

**Algoritmos e Estruturas de Dados**

Professor Tomás Oliveira e Silva

Professor Pedro Lavrador

## 

## 

**Multi-Ordered Trees**

Ana Paradinha 102491 33,3%

Paulo Pinto 103234 33,3%

Tiago Carvalho 104142 33,3%

Índice

[1 Introdução 3](#_Toc94820290)

[2 Metodologia 4](#_Toc94820291)

[2.1 Função *tree\_insert* 4](#_Toc94820292)

[2.2 Função *find* 5](#_Toc94820293)

[2.3 Função *tree\_depth* 5](#_Toc94820294)

[2.3 Função *list* 6](#_Toc94820295)

[3 Resultados 7](#_Toc94820296)

[3.1 Tree Creation Time 7](#_Toc94820297)

[3.2 Tree Search Time 8](#_Toc94820298)

[3.3 Tree Depth 9](#_Toc94820299)

[3.4 Tree Creation Time Histogram 10](#_Toc94820300)

[3.5 Tree Depth Histogram 11](#_Toc94820301)

[3.6 Search Time Histogram 12](#_Toc94820302)

[4 Apêndice 16](#_Toc94820303)

[5 Conclusão 19](#_Toc94820304)

[6 Bibliografia 20](#_Toc94820305)

# 1 Introdução

No âmbito da unidade curricular Algoritmos e Estruturas de Dados foi-nos proposto o desenvolvimento do *script multi\_ordered\_tree.c* com recurso à linguagem *C* que permite guardar e processar registos de dados pessoais e acessá-los através de um indíce.

Assim sendo, o objetivo deste projeto prático é trabalhar com dados gerados aleatoriamente, através de uma *seed*, e organizá-los em árvores binárias. É através da criação destas árvores que podemos obter valores como o seu tempo de criação, o tempo que demora a encontrar todos os dados nas árvores e a profundidade, *depth*, de cada árvore.

Foi nos ainda pedido que tratássemos e apresentássemos os valores obtidos em gráficos feitos em *Matlab*, de forma a analisarmos a eficácia deste método de ordenação para diferentes valores iniciais de *seed* e número de pessoas.

Com efeito, no presente relatório descrevemos as metodologias utilizadas para chegar a uma solução, assim como os resultados a que chegámos.

# 2 Metodologia

Com a metodologia aplicada para resolver o problema conseguimos não só completar todas as funções propostas pelo professor, como também adicionar os extras que eram sugeridos, tais como um quarto índice, contendo o número de segurança social de cada pessoa e alterar a função list para se conseguir listar apenas as pessoas que correspondam a uma dada expressão regular. Tendo isto em conta, o código que se segue já está com essas alterações sobrepostas.

## 2.1 Função *tree\_insert*

De forma a criar as árvores binárias ordenadas, existe, na função *main*, este excerto de código que, para cada índice, insere dada uma das pessoas criadas em cada uma das árvores binárias a que o índice corresponde. Para tal é passada na função *tree\_insert* a *root* à qual o índice corresponde, um ponteiro com uma pessoa do *array* de *persons* e o índice em que se está a trabalhar no momento. É a partir desses 3 dados passados na invocação da função que, já dentro da mesma, se processa toda a organização das *nodes* em questão.

Primeiramente, verificamos, com recurso a um *if*, se o *link* se encontra inicializado. Caso não esteja, preenche-se esse lugar com *person*, caso contrário, recorre-se à função *compare\_tree\_nodes* (disponibilizada pelo professor) para comparar entre o *node link* e o *node person*. Em caso de o *person* ser inferior ao *link*, invoca-se a função *tree\_insert* novamente passando-lhe o *tree\_node\_t* correspondente ao índice do *array left* do *link*, o *person* e o índice; caso seja superior, invoca-se a função *tree\_insert*, mas passando o *tree\_node\_t* correspondente ao índice do *array right* do *link*, o *person* e o índice.

## 2.2 Função *find*

A função *find* tem como objetivo encontrar uma dada *person* utilizando um *índice* e a árvore binária a que esse índice corresponde.

A estrutura da função *find* é muito parecida com a da função *tree\_insert*. A única diferença é que primeiramente verificamos através de um *if* se o *link* se encontra inicializado ou se o *link* é a *person* que procuramos, utilizando a função *compare\_tree\_nodes* e igualando o seu *return* a 0. Caso uma destas condições seja verdadeira a função *find* irá retornar o *link*.

Caso o *person* não seja encontrado nesse *link*, então recorremos à função *compare\_tree\_nodes* para saber se devemos procurar no *array left* ou no *array right* do *link*, assim como fizemos para organizar os dados com a função *tree\_insert*.

## 2.3 Função *tree\_depth*

É através da função *tree\_depth* que descobrimos a *depth* que tem cada uma das quatro árvores que criamos. Para tal, invocamos a função *tree\_depth* passando-lhe o índice desejado e a árvore binária ordenada a que esse índice corresponde.

Já dentro da função, para começar, verificamos se a árvore binária passada se encontra inicializada e caso isto não se verifique a função *tree\_depth* retorna o valor 0. Depois utilizamos a recursividade para encontrar navegar entre os *arrays left* e *right* de cada *node* que constitui a árvore e guardamos os seus valores de depth nas variáveis *ld* e *rl*, respetivamente. Depois de todos os retornos de cada *node left* e *node right*, acontece uma comparação entre os valores de *ld* e *rd* e retorna-se o valor que for superior somando-lhe 1.

## 2.3 Função *list*

Como dito anteriormente neste relatório, certos detalhes do código foram elaborados a partir das sugestões de implementações extra do professor. No caso da função *list*, grande parte do código que a constitui foi elaborado de forma a que o utilizador do programa possa passar um argumento adicional corresponde a uma expressão regular com o objetivo de filtrar as *persons* que vão ser listadas. Caso o utilizador pretenda listar todas as *persons* que compõem a árvore binária, basta apenas não passar esse argumento adicional ao chamar o programa. Nesse caso, a expressão regular será “NULL”.

O modo de atuar da função *list* funciona através de duas invocações da própria função *list* mas passando-lhe o *array left* e depois o *array right* do *node link* atual. Assim sendo, a função irá procurar cada uma das *persons* listadas na árvore binária ordenadamente, da menor para a maior, dependendo do índice passado. É desta forma que o código que se designa a printar a informação desejada se encontra entre as duas invocações da função *list*.

Começa-se por verificar qual é o índice em que se está a trabalhar através de uma série de *ifs* e é a partir desses *ifs* que definimos o ponteiro *search* com os dados pessoais (nome, código postal, número de telefone ou número de segurança social). Como o número de telefone e o número de segurança social têm espaços pelo meio, decidimos copiar o ponteiro *search* para um novo *array*, *sus*, a qual serão retirados os espaços e que depois será usada para com os dados do *node* em que estamos.

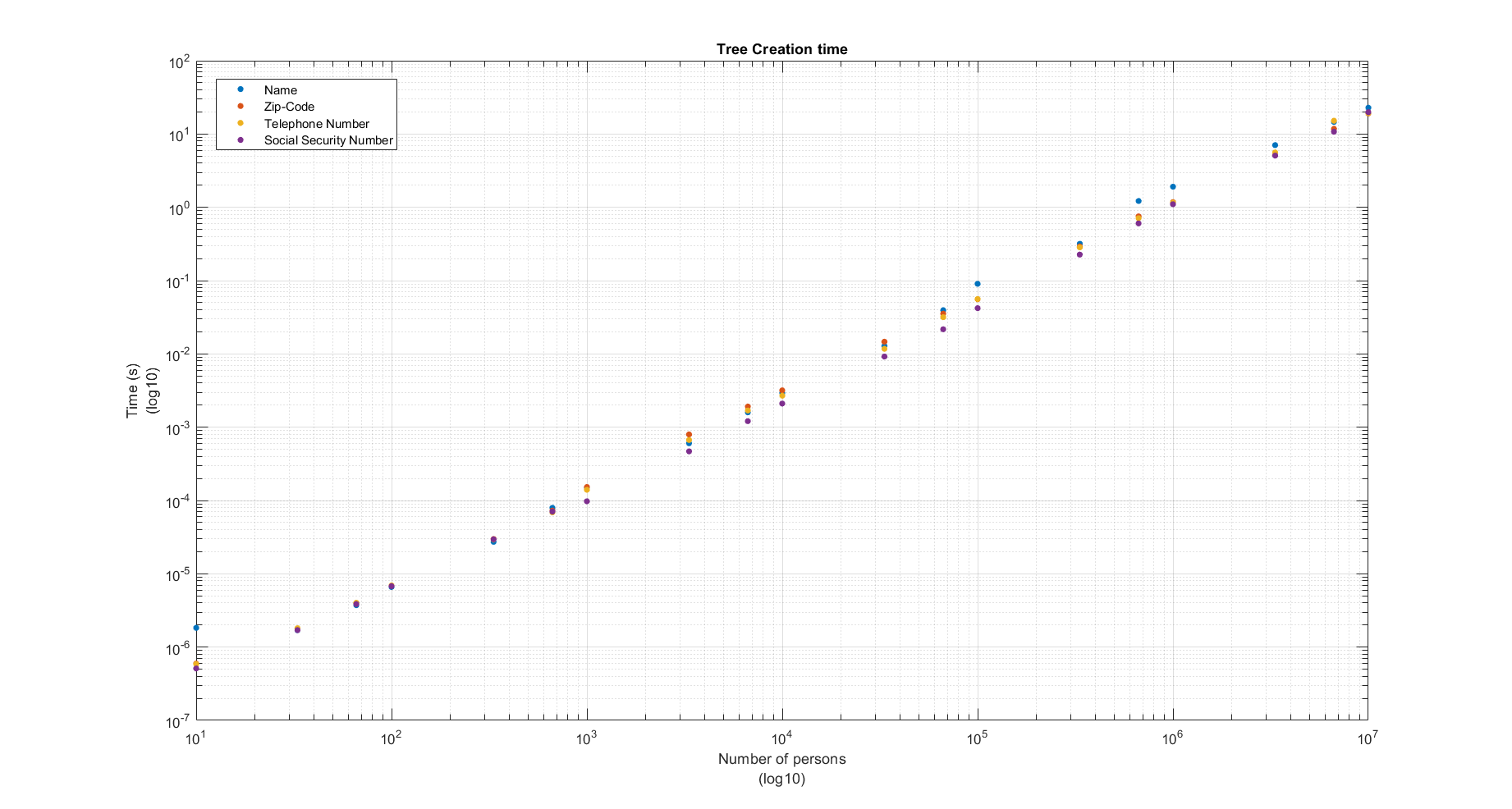
Utilizando a expressão regular passada, comparamos com a *string* “NULL” ou com o *array sus* e caso isto se verifique então os dados da *person* serão listados no terminal.

# 3 Resultados

## 3.1 Tree Creation Time

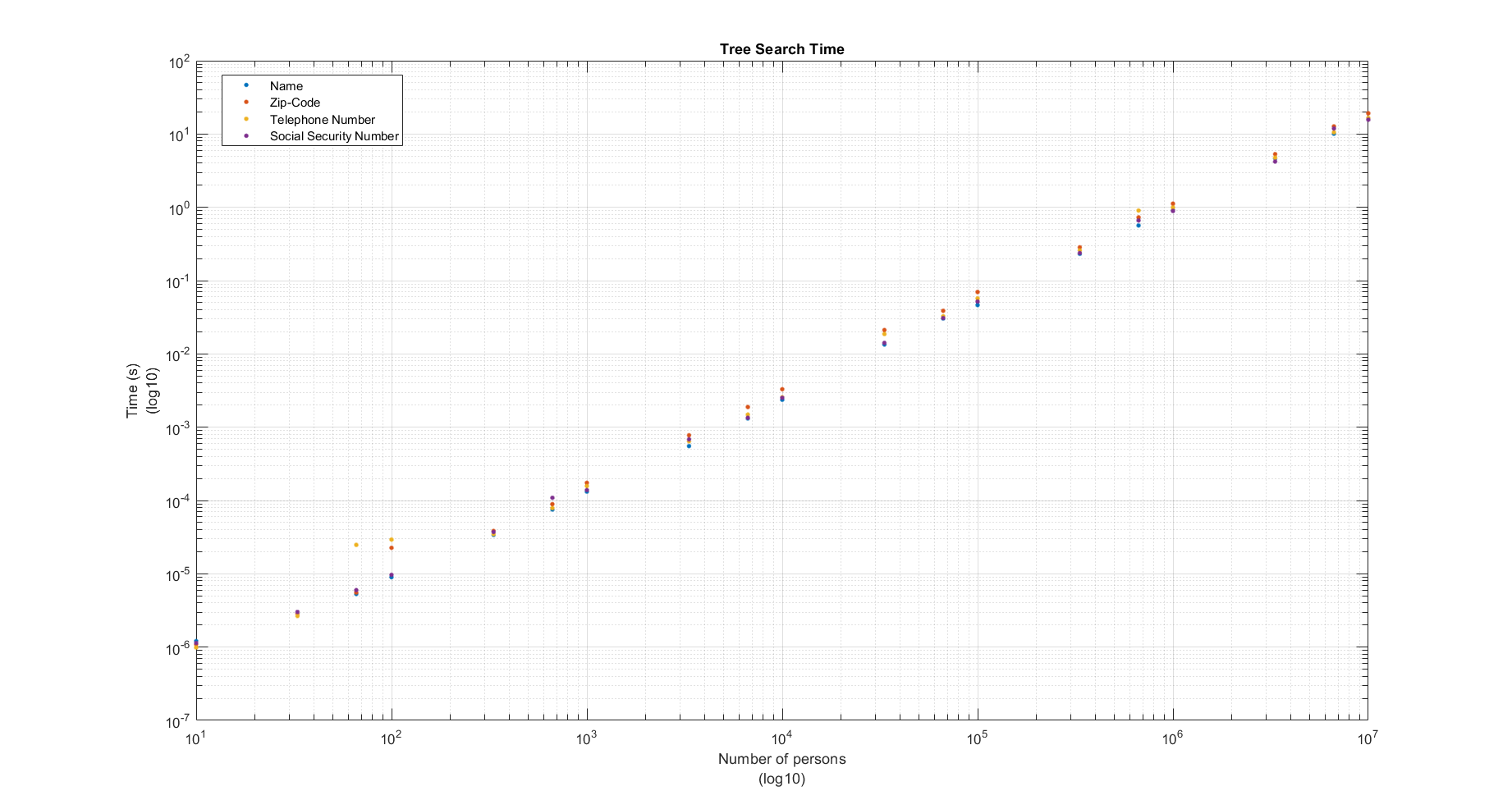
Com o objetivo de analisar os resultados obtidos criámos diversos gráficos que comparam o comportamento do script para cada índice, considerando diferentes critérios.

Nesse sentido, o primeiro gráfico diz respeito à evolução do tempo de criação de cada árvore em função do número de pessoas. Através dele podemos verificar que apesar de haver uma quantidade muito superior de nomes de pessoas, relativamente aos restantes atributos, esta não tem uma grande influência no tempo de criação das respetivas árvores binárias.



## 3.2 Tree Search Time

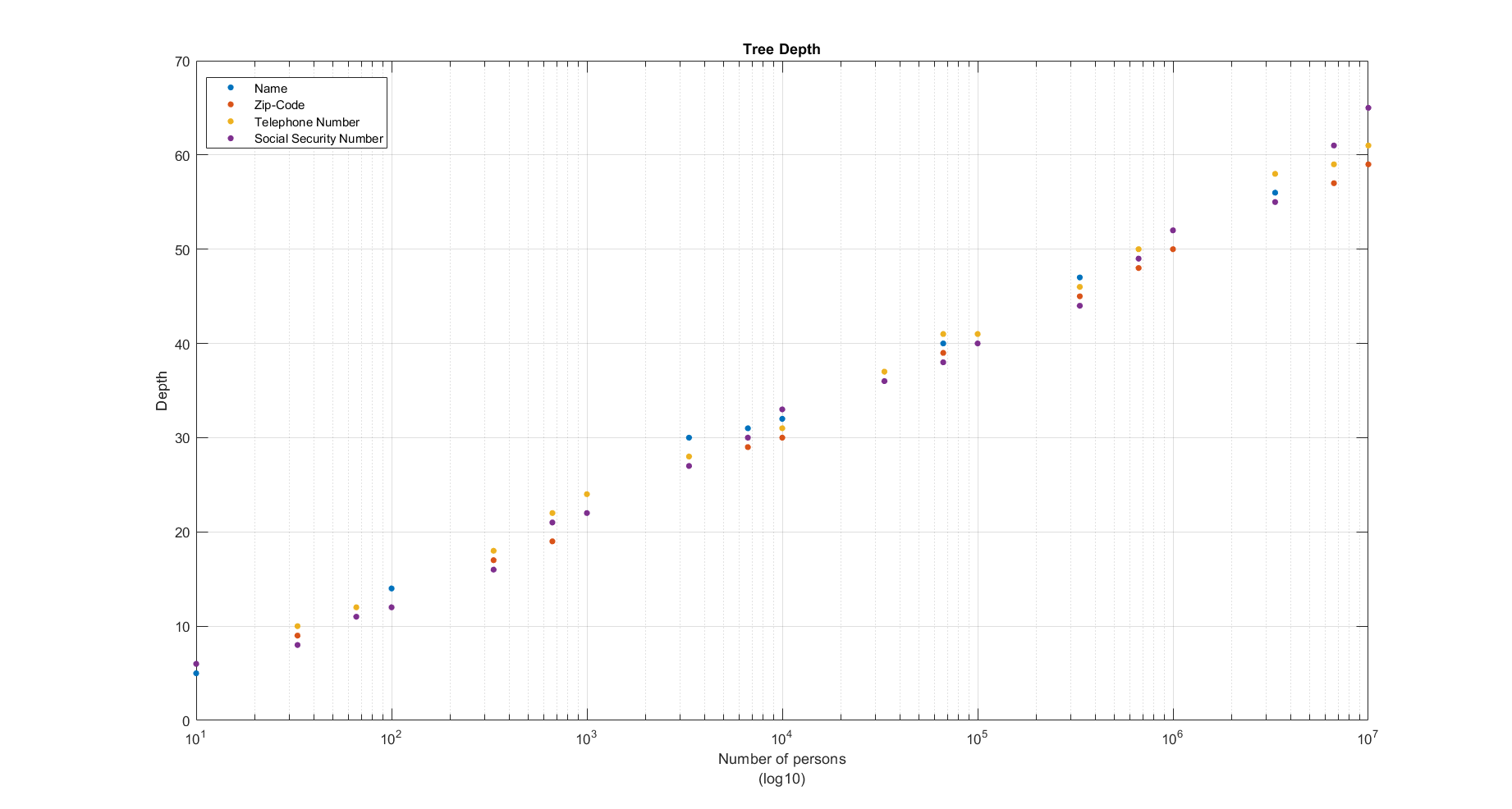
Além disso, o gráfico seguinte, que ilustra a evolução do tempo de procura em cada árvore em função do número de pessoas inseridas, permite-nos chegar à conclusão de que neste aspeto a quantidade de cada atributo também não provoca alterações na eficiência.



Também podemos verificar que a complexidade computacional dos algoritmos de inserção e procura na árvore binária é dada por O(n), uma vez que a evolução temporal de ambos é diretamente proporcional ao aumento do número de pessoas (n).

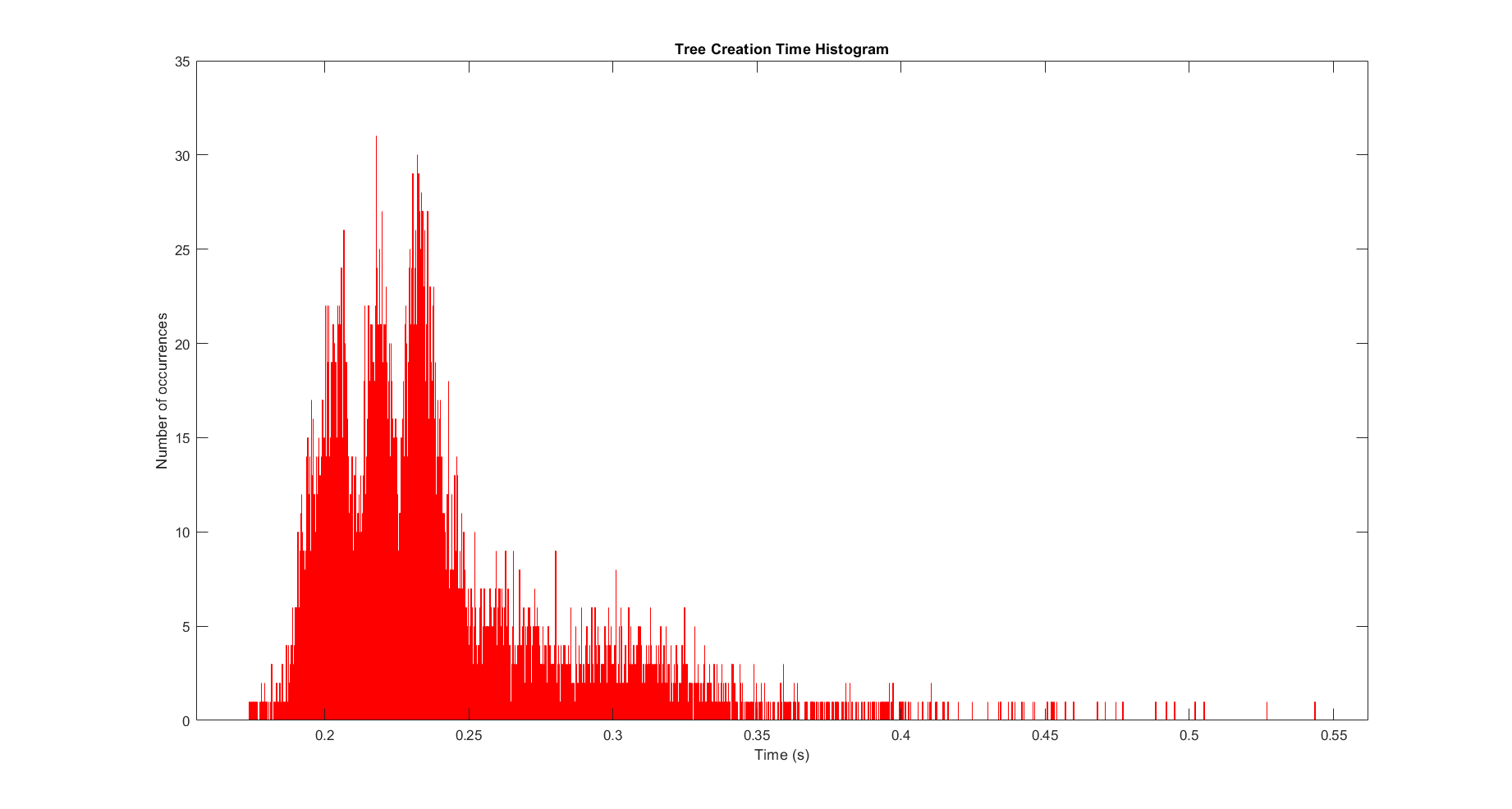
## 3.3 Tree Depth

Relativamente, à profundidade das árvores binárias criadas observamos que não há grandes discrepâncias entre os diferentes índices, para o mesmo número de pessoas adicionadas. Também podemos conferir que o mínimo valor obtido foi de 5, para o índice dos nomes, e o máximo 65, no número da segurança social.



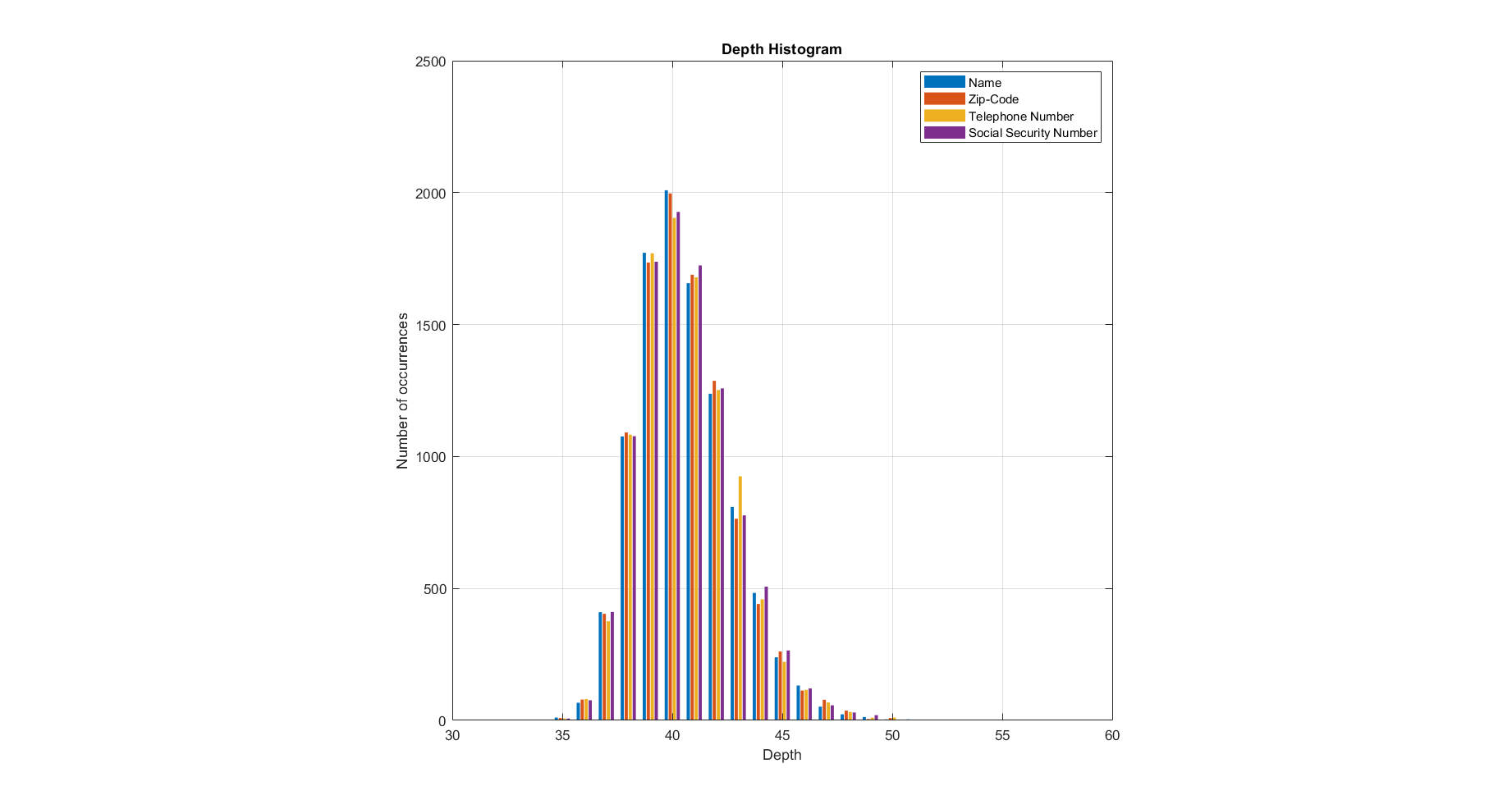
## 3.4 Tree Creation Time Histogram

Quando executamos o *script* para vários números mecanográficos, sempre com 10000 pessoas, verificamos que os tempos de criação mais comuns estão entre 0.2 e 0.25 segundos.



## 3.5 Tree Depth Histogram

Utilizando as mesmas características que no histograma anterior para a execução observamos que a profundidade mais obtida é 40 níveis.



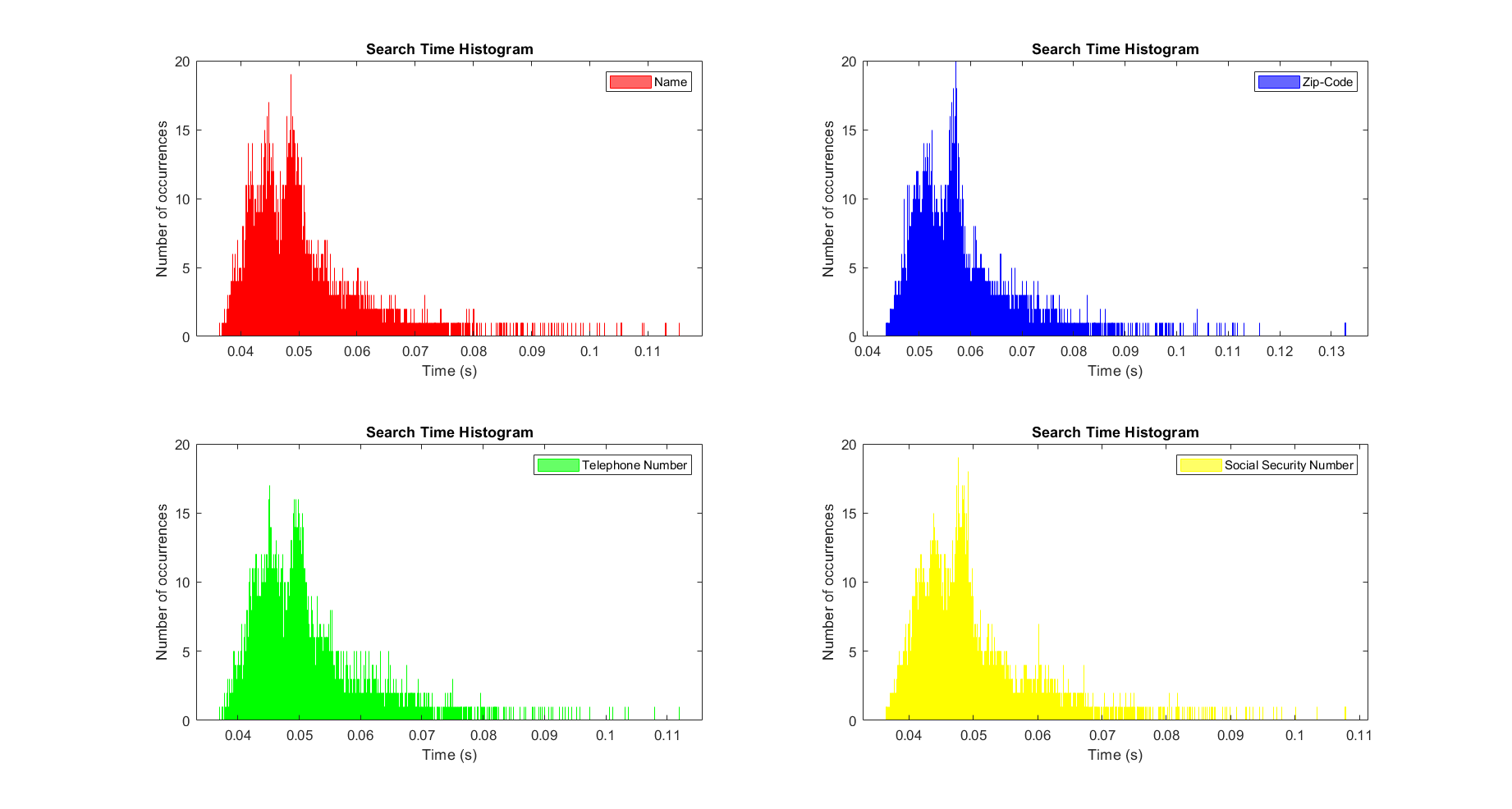
## 

## 

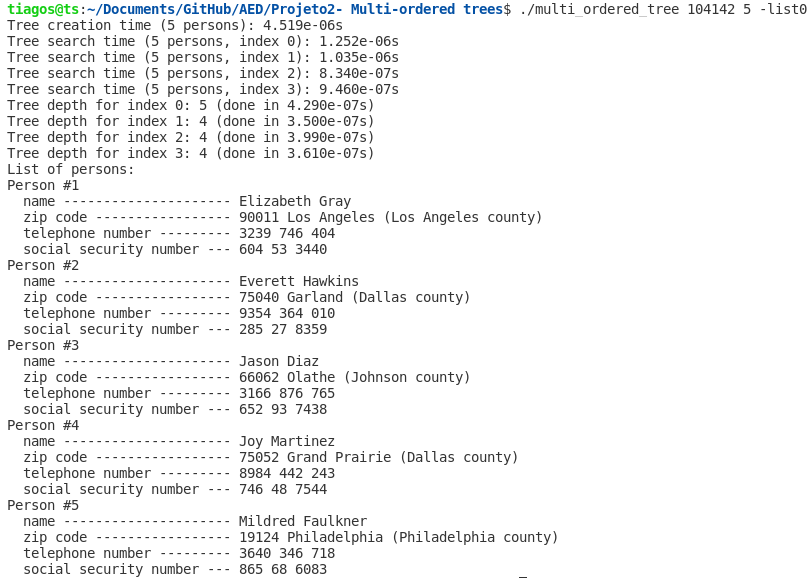
## 

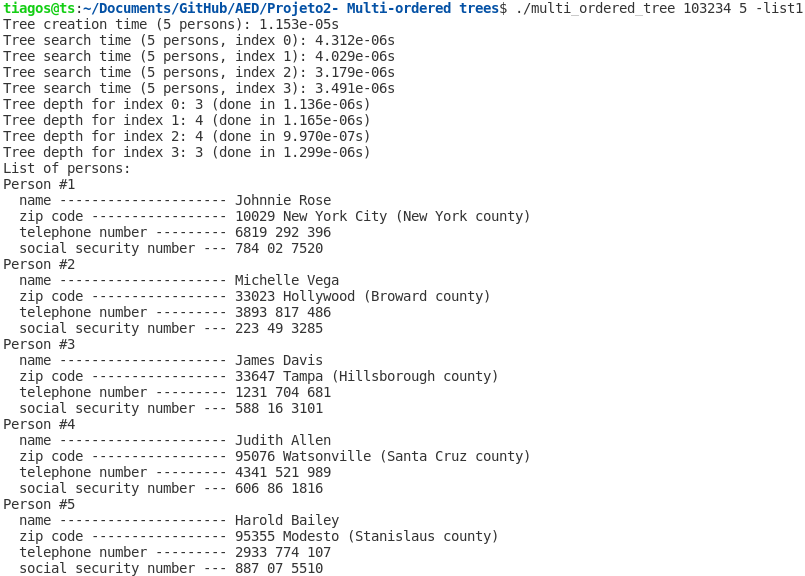
## 3.6 Search Time Histogram

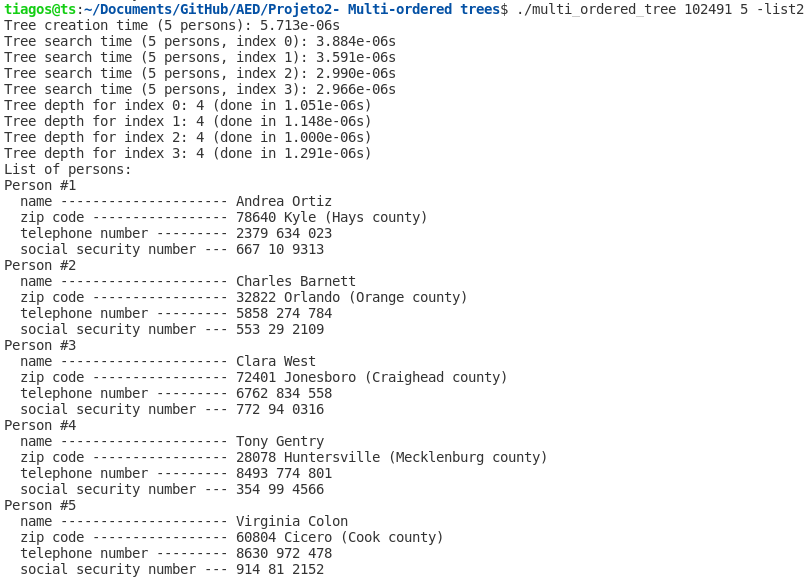
Simultaneamente, a nível dos tempos de procura, verificamos que são muitos semelhantes para todos os índices, rondando os 0.04 e os 0.06 segundos.

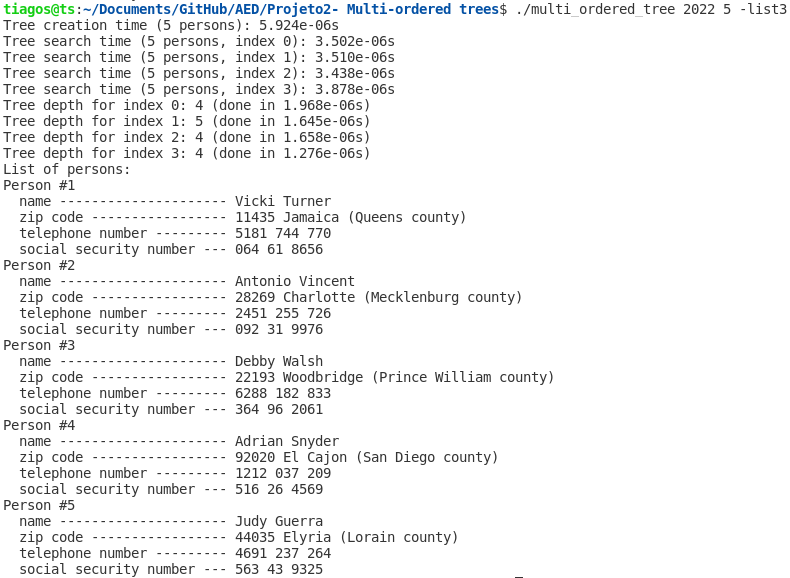


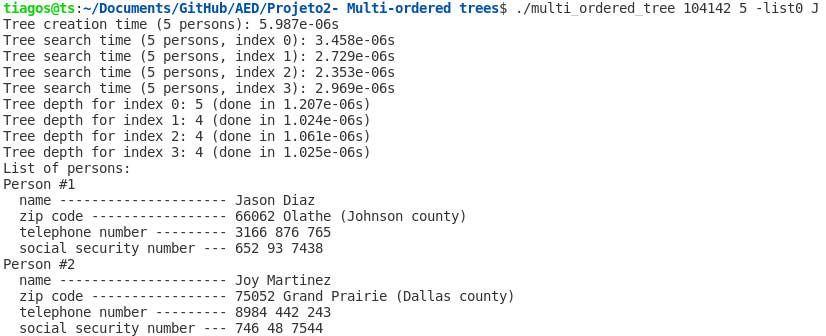
Adicionalmente, executamos o *script* com diferentes argumentos para verificar que todas as opções estavam implementadas corretamente.

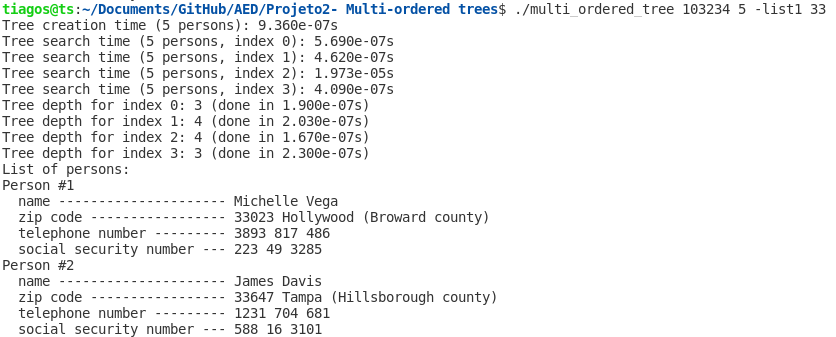


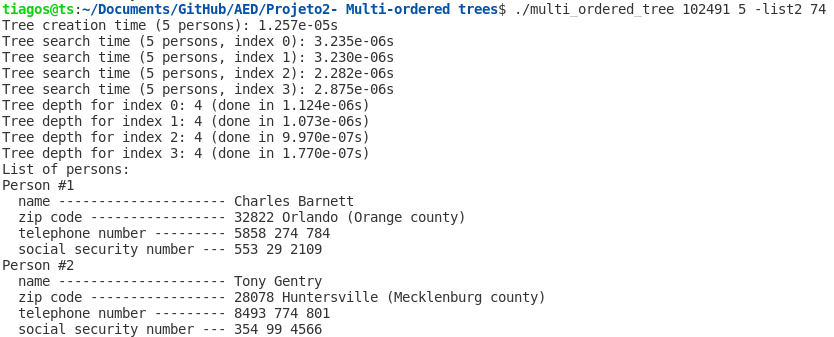


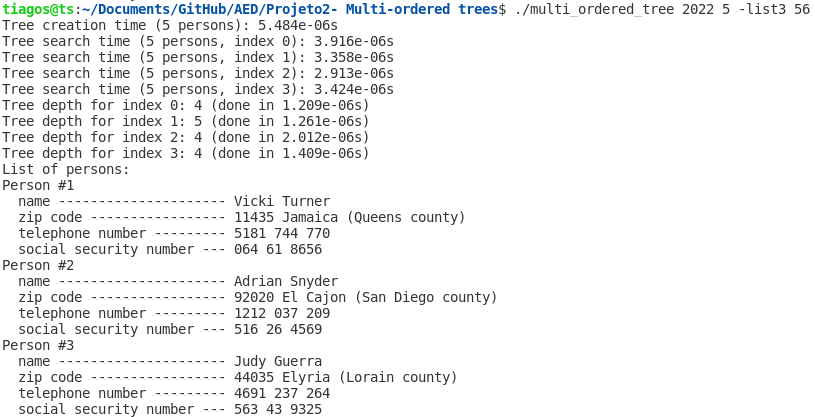












# 4 Apêndice

//

// AED, January 2022

//

// Solution of the second practical assignment (multi-ordered tree)

//

// Place your student numbers and names here

// Ana Raquel Paradinha 102491

// Paulo Pinto 103234

// Tiago Carvalho 104142

//

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "AED\_2021\_A02.h"

// MAX\_NAME\_SIZE

// the custom tree node structure

//

// we want to maintain three ordered trees (using the same nodes!), so we need three left and three right pointers

// so, when inserting a new node we need to do it three times (one for each index), so we will end up with 3 three roots

//

int ctr = 0;

typedef struct tree\_node\_s

{

char name[ MAX\_NAME\_SIZE + 1]; // index 0 data item

char zip\_code[MAX\_ZIP\_CODE\_SIZE + 1]; // index 1 data item

char telephone\_number[MAX\_TELEPHONE\_NUMBER\_SIZE + 1]; // index 2 data item

char social\_security\_number[MAX\_SOCIAL\_SECURITY\_NUMBER + 1]; // index 3 data item

struct tree\_node\_s \*left[4]; // left pointers (one for each index) ---- left means smaller

struct tree\_node\_s \*right[4]; // right pointers (one for each index) --- right means larger

}

tree\_node\_t;

//

// the node comparison function (do not change this)

//

int compare\_tree\_nodes(tree\_node\_t \*node1,tree\_node\_t \*node2,int main\_idx)

{

int i,c;

for(i = 0;i < 3;i++)

{

if(main\_idx == 0)

c = strcmp(node1->name,node2->name);

else if(main\_idx == 1)

c = strcmp(node1->zip\_code,node2->zip\_code);

else if(main\_idx == 2)

c = strcmp(node1->telephone\_number,node2->telephone\_number);

else

c = strcmp(node1->social\_security\_number,node2->social\_security\_number);

if(c != 0)

return c; // different on this index, so return

main\_idx = (main\_idx == 3) ? 0 : main\_idx + 1; // advance to the next index

}

return 0;

}

//

// tree insertion routine (place your code here)

//

void tree\_insert(tree\_node\_t \*\*link, tree\_node\_t \*person , int main\_index){

if(\*link == NULL){

(\*link) = person;

}

else if(compare\_tree\_nodes(\*link, person, main\_index) > 0){

tree\_insert(&((\*link)->left[main\_index]), person, main\_index);

}

else{

tree\_insert(&((\*link)->right[main\_index]), person, main\_index);

}

}

//

// tree search routine (place your code here)

//

tree\_node\_t \*find(tree\_node\_t \*link, tree\_node\_t \*person, int main\_index){

if(link == NULL || compare\_tree\_nodes(link, person, main\_index) == 0) {

return link;

}

else if(compare\_tree\_nodes(link, person, main\_index) > 0){

return find(link->left[main\_index],person, main\_index);

}

else {

return find(link->right[main\_index],person, main\_index);

}

}

//

// tree depth

//

int tree\_depth(tree\_node\_t \*link, int main\_index) {

if (link == NULL){return 0;}

int ld = tree\_depth(link->left[main\_index], main\_index);

int rd = tree\_depth(link->right[main\_index], main\_index);

if (ld > rd) {return ld + 1;}

else {return rd + 1;}

}

//

// list, i,e, traverse the tree (place your code here)

//

int list(tree\_node\_t \*link, int main\_index, char \*compare){

if(link != NULL){

list(link->left[main\_index], main\_index, compare);

char \*search;

if (main\_index == 0){

search = link->name;

} else if (main\_index == 1){

search = link->zip\_code;

} else if (main\_index == 2){

search = link->telephone\_number;

} else {

search = link->social\_security\_number;

}

char sus[strlen(search)];

strcpy(sus,search);

if (main\_index != 0) {

int i = 0, j = 0;

while (sus[i]) {

if (sus[i] != ' ')

sus[j++] = sus[i];

i++;

}

sus[j] = '\0';

}

if(strcmp(compare,"NULL") == 0 || strstr(sus,compare)){

ctr++;

printf("Person #%d\n",ctr);

printf(" name --------------------- %s\n",link->name);

printf(" zip code ----------------- %s\n",link->zip\_code);

printf(" telephone number --------- %s\n",link->telephone\_number);

printf(" social security number --- %s\n",link->social\_security\_number);

}

list(link->right[main\_index], main\_index, compare);

}

return EXIT\_SUCCESS;

}

//

// main program

//

int main(int argc,char \*\*argv){

double dt;

// process the command line arguments

if(argc < 3){

fprintf(stderr,"Usage: %s student\_number number\_of\_persons [options ...]\n",argv[0]);

fprintf(stderr,"Recognized options:\n");

fprintf(stderr," -list[N] # list the tree contents, sorted by key index N (the default is index 0)\n");

// place a description of your own options here

return 1;

}

int student\_number = atoi(argv[1]);

if(student\_number < 1 || student\_number >= 1000000){

fprintf(stderr,"Bad student number (%d) --- must be an integer belonging to [1,1000000{\n",student\_number);

return 1;

}

int n\_persons = atoi(argv[2]);

if(n\_persons < 3 || n\_persons > 10000000){

fprintf(stderr,"Bad number of persons (%d) --- must be an integer belonging to [3,10000000]\n",n\_persons);

return 1;

}

// generate all data

tree\_node\_t \*persons = (tree\_node\_t \*)calloc((size\_t)n\_persons,sizeof(tree\_node\_t)); // árvore para pôr as pessoas

if(persons == NULL){

fprintf(stderr,"Output memory!\n");

return 1;

}

aed\_srandom(student\_number);

for(int i = 0;i < n\_persons;i++){

random\_name(&(persons[i].name[0]));

random\_zip\_code(&(persons[i].zip\_code[0]));

random\_telephone\_number(&(persons[i].telephone\_number[0]));

random\_social\_security\_number(&(persons[i].social\_security\_number[0]));

for(int j = 0;j < 4;j++)

persons[i].left[j] = persons[i].right[j] = NULL; // make sure the pointers are initially NULL

}

// create the ordered binary trees

dt = cpu\_time();

tree\_node\_t \*roots[4]; // four indices, four roots

for(int main\_index = 0;main\_index < 4;main\_index++){

roots[main\_index] = NULL;

}

for(int i = 0;i < n\_persons;i++){

for(int main\_index = 0;main\_index < 4;main\_index++){

tree\_insert(&(roots[main\_index]),&(persons[i]), main\_index); // place your code here to insert &(persons[i]) in the tree with number main\_index

}

}

dt = cpu\_time() - dt;

printf("Tree creation time (%d persons): %.3es\n",n\_persons,dt);

// search the tree

for(int main\_index = 0;main\_index < 4;main\_index++){

dt = cpu\_time();

for(int i = 0;i < n\_persons;i++){

tree\_node\_t n = persons[i]; // make a copy of the node data

if(find(roots[main\_index],&(n), main\_index) != &(persons[i])) // place your code here to find a given person, searching for it using the tree with number main\_index

{

fprintf(stderr,"person %d not found using index %d\n",i,main\_index);

return 1;

}

}

dt = cpu\_time() - dt;

printf("Tree search time (%d persons, index %d): %.3es\n",n\_persons,main\_index,dt);

}

// compute the largest tree depth

for(int main\_index = 0;main\_index < 4;main\_index++){

dt = cpu\_time();

int depth = tree\_depth(roots[main\_index], main\_index); // place your code here to compute the depth of the tree with number main\_index

dt = cpu\_time() - dt;

printf("Tree depth for index %d: %d (done in %.3es)\n",main\_index,depth,dt);

}

// process the command line optional arguments

for(int i = 3;i < argc;i++){

if(strncmp(argv[i],"-list",5) == 0){ // list all (optional)

int main\_index = atoi(&(argv[i][5]));

if(main\_index < 0)

main\_index = 0;

if(main\_index > 3)

main\_index = 3;

printf("List of persons:\n");

if ((i+1) < argc){

(void)list(roots[main\_index], main\_index, argv[i+1]); // place your code here to traverse, in order, the tree with number main\_index

} else {

(void)list(roots[main\_index], main\_index, "NULL"); // place your code here to traverse, in order, the tree with number main\_index

}

}

// place your own options here

}

// clean up --- don't forget to test your program with valgrind, we don't want any memory leaks

free(persons);

return 0;

}

//  Function in random\_data.c

void random\_social\_security\_number(char social\_security\_number[MAX\_SOCIAL\_SECURITY\_NUMBER + 1])

{

  int n1 = aed\_random() % 1000;          //  000..999

  int n2 = aed\_random() % 100;           //  00..99

  int n3 = aed\_random() % 10000;         //  0000..9999

  if(snprintf(social\_security\_number,MAX\_SOCIAL\_SECURITY\_NUMBER + 1,"%03d %02d %04d",n1,n2,n3) >= MAX\_SOCIAL\_SECURITY\_NUMBER + 1){

    fprintf(stderr,"social security number too large (%03d) (%02d (%04d)\n",n1,n2,n3);

    exit(1);

  }

}

# 5 Conclusão

Com a realização deste trabalho, conseguimos adquirir bastante conhecimento acerca da utilização de árvores binárias e das suas vantagens na ordenação de elementos. Antes do início da execução do projeto, não tínhamos tanta noção da utilidade das árvores binárias, porém à medida que fomos progredindo e compreendendo o que o enunciado do problema nos propunha, percebemos que a partir delas (e de sistema de comparação) podemos ordenar da maneira que pretendemos qualquer conjunto de elementos.

Foi desta maneira que fomos desenvolvendo o código que nos era pedido, aos poucos e com cuidado, para que todos os elementos fossem colocados na posição correta da árvore binária.

Desta forma, agora com o código mais organizado e sintetizado, podemos concluir que nos sentimos realizados com o resultado obtido, porque na nossa opinião, conseguimos implementar uma solução direta do problema de ordenação que nos foi apresentado, uma vez que conseguimos obter árvores devidamente estruturadas e ordenadas, através de algoritmos com uma complexidade satisfatória e com tempos de execução relativamente bons.

# 6 Bibliografia

Para a realização deste trabalho utilizamos os slides teóricos e os guiões práticos disponibilizados pelo docente para nos auxiliar na compreensão do objetivo, assim como os seguintes sites (consultados entre os dias 15/01/22 e 27/01/22):

* <https://stackoverflow.com/>
* <https://www.geeksforgeeks.org/binary-tree-set-1-introduction/>
* <https://stackoverflow.com/questions/2603692/what-is-the-difference-between-tree-depth-and-height/2603707#2603707>