Trabalho Prático III
Interpretador de Escopo Dinâmico
BCC202 – Estruturas de Dados I

João Felipe Paiva Paixão (25.1.4014)
Mateus Oliveira Heleno (25.1.4007)

20 de fevereiro de 2026

1 Introdução

O presente trabalho tem como objetivo o desenvolvimento de um interpretador de comandos simples capaz de gerenciar variáveis em múltiplos escopos aninhados. Para solucionar o problema, foram utilizados conceitos avançados de Estruturas de Dados, especificamente a combinação de uma **Pilha Dinâmica** com **Árvores Binárias de Busca (ABB)**.

O projeto foi inteiramente implementado na linguagem C, utilizando a modularização por meio de Tipos Abstratos de Dados (TADs). A aplicação simula o comportamento de linguagens com escopo dinâmico, onde a busca por uma variável ocorre do escopo mais recente para o mais antigo.

2 Especificações do Problema

O problema consiste em ler um arquivo de texto contendo comandos de declaração e manipulação de variáveis e escopos. O interpretador deve processar quatro comandos básicos:

- **begin:** Abre um novo escopo (empilha um novo contexto).
- **end:** Fecha o escopo atual (desempilha e libera memória).
- **var:** Declara uma variável e seu valor no escopo atual.
- **print:** Busca e imprime o valor de uma variável, procurando primeiro no escopo atual e, caso não encontre, nos escopos anteriores (pilha abaixo).

A principal restrição técnica é a necessidade de implementar a estrutura de armazenamento como uma pilha de árvores, onde cada nó da pilha representa um escopo e contém uma árvore binária responsável por armazenar as variáveis daquele contexto.

3 Considerações Iniciais

Para o desenvolvimento, foram mantidas as ferramentas e boas práticas já utilizadas na disciplina:

- **Linguagem:** C.
- **Ambiente:** Compilação via `gcc` e edição no VS Code.
- **Nomenclatura:** Conforme exigido especificamente no PDF do trabalho, os arquivos foram nomeados como `filaprocessos.h` e `filaprocessos.c`, embora a lógica interna refira-se a Pilhas e Árvores.
- **Validação:** Uso do `Valgrind` para garantir ausência de vazamentos de memória (memory leaks), crucial devido à alta alocação dinâmica de nós de árvore e pilha.

4 Desenvolvimento

A solução foi estruturada de forma modular. A arquitetura de dados pode ser visualizada como uma "Pilha de Árvores".

4.1 Estruturas de Dados (TADs)

As estruturas principais definidas em `filaprocessos.h` são:

`No` **(Árvore):** Representa uma variável. Contém a chave (nome), o valor e ponteiros para esquerda e direita. Implementa uma Árvore Binária de Busca para garantir eficiência $O(\log n)$ na busca de variáveis dentro de um mesmo escopo.

`Escopo` **(Nó da Pilha):** Representa um bloco de código (`begin/end`). Contém o ponteiro para a raiz da árvore de variáveis deste escopo e um ponteiro para o escopo anterior.

`Pilha`**:** Gerencia a execução, mantendo um ponteiro para o `topo` (escopo atual).

4.2 Lógica de Execução e Busca

A função `executar` coordena a leitura do arquivo. A lógica de busca de variáveis (`buscarVar`) implementa o conceito de **Escopo Dinâmico**:

1. Verifica a árvore do escopo no topo da pilha.
2. Se a variável não for encontrada, desce para o escopo anterior (`prox`).
3. Repete o processo até encontrar a variável ou a pilha terminar (caso em que um erro é reportado).

Essa abordagem permite que variáveis locais "sombreiem"variáveis globais com o mesmo nome, um comportamento típico de linguagens de programação.

4.3 Detalhamento das Funções Principais

Nesta seção, descrevemos as três funções que compõem o núcleo lógico do interpretador, detalhando sua responsabilidade e o comportamento algorítmico adotado para o gerenciamento da Pilha de Árvores.

`executar(Pilha *p, int argc, char **argv)`

Atua como o motor do interpretador, sendo responsável por orquestrar a leitura do arquivo de entrada e a execução dos comandos.

- **Funcionamento:** A função opera em um laço de repetição, lendo o arquivo de texto palavra por palavra. Com base no comando identificado (`begin`, `var`, `print`, `end`), ela interage com a pilha, abrindo novos escopos (empilhando árvores), fechando contextos (desempilhando e liberando memória) ou chamando as funções de inserção e busca de variáveis.
- **Gestão de Segurança:** Contém validações rigorosas para evitar falhas de segmentação e vazamento de memória. Caso ocorra uma operação inválida (como acessar uma variável não declarada ou fechar um escopo inexistente), a função garante o fechamento seguro do arquivo e a deleção de toda a pilha antes de encerrar o programa.

`buscarVar(Pilha *p, char *varBusca)`

Implementa a essência do conceito de **Escopo Dinâmico** exigido na especificação do trabalho.

- **Funcionamento:** A busca inicia-se no escopo atual (topo da pilha). Caso a variável procurada não seja encontrada na Árvore Binária daquele escopo, a função desce sequencialmente para os escopos anteriores, iterando pelos ponteiros da pilha até a base.
- **Importância:** Esse comportamento de varredura profunda (de cima para baixo na pilha) garante que variáveis declaradas em escopos mais internos "sombreiem"corretamente variáveis com o mesmo nome definidas em escopos mais externos, retornando sempre o valor do contexto mais recente em execução.

`adicionaNo(No **raiz, char *var, char *valor)`

Gerencia a inserção e atualização de variáveis dentro da Árvore Binária de Busca (ABB) de um escopo específico.

- **Funcionamento:** Utiliza recursividade para percorrer a árvore com base na comparação alfabética (via `strcmp`) do nome da variável. Se atingir um ponteiro nulo, um novo nó é criado. Se encontrar uma chave idêntica já existente no mesmo escopo, a função atualiza o valor da variável.
- **Eficiência:** A escolha da Árvore Binária de Busca para armazenar as variáveis de cada escopo permite que as operações de declaração (`var`) e consulta (`print`) tenham complexidade média de $O(\log n)$, oferecendo um desempenho superior em comparação com o uso de listas encadeadas simples para buscas textuais.

4.4 Código Fonte: filaprocessos.h

A interface define as structs e protótipos para manipulação da Pilha e da Árvore.

Listing 1: Interface do TAD – filaprocessos.h

```
// Joao Felipe Paiva Paixao
// 25.1.4014
// Mateus Oliveira Heleno
// 25.1.4007

#ifndef FILAPROCESSOS_H
#define FILAPROCESSOS_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define TAM_PALA 16

// structs

typedef struct No // No da arvore
{
    char var[TAM_PALA];   // nome da variavel
    char valor[TAM_PALA]; // valor de variavel
    struct No *esq;
    struct No *dir;
} No;

typedef struct Escopo // No da pilha
{
    No *raiz;             // raiz da arvore
    struct Escopo *prox; // ponteiro pro escopo de baixo
    int tamAr;           // tamanho da arvore
} Escopo;

typedef struct // struct da pilha
{
    Escopo *topo; // ponteiro pro topo da pilha
    int tam;      // quantidade de escopos da pilha
} Pilha;

// TAD PILHA
Pilha *criarPilha();
Pilha *destroiPilha(Pilha *p);
void adicionaPilha(Pilha *p, Escopo *e);
void desempilha(Pilha *p);
void executar(Pilha *p, int argc, char **argv);

// TAD ESCOPO
Escopo *criarEscopo();
```

```c
Escopo *destroiEscopo(Escopo *e);
void adicionaVar(Pilha *p, char *var, char *valor);
char *buscarVar(Pilha *p, char *varBusca);
void imprimirVar(Pilha *p, char *varBusca);
No *buscarAr(No *raiz, char *varBusca);

// TAD NO
bool adicionaNo(No **raiz, char *var, char *valor);
No *criaNo(char *var, char *valor);
void destroiAr(No *no);

// aux
FILE *abreArquivo(int argc, char *argv[]);
int analiseArq(char *palavra);

#endif
```

4.5 Código Fonte: filaprocessos.c

A implementação contém a lógica de inserção na BST, manipulação da pilha e o loop principal de interpretação. Destaca-se o cuidado com a liberação de memória recursiva nas funções `destroiAr` e `destroiEscopo`.

Listing 2: Implementação das Funções – filaprocessos.c

```c
// Joao Felipe Paiva Paixao
// 25.1.4014
// Mateus Oliveira Heleno
// 25.1.4007

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#include "filaprocessos.h"

Pilha *criarPilha()
{
    Pilha *p = (Pilha *)malloc(sizeof(Pilha));

    if (!p)
        return NULL;

    p->topo = NULL;
    p->tam = 0;

    return p;
}

Pilha *destroiPilha(Pilha *p)
{
```

```
28      if (!p)
29          return NULL;
30
31      while (p->topo != NULL) // esvazia a pilha antes de liberar a
            estrutura
32      {
33          desempilha(p);
34      }
35
36      free(p);
37      return NULL;
38  }
39
40  void adicionaPilha(Pilha *p, Escopo *e)
41  {
42      if (!p || !e)
43          return;
44
45      e->prox = p->topo; // o prox do novo e o antigo topo
46      p->topo = e;       // atualiza topo
47      p->tam++;
48  }
49
50  void desempilha(Pilha *p)
51  {
52      if (!p || p->tam == 0)
53          return;
54
55      Escopo *aux = p->topo;
56      p->topo = p->topo->prox; // topo desce pro escopo de baixo
57
58      p->tam--;
59      destroiEscopo(aux); // libera o no da pilha e da arvore
60  }
61
62  void executar(Pilha *p, int argc, char **argv)
63  {
64      FILE *arq = abreArquivo(argc, argv);
65      if (!arq)
66          return;
67
68      char palavra[TAM_PALA];
69
70      while (fscanf(arq, "%s", palavra) != EOF) // leitura linha a
           linha
71      {
72
73          int comando = analiseArq(palavra);
74
75          switch (comando)
76          {
```

```
77          case 1: // begin
78          {
79              Escopo *e = criarEscopo();
80              adicionaPilha(p, e);
81              break;
82          }
83
84          case 2: // var
85          {
86              char valor[TAM_PALA], var[TAM_PALA], lixo[TAM_PALA];
87
88              if (p->topo == NULL) // verifica se tem espaco
89              {
90                  printf("Escopo nao aberto\n");
91                  destroiPilha(p);
92                  fclose(arq);
93                  exit(1);
94              }
95
96              fscanf(arq, "%s %s %s", var, lixo, valor); // "var x
                    = 10"
97              adicionaVar(p, var, valor);
98              break;
99          }
100
101         case 3: // print
102         {
103             char varBusca[TAM_PALA];
104             if (p->topo == 0)
105             {
106                 printf("Escopo nao aberto\n");
107                 destroiPilha(p);
108                 fclose(arq);
109                 exit(1);
110             }
111
112             fscanf(arq, "%s", varBusca);
113             char *resultado = buscarVar(p, varBusca);
114
115             if (resultado == NULL)
116             {
117                 printf("Variavel %s nao declarada\n", varBusca);
118                 destroiPilha(p);
119                 fclose(arq); // evita o vazamento de memoria
120                 exit(1);
121             }
122             else
123             {
124                 printf("%s\n", resultado);
125             }
126             break;
```

```
127         }
128         case 4: // end
129         {
130             if (p->topo == NULL)
131             {
132                 printf("Escopo nao aberto\n");
133                 destroiPilha(p);
134                 fclose(arq);
135                 exit(1);
136             }
137             desempilha(p);
138             break;
139         }
140         }
141     }
142     if (p->tam > 0) // se sobrar escopo na pilha
143     {
144         printf("Escopo n o fechado\n");
145         fclose(arq);
146         destroiPilha(p);
147         exit(1);
148     }
149 
150     fclose(arq);
151 }
152 
153 Escopo *criarEscopo()
154 {
155     Escopo *e = (Escopo *)malloc(sizeof(Escopo));
156     if (!e)
157         return NULL;
158 
159     e->raiz = NULL;
160     e->prox = NULL;
161     e->tamAr = 0;
162 
163     return e;
164 }
165 
166 Escopo *destroiEscopo(Escopo *e)
167 {
168     if (!e)
169         return NULL;
170 
171     destroiAr(e->raiz); // destroi recursivamente a arvore do
        escopo
172 
173     e->raiz = NULL;
174     e->tamAr = 0;
175 
176     free(e);
```

```
177     return NULL;
178 }
179
180 void adicionaVar(Pilha *p, char *var, char *valor) // adiciona
    variavel na arvore do escopo atual
181 {
182
183     if (p == NULL)
184         return;
185
186     if (adicionaNo(&(p->topo->raiz), var, valor)) // se retorna
        true eh uma chave nova se retorna false eh uma chave ja
        existente
187         p->topo->tamAr++;
188 }
189
190 char *buscarVar(Pilha *p, char *varBusca) // busca a chave nas
    pilhas
191 {
192     Escopo *atual = p->topo;
193
194     while (atual != NULL) // passa pelos escopos do topo ate a
        base
195     {
196         No *noEncontrado = buscarAr(atual->raiz, varBusca);
197         if (noEncontrado != NULL)
198         {
199             return noEncontrado->valor; // escopo mais recente
                encontrado
200         }
201         atual = atual->prox; // desce pro escopo anterior
202     }
203     return NULL; // retorna NULL sinalizando erro, tratamento na
        funcao executar
204 }
205
206 void imprimirVar(Pilha *p, char *varBusca) // funcao de impressao
207 {
208     char resultado[TAM_PALA];
209     strcpy(resultado, buscarVar(p, varBusca));
210     printf("%s\n", resultado);
211 }
212
213 No *buscarAr(No *raiz, char *varBusca) // busca recursiva na
    arvore binaria
214 {
215     if (raiz == NULL)
216         return NULL;
217
218     int comp = strcmp(varBusca, raiz->var);
219
```

```
220     if (comp < 0)
221         return buscarAr(raiz->esq, varBusca); // procura na
               esquerda
222     else if (comp > 0)
223         return buscarAr(raiz->dir, varBusca); // procura na
               direira
224     else
225         return raiz; // encontrou
226 }
227
228 bool adicionaNo(No **raiz, char *var, char *valor) // insere na
       arvore, retorna true se criar e false se nao
229 {
230     if (*raiz == NULL)
231     {
232         *raiz = criaNo(var, valor);
233         return true;
234     }
235
236     int comp = strcmp(var, (*raiz)->var);
237
238     if (comp < 0)
239         return adicionaNo((&(*raiz)->esq), var, valor);
240     else if (comp > 0)
241         return adicionaNo((&(*raiz)->dir), var, valor);
242     else
243     {
244         strcpy((*raiz)->valor, valor);
245         return false;
246     }
247
248     return true;
249 }
250
251 No *criaNo(char *var, char *valor)
252 {
253     No *novo = (No *)malloc(sizeof(No));
254     if (!novo)
255         return NULL;
256
257     strcpy(novo->var, var);
258     strcpy(novo->valor, valor);
259     novo->esq = novo->dir = NULL;
260     return novo;
261 }
262
263 void destroiAr(No *no)
264 {
265     if (!no)
266         return;
267
```

```
268     destroiAr(no->esq);
269     destroiAr(no->dir);
270 
271     free(no);
272 }
273 
274 FILE *abreArquivo(int argc, char *argv[])
275 {
276     if (argc < 2)
277     {
278         printf("Uso: %s nome_do_arquivo\n", argv[0]);
279         return NULL;
280     }
281 
282     FILE *arq = fopen(argv[1], "r");
283     if (arq == NULL)
284     {
285         printf("Erro ao abrir o arquivo.\n");
286         return NULL;
287     }
288 
289     return arq;
290 }
291 
292 int analiseArq(char *palavra)
293 {
294     if (!strcmp(palavra, "begin"))
295         return 1;
296     if (!strcmp(palavra, "var"))
297         return 2;
298     if (!strcmp(palavra, "print"))
299         return 3;
300     if (!strcmp(palavra, "end"))
301         return 4;
302     return 5; // erro
303 }
```

4.6 Código Fonte: tp.c

O arquivo principal inicializa a estrutura e dispara a execução.

Listing 3: Programa Principal – tp.c

```
1 // Joao Felipe Paiva Paixao
2 // 25.1.4014
3 // Mateus Oliveira Heleno
4 // 25.1.4007
5 
6 #include <stdio.h>
7 #include <string.h>
8 #include <stdlib.h>
9 #include <stdbool.h>
```

```
10
11 #include "filaprocessos.h"
12
13 int main(int argc, char **argv)
14 {
15
16     Pilha *p = criarPilha(); // cria a pilha de escopos
17
18     executar(p, argc, argv); // executa o interpretador passando
       os nomes de arquivos
19
20     destroiPilha(p); // limpa a memoria alocada
21
22     return 0;
23 }
```

5 Resultados

Os testes realizados validaram o comportamento correto de empilhamento e desempilhamento de escopos, bem como a busca em profundidade na pilha.

A tabela apresenta um caso de teste complexo onde uma variável `x` é redeclarada em um escopo interno, demonstrando o sombreamento correto, e acessada após o fechamento do escopo, retornando ao valor original.

Tabela 1: Exemplo de Execução (Escopos Aninhados)

Entrada (Arquivo)	**Saída Gerada**

`begin`
`var x = 10`
`begin`
`var x = 20`
`print x`
`end`
`print x`
`end`
	20
	
	10

5.1 Análise de Memória

A execução foi monitorada via `Valgrind`. O relatório confirmou que todos os `mallocs` realizados para criar nós da árvore e elementos da pilha foram devidamente liberados pelos destrutores (`destroiPilha`, `destroiEscopo`, `destroiAr`), resultando em zero vazamentos.

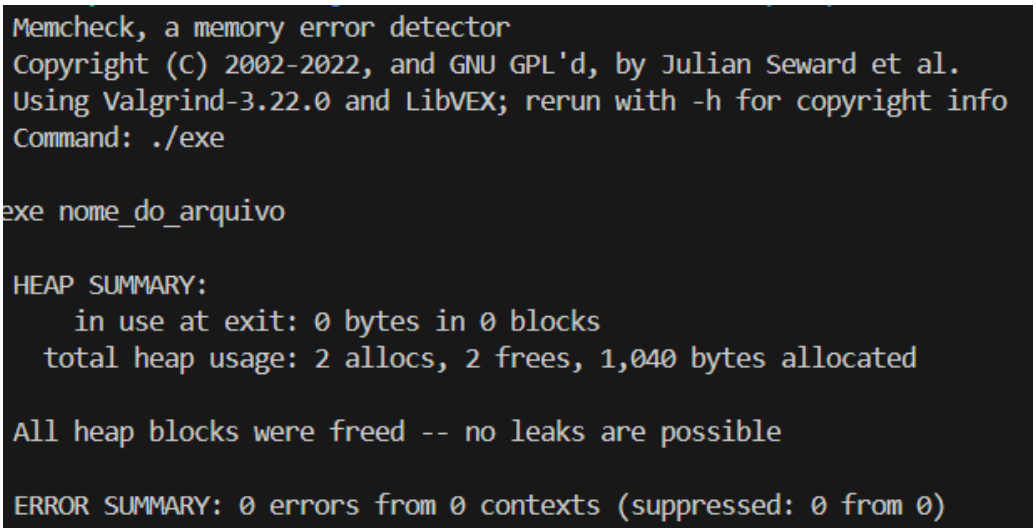

Figura 1: Comprovação de ausência de vazamentos de memória (Memory Leaks).

6 Conclusão

O trabalho permitiu consolidar o conhecimento sobre estruturas de dados hierárquicas e lineares trabalhando em conjunto. A implementação de uma "Pilha de Árvores"mostrou-se uma solução elegante e eficiente para o problema de gerenciamento de escopos dinâmicos.

A maior dificuldade encontrada foi garantir o gerenciamento correto dos ponteiros ao desalocar um escopo intermediário e assegurar que a busca percorresse a pilha corretamente sem falhas de segmentação. O resultado final atende a todos os requisitos propostos, com código modular, documentado e livre de erros de memória.