Instituto Federal de Educação Ciência e Tecnologia
do Norte de Minas Gerais - IFNMG
Bacharelado em Ciência da Computação

Disciplina de Algoritmos e Estruturas de Dados II

# Trabalho Prático 3

Daniel Veloso Braga
Gabriel Henrique Vieira da Mata
Thiago Emanuel Silva Antunes Lopes

# Sumário


**1 Introdução** **1**

**2 Descrição do Trabalho Prático** **2**

2.1 Entendendo a árvore persistente e suas utilidades . . . . . 2

2.2 Diferença entre os métodos e vantagens . . . . . 2

2.3 Desenvolvimento do método de cópia da árvore . . . . . 3

2.3.1 *Método Cópia* . . . . . 3

2.3.2 Métodos auxiliares ao método de cópia . . . . . 3

2.4 Desenvolvimento do método de árvore persistente . . . . . 5

2.4.1 Método Persistente . . . . . 5

2.4.2 Métodos Internos ao método persistente . . . . . 6

2.5 *Remover* . . . . . 9

2.6 *Busca* . . . . . 10

**3 Estudo Analítico entre os métodos** **12**

3.1 Utilização de espaço e tempo . . . . . 12

# Capítulo 1

# Introdução

O presente relatório proposto pelo Professor Wagner Barros na disciplina Algoritmo e Estrutura de Dados II e realizado pelos discentes Daniel Veloso Braga, Gabriel Henrique e Thiago Emanuel do $2^o$ Período do Curso Bacharel em Ciência da Computação do Instituto Federal do Norte de Minas Gerais visa expor/detalhar a criação e funcionamento do terceiro trabalho prático sobre árvores persistentes.

O objetivo do trabalho era implementar em C++ o TAD utilizando Classes e alguns de seus métodos.

# Capítulo 2

# Descrição do Trabalho Prático

## 2.1 Entendendo a árvore persistente e suas utilidades

A árvore de persistência em si é utilizada em problemas em que o algoritmo precisa salvar informações sobre versões anteriores dos dados. Por exemplo, uma loja que necessite gravar todo o seu estoque de cada dia e atualizar eles. Existem dois métodos para essa gravação dos dados, são eles: Copiar todos os dados já salvos e adicionar o que é necessário, ou criar um método persistente que aproveita os dados já salvos em versões anteriores. O método persistente vêm com o objetivo de melhorar a eficiência e a quantidade de espaço necessária.

## 2.2 Diferença entre os métodos e vantagens

O método da árvore persistente, busca aproveitar os dados já salvos em versões anteriores do programa, reduzindo assim, o tempo necessário para inserir um novo conjunto de dados, e o espaço físico para a inserção desses dados. Já a cópia da árvore inteira a cada atualização, reduz a velocidade do programa e sua eficiência, e enfim, exige uma quantidade de armazenamento físico grande para a nova inserção de dados.

## 2.3 Desenvolvimento do método de cópia da árvore

### 2.3.1 *Método Cópia*

Após a captura dos dados que estão dentro do arquivo, o método de cópia é utilizado para criar uma árvore de acordo com esses arquivos e realizar as operações necessárias descrita na própria leitura do aquivo. Foi criado um contador para contar a quantidade de versões, sendo que cada versão é correspondida por cada linha do arquivo lida . Então, cria-se uma vetor de árvores para armazenar as cópias das árvores com versões anteriores.

#### Implementação do *Método*

```
1     arvore historico[contadordeversao];
2         for (int i = 0; i < contadordeversao; i++) {
3                 if (i == 0) {
4                         arvore arvoreaux;
5                         arvoreaux.atualizar(produtos[i]);
6                         historico[i] = arvoreaux;
7                 } else {
8                         arvore temp = historico[i - 1];
9                         temp.atualizar(produtos[i]);
10                        historico[i] = temp;
11                }
12        }
```

### 2.3.2 Métodos auxiliares ao método de cópia

Percebe-se que foi utilizado métodos auxiliares para a realização da cópia da árvore e o funcionamento da mesma. Essas são descritas abaixo:

#### Método interno : Construtor de cópia

Com o construtor da cópia podemos então copiar a árvore passada como referência.

```
1 arvore::arvore(arvore& st) {
2         if (st.raiz == NULL)
3                 raiz = NULL;
4         else
```

```
5                   copiararvore(this->raiz, st.raiz); //--> metodo de
                       copia
6
7   }
```

### Método interno: Sobrecarga do operador "="

Foi criado uma sobrecarga interna à arvore para realizar operações sobre esta mesma. Neste método de cópia, o operador irá copiar toda á arvore que estará após o operador = para aquela que estará antes deste. Ex.: $arvore1 = arvore2$, portanto a $arvore1$ será a cópia da $arvore2$.

```
1
2   void arvore::copiararvore(Node *& n, Node *& sn) {
3           if (sn == NULL)
4                   n = NULL;
5           else {
6                   n = new Node(sn->item);
7                   copiararvore(n->esquerda, sn->esquerda);
8                   copiararvore(n->direita, sn->direita);
9
10          }
11  }
12
13  void arvore::operator=(arvore& st) {
14          if (st.raiz == NULL)
15                  raiz = NULL;
16          else
17                   copiararvore(this->raiz, st.raiz); //--> metodo de
                        copia
```

### Método interno da árvore: Atualizar

Neste método, como o próprio nome diz, ele atualiza a árvore de acordo com o par de dados passado como referência. Irá adicionar ou remover produtos de acordo com que o arquivo pede. Como descrito acima o operador irá copiar árvore e logo em seguida o método Atualizar irá adicionar o novo par, seguindo as ordens do arquivo.

```
1   void arvore::atualizar(pair<int, int> r) {
```

```
        if (buscar(r)) {
                Node *aux = pegarnodo(r, raiz);
                if (aux) {
                        aux->item.second += r.second;
                        if (aux->item.second <= 0)
                                remover(r);
                } else
                        cout << "Erro ao atualizar estoque!" << endl;
        } else {
                raiz = inserir(r, raiz);
        }
}
```

## 2.4 Desenvolvimento do método de árvore persistente

### 2.4.1 Método Persistente

Com os dados obtidos do arquivo é feito um contador para a contagem de versões do programa. Esse contador é igual a quantidade de datas do programa, sendo assim, cria-se um vetor de árvores contíguo para armazenar os aproveitamentos das arvores anteriores, como o método persistente propõe. O método propõe que em vez da cópia da árvore, criar apontadores para aqueles nós desta árvore que não foram modificados e adicionar somente em nós que foram modificados.

```
arvore historicopers [contadordeversao];
        for (int i = 0; i < contadordeversao; i++) {
                if (i == 0) {
                        arvore arvoreaux;
                        arvoreaux.atualizar(produtos[i]);
                        historicopers[i] = arvoreaux;
                } else {
                        arvore arvoreaux;
                        arvoreaux.metodopers(produtos[i],
                           historicopers[i - 1]);
                        historicopers[i] = arvoreaux;
                }
```

```
12          }
```

### 2.4.2 Métodos Internos ao método persistente

Como podemos perceber, igualmente no método de cópia, foram utilizados métodos internos para o funcionamento do método persistente. Seguem eles:

#### Método interno: Construtor de cópia (Apontar árvore)

Igualmente no outro método de cópia, o método persistente também possui um construtor de cópia, porém este construtor, diferente do construtor do método de cópia, não irá copiar a arvore, apenas apontar para ela, criando assim um apontador de árvore, e não uma nova árvore. Portanto, não é alocado uma nova árvore, apenas é apontado.

```
1  arvore::arvore(arvore& st) {
2          if (st.raiz == NULL)
3                  raiz = NULL;
4          else
5
6                  this->raiz=st.raiz; \\--> metodo persistente
7  }
```

#### Método interno: Sobrecarga do Operador =

Utiliza-se também a sobrecarga de operadores neste método para que possamos apontar uma certar árvore para a outra. Por exemplo: $arvore1 = arvore2$, temos aqui que a $arvore1$ irá apontar para a $arvore2$ e não copia-la.

```
1
2  void arvore::operator=(arvore& st) {
3          if (st.raiz == NULL)
4                  raiz = NULL;
5          else
6                  this->raiz = st.raiz;\\-->metodo persistente
```

#### Método interno: Atualizar

O método interno Atualizar, é também utilizado aqui, para que se adicione o primeiro item em uma árvore para que se aproveite os seus caminhos. Percebe-se que o método Adicionar é

utilizado somente uma vez, dentro da repetição, quando o $i = 0$.

```
void arvore::atualizar(pair<int, int> r) {
        if (buscar(r)) {
                Node *aux = pegarnodo(r, raiz);
                if (aux) {
                        aux->item.second += r.second;
                        if (aux->item.second <= 0)
                                remover(r);
                } else
                        cout << "Erro ao atualizar estoque!" << endl;
        } else {
                raiz = inserir(r, raiz);
        }
}
```

### Método interno: Método Persistente

Após atualizar a árvore, criando-a com os primeiros dados, como descrito acima, é utilizado o método persistente. Este método aproveita os nós não modificados, apenas criando apontadores para eles. Para aqueles nós que serão modificados, ou seja, adicionados na árvore, é alocado um novo nó com o item desejado nele. como sugere a imagem a seguir.

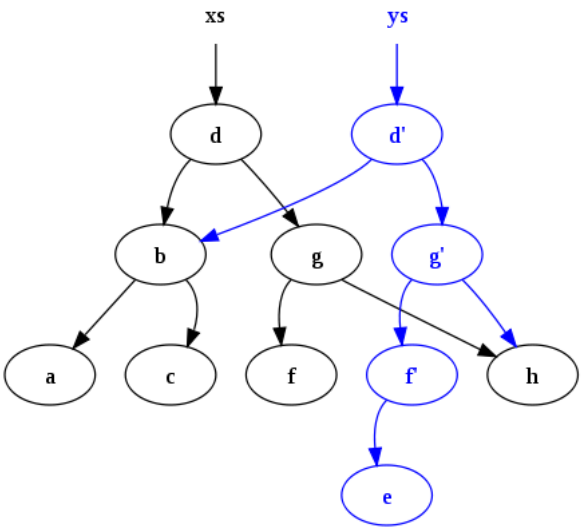


Figura 2.1: Método Persistente

### Implementação do método persistente

```
void arvore::metodopers(pair<int, int> r, arvore& t) {
        raiz = new Node(t.raiz->item);
        Node *anterior = t.raiz, *atual = raiz;
        while (anterior != NULL && anterior->item.first != r.first) {
                if (r.first > anterior->item.first) {
                        atual->esquerda = anterior->esquerda;
                        anterior = anterior->direita;
                        if (anterior != NULL)
                                atual->direita = new Node(anterior->
                                   item);
                        else
                                atual->direita = new Node(r);
                        atual = atual->direita;
                } else {
                        atual->direita = anterior->direita;
                        anterior = anterior->esquerda;
                        if (anterior != NULL)
                                atual->esquerda = new Node(anterior->
                                   item);
                        else
                                atual->esquerda = new Node(r);
                        atual = atual->esquerda;
                }
        }
        if (anterior != NULL) {
                atual->esquerda = anterior->esquerda;
                atual->direita = anterior->direita;
                atual->item.second += r.second;
                if (atual->item.second <= 0)
                        remover(atual->item);
        } else {
                atual->item.second = r.second;
        }
}
```

## 2.5 *Remover*

O método interno remover possui sua própria seção pois é utilizado tanto para o método de cópia quanto para o método persistente. Este método é comum aos métodos de remoção em árvores binárias.A seguir a implementação deste:

```
1  Node* arvore::remover(pair<int, int> r, Node* n) {
2          Node *temporario = NULL;
3          if (n != NULL) {
4                  if (r.first == n->item.first) {
5                          temporario = n;
6                          n = removernodo(n);
7                  }
8                  else if (r.first > n->item.first)
9                          n->direita = remover(r, n->direita);
10                 else
11                         n->esquerda = remover(r, n->esquerda);
12         }
13         if (temporario)
14                 delete temporario;
15         return n;
16 }
17
18 Node* arvore::removernodo(Node* n) {
19         if (n->esquerda == NULL)
20                 return n->direita;
21         else if (n->direita == NULL)
22                 return n->esquerda;
23         else
24                 return removermax(n);
25 }
26
27 Node* arvore::removermax(Node* n) {
28         if (n->esquerda->direita == NULL) {
29                 n->esquerda->direita = n->direita;
30                 return n->esquerda;
31         } else {
```

```
                Node *nodoaux1 = n->esquerda;
                Node *nodoaux2 = n->esquerda->direita;
                while (nodoaux2->direita != NULL) {
                        nodoaux1 = nodoaux2;
                        nodoaux2 = nodoaux2->direita;
                }
                nodoaux1->direita = nodoaux2->esquerda;
                nodoaux2->esquerda = n->esquerda;
                nodoaux2->direita = n->direita;
                return nodoaux2;
        }
}
```

## 2.6 *Busca*

Assim como o método de remoção, o método de busca é utilizado tanto no método de cópia quanto no método persistente. Com o vector *datas* e a criação contígua, tomando referência o vector *datas*, do vetor de árvores, é pedido ao usuário digitar uma data e uma hora desejada para se obter os dados até aquele certa data e hora. Com isso o usuário terá as informações impressas na tela.

```
 string abc;
    cout<<"Digite a data para recuperacao dos registros"<<endl;
    getchar();
    getline(cin,abc);

    int k = -1;
    for (int j = 0; j < static_cast<int>(data.size()); j++) {

        if (abc == data[j]) {

            k = j;
            if (k != -1) {

                historicopers[k].printar();

```

```
16                      break;
17                  }
18              }
19
20      }
```

# Capítulo 3

# Estudo Analítico entre os métodos

## 3.1 Utilização de espaço e tempo

O presente estudo analítico foi realizado com os arquivos propostos pelo Professor Wagner Barros. Os arquivos são divididos em arquivos para testes de funcionamento e arquivos para teste de desempenho.

Nos arquivos de teste de funcionamento, os dois métodos possuíram êxito em realizar as operações, isso se deve a pequena quantidade de versões. Por exemplo, no arquivo de teste de funcionamento *compra-venda-1a.txt* existem apenas 10 versões ( quantidade de linhas ).

Não obstante, nos arquivos de teste de desempenho, apenas o método persistente obteve sucesso em realizar as operações, esse fato ocorreu pois no método de cópia há uma alocação de árvore para cada versão, por exemplo, no arquivo de teste de desempenho *compra-venda-1c.txt* existem 100000 versões, portanto o sistema operacional por sua própria natureza irá matar o programa pela quantidade de alocações que está sendo realizada, dando o erro *bad alloc*. Contudo, no método persistente, o arquivo descrito foi facilmente processado, já que neste método é utilizado apontadores para todo nodo não modificado e não uma cópia de toda a árvore, sendo assim, este método não ocupa tanta memória para realizar as operações, diferentemente do método de cópia.

Conclui-se então que a persistência é melhor do que a cópia, já que essa não precisa de uma grande quantidade de alocação de memória, sendo inquestionavelmente melhor.

O estudo analítico de tempo de execução não foi possível, já que os arquivos de teste de desempenho não rodaram no método de cópia.