Estruturas de Dados I

Sacos Ordenados

Igor Machado Coelho

02/11/2026

1 Tópico Avançado: Saco Ordenado

2 Agradecimentos

Section 1

Tópico Avançado: Saco Ordenado

Saco Ordenado: Motivação

Esta parte é **curta** e serve como **motivação** para as próximas aulas.

No Saco, a `busca` custa $O(n)$: sem ordem, é preciso olhar tudo (desafio do Percorrível!).

E se mantivermos os elementos **sempre ordenados**, como no Conjunto Ordenado?

- a busca binária volta a funcionar: $O(\log n)$
- surgem perguntas **sobre a ordem**: menor, maior, "quantos `x`?", "quais entre `a` e `b`?"
- mas inserir e remover passam a custar $O(n)$...

Saco Ordenado

Um *Saco Ordenado* é um Saco cujo vetor satisfaz a **invariante**:

$$v[0] \leq v[1] \leq \cdots \leq v[N-1] \qquad (\leq \text{, pois há repetições!})$$

Consequência: seu iterador visita os elementos **em ordem crescente**.

Lembrete: *no Saco, removíamos trazendo o último para o buraco. Por que isso agora é* ***proibido****?*

Saco × Saco Ordenado

Operação	Saco (vetor)	Saco Ordenado (vetor)
`adiciona`	$O(1)$	$O(n)$: abre espaço
`remove`	$O(1)$: traz o último	$O(n)$: fecha o buraco
`busca`	$O(n)$	$\mathbf{O}(\log \mathbf{n})$
percorrer	$O(n)$, ordem qualquer	$O(n)$, **ordem crescente**
contar cópias de `x`	$O(n)$	$\mathbf{O}(\log \mathbf{n})$
menor / maior	$O(n)$	$\mathbf{O}(\mathbf{1})$: nas pontas
intervalo $[a, b]$ (k elementos)	$O(n)$	$\mathbf{O}(\log \mathbf{n} + \mathbf{k})$

Vale a pena quando há **muitas consultas** e **poucas atualizações**.

Buscas por Ordem: buscaMaiorIgual e buscaMaior

Motivação: pense em uma estante. Para guardar mais um `d` em uma estante ordenada, há um **primeiro** e um **último** lugar possível:

```
 a | d   d   d | f
   ^           ^

 0   1   2   3   4
buscaMaiorIgual('d') = 1      buscaMaior('d') = 4
(1o elemento >= d)            (1o elemento > d)
```

- Entre as duas posições estão **todas as cópias** de `d`
- Se as duas coincidem, `d` **não está** no saco

Na STL existem `lower_bound` (buscaMaiorIgual) e `upper_bound` (buscaMaior), e no Java `ceiling` e `higher`.

Revisão: Iterador de Vetor

O Saco Ordenado usa o mesmo iterador do Saco em vetor (veja mais detalhes nos slides de **Sacos**):

```
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++; }
};
static_assert(IteradorTAD<IteradorVetor1, char>);
```

- `le()`: lê o elemento atual (é uma **Vista**)
- `proximo()`: avança uma posição
- `terminou()`: chegou em `fim`?

Um par (`atual`, `fim`) delimita uma **faixa** do vetor: as buscas deste tópico só escolhem onde ela começa e termina!

Conceito: Saco Ordenado

Um *Saco Ordenado* é um Saco com elementos **comparáveis** e com as duas **buscas por ordem**:

```
template <typename S, typename T>
concept SacoOrdenadoTAD = SacoTAD<S, T> &&
   std::totally_ordered<T> &&  // comparáveis
   requires(S s, T t) {
   // 1a posição com elemento >= t
   { s.buscaMaiorIgual(t) } -> IteradorTAD<T>;
   // 1a posição com elemento > t
   { s.buscaMaior(t) } -> IteradorTAD<T>;
};
```

```
struct SacoOrdenado1 {
   char elementos[MAX_N]; // sempre orde
   int N;                 // num. de ele
   // ... operações do Saco ...
   IteradorVetor1 buscaMaiorIgual(char x
   IteradorVetor1 buscaMaior(char x);
};
static_assert(SacoOrdenadoTAD<SacoOrdena
```

Atenção: o conceito verifica que as buscas **existem**, mas não que o vetor está **ordenado**: essa invariante o compilador não consegue verificar!

Implementação: SacoOrdenado1

Consideraremos um vetor de caracteres pré-alocado com capacidade `MAX_N`, **sempre ordenado**.

```
struct SacoOrdenado1 {
    char elementos[MAX_N]; // sempre ordenado
    int N;                 // num. de elementos
    void cria();                              // iguais ao SacoVetor1:
    void cria(IteradorTAD<char> auto it); // cópia
    void libera();
    void adiciona(char x);
    IteradorVetor1 iterador();
    IteradorVetor1 busca(char x);
    IteradorVetor1 remove(IteradorVetor1 it);
    int tamanho();
    IteradorVetor1 buscaMaiorIgual(char x);
    IteradorVetor1 buscaMaior(char x);
};
static_assert(SacoOrdenadoTAD<SacoOrdenado1, char>);
```

```
void cria()    { N = 0; }
void libera()  { N = 0; }
int tamanho()  { return N; }
IteradorVetor1 iterador() {
   return {.atual = elementos,
           .fim   = elementos + N};
}
```

O `cria(it)`, que cria a partir de **outro saco**, veremos mais adiante.

Implementação: Buscas

```
// primeira posição com elemento >= x
IteradorVetor1 buscaMaiorIgual(char x) {
   // busca em [ini, fim)
   int ini = 0, fim = N;
   while (ini < fim) {
      int meio = ini + (fim - ini) / 2;
      if (elementos[meio] < x)
         ini = meio + 1;
      else
         fim = meio;
   }
   return {.atual = elementos + ini,
           .fim   = elementos + N};
}
```

```
// primeira posição com elemento > x
IteradorVetor1 buscaMaior(char x) {
   // idêntica, trocando a comparação:
   //   if (elementos[meio] <= x)
   ...
}

IteradorVetor1 busca(char x) {
   auto it = buscaMaiorIgual(x);
   if (!it.terminou() && it.le() == x)
      return it;
   return {.atual = elementos + N,
           .fim   = elementos + N};
}
```

Implementação: Adiciona e Remove

```
void adiciona(char x) {
   // após as cópias de x
   int p = buscaMaior(x).atual - elementos;
   for (int i = N; i > p; i--)
      elementos[i] = elementos[i - 1];
   elementos[p] = x;       // abriu espaço
   N++;
}
```

```
IteradorVetor1 remove(IteradorVetor1 it)
   int i = it.atual - elementos;
   for (int j = i; j < N - 1; j++)
      elementos[j] = elementos[j + 1];
   N--;                          // fechou o b
   return {.atual = elementos + i,
           .fim   = elementos + N};
}
```

Ambas $O(n)$: achar o lugar é rápido, **deslocar** é que custa. O contrato de `remove` é o mesmo do Saco!

Usando as Buscas: Contar e Intervalo

```
int conta(char x) {
   return buscaMaior(x).atual
        - buscaMaiorIgual(x).atual;
}

// elementos em [a, b]   (pre: a <= b)
IteradorVetor1 intervalo(char a, char b) {
   return
   {.atual = buscaMaiorIgual(a).atual,
    .fim   = buscaMaior(b).atual};
}
```

```
// saco: a b d d d f g
println("{}", s.conta('d'));  // 3
println("{}", s.conta('c'));  // 0

auto it = s.intervalo('b', 'f');
while (!it.terminou()) {
   println("{}", it.le()); // b d d d f
   it.proximo();
}
```

O intervalo é apenas um **iterador com outro fim**: uma vista de uma faixa do saco!

Criando a Partir de Outro Saco em $O(n \log n)$

Como criar um Saco Ordenado com os n elementos de outro saco?

Estratégia	Custo
n chamadas de `adiciona`	$O(n^2)$: cada uma desloca
copiar + Insertion Sort	$O(n^2)$ no pior caso, $O(n)$ se já ordenado
copiar + Merge Sort	$\mathbf{O(n \log n)}$ sempre

```
void cria(IteradorTAD<char> auto it) {  // iterador de QUALQUER saco
   N = 0;
   while (!it.terminou()) {              // O(n): copia
      elementos[N] = it.le();
      N++;       it.proximo();
   }
   ordena();   // O(n log n) com Merge Sort (aula de Conjuntos Simples)
}
```

Desafio: faça o `ordena()`! Se a origem for **outro Saco Ordenado**, qual ordenação você escolheria?

Para Onde Vamos

- Na STL: `std::multiset` (árvore) e `std::flat_multiset` (C++23, vetor ordenado: é o nosso!)
- **Dicionário Ordenado:** o mesmo Saco Ordenado, sem repetições, com pares (chave, valor) comparados pela **chave**. As buscas `buscaMaiorIgual`, `buscaMaior` e `intervalo` serão as mesmas!
- **Árvores:** como ter consultas **e** atualizações em $O(\log n)$?

Pergunta final: *no Saco Ordenado, achamos o lugar em* $O(\log n)$*, mas deslocamos em* $O(n)$*. Que estrutura evitaria o deslocamento?*

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 2

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)