Curso: Ciência da Computação

Disciplina: Estruturas de Dados

Profª Luciana Ap. Oliveira Betetto

Aula 06 - Métodos de Ordenação Interna

1.1 Introdução

A eficiência do manuseio de dados muitas vezes pode ser substancialmente aumentada se os dados forem dispostos de acordo com algum critério de ordem.

Ordenar indica o processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. A atividade de colocar as coisas em ordem está presente na maioria das aplicações em que os objetos armazenados têm de ser pesquisados e recuperados.

O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. Imagine como seria difícil utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética. O mesmo pode ser dito com relação a dicionários, índices de livros, folhas de pagamento, contas bancárias, tabelas, arquivos e outros materiais organizados alfabeticamente. A conveniência de usar dados ordenados é inquestionável e precisa ser aplicada também à ciência da computação.

Os métodos de ordenação ilustram claramente como um determinado problema pode ser resolvido por meio de diferentes algoritmos, cada um apresentando vantagens e desvantagens, que devem ser consideradas sob o angulo da aplicação a que se destinam.

Diversos fatores influem na eficiência de um algoritmo de ordenação (isto implica que não há um único algoritmo de ordenação que seja melhor para todas as situações), tais como:

- Número de registros a serem ordenados;
- A memória interna disponível (se cabe ou não todos os registros);

- O grau de ordenação já existente;
- Os tipos de meios de armazenamento em que os registros estão armazenados;
- A complexidade e os requisitos de armazenamento do algoritmo de ordenação;
- Se os registros deverão ou não ser removidos ou inseridos periodicamente.

Um método de ordenação é dito "estável" se a ordem relativa dos itens com chaves iguais mantém-se inalterada depois de efetuada a ordenação. Em geral, esta propriedade é desejável, especialmente quando os elementos já estiverem ordenados em relação a uma ou mais chaves secundárias. Por exemplo, se uma lista alfabética de nomes de funcionários de uma empresa é ordenada pelo campo salário, então um método estável produz uma lista em que os funcionários com o mesmo salário aparecem em ordem alfabética.

Os métodos de ordenação são em geral, classificados em dois grandes grupos:

- Ordenação interna
  - Se o arquivo a ser ordenado é pequeno, ou seja, a lista de registros pode ser mantida inteiramente na memória principal durante a ordenação.
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.

- Ordenação externa
  - Se o arquivo a ser ordenado não cabe na memória principal e, por isso, tem de ser armazenado em fita ou disco.
  - Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.

Classificação dos métodos de ordenação interna:

Métodos simples ou diretos:

- Adequados para conjuntos pequenos;
- Requerem $O(n^2)$ [1] comparações;
- Produzem programas pequenos e fáceis de entender.

[1] $O(n^2)$: um algoritmo de complexidade $O(n^2)$ é dito ter complexidade quadrática; ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro; quando n é mil, o número de operações é da ordem de 1 milhão; sempre que n dobra, o tempo de execução é multiplicado por 4; úteis para resolver problemas de tamanho relativamente pequenos.

Métodos eficientes:

- Adequados para conjuntos maiores;
- Requerem O(n log n) [2] comparações;
- Usam menos comparações;
- As comparações são mais complexas nos detalhes.

Os algoritmos trabalham sobre os registros de um arquivo. Apenas uma parte do registro, chamada chave, é utilizada para controlar a ordenação. Além da chave, podem existir outros componentes em um registro, os quais não tem influência no processo de ordenar, a não ser pelo fato de que permanecem com a mesma chave. A escolha do tipo para a chave é arbitrária. Qualquer tipo sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado. As ordens numérica e alfabética são as mais usuais.

A ordenação final de dados pode ser obtida em uma variedade de modos, mas somente algumas delas podem ser consideradas significativas e eficientes. Para decidir qual método é o melhor, certos critérios de eficiência têm que ser estabelecidos, e um método para comparar quantitativamente diferentes algoritmos precisa ser selecionado.

Para tornar a comparação independente da máquina, certas propriedades críticas dos algoritmos de ordenação precisam ser definidas quando se comparam métodos alternativos. Duas dessas propriedades são o número de comparações e o número de movimentos de dados. Para ordenar um conjunto de dados, eles têm que ser comparados e movidos conforme necessário; a eficiência dessas duas operações depende do tamanho do conjunto de dados.

A quantidade extra de memória auxiliar utilizada pelo algoritmo é também um aspecto importante. Uma característica fundamental dos métodos de ordenação corresponde ao uso econômico da memória disponível (algoritmos ideais são aqueles que não utilizem estruturas auxiliares para a ordenação). Os métodos que utilizam a estrutura vetor e executam a permutação dos itens no próprio vetor, exceto para a utilização de uma pequena tabela ou pilha, são os preferidos – esses métodos são conhecidos como algoritmos que ordenam *in situ*.

Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto para ordenar um arquivo. Considerando n o número de elementos a serem ordenados, as medidas de complexidade relevantes para os algoritmos de ordenação interna são:

- Número de comparações entre as chaves (C(n))
- Número de movimentos realizados (M(n))

[2] O(n log n): típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e junta as soluções depois; quando n é 1 milhão, nlog2n é cerca de 20 milhões.

1.2 Métodos de Ordenação Diretos (número de comparações da ordem de n^2)

Consistem em métodos simples e de fácil compreensão e apresentam bom desempenho para conjuntos de dados pequenos.

1.2.1 Inserção Direta

Método preferido dos jogadores de cartas**.** Este é um dos mais simples algoritmos de ordenação. Ele consiste em, dado um vetor para ordenação, percorrer elemento por elemento deslocando-o e inserindo-o na posição ordenada. A idéia é formar um bloco de valores ordenados e outro de desordenados, e ir passando os valores de um bloco a outro.

Uma ordenação por inserção inicia considerando os dois primeiros elementos: data[0] e data[1]. Se eles estão fora de ordem, um intercâmbio se realiza. Então, um terceiro elemento, data[2], é considerado e inserido em seu lugar apropriado. Se data[2] é menor do que data[0] e data[1], estes dois são mudados em uma posição; data[0] é colocado na posição 1, data[1] na posição 2 e data[2] na posição 0. Se data[2] é menor do que data[1] e não menor do que data[0], somente data[1] é movido para a posição 2 e seu lugar é tomado por data[2]. Se, finalmente, data[2] não é menor do que seus predecessores, ele permanece em sua posição. Cada elemento data[i] é inserido em seu local apropriado j de tal forma que $0 \leq j \leq i$ e todos os elementos maiores que data[i] são movidos em posição.

Algoritmo:

- Em cada passo a partir de i=2 faça:
  - Selecione o i-ésimo item da sequência fonte.
  - Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.

Uma vantagem em utilizar a ordenação por inserção é que ela ordena somente quando é realmente necessário. Se os elementos já estão em ordem, nenhum movimento substancial é realizado. Uma desvantagem é que, se um item está sendo inserido, todos os elementos maiores do que ele têm que ser movidos. A inserção não é localizada e pode exigir que se mova um número significativo de elementos, gerando um retardo substancial na execução.

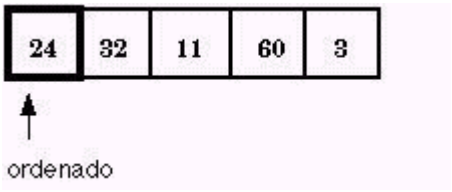
Exemplo 1: (as chaves em negrito representam a sequência destino):

	1	2	3	4	5	6
Chaves iniciais:	***O***	*R*	*D*	*E*	*N*	*A*
i = 1	***O***	***R***	*D*	*E*	*N*	*A*
i = 2	***D***	***O***	***R***	*E*	*N*	*A*
i = 3	***D***	***E***	***O***	***R***	*N*	*A*
i = 4	***D***	***E***	***N***	***O***	***R***	*A*
i = 5	***A***	***D***	***E***	***N***	***O***	***R***

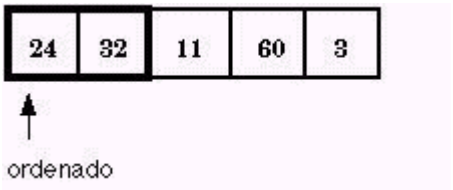
No Exemplo 1, a colocação do item no seu lugar apropriado na sequência de destino é realizada movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia. Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo: (i) um item com chave menor que o item em consideração é encontrado; (ii) o final da seqência destino é atingido à esquerda.

Exemplo 2:

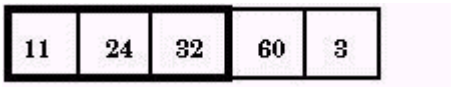
No início, o bloco de ordenados é apenas o primeiro elemento.



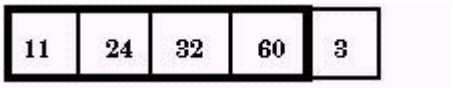
Então, faz-se o segundo elemento uma chave de comparação e compara-se com o primeiro elemento. Como não é maior, ele é inserido após o primeiro valor (mesma posição).



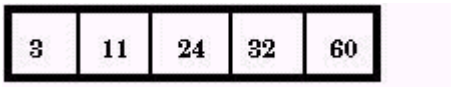
Na segunda comparação, o terceiro valor é a nova chave e é comparado com o último do bloco ordenado. Como a chave é menor, é comparada com o próximo. Como ainda é menor, a chave é inserida então na primeira posição e os demais valores deslocados para a direita (v[i+1] = v[i]).



Agora a chave passa a ser o quarto valor, e este é comparado com os valores anteriores, sempre começando do último. Como este é menor, então ele é inserido após o último elemento, ficando na mesma posição.



Por fim, a chave é o último elemento do vetor, e é então comparado com todos os demais, sempre começando do último: 3 é comparado com 60, 32, 24 e 11, como sempre é menor, será colocado na primeira posição, e os demais valores serão deslocados uma posição para a direita. E o vetor final é o que vemos a seguir:



Análise:

- Complexidade: $O(n^2)$
- Número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- Número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o conjunto está “quase” ordenado.
- É um bom método quando se deseja adicionar poucos itens a um conjunto já ordenado e depois obter outro conjunto ordenado, pois o custo é linear.
- É um método estável, pois ele deixa os registros com chaves iguais na mesma posição relativa.

1.2.2 Seleção Direta

A ordenação por seleção é uma tentativa de localizar as trocas dos elementos do conjunto, encontrando um elemento mal colocado primeiro e ajustando-o em sua posição final.

É considerado um dos algoritmos mais simples de ordenação, cujo princípio de funcionamento é o seguinte: selecione o menor item do conjunto e a seguir troque-o com o item que está na primeira posição. Repita essas duas operações com os n-1 itens restantes, depois com os n-2 itens, até que reste apenas um elemento.

Exemplo 1:

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 1	***A***	*R*	*D*	*E*	*N*	***O***
i = 2	*A*	***D***	***R***	*E*	*N*	*O*
i = 3	*A*	*D*	***E***	***R***	*N*	*O*
i = 4	*A*	*D*	*E*	***N***	***R***	*O*
i = 5	*A*	*D*	*E*	*N*	***O***	***R***

Análise:

- Complexidade: $O(n^2)$
- Vantagens:
  - Custo linear no tamanho da entrada para o número de movimentos de registros.
  - É o algoritmo a ser utilizado para conjuntos com registros muito grandes.
  - É muito interessante para conjuntos pequenos.
- Desvantagens:
  - fato do conjunto já estar ordenado não ajuda em nada, pois o custo continua quadrático.
  - O algoritmo não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

1.3. Métodos de Ordenação por Permutação ou Troca

Nesses métodos a permutação entre dois elementos é a principal característica do processo.

1.3.1 Bolha (Bubble Sort)

Este método não é o mais eficiente, mas é um dos mais conhecidos devido à sua simplicidade. Tomando-se o conjunto a ser ordenado um vetor com n elementos, o Método da Bolha percorre o vetor comparando os elementos vizinhos entre si. Caso estejam fora de ordem, os mesmos são trocados de posição. Procede-se assim até o final do vetor e este processo é repetido até que nenhuma troca seja realizada.

Na rotina 1, após a primeira varredura o maior elemento do vetor estará na n-ésima posição, já que a ordenação é crescente e o processo de comparação ocorre do inicio para o fim do vetor. Na rotina 2 também após a primeira varredura o maior elemento do vetor estará na n-ésima posição, já que a ordenação é crescente e o processo de comparação se dá do fim para o inicio do vetor.

Análise

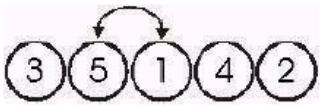
- Complexidade: $O(n^2)$
- É um algoritmo bem simples, porém lento para grande quantidade de dados;
- no entanto, serve como base para algoritmos mais elaborados.

Exemplo 1:

Vetor inicial:

O primeiro passo é fazer a comparação entre os dois elementos das primeiras posições:

Assim, verificamos que neste caso os dois primeiros elementos estão desordenados entre si, logo devemos trocá-los de posição. E assim continuamos com as comparações dos elementos subsequentes:

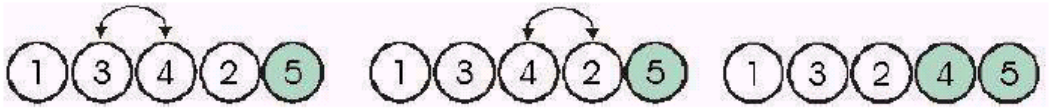


Aqui, mais uma vez, verificamos que os elementos estão desordenados entre si. Devemos fazer a troca e continuar nossas comparações até o final do arranjo:

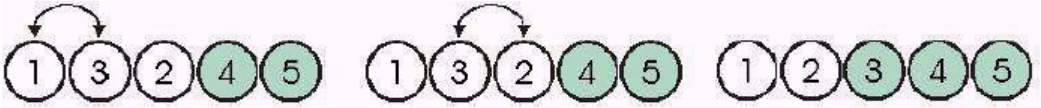
Após este primeiro passo, que compreende a primeira passagem pelo arranjo fazendo as comparações e as trocas necessárias, verificamos que o maior elemento, o número 5, foi parar na última posição, seu lugar correto no arranjo ordenado. Pode-se dizer que o número 5 "borbulhou" para a sua posição correta, lá no final do arranjo.

O próximo passo agora será repetir o processo de comparações e trocas desde o início do arranjo. Só que dessa vez o processo não precisará comparar o penúltimo com o último elemento, pois o último número, o 5, está em sua posição correta no arranjo. Vamos ao processo:

Novamente comparam-se os dois primeiros elementos do arranjo. Neste caso, verificamos que será necessária a troca de lugares entre os elementos. Em seguida vamos continuando as comparações até o final (lembrando que a última posição já está ordenada no arranjo):

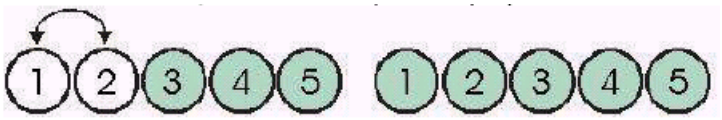


Nestes passos podemos verificar com bastante clareza que o segundo elemento de maior valor "borbulhou" para sua posição correta. Agora precisaremos repetir o processo novamente,



mas desta vez, além de não precisarmos levar em consideração o último elemento do arranjo (no caso o número 5) que já está ordenado, também não precisaremos levar em consideração o penúltimo elemento do arranjo (no caso o número 4) pois ele também está em sua posição correta. Vamos então continuar o processo:

Mais uma vez o elemento de maior valor, o número 3, "borbulhou" para sua posição correta. Basta agora mais um processo para que todo o arranjo fique ordenado. (OBS: Neste caso o arranjo já está ordenado devido as disposições iniciais do arranjo, mas não é possível o algoritmo saber se todo o arranjo está ordenado ou não, e é exatamente por isso que precisaremos realizar mais um processo).



1.3.2 ShellSort (inserção)

Este método, proposto por Shell em 1959, é bem mais eficiente que o Método da Bolha, e consiste em uma extensão do algoritmo de ordenação por inserção. Inicialmente, ele distribui os elementos aproximadamente onde eles ficarão na ordenação final e em seguida, determina seu posicionamento exato. O grande segredo do método está na estimativa da posição final do elemento.

O conceito chave do Método Shell é o intervalo que representa a distância entre os elementos comparados. Por exemplo, ao definir intervalo 3, o primeiro elemento será comparado com o quarto, o segundo elemento com o quinto, o terceiro com o sexto e assim por diante.

Exemplo 1:

Na primeira passada (h=4), o *O* e o *N* (posições 1 e 5) são comparados e trocados; a seguir o *R* e o *A* (posições 2 e 6) são comparados e trocados. Na segunda passada (h=2), *N*, *D* e *O*, nas posições 1, 3 e 5, são re-arranjados para resultar em *D*, *N* e *O* nestas mesmas posições; da mesma forma *A*, *E* e *R*, nas posições 2, 4 e 6, são comparados e mantidos nos seus lugares. A última passada (h=1) corresponde ao algoritmo de inserção: entretanto, nenhum item tem de se mover para posições muito distantes.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

Análise

- A razão da eficiência do algoritmo ainda não é conhecida, pois ninguém ainda foi capaz de analisar o algoritmo.
- A sua análise contém alguns problemas matemáticos muito difíceis, a começar pela própria sequência de incrementos.
- O que se sabe é que cada incremento não deve ser múltiplo do anterior.

- Vantagens:
  - Shellsort é uma ótima opção para conjuntos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.
- Desvantagens:
  - tempo de execução do algoritmo é sensível à ordem inicial do conjunto.

Exemplo 2:

Vetor não ordenado

1	2	3	4	5	6	7	8	9	10
5	8	0	1	7	9	2	6	3	4

Após varredura com intervalo 5:

1	2	3	4	5	6	7	8	9	10
5	2	0	1	4	9	8	6	3	7

Após varredura com intervalo 4:

1	2	3	4	5	6	7	8	9	10
4	2	0	1	3	7	8	6	5	9

Após varredura com intervalo 3:

1	2	3	4	5	6	7	8	9	10
1	2	0	4	3	5	8	6	7	9

Após varredura com intervalo 2:

1	2	3	4	5	6	7	8	9	10
0	2	1	4	3	5	7	6	8	9

Após varredura com intervalo 1:

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Pode-se verificar que após a varredura com intervalo 2 o vetor está praticamente ordenado, assim para intervalo 1, que são as comparações entre elementos adjacentes, o vetor se torna completamente ordenado.

O valor inicial do intervalo é arbitrário, embora seja comum defini-lo como a metade inteira da quantidade de elementos que o conjunto a ser ordenado possui, por exemplo, para um vetor com 13 elementos o intervalo inicial seria 6.

1.4. Métodos de Ordenação por Particionamento

A ordenação por particionamento consiste em particionar os dados de uma lista a ser ordenada em 2 ou mais sub-listas, ordenar cada uma separadamente e depois combiná-las apropriadamente para obter a lista ordenada. Esta forma de ordenação naturalmente leva à algoritmos recursivos.

1.4.1 QuickSort (ordenação rápida)

É o algoritmo de ordenação interna mais rápido que se conhece para uma variedade de situações, sendo provavelmente mais utilizado do que qualquer outro algoritmo. Este método é considerado o melhor algoritmo de ordenação atualmente, sendo baseado na ideia de partições. O algoritmo foi criado por C.A.R. Hoare em 1960, quando visitava a Universidade de Moscou como estudante e foi publicado em 1962, após uma série de refinamentos.

A idéia básica é a de partir o problema de ordenar um conjunto com n itens em dois problemas menores. A seguir, os problemas menores são ordenados independentemente e depois os resultados são combinados para produzir a solução do problema maior – se baseia no paradigma de dividir e conquistar.

A parte mais delicada do método é relativa ao procedimento partição, o qual tem que rearranjar o vetor vet[esq..dir] através da escolha arbitrária de um elemento x do vetor denominado pivô, de tal forma que ao final o vetor vet está particionado em uma parte esquerda com chaves menores ou iguais a x e uma parte direita com chaves maiores ou iguais a x.

O pivô pode ser escolhido de diversas formas:

- aleatoriamente
- o primeiro valor da lista
- o valor médio entre os extremos da lista
- a média de um pequeno conjunto de valores retirado da lista

Este método é essencialmente recursivo, já que adota o mesmo procedimento para as sucessivas partições.

Algoritmo

1. Escolha arbitrariamente um item do vetor e coloque-o em x.
2. Percorra o vetor a partir da esquerda até que um item vet[i] $\geq$ x seja encontrado. Da mesma maneira, percorra o vetor a partir da direita até que um item vet[j] $\leq$ x seja encontrado.
3. Como os dois itens vet[i] e vet[j] estão fora de lugar no vetor final então troque-os de lugar.
4. Continue este processo até que os índices i e j se cruzem em algum ponto do vetor.

Ao final, o vetor [esq .. dir] está particionado de tal forma que:

- Os itens em vet[esq], vet[esq+1], ..., vet[j] são menores ou iguais a x;
- Os itens em vet[i], vet[i+1], ..., vet[dir] são maiores ou iguais a x;

Análise

- Complexidade O(n log n).

- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Vantagens:
  - É extremamente eficiente.
  - Requer cerca de n log n comparações em média para ordenar n itens.
- Desvantagens:
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada e difícil:
  - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.

Exemplo 1: Considere o conjunto apresentado a seguir (processo de partição).

1	2	3	4	5	6
O	*R*	***D***	*E*	*N*	*A*
A	*R*	***D***	*E*	*N*	*O*
A	***D***	*R*	*E*	*N*	*O*

Partição do vetor

O item x é escolhido como vet[(i+j) div 2]. Como inicialmente i = 1 e j = 6, então x = vet[3] = D, destacado em negrito na segunda linha da mesma figura.

A varredura a partir da posição 1 pára no item O e a varredura a partir da posição 6 pára no item A, sendo os dois trocados como mostrado na terceira linha da figura.

A seguir a varredura a partir da posição 2 pára no item R e a varredura a partir da posição 5 pára no item D, e então os dois itens são trocados, como mostrado na Quarta linha. Nesse momento i e j se cruzam (i = 3 e j = 2), o que encerra o processo de partição.

Após obter os dois pedaços do vetor através do procedimento partição, cada pedaço é ordenado recursivamente. O procedimento ordena é recursivo e o vetor vet é global.

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	***D***	*R*	*E*	*N*	*O*
2	***A***	*D*				
3			***E***	*R*	*N*	*O*
4				***N***	*R*	*O*
5					*O*	***R***
	A	*D*	*E*	*N*	*O*	*R*

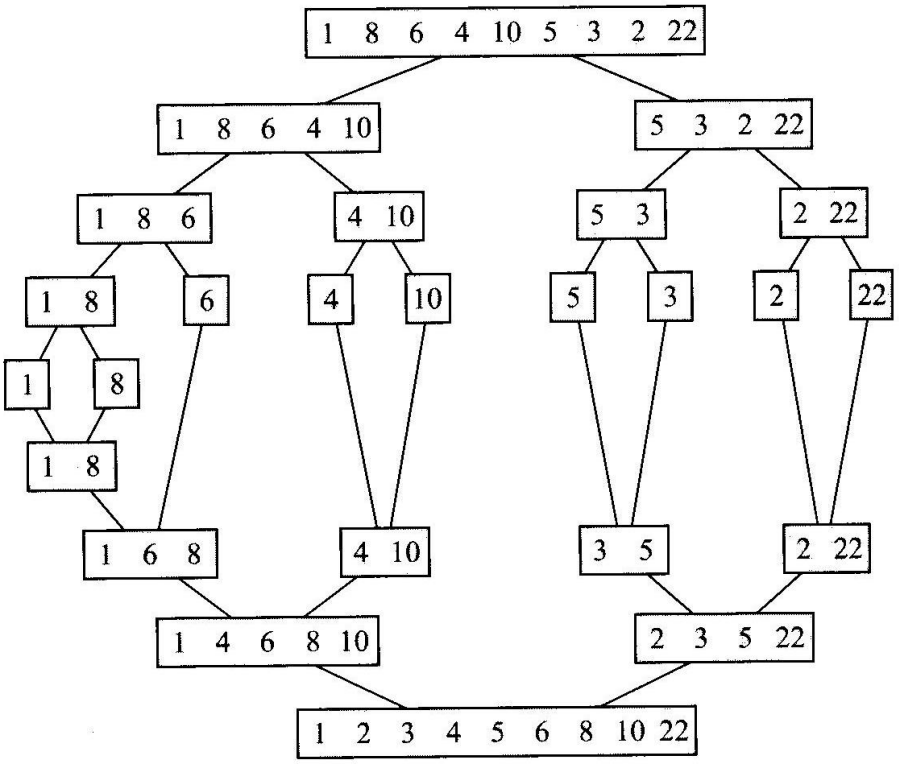
1.4.2 MergeSort

Este algoritmo de ordenação por particionamento, subdivide a lista de dados em 2 sub-listas de igual comprimento, ordena cada sub-lista separadamente e finalmente as intercala apropriadamente para obter a lista ordenada. A ordenação de cada sub-lista é feita de forma idêntica á descrita acima, assim, o algoritmo é recursivo.

O processo de intercalação é chamado de merge derivando do inglês to merge, que significa intercalar. Numa primeira etapa os elementos de 2 sub-listas são comparados e o menor é colocado na lista final ordenada de tal forma que os elementos das 2 sub-listas sejam intercalados de forma ordenada. Esta etapa do processo de merge é conhecida por balance line. Numa segunda etapa se ainda restarem elementos em uma das listas estes serão copiados para o final da lista ordenada.

Já que o processo de merge intercala valores de 2 sub-listas provenientes da lista original a lista ordenada tem que ser criada numa outra variável, assim este algoritmo de ordenação necessita de uma lista auxiliar.

Exemplo 1: considere o conjunto 1, 8, 6, 4, 10, 5, 3, 2, 22



Análise

- Complexidade O(n log n).
- Desvantagem:
  - Necessidade de armazenamento adicional para fundir os conjuntos, o que, para grandes quantidades de dados, poderá ser um obstáculo insuperável (uma solução é utilizar listas encadeadas).

1.4.3 Heapsort (ordenação por monte)

Esse algoritmo possui o mesmo princípio de funcionamento da ordenação por seleção.

Algoritmo:

1. Selecione o menor item do vetor.

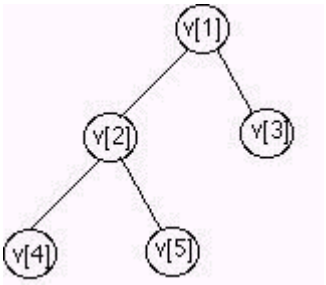
2. Troque-o com o item da primeira posição do vetor.

3. Repita estas duas operações com os n -1 itens restantes, depois com os n-2 itens, e assim sucessivamente.

O custo para encontrar o menor (ou o maior) item entre n itens é n-1 comparações. Isso pode ser reduzido utilizando uma fila de prioridades.

A seleção em árvore é assim chamada por realizar uma distribuição lógica dos valores numa hierarquia de árvore binária. É chamada também de *heapsort* por ser baseada na formação de um *heap* , um seja, uma formação triangular de valores, onde há um menor (ou maior) no topo e dois maiores (ou menores) nas duas outras extremidades.

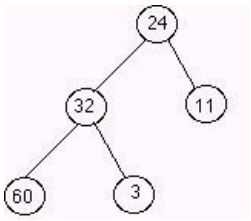
A distribuição em árvore se dá na seguinte ordem: seja um vetor **v** de 5 elementos, a sua árvore seria:



Assim, para o vetor

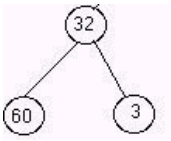
24	32	11	60	3

a árvore seria:

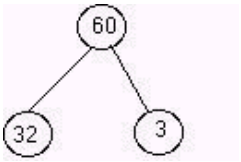


Uma vez formada a árvore, o próximo passo é a formação do *heap* das subárvores.

Inicia-se pelo mais baixo nível à direita. No exemplo, inicia-se pela subárvore:



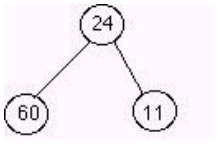
e ordena-se, então, formando o *heap* , colocando o maior valor no topo:



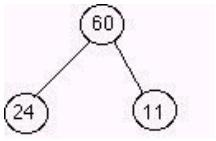
e o vetor fica então com os valores:

24	60	11	32	3

Passa-se à formação do *heap* a outra subárvore:



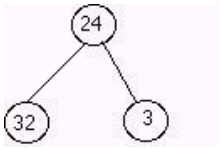
que passa a



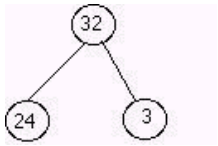
e o vetor fica:

60	24	11	32	3

Formou-se o *heap* na segunda subárvore, mas por outro lado desarranjou a primeira, que ficou:



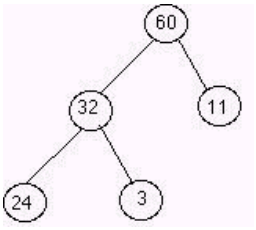
e formando-se o novo *heap* :



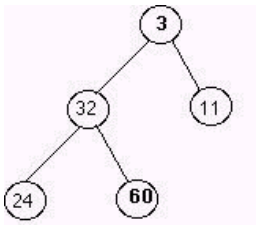
e a ordem no vetor:

60	32	11	24	3

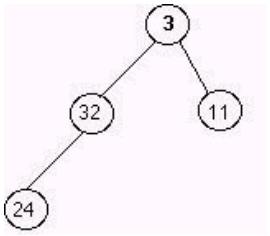
e a árvore completa:



Após ajustadas todas as subárvores, passa-se à troca do valor da raiz com o valor correspondente ao do último elemento do vetor.



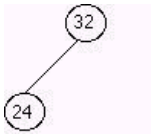
Uma vez feita a troca, elimina-se a folha da árvore:



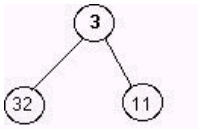
ficando o vetor com a seguinte ordenação:

3	32	11	24	60

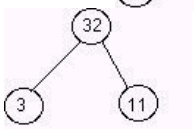
Uma vez retirada a folha, reinicia-se a formação de *heaps* . A primeira subávore não necessita alteração:



mas a segunda...



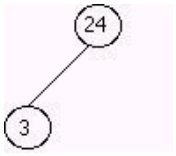
resulta em:



ficando o vetor:

32	3	11	24	60

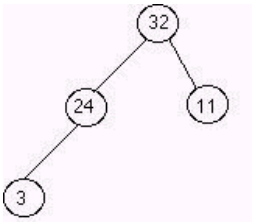
e a nova formação da primeira subárvore, modificada pela segunda:



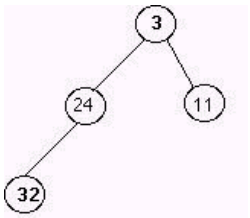
alterando também o vetor:

32	24	11	3	60

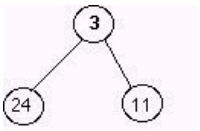
E a árvore completa:



Troca-se, então, o valor raiz com o último elemento, colocando aquele na posição correta...



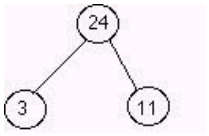
... retirando-o da árvore:



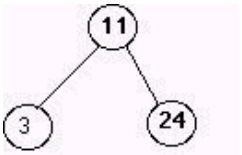
ficando o vetor na forma:

3	24	11	32	60

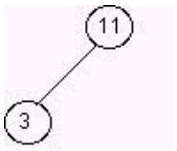
Formamos então o *heap* da árvore resultante...



... e trocamos o valor da raiz...



... e retiramos a última folha...



... para formar o vetor quase ordenado:

11	3	24	32	60

Para concluir, como não há troca, apenas passa-se a raiz para o lugar da folha e temos o vetor ordenado:

3	11	24	32	60

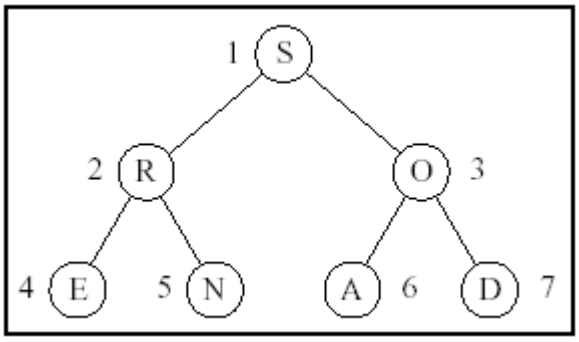
Heaps (ou "montes")

Seqüência de itens com chaves c[1], c[2], ..., c[n], tal que:

c[i] >= c[2i];

c[i] >= c[2i + 1]; para todo i = 1, 2, ..., n/2.

A definição pode ser facilmente visualizada em uma árvore binária completa:



Árvore binária completa

- Os nós são numerados de 1 a n.
- O primeiro nó é chamado raiz.
- O nó [k/2] é o pai do nó k, para 1 < k <= n.
- Os nós 2k e 2k + 1 são os filhos à esquerda e à direita do nó k, para 1 <= k <= [k/2].
- As chaves na árvore satisfazem a condição do *heap*.
- A chave em cada nó é maior do que as chaves em seus filhos.
- A chave no nó raiz é a maior chave do conjunto.
- Uma árvore binária completa pode ser representada por um array:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*

- A representação é extremamente compacta.
- Permite caminhar pelos nós da árvore facilmente.
- Os filhos de um nó i estão nas posições 2i e 2i + 1.
- pai de um nó i está na posição i div 2.
- Na representação do *heap* em um arranjo, a maior chave está sempre na posição 1 do vetor.

- Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.
- Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.
- O algoritmo não necessita de nenhuma memória auxiliar.

Algoritmo

1. Construir o *heap*.
2. Troque o item na posição 1 do vetor (raiz do *heap*) com o item da posição n.
3. Use o procedimento Refaz para reconstituir o *heap* para os itens vet[1], vet[2], ... vet[n-1].
4. Repita os passos 2 e 3 com os n-1 itens restantes, depois com os n-2, até que reste apenas um item.

Exemplo 1:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*
R	***N***	*O*	*E*	***D***	*A*	***S***
O	*N*	***A***	*E*	*D*	*R*	
N	***E***	*A*	***D***	*O*		
E	***D***	*A*	*N*			
D	***A***	*E*				
A	*D*					

Análise

- Heapsort gasta um tempo de execução proporcional a n log n, no pior caso.
- Vantagens:

  O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- Desvantagens:

  O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
- Recomendado:

  Para aplicações que não podem tolerar eventualmente um caso desfavorável.

  Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.

1.5 Comparação entre Alguns Métodos

A ordenação interna é utilizada quando todos os registros do arquivo cabem na memória principal. Alguns métodos de ordenação interna mediante comparação de chaves foram apresentamos. Estudamos dois métodos simples (Seleção e Inserção) que requerem $O(n^2)$ comparações e três métodos eficientes (Shellsort, Quicksort e Heapsort) que requerem O (n log n) comparações.

As tabelas 1, 2 e 3 abaixo apresentam quadros comparativos do tempo total real para ordenar arranjos com 500, 5.000, 10.000 e 30.000 registros na ordem aleatória, na ordem ascendente e na ordem descendente, respectivamente. Em cada tabela, o método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele. Assim, na tabela 1, o Shellsort levou o dobro do tempo do Quicksort para ordenar 30.000 registros.

	500	**5.000**	**10.000**	**30.000**
Inserção	11,3	87	161	-
Seleção	16,2	124	228	-
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

Tabela 1 – Ordem aleatória dos registros.

	500	**5.000**	**10.000**	**30.000**
Inserção	1	1	1	1
Seleção	128	1.524	3.066	-
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

Tabela 2 – Ordem ascendente dos registros.

	500	**5.000**	**10.000**	**30.000**
Inserção	40,3	305	575	-
Seleção	29,3	221	417	-
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

Tabela 3 – Ordem descendente dos registros.

Observações sobre os métodos:

1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza.
2. Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta.
4. A relação Heapsort/Quicksort se mantém constante para todos os tamanhos.
5. Para conjuntos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort.
6. Quando o tamanho da entrada cresce, o Heapsort é mais rápido que o Shellsort.
7. Inserção é o mais rápido para qualquer tamanho se os elementos estão ordenados.
8. Inserção é o mais lento para qualquer tamanho se os elementos estão em ordem descendente.
9. Entre os algoritmos de custo $O(n^2)$, o Inserção é melhor para todos os tamanhos aleatórios experimentados.
10. Shellsort é bastante sensível à ordenação ascendente ou descendente da entrada.
11. Em conjuntos do mesmo tamanho, o Shellsort executa mais rápido para conjuntos ordenados.
12. Quicksort é sensível à ordenação ascendente ou descendente da entrada.
13. Em conjuntos do mesmo tamanho, o Quicksort executa mais rápido para conjuntos ordenados.
14. Quicksort é o mais rápido para qualquer tamanho para conjuntos na ordem ascendente.
15. Heapsort praticamente não é sensível à ordenação da entrada.

Método da Inserção:

- É o mais interessante para conjuntos com menos do que 20 elementos.
- Possui comportamento melhor do que o método da **bolha (Bubblesort)**
- Sua implementação é tão simples quanto às implementações do Bubblesort e Seleção.
- Para conjuntos já ordenados, o método é O(n).
- O custo é linear para adicionar alguns elementos a um conjunto já ordenado.

Método da Seleção:

- É vantajoso quanto ao número de movimentos de registros, que é O(n).
- Deve ser usado para conjuntos com registros muito grandes, desde que o tamanho do conjunto não exceda 1.000 elementos.

Shellsort:

- É o método a ser escolhido para a maioria das aplicações por ser muito eficiente para conjuntos de tamanho moderado.
- Mesmo para conjuntos grandes, o método é cerca de apenas duas vezes mais lento do que o Quicksort.
- Sua implementação é simples e geralmente resulta em um programa pequeno.
- Não possui um pior caso ruim e quando encontra um conjunto parcialmente ordenado trabalha menos.

Quicksort:

- É o algoritmo mais eficiente que existe para uma grande variedade de situações.
- É um método bastante frágil no sentido de que qualquer erro de implementação pode ser difícil de ser detectado.
- O algoritmo é recursivo, o que demanda uma pequena quantidade de memória adicional.
- Seu desempenho é da ordem de $O(n^2)$ operações no pior caso.
- O principal cuidado a ser tomado é com relação à escolha do pivô.
- A escolha do elemento do meio do arranjo melhora muito o desempenho quando o conjunto está total ou parcialmente ordenado.
- O pior caso tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios.
- Geralmente se usa a mediana de uma amostra de três elementos para evitar o pior caso.
- Esta solução melhora o caso médio ligeiramente.
- Outra importante melhoria para o desempenho do Quicksort é evitar chamadas recursivas para pequenos subconjuntos.
- Para isto, basta chamar um método de ordenação simples nos conjuntos pequenos.
- A melhoria no desempenho é significativa, podendo chegar a 20% para a maioria das aplicações

Heapsort:

- É um método de ordenação elegante e eficiente. Apesar de ser cerca de duas vezes mais lento do que o Quicksort, não necessita de nenhuma memória adicional.
- Executa sempre em tempo proporcional a n log n.
    - Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o Heapsort.