Trabalho prático 1

Resolvedor de Expressões Numéricas

Iago da Silva Rodrigues Alves – 2022035881

iagosilva92@ufmg.br

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG – Brasil

1. Introdução

O projeto em questão é uma implementação de uma calculadora de expressões matemáticas em C++, que possui recursos para a validação e resolução de expressões complexas. As expressões podem ser fornecidas em notação infixa ou pós-fixa, e a implementação é capaz de lidar com a presença de parênteses e múltiplos operadores e operandos. Dessa forma, nessa implementação, o programa é capaz de lidar com expressões de até 1000 caracteres e verificar se a expressão é válida, além de ler, converter e resolver expressões matemáticas com uma interface simples.

2. Método

2.1 Ferramentas

Esse projeto utiliza as seguintes ferramentas e especificações:

- Linguagem: *C++11*
- Compilação: *g++ 11.3.0*
- Sistema Operacional: *Ubuntu 22.04*
- GNU Make 4.3

2.2 Tipos Abstratos de Dados

Este projeto contém quatro arquivos de cabeçalho para definição dos TAD's, são eles:

Expressao.hpp que engloba as classes Expressao e Node. A classe Expressão é uma classe abstrata que define a interface para as expressões matemáticas. A classe possui três métodos virtuais puros que devem ser implementados pelas classes derivadas, eles possuem as funções principais para a manipulação da expressão, são elas: ler, avaliar e converter. A classe Node é uma classe auxiliar que é utilizada para

construir a árvore binária. Cada nó da árvore representa um operando ou um operador da expressão.

Infixa.hpp que abrange a classe Infixa. A classe infixa é uma classe derivada da classe base Expressao. Ela é usada para representar uma expressão matemática na notação infixa, assim uma de suas finalidades é implementar os métodos virtuais da classe base visando a manipulação de expressões no formato infixo.

Posfixa.hpp que abrange a classe Posfixa. A classe Posfixa é também uma classe derivada da classe base Expressao. Ela é usada para representar uma expressão matemática na notação posfixa e, portanto, desenvolve os métodos virtuais da classe base com esse propósito.

Pilha.hpp que abrange a classe Pilha. A classe Pilha é uma classe auxiliar que é utilizada para armazenar os operandos e os operadores nas operações com a expressão. A classe Pilha foi desenvolvida como um template para torná-la mais genérica e permitir o armazenamento de diferentes tipos de dados. Vale ressaltar que a pilha onde é encadeada.

2.3 Funções

O programa possui quatro funções principais para sua funcionalidade, são elas: Função **avaliaExpressao** que avalia a expressão matemática e retorna um booleano indicando se a avaliação foi bem-sucedida ou não; Função **lerExpressao** que lê a expressão e armazena se a mesma for válida; Função **converteExpressao** que converte a expressão matemática da notação infixa para a notação pós-fixa ou vice-versa; Função **resolveExpressao** que resolve a expressão matemática;

Os três primeiros métodos são virtuais na classe Expressão e são implementados nas classes Infixa e Posfixa. Já o método **resolveExpressao,** como ambas as expressões são armazenadas da mesma forma. esse método é implementado diretamente na classe pai.

3. Análise de Complexidade

Considerando as funções principais citadas acima, analisando suas complexidades percebi que todas possuem o mesmo valor: O(n). Isso ocorre, pois, essas funções sempre irão depender do tamanho n da expressão armazenada.

Outrossim, é válido destacar que além das funções principais mencionadas acima, existem outras funções no projeto que possuem complexidades diferentes. Algumas funções possuem complexidade constante, O(1), independentemente do tamanho da expressão ou da árvore, como é o caso das funções "ehOperador" e

"expressaoExiste". Já outras funções, como "converteRecursivo" e "resolveRecursivo", possuem complexidade linear em relação ao número de nós da árvore, já que percorrem todos os nós para realizar as operações necessárias.

Ao analisar as complexidades das funções principais e demais funções do projeto, é possível entender como o tempo de execução é afetado pelo tamanho da entrada. As funções principais possuem complexidade linear O(n) em relação ao tamanho da expressão armazenada, enquanto outras funções possuem complexidade constante O(1).

Além disso, é importante ressaltar a complexidade de espaço, que está relacionada ao número de nós criados e à utilização de memória durante a execução do programa. A complexidade de espaço varia de acordo com o tamanho da expressão, ao construir a árvore de expressão, serão criados nós correspondentes a cada elemento da expressão, ocupando espaço na memória. Portanto, a complexidade de espaço é influenciada pelo tamanho da expressão tendo em vista o número de nós necessários para representar a árvore.

4. Estratégias de Robustez

O programa implementa mecanismos de programação defensiva e tolerância a falhas a fim de garantir a robustez e confiabilidade do sistema e para isso ele utiliza a biblioteca **<stdexcept>.** Podemos observar a utilização desses mecanismos nas funções **expressaoExiste** e **printLerExpressao**. A função **expressaoExiste** é responsável por verificar se a expressão existe, ou seja, se a string contendo a expressão não está vazia. Caso a expressão não exista, imprime uma mensagem de erro na saída de erro padrão e interrompe a execução. Isso evita que o programa continue executando operações com uma expressão inválida, prevenindo erros mais graves. A função **printLerExpressao**, por sua vez, é responsável por validar a expressão lida do arquivo de entrada. Caso a expressão seja inválida, a função imprime uma mensagem de erro na saída de erro padrão. Isso garante que apenas expressões válidas serão manipuladas pelo programa.

Além dos mecanismos de programação defensiva, o programa implementa também mecanismos de tolerância a falhas. Esses mecanismos são utilizados para lidar com erros que possam ocorrer durante a execução do programa e que não possam ser prevenidos. No caso desse programa, podemos observar a utilização desses mecanismos nas funções **printExpressaoPosfixa**, **printExpressaoInfixa**, **printSolucaoPosfixa** e **printSolucaoInfixa.** Todas essas funções são responsáveis por realizar operações em expressões aritméticas, como converter uma expressão infixa em posfixa ou calcular o valor de uma expressão posfixa. Caso ocorra algum erro durante essas operações, como uma divisão por zero, a função captura a exceção

e imprime uma mensagem de erro na saída de erro padrão. Exemplo outra de robustez aplicada no programa:

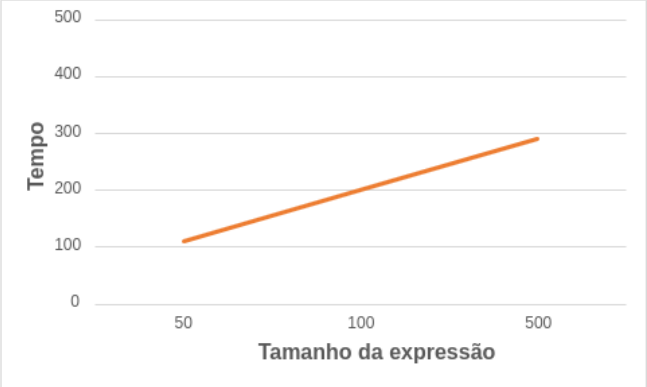
```cpp
// Abre o arquivo de entrada
std::ifstream arquivo("arquivo.txt");
if (!arquivo.is_open()) {
  std::cerr << "Erro ao abrir o arquivo de entrada" << std::endl;
  return 1;
}
```

Outrossim, o programa não utiliza a declaração do namespace std para evitar conflitos de nomes e a poluição do namespace. E também todos os construtores presentes seguem a implementação na forma "member initializer constructor" que traz benefícios como aumento de performance e redução de complexidade de código. Evitando que os membros sejam inicializados na implementação do construtor

5. Análise Experimental

Além da análise de complexidade teórica, outra abordagem importante para avaliar o desempenho de um algoritmo é a análise experimental. No entanto, é importante ressaltar que a análise experimental também tem suas limitações, uma vez que os resultados podem ser afetados por fatores externos, como a arquitetura do computador, o sistema operacional e a carga de trabalho do sistema no momento dos testes. Por isso, é importante realizar vários testes e calcular a média dos resultados para obter uma estimativa mais precisa do desempenho do algoritmo.

Abaixo trago um gráfico que compara o tempo de execução do programa em microssegundos e o tamanho das expressões fornecidas em número de caracteres. Os testes foram feitos com a passagem dos parâmetros de leitura, conversão e resolução.



Ao plotar o gráfico tempo x tamanho da expressão, é possível identificar padrões e tendências na curva, que podem indicar a complexidade do algoritmo. A curva claramente apresenta uma inclinação linear, isso sugere que a complexidade é proporcional ao tamanho da entrada. Logo, significa que, à medida que o tamanho da entrada aumenta, o tempo de execução do programa aumenta linearmente.

6. Conclusões

Neste projeto foi desenvolvida uma implementação em C++ de uma calculadora para expressões matemáticas que apresenta recursos de validação, conversão e resolução de expressões complexas. A implementação pode lidar com expressões fornecidas em notação infixada ou posfixada.

Posto isso, para alcançar esse objetivo, foram aplicados diversos conceitos e técnicas aprendidas em sala de aula, tais como modularização, depuração e a construção de tipos abstratos de dados. Outrossim, o programa foi projetado para garantir a robustez e a tolerância a falhas, contando com mecanismos de programação defensiva e estratégias de tratamento de exceções, fornecendo uma base sólida para futuras extensões e melhorias.

7. Bibliografia

Slides da disciplina de Estrutura de Dados. Disponibilizados via Moddle. 1° semestre 2023.

3.12 Expression Trees | Binary Expression Tree | Data Structures Tutorials. *Jenny's Lectures CS IT. Disponível em: < https://www.youtube.com/watch?v=2Z6g3kNymd0>*
Acesso em: 03 maio 23.

3.13 Expression Tree from Postfix | Data Structures Tutorials. Jenny's Lectures CS IT.
Disponível em: <https://www.youtube.com/watch?v=WHs-wSo33MM>
Acesso em: 03 maio 23

8. Instruções para compilação e execução

Passo 1: Baixe o arquivo zipado referente ao trabalho e extraia seu conteúdo. Abra o diretório **TP1**.

Passo 2: Para compilar, digite o comando make no terminal.

Passo 3: Crie um arquivo texto (.txt) dentro do diretório. Nesse arquivo coloque as entradas para a execução do programa.

Atenção: Cada linha do arquivo indica uma entrada para o programa, existem três tipos de entrada. Entrada para leitura e armazenamento: **LER TIPOEXP EXP**, onde **TIPOEXP** deve ser **INFIXA** ou **POSFIXA** e **EXP** deve ser a expressão matemática. Entrada para conversão: Apenas **INFIXA** ou **POSFIXA**, saiba que para sua aplicação deve existir uma expressão armazenada com tipo contrário ao da conversão. Entrada para solução: **RESOLVE.**

Passo 4: Para executar o programa existem duas alternativas.

1. **make run ARGS="-o nomedoarquivo.txt"**
2. **bin/resolvedor -o nomedoarquivo.txt**

Limpeza: Para limpar os diretórios criados, contendo os objetos e o executável, digite **make clean** no terminal.

Observação: O programa foi criado para ser executado recebendo um arquivo de entrada, logo o argumento –o é imprescindível para execução.