MC-102 — Aula 14
Busca e Ordenação

Instituto de Computação – Unicamp

Primeiro Semestre de 2006

Roteiro

Busca

Definição do problema

- Dada uma coleção de n elementos, pretende-se saber se um determinado elemento `valor` está presente nessa coleção.
- Para efeitos práticos, vamos supor que essa coleção é implementada como sendo um vetor de n elementos inteiros: `vetor[0]..vetor[n-1]`.

Busca seqüencial

Uma solução: busca seqüencial

Percorrer o vetor desde a primeira posição até a última. Para cada posição i, comparamos `vetor[i]` com `valor`.

- Se forem iguais dizemos que `valor` existe.
- Se chegarmos ao fim do vetor sem sucesso dizemos que `valor` não existe.

Busca seqüencial

- 1º passo — inicialização

```
i = 0;
encontrado = 0; /*Falso*/
```

- 2º passo — busca

```
while (i < TAMANHO && !encontrado) {
  if (vetor[i] == valor) {
    encontrado = 1; /*Verdadeiro*/
  } else {
    i++;
  }
}
```

Busca seqüencial

- 3º passo — tratamento do resultado

```
if (encontrado) {
  printf ("Valor %d está na posicao %d\n",
          vetor[i], i);
} else {
  printf ("Valor %d não encontrado\n",
          valor);
}
```

Veja o programa completo em `sequencial.c`.

Busca seqüencial

Quanto tempo a busca seqüencial demora para executar? Em outras palavras, quantas vezes a comparação `valor == vetor[i]` é executada?

- Caso valor não esteja presente no vetor, n vezes.
- Caso valor esteja presente no vetor:
  - 1 vez no melhor caso (valor está na primeira posição).
  - n vezes no pior caso (valor está na última posição).
  - $\frac{n}{2}$ vezes no caso médio.

Veja exemplos em `seq-random.c` e `seq-random2.c`.

Busca binária

Supondo agora que o vetor inicial está ordenado em ordem crescente. Será que é possível resolver o problema de modo mais eficiente?

Outra solução: busca binária

- Caso a lista esteja ordenada, sabemos que, para qualquer i e j, $i < j$, se, e somente se, $A[i] \leq A[j]$.
- Portanto, comparando um determinado elemento com o elemento procurado, saberemos:
  - Se o elemento procurado é o elemento comparado;
  - se ele está antes do elemento comparado ou;
  - se está depois.

Busca binária

Outra solução: busca binária

- Se fizermos isso sempre com o elemento do meio da lista, a cada comparação dividiremos a lista em duas, reduzindo nosso tempo de busca.
- Se em um determinado momento o vetor, após sucessivas divisões, tiver tamanho zero, então o elemento não está no vetor.

Veja uma implementação em `binaria.c`.

Busca binária

Quanto tempo a busca binária demora para executar? Em outras palavras, quantas vezes a comparação `valor == vetor[i]` é executada?

- Caso valor não exista no vetor, $log_2(n)$ vezes.
- Caso valor exista no vetor:
  - 1 vez no melhor caso (valor é a mediana do vetor).
  - $log_2(n)$ vezes no caso médio.
- O logaritmo de base 2 aparece porque sempre dividimos o intervalo ao meio.

Veja um exemplo em `bin-random.c`.

Qual dos dois algoritmos é melhor?

- Para $n = 1000$, o algoritmo de busca seqüencial irá executar 1000 comparações no pior caso, 500 operações no caso médio.
- Por sua vez, o algoritmo de busca binária irá executar 10 comparações no pior caso, para o mesmo n.
- O algoritmo de busca binária supõe que o vetor está ordenado. Ordenar um vetor também tem um custo, superior ao custo da busca seqüencial.
- Se for para fazer uma só busca, não vale a pena ordenar o vetor. Por outro lado, se pretendermos fazer muitas buscas, o esforço da ordenação já poderá valer a pena.

Ordenação

Definição do problema

Dada uma coleção de n elementos, representada em um vetor de 0 a $n-1$, deseja-se obter uma outra coleção, cujos elementos estejam ordenados segundo algum critério de comparação entre os elementos.

Ordenação por inserção

Uma solução: ordenação por inserção

Para cada elemento do vetor, coloque-o em sua posição correspondente.

- Durante o processo de ordenação por inserção a lista fica dividida em duas sub-listas, uma com os elementos já ordenados e a outra com elementos ainda por ordenar.
- No início, a sub-lista ordenada é formada trivialmente apenas pelo primeiro elemento da lista.
- A cada etapa i, o i-ésimo elemento é inserido em seu lugar apropriado entre os $i - 1$ elementos já ordenados. Os índices dos itens a serem inseridos variam 2 a *tamanho*.

Ordenação por inserção

Uma possível execução

	V[0]	V[1]	V[2]	V[3]	V[4]
Etapa	9	8	7	6	5
1	{8	9}	{7	6	5}
2	{7	8	9}	{6	5}
3	{6	7	8	9}	{5}
4	{5	6	7	8	9}

Veja uma implementação em `insercao.c`.

Ordenação por inserção

Quanto tempo a ordenação por inserção demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

- Em cada etapa i são executadas no máximo, i comparações pois o loop interno é executado para j de $i - 1$ até 0. Como i varia de 0 até $tamanho - 1$, o número total de comparações para ordenar a lista toda é da ordem de $tamanho^2$.
- Para cada uma das comparações é feita a troca de posição de dois elementos. Desta forma, o número de trocas também é da ordem de $tamanho^2$.

Ordenação por seleção

Uma outra solução: ordenação por seleção

A ordenação por seleção consiste, em cada etapa, em selecionar o maior (ou o menor) elemento e alocá-lo em sua posição correta dentro da futura lista ordenada.

- Como na ordenação por inserção, na ordenação por seleção a lista com m registros fica decomposta em duas sub-listas, uma contendo os ítens já ordenados e a outra com os restantes ainda não ordenados.
- No início a sub-lista ordenada é vazia e a outra contém todos os demais. No final do processo a sub-lista ordenada apresentará $tamanho - 1$ ítens e a outra apenas 1.

Ordenação por seleção

Uma possível execução

	V[0]	V[1]	V[2]	V[3]	V[4]
Etapa	5	9	1	4	3
1	{5	3	1	4}	{9}
2	{4	3	1}	{5	9}
3	{1	3}	{ 4	5	9}
4	{1}	{3	4	5	9}

Veja uma implementação em `selecao.c`.

Ordenação por seleção

Quanto tempo a ordenação por seleção demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

- A comparação é feita no loop mais interno. Para cada valor de i são feitas $tamanho - i$ comparações dentro do loop mais interno. Como i varia de 0 até $tamanho - 1$, o número total de comparações para ordenar a lista toda é, novamente, da ordem de $tamanho^2$.
- Porém, a troca de dois elementos do vetor só ocorre quando o maior elemento foi encontrado. Como i varia de 0 até $tamanho - 1$, o número máximo de trocas é da ordem de $tamanho$.

Ordenação por bolha

Uma outra solução: ordenação por bolha

Um método simples de ordenação por troca é a estratégia conhecida como bolha que consiste em cada etapa "borbulhar", através de trocas, o maior elemento para o fim da lista.

- Inicialmente percorre-se a lista dada da esquerda para a direita, comparando pares de elementos consecutivos, trocando de lugar os que estão fora da ordem. Em cada troca, o maior elemento é deslocado uma posição para a direita.

Ordenação por bolha

Uma possível execução

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
1	10	9	7	6	V[0] e V[1]
	9	10	7	6	V[1] e V[2]
	9	7	10	6	V[2] e V[3]
	9	7	6	10	Fim da Varredura 1

Ordenação por bolha

Após a primeira varredura o maior elemento encontra-se alocado em sua posição definitiva na lista ordenada. Podemos deixá-lo de lado e efetuar a segunda varredura na sub-lista $V[0], V[1], V[2]$.

Uma possível execução

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
2	9	7	6	10	V[0] e V[1]
	7	9	6	10	V[1] e V[2]
	7	6	9	10	Fim da Varredura 2

Ordenação por bolha

Após a segunda varredura o maior elemento da sub-lista $V[0], V[1], V[2]$ encontra-se alocado em sua posição definitiva. A próxima sub-lista a ser ordenada é $V[0], V[1]$.

Uma possível execução

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
3	7	6	9	10	V[0] e V[1]
	6	7	9	10	Fim da Varredura 3

Veja uma implementação em `bubble.c`.

Ordenação por bolha

Quanto tempo a ordenação por bolha demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

- No algoritmo da bolha observamos que existem $tamanho - 1$ varreduras (etapas) e que em cada varredura o número de comparações diminui de uma unidade, variando de $tamanho - 1$ até 1. Portanto, para ordenar a lista toda novamente precisamos de aproximadamente $tamanho^2$ comparações.
- Para cada uma destas comparações, pode ser necessário fazer uma troca. Logo, o número máximo de trocas a ser realizada será da ordem de $tamanho^2$.