LICENCIATURA EM ENGENHARIA INFORMÁTICA
ALGORITMOS E ESTRUTURAS DE DADOS
2021/2022

Multi-ordered Trees

01/02/2022

Airton Moreira | 100480 | 50%

Victor Barros Melo | 101099 | 50%

Índice

Introdução

Este relatório refere-se ao segundo trabalho prático proposto na disciplina de Algoritmos e Estruturas de Dados que tem como objetivo o desenvolvimento de uma aplicação em C para a simulação de uma base de dados incluindo o Nome, Código Postal (Zip Code), Número de Telemóvel e Número de Identificação de Segurança Social (NISS).

A resolução deste trabalho tem foco principal na implementação dos conceitos da Estrutura de dados de Árvores Binárias Quádruplas e a ênfase na eficiência desta estrutura principalmente nos métodos de listagem, procura de todas as entidades com o mesmo valor de um determinado campo (nome, zip code, etc.) e na busca por um nó específico, onde o algoritmo executa-os com grande eficiência.

Funções, Estruturas e Métodos Implementados

Estrutura da Árvore Binária Quádrupla

A `struct tree_node_s` consiste em um nó da nossa árvore binária a ser criada. Cada nó terá um campo de *Nome* (***name***), *Código Postal* (***zip_code***), *Número de Telemóvel* (***telephone_number***) e também foi adicionado o requisito extra (inserir um quarto campo) *Número de Identificação de Segurança Social* (***niss***). Cada índice desta *struct* corresponde a um destes campos de dado os quais são *arrays* de caracteres (***char ****) e cada um tem seu tamanho definido nas constantes do ficheiro *AED_2021_A02.h*.

Cada nó contém também a definição dos ponteiros para seus nós esquerdo e direito, onde o nó da esquerda deve conter dados de menor magnitude (comparados pela função ***compare_tree_nodes***) e o nó direito contém os maiores dados, ambos os nós contêm os mesmos quatro campos.

```
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 niss[MAX_TELEPHONE_NUMBER_SIZE - 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;
```

Algumas adaptações foram necessárias em alguns ficheiros para a implementação do quarto campo de dados, como a criação da função ***random_niss*** que cria um niss aleatório para cada pessoa na base de dados.

Ficheiro: randomData.c

```
void random_niss(char NISS[MAX_NISS + 1])
{
  int n1 = 1000 + aed_random() % 9000; // 1000..9999
  int n2 = aed_random() % 1000;        //  000..999
  int n3 = aed_random() % 100;
  if(snprintf(NISS,MAX_NISS + 1,"%04d%03d%04d",n1,n2,n3) > MAX_NISS +1)
  {
    fprintf(stderr,"niss number too large (%04d) (%03d (%04d)\n",n1,n2,n3);
    exit(1);
  }
}
```

Também foram criadas novas constantes para a implementação do niss, como a MAX_NISS que define o máximo de dígitos que um niss deve ter.

Ficheiro: AED_2021_A02.h

```
#define MAX_NAME_SIZE              31
#define MAX_ZIP_CODE_SIZE          63
#define MAX_TELEPHONE_NUMBER_SIZE  15
#define MAX_NISS  15

// from random_number.c
void aed_srandom(int seed);
int aed_random(void);

// from random_data.c
void random_name(char name[MAX_NAME_SIZE + 1]);
void random_zip_code(char zip_code[MAX_ZIP_CODE_SIZE + 1]);
void random_telephone_number(char telephone_number[MAX_TELEPHONE_NUMBER_SIZE
+ 1]);
void random_niss(char NISS[MAX_NISS + 1]);
```

Compare Tree Nodes

A função *compare_tree_nodes* tem como objetivo comparar o conteúdo de dois nós (***node1** e ***node2***) e utiliza também o inteiro que corresponde ao índice da ***struct tree_node_t***, ou seja, qual dado será utilizado para a comparação (vai de 0 a 3). Verifica-se qual campo desejado por uma estrutura de ***if's*** e compara as strings pelo método strcmp, onde é acessado o campo desejado de cada nó, como por exemplo, se o campo desejado for 0 (***name***), acessamos o campo dos nós pela estrutura (***node1->name, node2->name***). Se ***c for igual a zero***, **ambos os campos têm o mesmo valor**, **se *c for menor que zero* o primeiro campo é menor que o segundo**, **caso *c seja maior que zero* o primeiro campo é maior que o segundo**. Após comparar, incrementamos o índice e este processo se repete até compararmos todos os campos.

```
int compare_tree_nodes(tree_node_t *node1, tree_node_t *node2, int main_idx)
{
  int i, c;

  for (i = 0; i < 4; i++)
  {
    if (main_idx == 0)
      c = strcmp(node1->name, node2->name); // strcmp compare two strings, if they
are equal this function return 0
    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->niss, node2->niss);
    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 Insert

A função ***`tree_insert`*** tem o objetivo de inserir um novo nó na árvore. Primeiro, verificamos se o conteúdo do ponteiro para o nó ***link*** está vazio e se a árvore foi criada corretamente, esta verificação é executada no início. Após isso, fazemos a comparação do nó ***link*** com o nó 2, neste caso chamado de *parent*, através da função **`compare_tree_nodes`** e guardamos o valor retornado no inteiro C e após isso, se o conteúdo de ***link*** for maior que ***parent***, faremos a inserção recursiva, onde chamamos novamente a função **`tree_insert`** e comparamos o conteúdo do nó esquerdo de ***link*** com o ***parent***, a partir do ***tree_index***, caso o conteúdo de *link* seja menor que *parent,* fazemos o mesmo, porém para o nó direito.

```
void tree_insert(tree_node_t **link, tree_node_t *parent, int tree_index) // parent
here could be the tree_node *node (just name ch)
{
  if (*link == NULL)
  {                 // verifica se a árvore está vazia (criada corretamente)
    *link = parent; // executada somente no início da função, ou quando vamos
                    // juntar um novo no!
    return;
  }
  int c = compare_tree_nodes(*link, parent, tree_index); // c give us the order
  // link==node
  // c==0 node is equal , cant appen
  // C<0 node 1  aparece antes de node 2 na sequência
  // C>0 node1 aparece depois de node 2 na sequência
  //  else if (tree_index <= (*link)->tree_index)
  if (c > 0)
    tree_insert(&((*link)->left[tree_index]), parent, tree_index); // call by
pointer - aceder o valor da variável apontada no interior da função.
  else
    tree_insert(&((*link)->right[tree_index]), parent, tree_index);
}
```

Tree Find

A função ***find** retorna o ponteiro para o ***tree_node_t** que é encontrado pela função. Temos como parâmetro o ponteiro para o ponteiro de ***link** e o nó que contém os dados desejados a partir do ***tree_index**. Começamos por comparar ambos os nós pela função ***compare_tree_nodes** entre ***link** e ***data**, onde verificamos se o conteúdo de ***link** é vazio ou se a comparação retornou zero (conteúdos iguais) o que deve retornar o ponteiro para o nó ***link**, feito no início para verificar se foi criado corretamente e no final quando a comparação finalmente retorna conteúdos iguais entre os nós.
A função começa por buscar recursivamente pelo nó esquerdo de ***link** se for maior que ***data**, caso seja menor, buscaremos pelo nó direito de ***link**, o que irá percorrer a árvore toda a procura de nós iguais e retornando o ponteiro para estes nós.

```
tree_node_t *find(tree_node_t **link, tree_node_t *data, int tree_index)
{
  int c = compare_tree_nodes(*link, data, tree_index);
  if (*link == NULL || c == 0)
    return *link;
  if (c > 0)
    return find(&((*link)->left[tree_index]), data, tree_index);
  else
    return find(&((*link)->right[tree_index]), data, tree_index);
}
```

Tree Depth

A função ***tree_depth*** retorna um inteiro que corresponde a profundidade da árvore. Começamos por verificar se o conteúdo ou o ponteiro de ***link*** é vazio, pois se for não terá nenhum nó filho, logo a profundidade da árvore é zero. Salvamos no inteiro ***altesq*** a altura do nó esquerdo de ***link*** e em ***altdire*** a altura do nó direito. Se a altura do nó ***esquerdo for maior que o direito***, retornamos a profundidade da árvore como sendo a ***altesq + 1***, pois a profundidade a ser considerada é a do nó com maior altura + 1, caso a ***maior altura seja a do nó direito***, retornamos ***altdire + 1*** e assim recursivamente percorre a altura de todos os nós da árvore.

```
int tree_depth(tree_node_t **link, int tree_index)
{

  if (*link == NULL)
    return 0;
  if (link == NULL)
    return 0;

  int altesq = tree_depth(&((*link)->left[tree_index]), tree_index);
  int altdire = tree_depth(&((*link)->right[tree_index]), tree_index);

  if (altesq > altdire)
    return (altesq + 1);
  else
    return (altdire + 1);
}
```

Tree List

A função ***list*** tem como objetivo fazer o print de todas os campos de todos os nós da árvore binária. Começamos a verificar se o nó ***link*** está vazio, pois se estiver, chegamos ao final da árvore ou houve algum erro na criação do nó. Foi determinada a ordem natural de listagem da árvore binária, portanto primeiro acessamos o nó esquerdo de ***link,*** vamos fazer o print do número correspondente aquela pessoa na base de dados (assim como foi mostrado no output do Sr.Professor) pela variável global ***PERSON_COUNTER*** e dos campos: ***name, zip_code, telephone_number e niss,*** acessados por call by pointer. Prosseguimos por acessar o nó direito e a repetir este processo de listagem recursivamente enquanto incrementamos ***PERSON_COUNTER.***
A função só será utilizada se o usuário ao rodar o programa com o argumento `-list[N]` onde N representa por qual campo a listagem deve ordenar de forma crescente, 0 para name, 1 para zip_code, 2 para telephone_number e 3 para niss.

```
void list(tree_node_t *link, int tree_index)
{
  if (link == NULL)
    return;
  list(link->left[tree_index], tree_index);
                                                    // visita o filho esq
  printf("\n Person #%d \n", PERSON_COUNTER++);
                                                      // imprime a pessoa
  printf("Name ------ %s\n", link->name);
                                                // imprime a info nome da link
  printf("Zip code ------ %s\n", link->zip_code);
                                             // imprime a info zip_code da link
  printf("Telephone number --- %s\n", link->telephone_number);
                                  // imprime a info telephone_number da link
  printf("NISS --- %s\n", link->niss);
                                                // imprime a info niss da link
  list(link->right[tree_index], tree_index);
                                                    // visita o filho dir
}
```

Tree List Find

A função ***list_find*** tem como objetivo a procura e a listagem de nós que contenham a mesma informação em determinado campo (ex.: mesmo *zip_code*). É baseada na função ***list*** e um requisito extra. A função só será utilizada se o usuário ao rodar o programa com uma das opções como argumento `-zip[N] -niss[N] -name[N]` onde N representa neste caso o ***zip_code, niss ou nome*** desejado para a busca e listagem.
Tem funcionamento muito próximo a ***list*** onde fazemos exatamente o mesmo tipo de listagem, da esquerda para a direita em ordem natural, porém após verificar se o ***link*** é nulo fazemos uma estrutura de ***if's*** baseado na opção passada como argumento, que corresponde a qual campo que vamos procurar (*0 name, 2 zip_code, 3 telephone_number e 4 niss*) e dependendo da opção selecionada, faremos uma comparação de strings (***strcmp***) entre os dados do nó ***link*** e ***search (string)*** que pretende-se encontrar. O resultado dessa busca fica guardado no inteiro comp
que se tiver o valor ***zero*** corresponde que o valor de ***link e search*** são iguais e prossegue para o print das informações do nó esquerdo, assim como na função ***list***. Após imprimir todos os campos e o contador de pessoas, prossegue-se para a listagem do nó direito e assim em diante recursivamente, até chegar ao final na Árvore.No caso de uma busca usando a opção -name pode se fazer tanto usando o nome completo da pessoa , ou só o primeiro nome(nesse casso é apresentado a(s) pessoa(s) que tem o string desejado.

```
void list_find(tree_node_t *link, int tree_index, char *search,int option)
{
  if (link == NULL)
    return;


  list_find(link->left[tree_index], tree_index, search,option);
// visita o filho esq

  int comp;
  if (option==1){
    comp = strcmp (link->nif,search);
  }
  else if(option==0){
      comp = strcmp (link->zip_code,search);
  }
  else{
    char *copiedname = (char*)calloc(strlen(link->name)+1,sizeof(char));
    strcpy(copiedname,link->name);
    char *test = (char*)calloc(strlen(link->name)+1,sizeof(char));
    comp = strcmp (link->name,search);
```

```
    if (comp!=0)
    {
      for (int  i = 0; i < strlen(link->name)-1; i++)
    {
      if ((link->name)[i]==' ' )
      {
       break;
      }
      else{
        test[i]=(link->name)[i];
        comp=strcmp(test,search);
      }
    }
    }
      free(copiedname);
      free(test);
  }

  if (comp==0)
    {

      printf("\n Person #%d \n", PERSON_COUNTER++);                //
imprime a pessoa

       if (option==2){
        printf("Name ------ "ANSI_COLOR_RED  "%s\n" ANSI_COLOR_RESET ,
link->name);

      }
      else
      {
        printf("Name ------ %s\n", link->name);                     //
imprime a info nome da link

      }
       if (option==0){
        printf("Zip code------ "  ANSI_COLOR_RED  "%s\n" ANSI_COLOR_RESET ,
link->zip_code);
      }
      else  {
         printf("Zip code --- %s\n", link->zip_code);
```

```
      }
      //printf("Zip code ------ %s\n", link->zip_code);              //
imprime a info zip_code da link
      printf("Telephone number --- %s\n", link->telephone_number); //
imprime a info telephone_number da link
      if (option==1){
        printf("NISS --- " ANSI_COLOR_RED  "%s \n" ANSI_COLOR_RESET ,
link->nif);
      }
      else{
         printf("NISS --- %s\n", link->nif);
      }
    }
  list_find(link->right[tree_index], tree_index, search,option);
// visita o filho dir
}
```

Main

Onde implementamos para a execução e a chamada de todas as funções construídas, além de alterações no tratamento do input para acomodar 4 campos e nossas funções extras .

```
int main(int argc, char **argv)
{
    // Printing everything into a file!
    FILE *file_ct;
    FILE *file_st;
    FILE *file_td;
    FILE *file_tdWpersons;
    // freopen("multi_ordered_tree_times.txt", "w", stdout);
    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
        fprintf(stderr, "  -zip [N]             # list all person with a given zip
code\n");
        fprintf(stderr, "  -name [N]             # list people with the given name \n");
        fprintf(stderr, "  -nif [N]             # list the  person who has the given nif
code\n");
        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)); // calloc arguments 0- numerber of elements(i.e,nodes )  1 size in
bytes of each elements

// this creat all nodes we need and put zero to not have trash
    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_nif(&(persons[i].nif[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]; // three indices, three 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);

    dt = cpu_time() - dt;
    file_ct = fopen("creation_time.txt", "a"); // output tree creation time
    printf("Tree creation time (%d persons): %.3es\n", n_persons, dt);
    fprintf(file_ct, "%d %.3es\n", n_persons, dt);
    fclose(file_ct);
    // search the tree

    // print to a file
    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;
        file_st = fopen("search_time.txt", "a"); // output tree creation time
        printf("Tree search time (%d persons, index %d): %.3es\n", n_persons,
main_index, dt);
        fprintf(file_st, "%d %d %.3es\n", n_persons, main_index, dt);
        fclose(file_st);
    }

    // compute the largest tree depdth
    for (int main_index = 0; main_index < 4; main_index++)
    {
        file_tdWpersons = fopen("treeDepth_timeWpersons.txt", "a"); // output tree
creation time
```

```
        file_td = fopen("treeDepth_time.txt", "a");                // output tree
creation time

        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);
        fprintf(file_td, "%d %d %.3es\n", main_index, depth, dt);
        fclose(file_td);

        fprintf(file_tdWpersons, "%d %d %d %.3es\n", n_persons, main_index, depth, dt);
        fclose(file_tdWpersons);
    }

    // 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]));
            // main_index=mostrar por orden de :: 0-por nome 1-pelo zip code e 2-por
telefone number 3- pelo NISS
            if (main_index < 0)
                main_index = 0;
            if (main_index > 3)
                main_index = 3;

            switch (main_index)
            {
            case 0:
                printf("\n          .....Ordered by Name.....\n ");
                break;
            case 1:
                printf("\n          .....Ordered by Zip Code.....\n ");
                break;
            case 2:
                printf("\n          .....Ordered by Telephone Number.....\n ");
                break;
            case 3:
                printf("\n          .....Ordered by NISS.....\n ");
                break;
            default:
                printf("\n          .....Ordered by NISS.....\n");
                break;
            }
            (void)list(&persons[0], main_index);
        }

        // 0 for zip code
        // 1 for niss
        // 2 for name

        if (strncmp(argv[i], "-zip", 5) == 0 && argc > 4)
        {
```

```
            int len1 = strlen(argv[i + 1]) + 1;
            char zipcode[len1];
            strcpy(zipcode, argv[i + 1]); // copy string from argv[i+1] to variable
            printf("\n --Desired zip %s--- \n", zipcode);

            (void)list_find(&persons[0], 0, zipcode, 0);
        }
        if (strncmp(argv[i], "-niss", 5) == 0 && argc > 4)
        {
            printf("\n --Desired niss %s--- \n", argv[i + 1]);
            (void)list_find(&persons[0], 0, argv[i + 1], 1);
        }
        if (strncmp(argv[i], "-name", 5) == 0 && argc > 4)
        {
            printf("\n --Desired name %s--- \n", argv[i + 1]);

            (void)list_find(&persons[0], 0, argv[i + 1], 2);
        }
    }
    free(persons);
    return 0;
}
```

Testes e Outputs

Output da execução normal do programa

Assim como no output mostrado pelo professor, executamos com os mesmos parâmetros.
./multi_ordered_tree 2021 10000000

```
Tree creation time (1000000 persons): 9.000e+00s
Tree search time (1000000 persons, index 0): 3.299e+00s
Tree search time (1000000 persons, index 1): 2.588e+00s
Tree search time (1000000 persons, index 2): 2.742e+00s
Tree search time (1000000 persons, index 3): 2.135e+00s
Tree depth for index 0: 49 (done in 1.125e-01s)
Tree depth for index 1: 51 (done in 1.232e-01s)
Tree depth for index 2: 51 (done in 1.213e-01s)
Tree depth for index 3: 50 (done in 8.511e-02s)
```

Output da função: List

Testamos o output da função em ordem crescente, primeiro por nome, depois zip_code, telephone_number e por último niss. Deixamos à amostra as informações sobre o tempo de criação, busca e profundidade apenas no primeiro output, já que não diferem significativamente dos outros.

./multi_ordered_tree 2021 4 -list0

```
Tree creation time (4 persons): 2.200e-06s
Tree search time (4 persons, index 0): 2.700e-06s
Tree search time (4 persons, index 1): 3.700e-06s
Tree search time (4 persons, index 2): 8.600e-06s
Tree search time (4 persons, index 3): 3.900e-06s
Tree depth for index 0: 4 (done in 2.400e-06s)
Tree depth for index 1: 3 (done in 3.700e-06s)
Tree depth for index 2: 3 (done in 2.900e-06s)
Tree depth for index 3: 3 (done in 3.200e-06s)

          .....Ordered by Name.....

 Person #1
Name ------ Chong Kirby
Zip code ------ 11211 Brooklyn (Kings county)
Telephone number --- 2556 934 010
NISS --- 45263490065

 Person #2
Name ------ David Mills
Zip code ------ 94112 San Francisco (San Francisco county)
Telephone number --- 6061 785 619
NISS --- 57058590044

 Person #3
Name ------ Dawn Santos
```

```
Zip code ------ 30101 Acworth (Cobb county)
Telephone number --- 5114 054 422
NISS --- 93885160065

 Person #4
Name ------ Luke Hall
Zip code ------ 11215 Brooklyn (Kings county)
Telephone number --- 7362 997 722
NISS --- 80262150047
```

./multi_ordered_tree 2021 4 -list1

```
  .....Ordered by Zip Code.....

 Person #1
Name ------ Chong Kirby
Zip code ------ 11211 Brooklyn (Kings county)
Telephone number --- 2556 934 010
NISS --- 45263490065

 Person #2
Name ------ Luke Hall
Zip code ------ 11215 Brooklyn (Kings county)
Telephone number --- 7362 997 722
NISS --- 80262150047

 Person #3
Name ------ Dawn Santos
Zip code ------ 30101 Acworth (Cobb county)
Telephone number --- 5114 054 422
NISS --- 93885160065

 Person #4
Name ------ David Mills
Zip code ------ 94112 San Francisco (San Francisco county)
Telephone number --- 6061 785 619
NISS --- 57058590044
```

./multi_ordered_tree 2021 4 -list2

```
          .....Ordered by Telephone Number.....

 Person #1
Name ------ Chong Kirby
Zip code ------ 11211 Brooklyn (Kings county)
Telephone number --- 2556 934 010
NISS --- 45263490065

 Person #2
Name ------ Dawn Santos
Zip code ------ 30101 Acworth (Cobb county)
Telephone number --- 5114 054 422
NISS --- 93885160065

 Person #3
Name ------ David Mills
Zip code ------ 94112 San Francisco (San Francisco county)
Telephone number --- 6061 785 619
NISS --- 57058590044

 Person #4
Name ------ Luke Hall
Zip code ------ 11215 Brooklyn (Kings county)
Telephone number --- 7362 997 722
NISS --- 80262150047
```

./multi_ordered_tree 2021 4 -list3

```
.....Ordered by NISS.....

 Person #1
Name ------ Chong Kirby
Zip code ------ 11211 Brooklyn (Kings county)
Telephone number --- 2556 934 010
NISS --- 45263490065

 Person #2
Name ------ David Mills
Zip code ------ 94112 San Francisco (San Francisco county)
Telephone number --- 6061 785 619
NISS --- 57058590044

 Person #3
Name ------ Luke Hall
Zip code ------ 11215 Brooklyn (Kings county)
Telephone number --- 7362 997 722
NISS --- 80262150047

 Person #4
Name ------ Dawn Santos
Zip code ------ 30101 Acworth (Cobb county)
Telephone number --- 5114 054 422
NISS --- 93885160065
```

Output da função: List Find

Testamos a função primeiro pelo nome completo e também só com primeiro nome, depois pelo zip code e ainda a busca pelo indivíduo que possui um tal niss. Passamos a string desejada para a busca entre aspas duplas (ex.:“desired_match“). Informações sobre o tempo de criação, busca e profundidade estão disponíveis em todos os outputs para uma melhor visualização da performance.

./multi_ordered_tree 2021 100000 -name "Andrew Davis"

```
Tree creation time (100000 persons): 4.366e-01s
Tree search time (100000 persons, index 0): 1.341e-01s
Tree search time (100000 persons, index 1): 1.177e-01s
Tree search time (100000 persons, index 2): 1.095e-01s
Tree search time (100000 persons, index 3): 1.058e-01s
Tree depth for index 0: 39 (done in 7.026e-03s)
Tree depth for index 1: 43 (done in 7.043e-03s)
Tree depth for index 2: 41 (done in 1.414e-02s)
Tree depth for index 3: 38 (done in 1.247e-02s)

 --Desired name Andrew Davis---

 Person #1
Name ------ Andrew Davis
Zip code --- 10128 New York City (New York county)
Telephone number --- 5795 592 160
NISS --- 85236790031

 Person #2
Name ------ Andrew Davis
Zip code --- 11372 Jackson Heights (Queens county)
Telephone number --- 3289 737 044
NISS --- 61044130017

 Person #3
Name ------ Andrew Davis
Zip code --- 20878 Gaithersburg (Montgomery county)
Telephone number --- 1202 349 271
NISS --- 95544470005

 Person #4
Name ------ Andrew Davis
Zip code --- 90640 Montebello (Los Angeles county)
Telephone number --- 3990 764 630
NISS --- 21951350015
```

./multi_ordered_tree 2022 10000 -name 'Pedro' (lista todas cujo primeiro nome é Pedro)

```
Tree creation time (10000 persons): 1.337e-02s
Tree search time (10000 persons, index 0): 3.217e-03s
Tree search time (10000 persons, index 1): 3.625e-03s
Tree search time (10000 persons, index 2): 3.494e-03s
Tree search time (10000 persons, index 3): 2.635e-03s
Tree depth for index 0: 32 (done in 7.147e-04s)
Tree depth for index 1: 32 (done in 6.017e-04s)
Tree depth for index 2: 30 (done in 5.205e-04s)
Tree depth for index 3: 30 (done in 4.942e-04s)

 --Desired name Pedro---

 Person #1
Name ------ Pedro Barr
Zip code --- 92021 El Cajon (San Diego county)
Telephone number --- 3205 136 911
NISS --- 97820340083

 Person #2
Name ------ Pedro Davis
Zip code --- 95123 San Jose (Santa Clara county)
Telephone number --- 1802 717 641
NISS --- 17061370075

 Person #3
Name ------ Pedro Moore
Zip code --- 85142 Queen Creek (Maricopa county)
Telephone number --- 7331 500 137
NISS --- 65907680021

 Person #4
Name ------ Pedro Palmer
Zip code --- 10977 Spring Valley (Rockland county)
Telephone number --- 1571 974 594
NISS --- 39413410064

 Person #5
Name ------ Pedro Rush
Zip code --- 85308 Glendale (Maricopa county)
Telephone number --- 2158 019 615
NISS --- 44722200025

 Person #6
Name ------ Pedro Strickland
Zip code --- 96706 Ewa Beach (Honolulu county)
Telephone number --- 7357 958 597
NISS --- 73626610011

 Person #7
Name ------ Pedro Torres
Zip code --- 92201 Indio (Riverside county)
Telephone number --- 2696 105 839


NISS --- 11413950015
```

./multi_ordered_tree 2021 10000 -zip "11372 Jackson Heights (Queens county)"

```
Tree creation time (10000 persons): 1.675e-02s
Tree search time (10000 persons, index 0): 6.261e-03s
Tree search time (10000 persons, index 1): 5.057e-03s
Tree search time (10000 persons, index 2): 5.390e-03s
Tree search time (10000 persons, index 3): 4.770e-03s
Tree depth for index 0: 29 (done in 8.622e-04s)
Tree depth for index 1: 33 (done in 5.795e-04s)
Tree depth for index 2: 32 (done in 7.922e-04s)
Tree depth for index 3: 31 (done in 7.143e-04s)

 --Desired zip 11372 Jackson Heights (Queens county)---

 Person #1
Name ------ Brent Moreno
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 4384 846 879
NISS --- 50706790083

 Person #2
Name ------ Chris White
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 5738 660 041
NISS --- 26690610083

 Person #3
Name ------ Christina Hill
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 6653 778 100
NISS --- 12343170060

 Person #4
Name ------ David Palmer
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 1910 287 930
NISS --- 81261070011

 Person #5
Name ------ Erik Scott
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 8449 238 191
NISS --- 50349280082

 Person #6
Name ------ Janice Hernandez
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 6709 519 688
NISS --- 35656540071

 Person #7
Name ------ Lora Miles
Zip code------ 11372 Jackson Heights (Queens county)
```

```
Telephone number --- 8166 326 163
NISS --- 22446690082

 Person #8
Name ------ Luther Page
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 9611 358 905
NISS --- 41039810097

 Person #9
Name ------ Matt Martin
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 3593 981 481
NISS --- 67825360001

 Person #10
Name ------ Melissa Roman
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 7775 672 502
NISS --- 85030450053

 Person #11
Name ------ Nancy Hendrix
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 1701 590 583
NISS --- 29896090087

 Person #12
Name ------ Patsy Rivera
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 1705 678 456
NISS --- 32442550045

 Person #13
Name ------ Rodolfo Rodriguez
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 7461 143 017
NISS --- 91790120061

 Person #14
Name ------ William Harper
Zip code------ 11372 Jackson Heights (Queens county)
Telephone number --- 6864 922 900
NISS --- 77855710034
```

./multi_ordered_tree 2021 100000 -niss "12948370094"

```
Tree creation time (100000 persons): 4.332e-01s
Tree search time (100000 persons, index 0): 1.510e-01s
Tree search time (100000 persons, index 1): 1.769e-01s
Tree search time (100000 persons, index 2): 1.341e-01s
Tree search time (100000 persons, index 3): 9.340e-02s
Tree depth for index 0: 39 (done in 8.940e-03s)
Tree depth for index 1: 43 (done in 9.515e-03s)
Tree depth for index 2: 41 (done in 9.146e-03s)
Tree depth for index 3: 38 (done in 1.061e-02s)

 --Desired niss 12948370094---

 Person #1
Name ------ Zachary Cabrera
Zip code --- 34711 Clermont (Lake county)
Telephone number --- 2765 496 725
NISS --- 12948370094
```

Testando o script com o Valgrind

valgrind -s --track-origins=yes ./multi_ordered_tree 2022 100000 -name 'Pedro Ibarra'

```
==281== Memcheck, a memory error detector
==281== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==281== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==281== Command: ./multi_ordered_tree 2022 100000 -name Pedro\ Ibarra
==281==
Tree creation time (100000 persons): 6.202e+00s
Tree search time (100000 persons, index 0): 1.763e+00s
Tree search time (100000 persons, index 1): 2.293e+00s
Tree search time (100000 persons, index 2): 1.706e+00s
Tree search time (100000 persons, index 3): 1.672e+00s
Tree depth for index 0: 40 (done in 5.030e-02s)
Tree depth for index 1: 40 (done in 5.243e-02s)
Tree depth for index 2: 39 (done in 6.069e-02s)
Tree depth for index 3: 42 (done in 6.160e-02s)

 --Desired name Pedro Ibarra---

 Person #1
Name ------ Pedro Ibarra
Zip code --- 92553 Moreno Valley (Riverside county)
Telephone number --- 3926 775 514
NISS --- 28358830018
==281==
==281== HEAP SUMMARY:
==281==     in use at exit: 6,136 bytes in 13 blocks
==281==   total heap usage: 100,015 allocs, 100,002 frees, 20,597,032 bytes allocated
==281==
==281== LEAK SUMMARY:
==281==    definitely lost: 0 bytes in 0 blocks
==281==    indirectly lost: 0 bytes in 0 blocks
==281==      possibly lost: 0 bytes in 0 blocks
==281==    still reachable: 6,136 bytes in 13 blocks
==281==         suppressed: 0 bytes in 0 blocks
==281== Rerun with --leak-check=full to see details of leaked memory
==281==
==281== For lists of detected and suppressed errors, rerun with: -s
==281== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Estatísticas de Desempenho

Obtendo diversas execuções

Após correrrmos o programa durante várias horas, obtivemos os tempos de criação e busca e a profundidade da árvore. Primeiro, criamos um script em *bash* com o objetivo de correrrmos o programa incrementando o número de pessoas para cada número mecanográfico.
Começamos por executar de 4 a 10.000 pessoas com o incremento de 1 por run, depois vamos de 10.000 a 50.000 de 100 em 100 pessoas e por último vamos de 50.000 a 10.000.000 de pessoas. Todavia, visto o imenso tempo de execução para chegar aos 10 mi de pessoas, decidimos parar em 2.253.000 para gerarmos nossos gráficos e estatística de desempenho.

```
#!/bin/bash
#### RODANDO COM O NMEC = 101099
while (true); do
   for i in {4..10000..1}
   do
      echo runing for: $i >> search_time.txt
      echo runing for: $i >> creation_time.txt
      echo runing for: $i >> treeDepth_time.txt
      ./multi_ordered_tree 101099 $i
      echo $'\n' >> search_time.txt
      echo $'\n' >> creation_time.txt
      echo $'\n' >> treeDepth_time.txt
   done
   for i in {10000..50000..100}
   do
      echo runing for: $i >> search_time.txt
      echo runing for: $i >> creation_time.txt
      echo runing for: $i >> treeDepth_time.txt
      ./multi_ordered_tree 101099 $i
      echo $'\n' >> search_time.txt
      echo $'\n' >> creation_time.txt
      echo $'\n' >> treeDepth_time.txt
   done
   for i in {50000..10000000..1000}
   do
      echo runing for: $i >> search_time.txt
      echo runing for: $i >> creation_time.txt
      echo runing for: $i >> treeDepth_time.txt
      ./multi_ordered_tree 101099 $i
      echo $'\n' >> search_time.txt
      echo $'\n' >> creation_time.txt
      echo $'\n' >> treeDepth_time.txt
   done
done
```

Obtendo o Output das Estatísticas

Para obtermos o output das diversas estatísticas de forma a conseguirmos fazer os gráficos desejados, tivemos de salvar o output de cada execução com o script acima citado em ficheiros ***.txt*** , o que foi implementado na secção *main* do programa ***multi_ordered_tree.c.***

```
dt = cpu_time() - dt;
file_ct = fopen("creation_time.txt", "a"); // output tree creation time
printf("Tree creation time (%d persons): %.3es\n", n_persons, dt);
fprintf(file_ct, "%d %.3es\n", n_persons, dt);
fclose(file_ct);
```

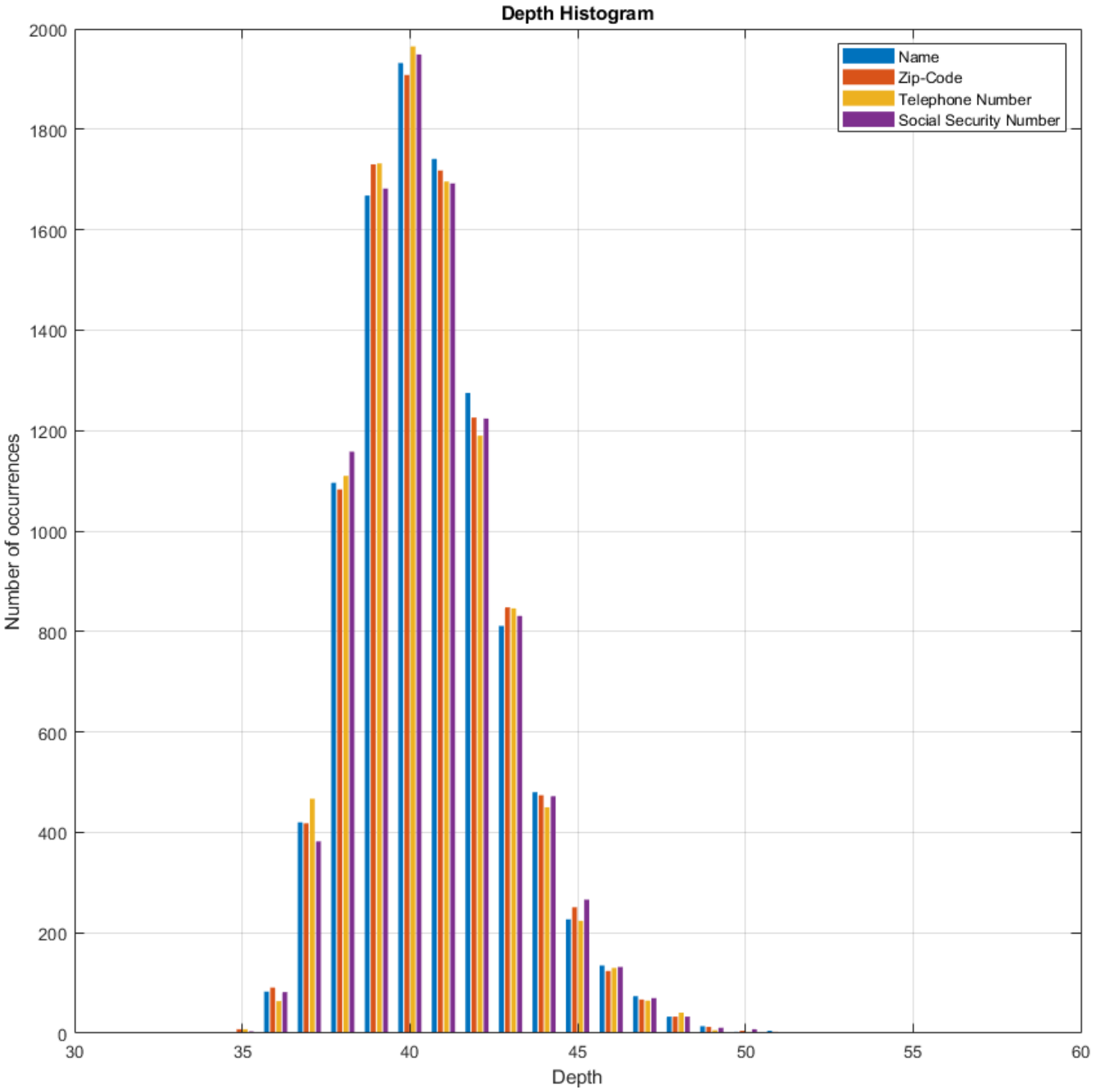
```
dt = cpu_time() - dt;
file_st = fopen("search_time.txt", "a"); // output tree search time
printf("Tree search time (%d persons, index %d): %.3es\n", n_persons, main_index,
dt);
fprintf(file_st, "%d %d %.3es\n", n_persons, main_index, dt);
fclose(file_st);
```

```
dt = cpu_time() - dt;
printf("Tree depth for index %d: %d (done in %.3es)\n", main_index, depth, dt);
fprintf(file_td, "%d %d %.3es\n", main_index, depth, dt); // output tree depth
fclose(file_td);
```

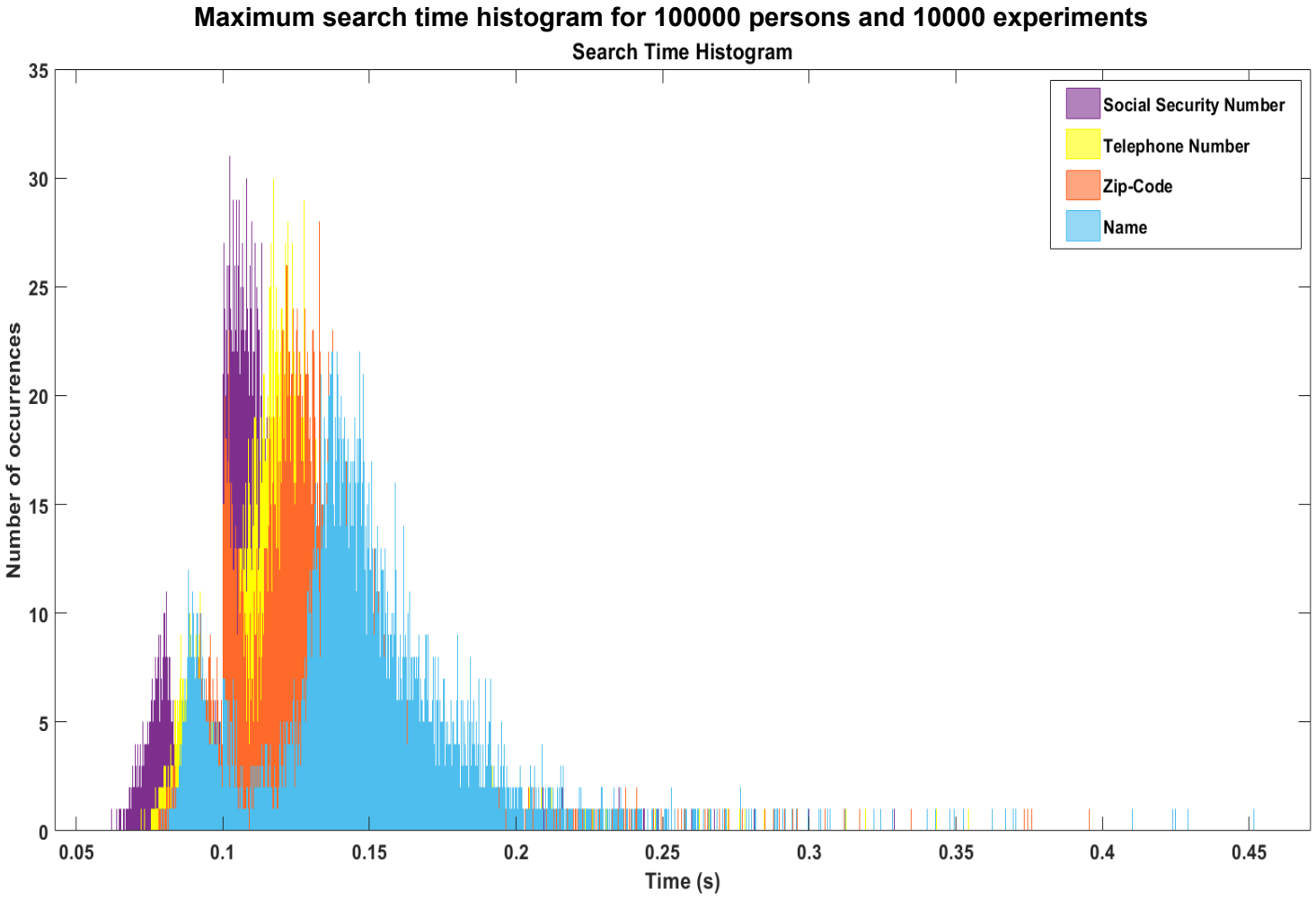
Gráficos

Tree Depth Histogram

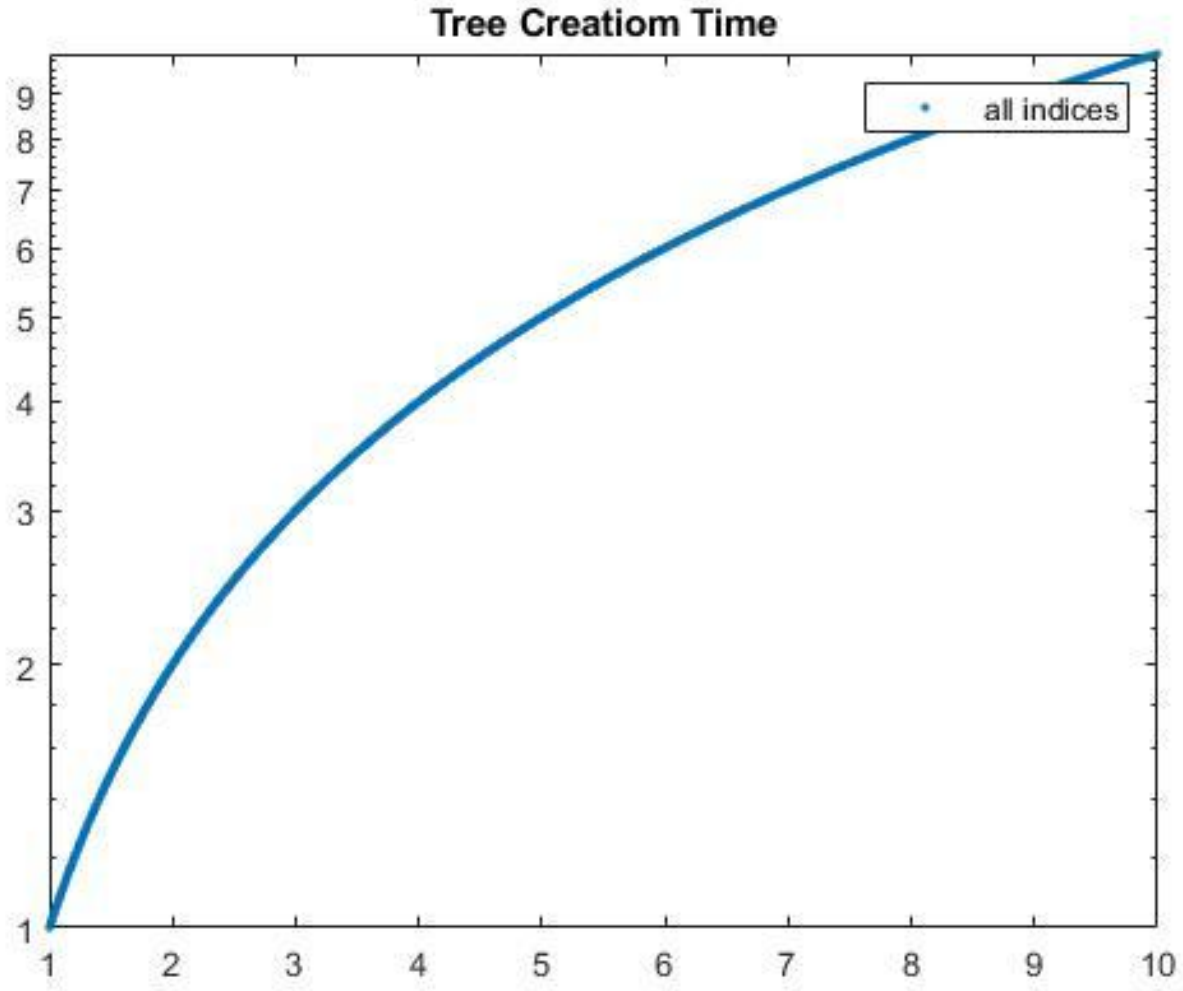
Maximum depth histogram for 100000 persons and 10000 experiments



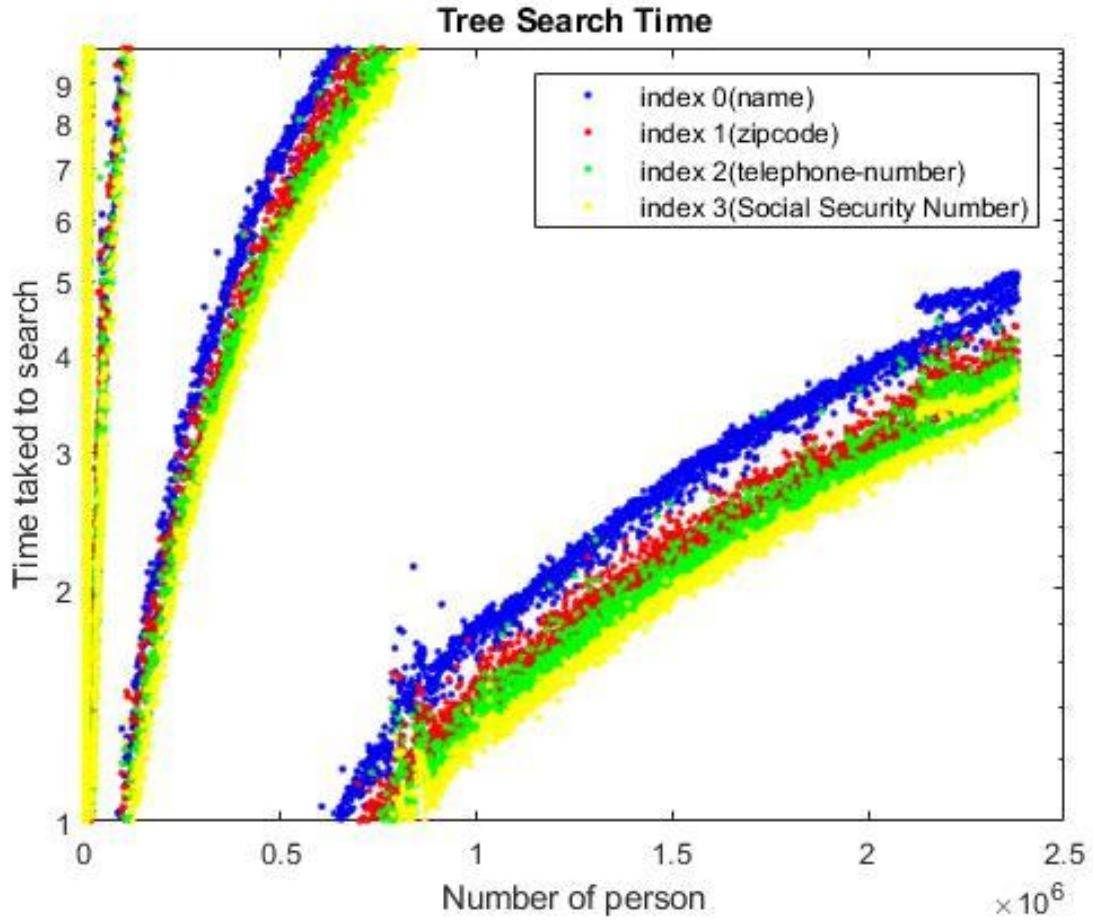
Search Time Histogram



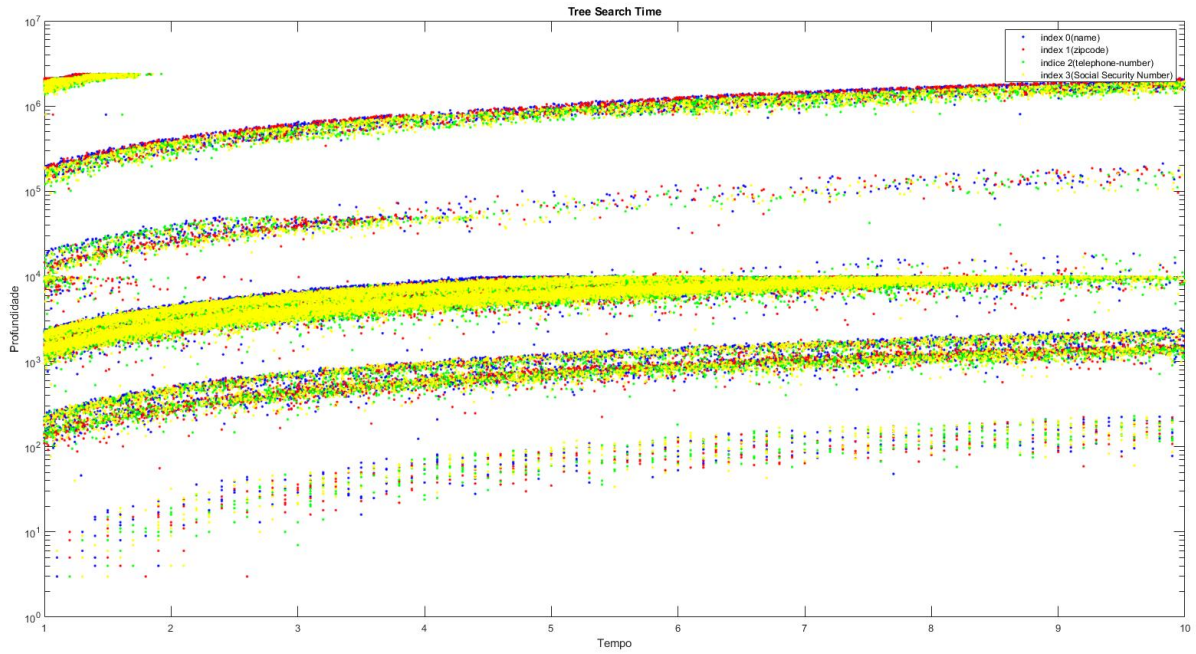
Tree Creation Time Graph (using 100480 nmec)



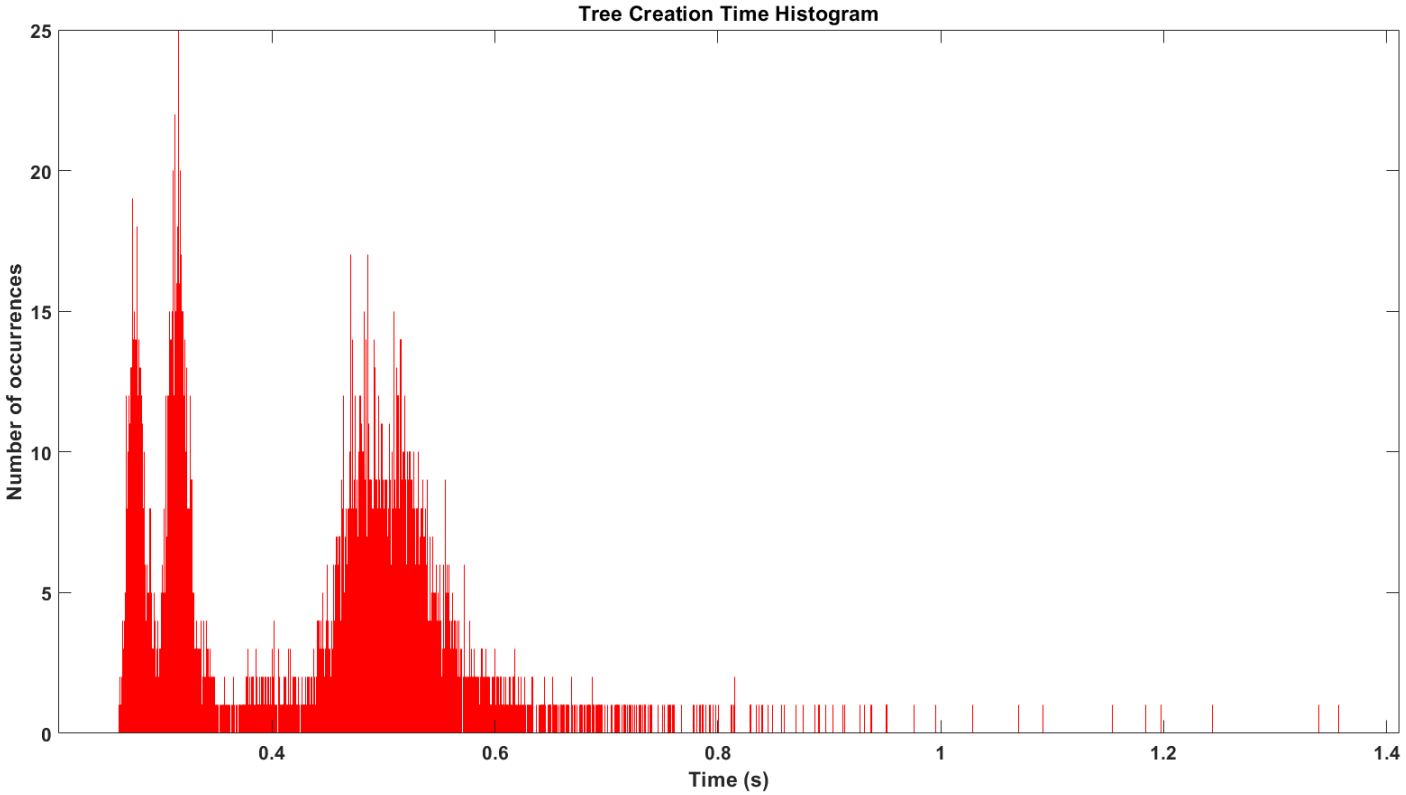
Tree Search Time Graph for 2381000 Persons (using 100480 nmec)



Tree Depth Graph for 2381000 persons(using 100480 nmec)



Tree Creation Time Histogram for 10.000 persons and 10.000 experiments (using 101099 nmec)



Tratamentos do Output

Da maneira como fizemos o trabalho foi necessário usar outras linguagens de programação para fazer o tratamento dos outputs que são necessários para fazer o gráfico usando o matlab

Bash

Para tratar o output usando o bash , desenvolvemos um script que elimina os caracteres que não serão usados como por exemplo: “+” “-” “e” , e eventuais espaços

```
#!/bin/bash
input=$1".txt"
while IFS= read -r line
do
  if [[ $line != '\n' ]]
  then
  echo $line | sed 's/e//' | cut -f1 -d "-" | cut -f1 -d "+"   >>
"cleaned_$1.txt"  ## -v, --invert-match select non-matching lines
  fi
done < "$input"
```

o script então lê cada linha do ficheiro que é passado como argumento , e cria um novo ficheiro .***txt*** com todas as alterações feitas.

Python

Da maneira como o script em matlab foi feito , também foi necessário usar o python para criar outros ficheiros ***.txt*** com os valores de cada índice,todo esse trabalho extra poderia ser simplesmente tratado com um melhor ajustamento no output do ”./multi_ordered_tree”, mas como já tínhamos os scripts rodando por horas , resolvemos então aproveita-las e tentar arranjar soluções , depois de usar o bash , ***python*** foi a solução mais fácil e rápida que conseguimos pensar

```
from asyncore import write
f = open("nomedoficheiro.txt", "r")
f1 = open("index0.txt", "a")
f2 = open("index1.txt", "a")
f3 = open("index2.txt", "a")
f4 = open("index3.txt", "a")

for x in f:
     for y in range(len(x)):

        if(x[y]==" "):
            indice=x[y+1]
            break
     if(indice=="0"):
      f1.write(x)
     if(indice=="1"):
      f2.write(x)
     if(indice=="2"):
      f3.write(x)
     if(indice=="3"):
      f4.write(x)
```

Este script em ***python***, abre o ficheiro desejado, e cria outros 3 ficheiros ***.txt*** com o conteúdo de cada índice.

Matlab

Foi utilizado para a plotagem e tratamento dos dados para os histogramas, onde importamos os dados dos ficheiros .***txt*** dados no output do nosso programa principal em ***C.***

```
%%Plotting maxTreeDepth_Time
dic=readtable('treeDepth_time_nmec.txt')
% nMec = [990000 : 1 : 999999];
depths = table2array(dic(:,2));
index = 1;
for i=1:4:40000
    depth1(index) = depths(i,1);
    depth2(index) = depths(i+1,1);
    depth3(index) = depths(i+2,1);
    depth4(index) = depths(i+3,1);
    index=index+1;
end
binspan=(34:56);
A1 = hist(depth1,binspan);
B1 = hist(depth2,binspan);
C1 = hist(depth3,binspan);
D1 = hist(depth4,binspan);
figure(1);
x = 34:1:56;
y = [A1;B1;C1;D1];
bar(x,y,EdgeColor="none");
axis square;
grid on;
xlabel("Depth");
ylabel("Number of occurrences");
title('Depth Histogram');
legend("Name","Zip-Code","Telephone Number", "Social Security Number");
%%Plotting search_time histogram
% search=readtable('search_Time_nmec.txt')
% passando a table para .txt para input facilitado
% writetable((search(:,3)), 'searchTimeHistogram.txt')
stimes = dlmread('searchTimeHistogram.txt');
index = 1;
for i=1:4:40000
    search1(index) = stimes(i,1);
    search2(index) = stimes(i+1,1);
    search3(index) = stimes(i+2,1);
    search4(index) = stimes(i+3,1);
    index=index+1;
end
figure(2);
```

```
histogram(search4,"BinWidth",0.000001,FaceColor="magenta",EdgeColor="mag
enta");
hold on
histogram(search3,"BinWidth",0.000001,FaceColor="yellow",EdgeColor="yell
ow");
hold on
histogram(search2,"BinWidth",0.000001,FaceColor="red",EdgeColor="red");
hold on
histogram(search1,"BinWidth",0.000001,FaceColor="blue",EdgeColor="blue")
;
title('Search Time Histogram');
xlabel("Time (s)");
ylabel("Number of occurrences");
legend("Social Security Number", "Telephone Number", "Zip-Code",
"Name");
hold off
```

```
Plotting Tree_Creation_Time Histogram
% creation=readtable('creation_time_nmec.txt')
% % passando a table para .txt para input facilitado
% writetable((creation(:,2)), 'creationTimeHistogram.txt')
a = dlmread('creationTimeHistogram.txt');
for i=1:10000
    c1(i) = a(i,1);
end
histogram(c1,"BinWidth",0.0001,EdgeColor="red");
xlabel("Time (s)");
ylabel("Number of occurrences");
title('Tree Creation Time Histogram');
```