Aula 2 · Mais estruturas da STL: Listas e Árvores

Desafios de Programação

Fernando Kiotheka Victor Alflen

UFPR

15/06/2022

O par ordenado (`pair`) e a tupla (`tuple`)

O `pair<T, U>` é um par ordenado é composto de dois elementos do tipo `T` e `U` respectivamente. Ele é extremamente útil porque você não precisa criar uma `struct` quando precisar de um par de elementos, e além disso, ele é (como o nome diz) **ordenável**. Em algumas situações, o `tuple<T u, U y, ...>` também é adequado, mas ele tem acessos mais chatinhos.

- `pair<T, U> p (T x, U y)`: Cria um par com `x` e `y`.
- `p.first`: Acessa o primeiro item do par.
- `p.second`: Acessa o segundo item do par.
- `make_pair(T x, U y)`: Cria um par com `x` e `y`.
- `tuple<T, U, ...> t (T x, U y, ...)`: Cria uma tupla com `x`, `y`, ...
- `make_tuple(T x, U y, ...)`: Cria uma tupla com `x`, `y`, ...
- `get<i>(t_or_p)`: Acessa a posição `i` da tupla ou par `t_or_p`.

Exemplo de pair e tuple

Código pair.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define xx first
#define yy second
int main() {
    pair<int, int> coords (1, 2);
    cout << coords.xx << " " << coords.yy << ", ";
    coords.xx += 3;
    cout << coords.xx << " " << coords.yy << "\n";
    if (make_tuple(1, 2, 3) < make_tuple(1, 2, 2))
        cout << "1,2,3 < 1,2,2\n";
    if (make_tuple(1, 2, -3) < make_tuple(1, 2, -2))
        cout << "1,2,-3 < 1,2,-2\n";
}
```

Entrada	Saída
	1 2, 4 2 1,2,-3 < 1,2,-2

A fila de duas extremidades (deque)

O deque<T> é diferente do vector porque não garante que tudo seja guardado em uma única região contígua de memória. Porém, ainda assim, o deque garante todas as operações que o vector faz, mais algumas:

- `d.push_front(T x)` [$\mathcal{O}(1)$]: Adiciona um valor ao início.
- `d.pop_front()` [$\mathcal{O}(1)$]: Remove o elemento do início.

O custo é apenas o fato do deque ser algumas vezes mais complicado que o vector, implicando em uma constante maior. Se você sabe que não vai usar essa gama de operações, não vale a pena utilizar o deque diretamente. Se você quiser uma fila ou pilha, utilize stack<T> e queue<T>.

A pilha (stack)

A stack<T> é uma pilha, o último que entra é o primeiro que sai (LIFO). Ela é implementada como uma abstração sobre o deque<T>. Se você quiser o vector<T> ou list<T> como abstração, pode também, declarando: stack<T, vector<T>>.

Código stack.cpp

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

int main() {
    stack<int> s;
    s.push(2); s.push(3);
    cout << s.top() << " sz=" << s.size() << ", "; s.pop();
    cout << s.top() << " sz=" << s.size() << ", "; s.pop();
    cout << "empty=" << s.empty() << "\n";
}
```

Entrada	Saída
	3 sz=2, 2 sz=1, empty=1

A fila (queue)

A queue<T> é uma fila, o primeiro que entra é o primeiro que sai (FIFO). Por padrão é implementada como uma abstração sobre o deque<T>, mas é possível usar a `list<T>`.

Código `queue.cpp`

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

int main() {
    queue<int> q;
    q.push(5); q.push(6);
    cout << q.front() << " " << q.back() << " ";
    cout << "sz=" << q.size() << ", "; q.pop();
    cout << q.front() << " " << q.back() << " ";
    cout << "sz=" << q.size() << ", "; q.pop();
    cout << "empty=" << q.empty() << "\n";
}
```

Entrada	Saída
	`5 6 sz=2, 6 6 sz=1, empty=1`

O mínimo de todos os subvetores de tamanho K

Suponha que ganhamos um vetor A de tamanho N e um dado $K \leq N$. Temos que achar o mínimo de cada subvetor de tamanho K neste vetor.
Por exemplo para $K = 3$:

$$v = \left[{}_0 1 \, {}_1 2 \, {}_2 3 \, {}_3 3 \, {}_4 5 \, {}_5 8 \, {}_6 \right]$$

$$\begin{aligned}
\min\{v[0], v[1], v[2]\} &= \{1, 2, 3\} = 1 \\
\min\{v[1], v[2], v[3]\} &= \{2, 3, 3\} = 2 \\
\min\{v[2], v[3], v[4]\} &= \{3, 3, 5\} = 3 \\
\min\{v[3], v[4], v[5]\} &= \{3, 5, 8\} = 3
\end{aligned}$$

Como fazer isso **rápido**?

Nossa primeira estrutura de dados: A fila mínima

Queremos fazer uma modificação na fila de forma que seja possível achar o menor elemento na fila em tempo constante ($\mathcal{O}(1)$).
Podemos fazer de inúmeras maneiras, mas uma que serve para nós é uma opção que:

- É uma fila que não vai guardar todos os elementos, apenas os necessários para achar o mínimo (não vamos usar a fila efetivamente, então não tem problema)
- Precisamos sempre saber o valor do elemento que queremos remover da fila (também é fácil)

Então vamos a solução!

A ideia da estrutura de fila mínima

A ideia é manter uma fila em ordem não-descrescente (o menor valor será o próximo a sair da fila).

- Para achar o mínimo, é só consultar o próximo a sair da fila.
- Para adicionar um elemento, é só verificar os últimos que entraram na fila, e removeremos todos os elementos maiores que o que vamos adicionar.
- Para remover um elemento, verificamos se o próximo elemento a sair da fila é o que queremos remover, e se for, o removemos.

Resolvendo o mínimo de subvetores

Código `minqueue.cpp`

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k; cin >> n >> k;
    vector<int> v (n); for (auto& x : v) { cin >> x; }
    deque<int> q;
    for (int i = 0; i < n; i++) {
        if (i >= k && !q.empty() && q.front() == v[i-k]) {
            q.pop_front();
        }
        while (!q.empty() && q.back() > v[i]) {
            q.pop_back();
        }
        q.push_back(v[i]);
        if (i+1 >= k) { cout << q.front() << " "; }
    }
}
```

Exemplos do mínimo de subvetores

Entrada	Saída
6 3 1 2 3 3 5 8	1 2 3 3
10 4 1 4 9 8 7 3 5 6 9 7	1 4 3 3 3 3 5

Algoritmos, Algoritmos!

Como que a gente ordena um vetor? Com iteradores e funções!

- `reverse(iterator begin, end)` [$\mathcal{O}(n \lg n)$]: Inverte os elementos do intervalo $[begin, end)$.
- `is_sorted(iterator begin, end)` [$\mathcal{O}(n)$]: O intervalo $[begin, end)$ é ordenado?
- `sort(iterator begin, end, bool(T, T) cmp = less<T>)` [$\mathcal{O}(n \lg n)$]: Ordena o intervalo $[begin, end)$.
- `stable_sort(iterator begin, end, bool(T, T) cmp = less<T>)` [$\mathcal{O}(n \lg n)$]: Ordena o intervalo $[begin, end)$ de forma estável (mantém a ordem de elementos iguais).
- `unique(iterator begin, end, bool(T, T) eq = ==<T>)` [$\mathcal{O}(n)$]: Remove duplicados do intervalo ordenado $[begin, end)$ e retorna um novo end.
- `remove(iterator begin, end, T val)` [$\mathcal{O}(n)$]: Remove iguais a `val` no intervalo $[begin, end)$ e retorna um novo end.

Exemplo de ordenação

Código `sort.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#define p(XS) for (int x : XS) cout << x << ' '; \
    cout << "s=" << is_sorted(XS.begin(), XS.end()) << "\n";
int main() {
    vector<int> xs { 4, 6, 5, 1, 1, 2, 3 };
    sort(xs.begin(), xs.end()); p(xs);
    xs.erase(unique(xs.begin(), xs.end()), xs.end()); p(xs);
    sort(xs.rbegin(), xs.rend()); p(xs);
    sort(xs.begin(), xs.end(), [](int a, int b) {
        return a*3 < b*2; }); p(xs);
}
```

Entrada	Saída
	1 1 2 3 4 5 6 s=1 1 2 3 4 5 6 s=1 6 5 4 3 2 1 s=0 1 3 2 6 5 4 s=0

A lista (`list`)

A `list<T>` é aquela lista duplamente encadeada de Algoritmos II, ela voltou. Como você já sabe, elas são boas pra se inserir, deletar, mas não para se acessar aleatoriamente. Ocasionalmente são úteis.

- `list<T> l (int n = 0, T val = {})` $[\mathcal{O}(n)]$: Cria uma `list` do tipo `T` com `n` elementos inicializados em `val`.
- `l.resize(int n, T val = {})` $[\mathcal{O}(n)]$: Muda o tamanho do vetor preenchendo com `val` se `n` for maior que o atual.
- `l.size()` $[\mathcal{O}(1)]$: Retorna o tamanho (cuidado: `unsigned`!).
- `l.front()`, `l.back()` $[\mathcal{O}(1)]$: Acessa o primeiro, último.
- `l.push_back(T x)` $[\mathcal{O}(1)]$: Adiciona um valor ao final.
- `l.pop_back()` $[\mathcal{O}(1)]$: Remove o elemento do final.
- `l.push_front(T x)` $[\mathcal{O}(1)]$: Adiciona um valor ao início.
- `l.pop_front()` $[\mathcal{O}(1)]$: Remove o elemento do início.
- `l.clear()` $[\mathcal{O}(1)]$: Remove todos os elementos.
- `l.empty()` $[\mathcal{O}(1)]$: Está vazio?

Iteradores da `list`

Os iteradores da `list` não podem ser iguais aos do `vector` por motivos óbvios, não dá pra simplesmente somar um valor e descobrir a localização de um elemento. Por esse motivo eles só suportam operações `++` e `--`. Se você quiser avançar em $\mathcal{O}(n)$ em qualquer iterador é só usar `advance(iterator it, int delta)`. Usando os iteradores obtidos por `l.begin()` e `l.end()` é possível:

- `l.insert(iterator it, int val)` [$\mathcal{O}(1)$]: Insere após o iterador `it` o valor `val`.
- `l.erase(iterator it)` [$\mathcal{O}(1)$]: Remove o elemento apontado pelo iterador `it`.
- `l.erase(iterator first, iterator last)` [$\mathcal{O}(d)$]: Remove os elementos dados pelo intervalo $[first, last)$. d é a quantidade removida.
- `l.splice(iterator first, list m)` [$\mathcal{O}(1)$]: Transfere os elementos da lista `m` (remove tudo de `m`) após o iterador `it`.

Aquela heap que você falou? (priority_queue)

A priority_queue<T> encapsula um vector rodando o algoritmo de max heap de Algoritmos III. O C++ também expõe make_heap, push_heap e pop_heap se você quiser inventar fazer no seu próprio vetor. Assim, push e pop são $\mathcal{O}(\lg n)$.

Código priority-queue.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    priority_queue<int, vector<int>, less<int>> pq;
    pq.push(25); pq.push(25); pq.push(30); pq.push(20);
    while (!pq.empty()) {
        cout << pq.top() << " (" << pq.size() << ") ";
        pq.pop();
    } cout << "\n";
}
```

Entrada	Saída
	30 (4) 25 (3) 25 (2) 20 (1)

O conjunto (`set`)

Já o `set<T>` é para conjuntos. E é no sentido matemático, não há repetições. Ele mantém os elementos ordenados no seu interior através de uma estrutura em árvore (geralmente uma rubro-negra). Esta estrutura possui iteradores não somáveis.

- `set<T> s (bool(T, T) cmp = less<T>)` [$\mathcal{O}(1)$]: Cria um `set` do tipo `T` usando a função de comparação `cmp`.
- `s.size()` [$\mathcal{O}(1)$]: Retorna o tamanho (cuidado: `unsigned`!).
- `s.clear()` [$\mathcal{O}(1)$]: Remove todos os elementos.
- `s.empty()` [$\mathcal{O}(1)$]: É vazio?
- `s.find(T v)` [$\mathcal{O}(\lg n)$] Acha o `v` e retorna seu iterador (ou `s.end()` se não encontrar).
- `s.insert(T v)` [$\mathcal{O}(\lg n)$] Insere `v`. Retorna um par de iterador e um booleano indicando se é um elemento novo.
- `s.erase(T v)` [$\mathcal{O}(\lg n)$] Remove `v` (se existir).
- `s.count(T v)` [$\mathcal{O}(\lg n)$] Retorna quantidade de `v` (0 ou 1).

Exemplo de set

Código `set.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    set<int> s;
    s.insert(2); s.insert(2); s.insert(5); s.insert(7);
    for (auto x : s) { cout << x << " "; }
    set<int>::iterator it = s.find(5); it--;
    cout << "it=" << *it << "\n";
    s.erase(5); s.erase(8); s.erase(it);
    cout << "2? " << s.count(2) << ", 5? " << s.count(5);
    cout << ", 7 e novo? " << s.insert(7).second << "\n";
    cout << "sz=" << s.size() << "\n";
}
```

Entrada	Saída
	2 5 7 it=2 2? 0, 5? 0, 7 e novo? 0 sz=1

Que tal um set com valor? (map)

Na mesma ideia do conjunto `set<T>`, temos o `map<K, V>` que mantém um conjunto de chaves `K` e associa a cada uma delas um `V`. Todas as operações do `set` se aplicam, mais essas:

- `m[K key] = (V val)` [$\mathcal{O}(\lg n)$]: Cria ou sobrescreve a chave `key` com o valor `val`.
- `m[K key]` [$\mathcal{O}(\lg n)$]: Pega o valor da chave `key` (cuidado: **cria um valor vazio** se não existir! Verifique com `m.count(key)`).

Código `map.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    map<string, int> num; num["fever"] = 2;
    cout << "0 = " << num["marcio"] << ", ";
    for (auto [k, v] : num) cout << k << ": " << v << " ";
}
```

Entrada	Saída
	`0 = 0, fever: 2 marcio: 0`

Conjuntos... com elementos repetidos? (`multiset`)

O `multiset<T>` mantém um conjunto muito similar o `set`, com repetições! É preciso ter cuidado com a complexidade porém, as vezes vale a pena usar um `map<T, int>`.

- `multiset<T> m (bool(T, T) cmp = less<T>)` [$\mathcal{O}(1)$]: Cria um `multiset` do tipo `T` usando a função de comparação `cmp`.
- `m.size()` [$\mathcal{O}(1)$]: Retorna o tamanho (cuidado: `unsigned`!).
- `m.clear()` [$\mathcal{O}(1)$]: Remove todos os elementos.
- `m.empty()` [$\mathcal{O}(1)$]: É vazio?
- `m.find(T v)` [$\mathcal{O}(\lg n)$] Acha um `v` (não necessariamente o 1ž) e retorna seu iterador (ou `m.end()` se não encontrar).
- `m.insert(T v)` [$\mathcal{O}(\lg n)$] Insere `v` e retorna o iterador.
- `m.erase(T v)` [$\mathcal{O}(n)$] Remove todos os `v`.
- `m.erase(iterator it)` [$\mathcal{O}(\lg n)$] Remove o apontado por `it`.
- `m.count(T v)` [$\mathcal{O}(n)$] Retorna quantidade de `v` (cuidado!).

Quero meu $\mathcal{O}(1)$! (`unordered_*`)

Existem versões do `set`, do `multiset` e do `map` que usam *hashing* ao invés da manutenção de uma árvore balanceada. Elas são `unordered_set`, `unordered_multiset` e `unodered_map` respectivamente. Elas prometem $\mathcal{O}(1)$ amortizado nas operações que são normalmente $\mathcal{O}(\lg n)$. Isso é bom, mas é perigoso:

- A ordenação você vai perder logo de cara, como diz o nome.
- O pior caso é $\mathcal{O}(n)$, que é o caso onde o *hashing* dá resultados desastrosos (tudo dá conflito).
- A função de *hashing* padrão do C++ é péssima, e o setters maldosos **vão abusar disso para criar testes extremos**.
- Então em geral, pra usar os `unordered_*`, você vai precisar definir alguns parâmetros, customizar a função de *hashing*, etc.

Mais detalhes em
`https://codeforces.com/blog/entry/62393`.

E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Mais informações sobre a fila e a pilha mínima: `https://cp-algorithms.com/data_structures/stack_queue_modification.html`
- Não é necessário você entender absolutamente tudo agora, com o tempo você ficará mais habituado com a STL.