Algoritmos e Estruturas de Dados 2021/2022

Multi-Ordered Binary Trees

Segundo Trabalho Prático da disciplina de AED.
Professores: Tomás Oliveira e Silva; Pedro Lavrador

Trabalho Realizado por: 102435 Rafael Remígio 50%;
104360 João Correia 50%;

universidade de aveiro

Índice

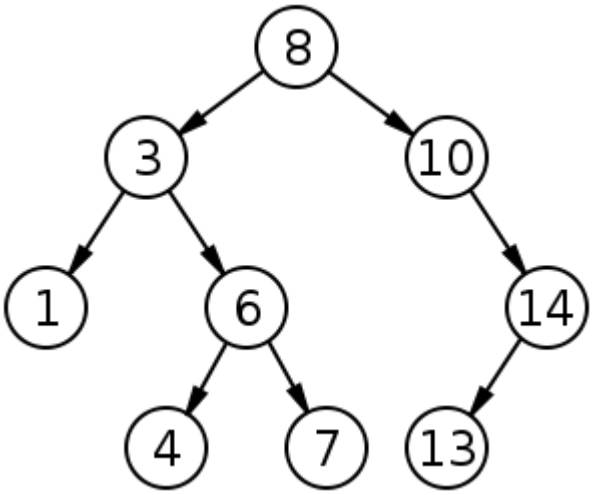
Introdução

Uma das áreas mais importantes da Ciência da Computação e no desenvolvimento de Software é o armazenamento e processamento de informação de forma eficaz e pouco dispendiosa. Diferentes tipos de estruturas de dados oferecem diversas vantagens e desvantagens. Neste trabalho focaremos apenas em Árvores Binárias, nas vantagens da sua utilização tal como na implementação de diversos algoritmos para percorrer e criar estas Árvores.

Árvore Binária:

Neste caso falaremos de Árvores Binárias de Busca que podem ser definidas por um conjunto de Nós onde cada Nó possuí a seguinte informação:

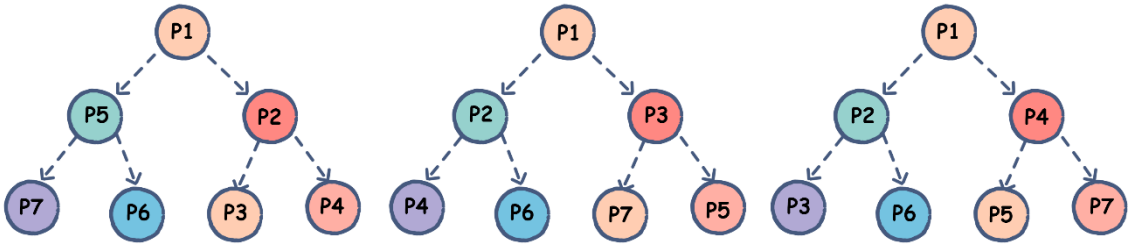
- Dados a guardar
- Ponteiro para o ramo da esquerda (left child)
- Ponteiro para o ramo da direita (right child)
- Ponteiro para o Parente **(Opcional)**
- Todos os Nós da subárvore da esquerda possuem um valor inferior ao da raiz e todos os Nós da subárvore da direita possuem um valor superior ao da raiz



Neste trabalho prático foi nos pedido a organização em árvores binarias de Nós com três campos:

- Nome
- Zip Code
- Número de Telefone

É necessária a criação de 3 árvores, uma por cada campo, sendo que deve ser possível percorrer, pesquisar e saber o tamanho de cada Árvore criada.

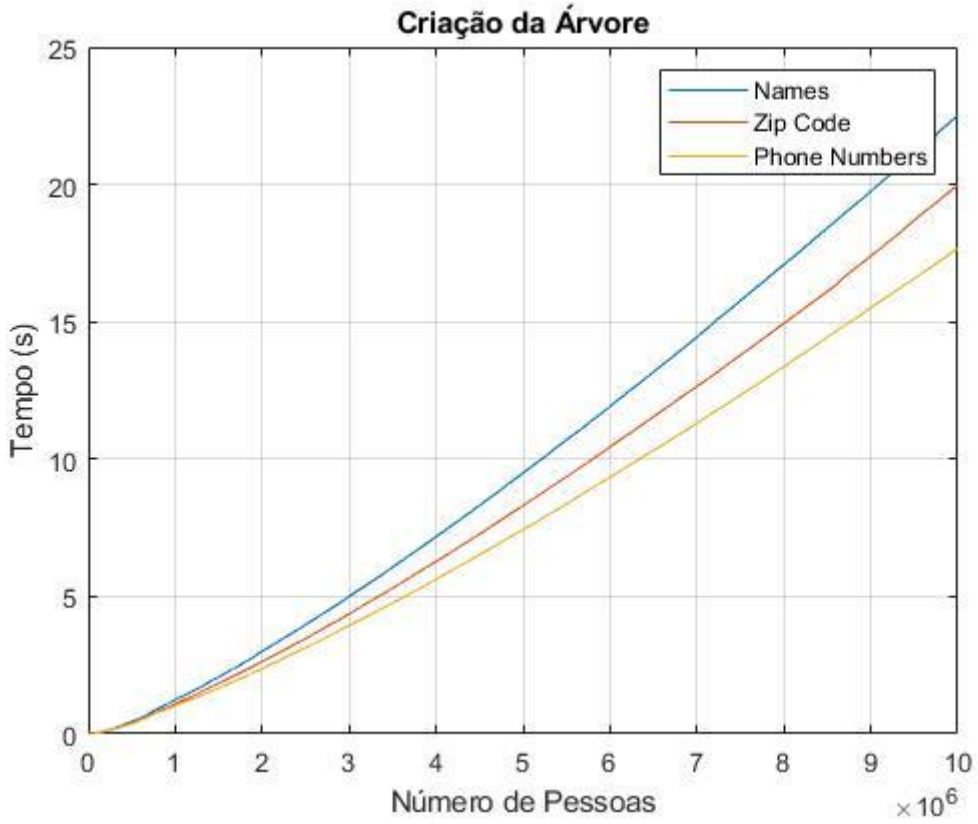


Inserção de elementos

A operação de inserção deve ter em cuidado os princípios da árvore binária de busca, sendo então necessário adicionar novos nodes sempre como folhas da árvore.

- Percorremos a árvore recursivamente comparando o valor adicionar com o valor do Nó onde nos encontramos. Caso o Nó tiver um valor maior que o node atual tentamos adicionar à direita caso contrário tentamos adicionar a esquerda. Desde modo o Nó é sempre inserido numa posição de folha não pondo em causa a estrutura da árvore
- A comparação é realizada relativamente ao campo específico através de uma função que recebe as duas Pessoas e o índex relativo há árvore que estamos a criar comparando destes modos os campos desejados. Se os campos forem iguais a função irá comparar outro campo.
- As três árvores diferentes são criadas através de um array de ponteiros, onde cada índex é referente a uma árvore diferente com diferentes campos a serem comparados

Esta operação tem de complexidade temporal em média O(log(n)) sendo o pior caso O(n). A criação completa da árvore tem de complexidade O(n*log(n)).



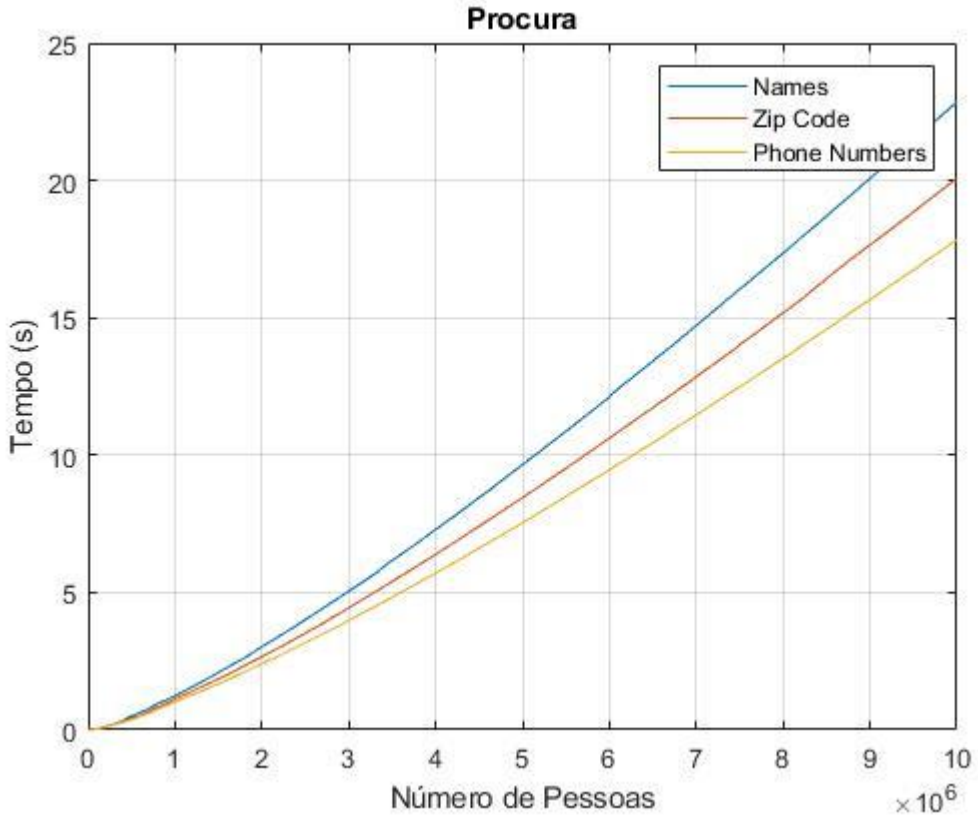

```
void tree_insert( tree_node_t** rootp, tree_node_t* node,int main_idx)
{
  if ( *rootp == NULL){
    *rootp = node;
    return;
  }
  int c = compare_tree_nodes(*rootp,node,main_idx);
  if (c < 0)
  {
    tree_insert(&((*rootp)->right[main_idx]), node,main_idx);
    return;
  }
  else
  {
    tree_insert(&((*rootp)->left[main_idx]), node,main_idx);
    return;
  }
  return;

}
```

Procura

De forma semelhante à inserção na árvore, a procura de elementos também tira proveito das propriedades de árvores binárias de busca.

- Percorremos a árvore recursivamente comparando o valor da pessoa a procurar com o valor do Nó onde nos encontramos. Se este for igual devolvemos este Nó, se for maior ou menor movemo-nos para a direita ou esquerda da árvore respetivamente.
- Se encontrarmos um Nó inexistente, indica que chegamos ao fim da árvore, isto indica que a pessoa que procurávamos não existe dentro de esta árvore
- As três árvores diferentes são procuradas através de um array de ponteiros, onde cada índex é referente a uma árvore diferente com diferentes campos a serem comparados.

A procura tal como a inserção, tem de complexidade temporal em média O(log(n)) sendo o pior caso O(n). A procura de todos os elementos da árvore tem complexidade temporal O(n*log(n)).

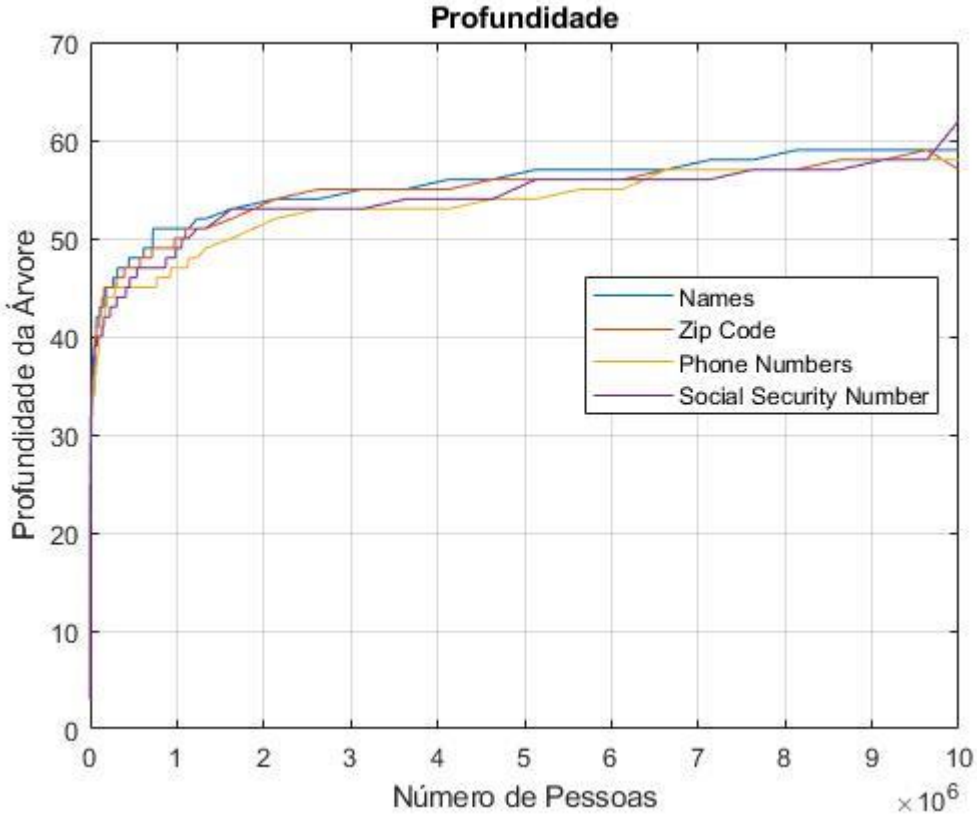


```
tree_node_t* find(tree_node_t** rootp,int main_idx,tree_node_t* person)
{
  if ((*rootp) == NULL){
    printf("here");
    return NULL;
  }
  if (compare_tree_nodes(*rootp,person,main_idx)==0)
  {
    return *rootp;
  }
  else if (compare_tree_nodes(*rootp,person,main_idx) > 0)
  {
    return find(&((*rootp)->left[main_idx]),main_idx,person);
  }
  else
  {
    return find(&((*rootp)->right[main_idx]),main_idx,person);
  }
  return NULL;
}
```

Altura/Profundidade da árvore

O tempo de inserção e procura de itens na árvore é impactado pelo tamanho desta, quão menor for a árvore menor é o caminho a percorrer até chegar a um node específico. Sendo que as pessoas a serem inseridas são dadas por ordem pseudoaleatória o tamanho da árvore não será o mínimo. No melhor caso a árvore terá profundidade log(n).

- A altura da árvore é descoberta percorrendo de forma recursiva todos os nodes até chegarmos a um node inexistente, retornando então 0, e retornando o máximo da altura da subárvore esquerda e da direita mais 1.
- O algoritmo da descoberta do tamanho de uma árvore tem de complexidade computacional O(n)



```c
int tree_depth(tree_node_t** root, int main_idx)
{
  if ( *root == NULL){
    return 0;
  }
  int leftheight = tree_depth(&((*root)->left[main_idx]),main_idx);
  int rightheight = tree_depth(&((*root)->right[main_idx]),main_idx);

  if (leftheight > rightheight)
  {
    return leftheight + 1;
  }
  else{
    return rightheight + 1 ;
  }
}
```

Listagem

A listagem das diferentes pessoas por ordem relativa a um respetivo campo pode ser feita de diversas formas de modo a conseguir diferentes tipos de listagens. Neste problema decidimos implementar 3 diferentes listagens, uma **listagem por largura (breadth)**, uma **listagem por altura de pré-ordem(preorder)** e uma **listagem por altura ordem simétrica(inorder).**

- **Breadth search –** Implementada através da utilização da estrutura de dados fila (queue), onde visitamos as Pessoas no fim da fila e adicionamos nós no final da fila.

```
Multiordered_Binary_Trees./remigio 12323 3
----------------------
----------------------
Name ----------------- Lisa Walker
ZipCode ------------------ 30188 Woodstock (Cherokee county)
Telephone Number ----------------- 3036 087 680
----------------------
----------------------
Name ----------------- Diana Stewart
ZipCode ------------------ 60651 Chicago (Cook county)
Telephone Number ----------------- 1407 114 127
----------------------
----------------------
Name ----------------- Ann Case
ZipCode ------------------ 94110 San Francisco (San Francisco county)
Telephone Number ----------------- 8319 442 913
```

```c
void traverse_breadth_first(tree_node_t *link,int main_idx)
{
  queue q1;
  init_queue(&q1);
  enqueue(&q1,link);
  while(q1.head != NULL)
  {
    link = dequeue(&q1);
    if(link != NULL)
    {
      visit_node(link);
      enqueue(&q1,link->left[main_idx]);
      enqueue(&q1,link->right[main_idx]);
    }
  }
}

// queue implementation

typedef struct node {
    tree_node_t* val;
    struct node *next;
} node_q;
```

```c
typedef struct queue
{
    node_q* head;
    node_q* tail;
} queue;


void init_queue(queue* q){
    q->head = NULL;
    q->tail = NULL;
}

void enqueue(queue* q, tree_node_t* value) {
    // create new node
    node_q * new_node = malloc(sizeof(node_q));
    if (new_node==NULL) { return;}    //if malloc fails
    new_node->val = value;
    new_node->next = NULL;

    // if there is a tail we just connect that tail to this node
    if (q->tail != NULL){
        q->tail->next = new_node;
    }
    q->tail = new_node;

    // if there is no head we set this one also as head
    if (q->head == NULL) {
        q->head = new_node;
    }
    return;
}

tree_node_t* dequeue(queue* q) {
    if (q->head == NULL) {return NULL;}
    node_q * tmp = q->head;
    tree_node_t* result = tmp->val;
    q->head = q->head->next;
    if (q->head == NULL){
        q->tail = NULL;
    }
    free(tmp);
    return result;
}
```

- **Depth pre-order search –** Percorre-se recursivamente todos os nós visitando primeiro a atual raiz, seguida da subárvore à esquerda e depois a subárvore à direita.

```
Multiordered_Binary_Trees./remigio 123235 3
----------------------
----------------------
Name ----------------- Linda Franco
ZipCode ----------------- 43081 Westerville (Franklin county)
Telephone Number ----------------- 7091 352 001
----------------------
----------------------
Name ----------------- Jeffrey Arellano
ZipCode ----------------- 75034 Frisco (Collin county)
Telephone Number ----------------- 4288 790 986
----------------------
----------------------
Name ----------------- Taylor Burnett
ZipCode ----------------- 77479 Sugar Land (Fort Bend county)
Telephone Number ----------------- 1112 205 779
```

```c
int list_pre_order(tree_node_t* node,int main_idx)
{

  if (node !=NULL){
    visit_node(node);
    if (node->left[main_idx] != NULL){
      list_pre_order(node->left[main_idx],main_idx);
    }
    if (node->right[main_idx] != NULL){

      list_pre_order(node->right[main_idx],main_idx);
    }

  }
  return 1;

}
```

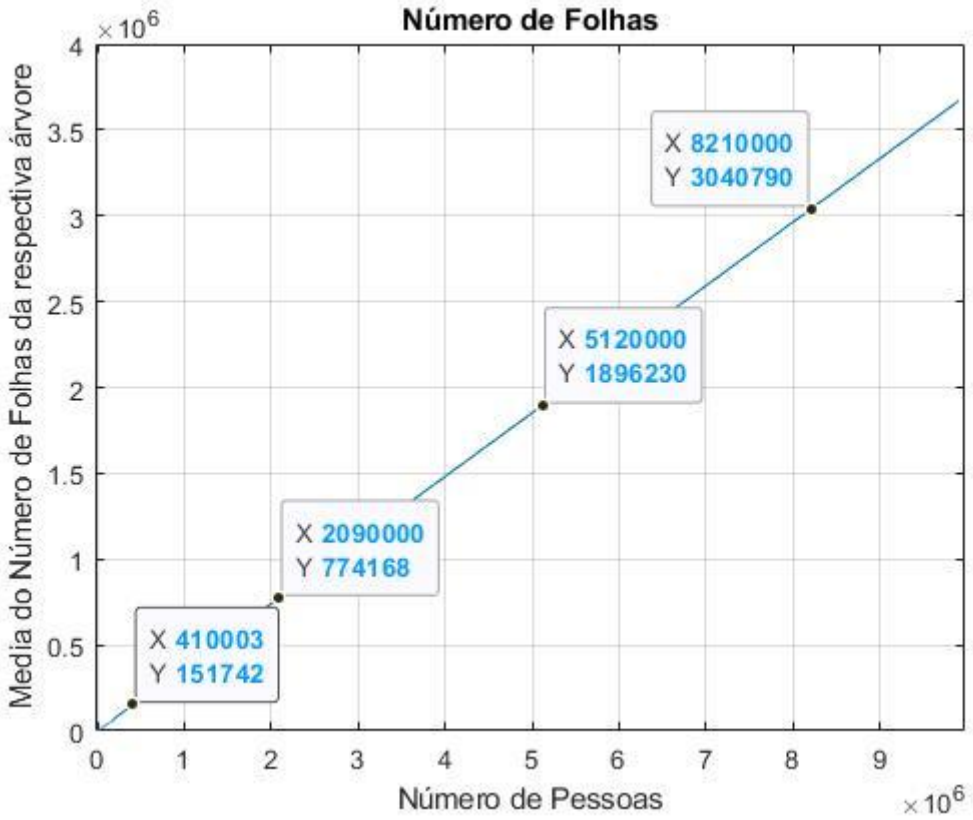
- **Depth in-order search –** Semelhante à anterior com diferença que vistamos toda a subárvore a esquerda, depois a raiz atual e finalmente toda a subárvore a direita. Esta listagem tem a particularidade de listar os Nós por ordem crescente.

```
Multiordered_Binary_Trees./remigio 123235 3
----------------------
----------------------
Name ----------------- Jeffrey Arellano
ZipCode ----------------- 75034 Frisco (Collin county)
Telephone Number ----------------- 4288 790 986
----------------------
----------------------
Name ----------------- Linda Franco
ZipCode ----------------- 43081 Westerville (Franklin county)
Telephone Number ----------------- 7091 352 001
----------------------
----------------------
Name ----------------- Taylor Burnett
ZipCode ----------------- 77479 Sugar Land (Fort Bend county)
Telephone Number ----------------- 1112 205 779
```

```
int list_in_order(tree_node_t* node,int main_idx)
{

  if (node !=NULL){
    if (node->left[main_idx] != NULL){
      list_in_order(node->left[main_idx],main_idx);
    }
        visit_node(node);
    if (node->right[main_idx] != NULL){

      list_in_order(node->right[main_idx],main_idx);
    }

  }
  return 1;

}
```

Número de Folhas

O número de folhas de uma árvore é calculado percorrendo todos os nós até encontrarmos uma folha, isto é, encontrar uma pessoa onde o ponteiro para a esquerda e para direita são nulos, onde então retornamos 1. (Como podemos ver pelo seguinte gráfico o número de folhas é aproximadamente (n+1)/2)




```
int leafCount(tree_node_t** rootp,int main_idx)
{
  if ( *rootp == NULL){
    return 0;
  }
  if ((*rootp)->left[main_idx] == NULL && (*rootp)->right[main_idx] ==
NULL)
  {
    return 1;
  }
  else
  {
    return leafCount(&(*rootp)->left[main_idx],main_idx) +
leafCount(&(*rootp)->right[main_idx],main_idx);
  }
}
```

Pessoas com atributos iguais

De forma semelhante a procura de nós, podemos também pesquisar pessoas com nomes, números de telefone e zip codes específicos. A comparação é feita entre o conjunto de caracteres e o atributo da pessoa. Dependendo do valor dessa comparação percorremos a árvore pelos ramos até chegarmos a primeira pessoa com um campo igual. Quando encontramos uma igualdade testamos para os seus filhos se os campos também são iguais ao conjunto de caracteres movemo-nos para esse nó, se não desprezamos esse nó.

```
int compareCamp(char *camp,tree_node_t* node,int main_idx){
  switch (main_idx)
  {
  case 0:
      return strcmp((node)->name,camp);
    break;
  case 1:
      return strcmp((node)->zip_code,camp);
    break;
  case 2:
      return strcmp((node)->telephone_number,camp);
    break;

  }

}




void sameType(tree_node_t** rootp,char * camp,int main_idx){

  if ((*rootp)==NULL){return;}
  int c = compareCamp(camp,*rootp,main_idx);
  if (c == 0)
  {
    visit_node(*rootp);
    if ((*rootp)->left[main_idx] != NULL){
      if (compareCamp(camp,((*rootp)->left[main_idx]),main_idx) == 0){
         sameType(&((*rootp)->left[main_idx]),camp,main_idx);
      }
    }
    if ((*rootp)->right[main_idx] != NULL){
      if (compareCamp(camp,((*rootp)->right[main_idx]),main_idx) == 0){
        sameType(&((*rootp)->right[main_idx]),camp,main_idx);
      }
    }
  }
```

```
  else if (c > 0)
  {
    sameType(&((*rootp)->left[main_idx]),camp,main_idx);
  }
  else
  {
    sameType(&((*rootp)->right[main_idx]),camp,main_idx);
  }
  return;
}
```

```
/Binary_Trees/./remigio 12435 100000 "10469 Bronx (Bronx county)" 1
----------------------
----------------------
Name ----------------- Ingrid Gutierrez
ZipCode ------------------ 10469 Bronx (Bronx county)
Telephone Number ------------------ 6562 679 379
----------------------
----------------------
Name ----------------- Aaron Murphy
ZipCode ------------------ 10469 Bronx (Bronx county)
Telephone Number ------------------ 6706 951 758
----------------------
----------------------
Name ----------------- John Ruiz
ZipCode ------------------ 10469 Bronx (Bronx county)
Telephone Number ------------------ 6695 870 143
----------------------
----------------------
Name ----------------- Patricia Thompson
ZipCode ------------------ 10469 Bronx (Bronx county)
Telephone Number ------------------ 6636 264 659
----------------------
----------------------
Name ----------------- John Mendoza
ZipCode ------------------ 10469 Bronx (Bronx county)
Telephone Number ------------------ 6584 214 910
```

Novo atributo da estrutura Pessoa

Decidimos adicionar à estrutura Pessoa o atributo “social security number”, para isto é necessário gerar um conjunto de caracteres pseudo-aleatórios com o formato

- “%d%d%d %d%d %d%d%d”, onde %d representa um algarismo de 0 a 9 inclusive.

Simplesmente adaptamos a geração de números pseudo-aleatórios já previamente disponibilizada para gerar os números de telemóvel

É então só necessário adicionar este conjunto de caracteres à estrutura Pessoa e acrescentar um ponteiro a cada conjunto de ponteiros de modo a gerar árvores com este campo.

A função de comparação é então também modificada para poder fazer a comparação com este campo.

```
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 security_number[MAX_SECURITY_NUMBER_SIZE + 1];
  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;
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); // compara nome
      else if(main_idx == 1)
        c = strcmp(node1->zip_code,node2->zip_code); // compara zip
      else if(main_idx == 2)
        c = strcmp(node1->telephone_number,node2->telephone_number); //
compara numero
      else
```

```
        c = strcmp(node1->security_number,node2->security_number); //
compara sec numero
      if(c != 0)
        return c; // different on this index, so return (sao diferentes,
retorna c)
      main_idx = (main_idx == 3) ? 0 : main_idx + 1; // advance to the
next index (sao iguais, bora para o prox)
    }
    return 0;
  }
```

```
void random_security_number(char security_number[MAX_SECURITY_NUMBER_SIZE
+ 1])
{
  int n1 = aed_random() % 1000; // 000..999
  int n2 = aed_random() % 100;         //  00..99
  int n3 = aed_random() % 10000;         //  0000..9999
  if(snprintf(security_number,MAX_SECURITY_NUMBER_SIZE + 1,"%03d-%02d-
%04d",n1,n2,n3) >= MAX_SECURITY_NUMBER_SIZE + 1)
  {
    fprintf(stderr,"security number too large (%04d) (%03d
(%03d)\n",n1,n2,n3);
    exit(1);
  }
}
```

Número de nós numa altura/profundidade especifica e posição

Para um melhor entendimento e avaliação da estrutura das árvores binárias criadas, achamos relevante a criação de uma função capaz de mostrar o número de nós a profundidades especificas tal como a posição especifica de uma certa Pessoa. Estas funções seguem a mesma estrutura das funções anteriores.

- **Número de nós a profundidade especifica –** nesta função percorremos todos os nodes começando na raiz, tanto para a esquerda como para a direita, decrementando a variável da profundidade até chegarmos ao fim de uma ramificação ou a variável da profundidade chegar a 0, isto é encontrar um nó nessa especifica profundida, então retornando 1.

```
int deapthNodes(tree_node_t** rootp, int main_idx,int depth)
{
  if ( *rootp == NULL){
    return 0;
  }
  if (depth == 0){
    return 1;
  }
  return deapthNodes(&(*rootp)->left[main_idx],main_idx,depth-1) +
deapthNodes(&(*rootp)->right[mainx_idx],main_idx,depth-1);
}
```

```
/Binary_Trees/./remigio 12435 100000
Número de nós na profundidade 20 --> 9101
```

- **Posição especifica de nó –** tal como a função de procura anteriormente falada, procuramos um nó específico na árvore da mesma forma, sendo que desta vez passamos também a variável do caminho percorrido até este ser encontrado. Quando esta pessoa é encontrada, escrevemos a sua posição em relação ao caminho percorrido.

```
tree_node_t* node_depth(tree_node_t** rootp,int main_idx,
tree_node_t* person,int rights, int lefts)
{
  if (compare_tree_nodes(*rootp,person,main_idx)==0)
  {
    printf("The position is at %d rigths and %d lefts, and has depth
%d",rights,lefts,rights + lefts);
    return *rootp;
  }
  else if (compare_tree_nodes(*rootp,person,main_idx) > 0)
  {
    return node_depth(&((*rootp)-
>left[main_idx]),main_idx,person,rights,lefts+1);
  }
  else
  {
    return node_depth(&((*rootp)-
>right[main_idx]),main_idx,person,rights +1,lefts);
  }
  return NULL;
}
```

```
/Binary_Trees/./remigio 12435 100000
Número de nós na profundidade 20 --> 9101
 Pessoa nº 20 ->Encontra-se na posição 3 direitas e 2 esquerdas, e tem profundidade 5
```

Árvores equilibradas

Como anteriormente falado o tempo de adição e procura é maioritariamente afetado pela estrutura da árvore. A estrutura ideal para inserção e remoção é uma árvore equilibrada, isto é, uma árvore onde todos os Nós possuem a **diferença entre a subárvores direita e esquerda não difere mais que 1**. O tempo de adição e procura numa árvore possui sempre **complexidade O(log(n))** enquanto que numa árvore não equilibrada o pior caso pode ser O(n).

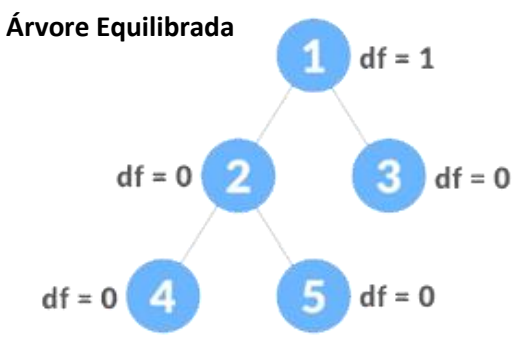
Sabendo estas vantagens, como criamos árvores equilibradas?, mais especificamente, as usadas por nós, **árvores AVL** (criada por Georgy Adelson-Velsky e Yevgeniy Landis).

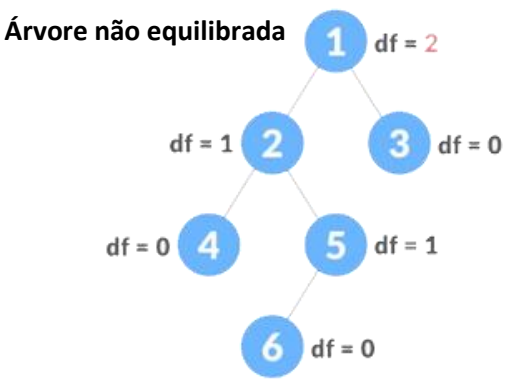
Os 3 componentes na criação destas árvores é a inserção como previamente efetuada, a descoberta do valor de equilíbrio, e as rotações necessárias.

Descoberta do fator de equilíbrio

A descoberta do equilíbrio de uma árvore é simplesmente a diferença da altura da subárvore esquerda com a subárvore direita. Este equilíbrio é aquilo que dita a necessidade de rotações ou se a árvore se encontra balanceada.

A altura de cada árvore é algo que podemos descobrir através da função falada anteriormente, o problema desta implementação é que se torna incrivelmente dispendiosa visto que isto exige percorrer por todos os nós existentes da subárvore. Por esta razão decidimos implementar na própria estrutura da pessoa o tamanho da árvore da qual este é raiz. Isto feito começando com a altura em 0 e incrementando 1 sempre que adicionamos um item a sua subárvore. Deste modo a recolha do tamanho de cada subárvore passa a ter O(1).





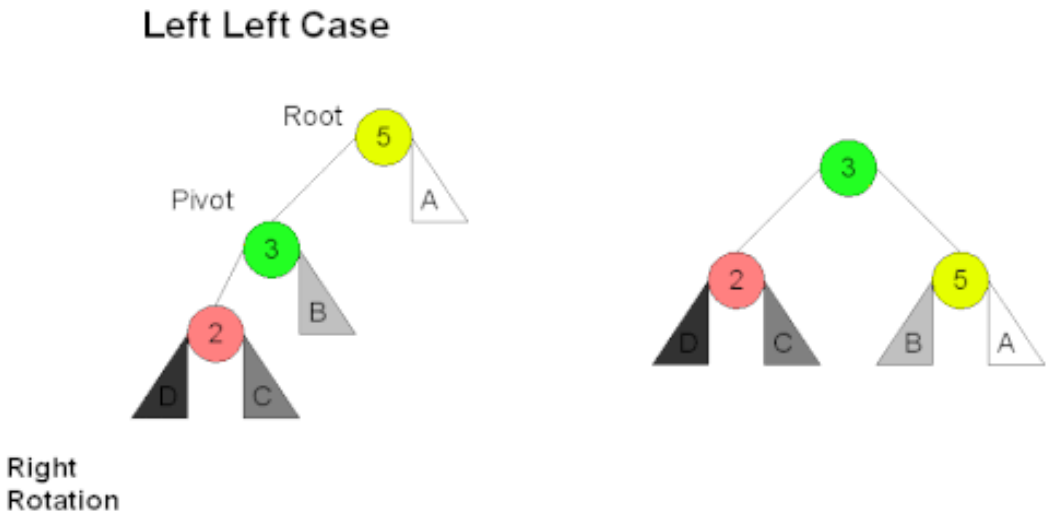
```c
int height(tree_node_t *node){
  if (node == NULL){
    return 0;
  }
  return node->height;
}
int getBalance(tree_node_t *N, int main_idx)
{
    if (N == NULL)
        return 0;
    return height(N->left[main_idx]) - height(N->right[main_idx]);
}
```

Rotações

Sempre que é inserido ou removido um Nó da árvore é necessário reestruturar a árvore e modo a manter a estrutura de uma árvore equilibrada. Estas rotações podem ser de 4 tipos.

→ **Rotação para a direita**

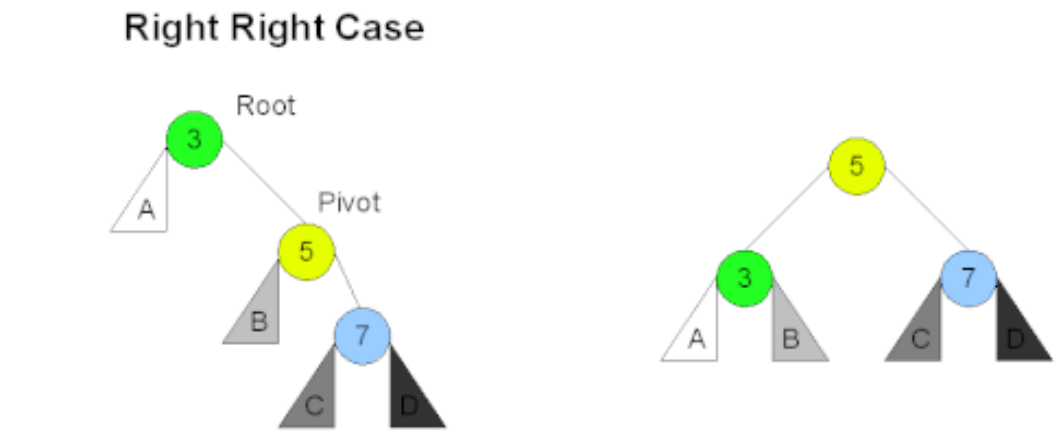
Se o fator de equilíbrio for maior que 1 e se o Nó adicionado pertencer for menor que o filho esquerdo da raiz atual.



Right Rotation

→ **Rotação para a esquerda**

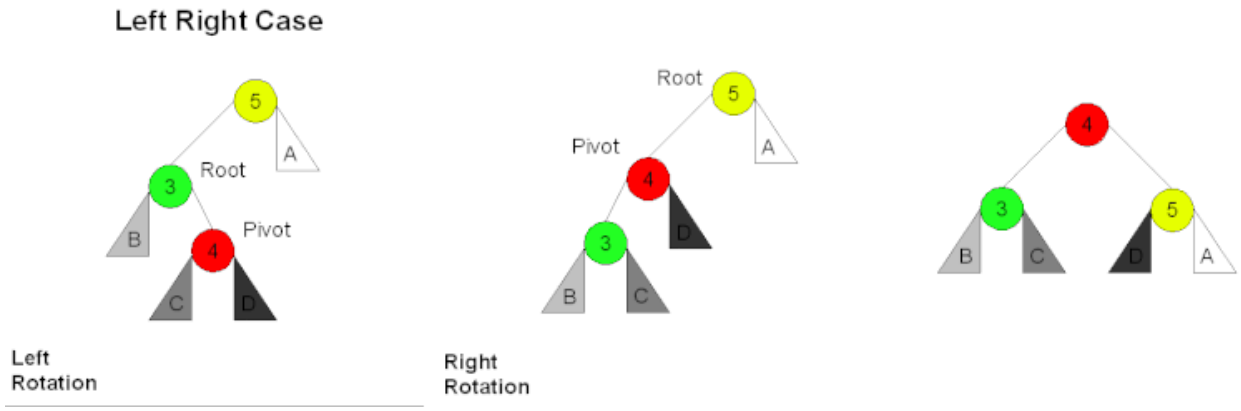
Se o fator de equilíbrio for menor que -1 e se o Nó adicionado pertencer for maior que o filho direito da raiz atual.



Left Rotation

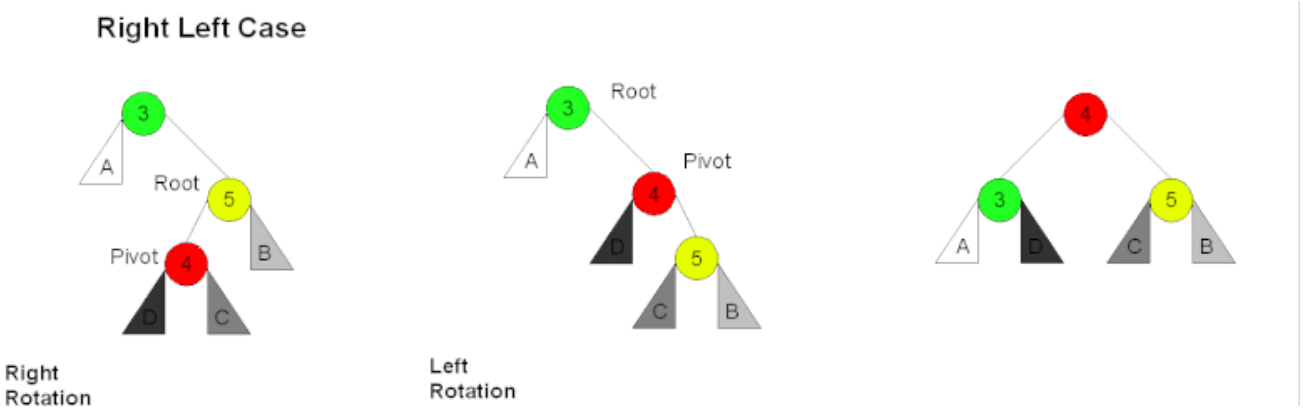
→ **Rotação para a esquerda seguida para a direita**

Se o fator de equilíbrio for menor que -1 e se o Nó adicionado pertencer for menor que o filho direito da raiz atual.



→ **Rotação para a direita seguida para a esquerda**

Se o fator de equilíbrio for maior que 1 e se o Nó adicionado pertencer for menor que o filho esquerdo da raiz atual.



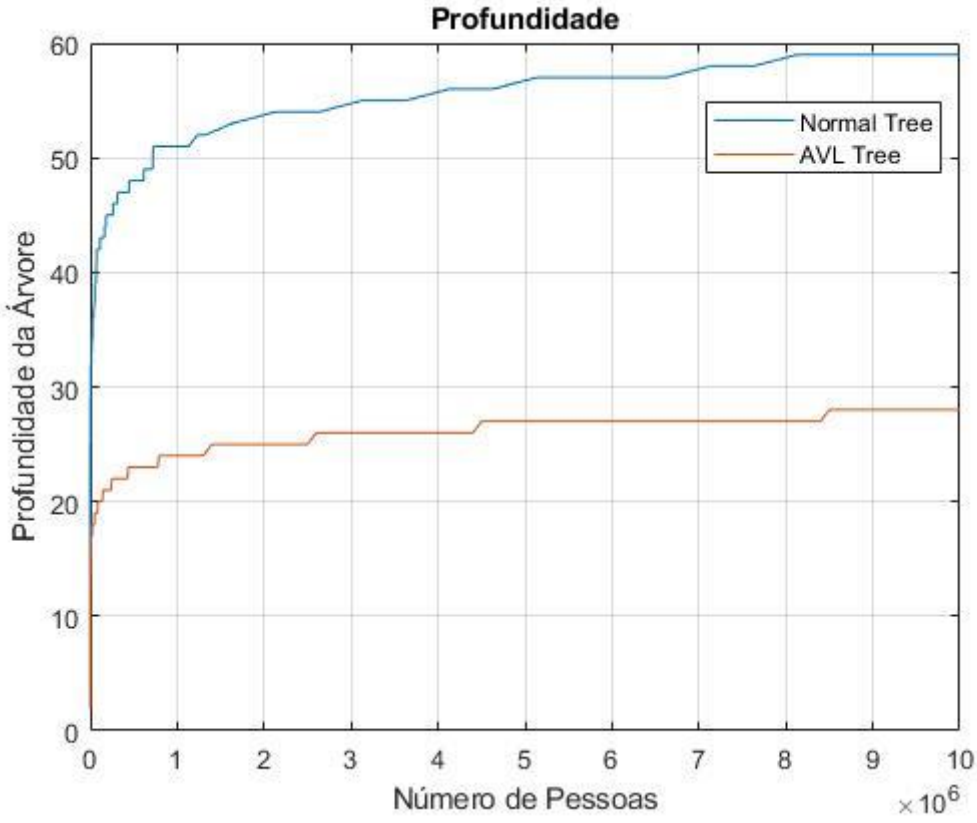
```c
typedef struct tree_node_s
{
  char name[MAX_NAME_SIZE + 1];
  char zip_code[MAX_ZIP_CODE_SIZE + 1];
  char telephone_number[MAX_TELEPHONE_NUMBER_SIZE + 1];
  char security_number[MAX_SECURITY_NUMBER_SIZE + 1];
  struct tree_node_s *left[4];
  struct tree_node_s *right[4];
  int height;       // heigth of the tree from node for AVL trees
}
tree_node_t;


int height(tree_node_t *node){
  if (node == NULL){
    return 0;
  }
  return node->height;
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
tree_node_t *rightRotate(tree_node_t *y,int main_idx)
{
    tree_node_t *x = y->left[main_idx];
    tree_node_t *T2 = NULL;
     if (x ){
       T2 = x->right[main_idx];
     }

    // Perform rotation

    if (x){
       x->right[main_idx] = y;
     }
    y->left[main_idx] = T2;

    y->height = max(height(y->left[main_idx]),height( y->right[main_idx]))+1;
    if (x){
      x->height = max(height(x->left[main_idx]), height(x->right[main_idx]))+1;
     }


    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
```

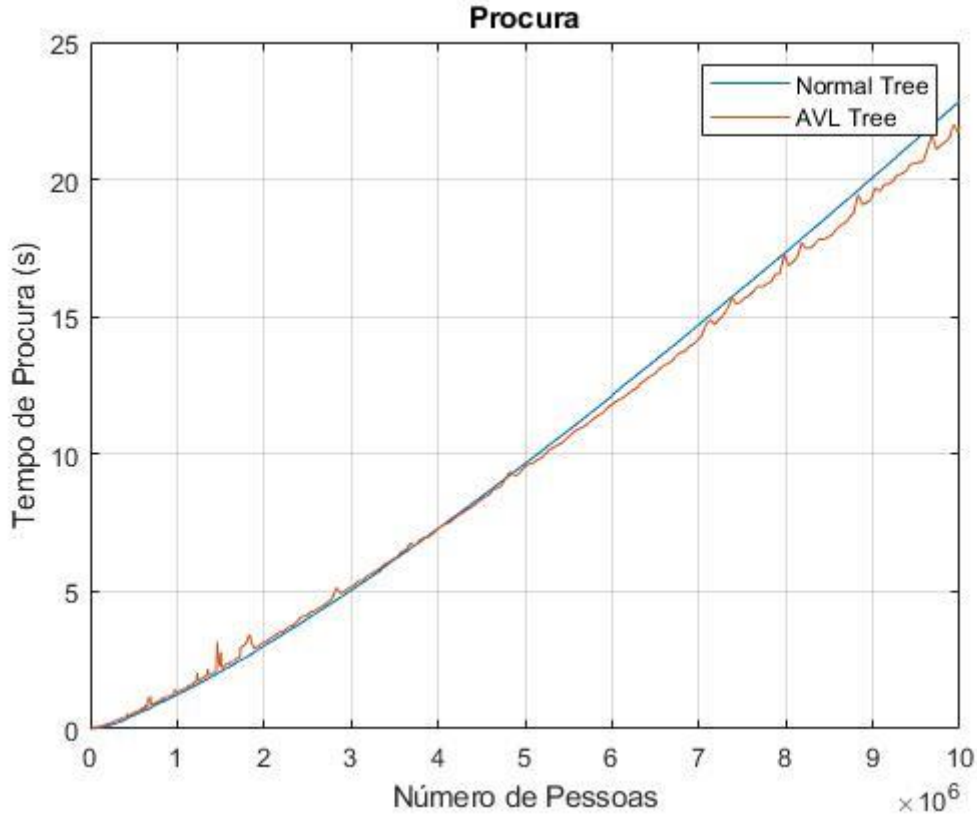
```
tree_node_t *leftRotate(tree_node_t *x,int main_idx)
{
    tree_node_t *y = x->right[main_idx];
    tree_node_t *T2 = NULL;
     if (y){
       T2 = y->left[main_idx];
     }
    // Perform rotation
    if (y){
       y->left[main_idx] = x;
     }
    x->right[main_idx] = T2;



    x->height = max(height(x->left[main_idx]), height(x->right[main_idx]))+1;
    if (y){
       y->height = max(height(y->left[main_idx]), height(y->right[main_idx]))+1;
     }
    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(tree_node_t *N, int main_idx)
{
    if (N == NULL)
        return 0;
    return height(N->left[main_idx]) - height(N->right[main_idx]);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
tree_node_t* insert(tree_node_t* node, tree_node_t* person, int main_idx)
{

    if (node == NULL){
      return person;
    }
    int *c;
    c = (int *) malloc(sizeof(int));
    *c = compare_tree_nodes(node,person,main_idx);

    if (*c > 0)
    {
       node->left[main_idx]  = insert(node->left[main_idx], person, main_idx);
    }
    else
    {
```

```
      node->right[main_idx]  = insert(node->right[main_idx], person,main_idx);
    }

    // increase the height of the tree

    node->height = 1 + max( height(node->right[main_idx]) ,height( node-
>left[main_idx]));


    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int *balance;
    balance = (int *) malloc(sizeof(int));
    *balance = getBalance(node,main_idx);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (*balance > 1 && compare_tree_nodes(person,node->left[main_idx],main_idx) < 0)
    {
        return rightRotate(node,main_idx);
    }
    // Right Right Case
    if (*balance < -1 && compare_tree_nodes(person,node->right[main_idx],main_idx)>
0)
        return leftRotate(node,main_idx);

    // Left Right Case
    if (*balance > 1 && compare_tree_nodes(person,node->left[main_idx],main_idx) > 0)
    {

        node->left[main_idx] =  leftRotate(node->left[main_idx],main_idx);
        return rightRotate(node,main_idx);
    }

    // Right Left Case
    if (*balance < -1 && compare_tree_nodes(person,node->right[main_idx],main_idx)<
0)
    {
        node->right[main_idx] = rightRotate(node->right[main_idx],main_idx);
        return leftRotate(node,main_idx);
    }
    /* return the (unchanged) node pointer */
    free(c);
    free(balance);
    return node;
}
```

Gráficos de árvores AVL

As vantagens do uso de árvores equilibradas são facilmente visíveis através de gráficos.





Nota: A melhoria não é muito visível devido ao facto da recolha de dados ter sido feita com 2 computadores com CPUs significativamente diferentes

Tal como o facto do input para a procura ser nos dado iterativamente e pela mesma ordem de adição também prejudica a visualização da melhor velocidade das árvores equilibradas

Conclusão

Este Projeto demonstra a eficácia e a importância da utilização de Árvores Binárias no tratamento de dados e como estas são fundamentais em diversos sistemas utilizados diariamente.

Conseguimos implementar as funções necessárias a completar o trabalho, conseguindo inserir elementos, procurar e descobrir o tamanho de árvores. Também conseguimos o desenvolvimento de funções que nos ajudaram numa melhor compreensão de árvores binárias e como operar com estas

A criação de árvores equilibradas, mais especificamente, árvores AVL demonstrou-se a parte mais desafiadora deste Projeto, mas é também a mais demonstrativa das vantagens da utilização de árvores binárias.

```
/Binary_Trees/./remigio 12435 100000
Tree creation time (100000 persons): 1.562e-01s
Tree creation time (100000 persons): 1.719e-01s
Tree creation time (100000 persons): 1.094e-01s
Tree creation time (100000 persons): 1.406e-01s
Tree search time (100000 persons, index 0): 1.094e-01s
Tree search time (100000 persons, index 1): 1.094e-01s
Tree search time (100000 persons, index 2): 1.250e-01s
Tree search time (100000 persons, index 3): 9.375e-02s
Tree depth for index 0: 39 (done in 0.000e+00s)
Tree depth for index 1: 37 (done in 1.562e-02s)
Tree depth for index 2: 39 (done in 1.562e-02s)
Tree depth for index 3: 42 (done in 0.000e+00s)

 AVL TREES -----------------

Tree creation time (100000 persons): 1.562e-01s
Tree creation time (100000 persons): 2.031e-01s
Tree creation time (100000 persons): 2.656e-01s
Tree creation time (100000 persons): 2.031e-01s
Tree depth for index 0: 20 (done in 0.000e+00s)
Tree depth for index 1: 20 (done in 0.000e+00s)
Tree depth for index 2: 20 (done in 1.562e-02s)
Tree depth for index 3: 20 (done in 0.000e+00s)
```

Código Utilizado

```
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 security_number[MAX_SECURITY_NUMBER_SIZE + 1];
  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
  int height;                                           // heigth of the tree from node for AVL trees
}
tree_node_t;

int max(int a, int b)
{
    return (a > b)? a : b;
}



// queue implementation

typedef struct node {
    tree_node_t* val;
    struct node *next;
} node_q;

typedef struct queue
{
    node_q* head;
    node_q* tail;
} queue;



void init_queue(queue* q){
    q->head = NULL;
    q->tail = NULL;
}

void enqueue(queue* q, tree_node_t* value) {
    // create new node
    node_q * new_node = malloc(sizeof(node_q));
    if (new_node==NULL) { return;}    //if malloc fails
    new_node->val = value;
    new_node->next = NULL;

    // if there is a tail we just connect that tail to this node
    if (q->tail != NULL){
```

```
        q->tail->next = new_node;
    }
    q->tail = new_node;

    // if there is no head we set this one also as head
    if (q->head == NULL) {
        q->head = new_node;
    }
    return;
}

tree_node_t* dequeue(queue* q) {
    if (q->head == NULL) {return NULL;}
    node_q * tmp = q->head;
    tree_node_t* result = tmp->val;
    q->head = q->head->next;
    if (q->head == NULL){
        q->tail = NULL;
    }
    free(tmp);
    return result;
}









//
// 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); // compara nome
      else if(main_idx == 1)
        c = strcmp(node1->zip_code,node2->zip_code); // compara zip
      else if(main_idx == 2)
        c = strcmp(node1->telephone_number,node2->telephone_number); // compara numero
      else
        c = strcmp(node1->security_number,node2->security_number); // compara sec numero
      if(c != 0)
        return c; // different on this index, so return (sao diferentes, retorna c)
      main_idx = (main_idx == 3) ? 0 : main_idx + 1; // advance to the next index (sao iguais, bora para o prox)
```

```
    }
    return 0;
  }


//
// ALL THE STUFF FOR AVL TREES
//


int height(tree_node_t *node){
  if (node == NULL){
    return 0;
  }
  return node->height;
}


// A utility function to right rotate subtree rooted with y
// See the diagram given above.
tree_node_t *rightRotate(tree_node_t *y,int main_idx)
{
    tree_node_t *x = y->left[main_idx];
    tree_node_t *T2 = NULL;
     if (x ){
       T2 = x->right[main_idx];
     }


    // Perform rotation


    if (x){
       x->right[main_idx] = y;
     }
    y->left[main_idx] = T2;


    y->height = max(height(y->left[main_idx]),height( y->right[main_idx]))+1;
    if (x){
      x->height = max(height(x->left[main_idx]), height(x->right[main_idx]))+1;
     }



    // Return new root
    return x;
}


// A utility function to left rotate subtree rooted with x
// See the diagram given above.
tree_node_t *leftRotate(tree_node_t *x,int main_idx)
{
    tree_node_t *y = x->right[main_idx];
    tree_node_t *T2 = NULL;
     if (y){
```

```
       T2 = y->left[main_idx];
     }
    // Perform rotation
    if (y){
       y->left[main_idx] = x;
     }
    x->right[main_idx] = T2;




    x->height = max(height(x->left[main_idx]), height(x->right[main_idx]))+1;
    if (y){
       y->height = max(height(y->left[main_idx]), height(y->right[main_idx]))+1;
     }
    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(tree_node_t *N, int main_idx)
{
    if (N == NULL)
        return 0;
    return height(N->left[main_idx]) - height(N->right[main_idx]);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
tree_node_t* insert(tree_node_t* node, tree_node_t* person, int main_idx)
{

    if (node == NULL){
      return person;
    }
    int *c;
    c = (int *) malloc(sizeof(int));
    *c = compare_tree_nodes(node,person,main_idx);

    if (*c > 0)
    {
       node->left[main_idx]  = insert(node->left[main_idx], person, main_idx);
    }
    else
    {
      node->right[main_idx]  = insert(node->right[main_idx], person,main_idx);
    }

    // increase the height of the tree
```

```
    node->height = 1 + max( height(node->right[main_idx]) ,height( node->left[main_idx]));



    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int *balance;
    balance = (int *) malloc(sizeof(int));
    *balance = getBalance(node,main_idx);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (*balance > 1 && compare_tree_nodes(person,node->left[main_idx],main_idx) < 0)
    {
        return rightRotate(node,main_idx);
    }
    // Right Right Case
    if (*balance < -1 && compare_tree_nodes(person,node->right[main_idx],main_idx)> 0)
        return leftRotate(node,main_idx);

    // Left Right Case
    if (*balance > 1 && compare_tree_nodes(person,node->left[main_idx],main_idx) > 0)
    {

        node->left[main_idx] =  leftRotate(node->left[main_idx],main_idx);
        return rightRotate(node,main_idx);
    }

    // Right Left Case
    if (*balance < -1 && compare_tree_nodes(person,node->right[main_idx],main_idx)< 0)
    {
        node->right[main_idx] = rightRotate(node->right[main_idx],main_idx);
        return leftRotate(node,main_idx);
    }
    /* return the (unchanged) node pointer */
    free(c);
    free(balance);
    return node;
}



//
// tree insertion routine (place your code here)
//

void tree_insert( tree_node_t** rootp, tree_node_t* node,int main_idx)
```

```
{
  if ( *rootp == NULL){
    *rootp = node;
    return;
  }
  int c = compare_tree_nodes(*rootp,node,main_idx);
  if (c < 0)
  {
    tree_insert(&((*rootp)->right[main_idx]), node,main_idx);
    return;
  }
  else
  {
    tree_insert(&((*rootp)->left[main_idx]), node,main_idx);
    return;
  }
  return;

}



//
// tree search routine (place your code here)
//

tree_node_t* find(tree_node_t** rootp,int main_idx,tree_node_t* person)
{
  if ((*rootp) == NULL){
    printf("here");
    return NULL;
  }
  if (compare_tree_nodes(*rootp,person,main_idx)==0)
  {
    return *rootp;
  }
  else if (compare_tree_nodes(*rootp,person,main_idx) > 0)
  {
    return find(&((*rootp)->left[main_idx]),main_idx,person);
  }
  else
  {
    return find(&((*rootp)->right[main_idx]),main_idx,person);
  }
  return NULL;
}



//
// tree depdth
```

```
//


int tree_depth(tree_node_t** root, int main_idx)
{
  if ( *root == NULL){
    return 0;
  }
  int leftheight = tree_depth(&((*root)->left[main_idx]),main_idx);
  int rightheight = tree_depth(&((*root)->right[main_idx]),main_idx);

  if (leftheight > rightheight)
  {
    return leftheight + 1;
  }
  else{
    return rightheight + 1 ;
  }
}



//
// list, i,e, traverse the tree (place your code here)
//
void visit_node(tree_node_t* node)
{
  printf("----------------------\n");
  printf("----------------------\n");
  printf("Name ----------------- %s\n",node->name);
  printf("ZipCode ----------------- %s\n",node->zip_code);
  printf("Telephone Number ----------------- %s\n",node->telephone_number);

  return;
}

void traverse_breadth_first(tree_node_t *link,int main_idx)
{
  queue q1;
  init_queue(&q1);
  enqueue(&q1,link);
  while(q1.head != NULL)
  {
    link = dequeue(&q1);
    if(link != NULL)
    {
      visit_node(link);
      enqueue(&q1,link->left[main_idx]);
      enqueue(&q1,link->right[main_idx]);
    }
  }
```

```
}



int list_in_order(tree_node_t* node,int main_idx)

{


  if (node !=NULL){

    if (node->left[main_idx] != NULL){

      list_in_order(node->left[main_idx],main_idx);

    }

        visit_node(node);

    if (node->right[main_idx] != NULL){


      list_in_order(node->right[main_idx],main_idx);

    }


  }

  return 1;


}




int list_pre_order(tree_node_t* node,int main_idx)

{


  if (node !=NULL){

    visit_node(node);

    if (node->left[main_idx] != NULL){

      list_pre_order(node->left[main_idx],main_idx);

    }

    if (node->right[main_idx] != NULL){


      list_pre_order(node->right[main_idx],main_idx);

    }


  }

  return 1;


}



//

// Find depth of node

//


tree_node_t* node_depth(tree_node_t** rootp,int main_idx,tree_node_t* person,int rights, int lefts)

{

  if (compare_tree_nodes(*rootp,person,main_idx)==0)
```

```
  {
    printf("found it\n");
    printf("the position is at %d rigths and %d lefts, and has depth %d",rights,lefts,rights + lefts);
    return *rootp;
  }
  else if (compare_tree_nodes(*rootp,person,main_idx) > 0)
  {
    return node_depth(&((*rootp)->left[main_idx]),main_idx,person,rights,lefts+1);
  }
  else
  {
    return node_depth(&((*rootp)->right[main_idx]),main_idx,person,rights +1,lefts);
  }
  return NULL;
}

//
//  how many nodes
//

int numberNodes(tree_node_t** root, int main_idx)
{
  if ( *root == NULL){
    return 0;
  }
  int leftnodes= numberNodes(&((*root)->left[main_idx]),main_idx);
  int rightnodes = numberNodes(&((*root)->right[main_idx]),main_idx);

    return rightnodes + leftnodes+ 1 ;
}

//
//  how many nodes is each level
//

int deapthNodes(tree_node_t** rootp, int main_idx,int depth)
{
  if ( *rootp == NULL){
    return 0;
  }
  if (depth == 0){
    return 1;
  }
  return deapthNodes(&(*rootp)->left[main_idx],main_idx,depth-1) + deapthNodes(&(*rootp)->right[main_idx],main_idx,depth-1);
}



//
//  how many leaf nodes
```

```
//
int leafCount(tree_node_t** rootp,int main_idx)
{
  if ( *rootp == NULL){
    return 0;
  }
  if ((*rootp)->left[main_idx] == NULL && (*rootp)->right[main_idx] == NULL)
  {
    return 1;
  }
  else
  {
    return leafCount(&(*rootp)->left[main_idx],main_idx) + leafCount(&(*rootp)->right[main_idx],main_idx);
  }
}


// number with same camp


int compareCamp(char *camp,tree_node_t* node,int main_idx){
  switch (main_idx)
  {
  case 0:
      return strcmp((node)->name,camp);
    break;
  case 1:
      return strcmp((node)->zip_code,camp);
    break;
  case 2:
      return strcmp((node)->telephone_number,camp);
    break;


  }


}
```

```
void sameType(tree_node_t** rootp,char * camp,int main_idx){


  if ((*rootp)==NULL){return;}
  int c = compareCamp(camp,*rootp,main_idx);
  if (c == 0)
  {
    visit_node(*rootp);
    if ((*rootp)->left[main_idx] != NULL){
      if (compareCamp(camp,((*rootp)->left[main_idx]),main_idx) == 0){
         sameType(&((*rootp)->left[main_idx]),camp,main_idx);
      }
    }
    if ((*rootp)->right[main_idx] != NULL){
      if (compareCamp(camp,((*rootp)->right[main_idx]),main_idx) == 0){
        sameType(&((*rootp)->right[main_idx]),camp,main_idx);
      }
    }
  }
  else if (c > 0)
  {
    sameType(&((*rootp)->left[main_idx]),camp,main_idx);
  }
  else
  {
    sameType(&((*rootp)->right[main_idx]),camp,main_idx);
  }
  return;
}
```