Universidade Presbiteriana Mackenzie

ESTRUTURA DE DADOS II

Aplicação 1 (Apl1) - Árvore binária de expressão aritmética
Atividade (máx. três alunos)

Objetivo

Implementar uma árvore binária de expressão aritmética em Java e testar a sua implementação, de forma que o programa consiga avaliar expressões aritméticas corretamente.

Instruções

- A atividade deve ser resolvida usando a linguagem Java.
- A árvore usada na sua solução deve ser uma implementação de sua autoria (pode usar como base o conteúdo visto em aula), isto é, não deve usar estruturas relacionadas a árvores que são oferecidas pela linguagem Java (projetos que usarem tais estruturas serão desconsiderados – zero).
- Caso necessário, sua solução pode usar as estruturas de pilha, fila e/ou lista encadeada oferecidas pela linguagem Java.
- Inclua a identificação do grupo (nome completo e RA de cada integrante) no início de cada arquivo de código, como comentário.
- Inclua todas as referências (livros, artigos, sites, vídeos, entre outros) consultadas para solucionar a atividade como comentário no arquivo .java que contém a main().

Problema

Um exemplo de utilização de árvores binárias é a avaliação de expressões. Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos. Nessa árvore, os nós folhas representam operandos e os nós internos, operadores. Uma árvore que representa, por exemplo, a expressão $(3 + 6) * (4 - 1) + 5$ *é ilustrada na Figura 1 a seguir.*

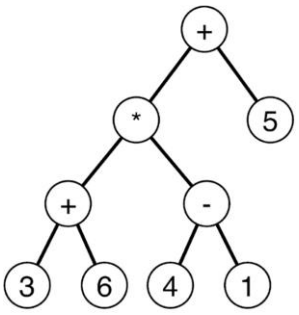

*Figura 1 - Árvore da expressão (3 + 6) * (4 - 1) + 5.*

CELES, W.; CERQUEIRA, R.; RANGEL, J. L. **Introdução a estruturas de dados: com técnicas de programação em C**. Rio de Janeiro: Elsevier, 2016 (adaptado).

ESTRUTURA DE DADOS II

Com base no conteúdo estudado na disciplina e no excerto anterior sobre um problema que pode ser resolvido usando árvore binária, implemente um programa Java que monta uma árvore binária de expressão aritmética a partir de uma expressão informada pelo usuário e que calcula o resultado da expressão.

Funcionamento do programa

O **funcionamento/comportamento do programa** deve atender aos seguintes requisitos:

0. O programa deve apresentar um menu de opções contendo 5 opções:

1. Entrada da expressão aritmética na notação infixa.
2. Criação da árvore binária de expressão aritmética.
3. Exibição da árvore binária de expressão aritmética.
4. Cálculo da expressão (realizando o percurso da árvore).
5. Encerramento do programa.

1. Entrada da expressão aritmética na notação infixa

O usuário do programa informa uma expressão aritmética na notação infixa.

O programa deve validar a expressão informada pelo usuário de acordo com a descrição a seguir:

- A expressão deve conter **apenas números inteiros e/ou decimais**, **parênteses** (prioridade), **e** operadores, sendo que devem ser suportados apenas **quatro operadores binários**: `+` (adição), `-` (subtração), `*` (multiplicação) e `/` (divisão).
- Para qualquer expressão válida, o programa deve exibir uma mensagem em tela informando que a expressão é válida. Isso pode ser feito logo após o usuário informar a expressão ou durante a criação da árvore binária de expressão aritmética.
- Para qualquer expressão que contenha algum erro, o programa deve exibir uma mensagem em tela informando que a expressão é inválida (não é necessário indicar o problema na expressão, embora seja um desafio interessante). Isso pode ser feito logo após o usuário informar a expressão ou durante a criação da árvore binária de expressão aritmética.

Alguns exemplos de expressões válidas:

```
1+2*3
(45+20) * 2 - 15
      0.5*3/0.25
(7 + 3) / (6 – 4) * 9
1.5   + (2    -  (3   + 4)) * 5.1
```

Observe que os espaços em branco inseridos na expressão devem ser ignorados pelo programa, e que é possível informar uma expressão válida em que números e operadores estão juntos (isto é, sem espaços em branco como separador).

Alguns exemplos de expressões inválidas:

```
x * y          (erro: x e y não são números inteiros/decimais nem operadores)
1+2*           (erro: o operador de multiplicação é binário, mas falta um operando na expressão)
```

((10-2)*3 (erro: há dois parênteses de abertura e apenas um de fechamento)
5 % 2 (erro: operador % é inválido)
1 + 2 = 3 (erro: operador = é inválido)

2. Criação da árvore binária de expressão aritmética

O programa deve processar a expressão aritmética na notação infixa informada na opção 1 e montar uma árvore binária de expressão aritmética correspondente.

3. Exibição da árvore binária de expressão aritmética

O programa deve percorrer a árvore criada na opção 2 e exibir o conteúdo de cada nó (o valor armazenado em cada nó) em tela.

Devem ser exibidas três versões da árvore: o conteúdo da árvore em pré-ordem, em ordem e em pós-ordem.

4. Cálculo da expressão (realizando o percurso da árvore)

O programa deve percorrer a árvore e, durante o percurso, realizar o cálculo da expressão. Ao final do percurso, o resultado do cálculo da expressão deve ser exibido em tela.

5. Encerramento do programa

O programa é encerrado quando o usuário escolher a opção 5.

Implementação do programa

A **implementação do programa** deve atender aos seguintes requisitos:

1. Deve existir uma classe base que representa um **nó** da árvore binária. Os atributos e métodos da classe devem ser definidos por você (ou seja: quais propriedades e comportamentos devem existir em um nó?).

No entanto, um método do tipo float deve aparecer obrigatoriamente na sua classe base que representa o nó:

OPERAÇÃO	DESCRIÇÃO
visitar() (ou algum nome similar)	Deve ser executado quando o nó é visitado durante o percurso da árvore para cálculo da expressão. Na classe base, esse método deve retornar Float.NaN.

2. Deve existir uma classe usada para representar um **nó** que armazena um **operando**. Essa classe deve ser uma subclasse da classe base criada no item 1.

O método visitar() (ou equivalente) deve sobrescrever a implementação da classe base da seguinte maneira:

OPERAÇÃO	DESCRIÇÃO

visitar() (ou algum nome similar)	Deve ser executado quando o nó é visitado durante o percurso da árvore para cálculo da expressão. Na classe do nó que armazena um operando, esse método deve retornar o valor do operando.

3. Deve existir uma classe usada para representar um **nó** que armazena um **operador**. Essa classe deve ser uma subclasse da classe base criada no item 1.

O método visitar() (ou equivalente) deve sobrescrever a implementação da classe base da seguinte maneira:

OPERAÇÃO	DESCRIÇÃO
visitar() (ou algum nome similar)	Deve ser executado quando o nó é visitado durante o percurso da árvore para cálculo da expressão. Na classe do nó que armazena um operador, esse método deve retornar o resultado da operação indicada pelo operador.

4. Deve existir uma classe que representa uma **árvore binária**, que será usada para criar uma árvore binária de expressão aritmética na memória. Os atributos e métodos da classe devem ser definidos por você (ou seja: quais propriedades e comportamentos devem existir em uma árvore binária?).

Dica: pelo que já foi descrito no enunciado, a árvore deve ser percorrida de quatro maneiras diferentes: três percursos que exibem o conteúdo de cada nó (pré-ordem, em ordem, pós-ordem) e um percurso que realiza o cálculo da expressão aritmética.

5. Deve existir uma classe com o método public static void main(String[] args) { ... } que apresenta o menu de opções ao usuário e executa as opções selecionadas.

Apresentação do projeto na aula

Além da solução escrita em Java, o grupo deverá apresentar o projeto rodando e explicar (apenas para o professor): o processamento (análise) da expressão aritmética informada pelo usuário; como a árvore binária de expressão aritmética é criada a partir da expressão informada pelo usuário; como a expressão, agora definida como uma árvore binária, é calculada para se obter o resultado da expressão.

As aulas dos dias 23/09/2024 e 30/09/2024 serão reservadas para a apresentação dos trabalhos – a ordem de apresentação será definida por sorteio, no início de cada aula.

Entrega

Código:

Compacte o código-fonte (somente arquivos *.java) no **formato zip**.

Atenção: O arquivo zip não deve conter arquivos intermediários e/ou pastas geradas pelo compilador/IDE (ex. arquivos *.class, etc.).

Prazo de entrega: via link do Moodle até 30/09/2024 23:59.

Critérios de avaliação

A nota da atividade é calculada de acordo com os critérios da tabela a seguir.

ITEM AVALIADO	PONTUAÇÃO MÁXIMA
0. Funcionamento: Menu de opções.	0,2
1. Funcionamento: Leitura e validação da expressão aritmética na notação infixa.	1,0
2. Funcionamento: Criação da árvore binária de expressão aritmética.	2,0
3. Funcionamento: Exibição da árvore binária de expressão aritmética.	1,0
4. Funcionamento: Cálculo da expressão (realizando o percurso da árvore).	1,0
1. Implementação: Classe base do nó.	0,6
2. Implementação: Subclasse nó armazenando operando.	0,4
3. Implementação: Subclasse nó armazenando operador.	0,4
4. Implementação: Classe árvore binária.	0,2
5. Implementação: Classe com main().	0,2
6. Apresentação do projeto	3,0
Bônus opcional: Suporte ao operador unário - (negação).	0,5
Bônus opcional: Subclasses para cada operação matemática.	0,5

Tabela 1 - Critérios de avaliação.

A tabela a seguir contém critérios de avaliação que podem **reduzir** a nota final da atividade.

ITEM INDESEJÁVEL	REDUÇÃO DE NOTA
O projeto é cópia de outro projeto.	Projeto é zerado
O projeto usa estruturas relacionadas a árvores que são oferecidas pela linguagem Java.	Projeto é zerado
Há erros de compilação e/ou o programa trava/"quebra" durante a execução[1].	-1,0
Não há identificação do grupo (código-fonte). Não há indicação de referências (código-fonte). Arquivos enviados em formatos incorretos. Arquivos e/ou pastas intermediárias que são criadas no processo de compilação ou pela IDE foram enviadas junto com o código-fonte.	-1,0

Tabela 2 - Critérios de avaliação (redução de nota).

O código-fonte será compilado com o compilador javac (21.0.2) na plataforma Windows da seguinte forma:

```
> javac *.java -encoding utf8
```

O código compilado será executado com java (21.0.2) na plataforma Windows da seguinte forma:

```
> java <Classe>
```

[1] Sobre erros de compilação: considere apenas erros. Não há problema se o projeto tiver *warnings* (embora *warnings* podem avisar sobre possíveis travamentos em tempo de execução, como loop infinito, divisão por zero, etc.).

Sendo que <Classe> deve ser substituído pelo nome da classe que contém o método public static void main(String[] args).

Dicas

Dica 1. Revisitando um assunto de Estrutura de Dados I, a Figura 2 a seguir descreve um algoritmo para conversão de uma expressão aritmética em notação infixa para a notação posfixa.

Algoritmo para conversão de expressão infixa para posfixa

Um algoritmo para conversão de uma expressão infixa qualquer para posfixa seria:

- Inicie com uma pilha vazia;
- Realize uma varredura na expressão infixa, copiando todos os identificadores encontrados diretamente para a expressão de saída.

  a) Ao encontrar um operador:
  1. Enquanto a pilha não estiver vazia e houver no seu topo um operador com prioridade maior ou igual ao encontrado, desempilhe o operador e copie-o na saída;
  2. Empilhe o operador encontrado;

  b) Ao encontrar um parêntese de abertura, empilhe-o;

  c) Ao encontrar um parêntese de fechamento, remova um símbolo da pilha e copie-o na saída, até que seja desempilhado o parêntese de abertura correspondente.

- Ao final da varredura, esvazie a pilha, movendo os símbolos desempilhados para a saída.

Um exemplo de execução do algoritmo de conversão de infixa para posfixa, supondo a expressão **A*(B+C)/D**, é apresentado abaixo:

Símbolo	Ação	Pilha	Saída
A	copia para a saída	P:[]	A
*	pilha vazia, empilha	P:[*]	A
(	sempre deve ser empilhado	P:[(, *]	A
B	copia para a saída	P:[(,*]	AB
+	prioridade maior, empilha	P:[+, (, *]	AB
C	copia para a saída	P:[+, (, *]	ABC
)	desempilha até achar '('	P:[*]	ABC+
/	prioridade igual, desempilha	P:[/]	ABC+*
D	copia para a saída	P:[/]	ABC+*D
	final, esvazia pilha	P:[]	ABC+*D/

Figura 2 - Conversão de expressão infixa para posfixa.

Analisando este algoritmo, podemos nos perguntar: Seria possível adaptar o algoritmo para me auxiliar na criação da árvore binária de expressão aritmética? Se sim, o que devo alterar no algoritmo?

O algoritmo da Figura 2 usa uma pilha. Será que preciso usar uma pilha para resolver esta atividade? Se sim, qual é o tipo da pilha? Em que momento e por que devo armazenar algo na pilha? De onde vem os elementos

que insiro na pilha? Em que momento e por que devo remover algo da pilha? Para onde vão os elementos removidos da pilha?

Dica 2. Como o programa deve considerar que expressões sem espaços em branco são válidas (isto é, operandos e operadores aparecem "colados um no outro") e que uma expressão pode ter diversos espaços em branco em sequência, talvez não seja possível usar o método split() da classe String do Java.

Uma possibilidade para essa funcionalidade é implementar o seu próprio *tokenizer* (processar uma string e identificar os *tokens* válidos – nesta atividade, os *tokens* válidos são: números inteiros, números decimais, operadores e parênteses).

A Figura 4 na próxima página apresenta uma implementação de um *tokenizer* básico em Java que reconhece números inteiros e os símbolos + e *, além de ignorar espaços em branco.

Algumas observações sobre o código do *tokenizer* de exemplo:

- O código está reduzido por motivos de espaço e formatação deste documento.
- Há um StringBuilder sb para fazer a concatenação de strings (usado no reconhecimento de números inteiros).
- Para reconhecer um número inteiro, o caractere atual da string deve ser um dígito. Quando isso ocorre, o conteúdo do StringBuilder sb é reiniciado com a instrução sb.setLength(0), pois estamos reusando o StringBuilder sb durante o processo de reconhecimento de tokens. Na sequência, o código entra em um loop para concatenar o conteúdo atual de sb com o dígito lido da string, sendo que o loop é encerrado assim que o próximo caractere lido da string *não* for um dígito. Com o loop encerrado, temos um número inteiro (representado como string) em sb, e tal valor é adicionado à lista de tokens via tokens.add(sb.toString()).
- Como os operadores de adição e multiplicação são representados apenas por um único símbolo cada, não há a necessidade de concatenar strings, sendo um código mais direto (adição do símbolo como string na lista de tokens). Porém, é importante lembrar de avançar para o próximo caractere da string.

A execução do código VeryBasicTokenizer gera a saída da Figura 3 (observe que tokenize() é encerrado por não reconhecer a letra A como um token válido).

```
Token não reconhecido: A
Encerrando...

token[0]: 25
token[1]: +
token[2]: 32
token[3]: *
token[4]: 1
token[5]: +
```

Figura 3 - Saída do "Tokenizer básico em Java".

Analise o código da Figura 4 e se pergunte: Como posso reconhecer um número decimal? Como posso reconhecer os outros operadores? E os parênteses? Será que eles são necessários nesse processo? Devo usar outro tipo de dado no lugar da `List<String> tokens`? Posso manter os tokens como `String`s ou devo converter para outro tipo de dado, como `float` e `char`, por exemplo? Em que momento do programa que o método `tokenize()` deve ser executado? Esse método deve operar em conjunto com algum outro algoritmo?

```java
import java.util.ArrayList;
import java.util.List;

public class VeryBasicTokenizer {
  private char[] input;
  private int index;

  public VeryBasicTokenizer(String str) {
    input = str.toCharArray();
    index = 0;
  }

  // Avança para o próximo caractere e retorna seu valor.
  // Ou retorna \0 quando chegou no final da string.
  private char getNextChar() {
    if (index >= input.length) { return '\0'; }
    return input[index++];
  }

  // Separa a string em tokens e retorna uma lista de strings,
  // sendo que cada string é um token reconhecido pelo método.
  public List<String> tokenize() {
    List<String> tokens = new ArrayList<>();
    StringBuilder sb = new StringBuilder();
    char currChar = getNextChar();

    boolean isTokenizing = true;
    while (isTokenizing) {
      // Ignora espaços em branco.
      while (Character.isWhitespace(currChar))
        currChar = getNextChar();

      if (Character.isDigit(currChar)) { // Reconhece um número inteiro.
        sb.setLength(0);
        while (Character.isDigit(currChar)) {
          sb.append(currChar);
          currChar = getNextChar();
        }
        tokens.add(sb.toString());

      } else if (currChar == '+') { // Reconhece símbolo +
        tokens.add("+");
        currChar = getNextChar();

      } else if (currChar == '*') { // Reconhece símbolo *
        tokens.add("*");
        currChar = getNextChar();

      } else if (currChar == '\0') {
        System.out.println("Chegou ao final da string.");
        isTokenizing = false;

      } else {
        System.out.println("Token não reconhecido: " + currChar);
        isTokenizing = false;
      }
    }

    System.out.println("Encerrando...\n");
    return tokens;
  }

  public static void main(String args[]) {
    VeryBasicTokenizer vbt = new VeryBasicTokenizer("25+   32      * 1 + A");

    List<String> tokens = vbt.tokenize();
    for (int i = 0; i < tokens.size(); ++i)
      System.out.println("token[" + i + "]: " + tokens.get(i));
  }
}
```

Figura 4 - Tokenizer básico em Java.