Grupo 4 - Trabalho Estrutura de Dados 1 2025/1

1.0

Gerado por Doxygen 1.14.0

Capítulo 1

Índice das Estruturas de Dados

1.1 Estruturas de Dados

Aqui estão as estruturas de dados, uniões e suas respectivas descrições:

Capítulo 2

Índice dos Arquivos

2.1 Lista de Arquivos

Esta é a lista de todos os arquivos e suas respectivas descrições:

Capítulo 3

Estruturas

3.1 Referência da Estrutura Nó

Estrutura para o nó de uma pilha que armazena tipo genérico de dado em seu campo chave.

`#include <Pilha.h>`

3.1.1 Descrição detalhada

Estrutura para o nó de uma pilha que armazena tipo genérico de dado em seu campo chave.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- Pilha.h

3.2 Referência da Estrutura no

`#include <Pilha.h>`

Campos de Dados

- void * chave

  Campo para armazenar os dados.
- struct no * prox

  Ponteiro para o próximo nó da pilha.

3.2.1 Descrição detalhada

Definição na linha 25 do arquivo Pilha.h.

3.2.2 Campos

3.2.2.1 chave

```
void* chave
```

Campo para armazenar os dados.

Definição na linha 27 do arquivo Pilha.h.

3.2.2.2 prox

```
struct no* prox
```

Ponteiro para o próximo nó da pilha.

Definição na linha 28 do arquivo Pilha.h.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- Pilha.h

3.3 Referência da Estrutura Pilha

Estrutura que armazena o topo de uma pilha.

```
#include <Pilha.h>
```

Campos de Dados

- No * topo
- int tamanho

3.3.1 Descrição detalhada

Estrutura que armazena o topo de uma pilha.

Definição na linha 35 do arquivo Pilha.h.

3.3.2 Campos

3.3.2.1 tamanho

```
int tamanho
```

Definição na linha 38 do arquivo Pilha.h.

3.3.2.2 topo

```
No* topo
```

Definição na linha 37 do arquivo Pilha.h.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- Pilha.h

3.4 Referência da Estrutura Posicao

Representa uma posição no mapa com coordenadas (x, y).

Campos de Dados

- int x
  Coordenada x (linha)
- int y
  Coordenada y (coluna)

3.4.1 Descrição detalhada

Representa uma posição no mapa com coordenadas (x, y).

Definição na linha 28 do arquivo programa3.c.

3.4.2 Campos

3.4.2.1 x

```
int x
```

Coordenada x (linha)

Definição na linha 30 do arquivo programa3.c.

3.4.2.2 y

```
int y
```

Coordenada y (coluna)

Definição na linha 31 do arquivo programa3.c.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- programa3.c

Capítulo 4

Arquivos

4.1 Referência do Arquivo Pilha.c

Arquivo com a implementação das funções para a pilha.

```
#include <stdlib.h>
#include <stdio.h>
#include "Pilha.h"
```

Funções

- No ∗ criaNo (void ∗valor)

  Cria um novo nó para a pilha com o valor passado como argumento, por padrão o campo `prox` *aponta para* `NULL`*.*
- Pilha ∗ criaPilha ()

  Cria uma pilha vazia com tamanho 0.
- void empilha (Pilha ∗p, void ∗valor)

  Empilha um nó com chave `valor` *na pilha* `p`*.*
- void ∗ desempilha (Pilha ∗p)

  Desempilha um elemento da pilha `p`*, liberando a memória que ele ocupa e retornando o valor que estava no topo.*
- int estaVazia (Pilha ∗p)

  Verifica se a pilha está vazia ou não.
- void imprimePilha (Pilha ∗p, void(∗imprimeCB)(void ∗))

  Imprime todos elementos da pilha.
- void esvaziaPilha (Pilha ∗p)

  Esvazia uma dada pilha, liberando a memória de cada nó (sem liberar o ponteiro da pilha em si).
- void ∗ topoPilha (Pilha ∗p)

  Retorna o valor armazenado no topo da pilha, sem removê-lo.

4.1.1 Descrição detalhada

Arquivo com a implementação das funções para a pilha.

Autores

Davi Brandão de Souza
Mauricio Zanetti Neto
Pedro Henrique Alves do Nascimento
Silvio Eduardo Bellinazzi de Andrade

agosto 2025

Definição no arquivo Pilha.c.

4.1.2 Funções

4.1.2.1 criaNo()

```
No * criaNo (
            void * valor)
```

Cria um novo nó para a pilha com o valor passado como argumento, por padrão o campo `prox` aponta para `NULL`.

Parâmetros

valor	Ponteiro para o valor que será inserido no nó criado.

Retorna

Retorna o ponteiro para o nó criado com o valor dado.

Definição na linha 22 do arquivo Pilha.c.

```
00023 {
00024     No *novoNo = (No *)malloc(sizeof(No));
00025     if (novoNo == NULL)
00026     {
00027         perror("Erro ao alocar nó.\n");
00028         exit(1);
00029     }
00030     novoNo->chave = valor;
00031     novoNo->prox = NULL;
00032     return novoNo;
00033 }
```

4.1.2.2 criaPilha()

```
Pilha * criaPilha ()
```

Cria uma pilha vazia com tamanho 0.

Retorna

Pilha criada.

Definição na linha 35 do arquivo Pilha.c.

```
00036 {
00037     Pilha *novaPilha = (Pilha *)malloc(sizeof(Pilha));
00038     if (novaPilha == NULL)
00039     {
00040         perror("Erro ao alocar pilha.\n");
00041         exit(1);
00042     }
00043     novaPilha->topo = NULL;
00044     novaPilha->tamanho = 0;
00045     return novaPilha;
00046 }
```

4.1.2.3 desempilha()

```
void * desempilha (
            Pilha * p)
```

Desempilha um elemento da pilha `p`, liberando a memória que ele ocupa e retornando o valor que estava no topo.

Parâmetros

p	Pilha que terá um elemnento desempilhado.

Retorna

Valor que estava armazenado no topo da pilha.

Definição na linha 59 do arquivo Pilha.c.

```
00060 {
00061     if (p == NULL || p->topo == NULL)
00062     {
00063         return NULL;
00064     }                   // Verificando os casos em que a pilha está vazia
00065     No *aux = p->topo; // Nó auxiliar que armazena o endereço inicial de p.
00066     void *chave = aux->chave;
00067     p->topo = aux->prox;
00068     p->tamanho--;
00069     free(aux);
00070     return chave;
00071 }
```

4.1.2.4 empilha()

```
void empilha (
            Pilha * p,
            void * valor)
```

Empilha um nó com chave `valor` na pilha `p`.

Parâmetros

p	Pilha na qual o elemento será empilhado.
valor	Ponteiro para o valor que será empilhado na pilha.

Retorna

void.

Definição na linha 48 do arquivo Pilha.c.

```
00049 {
00050     if (p != NULL)
00051     {
00052         No *noEmp = criaNo(valor); // Nó que vai ser empilhado na pilha.
00053         noEmp->prox = p->topo;
00054         p->topo = noEmp;
00055         p->tamanho++;
00056     }
00057 }
```

4.1.2.5 estaVazia()

```
int estaVazia (
            Pilha * p)
```

Verifica se a pilha está vazia ou não.

Parâmetros

p	A pilha a ser verificada.

Retorna

Retorna 1 se a pilha está vazia ou se não está inicializada, e 0 se não.

Definição na linha 73 do arquivo Pilha.c.

```
00074 {
00075     if (p == NULL)
00076     {
00077         return 1;
00078     }
00079     return (p->tamanho == 0);
00080 }
```

4.1.2.6 esvaziaPilha()

```
void esvaziaPilha (
            Pilha * p)
```

Esvazia uma dada pilha, liberando a memória de cada nó (sem liberar o ponteiro da pilha em si).

Parâmetros

p	Pilha a ser esvaziada.

Retorna

void.

Definição na linha 97 do arquivo Pilha.c.

```
00098 {
00099     if (p == NULL)
00100     {
00101         return;
00102     }
00103     while (p->topo != NULL)
00104     {
00105         desempilha(p);
00106     }
00107 }
```

4.1.2.7 imprimePilha()

```
void imprimePilha (
            Pilha * p,
            void(* imprimeCB )(void *))
```

Imprime todos elementos da pilha.

Parâmetros

p	Pilha a ser impressa.
imprimeCB	Função *callback* para imprimir cada elemento da pilha.

Retorna

void.

Definição na linha 82 do arquivo Pilha.c.

```
00083 {
00084     if (p == NULL)
00085     {
00086         return;
00087     }
00088     No *aux = p->topo;
00089     while (aux != NULL)
00090     {
00091         imprimeCB(aux->chave);
00092         aux = aux->prox;
00093     }
00094     printf("\n");
00095 }
```

4.1.2.8 topoPilha()

```
void * topoPilha (
            Pilha * p)
```

Retorna o valor armazenado no topo da pilha, sem removê-lo.

Parâmetros

p	A pilha a ser inspecionada.

Retorna

Ponteiro para o valor que está no topo da pilha, ou NULL caso a pilha esteja vazia ou não tenha sido inicializada.

Definição na linha 109 do arquivo Pilha.c.

```
00110 {
00111     if (p == NULL || estaVazia(p))
00112     {
00113         return NULL;
00114     }
00115     return p->topo->chave;
00116 }
```

4.2 Pilha.c

Ir para a documentação desse arquivo.

```
00001 /* Grupo 4
00002  * Davi Brandão de Souza
00003  * Mauricio Zanetti Neto
00004  * Pedro Henrique Alves do Nascimento
00005  * Silvio Eduardo Bellinazzi de Andrade
00006  */
00007
00018 #include <stdlib.h>
00019 #include <stdio.h>
00020 #include "Pilha.h"
00021
00022 No *criaNo(void *valor)
00023 {
00024     No *novoNo = (No *)malloc(sizeof(No));
00025     if (novoNo == NULL)
00026     {
```

```
00027         perror("Erro ao alocar nó.\n");
00028         exit(1);
00029     }
00030     novoNo->chave = valor;
00031     novoNo->prox = NULL;
00032     return novoNo;
00033 }
00034
00035 Pilha *criaPilha()
00036 {
00037     Pilha *novaPilha = (Pilha *)malloc(sizeof(Pilha));
00038     if (novaPilha == NULL)
00039     {
00040         perror("Erro ao alocar pilha.\n");
00041         exit(1);
00042     }
00043     novaPilha->topo = NULL;
00044     novaPilha->tamanho = 0;
00045     return novaPilha;
00046 }
00047
00048 void empilha(Pilha *p, void *valor)
00049 {
00050     if (p != NULL)
00051     {
00052         No *noEmp = criaNo(valor); // Nó que vai ser empilhado na pilha.
00053         noEmp->prox = p->topo;
00054         p->topo = noEmp;
00055         p->tamanho++;
00056     }
00057 }
00058
00059 void *desempilha(Pilha *p)
00060 {
00061     if (p == NULL || p->topo == NULL)
00062     {
00063         return NULL;
00064     }                   // Verificando os casos em que a pilha está vazia
00065     No *aux = p->topo; // Nó auxiliar que armazena o endereço inicial de p.
00066     void *chave = aux->chave;
00067     p->topo = aux->prox;
00068     p->tamanho--;
00069     free(aux);
00070     return chave;
00071 }
00072
00073 int estaVazia(Pilha *p)
00074 {
00075     if (p == NULL)
00076     {
00077         return 1;
00078     }
00079     return (p->tamanho == 0);
00080 }
00081
00082 void imprimePilha(Pilha *p, void (*imprimeCB)(void *))
00083 {
00084     if (p == NULL)
00085     {
00086         return;
00087     }
00088     No *aux = p->topo;
00089     while (aux != NULL)
00090     {
00091         imprimeCB(aux->chave);
00092         aux = aux->prox;
00093     }
00094     printf("\n");
00095 }
00096
00097 void esvaziaPilha(Pilha *p)
00098 {
00099     if (p == NULL)
00100     {
00101         return;
00102     }
00103     while (p->topo != NULL)
00104     {
00105         desempilha(p);
00106     }
00107 }
00108
00109 void *topoPilha(Pilha *p)
00110 {
00111     if (p == NULL || estaVazia(p))
00112     {
00113         return NULL;
```

```
00114     }
00115     return p->topo->chave;
00116 }
```

4.3 Referência do Arquivo Pilha.h

Arquivo header com a declaração de structs e funções para implementação da pilha utilizando lista simplesmente encadeada com nó cabeça.

Estruturas de Dados

- struct no
- struct Pilha

  Estrutura que armazena o topo de uma pilha.

Definições de Tipos

- typedef struct no No

Funções

- No ∗ criaNo (void ∗valor)

  Cria um novo nó para a pilha com o valor passado como argumento, por padrão o campo `prox` aponta para `NULL`.
- Pilha ∗ criaPilha ()

  Cria uma pilha vazia com tamanho 0.
- void empilha (Pilha ∗p, void ∗valor)

  Empilha um nó com chave `valor` na pilha `p`.
- void ∗ desempilha (Pilha ∗p)

  Desempilha um elemento da pilha `p`, liberando a memória que ele ocupa e retornando o valor que estava no topo.
- int estaVazia (Pilha ∗p)

  Verifica se a pilha está vazia ou não.
- void imprimeCB (void ∗valor)

  Função callback para impressão de tipo genérico de dado.
- void imprimePilha (Pilha ∗p, void(∗imprimeCB)(void ∗))

  Imprime todos elementos da pilha.
- void esvaziaPilha (Pilha ∗p)

  Esvazia uma dada pilha, liberando a memória de cada nó (sem liberar o ponteiro da pilha em si).
- void ∗ topoPilha (Pilha ∗p)

  Retorna o valor armazenado no topo da pilha, sem removê-lo.

4.3.1 Descrição detalhada

Arquivo header com a declaração de structs e funções para implementação da pilha utilizando lista simplesmente encadeada com nó cabeça.

Autores

Davi Brandão de Souza

Mauricio Zanetti Neto

Pedro Henrique Alves do Nascimento

Silvio Eduardo Bellinazzi de Andrade

agosto 2025

Definição no arquivo Pilha.h.

4.3.2 Definições dos tipos

4.3.2.1 No

```
typedef struct no No
```

4.3.3 Funções

4.3.3.1 criaNo()

```
No * criaNo (
            void * valor)
```

Cria um novo nó para a pilha com o valor passado como argumento, por padrão o campo `prox` aponta para `NULL`.

Parâmetros

valor	Ponteiro para o valor que será inserido no nó criado.

Retorna

Retorna o ponteiro para o nó criado com o valor dado.

Definição na linha 22 do arquivo Pilha.c.

```
00023 {
00024     No *novoNo = (No *)malloc(sizeof(No));
00025     if (novoNo == NULL)
00026     {
00027         perror("Erro ao alocar nó.\n");
00028         exit(1);
00029     }
00030     novoNo->chave = valor;
00031     novoNo->prox = NULL;
00032     return novoNo;
00033 }
```

4.3.3.2 criaPilha()

```
Pilha * criaPilha ()
```

Cria uma pilha vazia com tamanho 0.

Retorna

Pilha criada.

Definição na linha 35 do arquivo Pilha.c.

```
00036 {
00037     Pilha *novaPilha = (Pilha *)malloc(sizeof(Pilha));
00038     if (novaPilha == NULL)
00039     {
00040         perror("Erro ao alocar pilha.\n");
00041         exit(1);
00042     }
00043     novaPilha->topo = NULL;
00044     novaPilha->tamanho = 0;
00045     return novaPilha;
00046 }
```

4.3.3.3 desempilha()

```
void * desempilha (
            Pilha * p)
```

Desempilha um elemento da pilha `p`, liberando a memória que ele ocupa e retornando o valor que estava no topo.

Parâmetros

p	Pilha que terá um elemnento desempilhado.

Retorna

Valor que estava armazenado no topo da pilha.

Definição na linha 59 do arquivo Pilha.c.

```
00060 {
00061     if (p == NULL || p->topo == NULL)
00062     {
00063         return NULL;
00064     }                   // Verificando os casos em que a pilha está vazia
00065     No *aux = p->topo; // Nó auxiliar que armazena o endereço inicial de p.
00066     void *chave = aux->chave;
00067     p->topo = aux->prox;
00068     p->tamanho--;
00069     free(aux);
00070     return chave;
00071 }
```

4.3.3.4 empilha()

```
void empilha (
              Pilha * p,
              void * valor)
```

Empilha um nó com chave `valor` na pilha `p`.

Parâmetros

p	Pilha na qual o elemento será empilhado.
valor	Ponteiro para o valor que será empilhado na pilha.

Retorna

void.

Definição na linha 48 do arquivo Pilha.c.

```
00049 {
00050     if (p != NULL)
00051     {
00052         No *noEmp = criaNo(valor); // Nó que vai ser empilhado na pilha.
00053         noEmp->prox = p->topo;
00054         p->topo = noEmp;
00055         p->tamanho++;
00056     }
00057 }
```

4.3.3.5 estaVazia()

```
int estaVazia (
              Pilha * p)
```

Verifica se a pilha está vazia ou não.

Parâmetros

p	A pilha a ser verificada.

Retorna

Retorna 1 se a pilha está vazia ou se não está inicializada, e 0 se não.

Definição na linha 73 do arquivo Pilha.c.

```
00074 {
00075     if (p == NULL)
00076     {
00077         return 1;
00078     }
00079     return (p->tamanho == 0);
00080 }
```

4.3.3.6 esvaziaPilha()

```
void esvaziaPilha (
            Pilha * p)
```

Esvazia uma dada pilha, liberando a memória de cada nó (sem liberar o ponteiro da pilha em si).

Parâmetros

p	Pilha a ser esvaziada.

Retorna

void.

Definição na linha 97 do arquivo Pilha.c.

```
00098 {
00099     if (p == NULL)
00100     {
00101         return;
00102     }
00103     while (p->topo != NULL)
00104     {
00105         desempilha(p);
00106     }
00107 }
```

4.3.3.7 imprimeCB()

```
void imprimeCB (
            void * valor)
```

Função *callback* para impressão de tipo genérico de dado.

Parâmetros

valor	Ponteiro para o dado a ser impresso.

Retorna

void.

4.3.3.8 imprimePilha()

```
void imprimePilha (
            Pilha * p,
            void(* imprimeCB )(void *))
```

Imprime todos elementos da pilha.

Parâmetros

p	Pilha a ser impressa.
imprimeCB	Função *callback* para imprimir cada elemento da pilha.

Retorna

void.

Definição na linha 82 do arquivo Pilha.c.

```
00083 {
00084     if (p == NULL)
00085     {
00086         return;
00087     }
00088     No *aux = p->topo;
00089     while (aux != NULL)
00090     {
00091         imprimeCB(aux->chave);
00092         aux = aux->prox;
00093     }
00094     printf("\n");
00095 }
```

4.3.3.9 topoPilha()

```
void * topoPilha (
            Pilha * p)
```

Retorna o valor armazenado no topo da pilha, sem removê-lo.

Parâmetros

p	A pilha a ser inspecionada.

Retorna

Ponteiro para o valor que está no topo da pilha, ou NULL caso a pilha esteja vazia ou não tenha sido inicializada.

Definição na linha 109 do arquivo Pilha.c.

```
00110 {
00111     if (p == NULL || estaVazia(p))
00112     {
00113         return NULL;
00114     }
00115     return p->topo->chave;
00116 }
```

4.4 Pilha.h

Ir para a documentação desse arquivo.

```
00001 /* Grupo 4
00002  * Davi Brandão de Souza
00003  * Mauricio Zanetti Neto
00004  * Pedro Henrique Alves do Nascimento
00005  * Silvio Eduardo Bellinazzi de Andrade
00006  */
00007
00018 #ifndef PILHA_H
00019 #define PILHA_H
00020
00025 typedef struct no
00026 {
00027     void *chave;
00028     struct no *prox;
00029 } No;
00030
00035 typedef struct
00036 {
00037     No *topo;
00038     int tamanho;
00039 } Pilha;
00040
00046 No *criaNo(void *valor);
00047
00052 Pilha *criaPilha();
00053
00060 void empilha(Pilha *p, void *valor);
00061
00067 void *desempilha(Pilha *p);
00068
00074 int estaVazia(Pilha *p);
00075
00081 void imprimeCB(void *valor);
00082
00089 void imprimePilha(Pilha *p, void (*imprimeCB)(void *));
00090
00096 void esvaziaPilha(Pilha *p);
00097
00103 void *topoPilha(Pilha *p);
00104
00105 #endif
```

4.5 Referência do Arquivo programa1.c

Arquivo com a implementação do programa 1 para validação de expressões matemáticas.

```
#include "Pilha.h"
#include <stdlib.h>
#include <stdio.h>
```

Funções

- int validarExpressao (char *exp)

  Função que valida expressões matemáticas conforme seus delimitadores. Lê a expressão caractere a caractere e empilha somente os delimitadores de abertura, se o próximo caractere delimitador for de fechamento e for o correspondente, desempilha o elemento do topo e continua. Leva em consideração a hierarquia dos delimitadores, não pode haver chaves ou colchetes dentro de parênteses, nem chaves dentro de colchetes.

- int main ()

  Função para utilizar o programa, aceita como entrada a expressão que deseja que seja validada e imprime na tela se é válida ou não.

4.5.1 Descrição detalhada

Arquivo com a implementação do programa 1 para validação de expressões matemáticas.

Autores

Davi Brandão de Souza

Mauricio Zanetti Neto

Pedro Henrique Alves do Nascimento

Silvio Eduardo Bellinazzi de Andrade

agosto 2025

Definição no arquivo programa1.c.

4.5.2 Funções

4.5.2.1 main()

```
int main ()
```

Função para utilizar o programa, aceita como entrada a expressão que deseja que seja validada e imprime na tela se é válida ou não.

Definição na linha 90 do arquivo programa1.c.

```
00090           {
00091     int execucao = 1;
00092     char expressao[512];
00093     while (execucao != 0){
00094     printf("Digite a expressão: ");
00095     getchar();
00096     fgets(expressao, 512, stdin);
00097     if (validarExpressao(expressao)){ printf("VALIDA\n"); }
00098     else{ printf("INVALIDA\n"); }
00099     printf("Deseja validar outra expressão? (0-Não/1-Sim) ");
00100     scanf(" %d", &execucao);
00101     }
00102     return EXIT_SUCCESS;
00103 }
```

4.5.2.2 validarExpressao()

```
int validarExpressao (
            char * exp)
```

Função que valida expressões matemáticas conforme seus delimitadores. Lê a expressão caractere a caractere e empilha somente os delimitadores de abertura, se o próximo caractere delimitador for de fechamento e for o correspondente, desempilha o elemento do topo e continua. Leva em consideração a hierarquia dos delimitadores, não pode haver chaves ou colchetes dentro de parênteses, nem chaves dentro de colchetes.

Parâmetros

exp	Expressão que deve ser validada (*string*).

Retorna

Retorna 0 para uma expressão inválida e 1 para uma expressão válida.

Definição na linha 32 do arquivo programa1.c.

```
00033 {
00034     if (exp == NULL){ return 0; }
00035     Pilha *p = criaPilha();
00036     int i = 0;
00037     while (exp[i] != '\0'){
00038     char caractere = exp[i];
00039     if (caractere == '(' || caractere == '[' || caractere == '{'){ // Insere na pilha somente os
      caracteres de abertura
00040         /* Para tratar a opção b, foi feito uso dos valores dos caracteres de delimitação
00041          * na tabela ASCII, onde os delimitadores com maior precedência sempre têm valores
00042          * númericos menores que os de menor precedência.
00043          *
00044          * Exemplo: (A{B-C}*E), com '(' estando no topo da pilha, tenta-se empilhar '{',
00045          * os respectivos valores desses caracteres em decimal são 40 e 123, então a comparação
00046          * feita abaixo fica: 40 - 123 < 0 -> verdadeiro, ou seja, a expressão é inválida pela
00047          * opção b, pois o '{' está numa posição mais interna que '('.
00048          */
00049         if (!estaVazia(p) && (*(char *)p->topo->chave - caractere < 0)){
00050         esvaziaPilha(p);
00051         free(p);
00052         return 0;
00053         }
00054         char *dadoEmpilhar = (char *)malloc(sizeof(char));
00055         if (dadoEmpilhar == NULL){
00056         perror("Erro ao alocar dado.\n");
00057         exit(1);
00058         }
00059         *dadoEmpilhar = caractere;
00060         empilha(p, dadoEmpilhar);
00061     }
00062     else if (caractere == ')' || caractere == ']' || caractere == '}'){
00063         if (estaVazia(p)){
00064         esvaziaPilha(p);
00065         free(p);
00066         return 0;
00067         }
00068         char topo = *(char *)desempilha(p);
00069
00070         if ((caractere == ')' && topo != '(') ||
00071         (caractere == ']' && topo != '[') ||
00072         (caractere == '}' && topo != '{')){
00073         esvaziaPilha(p);
00074         free(p);
00075         return 0;
00076         }
00077     }
00078     i++;
00079     }
00080     int valida = estaVazia(p); // A pilha deve estar vazia para garantir que não há delimitadores de
      abertura sem fechamento
00081     esvaziaPilha(p);
00082     free(p);
00083
00084     return valida;
00085 }
```

4.6 programa1.c

Ir para a documentação desse arquivo.

```
00001 /* Grupo 4
00002  * Davi Brandão de Souza
00003  * Mauricio Zanetti Neto
00004  * Pedro Henrique Alves do Nascimento
00005  * Silvio Eduardo Bellinazzi de Andrade
00006  */
00007
00018 #include "Pilha.h"
00019 #include <stdlib.h>
00020 #include <stdio.h>
00021
00032 int validarExpressao(char *exp)
00033 {
00034     if (exp == NULL){ return 0; }
```

```
00035     Pilha *p = criaPilha();
00036     int i = 0;
00037     while (exp[i] != '\0'){
00038     char caractere = exp[i];
00039     if (caractere == '(' || caractere == '[' || caractere == '{'){ // Insere na pilha somente os
      caracteres de abertura
00040         /* Para tratar a opção b, foi feito uso dos valores dos caracteres de delimitação
00041          * na tabela ASCII, onde os delimitadores com maior precedência sempre têm valores
00042          * númericos menores que os de menor precedência.
00043          *
00044          * Exemplo: (A{B-C}*E), com '(' estando no topo da pilha, tenta-se empilhar '{',
00045          * os respectivos valores desses caracteres em decimal são 40 e 123, então a comparação
00046          * feita abaixo fica: 40 - 123 < 0 -> verdadeiro, ou seja, a expressão é inválida pela
00047          * opção b, pois o '{' está numa posição mais interna que '('.
00048          */
00049         if (!estaVazia(p) && (*(char *)p->topo->chave - caractere < 0)){
00050         esvaziaPilha(p);
00051         free(p);
00052         return 0;
00053         }
00054         char *dadoEmpilhar = (char *)malloc(sizeof(char));
00055         if (dadoEmpilhar == NULL){
00056         perror("Erro ao alocar dado.\n");
00057         exit(1);
00058         }
00059         *dadoEmpilhar = caractere;
00060         empilha(p, dadoEmpilhar);
00061     }
00062     else if (caractere == ')' || caractere == ']' || caractere == '}'){
00063         if (estaVazia(p)){
00064         esvaziaPilha(p);
00065         free(p);
00066         return 0;
00067         }
00068         char topo = *(char *)desempilha(p);
00069
00070         if ((caractere == ')' && topo != '(') ||
00071         (caractere == ']' && topo != '[') ||
00072         (caractere == '}' && topo != '{')){
00073         esvaziaPilha(p);
00074         free(p);
00075         return 0;
00076         }
00077     }
00078     i++;
00079     }
00080     int valida = estaVazia(p); // A pilha deve estar vazia para garantir que não há delimitadores de
      abertura sem fechamento
00081     esvaziaPilha(p);
00082     free(p);
00083
00084     return valida;
00085 }
00086
00090 int main() {
00091     int execucao = 1;
00092     char expressao[512];
00093     while (execucao != 0){
00094     printf("Digite a expressão: ");
00095     getchar();
00096     fgets(expressao, 512, stdin);
00097     if (validarExpressao(expressao)){ printf("VALIDA\n"); }
00098     else{ printf("INVALIDA\n"); }
00099     printf("Deseja validar outra expressão? (0-Não/1-Sim) ");
00100     scanf(" %d", &execucao);
00101     }
00102     return EXIT_SUCCESS;
00103 }
```

4.7 Referência do Arquivo programa2.c

Algoritmo para Avaliação de Expressões em Notação Pós-Fixa.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Pilha.h"
```

Definições e Macros

- #define MAX_EXPRESSAO 100

Funções

- int ehOperador (char c)

  Verifica o operador . A função verifica se o caractere é um dos operadores aritméticos.
- int precedencia (char op)

  Define a prioridade dos operadores. A função retorna um valor inteiro que representa a precedência do operador. Operadores com maior precedência retornam valores maiores.
- void infixaParaPosfixa (char ∗infixa, char ∗posfixa)

  Converte uma expressão infixa para posfixa. A função armazena os operandos diretamente na saída e usa uma pilha para gerenciar os operadores. Os operadores são empilhados e desempilhados conforme a precedência, e os parênteses são tratados adequadamente.
- float avaliarPosfixa (char ∗posfixa, float valores[ ])

  Interpreta e avalia uma expressão aritmética em notação polonesa reversa A função armazena os valores das variáveis em uma pilha e realiza as operações conforme encontra operadores na expressão.
- void marcarLetrasUsadas (char ∗expr, int usadas[ ])

  Marca as letras usadas. Percorre a expressão e marca as letras usadas no array.
- int main ()

  Função principal do programa. Lê o tipo de expressão (infixa ou posfixa), converte se necessário, solicita os valores das variáveis usadas e avalia a expressão.

4.7.1 Descrição detalhada

Algoritmo para Avaliação de Expressões em Notação Pós-Fixa.

Autores

Davi Brandão de Souza

Mauricio Zanetti Neto

Pedro Henrique Alves do Nascimento

Silvio Eduardo Bellinazzi de Andrade

Definição no arquivo programa2.c.

4.7.2 Definições e macros

4.7.2.1 MAX_EXPRESSAO

```
#define MAX_EXPRESSAO 100
```

Definição na linha 22 do arquivo programa2.c.

4.7.3 Funções

4.7.3.1 avaliarPosfixa()

```
float avaliarPosfixa (
            char * posfixa,
            float valores[])
```

Interpreta e avalia uma expressão aritmética em notação polonesa reversa A função armazena os valores das variáveis em uma pilha e realiza as operações conforme encontra operadores na expressão.

Parâmetros

posfixa	Expressão em notação polonesa reversa.
valores	Array de valores das variáveis (A-J).

Retorna

Resultado da avaliação da expressão.

Definição na linha 145 do arquivo programa2.c.

```
00146 {
00147     Pilha *pilha = criaPilha();
00148
00149     for (int i = 0; posfixa[i] != '\0'; i++) // percorre a expressão posfixa
00150     {
00151         char c = posfixa[i];
00152
00153         if (c == ' ')
00154             continue;
00155
00156         if (c >= 'A' && c <= 'J') // se for operando, empilha o valor correspondente
00157         {
00158             float *val = malloc(sizeof(float));
00159             *val = valores[c - 'A'];
00160             empilha(pilha, val);
00161         }
00162         else if (ehOperador(c))
00163         {
00164             if (estaVazia(pilha))
00165             {
00166                 // Erro: Falta operandos para o operador
00167                 exit(1);
00168             }
00169             float *op2 = (float *)desempilha(pilha);
00170
00171             if (estaVazia(pilha))
00172             {
00173                 // Erro: Falta operandos para o operador
00174                 exit(1);
00175             }
00176             float *op1 = (float *)desempilha(pilha); // desempilha os dois operandos
00177
00178             float *resultado = malloc(sizeof(float));
00179
00180             switch (c) // realiza a operação correspondente
00181             {
00182             case '+':
00183                 *resultado = *op1 + *op2;
00184                 break;
00185             case '-':
00186                 *resultado = *op1 - *op2;
00187                 break;
00188             case '*':
00189                 *resultado = *op1 * *op2;
00190                 break;
00191             case '/':
00192                 if (*op2 == 0.0f)
00193                 {
00194                     // Erro: Divisão por zero
00195             printf("Divisão por zero.\n");
00196                     exit(1);
00197                 }
00198                 *resultado = *op1 / *op2;
00199                 break;
00200             case '^':
00201                 *resultado = pow(*op1, *op2);
00202                 break;
00203             }
00204
00205             free(op1);
00206             free(op2);
00207             empilha(pilha, resultado);
00208         }
00209         else
00210         {
00211             // Erro: Caractere inválido
00212             exit(1);
00213         }
00214     }
```

```
00215
00216     if (estaVazia(pilha))
00217     {
00218         // Erro: Expressão vazia ou inválida
00219         exit(1);
00220     }
00221     float resultadoFinal = *((float *)desempilha(pilha));
00222
00223     if (!estaVazia(pilha))
00224     {
00225         // Erro: Expressão possui operandos no final
00226         exit(1);
00227     }
00228
00229     free(pilha);
00230     return resultadoFinal;
00231 }
```

4.7.3.2 ehOperador()

```
int ehOperador (
            char c)
```

Verifica o operador . A função verifica se o caractere é um dos operadores aritméticos.

Parâmetros

c	Caractere a ser verificado.

Retorna

1 se for um operador, 0 caso contrário.

Definição na linha 30 do arquivo programa2.c.

```
00031 {
00032     return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
00033 }
```

4.7.3.3 infixaParaPosfixa()

```
void infixaParaPosfixa (
            char * infixa,
            char * posfixa)
```

Converte uma expressão infixa para posfixa. A função armazena os operandos diretamente na saída e usa uma pilha para gerenciar os operadores. Os operadores são empilhados e desempilhados conforme a precedência, e os parênteses são tratados adequadamente.

Parâmetros

infixa	Expressão infixa a ser convertida.
posfixa	Buffer onde a expressão posfixa será armazenada.

Definição na linha 66 do arquivo programa2.c.

```
00067 {
00068     Pilha *operadores = criaPilha();
00069     int j = 0;
00070 
00071     for (int i = 0; infixa[i] != '\0'; i++) // percorre a expressão infixa
00072     {
00073         char c = infixa[i];
00074 
00075         if (c == ' ')
00076             continue;
00077 
00078         if (c >= 'A' && c <= 'J')
00079         {
00080             posfixa[j++] = c; // adiciona operandos diretamente à saída
00081         }
00082         else if (c == '(')
00083         {
00084             char *abre = malloc(sizeof(char)); // aloca memória para o caractere
00085             *abre = c;
00086             empilha(operadores, abre);
00087         }
00088         else if (c == ')')
00089         {
00090             while (!estaVazia(operadores) && *((char *)topoPilha(operadores)) != '(')
00091             {
00092                 char *op = (char *)desempilha(operadores);
00093                 posfixa[j++] = *op; // adiciona operadores à saída até encontrar o parêntese de
      abertura
00094                 free(op);
00095             }
00096             if (estaVazia(operadores))
00097             {
00098                 // Erro: Parêntese de fechamento sem um par de abertura correspondente
00099                 exit(1);
00100             }
00101             free(desempilha(operadores));
00102         }
00103         else if (ehOperador(c))
00104         {
00105             while (!estaVazia(operadores) && precedencia(*((char *)topoPilha(operadores))) >=
      precedencia(c)) // desempilha operadores com maior ou igual precedência
00106             {
00107                 char *op = (char *)desempilha(operadores);
00108                 posfixa[j++] = *op;
00109                 free(op);
00110             }
00111             char *op = malloc(sizeof(char));
00112             *op = c;
00113             empilha(operadores, op);
00114         }
00115         else
00116         {
00117             // Erro: Caractere inválido
00118             exit(1);
00119         }
00120     }
00121 
00122     while (!estaVazia(operadores))
00123     {
00124         if (*((char *)topoPilha(operadores)) == '(')
00125         {
00126             // Erro: Parêntese de abertura sem um par de fechamento correspondente
00127             exit(1);
00128         }
00129         char *op = (char *)desempilha(operadores);
00130         posfixa[j++] = *op;
00131         free(op);
00132     }
00133 
00134     posfixa[j] = '\0';
00135     free(operadores);
00136 }
```

4.7.3.4 main()

```
int main ()
```

Função principal do programa. Lê o tipo de expressão (infixa ou posfixa), converte se necessário, solicita os valores das variáveis usadas e avalia a expressão.

Retorna

Código de saída do programa.

Definição na linha 253 do arquivo programa2.c.

```
00254 {
00255     char tipo;
00256     char infixa[MAX_EXPRESSAO], posfixa[MAX_EXPRESSAO];
00257     float valores[10];
00258     int usadas[10] = {0};
00259 
00260     printf("Digite o tipo da entrada, a-infixa/b-posfixa: ");
00261     scanf("%c", &tipo);
00262 
00263     if (tipo == 'a' || tipo == 'A')
00264     {
00265         // Entrada infixa
00266     printf("Digite a expressão: ");
00267         scanf(" %99[^\n]", infixa);
00268         infixaParaPosfixa(infixa, posfixa);
00269         printf("%s\n", posfixa); // converte para pósfixa
00270     }
00271     else if (tipo == 'b' || tipo == 'B')
00272     {
00273     printf("Digite a expressão: ");
00274         scanf(" %99[^\n]", posfixa);
00275     }
00276     else
00277     {
00278         // Erro: Tipo inválido
00279     printf("Erro, tipo inválido.\n");
00280         return 1;
00281     }
00282 
00283     marcarLetrasUsadas(posfixa, usadas);
00284 
00285     // Percorre as letras de A a J
00286     for (char c = 'A'; c <= 'J'; c++)
00287     {
00288         if (usadas[c - 'A']) //se a letra for usada sera solicitado o valor
00289         {
00290         printf("Digite o valor de %c: ", c);
00291         while(scanf("%f", &valores[c - 'A']) != 1){
00292         while (getchar() != '\n');
00293         printf("Digite o valor de %c: ", c);
00294         }
00295         }
00296     }
00297 
00298     float resultado = avaliarPosfixa(posfixa, valores);
00299     printf("Resultado: %.2f\n", resultado);
00300 
00301     return 0;
00302 }
```

4.7.3.5 marcarLetrasUsadas()

```
void marcarLetrasUsadas (
            char * expr,
            int usadas[])
```

Marca as letras usadas. Percorre a expressão e marca as letras usadas no array.

Parâmetros

expr	Expressão a ser analisada.
usadas	Array de inteiros onde cada índice representa uma letra (0 para A, 1 para B, ..., 9 para J).

Definição na linha 238 do arquivo programa2.c.

```
00239 {
00240     for (int i = 0; expr[i] != '\0'; i++)
00241     {
00242         char c = expr[i];
00243         if (c >= 'A' && c <= 'J')
00244         {
00245             usadas[c - 'A'] = 1;
00246         }
00247     }
00248 }
```

4.7.3.6 precedencia()

```
int precedencia (
            char op)
```

Define a prioridade dos operadores. A função retorna um valor inteiro que representa a precedência do operador. Operadores com maior precedência retornam valores maiores.

Parâmetros

op	Operador a ser verificado.

Retorna

Precedência do operador.

Definição na linha 42 do arquivo programa2.c.

```
00043 {
00044     switch (op)
00045     {
00046     case '^':
00047         return 3;
00048     case '*':
00049     case '/':
00050         return 2;
00051     case '+':
00052     case '-':
00053         return 1;
00054     default:
00055         return 0;
00056     }
00057 }
```

4.8 programa2.c

Ir para a documentação desse arquivo.

```
00001 /* Grupo 4
00002  * Davi Brandão de Souza
00003  * Mauricio Zanetti Neto
00004  * Pedro Henrique Alves do Nascimento
00005  * Silvio Eduardo Bellinazzi de Andrade
00006  */
00007
00016
00017 #include <stdio.h>
00018 #include <stdlib.h>
00019 #include <math.h>
00020 #include "Pilha.h"
```

```
00021
00022 #define MAX_EXPRESSAO 100
00023
00030 int ehOperador(char c)
00031 {
00032     return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
00033 }
00034
00042 int precedencia(char op)
00043 {
00044     switch (op)
00045     {
00046     case '^':
00047         return 3;
00048     case '*':
00049     case '/':
00050         return 2;
00051     case '+':
00052     case '-':
00053         return 1;
00054     default:
00055         return 0;
00056     }
00057 }
00058
00066 void infixaParaPosfixa(char *infixa, char *posfixa)
00067 {
00068     Pilha *operadores = criaPilha();
00069     int j = 0;
00070
00071     for (int i = 0; infixa[i] != '\0'; i++) // percorre a expressão infixa
00072     {
00073         char c = infixa[i];
00074
00075         if (c == ' ')
00076             continue;
00077
00078         if (c >= 'A' && c <= 'J')
00079         {
00080             posfixa[j++] = c; // adiciona operandos diretamente à saída
00081         }
00082         else if (c == '(')
00083         {
00084             char *abre = malloc(sizeof(char)); // aloca memória para o caractere
00085             *abre = c;
00086             empilha(operadores, abre);
00087         }
00088         else if (c == ')')
00089         {
00090             while (!estaVazia(operadores) && *((char *)topoPilha(operadores)) != '(')
00091             {
00092                 char *op = (char *)desempilha(operadores);
00093                 posfixa[j++] = *op; // adiciona operadores à saída até encontrar o parêntese de
      abertura
00094                 free(op);
00095             }
00096             if (estaVazia(operadores))
00097             {
00098                 // Erro: Parêntese de fechamento sem um par de abertura correspondente
00099                 exit(1);
00100             }
00101             free(desempilha(operadores));
00102         }
00103         else if (ehOperador(c))
00104         {
00105             while (!estaVazia(operadores) && precedencia(*((char *)topoPilha(operadores))) >=
      precedencia(c)) // desempilha operadores com maior ou igual precedência
00106             {
00107                 char *op = (char *)desempilha(operadores);
00108                 posfixa[j++] = *op;
00109                 free(op);
00110             }
00111             char *op = malloc(sizeof(char));
00112             *op = c;
00113             empilha(operadores, op);
00114         }
00115         else
00116         {
00117             // Erro: Caractere inválido
00118             exit(1);
00119         }
00120     }
00121
00122     while (!estaVazia(operadores))
00123     {
00124         if (*((char *)topoPilha(operadores)) == '(')
00125         {
```

```
00126             // Erro: Parêntese de abertura sem um par de fechamento correspondente
00127             exit(1);
00128         }
00129         char *op = (char *)desempilha(operadores);
00130         posfixa[j++] = *op;
00131         free(op);
00132     }
00133
00134     posfixa[j] = '\0';
00135     free(operadores);
00136 }
00137
00145 float avaliarPosfixa(char *posfixa, float valores[])
00146 {
00147     Pilha *pilha = criaPilha();
00148
00149     for (int i = 0; posfixa[i] != '\0'; i++) // percorre a expressão posfixa
00150     {
00151         char c = posfixa[i];
00152
00153         if (c == ' ')
00154             continue;
00155
00156         if (c >= 'A' && c <= 'J') // se for operando, empilha o valor correspondente
00157         {
00158             float *val = malloc(sizeof(float));
00159             *val = valores[c - 'A'];
00160             empilha(pilha, val);
00161         }
00162         else if (ehOperador(c))
00163         {
00164             if (estaVazia(pilha))
00165             {
00166                 // Erro: Falta operandos para o operador
00167                 exit(1);
00168             }
00169             float *op2 = (float *)desempilha(pilha);
00170
00171             if (estaVazia(pilha))
00172             {
00173                 // Erro: Falta operandos para o operador
00174                 exit(1);
00175             }
00176             float *op1 = (float *)desempilha(pilha); // desempilha os dois operandos
00177
00178             float *resultado = malloc(sizeof(float));
00179
00180             switch (c) // realiza a operação correspondente
00181             {
00182             case '+':
00183                 *resultado = *op1 + *op2;
00184                 break;
00185             case '-':
00186                 *resultado = *op1 - *op2;
00187                 break;
00188             case '*':
00189                 *resultado = *op1 * *op2;
00190                 break;
00191             case '/':
00192                 if (*op2 == 0.0f)
00193                 {
00194                     // Erro: Divisão por zero
00195             printf("Divisão por zero.\n");
00196                     exit(1);
00197                 }
00198                 *resultado = *op1 / *op2;
00199                 break;
00200             case '^':
00201                 *resultado = pow(*op1, *op2);
00202                 break;
00203             }
00204
00205             free(op1);
00206             free(op2);
00207             empilha(pilha, resultado);
00208         }
00209         else
00210         {
00211             // Erro: Caractere inválido
00212             exit(1);
00213         }
00214     }
00215
00216     if (estaVazia(pilha))
00217     {
00218         // Erro: Expressão vazia ou inválida
00219         exit(1);
```

```
00220     }
00221     float resultadoFinal = *((float *)desempilha(pilha));
00222
00223     if (!estaVazia(pilha))
00224     {
00225         // Erro: Expressão possui operandos no final
00226         exit(1);
00227     }
00228
00229     free(pilha);
00230     return resultadoFinal;
00231 }
00232
00238 void marcarLetrasUsadas(char *expr, int usadas[])
00239 {
00240     for (int i = 0; expr[i] != '\0'; i++)
00241     {
00242         char c = expr[i];
00243         if (c >= 'A' && c <= 'J')
00244         {
00245             usadas[c - 'A'] = 1;
00246         }
00247     }
00248 }
00253 int main()
00254 {
00255     char tipo;
00256     char infixa[MAX_EXPRESSAO], posfixa[MAX_EXPRESSAO];
00257     float valores[10];
00258     int usadas[10] = {0};
00259
00260     printf("Digite o tipo da entrada, a-infixa/b-posfixa: ");
00261     scanf("%c", &tipo);
00262
00263     if (tipo == 'a' || tipo == 'A')
00264     {
00265         // Entrada infixa
00266     printf("Digite a expressão: ");
00267         scanf(" %99[^\n]", infixa);
00268         infixaParaPosfixa(infixa, posfixa);
00269         printf("%s\n", posfixa); // converte para pósfixa
00270     }
00271     else if (tipo == 'b' || tipo == 'B')
00272     {
00273     printf("Digite a expressão: ");
00274         scanf(" %99[^\n]", posfixa);
00275     }
00276     else
00277     {
00278         // Erro: Tipo inválido
00279     printf("Erro, tipo inválido.\n");
00280         return 1;
00281     }
00282
00283     marcarLetrasUsadas(posfixa, usadas);
00284
00285     // Percorre as letras de A a J
00286     for (char c = 'A'; c <= 'J'; c++)
00287     {
00288         if (usadas[c - 'A']) //se a letra for usada sera solicitado o valor
00289         {
00290         printf("Digite o valor de %c: ", c);
00291         while(scanf("%f", &valores[c - 'A']) != 1){
00292         while (getchar() != '\n');
00293         printf("Digite o valor de %c: ", c);
00294         }
00295         }
00296     }
00297
00298     float resultado = avaliarPosfixa(posfixa, valores);
00299     printf("Resultado: %.2f\n", resultado);
00300
00301     return 0;
00302 }
```

4.9 Referência do Arquivo programa3.c

Implementação das funções para contagem de cômodos em uma planta de casa.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdbool.h>
#include "Pilha.h"
```

Estruturas de Dados

- struct Posicao

  Representa uma posição no mapa com coordenadas (x, y).

Definições e Macros

- #define MAX 1000

Funções

- bool posicaoValida (int x, int y, int n, int m, char mapa[MAX][MAX], bool processado[MAX][MAX])

  Verifica se uma posição (x,y) é válida para visitar no mapa.
- void visitarComodo (Pilha *pilha, int x0, int y0, int n, int m, char mapa[MAX][MAX], bool processado[MAX][MAX])

  Explora um cômodo do mapa a partir da posição inicial usando busca em profundidade.
- int contaComodos (int n, int m, char mapa[MAX][MAX])

  Conta o número de cômodos (áreas conectadas de '.') no mapa.
- int main ()

  Função principal do programa.

4.9.1 Descrição detalhada

Implementação das funções para contagem de cômodos em uma planta de casa.

Autores

Davi Brandão de Souza

Mauricio Zanetti Neto

Pedro Henrique Alves do Nascimento

Silvio Eduardo Bellinazzi de Andrade

Definição no arquivo programa3.c.

4.9.2 Definições e macros

4.9.2.1 MAX

```
#define MAX 1000
```

Definição na linha 22 do arquivo programa3.c.

4.9.3 Funções

4.9.3.1 contaComodos()

```
int contaComodos (
            int n,
            int m,
            char mapa[MAX][MAX])
```

Conta o número de cômodos (áreas conectadas de '.') no mapa.

Percorre o mapa, e para cada ponto não processado que seja '.' chama a função visitarComodo, que marca todo o cômodo como processado.

Parâmetros

n	Número de linhas do mapa.
m	Número de colunas do mapa.
mapa	Matriz do mapa.

Retorna

O total de cômodos encontrados no mapa.

Definição na linha 120 do arquivo programa3.c.

```
00121 {
00122     bool processado[MAX][MAX] = {false};
00123     Pilha *pilha = criaPilha();
00124     int totalComodos = 0;
00125
00126     for (int i = 0; i < n; i++)
00127     {
00128         for (int j = 0; j < m; j++)
00129         {
00130             if (mapa[i][j] == '.' && !processado[i][j])
00131             {
00132                 totalComodos++;
00133                 visitarComodo(pilha, i, j, n, m, mapa, processado);
00134             }
00135         }
00136     }
00137     esvaziaPilha(pilha);
00138     free(pilha);
00139     return totalComodos;
00140 }
```

4.9.3.2 main()

```
int main ()
```

Função principal do programa.

Lê as dimensões do mapa, lê o mapa da entrada, calcula o número de cômodos e imprime o resultado.

Retorna

0 se a execução ocorreu com sucesso.

Definição na linha 151 do arquivo programa3.c.

```
00152 {
00153     int n, m;
00154     char mapa[MAX][MAX];
00155     scanf("%d %d", &n, &m);
00156
00157     for (int i = 0; i < n; i++)
00158     {
00159         scanf("%s", mapa[i]);
00160     }
00161     int totalComodos = contaComodos(n, m, mapa);
00162     printf("%d\n", totalComodos);
00163     return 0;
00164 }
```

4.9.3.3 posicaoValida()

```
bool posicaoValida (
            int x,
            int y,
            int n,
            int m,
            char mapa[MAX][MAX],
            bool processado[MAX][MAX])
```

Verifica se uma posição (x,y) é válida para visitar no mapa.

Uma posição é válida se estiver dentro dos limites, for um espaço livre ('.') e ainda não foi processada.

Parâmetros

x	Coordenada x (linha) da posição.
y	Coordenada y (coluna) da posição.
n	Número de linhas do mapa.
m	Número de colunas do mapa.
mapa	Matriz que representa o mapa, com '.' para espaço livre.
processado	Matriz booleana que indica posições já processadas (agendadas para visitação).

Retorna

true se a posição for válida para visitar, false caso contrário.

Definição na linha 49 do arquivo programa3.c.

```
00050 {
00051     if (x >= 0 && x < n && y >= 0 && y < m && mapa[x][y] == '.' && !processado[x][y])
00052         return true;
00053     else
00054         return false;
00055 }
```

4.9.3.4 visitarComodo()

```
void visitarComodo (
            Pilha * pilha,
            int x0,
            int y0,
            int n,
            int m,
            char mapa[MAX][MAX],
            bool processado[MAX][MAX])
```

Explora um cômodo do mapa a partir da posição inicial usando busca em profundidade.

Utilizando uma pilha, a função empilha posições válidas conectadas, e desempilha para explorar seus vizinhos, marcando posições como processadas para evitar visitas repetidas.

Parâmetros

pilha	Ponteiro para a pilha usada para controle da exploração.
x0	Coordenada x (linha) inicial para começar a visita.

y0	Coordenada y (coluna) inicial para começar a visita.
n	Número de linhas do mapa.
m	Número de colunas do mapa.
mapa	Matriz do mapa com '.' indicando espaços livres.
processado	Matriz booleana que indica posições já processadas.

Definição na linha 72 do arquivo programa3.c.

```
00073 {
00074     Posicao *inicio = (Posicao *)malloc(sizeof(Posicao));
00075     if (inicio == NULL)
00076     {
00077         printf("Erro de alocacao. \n");
00078         exit(1);
00079     }
00080     inicio->x = x0;
00081     inicio->y = y0;
00082     empilha(pilha, inicio);
00083     processado[x0][y0] = true;
00084 
00085     while (!estaVazia(pilha))
00086     {
00087         Posicao *ultima = (Posicao *)desempilha(pilha);
00088         int dir_x[4] = {-1, 1, 0, 0};
00089         int dir_y[4] = {0, 0, -1, 1};
00090 
00091         for (int i = 0; i < 4; i++)
00092         {
00093             int vizinhoX = ultima->x + dir_x[i];
00094             int vizinhoY = ultima->y + dir_y[i];
00095             if (posicaoValida(vizinhoX, vizinhoY, n, m, mapa, processado))
00096             {
00097                 Posicao *vizinho = (Posicao *)malloc(sizeof(Posicao));
00098                 vizinho->x = vizinhoX;
00099                 vizinho->y = vizinhoY;
00100                 empilha(pilha, vizinho);
00101                 processado[vizinhoX][vizinhoY] = true;
00102             }
00103         }
00104         free(ultima);
00105     }
00106 }
```

4.10 programa3.c

Ir para a documentação desse arquivo.

```
00001 /* Grupo 4
00002  * Davi Brandão de Souza
00003  * Mauricio Zanetti Neto
00004  * Pedro Henrique Alves do Nascimento
00005  * Silvio Eduardo Bellinazzi de Andrade
00006  */
00007 
00016 
00017 #include <stdio.h>
00018 #include <stdlib.h>
00019 #include <stdbool.h>
00020 #include "Pilha.h"
00021 
00022 #define MAX 1000
00023 
00028 typedef struct
00029 {
00030     int x;
00031     int y;
00032 } Posicao;
00033 
00048 
00049 bool posicaoValida(int x, int y, int n, int m, char mapa[MAX][MAX], bool processado[MAX][MAX])
00050 {
00051     if (x >= 0 && x < n && y >= 0 && y < m && mapa[x][y] == '.' && !processado[x][y])
00052         return true;
00053     else
```

```
00054         return false;
00055 }
00056
00071
00072 void visitarComodo(Pilha *pilha, int x0, int y0, int n, int m, char mapa[MAX][MAX], bool
      processado[MAX][MAX])
00073 {
00074     Posicao *inicio = (Posicao *)malloc(sizeof(Posicao));
00075     if (inicio == NULL)
00076     {
00077         printf("Erro de alocacao. \n");
00078         exit(1);
00079     }
00080     inicio->x = x0;
00081     inicio->y = y0;
00082     empilha(pilha, inicio);
00083     processado[x0][y0] = true;
00084
00085     while (!estaVazia(pilha))
00086     {
00087         Posicao *ultima = (Posicao *)desempilha(pilha);
00088         int dir_x[4] = {-1, 1, 0, 0};
00089         int dir_y[4] = {0, 0, -1, 1};
00090
00091         for (int i = 0; i < 4; i++)
00092         {
00093             int vizinhoX = ultima->x + dir_x[i];
00094             int vizinhoY = ultima->y + dir_y[i];
00095             if (posicaoValida(vizinhoX, vizinhoY, n, m, mapa, processado))
00096             {
00097                 Posicao *vizinho = (Posicao *)malloc(sizeof(Posicao));
00098                 vizinho->x = vizinhoX;
00099                 vizinho->y = vizinhoY;
00100                 empilha(pilha, vizinho);
00101                 processado[vizinhoX][vizinhoY] = true;
00102             }
00103         }
00104         free(ultima);
00105     }
00106 }
00107
00119
00120 int contaComodos(int n, int m, char mapa[MAX][MAX])
00121 {
00122     bool processado[MAX][MAX] = {false};
00123     Pilha *pilha = criaPilha();
00124     int totalComodos = 0;
00125
00126     for (int i = 0; i < n; i++)
00127     {
00128         for (int j = 0; j < m; j++)
00129         {
00130             if (mapa[i][j] == '.' && !processado[i][j])
00131             {
00132                 totalComodos++;
00133                 visitarComodo(pilha, i, j, n, m, mapa, processado);
00134             }
00135         }
00136     }
00137     esvaziaPilha(pilha);
00138     free(pilha);
00139     return totalComodos;
00140 }
00141
00150
00151 int main()
00152 {
00153     int n, m;
00154     char mapa[MAX][MAX];
00155     scanf("%d %d", &n, &m);
00156
00157     for (int i = 0; i < n; i++)
00158     {
00159         scanf("%s", mapa[i]);
00160     }
00161     int totalComodos = contaComodos(n, m, mapa);
00162     printf("%d\n", totalComodos);
00163     return 0;
00164 }
```

Índice Remissivo