Notebook de Programação Competitiva

October 1, 2025

Contents

1 Estrutura De Dados

1.1 Fila

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    queue<int> q;
    deque<int> dq;

    q.push(3); // Adiciona elemento na fila
    q.pop(); // Remove elemento no inicio
    q.empty(); // Verifica se a fila está vazia
    q.size(); // Retorna tamanho da fila
    q.front(); // Retorna valor no inicio da fila
    q.back(); // Retorna valor no fim da fila
    dq.push_front(3); // Adiciona elemento no inicio
    dq.pop_front(); // Remove elemento no inicio
    dq.push_back(3); // Adiciona elemento no fim
    dq.pop_back(); // Remove elemento no fim

}
```

Listing 1: 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 $x_1, x_2, \ldots, x_N$, onde x_i é o elemento obtido na i-ésima extração, tais que a função $F : \mathbb{N} \to T$, 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.

```cpp
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class MonoQueue {
public:
    void push(const T& x) {
        while (not st.empty() and st.back() > x)
            st.pop_back();
        st.emplace_back(x);
    }

    void pop() {
        st.pop_front();
    }

    auto back() const {
        return st.back();
    }

    auto front() const {
        return st.front();
    }

```

```
    bool empty() const {
        return st.empty();
    }

private:
    deque<T> st;
};

template <typename T>
ostream& operator<<(ostream& os, const MonoQueue<T>& ms) {
    auto temp(ms);
    while (not temp.empty()) {
        cout << temp.front() << ' ';
        temp.pop();
    }
    return os;
}

int main() {
    vector<int> as{1, 4, 3, 4, 2, 1, 3};
    MonoQueue<int> mq;

    for (auto& a : as) {
        mq.push(a);
        cout << mq << '\n';
    }

    return 0;
}
```

Listing 2: fila monotona.cpp

1.2 Heap

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

int main()
{
    priority_queue<int> pq; // Por padrão max-heap
    priority_queue<int, vector<int>, greater<int>> pqMin; // Para criar uma min-heap

    pq.push(30); // Adiciona elemento
    pq.top(); // Acessa elemento de maior prioridade
    pq.pop(); // Remove elemento de maior prioridade
    pq.size(); // Retorna tamanho da fila
    pq.empty(); // Retorna se a fila está vazia

}
```

Listing 3: heap.cpp

1.3 Pilha

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

int main()
{
    stack<int> st;

    st.push(3); // Adiciona elemento na pilha
    st.pop(); // Remove elemento no topo
    st.empty(); // Verifica se a pilha está vazia
    st.size(); // Retorna tamanho da pilha
    st.top(); // Retorna valor no topo da pilha

    // Exemplo de delimitador com pilha
    string n, resultado = "";
    stack<char> st;
```

```
    cin >> n;
    for(size_t i = 0; i < n.length(); i++)
    {
        if(n[i] == '(')
        {
            st.push(n[i]);
            resultado += '(';
        }
        else if(n[i] == ')' && st.size() > 0)
        {
            st.pop();
            resultado += ')';
        }
        else if(n[i] == ')' && st.size() == 0)
        {
            resultado = '(' + resultado + ')';
        }
    }
    while(st.size() > 0)
    {
        resultado += ')';
        st.pop();
    }
    cout << resultado << '\n';
}
```

Listing 4: 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 $x_1, x_2, \ldots, x_N$, onde x_i é o elemento obtido na i-ésima extração, tais que a função $F : \mathbb{N} \to T$, 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 $a_1, a_2, \ldots, a_N$ em $O(N^2)$ por meio de uma busca completa.
- Para cada índice i, é preciso avaliar todos os elementos a_j, com $j = 1, 2, \ldots, i-1$.
- 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).

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

template <typename T>
class MonoStack {
public:
    void push(const T& x) {
        while (not st.empty() and st.top() > x)
            st.pop();
        st.emplace(x);
    }

    void pop() {
        st.pop();
    }

    auto top() const {
        return st.top();
```

```
    }

    bool empty() const {
        return st.empty();
    }

private:
    stack<T> st;
};

template <typename T>
ostream& operator<<(ostream& os, const MonoStack<T>& ms) {
    auto temp(ms);
    while (not temp.empty()) {
        cout << temp.top() << ' ';
        temp.pop();
    }
    return os;
}

int main() {
    vector<int> as{1, 4, 3, 4, 2, 1, 3};
    MonoStack<int> ms;

    for (auto& a : as) {
        ms.push(a);
        cout << ms << '\n';
    }

    return 0;
}
```

Listing 5: pilha monotona.cpp

1.4 Vector

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

int main()
{
    int n = 5;
    vector<int> vet(n, 0); // Cria vetor de tamanho 5, com todos valores = 0

    vet.push_back(5); // Adiciona valor no fim do vetor
    vet.pop_back(); // Remove ultimo valor do vetor
    vet.size(); // Retorna tamanho do vetor
    vet.clear(); // Remove todos os elementos mas não libera a memória alocada
    sort(vet.begin(), vet.end()) // Ordenação, greater<int> para colocar em ordem
    decrescente

    vector<vector<int>> matriz(n, vector<int>(n, -2)); // Declara Matriz de tamanho NxN
    for (int i = 0; i < n; i++) {           // percorre as linhas
        for (int j = 0; j < n; j++) {       // percorre as colunas
            cout << matriz[i][j];    // acessa elemento [i][j]
        }
    }
}
```

Listing 6: vector.cpp

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 $P[j]$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  int n, num;
  cin >> n;
  vector<long long> vet(n * 2), soma(n * 2 + 1);
  for (int i = 0; i < n; i++) {
    cin >> num;
    vet[i] = num;
    vet[i + n] = num;
  }
  soma[0] = 0;
  for (int i = 0; i < n * 2; i++) {
    soma[i + 1] = soma[i] + vet[i];
  }
}
```

Listing 7: soma de prefixos.cpp