Métodos de classificação
Prof. João B. Oliveira

Métodos de classificação

Desde seus primeiros estudos, os métodos de classificação de dados foram divididos em duas grandes categorias:

- Classificação interna

  São métodos nos quais todo o trabalho de classificação é feito em memória principal, ou seja, não são usadas áreas auxiliares como disco ou fita magnética.

- Classificação externa

  Nestes métodos, admite-se que seja usada memória externa, ou seja, áreas auxiliares para dados temporários, tipicamente em disco ou fita.

Nas décadas de 50 e 60 muita ênfase era dada aos métodos de classificação externa, já que as máquinas da época tinham uma memória muito limitada (tipicamente cerca de 16 kb) e portanto o uso de áreas auxiliares era essencial para trabalhar com qualquer quantidade razoável de dados. Com o passar do tempo, as máquinas receberam memória cada vez mais rápida e barata, e os métodos de classificação interna se tornaram cada vez mais importantes. Hoje, os métodos de classificação externa são considerados bem menos importantes, embora ainda sejam usados por qualquer bom programa de ordenação em situações especiais. Por exemplo, o programa `sort` usado no sistema operacional UNIX e suas variantes faz a ordenação de dados em memória interna, mas se a quantidade de dados excede a memória da máquina usada, ele automaticamente usa arquivos em disco para fazer parte da ordenação.

Classificação por inserção[1]

Nestes métodos, o princípio básico é sempre o mesmo: considerar o vetor como uma série de dados desorganizados, retirar um elemento por vez e colocá-lo no lugar adequado em uma parte já organizada, inserindo-o em sua devida posição. Desta forma os métodos não se preocupam com qual elemento estão retirando, apenas tomam cuidado no momento de recolocá-lo no vetor para que este entre na posição adequada.

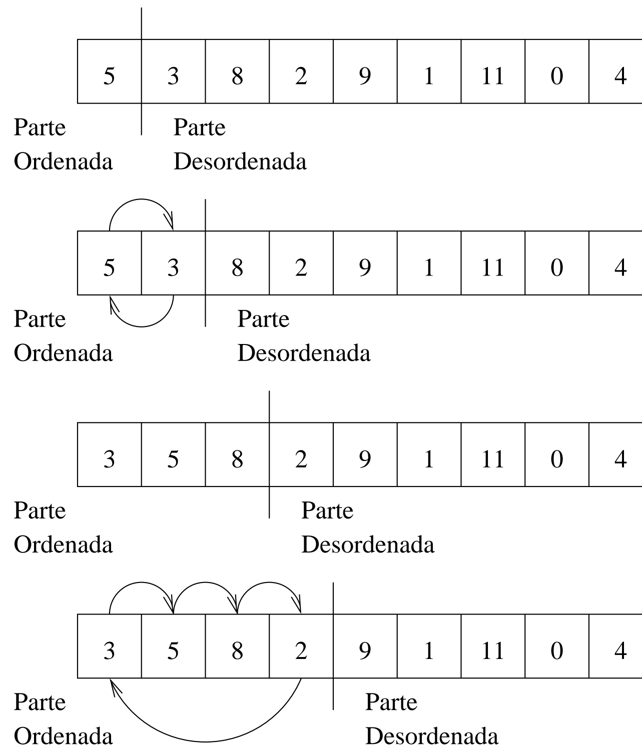
Insertion sort

A classificação por inserção é um método extremamente simples de fazer a ordenação de um vetor, com baixa eficiência mas que pode ser usado para vetores de tamanho pequeno por sua simplicidade de programação. Esquematicamente, o algoritmo é o seguinte:

- O vetor é dividido em uma parte ordenada e uma parte desordenada. No início, somente o primeiro elemento do vetor compõe a parte desordenada.
- Um a um os elementos da parte desordenada são colocados na posição apropriada na parte ordenada, jogando-se elementos mais para a frente de acordo com a necessidade de reposicionamento.
- O algoritmo termina quando não há mais elementos na parte desordenada.

[1] Atenção: apesar dos programas apresentados estarem escritos em C, assume-se que os vetores tem seus índices começando em 1 e indo até o número de elementos do vetor, ou seja, na faixa 1...n, e não na faixa 0...n-1. Isto foi adotado para deixar o código mais claro.

A parte mais custosa deste algoritmo é abrir espaço na parte ordenada para um novo elemento, uma vez que o elemento a ser inserido pode entrar em uma posição qualquer dentro da parte ordenada, podendo deslocar **todos** os outros elementos e fazendo-os avançar um posição. Confira na ilustração abaixo:

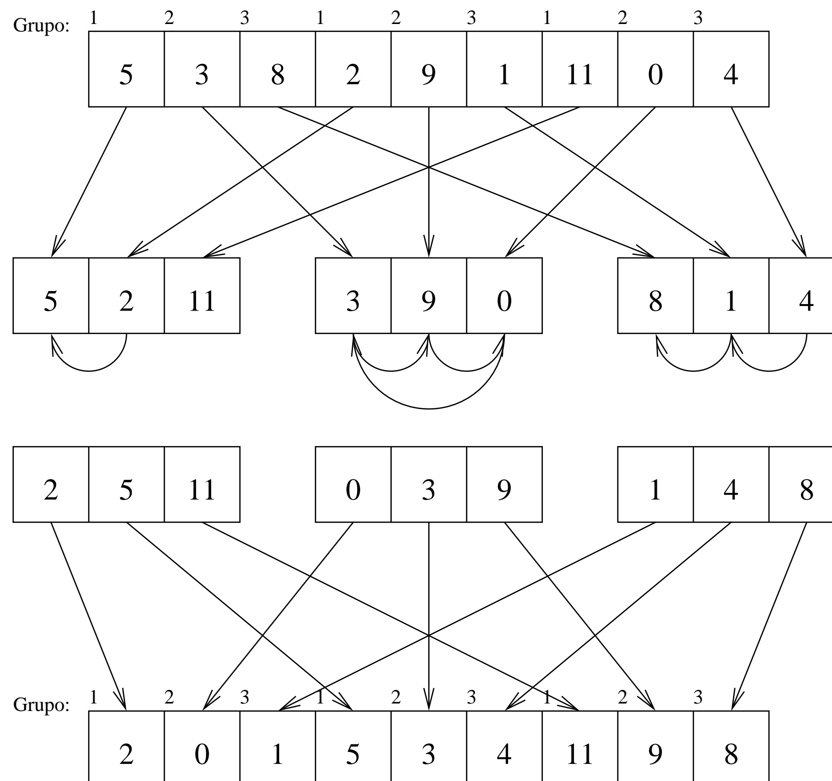


Uma possível implementação para o *insertion sort* segue abaixo.

```
void    insertion_sort (vetor v, int tam) {
  int     i, j, k, tmp;
  bool    achei;

  for (i = 2; i <= tam; i++) {
    j = 1;
    achei = FALSE;
    while ((j < i) && (!achei)) {
      if (v[i] < v[j])
        achei = TRUE;
      else
        j++;
    }
    if (achei == TRUE) {
      tmp = v[i];
      k = i - 1;
      while (k >= j) {
        v[k + 1] = v[k];
        k--;
      }
      v[j] = tmp;
    }
  }
}
```

Shellsort

Como o problema básico do *insertion sort* é a série longa de movimentações de elementos, com muitos deslocamentos por uma distância curta (sempre apenas uma posição), em 1959 Donald L. Shell teve a idéia de aplicar este algoritmo simples a pedaços *diferentes* do vetor, de forma que as trocas não mais fossem de curta distância, fazendo com que os elementos se movessem em grandes saltos pelo vetor. O algoritmo funciona de forma engenhosa pois os pedaços diferentes do vetor não são contíguos, mas sim intercalados. Esta é, na verdade, uma condição essencial para que o *shellsort* funcione bem. A figura abaixo mostra o vetor original dividido em três pedaços e como o *shellsort* opera sobre estes pedaços.



Através da intercalação, movimentos aparentemente curtos nos vetores menores representam na verdade uma distância bem maior quando vistos no vetor original. Note-se que o processo de ordenação dos vetores menores pode ser o próprio *insertion sort* visto anteriormente. Deve-se dizer ainda que em momento algum os vetores menores existem realmente, ou seja, o algoritmo opera sempre sobre o vetor principal, sabendo localizar cada elemento dos vetores menores.

Uma característica fundamental do *shellsort* é que ao final de um passo (um passo = uma série de divisões sobre o vetor, como mostrado na figura) o vetor **não** está necessariamente ordenado, mas seus elementos estão em geral melhor posicionados. Sendo assim, os passos são repetidos dividindo-se o vetor em um número cada vez menor de pedaços. Para que o processo de ordenação termine com sucesso, é fundamental que no final seja executado um passo com apenas **um** vetor, ou seja, sobre o vetor original. É este último passo que finalmente coloca os elementos em seu lugar e garante o sucesso da ordenação. Este último passo corresponde a um *insertion sort* normal, mas como os elementos já foram movidos por longas distâncias nos passos anteriores, o trabalho de movomentação é agora muito reduzido.

O código do *shellsort* está a seguir:

```
/* Rotina que efetua cada passo de ordenação nos subvetores.
Note como ela está parametrizada! */
void    passo_shell (vetor v, int r, int s, int n) {
       /* Pergunta: qual o papel de r, s, e n? */
  int     i, j, k, tmp;
  bool    achei;

  for (i = s + r; i <= n; i += r) {
    j = s;
    achei = FALSE;
    while ((j < i) && (!achei)) {
      if (v[i] < v[j])
        achei = TRUE;
      else
        j += r;
    }
    if (achei == TRUE) {
      tmp = v[i];
      k = i - r;
      while (k >= j) {
        v[k + r] = v[k];
        k -= r;
      }
      v[j] = tmp;
    }
  }
}

/* Rotina que faz a gerência dos passos do shellsort */
void    shellsort (vetor v, int tam) {
  int     np, i, j, inc;  /* np = numero de passos */

  np = pergunta_num_passos ();

  for (i = np; i >= 0; i--) {
         /* O cálculo não será mostrado */
    inc = 1 << i;
    for (j = 1; j <= inc; j++)
      passo_shell (v, inc, j, tam);
  }
}
```

Classificação por troca

Nestes métodos, o princípio básico é o seguinte: pares de elementos são sempre examinados, e caso o par de elementos esteja na ordem inversa à desejada, os elementos são trocados de posição. Enquanto o *bubblesort* é bastante simples e troca de posição elementos que são vizinhos, o *quicksort* tenta trocar de posição elementos que estão afastados, procurando ganhar na distância a cada movimento.

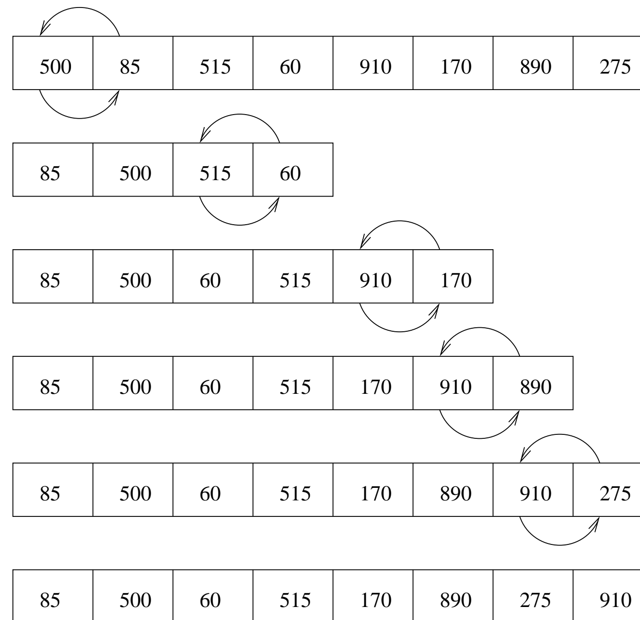
Bubblesort

O *bubblesort* (também chamado de método da bolha) é um método bastante primitivo de ordenar um vetor, mas que pode ser usado para vetores pequenos, já que sua programação é bastante rápida e simples. No entanto, à medida que o número de elementos do vetor cresce o desempenho do *bubblesort* cai dramaticamente. Em termos gerais, o algoritmo é o seguinte:

- A cada passo, cada elemento é comparado com o próximo. Se eles estiverem fora de ordem, são trocados de posição.

- Realizam-se tantos destes passos até que não sejam feitas mais trocas.

A partir desta descrição, pode-se notar que as trocas são sempre feitas entre elementos adjacentes, o que prejudica o desempenho do algoritmo. Também não se sabe de antemão quantos pasos serão necessários para colocar o vetor em ordem. A figura abaixo mostra um vetor e como um passo do *bubblesort* opera.



Depois deste passo, pode-se perceber que o maior dos elementos do vetor foi para a última posição, enquanto os outros podem estar ainda fora de ordem. Um segundo passo colocará o segundo maior elemento na penúltima posição, e assim por diante. Por esta lógica, seriam precisos $n-1$ passos (onde n é o número de elementos do vetor) para garantir a ordenação, mas os algoritmos mantém um controle de quando foram feitas trocas, de forma que se em um passo não houveram trocas garante-se que o vetor está ordenado.

O código do *bubblesort* está a seguir:

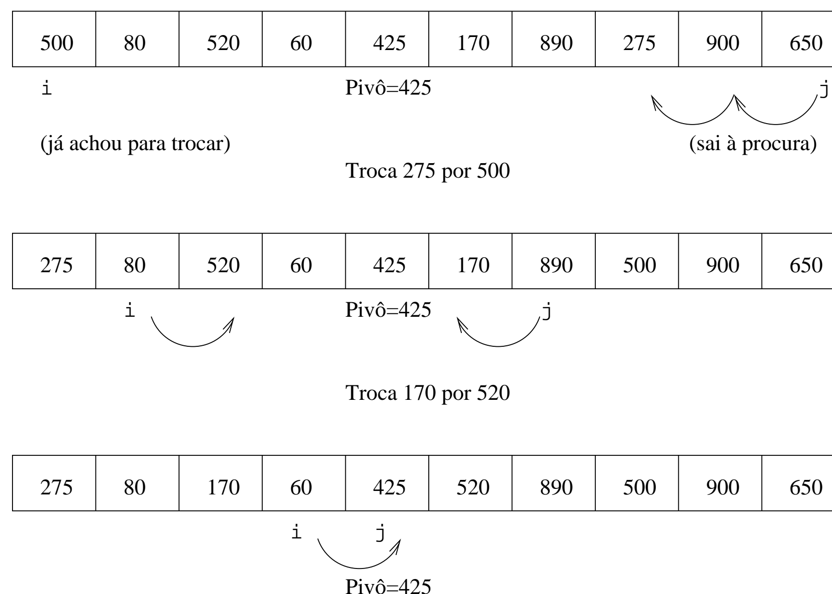
```
/* Rotina básica para bubblesort */
void    bubblesort (vetor v, int tam) {
  int     i, lim, k, tmp;
  bool    troca = TRUE;

  lim = tam - 1;
  while (troca == TRUE) {
    troca = FALSE;
    for (i = 1; i <= lim; i++) {
      if (v[i] > v[i + 1]) {
        tmp = v[i];
        v[i] = v[i + 1];
        v[i + 1] = tmp;
        troca = TRUE;
        k = i;
      }
    }
    lim = k;
  }
}
```

Quicksort

Baseando-se no princípio de trocar elementos de lugar, o conhecido cientista C. A. R. Hoare criou em 1962 um dos melhores métodos já elaborados para ordenação, o *quicksort.* Este método também troca pares de elementos, mas supera o principal problema do *bubblesort*, que é a distância curta entre as movimentações. Dentro do quicksort, elementos podem se mover distâncias muito longas com facilidade. Para implementar esta idéia, Hoare imaginou um algoritmo recursivo baseado no seguinte princípio:

- Escolhe-se arbitrariamente um elemento do vetor (geralmente um elemento no meio do vetor). Este elemento será chamado de pivô. Isto separa o vetor em três partes: à esquerda do pivô, o pivô e sua direita.)
- O algoritmo começa a separar os elementos, tentando fazer com que todos os elementos menores do que o pivô vão para a esquerda e todos os maiores vão para a direita. Isto acontece de uma forma bastante engenhosa, usando-se dois contadores `i` e `j`:
  - Com o contador `i` partindo de 1 e sendo incrementado, o algoritmo vem examinando o lado esquerdo *de fora para o centro*, procurando um elemento maior do que o pivô (para passà-lo para o outro lado.)
  - Com o contador `j` igual ao número de elementos do vetor e sendo decrementado, o algoritmo examina o lado direito *também de fora para o centro*, procurando um elemento menor do que o pivô.
- Uma vez que dois elementos são achados para a troca de lado, eles são trocados entre si.
- O algoritmo continua examinando os elementos, incrementando `i` e decrementando `j` do ponto de onde havia parado, procurando novas trocas.
- Quando `i` e `j` se encontrarem, o processo se repete recursivamente para as duas metades (esquerda de `i` e `j`, direita de `i` e `j`.)

É bem mais simples entender o algoritmo com uma ilustração de seu funcionamento:



Nesta situação os dois se encontram e o processo é repetido para as duas metades.

No exemplo acima está mostrado como dois contadores verificam os elementos de cada lado do pivô para encontrar dois elementos que precisem trocar de lugar. Isto ocorre de fora para

dentro até que em algum momento os dois contadores se encontram. Nesta situação, temos de um lado elementos menores do que o pivô e do outro elementos maiores ou iguais a ele.

É fundamental perceber a importância de três detalhes:

1. Quando os dois contadores se encontram os elementos já foram separados em dois grupos (maiores e menores que o pivô, mais este), e cada grupo pode ser ordenado separadamente.

2. Iniciar a procura de fora para dentro garante que os pares de elementos achados serão movidos uma grande distância em um só movimento.

3. O valor do pivô é armazenado em uma variável *separada*, pois o elemento que foi escolhido como o pivô inicial pode ser movido também.

Na repetição do processo para os dois lados, é escolhido um novo pivô para cada metade e o algoritmo continua como mostrado. Traduzindo para código, teríamos algo como a seguir:

```
/* Rotina que efetua a ordenação nos subvetores. */
void    particao (vetor v, int esq, int dir, int *i, int *j) {

  int     pivo = v[(esq + dir) / 2];  /* Aproximadamente no meio... */

  do {
    while (pivo > v[esq]) esq++;  /* Procura na esquerda... */
    while (pivo < v[dir]) dir--;  /* Procura na direita... */
    if (esq <= dir) {
           /* Se achou um par faz a troca e avanca i e j */
      int tmp = v[esq];
      v[esq] = v[dir];
      v[dir] = tmp;
      esq++;
      dir--;
    }
  }
  while (esq < dir);
  *i = esq;
  *j = dir;
}

/* Rotina que faz a gerência dos passos do quicksort */
void    quicksort (vetor v, int esq, int dir) {
  int     i, j;

  particao (v, esq, dir, &i, &j);
  if (esq < j) quicksort (v, esq, j);
  if (dir > i) quicksort (v, i, dir);
}
```

Desta forma, a partir do exemplo mostrado no código, a chamada inicial para o *quicksort* é feita como

```
quicksort(vetor, 1, tam);
```

Classificação por seleção

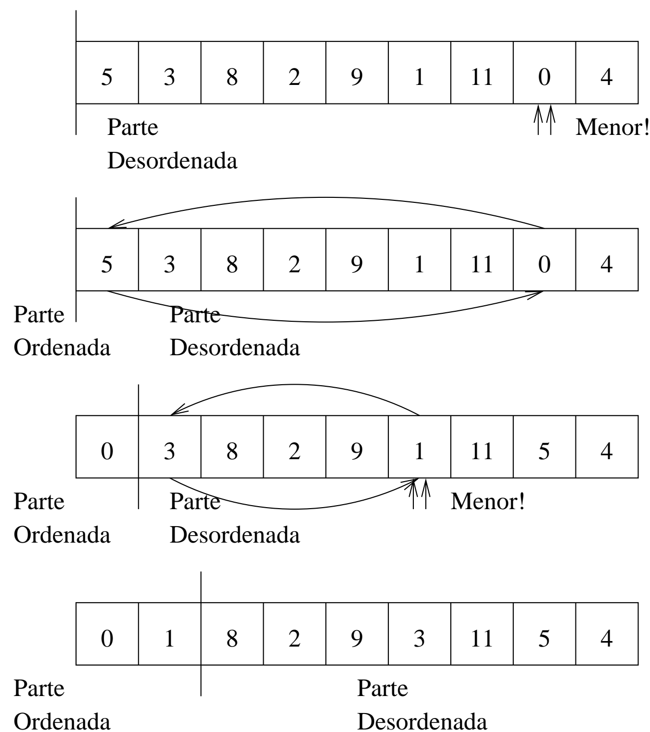
Nestes métodos, o princípio básico é o seguinte: considerar o vetor como uma série de dados desorganizados, retirar a cada vez o menor elemento desta parte desorganizada e colocá-lo na última posição da parte organizada. Pode-se dizer que é um princípio de funcionamento totalmente oposto ao dos métodos de inserção direta, que não escolhem o elemento a buscar e o inserem na posição correta. Agora, busca-se um determinado elemento (o menor) pa rainseri-lo facilmente na última posição da parte organizada.

Seleção direta

Esta é uma implementação imediata do princípio descrito acima:

- A cada passo encontra-se o menor elemento da parte desorganizada
- Troca-se este elemento com o primeiro elemento da parte desorganizada
- Atualiza-se o tamanho do segmento desorganizado (menos um elemento)
- O processo é repetido até que a parte desorganizada fique com apenas um elemento.

A figura abaixo mostra um vetor e como um passo da *seleção direta* opera.



Traduzindo para código, teríamos algo como a seguir:

```
/* Rotina básica para selection sort */
void    selectionsort (vetor * v, int tam) {
  int     i, pos_menor, tmp, j;

  for (i = 1; i <= tam - 1; i++) {
    pos_menor = i;
    for (j = i + 1; j <= tam; j++)
      if (v[j] < v[pos_menor])
        pos_menor = j;
    tmp = v[i];
    v[i] = v[pos_menor];
    v[pos_menor] = tmp;
  }
}
```

Heapsort

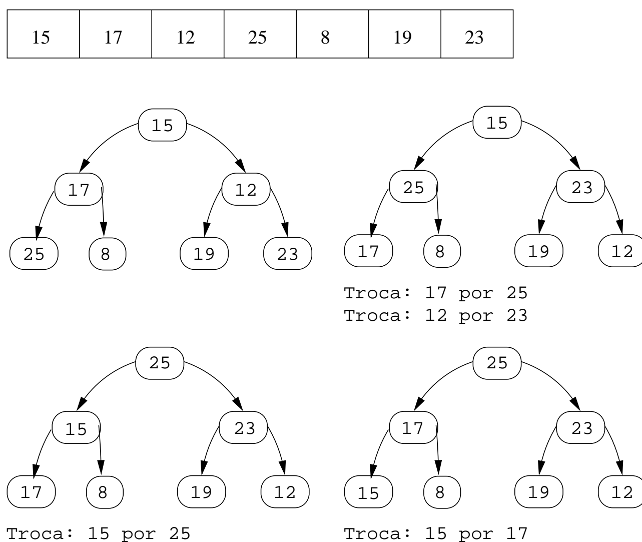
O *heapsort* (ou seleção em árvore) é um dos melhores métodos de ordenação, baseando-se em uma idéia completamente diferente das usadas nos métodos anteriores. Enquanto os métodos vistos até agora sempre operam sobre o vetor como uma lista linear de elementos, o *heapsort* funciona como se os elementos estivessem em uma **árvore binária**, mas isso acontece **sem** que uma árvore ou qualquer outra estrutura auxiliar seja criada. Essencialmente, o algoritmo usa um sistema bastante simples de endereçamento:

- O nodo raiz da árvore é o elemento da posição 1 no vetor.
- Para um dado nodo na posição i do vetor, temos a seguinte situação:
  - Seu filho esquerdo está na posição $2 * i$.
  - Seu filho direito está na posição $2 * i + 1$.

  Note que para qualquer nodo i (exceto a raiz, naturalmente), sempre podemos saber quem é seu pai, pois este estará na posição $i \div 2$ (esta é uma divisão inteira!).

Baseado neste esquema de endereçamento, o algoritmo consiste de duas fases que se repetem várias vezes:

Formação do heap: um heap é um tipo especial de árvore binária onde cada nodo pai tem um valor maior do que seus dois filhos. Uma vez que todos os nodos pais tem valores maiores do que seus filhos, obrigatoriamente o maior valor existente estará na raiz.
A transformação da árvore em heap é feita do maior nível até a raiz, trocando-se cada nodo com o maior de seus filhos e repetindo-se este processo até quando seja necessário. Veja na figura abaixo um exemplo da transformação de uma árvore comum em um heap:

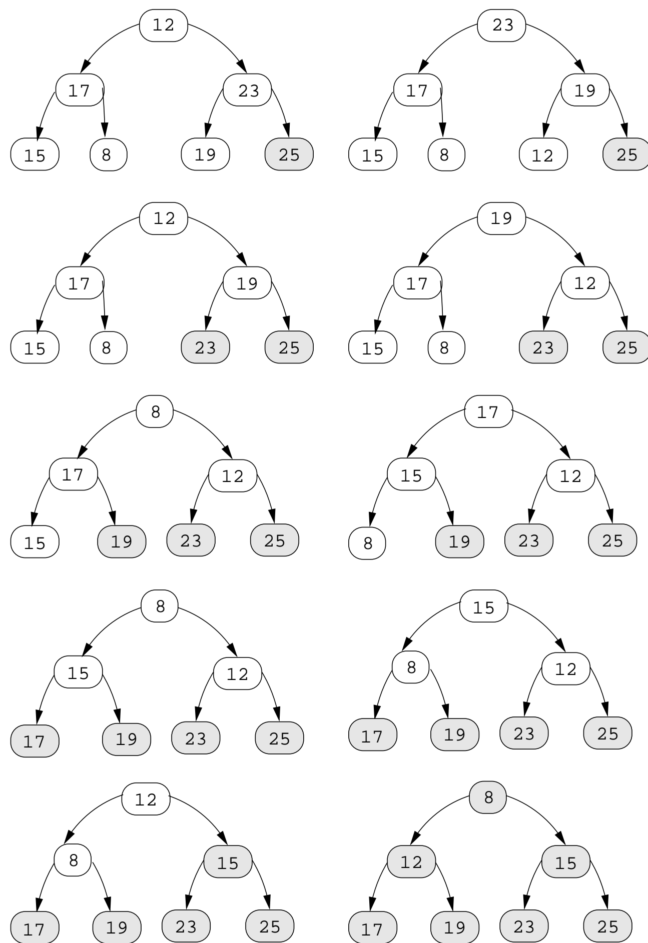


É importante perceber que o processo de transformação pode visitar nodos mais de uma vez, pois uma alteração pode forçar o algoritmo a rever partes já visitadas da árvore. Neste exemplo, a troca de 15 por 25 no terceiro passo implica em trocar também 15 por 17, caso contrário não teríamos um heap. Também é importante notar que a montagem da árvore **nunca** ocorreu, os elementos foram sempre procurados através de seus índices sobre o vetor original, o que possibilita uma grande velocidade de acesso.

Classificação dos elementos no heap: esta fase é bastante simples, e consiste basicamente em trocar o elemento que está na raiz do heap pelo elemento em último lugar no heap. No exemplo acima, o nodo com valor 25 seria trocado pelo nodo 12. Ao fazer-se isto, temos a seguinte situação:

1. Verificando no vetor, o elemento 25 terá ido para a sua posição correta (o último lugar, já que ele é o maior de todos).
2. Uma vez que o último elemento está no lugar certo, podemos continuar o algoritmo supondo que o vetor ficou menor (em um elemento), e procurando agora o penúltimo elemento.
3. O elemento que foi para a raiz pode ter deixado a árvore sem ser um heap (isto acontece se ele for menor do que um dos filhos). Neste caso, deve-se voltar à primeira fase e montar o heap outra vez.

Continuando com o algoritmo e repetindo os passos acima, teríamos a seguinte série de transformações (as operações para formar o heap não são mostradas aqui):



À medida que os maiores elementos da parte restante são encontrados, o vetor ordenado vai sendo montado de trás para a frente. Em nosso exemplo, os elementos marcados em cinza não são mais alterados. Por último, o menor elemento é encontrado e posto na raiz da árvore, ou seja, na posição inicial do vetor.

Em uma primeira análise o algoritmo pode parecer complicado, mas a implementação é bastante simples:

```
/* Rotinas para heapsort */

int     parent (int i) { return i / 2; }
int     left (int i)   { return 2 * i; }
int     right (int i)  { return 2 * i + 1; }

void    heapify (vetor v, int tam, int i) {
  int     l, r, largest, tmp;

  l = left (i);
  r = right (i);

  if ((l <= tam) && (v[l] > v[i]))
    largest = l;
  else
    largest = i;

  if ((r <= tam) && (v[r] > v[largest]))
    largest = r;
  if (largest != i) {
    tmp = v[largest];
    v[largest] = v[i];
    v[i] = tmp;
    heapify (v, tam, largest);
  }
}

void    build_heap (vetor v, int tam) {
  int     i;

  for (i = tam / 2; i; i--)
    heapify (v, tam, i);
}

void    heapsort (vetor v, int tam) {
  int     i, tmp;

  build_heap (v, tam);
  for (i = tam; i > 1; i--) {
    tmp = v[1];
    v[1] = v[i];
    v[i] = tmp;
    heapify (v, --tam, 1);
  }
}
```

Perguntas

1. Qual o pior caso possível para o *insertion sort*? Qual o melhor?
2. Sugira uma forma de calcular o número de passos adequado para ser usado no *shellsort*, e justifique sua resposta.
3. O *bubblesort* precisa de quantas passagens para ordenar um vetor?
4. Para o *bubblesort*, estime o número de vezes em que o algoritmo compara pares de valores do vetor.
5. Implemente o quicksort e faça-o imprimir mensagens, indicando que trechos do vetor estão sendo ordenados de cada vez.

6. Compare *selection sort* e *insertion sort* e diga qual deles é mais rápido (ou eficiente), e porquê.

7. Explique porque o *heapsort* prefere operar sobre o vetor como se este fosse uma árvore, enquanto todos os outros métodos operam sobre vetores ou sub-vetores.

8. O *heapsort* leva o maior elemento do vetor até o topo do heap, e depois o move para o final do vetor. Explique porque não é possível mover o menor elemento do vetor até o topo e depois deixá-lo lá, uma vez que ele está no lugar certo.

9. Você deseja saber qual o elemento de um vetor que estaria na n-ésima posição se o vetor fosse ordenado, mas você não quer (ou não pode) ter o trabalho de fazer a ordenação. Mostre como podemos usar um algoritmo baseado no Quicksort para determinar o elemento correto, sem ordenar o vetor.

10. Dado um vetor com números inteiros, escreva um algoritmo para verificar se há números repetidos e emitir um aviso. Faça isso sob as seguintes condições:

    (a) Você sabe que o vetor está ordenado.

    (b) Você não sabe nada sobre a ordenação do vetor.

    (c) Avalie o desempenho dos dois algoritmos acima, em notação $O()$.

    (d) Avalie a seguinte proposta: passar pelo vetor, inserindo todos os elementos em uma árvore balanceada e verificando se há repetição no momento de inserir. Como fica o desempenho em notação $O()$?

11. O algoritmo abaixo é um algoritmo de classificação. Determine se ele é estável ou instável:

```
função sort(array)
   list less, equal, greater;
   if (length(array) <= 1) return array;
   escolha um elemento XX do array;
   para cada x no array
     if x < XX then append x em less
     if x = XX then append x em equal
     if x > XX then append x em greater
   retorne concat(sort(less), equal, sort(greater));
```

12. Explique o que fazer se desejamos imitar o Heapsort e mapear árvores sobre um vetor, para o caso de árvores ternárias, quaternárias e 5-árias. Como os filhos de cada nodo devem ser endereçados?