Ordenação e Busca

Introdução à Programação Competitiva



Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga



- Introdução
- Ordenação
- Busca

IntroduçãoOrdenaçãoBusca



Sumário

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 $V=(v_0,\ldots,v_{n-1})$ e produz uma sequência de saída $V'=(v'_0,\ldots,v'_{n-1})$, uma permutação de V, com $v'_0< v'_1\ldots < v'_{n-1}$, 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.



Ordenação





- sort
- funções comparadoras
- operador <
- functores
- funções lambda



- 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 $\Theta(n \lg n)$.



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



- 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.



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



- Ordenação
 - sort
 - funções comparadoras
 - operador <
 - functores
 - funções lambda



- 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.



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



```
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, selecionar aquela
   // com major renda mensal.
    if (lhs.idade < rhs.idade) {
        return true;
   }
    if (lhs.idade == rhs.idade) {
        return lhs.renda_mensal > rhs.renda_mensal;
   return false;
```



```
vector<pessoa> v;
// ...
sort(v.begin(), v.end(), cmp_pessoa);
```





- sort
- funções comparadoras
- $\bullet \ {\sf operador} <$
- functores
- 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 <

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





- sort
- funções comparadoras
- ullet operador <
- functores
- funções lambda



Functores

- Também é possível utilizar functores 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

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



Functores

```
vector<pessoa> v;
// ...
sort(v.begin(), v.end(), cmp_pessoa());
```





- sort
- funções comparadoras
- operador <
- functores
- 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

```
vector<pessoa> v;
// ...
sort(v.begin(), v.end(), [](pessoa &lhs, pessoa &rhs) -> bool {
    if (lhs.idade < rhs.idade) {
        return true;
    }
    if (lhs.idade == rhs.idade) {
        return lhs.renda_mensal > rhs.renda_mensal;
    }
    return false;
}
```







Busca

 Agora que sabemos ordenar uma sequência qualquer de elementos, podemos pesquisar sobre ela mais eficientemente utilizando uma técnica chamada de 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: $m \leftarrow \lfloor \frac{n}{2} \rfloor$.
- \bullet Se a chave corresponde à $V[m], {\tt ent} \tilde{\tt ao}$ encontramos a chave no ponto médio.



- Se a chave não é igual ao elemento V[m], temos duas opções.
 - lacksquare 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].



- ullet 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: $\Theta(\lg n)$.

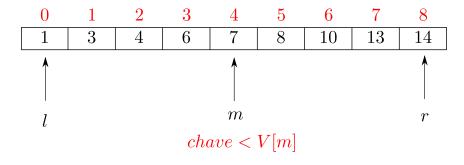


$$chave = 4$$

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

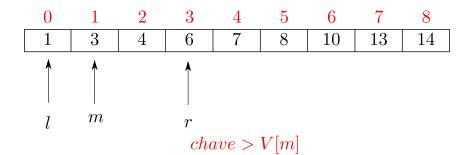


$$chave = 4$$



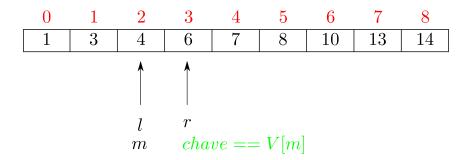


$$chave = 4$$





$$chave = 4$$



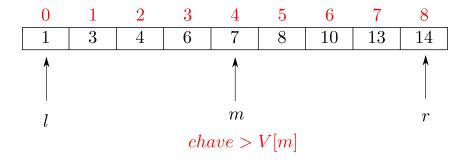


$$chave = 11$$

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

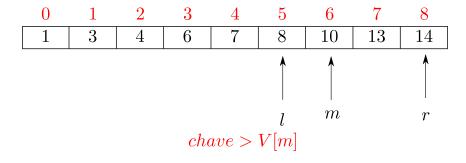


$$chave = 11$$



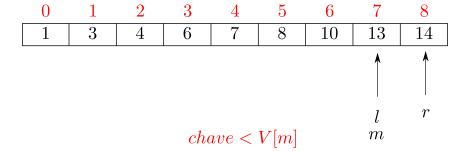


$$chave = 11$$



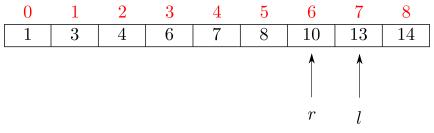


$$chave = 11$$





$$chave = 11$$





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

```
int main() {
    vector<int> v = {10, 20, 30, 30, 20, 10, 10, 20};
    sort(v.begin(), v.end());
    auto low = lower_bound(v.begin(), v.end(), 20); //
    auto up = upper_bound(v.begin(), v.end(), 20); //
    cout << "lower_bound at position " << (low - v.begin()) << '\n';
    cout << "upper_bound at position " << (up - v.begin()) << '\n';
    return 0;
}</pre>
```



Lower Bound e Upper Bound

ullet Em outras palavras, se 1b 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

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