universidade de aveiro

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

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

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.

2.4 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**”.

Assim, começamos por declarar as variáveis **c** e **search**, que verificam se o *node* atual corresponde à expressão regular passada como argumento à função e guardam a informação do *node* atual no index selecionado, respetivamente. Assim sendo, a função irá procurar cada uma das ***persons*** listadas na árvore binária ordenadamente, da menor para a maior, dependendo do índice passado.

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***, ao qual serão retirados os espaços e que depois será usada para com os dados do ***node*** em que estamos.

Com efeito, 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 em função do valor da variável **c**.

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.

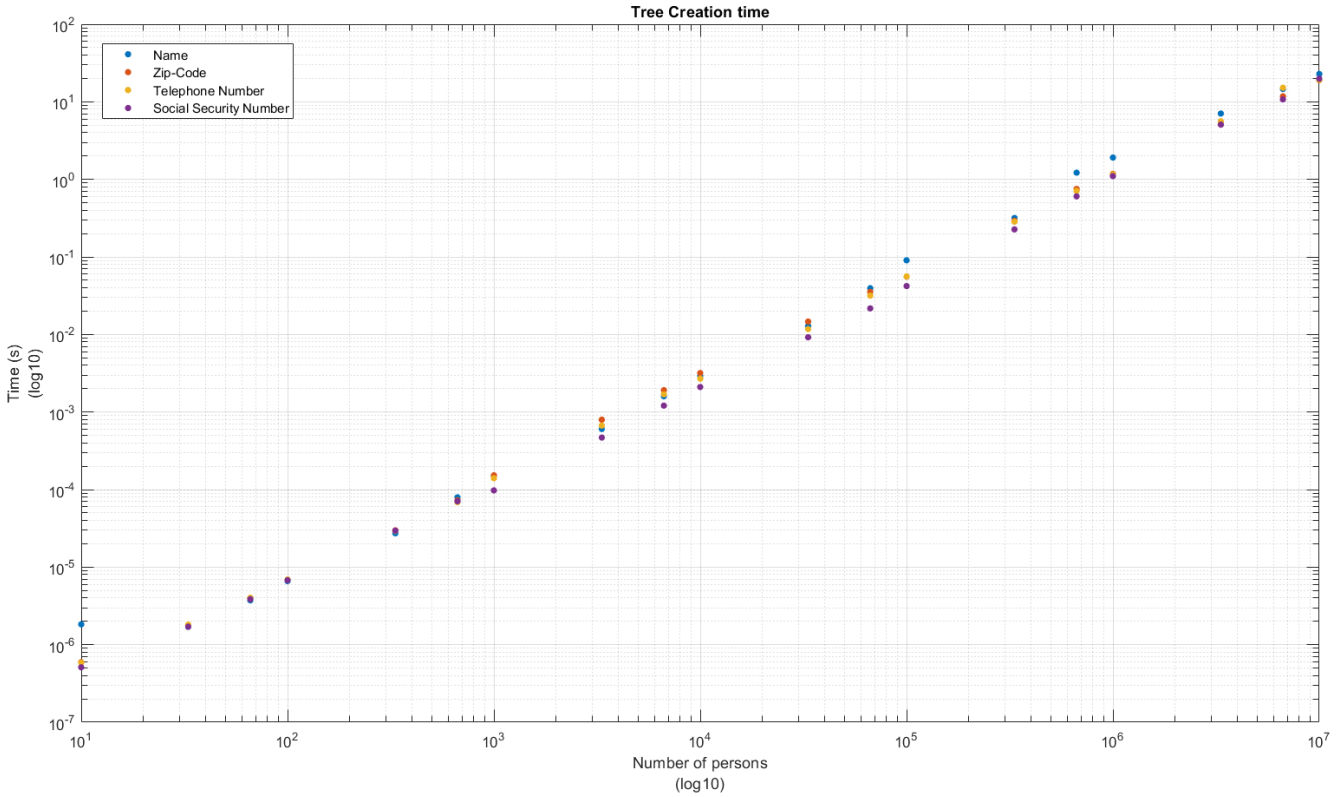
Para realizar a impressão dos resultados criámos a função *listprint,* com o objetivo de evitar a repetição do mesmo código.

3 Resultados

3.1 Tree Creation Time

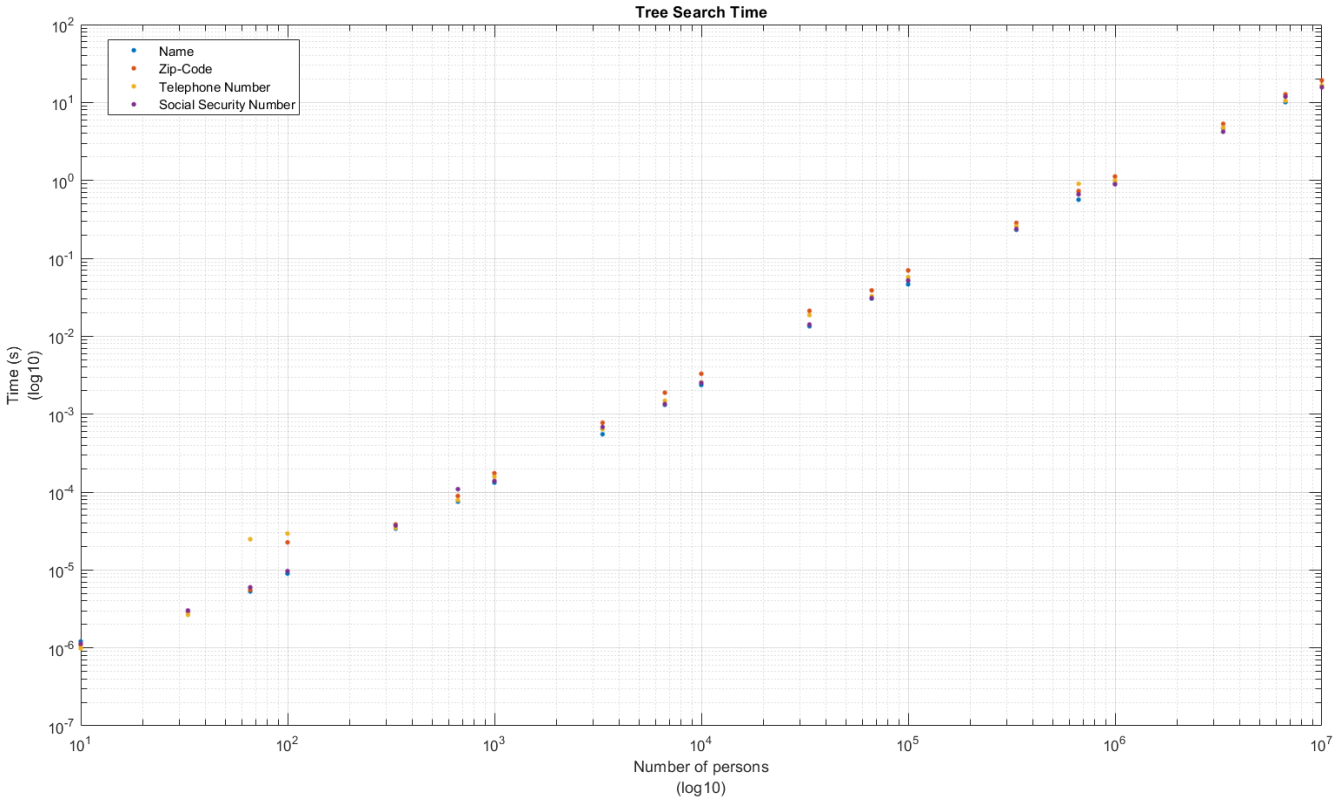
Com o objetivo de analisar os resultados obtidos criámos diversos gráficos que comparam o comportamento do script para cada índice, considerando diferentes critérios.

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

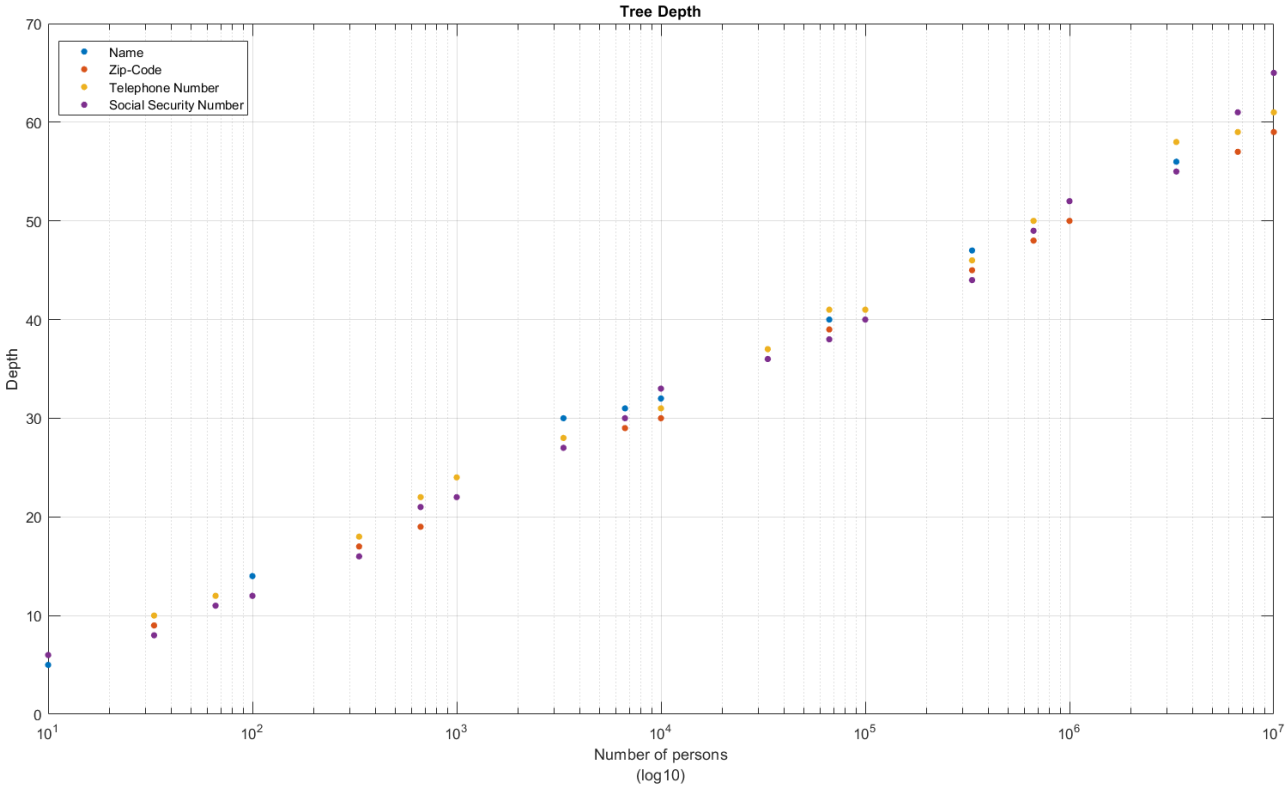
Além disso, o gráfico seguinte, que ilustra a evolução do tempo de procura em cada árvore em função do número de pessoas inseridas, permite-nos chegar à conclusão de que neste aspeto a quantidade de cada atributo também não provoca alterações na eficiência.



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).

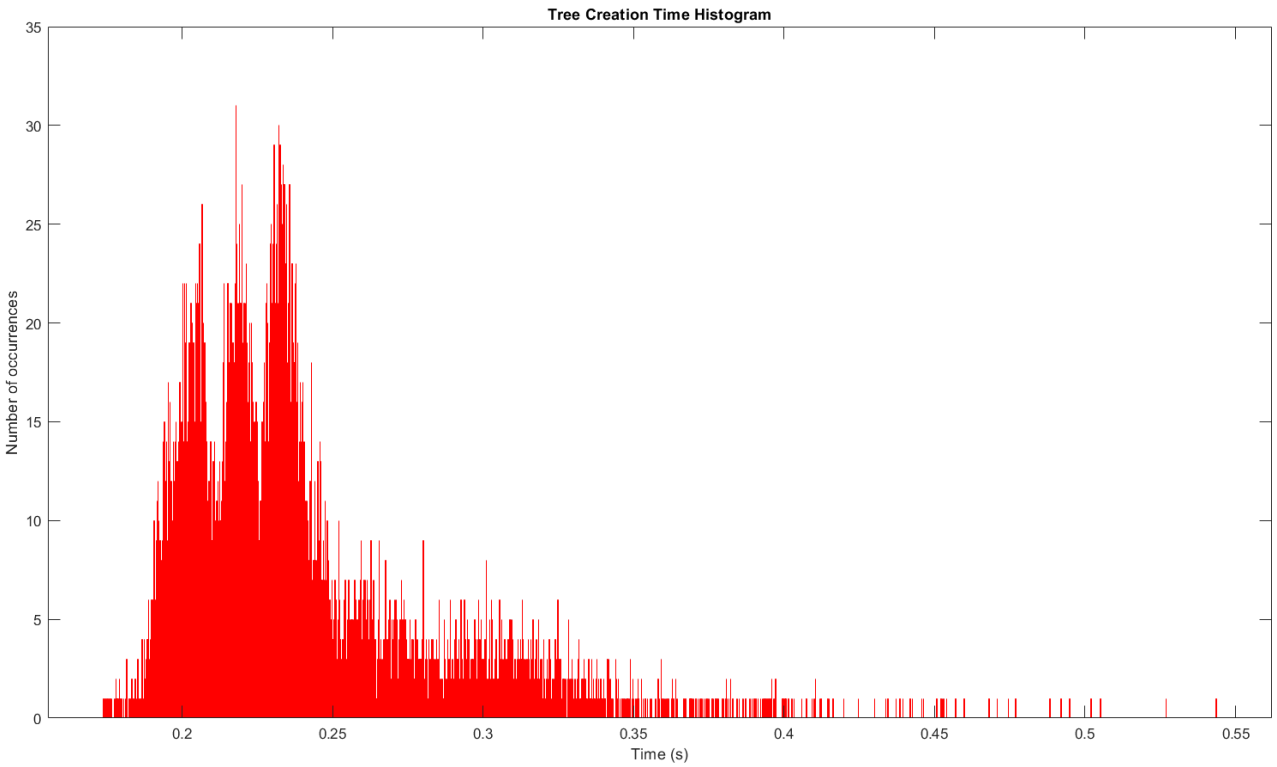
3.3 Tree Depth

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.



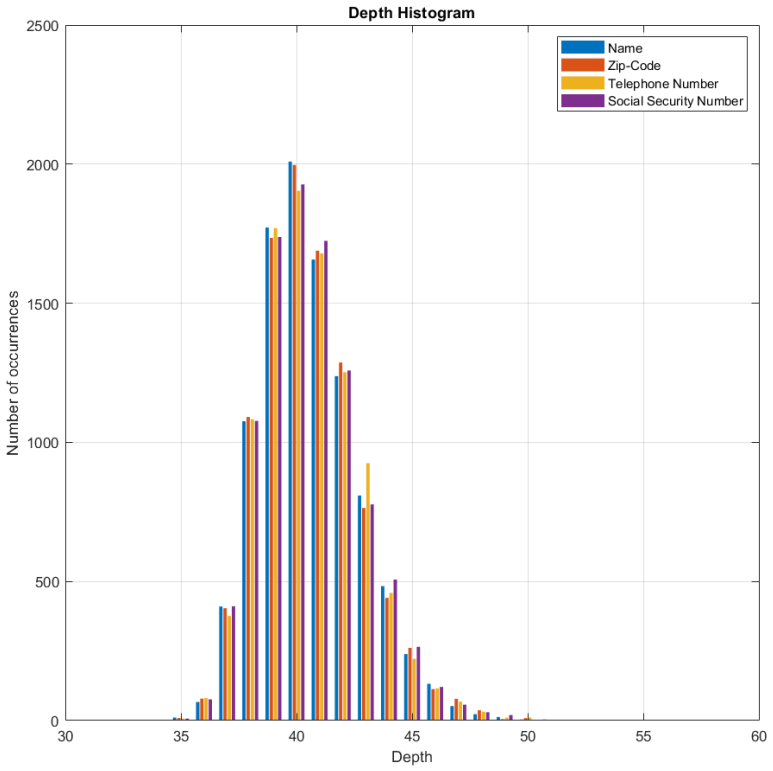
3.4 Tree Creation Time Histogram

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.



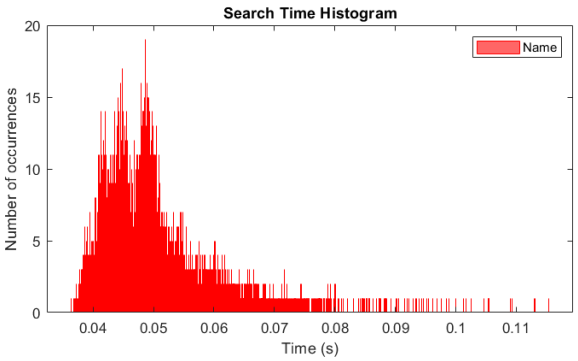
3.5 Tree Depth Histogram

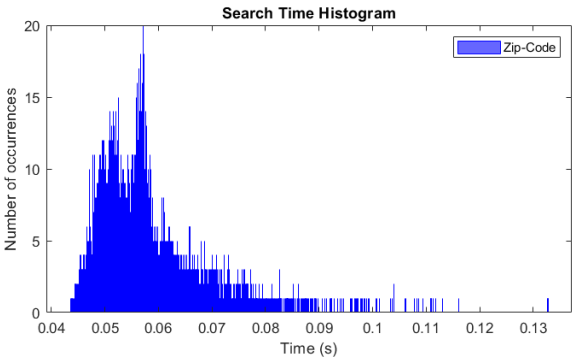
Utilizando as mesmas características que no histograma anterior para a execução observamos que a profundidade mais obtida é 40 níveis.

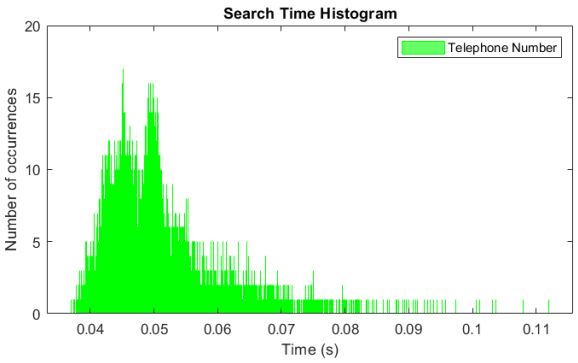


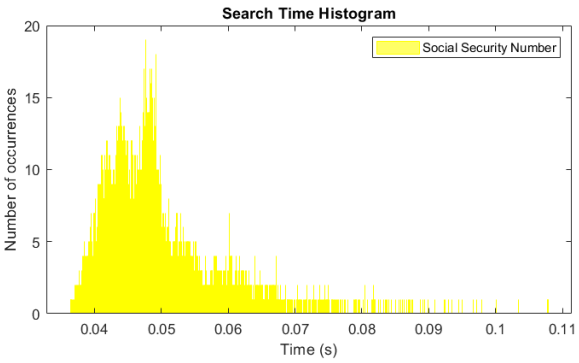
3.6 Search Time Histogram

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.

```
pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 102491 5 -list0
Tree creation time (5 persons): 3.000e-06s
Tree search time (5 persons, index 0): 1.000e-06s
Tree search time (5 persons, index 1): 6.000e-07s
Tree search time (5 persons, index 2): 5.000e-07s
Tree search time (5 persons, index 3): 6.000e-07s
Tree depth for index 0: 4 (done in 4.000e-07s)
Tree depth for index 1: 4 (done in 3.000e-07s)
Tree depth for index 2: 4 (done in 2.000e-07s)
Tree depth for index 3: 4 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Andrea Ortiz
  zip code ------------------ 78640 Kyle (Hays county)
  telephone number ---------- 2379 634 023
  social security number --- 667 10 9313
Person #2
  name ---------------------- Charles Barnett
  zip code ------------------ 32822 Orlando (Orange county)
  telephone number ---------- 5858 274 784
  social security number --- 553 29 2109
Person #3
  name ---------------------- Clara West
  zip code ------------------ 72401 Jonesboro (Craighead county)
  telephone number ---------- 6762 834 558
  social security number --- 772 94 0316
Person #4
  name ---------------------- Tony Gentry
  zip code ------------------ 28078 Huntersville (Mecklenburg county)
  telephone number ---------- 8493 774 801
  social security number --- 354 99 4566
Person #5
  name ---------------------- Virginia Colon
  zip code ------------------ 60804 Cicero (Cook county)
  telephone number ---------- 8630 972 478
  social security number --- 914 81 2152

pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 103234 5 -list1
Tree creation time (5 persons): 2.900e-06s
Tree search time (5 persons, index 0): 9.000e-07s
Tree search time (5 persons, index 1): 6.000e-07s
Tree search time (5 persons, index 2): 8.000e-07s
Tree search time (5 persons, index 3): 6.000e-07s
Tree depth for index 0: 3 (done in 3.000e-07s)
Tree depth for index 1: 4 (done in 3.000e-07s)
Tree depth for index 2: 4 (done in 2.000e-07s)
Tree depth for index 3: 3 (done in 5.000e-07s)
List of persons:
Person #1
  name ---------------------- Johnnie Rose
  zip code ------------------ 10029 New York City (New York county)
  telephone number ---------- 6819 292 396
  social security number --- 784 02 7520
Person #2
  name ---------------------- Michelle Vega
  zip code ------------------ 33023 Hollywood (Broward county)
  telephone number ---------- 3893 817 486
  social security number --- 223 49 3285
Person #3
  name ---------------------- James Davis
  zip code ------------------ 33647 Tampa (Hillsborough county)
  telephone number ---------- 1231 704 681
  social security number --- 588 16 3101
Person #4
  name ---------------------- Judith Allen
  zip code ------------------ 95076 Watsonville (Santa Cruz county)
  telephone number ---------- 4341 521 989
  social security number --- 606 86 1816
Person #5
  name ---------------------- Harold Bailey
  zip code ------------------ 95355 Modesto (Stanislaus county)
  telephone number ---------- 2933 774 107
  social security number --- 887 07 5510
```

```
pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 102491 5 -list2
Tree creation time (5 persons): 1.600e-06s
Tree search time (5 persons, index 0): 1.000e-06s
Tree search time (5 persons, index 1): 8.000e-07s
Tree search time (5 persons, index 2): 8.000e-07s
Tree search time (5 persons, index 3): 7.000e-07s
Tree depth for index 0: 4 (done in 2.000e-07s)
Tree depth for index 1: 4 (done in 2.000e-07s)
Tree depth for index 2: 4 (done in 4.000e-07s)
Tree depth for index 3: 4 (done in 4.000e-07s)
List of persons:
Person #1
  name ---------------------- Andrea Ortiz
  zip code ------------------ 78640 Kyle (Hays county)
  telephone number ---------- 2379 634 023
  social security number --- 667 10 9313
Person #2
  name ---------------------- Charles Barnett
  zip code ------------------ 32822 Orlando (Orange county)
  telephone number ---------- 5858 274 784
  social security number --- 553 29 2109
Person #3
  name ---------------------- Clara West
  zip code ------------------ 72401 Jonesboro (Craighead county)
  telephone number ---------- 6762 834 558
  social security number --- 772 94 0316
Person #4
  name ---------------------- Tony Gentry
  zip code ------------------ 28078 Huntersville (Mecklenburg county)
  telephone number ---------- 8493 774 801
  social security number --- 354 99 4566
Person #5
  name ---------------------- Virginia Colon
  zip code ------------------ 60804 Cicero (Cook county)
  telephone number ---------- 8630 972 478
  social security number --- 914 81 2152

pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 2022 5 -list3
Tree creation time (5 persons): 1.900e-06s
Tree search time (5 persons, index 0): 1.100e-06s
Tree search time (5 persons, index 1): 7.000e-07s
Tree search time (5 persons, index 2): 7.000e-07s
Tree search time (5 persons, index 3): 6.000e-07s
Tree depth for index 0: 4 (done in 3.000e-07s)
Tree depth for index 1: 5 (done in 2.000e-07s)
Tree depth for index 2: 4 (done in 2.000e-07s)
Tree depth for index 3: 4 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Vicki Turner
  zip code ------------------ 11435 Jamaica (Queens county)
  telephone number ---------- 5181 744 770
  social security number --- 064 61 8656
Person #2
  name ---------------------- Antonio Vincent
  zip code ------------------ 28269 Charlotte (Mecklenburg county)
  telephone number ---------- 2451 255 726
  social security number --- 092 31 9976
Person #3
  name ---------------------- Debby Walsh
  zip code ------------------ 22193 Woodbridge (Prince William county)
  telephone number ---------- 6288 182 833
  social security number --- 364 96 2061
Person #4
  name ---------------------- Adrian Snyder
  zip code ------------------ 92020 El Cajon (San Diego county)
  telephone number ---------- 1212 037 209
  social security number --- 516 26 4569
Person #5
  name ---------------------- Judy Guerra
  zip code ------------------ 44035 Elyria (Lorain county)
  telephone number ---------- 4691 237 264
  social security number --- 563 43 9325
```

```
pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 104142 5 -list0 "J"
Tree creation time (5 persons): 1.700e-06s
Tree search time (5 persons, index 0): 1.100e-06s
Tree search time (5 persons, index 1): 7.000e-07s
Tree search time (5 persons, index 2): 6.000e-07s
Tree search time (5 persons, index 3): 6.000e-07s
Tree depth for index 0: 5 (done in 5.000e-07s)
Tree depth for index 1: 4 (done in 2.000e-07s)
Tree depth for index 2: 4 (done in 3.000e-07s)
Tree depth for index 3: 4 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Jason Diaz
  zip code ------------------ 66062 Olathe (Johnson county)
  telephone number ---------- 3166 876 765
  social security number ---- 652 93 7438
Person #2
  name ---------------------- Joy Martinez
  zip code ------------------ 75052 Grand Prairie (Dallas county)
  telephone number ---------- 8984 442 243
  social security number ---- 746 48 7544

pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 103234 5 -list1 33
Tree creation time (5 persons): 1.600e-06s
Tree search time (5 persons, index 0): 9.000e-07s
Tree search time (5 persons, index 1): 6.000e-07s
Tree search time (5 persons, index 2): 5.000e-07s
Tree search time (5 persons, index 3): 5.000e-07s
Tree depth for index 0: 3 (done in 4.000e-07s)
Tree depth for index 1: 4 (done in 2.000e-07s)
Tree depth for index 2: 4 (done in 3.000e-07s)
Tree depth for index 3: 3 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Michelle Vega
  zip code ------------------ 33023 Hollywood (Broward county)
  telephone number ---------- 3893 817 486
  social security number ---- 223 49 3285
Person #2
  name ---------------------- James Davis
  zip code ------------------ 33647 Tampa (Hillsborough county)
  telephone number ---------- 1231 704 681
  social security number ---- 588 16 3101

pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 102491 5 -list2 67
Tree creation time (5 persons): 1.800e-06s
Tree search time (5 persons, index 0): 9.000e-07s
Tree search time (5 persons, index 1): 7.000e-07s
Tree search time (5 persons, index 2): 5.000e-07s
Tree search time (5 persons, index 3): 6.000e-07s
Tree depth for index 0: 4 (done in 4.000e-07s)
Tree depth for index 1: 4 (done in 3.000e-07s)
Tree depth for index 2: 4 (done in 3.000e-07s)
Tree depth for index 3: 4 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Clara West
  zip code ------------------ 72401 Jonesboro (Craighead county)
  telephone number ---------- 6762 834 558
  social security number ---- 772 94 0316

pjnp5@DESKTOP-DOICTN4:/mnt/c/Users/paulo/OneDrive/Documentos/GitHub/AED_PP/Projeto2- Multi-ordered trees$ ./multi_ordered_tree 2022 5 -list3 56
Tree creation time (5 persons): 1.500e-06s
Tree search time (5 persons, index 0): 1.000e-06s
Tree search time (5 persons, index 1): 6.000e-07s
Tree search time (5 persons, index 2): 5.000e-07s
Tree search time (5 persons, index 3): 8.000e-07s
Tree depth for index 0: 4 (done in 2.000e-07s)
Tree depth for index 1: 5 (done in 3.000e-07s)
Tree depth for index 2: 4 (done in 2.000e-07s)
Tree depth for index 3: 4 (done in 2.000e-07s)
List of persons:
Person #1
  name ---------------------- Judy Guerra
  zip code ------------------ 44035 Elyria (Lorain county)
  telephone number ---------- 4691 237 264
  social security number ---- 563 43 9325
```

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 depdth
//
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){
    int c = 0; char *search;
    if (main_index == 0){
      search = link->name;
      c = strcmp(link->name,compare);
    } else if (main_index == 1){
      search = link->zip_code;
      c = strcmp(link->zip_code,compare);
    } else if (main_index == 2){
      search = link->telephone_number;
      c = strcmp(link->telephone_number,compare);
    } else {
      search = link->social_security_number;
```

```
      c = strcmp(link->social_security_number,compare);
    }
    if(strcmp(compare,"NULL")==0){
      list(link->left[main_index], main_index, compare);
      listprint(link, search, compare);
      list(link->right[main_index], main_index, compare);
    } else {
      char *subSearch; int len = strlen(compare);
      if(strlen(search) < strlen(compare)) {
        len = strlen(search);
      }
      subSearch = (char*)malloc(sizeof(char) * (len+1));
      strncpy(subSearch, search, len);
      if (c > 0){
        list(link->left[main_index], main_index, compare);
        listprint(link, subSearch, compare);
      } else {
        listprint(link, subSearch, compare);
        list(link->right[main_index], main_index, compare);
      }
    }
  }
  return EXIT_SUCCESS;
}

int listprint(tree_node_t *link, char *search, char *compare){
  if(strcmp(compare,"NULL") == 0 || strcmp(search,compare) == 0){
    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);
  }
}


//
// 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)); // arvore
para por 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 depdth
  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;
}
```

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