Estruturas de Dados I

Pilhas

Igor Machado Coelho

16/09/2020 - 13/04/2023

1. Pilhas
2. Tipo Abstrato: Pilha
3. Pilhas Sequenciais
4. Pilhas Encadeadas
5. Implementação Avançada com Ponteiros Inteligentes
6. Pilhas Genéricas e Conceitos de Pilha
7. Pilhas na Biblioteca Padrão e Aplicações
8. Análise de Complexidade
9. Agradecimentos

Section 1

Pilhas

Pré-Requisitos

São requisitos para essa aula o conhecimento de:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- Interesse em aprender C/C++
- Noções de tipos de dados
- Noções de listas e encadeamento

Section 2

Tipo Abstrato: Pilha

Pilha

A Pilha (do inglês *Stack*) é um Tipo Abstrato de Dado (TAD) que pode ser compreendida como vemos no cotidiano.

Em uma *pilha de pratos*, por exemplo:

- Só se consegue "inserir" (empilhar) novos pratos no *topo* da pilha
- Só podemos "remover" (desempilhar) os pratos do *topo* da pilha



Figure 1: Pilhas

Pilhas na computação

Pilhas são estruturas fundamentais na própria computação.

Por exemplo, as chamadas de uma função recursiva podem ser feitas utilizando uma pilha!

. . . e é precisamente desta maneira que o sistema operacional consegue executar várias de suas funções internas!

Linguagens de programação como Java, C# e Python são implementadas através de operações em pilhas.

Operações de uma Pilha

Uma Pilha é uma estrutura de dados linear (assim como estruturas de lista), consistindo de 3 operações básicas:

- topo
- empilhar (*push*)
- desempilhar (*pop*)

Seu comportamento é descrito como LIFO (last-in first-out), ou seja, o *último* elemento a entrar na pilha será o *primeiro* a sair.

Implementações

De forma geral, uma pilha pode ser implementada utilizando uma lista linear, porém com acesso aos elementos restritos a uma única extremidade dessa lista.

$$\rightleftarrows \mid C \mid B \mid A \mid$$

Em C/C++, os métodos esperados para uma pilha de tipo `t` são: `topo()`, `empilha(t)`, `desempilha()`, `tamanho()`.

Section 3

Pilhas Sequenciais

Pilhas Sequenciais

As Pilhas Sequenciais utilizam um array para armazenar os dados. Assim, os dados sempre estarão em um *espaço contíguo* de memória.

Implementação

Consideraremos uma pilha sequencial com, no máximo, `MAX_N` elementos no vetor `v` do tipo caractere.

```
constexpr int MAX_N = 100'000; // capacidade máxima da pilha
struct PilhaSeq1 {
  char v[MAX_N];              // elementos na pilha
  int N;                      // num. de elementos na pilha
  void cria();                // inicializa agregado
  void libera();              // finaliza agregado
  char topo();
  void empilha(char dado);
  char desempilha();
  int tamanho();
};
```

Utilização da Pilha

Antes de completar as funções pendentes, utilizaremos a `PilhaSeq1`:

```
import std;
int main() {
    PilhaSeq1 p;
    p.cria();          // nosso contrato no curso!
    p.empilha('A');  p.empilha('B');  p.empilha('C');
    println("{}", p.topo());
    println("{}", p.desempilha());
    p.empilha('D');
    while(p.tamanho() > 0)  println("{}", p.desempilha());
    p.libera();      // nosso contrato no curso!
    return 0;
}
```

Verifique as impressões em tela:

Utilização da Pilha

Antes de completar as funções pendentes, utilizaremos a `PilhaSeq1`:

```
import std;
int main() {
    PilhaSeq1 p;
    p.cria();          // nosso contrato no curso!
    p.empilha('A');  p.empilha('B');  p.empilha('C');
    println("{}", p.topo());
    println("{}", p.desempilha());
    p.empilha('D');
    while(p.tamanho() > 0)  println("{}", p.desempilha());
    p.libera();      // nosso contrato no curso!
    return 0;
}
```

Verifique as impressões em tela:

C C D B A

Implementação: Cria e Libera

A operação `cria` inicializa a pilha para uso, e a função `libera` desaloca os recursos dinâmicos.

```
void PilhaSeq1::cria() {
    N = 0;      // this->N = 0;
}

void PilhaSeq1::libera() {
    // nenhum recurso dinâmico para desalocar
}
```

Implementação: Empilha / Desempilha

A operação `empilha` em uma pilha sequencial adiciona um novo elemento ao topo da pilha. A operação `desempilha` em uma pilha sequencial remove e retorna o último elemento da pilha.

```
void PilhaSeq1::empilha(char dado) {
    v[N] = dado;
    N++;                    // N = N + 1
}

char PilhaSeq1::desempilha() {
    N--;                    // N = N - 1
    return v[N];
}
```

Implementação: Topo

A operação de topo em uma pilha sequencial retorna o último elemento empilhado.

```
char PilhaSeq1::topo() {
    return v[N-1];
}
int PilhaSeq1::tamanho() {
    return N;
}
```

Desafio: *O que aconteceria se a pilha estivesse vazia e o* `topo()` *fosse invocado? Como permitir que o programa continue mesmo após situações inesperadas como essa?*

Dica: *Retorne um* `char` ***opcional****, com uma pequena modificação na função* `topo()`*.*

Exemplo: `auto PilhaSeq1::topo() -> std::optional<char> {...}`

Exemplo de uso

Considere uma pilha sequencial (`MAX_N=5`): `PilhaSeq1 p; p.cria();`

```
p.N: | 0 |      p.v: |   |   |   |   |   |
                                0   1   2   3   4
```

Agora, empilhamos `A`, `B` e `C`, e depois desempilhamos uma vez.

```
p.N: | 1 |      p.v: | A |   |   |   |   |
                      0   1   2   3   4

p.N: | 2 |      p.v: | A | B |   |   |   |
                      0   1   2   3   4

p.N: | 3 |      p.v: | A | B | C |   |   |
                      0   1   2   3   4

p.N: | 2 |      p.v: | A | B |   |   |   |
                      0   1   2   3   4
```

Qual o topo atual da pilha?

Análise Preliminar: Pilha Sequencial

A Pilha Sequencial tem a vantagem de ser bastante simples de implementar, ocupando um espaço constante (na memória) para todas operações.

Porém, existe a limitação física de `MAX_N` posições imposta pela alocação estática, não permitindo que a pilha ultrapasse esse limite.

Desafio: implemente uma Pilha Sequencial utilizando alocação dinâmica para o vetor `v`. Assim, quando não houver espaço para novos elementos, aloque mais espaço na memória (copiando elementos existentes para o novo vetor).

Dica: Experimente a estratégia de *dobrar a capacidade* da pilha (quando necessário), e reduzir à metade a capacidade (quando necessário). Essa estratégia é bastante eficiente, mas requer alteração nos métodos `cria`, `libera`, `empilha` e `desempilha`.

Section 4

Pilhas Encadeadas

Pilhas Encadeadas

A implementação do TAD Pilha pode ser feito através de uma *estrutura encadeada* com alocação dinâmica de memória.

A vantagem é não precisar pre-determinar uma capacidade máxima da pilha (o limite é a memória do computador!). A desvantagem é depender de implementações ligeiramente mais complexas.

Implementação com ponteiros

Consideraremos uma pilha encadeada, utilizando um agregado `NoPilha1` para conectar cada elemento da pilha:

```
struct NoPilha1 {
    char dado;
    NoPilha1* prox;
};
```

```
struct PilhaEnc1 {
    NoPilha1* inicio;
    int N;
    void cria();
    void libera();
    char topo();
    void empilha(char dado);
    char desempilha();
    int  tamanho();
};
```

Implementação: Cria

```
void PilhaEnc1::cria() {
    N = 0;       // zero elementos na pilha
    inicio = 0; // endereço zero de memória
}
```

Exemplo de uso

Variável local do tipo Pilha Encadeada:

```
PilhaEnc1 p;
p.cria();
```

Visualização da memória

p.N: 0 **p.inicio: 0** $topo \leftarrow \epsilon$

0	4	...	100	104	108	112	116	...	8GiB

Implementação: Empilha

```
void PilhaEnc1::empilha(char v) {
  auto no = new NoPilha1{.dado = v, .prox = inicio};
  inicio = no;
  N++;                    // N = N + 1
}
```

Na memória: p.empilha('A'); p.empilha('B');

p.N: 0 **p.inicio: 0** *topo* $\leftarrow \epsilon$

0	4	...	100	104	108	112	116	...	8GiB

p.N: 1 **p.inicio: 112** *topo* $\leftarrow A$

						A	0		
0	4	...	100	104	108	112	116	...	8GiB

p.N: 2 **p.inicio: 100** *topo* $\leftarrow B \leftarrow A$

			B	112		A	0		
0	4	...	100	104	108	112	116	...	8GiB

Implementação: Topo

```
char PilhaEnc1::topo() {
    return inicio->dado;        // (*inicio).dado;
    // ou simplesmente...
    // return this->inicio->dado;
}
```

Na memória: p.topo();

p.N: 2 **p.inicio: 100** *topo* $\leftarrow B \leftarrow A$

				B	112			A	0			
0		4	...	100	104		108	112	116	...		8GiB

Implementação: Desempilha

```
char PilhaEnc1::desempilha() {
    NoPilha1* p = inicio->prox;
    char r = inicio->dado;
    delete inicio;
    std::println("{}", inicio); // qual valor?
    inicio = p;
    N--;                  // N=N-1
    return r;
}
```

```
// relembrando...
struct NoPilha1 {
    char dado;
    NoPilha1* prox;
};
```

Na memória: p.desempilha();

p.N: 2 **p.inicio: 100** *topo* $\leftarrow B \leftarrow A$

				B	112		A	0				
0		4	...	100	104	108	112	116	...	8GiB		

p.N: 1 **p.inicio: 112** *topo* $\leftarrow A$

							A	0				
0		4	...	100	104	108	112	116	...	8GiB		

Implementação: Libera

```
void PilhaEnc1::libera() {
   while (N > 0) {
      NoPilha1* p = inicio->prox;
      delete inicio;   inicio = p;    N--;
   }
}
```

Na memória: p.libera();

p.N: 2 **p.inicio: 100** *topo* $\leftarrow B \leftarrow A$

				B	112		A	0		
0	4	...	100	104	108	112	116	...	8GiB	

p.N: 1 **p.inicio: 112** *topo* $\leftarrow A$

							A	0		
0	4	...	100	104	108	112	116	...	8GiB	

p.N: 0 **p.inicio: 0** *topo* $\leftarrow \epsilon$

0	4	...	100	104	108	112	116	...	8GiB	

Section 5

Implementação Avançada com Ponteiros Inteligentes

Implementação com ponteiros inteligentes

Para uma implementação mais segura, é possível utilizar *smart pointers*. Em especial, basta utilizar o `std::unique_ptr`. Para simplificar a sintaxe, consideramos o seguinte "atalho" para o nome dos ponteiros únicos:

```
template<typename T>
using uptr = std::unique_ptr<T>;
```

Dessa forma, basta escrever `uptr<int>` para representar um `std::unique_ptr<int>`.

Implementação com ponteiros inteligentes

Consideraremos uma pilha encadeada, utilizando um agregado `NoPilha2` para conectar cada elemento da pilha:

```
struct NoPilha2 {
  char dado;
  uptr<NoPilha2> prox;
};
```

```
struct PilhaEnc2 {
    uptr<NoPilha2> inicio;
    int N;
    auto cria()   -> void;
    auto libera() -> void;
    auto topo()                  -> char;
    auto empilha (char dado) -> void;
    auto desempilha()            -> char;
    auto tamanho()               -> int;
};
```

Implementação: Cria

```
auto PilhaEnc2::cria() -> void {
    this->N = 0;            // zero elementos na pilha
    // this->inicio = 0; // não é necessário inicializar
}
```

Implementação: Empilha

```
auto PilhaEnc2::empilha(char v) -> void {
  this->inicio = std::make_unique<NoPilha2>(
    NoPilha2{.dado = v, .prox = std::move(this->inicio)}
  );
  this->N++;                    // N = N + 1
}
```

Implementação: Desempilha

```
auto PilhaEnc2::desempilha() -> char {
    char r = this->inicio->dado;
    this->inicio = std::move(this->inicio->prox);
    this->N--;              //N=N-1
    return r;
}
```

Implementação: Libera (inseguro)

```
auto PilhaEnc2::libera() -> void {
    this->inicio = nullptr;    // ou... this->inicio.reset();
    // todo o resto é destruído automaticamente
    // CUIDADO com estouro de pilha (stack overflow!)
}
```

Implementação: Libera (seguro)

```
auto PilhaEnc2::libera() -> void {
    // seguro contra stack overflow
    while (this->tamanho() > 0) {
        this->inicio = std::move(this->inicio->prox);
        this->N--;
    }
}
```

Análise Preliminar: Pilha Encadeada

A Pilha Encadeada é flexível em relação ao espaço de memória, permitindo maior ou menor utilização.

Como desvantagem tende a ter acessos de memória ligeiramente mais lentos, devido ao espalhamento dos elementos por toda a memória do computador (perdendo as vantagens de acesso rápido na *memória cache*, por exemplo).

Também é considerada como desvantagem o gasto de espaço extra com ponteiros em cada elemento, o que não acontece na Pilha Sequencial.

Section 6

Pilhas Genéricas e Conceitos de Pilha

Tarefa: Pilha Sequencial Genérica

Uma implementação genérica da pilha sequencial pode ser feita utilizando templates, inclusive para o limite de capacidade (permitindo maior personalização caso a caso).

```
template<typename T, int MAX_N>
class PilhaSeq
{
public:
  T v[MAX_N];                  // elementos na pilha
  int N;                       // num. de elementos na pilha
  auto cria()    -> void;      // inicializa agregado
  auto libera() -> void;       // finaliza agregado
  auto topo()              -> T;
  auto empilha(T dado) -> void;
  auto desempilha()        -> T;
  auto tamanho()           -> int;
};
```

Utilizando a Pilha Genérica

Antes de completar as funções pendentes, utilizaremos a `PilhaSeq`:

```
int main () {
    PilhaSeq<char, 100'000> p;
    p.cria();
    p.empilha('A');
    p.empilha('B');
    p.empilha('C');
    print("{}\n", p.topo());
    print("{}\n", p.desempilha());
    p.empilha('D');
    while(p.tamanho() > 0)
        print("{}\n", p.desempilha());
    p.libera();
    return 0;
}
```

Verifique as impressões em tela: *C C D B A*

Definição do *Conceito* PilhaTAD em C++

O *conceito* de pilha somente requer suas três operações básicas. Como consideramos uma *pilha genérica* (pilha de inteiro, char, etc), definimos um *conceito genérico* chamado PilhaTAD:

```
template<typename Agregado, typename Tipo>
concept PilhaTAD = requires(Agregado a, Tipo t)
{
    // requer operação 'topo'
    { a.topo() }       -> std::same_as<t>;
    // requer operação 'empilha' sobre tipo 't'
    { a.empilha(t) }   -> std::same_as<void>;
    // requer operação 'desempilha'
    { a.desempilha() } -> std::same_as<t>;
    // requer operação 'tamanho'
    { a.tamanho() }    -> std::same_as<int>;
};
```

Verificando se PilhaSeq1 satisfaz conceito PilhaTAD

O static_assert pode ser usado para assegurar a corretude de implementação do conceito PilhaTAD:

```
constexpr int MAX_N = 100'000; // capacidade máxima da pilha
struct PilhaSeq1 {
  char v[MAX_N];                // elementos na pilha
  int N;                        // num. de elementos na pilha
  // implementa métodos da Pilha
  // ...
};

// verifica se agregado PilhaSeq1 satisfaz conceito PilhaTAD
static_assert(PilhaTAD<PilhaSeq1, char>);
```

Verificando se PilhaEnc1 satisfaz conceito PilhaTAD

O static_assert pode ser usado para assegurar a corretude de implementação do conceito PilhaTAD:

```
struct NoPilha1 {
    char dado;
    NoPilha1* prox;
};
```

```
struct PilhaEnc1 {
    NoPilha1* inicio;
    int N;
    // implementa métodos da Pilha
    // ...
};
// verifica agregado PilhaEnc1
static_assert(PilhaTAD<PilhaEnc1, char>);
```

Tarefa: Implemente uma PilhaEnc genérica

Implemente uma pilha encadeada genérica PilhaEnc, que satisfaz o conceito PilhaTAD:

```
template<typename T>
struct NoPilhaEnc {
  T dado;
  NoPilhaEnc<T>* prox;
};

template<typename T>
struct PilhaEnc {
  NoPilhaEnc<T>* inicio;  // elementos na pilha de tipo T
  int N;                  // num. de elementos na pilha
  // implementa métodos da Pilha
  // ...
};

// verifica se agregado PilhaSeq satisfaz conceito PilhaTAD
```

Section 7

Pilhas na Biblioteca Padrão e Aplicações

Uso da std::stack

Em C/C++, é possível utilizar implementações *prontas* do TAD Pilha. A vantagem é a grande eficiência computacional e amplo conjunto de testes, evitando erros de implementação.

Na STL, basta fazer `import std;` e usar métodos `push`, `pop` e `top`.

```
import std;

int main() {
    std::stack<char> p;        // pilha de char
    p.push('A');
    p.push('B');
    println("{}", p.top());   // imprime B
    p.pop();
    println("{}", p.top());   // imprime A
    return 0;
}
```

Problema prático com recursão vs pilha

Como reverter uma string? Exemplo: `rev("ESCOLA") => "ALOCSE"`.

```
std::string rev(std::string s) {
  if (s.length() <= 1) return s;
  // exclui primeiro caractere de s e adiciona no fim
  // NOTA: "ESCOLA".substr(2) => "COLA"
  std::string r = rev(s.substr(1)) + s[0];
  return r;
}
```

Solução do problema prático com pilha

Solução com pilha (sem recursão):

```
std::string revs(std::string s) {
  std::stack<char> pchars;
  for (char c : s) pchars.push(c);
  std::string r;
  // reconstroi string ao contrário
  while (pchars.size() > 0) {
    r += pchars.top();
    pchars.pop();
  }
  return r;
}
```

Definindo um TAD para `std::stack`

Desafio: escreva um *conceito* (utilizando o recurso C++ Concepts) para o `std::stack` da STL, considerando operações `push`, `pop` e `top`.

Dica: Utilize o *conceito* `PilhaTAD` apresentado no curso, e faça os devidos ajustes. Verifique se `std::stack` passa no teste com `static_assert`.

Você pode compilar o código proposto (começando pelo slide anterior em um arquivo chamado `material/3-pilhas/main_pilha_stl.cpp`) usando o CMake 4.

Fim implementações

Fim parte de implementações.

Section 8

Análise de Complexidade

Pilha: Revisão Geral

- Para que serve uma pilha?
- Quais são os 3 métodos de uma pilha?
- Qual é a complexidade de cada método em uma Pilha Sequencial?
- Qual é a complexidade de cada método em uma Pilha Encadeada?
- Quais as vantagens e desvantagens de cada implementação de pilha?

Bibliografia Recomendada

Além da bibliografia do curso, recomendamos para esse tópico:

- Szwarcfiter, J.L; Markenzon, L. Estruturas de Dados e seus Algoritmos. Rio de Janeiro, LTC, 1994. Bibliografia Adicional:
- Cerqueira, R.; Celes, W.; Rangel, J.L. Introdução a estruturas de dados: com técnicas de programação em C. Editora, 2004.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein Algoritmos: Teoria e Prática. Ed. Campus, 2002.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 3rd ed.. The MIT Press, 2009.
- Preiss, B.R. Estruturas de Dados e Algoritmos Ed. Campus, 2000;
- Knuth, D.E. The Art of Computer Programming - Vols I e III. 2nd Edition. Addison Wesley, 1973.
- Graham, R.L., Knuth, D.E., Patashnik, O. Matemática Concreta. Segunda Edição, Rio de Janeiro, LTC, 1995.
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: https://github.com/isocpp/CppCoreGuidelines

Section 9

Agradecimentos

Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- https://igormcoelho.github.io/ilectures-pandoc/

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

Licença: CC-BY 4.0 2020

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)