**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%;**

**Índice**

**Introdução1**

Escrever título do capítulo (nível 2)2

Escrever título do capítulo (nível 3)3

**Escrever título do capítulo (nível 1)4**

Escrever título do capítulo (nível 2)5

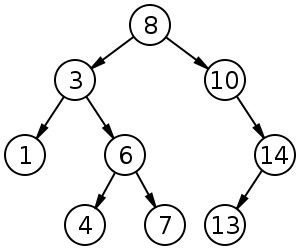
Escrever título do capítulo (nível 3)6

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.

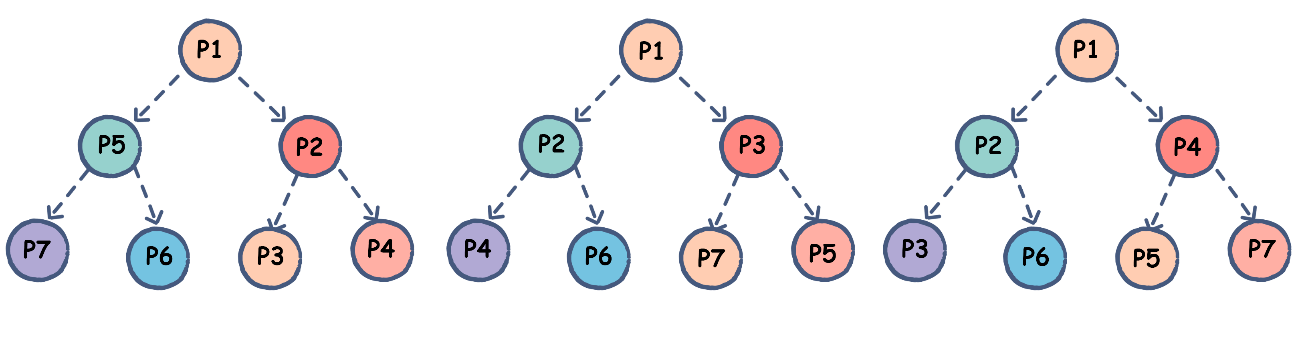
Árvore Binária:

Neste caso falaremos de Árvores Binárias de Busca que podem ser definidas por um conjunto de Nós onde cada node 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:

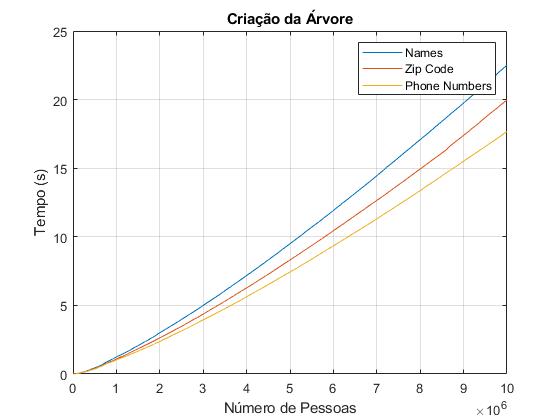
* Nome
* Zip Code
* Número de Telefone

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 node onde nos encontramos. Caso o node tiver um valor maior que o node atual tentamos adicionar à direita caso contrário tentamos adicionar a esquerda. Desde modo o node 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 arvore 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;

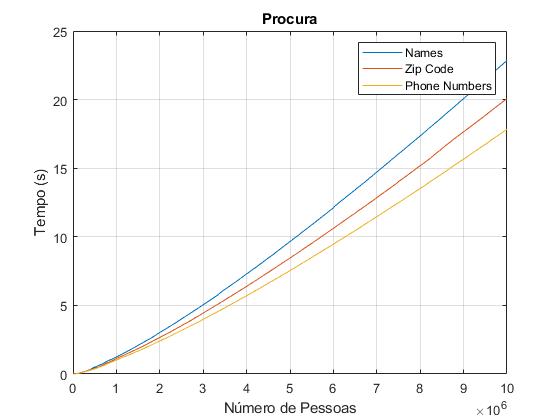
}

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 node onde nos encontramos. Se este for igual devolvemos este node, se for maior ou menos movemo-nos para a direita ou esquerda da árvore respetivamente.
* Se encontrarmos um node 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);

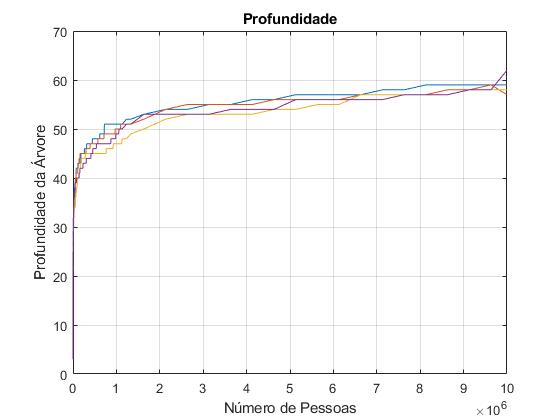
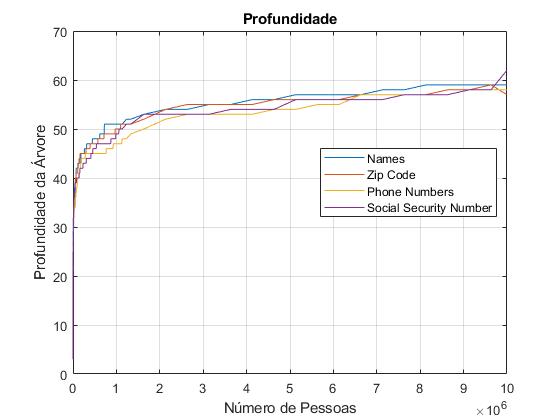
  }

  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)

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{

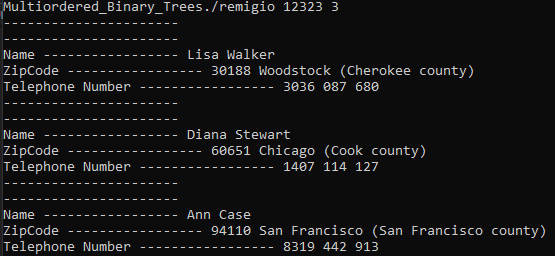
    return rightheight + 1 ;

  }

}

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.

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;

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){

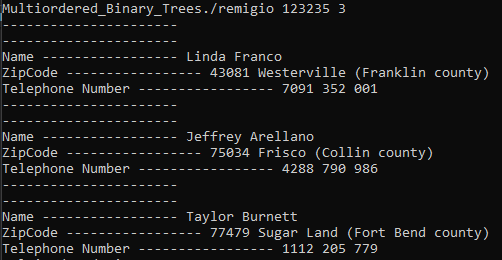
        q->tail = NULL;

    }

    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.

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){

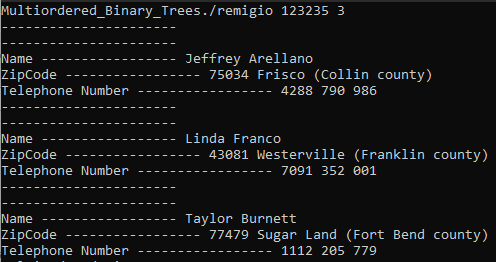
      list\_pre\_order(node->right[main\_idx],main\_idx);

    }

  }

  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.

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);

    }

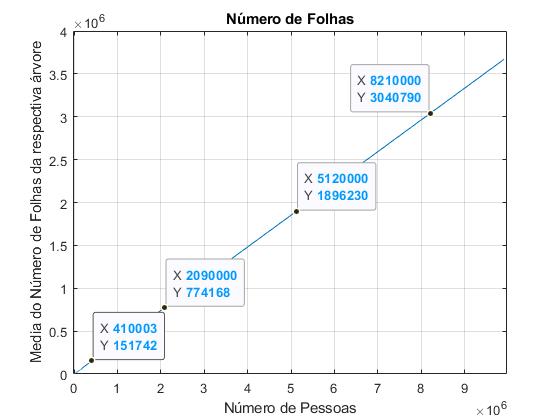
  }

  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);

  }

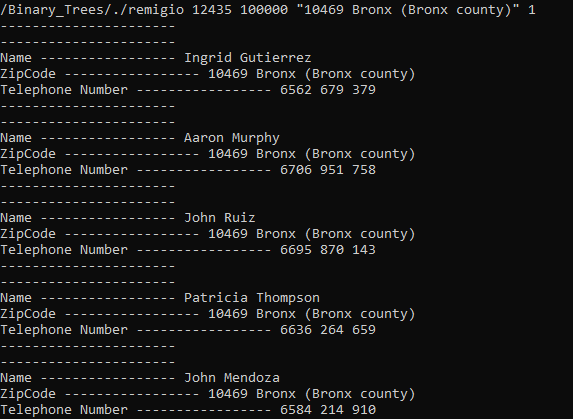
  else

  {

    sameType(&((\*rootp)->right[main\_idx]),camp,main\_idx);

  }

  return;

}

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);

****}

* **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;

****}

Árvores equilibradas

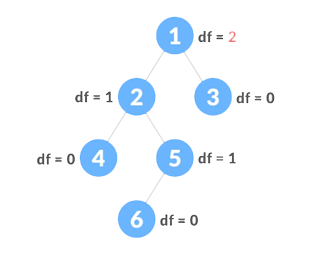
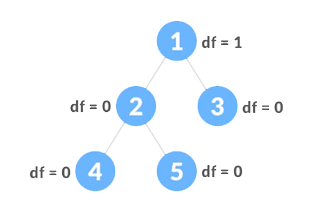
Como anteriormente falado o tempo de adição e procura é maioritariamente afetado pela estrutura da árvore. A estrutura ideal para inserção e remoção é uma árvore equilibrada, isto é, uma árvore onde todos os Nós possuem a **diferença entre a subárvores direita e esquerda não difere mais que 1**. O tempo 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](https://pt.wikipedia.org/wiki/Georgy_Adelson-Velsky) e [Yevgeniy Landis](https://pt.wikipedia.org/wiki/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 tamanha de cada subárvore passa a ter O(1).

**Árvore não equilibrada**

**Árvore Equilibrada**

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;

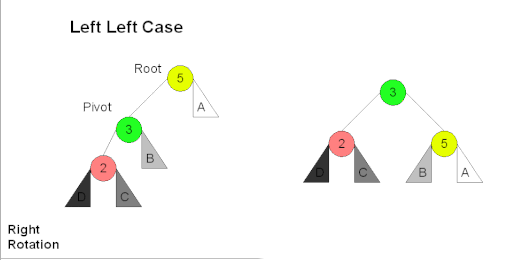
    return height(N->left[main\_idx]) - height(N->right[main\_idx]);

}

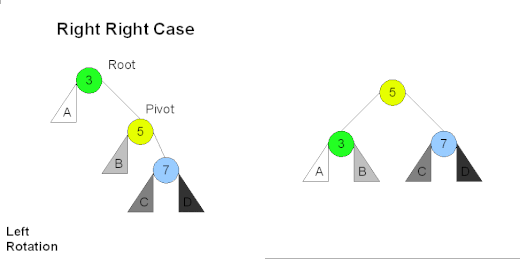
Rotações

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.

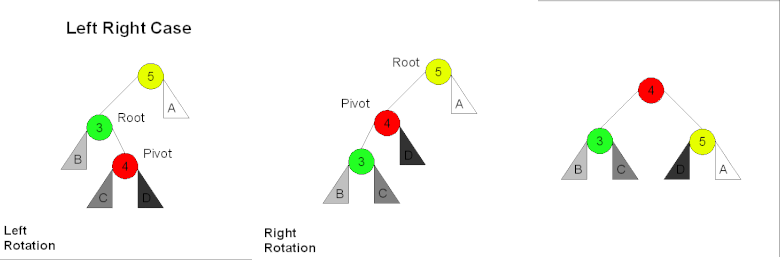
* **Rotação para a direita**

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

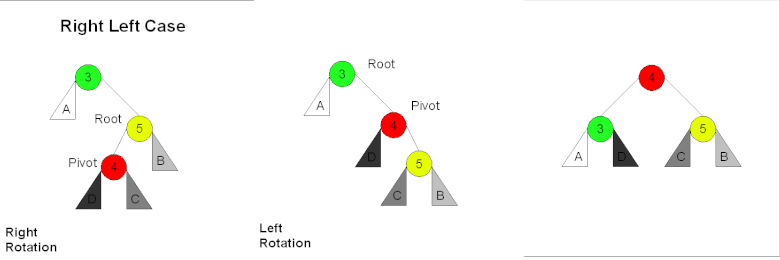
* **Rotação para a esquerda**

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

* **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.

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);

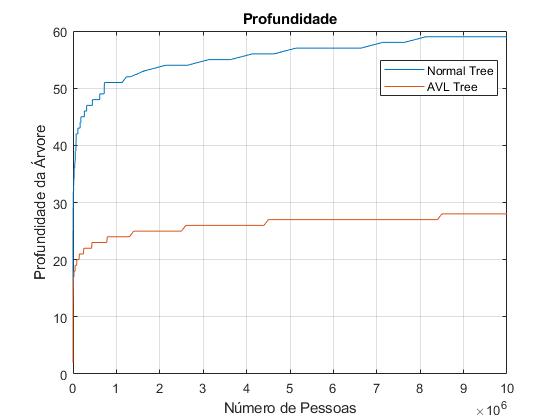
    free(balance);

    return node;

}

Gráficos de árvores AVL

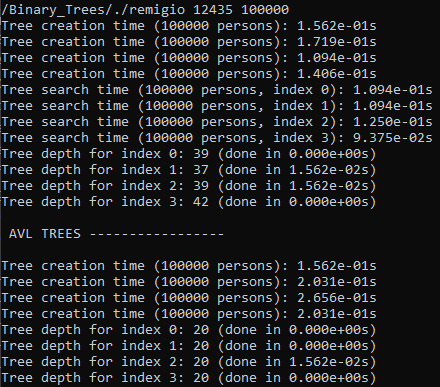
As vantagens do uso de árvores equilibradas são facilmente visíveis através de gráficos.



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.

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;

}