2. ALGORITMOS DE ORDENAÇÃO INTERNA

São algoritmos que trabalham sobre registros de uma tabela organizada em forma de um vetor. Entretanto, apenas uma parte do registro, chamada chave de classificação, é utilizada para controlar a ordenação.

2.1 Ordenação por Seleção

Os métodos que formam a família de classificação por seleção caracterizam-se por procurarem, a cada repetição, a chave de menor (ou maior) valor do vetor e colocá-la na sua posição definitiva, por permutação com a que ocupa aquela posição. O vetor a ser classificado fica reduzido de um elemento.

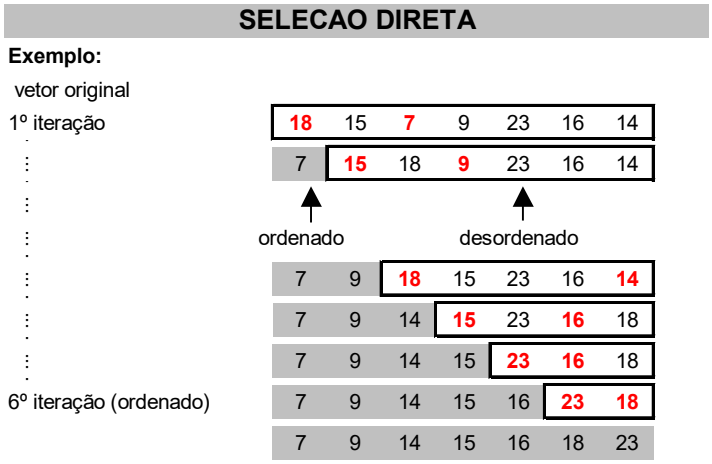
O mesmo processo é repetido até que fique reduzido a um único elemento, quando então a classificação estará concluída.

Os métodos de classificação por seleção diferenciam-se pelo critério utilizado para selecionar, a cada repetição, o elemento de menor (ou maior) valor.

2.1.1 Método Seleção Direta

➢ Princípios de Funcionamento

- Selecionar o item de menor chave e trocá-lo com o item que está na primeira posição do arquivo;
- Repetir este procedimento para os (n-1) elementos restantes, depois para os (n-2) elementos restantes, e assim sucessivamente, até se obter o arquivo ordenado.



➢ Analise do Caso:

Este método não é influenciado pela ordem inicial dos elementos do vetor. O pior caso ocorre nas comparações dos elementos de vetor muito grande.

➢ Considerações:

Deve-se observar que este método não é estável e que o fato de uma entrada estar quase ordenada não favorece a solução.

➢ Implementação do Algoritmo de Seleção Direta em pseudocódigo:

```
constante:
                TAMANHO = 15
variáveis:
        vetor[1...TAMANHO]: inteiro
        i, j, trab, menor: inteiro

início
        para i ← 1 até TAMANHO faça
                leia (vetor[i])
        fim para
        para i ← 1 até TAMANHO-1  faça
                menor ← i
                para j ← i + 1 até  TAMANHO faça
                        se vetor[j] < vetor[menor] então
                                menor ← j
                        fim se
                fim para
                trab ← vetor[i]
                vetor[i] ← vetor[menor]
                vetor[menor] ← trab
        fim para
fim
```

➢ Implementação do Algoritmo de Seleção Direta em Java:

```
public void seleçãoDireta ( ){
      int i, j, minimo;
      Item temp;
      for (i=0; i<this.nElem-1;i++){
            minimo = i;
            for (j = i+1; j < this.nElem; j++)
                  if (this.vetor[j].getChave<this.vetor[minimo].getChave)
                        minimo = j;
            temp = this.vetor[minimo];
            this.vetor[minimo] = this.vetor[i];
            this.vetor[i] = temp;
      }
}
```

2.1.2 Método Seleção em Arvore – Heapsort

➢ Princípios de Funcionamento:

É o mesmo da seleção direta, entretanto utiliza uma estrutura de dados heap. Primeiro monta-se uma arvore binária heap, contendo os itens do arquivo e utiliza-se essa para a seleção das chaves na ordem desejada.

➢ Uma heap é uma arvore com as seguintes características:

- a maior chave está sempre na raiz da árvore.
- o sucessor à esquerda do elemento de índice i é o elemento de índice 2i e o sucessor à direita é o elemento de índice 2i + 1, sendo que a chave de cada nó é maior ou igual às chaves de seus filhos.

➢ Etapas:

- Montar a heap. A transformação é feita do último nível da árvore para a raiz, colocando em cada nó o elemento de maior chave entre ele e seus filhos até que se obtenha a heap.
- trocar o elemento da raiz (que possui a maior chave) com o elemento do nó que está na última posição do vetor. A seguir, isolar esse elemento e repetir o processo com os elementos restantes.

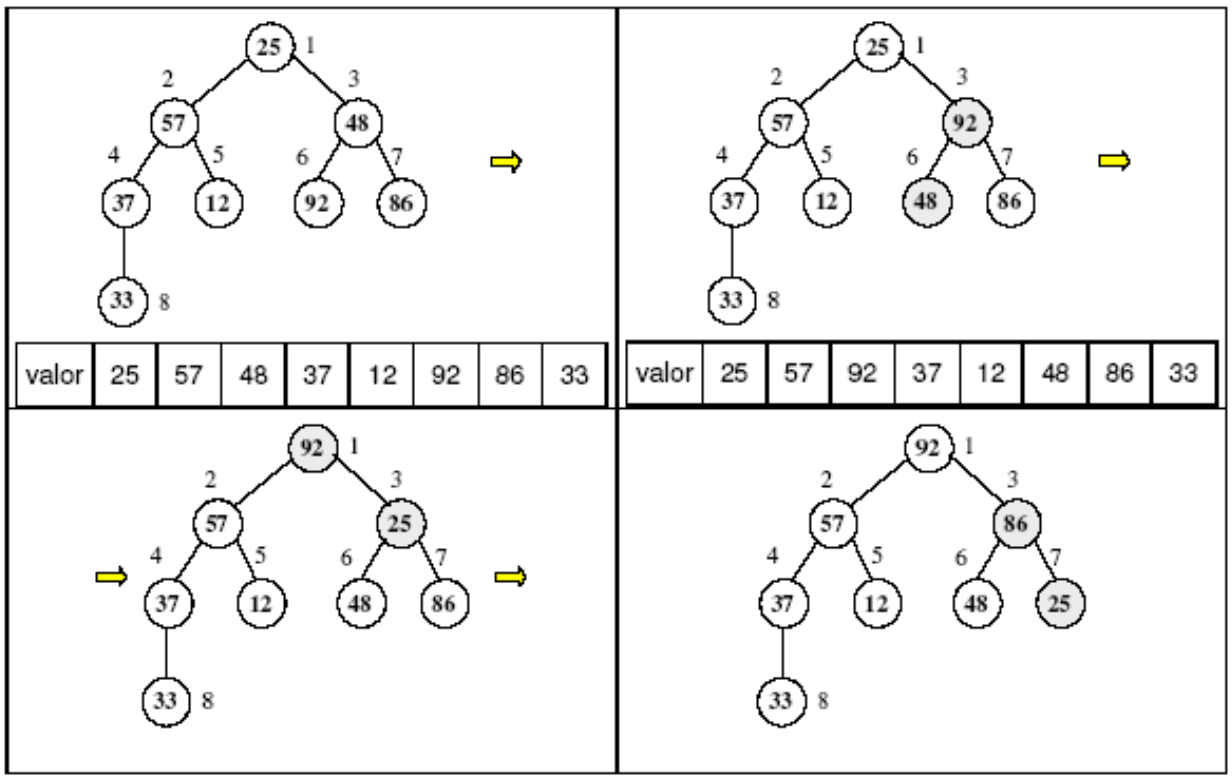
Seja um Heap representado por uma árvore binária. A árvore é formada por nodos e de cada nodo apresenta um nodo pai (parent) e até dois nodos filhos (children). A árvore apresenta um nodo denominado

raiz (root) a partir do qual são criados os filhos, seguindo a regra anterior. A condição para que uma árvore seja um heap é que a chave em cada nodo seja maior do que (ou igual a) as chaves presentes nos nodos filhos.

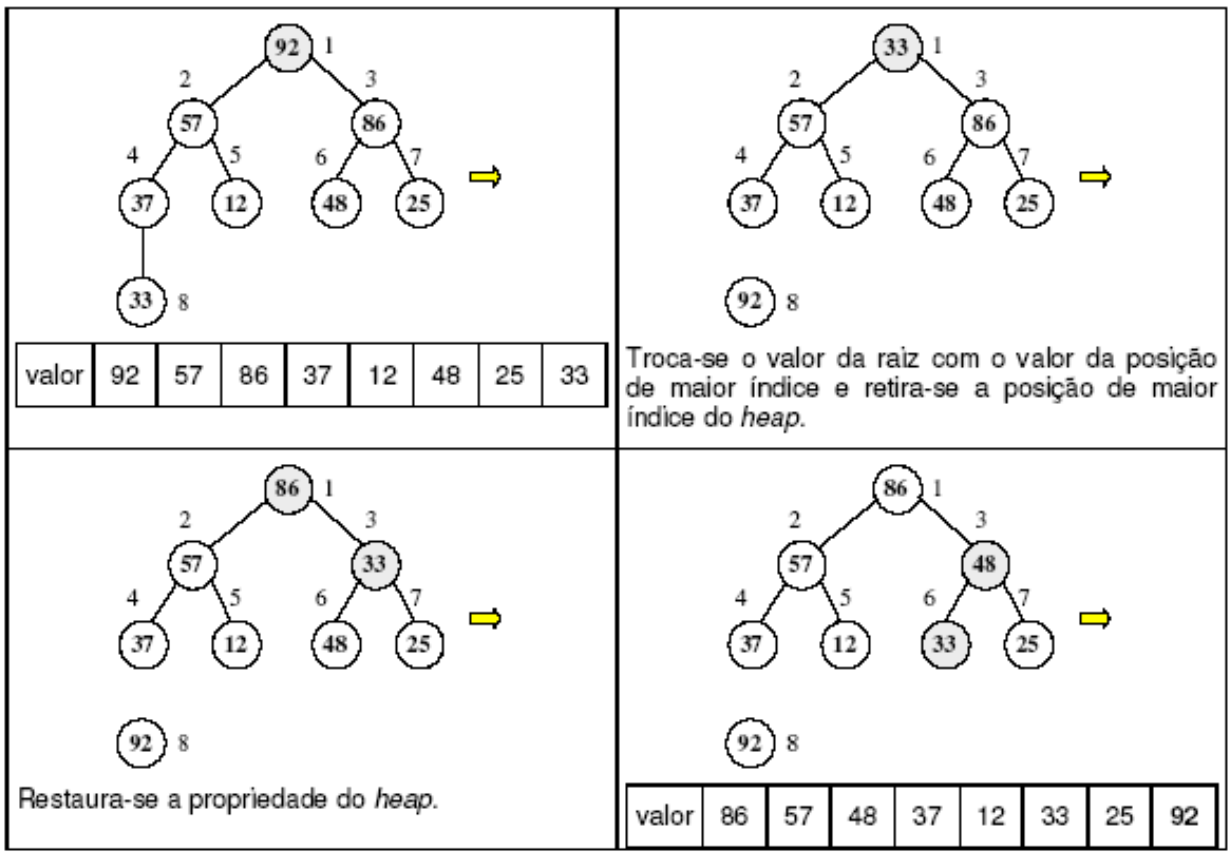
Exemplo:

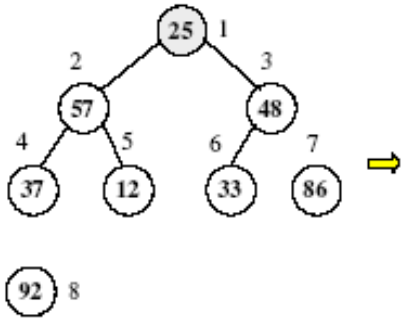
posição no vetor	1	2	3	4	5	6	7	8
valor	25	57	48	37	12	92	86	33

Transformação da árvore em *heap*

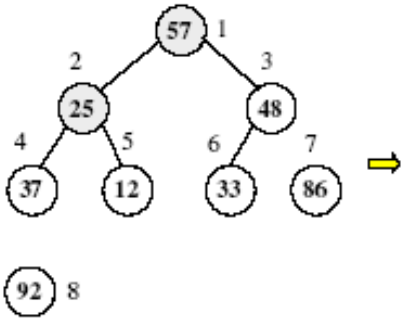


Classificação

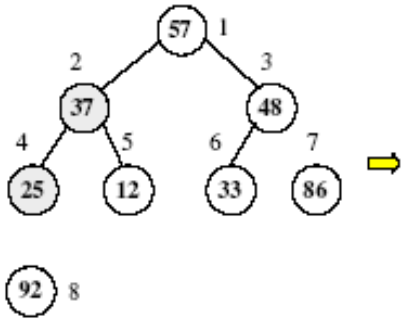




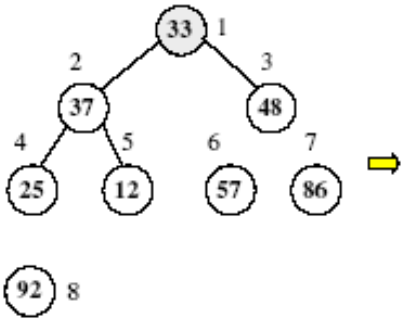
Troca-se o valor da raiz com o valor da posição de maior índice e retira-se a posição de maior índice do *heap*



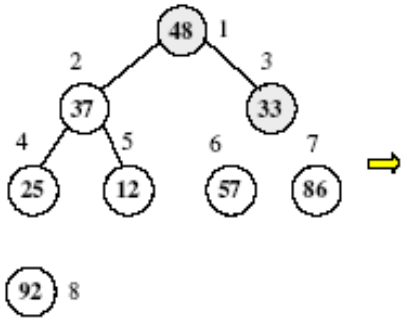
Restaura-se a propriedade do *heap*



valor	57	37	48	25	12	33	86	92

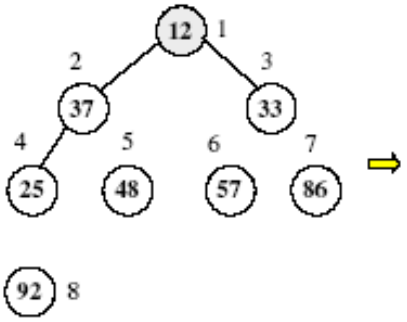


Troca-se o valor da raiz com o valor da posição de maior índice e retira-se a posição de maior índice do *heap*

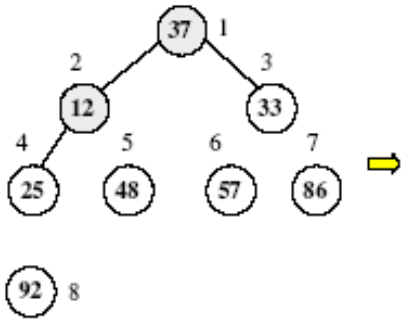


Restaura-se a propriedade do *heap*

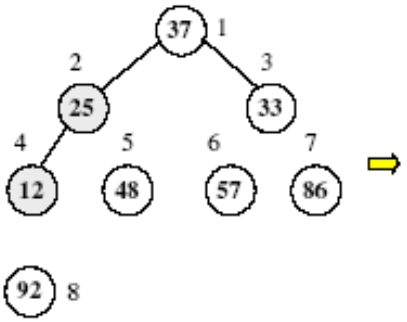
valor	48	37	33	25	12	57	86	92



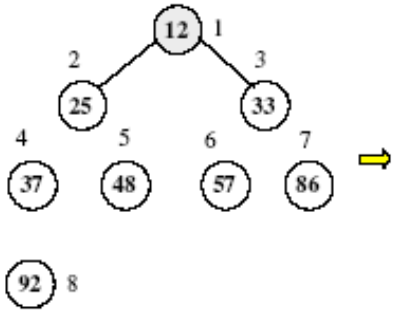
Troca-se o valor da raiz com o valor da posição de maior índice e retira-se a posição de maior índice do *heap*



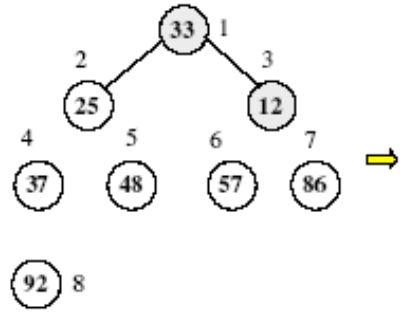
Restaura-se a propriedade do *heap*



valor	37	25	33	12	48	57	86	92

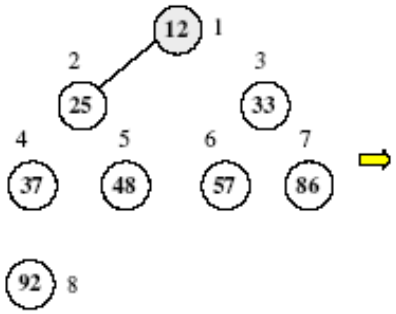


Troca-se o valor da raiz com o valor da posição de maior índice e retira-se a posição de maior índice do *heap*

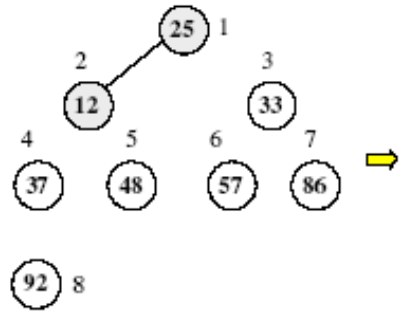


Restaura-se a propriedade do *heap*

valor	33	25	12	37	48	57	86	92

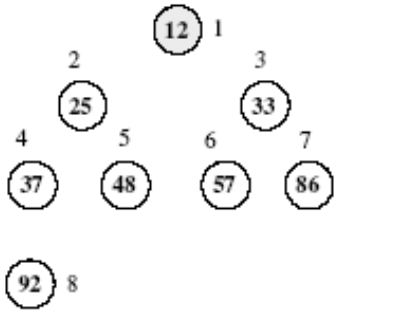


Troca-se o valor da raiz com o valor da posição de maior índice e retira-se a posição de maior índice do *heap*



Restaura-se a propriedade do *heap*

valor	25	12	33	37	48	57	86	92



Fim do processo

Ao final do processo, o vetor estará ordenado:

posição	1	2	3	4	5	6	7	8
valor	12	25	33	37	48	57	86	92

➢ Análise de Caso:

O melhor caso deve-se quando o vetor está invertido e pior caso acontece quando ele estiver previamente ordenado.

➢ Considerações:

Assim como a seleção direta, este algoritmo não é estável.

➢ Implementação do Algoritmo Heapsort.

```
public void heapSort (){
      int dir = nElem-1;
      int esq = (dir-1)/2;
      Item temp;
      while (esq >= 0)
            refazHeap (esq--, this.nElem-1);
      while (dir > 0){
            temp = this.vetor[0];
            this.vetor [0] = this.vetor [dir];
```

```
            this.vetor [dir--] = temp;
            refazHeap(0, dir);
      }
}
private void refazHeap (int esq, int dir){
      int i = esq;
      int MaiorFolha = 2*i+1;
      Item raiz = this.vetor[i];
      boolean heap = false;
      while ((MaiorFolha <= dir) && (!heap)){
            if (MaiorFolha < dir)
                  if (this.vetor[MaiorFolha].getChave()<
                              this.vetor[MaiorFolha+1].getChave())
                        MaiorFolha++;
            if (raiz.getChave() < this.vetor[MaiorFolha].getChave()) {
                  this.vetor[i] = this.vetor[MaiorFolha];
                  i = MaiorFolha;
                  MaiorFolha = 2*i+1;
            }
            else
                  heap = true;
      }
      this.vetor[i] = raiz;
}
```

2.2 Ordenação por Inserção

A característica comum a todos os métodos de classificação por inserção é que os “n” dados a serem ordenados são divididos em dois segmentos: um já ordenado e outro desordenado. A ordenação do vetor é efetivada pela inserção de cada um dos elementos em sua posição correta dentro do segmento ordenado.

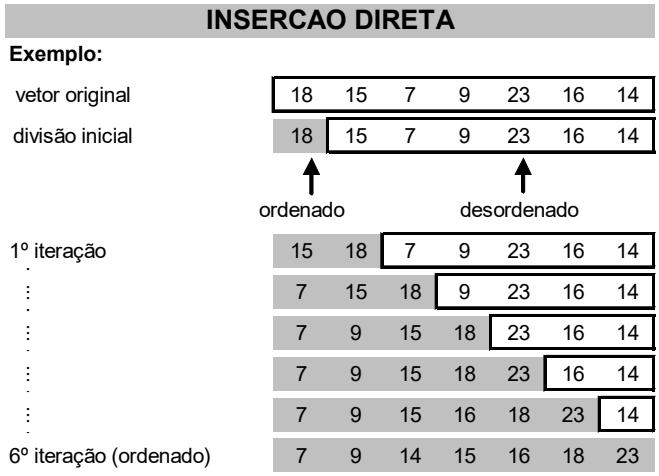
Os métodos desta família diferem um dos outros apenas pela forma como localizam a posição relativa em que cada elemento deve ser inserido no segmento já ordenado.

A seguir, veremos dois métodos de classificação por inserção, que são os métodos da Inserção Direta e o método dos incrementos decrescentes – Shellsort.

2.2.1 Método da Inserção direta

- Princípios de Funcionamento:

Neste método, o arquivo é dividido em dois segmentos. Inicialmente, o primeiro segmento contém um único elemento e conseqüentemente está ordenado. O segundo segmento contém os n-1 elementos restantes. A cada passo, a partir de i = 2, o i-ésimo elemento é transferido de segundo segmento para o primeiro, sendo inserido em sua posição apropriada.



Este método de ordenação é o mais rápido entre os outros métodos considerados básicos – Bubblesort e Seleção Direta.

- Análise de Casos:

O desempenho deste método é influenciado pela ordem inicial dos elementos do vetor. O pior caso ocorre quando o vetor a ser ordenada se apresenta na ordem contrária à desejada. Neste caso, a quantidade de comparações e de trocas de posições é igual a: **((n*n)-n)/2**

O melhor caso ocorre quando o vetor já se encontra ordenada e, portanto, nenhuma transposição será necessária, havendo a necessidade apenas de se executar **(n –1)** comparações.

O caso médio pode ser também obtido pela média aritmética entre o melhor e o pior caso.

- Considerações:

É um algoritmo estável.

- Implementação do Algoritmo Inserção Direta em pseudo-código:

```
Função Inserção_simples (V[ ] : Vetor de  inteiro, maximo: inteiro)
Inicio
                i, j, k, trab: inteiro

                para j ← 2 até maximo  faça
                        trab ← vetor[j]
                        i ← j - 1
                        enquanto  (i > 0)  e  (vetor[i] > trab) faça
                                vetor[i+1] ← vetor[i]
                                i ← i - 1
                        fim enquanto
                        vetor[i+1] ← trab
                fim para
fim
```

- Implementação do Algoritmo Inserção Direta em Java:

```
public void inserçãoDireta(){
      int i, j;
      Item temp;
      for (i=1; i < this.nElem; i++){
            temp = this.vetor[i];
            j = i-1;
            while ((j >= 0)&&(this.vetor[j].getChave()>temp.getChave())){
                  this.vetor [j+1] = this.vetor[j--];
```

```
            }
            this.vetor [j+1] = temp;
      }
}
```

2.2.2 Método de Inserção através de Incrementos Decrescentes – ShellSort

É um refinamento do método de ordenação por inserção direta.

- Princípios de Funcionamento:

A ideia básica consiste na introdução de passos nos quais os elementos defasados de uma mesma distância (incremento) são agrupados e ordenados separadamente. Cada ordenação de distância h é programada na forma de uma inserção direta e cada passo se beneficia das informações colhidas nos passos anteriores. Qualquer sequência de incrementos é aceitável, desde que a última sequência seja unitária.

Para implementar o método, o vetor é dividido em segmentos, assim formados:

seg 1: C[1], C[**h**+1], C[2**h** +1],...
seg 2: C[2], C[**h** +2], C[2**h** +2],...
...
seg **h**: C[**h**], C[**h** + **h**], C[2**h** + **h**],...

Num primeiro passo, para um **h** inicial, os segmentos assim formados são então classificados por inserção direta.

Num segundo passo, o incremento **h** é diminuído (a metade do valor anterior), dando origem a novos segmentos, os quais também serão classificados por inserção direta. A cada passo o vetor torna-se parcialmente ordenado.

Este processo se repete até que **h** seja igual a 1. Quando for feita a classificação com **h**=1, o vetor estará todo ordenado.

SHELLSORT

Exemplo:

vetor original	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1º iteração	17	25	49	12	18	23	45	38	53	42	27	13	11	28	10	14
H=4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2º iteração	11	23	10	12	17	25	27	13	18	28	445	14	53	42	49	38
H=2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3º iteração	10	12	11	13	17	14	18	23	27	25	45	28	49	38	53	42
H=1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
vetor ordenado	10	11	12	13	14	17	18	23	25	27	28	38	42	45	49	53

A escolha do número de incrementos e sua sequência específica ficam a critério do programador.

- Análise de Casos

O desempenho deste método é fortemente influenciado pela ordem inicial dos elementos do vetor. O pior caso ocorre quando o vetor a ser ordenado se apresenta na ordem contrária à desejada.

O melhor caso ocorre quando o vetor já se encontra ordenada e, portanto nenhuma transposição será necessária.

A análise deste algoritmo identifica alguns problemas matemáticos bastantes difíceis, muitos dos quais ainda não foram resolvidos. Em particular, não se sabe qual é a escolha de incrementos que deverá

fornecer os melhores resultados. Entretanto, um fato surpreendente é que eles não podem ser múltiplos uns dos outros.

- Implementação do Algoritmo em pseudocódigo:

```
Função shellsort (V[ ] : Vetor de inteiro, maximo: inteiro, h[ ] : vetor de inteiro, m : inteiro)
início
        i, j, k, trab: inteiro

        para k ← m até 1 decr 1 faça

                para j ← h[k] +1 até maximo faça
                        trab ← vetor[j]
                        i ← j – h[k]
                        enquanto (i > 0) e (vetor[i] > trab) faça
                                vetor[i+h[k]] ← vetor[i]
                                i ← i – h[k]
                        fim enquanto
                        vetor[i+h[k]] ← trab
                fim para

        fim para
fim
```

- Implementação do Algoritmo em Java:

```
public void shellSort (){
        int i, j, h;
        Item temp;
        h = 1;
        do{
                h = 3*h+1;
        }while (h < this.nElem);
        do{
                h = h/3;
                for (i=h; i < this.nElem; i++){
                        temp = this.vetor[i];
                        j = i;
                        while (this.vetor[j-h].getChave() > temp.getChave()){
                                this.vetor[j] = this.vetor[j-h];
                                j -= h;
                                if (j < h)
                                        break;
                        }
                        this.vetor [j] = temp;
                }
        }while (h != 1);
}
```

2.3 Ordenação por Permutação

Os métodos de classificação por trocas caracterizam-se por efetuarem a classificação por comparação entre pares de chaves, trocando-as de posição caso estejam fora de ordem no par.

2.3.1 Método da Bolha – Bubblesort

- Princípios de Funcionamento:

Em cada passo, se necessário, faz-se a traço entre os pares de itens adjacentes, de modo que ao término do primeiro passo, o item de menor chave esteja no início de arquivo, ao término do segundo passo o item com a segunda menor chave esteja na segunda posição do arquivo e assim sucessivamente. Em cada passo a menor chave vai sendo deslocada para o início do arquivo.

BUBBLESORT

Exemplo:

vetor original					
1º varredura	**28**	**26**	30	24	25
⋮	26	**28**	**30**	24	25
⋮	26	28	**30**	**24**	25
⋮	26	28	24	**30**	**25**
fim da 1º varredura	26	28	24	25	30
2º varredura	**26**	**28**	24	25	30
fim da 2º varredura	26	24	25	28	30
3º varredura	**26**	**24**	25	28	30
fim da 3º varredura	26	24	25	28	30

➢ Análise de Caso:

O desempenho do método Bubblesort no pior caso é sofrível e ocorre sempre que os elementos se encontrarem na ordem inversa à desejada. Isto ocorre porque, a cada iteração, apenas uma chave será colocada na posição correta.

No melhor caso, quando os elementos já estão na ordem correta, apenas n–1 comparações são necessárias para se verificar que nenhuma troca será necessária.

➢ Considerações:

É um algoritmo estável, sendo que pode ser aperfeiçoado através de um indicador de troca ou não. Se não ocorreu nenhuma troca, então o arquivo está ordenado e determina-se precocemente o término do algoritmo.

➢ Implementação do Algoritmo Bubblesort em pseudocódigo:

```
Função bubblesort (vetor[ ]: inteiro; MAXIMO : inteiro)

variáveis
        j, i, trab, limite: inteiro
        troca: lógico

inicio

        troca ← true;
        limite ← MAXIMO – 1
        enquanto (troca = true) faça
        inicio
                troca ← false
                para i ← até limite faça
                inicio
                                se vetor[i] > vetor [i + 1] então
                                        trab ← vetor[i]
                                        vetor[i] ← vetor[i + 1]
                                        vetor[i + 1] ← trab
                                        troca ← true
                                fim se
                fim para
                limite ← limite - 1
        fim enquanto
fim
```

➢ Implementação do Algoritmo Bubblesort em Java:

```
public void bubblesort (){
      int n, i, j;
      Item temp;
      n = this.nElem-1;
      do{
            i = 0;
            for (j = 0; j < n; j++)
                  if (this.vetor[j].getChave() > this.vetor[j+1].getChave()){
                        temp = this.vetor[j];
                        this.vetor[j] = this.vetor[j+1];
                        this.vetor[j+1] = temp;
                        i = j;
                  }
            n = i;
      }while (n >= 1);
}
```

2.3.2 Método da Coqueteleira – Shakesort

É um método bastante semelhante ao Bubblesort. Visando melhorar o desempenho, alterna-se o sentido do movimento a cada passo e mantêm-se indicadores das posições da última troca.

➢ Princípios de Funcionamento:
Sejam esq, dir e k indicadores da esquerda, direita e da posição da última troca, respectivamente. Inicialmente esq = 2 e dir = n. A primeira passada é feita do final para o começo do arquivo, a segunda do começo para o final e assim sucessivamente. A cada passada do final do arquivo para seu começo, garante-se que os elementos a1, a2,...ak-1 estão em ordem e faz-se esq = k + 1. Analogicamente, a cada passada em direção ao final do arquivo, garante-se que os elementos ak, ak-1,...na estão em ordem e faz-se dir = K – 1.

SHAKESORT

Exemplo:

vetor original					
1º varredura	**28**	**26**	30	24	25
	26	**28**	**30**	24	25
	26	28	**30**	**24**	25
	26	28	24	**30**	**25**
	26	28	24	25	30
	26	28	**24**	**25**	30
	26	**28**	**24**	25	30
	26	**24**	28	25	30
fim da 2º varredura	24	26	28	25	30

➢ Análise de Caso:
O desempenho do método Shakesort no pior caso é sofrível e ocorre sempre que os elementos se encontrarem na ordem inversa à desejada.

No melhor caso, quando os elementos já estão na ordem correta, apenas n–1 comparações são necessárias para se verificar que nenhuma troca será necessária.

➢ Considerações:
É um algoritmo estável.

➢ Implementação do Algoritmo Shakesort.

```
public void shakersort (){
```

```
        int esq, dir, i, j;
        Item temp;
        esq = 1;
        dir = this.nElem - 1;
        j = dir;
        do{
              for (i = dir ; i >= esq; i -- )
                    if (this.vetor[i-1].getChave() > this.vetor[i].getChave()){
                          temp = this.vetor[i];
                          this.vetor[i] = this.vetor[i-1];
                          this.vetor[i-1] = temp;
                          j = i;
                    }
              esq = j+1;
              for (i = esq ; i <= dir; i++)
                    if (this.vetor[i-1].getChave() > this.vetor[i].getChave()){
                          temp = this.vetor[i];
                          this.vetor[i] = this.vetor[i-1];
                          this.vetor[i-1] = temp;
                          j = i;
                    }
              dir = j-1;
        }while (esq <= dir);
}
```

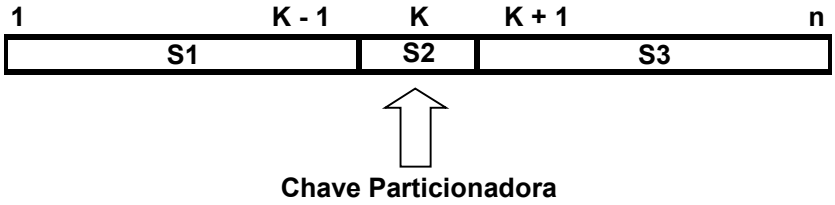
2.4 Ordenação por Partição

2.4.1 QuickSort

É um algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, sendo provavelmente mais utilizado do que qualquer outro.

➢ Princípios de funcionamento:

- escolher arbitrariamente um item do vetor como pivô;
- percorrer o vetor V a partir de seu início, até encontrar um item com chave maior ou igual à chave do pivô, índice i;
- percorrer o vetor V do final para o inicio até encontrar um item com chave menor ou igual à chave do pivô, índice j;
- trocar os itens V[i] e V[j];
- continuar o percurso e troca até que os dois ponteiros i e j se cruzem.



➢ Análise de Casos:

O melhor caso do Quicksort ocorre quando os segmentos obtidos nas partições do vetor têm tamanhos iguais ou semelhantes. Neste caso o desempenho do método para a ordenação é **O(nlog2 n).**

O pior caso ocorre quando a partição da tabela resulta um segmento vazio e o outro segmento com (tamanho – 1) elementos. Tipicamente, esta situação ocorre quando o vetor já se encontra ordenado e a chave particionadora escolhida é a primeira do vetor.

➢ Considerações:

Necessita de uma pequena pilha como memória auxiliar.
Implementação relativamente difícil.

Este algoritmo é instável.

- Implementação do Algoritmo em pseudocódigo:

```
Função QuickSort (V[ ] : Vetor de  inteiro, esquerda, direita : inteiro)
Inicio
        variáveis
        trab, aux : inteiro
        i, j : inteiro
        i ← esquerda
        j ← direita
        trab ← V[(esquerda + direita) div 2]
        repita
                enquanto trab > V[i]  faça
                inicio
                        i ← i + 1;
                fim enquanto

                enquanto trab < V[j]  faça
                inicio
                        j ← j - 1;
                fim enquanto

                se i ≤ j  então
                inicio
                        aux ← V[i]
                        V[i] ← V[j]
                        V[j] ← aux
                        i ← i + 1
                        j ← j – 1
                fim se
        até i > j

        se esq < j  então   QuickSort (V, esquerda, j)

        se dir  > i  então   QuickSort (V, i,direita)

fim
```

PARTIÇÃO

- Implementação do Algoritmo em Java:

```
public void quicksort (){
      ordena (0, this.nElem-1);
}
private void ordena (int esq, int dir){
      int pivo, i = esq, j = dir;
      Item temp;
      pivo = this.vetor[(i+j)/2].getChave();
      do {
            while (this.vetor[i].getChave() < pivo) i++;
            while (this.vetor[j].getChave() > pivo) j--;
            if (i <= j) {
                  temp = this.vetor[i];
                  this.vetor[i] = this.vetor[j];
                  this.vetor[j] = temp;
                  i++;
                  j--;
            }
      } while (i <= j);
      if (esq < j){
            ordena (esq, j);
```

```
        }
        if (dir > i){
              ordena (i, dir);
}
```