**DCC – ICEX – UFMG – ESTRUTURAS DE DADOS  
Aluno: Antonio Carlos da Anunciação 2018019443**

Lista de Exercícios **4** – Ordenação

**1 –** Em sala, vimos 7 diferentes métodos de ordenação: Bolha, Seleção, Inserção, Quicksort, Mergesort, Heapsort e Shellsort. Monte uma tabela comparativa dos métodos contendo: uma breve descrição (em suas palavras) de como é o método, suas principais vantagens e desvantagens. Monte a tabela de acordo com o modelo abaixo:

|  |  |  |  |
| --- | --- | --- | --- |
| **Método** | **Descrição** | **Vantagens** | **Desvantagens** |
| Bolha | Passa no arquivo e troca os elementos adjacentes que estão fora de ordem, até que os registro estejam ordenados | Estável, simples de implementar | Não adaptável, a quantidade de trocas pode inviabilizar o método |
| Seleção | Seleciona a posição do maior/menor elemento, e o troca para sua posição correta | Custo linear, bom para ser usado quando temos arquivos grandes | Não estável, não adaptável. |
| Inserção | As chaves são ordenadas no sentido do vetor, o método avança e troca as posição A[ i ] com as A[ j ] com j < i, ou seja o método faz a troca enquanto a condição de A[ i ] > A[ j ] for verdadeira | Algoritmo estável, ideal para ser usado om arquivos pré-ordenados. | Alto custo de movimentação no vetor. |
| Quicksort | Conquistar e dividir, divide o problema em um conjunto de n itens em dois grupos menores e os ordena de forma independente, no final os resultados são combinados e termos o vetor original ordenado. | Extremamente eficiente para ordenar arquivos, pilha auxiliar pequena, complexidade para comparações nlog(n) em média. | Pior caso tem complexidade o(n²), difícil implementação, método não estável. |
| Mergesort | O mergesort é baseado na ideia do quicksort, de dividir e conquistar, após ter dividido totalmente o vetor ele aplica o quicksort e vai unindo os resultados. | Algoritmo estável, custo nlog(n), bom para quando o custo para o pior caso não pode ser tolerado. | Algoritmo não adaptável, requer um espaço extra proporcional a n. |
| Heapsort | Mesmo principio de operação que o Seleção, porém para reduzir o custo é montado um fila de prioridades (Heap) com a seguinte regra: c[i] >= c[2i], c[i] >= c[2i + 1] | Comportamento sempre nlog(n). | Anel interno complexo.  Não é estável. |
| Shellsort | Variação do algoritmo de inserção, a classificação é feita separando os elementos um do outro e reduz sucessivamente o intervalo entre os elementos a serem classificados. | Fácil implementação. | Método não é estável, tempo de execução sensível a ordem inicial dos arquivos. |

**2 –** Para os métodos simples (Bolha, Seleção e Inserção), dado o vetor abaixo, mostre as trocas que serão feitas e como ficará o vetor para as 3 primeiras trocas efetuadas em cada método:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **2** | **5** | **1** | **7** | **6** | **8** | **4** | **3** |

***Bolha: 2, 1, 5, 6, 7, 4, 8, 3***

***Seleção: 1, 2, 3, 7, 6, 8, 4, 5***

***Inserção: 1, 2, 5, 6, 7, 8, 4, 3***

**3 –** Dado o vetor abaixo:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **3** | **6** | **4** | **2** | **5** | **8** | **7** | **1** |

a) O vetor é um heap? Justifique

***Não, para ser um heap o vetor deve obedecer a seguinte ordem: V[i] > V[2i], e V[i] > V[2i+1], sendo assim:***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **3** | **6** | **4** | **2** | **5** | **8** | **7** | **1** |
| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |

***Podemos ver que o vetor não obedece a ordem mostra acima.***

b) Caso não seja, mostre o passo a passo o método para transformá-lo em heap.

|  |
| --- |
| void heap(int lista[], int tamanho, int i){  int maior = i;  int esquerda = 2\*i + 1;  int direita = 2\*i + 2;    if (esquerda < tamanho && lista[esquerda] > lista[maior]) maior = esquerda;  if (direita < tamanho && lista[direita] > lista[maior]) maior = direita;    if (maior != i){  troca(&lista[i], &lista[maior]);  heap(lista, tamanho, maior);  }  }    void heapsort(int lista[], int tamanho){  for (int i = tamanho/2 - 1; i >= 0; i--) heap(lista, tamanho, i);  for (int i = tamanho-1; i>=0; i--){  troca(&lista[0], &lista[i]);  heap(lista, i, 0);  }  } |

c) Ordene o vetor utilizando o heapsort.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** |

d) No momento de montar o heap, funcionária se se chamasse a função Refaz direto para todo vetor? Por que?

***Não, porque o Heap é montado ramo a ramo da “arvore”.***

e) Após colocar cada elemento em sua posição no heapsort, é preciso reconstruir o heap (como feito inicialmente)? Por que?

***É por causa da definição do método, onde o objetivo é ter acesso rápido ou maior/menor elemento.***

**4 –** Pensando na ordenação utilizando o Quicksort, o que acontece quando:

a) Se o vetor tiver ordenado, e o pivô for o 1º. ou último elemento do vetor. Justifique.

***Teremos o pior caso, porque o método sera chamado recursivamente todas as n vezes, onde o tamanho do vetor.***

b) Se o vetor tiver ordenado, e o pivô for a mediana de 3. Justifique.

***Melhor caso médio, dado que a chance de evitar os piores casos (bordas)são aumentadas.***

**5 –** Considerando o vetor abaixo:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | **3** | **9** | **4** | **8** | **2** | **7** | **6** | **5** | **1** |

a) Ordene o vetor usando o Quicksort. Considere o pivô como sendo o elemento central.  
A cada partição resolva primeiro o lado esquerdo, até que 1 2 3 estejam em ordem.  
b) Ordene o vetor usando o Mergesort. Mostre claramente o passo-a-passo até que o primeiro lado do vetor esteja ordenado.  
c) Ordene o vetor usando o Shellsort. Mostre como ficará o vetor, após as trocas para h=4 e h=3.

**6 –** Ordene o vetor abaixo utilizando o método do **Quicksort Não Recursivo**, selecionando o elemento central como pivô:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **23** | **100** | **53** | **39** | **8** | **15** | **91** | **74** | **51** | **19** |

a) Ordene a primeira vez, selecionando sempre o menor lado para empilhar.  
b) Ordene novamente, selecionando sempre o maior lado para empilhar. o vetor usando o Quicksort.  
c) A seleção do lado a ser empilhado faz diferença na altura da pilha? Qual? Por que?

**7 –** Ordene este mesmo vetor utilizando o Quicksort recursivo. Ordene o vetor passo a passo. Neste caso faz diferença se o maior ou menor lado é executado primeiro? Explique