);

      printf("Post ID: %s\n",t->postId);
      printf("Total Reach: %d Engaged users: %d\n",t->totalReach, t->engagedUser);

      if(t->side) // if there is side element, recursively calling again
          displayPosts(t->side);

      displayPosts(t->left);
   }
}

void mostReach(AVLTree t){
```

```
/*    Since there is only one while loop, and this while loop traverses "number of right nodes from the root node" times, and since we are not looking every node in the tree,
   the complexity is = O(log(n)). I think there is no other way to decrease the complexity, we sorted our AVLTree based on
   total reach value, since in this function we are looking for the biggest total reach's node, we need to iterate
   number of right nodes from the root node times to right which is log(n).

         Best Case: O(1), if there is only one node.
   */
   printf("\nMaximum Total Reach:\n");
   printf("====================\n");

   AVLTree traversal = t;

   while(traversal->right != NULL) // coming to the rightest node in the tree
      traversal = traversal->right;

   printf("Post Id: %s\n",traversal->postId);
   printf("Permalink: %s\n",traversal->permaLink);
   printf("Type: %s\n",traversal->type);
   printf("Posted: %s\n",traversal->posted);
   printf("Total Reach: %d\n",traversal->totalReach);
   printf("Engaged users: %d\n",traversal->engagedUser);

   while(traversal->side){ // if there is existing mostReach in the tree, display it too
      printf("\nPost Id: %s\n",traversal->side->postId);
      printf("Permalink: %s\n",traversal->side->permaLink);
      printf("Type: %s\n",traversal->side->type);
      printf("Posted: %s\n",traversal->side->posted);
      printf("Total Reach: %d\n",traversal->side->totalReach);
      printf("Engaged users: %d\n",traversal->side->engagedUser);
```

```
        traversal = traversal->side;
    }

}

Queue createQueue(){
        Queue q;
        q = (Queue)malloc(sizeof(Queue));
        if (q == NULL){
                printf("Out of space!");
                exit(-1);
        }

        q->size = 0;
        q->front = (struct QueueNode*)malloc(sizeof(struct QueueNode));
        if (q->front == NULL){
                printf("Out of space!");
                exit(-1);
        }

        q->front->next = NULL;
        q->rear = q->front;
        return q;
}

void Enqueue(Queue q, AVLTree x){
        struct QueueNode *temp;
        temp = (struct QueueNode*)malloc(sizeof(struct QueueNode));
        if (temp == NULL){
                printf("Out of space!");
```

```
                exit(-1);
        }

        temp->next = NULL;
        temp->val = x;

        q->rear->next = temp;
        q->rear = temp;
        q->size++;
}

AVLTree Dequeue(Queue q){
        AVLTree x;
        struct QueueNode *removal;
        removal = q->front->next;
        x = removal->val;

        q->front->next = removal->next;
        free(removal);
        q->size--;

        if (q->size == 0)
                q->rear = q->front;

        return x;
}

void mostEngaged(AVLTree t, int max){
        /*      Since this function has only two while loops, and it traverses "number of the nodes in the AVLTree" times, because
        it does the enqueue and dequeue operation. It will enqueue the current node's left, right, and side, and will dequeue.
```

So number of the nodes in the AVLTree. If we say n = number of the nodes in the AVLTree, the complexity is O(n). Because we have two while loops,

O(n) + O(n) = O(n).

I couldn't find another way to decrase this complexity, since this function searches the engagedUser member, it can be any node

in the tree. So we need at least to iterate every node in the tree, and I did that using enqueue and dequeue operations.

Best Case: O(1), if there is only one node.

*/

```
char postId[35], permaLink[75], type[10], posted[35];
int totalReach, engagedUser;

AVLTree temp = t;
Queue q;
q = createQueue();

while(temp != NULL){ // finding the mostEngaged value.
        if (max < temp->engagedUser) // keeping the mostEngaged node.
                max = temp->engagedUser;

        if (temp->side)
                Enqueue(q, temp->side);
        if (temp->left)
                Enqueue(q, temp->left);
        if (temp->right);
                Enqueue(q, temp->right);

        temp = Dequeue(q);
}
```

```
if(t==NULL){
        printf("\nt is null\n");
        exit(-1);
}

temp = t;



while(temp != NULL){ // finding that value and displaying
        if (max == temp->engagedUser){ // keeping the mostEngaged node.
                printf("\nMaximum Engaged Users:\n");
                printf("====================\n");
                printf("Post Id: %s\n",temp->postId);
                printf("Permalink: %s\n",temp->permaLink);
                printf("Type: %s\n",temp->type);
                printf("Posted: %s\n",temp->posted);
                printf("Total Reach: %d\n",temp->totalReach);
                printf("Engaged users: %d\n",temp->engagedUser);


        }


        if (temp->side)
                Enqueue(q, temp->side);
        if (temp->left)
                Enqueue(q, temp->left);
        if (temp->right);
                Enqueue(q, temp->right);



        temp = Dequeue(q);
}
```

```
}
}


//avltree.h
struct AVLTreeNode{
        char postId[35], permaLink[75], type[10], posted[35];
        int totalReach, engagedUser;
        int height;
        struct AVLTreeNode *left;
        struct AVLTreeNode *right;
        struct AVLTreeNode *side;
};
typedef struct AVLTreeNode *AVLTree;

int HeightAvlTree(AVLTree);
AVLTree RightRotation(AVLTree);
AVLTree LeftRotation(AVLTree);
AVLTree DoubleRightRotation(AVLTree);
AVLTree DoubleLeftRotation(AVLTree);
int MaxOfTwo(int, int);

//queue.h
struct QueueNode{
        AVLTree val;
        struct QueueNode *next;
};

struct QueueRecord{
        struct QueueNode *front;
        struct QueueNode *rear;
```

```
        int size;
};
typedef struct QueueRecord *Queue;

Queue createQueue();
void Enqueue(Queue, AVLTree);
AVLTree Dequeue(Queue);
```