# Relatório do Trabalho Prático III (TP III)

Árvore Binária de Busca e Pilha

**Nome dos alunos:**
Maurício de Oliveira Santos Rodrigues
João Pedro Seabra Nogueira

**Professora:** Karla A S Joriatti

26 de fevereiro de 2026

## Resumo

Este relatório descreve a implementação de um gerenciador de escopos e variáveis utilizando uma Pilha de Árvores Binárias de Busca (BST). O objetivo é simular o comportamento de linguagens de programação no tratamento de escopos dinâmicos, permitindo a declaração, atribuição e recuperação de variáveis em diferentes níveis de profundidade, garantindo a integridade dos dados e o fechamento correto de blocos.

## Sumário

# 1 Implementação

## 1.1 Visão geral

O programa processa um fluxo de comandos que simulam o controle de memória de um interpretador. Através dos comandos `begin` e `end`, o sistema empilha e desempilha contextos. Dentro de cada contexto, o comando `var` armazena dados em uma árvore binária, e o comando `print` realiza a busca escalonada: do escopo mais interno para o mais externo.

## 1.2 Arquitetura e TADs

A solução foi modularizada para garantir a manutenibilidade e seguir as diretrizes do enunciado:

- **tp.c**: Contém a função principal e o loop de execução.
- **filaprocessos.h**: Define as assinaturas das funções e as estruturas de dados.
- **filaprocessos.c**: Implementa a lógica das estruturas de dados.

As principais estruturas são:

- `Variavel`: Contém os campos `nome` e `conteudo`.
- `No`: Estrutura de nó para a Árvore Binária de Busca (BST).
- `Pilha`: Lista encadeada onde cada célula aponta para a raiz de uma BST.

## 1.3 Funcionamento das principais funções

Abaixo, detalham-se as funções que compõem a lógica modular do sistema:

- **executar**: É o núcleo do programa. Utiliza um laço de repetição associado ao `scanf` para processar a entrada padrão. Ela gerencia o estado da `Pilha` e redireciona os comandos para as funções específicas, além de realizar a verificação de erros fatais (como o fechamento de um escopo inexistente).

```c
//executa a sequencia de comandos
void executar(Pilha *pilha, Erro *erro){
char comando[15];

//recebe os comandos ate o scanf chegar ao final do arquivo de
  entrada (EOF)
while(scanf("%s", comando) != EOF) {
  //o comando begin cria um novo escopo no topo da pilha
  if(strcmp(comando, "begin") == 0)
    adicionaPilha(pilha);

  //o comando end desemplilha o escopo do topo da pilha
  else if(strcmp(comando, "end") == 0){
    Arvore *raiz = NULL;

    //se falhar libera a pilha e retorna mensagem de erro
    if (!desempilha(pilha, &raiz)){
      erro->tipo = 1;
      strcpy(erro->mensagemErro, "Escopo nao aberto");
      destroiPilha(pilha);
      return;
    }

    //libera o escopo desempilhado
    raiz = destroiEscopo(raiz);
```

```
25      }
26
27      //o comando print procura a variavel ligada a chave passada e
        imprime seu valor
28      else if(strcmp(comando, "print") == 0){
29        Variavel var;
30        char nome[MAX_NOME];
31
32        //recebe o nome da variavel que sera printada
33        scanf("%s", nome);
34
35        //procura pela variavel no escopo atual e nos anteriores
36        if(buscaEmEscopos(pilha, nome, &var))
37          printf("%s\n", var.conteudo); //caso encontre imprime seu
        valor
38
39        else{
40          //erro de variavel nao encontrada
41          erro->tipo = 2;
42
43          //funcao para adicionar o nome da variavel na mensagem de erro
44          sprintf(erro->mensagemErro, "Variavel %s nao declarada", nome)
        ;
45
46          //detroi a pilha
47          destroiPilha(pilha);
48
49          return;
50        }
51      }
52
53      //o comando var inicializa uma variavel ou atribui um novo valor
        caso ja tenha sido declarada no escopo
54      else if(strcmp(comando, "var") == 0){
55        char nome[MAX_NOME];
56        char conteudo[MAX_CONTEUDO];
57        char lixo[5]; //usado como filtro no scanf
58
59        //recebe o nome e valor da variavel (mascara: "<nome> = <valor
        >")
60        scanf("%s %s %s", nome, lixo, conteudo);
61
62        Variavel novaVar;
63        strcpy(novaVar.nome, nome);
64        strcpy(novaVar.conteudo, conteudo);
65
66        Arvore *temp;
67        temp = getTopo(pilha); //pega o escopo no topo da pilha (sem
        liberar a celula)
68
69        if(temp){ //caso o escopo tenha sido inicializado
70          adicionaVar(temp, &novaVar); //adiciona a variavel
71        }
72        else{
73          erro->tipo = 1; //erro de escopo nao aberto
74          strcpy(erro->mensagemErro, "Escopo nao aberto");
75          destroiPilha(pilha); //libera a pilha
76          return;
77        }
78
79      }
```

```
80  }
81
82  if(getTopo(pilha)){ //caso ainda tenha um escopo aberto no topo da
      pilha
83    erro->tipo = 3; //erro de escopo nao fechado
84    strcpy(erro->mensagemErro, "Escopo n o fechado");
85    destroiPilha(pilha);
86  }
87
88  }
89
```

- **adicionaVar**: Implementada de forma recursiva, esta função navega pela BST do escopo atual comparando os nomes das variáveis via `strcmp`. Se a variável já existe, seu valor é atualizado; caso contrário, um novo `No` é alocado e inserido na folha apropriada, mantendo a propriedade de ordenação da árvore.

```
1   //adiciona uma variavel ao escopo atual
2   bool adicionaVar(Arvore *raiz, Variavel *var){
3   //caso a arvore nao existe retorna false
4   if(!raiz)
5     return false;
6
7   //caso a arvore ou sub-arvore esteja vazia, inicializa o No raiz
8   if(!(*raiz)){
9     No *novo = (No*) malloc(sizeof(No));
10    if(!novo)
11      return false;
12
13    novo->valor = *var;
14    novo->pDir = NULL;
15    novo->pEsq = NULL;
16
17    *raiz = novo;
18    return true;
19  }
20
21  //determina se a incersao deve ocorrer na sub-arvore da esquerda ou
      da direita
22  int result = strcmp((*raiz)->valor.nome, var->nome);
23
24  if(result > 0)
25    return adicionaVar(&(*raiz)->pEsq, var);
26
27  if(result < 0)
28    return adicionaVar(&(*raiz)->pDir, var);
29
30  //sobrescreve o valor da variavel caso ela ja exista
31  (*raiz)->valor = *var;
32
33  return true;
34  }
35
36  //busca o valor de uma variavel na pilha
37  bool arvoreBusca(Arvore *raiz, char *chave, Variavel *var){
38  //verifica se a arvore nao existe
39  if(!raiz)
40    return false;
41
42  //verifica se o no raiz eh invalido
43  if(!(*raiz))
```

```
    return false;

  //recebe o resultado da comparacao
  int compara = strcmp((*raiz)->valor.nome, chave);

  //se nome eh maior que a chave, segue para a esquerda
  if(compara > 0)
    return arvoreBusca(&(*raiz)->pEsq, chave, var);

  //se nome eh menor que a chave, segue para a direita
  if(compara < 0)
    return arvoreBusca(&(*raiz)->pDir, chave, var);

  //atualiza o valor da variavel ao encontrar na arvore
  *var = (*raiz)->valor;

  return true;
  }

```

- **buscaEmEscopos**: Esta função implementa a regra de escopo dinâmico. Ela percorre a `Pilha` do topo (escopo mais interno) para a base (escopo global). Para cada nível da pilha, ela invoca a `arvoreBusca`. Se a variável for encontrada, a busca é interrompida e o valor é retornado, garantindo que variáveis locais 'ocultem' variáveis globais homônimas.

```
//busca o valor em todos os escopos da pilha
bool buscaEmEscopos(Pilha *pilha, char *chave, Variavel *var){
//verifica se a pilha eh invalida
if(!pilha)
  return false;

//verifica se o topo eh invalido
if(!pilha->topo)
  return false;


//realiza a busca binaria em cada escopo anterior ao atual
Celula *aux = pilha->topo;
while(aux){
  if(arvoreBusca(aux->raiz, chave, var))
    return true;

  aux = aux->prox;
}

return false;
}

```

- **arvoreBusca**: Uma função de busca binária clássica que opera sobre a estrutura de dados `Arvore`. Sua complexidade média é logarítmica, o que torna a recuperação de variáveis eficiente mesmo em escopos com grande volume de dados.

```
//busca o valor em todos os escopos da pilha
bool buscaEmEscopos(Pilha *pilha, char *chave, Variavel *var){
//verifica se a pilha eh invalida
if(!pilha)
  return false;

//verifica se o topo eh invalido
if(!pilha->topo)
```

```
    return false;


  //realiza a busca binaria em cada escopo anterior ao atual
  Celula *aux = pilha->topo;
  while(aux){
    if(arvoreBusca(aux->raiz, chave, var))
      return true;

    aux = aux->prox;
  }

  return false;
  }

```

- **adicionaPilha e desempilha**: Responsáveis pelo gerenciamento de memória dos contextos. `adicionaPilha` aloca uma nova célula e inicializa uma árvore vazia para o novo escopo. `desempilha` remove o topo da pilha e retorna o ponteiro da árvore para que ela possa ser devidamente destruída.

```
  //adiciona um novo escopo na pilha
  bool adicionaPilha(Pilha *pilha){
  //verifica se a pilha eh invalida
  if(!pilha)
    return false;

  //cria a nova celula
  Celula *nova = (Celula*) malloc(sizeof(Celula));

  //verifica se a alocacao falhou
  if(!nova)
    return false;

  //adiciona um escopo vazio no topo da pilha
  Arvore *raiz;
  raiz = criarEscopo();
  nova->raiz = raiz;
  nova->prox = pilha->topo;
  pilha->topo = nova;

  return true;
  }

  //libera a celula topo e devolve seu escopo
  bool desempilha(Pilha *pilha, Arvore **raiz){
  //verifica se a pilha eh invalida
  if(!pilha)
    return false;

  //se o topo for invalido, retorna
  Celula *aux = pilha->topo;
  if(!aux)
    return false;

  //salva as informacoes na variavel
  *raiz = aux->raiz;
  pilha->topo = aux->prox;

  //libera o escopo
  free(aux);
```

```
41
42  return true;
43  }
44
```

- **destroiEscopoAux**: Função auxiliar recursiva que executa um caminhamento **pós-ordem**. Ela garante que os filhos (esquerdo e direito) de um nó sejam desalocados antes do próprio nó pai, evitando referências perdidas e garantindo que toda a memória dinâmica da árvore seja limpa ao fechar um bloco (`end`).

```
1   //realiza o caminhamento pos ordem para destruir todos os filhos
2   void destroiEscopoAux(Arvore *raiz){
3   //se a raiz eh nula, chegou a uma posicao invalida da arvore
4   if(!raiz)
5     return;
6
7   //se o conteudo da raiz eh nulo, retorna
8   if(*raiz == NULL)
9     return;
10
11  //destroi os filhos (esquerda e direita)
12  destroiEscopoAux(&(*raiz)->pEsq);
13  destroiEscopoAux(&(*raiz)->pDir);
14
15  //com todos os filhos destruidos, libera a memoria da arvore de fato
16  free(*raiz);
17  }
18
```

# 2 Impressões gerais

A implementação deste trabalho permitiu uma compreensão prática sobre a simulação de escopos dinâmicos, um conceito fundamental na construção de compiladores e interpretadores. O maior desafio técnico residiu na natureza híbrida da arquitetura: enquanto a **Pilha** gerencia a cronologia e a hierarquia dos blocos (`begin/end`), a **Árvore Binária de Busca (BST)** otimiza a recuperação de dados dentro de cada contexto.

Durante o desenvolvimento, a manipulação de ponteiros duplos para a raiz das árvores exigiu atenção redobrada, especialmente para garantir que as inserções no escopo atual não afetassem os escopos pais de forma indevida. Além disso, a implementação da busca recursiva escalonada — que atravessa a pilha da célula de topo até a base — revelou-se um exercício crítico de lógica para garantir que a precedência das variáveis locais sobre as globais fosse respeitada.

Por fim, a etapa de depuração com a ferramenta `Valgrind` foi indispensável. A destruição de um escopo não é uma operação simples de `free`, mas sim um percurso pós-ordem completo na BST. Garantir que cada nó de variável fosse devidamente liberado antes da célula da pilha evitou vazamentos de memória (*memory leaks*), assegurando a estabilidade do simulador mesmo em cenários de alta profundidade de recursão ou múltiplos blocos aninhados.

# 3 Análise de Complexidade

## 3.1 Tempo

- **Inserção/Busca na Árvore**: $O(\log n)$ no caso médio e $O(n)$ no pior caso (árvore degenerada).
- **Busca em Escopos**: $O(n)$, pois no pior caso deve-se percorrer todos os escopos e todos os nós de cada árvore.
- **Finalização**: $O(n)$ para desalocar todos os elementos do sistema.

### 3.2 Espaço

A complexidade de espaço é $O(n)$, onde $n$ representa o total de variáveis e escopos alocados simultaneamente na memória.

## 4 Análise de Resultados

Para verificar se a implementação da gerência de escopo e das instruções de atribuição e impressão estava funcionando corretamente, foi utilizado o arquivo de teste `teste1.in`. Esse teste foi escolhido porque avalia principalmente a capacidade do interpretador de lidar com escopos aninhados e com a sopreposição do valor de variáveis dentro de blocos internos.

### 4.1 Teste 1: Sem erros

#### 4.1.1 Descrição do Teste

O arquivo de entrada define dois níveis de escopo:

- **Escopo Global (Externo):** Declara a variável `x` com o valor `oi`.
- **Escopo Local (Interno):** Declara as variáveis `y` e `z`, além de redefinir `y` e `x` dentro deste bloco.

A principal finalidade desse teste é garantir que, ao sair do escopo interno, o valor original de `x` (definido no escopo externo) continue preservado e que as variáveis locais sejam corretamente descartadas.

#### 4.1.2 Comparação de Resultados

A tabela abaixo apresenta a comparação entre a saída esperada e a saída obtida durante a execução do programa:

| Instrução | Saída Esperada | Saída Obtida |
|---|---|---|
| `print y` (interno) | 3 | 3 |
| `print x` (interno) | 4 | 4 |
| `print x` (externo) | oi | oi |

Tabela 1: Comparação entre resultados esperados e obtidos.

Observa-se que todas as saídas coincidem exatamente com o resultado esperado, o que indica que o comportamento do interpretador está de acordo com o especificado.


Figura 1: Print do Terminal - Teste 1

### 4.2 Teste 2: Erro de Variável Não Declarada

Além do teste anterior, foi realizado um novo experimento com o arquivo `teste4.in`, com o objetivo de verificar se o interpretador identifica corretamente o uso de variáveis que não foram declaradas no escopo atual ou em escopos superiores.

### 4.2.1 Descrição do Teste

O arquivo `teste4.in` possui a seguinte estrutura:

- No escopo externo, são declaradas as variáveis `x = 2` e `z = 7.5`.
- Em seguida, é aberto um escopo interno, no qual:
  - `print x` é executado (variável válida, herdada do escopo externo).
  - `print y` é executado (variável `y` não foi declarada).
- Após o bloco interno, há ainda a instrução `print z`.

O comportamento esperado é que o programa imprima corretamente o valor de `x` e, ao tentar acessar `y`, gere uma mensagem de erro indicando que a variável não foi declarada, interrompendo a execução normal.

**Saída Esperada (`teste4.out`)**

```
2
Variavel y nao declarada
```

**Saída Obtida**

Conforme observado na execução no terminal, a saída produzida foi:

```
2
Variavel y nao declarada
```

A saída obtida corresponde exatamente à saída esperada.

### 4.2.2 Conclusão

Esse teste demonstra que o mecanismo de busca na **Pilha** não apenas respeita a hierarquia de escopos, mas também trata corretamente situações de erro. Ao não encontrar a variável `y` em nenhum dos níveis da pilha, o interpretador emite a mensagem adequada e evita comportamento indefinido.

Com isso, é possível concluir que a verificação de declaração prévia de variáveis está implementada corretamente, garantindo que apenas identificadores previamente definidos possam ser utilizados. Dessa forma, o sistema impede o acesso a variáveis inexistentes e evita comportamentos inesperados durante a execução. Além disso, o tratamento de erro está de acordo com o especificado nos arquivos de saída esperados, demonstrando consistência e confiabilidade na implementação.

Portanto, além de funcionar corretamente em cenários válidos (como no `teste1.in`), o interpretador também se comporta adequadamente diante de situações inválidas, aumentando a robustez da implementação.

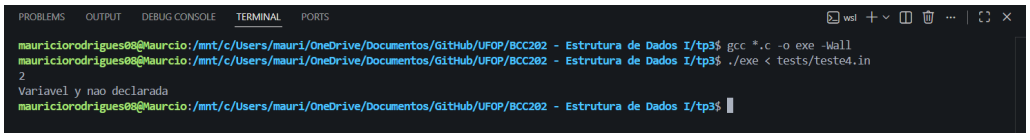


Figura 2: Print do Terminal - Teste 2

### 4.3 Conclusão da Análise

Conforme evidenciado nas imagens do terminal anexada, o programa foi compilado com a flag `-Wall` sem a geração de avisos (*warnings*) e produziu uma saída idêntica à definida nos arquivos `.out`.

Esses resultados permitem concluir que:

1. O mecanismo de **Pilha** está operando corretamente, criando e removendo escopos conforme a entrada e saída dos blocos.
2. A **Sobreposição do Valor de Variáveis** funciona como esperado, permitindo que `x = 4` no escopo interno não altere permanentemente o valor `x = oi` do escopo externo, no caso do Teste 1.
3. A **Busca e Recuperação de Valores** respeitam adequadamente a hierarquia dos blocos `begin...end`, garantindo consistência na execução.
4. A **Verificação de Declaração Prévia** de variáveis está funcionando corretamente e emitindo mensagem de erro, quando necessário.

De modo geral, o teste demonstra que a implementação atende aos requisitos propostos para controle de escopo e manipulação de variáveis.

## 5 Conclusão

O trabalho foi concluído atendendo aos requisitos de manipulação de TADs complexos. A utilização da BST otimizou a busca em relação a uma lista simples, e a lógica de pilha garantiu a simulação correta do tempo de vida das variáveis. O uso do Valgrind confirmou a integridade da gestão de memória.

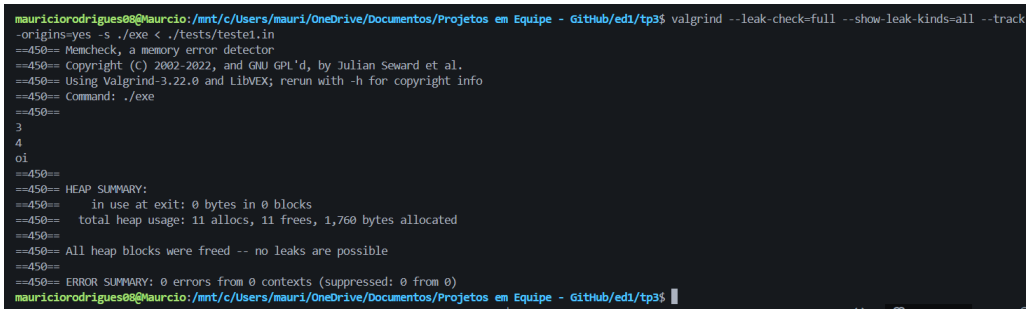


Figura 3: Print de Execução - Valgrind