Trabalho 2

Algoritmo e Estruturas de Dados - 4645G-04

Professor: Henry Cabral

Turma: 12

Pedro Henrique Nunes Oliveira

Introdução

O programa constrói uma hierarquia organizacional a partir de pares (supervisor, subordinado) lidos de um arquivo e verifica se uma ordem é autorizada segundo regras hierárquicas (autorizante dois níveis acima ou três colegas no nível acima). A verificação hierárquica é importante para assegurar fluxos de autorização formais.

Estrutura de dados usada

Utiliza-se uma árvore genérica implementada por objetos Tree (derivados de TreeTAD). Cada nó guarda o item (nome), ponteiro para o pai e um vetor de ponteiros para filhos (std::vector<TreeTAD*>), representando diretamente relações supervisor → subordinado.

```
class Tree : public TreeTAD{
private:
    TreeTAD* parent;                    // Ponteiro para o nó pai
    string item;                        // Item armazenado no nó
    std::vector<TreeTAD*> children;     // Vetor para armazenar nós filhos

public:
    // Construtor
    Tree(string value){
        this->item = value;
        this->parent = nullptr;
    }

    // Destrutor
    ~Tree() {
        for (TreeTAD* child : children) {
            delete child;               // Libera a memória dos filhos
        }
```

```
    }

    // Adiciona um nó filho
    void addChild(TreeTAD* tree) {
        children.push_back(tree);
        Tree* tree2 = dynamic_cast<Tree*>(tree);
        if (tree2) {
            tree2->parent = this;           // Define o pai do nó filho
        }
    }

    // Obtém o tamanho da árvore (n° de nós)
    int size() {
        int totalSize = 1; // Contando o nó atual
        for (TreeTAD* child : children) {
            totalSize += child->size(); // Contando nós filhos
        }
        return totalSize;
    }

    // Obtém o item armazenado no nó
    const string& getItem() const {
        return item;
    }

    // Obtém o nó pai
    TreeTAD* getParent(){
        return parent;
    }

    // Obtém um nó filho na posição especificada
    TreeTAD* getChild(int index) {
        if (index >= 0 && index < static_cast<int>(children.size())) {
            return children[index];
        }
        return nullptr;
    }

    // Retorna referência privada do numero de filhos
    const std::vector<TreeTAD*>& getChildren() const {
        return children;
    }
};
```

Código do algoritmo de leitura

```
    // ============================
    // LEITURA DO ARQUIVO
    // ============================
    ifstream file("../nomes.txt");
    if (!file.is_open()) {
        cerr << "Erro ao abrir o arquivo." << endl;
        return 1;
    }

    int n;
    if (!(file >> n)) {
        cerr << "não conseguiu ler o número de relações." << endl;
        return 1;
    }

    vector<string> nomes;
    string nome;
    while (file >> nome) {
        // Tira aspas
        if (!nome.empty() && nome.front() == '"') nome.erase(0,1);
        if (!nome.empty() && nome.back() == '"') nome.pop_back();
        nomes.push_back(nome);
    }
    file.close();

    if (nomes.empty()) {
        cerr << "Arquivo não contém pares de nomes." << endl;
        return 1;
    }
```

Explicação da leitura

O arquivo é lido palavra por palavra; o primeiro inteiro lido é armazenado em *n*. Em seguida, pares de strings (pai, filho) são coletados em um vetor *nomes*. O algoritmo cria o nó raiz com o primeiro nome e itera sobre os pares, mantendo *pai_atual*. Quando o nome do pai muda, procura-se linearmente na lista de nós já criados para identificar o nó correspondente; o filho é criado e anexado ao vetor de filhos do *pai_atual*.

Código da função de verificação

```
bool verificar(Tree* node){
    if (!node) return false;
    if (!node->getParent()){
        cout << "É a raiz" << endl;
```

```
        return true;
    }

    // Primeira opção: Verifica se existe alguém 2 niveis acima
    Tree* primeiroPai = dynamic_cast<Tree*>(node->getParent());
    if(primeiroPai){
        Tree* segundoPai = dynamic_cast<Tree*>(primeiroPai->getParent());
        if(segundoPai){
            return true;
        }
    }

    // Segunda opção: Verifica se existe 3 pessoas um nivel acima
    if (primeiroPai && primeiroPai->getChildren().size() >= 3) {
        return true;
    }

    return false;
}
```

Na função *main()* é criada a hierarquia e é chamada a função *verificar()*:

```
    // ========================
    // CRIAÇÃO DA HIERARQUIA
    // ========================
    vector<Tree*> nos;
    Tree* raiz = new Tree(nomes[0]);
    nos.push_back(raiz);

    Tree* pai_atual = raiz;
    string pai_anterior = nomes[0];

    for (size_t i = 0; i + 1 < nomes.size(); i += 2) {
        string paiNome = nomes[i];
        string filhoNome = nomes[i + 1];

        // Se o pai mudou, encontra o nó correspondente na lista
        if (paiNome != pai_anterior) {
            for (Tree* no : nos) {
                if (no->getItem() == paiNome) {
                    pai_atual = no;
                    break;
                }
            }
            pai_anterior = paiNome;
        }

        Tree* filho = new Tree(filhoNome);
```

```
        pai_atual->addChild(filho);
        nos.push_back(filho);
    }

    // ==========================
    // VERIFICAÇÃO DE CONDIÇÕES
    // ==========================
    cout << "\n=== Verificação de Condições ===\n" << endl;
    // Verifica a condição para o nó "Ana"
    for (Tree* no : nos) {
        if (no->getItem() == "Ana") {
            if (verificar(no)) {
                cout << "Ana atende à condição." << endl;
            } else {
                cout << "Ana não atende à condição." << endl;
            }
            break;
        }
    }
```

Explicação da verificação

A função retorna verdadeiro se: o nó é a raiz; ou existe um avô (dois níveis acima); ou o pai do nó possui ao menos três filhos (três autorizações no nível imediatamente superior). A verificação faz apenas acessos ao pai, ao avô e ao tamanho do vetor de filhos (operações de custo constante).

Casos de teste e saídas esperadas

Exemplo 1:

Entrada (nomes.txt):

6

"Joao" "Maria"

"Joao" "Carlos"

"Maria" "Ana"

"Maria" "Bruno"

"Carlos" "Paula"

"Carlos" "Rafael"

Saida:

Joao -> Maria

Joao -> Carlos

Maria -> Ana
Maria -> Bruno
Carlos -> Paula
Carlos -> Rafael

=== Estrutura da Árvore ===
Joao
  Maria
    Ana
    Bruno
  Carlos
    Paula
    Rafael

=== Verificação de Condições ===
Ana atende à condição.

Exemplo 2

Entrada (nomes2.txt):
1
"Joao" "Ana"

Saída:
Joao -> Ana

=== Estrutura da Árvore ===
Joao
  Ana

=== Verificação de Condições ===
Ana não atende à condição.

Análise da complexidade

➢ Verificar(node) executa um número constante de operações (acesso ao pai, possível acesso ao avô e checagem do tamanho do vetor de filhos): tempo O(1) no melhor e pior caso.

➢ A construção da árvore, como implementada (busca linear na lista de nós quando o pai muda), pode ser O(N^2) no pior caso (N = número de pares/nós);

➢ Memória O(N) para armazenar nós.

Conclusão

A implementação usa uma árvore genérica simples com ponteiros pai/filhos, leitura sequencial de pares e uma verificação eficiente O(1) por nó. A principal limitação identificada é a busca linear por pai durante a construção (pode-se otimizar com um dicionário mapa<string,Tree*> para reduzir a construção para O(N)). A solução é funcional e facilita extensões e otimizações futuras.

Código Completo:

tree.cpp:

```
#include "TreeTAD.h"
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <string>

using namespace std;

class Tree : public TreeTAD{
private:
    TreeTAD* parent;                    // Ponteiro para o nó pai
    string item;                        // Item armazenado no nó
    std::vector<TreeTAD*> children;     // Vetor para armazenar nós filhos

public:
```

```
// Construtor
Tree(string value){
    this->item = value;
    this->parent = nullptr;
}

// Destrutor
~Tree() {
    for (TreeTAD* child : children) {
        delete child;              // Libera a memória dos filhos
    }
}

// Adiciona um nó filho
void addChild(TreeTAD* tree) {
    children.push_back(tree);
    Tree* tree2 = dynamic_cast<Tree*>(tree);
    if (tree2) {
        tree2->parent = this;           // Define o pai do nó filho
    }
}

// Obtém o tamanho da árvore (n° de nós)
int size() {
    int totalSize = 1; // Contando o nó atual
    for (TreeTAD* child : children) {
        totalSize += child->size(); // Contando nós filhos
    }
    return totalSize;
}

// Obtém o item armazenado no nó
const string& getItem() const {
    return item;
}

// Obtém o nó pai
TreeTAD* getParent(){
    return parent;
}

// Obtém um nó filho na posição especificada
TreeTAD* getChild(int index) {
    if (index >= 0 && index < static_cast<int>(children.size())) {
        return children[index];
    }
    return nullptr;
}
```

```
    // Retorna referência privada do numero de filhos
    const std::vector<TreeTAD*>& getChildren() const {
        return children;
    }
};

void printTree(Tree* node, int nivel = 0) {
    if (!node) return;
    for (int i = 0; i < nivel; ++i) cout << "  ";
    cout << node->getItem() << endl;
    for (TreeTAD* child : node->getChildren()) {
        Tree* childTree = dynamic_cast<Tree*>(child);
        if (childTree) {
            printTree(childTree, nivel + 1);
        }
    }
}

// Função verificar ordem
//  > 1 nó dois niveis acima
//  ou
//  > 3 nós um nível acima
bool verificar(Tree* node){
    if (!node) return false;
    if (!node->getParent()){
        cout << "É a raiz" << endl;
        return true;
    }

    // Primeira opção: Verifica se existe alguém 2 niveis acima
    Tree* primeiroPai = dynamic_cast<Tree*>(node->getParent());
    if(primeiroPai){
        Tree* segundoPai = dynamic_cast<Tree*>(primeiroPai->getParent());
        if(segundoPai){
            return true;
        }
    }

    // Segunda opção: Verifica se existe 3 pessoas um nivel acima
    if (primeiroPai && primeiroPai->getChildren().size() >= 3) {
        return true;
    }

    return false;
}

int main(){
```

```cpp
    // ==========================
    // LEITURA DO ARQUIVO
    // ==========================
    ifstream file("../nomes.txt");
    if (!file.is_open()) {
        cerr << "Erro ao abrir o arquivo." << endl;
        return 1;
    }

    int n;
    if (!(file >> n)) {
        cerr << "não conseguiu ler o número de relações." << endl;
        return 1;
    }

    vector<string> nomes;
    string nome;
    while (file >> nome) {
        //Tira aspas
        if (!nome.empty() && nome.front() == '"') nome.erase(0,1);
        if (!nome.empty() && nome.back() == '"') nome.pop_back();
        nomes.push_back(nome);
    }
    file.close();

    if (nomes.empty()) {
        cerr << "Arquivo não contém pares de nomes." << endl;
        return 1;
    }

    // Teste imprimir pares
    for (size_t i = 0; i + 1 < nomes.size(); i += 2){
        cout << nomes[i] << " -> " << nomes[i + 1] << endl;
    }

    // ==========================
    // CRIAÇÃO DA HIERARQUIA
    // ==========================
    vector<Tree*> nos;
    Tree* raiz = new Tree(nomes[0]);
    nos.push_back(raiz);

    Tree* pai_atual = raiz;
    string pai_anterior = nomes[0];

    for (size_t i = 0; i + 1 < nomes.size(); i += 2) {
        string paiNome = nomes[i];
        string filhoNome = nomes[i + 1];
```

```
        // Se o pai mudou, encontra o nó correspondente na lista
        if (paiNome != pai_anterior) {
            for (Tree* no : nos) {
                if (no->getItem() == paiNome) {
                    pai_atual = no;
                    break;
                }
            }
            pai_anterior = paiNome;
        }

        Tree* filho = new Tree(filhoNome);
        pai_atual->addChild(filho);
        nos.push_back(filho);
    }

    cout << "\n=== Estrutura da Árvore ===\n";
    printTree(raiz);

    // ========================
    // VERIFICAÇÃO DE CONDIÇÕES
    // ========================
    cout << "\n=== Verificação de Condições ===\n" << endl;
    // Verifica a condição para o nó "Ana"
    for (Tree* no : nos) {
        if (no->getItem() == "Ana") {
            if (verificar(no)) {
                cout << "Ana atende à condição." << endl;
            } else {
                cout << "Ana não atende à condição." << endl;
            }
            break;
        }
    }
    delete raiz; // libera memória
    return 0;
}
```

TreeTAD.h

```
// ****************************************
//  Classe abstrata TreeTAD
// ****************************************

// É o tipo abstrato de dados genérico para que outros tipos de árvore possam usar o
mesmo TAD
```

```cpp
#ifndef TREETAD_H
#define TREETAD_H

#include <string>
#include <vector>

class TreeTAD {
public:
    virtual ~TreeTAD() = default;

    // Manipulação de filhos
    virtual void addChild(TreeTAD* child) = 0;
    virtual TreeTAD* getChild(int index) = 0;
    virtual const std::vector<TreeTAD*>& getChildren() const = 0;

    // Informações do nó
    virtual int size() = 0;
    virtual const std::string& getItem() const = 0;
    virtual TreeTAD* getParent() = 0;
};

#endif // TREETAD_H
```