Estrutura de Dados para Competição

Prova 1

September 30, 2025

Contents

1 Guia Prático: Vetores (`std::vector`)

Vetores são arrays dinâmicos que podem crescer e encolher de tamanho. São uma das estruturas de dados mais fundamentais e úteis na STL.

1.1 Header

Para usar `std::vector`, você precisa incluir o seguinte header:

```
#include <vector>
```

1.2 Declaração

Existem várias formas de declarar um vetor:

```
// Vetor vazio de inteiros
std::vector<int> v1;

// Vetor de strings com 10 elementos, todos inicializados como ""
std::vector<std::string> v2(10);

// Vetor de doubles com 5 elementos, todos inicializados com o valor 3.14
std::vector<double> v3(5, 3.14);

// Vetor inicializado com valores espec ficos
std::vector<int> v4 = {10, 20, 30, 40, 50};

// Copiando um vetor existente
std::vector<int> v5 = v4;
```

1.3 Principais Funções e Operações

1.4 Iterando sobre um Vetor

A forma mais comum e moderna de percorrer um vetor é com um *range-based for loop*.

```
std::vector<int> numeros = {1, 2, 3, 4, 5};

// Para ler os valores
for (int numero : numeros) {
    std::cout << numero << " ";
}
// Sa da: 1 2 3 4 5

// Para modificar os valores (usando refer ncia &)
for (int &numero : numeros) {
    numero *= 2;
}
// Agora o vetor 'numeros' cont m {2, 4, 6, 8, 10}
```

1.5 Exemplo Prático: Soma de Elementos

Problema: Leia um número N, seguido por N inteiros. Armazene-os em um vetor, calcule a soma de todos os elementos e imprima o resultado.

Solução em C++:

```
#include <iostream>
#include <vector>
#include <numeric> // Para std::accumulate

int main() {
```

Operação	Descrição	Exemplo	Complexidade
`v.push_back(valor)`	Adiciona um elemento no final do vetor.	`v.push_back(60);`	O(1) amortizado
`v.pop_back()`	Remove o último elemento.	`v.pop_back();`	O(1)
`v.size()`	Retorna o número de elementos no vetor.	`int tam = v.size();`	O(1)
`v.empty()`	Retorna `true` se o vetor estiver vazio, `false` caso contrário.	`if (v.empty()) {...}`	O(1)
`v[i]`	Acessa o elemento na posição `i` (sem verificação de limites).	`int primeiro = v[0];`	O(1)
`v.at(i)`	Acessa o elemento na posição `i` (com verificação de limites).	`int segundo = v.at(1);`	O(1)
`v.front()`	Retorna uma referência para o primeiro elemento.	`int primeiro = v.front();`	O(1)
`v.back()`	Retorna uma referência para o último elemento.	`int ultimo = v.back();`	O(1)
`v.clear()`	Remove todos os elementos do vetor.	`v.clear();`	O(N)
`sort(v.begin(), v.end())`	Ordena o vetor (requer `#include <algorithm>`).	`sort(v.begin(), v.end());`	O(N log N)

Table 1: Principais operações com `std::vector`.

```
6     // Acelera a leitura e escrita
7     std::ios_base::sync_with_stdio(false);
8     std::cin.tie(NULL);
9
10    int n;
11    std::cout << "Digite o numero de elementos: ";
12    std::cin >> n;
13
14    std::vector<int> meu_vetor(n);
15    long long soma = 0;
16
```

```
    std::cout << "Digite os " << n << " elementos: ";
    for (int i = 0; i < n; ++i) {
        std::cin >> meu_vetor[i];
        soma += meu_vetor[i];
    }

    // Outra forma de calcular a soma usando a biblioteca <numeric>
    // long long soma_alternativa = std::accumulate(meu_vetor.begin(), meu_vetor.end(), 0LL);

    std::cout << "O vetor eh: ";
    for (int i = 0; i < n; ++i) {
        std::cout << meu_vetor[i] << " ";
    }
    std::cout << std::endl;

    std::cout << "A soma dos elementos eh: " << soma << std::endl;

    return 0;
}
```

2 Guia Prático: Pilhas (`std::stack`)

Pilhas são uma estrutura de dados que opera no princípio **LIFO (Last-In, First-Out)**. O último elemento a ser inserido é o primeiro a ser removido. Pense em uma pilha de pratos.

A `std::stack` na STL é um "adaptador de contêiner", o que significa que ela usa outra estrutura por baixo (por padrão, `std::deque`) para gerenciar os dados.

2.1 Header

Para usar `std::stack`, inclua o seguinte header:

```
#include <stack>
```

2.2 Declaração

A declaração é simples e você só precisa especificar o tipo de dado que a pilha irá armazenar.

```
// Pilha de inteiros
std::stack<int> s1;

// Pilha de strings
std::stack<std::string> s2;
```

2.3 Principais Funções e Operações

A `std::stack` tem um conjunto de funções bem direto ao ponto, seguindo o princípio LIFO.

Operação	Descrição	Exemplo	Complexidade
`s.push(valor)`	Insere um elemento no topo da pilha.	`s.push(10);`	O(1)
`s.pop()`	Remove o elemento do topo da pilha. **Não retorna o valor!**	`s.pop();`	O(1)
`s.top()`	Retorna uma referência para o elemento no topo da pilha.	`int topo = s.top();`	O(1)
`s.size()`	Retorna o número de elementos na pilha.	`int tam = s.size();`	O(1)
`s.empty()`	Retorna `true` se a pilha estiver vazia, `false` caso contrário.	`if (s.empty()) {...}`	O(1)

Table 2: Principais operações com `std::stack`.

Importante: Para acessar e depois remover o elemento do topo, você precisa de duas operações:

```
if (!minha_pilha.empty()) {
    int valor_do_topo = minha_pilha.top(); // 1. Acessa
    minha_pilha.pop();                     // 2. Remove
    // Agora pode usar 'valor_do_topo'
}
```

Tentar usar `.top()` ou `.pop()` em uma pilha vazia resulta em comportamento indefinido (geralmente, um erro em tempo de execução). Sempre verifique com `.empty()` antes.

2.4 Aplicações Comuns

- **Verificação de parênteses/chaves/colchetes balanceados:** A aplicação clássica.
- **Navegação:** Botão de "voltar" em navegadores ou editores de texto.

- **Algoritmos:** Busca em profundidade (DFS) pode ser implementada com uma pilha.
- **Avaliação de expressões:** Converter expressões infixas para pós-fixas e calculá-las.

2.5 Exemplo Prático: Parênteses Balanceados

Problema: Dada uma string contendo apenas os caracteres `(`, `)`, `{`, `}`, `[` e `]`, determine se a string de entrada é válida. Uma string de entrada é válida se:

1. Os colchetes abertos devem ser fechados pelo mesmo tipo de colchetes.
2. Os colchetes abertos devem ser fechados na ordem correta.

Solução em C++:

```cpp
#include <iostream>
#include <stack>
#include <string>

bool estaoBalanceados(std::string expressao) {
    std::stack<char> pilha;

    for (char c : expressao) {
        // Se for um caractere de abertura, empilha
        if (c == '(' || c == '{' || c == '[') {
            pilha.push(c);
        }
        // Se for um caractere de fechamento
        else if (c == ')' || c == '}' || c == ']') {
            // Se a pilha estiver vazia, nao ha nada para fechar. Erro.
            if (pilha.empty()) {
                return false;
            }
            // Se o topo nao corresponder ao fechamento. Erro.
            char topo = pilha.top();
            if ((c == ')' && topo != '(') ||
                (c == '}' && topo != '{') ||
                (c == ']' && topo != '[')) {
                return false;
            }
            // Se correspondeu, desempilha
            pilha.pop();
        }
    }

    // No final, a pilha deve estar vazia para ser balanceada.
    return pilha.empty();
}

int main() {
    std::string s1 = "()[]{}";
    std::string s2 = "([)]";
    std::string s3 = "{[]}";
    std::string s4 = "]";

    std::cout << s1 << ": " << (estaoBalanceados(s1) ? "Balanceado" : "Nao Balanceado
    ") << std::endl;
    std::cout << s2 << ": " << (estaoBalanceados(s2) ? "Balanceado" : "Nao Balanceado
    ") << std::endl;
    std::cout << s3 << ": " << (estaoBalanceados(s3) ? "Balanceado" : "Nao Balanceado
    ") << std::endl;
    std::cout << s4 << ": " << (estaoBalanceados(s4) ? "Balanceado" : "Nao Balanceado
    ") << std::endl;
```

```
45
46      return 0;
47  }
```

3 Guia Prático: Pilhas Monótonas

Uma Pilha Monótona é uma técnica/padrão de algoritmo que usa uma `std::stack` para manter uma sequência de elementos em ordem estritamente crescente ou decrescente.

É uma ferramenta poderosa para resolver problemas que envolvem encontrar o "próximo" ou "anterior" elemento maior/menor para cada elemento em um array. A principal vantagem é que ela faz isso em tempo linear, **O(N)**.

3.1 Ideia Principal

A pilha monótona processa os elementos de uma sequência (geralmente um `std::vector`). Para cada elemento, ela ajusta o conteúdo da pilha para manter a propriedade monotônica.

- **Pilha Monotônica Crescente:** Os elementos na pilha estão sempre em ordem crescente, da base para o topo (`base <= ... <= topo`). É útil para encontrar o **próximo elemento menor** ou o **elemento anterior menor**.
- **Pilha Monotônica Decrescente:** Os elementos na pilha estão sempre em ordem decrescente, da base para o topo (`base >= ... >= topo`). É útil para encontrar o **próximo elemento maior** ou o **elemento anterior maior**.

O truque: Quando um novo elemento `x` vai ser inserido, a pilha remove (dá `pop`) todos os elementos do topo que violam a propriedade monotônica em relação a `x`. O elemento que está no topo *depois* dos `pop`s (se houver) é o elemento anterior que satisfaz a condição. O elemento `x` que *causou* os `pop`s é o próximo elemento que satisfaz a condição para todos os elementos que foram removidos.

3.2 Implementação (Template)

O padrão de código para encontrar o "Próximo Elemento Maior" (usando uma pilha monótona decrescente) geralmente se parece com isto:

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_map> // Ou um vector para o resultado

// nums: o array de entrada
// retorna: um array onde resultado[i]    o pr ximo elemento maior que nums[i]
std::vector<int> proximoElementoMaior(const std::vector<int>& nums) {
    int n = nums.size();
    std::vector<int> resultado(n, -1); // Inicializa com -1 (sem pr ximo maior)
    std::stack<int> pilha; // Armazena  NDICES , n o valores!

    // Itera sobre o array
    for (int i = 0; i < n; ++i) {
        // Enquanto a pilha n o estiver vazia e o elemento atual for MAIOR
        // que o elemento no  ndice  do topo da pilha...
        while (!pilha.empty() && nums[i] > nums[pilha.top()]) {
            // ...isso significa que nums[i]    o "pr ximo elemento maior"
            // para o elemento no  ndice  pilha.top().
            int indice_anterior = pilha.top();
            resultado[indice_anterior] = nums[i];
            pilha.pop();
        }
        // Empilha o  NDICE  do elemento atual.
        pilha.push(i);
    }
    // Elementos cujos  ndices  ainda est o na pilha n o t m um pr ximo maior.
    // Como j  inicializamos o resultado com -1, n o precisamos fazer nada.
```

```
29
30      return resultado;
31  }
```

Por que armazenar índices? Armazenar índices em vez de valores permite saber a posição original do elemento, o que é crucial para preencher o array de resultado corretamente.

3.3 Aplicações Comuns

- **Next/Previous Greater/Smaller Element:** O caso de uso canônico.
- **Largest Rectangle in Histogram:** Encontrar a maior área de um retângulo em um histograma. A pilha monótona ajuda a encontrar, para cada barra, a primeira barra menor à esquerda e à direita.
- **Stock Span Problem:** Para cada dia, encontrar o número de dias consecutivos anteriores em que o preço da ação foi menor ou igual.
- Problemas de "range" onde um elemento `nums[i]` é o mínimo ou máximo.

3.4 Exemplo Prático: Next Greater Element I (LeetCode 496)

Problema: Você recebe dois arrays, `nums1` e `nums2`, onde `nums1` é um subconjunto de `nums2`. Para cada elemento em `nums1`, encontre o primeiro elemento maior à sua direita em `nums2`.

Exemplo: `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`.

- Para `4` em `nums1`, não há elemento maior à sua direita em `nums2`. Resultado: -1.
- Para `1` em `nums1`, o próximo maior é `3` em `nums2`. Resultado: 3.
- Para `2` em `nums1`, não há elemento maior à sua direita em `nums2`. Resultado: -1.

Saída esperada: `[-1, 3, -1]`

Solução em C++: A estratégia é pré-calcular o "próximo elemento maior" para *todos* os elementos de `nums2` usando uma pilha monótona. Depois, usamos um mapa para consultar rapidamente os resultados para os elementos de `nums1`.

```
1  #include <iostream>
2  #include <vector>
3  #include <stack>
4  #include <unordered_map>
5
6  std::vector<int> nextGreaterElement(std::vector<int>& nums1, std::vector<int>& nums2)
       {
7      // Mapa para armazenar: {elemento -> proximo_maior}
8      std::unordered_map<int, int> proximo_maior_map;
9      // Pilha mon tona (decrescente) para valores
10     std::stack<int> pilha;
11
12     // Itera REVERSAMENTE por nums2 para encontrar o pr ximo maior
13     for (int i = nums2.size() - 1; i >= 0; --i) {
14         int num = nums2[i];
15         // Remove elementos da pilha que s o menores ou iguais ao n mero atual
16         while (!pilha.empty() && pilha.top() <= num) {
17             pilha.pop();
18         }
19
20         // Se a pilha n o estiver vazia, o topo    o pr ximo maior
21         // Se estiver vazia, n o h   pr ximo maior
22         proximo_maior_map[num] = pilha.empty() ? -1 : pilha.top();
23
```

```
        // Empilha o n mero atual para as pr ximas  itera  es
        pilha.push(num);
    }

    // Monta o resultado para nums1 consultando o mapa
    std::vector<int> resultado;
    for (int num : nums1) {
        resultado.push_back(proximo_maior_map[num]);
    }

    return resultado;
}
```

4 Guia Prático: Filas (`std::queue`)

Filas são uma estrutura de dados que opera no princípio **FIFO (First-In, First-Out)**. O primeiro elemento a ser inserido é o primeiro a ser removido. Pense em uma fila de pessoas esperando para serem atendidas.

Assim como a `std::stack`, a `std::queue` na STL é um "adaptador de contêiner" que usa outra estrutura por baixo (por padrão, `std::deque`).

4.1 Header

Para usar `std::queue`, inclua o seguinte header:

```
#include <queue>
```

4.2 Declaração

A declaração é direta, especificando o tipo de dado que a fila irá conter.

```
// Fila de inteiros
std::queue<int> q1;

// Fila de pares (muito comum em BFS)
std::queue<std::pair<int, int>> q2;
```

4.3 Principais Funções e Operações

Operação	Descrição	Exemplo	Complexidade
`q.push(valor)`	Insere um elemento no **final** da fila.	`q.push(10);`	O(1)
`q.pop()`	Remove o elemento da **frente** da fila. **Não retorna o valor!**	`q.pop();`	O(1)
`q.front()`	Retorna uma referência para o elemento na **frente** da fila.	`int primeiro = q.front();`	O(1)
`q.back()`	Retorna uma referência para o elemento no **final** da fila.	`int ultimo = q.back();`	O(1)
`q.size()`	Retorna o número de elementos na fila.	`int tam = q.size();`	O(1)
`q.empty()`	Retorna `true` se a fila estiver vazia, `false` caso contrário.	`if (q.empty()) { ... }`	O(1)

Table 3: Principais operações com `std::queue`.

Importante: Para acessar e depois remover o elemento da frente, você precisa de duas operações, assim como na pilha:

```
if (!minha_fila.empty()) {
    int primeiro_da_fila = minha_fila.front(); // 1. Acessa
    minha_fila.pop();                          // 2. Remove
    // Agora pode usar 'primeiro_da_fila'
}
```

Tentar usar `.front()`, `.back()` ou `.pop()` em uma fila vazia resulta em comportamento indefinido. Sempre verifique com `.empty()` antes.

4.4 Aplicações Comuns

- **Busca em Largura (BFS - Breadth-First Search):** A aplicação mais importante em grafos e árvores para encontrar o caminho mais curto em grafos não ponderados.
- **Simulações:** Modelar qualquer processo onde a ordem de chegada importa (fila de impressão, atendimento ao cliente, etc.).
- **Buffering:** Em redes e sistemas operacionais, filas são usadas para armazenar dados que chegam mais rápido do que podem ser processados.

4.5 Exemplo Prático: Simulador de Tarefas

Problema: Simule um processador que executa tarefas em ordem de chegada. Cada tarefa é representada por uma string. Adicione algumas tarefas à fila e processe-as uma por uma, imprimindo qual tarefa está sendo executada.

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <vector>

void processarFila(std::queue<std::string>& tarefas) {
    std::cout << "Iniciando processamento de " << tarefas.size() << " tarefas." << std::endl;
    int contador = 1;
    while (!tarefas.empty()) {
        std::string tarefa_atual = tarefas.front();
        tarefas.pop();
        std::cout << "  " << contador++ << ". Executando: " << tarefa_atual << std::endl;
    }
    std::cout << "Todas as tarefas foram processadas." << std::endl;
}

int main() {
    std::queue<std::string> fila_de_tarefas;
    fila_de_tarefas.push("Compilar o kernel");
    fila_de_tarefas.push("Baixar videos de gatos");
    fila_de_tarefas.push("Resolver problema do AtCoder");
    fila_de_tarefas.push("Fazer cafe");
    processarFila(fila_de_tarefas);
    return 0;
}
```

5 Guia Prático: Filas Monótonas (com `std::deque`)

Uma Fila Monótona (ou "Monotonic Queue") é um padrão de algoritmo usado para encontrar o elemento mínimo ou máximo em uma **janela deslizante (sliding window)** de tamanho fixo sobre um array, de forma muito eficiente. A complexidade total é **O(N)**.

A estrutura de dados ideal para implementar uma fila monótona é a `std::deque` (double-ended queue), pois ela permite inserções e remoções eficientes tanto no início quanto no final.

5.1 Header

Para usar `std::deque`, inclua o header:

```
#include <deque>
```

5.2 Ideia Principal

O objetivo é manter uma `deque` de **índices** dos elementos da janela atual. A propriedade chave é que os **valores** correspondentes a esses índices estarão sempre em ordem monotônica.

- **Fila Monotônica Decrescente (para achar o máximo):** Os valores `nums[deque.front()] >= nums[deque[1]] >= ...` são mantidos. O elemento na frente da deque (`deque.front()`) é sempre o índice do **maior** elemento na janela atual.
- **Fila Monotônica Crescente (para achar o mínimo):** Os valores `nums[deque.front()] <= nums[deque[1]] <= ...` são mantidos. O elemento na frente da deque (`deque.front()`) é sempre o índice do **menor** elemento na janela atual.

5.3 Exemplo Prático: Sliding Window Maximum (LeetCode 239)

Problema: Dado um array `nums` e um tamanho de janela `k`, encontre o valor máximo em cada janela deslizante de tamanho `k`.

Solução em C++:

```
#include <iostream>
#include <vector>
#include <deque>

std::vector<int> maxSlidingWindow(const std::vector<int>& nums, int k) {
    std::vector<int> resultado;
    // A deque armazena  NDICES  dos elementos de `nums`
    std::deque<int> dq;

    for (int i = 0; i < nums.size(); ++i) {
        // 1. Remover  ndices  que est o fora da janela atual
        if (!dq.empty() && dq.front() == i - k) {
            dq.pop_front();
        }

        // 2. Manter a propriedade monot nica (decrescente)
        // Remove da parte de tr s todos os  ndices  de elementos menores que o
    atual.
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
            dq.pop_back();
        }

        // Adiciona o  ndice  do elemento atual
        dq.push_back(i);

        // 3. Adicionar o m ximo da janela ao resultado
```

```
        // A janela est  completamente formada a partir do  ndice  k-1.
        if (i >= k - 1) {
            resultado.push_back(nums[dq.front()]);
        }
    }
    return resultado;
}

int main() {
    std::vector<int> nums = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;
    std::vector<int> res = maxSlidingWindow(nums, k);

    std::cout << "Maximos da janela deslizante: ";
    for (int val : res) {
        std::cout << val << " ";
    }
    std::cout << std::endl; // Saida: 3 3 5 5 6 7
    return 0;
}
```

6 Guia Prático: Árvores Binárias na STL (`set`, `map`, etc.)

Em programação competitiva, raramente você implementará uma árvore do zero. Em vez disso, você usará as estruturas de contêiner da STL que são implementadas como árvores binárias de busca auto-balanceadas (geralmente Árvores Rubro-Negras). Isso garante performance de **O(log N)** para a maioria das operações.

6.1 `std::set`

O `std::set` é um contêiner que armazena elementos **únicos** e **ordenados**.

- **Propriedades:** Não permite duplicatas. Os elementos são mantidos em ordem crescente.
- **Uso principal:** Manter uma coleção de itens únicos, verificar rapidamente a existência de um item, e iterar sobre eles em ordem.
- **Header:** `#include <set>`

6.1.1 Principais Funções

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> meu_set;

    // Inser  o: O(log N)
    meu_set.insert(30);
    meu_set.insert(10);
    meu_set.insert(50);
    meu_set.insert(10); // Ignorado, pois 10 j  existe

    // Itera  o (sempre em ordem!)
    std::cout << "Set: ";
    for (int val : meu_set) {
        std::cout << val << " "; // Sa da: 10 30 50
    }
    std::cout << std::endl;

    // Verifica  o de exist ncia: O(log N)
    if (meu_set.count(30)) { // count retorna 1 se existe, 0 se n o
        std::cout << "30 esta no set." << std::endl;
    }

    // Remo  o: O(log N)
    meu_set.erase(10);
    std::cout << "Apos remover 10: ";
    for (int val : meu_set) {
        std::cout << val << " "; // Sa da: 30 50
    }
    std::cout << std::endl;

    // Encontrar o primeiro elemento >= X (lower_bound)
    auto it = meu_set.lower_bound(35);
    if (it != meu_set.end()) {
        std::cout << "Lower bound de 35: " << *it << std::endl; // Sa da: 50
    }
    return 0;
}
```

6.2 `std::map`

O `std::map` armazena pares **chave-valor**, com chaves **únicas** e **ordenadas** pela chave.

- **Propriedades:** Cada chave é única. O mapa é ordenado pelas chaves.
- **Uso principal:** Associar um valor a uma chave (dicionário, frequência de elementos).
- **Header:** `#include <map>`

6.2.1 Principais Funções

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> idades;

    // Inser  o/Atualiza  o com []: O(log N)
    idades["Alice"] = 30;
    idades["Bob"] = 25;
    idades["Charlie"] = 30;
    idades["Alice"] = 31; // Atualiza o valor para a chave "Alice"

    // Acesso: O(log N)
    std::cout << "Idade de Bob: " << idades["Bob"] << std::endl;

    // Itera  o (ordenado pela chave!)
    for (const auto& par : idades) {
        std::cout << par.first << ": " << par.second << std::endl;
    }

    // Verifica  o de exist ncia
    if (idades.count("David")) {
        std::cout << "David existe." << std::endl;
    } else {
        std::cout << "David nao existe." << std::endl;
    }
    return 0;
}
```

6.3 `multiset` e `multimap`

- `std::multiset`: Igual ao `set`, mas permite elementos **duplicados**. Útil quando você precisa contar múltiplas ocorrências e ainda manter a ordem.
- `std::multimap`: Igual ao `map`, mas permite chaves **duplicadas**. O acesso com `[]` não funciona. Use `multimap.insert({chave, valor})`.

6.4 Árvores com Estatísticas de Ordem (Avançado)

Esta é uma ferramenta extremamente poderosa para problemas mais difíceis, disponível nas extensões GNU C++. Ela é uma árvore que, além de todas as operações do `set`, pode responder em **O(log N)**:

1. `order_of_key(k)`: Quantos elementos na árvore são estritamente menores que `k`?
2. `find_by_order(k)`: Qual é o k-ésimo menor elemento (indexado do zero)?

- **Headers e namespace:** `#include <ext/pb_ds/assoc_container.hpp>`, `#include <ext/pb_ds/tree_pol` `using namespace __gnu_pbds;`
- **Declaração:** `tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>` `ordered_set;`

6.4.1 Exemplo Prático

```cpp
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

typedef tree<int, null_type, std::less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;

int main() {
    ordered_set o_set;
    o_set.insert(1);
    o_set.insert(2);
    o_set.insert(4);
    o_set.insert(8);
    o_set.insert(16);

    // Qual o 2   menor elemento? (0-indexed)
    // find_by_order(k) -> retorna um iterador
    std::cout << "2o menor (k=1): " << *o_set.find_by_order(1) << std::endl; //
    Sa da: 2
    std::cout << "4o menor (k=3): " << *o_set.find_by_order(3) << std::endl; //
    Sa da: 8

    // Quantos elementos s o menores que 7?
    // order_of_key(k)
    std::cout << "Elementos < 7: " << o_set.order_of_key(7) << std::endl; // Sa da:
    3 (1, 2, 4)
    std::cout << "Elementos < 1: " << o_set.order_of_key(1) << std::endl; // Sa da:
    0

    return 0;
}
```

Atenção: `ordered_set` não faz parte do padrão C++, então pode não estar disponível em todos os compiladores (mas está nos juízes online mais comuns como Codeforces, AtCoder, etc.).

7 Guia Prático: Heaps Binárias (`std::priority_queue`)

Uma Heap Binária é uma estrutura de dados baseada em árvore que satisfaz a "propriedade do heap": em uma **max-heap**, para qualquer nó `P`, seu valor é maior ou igual aos valores de seus filhos. O maior elemento está sempre na raiz. Em uma **min-heap**, é o oposto, com o menor elemento na raiz.

Em C++, a maneira padrão de usar heaps é com a `std::priority_queue`, que é um adaptador de contêiner.

- **Uso principal:** Obter acesso rápido ao elemento de maior (ou menor) prioridade em uma coleção.
- **Complexidades:** Inserção (`push`) é O(log N), remoção (`pop`) é O(log N), e acesso ao topo (`top`) é O(1).
- **Header:** `#include <queue>`

7.1 Max-Heap (Fila de Prioridade Máxima)

Por padrão, `std::priority_queue` é uma max-heap.

```cpp
#include <iostream>
#include <queue>
#include <vector>

int main() {
    // Declara  o padr o de uma max-heap de inteiros
    std::priority_queue<int> max_pq;

    // Inserindo elementos
    max_pq.push(10);
    max_pq.push(30);
    max_pq.push(20);
    max_pq.push(5);

    std::cout << "Processando a max-heap:" << std::endl;
    while (!max_pq.empty()) {
        // .top() acessa o maior elemento (O(1))
        std::cout << "Topo: " << max_pq.top() << std::endl;
        // .pop() remove o maior elemento (O(log N))
        max_pq.pop();
    }
    // Sa da (em ordem): 30, 20, 10, 5
    return 0;
}
```

7.2 Min-Heap (Fila de Prioridade Mínima)

Para problemas como Dijkstra ou encontrar os menores elementos, você precisa de uma min-heap. A declaração é mais longa, mas o padrão é sempre o mesmo.

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <functional> // Para std::greater

int main() {
    // Declara  o de uma min-heap de inteiros
    // std::priority_queue<Tipo, Cont iner, Comparador>
    std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;

    min_pq.push(10);
```

```
    min_pq.push(30);
    min_pq.push(20);
    min_pq.push(5);

    std::cout << "Processando a min-heap:" << std::endl;
    while (!min_pq.empty()) {
        std::cout << "Topo: " << min_pq.top() << std::endl;
        min_pq.pop();
    }
    // Sa da (em ordem): 5, 10, 20, 30
    return 0;
}
```

7.3 Heap com Tipos Customizados (Pares e Structs)

É muito comum usar `priority_queue` com `std::pair` ou `structs`, por exemplo, para o algoritmo de Dijkstra `(distância, vértice)`.

7.3.1 Com `std::pair`

Por padrão, uma `priority_queue` de pares ordena pelo `first` elemento (max-heap). Para simular uma min-heap, um truque comum em competições é inserir o primeiro elemento com valor negativo.

```
#include <iostream>
#include <queue>
#include <utility> // Para std::pair

int main() {
    // Min-heap para pares (dist, vertice) usando o truque do negativo
    // A priority_queue vai ordenar pela maior dist ncia , ent o -5 > -10.
    std::priority_queue<std::pair<int, int>> pq_dijkstra;

    pq_dijkstra.push({-5, 1});  // Dist ncia 5, v rtice 1
    pq_dijkstra.push({-10, 2}); // Dist ncia 10, v rtice 2
    pq_dijkstra.push({-3, 0});  // Dist ncia 3, v rtice 0

    std::cout << "Processando min-heap de pares (com truque negativo):" << std::endl;
    while (!pq_dijkstra.empty()) {
        std::pair<int, int> topo = pq_dijkstra.top();
        pq_dijkstra.pop();
        std::cout << "Distancia: " << -topo.first << ", Vertice: " << topo.second <<
    std::endl;
    }
    // O topo ser  {-3, 0}, que corresponde    menor dist ncia real.
    return 0;
}
```

7.3.2 Com `struct` (Método mais limpo)

Você pode definir uma `struct` e sobrecarregar o operador `<`.

```
#include <iostream>
#include <queue>

struct Estado {
    int dist;
    int vertice;

    // Sobrecarga do operador < para que a priority_queue funcione como MIN-HEAP
    // Retorna true se 'this' tem MENOR prioridade que 'outro'
    // Para min-heap, queremos que o menor 'dist' tenha MAIOR prioridade.
```

```
    // Ent o , se 'this->dist'    MAIOR que 'outro.dist', 'this' tem MENOR prioridade
    .
    bool operator<(const Estado& outro) const {
        return dist > outro.dist; // Inverte a ordem padr o para min-heap
    }
};

int main() {
    std::priority_queue<Estado> pq_dijkstra;
    pq_dijkstra.push({5, 1});
    pq_dijkstra.push({10, 2});
    pq_dijkstra.push({3, 0});

    std::cout << "Processando min-heap de structs:" << std::endl;
    while (!pq_dijkstra.empty()) {
        Estado topo = pq_dijkstra.top();
        pq_dijkstra.pop();
        std::cout << "Distancia: " << topo.dist << ", Vertice: " << topo.vertice <<
    std::endl;
    }
    // O topo ser  {3, 0}
    return 0;
}
```

7.4 Exemplo Prático: K-ésimo Maior Elemento

Problema: Encontre o k-ésimo maior elemento em um array.

Solução: Use uma **min-heap** de tamanho `k`. Itere pelo array. Se a heap tiver menos de `k` elementos, apenas insira. Se a heap já tiver `k` elementos, compare o elemento atual `num` com o topo da min-heap (`min_pq.top()`, que é o menor dos `k` maiores vistos até agora). Se `num` for maior, remova o topo e insira `num`.

No final, o topo da min-heap será o k-ésimo maior elemento do array.

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional> // Para std::greater

int findKthLargest(const std::vector<int>& nums, int k) {
    std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;

    for (int num : nums) {
        if (min_pq.size() < k) {
            min_pq.push(num);
        } else if (num > min_pq.top()) {
            min_pq.pop();
            min_pq.push(num);
        }
    }

    return min_pq.top();
}

int main() {
    std::vector<int> nums = {3, 2, 1, 5, 6, 4};
    int k = 2;
    std::cout << k << "-esimo maior elemento: " << findKthLargest(nums, k) << std::
    endl; // Sa da: 5

    std::vector<int> nums2 = {3, 2, 3, 1, 2, 4, 5, 5, 6};
    k = 4;
```

```
28      std::cout << k << "-esimo maior elemento: " << findKthLargest(nums2, k) << std::
        endl; // Sa da: 4
29      return 0;
30  }
```

8 Guia Prático: Implementação de Árvores Binárias (de Busca)

Antes de usar as poderosas árvores da STL (`set`, `map`), é fundamental entender como uma Árvore Binária de Busca (Binary Search Tree - BST) funciona por baixo dos panos. A implementação manual solidifica os conceitos de ponteiros, recursão e as propriedades da árvore.

8.1 Definição e Propriedades

Uma **Árvore Binária** é uma estrutura de dados em que cada nó tem no máximo dois filhos, referenciados como filho da esquerda e filho da direita.

Uma **Árvore Binária de Busca (BST)** é uma árvore binária com as seguintes propriedades para qualquer nó `N`:

1. Todos os valores na subárvore esquerda de `N` são **menores** que o valor de `N`.
2. Todos os valores na subárvore direita de `N` são **maiores** que o valor de `N`.
3. Ambas as subárvores (esquerda e direita) também são árvores binárias de busca.

Esta propriedade permite buscas, inserções e remoções muito eficientes (em média O(log N)).

8.2 Estrutura do Nó (Node)

A base de uma árvore é o seu nó. Em C++, geralmente o definimos com uma `struct` ou `class`.

```
struct Node {
    int data;
    Node* left;
    Node* right;

    // Construtor para facilitar a cria  o de um novo n
    Node(int value) {
        data = value;
        left = nullptr; // ou NULL
        right = nullptr; // ou NULL
    }
};
```

8.3 Operações Fundamentais

As operações são geralmente implementadas de forma recursiva, aproveitando a natureza da estrutura.

8.3.1 Inserção

Para inserir um valor, começamos pela raiz. Se o valor for menor que o nó atual, vamos para a esquerda; se for maior, vamos para a direita. Repetimos até encontrar um lugar vago (`nullptr`).

```
Node* insert(Node* root, int value) {
    // Se a  rvore /sub rvore estiver vazia, criamos o novo n  aqui
    if (root == nullptr) {
        return new Node(value);
    }

    // Caso contr rio, descemos pela  rvore  recursivamente
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    // Se o valor j  existe, n o fazemos nada
```

```
14
15      return root;
16  }
```

8.3.2 Busca

A busca segue a mesma lógica da inserção para encontrar um valor.

```
Node* search(Node* root, int value) {
    // Se a raiz    nula ou o valor foi encontrado
    if (root == nullptr || root->data == value) {
        return root;
    }

    // Valor    maior, ent o est  na sub rvore direita
    if (value > root->data) {
        return search(root->right, value);
    }

    // Valor    menor, ent o est  na sub rvore esquerda
    return search(root->left, value);
}
```

8.3.3 Travessias (Traversals)

Existem 3 formas principais de visitar todos os nós de uma árvore (Busca em Profundidade - DFS):

a) In-order (Esquerda, Raiz, Direita) Visita os nós em ordem crescente de valor. É a travessia mais comum para uma BST.

```
void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    std::cout << root->data << " ";
    inorderTraversal(root->right);
}
```

b) Pre-order (Raiz, Esquerda, Direita) Útil para criar uma cópia da árvore.

```
void preorderTraversal(Node* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

c) Post-order (Esquerda, Direita, Raiz) Útil para deletar a árvore, pois os filhos são processados antes do pai.

```
void postorderTraversal(Node* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    std::cout << root->data << " ";
}
```

8.4 Exemplo Prático Completo

Vamos juntar tudo: criar uma árvore, inserir elementos, fazer as travessias e buscar um valor.

```
#include <iostream>

// Defini  es da struct Node e das fun  es insert, search, e traversals
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}

Node* search(Node* root, int value) {
    if (root == nullptr || root->data == value) {
        return root;
    }
    if (value > root->data) {
        return search(root->right, value);
    }
    return search(root->left, value);
}

void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    std::cout << root->data << " ";
    inorderTraversal(root->right);
}

void preorderTraversal(Node* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

void postorderTraversal(Node* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    std::cout << root->data << " ";
}


int main() {
```

```
    Node* root = nullptr; // A  rvore  come a vazia

    // Inserindo elementos
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Travessia In-order (deve imprimir em ordem ordenada)
    std::cout << "Travessia In-order: ";
    inorderTraversal(root); // Sa da: 20 30 40 50 60 70 80
    std::cout << std::endl;

    // Travessia Pre-order
    std::cout << "Travessia Pre-order: ";
    preorderTraversal(root); // Sa da: 50 30 20 40 70 60 80
    std::cout << std::endl;

    // Travessia Post-order
    std::cout << "Travessia Post-order: ";
    postorderTraversal(root); // Sa da: 20 40 30 60 80 70 50
    std::cout << std::endl;

    // Buscando um valor
    int valor_a_buscar = 40;
    Node* resultado = search(root, valor_a_buscar);
    if (resultado != nullptr) {
        std::cout << "Valor " << valor_a_buscar << " encontrado na arvore." << std::
    endl;
    } else {
        std::cout << "Valor " << valor_a_buscar << " NAO encontrado na arvore." <<
    std::endl;
    }

    valor_a_buscar = 99;
    resultado = search(root, valor_a_buscar);
    if (resultado != nullptr) {
        std::cout << "Valor " << valor_a_buscar << " encontrado na arvore." << std::
    endl;
    } else {
        std::cout << "Valor " << valor_a_buscar << " NAO encontrado na arvore." <<
    std::endl;
    }

    // A remo  o  mais complexa e omitida aqui para simplicidade,
    // mas envolve 3 casos: n  folha, n  com 1 filho, e n  com 2 filhos.
    // Para deletar a  rvore  inteira, uma travessia post-order  ideal.

    return 0;
}
```

Nota sobre Remoção: A remoção de um nó com dois filhos é a parte mais complexa. A estratégia padrão é encontrar o sucessor in-order (o menor nó na subárvore direita) ou o predecessor in-order (o maior nó na subárvore esquerda), copiar seu valor para o nó que está sendo removido e, em seguida, remover recursivamente esse sucessor/predecessor (que terá no máximo um filho, caindo em um caso mais simples).

9 Guia Prático: Funções Úteis da STL para Competição

Além dos contêineres, a STL oferece um arsenal de algoritmos e funções genéricas que operam sobre eles. Dominar essas funções é crucial para resolver problemas de forma rápida e eficiente.

9.1 Header `<algorithm>`

Este é o header mais importante. A maioria das funções aqui opera em intervalos definidos por iteradores (como `v.begin()` e `v.end()`).

9.1.1 `std::sort`

Ordena um intervalo. A complexidade é O(N log N).

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional> // Para std::greater

// Exemplo de uso em main()
int main() {
    std::vector<int> v = {5, 2, 8, 1, 9};

    // Ordena o padr o (crescente)
    std::sort(v.begin(), v.end());
    // v agora {1, 2, 5, 8, 9}
    std::cout << "Crescente: "; for(int x : v) std::cout << x << " "; std::cout << std::endl;

    // Ordena o decrescente
    std::sort(v.begin(), v.end(), std::greater<int>());
    // v agora {9, 8, 5, 2, 1}
    std::cout << "Decrescente: "; for(int x : v) std::cout << x << " "; std::cout << std::endl;
    return 0;
}
```

9.1.2 `std::binary_search`, `lower_bound`, `upper_bound`

Realizam busca binária em um intervalo **previamente ordenado**. Complexidade O(log N).

- `binary_search`: Retorna `true` ou `false` se o elemento existe.
- `lower_bound`: Retorna um iterador para o **primeiro elemento que não é menor** que o valor (ou seja, >= valor).
- `upper_bound`: Retorna um iterador para o **primeiro elemento que é maior** que o valor (ou seja, > valor).

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {10, 20, 30, 30, 40, 50};

    bool existe = std::binary_search(v.begin(), v.end(), 30); // true
    std::cout << "30 existe? " << (existe ? "Sim" : "Nao") << std::endl;

    auto it_lower = std::lower_bound(v.begin(), v.end(), 30);
```

```
    // aponta para o primeiro 30 ( ndice  2)
    int pos_lower = it_lower - v.begin(); // pos_lower = 2
    std::cout << "lower_bound de 30 esta na posicao: " << pos_lower << std::endl;

    auto it_upper = std::upper_bound(v.begin(), v.end(), 30);
    // aponta para 40 ( ndice  4)
    int pos_upper = it_upper - v.begin(); // pos_upper = 4
    std::cout << "upper_bound de 30 esta na posicao: " << pos_upper << std::endl;

    // A quantidade de elementos iguais a 30    pos_upper - pos_lower = 2
    std::cout << "Quantidade de 30s: " << pos_upper - pos_lower << std::endl;
    return 0;
}
```

9.1.3 `std::min_element`, `std::max_element`

Encontram o menor e o maior elemento em um intervalo. Retornam um iterador. Complexidade O(N).

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {5, 2, 8, 1, 9};
    auto it_min = std::min_element(v.begin(), v.end()); // aponta para 1
    auto it_max = std::max_element(v.begin(), v.end()); // aponta para 9
    std::cout << "Min: " << *it_min << ", Max: " << *it_max << std::endl;
    return 0;
}
```

9.1.4 `std::reverse`

Inverte a ordem dos elementos em um intervalo. Complexidade O(N).

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::reverse(v.begin(), v.end());
    // v agora    {5, 4, 3, 2, 1}
    std::cout << "Vetor invertido: ";
    for(int x : v) std::cout << x << " ";
    std::cout << std::endl;
    return 0;
}
```

9.1.5 `std::next_permutation`

Gera a próxima permutação lexicograficamente maior de um intervalo. Essencial para "força bruta" em problemas com N pequeno. Retorna `false` quando não há mais permutações.

```
#include <iostream>
#include <string>
#include <algorithm> // Para std::sort e std::next_permutation

int main() {
    std::string s = "321"; // Come a com a maior permuta  o para demonstrar todas
    std::sort(s.begin(), s.end()); // Garante que come a da menor permuta  o:
    "123"
```

```
8
9       std::cout << "Permutacoes de " << s << ":" << std::endl;
10      do {
11          std::cout << s << std::endl;
12      } while (std::next_permutation(s.begin(), s.end()));
13      // Imprime: 123, 132, 213, 231, 312, 321
14      return 0;
15  }
```

9.1.6 std::unique

Remove elementos duplicados **consecutivos** de um intervalo. Para remover todas as duplicatas, ordene primeiro. Ele não redimensiona o contêiner, apenas move os elementos únicos para o início e retorna um iterador para o novo fim. Use o "erase-remove idiom" para de fato apagar os elementos.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm> // Para std::sort e std::unique
4
5  int main() {
6      std::vector<int> v = {1, 2, 2, 3, 3, 3, 1, 1};
7      std::cout << "Vetor original: ";
8      for(int x : v) std::cout << x << " ";
9      std::cout << std::endl;
10
11     std::sort(v.begin(), v.end()); // v = {1, 1, 1, 2, 2, 3, 3, 3}
12     std::cout << "Vetor ordenado: ";
13     for(int x : v) std::cout << x << " ";
14     std::cout << std::endl;
15
16     auto last = std::unique(v.begin(), v.end());
17     // v agora    {1, 2, 3, ... lixo ...}
18     std::cout << "Apos unique (antes de erase): ";
19     for(int x : v) std::cout << x << " ";
20     std::cout << std::endl;
21
22     v.erase(last, v.end());
23     // v agora    {1, 2, 3}
24     std::cout << "Apos erase: ";
25     for(int x : v) std::cout << x << " ";
26     std::cout << std::endl;
27     return 0;
28 }
```

9.2 Header <numeric>

9.2.1 std::accumulate

Soma os elementos de um intervalo a partir de um valor inicial. Complexidade O(N).

```
1  #include <iostream>
2  #include <vector>
3  #include <numeric> // Para std::accumulate
4
5  int main() {
6      std::vector<int> v = {1, 2, 3, 4, 5};
7      long long soma = std::accumulate(v.begin(), v.end(), 0LL);
8      // soma = 15. Use 0LL para somas que podem estourar um int.
9      std::cout << "Soma dos elementos: " << soma << std::endl;
10     return 0;
11 }
```

9.2.2 `std::gcd` e `std::lcm` (C++17)

Calcula o Máximo Divisor Comum (MDC) e o Mínimo Múltiplo Comum (MMC).

```cpp
#include <iostream>
#include <numeric> // Para std::gcd e std::lcm (C++17)

int main() {
    int mdc = std::gcd(12, 18); // mdc = 6
    int mmc = std::lcm(12, 18); // mmc = 36
    std::cout << "MDC(12, 18): " << mdc << std::endl;
    std::cout << "MMC(12, 18): " << mmc << std::endl;
    return 0;
}
```

9.3 Header `<string>`

9.3.1 `substr`, `stoi`, `to_string`

Funções essenciais para manipulação de strings.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s = "hello world";
    std::string sub = s.substr(6, 5); // sub = "world"
    std::cout << "Substring: " << sub << std::endl;

    std::string num_str = "123";
    int num = std::stoi(num_str); // num = 123
    std::cout << "String para int: " << num << std::endl;

    std::string converted = std::to_string(456); // converted = "456"
    std::cout << "Int para string: " << converted << std::endl;
    return 0;
}
```

9.4 Macros e Snippets Úteis

Em competições, é comum ver macros para acelerar a escrita.

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::sort

// Para iterar sobre qualquer cont iner
#define ALL(x) x.begin(), x.end()

// Loop for mais curto
#define FOR(i, a, b) for (int i = (a); i < (b); ++i)

int main() {
    std::vector<int> my_vector = {5, 2, 8, 1, 9};
    std::sort(ALL(my_vector)); // Exemplo de uso com sort
    std::cout << "Vetor ordenado com ALL: ";
    for(int x : my_vector) std::cout << x << " ";
    std::cout << std::endl;

    std::cout << "Loop com FOR: ";
    FOR(i, 0, 5) {
        std::cout << i << " "; // 0 1 2 3 4
    }
```

```
22      std::cout << std::endl;
23
24      // Sincroniza  o de I/O (colocar no in cio do main)
25      // std::ios_base::sync_with_stdio(false);
26      // std::cin.tie(NULL);
27      return 0;
28  }
```

10 Guia de Exercícios Práticos de C++ para Competição

A melhor maneira de solidificar o conhecimento das estruturas de dados e algoritmos é resolvendo problemas. Abaixo está uma lista de exercícios sugeridos para cada tópico que cobrimos. Tente resolver o problema primeiro antes de procurar a solução.

10.1 Vetores e Algoritmos Gerais

Estes problemas focam no uso de `std::vector` e funções do `<algorithm>`.

- **Problema: "Soma de Intervalos" (Prefix Sum)**
  - **Descrição:** Você recebe um array de N números e Q perguntas. Cada pergunta consiste em dois índices, `i` e `j`. Para cada pergunta, você deve responder a soma de todos os elementos do array entre `i` e `j` (inclusive).
  - **Técnica Principal:** Crie um array de "somas de prefixo" para poder responder cada pergunta em O(1).
- **Problema: "Encontrar Primeiro e Último Elemento em Array Ordenado"**
  - **Descrição:** Dado um array de inteiros ordenado, e um valor `X`, encontre o índice da primeira e da última ocorrência de `X`.
  - **Técnica Principal:** `std::lower_bound` e `std::upper_bound`.
- **Problema: "Gerando Permutações"**
  - **Descrição:** Dada uma string ou um conjunto de números, imprima todas as permutações únicas possíveis em ordem lexicográfica.
  - **Técnica Principal:** `std::sort` seguido de um loop com `std::next_permutation`.

10.2 Pilhas (`std::stack`)

- **Problema: "Parênteses Válidos"**
  - **Descrição:** Dada uma string contendo apenas `(`, `)`, `{` , `}`, `[` e `]`, determine se a string é balanceada.
  - **Técnica Principal:** `std::stack` para "casar" os parênteses de abertura e fechamento.
- **Problema: "Temperaturas Diárias"**
  - **Descrição:** Você recebe um array de temperaturas diárias. Retorne um array onde, para cada dia, está o número de dias que você precisa esperar até um dia mais quente. Se não houver, coloque 0.
  - **Técnica Principal:** Pilha Monótona (decrescente).

10.3 Filas (`std::queue` e `std::deque`)

- **Problema: "Caminho Mais Curto no Labirinto"**
  - **Descrição:** Em uma grade 2D representando um labirinto (com paredes e caminhos livres), encontre o comprimento do caminho mais curto de um ponto de partida 'S' até um ponto de chegada 'E'.
  - **Técnica Principal:** Busca em Largura (BFS) usando uma `std::queue` para armazenar as posições a visitar.

- **Problema: "Máximo da Janela Deslizante"**
  - **Descrição:** Dado um array e um número `k`, encontre o valor máximo em cada janela de tamanho `k` que desliza pelo array.
  - **Técnica Principal:** Fila Monótona (decrescente) implementada com `std::deque`.

10.4 Árvores da STL (`std::set`, `std::map`)

- **Problema: "Contagem de Frequência de Palavras"**
  - **Descrição:** Dado um texto, conte quantas vezes cada palavra aparece. Ignore maiúsculas/minúsculas.
  - **Técnica Principal:** `std::map<string, int>` para mapear cada palavra à sua contagem.
- **Problema: "Verificar Duplicatas Próximas"**
  - **Descrição:** Dado um array de inteiros e um inteiro `k`, determine se existem dois índices distintos `i` e `j` no array tais que `nums[i] == nums[j]` e a diferença absoluta entre `i` e `j` é no máximo `k`.
  - **Técnica Principal:** `std::set` usado como uma janela deslizante para manter os últimos `k` elementos vistos.

10.5 Heaps (`std::priority_queue`)

- **Problema: "Mesclar K Listas Ordenadas"**
  - **Descrição:** Dadas `K` listas de números, todas já ordenadas, mescle-as em uma única lista ordenada.
  - **Técnica Principal:** Use uma `min-heap` para manter o menor elemento de cada uma das `K` listas. A cada passo, extraia o menor da heap e insira o próximo da mesma lista.
- **Problema: "Pontos Mais Próximos da Origem"**
  - **Descrição:** Dado um array de pontos `(x, y)` no plano, encontre os `K` pontos mais próximos da origem (0,0).
  - **Técnica Principal:** Use uma `max-heap` de tamanho `K`. Para cada ponto, calcule sua distância ao quadrado. Se a heap tiver menos de `K` pontos, insira. Se a distância do ponto atual for menor que a da do topo da heap, remova o topo e insira o ponto atual.

10.6 Desafios (Combinando Estruturas)

- **Problema: "Mediana de um Fluxo de Dados"**
  - **Descrição:** Crie uma estrutura de dados que suporte duas operações: `addNum(num)` (adiciona um número de um fluxo de dados) e `findMedian()` (retorna a mediana de todos os elementos adicionados até agora).
  - **Técnica Principal:** Use duas heaps: uma `max-heap` para a metade inferior dos números e uma `min-heap` para a metade superior. Mantenha as heaps balanceadas em tamanho.
- **Problema: "Implemente um LRU Cache"**
  - **Descrição:** Projete uma estrutura de dados para um cache com política de remoção "Least Recently Used" (LRU). Deve suportar as operações `get(key)` e `put(key, value)` em tempo médio O(1).
  - **Técnica Principal:** Uma combinação de `std::map` (ou `unordered_map`) para acesso rápido e `std::list` (ou `deque`) para manter a ordem de uso recente.