

IMD0030

LINGUAGEM DE PROGRAMAÇÃO I

Aula 21 – Implementação de Tipos Abstratos de Dados (TADs) genéricos
(material baseado nas notas de aula do Prof. Silvio Sampaio)

Tipos Abstratos de Dados (TADs)

- Objetivo desta aula: apresentar a implementação de TADs genéricos utilizando *template* de classe em C++
 - Para isso, revisaremos:
 - Os conceitos de Tipos Abstratos de Dados (TAD) e a TAD Pilha
 - Como implementar *templates* de classes na linguagem de programação C++
 - Ao final da aula, espera-se que o aluno seja capaz de:
 - Implementar TADs genéricas (capaz de manipular qualquer tipo de dado) usando *templates* de classes na linguagem de programação C++
-

Tipos Abstratos de Dados (TADs)

- Variação da implementação
 - Há diferentes implementações possíveis para o mesmo tipo de dado
 - Todas definem o mesmo domínio e não mudam o significado das operações
 - Em programas reais, as implementações dos tipos de dados (modo como os dados são representados) são modificadas constantemente para se tornarem mais velozes, eficientes, claras, etc.
 - Essas mudanças têm grande impacto nos programas usuários do tipo de dado
 - Como podemos modificar as implementações dos tipos de dados com o menor impacto possível para os programas?
 - Como podemos encapsular (esconder) de quem usa um determinado tipo de dado a forma concreta como este tipo foi implementado?
 - **Resposta: TIPOS ABSTRATOS DE DADOS (TADs)**
-

Tipos Abstratos de Dados (TADs)

- Um TAD especifica o tipo de dado (domínio e operações) sem referência a detalhes da implementação
 - Minimiza código do programa que usa detalhes de implementação
 - Dá mais liberdade para mudar implementação com menor impacto nos programas
 - Minimiza custos
 - Os programas que usam o TAD não “conhecem” as implementações dos TADs
 - Apenas fazem uso do TAD através de suas operações (disponibilizadas através de sua interface)
 - Em resumo, um TAD especifica tudo que se precisa saber para usar um determinado tipo de dado
 - Não faz referência à maneira com a qual o tipo de dado será (ou é) implementado
 - Com o uso de TADs, os sistemas ficam divididos em:
 - Programas usuários: a parte que usa o TAD
 - Implementação: a parte que implementa o TAD
-

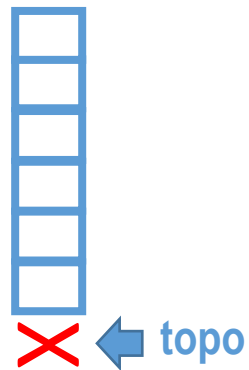
POO e TADs

- Abstração
 - Quando definimos um TAD, nos concentramos nos aspectos essenciais do tipo de dado (operações) e abstraímos a forma como cada operação será implementada
- Encapsulamento
 - O TAD provê um mecanismo de encapsulamento de um tipo de dado, permitindo separar a especificação (aspecto externo) de sua implementação (aspecto interno)

TAD Pilha

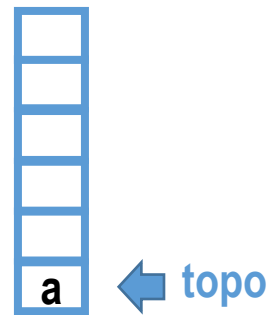
Operações básicas

Pilha p(6);



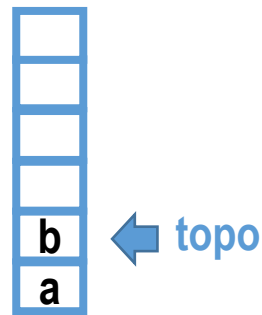
p.size() = 0
p.empty() = true
p.full() = false

p.push('a')



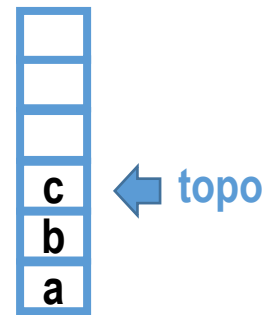
p.size() = 1
p.empty() = false
p.full() = false

p.push('b')



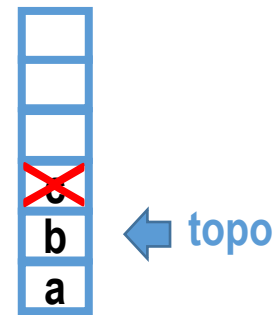
p.size() = 2
p.empty() = false
p.full() = false

p.push('c')

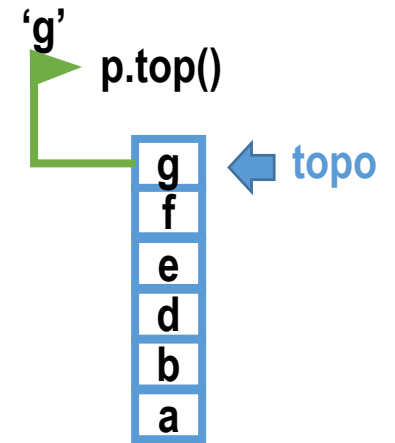


p.size() = 3
p.empty() = false
p.full() = false

p.pop()



p.size() = 2
p.empty() = false
p.full() = false



p.size() = 6
p.empty() = false
p.full() = true

TAD Pilha

Definição

```
#ifndef Pilha_H
#define Pilha_H

#include <iostream>

template <typename T>
class Pilha {
private:
    T* m_elementos;    // Elementos armazenados na pilha
    int m_tamanho;     // Qtde atual de elementos
    int m_capacidade;  // Qtde Max de elementos
public:
    Pilha (int n_capacidade = 50);
    ~Pilha ();
    bool empty ();
    bool full ();
    T& top ();
    int push ( T novo );
    int pop ();
    int size();
};
```

TAD Pilha

Construtor e Destrutor

```
template <typename T>
Pilha<T>::Pilha (int n_capacidade) :
m_tamanho(0), m_capacidade(n_capacidade)
{
    m_elementos = new T[n_capacidade];
}
```

```
template <typename T>
Pilha<T>::~~Pilha ()
{
    delete [] m_elementos;
}
```

TAD Pilha

Métodos de teste: pilha vazia? | pilha cheia?

```
template <typename T>
bool Pilha<T>::empty ()
{
    return m_tamanho == 0;
}

template <typename T>
bool Pilha<T>::full ()
{
    return m_tamanho == m_capacidade;
}
```

TAD Pilha

Operações básicas de uma pilha: push (empilha) | pop (desempilha)

```
template <typename T>
int Pilha<T>::push ( T novo )
{
    if (full())
        return 0;
    m_elementos[m_tamanho++] =
novo;
    return 1;
}
```

```
template <typename T>
int Pilha<T>::pop ()
{
    if (empty())
        return 0;
    m_tamanho--;
    return 1;
}
```

TAD Pilha

Operações básicas: top (elemento no topo)

```
template <typename T>
T& Pilha<T>::top ()
{
    if (empty()) {
        std::cerr << "Acesso invalido a elemento no topo.
            O programa sera fechado!" << std::endl;
        exit(EXIT_FAILURE);
    }
    return m_elementos[m_tamanho-1];
}
```

TAD Pilha

Operações básicas: size (tamanho atual)

```
template <typename T>
int Pilha<T>::size ()
{
    return m_tamanho;
}

#endif
```

TAD Pilha

Utilizando a TAD Pilha

```
#include "pilha.h"
```

```
[...]
```

```
Pilha<int> pilha;
```

```
cout << "Tamanho: " << pilha.size() << endl;
```

```
pilha.push(10);
```

```
pilha.push(15);
```

```
pilha.push(20);
```

```
cout << pilha.top() << endl;
```

```
pilha.pop();
```

```
cout << "Tamanho: " << pilha.size() << endl;
```

```
cout << pilha.top() << endl;
```

```
pilha.pop();
```

```
cout << pilha.top() << endl;
```

```
pilha.pop();
```

```
[...]
```

TAD Pilha

Utilizando a TAD Pilha

```
[...]  
Pilha<string> pilha4;  
pilha4.push("Ana");  
pilha4.push("Maria");  
pilha4.push("Joao");  
cout << pilha4.top() << endl;  
pilha4.pop();  
cout << pilha4.top() << endl;  
pilha4.pop();  
cout << pilha4.top() << endl;  
pilha4.pop();  
[...]
```

Generalizando em uma Lista Ordenada

- Uma lista linear representa uma coleção ordenada de elementos de um mesmo tipo
 - A palavra “ordenada” implica que, dado um elemento da coleção, podemos identificar seu sucessor e seu predecessor
 - Uso de sentinelas: os primeiro e último elementos da lista são considerados elementos especiais, pois não estão definidos o predecessor do primeiro elemento nem o sucessor do último elemento
 - Juntamente com esta definição de lista, pode-se ter um conjunto de operações a serem implementadas por uma lista:
 - Criar a lista
 - Inserir elemento
 - Remover elemento
 - Procurar elemento
 - Listar os elementos presentes na lista
 - Tamanho atual da lista
-

Generalizando em uma Lista Ordenada

- Restrições nas operações determinam diferentes tipos de lista (pilhas, filas, etc.)
 - Há dois tipos de implementações possíveis de listas lineares:
 - Sequencial (armazena itens em posições contíguas de memória - uso de vetores)
 - Encadeada (utiliza posições não contíguas de memória para armazenar itens - uso de ponteiros)
 - Listas são úteis em aplicações diversas tais como manipulação simbólica, gerência de memória, simulação e compiladores
-