PRACTICA 2

Sistemas Operativos 2

Pedro Pizarro
David Martín

Indexación del árbol por cadenas de caracteres

Ya que hemos modificado el árbol para que las keys sean cadenas de caracteres:

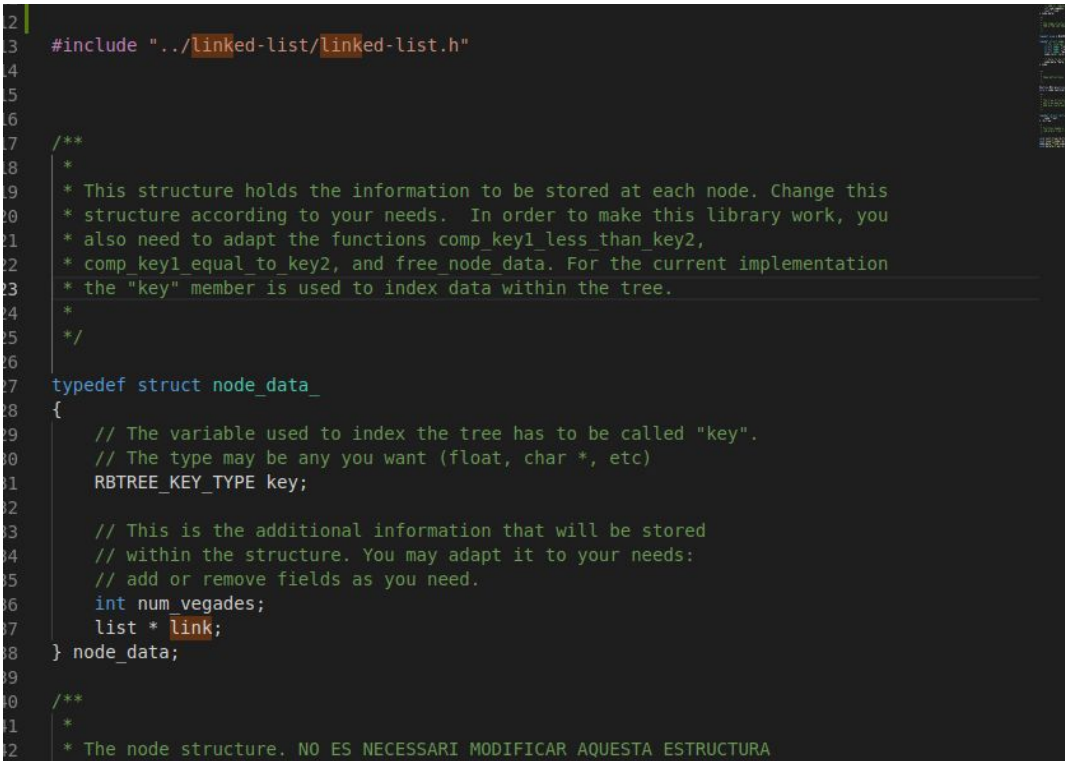
```
9
10  #define RBTREE_KEY_TYPE char*
```

Hemos modificado los “compares” para que pueda indexar con cadenas de caracteres con la función strcmp que si devuelve < 0 key1 es mayor que key 2, al revés si es >0 y si es = 0 las keys son iguales.

```
static int compare_key1_less_than_key2(RBTREE_KEY_TYPE key1, RBTREE_KEY_TYPE key2)
{
    int rc;

    rc = 0;

    if (strcmp(key1,key2) < 0)
        rc = 1;

    return rc;
}

/**
 *
 * Compares if key1 is equal to key2. Should return 1 (true) if condition
 * is satisfied, 0 (false) otherwise.
 *
 */

static int compare_key1_equal_to_key2(RBTREE_KEY_TYPE key1, RBTREE_KEY_TYPE key2)
{
    int rc;

    rc = 0;

    if (strcmp(key1,key2)==0){
        rc = 1;
    }

    return rc;
}

/**
```

Lista enlazada en cada nodo del árbol

También habíamos modificado la estructura, de forma que cada nodo tuviese una lista indexada:



```c
#include "../linked-list/linked-list.h"



/**
 *
 * This structure holds the information to be stored at each node. Change this
 * structure according to your needs.  In order to make this library work, you
 * also need to adapt the functions comp_key1_less_than_key2,
 * comp_key1_equal_to_key2, and free_node_data. For the current implementation
 * the "key" member is used to index data within the tree.
 *
 */

typedef struct node_data_
{
    // The variable used to index the tree has to be called "key".
    // The type may be any you want (float, char *, etc)
    RBTREE_KEY_TYPE key;

    // This is the additional information that will be stored
    // within the structure. You may adapt it to your needs:
    // add or remove fields as you need.
    int num_vegades;
    list * link;
} node_data;

/**
 *
 * The node structure. NO ES NECESSARI MODIFICAR AQUESTA ESTRUCTURA
```

De esta forma compliriamos la estructura que pide la práctica, en cada nodo tendríamos una lista enlazada, la cual también tendría una key, esta sería una cadena de caracteres, como en el árbol binario, siendo cada una el destino del nodo del árbol proveniente.

Errores

```
// Cogemos la lista del nodo encontrado
lista_origen = n_data->link;

//NO FUNCIONA , no sabemos porque.

// buscamos si el destino esta en la lista
l_data = find_list(lista_origen, destino);

if (l_data != NULL) {
    // Si está sumamos un vuelo y el retraso al total
    l_data->num_times++;
    l_data->retraso = (l_data->retraso* l_data->num_times-1) + retraso/ l_data->num_t

} else {
    // Si no esta en la lista reservamos memoria y lo añadimos
    l_data = malloc(sizeof(list_data));
    l_data->key = a;
    l_data->retraso = retraso;
    l_data->num_times = 1;

    insert_list(lista_origen, l_data);
}
```

Exactamente en la línea “l_data = find_list(lista_origen, destino);” lanza un error de violación de segmento. Por el momento no sabemos porque lo hace. Pero debido a esto no hemos podido hacer correctamente las pruebas ni poder liberar correctamente la memoria.

De todas formas, la idea para solucionar el resto de la práctica era:
Como se puede observar arriba, cada vez que se insertaba un elemento en la lista, si este estaba repetido, se suma al valor del “retraso” el próximo retraso a este destino. Por lo tanto en el momento que tengamos que saber el retraso promedio ya lo teníamos calculado. Solo había que coger de la estructura “retraso”:

```
char * value = argv[3];

//Encontramos el nodo que corresponde al origen que buscamos
n_data = find_node(tree, value);

//cogemos el item de la lista que contiene el retraso
l_item = n_data->link->first;

//mientras no sea null
while (l_item != NULL) {
    l_data = l_item->data;
    //Imprimimos el destino y el retraso previamente calculado en la inserción.
    printf("Key %s appears %d times\n", l_data->key, l_data->retraso);
    l_item = l_item->next;
}
```

Y respecto a mostrar el origen con más destinos, en vez de recorrer el árbol se me ha ocurrido que ya que tenemos todos los orígenes en el csv que insertamos, deberíamos darle uso:

Se vuelve a leer el documento, y por cada uno de los orígenes se comprueba con la variable num_items de cada lista de destinos en un contador temporal, si al final este contador ha sido el maximo, este es un origen con más destinos, simplemente queda enseñarlo por pantalla.

```
SIZE = atoi(str);
char *result = malloc(sizeof(char)*4);
int var,max= 0;

for(int i=0; i < SIZE; i++){
    if( fgets (str, 100, fp)!=NULL )
    {
        puts(str);
        str[strlen(str) -1] = '\0';

        //Todos han sido añadidos anteriormente
        n_data = find_node(tree, str);

        var = n_data->link->num_items;

        //si el valor obtenido es superior al maximo encontrado
        if (var>max){
            //Sera de medida 4 (los 3 caracteres y el \0)
            result = str;
            max = var;
        }

    }
}

printf(" El aeropuerto con mas destinaciones es: %s", result);

fclose(fp);
```