

Ordenação e Busca

Introdução à Programação Competitiva



**INSTITUTO
FEDERAL**
Brasília

Prof. Daniel Saad Nogueira
Nunes

IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

- 1 Introdução
- 2 Ordenação
- 3 Busca



Sumário

1 Introdução



Ordenação

- Um dos problemas clássicos da Ciência da Computação é o de ordenar uma coleção de elementos de acordo com alguma relação de ordem.
- Exemplos:
 - ▶ Ordenar uma sequência de números naturais de acordo com a ordem crescente.
 - ▶ Ordenar uma sequência de palavras de acordo com a ordem lexicográfica.
 - ▶ Ordenar uma sequência de pessoas de acordo com o número de CPF.



Ordenação

- Formalmente o problema da ordenação consiste de uma sequência de entrada $V = (v_0, \dots, v_{n-1})$ e produz uma sequência de saída $V' = (v'_0, \dots, v'_{n-1})$, uma permutação de V , com $v'_0 < v'_1 \dots < v'_{n-1}$, sendo $<$ uma relação de ordem.



Busca

- Outro problema interessante na Ciência da Computação é o problema de busca. Dados uma sequência V de elementos e um elemento k , o objetivo é dizer se k ocorre em V , e em caso afirmativo, determinar a posição de ocorrência na sequência.
- A busca pode ser utilizada em uma sequência ordenada ou não.
- Contudo, se a sequência estiver ordenada, técnicas especiais de busca permitem responder rapidamente a pergunta.
- Muitos problemas, para serem resolvidos eficientemente, requerem que primeiro ordenemos a sequência de entrada para então buscar elementos com mais facilidade.



Ordenação e Busca

- Nesta aula veremos como ordenar sequências de elementos e buscar dados sobre elas eficientemente.



Sumário

2 Ordenação



Sumário

2 Ordenação

- sort
- funções comparadoras
- operador <
- funtores
- funções lambda



Sort

- O C++ possui uma função `sort`, que pode ser utilizado sobre objetos do tipo `vector`.
- Recebe dois argumentos: o iterador para o início e para o fim do vetor.
- Esta função ordena os itens de acordo com o operador `<`.
- A complexidade do algoritmo utilizado é $\Theta(n \lg n)$.



Sort

```
1 vector<int> v;  
2 // ...  
3 sort(v.begin(), v.end());
```



Sort

- A função `sort` também pode ser utilizada em arrays convencionais.
- Basta passar o endereço de início do vetor e o endereço à direita do endereço final.



Sort

```
1  int v[100];  
2  // ...  
3  sort(v,v+100);
```



Sumário

2 Ordenação

- sort
- **funções comparadoras**
- operador <
- funtores
- funções lambda



Funções comparadoras

- Opcionalmente, a função `sort` recebe como terceiro parâmetro a função a ser utilizada para comparação.
- A função deverá receber como argumento dois objetos (ou duas referências) do tipo do vetor a ser ordenado e retornar um `bool` com valor `true` se o primeiro item deve preceder o segundo após a ordenação e `false`, caso contrário.



Funções comparadoras

```
struct pessoa {  
    string nome;  
    int idade;  
    double renda_mensal;  
};
```



Funções comparadoras

```
bool cmp_pessoa(const pessoa &lhs, const pessoa &rhs) {  
    // A função estabelece uma ordem entre duas pessoas  
    // Critério: primeiro pessoas mais novas e  
    // caso tenham a mesma idade, seleccionar aquela  
    // com maior renda mensal.  
    if (lhs.idade < rhs.idade) {  
        return true;  
    }  
    if (lhs.idade == rhs.idade) {  
        return lhs.renda_mensal > rhs.renda_mensal;  
    }  
    return false;  
}
```



Funções comparadoras

```
vector<peessoa> v;  
// ...  
sort(v.begin(), v.end(), cmp_peessoa);
```



Sumário

2 Ordenação

- sort
- funções comparadoras
- **operador <**
- funtores
- funções lambda



Operador <

- Caso o operador `<` tenha sido sobrecarregado por uma classe, ele é utilizado para estabelecer a ordem dos elementos se uma função comparadora não for colocada como terceiro parâmetro do `sort`.



Operador <

```
1 struct pessoa {
2     string nome;
3     int idade;
4     double renda_mensal;
5     // Critério: primeiro pessoas mais novas e
6     // caso tenham a mesma idade, selecionar aquela
7     // com maior renda mensal.
8     bool operator<(const pessoa &rhs) const {
9         if (this->idade < rhs.idade) {
10             return true;
11         }
12         if (this->idade == rhs.idade) {
13             return this->renda_mensal > rhs.renda_mensal;
14         }
15         return false;
16     }
17 };
```



Sumário

2 Ordenação

- sort
- funções comparadoras
- operador <
- **funtores**
- funções lambda



Funtores

- Também é possível utilizar funtores para estabelecer a ordem entre os elementos.
- Um functor, assim como a função comparadora, recebe dois objetos (ou referências) do tipo do vetor a ser ordenado e retorna verdadeiro se e somente se o primeiro objeto deve preceder o segundo após a ordenação.
- Uma instância da classe que implementa o functor deve ser passado como terceiro parâmetro.



Functores

```
1 struct cmp_pessoa {
2     bool operator()(const pessoa &lhs, const pessoa &rhs) {
3         // Critério: primeiro pessoas mais novas e
4         // caso tenham a mesma idade, selecionar aquela
5         // com maior renda mensal.
6         if (lhs.idade < rhs.idade) {
7             return true;
8         }
9         if (lhs.idade == rhs.idade) {
10            return lhs.renda_mensal > rhs.renda_mensal;
11        }
12        return false;
13    }
14 };
```



Funtores

```
vector<peessoa> v;  
// ...  
sort(v.begin(), v.end(), cmp_peessoa());
```



Sumário

2 Ordenação

- sort
- funções comparadoras
- operador <
- funtores
- funções lambda



Funções lambda

- Também é possível informar uma função lambda no terceiro argumento da função `sort`.
- A assinatura da função deve seguir as mesmas regras da função comparadora explicada anteriormente.



Funções lambda

```
1  vector<peessoa> v;
2  // ...
3  sort(v.begin(), v.end(), [](peessoa &lhs, peessoa &rhs) -> bool {
4      if (lhs.idade < rhs.idade) {
5          return true;
6      }
7      if (lhs.idade == rhs.idade) {
8          return lhs.renda_mensal > rhs.renda_mensal;
9      }
10     return false;
11 });
```



Sumário

3 Busca



Busca

- Agora que sabemos ordenar uma sequência qualquer de elementos, podemos pesquisar sobre ela mais eficientemente utilizando uma técnica chamada de **busca binária**.



Busca Binária

- Levando em consideração que o vetor está ordenado, podemos efetuar a busca binária.
- Ela funciona da seguinte forma. Suponha que a sequência $V[0, n - 1]$.
- Inicialmente calcula-se o ponto médio da sequência: $m \leftarrow \lfloor \frac{n}{2} \rfloor$.
- Se a chave corresponde à $V[m]$, então encontramos a chave no ponto médio.



Busca Binária

- Se a chave não é igual ao elemento $V[m]$, temos duas opções.
 - ① A chave é menor do que $V[m]$.
 - ② A chave é maior do que $V[m]$.
- No primeiro caso, sabemos que se a chave se encontra em V , ela deve estar no intervalo $V[0, m - 1]$.
- No segundo, concluímos que se a chave está em V , ela se encontra em $V[m + 1, n - 1]$.
- Isso é permitido pois sabemos que o vetor está ordenado. Logo, sabemos que todos os elementos à esquerda de $V[m]$ são menores ou iguais à $V[m]$. Simetricamente, todos os elementos à direita de $V[m]$ são maiores ou iguais a $V[m]$.



Busca Binária

- Caso a chave não corresponda ao elemento $V[m]$, continuamos a busca no subvetor à esquerda de $V[m]$ ou à direita de $V[m]$ utilizando a mesma estratégia!
- Descartamos metade dos elementos com uma única comparação!
- Complexidade: $\Theta(\lg n)$.



Busca Binária

chave = 4

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14



Busca Binária

$chave = 4$

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14
l				m				r

$chave < V[m]$



Busca Binária

$chave = 4$

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14
↑	↑		↑					
l	m		r					

$chave > V[m]$



Busca Binária

chave = 4

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14

↑
l
m

↑
r

chave == $V[m]$



Busca Binária

chave = 11

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14



Busca Binária

$chave = 11$

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14
l				m				r

$chave > V[m]$



Busca Binária

chave = 11

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14



l



m



r

chave > $V[m]$



Busca Binária

chave = 11

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14



l

m



r

chave < $V[m]$



Busca Binária

chave = 11

0	1	2	3	4	5	6	7	8
1	3	4	6	7	8	10	13	14

r l

Chave não encontrada!



Busca Binária

```
1  int busca_binaria(vector<int> &v, int key) {
2      int l = 0;
3      int r = v.size() - 1;
4      while (l <= r) {
5          int mid = l + (r - 1) / 2; // evita overflow
6          if (chave == v[mid]) {
7              return mid; /**Retorna a posição da chave**/
8          } else if (chave < v[mid]) {
9              r = mid - 1;
10         } else {
11             l = mid + 1;
12         }
13     }
14     return -1; /**Chave não encontrada**/
15 }
```



Lower Bound e Upper Bound

- O C++ providencia duas funções que utilizam a técnica de busca binária. Ambas recebem um iterador para o início e o fim da sequência a ser pesquisada. Só podem ser utilizadas sob um vetor ordenado.
- `lower_bound`: retorna um iterador para o primeiro elemento que **não é menor** do que a chave pesquisada ou o iterador `end()`;
- `lower_bound` retorna um iterador para o primeiro elemento que **é maior** do que a chave pesquisada ou o iterador `end()`;



Lower Bound e Upper Bound

```
1 int main() {
2     vector<int> v = {10, 20, 30, 30, 20, 10, 10, 20};
3     sort(v.begin(), v.end()); // 10 10 10 20 20 20 30 30
4     auto low = lower_bound(v.begin(), v.end(), 20); // ~
5     auto up = upper_bound(v.begin(), v.end(), 20); // ~
6     cout << "lower_bound at position " << (low - v.begin()) << '\n';
7     cout << "upper_bound at position " << (up - v.begin()) << '\n';
8     return 0;
9 }
```



Lower Bound e Upper Bound

- Em outras palavras, se `lb` é a posição retornada por `lower_bound` e `ub` é a posição retornada por `upper_bound`, então os elementos de valor pesquisado estão no intervalo $[lb, ub)$ do vetor ordenado.



Equal range

- A função `equal_range` funciona como uma aplicação de `lower_bound` e `upper_bound`.
- Ela retorna um **par** de iteradores para as posições de `lower_bound` e `upper_bound`



Equal range

```
1  int main() {
2      vector<int> v = {10, 20, 30, 30, 20, 10, 10, 20};
3      sort(v.begin(), v.end());    // 10 10 10 20 20 20 30 30
4      auto [low,up] = equal_range(v.begin(), v.end(), 20);
5      cout << "lower_bound at position " << (low - v.begin()) << '\n';
6      cout << "upper_bound at position " << (up - v.begin()) << '\n';
7      return 0;
8  }
```