universidade de aveiro

deti

Algoritmos e Estruturas de Dados

Um estudo de:

Multi-ordered Trees

André Butuc (103530) - 33.3%

Gonçalo Silva (103668) - 33.3%

João Matos (103182) - 33.3%

Índice

Introdução

O presente relatório enquadra-se no segundo trabalho prático para a cadeira de “Algoritmos e Estruturas de Dados”, tendo como objetivo o estudo relativo a uma estrutura de dados do tipo “Multi-ordered trees”. Inicialmente temos uma estrutura de dados “tree_node_t” que mapeia as informações de uma pessoa, nomeadamente o seu nome (index0), código postal (index1), número de telemóvel (index2) e número de segurança social (index3) em quatro árvores binárias ordenadas. Estas árvores coexistem na mesma estrutura, podendo-se pensar que se encontram “sobrepostas”.

Os resultados apresentados neste relatório relativos ao estudo da estrutura mencionada foram obtidos através da utilização e do desenvolvimento de funções, como, por exemplo, “tree_insert”, “find”, “tree_depth” e “list”, tendo sido utilizadas as funções “compare_tree_nodes”, “cpu_time” e funções geradoras de dados pseudo-aleatórios fornecidas pelo professor Tomás Silva.

Para cada função desenvolvida para a “multi-ordered tree” verificámos como o tempo de execução em segundos aumentava consoante o crescimento do número de pessoas presentes na árvore, tendo sido feita esta avaliação, percorrendo os números de pessoas espaçados na escala logarítmica desde o valor 10 até 10 000 000 pessoas, sendo para cada valor foram avaliados 100 valores de “seeds" (consideradas experiências nos gráficos deste relatório). Os valores obtidos na avaliação foram escritos em ficheiros de texto com uma formatação específica para facilitar o seu “parsing” no Matlab e facilitar a criação dos gráficos.

Em cada caso de estudo, criámos dois tipos de gráficos, sendo um o “plot” de pontos soltos de cada árvore binária ordenada e o outro histogramas seguindo a mesma estrutura do “plot”, sendo assim possível avaliar como o tempo de execução e o “maximum tree depth”, evoluem à medida do aumento do número de pessoas e à medida que a “seed” é alterada (nesta última é observado se os resultados são precisos, ou seja, se rondam o mesmo espetro de valores).

Criação das árvores

Para criar as árvores ordenadas foi utilizado a função “tree_insert”, esta função usufrui da função “compare_tree_nodes”, fornecida pelo professor Tomás Silva, que permite manter a propriedade da árvore sempre que novos “nodes” são inseridos na árvore. A função “tree_insert” é uma função recursiva que vai percorrendo a árvore binária em questão, avaliando em cada nível a propriedade de ordenação em “compare_tree_nodes”, até alcançar um espaço na mesma, identificado por “NULL”, passando depois o “node_data” a ser uma folha da árvore binária no espaço encontrado. A propriedade de ordenação é a seguinte: 1) se o “node_data” for inferior ao “node” que está na árvore, que na função “tree_insert” é dado por “rootptr[idx]”, então “desce-se” para o ramo à esquerda de “rootptr[idx]”; 2) se o “node_data” for superior ao “rootptr[idx], então “desce-se” para o ramo à direita de “rootptr[idx]”.

Como indicado na introdução avaliámos como evoluia o tempo de execução deste processo à medida que o tamanho do problema aumentava, tendo sido obtido os seguintes gráficos:

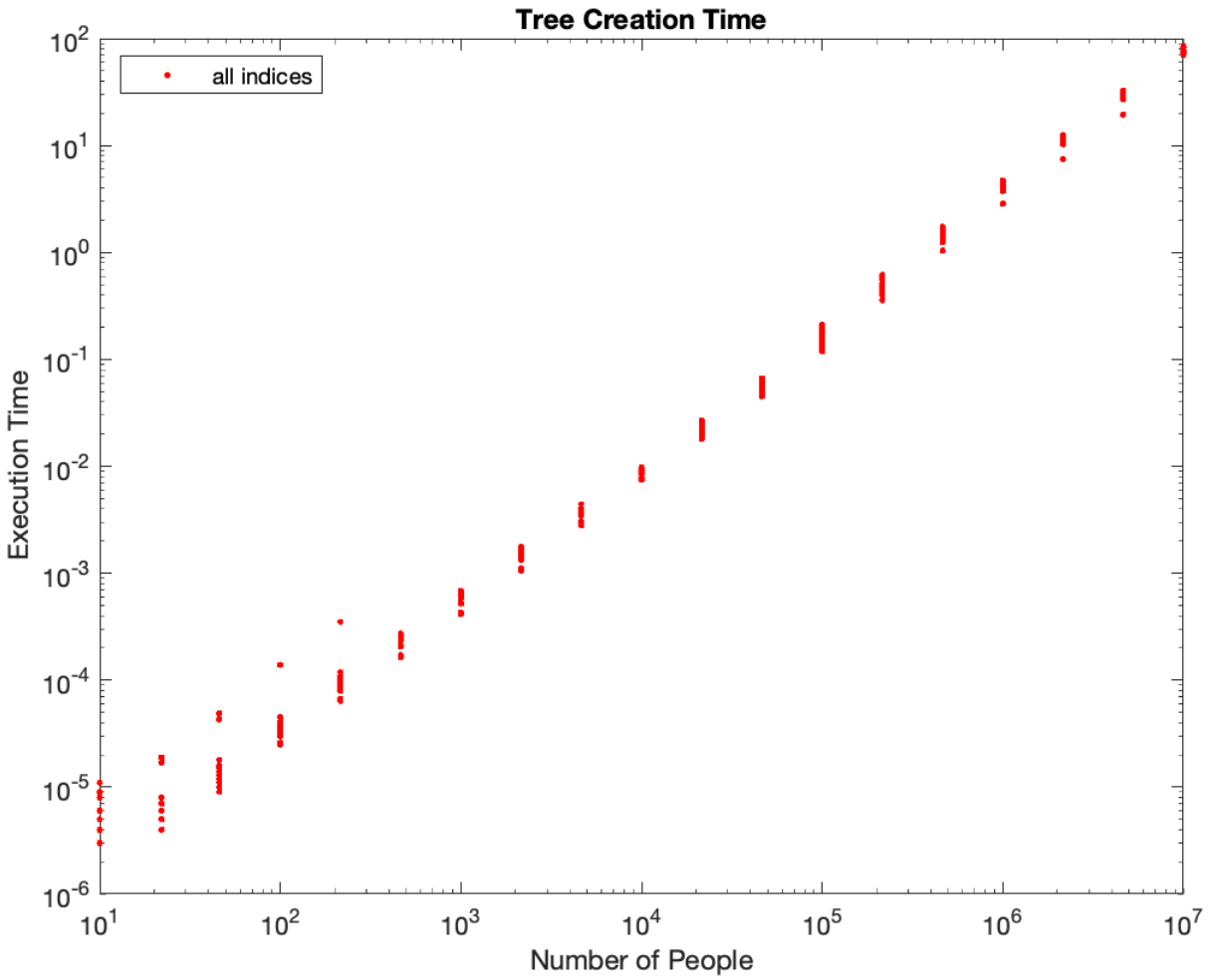

Figura 1 - Tempo de Criação da Estrutura “Multi-ordered tree”

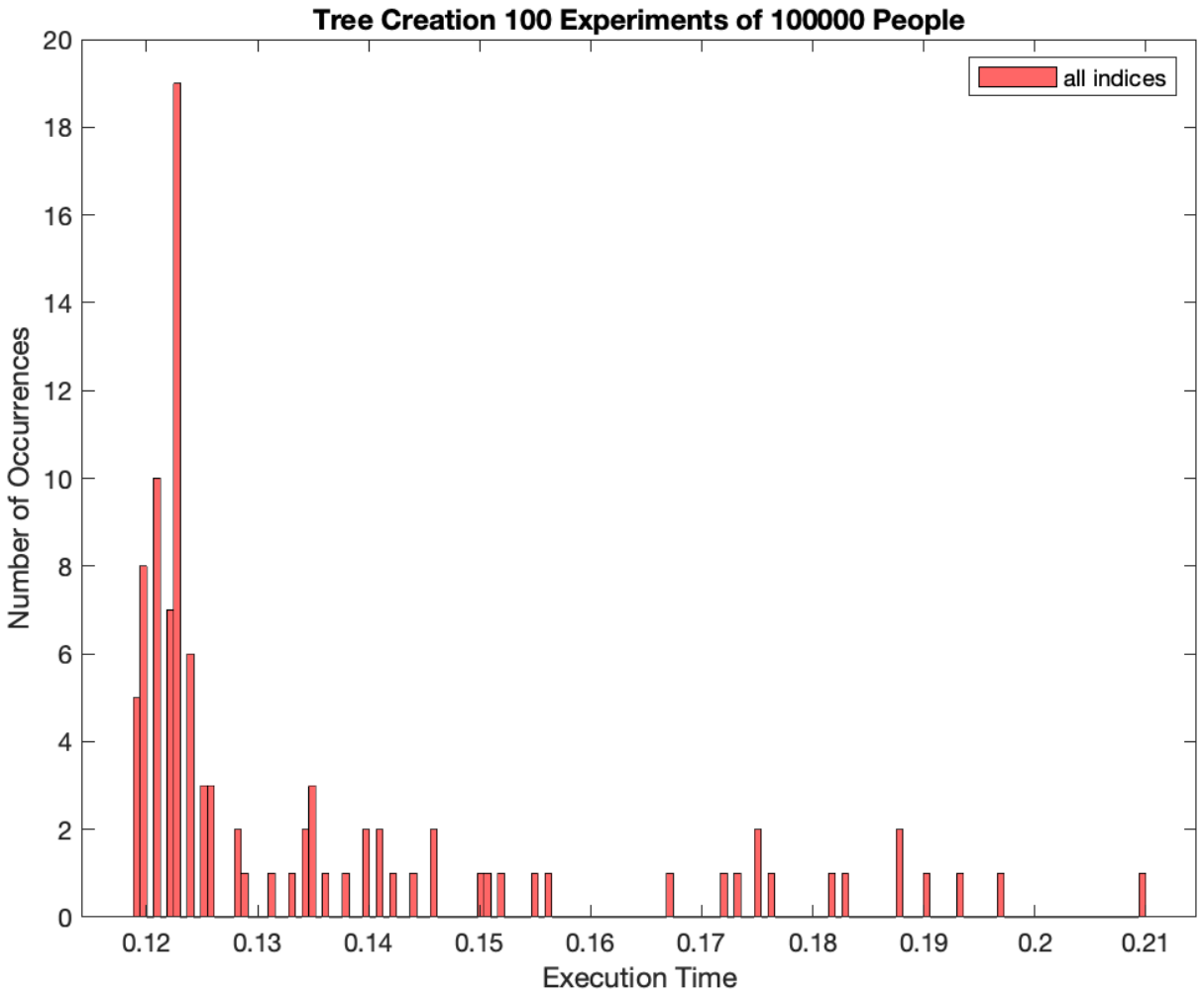

Figura 2 - Histograma do Tempo de Criação da Estrutura "Multi-ordered tree" para 100 000 pessoas

Na figura 1, podemos observar que o crescimento do tempo de execução aparenta ser, de facto, linear, sendo possível identificar alguns "pontos" soltos em determinados valores de números de pessoas que se afastam das restantes experiências realizadas. Para o valor 10 000 000 de pessoas, o tempo de execução ronda aproximadamente os 80 segundos.

Ao ampliarmos a figura 1, por exemplo nos tempos de execução equivalentes a 100 000 pessoas, iremos observar que os pontos, apesar de próximos, distam entre si, formando, à primeira vista, uma linha vertical. Para auxiliar no estudo da observação anterior, elaborámos um histograma (figura 2) que mapeia os tempos de execução para as estruturas que contêm 100 000 pessoas. Neste histograma os tempos de execução concentravam-se mais no intervalo de [0.12, 0.13] segundos, mas apresentam-se todos precisos entre si, sem desprezar que existem, como mencionado no parágrafo anterior, pontos "soltos", concluindo que são estes que contribuem para a ilusão da linha vertical.

Uma possível explicação para a existência de valores de tempos de execução que "fogem" à norma para o mesmo número de pessoas, é o fator da variação da "seed" que poderá provocar a geração de dados de pessoas que implicam mais comparações, mais chamadas à função "compare_tree_nodes" e, consequentemente, mais chamadas recursivas da função "tree_insert". Estes dados poderiam ter

por exemplo, o mesmo código postal, o que leva à intervenção das outras árvores binárias para criar um critério de desempate (funcionalidade própria da função “compare_tree_nodes”).

Pesquisa na árvore

A pesquisa na árvore é efetuada com a função recursiva “find” que tem como parâmetros de entrada um ponteiro “link” para a estrutura “multi-ordered tree” onde irá ser procurado o parâmetro “n”. Também é passado um inteiro “idx” que indica de qual árvore ordenada se trata (se é do nome, do código postal, do número de telemóvel ou do número da segurança social).

À semelhança da função “tree_insert” é utilizada a função “compare_tree_nodes” para avaliar se o caminho que teremos que percorrer para encontrar o “node” pretendido irá se seguir pelo ramo da esquerda ou pelo ramo da direita do “node” em que nos situamos pontualmente, sendo respeitados os critérios que levam a esta decisão, enunciados no tópico “Criação das árvores”.

A função recursiva apresenta duas condições de paragem, nomeadamente, se é atingida uma folha e ainda não foi encontrado o “node” pretendido, retornando dessa forma “NULL” ou então retorna o “node” pretendido quando este é encontrado, ou seja, quando o “compare_tree_nodes” retorna 0, indicando que o “node n” é igual ao “node link[idx]”.

O tempo de execução avaliado nesta função é referente ao tempo que demoram a ser pesquisados todos os “nodes” presentes nas árvores, tendo sido avaliado à semelhança da função “tree_insert”, como evoluia este tempo à medida que a dimensão do problema, nomeadamente o número de pessoas aumentava, tendo sido obtido os seguintes gráficos:

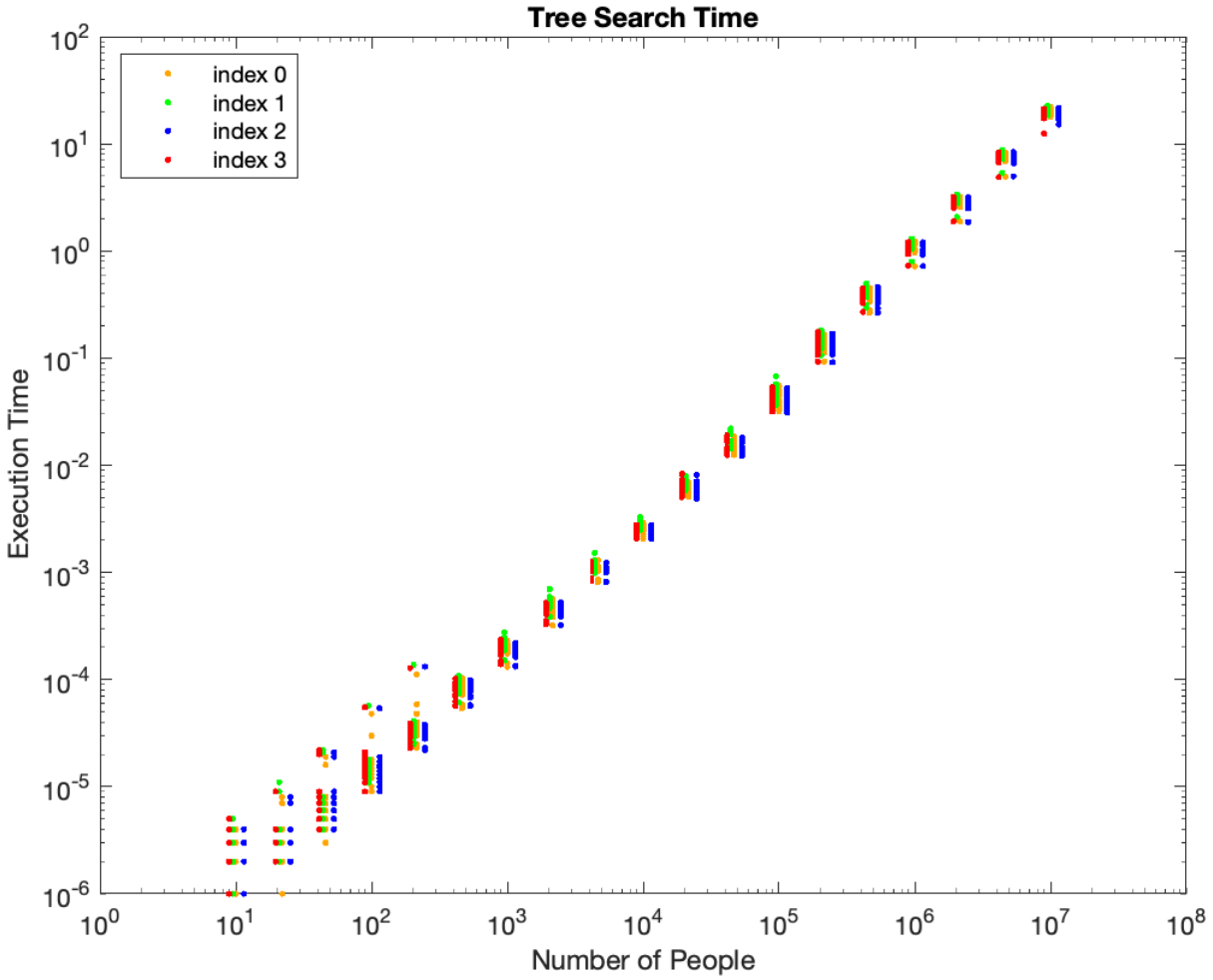

Figura 3 - Tempo de Pesquisa de todos os nodes das 4 Árvores Binárias Ordenadas

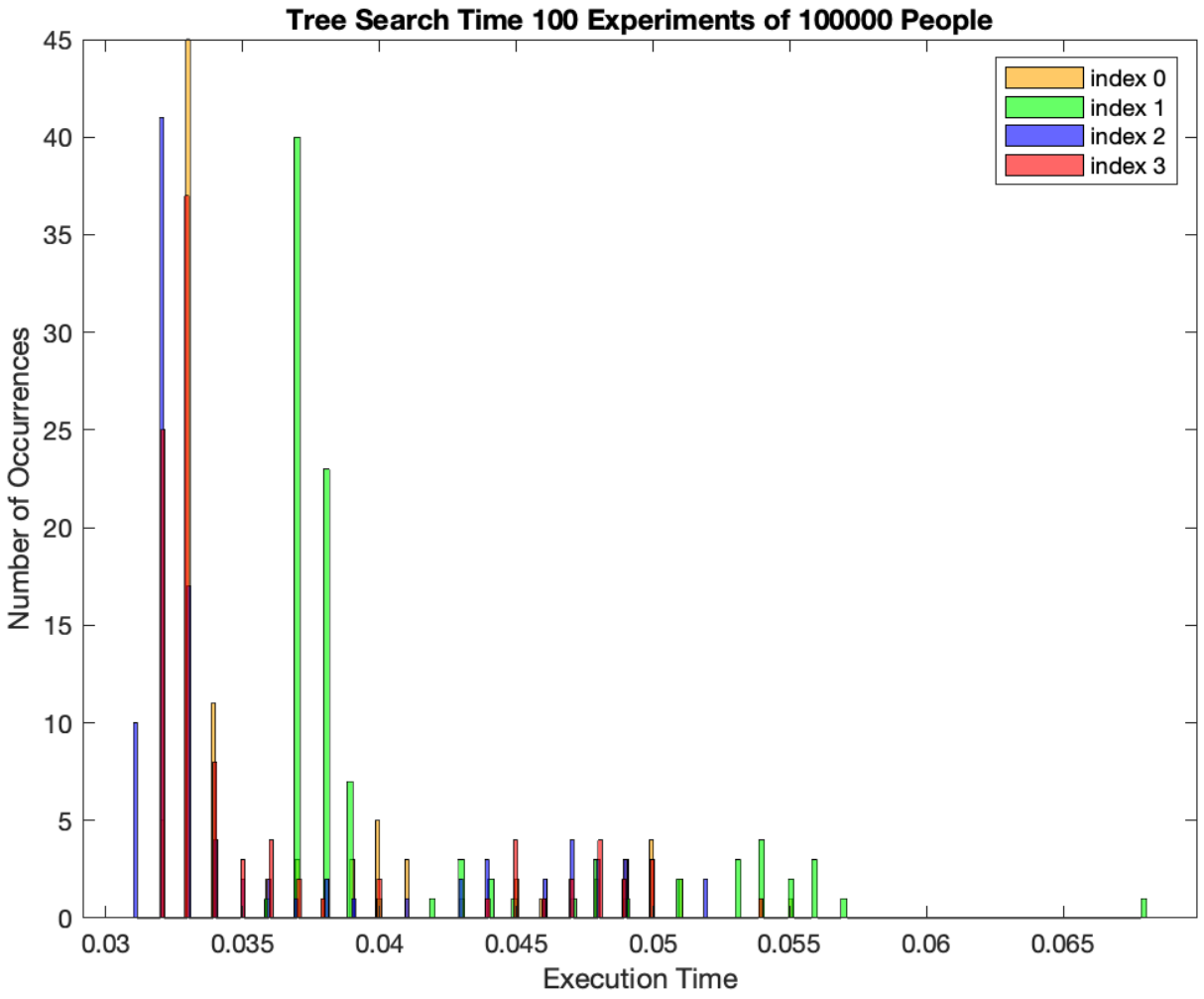

Figura 4 - Histograma dos Tempos de Pesquisa de todos os nodes da estrutura que contém 100 000 pessoas

Nota: na figura 3, o conjunto dos pontos index1, index2 e index3 foram desviados ao nível das suas abcissas para permitir a melhor visualização dos pontos (foi observado que, pelo facto de todas as árvores binárias ordenadas que integram a estrutura "multi-ordered trees" terem a mesma dimensão, sem a translação, os pontos e as cores dos mesmos misturavam-se, não permitindo o visionamento das propriedades pretendidas), logo o gráfico não representa inteiramente a veracidade dos dados.

Na figura 3 é observada a mesma propriedade que na figura 1, nomeadamente, o crescimento linear do tempo de execução. Faz de certa forma sentido esta propriedade permanecer, já que no processo de criação das árvores já existe um processo interno que "procura" o lugar para inserir os novos elementos. Sendo que a função "find" faz a mesma procura, mas para "conferir" a presença de um elemento, é expectável que ambos os processos tenham o mesmo crescimento temporal.

É também de notar que os resultados aparentam ser menos precisos para certos valores de “seeds” em números mais pequenos de pessoas (intervalo de 10 a 215), o que não acontece com tanta frequência em valores superiores de pessoas.

Por questões de comparação de trabalho computacional, para o valor 10 000 000 de pessoas, o tempo de execução da pesquisa ronda aproximadamente os 20 segundos, cerca de 4 vezes menor que o tempo de execução da criação da estrutura.

À semelhança da figura 2, também foi feito um “zoom” na região das 100 000 pessoas na figura 3 em formato de histograma. Neste histograma, a propriedade observada na figura 2 mantém-se, existem claramente aglomerados na generalidade das árvores de tempos de execução (da ordem dos 40) e acontecimentos pontuais de “seeds” que geram dados que exigem mais cálculos e instruções de comparação (este na ordem inferior aos aglomerados, nomeadamente ordem 5). Para além disso, o histograma permite observar de forma mais fácil (apesar de na figura 3 ser possível observar que em praticamente todos os números de pessoas os pontos verdes, referentes à árvore dos códigos postais, encontram-se mais acima) que a pesquisa é mais demorada na árvore dos códigos postais. A razão deve-se à quantidade reduzida de códigos postais (em comparação com a dimensão dos restantes dados), nomeadamente de 500 códigos postais, o que leva a que haja mais instruções de “desempate” em cada “node”, estas instruções consistem em verificar o conteúdo restante da pessoa, como por exemplo, o nome, para ser possível avaliar o percurso que é necessário tomar dentro da árvore até chegar ao “node” pretendido.

Profundidade máxima da árvore

O cálculo da profundidade máxima foi feito através da função “tree_depth”. Esta função, à semelhança das duas anteriores, é recursiva, sendo que em cada recursão a árvore é percorrida até chegar às suas folhas, quando esta condição é atingida, a profundidade da esquerda da árvore e a profundidade da direita são incrementadas de forma uniforme (fruto das condições que levam à incrementação). Quando é atingida a primeira chamada recursiva é avaliada qual profundidade será retornada, sendo que esta será a superior das duas, possibilitando assim a obtenção da profundidade máxima da árvore.

Para o estudo da profundidade máxima da árvore foi seguido o mesmo modelo gráfico anterior:

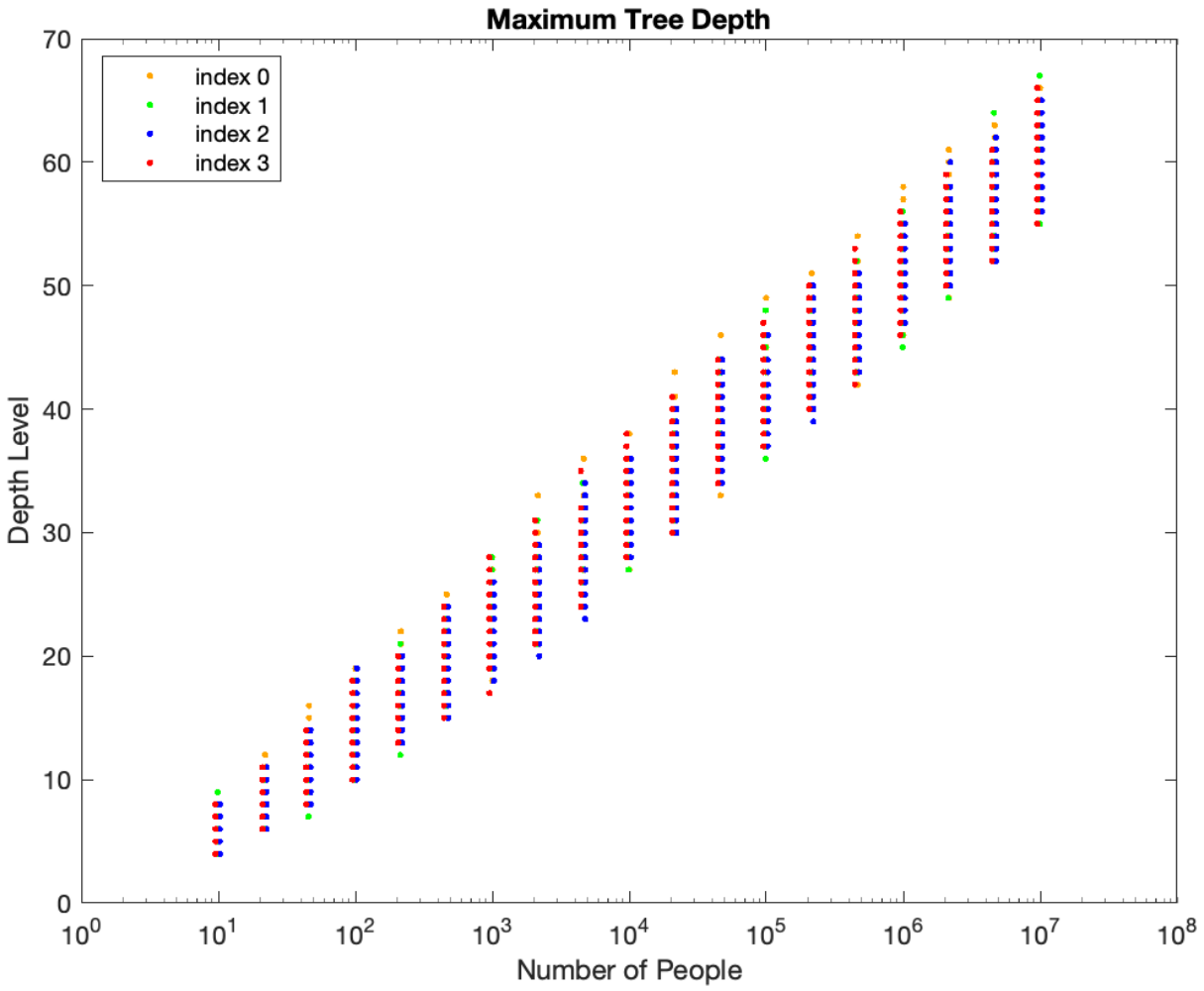

Figura 5 - Profundidade Máxima das 4 Árvores Binárias Ordenadas

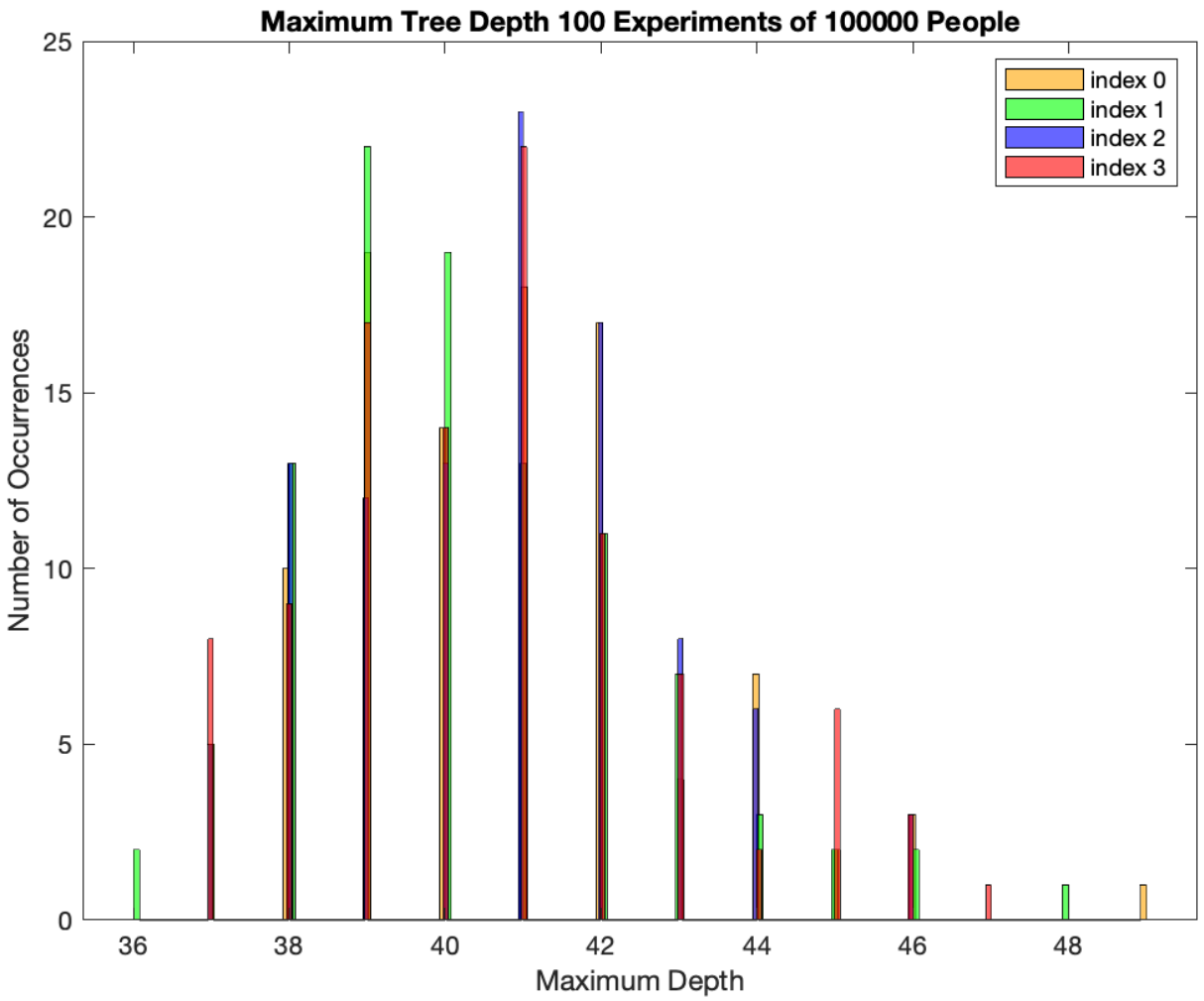

Figura 6 - Histograma da Profundidade Máxima das 4 Árvores Binárias Ordenadas

Nota: na figura 5, o conjunto dos pontos index1, index2 e index3 foram desviados ao nível das suas abcissas para permitir a melhor visualização dos pontos (foi observado que, pelo facto de todas as árvores binárias ordenadas que integram a estrutura "multi-ordered trees" terem a mesma dimensão, sem a translação, os pontos e as cores dos mesmos misturavam-se, não permitindo o visionamento das propriedades pretendidas), logo o gráfico não representa inteiramente a veracidade dos dados.

Na figura 5, a profundidade máxima cresce aparentemente de forma linear à medida que o número de pessoas aumenta. Mais uma vez, esta propriedade é expectável, pois quanto maior for a dimensão do problema, maior a probabilidade de existirem bifurcações que levam a mais ramificações e a consequente expansão da árvore e o aumento dos seus "níveis".

A figura 6, consiste num histograma, que "amplia" a porção do gráfico referente ao número 100 000 de pessoas. No histograma ocorre o problema mencionado na nota, no entanto, pelo facto de acontecer indica-nos que entre as árvores, a profundidade máxima não é muito diferente. Os mesmos comentários, relativos à precisão dos histogramas anteriores (figuras 2 e 4), aplicam-se ao presente histograma.

Em termos da análise dos tempos de execução da função “tree_depth”, obtemos os seguintes gráficos:

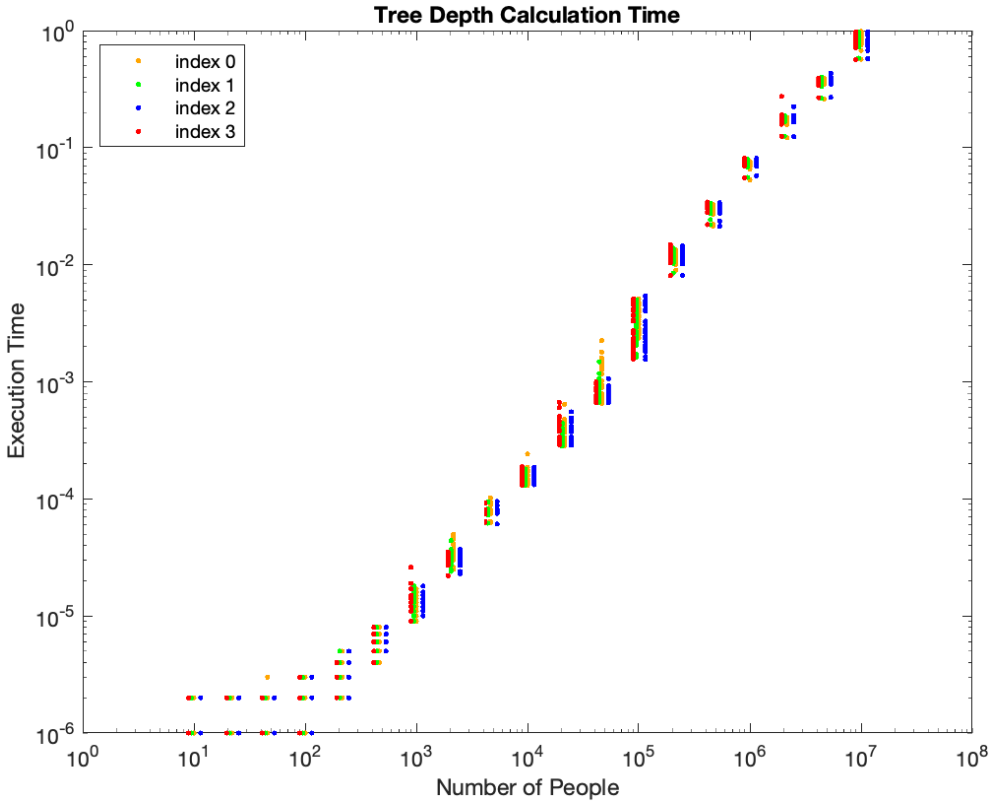

Figura 7 - Tempo do Cálculo da Profundidade Máxima das 4 Árvores Binárias Ordenadas

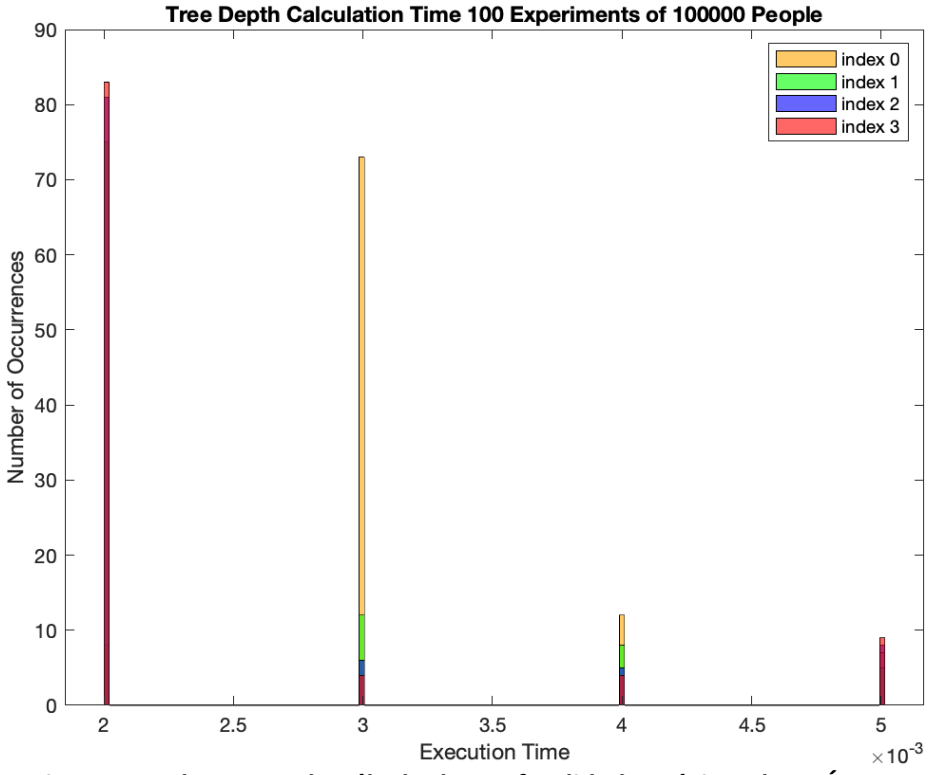

Figura 8 - Histograma do Tempo de Cálculo da Profundidade Máxima das 4 Árvores Binárias Ordenadas

Pelo facto da figura 7 e 8 tratarem de tempos que são praticamente residuais, sendo que até para 10 000 000 pessoas o tempo de execução ronda aproximadamente os 0.8 segundos (o que em comparação dos tempos de execução para 10 000 000 anteriores é vestigial), o estudo do tempo de execução, neste caso de estudo não é relevante.

Listagem seletiva

A funcionalidade da listagem seletiva foi obtida através da reutilização da função “list” que inicialmente listava os elementos da árvores binária ordenada em questão, que era passado como argumento em formato de “-list{indice_arvore}”, se fosse, por exemplo “-list0”, listava somente o conteúdo da árvore dos nomes de forma ordenada (neste caso, por ordem alfabética):

```
List of persons:
Person #1
        name= Daniel Kelly
        zip_code= 10314 Staten Island (Richmond county)
        telephone_number= 6670 529 034
        Security Social number= 128 04 7654
Person #2
        name= Danielle Pacheco
        zip_code= 94541 Hayward (Alameda county)
        telephone_number= 1117 025 781
        Security Social number= 517 80 7685
Person #3
        name= Derek Davis
        zip_code= 95111 San Jose (Santa Clara county)
        telephone_number= 3509 685 800
        Security Social number= 593 56 6701
Person #4
        name= Herbert Wilson
        zip_code= 93722 Fresno (Fresno county)
        telephone_number= 8374 899 815
        Security Social number= 718 18 5145
Person #5
        name= Jillian Hernandez
        zip_code= 78521 Brownsville (Cameron county)
        telephone_number= 6027 019 168
        Security Social number= 167 10 3859
Person #6
        name= Nellie Thornton
        zip_code= 77479 Sugar Land (Fort Bend county)
        telephone_number= 3698 410 846
        Security Social number= 104 88 9348
Person #7
        name= Pasquale Rivera
        zip_code= 79936 El Paso (El Paso county)
        telephone_number= 7053 252 516
        Security Social number= 287 48 1431
Person #8
        name= Robert Dawson
        zip_code= 60639 Chicago (Cook county)
        telephone_number= 5359 287 312
        Security Social number= 434 29 2075
Person #9
        name= Sara Silva
        zip_code= 19143 Philadelphia (Philadelphia county)
        telephone_number= 2044 584 709
        Security Social number= 015 96 2060
Person #10
        name= Wayne Marsh
        zip_code= 78640 Kyle (Hays county)
        telephone_number= 8298 014 010
        Security Social number= 339 09 9297
```

Figura 9 - Print do resultado da função “list”, com a opção seletiva

No entanto, para adicionar a funcionalidade pretendida adicionámos dois parâmetros extra, nomeadamente “int val” (que permite saber se a funcionalidade está ativa e, caso esteja, que tipo de seleção se trata) e “char *str” (a condição que define a listagem seletiva, nomeadamente os dados da pessoa que irá ser imprimida, deverá conter os dados contidos em str) .

A variável “val” é obtida através da identificação da “flag -select key_str” nos argumentos, sendo que a expressão “key_str” é passada numa função “validate” que identifica o tipo de informação de que

se trata (ou seja, se é um nome, código postal, número de telemóvel ou segurança social) que retorna um valor indicador do campo em questão.

De seguida, tendo os dados necessários para a funcionalidade extra, dentro da função é utilizada a função interna “strcmp” para avaliar caso o “node” que esteja na presente instrução recursiva contêm os dados “key_str”, caso contenha é chamada uma função “print_info”, desenvolvida pelo grupo, que imprime os dados das pessoas que cumprem a condição necessária.

De forma a testar a eficiência da função de listagem seletiva criamos uma estrutura “multi-ordered tree” com 10 000 000 pessoas, pesquisamos o nome “Derek Devis” e registámos o tempo que demorou, tendo sido este de aproximadamente 1 segundo para 56 “matches”, como pode ser observado na figura 10:

```
        Security Social number= 322 22 3293
Person #54
        name= Derek Davis
        zip_code= 956 Bayamon (Bayamon county)
        telephone_number= 9744 963 663
        Security Social number= 952 30 8622
Person #55
        name= Derek Davis
        zip_code= 95608 Carmichael (Sacramento county)
        telephone_number= 8248 105 908
        Security Social number= 913 93 4063
Person #56
        name= Derek Davis
        zip_code= 99301 Pasco (Franklin county)
        telephone_number= 1339 998 267
        Security Social number= 686 83 3640
1.071e+00
```

Figura 10 - Print do tempo de execução da função “list”, com a opção seletiva, “Derek Davis”

Avaliação da disposição das árvores

Para avaliarmos como as árvores na estrutura “multi-ordered tree” estavam dispostas, elaborámos uma função chamada “count_nodes” que em cada nível da árvore contava todos os seus nós. Esta função recursiva permitiu-nos avaliar o formato em termos de “largura” e “expansão” da árvore, sendo que esta, por exemplo para as árvores dos nomes de 10 milhões de pessoas, entre os níveis médios de 19-39 apresentava-se extremamente mais densas do que nos seus pólos, como se pode observar na seguinte figura:

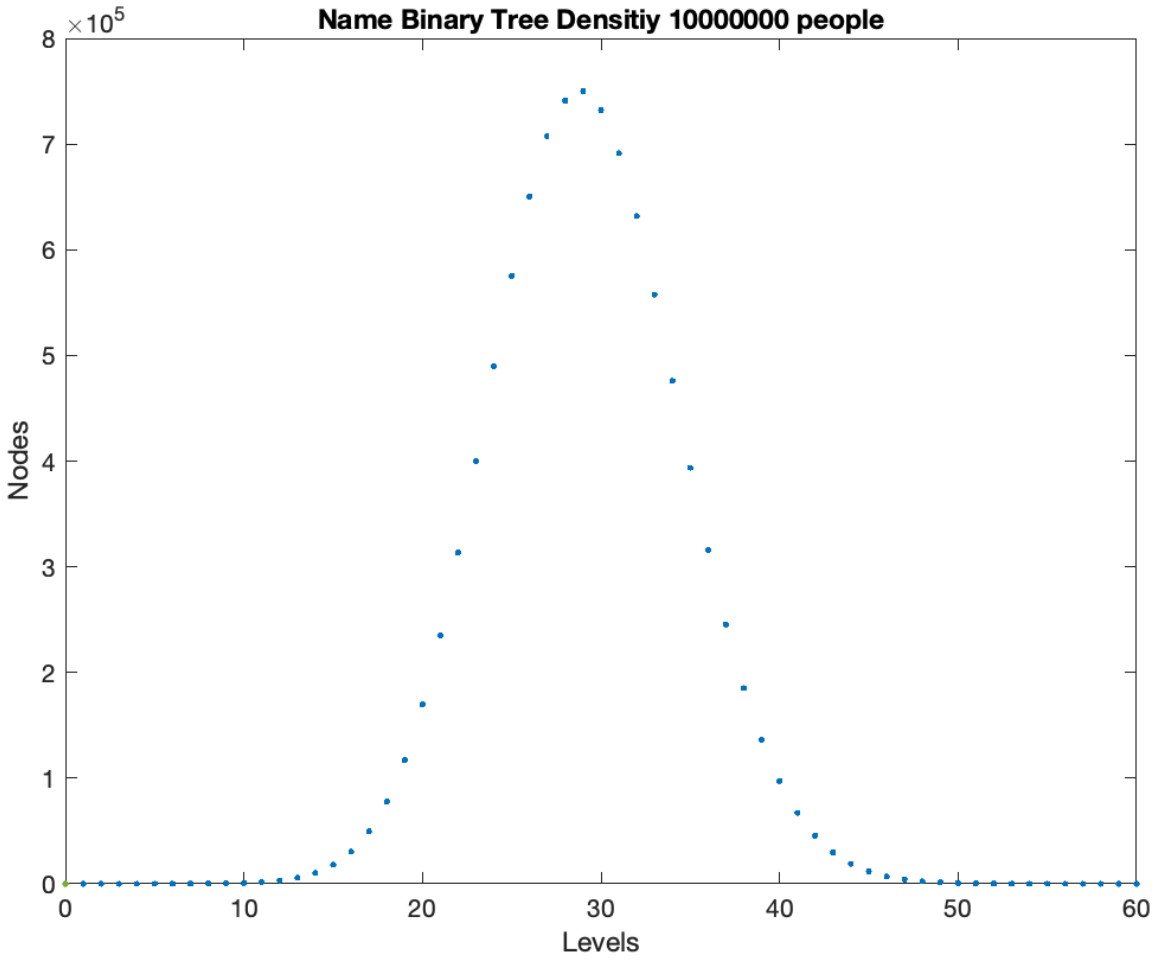

Figura 11 - Dispersão dos nodes pelos níveis da árvore binária dos nomes para 10 milhões de pessoas

Como se pode observar pela figura 11, o momento de maior densidade é praticamente a metade da árvore binária.

Conclusão

Após o estudo minucioso de cada propriedade da estrutura de dados “multi-ordered” tree, nomeadamente o tempo de execução da sua criação, pesquisa e profundidade podemos concluir que, de facto, esta estrutura é extremamente eficaz no armazenamento de grandes quantidades de informação. Se colocarmos a dimensão do problema em perspetiva, ao ser possível criar e pesquisar informação na escala de 10 milhões de pessoas, no fundo estamos a conseguir representar uma nação inteira, por exemplo, Portugal, numa estrutura de complexidade não muito elevada como é a árvore binária ordenada.

São nestes casos de estudos que realmente observamos a importância da escolha adequada de estruturas de dados de armazenamento de informação que tanto permitem expandir ao longo de tempo e de forma rápida (podemos imaginar por exemplo uma base de dados para uma rede social de rápido crescimento), como também aceder à mesma em tempo real adequado. De facto, o resultado da figura 10 enquadra-se com o ponto de vista anteriormente mencionado, já que para um utilizador normal, 1 segundo é um tempo de execução bastante bom para fazer uma pesquisa de um nome numa lista de 10 milhões de pessoas, passando por despercebido aos olhos de qualquer pessoa.

Em conclusão, este trabalho foi bastante útil para consolidar a matéria teórica que foi exposta nas aulas teóricas relativamente ao tópico das árvores binárias e nos permitiu compreender como estas funcionam internamente e desenvolver funções recursivas que podemos fazer para as estudar e usar.

Código Desenvolvido

```
// AED, January 2022
//
// Solution of the second practical assignement (multi-ordered tree)
//
// Place your student numbers and names here
// André Butuc-103530
// Gonçalo Silva-103668
// João Matos-103182

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "AED_2021_A02.h"

// 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 upo with 3 three roots

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[MAX_SOCIAL_SECURITY_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;

//
// 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 < 4; 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, node2->social_security);
   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 **rootptr, tree_node_t *node_data, int idx) // idx=
index--main_index
{
 if (rootptr[idx] == NULL)
 {
   rootptr[idx] = node_data;
   return;
 }
 else if (compare_tree_nodes(node_data, rootptr[idx], idx) < 0)
 {
   tree_insert(rootptr[idx]->left, node_data, idx);
 }
 else if (compare_tree_nodes(node_data, rootptr[idx], idx) > 0)
 {
   tree_insert(rootptr[idx]->right, node_data, idx);
 }
 return;
 //  node1/node2 são argumentos de passagem da função compare_tree_nodes
 //  compare_tree_nodes pode returnar 0 (conteudo do node1 já está na tree), um valor <0
(conteudo de node1 tem um codigo ASCII mais "pequeno" que o do node2,
 //  logo descemos na arvore para a esquerda do node2) eum valor >0 (conteudo de node1 tem
um codigo ASCII "maior" que o do node2, logo descemos na arvore para
 //  a direita do node2)
}
//
// tree search routine (place your code here)
//
```

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

//
// tree depth
//
// tree depth= height+1
int tree_depth(tree_node_t **rootptr, int idx)
{
 if (rootptr[idx] == NULL)
 {
   // arvore não existe
   return 0;
 }
 int left_depth = tree_depth(rootptr[idx]->left, idx);
 int right_depth = tree_depth(rootptr[idx]->right, idx);

 if (left_depth > right_depth)
 {
   return left_depth + 1;
 }
 else
 {
   return right_depth + 1;
 }
}

void print_info(tree_node_t **link, int *cont, int idx)
{
 *cont = *cont + 1;
 printf("Person #%d\n", *cont);
 printf("\tname= %s\n", link[idx]->name);
 printf("\tzip_code= %s\n", link[idx]->zip_code);
 printf("\ttelephone_number= %s\n", link[idx]->telephone_number);
```

```
 printf("\tSecurity Social number= %s\n", link[idx]->social_security);
}
//  list, i,e, traverse the tree (place your code here)
//
void list(tree_node_t **link, int idx, int *cont, int val, char *str)
{

 if (link[idx] != NULL)
 {
   if (link[idx]->left)
   {
     list(link[idx]->left, idx, cont, val, str);
   }

   if (val >= 0)
   {
     if (val == 0)
     {
       if (strcmp(str, link[idx]->name) == 0)
       {
         print_info(link, cont, idx);
       }
     }
     if (val == 1)
     {
       if (strcmp(str, link[idx]->zip_code) == 0)
       {
         print_info(link, cont, idx);
       }
     }
     if (val == 2)
     {
       if (strcmp(str, link[idx]->telephone_number) == 0)
       {
         print_info(link, cont, idx);
       }
     }
     if (val == 3)
     {
       if (strcmp(str, link[idx]->social_security) == 0)
       {
         print_info(link, cont, idx);
       }
     }
   }
   else
   {
     print_info(link, cont, idx);
```

```
   }

   if (link[idx]->right)
   {
     list(link[idx]->right, idx, cont, val, str);
   }
 }
}

int count_nodes(tree_node_t **roots, int current_level, int level, int main_index)
{
 if (roots[main_index] == NULL)
   return 0;
 if (current_level == level)
   return 1;

 return count_nodes(roots[main_index]->left, current_level + 1, level, main_index) +
count_nodes(roots[main_index]->right, current_level + 1, level, main_index);
}

int validate(char *str)
{

 if (isalpha(str[0]))
 {
   // printf("nome\n");
   return 0; // codigo-->name
 }
 else if (str[5] == ' ')
 {
   // printf("zip code\n");
   return 1; // codigo-->zip code
 }
 else if (str[4] == ' ')
 {
   // printf("telephone\n");
   return 2; // codigo-->telephone
 }
 else if (str[3] == ' ')
 {
   // printf("social security\n");
   return 3; // codigo-->social security
 }
 return -1;
}

//
// 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;

 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_n[] = {10, 22,  46, 100,  215,  464,  1000, 2154, 4642, 10000,  21544,
46416,  100000, 215443, 464159, 1000000,  2154435,  4641589,  10000000};
 // int n_persons;

 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
 // FILE *fcreation = fopen("tree_creation.txt", "w");
 // FILE *fsearch = fopen("tree_search.txt", "w");
 // FILE *fdepth = fopen("tree_depth.txt", "w");
 // for (student_number=10000; student_number < 10100; student_number++){
 //   for (int index=0; index<19; index++){
 //     n_persons=n_persons_n[index];

 tree_node_t *persons = (tree_node_t *)calloc((size_t)n_persons, sizeof(tree_node_t));
 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(&(persons[i].social_security[0]));

   for (int j = 0; j < 3; 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, &(persons[i]), main_index); // place your code here to insert
&(persons[i]) in the tree with number main_index

 // completar AQUI

 dt = cpu_time() - dt;
 printf("Tree creation time (%d persons): %.3es\n", n_persons, dt);
 //fprintf(fcreation, "%d  %.3e\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, 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;
   //fprintf(fsearch, "%d  %d  %.3e\n",n_persons, main_index, 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); // 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(fdepth, "%d  %d  %d  %.3e\n",n_persons, main_index, depth,dt);
}
// process the command line optional arguments
int main_index = 0;
char str[50];
int val = -2;
int cont = 0;
for (int i = 3; i < argc; i++)
{
  if (strncmp(argv[i], "-list", 5) == 0)
  { // list all (optional)
    main_index = atoi(&(argv[i][5]));
    if (main_index < 0)
      main_index = 0;
    if (main_index >= 3)
      main_index = 3;
  }
  // place your own options here
  if (strncmp(argv[i], "-select", 7) == 0)
  {
    strcpy(str, argv[i + 1]);
    val = validate(str);
    if (val == -1)
    {
      printf("String de pesquisa com formato inválido\n");
      exit(1);
    }
  }
}

printf("List of persons:\n");
(void)list(roots, main_index, &cont, val, str); // place your code here to traverse, in
order, the tree with number main_index

if (cont == 0)
{
  printf("No match found\n");
}
```

```
 int level, current_level, n_nodes;

 current_level = 0;
for (int main_index = 0; main_index < 1; main_index++)
 {
   n_nodes = 0;
   level = 0;
   int depth = tree_depth(roots, main_index);
   if (main_index == 0)
     printf("name_tree");
   if (main_index == 1)
     printf("zip_code_tree");
   if (main_index == 2)
     printf("telephone_tree");
   if (main_index == 3)
     printf("social_security_tree");

   for (int i = 0; i < depth; i++)
   {
     n_nodes = count_nodes(roots, current_level, level, main_index);
     fprintf(f, "%d  %d\n", level, n_nodes);
     level++;
   }
 }


 // clean up --- don't forget to test your program with valgrind, we don't want any memory
leaks
 free(persons);
 // fclose(fcreation);
 // fclose(fsearch);
 // fclose(fdepth);
 return 0;
}
```

"tree_creation.m"

```
clear

clc

close all

fn = "tree_creation.txt";

f = fopen(fn);

count = 1;

pessoas = zeros(1, 1900);

tempos = zeros(1, 1900);

while 1
```

```
    tline = fgetl(f);
    if ~ischar(tline), break, end
    data = split(tline);
    pessoas(count) = str2double(data{1,1});
    tempos(count) = str2double(data{2,1});
    count = count+1;
end
loglog(pessoas, tempos, ".r");
legend("all indices");
xlabel("Number of People")
ylabel("Execution Time")
legend('Location', 'northwest');
title("Tree Creation Time")
fclose(f);
```

tree_creation_histogram.m

```
clc
close all
clear
filename = "tree_search.txt";
f = fopen(filename);
target = 100000;
tempos0 = zeros(1, 100);
count0 = 1;
tempos1 = zeros(1, 100);
count1 = 1;
tempos2 = zeros(1, 100);
count2 = 1;
tempos3 = zeros(1, 100);
count3 = 1;
while 1
    tline = fgetl(f);
    if ~ischar(tline), break, end
    data = split(tline);
    if (str2double(data{1,1}) == target)
        index = str2double(data{2,1});
        tempo = round(str2double(data{3,1}),3);

        if index == 0
            tempos0(count0) = tempo;
            count0 = count0 + 1;
        elseif index == 1
            tempos1(count1) = tempo;
```

```
            count1 = count1 + 1;
        elseif index == 2
            tempos2(count2) = tempo;
            count2 = count2 + 1;
        elseif index == 3
            tempos3(count3) = tempo;
            count3 = count3 + 1;
        end
    end
end
histogram(tempos0, 150, "Facecolor", "#FFA500")
hold on
histogram(tempos1, 150, "Facecolor", "g")
histogram(tempos2, 150, "Facecolor", "b")
histogram(tempos3, 150, "Facecolor", "red")
hold off
title("Tree Search Time 100 Experiments of 100000 People")
xlabel("Execution Time")
ylabel("Number of Occurrences")
legend("index 0", "index 1", "index 2", "index 3")
fclose(f);
```

tree_depth.m

```
clear
clc
close all
fn = "tree_depth.txt";
f = fopen(fn);
pessoas = zeros(1, 1900);
depth0 = zeros(1, 1900);
count0 = 1;
depth1 = zeros(1, 1900);
count1 = 1;
depth2 = zeros(1, 1900);
count2 = 1;
depth3 = zeros(1, 1900);
count3 = 1;
while 1
    tline = fgetl(f);
    if ~ischar(tline), break, end
    data = split(tline);
    pessoa = str2double(data{1,1});
    index = str2double(data{2,1});
```

```
    depth = str2double(data{3,1});
    tempo = str2double(data{4,1});
    if index == 0
        pessoas(count0) = pessoa;
        depth0(count0) = depth;
        count0 = count0 + 1;
    elseif index == 1
        depth1(count1) = depth;
        count1 = count1 + 1;
    elseif index == 2
        depth2(count2) = depth;
        count2 = count2 + 1;
    elseif index == 3
        depth3(count3) = depth;
        count3 = count3 + 1;
    end
end
semilogx(pessoas, depth0, ".", "Color", "#FFA500")
hold on
semilogx(pessoas-pessoas*0.010, depth1, ".g")
semilogx(pessoas+pessoas*0.025, depth2, ".b")
semilogx(pessoas-pessoas*0.045, depth3, ".", "Color", "red")
hold off
legend("index 0", "index 1", "index 2", "index 3");
xlabel("Number of People")
ylabel("Depth Level")
legend('Location', 'northwest');
title("Maximum Tree Depth")
fclose(f);
```

tree_depth_histogram.m

```
clc
close all
clear
filename = "tree_depth.txt";
f = fopen(filename);
target = 100000;
tempos0 = zeros(1, 100);
count0 = 1;
tempos1 = zeros(1, 100);
count1 = 1;
tempos2 = zeros(1, 100);
count2 = 1;
```

```
tempos3 = zeros(1, 100);
count3 = 1;
while 1
   tline = fgetl(f);
   if ~ischar(tline), break, end
   data = split(tline);
   if (str2double(data{1,1}) == target)
       index = str2double(data{2,1});
       depth = round(str2double(data{3,1}),3);

       if index == 0
           tempos0(count0) = depth;
           count0 = count0 + 1;
       elseif index == 1
           tempos1(count1) = depth;
           count1 = count1 + 1;
       elseif index == 2
           tempos2(count2) = depth;
           count2 = count2 + 1;
       elseif index == 3
           tempos3(count3) = depth;
           count3 = count3 + 1;
       end
   end
end
histogram(tempos0, 150, "Facecolor", "#FFA500")
hold on
histogram(tempos1, 150, "Facecolor", "g")
histogram(tempos2, 150, "Facecolor", "b")
histogram(tempos3, 150, "Facecolor", "red")
hold off
title("Maximum Tree Depth 100 Experiments of 100000 People")
xlabel("Maximum Depth")
ylabel("Number of Occurrences")
legend("index 0", "index 1", "index 2", "index 3")
fclose(f);
```

tree_depth_time.m

```
clear
clc
close all
fn = "tree_depth.txt";
f = fopen(fn);
```

```
pessoas = zeros(1, 1900);

tempos0 = zeros(1, 1900);

count0 = 1;

tempos1 = zeros(1, 1900);

count1 = 1;

tempos2 = zeros(1, 1900);

count2 = 1;

tempos3 = zeros(1, 1900);

count3 = 1;

while 1

   tline = fgetl(f);

   if ~ischar(tline), break, end

   data = split(tline);

   pessoa = str2double(data{1,1});

   index = str2double(data{2,1});

   tempo = str2double(data{4,1});

   if index == 0

       pessoas(count0) = pessoa;

       tempos0(count0) = tempo;

       count0 = count0 + 1;

   elseif index == 1

       tempos1(count1) = tempo;

       count1 = count1 + 1;

   elseif index == 2

       tempos2(count2) = tempo;

       count2 = count2 + 1;

   elseif index == 3

       tempos3(count3) = tempo;

       count3 = count3 + 1;

   end

end

loglog(pessoas, tempos0, ".", "Color", "#FFA500")

hold on

loglog(pessoas-pessoas*0.050, tempos1, ".g")

loglog(pessoas+pessoas*0.15, tempos2, ".b")

loglog(pessoas-pessoas*0.105, tempos3, ".", "Color", "red")

hold off

legend("index 0", "index 1", "index 2", "index 3");

xlabel("Number of People")

ylabel("Execution Time")

legend('Location', 'northwest');

title("Tree Depth Calculation Time")

fclose(f);
```

tree_depth_time_histogram.m

```
clc
close all
clear
filename = "tree_depth.txt";
f = fopen(filename);
target = 100000;
tempos0 = zeros(1, 100);
count0 = 1;
tempos1 = zeros(1, 100);
count1 = 1;
tempos2 = zeros(1, 100);
count2 = 1;
tempos3 = zeros(1, 100);
count3 = 1;
while 1
   tline = fgetl(f);
   if ~ischar(tline), break, end
   data = split(tline);
   if (str2double(data{1,1}) == target)
       index = str2double(data{2,1});
       tempo = round(str2double(data{4,1}),3);

       if index == 0
           tempos0(count0) = tempo;
           count0 = count0 + 1;
       elseif index == 1
           tempos1(count1) = tempo;
           count1 = count1 + 1;
       elseif index == 2
           tempos2(count2) = tempo;
           count2 = count2 + 1;
       elseif index == 3
           tempos3(count3) = tempo;
           count3 = count3 + 1;
       end
   end
end
histogram(tempos0, 150, "Facecolor", "#FFA500")
hold on
histogram(tempos1, 150, "Facecolor", "g")
histogram(tempos2, 150, "Facecolor", "b")
```

```
histogram(tempos3, 150, "Facecolor", "red")
hold off
title("Tree Depth Calculation Time 100 Experiments of 100000 People")
xlabel("Execution Time")
ylabel("Number of Occurrences")
legend("index 0", "index 1", "index 2", "index 3")
fclose(f);
```

tree_search.m

```
clear
clc
close all
fn = "tree_search.txt";
f = fopen(fn);
pessoas = zeros(1, 1900);
tempos0 = zeros(1, 1900);
count0 = 1;
tempos1 = zeros(1, 1900);
count1 = 1;
tempos2 = zeros(1, 1900);
count2 = 1;
tempos3 = zeros(1, 1900);
count3 = 1;
while 1
   tline = fgetl(f);
   if ~ischar(tline), break, end
   data = split(tline);
   pessoa = str2double(data{1,1});
   index = str2double(data{2,1});
   tempo = str2double(data{3,1});
   if index == 0
       pessoas(count0) = pessoa;
       tempos0(count0) = tempo;
       count0 = count0 + 1;
   elseif index == 1
       tempos1(count1) = tempo;
       count1 = count1 + 1;
   elseif index == 2
       tempos2(count2) = tempo;
       count2 = count2 + 1;
   elseif index == 3
```

```
        tempos3(count3) = tempo;
        count3 = count3 + 1;
    end
end
loglog(pessoas, tempos0, ".", "Color", "#FFA500")
hold on
loglog(pessoas-pessoas*0.050, tempos1, ".g")
loglog(pessoas+pessoas*0.15, tempos2, ".b")
loglog(pessoas-pessoas*0.105, tempos3, ".", "Color", "red")
hold off
legend("index 0", "index 1", "index 2", "index 3");
xlabel("Number of People")
ylabel("Execution Time")
legend('Location', 'northwest');
title("Tree Search Time")
fclose(f);
```

tree_search_histogram.m

```
clc
close all
clear
filename = "tree_creation.txt";
f = fopen(filename);
count = 1;
target = 100000;
tempos = zeros(1, 100);
while 1
    tline = fgetl(f);
    if ~ischar(tline), break, end
    data = split(tline);
    if (str2double(data{1,1}) == target)
        tempos(count) = round(str2double(data{2,1}),3);
        count = count+1;
    end
end
histogram(tempos, 150, "Facecolor", "r")
title("Tree Creation 100 Experiments of 100000 People")
xlabel("Execution Time")
ylabel("Number of Occurrences")
legend("all indices")
fclose(f);

clear
```

nodes.m

```
close all

clc

fn = "nodes.txt";

f = fopen(fn);

count = 1;

node = zeros(61);

levels = zeros(61);

while 1

   tline = fgetl(f);

   if ~ischar(tline), break, end

   data = split(tline);

   levels(count) = str2double(data{1,1});

   node(count) = str2double(data{2,1});

   count = count+1;

end

figure(1);

plot(levels, node, ".");

title("Name Binary Tree Densitiy 10000000 people")

xlabel("Levels");

ylabel("Nodes")
```