**INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO**
**CURSO:** TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
**DISCIPLINA:** ALGORITMOS E ESTRUTURAS DE DADOS
**PROFESSOR:** RAMIDE DANTAS
**ASSUNTO:** PILHAS, FILAS E LISTAS

# Prática 04

## Parte 0: Preparação

Passo 1: Crie um projeto (ex.: Pratica-04) e adicione os arquivos que acompanham a prática.

> **ATENÇÃO:** A prática contém vários arquivos de teste. Você NÃO deve precisar modificar os testes nesse arquivo. Mesmo que os testes terminem com "OK" não significa que sua implementação esteja 100% correta.

Passo 2: Estude o arquivo **main.cpp**:

> Esse arquivo facilita a criação de um projeto só para a prática, permitindo escolher qual função *main()* rodar (basta comentar/descomentar).

## Parte 1: Trabalhando com pilhas

Passo 1: Estude os arquivos **pilha_teste.cpp** e **polonesa.cpp**.

- **pilha_teste.cpp:** realiza um conjunto de testes para validar o funcionamento da pilha.
- **polonesa.cpp**: implementa uma calculadora polonesa simples baseada em pilha.

Passo 2: Crie um arquivo chamado **pilha.h** e implemente nele a classe `Pilha` como a seguir:

```
template <class T>
class Pilha {
private:
       // Atributos para array de items, capacidade e topo da pilha
public:
       Pilha(int capacidade) {
         // inicializa capacidade e topo, instancia array de items
       }
       ~Pilha() {
         // destrua o array de items
       }
       void empilha(T item) {
         // empilha um item no topo da pilha;
         // lança std::runtime_error("Estouro da pilha") se cheia
       }
       T desempilha() {
         // remove um item do topo da pilha;
         // lança std::runtime_error("Pilha vazia") se vazia
       }
       int tamanho() {
         // retorna o número de elementos na pilha.
       }
};
```

> Dê implementações adequadas às funções acima, conforme os comentários.

Passo 3: Compile e teste a aplicação, verificando se o resultado é o esperado.

> Rode o **pilha_teste.cpp** (`mainPilha()` em **main.cpp**) primeiro para verificar sua implementação da Pilha. Depois teste o **polonesa.cpp** (`mainPol()`) para ver se o resultado gerado é o esperado.

Passo 4: **(Desafio/Opcional)** Tente usar a classe `std::stack` da **STL** (`<stack>`) no **polonesa.cpp** no lugar da sua pilha e veja se os resultados são os mesmos.

**Parte 2: Trabalhando com Filas**

Passo 1: Estude os arquivo **fila_teste.cpp** e **impressora.cpp**.

- **fila_teste.cpp:** realiza um conjunto de testes para validar o funcionamento da fila.
- **impressora.cpp**: simula uma fila de impressão: o usuário submete documentos que aguardam até que a impressora os imprima. A fila é um *buffer* que permite ao usuário e impressora trabalharem de forma paralela. (**OBS.:** Configure o projeto para usar C++11.)

Passo 2: Crie o arquivo **fila.h** e implemente a classe `Fila` conforme a declaração a seguir:

> **IMPORTANTE:** A fila deve usar o conceito de "*buffer circular*" visto em aula, usando resto da divisão (%) conforme explicado no material de aula.

```
template <class T>
class Fila {
private:
       // array de itens, capacidade, tamanho, posição inicial, etc.
public:
       Fila(int cap) {
         // inicializar array de items, capacidade, tamanho, posição inicial
       }

       ~Fila() {
         // destruir array de itens
       }

       void enfileira(const T & item) {
         // lança std::runtime_error("Fila cheia") caso cheia
         // adiciona um item ao final da fila usando buffer circular
       }

       T desenfileira() {
         // lança std::runtime_error("Fila vazia") caso vazia
         // remove um item do inicio da fila usando buffer circular
       }

       int cheia() {
         // retorna 1 se cheia, 0 caso contrário
       }

       int vazia() {
         // retorna 1 se vazia, 0 caso contrário
       }

       int tamanho() {
         // retorna a quantidade de itens atualmente na fila
       }
};
```

Passo 3: Compile e teste a aplicação, verificando se o resultado é o esperado.

> **fila_teste.cpp** (`mainFila()` em **main.cpp**)**:** deve exibir OK para todos os testes.
>
> **impressora.cpp** (`mainImp()` em **main.cpp**): neste código há um laço infinito: a cada iteração o usuário tem uma probabilidade de 70% de adicionar um novo documento à fila; já a impressora tem 30% de chance de imprimir da fila. Haverá um momento em que a fila estará cheia e o usuário não consegue adicionar mais documentos (só depois que a impressora retira um item). Modifique as probabilidades de forma que o fila fique quase sempre vazia.

Passo 4: **(Desafio/Opcional)** Mude **impressora.cpp** para usar `std::queue` da **STL** (`<queue>`).

**Parte 3: Trabalhando com Listas**

Passo 1: Estude os arquivo **lista_teste.cpp** e **lista_char.cpp**.

- **lista_teste.cpp**: realiza testes simples para validar o funcionamento da lista.
- **lista_char.cpp**: exemplo de uso da lista com caracteres.

Passo 2: Crie o arquivo **lista.h** e implemente a classe `Lista` conforme a declaração a seguir:

```
template <class T>
class Lista {

private:
      // atributos:
      //  -  itens da lista (ponteiro para T usado como array)
      //  -  capacidade e tamanho atual (inteiros)

public:
       Lista(int capacidade) {
         // inicilização do array de itens, capacidade e tamanho
       }

       ~Lista() {
         //destruição do array
       }

       // adiciona um item ao final da lista
       void adiciona (const T & item) {
         // lança std::runtime_error("Lista cheia") caso capacidade esgotada
       }

       // retorna um item pelo indice (começa em 1);
       T pega(int idx) {
         // lança std::runtime_error("Indice inválido") se posição inválida
         // ATENÇÃO: posições válidas são de 1 a N (= capacidade).
       }

       // insere um item na posição indicada (a partir de 1).
       void insere (int idx, const T & item) {
         // lança std::runtime_error("Lista cheia") caso cheia
         // lança std::runtime_error("Indice invalido") se posição inválida
         // se a lista contém N itens, só é possivel inserir até a posição N
         // deve deslocar itens existentes uma posição para a direita
       }

       // remove o item de uma posição indicada (a partir de 1)
       void remove(int idx) {
         // lança std::runtime_error("Indice invalido") se posição inválida
         // desloca itens uma posicao a esquerda sobre o item removido
       }

       void exibe() {
         // exibe os itens da saida padrão separados por espaços
       }

       int tamanho() {
         // retorna a quantidade de itens atualmente na lista
       }
};
```

**ATENÇÃO:**

- Do ponto de vista do usuário da lista (isto é, em **lista_teste.cpp** e **lista_char.cpp**), ela deve ser indexada a partir de 1, e não de 0. Isto é, numa lista com 10 itens, os índices válidos dos elementos vão de 1 a 10. A implementação interna usando *arrays* deve evitar desperdício de memória (*array* maior que o necessário) e acesso a índices fora dos limites do *array*.
- Os métodos `insere()` e `remove()` precisam deslocar os itens existentes no *array* para direita (`insere()`) ou para a esquerda (`remove()`), dependendo da situação.

Passo 3: Compile e teste a aplicação, verificando o resultado (ambos os arquivos).

**lista_teste.cpp:** (`mainLista()` em **main.cpp**) deve exibir OK para todos os testes.

Em **lista_char.cpp** (`mainChar()` em **main.cpp**), teste modificações nas adições, inserções e remoções.

Passo 4: **(Desafio/opcional)** Modifique **lista_char.cpp** para usar `std::list` da **STL** (`<list>`).