IMPLEMENTATION OF HASH TABLE

Engenharia Informática

Leandro Silva 93446 / Mário Silva 93340

2019/2020

Contribution percentage: 50/50

Index

Introduction

For this project we were asked to make a program that could get information about the words used on a text file and that used a hash table with separate chaining, that grows dynamically and each entry of the table should point to a linked list or an ordered binary tree.

We were also already given by the professor some code to read and get some information from the text file.

So we decided to start by doing our project using linked lists, because we though it would be easier to implement since it doesn't necessarily need any recursive function, although, at the end we realized that doing it with ordered binary trees was a bit simpler if we didn't used the dynamic resize that we will explain why it wasn't that important to make even though we did make it.

Solutions and Used Methods

➢ Separate Chaining

In the method known as separate chaining, each bucket is independent, and has some sort of structure of entries with the same index, structures that are efficient in time and space for these cases are preferred.

▪ Separate chaining with linked lists

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per bucket. For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure, so the worst-case cost is proportional to the number n of entries in the table.

▪ Separate chaining with binary trees

Instead of a list, we can use any other data structure that supports the required operations. For example, by using a self-balancing binary search tree, the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to O (log n) rather than O (n). However, this introduces extra complexity into the implementation, and may cause even worse performance for smaller hash tables, where the time spent inserting into and balancing the tree is greater than the time needed to perform a linear search on all of the elements of a list, therefore we decided to just use a ordered binary tree.

1. Linked List

```c
typedef struct node{
    char word[64];
    long last_pos;
    long count;
    long minDst;
    long maxDst;
    long totalDst;
    struct node *next;
} node;
```

A linked list is a linear data structure that consists of nodes, each node them contain:

• An array of characters that contain the word.

• Five numbers (longs) one that is the last position that the word was found, two for the minimum and maximum distances between the same word and another for the total distance between all the words.

• A pointer to next node in the list.

We have two functions that interact with the node, one to create it and another to update it. To create a new node, we need to first allocate memory and initialize all of its parameters with the value of 0, therefore we use *calloc*.

```c
static node *newLinkedWord(int pos, char *word){
    node *n=(node *)calloc(1,sizeof(node));
    if (n == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    strcpy(n->word,word);
    n->count=1;
    n->last_pos = pos;
    return n;
}
```

```c
void updNode(int pos, node *tmp){
    tmp->count++;
    int dst=pos-tmp->last_pos;
    if(tmp->count == 2)
        tmp->minDst = dst;
    if (dst<tmp->minDst){
        tmp->minDst=dst;
    }
    if (dst>tmp->maxDst){
        tmp->maxDst=dst;
    }
    tmp->totalDst+=dst;
    tmp->last_pos=pos;
}
```

And we do the same for the hash table, which is basically an array of nodes that point to another node or not, making the linked list.

```
hash_table_link = (node **)calloc(hash_size_link,sizeof(node *));
```

Then we calculate the hash value, using a hash function provided by the professor on the classes slides and then we check the node on the position of the index returned by the hash function, if the word is there, we update its parameters, else we create a new node and put it at the beginning of the list by putting the new node pointing to the old node that was there (or to NULL) and place the new node on the hash table.

```
int hashVal=hash_function(fd->word,hash_size_link);
node *tmp=hash_table_link[hashVal];
bool found_flag=false;

while(tmp!=NULL){
    if (strcmp(tmp->word, fd->word)==0) {
        found_flag=true;
        updNode(fd->word_pos, tmp);
        break;
    }
    tmp=tmp->next;
}
if (found_flag==false){
    tmp=newLinkedWord(fd->word_pos, fd->word);
    tmp->next=hash_table_link[hashVal];
    hash_table_link[hashVal]=tmp;
}
```

To do the resize we first do the *calloc* to a new temporary table that is 1.5 bigger than the hash table being used, then we go through the hash table and once we find a node, we save the node that is being pointed by the current node, and we calculate the hash of the current node and put it at the beginning of the temporary table just like we do normally, then we just do this for the rest of the linked list of the old hash table.

After that, we change the size of the hash table using the *realloc* method, which increases the size stored in the memory and we do *memset* to initialize all parameters at 0, then we just copy all content of the temporary table to the actual hash table used and at the end free the memory occupied by the temporary one.

```
void resize_LinkedList(){
    node** new_table=(node **)calloc(hash_size_link*1.5,sizeof(node *));
    for (int i=0;i<hash_size_link;i++){
        if (hash_table_link[i]!=NULL){
            node *tmp=hash_table_link[i];
            while (tmp!=NULL) {
                node *tmp2=tmp->next;
                int new_hash=hash_function(tmp->word,(hash_size_link*1.5));
                tmp->next=new_table[new_hash];
                new_table[new_hash]=tmp;
                tmp=tmp2;
            }
        }
    }
    hash_size_link=hash_size_link*1.5;
    hash_table_link=(node
**)realloc(hash_table_link,hash_size_link*sizeof(node *));
    memset(hash_table_link, 0, hash_size_link*sizeof(node *));
    for (int i=0;i<hash_size_link;i++){
        hash_table_link[i]=new_table[i];
    }
    free(new_table);

}
```

Finally, to print all the content of the hash table we just go through all positions of the hash table and when it isn't NULL, we call this function bellow to print the nodes and go for the next one until one of the nodes points to NULL.

```
static void printLinkedList(node *tmp){
    while (tmp != NULL) {
        printf("%-20s Count: %-8ld FirstP: %-13ld LastP: %-13ld MaxD: %-13ld
MinD: %-13ld TotalD: %-13ld AveD: %-ld\n", tmp->word, tmp->count, tmp->last_pos
- tmp->totalDst, tmp->last_pos, tmp->maxDst, tmp->minDst, tmp->totalDst, tmp-
>totalDst/tmp->count);
        tmp = tmp->next;
    }
}
```

2. Ordered Binary Tree

```c
typedef struct tree_node{
    char word[64];
    long last_pos;
    long count;
    long minDst;
    long maxDst;
    long totalDst;
    struct tree_node *right;
    struct tree_node *left;

} tree_node;
```

A binary tree is a dynamic data structure made by nodes, each of them contain:

• An array of characters that contain the word.

• Five numbers (longs) one that is the last position that the word was found, two for the minimum and maximum distances between the same word and another for the total distance between all the words.

• A pointer to the right and another one to the left that point to the right and left child where, the left child is smaller and the right is bigger than the actual node.

We have two functions that interact with the node, one to create it and another to update it. To create a new node, we need to first allocate memory and initialize all of its parameters with the value of 0, therefore we use *calloc*.

```c
static tree_node *newTreeWord(int pos, char *word) {
    tree_node* t_node = (tree_node *)calloc(1,sizeof(tree_node));
    if (t_node == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    strcpy(t_node->word,word);
    t_node->count=1;
    t_node->last_pos = pos;

    return t_node;
}
```

```c
void updTreenode(int pos, tree_node *tmp){
    tmp->count++;
    int dst=pos-tmp->last_pos;
    if(tmp->count == 2)
        tmp->minDst = dst;
    if (dst<tmp->minDst){
        tmp->minDst=dst;
    }
    if (dst>tmp->maxDst){
        tmp->maxDst=dst;
    }
    tmp->totalDst+=dst;
    tmp->last_pos=pos;
}
```

And we do the same for the hash table, which contains pointers to the node that is the head (or root) of the binary tree.

```
hash_table_tree = (tree_node **)calloc(hash_size_tree,sizeof(tree_node *));
```

Then we calculate the hash value, using a hash function provided by the professor on the classes slides and then we check the node on the position of the index returned by the hash function, if it's NULL we create a new node, else by using a recursive function we go through the ordered binary tree, do the comparisons and insert the node in the right place or updating it if it's present.

```
int hashVal=hash_function(fd2->word,hash_size_tree);

if (hash_table_tree[hashVal]==NULL) {
    hash_table_tree[hashVal]=newTreeWord(fd2->word_pos, fd2->word);
}
else {
    tree_node *head=hash_table_tree[hashVal];
    insert(head, fd2->word_pos, fd2->word);
}
```

```
static tree_node* insert(tree_node* head, int pos, char *word) {
    if (head == NULL) {
         head=newTreeWord(pos, word);
    } else if (strcmp(head->word, word) < 0) {
        head->left=insert(head->left, pos, word);
    } else if (strcmp(head->word, word) > 0) {
        head->right=insert(head->right, pos, word);
    } else {
        updTreenode(pos, head);
    }
    return head;
}
```

To do the resize we first do the *calloc* to a new temporary table that is 1.5 bigger than the hash table being used, then we go through the hash table and use a recursive function to travel through the binary tree of the "old" hash table, we go trhough the binary tree by first all the left nodes, and then the right nodes, after that we make sure to remove the left and right pointer of the node, and then get the hash code of the word, put it in the new table or if there's

already a binary tree there, we use another recursive function to insert it, that will do the comparisons needed and then change the pointer of one node in the binary tree to this new node that was in the previous hash table.

```
static tree_node* insert_Resize(tree_node* head, tree_node* to_put){
    if (head == NULL){
        head=to_put;
    } else if (strcmp(head->word, to_put->word) < 0) {
        head->left=insert_Resize(head->left, to_put);
    } else if (strcmp(head->word, to_put->word) > 0) {
        head->right=insert_Resize(head->right, to_put);
    }
    return head;
}
static void travel_Tree(tree_node** new_table, tree_node* tmp, int hash_size){
    if (tmp==NULL) return;
    travel_Tree(new_table,tmp->left, hash_size);
    travel_Tree(new_table,tmp->right, hash_size);
    int new_hash=hash_function(tmp->word,(hash_size*1.5));
    tree_node* head=new_table[new_hash];
    tmp->left=NULL;
    tmp->right=NULL;

    if (head==NULL){
        new_table[new_hash]=tmp;
    } else {
        insert_Resize(head, tmp);
    }
}
```

```
void resize_OBTree(){
    tree_node** new_table=(tree_node
**)calloc(hash_size_tree*1.5,sizeof(tree_node *));
    for (int i=0;i<hash_size_tree;i++){
        tree_node *head=hash_table_tree[i];
        travel_Tree(new_table, head, hash_size_tree);
    }
    hash_size_tree=hash_size_tree*1.5;
    hash_table_tree=(tree_node
**)realloc(hash_table_tree,hash_size_tree*sizeof(tree_node *));
    memset(hash_table_tree, 0, hash_size_tree*sizeof(tree_node *));
    for (int i=0;i<hash_size_tree;i++){
        hash_table_tree[i]=new_table[i];
    }
    free(new_table);
}
```

Finally, to print all the content of the hash table we just go through all positions of the hash table and when it isn't NULL, we call this recursive function bellow to go through the binary tree first on the left side, and then on the right side, meaning it's by an ascending order.

```
static void printTree(tree_node* head) {
    if (head!=NULL){
        printTree(head->left);
        printf("%-20s Count: %-8ld FirstP: %-13ld LastP: %-13ld MaxD: %-13ld
MinD: %-13ld TotalD: %-13ld AveD: %-ld\n",
        head->word, head->count, head->last_pos - head->totalDst, head-
>last_pos, head->maxDst, head->minDst, head->totalDst, head->totalDst/head-
>count);
        printTree(head->right);
    }
}
```

Results Interpretation

- **Ordered Binary Tree**

  Average time to implement hash table:

  - With dynamic resize when number of words is three times the size of the hash table: **0.085 seconds**.
  - with dynamic resize when number of words is half the size of the hash table: **0.13 seconds**.
  - without dynamic resize**: 0.09 seconds**.

Implementation of a dynamic resize of the hash table with ordered binary trees it isn't that much relevant per say, because the main advantage of using the ordered binary trees is to increase the search efficiency, in that sence, doing a dynamic resize wouldn't be worth it since the trees would only have around 1 to 3 elements average and that isn't helpful in terms of search efficiency, it would only be worth it if we were dealing with a much larger number of words, otherwise it's better to have much larger binary trees because it would reduce by a lot the number of comparisons necessary and increase efficiency.

The downside to not using the dynamic resize is that in terms of execution times, we tested out and it increases just a bit more than using the dynamic resize when the number of words is three times the size of the hash table, because it has to do much more comparisons to insert an element to the right place of the tree.

Therefore, we think the best option to use would be an implementation with ordered binary tree **without** dynamic resize, because execution time is still very low and if we would want to search for words (assuming to search we don't know the hash function used) it would be much faster.

- **Linked List**

  Average time to implement hash table:

  - with dynamic resize when number of words is double of the hash table size: **0.10 seconds**.
  - with dynamic resize when number of words is half of the hash table size: **0.13 seconds**.
  - without dynamic resize: **0.19 seconds**.

For the implementation of a hash table using linked lists, after testing the execution times we quickly realized that without dynamic resize it takes much longer, because it would have to go through the entire linked list to make sure the word is there or not, so insertion time wise we definitely need to implement a dynamic resize function. Now we decided to test resizing when the number of words is double and half the size of the hash table, and noticed its better when the number of words is double, we believe that doing the resize also takes a bit of time to do it since it needs to copy, so by doing it a little bit less, we managed to increase the execution time a bit more.

In terms of search efficiency (assuming to search we don't know the hash function used) we think it is the same using a dynamic resize or not, since it still needs to check the positions and go through the linked list one by one, so it wouldn't really increase or decrease much the time.

Conclusion

With this project we learned a lot more about the complexity of building a hash table.

We had to first analyse the best ways of implementing it and then studying a bit more how they work, then we had to look into efficiency wise, the search and the execution times, also we had to take in consideration the complexity of the memory used to make sure we only use just as much as necessary so we studied the dynamic resizing and its efficiency.

Also understood more about which type of data structure to use and what are the best scenarios each one should be applied.

References

https://www.geeksforgeeks.org/data-structures/linked-list/

https://www.geeksforgeeks.org/binary-search-tree-data-structure/

https://www.geeksforgeeks.org/hashing-data-structure/

https://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o

https://stackoverflow.com/questions/36436599/inserting-binary-tree-node

Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>

static double elapsed_time(void)
{
   static struct timespec last_time,current_time;

   last_time = current_time;

if(clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&current_
time) != 0)
      return -1.0; // clock_gettime() failed!!!
   return          ((double)current_time.tv_sec -
(double)last_time.tv_sec)
            + 1.0e-9 * ((double)current_time.tv_nsec -
(double)last_time.tv_nsec);
}

//////////////////HASH-FUNCTION//////////////////

unsigned int hash_function(const char *str, unsigned int s)
{
   unsigned int h;
   for (h = 0u; *str != '\0'; str++)
      h = 157u * h + (0xFFu & (unsigned int)*str); //
arithmetic overflow may occur here (just ignore it!)
   return h % s;                          // due to the unsigned
int data type, it is guaranteed that 0 <= h % s < s
}


/////////////////ORDERED-BINARY-TREE///////////////

typedef struct tree_node{
   char word[64];
   long last_pos;
   long count;
   long minDst;
   long maxDst;
   long totalDst;
   struct tree_node *right;
   struct tree_node *left;
} tree_node;

static tree_node *newTreeWord(int pos, char *word) {
   tree_node* t_node = (tree_node
*)calloc(1,sizeof(tree_node));
   if (t_node == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
   }
   strcpy(t_node->word,word);
   t_node->count=1;
   t_node->last_pos = pos;
   return t_node;
}

void updTreenode(int pos, tree_node *tmp){
   tmp->count++;
   int dst=pos-tmp->last_pos;
   if(tmp->count == 2)
      tmp->minDst = dst;
   if (dst<tmp->minDst){
      tmp->minDst=dst;
   }
   if (dst>tmp->maxDst){
      tmp->maxDst=dst;
   }
   tmp->totalDst+=dst;
   tmp->last_pos=pos;
}

static tree_node* insert(tree_node* head, int pos, char
*word) {
   if (head == NULL) {
       head=newTreeWord(pos, word);
   } else if (strcmp(head->word, word) < 0) {
      head->left=insert(head->left, pos, word);
   } else if (strcmp(head->word, word) > 0) {
      head->right=insert(head->right, pos, word);
   } else {
      updTreenode(pos, head);
   }
   return head;
}

static tree_node* insert_Resize(tree_node* head,
tree_node* to_put){
   if (head == NULL){
      head=to_put;
   } else if (strcmp(head->word, to_put->word) < 0) {
      head->left=insert_Resize(head->left, to_put);
   } else if (strcmp(head->word, to_put->word) > 0) {
      head->right=insert_Resize(head->right, to_put);
   }
   return head;
}

static void travel_Tree(tree_node** new_table,
tree_node* tmp, int hash_size){
   if (tmp==NULL) return;
   travel_Tree(new_table,tmp->left, hash_size);
   travel_Tree(new_table,tmp->right, hash_size);
```

```
   int new_hash=hash_function(tmp->word,(hash_size*1.5));
   tree_node* head=new_table[new_hash];
   tmp->left=NULL;
   tmp->right=NULL;

   if (head==NULL){
      new_table[new_hash]=tmp;
   } else {
      insert_Resize(head, tmp);
   }
}

   static int count2=0;

static void printTree(tree_node* head) {
   if (head!=NULL){
      printTree(head->left);
      printf("%-20s Count: %-8ld FirstP: %-13ld LastP: %-13ld MaxD: %-13ld MinD: %-13ld TotalD: %-13ld AveD: %-ld\n",
      head->word, head->count, head->last_pos - head->totalDst, head->last_pos, head->maxDst, head->minDst, head->totalDst, head->totalDst/head->count);
      count2++;
      printTree(head->right);
   }
}

//////////////////LINKED-LIST//////////////////

typedef struct node{
   char word[64];
   long last_pos;
   long count;
   long minDst;
   long maxDst;
   long totalDst;
   struct node *next;
} node;

static node *newLinkedWord(int pos, char *word){
   node *n=(node *)malloc(sizeof(node));
   memset(n, 0,sizeof(node));
   if (n == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
   }
   strcpy(n->word,word);
   n->count=1;
   n->last_pos = pos;
   return n;
}

void updNode(int pos, node *tmp){
   tmp->count++;
   int dst=pos-tmp->last_pos;
   if(tmp->count == 2)
      tmp->minDst = dst;
   if (dst<tmp->minDst){
      tmp->minDst=dst;
   }
   if (dst>tmp->maxDst){
      tmp->maxDst=dst;
   }
   tmp->totalDst+=dst;
   tmp->last_pos=pos;
}
static int count1=0;
static void printLinkedList(node *tmp){
   while (tmp != NULL) {
      count1++;
      printf("%-20s Count: %-8ld FirstP: %-13ld LastP: %-13ld MaxD: %-13ld MinD: %-13ld TotalD: %-13ld AveD: %-ld\n",
      tmp->word, tmp->count, tmp->last_pos - tmp->totalDst, tmp->last_pos, tmp->maxDst, tmp->minDst, tmp->totalDst, tmp->totalDst/tmp->count);
      tmp = tmp->next;
   }
   return;
}


//////////////////READ-FILE//////////////////

typedef struct file_data{
   // public data
   long word_pos;    // zero-based
   long word_num;    // zero-based
   char word[64];
   // private data
   FILE *fp;
   long current_pos; // zero-based
} file_data;

int isalpha(int c);
int tolower(int c);

int open_text_file(char *file_name,file_data *fd){
   fd->fp = fopen(file_name,"rb");

   if(fd->fp == NULL)
      return -1;
   fd->word_pos = -1;
   fd->word_num = -1;
   fd->word[0] ='\0';
   fd->current_pos = -1;
   return 0;
}

void close_text_file(file_data *fd){
   fclose(fd->fp);
```

```
    fd->fp = NULL;
}

int read_word(file_data *fd){
    int i,c;
    // skip white spaces
    do{
        c = fgetc(fd->fp);
        if(c == EOF)
            return -1;
        fd->current_pos++;
    }while(!isalpha(c));
    // record word
    fd->word_pos = fd->current_pos;
    fd->word_num++;
    fd->word[0] = tolower(c);
    for(i = 1; i < (int)sizeof(fd->word) - 1; i++){
        c = fgetc(fd->fp);
        if(c == EOF)
            break;  // end of file
        fd->current_pos++;
        if(!isalpha(c))
            break;  // terminate word
        fd->word[i] = tolower(c);
    }
    fd->word[i] ='\0';
    return 0;
}

/////////////////GLOBAL-VARIABLES//////////////////

static node** hash_table_link;
static int hash_size_link = 1000u;

static tree_node** hash_table_tree;
static int hash_size_tree = 1000u;


/////////RESIZE-HASH-TABLE-WITH-LINKED-LISTS////////

void resize_LinkedList(){
    node** new_table=(node
**)calloc(hash_size_link*1.5,sizeof(node *));
    for (int i=0;i<hash_size_link;i++){
        if (hash_table_link[i]!=NULL){
            node *tmp=hash_table_link[i];
            while (tmp!=NULL) {
                node *tmp2=tmp->next;
                int new_hash=hash_function(tmp-
>word,(hash_size_link*1.5));
                tmp->next=new_table[new_hash];
                new_table[new_hash]=tmp;
                tmp=tmp2;
            }
        }
    }
    hash_size_link=hash_size_link*1.5;
    hash_table_link=(node
**)realloc(hash_table_link,hash_size_link*sizeof(node *));
    memset(hash_table_link, 0, hash_size_link*sizeof(node
*));
    for (int i=0;i<hash_size_link;i++){
        hash_table_link[i]=new_table[i];
    }
    free(new_table);
}

/////RESIZE-HASH-TABLE-WITH-ORDERED-BINARY-
TREES////

void resize_OBTree(){
    tree_node** new_table=(tree_node
**)calloc(hash_size_tree*1.5,sizeof(tree_node *));
    for (int i=0;i<hash_size_tree;i++){
        tree_node *head=hash_table_tree[i];

        travel_Tree(new_table, head, hash_size_tree);
    }
    hash_size_tree=hash_size_tree*1.5;
    hash_table_tree=(tree_node
**)realloc(hash_table_tree,hash_size_tree*sizeof(tree_no
de *));
    memset(hash_table_tree, 0,
hash_size_tree*sizeof(tree_node *));
    for (int i=0;i<hash_size_tree;i++){
        hash_table_tree[i]=new_table[i];
    }
    free(new_table);
}

//////////////////MAIN-FUNCTION//////////////////

int main(int argc,char **argv) {
    (void)elapsed_time();

//////////Hash-Table-With-Linked-Lists//////////////

    file_data *fd = (file_data *)malloc(sizeof(file_data));
    memset(fd, 0, sizeof(file_data));
    if (open_text_file("SherlockHolmes.txt", fd) == -1) {
        perror("Ficheiro inexistente");
        exit(1);
    }

    hash_table_link = (node
**)calloc(hash_size_link,sizeof(node *));

    while(read_word(fd) == 0){
        if(fd->word_num > hash_size_link*2){
            // Code to See How Hash Grows Dinnamicly
            printf("Size Before Resizing: %d\n", hash_size_link);
            resize_LinkedList();
            printf("Size After Resizing: %d\n", hash_size_link);
        }
```

```
    int hashVal=hash_function(fd->word,hash_size_link);
    node *tmp=hash_table_link[hashVal];
    bool found_flag=false;

    while(tmp!=NULL){
      if (strcmp(tmp->word, fd->word)==0) {
        found_flag=true;
        updNode(fd->word_pos, tmp);
        break;
      }
      tmp=tmp->next;
    }
    if (found_flag==false){
      tmp=newLinkedWord(fd->word_pos, fd->word);
      tmp->next=hash_table_link[hashVal];
      hash_table_link[hashVal]=tmp;
    }
  }

  // Code to Print Hash Table With Linked Lists
  for (int i = 0; i < hash_size_link; i++) {
      node *tmp = hash_table_link[i];
    if (tmp != NULL) {
      printf("New Linked List:\n");
      printLinkedList(tmp);
      printf("\n");
    }
  }
  double cpu_time1 = elapsed_time();


////////Hash-Table-With-Ordered-Binary-Trees////////
  (void)elapsed_time();
  file_data *fd2 = (file_data *)malloc(sizeof(file_data));
  memset(fd2, 0, sizeof(file_data));
  if (open_text_file("SherlockHolmes.txt", fd2) == -1) {
    perror("Ficheiro inexistente");
    exit(1);
  }

  hash_table_tree = (tree_node
**)calloc(hash_size_tree,sizeof(tree_node *));
  while(read_word(fd2) == 0){
    if(fd2->word_num > hash_size_tree*3){
      printf("Size Before Resizing: %d\n",
hash_size_tree);
      resize_OBTree();
      printf("Size After Resizing: %d\n", hash_size_tree);
    }

    int hashVal=hash_function(fd2-
>word,hash_size_tree);

    if (hash_table_tree[hashVal]==NULL){
        hash_table_tree[hashVal]=newTreeWord(fd2-
>word_pos, fd2->word);
    }
    else {
        tree_node *head=hash_table_tree[hashVal];
        insert(head, fd2->word_pos, fd2->word);
    }

  }
  // Code to Print Hash Table With Ordered Binary Trees
  for (int i = 0; i < hash_size_tree; i++) {
    tree_node *head = hash_table_tree[i];
    if (head!=NULL){
      printf("New Ordered Binary Tree:\n");
      printTree(head);
      printf("\n");
    }
  }
  double cpu_time2 = elapsed_time();

  printf("Time to implement hash table using linked
lists: %f\n", cpu_time1);
  printf("Time to implement hash table using ordered
binary trees: %f\n", cpu_time2);
  printf("Number of different elements using linked
lists: %d\n", count1);
  printf("Number of different elements using ordered
binary trees: %d\n", count2);

  return 0;
}
```