Estruturas de Dados I

Conjuntos Simples, Busca e Ordenação (Slide ainda em revisão! Não use!)

Igor Machado Coelho

01/09/2026

1. Conjuntos, Busca e Ordenação
2. Tipo Abstrato: Conjunto
3. Conjunto Simples Sequencial
4. Conjunto Sequencial Dinâmico
5. Análise Amortizada
6. Algoritmos de Busca
7. Algoritmos de Ordenação
8. Conjunto Ordenado
9. E depois? Vetores que se reorganizam
10. Análise de Complexidade
11. Agradecimentos

Section 1

Conjuntos, Busca e Ordenação

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 e de vetores (arrays)
- Noções de listas e encadeamento
- Noções de análise assintótica (notação $O(\cdot)$)

Section 2

Tipo Abstrato: Conjunto

Conjunto

O Conjunto (do inglês *Set*) é um Tipo Abstrato de Dado (TAD) que segue a ideia matemática de conjunto:

- **não há repetição**: cada elemento aparece no máximo uma vez
- **não há ordem intrínseca**: $\{A, B\} = \{B, A\}$
- a pergunta fundamental é *"o elemento x pertence?"*

$$C = \{\ A,\ B,\ C\ \} \qquad x \in C\ ?$$

Qualquer elemento pode entrar ou sair a qualquer momento.

Operações de um Conjunto Simples

Chamaremos de **Conjunto Simples** a versão mínima do TAD, com 4 operações:

- `pertence(x)` — consulta (*membership*)
- `insere(x)` — insere, se ainda não existir
- `remove(x)` — remove, se existir
- `tamanho()` — cardinalidade $|C|$

Convenção do curso: `insere` e `remove` devolvem `bool`, indicando se o conjunto foi de fato **modificado**.

Operações de um Conjunto Simples

Chamaremos de **Conjunto Simples** a versão mínima do TAD, com 4 operações:

- `pertence(x)` — consulta (*membership*)
- `insere(x)` — insere, se ainda não existir
- `remove(x)` — remove, se existir
- `tamanho()` — cardinalidade $|C|$

Convenção do curso: `insere` e `remove` devolvem `bool`, indicando se o conjunto foi de fato **modificado**.

Um Conjunto *completo* acrescentaria união, interseção, diferença e percurso (iteração) — deixaremos isso para depois. **Comece pelo mínimo!**

Invariantes do TAD Conjunto

Todo TAD é definido não só por *o que ele faz*, mas por *o que ele promete*.

Para o Conjunto Simples, as promessas (**invariantes**) são:

1. $0 \leq N \leq MAX_N$
2. não existem $i \neq j$ com `v[i] == v[j]` (sem repetição)

E as pós-condições:

- após `insere(x)`, vale `pertence(x)`
- após `remove(x)`, vale `!pertence(x)`
- `tamanho()` só muda quando a operação devolve `true`

Guarde essa ideia: *invariante é o que separa uma "coleção de dados" de uma "estrutura de dados".*

Section 3

Conjunto Simples Sequencial

Conjunto Sequencial (não ordenado)

A implementação mais direta usa um **vetor**, exatamente como na Pilha Sequencial: os elementos ficam em um *espaço contíguo* de memória, nas posições `0..N-1`.

A diferença é que **a ordem das posições não importa** — o vetor é apenas um "saco" de elementos distintos.

Implementação

Consideraremos um conjunto sequencial com, no máximo, `MAX_N` elementos no vetor `v` do tipo caractere.

```
constexpr int MAX_N = 100'000;  // capacidade máxima do conju
struct ConjuntoSeq1 {
    char v[MAX_N];              // elementos do conjunto
    int N;                      // num. de elementos (cardinalidade
    void cria();                // inicializa agregado
    void libera();              // finaliza agregado
    bool pertence(char x);
    bool insere(char x);
    bool remove(char x);
    int  tamanho();
};
```

Versões genéricas (com `template`) ficam para o fim da aula — primeiro, o essencial.

Utilização do Conjunto

Antes de completar as funções pendentes, utilizaremos o `ConjuntoSeq1`:

```
import std;
int main() {
    ConjuntoSeq1 c;
    c.cria();                          // nosso contrato no curso!
    c.insere('A');  c.insere('B');  c.insere('A');
    println("{}", c.tamanho());
    println("{}", c.pertence('B'));
    println("{}", c.remove('B'));
    println("{}", c.pertence('B'));
    c.libera();                        // nosso contrato no curso!
    return 0;
}
```

Verifique as impressões em tela:

Utilização do Conjunto

Antes de completar as funções pendentes, utilizaremos o `ConjuntoSeq1`:

```
import std;
int main() {
    ConjuntoSeq1 c;
    c.cria();                          // nosso contrato no curso!
    c.insere('A');  c.insere('B');  c.insere('A');
    println("{}", c.tamanho());
    println("{}", c.pertence('B'));
    println("{}", c.remove('B'));
    println("{}", c.pertence('B'));
    c.libera();                        // nosso contrato no curso!
    return 0;
}
```

Verifique as impressões em tela:

2 true true false

Implementação: Cria e Libera

```
void ConjuntoSeq1::cria() {
    N = 0;                  // conjunto vazio
}

void ConjuntoSeq1::libera() {
    // nenhum recurso dinâmico para desalocar
}
```

O `tamanho()` é imediato:

```
int ConjuntoSeq1::tamanho() {
    return N;
}
```

Implementação: Pertence (busca linear)

Sem nenhuma organização interna, só resta **olhar elemento por elemento**:

```
bool ConjuntoSeq1::pertence(char x) {
    for (int i = 0; i < N; i++)
        if (v[i] == x)
            return true;        // achou!
    return false;               // varreu tudo e não achou
}
```

No **pior caso** (elemento ausente), fazemos N comparações: custo $O(N)$.

Desafio: *quantas comparações são feitas, em média, quando o elemento* ***está*** *no conjunto? E quando* ***não está****?*

Implementação: Insere

Inserir exige **garantir a invariante de não-repetição**:

```
bool ConjuntoSeq1::insere(char x) {
    if (pertence(x))        // O(N): mantém a invariante!
        return false;       // já existe: nada muda
    v[N] = x;               // O(1): coloca no fim
    N++;
    return true;
}
```

*Note que o custo $O(N)$ da inserção **não** vem de "colocar o elemento", e sim de **verificar** que ele ainda não estava lá.*

Implementação: Remove

Como a ordem não importa, podemos **tapar o buraco com o último elemento**:

```
bool ConjuntoSeq1::remove(char x) {
    for (int i = 0; i < N; i++)
        if (v[i] == x) {
            v[i] = v[N - 1];    // traz o último para o buraco
            N--;
            return true;
        }
    return false;
}
```

Desafio: *por que essa técnica seria* ***proibida*** *em um vetor ordenado?*

Exemplo de uso

Considere um conjunto sequencial (`MAX_N=5`): `ConjuntoSeq1 c;`
`c.cria();`

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

Inserimos `A`, `B` e `C`:

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

Tentamos inserir `B` novamente (devolve `false`, nada muda):

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

Removemos `A` (o último elemento vem ocupar a posição 0):

```
c.N: | 2 |      c.v: | C | B |   |   |   |
                       0   1   2   3   4
```

Análise Preliminar: Conjunto Sequencial

operação	custo
`pertence`	$O(N)$
`insere`	$O(N)$
`remove`	$O(N)$
`tamanho`	$O(1)$

Vantagens: simplicidade, memória contígua (ótimo uso de *cache*), zero espaço extra.

Desvantagem: **tudo custa** $O(N)$, porque toda operação depende de uma busca linear.

Análise Preliminar: Conjunto Sequencial

operação	custo
`pertence`	$O(N)$
`insere`	$O(N)$
`remove`	$O(N)$
`tamanho`	$O(1)$

Vantagens: simplicidade, memória contígua (ótimo uso de *cache*), zero espaço extra.

Desvantagem: **tudo custa** $O(N)$, porque toda operação depende de uma busca linear.

E ainda temos o `MAX_N`: **um limite físico** imposto pela alocação estática.

Section 4

Conjunto Sequencial Dinâmico

O problema do `MAX_N`

Com `constexpr int MAX_N = 100'000`, o agregado **sempre** ocupa 100 mil posições:

- se o conjunto tiver 3 elementos, desperdiçamos 99'997 posições
- se precisar de 100'001, o programa **falha** (e não há o que fazer)

Escolher `MAX_N` é adivinhar o futuro. A solução é **alocação dinâmica**: o vetor cresce conforme a necessidade.

Implementação: agregado dinâmico

Guardamos, além de N, a **capacidade** atual cap (espaço alocado):

```
struct ConjuntoSeqDin {
    char* v;        // vetor alocado dinamicamente (não é mais
    int cap;        // capacidade: quantas posições existem em
    int N;          // ocupação: quantas posições estão em uso
    void cria();
    void libera();
    void redimensiona(int novacap);   // operação interna
    bool pertence(char x);
    bool insere(char x);
    bool remove(char x);
    int  tamanho();
};
```

Invariante nova: $0 \leq N \leq cap$

Implementação: Cria e Libera

Agora o `libera()` deixa de ser vazio — ele é **obrigatório**!

```
void ConjuntoSeqDin::cria() {
    cap = 1;                    // capacidade inicial
    v = new char[cap];          // aloca no heap
    N = 0;
}

void ConjuntoSeqDin::libera() {
    delete[] v;                 // devolve a memória (note o delete
    v = nullptr;                // evita ponteiro pendurado
    cap = 0;    N = 0;
}
```

Cuidado: `delete[]` (com colchetes!) para memória alocada com `new[]`.

Implementação: Redimensiona

A operação central: alocar um vetor novo, **copiar** e devolver o antigo.

```
void ConjuntoSeqDin::redimensiona(int novacap) {
    char* novo = new char[novacap];
    for (int i = 0; i < N; i++)    // O(N): copia tudo!
        novo[i] = v[i];
    delete[] v;                    // devolve o vetor antigo
    v = novo;
    cap = novacap;
}
```

Essa é a operação cara. A pergunta toda da aula é: ***com que frequência ela acontece?***

Implementação: Insere e Remove

```
bool ConjuntoSeqDin::insere(char x) {
    if (pertence(x)) return false;
    if (N == cap)                       // vetor cheio?
        redimensiona(2 * cap);          // DOBRA a capacidade
    v[N] = x;   N++;
    return true;
}

bool ConjuntoSeqDin::remove(char x) {
    // ... busca e troca com o último, como antes ...
    N--;
    if (cap > 1 && N <= cap / 4)  // ocupação baixa?
        redimensiona(cap / 2);          // devolve memória
    return true;
}
```

Na memória: o crescimento

Inserindo `A`, `B`, `C`, `D`, `E` a partir de `cap=1`:

```
cap: 1  N: 1   | A |                                  insere A
cap: 2  N: 2   | A | B |                              cheio! dobra e copia
cap: 4  N: 3   | A | B | C |   |                      cheio! dobra e copia
cap: 4  N: 4   | A | B | C | D |                      cabe, sem cópia
cap: 8  N: 5   | A | B | C | D | E |   |   |   |    dobra, cop
```

Foram 5 inserções e apenas **4 realocações**, copiando $1 + 2 + 4 = 7$ elementos no total.

Note que a maior parte das inserções não copia nada.

Section 5

Análise Amortizada

O problema da análise de pior caso

Qual o custo de `insere` em `ConjuntoSeqDin` (ignorando a busca)?

- **Pior caso:** $O(N)$ — quando o vetor está cheio e precisa realocar
- **Melhor caso:** $O(1)$ — quando há espaço sobrando

Dizer que a inserção é $O(N)$ é **verdade, mas é pessimista demais**: esse pior caso não pode acontecer duas vezes seguidas!

O problema da análise de pior caso

Qual o custo de `insere` em `ConjuntoSeqDin` (ignorando a busca)?

- **Pior caso:** $O(N)$ — quando o vetor está cheio e precisa realocar
- **Melhor caso:** $O(1)$ — quando há espaço sobrando

Dizer que a inserção é $O(N)$ é **verdade, mas é pessimista demais**: esse pior caso não pode acontecer duas vezes seguidas!

A **análise amortizada** responde a outra pergunta: qual o custo *total* de uma **sequência** de n operações, dividido por n?

Método agregado: somando as cópias

Partindo de $cap = 1$, as realocações acontecem quando N vale $1, 2, 4, 8, \ldots$ — e cada uma copia exatamente esse tanto de elementos.

Após n inserções, o total de elementos copiados é:

$$1 + 2 + 4 + \cdots + 2^k = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 < 2n$$

pois $2^k < n$. Ou seja: o **trabalho total** de realocação em n inserções é $O(n)$.

$$\text{custo amortizado por inserção} = \frac{O(n)}{n} = O(1)$$

O crescimento dobrado é "de graça", no agregado.

Método contábil (o banqueiro)

Uma intuição alternativa: cada inserção paga **3 moedas**.

- 1 moeda paga a escrita do próprio elemento
- 2 moedas ficam **guardadas** com ele, como crédito

Quando o vetor dobra de *cap* para $2\,cap$, é preciso copiar *cap* elementos.

Método contábil (o banqueiro)

Uma intuição alternativa: cada inserção paga **3 moedas**.

- 1 moeda paga a escrita do próprio elemento
- 2 moedas ficam **guardadas** com ele, como crédito

Quando o vetor dobra de cap para $2\,cap$, é preciso copiar cap elementos.

Mas a metade "nova" do vetor ($cap/2$ elementos inseridos desde a última realocação) tem $2 \times cap/2 = cap$ moedas guardadas — **exatamente o que a cópia custa**.

O crédito nunca fica negativo, logo o custo amortizado é $O(1)$ por operação.

E se crescêssemos de 1 em 1?

Suponha `redimensiona(cap + 1)` a cada inserção. O total copiado seria:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

estratégia	trabalho total	amortizado
$cap + 1$	$O(n^2)$	$O(n)$
$cap + 100$	$O(n^2)$	$O(n)$
$2 \times cap$	$O(n)$	$\mathbf{O(1)}$
$1{,}5 \times cap$	$O(n)$	$\mathbf{O(1)}$

A lição: crescer por um valor *aditivo* é quadrático; crescer por um fator *multiplicativo* é linear. Qualquer fator > 1 serve — o `std::vector` costuma usar 2 ou 1,5.

A armadilha da redução (*thrashing*)

E por que reduzir em $N \leq cap/4$, e não em $N \leq cap/2$?

Imagine reduzir pela metade assim que a ocupação cair à metade, com $cap = 8$ e $N = 4$:

```
insere -> N=5, cheio? não... mas insere de novo -> dobra para
remove -> N=8 <= 16/4? não ... remove ate N=8 -> reduz para 8
```

Com o gatilho em $cap/2$, uma sequência alternada de `insere`/`remove` no ponto crítico **realoca a cada operação**: custo $O(n^2)$ no total!

Deixando uma **folga** (dobra em $N = cap$, reduz em $N = cap/4$), toda realocação é seguida de pelo menos $cap/2$ operações baratas.

Amortizado $\neq$ médio

Três conceitos que os alunos costumam confundir:

- **Caso médio:** depende de uma *hipótese probabilística* sobre a entrada (ex.: "as chaves chegam em ordem aleatória")
- **Amortizado:** é uma *garantia determinística* sobre a sequência inteira — vale mesmo para a entrada mais adversária possível
- **Pior caso de uma operação:** continua sendo $O(N)$! Uma inserção específica *pode* travar

Onde isso importa: *em sistemas de tempo real, uma latência $O(N)$ esporádica pode ser inaceitável, mesmo com ótimo custo amortizado.*

Cuidado: agregado com ponteiro cru

O `ConjuntoSeqDin` guarda um `char*`. Portanto, a cópia padrão do agregado é **rasa** (*shallow*):

```
ConjuntoSeqDin a;  a.cria();  a.insere('X');
ConjuntoSeqDin b = a;   // copia o PONTEIRO, não o vetor!
a.libera();
b.libera();             // ERRO: double free (dupla liberação)
```

Solução no curso: trate o agregado sempre por referência, ou use *smart pointers*, como fizemos nas pilhas:

```
template<typename T> using uptr = std::unique_ptr<T>;
uptr<char[]> v;    // libera sozinho, e proíbe cópia acidental
```

Tarefa: Conjunto Sequencial Dinâmico

Desafio 1: implemente `ConjuntoSeqDin` completo (`cria`, `libera`, `redimensiona`, `pertence`, `insere`, `remove`, `tamanho`) e teste com 10^6 inserções.

Desafio 2: instrumente o código com um contador global de **cópias**. Insira $n = 2^{20}$ elementos e verifique empiricamente que o total de cópias fica abaixo de $2n$.

Desafio 3: troque a estratégia para `cap + 1` e meça o tempo. Faça o gráfico de tempo $\times$ n para as duas estratégias.

Tarefa: Conjunto Sequencial Dinâmico

Desafio 1: implemente `ConjuntoSeqDin` completo (`cria`, `libera`, `redimensiona`, `pertence`, `insere`, `remove`, `tamanho`) e teste com 10^6 inserções.

Desafio 2: instrumente o código com um contador global de **cópias**. Insira $n = 2^{20}$ elementos e verifique empiricamente que o total de cópias fica abaixo de $2n$.

Desafio 3: troque a estratégia para `cap + 1` e meça o tempo. Faça o gráfico de tempo $\times$ n para as duas estratégias.

Pergunta final: *o custo amortizado da inserção é $O(1)$... mas qual é o custo real do `insere` no nosso conjunto, considerando o `pertence`?*

Tarefa: Conjunto Sequencial Dinâmico

Desafio 1: implemente `ConjuntoSeqDin` completo (`cria`, `libera`, `redimensiona`, `pertence`, `insere`, `remove`, `tamanho`) e teste com 10^6 inserções.

Desafio 2: instrumente o código com um contador global de **cópias**. Insira $n = 2^{20}$ elementos e verifique empiricamente que o total de cópias fica abaixo de $2n$.

Desafio 3: troque a estratégia para `cap + 1` e meça o tempo. Faça o gráfico de tempo $\times$ n para as duas estratégias.

Pergunta final: *o custo amortizado da inserção é $O(1)$... mas qual é o custo real do `insere` no nosso conjunto, considerando o `pertence`?*

Continua $O(N)$! A realocação nunca foi o gargalo — ***a busca é****.*

Tarefa: Conjunto Sequencial Dinâmico

Desafio 1: implemente `ConjuntoSeqDin` completo (`cria`, `libera`, `redimensiona`, `pertence`, `insere`, `remove`, `tamanho`) e teste com 10^6 inserções.

Desafio 2: instrumente o código com um contador global de **cópias**. Insira $n = 2^{20}$ elementos e verifique empiricamente que o total de cópias fica abaixo de $2n$.

Desafio 3: troque a estratégia para `cap + 1` e meça o tempo. Faça o gráfico de tempo $\times$ n para as duas estratégias.

Pergunta final: *o custo amortizado da inserção é $O(1)$... mas qual é o custo real do `insere` no nosso conjunto, considerando o `pertence`?*

*Continua $O(N)$! A realocação nunca foi o gargalo — **a busca é**.*

*E se o vetor estivesse **ordenado**?*

Section 6

Algoritmos de Busca

Busca Linear (sequencial)

Já a implementamos em `pertence`: percorre e compara.

```
template<typename T>
auto busca_linear(const T* v, int n, T x) -> std::optional<int
    for (int i = 0; i < n; i++)
        if (v[i] == x)
            return i;           // devolve a posição
    return std::nullopt;        // ausente
}
```

- **Não exige nenhuma pré-condição** sobre os dados
- Pior caso: $O(n)$ comparações
- É a **única opção** em dados desordenados

Busca Binária

Se o vetor está **ordenado**, cada comparação descarta *metade* do espaço de busca:

```
template<typename T>
auto busca_binaria(const T* v, int n, T x) -> std::optional<in
    int ini = 0, fim = n - 1;
    while (ini <= fim) {
        int meio = ini + (fim - ini) / 2;     // evita overflow!
        if (v[meio] == x)       return meio;
        else if (v[meio] < x)   ini = meio + 1;    // descarta esq
        else                    fim = meio - 1;    // descarta dir
    }
    return std::nullopt;
}
```

Cuidado: `(ini + fim) / 2` pode **estourar** o `int` para vetores grandes — um bug famoso, presente por anos na biblioteca padrão de Java.

Busca Binária: intuição

Busca por G em um vetor ordenado de 8 elementos:

```
| A | C | E | G | J | M | P | R |
  0   1   2   3   4   5   6   7
              ^ meio=3  -> v[3]==G, achou!
```

Busca por M:

```
| A | C | E | G | J | M | P | R |     meio=3: v[3]='G' < 'M'
  0   1   2   3   4   5   6   7       -> ini=4

|                 | J | M | P | R |   meio=5: v[5]=='M', achou
                    4   5   6   7
```

- $8 \to 4 \to 2 \to 1$: são $\log_2 n$ passos
- Com $n = 10^6$, a busca linear faz até 10^6 comparações; a binária, **20**

Busca Binária: pré-condição e invariante

A busca binária só é **correta** se o vetor estiver ordenado.

- **Pré-condição:** `v[0] <= v[1] <= ... <= v[n-1]`
- **Invariante do laço:** *se x está no vetor, então x está em $v[ini..fim]$*
- **Terminação:** o intervalo `fim - ini` diminui estritamente a cada iteração

Em C++26, essa pré-condição pode ser escrita no próprio código, com *contracts*:

```
template<typename T>
auto busca_binaria(const T* v, int n, T x) -> std::optional<in
    pre (std::is_sorted(v, v + n))
{
    // ...
}
```

Reflexão: *um algoritmo rápido com pré-condição violada não é rápido — é apenas errado.*

Busca na Biblioteca Padrão

Na STL, basta `import std;` — e note que as versões prontas trabalham com **intervalos** (`ranges`):

```
import std;

int main() {
    std::vector<char> v = {'A', 'C', 'E', 'G', 'J'};
    println("{}", std::ranges::binary_search(v, 'G'));  // true
    // lower_bound: 1a posição onde 'F' poderia ser inserido
    auto it = std::ranges::lower_bound(v, 'F');
    println("{}", it - v.begin());                      // 3
    // find: busca linear, funciona em qualquer vetor
    println("{}", std::ranges::find(v, 'Z') == v.end());// true
    return 0;
}
```

O `lower_bound` será nossa ferramenta favorita no conjunto ordenado!

Section 7

Algoritmos de Ordenação

Por que ordenar?

Ordenar é *caro* ($O(n \log n)$), mas **paga-se uma vez** e barateia tudo depois:

- busca em $O(\log n)$ em vez de $O(n)$
- duplicatas ficam vizinhas (detecção em uma passada)
- mínimo e máximo nas pontas
- união e interseção de conjuntos em tempo linear

Veremos três algoritmos $O(n^2)$ (simples e didáticos) e dois $O(n \log n)$.

Selection Sort (seleção)

Ideia: selecione o menor elemento restante e coloque-o na posição correta.

```
template<typename T>
auto selection_sort(T* v, int n) -> void {
    for (int i = 0; i < n - 1; i++) {
        int menor = i;
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[menor])
                menor = j;
        std::swap(v[i], v[menor]);
    }
}
```

- Comparações: sempre $\frac{n(n-1)}{2}$, ou seja, $O(n^2)$ **em todo caso**
- Trocas: apenas $n-1$ — útil quando **copiar é caro**

Insertion Sort (inserção)

Ideia: mantenha `v[0..i-1]` ordenado e **insira** `v[i]` no lugar certo, deslocando os maiores para a direita.

```
template<typename T>
auto insertion_sort(T* v, int n) -> void {
    for (int i = 1; i < n; i++) {
        T x = v[i];
        int j = i - 1;
        while (j >= 0 && v[j] > x) {
            v[j + 1] = v[j];          // desloca para a direita
            j--;
        }
        v[j + 1] = x;                 // encaixa
    }
}
```

- Pior caso $O(n^2)$, **melhor caso** $O(n)$ (vetor já ordenado)
- É *estável* e excelente para n pequeno ou vetores *quase* ordenados

Insertion Sort: a ponte com o Conjunto

Observe com atenção o laço interno do Insertion Sort.

Ele é exatamente a operação *"inserir um elemento em um vetor já ordenado"*.

```
| A | C | E | J | M |  <- inserir 'G'
  0   1   2   3   4

| A | C | E |   | J | M |   desloca J e M
| A | C | E | G | J | M |   encaixa G
```

Insertion Sort: a ponte com o Conjunto

Observe com atenção o laço interno do Insertion Sort.

Ele é exatamente a operação *"inserir um elemento em um vetor já ordenado"*.

```
| A | C | E | J | M |  <- inserir 'G'
  0   1   2   3   4

| A | C | E |   | J | M |   desloca J e M
| A | C | E | G | J | M |   encaixa G
```

Ou seja: **o Insertion Sort é o nosso Conjunto Ordenado sendo construído, uma inserção por vez.** Guarde isso — é o fio condutor do resto da aula.

Bubble Sort (bolha)

Ideia: compare vizinhos e troque; os maiores "borbulham" para o fim.

```
template<typename T>
auto bubble_sort(T* v, int n) -> void {
    bool trocou = true;
    for (int i = 0; i < n - 1 && trocou; i++) {
        trocou = false;
        for (int j = 0; j < n - 1 - i; j++)
            if (v[j] > v[j + 1]) {
                std::swap(v[j], v[j + 1]);
                trocou = true;
            }
    }
}
```

É o algoritmo mais fácil de escrever e o mais lento na prática (muitas trocas). Vale conhecê-lo pela **cultura** e pela facilidade de provar a corretude.

Merge Sort (intercalação)

Ideia: divisão e conquista — ordene as duas metades e **intercale**.

```
template<typename T>
auto merge_sort(std::vector<T>& v, int ini, int fim) -> void {
    if (fim - ini <= 1) return;                    // caso base
    int meio = ini + (fim - ini) / 2;
    merge_sort(v, ini, meio);                      // T(n/2)
    merge_sort(v, meio, fim);                      // T(n/2)
    intercala(v, ini, meio, fim);                  // O(n)
}
```

Recorrência: $T(n) = 2\,T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$

- **Garantia** de $O(n \log n)$ mesmo no pior caso, e é *estável*
- Custo: precisa de $O(n)$ de **memória auxiliar** para intercalar

Quick Sort (partição)

Ideia: escolha um **pivô**, particione (menores à esquerda, maiores à direita) e ordene as partes.

```
template<typename T>
auto quick_sort(std::vector<T>& v, int ini, int fim) -> void {
    if (fim - ini <= 1) return;
    int p = particiona(v, ini, fim);    // O(n)
    quick_sort(v, ini, p);
    quick_sort(v, p + 1, fim);
}
```

- Caso médio $O(n \log n)$, **in-place**, campeão na prática (cache!)
- Pior caso $O(n^2)$, com pivô mal escolhido (ex.: vetor já ordenado)

Desafio: *pesquise o pivô "mediana de três" e a* `std::introsort` *— como a biblioteca padrão evita o pior caso?*

Comparativo dos algoritmos de ordenação

algoritmo	melhor	médio	pior	memória	estável
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	não
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	sim
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	sim
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	sim
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	não

Limite inferior: qualquer ordenação *baseada em comparações* precisa de $\Omega(n \log n)$ comparações — há $n!$ permutações possíveis e $\log_2(n!) = \Theta(n \log n)$.

Ordenação na Biblioteca Padrão

```cpp
import std;

int main() {
    std::vector<int> v = {5, 3, 9, 1, 3};
    std::ranges::sort(v);                          // O(n log n)
    println("{}", v);                              // [1, 3, 3, 5,
    std::ranges::stable_sort(v);                   // preserva empa
    println("{}", std::ranges::is_sorted(v));      // true
    return 0;
}
```

Na prova e nas listas: *implemente à mão!* Em produção, use a STL — ela é testada, otimizada e provavelmente mais rápida que a nossa versão.

Section 8

Conjunto Ordenado

Conjunto Ordenado: a ideia

Mesmo TAD, **nova invariante**: o vetor está sempre ordenado.

1. $0 \leq N \leq MAX_N$
2. `v[i] != v[j]` para $i \neq j$ *(sem repetição)*
3. `v[0] < v[1] < ... < v[N-1]` ***(nova!** ordenado)*

Uma invariante mais forte **custa mais para manter**, mas **entrega mais**: agora `pertence` pode usar busca binária.

Implementação: Pertence e Insere

```
bool ConjuntoOrd1::pertence(char x) {
    return busca_binaria(v, N, x).has_value();    // O(log N)
}

bool ConjuntoOrd1::insere(char x) {
    int pos = lower_bound(v, N, x);    // O(log N): onde deveria
    if (pos < N && v[pos] == x)
        return false;                  // já existe
    for (int i = N; i > pos; i--)      // O(N): abre espaço!
        v[i] = v[i - 1];
    v[pos] = x;
    N++;
    return true;
}
```

Achamos o lugar em $O(\log N)$... e gastamos $O(N)$ para ***abrir espaço***.

O gargalo: o deslocamento

```
| A | C | E | J | M |          inserir 'G': lower_bound -> pos=
  0   1   2   3   4

| A | C | E |   | J | M |      desloca N-pos elementos  <- O(N)
| A | C | E | G | J | M |      encaixa                  <- O(1)
  0   1   2   3   4   5
```

operação	Conjunto Seq.	Conjunto Ord.
`pertence`	$O(N)$	$\mathbf{O}(\log \mathbf{N})$
`insere`	$O(N)$	$O(N)$
`remove`	$O(N)$	$O(N)$

Ganhamos muito na consulta e **não perdemos nada** na atualização — mas o $O(N)$ da inserção continua incomodando.

Definição do *Conceito* ConjuntoTAD em C++

Como fizemos com a Pilha, o *conceito* exige apenas as operações básicas — e vale tanto para ConjuntoSeq quanto para ConjuntoOrd:

```
template<typename Agregado, typename Tipo>
concept ConjuntoTAD = requires(Agregado a, Tipo t)
{
    // requer operação 'pertence' sobre tipo 'Tipo'
    { a.pertence(t) } -> std::same_as<bool>;
    // requer operação 'insere' sobre tipo 'Tipo'
    { a.insere(t)   } -> std::same_as<bool>;
    // requer operação 'remove' sobre tipo 'Tipo'
    { a.remove(t)   } -> std::same_as<bool>;
    // requer operação 'tamanho'
    { a.tamanho()   } -> std::same_as<int>;
};
```

Verificando os conceitos com `static_assert`

```
// três implementações bem diferentes, o mesmo TAD...
static_assert(ConjuntoTAD<ConjuntoSeq1, char>);
static_assert(ConjuntoTAD<ConjuntoSeqDin, char>);
static_assert(ConjuntoTAD<ConjuntoOrd1, char>);
```

Importante: o conceito verifica a **sintaxe** (quais operações existem e com que tipos), mas **não** verifica as invariantes semânticas.

Nenhum `static_assert` percebe que o seu `insere` aceitou um elemento repetido. Para isso existem **testes** e, em C++26, os **contracts**.

Tarefa: Conjunto Sequencial Genérico

Agora que o conjunto de char está claro, uma implementação **genérica** pode ser feita com templates, inclusive para o limite de capacidade:

```
template<typename T, int MAX_N>
class ConjuntoSeq
{
public:
    T v[MAX_N];                  // elementos do conjunto
    int N;                       // num. de elementos
    auto cria()    -> void;      // inicializa agregado
    auto libera()  -> void;      // finaliza agregado
    auto pertence(T x) -> bool;
    auto insere(T x)   -> bool;
    auto remove(T x)   -> bool;
    auto tamanho()     -> int;
};

// verifica se agregado ConjuntoSeq satisfaz conceito Conjunto
```

Conjuntos na Biblioteca Padrão

```cpp
import std;

int main() {
    std::set<char> a;                  // árvore balanceada: O(log n
    a.insert('C'); a.insert('A'); a.insert('C');
    println("{}", a.size());           // 2 (sem repetição!)
    println("{}", a.contains('A'));

    std::unordered_set<char> b;        // tabela hash: O(1) amortiza
    std::flat_set<char> c;             // C++23: vetor ordenado!
    return 0;
}
```

Note o `std::flat_set`: é *exatamente* o nosso `ConjuntoOrd` — consulta rápida, memória contígua, e a mesma inserção $O(n)$. A biblioteca padrão o adotou pelo desempenho de *cache*.

Section 9

E depois? Vetores que se reorganizam

O incômodo que sobrou

Recapitulando a nossa tabela:

estrutura	`pertence`	`insere`
vetor não ordenado	$O(N)$	$O(N)$
vetor ordenado	$O(\log N)$	$O(N)$
`std::set` (árvore)	$O(\log N)$	$O(\log N)$

A árvore vence na inserção... mas **espalha os nós pela memória**, perdendo o *cache*, e gasta ponteiros em cada elemento (como vimos na Pilha Encadeada).

O incômodo que sobrou

Recapitulando a nossa tabela:

estrutura	`pertence`	`insere`
vetor não ordenado	$O(N)$	$O(N)$
vetor ordenado	$O(\log N)$	$O(N)$
`std::set` (árvore)	$O(\log N)$	$O(\log N)$

A árvore vence na inserção... mas **espalha os nós pela memória**, perdendo o *cache*, e gasta ponteiros em cada elemento (como vimos na Pilha Encadeada).

A pergunta da próxima aula: *é possível manter os dados em um vetor contíguo e* ***mesmo assim*** *inserir em tempo logarítmico?*

Ideia: pagar caro, mas raramente

E se, em vez de deslocar tudo a cada inserção, deixássemos **espaços vazios** propositais no vetor, e só **reconstruíssemos** um trecho quando ele ficasse desequilibrado demais?

```
| A |   | C | E |   | J |   | M |     inserir 'G' -> usa uma f
| A |   | C | E | G | J |   | M |
```

- A maioria das inserções custa $O(\log n)$ (ou até $O(1)$)
- De vez em quando, uma inserção "azarada" custa $O(n)$ e reorganiza um trecho
- **Na média** (custo *amortizado*), o resultado é excelente

O bode expiatório (*scapegoat*)

Essa é a ideia das **árvores scapegoat** (*bode expiatório*) e das suas versões **implícitas em vetor**:

- não se guarda fator de balanceamento em cada nó (nada de ponteiros extras!)
- quando uma inserção deixa um trecho "torto demais", procura-se o **culpado**: o ancestral responsável pelo desequilíbrio
- esse trecho é **reconstruído inteiro**, perfeitamente balanceado, em $O(k)$

O nome vem daí: elege-se um *bode expiatório* e paga-se a conta de uma só vez.

Resultado: busca $O(\log n)$ garantida, inserção $O(\log n)$ **amortizada**, memória contígua, zero ponteiros.

Para onde vamos

Os próximos passos do curso montam essa escada:

1. **Vetor ordenado** (hoje) — busca binária $O(\log n)$, inserção $O(n)$
2. **Árvores de busca (BST)** — inserção $O(\log n)$, mas degeneram
3. **Árvores balanceadas (AVL)** — garantia $O(\log n)$, com custo de rotações
4. **Layout implícito** — a árvore *dentro* do vetor, sem ponteiros
5. **Reconstrução amortizada** (*scapegoat*) — o melhor dos dois mundos

Curiosidade: *esse é tema de pesquisa atual, e não apenas de livro-texto — há estruturas recentes explorando exatamente esse espaço de projeto.*

Section 10

Análise de Complexidade

Conjuntos, Busca e Ordenação: Revisão Geral

- Quais são as invariantes de um Conjunto? E de um Conjunto **Ordenado**?
- Por que `insere` custa $O(N)$ mesmo em um conjunto não ordenado?
- Qual a pré-condição da busca binária? O que acontece se ela for violada?
- Por que $(ini + fim)/2$ é perigoso?
- Qual algoritmo de ordenação usar para um vetor *quase* ordenado? Por quê?
- Por que nenhuma ordenação por comparação pode ser melhor que $O(n \log n)$?
- Qual a relação entre o Insertion Sort e a inserção em um conjunto ordenado?
- O que significa dizer que um custo é **amortizado**?

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.
- Galperin, I.; Rivest, R.L. Scapegoat Trees. SODA, 1993.

Section 11

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)