

Notebook de Programação Competitiva

October 1, 2025

Contents

1	Estrutura De Dados	2
1.1	Arvore Binaria	2
1.2	Fila	3
1.3	Heap	4
1.4	Pilha	4
1.5	Vector	6
2	Outros	7

1 Estrutura De Dados

1.1 Arvore Binaria

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     // SET
7     set<int> s;
8
9     s.insert(10); // Adiciona o elemento
10    s.size(); // Retorna o tamanho
11    s.erase(10); // Remove o elemento 10
12    s.empty(); // Verifica se está vazio
13    s.clear(); // Remove todos os elementos
14
15    // Itera e imprime os elementos ordenados
16    for (int elemento : s) {
17        cout << " " << elemento;
18    }
19
20    // Verifica se um elemento existe
21    if (s.count(5)) { // Retorna 1 se existe, 0 se não
22        cout << "0 elemento 5 existe no set." << endl;
23    }
24
25    // find() retorna um iterador para o elemento, ou s.end() se não encontrar
26    auto it_set = s.find(20);
27    if (it_set != s.end()) {
28        cout << "Elemento 20 encontrado!" << endl;
29    }
30
31    // MAP
32    map<string, int> m;
33
34    m["banana"] = 10; // Adiciona/atualiza o par {"banana", 10}
35    m.insert(make_pair("laranja", 20)); // Outra forma de inserir
36    cout << "Valor associado a 'banana': " << m["banana"] << endl; // Acessa o valor
    pela chave
37
38    // Itera sobre o map (chaves estarão em ordem alfabética: banana, laranja, maca)
39    for (auto const& [chave, valor] : m) {
40        cout << "- Chave: " << chave << ", Valor: " << valor << endl;
41    }
42
43    m.erase("maca"); // Remove o par com a chave "maca"
44
45    // MULTISSET
46    multiset<int> ms;
47
48    ms.insert(10);
49    ms.insert(10); // Adiciona 10 novamente
50    ms.insert(10); // Adiciona 10 uma terceira vez
51    ms.size(); // Retorna tamanho do multimap
52
53    // Itera e imprime (elementos ordenados: 5, 10, 10, 10, 20)
54    for (int elemento : ms) {
55        cout << " " << elemento;
56    }
57
58    ms.erase(10); // Remove TODAS as ocorrências do valor 10
59
60    // Para remover apenas uma ocorrência, use um iterador
61    ms.insert(30);
62    auto it_ms = ms.find(30); // Encontra a primeira ocorrência de 30
63    if (it_ms != ms.end()) {
64        ms.erase(it_ms); // Remove apenas o elemento apontado pelo iterador
65    }
66
67    // Multimap
68    multimap<string, int> mm;
```

```

69
70 // No multimap, não se pode usar o operador [], pois ele não saberia
71 // qual valor acessar se houvesse chaves duplicadas. Use insert()
72 mm.insert(make_pair("aluno", 10));
73 mm.insert(make_pair("professor", 9));
74 mm.insert(make_pair("aluno", 8)); // Chave "aluno" duplicada
75 mm.count("aluno"); // Conta número de valores para chave
76 mm.erase("professor"); // Remove todos os pares com a chave "professor".
77 // it->first é a chave, it->second é o valor
78
79 return 0;
80 }

```

Listing 1: arvore binaria.cpp

1.2 Fila

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     queue<int> q;
7     deque<int> dq;
8
9     q.push(3); // Adiciona elemento na fila
10    q.pop(); // Remove elemento no inicio
11    q.empty(); // Verifica se a fila está vazia
12    q.size(); // Retorna tamanho da fila
13    q.front(); // Retorna valor no inicio da fila
14    q.back(); // Retorna valor no fim da fila
15    dq.push_front(3); // Adiciona elemento no inicio
16    dq.pop_front(); // Remove elemento no inicio
17    dq.push_back(3); // Adiciona elemento no fim
18    dq.pop_back(); // Remove elemento no fim
19
20 }

```

Listing 2: fila.cpp

Fila monótona

Seja F uma fila de elementos do tipo T . A fila F é dita **monótona** se, quando extraídos todos os elementos de F , eles formam uma sequência x_1, x_2, \dots, x_N , onde x_i é o elemento obtido na i -ésima extração, tais que a função $F: \mathbb{N} \rightarrow T$, com $f(i) = x_i$, é monótona. A fila F será **não-decrescente** se f for **não-decrescente**; caso contrário, F será **não-decrescente**.

- Em filas monótonas é necessário manter a invariante da monotonicidade a cada inserção.
- Seja F uma fila não-decrescente e x um elemento a ser inserido em F .
- Se F estiver vazia, basta inserir x em F : o invariante estará preservado.
- Se F não estiver vazia, o mesmo acontece se $x \leq y$, onde y é o último de F .
- Contudo, se $x > y$, é preciso remover y antes da inserção de x .
- Após a remoção de y , é preciso confrontar x com o novo elemento que ocupará a última posição até que x possa ser inserido na última posição de F .

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 template <typename T>
5 class MonoQueue {
6 public:
7     void push(const T& x) {
8         while (not st.empty() and st.back() > x)
9             st.pop_back();
10        st.emplace_back(x);

```

```

11     }
12
13     void pop() {
14         st.pop_front();
15     }
16
17     auto back() const {
18         return st.back();
19     }
20
21     auto front() const {
22         return st.front();
23     }
24
25     bool empty() const {
26         return st.empty();
27     }
28
29 private:
30     deque<T> st;
31 };
32
33 template <typename T>
34 ostream& operator<<(ostream& os, const MonoQueue<T>& ms) {
35     auto temp(ms);
36     while (not temp.empty()) {
37         cout << temp.front() << ' ';
38         temp.pop();
39     }
40     return os;
41 }
42
43 int main() {
44     vector<int> as{1, 4, 3, 4, 2, 1, 3};
45     MonoQueue<int> mq;
46
47     for (auto& a : as) {
48         mq.push(a);
49         cout << mq << '\n';
50     }
51
52     return 0;
53 }

```

Listing 3: fila monotona.cpp

1.3 Heap

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     priority_queue<int> pq; // Por padrão max-heap
7     priority_queue<int, vector<int>, greater<int>> pqMin; // Para criar uma min-heap
8
9     pq.push(30); // Adiciona elemento
10    pq.top(); // Acessa elemento de maior prioridade
11    pq.pop(); // Remove elemento de maior prioridade
12    pq.size(); // Retorna tamanho da fila
13    pq.empty(); // Retorna se a fila está vazia
14
15 }

```

Listing 4: heap.cpp

1.4 Pilha

```

1 #include <bits/stdc++.h>
2 using namespace std;

```

```

3
4 int main()
5 {
6     stack<int> st;
7
8     st.push(3); // Adiciona elemento na pilha
9     st.pop(); // Remove elemento no topo
10    st.empty(); // Verifica se a pilha está vazia
11    st.size(); // Retorna tamanho da pilha
12    st.top(); // Retorna valor no topo da pilha
13
14    // Exemplo de delimitador com pilha
15    string n, resultado = "";
16    stack<char> st;
17    cin >> n;
18    for(size_t i = 0; i < n.length(); i++)
19    {
20        if(n[i] == '(')
21        {
22            st.push(n[i]);
23            resultado += '(';
24        }
25        else if(n[i] == ')' && st.size() > 0)
26        {
27            st.pop();
28            resultado += ')';
29        }
30        else if(n[i] == ')' && st.size() == 0)
31        {
32            resultado = '(' + resultado + ')';
33        }
34    }
35    while(st.size() > 0)
36    {
37        resultado += ')';
38        st.pop();
39    }
40    cout << resultado << '\n';
41 }

```

Listing 5: pilha.cpp

Pilha monótona

Seja P uma pilha de elementos do tipo T . A pilha P é dita **monótona** se, quando extraídos todos os elementos de P , eles formam uma sequência x_1, x_2, \dots, x_N , onde x_i é o elemento obtido na i -ésima extração, tais que a função $F: \mathbb{N} \rightarrow T$, com $f(i) = x_i$, é monótona. A pilha P será **não-decrescente** se f for **não-crescente**; caso contrário, P será **não-decrescente**.

- É possível determinar o maior elemento à esquerda para todos os elementos de uma sequência a_1, a_2, \dots, a_N em $O(N^2)$ por meio de uma busca completa.
- Para cada índice i , é preciso avaliar todos os elementos a_j , com $j = 1, 2, \dots, i - 1$.
- Contudo, é possível determinar estes valores em $O(N)$ com uma modificação no método de inserção de uma pilha não-crescente.
- A inserção em uma pilha não-crescente ocorre em duas etapas: manutenção do invariante e inserção do novo elemento.
- Finalizada a manutenção do invariante, os elementos que restam na pilha são todos maiores do que a_i e o elemento do topo será o maior elemento à esquerda de a_i .
- Em algumas implementações são mantidos os índices e não os valores da sequência propriamente ditos (ou pares com ambas informações).

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 template <typename T>

```

```

5 class MonoStack {
6 public:
7     void push(const T& x) {
8         while (not st.empty() and st.top() > x)
9             st.pop();
10        st.emplace(x);
11    }
12
13    void pop() {
14        st.pop();
15    }
16
17    auto top() const {
18        return st.top();
19    }
20
21    bool empty() const {
22        return st.empty();
23    }
24
25 private:
26     stack<T> st;
27 };
28
29 template <typename T>
30 ostream& operator<<(ostream& os, const MonoStack<T>& ms) {
31     auto temp(ms);
32     while (not temp.empty()) {
33         cout << temp.top() << ' ';
34         temp.pop();
35     }
36     return os;
37 }
38
39 int main() {
40     vector<int> as{1, 4, 3, 4, 2, 1, 3};
41     MonoStack<int> ms;
42
43     for (auto& a : as) {
44         ms.push(a);
45         cout << ms << '\n';
46     }
47
48     return 0;
49 }

```

Listing 6: pilha monotona.cpp

1.5 Vector

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     int n = 5;
7     vector<int> vet(n, 0); // Cria vetor de tamanho 5, com todos valores = 0
8
9     vet.push_back(5); // Adiciona valor no fim do vetor
10    vet.pop_back(); // Remove ultimo valor do vetor
11    vet.size(); // Retorna tamanho do vetor
12    vet.clear(); // Remove todos os elementos mas não libera a memória alocada
13    sort(vet.begin(), vet.end()) // Ordenação, greater<int> para colocar em ordem
    decrescente
14
15    vector<vector<int>> matriz(n, vector<int>(n, -2)); // Declara Matriz de tamanho NxN
16    for (int i = 0; i < n; i++) { // percorre as linhas
17        for (int j = 0; j < n; j++) { // percorre as colunas
18            cout << matriz[i][j]; // acessa elemento [i][j]
19        }
20    }

```

2 Outros

Soma de Prefixos

Soma de prefixos é um método que permite calcular a soma de qualquer subvetor contínuo em tempo constante, $O(1)$.

Dado um vetor A de tamanho N , seu vetor de soma de prefixos, P , é definido tal que $P[i]$ contém a soma de todos os elementos desde $A[0]$ até $A[i]$.

- **Cálculo Eficiente:** O vetor P pode ser calculado em tempo linear, $O(N)$, utilizando a seguinte recorrência:

$$P[i] = P[i - 1] + A[i], \quad \text{com o caso base } P[0] = A[0]$$

- **Aplicação Principal:** A soma de um subvetor de A do índice i ao j (inclusive) é calculada em tempo $O(1)$ da seguinte forma:

$$\text{soma}(A[i \dots j]) = P[j] - P[i - 1]$$

Para o caso especial em que $i = 0$, a soma é simplesmente $P[j]$.

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     int n, num;
6     cin >> n;
7     vector<long long> vet(n * 2), soma(n * 2 + 1);
8     for (int i = 0; i < n; i++) {
9         cin >> num;
10        vet[i] = num;
11        vet[i + n] = num;
12    }
13    soma[0] = 0;
14    for (int i = 0; i < n * 2; i++) {
15        soma[i + 1] = soma[i] + vet[i];
16    }
17 }
```

Listing 8: soma de prefixos.cpp