Sumário

Introdução **1**

Exercício 1 {ex1} **2**

Exercício 2 {ex2} **6**

Exercício 3 {ex3} **7**

Exercício 4 {ex4} **8**

Exercício 5 {ex5} **9**

Exercício 6 {ex6} **10**

Exercício 7 {ex7} **11**

Introdução

Esta lista de exercícios foi proposta pelo Prof. Dr. Igor Machado Coelho, como atividade continuada do tópico Estruturas Lineares na disciplina de Estrutura de Dados e Algoritmos, do Programa de Pós-graduação em Ciência da Computação (PGC) da Universidade Federal Fluminense (UFF), Campus Niterói.

O autor é servidor público no Instituto Federal de Rondônia (IFRO) - *Campus* Vilhena e atua como docente nas disciplinas da área de Engenharia Elétrica, também é aluno, em nível de doutorado, do PGC-UFF e participa do Projeto de Cooperação entre Instituições para Qualificação de Profissionais de Nível Superior (PCI), firmado entre o IFRO e a UFF.

Esta lista de exercícios serviu de oportunidade para que o autor tivesse uma introdução à orientação à objetos e estrutura de projeto utilizando boas práticas de programação. Foi um desafio sair do pensamento de programação estruturada, com o código em apenas um arquivo e passar a separar as funções em arquivos e subpastas. Tanto os arquivos de cabeçalho (.hpp) quanto as implementações (.cpp) foram separados na subpasta "include".

Todos os arquivos de cabeçalho começam com diretivas `#ifndef` e `#define` que evitam referência cíclica, e finalizam com a diretiva `#include`, que referencia o arquivo de implementação. A exemplo do arquivo *dequeEncadeado.hpp*, abaixo:

```
#ifndef _DEQUE_ENCADEADO_HPP_
#define _DEQUE_ENCADEADO_HPP_

// [...]

#include "dequeEncadeado.cpp"
#endif
```

Seguindo as diretrizes da lista de exercícios, este documento foca mais na lógica do que em programação. É uma discussão da proposta do algoritmo, dos aprendizados do percurso e uma análise da complexidade assintótica dos métodos implementados. Porém, todas as implementações foram testadas e aprovados pelo autor, com código disponibilizado no GitHub [**https://github.com/ecostadelle/lista_pilhas_filas**].

A seguir, cada capítulo deste documento apresenta a resposta de um exercício.

Exercício 1 {ex1}

A fim de responder essa questão implementei tanto o deque **sequencial** quanto o **encadeado**. Porém, me limitei a comentar apenas no deque **encadeado** porque foi o que trouxe maior aprendizado. Antes de implementá-lo, a alocação dinâmica era bem nebulosa para mim. Preferi utilizar o tipo genérico, de modo que a implementação pudesse ser reaproveitada em exercícios posteriores, a exemplo das alternativas dessa questão e do Exercício 7, que foi em ordem cronológica o 2º exercício a ser implementado.

Preferi, também, destinar apenas uma pasta para os arquivos Os arquivos de cabeçalho e foram A escolha pelo tipo genérico, associado à estrutura de diretórios

No arquivo de **cabeçalho**, foram definidas as estruturas necessárias para implementar o Deque Encadeado. A primeira classe (`NoDeque`) foi o nó, que armazena um dado do tipo genérico e dois ponteiros que apontam para os nós anterior (`*noAnterior`) e posterior (`*noSeguinte`).

```
template<typename TipoGenerico>
class NoDeque
{
public:
    TipoGenerico dado;
    NoDeque<TipoGenerico> *noSeguinte;
    NoDeque<TipoGenerico> *noAnterior;
};
```

O tipo genérico do dado armazenado precisa ser definido na declaração da variável e é atribuída à classe em tempo de compilação. Inclusive tive muitos problemas quando escolhi esse método, porque eu estava compilando os objetos separados e vinculando-os posteriormente.

Na segunda classe (`Deque`), foram declarados em modo privado os ponteiros que apontam apenas para dois nós, o inicial e o final. As interfaces padrão foram declaradas em modo público, de maneira que permitiam o acesso externo a classe.

```
template <typename TipoGenerico>
class Deque
{
private:
    NoDeque<TipoGenerico> *inicio;
    NoDeque<TipoGenerico> *fim;
```

```
7       int numeroElementos;
8
9   public:
10      Deque();
11      ~Deque();
12      void insereInicio(TipoGenerico v);
13      void insereFim(TipoGenerico v);
14      int tamanho();
15      TipoGenerico buscaInicio();
16      TipoGenerico buscaFim();
17      TipoGenerico removeInicio();
18      TipoGenerico removeFim();
19  };
```

Os métodos tiveram seu próprio arquivo de **implementação (dequeEncadeado.cpp)**. Além dos métodos solicitados através do `staticassert`, foram implementados um construtor `Deque()`, que inicia as variáveis em tempo constante $O(1)$, e um destrutor `~Deque()`, que ao se invocado, percorre todos os elementos do deque, liberando a memória e evitando "vazamento de memória". A complexidade do método destrutor é linearmente depende do tamanho armazenado em `numeroElementos` (n), ou seja, $O(n)$

```
1   #include "dequeEncadeado.hpp"
2
3   template <typename TipoGenerico>
4   Deque<TipoGenerico>::Deque()
5   {
6       this->numeroElementos = 0;
7       this->inicio = nullptr;
8       this->fim = nullptr;
9   }
10
11  template <typename TipoGenerico>
12  Deque<TipoGenerico>::~Deque()
13  {
14      while (this->numeroElementos > 0)
15          removeInicio();
16  }
```

No método `insereInicio` um nó é criado e alocado dinamicamente na memória, um ponteiro (`*no`) armazena o endereço e um dado é inserido. Caso o deque esteja vazio, os endereços de início e fim serão os mesmo do nó recém criado. Caso contrário, o campo `noSeguinte` do nó recém criado recebe o endereço daquele que era o início, e o campo `noAnterior`, daquele que era o inicio, recebe o endereço do novo nó e o número de elementos no deque é incrementado. Este método opera em tempo constante, ou seja, $O(1)$.

```
1   template <typename TipoGenerico>
2   void Deque<TipoGenerico>::insereInicio(TipoGenerico v)
```

```
3  {
4      NoDeque<TipoGenerico> *no =
5          new NoDeque
6          {.dado = v,
7           .noSeguinte = nullptr,
8           .noAnterior = nullptr};
9      if (numeroElementos == 0)
10     {
11         inicio = fim = no;
12     }
13     else
14     {
15         no->noSeguinte = inicio;
16         inicio->noAnterior = no;
17         inicio = no;
18     }
19     numeroElementos++;
20 }
21
22
23
24 template <typename TipoGenerico>
25 void Deque<TipoGenerico>::insereFim(TipoGenerico v)
26 {
27     NoDeque<TipoGenerico> *no =
28         new NoDeque<TipoGenerico>
29         {.dado = v,
30          .noSeguinte = nullptr,
31          .noAnterior = nullptr};
32     if (numeroElementos == 0)
33     {
34         inicio = fim = no;
35     }
36     else
37     {
38         no->noAnterior = fim;
39         fim->noSeguinte = no;
40         fim = no;
41     }
42     numeroElementos++;
43 }
44
45 template <typename TipoGenerico>
46 int Deque<TipoGenerico>::tamanho()
47 {
48     return this->numeroElementos;
49 }
50
51 template <typename TipoGenerico>
52 TipoGenerico Deque<TipoGenerico>::buscaInicio()
53 {
```

```
54      return inicio->dado;
55  }
56
57  template <typename TipoGenerico>
58  TipoGenerico Deque<TipoGenerico>::buscaFim()
59  {
60      return fim->dado;
61  }
62
63  template <typename TipoGenerico>
64  TipoGenerico Deque<TipoGenerico>::removeInicio()
65  {
66      NoDeque<TipoGenerico> *p = inicio;
67      inicio = inicio->noSeguinte;
68      TipoGenerico r = p->dado;
69      delete p;
70      numeroElementos--;
71      return r;
72  }
73
74  template <typename TipoGenerico>
75  TipoGenerico Deque<TipoGenerico>::removeFim()
76  {
77      NoDeque<TipoGenerico> *p = fim;
78      fim = fim->noAnterior;
79      TipoGenerico r = p->dado;
80      delete p;
81      numeroElementos--;
82      return r;
83  }
84
85  template <typename Agregado, typename Tipo>
86  concept DequeTAD = requires(Agregado a, Tipo t)
87  {
88      // requer operação de consulta ao elemento 'inicio'
89      {a.buscaInicio()};
90      // requer operação de consulta ao elemento 'fim'
91      {a.buscaFim()};
92      // requer operação 'insereInicio' sobre tipo 't'
93      {a.insereInicio(t)};
94      // requer operação 'insereFim' sobre tipo 't'
95      {a.insereFim(t)};
96      // requer operação 'removeInicio' e retorna tipo 't'
97      {a.removeInicio()};
98      // requer operação 'removeFim' e retorna tipo 't'
99      {a.removeFim()};
100 };
101 // testa se Deque está correto
102 static_assert(DequeTAD<Deque<char>, char>);
```

Exercício 2 {ex2}

Exercício 3 {ex3}

Exercício 4 {ex4}

Exercício 5 {ex5}

Exercício 6 {ex6}

Exercício 7 {ex7}

Para resolver este exercício, uma pilha armazenou os operadores e um vetor, a saída.

De modo que um laço percorre a entrada e armazena letras (variáveis) diretamente na saída e sinais (operações) na pilha até encontrar: 1. um parêntese fechado; ou 2. um operador de precedência inferior ao último encontrado.

Assim que uma das duas condições é satisfeita, duas coisas podem ocorrer, respectivamente: 1. o último operador é desempilhado, inserido no vetor de saída, e o um parêntese aberto também é desempilhado; 2. o operador de maior precedência é desempilhado, inserido na saída e o novo operador é empilhado.

Ao chegar ao final do laço, todos os operadores são desempilhados e inseridos na saída.

Com o objetivo de determinar as precedências, utilizou-se o código ASCII do caractere subtraído de 41 em módulo 6. Esta transformação faz com que o caractere '*' (decimal 42), torne-se 1; '+' (decimal 43), torne-se 2; '-' (decimal 45), torne-se 4; e, por fim, o caractere '/' (decimal 47), torne-se 0.