Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Curso de Bacharelado em Sistemas de Informação

Daniel Fucci
Pedro Fernando Christofoletti dos Santos
Bruno Maximo e Melo
Heitor Camilo de Freitas e Oliveira
Henrique Delgado Franzin

Monografia apresentada na disciplina SSC0503 -
Introdução à Ciência de Computação II, como
requisito parcial para obtenção da aprovação na mesma.

São Carlos/SP, 07 de Novembro de 2019

SUMÁRIO

1. Introdução

O trabalho tem como propósito implementar uma árvore AVL que realiza as operações de CRD de um CRUD tradicional (Create, Read, Update, Delete) de banco de dados sobre os nós contendo nome, CPF ou data de nascimento a partir de um arquivo texto em formato CSV disponibilizado previamente. O usuário ainda pode visualizar os nós de forma ordenada pelo campo desejado.

A implementação se dá com a linguagem C e é estruturada em arquivos que definem e implementam o algoritmo da árvore, chamados de tree.h e tree.c respectivamente, e de um arquivo main.c que possui o menu de interação com o usuário. Além dos arquivos citados, a solução também possui um Makefile para facilitar o processo de build do programa.

2. Fundamentação Teórica

A árvore AVL é uma árvore de busca binária balanceada com relação à altura de suas subárvores. Uma árvore AVL verifica a altura das subárvores da esquerda e da direita, garantindo que essa diferença não seja maior que 1 ou menor que -1. A essa diferença é dado o nome de fator de balanceamento que é calculado como:

$f_b = h_{esq} - h_{dir}$, onde h_{esq} é a altura da subárvore à esquerda e h_{dir} a altura da subárvore à direita.

O fator de balanceamento é calculado a cada nó, ou seja, para cada nó, a diferença de altura entre a subárvore da esquerda e da direita não pode estar fora do intervalo [-1,0,1]. O nó pode armazenar o fator de balanceamento ou a altura do próprio nó em relação à árvore.

Uma árvore necessita de funções de inserção, busca e remoção. A árvore AVL, como já é possível perceber, a cada inserção ou remoção tem a capacidade intrínseca de se autobalancear. Uma inserção pode fazer com que o fator de balanceamento de um nó se encontre fora do intervalo acima citado, desbalanceando a árvore. Então, após a inserção é necessário voltar até a raíz nó por nós atualizando as alturas. Se um novo fator de balanceamento para um determinado nó for 2 ou -2, ajustamos a árvore realizando uma operação chamada de *rotação* em torno desse nó. Ou seja, o autobalanceamento da árvore AVL ocorrerá instantaneamente toda vez que houver um nó com fator e balanceamento 2 ou -2. Se a inserção não ocorreu em um nó folha, mas aconteceu de desbalancear a árvore, pode ser necessário uma rotação dupla para balanceá-la, porém cada rotação dessa é no sentido contrário da outra, ou seja, direita-esquerda ou esquerda-direita, mas nunca direita-direita ou esquerda-esquerda.

A complexidade das operações de busca, inserção e remoção na árvore AVL, que chamaremos neste trabalho de operações CRD, são *O(log n)*. Mais detalhes na tabela a seguir:

Algoritmo	**Caso Médio**	**Pior Caso**
Espaço	O(n)	O(n)
Busca	O(log n)	O(log n)
Inserção	O(log n)	O(log n)
Remoção	O(log n)	O(log n)

3. Implementação

Como o trabalho especifica que devemos trabalhar com um registro de pessoa que deveria possuir nome, CPF e data de nascimento, então é necessário referenciar os registros em três árvores AVL diferente para fins de realizar as operações de CRD nos três campos já citados.

No menu de usuário há as seguintes opções:

```
printf("\n\nDigite o número da operação desejada\n\n");
printf("1-  Ler arquivo\n");
printf("2-  Consulta\n");
printf("3-  Inserir novo cadastro\n");
printf("4-  Remover cadastro\n");
printf("5-  Imprimir cadastros\n");
printf("6-  Apagar banco de dados\n\n");
printf("=============================================");
printf("\n\nDigite 0 para sair\n\n");
```

Figura 1. Menu de usuário

As opções 1, 6 e 0 serão desconsideradas para a explicação nessa monografia, pois são consideradas triviais e vamos nos ater às opções relacionadas às operações de CRD que no caso são as opções 2, 3, 4 e 5.

3.1. Consulta

A consulta é divida por campos em um novo menu como pode ser visto abaixo:

```
printf("\nDigite o número da operação desejada\n\n");
printf("1-  Consulta por CPF\n");
printf("2-  Conuslta por Nome\n");
printf("3-  Conuslta por Data de Nascimento\n\n");
```

Figura 2. Submenu de consulta

A partir do momento que o usuário escolhe a opção desejada o programa entra no algoritmo que realiza a devida consulta. Abaixo pode ser visto o código específico da árvore para cada uma consultas que são respectivamente as funções avltree_find(), avltree_search_names() e avltree_search_dates():

```c
void *avltree_find(const avltree *tree, const PESSOA *pessoa){
  void *temp = NULL;
  avlsearchresult result;
  result.node = NULL;
  result.parent = NULL;

  if (avltree_search(tree, &result, pessoa)) temp = result.node->pessoa;
  return temp;
}
```

Figura 3. Função avltree_find().

```c
static int avltree_search(const avltree *tree, avlsearchresult *result, const PESSOA *pessoa){
    int found = 0;
    result->node = tree->root;
    while (!found && result->node != NULL){
      int rv = tree->compare(result->node->pessoa, pessoa);
      if (rv == 0) found = 1;
      else{
        result->parent = result->node;
        if (rv > 0) result->node = result->node->left;
        else if (rv < 0) result->node = result->node->right;
      }
    }
    return found;
}
```

Figura 4. Função avltree_search() chamada pela avltree_find().

```c
static void avltree_for_each_recursive_names(const avltreenode *root, avltree_forfn fun, char *name){
  if (root->left != NULL)avltree_for_each_recursive_names(root->left, fun, name);
  if (StartsWith(root->pessoa->nome, name)) fun(root->pessoa);
  if (root->right != NULL)avltree_for_each_recursive_names(root->right, fun, name);
}

void avltree_search_names(const avltree *tree, char *name) {
    avltree_for_each_recursive_names(tree->root, (avltree_forfn)printPessoa, name);
}
```

Figura 5. Função avltree_search_names() e sua respectiva função recursiva.

```c
static void avltree_for_each_recursive_dates(const avltreenode *root, avltree_forfn fun, DATE *date1, DATE *date2){
  if (root->left != NULL)avltree_for_each_recursive_dates(root->left, fun, date1, date2);
  if (compareDate_Group(root->pessoa, date1, date2) == 0) fun(root->pessoa);
  if (root->right != NULL)avltree_for_each_recursive_dates(root->right, fun, date1, date2);
}

void avltree_search_dates(const avltree *tree, DATE *date1, DATE *date2){
   avltree_for_each_recursive_dates(tree->root, (avltree_forfn)printPessoa, date1, date2);
}
```

Figura 6. Função avltree_search_dates() e sua respectiva função recursiva.

3.2. Inserir novo registro

Diferente da consulta, a inserção não possui submenu, portanto abaixo se encontra apenas a função avltree_add() que implementa a inserção. Perceba que essa função chama a função que realiza o balanceamento chamada avltree_rebalance():

```
void *avltree_add(avltree *tree, PESSOA *pessoa){
  void *temp = NULL;
  avlsearchresult result;
  result.node = NULL;
  result.parent = NULL;

  if (avltree_search(tree, &result, pessoa)){
    temp = result.node->pessoa;
    result.node->pessoa = pessoa;
  }
  else{
    avltreenode *node = avltreenode_create(pessoa);
    if (result.node == tree->root) tree->root = node;
    else{
      int rv = tree->compare(pessoa, result.parent->pessoa);
      if (rv < 0) result.parent->left = node;
      else result.parent->right = node;
      node->parent = result.parent;
      avltree_rebalance(tree, node);
    }
    tree->count++;
  }
  return temp;
}
```

Figura 7. Função avltree_add().

3.3. Remover cadastro

Assim como a opção de consulta, a remoção possui submenu em que o usuário tem a oportunidade de remover por campos:

```
printf("\nDigite o número da operação desejada\n\n");
printf("1-  Remover pelo nome\n");
printf("2-  Remover pelo CPF\n");
printf("3-  Remover pela data de nascimento\n\n");
```

Figura 8. Submenu de remoção.

Neste caso a função que remove o campo nome, CPF e data de nascimento é a mesma, nomeada de avltree_remove() que chama sua função recursiva avltree_remove_node() que não iremos colocar neste trabalho por ser muito grande. Segue abaixo a avltree_remove:

```
void *avltree_remove(avltree *tree, const PESSOA *pessoa){
  void *temp = NULL;
  avlsearchresult result;
  result.node = NULL;
  result.parent = NULL;

  if (avltree_search(tree, &result, pessoa)){
    temp = result.node->pessoa;
    avltree_remove_node(tree, result.node);
  }
  return temp;
}
```

Figura 9. Função avltree_remove().

3.4. Imprimir cadastros

Esta opção também possui submenus e assim como no caso da remoção, a função executada para imprimir tanto o nome, quanto o CPF e a data de nascimento é a mesma, chamada de avltree_for_each() que chama sua função recursiva avltree_for_each_recusive():

```
printf("\nDigite o número da operação desejada\n\n");
printf("1-  Imprimir cadastros ordenados pelo Nome\n");
printf("2-  Imprimir cadastros ordenados pelo CPF\n");
printf("3-  Imprimir cadastros ordenados pela Data de Nascimento\n\n");
```

Figura 10. Submenu Imprimir cadastros.

```
static void avltree_for_each_recursive(const avltreenode *root, avltree_forfn fun){
  if (root->left != NULL)avltree_for_each_recursive(root->left, fun);
  fun(root->pessoa);
  if (root->right != NULL)avltree_for_each_recursive(root->right, fun);
}

void avltree_for_each(const avltree *tree, avltree_forfn fun){
  if (tree->root)avltree_for_each_recursive(tree->root, fun);
}
```

Figura 11. Função avltree_for_each().

4. Discussões

Analisamos se seria melhor criar funlçoes específicas para cada uma das três árvores AVL, mas então percebemos que a estrutura poderia ser a mesma variando apenas a função de comparação de elementos, reduzindo assim linhas de código que de outro modo seriam desnecessárias.

Uma dificuldade enfrentada foi realizar a busca entre datas de nascimento, mas foi uma dificuldade de lógica no projeto projeto da solução.

Utilizamos propositalmente o uso de funções estáticas sempre que fosse uma função a ser chamada por outras no próprio arquivo tree.c, portanto não necessita especificá-las no tree.h já que elas pertencem ao escopo direto da implementação.

5. Conclusão

Os testes realizados obtiveram sucesso e bom desempenho. Fomos avisados pela turma que em alguns casos as chamadas recursivas estouravam a pilha, mas no nosso caso isso não chegou a acontecer. Há duas razões possíveis para isso não ter acontecido. A primeira delas é que as máquinas que rodamos possui mais de 8GB de memória RAM. E a segunda, e mais provável, foi que as funções recursivas eram simples e portanto não devem afetar no consumo da pilha de maneira significativa.

Referências

http://wiki.icmc.usp.br/images/f/f0/AVL.pdf

https://pt.wikipedia.org/wiki/%C3%81rvore_AVL

https://www.youtube.com/watch?v=YkF76cOgtMQ