# Estruturas de Dados I

## Sacos

Igor Machado Coelho

29/03/2021-02/11/2026

1. Sacos e Listas
2. Tipo Abstrato: Saco
3. Lista Encadeada
4. Extensões de Listas Encadeadas
5. Vetores como Sacos
6. Análise de Complexidade
7. Agradecimentos

Section 1

# Sacos e Listas

# 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

# Material Complementar

Materiais complementares:

- Livro: R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011
- Códigos em Java: https://algs4.cs.princeton.edu/code/

Section 2

# Tipo Abstrato: Saco

# Saco

O *Saco* ou *Bolsa* (do inglês *Bag*) é um Tipo Abstrato de Dado (TAD) que serve para agregar elementos de um tipo pré-definido (estrutura homogênea).

Em um *saco de produtos*, por exemplo:

- Se consegue "inserir" (do inglês *add*) novos produtos
- Podemos "percorrer" ou "iterar" (do inglês *iterate*) sobre os produtos
- Podemos "buscar" ou "encontrar" (do inglês *find*) um produto específico
- Podemos "remover" (do inglês *remove*) um dado produto

O *Saco* é também conhecido como *multiset* e alguns autores não consideram operações de remoção. [1]

[1] De acordo com NIST *"An unordered collection of values that may have duplicates"*.

# Saco na computação

Sacos são estruturas fundamentais na própria computação, como o conceito de `multiset`. Diferente de um `set`, não se exige que elementos sejam únicos.

Em Python, a classe `Counter` representa um tipo de saco. Em `Java`, a interface Bag (da *Apache Commons Collections*) representa um saco. Em C++, pode-se considerar containers como `std::list` e `std::vector`, que são *sequências*, e também em `std::unordered_multiset` (saco sem ordem específica, com repetições).

Mas antes de definir um Saco, precisamos revisar outros componentes básicos: Iterador (que é uma Vista) e Percorrível.

# Revisão: Iterador e Percorrível

```
template <typename It, typename T>
concept IteradorTAD = VistaTAD<It> && std::same_as<typename It::Tipo, T> &&
   requires(It it) { { it.terminou() }; { it.proximo() }; };

template <typename P, typename T>
concept PercorrivelTAD = requires(P p) { { p.iterador() } -> IteradorTAD<T>; };
```

O Saco será um **Percorrível**: seu `iterador()` permite visitar todos os elementos.

Lembre-se que uma Vista basicamente tem um método `.le()` que retorna uma cópia do elemento da origem.

# Operações de um Saco (TAD)

Um Saco é um tipo abstrato de dado (TAD) **Percorrível** que carrega seus elementos e consiste de 4 operações básicas:

- adicionar (*add*)
- buscar (*find*)
- remover (*remove*): remove o elemento de uma posição (iterador) e retorna um iterador para o seguinte (*todos os demais iteradores do saco ficam inválidos*)
- tamanho (*size*): retorna o número de elementos

Também inclui sua operação **Percorrível** (de *range*/iterador):

- iterador (*begin/first/iterate*)

Note que o Saco não tem conceito de *indexação* nem uma ordem específica de percurso / ordenação dos elementos. Elementos podem ser repetidos.

# Implementações

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

Podemos responder as perguntas: *'a' existe? 'd' existe?*

*Quais estruturas de dados podem ser usadas para implementar as operações de um Saco?*

Apresentaremos inicialmente a implementação do TAD Saco através de Listas Encadeadas, embora também possa ser feito através de Arrays.

# Definição do *Conceito* Saco em C++

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

```
template <typename S, typename T>
concept SacoTAD = PercorrivelTAD<S, T> &&
   requires(S s, T t) {
   { s.adiciona(t) };  // insere elemento
   { s.busca(t) };     // posição de 't' (ou terminado)
   { s.remove(s.iterador()) } -> IteradorTAD<T>; // remove da posição 'it'
   { s.tamanho() };    // número de elementos
};
```

# Utilização do SacoTAD

Antes de completar as funções pendentes, utilizaremos um SacoTAD:

```
int main () {
   SacoTAD<char> auto s = ...; // alguma implementação
   s.cria();
   s.adiciona('c');
   s.adiciona('a');
   s.adiciona('b');
   s.adiciona('a');
   println("{}", s.busca('a').terminou()); // 0
   println("{}", s.busca('d').terminou()); // 1
   for (auto it = s.iterador(); !it.terminou(); it.proximo())
      println("{}", it.le()); // a b a c (em qual ordem?)
   s.remove(s.busca('b'));
   s.libera();
   ...
```

Section 3

# Lista Encadeada

# Lista Encadeada

As listas encadeadas possibilitam a inclusão de elementos em quantidade arbitrária (limitada à memória do computador), através de nós de encadeamento.

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

$$[inicio \rightarrow][a| \rightarrow]\ [b| \rightarrow]\ [a| \rightarrow]\ [c| \rightarrow]\ [S| \rightarrow 0]$$

O nó $S$ é uma **sentinela**: marca o fim da lista e sempre existe, mesmo com a lista vazia.

# Implementação (I)

Consideraremos uma lista encadeada de caracteres. Para tal, definimos um agregado que contenha elementos, e outro que represente um iterador.

```
struct NoEnc1 {
   char dado;
   NoEnc1* proximo;
};

struct IteradorNoEnc1 {
   using Tipo = char;  // tipo associado
   NoEnc1* no;         // no atual
   char le();          // da Vista (inglês 'current')
   bool terminou();    // inglês 'isDone'
   void proximo();     // inglês 'next'
};
```

# Implementação (II)

```
struct ListaEnc1 {
   NoEnc1* inicio; // inicio da lista
   int N;          // num. de elementos na lista
   void cria();    // inicializa agregado
   void libera();  // finaliza agregado
   void adiciona(char dado);
   IteradorNoEnc1 iterador();
   IteradorNoEnc1 busca(char dado);
   IteradorNoEnc1 remove(IteradorNoEnc1 it);
   int tamanho();
};
// verifica se agregado ListaEnc1 satisfaz conceito SacoTAD
static_assert(SacoTAD<ListaEnc1, char>);
```

# Utilização da ListaEnc1

Antes de completar as funções pendentes, utilizaremos a `ListaEnc1`:

```
int main () {
   ListaEnc1 l;
   l.cria();
   l.adiciona('c');
   l.adiciona('a');
   l.adiciona('b');
   l.adiciona('a');
   println("{}", l.busca('a').terminou());
   println("{}", l.busca('d').terminou());
   for(auto it = l.iterador(); !it.terminou(); it.proximo()) {
      println("{}", it.le());
   }
   l.libera();
   ...
```

# Implementação: Cria e Libera

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

```
struct ListaEnc1 {
...
void cria() {
   inicio = new NoEnc1{}; // sentinela: dado = 0, proximo = 0
   N = 0;
}

void libera() {
   while (inicio != 0) {
      auto* prox = inicio->proximo;
      delete inicio;   inicio = prox;
   }
   N = 0;
}
...
```

# Implementação: Adiciona

A operação adiciona em uma lista encadeada adiciona um novo elemento na "cabeça" da lista (no início).

```
struct ListaEnc1 {
...
int tamanho() { return N; }
...
void adiciona(char dado) {
    auto* no = new NoEnc1{.dado = dado, .proximo = inicio};
    inicio = no;
    N++; // N = N + 1
}
...
};
```

# Implementação: Iterador de Lista

O iterador terminou quando chega à **sentinela**, o único nó sem próximo.

```
struct IteradorNoEnc1 {
   using Tipo = char;  // tipo associado
   NoEnc1* no;
   char le() {
      return no->dado;
   }
   bool terminou() {
      return no->proximo == 0;
   }
   void proximo() {
      no = no->proximo;
   }
};
// verifica se agregado IteradorNoEnc1 satisfaz conceito IteradorTAD
```

# Implementação: Iterador na Lista

A operação `iterador` em uma lista encadeada retorna o iterador da lista, posicionado no começo.

```
struct ListaEnc1 {
...
IteradorNoEnc1 iterador() {
   IteradorNoEnc1 it{.no = inicio};
   return it;
}
...
};
```

# Implementação: Busca

A operação busca em uma lista encadeada utiliza o iterador da lista, posicionado no começo, para encontrar o elemento, retornando o iterador atualizado na posição correspondente.

```
struct ListaEnc1 {
...
IteradorNoEnc1 busca(char dado) {
   auto it = iterador();
   while (!it.terminou())
   {
      if (it.le() == dado)
         return it;
      it.proximo();
   }
   return it;
}
...
```

# Implementação: Busca Recursiva

A busca pode ser feita de forma recursiva também:

```
IteradorNoEnc1 buscarec(IteradorNoEnc1 it, char dado)
{
   if(it.terminou() || it.le() == dado)
      return it;
   else
   {
      it.proximo();
      return buscarec(it, dado);
   }
}
```

# Implementação: Remoção

A remoção de um elemento ocorre a partir de um iterador.

```
struct ListaEnc1 {
...
IteradorNoEnc1 remove(IteradorNoEnc1 it) {
   auto* prox = it.no->proximo; // sempre existe (no máximo, a sentinela)
   *it.no = *prox;              // copia o seguinte
   delete prox;
   N--;
   return it;                   // mesma posição, novo elemento
}
...
};
```

# Removendo Durante o Percurso

Após `remove`, **todos** os iteradores do saco ficam inválidos, **exceto o retornado**, que aponta para o elemento seguinte. Assim, para remover todas as ocorrências de um elemento, continuamos a partir dele:

```
// Exemplo de uso
ListaEnc1 s;
s.cria();
s.adiciona('x'); s.adiciona('c');
s.adiciona('x'); s.adiciona('a');
s.adiciona('b'); s.adiciona('x');
// remove todos os 'x' do saco 's'
auto it = s.iterador();
// lista: x b a x c x
```

```
while (!it.terminou()) {
   if (it.le() == 'x')
      // segue do retornado
      it = s.remove(it);
   else
      // só avança se manteve
      it.proximo();
}
// lista: b a c
s.libera();
```

**Desafio:** qual o custo desse laço na lista encadeada? E se `remove` não retornasse um iterador?

Section 4

# Extensões de Listas Encadeadas

# Extensões de Listas Encadeadas

A implementação de listas encadeadas tipicamente permite acesso bidirecional, com um ponteiro `anterior` além de um `proximo`. Este tipo de lista é chamado de *Lista Duplamente Encadeada*.

```
struct NoDuploEnc1 {
   char dado;
   NoDuploEnc1* anterior;
   NoDuploEnc1* proximo;
};
```

**Desafio:** Implemente as operações da lista!

**Dica:** para simplificar operações nos extremos da lista, pode-se utilizar um nó sentinela de Cabeça e Cauda.

# Avançado: Iteradores de *Range* em C++

É possível transformar a implementação da `ListaEnc1` para uso em *for* do tipo *range*, em C++. Exemplo:

```
int main() {
   auto s = ListaEnc1();
   s.cria();
   s.adiciona('c');
   s.adiciona('a');
   s.adiciona('b');
   s.adiciona('a');
   println("{}", s.busca('a').terminou()); // 0
   println("{}", s.busca('d').terminou()); // 1
   println("{}", buscarec(s.iterador(), 'd').terminou()); // 1
   for (auto x : s)
      println("{}", x); // a b a c
```

# Avançado: Implementação: Iterador de *Range* em C++

Por padrão, a linguagem C++ exige os métodos `begin()` e `end()` para início e fim do processo de iteração, no agregado `ListaEnc1`. Como nosso iterador sabe sozinho quando terminou, o `end()` retorna apenas um **marcador de fim** (`std::default_sentinel`, do C++20).

```
struct ListaEnc1 {
...
   // iterador de início
   IteradorNoEnc1 begin() {
      return iterador();
   }

   // marcador de final da iteração
   std::default_sentinel_t end() {
      return {};
   }
};
```

# Avançado: Implementação de Operadores no Iterador

**Tópico Avançado**: os métodos `terminou()`, `proximo()` e `le()` são substituídos pelos operadores `==`, `++` e `*`, no agregado `IteradorNoEnc1`.

```
struct IteradorNoEnc1 {
   NoEnc1* no;     // elemento atual
   ...             // demais métodos omitidos

   char operator*() { return le(); }

   bool operator==(std::default_sentinel_t s) {
      return terminou();  // chegou ao fim?
   }

   void operator++() { proximo(); }
};
```

# Fim implementações (listas)

Fim parte de implementações de listas.

Section 5

# Vetores como Sacos

# Vetores como Sacos

Os vetores podem ser utilizados como sacos, basta incluir os elementos sequencialmente à medida que forem adicionados.

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

$$|c|a|b|a|$$

0 1 2 3

# Implementação

Consideraremos um vetor de caracteres pré-alocado com capacidade `MAX_N`.

```
constexpr int MAX_N = 10000;
struct SacoVetor1 {
   char elementos[MAX_N]; // elementos
   int N;                 // num. de elementos
   void cria();           // inicializa agregado
   void libera();         // finaliza agregado
   void adiciona(char dado);
   IteradorVetor1 iterador();
   IteradorVetor1 busca(char dado);
   IteradorVetor1 remove(IteradorVetor1 it);
   int tamanho();
};
// verifica se agregado SacoVetor1 satisfaz conceito SacoTAD
static_assert(SacoTAD<SacoVetor1, char>);
```

# Utilização da SacoVetor1

Antes de completar as funções pendentes, utilizaremos a `SacoVetor1`:

```
int main () {
   auto s = SacoVetor1();
   s.cria();
   s.adiciona('c');
   s.adiciona('a');
   s.adiciona('b');
   s.adiciona('a');
   println("{}", s.busca('a').terminou());
   println("{}", s.busca('d').terminou());
   for (auto it = s.iterador(); !it.terminou(); it.proximo())
      println("{}", it.le()); // c a b a (nesta ordem)
   s.libera();
   ...
}
```

# Implementação: Cria e Libera

A operação `cria` inicializa a estrutura para uso, e a função `libera` não desaloca nada.

```
struct SacoVetor1 {
...
void cria()   { N = 0; }
void libera() { N = 0; }
...
};
```

# Implementação: Adiciona

A operação adiciona insere no final do vetor.

```
struct SacoVetor1 {
...
int tamanho() { return N; }
...
void adiciona(char dado) {
   elementos[N] = dado;
   N++; // N = N + 1
}
...
};
```

# Implementação: Iterador de Vetor

O iterador do vetor é o mesmo `IteradorVetorInt` dos conceitos, agora para `char`.

```
struct IteradorVetor1 {
   using Tipo = char;  // tipo associado
   const char* atual;  // posição atual
   const char* fim;    // após o último
   char le()       { return *atual; }
   bool terminou() { return atual == fim; }
   void proximo()  { atual++; }
};
// verifica se agregado IteradorVetor1 satisfaz conceito IteradorTAD
static_assert(IteradorTAD<IteradorVetor1, char>);
```

# Implementação: Iterador no Vetor

A operação `iterador` retorna o início do vetor e a posição logo após o último elemento.

```
struct SacoVetor1 {
...
IteradorVetor1 iterador() {
   return {.atual = elementos, .fim = elementos + N};
}
...
};
```

# Implementação: Busca

A busca no vetor é idêntica à da lista: só usa o iterador!

```
struct SacoVetor1 {
...
IteradorVetor1 busca(char dado) {
   auto it = iterador();
   while (!it.terminou() && it.le() != dado)
      it.proximo();
   return it;
}
...
};
```

# Implementação: Remoção

Como o Saco **não tem ordem**, basta trazer o último elemento para a posição removida: custo $O(1)$!

```
struct SacoVetor1 {
...
IteradorVetor1 remove(IteradorVetor1 it) {
   int i = it.atual - elementos;  // posição removida
   elementos[i] = elementos[N-1]; // traz o último para cá
   N--;
   return {.atual = elementos + i, .fim = elementos + N};
}
...
};
```

**Desafio:** por que o iterador retornado aponta para a **mesma** posição? E por que precisa de um novo `fim`?

# Fim implementações

Fim parte de implementações.

Section 6

# Análise de Complexidade

# Saco: Revisão Geral

- Para que serve um saco? Por que ele é **Percorrível**?
- O que o contrato de `remove` garante sobre os iteradores?
- Qual é a complexidade de cada operação em cada implementação?

# Saco: Revisão Geral

- Para que serve um saco? Por que ele é **Percorrível**?
- O que o contrato de `remove` garante sobre os iteradores?
- Qual é a complexidade de cada operação em cada implementação?

| Operação | Lista (sentinela) | Vetor (`MAX_N`) |
|---|---|---|
| `cria` | $O(1)$ (aloca a sentinela) | $O(1)$ |
| `libera` | $O(n)$ (libera cada nó) | $O(1)$ |
| `adiciona` | $O(1)$ (no início) | $O(1)$ (no fim, se houver espaço) |
| `iterador`, `tamanho` | $O(1)$ | $O(1)$ |
| `busca` | $O(n)$ | $O(n)$ |
| `remove` | $O(1)$ (copia o seguinte) | $O(1)$ (traz o último) |
| remover todos os `x` | $O(n)$ | $O(n)$ |
| espaço | $n+1$ nós | $MAX_N$ fixo |

**Reflexão:** as duas remoções são $O(1)$ por caminhos diferentes, e ambas só funcionam porque o Saco **não tem ordem**. A busca é $O(n)$ nas duas: nas próximas aulas faremos melhor!

# 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

Section 7

# 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)