Trabalho Prático 01 - Resolvedor de Expressão Numérica

Aluno: Raphael Henrique Braga Leivas
Matrícula: 2020028101

Universidade Federal de Minas Gerais - UFMG
Belo Horizonte

1 INTRODUÇÃO

O Trabalho Prático (TP) consiste essencialmente no desenvolvimento de um resolvedor de expressões numéricas, que recebe expressões algébricas em notação infixa e posfixa, contendo as 4 operações básicas de adição, subtração, multiplicação e divisão, e executa as seguintes operações:

- Leitura e armazenamento da expressão numérica de entrada;
- Conversão para notação posfixa;
- Conversão para notação infixa;
- Cálculo da expressão armazenada, retornando seu valor numérico caso ela seja válida.

Tendo em vista a aplicação de Estruturas de Dados vistas na disciplina, será usado a linguagem C++ para desenvolver duas classes, representando dois Tipos Abstratos de Dados (TADs), as quais serão usadas para atender às especificações do TP de forma modularizada. Assim, o TP será capaz de reforçar conceitos da disciplina vistos até agora e aprimorar o nosso conhecimento com estruturas já conhecidas - como pilhas e filas - e algoritmos.

2 MÉTODO

O software foi desenvolvido em linguagem C++ em ambiente Windows, sendo compilado em WSL Ubuntu 22.04 LTS com g++. O processo de compilação dos módulos é gerido por Makefile, assim mantendo a estrutura da pasta especificada em atividades da disciplina. Para verificar o gerenciamento de memória da aplicação é usado o valgrind, ao passo que para depurar a aplicação foi usado o gdb.

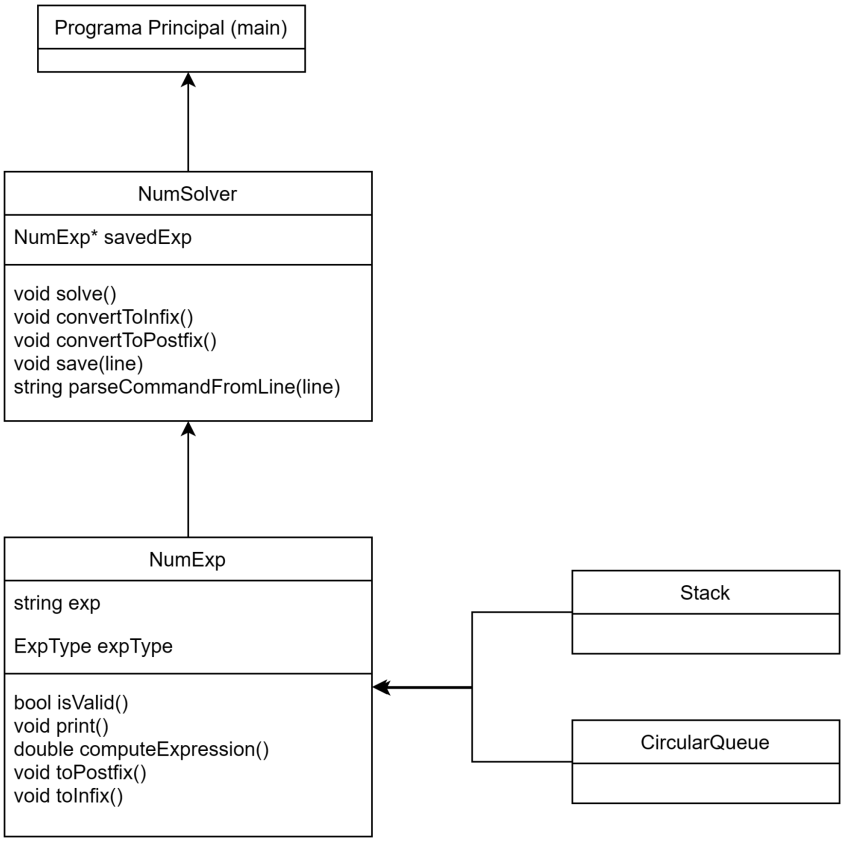
A análise em tempo de execução foi feita usando o gprof, com os dados da análise sendo escritos em arquivo txt após a compilação e execução do código.

2.1 ESTRUTURAS DE DADOS

Para visualizar os Tipos Abstratos de Dados usados e como eles se relacionam, considere o diagrama de classes da Figura 2.1. Como mostra a Figura 2.1, a solução foi implementada em camadas, de tal maneira que o programa principal chama apenas o TAD NumSolver, responsável pelas operações de alto nível de conversão e cálculo da expressão numérica. Note que, do ponto de vista do programa principal, não importa qual a representação interna da expressão ou quais algoritmos são usados: o programa principal apenas chama os métodos do TAD NumSolver e são retornados os resultados

na saída padrão do sistema std::cout. Assim, é garantido a modularização e facilita o processo de desenvolvimento e manutenção do código.

Figura 2.1. Diagrama de classes da solução implementada.



Fonte: elaboração própria.

O TAD NumSolver possui 5 métodos de alto nível implementados:

- void save(line): recebe uma linha inteira do arquivo de entrada, com a expressão numérica e seu tipo (se é infixa ou posfixa, informada na própria linha conforme a especificação do projeto). Se a expressão for válida, ela é salva no atributo de classe savedExp, do tipo NumExp, que será explicado mais à frente;
- void convertToInfix(): converte a expressão armazenada em savedExp para notação infixa;
- void convertToPostfix(): converte a expressão armazenada em savedExp para notação posfixa;
- void solve(): calcula o valor numérico da expressão armazenada, retornando seu valor na saída padrão std::cout;
- string parseCommandFromLine(line): é um método auxiliar que extrai o comando da linha de entrada e retorna para o programa principal, que chamará um dos 4 métodos acima dependendo do valor que o comando assumir. Os possíveis valores de comando são LER, RESOLVE, INFIXA e POSFIXA.

O TAD NumSolver chama um TAD de mais baixo nível, o NumExp, que é a classe que efetivamente representa uma expressão numérica no sistema. A classe possui dois atributos: exp, do tipo string, e expType, que é uma enumeração que assume valor INFIX

ou POSTFIX, representando assim o tipo de notação que a string exp representa. Note que o atributo exp armazena todo o conteúdo da expressão em forma de string, ao passo que o atributo expType atua como um indicador se essa string armazenada é do tipo infixa ou posfixa.

Além desses atributos, o TAD NumExp possui 5 métodos de baixo nível:

- bool isValid(): retorna booleano indicando se a string armazenada em exp é uma expressão válida, dependendo se ela é infixa ou posfixa;
- void print(): printa em std::cout a string em exp;
- double computeExpression(): calcula o valor da expressão numérica representada pela string exp;
- void toPostfix(): converte a string armazenada em exp para notação posfixa, atualizando também o valor do atributo expType;
- void toInfix(): converte a string armazenada em exp para notação infixa, atualizando também o valor do atributo expType.

Como os métodos do TAD NumExp efetivamente realizam as operações matemáticas e algorítmicas do sistema, elas precisam de ter acesso a estruturas de dados auxiliares como a pilha (Stack) e a fila (CircularQueue), implementadas de autoria própria. Essas estruturas são fundamentais na execução dos algoritmos das operações do sistema. O detalhamento dessas operações, bem como suas complexidades, estão exibidas nas seções seguintes desta documentação.

Note que os métodos dos TADs NumSolver e NumExp são bem parecidos - de fato, parecem que fazem a mesma coisa. Contudo, é importante ter essa modularização em camadas para facilitar o uso em certas operações que exigem mais de um método de NumExp na mesma chamada. Essa necessidade ficará mais clara nas seções seguintes.

2.2 IMPLEMENTAÇÃO DAS ESTRUTURAS

As classes Stack e CircularQueue implementam o TAD Pilha e Fila Circular na forma de arranjos e vetores, isto é, com alocação estática, de maneira semelhante às vistas na disciplina.

2.2.1 Classe NumExp

Como os métodos da classe NumExp são de baixo nível, é importante elaborar em mais detalhes como os algoritmos funcionam. Para a classe NumSolver, esse detalhamento não é tão importante pois seus métodos são de mais alto nível devido à modularização em camadas implementada.

O método computeExpression() calcula o valor da expressão numérica a partir do seguinte algoritmo, assumindo que a expressão já está na forma de posfixa. Caso não esteja em posfixa, é necessário que a camada que está chamando esse método (ou seja, a classe NumSolver) antes converta a expressão para posfixa antes de chamar esse método de cálculo.

1. Declara uma pilha de double, inicialmente vazia e tamanho igual à expressão que se deseja calcular;
2. Começa a percorrer cada item da expressão, delimitado por um espaço em branco;
3. Se encontrou um espaço vazio, pula para o próximo item;
4. Se o item atual é um operador, desempilha os dois últimos números da pilha, calcula a operação deles com esse operador, e empilha o resultado na stack;
5. Se o item encontrado é um número, empilha ele na pilha;
6. Quando acabar de percorrer os itens da expressão, o único número que sobrou na pilha é o resultado da expressão, e retorna ele.

O método toInfix() converte a expressão de infixa para posfixa, conforme o seguinte algoritmo:

1. Declara uma pilha de strings, inicialmente vazia e tamanho igual à expressão que se deseja converter;
2. Começa a percorrer cada item da expressão, delimitado por um espaço em branco;
3. Se encontrou um espaço vazio, pula para o próximo item;
4. Se o item atual é um operador, desempilha os dois últimos números da pilha, monta uma expressão infixa com parênteses com os dois números desempilhados e o operador, e empilha essa expressão na pilha;
5. Se o item atual é um número, empilha ele na pilha;
6. Quando terminar de percorrer a expressão, o que sobrou no topo da pilha é o resultado da string inteira. Desempilha ele e retorna.

O método toPostfix() converte a expressão de infixa para posfixa, conforme o seguinte algoritmo:

1. Declara duas pilhas de strings, uma para armazenar o resultado e outra para armazenar operadores (e parênteses que abrem e fecham), inicialmente vazias e de tamanho igual à expressão que se deseja converter;
2. Começa a percorrer cada item da expressão, delimitado por um espaço em branco;
3. Se encontrou um espaço vazio, pula para o próximo item;
4. Se o item atual é um parênteses fechando, puxa operadores da stack e joga eles para a pilha resultado até encontrar um parênteses abrindo. Quando chegar no parênteses abrindo, desempilha ele e volta para o onde parou nos itens da expressão;
5. Se o item atual é um parênteses abrindo, joga ele na stack dos operadores;
6. Se o item atual é um operador, olha quem está no topo da pilha. Se quem está no topo da pilha tem precedência maior que sobre ele, desempilha o topo, empilha o operador atual e empilha o operador que estava no topo novamente. Assim, garante-se a ordem de precedência na expressão final convertida. Se o topo da pilha tem precedência menor que o atual, joga o topo na pilha de resultado e empilha o atual na pilha de operadores;
7. Se o item atual é um número, joga ele na pilha de resultados;
8. Quando terminar de percorrer a expressão, vai desempilhando tudo o que está na pilha de operadores e jogando na pilha de resultado;

9. Ao final, concatena tudo o que está na pilha de resultado em uma string e retorna essa string;

Por fim, para o método isValid(), que verifica se a string em NumExp é válida, o algoritmo usado possui uma validação para expressões infixas e outra para expressões posfixas. O método de validação para infixa foi deixado em branco por dificuldades na implementação e falta de tempo. O método de validação de posfixas essencialmente percorre a string com um contador que simula a posição do topo da pilha. Caso esse ponteiro fique negativo (apresentando underflow na pilha) ou termine em uma posição diferente de 1 (representando overflow na pilha), o algoritmo retorna falso indicando que a expressão avaliada não é válida.

2.2.2 Classe NumSolver

Os métodos da classe NumSolver essencialmente chamam os métodos da classe NumExp, que está armazenada no atributo savedExp. No entanto, como pode ser visto no código fonte, às vezes é necessário chamar mais de um método de NumExp para realizar as operações de alto nível.

Por exemplo, ao chamar a função convertToInfix(), antes é verificado se há uma expressão salva, depois verifica se a expressão salva já está em notação infixa, pois nesse caso não é necessário converter nada. Por último, chama-se o método toInfix() de NumExp.

Demais implementações dos métodos de NumSolver podem ser compreendidos analisando diretamente o código fonte, uma vez que seus métodos são de alto nível.

3 ANÁLISE DE COMPLEXIDADE

Os algoritmos descritos na seção 2.2.1 possuem as complexidades de tempo discutidas abaixo. Em todos os casos, n representa o tamanho da expressão numérica de entrada. Será usado apenas a notação Big-O O pois, para efeitos de documentação, é interessante compreender apenas o limite superior (pior caso) do custo dos algoritmos.

- computeExpression(): temos apenas um loop que percorre a string de entrada uma única vez, e vai calculando o valor da expressão durante o percurso da string com o uso da pilha auxiliar. Assim, tem complexidade assintótica de $O(n)$;
- toInfix(): de modo bastante semelhante ao método computeExpression(), o método toInfix() apenas percorre a string de entrada uma única vez e vai usando a pilha auxiliar para a conversão. Assim, tem complexidade assintótica de $O(n)$;
- toPostfix(): São usadas duas pilhas e, no pior caso, a pilha dos operadores é percorrida inteiramente dentro do loop que varre a string de entrada, ou seja, temos dois loops nestados. Assim, temos complexidade assintótica de $O\left(n^2\right)$;
- isValid(): como apenas percorre a string uma única vez com o contador inteiro representando o topo da stack, temos apenas um loop e assim complexidade $O(n)$

Como os métodos da classe NumSolver apenas chamam os métodos de NumExp, eles possuem a mesma complexidade dos métodos discutidos acima.

Fica claro que o método toPostfix é não somente o algoritmo mais ineficiente do programa (único com $O\left(n^2\right)$), mas também é uma função bastante grande no código fonte quando comparada com as demais. De fato, a melhor opção seria usar uma árvore binária para operações que manipulam notação infixa em vez de usar duas pilhas para isso, as quais são mais adequadas para manipulação de expressões que já estão em posfixa.

Por uma questão de tempo e familiaridade maior com a Estrutura de Dados Pilha do que com a Árvore Binária (a qual foi aprendida 2 semanas antes da entrega desse TP), foi optado por usar esse algoritmo ineficiente com duas pilhas do que o algoritmo com árvores.

4 ESTRATÉGIAS DE ROBUSTEZ

Fazendo uso da modularização entre os TADs, a classe NumSolver de alto nível é responsável pela maior parte das verificações e proteções contra entradas indevidas no sistema, ao passo que os métodos da classe de baixo nível NumExp em geral apenas executam a operação pedida, sem uma verificação do argumento de entrada. Nesse cenário, os métodos de NumExp já recebem os argumentos filtrados pela classe NumSolver na camada superior.

No método save() de NumSolver, a expressão é salva apenas se ela é valida e se a linha de entrada especificou o comando de leitura indicando devidamente qual o tipo da notação que será salva. Se o tipo de notação for inválida, ou a expressão de entrada for inválida, o sistema não salva a expressão de entrada.

Nos métodos convertToInfix() e convertToPostfix() a conversão não é feita se não há expressão armazenada. Além disso, se a expressão a ser convertida já está na notação desejada, o algoritmo de conversão não é chamado, assim evitados gastos computacionais desnecessários.

Por fim, de modo análogo aos demais, o método solve() apenas efetivamente calcula o valor das expressões se haver uma expressão salva em NumSolver; caso não tenha, é retornado uma mensagem de erro em std::cout.

5 ANÁLISE EXPERIMENTAL

Em toda a análise experimental dessa seção, foi considerado a seguinte entrada de dados na aplicação:

LER POSFIXA 2.599367 9.647195 9.673411 9.050032 / 6.358308 - 8.651174 * 4.875167 2.911182 5.348410 2.736445 + 5.191413 * - - * * 1.915312 5.548351 2.050359 0.943320 + * + + 5.181741 5.625093 * * 8.694984 3.686034 / 9.440571 7.117509 / + - 9.446373 8.978700 / 3.580536 + + + 5.250495 1.305610 / * 0.441742 6.502549 7.095268 4.133539 / - * 4.930346 2.550228 / 3.125103 * 4.950563 * * * 6.550852 6.671211 1.809183 + / 3.824251 + * 0.923045 4.509118 6.480367 / 0.074564 / 4.707001 - + 4.743344 3.905694 2.327981 * * 1.462779 - * +

INFIXA

RESOLVE

5.1 GERENCIAMENTO DE MEMÓRIA

A Figura 5.1 exibe a saída do valgrind, usado para verificação do gerenciamento de memória da aplicação.

Figura 5.1. Saída do valgrind.

```
==42==
==42== LEAK SUMMARY:
==42==    definitely lost: 0 bytes in 0 blocks
==42==    indirectly lost: 0 bytes in 0 blocks
==42==      possibly lost: 0 bytes in 0 blocks
==42==    still reachable: 131,352 bytes in 3 blocks
==42==         suppressed: 0 bytes in 0 blocks
==42==
==42== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make: *** [Makefile:40: valgrind] Profiling timer expired
```

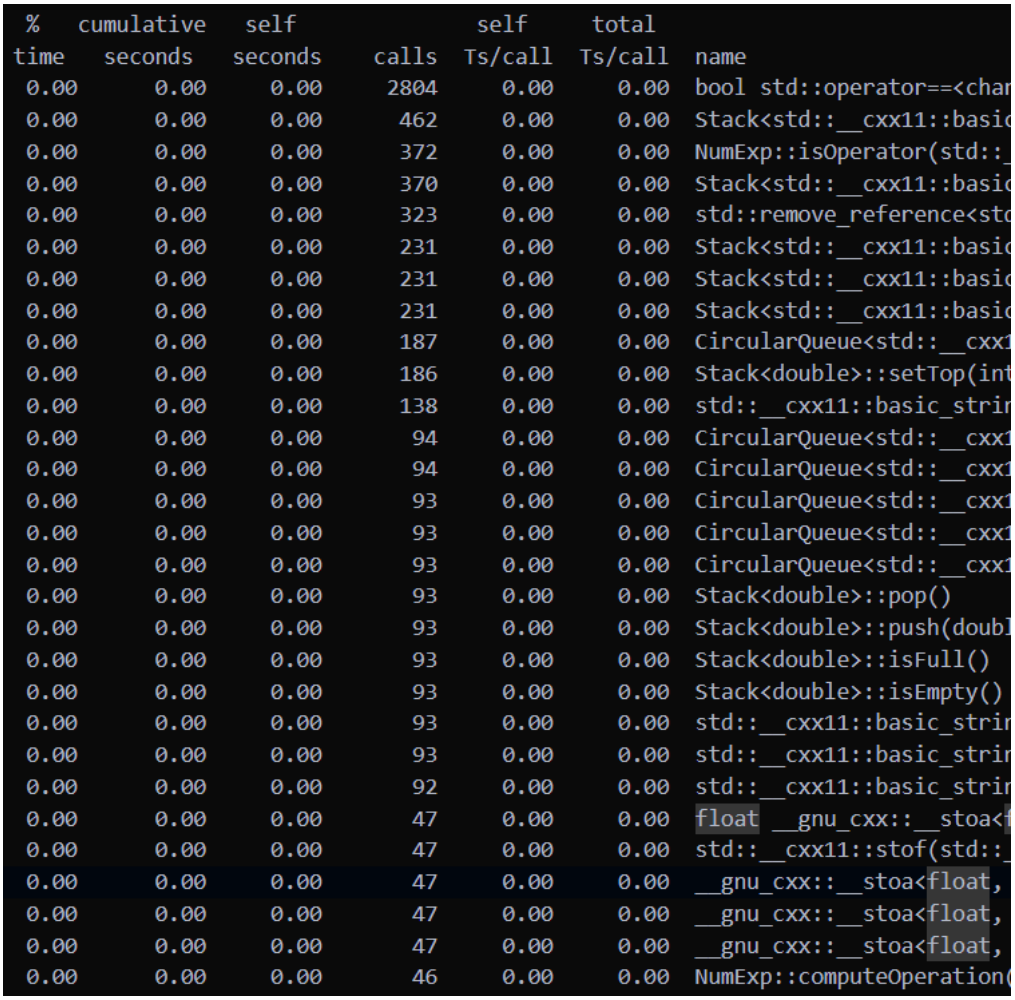
Fonte: elaboração própria.

Conforme exibido na Figura 5.1, a aplicação não exibe vazamentos de memória: todos os ponteiros e alocação de memória foram efetivamente desalocados ao final da execução do programa. O “still reachable” do valgrind às vezes acusa leaks de memória de sistema não alocada pelo código, sendo os principais indicadores na Figura 5.1 exibindo 0 bytes de vazamento de memória.

5.2 ANÁLISE DE EXECUÇÃO

Usando o gprof, as principais chamadas de funções no código durante a execução para a entrada acima estão exibidas na Figura 5.2.

Figura 5.2. Saída do gprof após análise das chamadas de funções em tempo de execução.



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	2804	0.00	0.00	bool std::operator==<char
0.00	0.00	0.00	462	0.00	0.00	Stack<std::__cxx11::basic
0.00	0.00	0.00	372	0.00	0.00	NumExp::isOperator(std::_
0.00	0.00	0.00	370	0.00	0.00	Stack<std::__cxx11::basic
0.00	0.00	0.00	323	0.00	0.00	std::remove_reference<st
0.00	0.00	0.00	231	0.00	0.00	Stack<std::__cxx11::basic
0.00	0.00	0.00	231	0.00	0.00	Stack<std::__cxx11::basic
0.00	0.00	0.00	231	0.00	0.00	Stack<std::__cxx11::basic
0.00	0.00	0.00	187	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	186	0.00	0.00	Stack<double>::setTop(int
0.00	0.00	0.00	138	0.00	0.00	std::__cxx11::basic_strin
0.00	0.00	0.00	94	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	94	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	93	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	93	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	93	0.00	0.00	CircularQueue<std::__cxx1
0.00	0.00	0.00	93	0.00	0.00	Stack<double>::pop()
0.00	0.00	0.00	93	0.00	0.00	Stack<double>::push(doubl
0.00	0.00	0.00	93	0.00	0.00	Stack<double>::isFull()
0.00	0.00	0.00	93	0.00	0.00	Stack<double>::isEmpty()
0.00	0.00	0.00	93	0.00	0.00	std::__cxx11::basic_strin
0.00	0.00	0.00	93	0.00	0.00	std::__cxx11::basic_strin
0.00	0.00	0.00	92	0.00	0.00	std::__cxx11::basic_strin
0.00	0.00	0.00	47	0.00	0.00	float __gnu_cxx::__stoa<f
0.00	0.00	0.00	47	0.00	0.00	std::__cxx11::stof(std::_
0.00	0.00	0.00	47	0.00	0.00	__gnu_cxx::__stoa<float,
0.00	0.00	0.00	47	0.00	0.00	__gnu_cxx::__stoa<float,
0.00	0.00	0.00	47	0.00	0.00	__gnu_cxx::__stoa<float,
0.00	0.00	0.00	46	0.00	0.00	NumExp::computeOperation(

Fonte: elaboração própria

Observamos de imediato, na análise da Figura 5.2, que as estruturas de dados relacionadas à operações são as chamadas mais feitas:

- O operador booleano de comparação == é chamado quase 3000 vezes, para aferir durante o loop de percurso da string de entrada se o item atual é um operador, número ou um espaço em branco;
- Em seguida, chamadas às estruturas auxiliares Stack e CircularQueue nos métodos de NumExp são chamadas entre 200 e 400 vezes durante a execução. De fato, devido à arquitetura em camadas implementada, os métodos de baixo nível que mais executam operações matemáticas são os mais chamados, ao passo que os métodos de alto nível de NumSolver são chamados apenas uma vez durante toda a execução;
- Usadas como estruturas auxiliares nos algoritmos, os TADs Pilha e Fila tem seus métodos são chamadas quase 100 vezes cada um no código: empilha, desempilha, enfileira e desenfileira são os mais invocados.

6 CONCLUSÕES

Tendo em vista os objetivos do TP, foi possível desenvolver um sistema que resolve as expressões numéricas e realiza as operações da especificação.

Além disso, foi possível aprender mais algoritmos de manipulação das expressões e aprofundar o conhecimento não somente de estruturas de dados conhecidas (como pilhas e filhas), mas também sobre a criação de novas estruturas e TADs que representam os dados de cada problema. Nesse caso, a criação dos TADs NumSolver e NumExp.

7 REFERÊNCIAS

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Pilhas e Filas. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Acesso em 07 de mai. de 2023.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Análise de Complexidade e Complexidade Assintótica. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Acesso em 07 de mai. de 2023.

8 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Siga o passo a passo abaixo para compilar e executar o TP:

1. Tenha um sistema Linux com compilador g++ e make instalado e configurado;
2. Rode o comando `make clean`
3. Rode o comando `make run`
4. Após o make run, dentro da pasta bin estará o arquivo binário que pode ser executado. Rode o comando `./bin/TP01 -f FILE_PATH`, onde FILE_PATH é o caminho completo do arquivo txt que possui os comandos e expressões de entrada no resolvedor. Exemplo:

```
./bin/TP01 -f /mnt/c/dev/estruturas-de-dados-2023-1/TP01/TP1entrada/entdouble.s36.n50.p.in
```