Sumário

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

a) Implementação de um Double Ended Queue (DEQUE) encadeado

A fim de responder essa questão implementei tanto o deque **sequencial** (as ligações para os arquivos no repositório aparecem em negrito) 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
{
```

```
4  private:
5      NoDeque<TipoGenerico> *inicio;
6      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 `concept`, 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 evita o "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, $O(1)$.

```
template <typename TipoGenerico>
void Deque<TipoGenerico>::insereInicio(TipoGenerico v)
{
    NoDeque<TipoGenerico> *no =
        new NoDeque
        {.dado = v,
         .noSeguinte = nullptr,
         .noAnterior = nullptr};
    if (numeroElementos == 0)
    {
        inicio = fim = no;
    }
    else
    {
        no->noSeguinte = inicio;
        inicio->noAnterior = no;
        inicio = no;
    }
    numeroElementos++;
}
```

No método `insereFim` ocorre um procedimento semelhante ao descrito anteriormente, porém na outra ponta. E opera, também, em tempo constante, ou seja, $O(1)$.

```
template <typename TipoGenerico>
void Deque<TipoGenerico>::insereFim(TipoGenerico v)
{
    NoDeque<TipoGenerico> *no =
        new NoDeque<TipoGenerico>
        {.dado = v,
         .noSeguinte = nullptr,
         .noAnterior = nullptr};
    if (numeroElementos == 0)
    {
        inicio = fim = no;
    }
    else
    {
        no->noAnterior = fim;
        fim->noSeguinte = no;
        fim = no;
    }
    numeroElementos++;
}
```

Já os métodos `tamanho`, `buscaInicio` e `buscaFim` apenas retornam variáveis que estão privadas, é um modo seguro de se criar uma interface. Operam em tempo constante ($O(1)$).

```
template <typename TipoGenerico>
int Deque<TipoGenerico>::tamanho()
```

```
3  {
4      return this->numeroElementos;
5  }
6
7  template <typename TipoGenerico>
8  TipoGenerico Deque<TipoGenerico>::buscaInicio()
9  {
10     return inicio->dado;
11 }
12
13 template <typename TipoGenerico>
14 TipoGenerico Deque<TipoGenerico>::buscaFim()
15 {
16     return fim->dado;
17 }
```

Os métodos `removeInicio` e `removeFim` operam em tempo constante ($O(1)$) e quando um deles é invocado, ele armazena o endereço do nó que será removido, coleta o dado armazenado, remove o nó, decrementa o número de elementos, define a nova ponta da fila (seja no início, seja no fim) e retorna o valor lido.

```
1  template <typename TipoGenerico>
2  TipoGenerico Deque<TipoGenerico>::removeInicio()
3  {
4      NoDeque<TipoGenerico> *p = inicio;
5      TipoGenerico r = p->dado;
6      delete p;
7      numeroElementos--;
8      inicio = inicio->noSeguinte;
9      return r;
10 }
11
12 template <typename TipoGenerico>
13 TipoGenerico Deque<TipoGenerico>::removeFim()
14 {
15     NoDeque<TipoGenerico> *p = fim;
16     TipoGenerico r = p->dado;
17     delete p;
18     numeroElementos--;
19     fim = fim->noAnterior;
20     return r;
21 }
```

Conforme solicitado, o `static_assert` verifica os métodos do `concept`, algumas pequenas alterações foram feitas: a palavra `bool` foi removida porque os código foi implementado em C++20, uma vez que o IntelliSense reclamou dos métodos em questão enquanto era utilizado o C++17; a segunda alteração foi o acréscimo da palavra “busca” nos métodos que realizavam essa ação, com o objetivo de melhorar a legibilidade do código. Abaixo estão os códigos do `concept` e do `static_assert`

```
template <typename Agregado, typename Tipo>
concept DequeTAD = requires(Agregado a, Tipo t)
{
    // requer operação de consulta ao elemento 'inicio'
    {a.buscaInicio()};
    // requer operação de consulta ao elemento 'fim'
    {a.buscaFim()};
    // requer operação 'insereInicio' sobre tipo 't'
    {a.insereInicio(t)};
    // requer operação 'insereFim' sobre tipo 't'
    {a.insereFim(t)};
    // requer operação 'removeInicio' e retorna tipo 't'
    {a.removeInicio()};
    // requer operação 'removeFim' e retorna tipo 't'
    {a.removeFim()};
};
// testa se Deque está correto
static_assert(DequeTAD<Deque<char>, char>);
```

b) Implementação de uma pilha utilizando um DEQUE

No arquivo de cabeçalho **pilhaDeque.hpp**, foram declarados, em uma classe, uma variável com o Tipo Abstrato de Dados (TAD) Deque (definindo o tipogenérico como `char`) e os protótipos das interfaces padrão do TAD pilhaDeque. Preferiu-se utilizar métodos com nomes semelhantes aos disponíveis na Biblioteca de Modelos Padrão (STL - *standard template library*) do C++.

```
#ifndef _PILHA_DEQUE_HPP_
#define _PILHA_DEQUE_HPP_

#include "dequeEncadeado.hpp"

class PilhaDeque
{
public:
    Deque<char> d;

    PilhaDeque();
    ~PilhaDeque();
    bool empty();
    char top();
    void push(char t);
    char pop();
};

#include "pilhaDeque.cpp"

#endif
```

No arquivo de implementação **pilhaDeque.cpp**, foram declarados os métodos utilizando as interfaces padrão do TAD `Deque`, limitando-se apenas em operar em uma das pontas. Com exceção do método destrutor, que opera em tempo linearmente dependente do número de elementos, $O(n)$, todos os demais métodos operam em tempo constante $O(1)$.

```cpp
#include <iostream>
#include "pilhaDeque.hpp"

PilhaDeque::PilhaDeque()
{
}

PilhaDeque::~PilhaDeque()
{
    d.~Deque();
}

bool PilhaDeque::empty()
{
    return (d.tamanho() == 0);
}

char PilhaDeque::top()
{
    return d.buscaFim();
}

void PilhaDeque::push(char t)
{
    d.insereFim(t);
}

char PilhaDeque::pop()
{
    return d.removeFim();
}

template <typename Agregado, typename Tipo>
concept PilhaTAD = requires(Agregado c, Tipo t)
{
    // requer operação de consulta ao elemento 'fim'
    {c.top()};
    // requer operação 'insereFim' sobre tipo 't'
    {c.push(t)};
    // requer operação 'removeFim' e retorna tipo 't'
    {c.pop()};
};
// testa se Pilha está correta
```

```
44 static_assert(PilhaTAD<PilhaDeque, char>);
```

c) Implementação de uma pilha utilizando um DEQUE

No arquivo de cabeçalho **filaDeque.hpp**, foram declarados, em uma classe, uma variável com o Tipo Abstrato de Dados (TAD) Deque (definindo o tipogenérico como `char`) e os protótipos das interfaces padrão do TAD `filaDeque`. Assim como no exercício anterior, preferiu-se utilizar métodos com nomes semelhantes aos disponíveis na Biblioteca de Modelos Padrão (STL - *standard template library*) do C++.

```
1  #ifndef _FILA_DEQUE_HPP_
2  #define _FILA_DEQUE_HPP_
3
4  #include "dequeEncadeado.hpp"
5
6  class FilaDeque
7  {
8  public:
9      Deque<char> d;
10
11     FilaDeque();
12     ~FilaDeque();
13     bool empty();
14     char front();
15     void push(char t);
16     char pop();
17 };
18
19 #include "FilaDeque.cpp"
20
21 #endif
```

No arquivo de implementação **filaDeque.cpp**, foram declarados os métodos utilizando as interfaces padrão do TAD `Deque`, limitando-se apenas em inserir dados em uma das pontas e remover na outra. Com exceção do método destrutor, que opera em tempo linearmente dependente do número de elementos, $O(n)$, todos os demais métodos operam em tempo constante $O(1)$.

```
1  #include <iostream>
2  #include "filaDeque.hpp"
3
4  FilaDeque::FilaDeque()
5  {
6  }
7
8  FilaDeque::~FilaDeque()
9  {
```

```
10      d.~Deque();
11  }
12
13  bool FilaDeque::empty()
14  {
15      return (d.tamanho() == 0);
16  }
17
18  char FilaDeque::front()
19  {
20      return d.buscaInicio();
21  }
22
23  void FilaDeque::push(char t)
24  {
25      d.insereFim(t);
26  }
27
28  char FilaDeque::pop()
29  {
30      return d.removeInicio();
31  }
32
33  template <typename Agregado, typename Tipo>
34  concept FilaTAD = requires(Agregado c, Tipo t)
35  {
36      // requer operação de consulta ao elemento 'inicio'
37      {c.front()};
38      // requer operação 'insereFim' sobre tipo 't'
39      {c.push(t)};
40      // requer operação 'removeInicio' e retorna tipo 't'
41      {c.pop()};
42  };
43  // testa se Fila está correta
44  static_assert(FilaTAD<FilaDeque, char>);
```

Exercício 2

Implementar uma Pilha que utiliza duas Filas como armazenamento interno

No arquivo de cabeçalho **pilha2F.cpp**, foi declarada uma fila genérica do STL, para armazenar o tipo `char` e os protótipos da interface padrão do TAD. Seguindo o que vem sendo aplicado nos exercícios anteriores, utilizou-se métodos com nomes semelhantes aos disponíveis na STL.

```cpp
#ifndef _PILHA_2F_HPP_
#define _PILHA_2F_HPP_

#include <queue> // Fila genérica em C++

class Pilha2F {
public:
    std::queue<char> f1; // Fila para 'char'
    std::queue<char> f2; // Fila para 'char'
    // SOMENTE espaço auxiliar CONSTANTE aqui
    // (nenhum vetor, lista, etc)
    // implementar métodos do TAD Pilha
    Pilha2F();
    ~Pilha2F();
    bool empty();
    char top();
    void push(char t);
    char pop();
};

#include "pilha2F.cpp"

#endif
```

No arquivo de implementação **pilha2F.cpp**, foram efetivados os métodos que permitiram a operação solicitada. No método construtor, foram inicializadas as variáveis da classe `Pilha2F`, este método opera em tempo constante ($O(1)$).

```cpp
Pilha2F::Pilha2F()
{
    // Inicialização das filas
```

```
4       f1 = std::queue<char>{};
5       f2 = std::queue<char>{};
6   }
```

Já o método destrutor, abaixo, percorre todos os elementos da pilha e os remove. Operando em tempo linearmente dependente do número de elementos armazenado na pilha (n), ou seja, opera em $O(n)$.

```
1   Pilha2F::~Pilha2F()
2   {
3       while (this->empty() == false)
4       {
5           this->pop();
6       }
7   }
```

O método `push` é o mais importante desta implementação, é nele que é executado a o algoritmo que permitiu a operação solicitada. Quando o usuário solicita um `push`, todos os elementos da primeira fila (`f1`) são removidos para a segunda (`f2`), o elemento é inserido na `f1` e todos os elementos da `f2` retornam para a `f1`. Deste modo, a cada nova inserção são necessárias operações em dobro se comparado com o número de elementos (n), mesmo sendo $O(2n)$, assintoticamente opera em tempo $O(n)$.

```
1   void Pilha2F::push(char t)
2   {
3       while (!f1.empty())
4       {
5           f2.push(f1.front());
6           f1.pop();
7       }
8
9       f1.push(t);
10
11      while (!f2.empty())
12      {
13          f1.push(f2.front());
14          f2.pop();
15      }
16  }
```

Manter o último elemento inserido na pilha na primeira posição da `f1`, favorece os métodos `top` e `pop` que operarão sempre no elemento que está na frente da `f1`. Operando sempre em tempo constante, $O(1)$.

```
1   char Pilha2F::top()
2   {
3       return f1.front();
4   }
5
```

```
6  char Pilha2F::pop()
7  {
8      char t = f1.front();
9      f1.pop();
10     return t;
11 }
```

O método `empty` verifica se ambas filas estão vazias, apesar de, em tese, a `f2` permanecer vazia a maior parte do tempo. Opera em tempo constante, $O(1)$.

```
1  bool Pilha2F::empty()
2  {
3      return f1.empty() && f2.empty();
4  }
```

O `static_assert` verifica se o os métodos solicitados pelo `concept` foram satisfeitos.

```
1
2  template <typename Agregado, typename Tipo>
3  concept PilhaTAD2F = requires(Agregado a, Tipo t)
4  {
5      {a.empty()};
6      {a.top()};
7      {a.push(t)};
8      {a.pop()};
9  };
10 // testa se Pilha está correta
11 static_assert(PilhaTAD2F<Pilha2F, char>);
```

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.