Técnicas para Programação Competitiva

Estruturas de Dados na STL do C++

Samuel da Silva Feitosa

Aula 3
2022/2

Estruturas de Dados

Introdução

- Em programação competitiva é crucial conhecer quais estruturas de dados estão disponíveis na biblioteca padrão da linguagem.
  - Isso acelera muito o desenvolvimento dos algoritmos.
- Veremos as principais estruturas de dados presentes na STL do C++.
  - vector, sets, maps, etc.
  - Referência: www.cplusplus.com

Arrays dinâmicos

- Em C++, os arrays são estruturas de dados de tamanho fixo.
  - Não é possível mudar o tamanho depois da criação.

```
int array[n];
```

- Um *array dinâmico* é um array que pode ter seu tamanho modificado durante a execução de um programa.
  - O C++ possui diversos arrays dinâmicos, sendo ***vector*** o mais conhecido.

Vectors (1)

- Permite adicionar e remover elementos ao **final da estrutura** eficientemente. Elementos podem ser acessados normalmente.

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

- Também podemos criar um *vector* indicando explicitamente os seus elementos:

```
vector<int> v = {2,4,2,5,1};
```

Vectors (2)

- Podemos ainda criar um *vector* indicando o seu tamanho e valores iniciais:

```
vector<int> a(8); // tamanho 8, valor inicial 0
vector<int> b(8,2); // tamanho 8, valor inicial 2
```

Vectors (3)

- Iterando *vectors*:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

```
for (auto x : v) {
    cout << x << "\n";
}
```

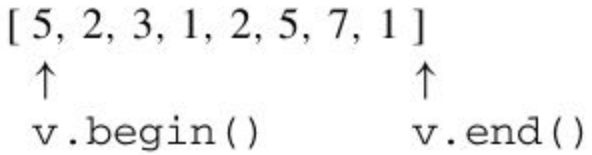
- O método *back()* retorna o último elemento de um *vector* e o método *pop_back()* remove o último elemento de um *vector*

```
vector<int> v = {2,4,2,5,1};
cout << v.back() << "\n"; // 1
v.pop_back();
cout << v.back() << "\n"; // 5
```

- *push_back* e *pop_back* trabalham em tempo O(1) na média.

Iterators e Ranges (1)

- Um *iterator* é uma variável que aponta para um elemento
  - *begin()* aponta para o primeiro e *end()* aponta para a posição **após** o último



- *range* é uma sequência de elementos consecutivos
  - Os iterators *begin()* e *end()* definem um range que contém todos os elementos da estrutura

```
sort(v.begin(),v.end());
reverse(v.begin(),v.end());
random_shuffle(v.begin(),v.end());
```

Iterators e Ranges (2)

- O elemento para o qual um *iterator* aponta pode ser acessado usando * :

```
cout << *v.begin() << "\n";
```

Funções úteis

- Dado um *range* onde os **valores** estão **ordenados**,
  - *lower_bound* retorna um *iterator* para o primeiro elemento com valor **pelo menos x** no *range*
  - *upper_bound* retorna um *iterator* para o primeiro elemento com valor **maior que x** no *range*

```
vector<int> v = {2,3,3,5,7,8,8,8};
auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);
cout << *a << " " << *b << "\n"; // 5 7
```

- *unique* remove os elementos duplicados em um dado *range*:

```
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());
```

Deque

- *deque* é um array dinâmico que permite a manipulação eficiente de ambos os lados da sua estrutura
  - Métodos: *push_back*, *pop_back*, *push_front*, *pop_front*

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

- Essas operações tem complexidade O(1) em média
  - Entretanto, *deques* possuem constantes de complexidade maiores do que vetores

Stack (pilha) e queue (fila)

- C++ também possui outras duas estruturas importantes, cuja implementação é baseada em uma *deque*
  - **stack** (pilha), com as operações *push*, *pop* e *top*
  - **queue** (fila), com as operações *push*, *pop*, *front* e *back*

```
stack<int> s;
s.push(2); // [2]
s.push(5); // [2,5]
cout << s.top() << "\n"; // 5
s.pop(); // [2]
cout << s.top() << "\n"; // 2
```

```
queue<int> q;
q.push(2); // [2]
q.push(5); // [2,5]
cout << q.front() << "\n"; // 2
q.pop(); // [5]
cout << q.back() << "\n"; // 5
```

Estruturas de Conjunto

- Uma estrutura de conjunto é uma estrutura de dados que armazena uma coleção de elementos
  - As suas operações básicas são inserção, busca e remoção
- A biblioteca de C++ contém dois tipos para conjuntos:
  - *set*, que é baseado em uma árvore binária balanceada e suas operações têm complexidade O(log *n*)
  - *unordered_set*, que é baseada em uma tabela hash e opera em média em tempo O(1)

Set (1)

- Principais operações:
  - *insert*, *erase*, *count*, *find*, …

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

Set (2)

- Uma propriedade importante de uma estrutura de conjunto é que todos os seus elementos são distintos

```
set<int> s;
s.insert(3);
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

- Não é possível acessar elementos de um *set* usando *[ ]*
- Podemos acessar todos os elementos de um *set* como a seguir:

```
cout << s.size() << "\n";
for (auto x : s) {
    cout << x << "\n";
}
```

Set (3)

- Podemos buscar por um elemento um *set* como a seguir:

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

- A principal diferença entre *set* e *unordered_set* é que o primeiro mantém uma ordem dos elementos
- Em um *set*, podemos acessar o maior e o menor elementos:

```
auto first = s.begin();
auto last = s.end(); last--;
cout << *first << " " << *last << "\n";
```

Estruturas de Multiconjunto

- Um multiconjunto é um conjunto que pode ter várias cópias de um mesmo valor.
  - C++ tem as estruturas *multiset* e *unordered_multiset*, com características similares a *set* e *unordered_set*

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

```
s.erase(5);
cout << s.count(5) << "\n"; // 0

s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

Map

- Um *map* é uma estrutura que consiste de pares chave-valor
  - Podem ser vistos como arrays generalizados
  - Enquanto as chaves de um array de tamanho *n* são sempre 0, 1, …, *n* - 1, as chaves em um *map* podem ser de qualquer tipo de dado
- C++ oferece as estruturas *map* e *unordered_map*:
  - *map*, que é baseado em árvores binárias balanceadas, onde o acesso aos seus elementos toma tempo O(log *n*)
  - *unordered_map*, que é baseado em tabelas hash e o acesso aos seus elementos leva em média tempo O(1)

Maps: Exemplos

- É possível manipular um *map* como um vetor, usar a função *count* para verificar a existência de elementos, e iterar sobre os elementos:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0

if (m.count("aybabtu")) {
    // key exists
}
```

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

Experimentos (1)

- Set x Sorting

Table 5.1 Results of an experiment where the number of unique elements in a vector was calculated. The first two algorithms insert the elements to a set structure, while the last algorithm sorts the vector and inspects consecutive elements

Input size n	`set` (s)	`unordered_set` (s)	Sorting (s)
10^6	0.65	0.34	0.11
$2 \cdot 10^6$	1.50	0.76	0.18
$4 \cdot 10^6$	3.38	1.63	0.33
$8 \cdot 10^6$	7.57	3.45	0.68
$16 \cdot 10^6$	17.35	7.18	1.38

Experimentos (2)

- Map x Array

Table 5.2 Results of an experiment where the most frequent value in a vector was determined. The two first algorithms use map structures, and the last algorithm uses an ordinary array

Input size n	`map` (s)	`unordered_map` (s)	Array (s)
10^6	0.55	0.23	0.01
$2 \cdot 10^6$	1.14	0.39	0.02
$4 \cdot 10^6$	2.34	0.73	0.03
$8 \cdot 10^6$	4.68	1.46	0.06
$16 \cdot 10^6$	9.57	2.83	0.11

Considerações Finais

- Nesta aula estudamos as principais estruturas de dados disponíveis na STL do C++.
  - Essas estruturas são muito úteis no contexto da programação competitiva, uma vez que não é preciso reimplementar estruturas básicas.
- Também verificamos questões de eficiência entre as estruturas disponíveis, para que seja possível decidir qual estrutura pode ser mais adequada para um determinado problema.