Ordenação

Livro "Projeto de Algoritmos" – Nívio Ziviani

Capítulo 4

http://www2.dcc.ufmg.br/livros/algoritmos/

Ordenação

- Introdução – Conceitos Básicos
- Ordenação Interna
  - Ordenação por Seleção
  - Ordenação por Inserção
  - ShellSort
  - QuickSort
  - HeapSort
  - MergeSort
  - Ordenação Digital
  - Ordenação Parcial

Ordenação

- Ordenação Externa
  - Intercalação Balanceada de Vários Caminhos
  - Implementação por meio de Seleção por Substituição
  - Considerações Práticas
  - Intercalação Polifásica
  - Quicksort Externo

Introdução – Conceitos Básicos

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.

Introdução – Conceitos Básicos

- Notação utilizada nos algoritmos:
  - Os algoritmos trabalham sobre os registros de um arquivo.
  - Cada registro possui uma chave utilizada para controlar a ordenação.
  - Podem existir outros componentes em um registro.

Introdução – Conceitos Básicos

- Estrutura de um registro:

```
typedef int ChaveTipo;
typedef struct Item {
    ChaveTipo Chave;
    /* outros componentes */
} Item;
```

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.

Introdução – Conceitos Básicos

- Um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.
- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.
- Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma.

Introdução – Conceitos Básicos

- Classificação dos métodos de ordenação:
  - **Ordenação interna:** arquivo a ser ordenado cabe todo na memória principal.
  - **Ordenação externa:** arquivo a ser ordenado não cabe na memória principal.

- Diferenças entre os métodos:
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - Em um método de ordenação externa, os registros são acessados seqüencialmente ou em grandes blocos.

Introdução – Conceitos Básicos

- A maioria dos métodos de ordenação é baseada em **comparações** das chaves.

- Existem métodos de ordenação que utilizam o princípio da **distribuição**.

Introdução – Conceitos Básicos

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

A < 2 < 3 < ... < 10 < J < Q < K

e

♣ < ♦ < ♥ < ♠

Introdução – Conceitos Básicos

- Algoritmo:
  1. Distribuir as cartas abertas em treze montes: ases, dois, três, : : :, reis.
  2. Colete os montes na ordem especificada.
  3. Distribua novamente as cartas abertas em quatro montes: paus, ouros, copas e espadas.
  4. Colete os montes na ordem especificada.

Introdução – Conceitos Básicos

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort ou bucketsort.**
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.
- O custo para ordenar um arquivo com n elementos é da ordem de O(n).

Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo n o número registros no arquivo, as medidas de complexidade relevantes são:
  - Número de comparações C(n) entre chaves.
  - Número de movimentações M(n) de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação ***in situ*** *são os* preferidos.

Ordenação Interna

- Métodos que utilizam listas encadeadas não são muito utilizados.

- Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

Ordenação Interna

- Classificação dos métodos de ordenação interna:
  - Métodos simples:
    - Adequados para pequenos arquivos.
    - Requerem $O(n^2)$ comparações.
    - Produzem programas pequenos.
  - Métodos eficientes:
    - Adequados para arquivos maiores.
    - Requerem $O(n \log n)$ comparações.
    - Usam menos comparações.
    - As comparações são mais complexas nos detalhes.
    - Métodos simples são mais eficientes para pequenos arquivos.

Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
typedef int Indice;
typedef Item Vetor[MaxTam + 1];
Vetor A;
```

- O índice do vetor vai de 0 até MaxTam, devido às chaves **sentinelas.**
- O vetor a ser ordenado contém chaves nas posições de 1 até n.

Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.

- Algoritmo:
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita essas duas operações com os n - 1 itens restantes, depois com os n - 2 itens, até que reste apenas um elemento.

Ordenação por Seleção

- O método é ilustrado abaixo:

	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**

- As chaves em negrito sofreram uma troca entre si.

Ordenação por Seleção

```
void Selecao (Item *A, Indice *n)
{ Indice i, j, Min;
  Item x;
  for (i = 1; i <= *n - 1; i++)
  { Min = i;
    for (j = i + 1; j <= *n; j++)
      if (A[j].Chave < A[Min].Chave) Min = j;
  x = A[Min];
  A[Min] = A[i];
  A[i] = x;
  }
}
```

Ordenação por Seleção

- **Análise**
  - Comparações entre chaves e movimentações de registros:

$$C(n) = n^2/2 - n/2$$

$$M(n) = 3(n - 1)$$

  - A atribuição Min = j é executada em média *n log n* vezes, Knuth (1973).

Ordenação por Seleção

- Vantagens:
  - Custo linear no tamanho da entrada para o número de movimentos de registros.
  - É o algoritmo a ser utilizado para arquivos com registros muito grandes.
  - É muito interessante para arquivos pequenos.
- Desvantagens:
  - O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
  - O algoritmo não é **estável.**

Ordenação por Inserção

- Método preferido dos jogadores de **cartas.**

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

Ordenação por Inserção

- O método é ilustrado abaixo:

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

- As chaves em negrito representam a seqüência destino.

Ordenação por Inserção

```
void Insercao(Item *A, Indice *n)
{ Indice i, j;
  Item x;
  for (i = 2; i <= *n; i++)
  { x = A[i]; j = i -1;
    A[0] = x; /* sentinela */
    while (x.Chave < A[j].Chave)
    { A[j+1] = A[j]; j--;
    }
    A[j+1] = x;
  }
}
```

Ordenação por Inserção

- Considerações sobre o algoritmo:
  - O processo de ordenação pode ser terminado pelas condições:
    - Um item com chave menor que o item em consideração é encontrado.
    - O final da seqüência destino é atingido à esquerda.
  - Solução:
    - Utilizar um registro **sentinela** na posição zero do vetor.

Ordenação por Inserção

- **Análise**

  - Seja C(n) a função que conta o número de comparações.

  - No anel mais interno, na i-ésima iteração, o valor de C_i é:

    - melhor caso : $C_i (n) = 1$
    - pior caso : $C_i (n) = i$
    - caso medio : $Ci(n) = 1/i (1 + 2 + ... + i) = (i+1)/2$

Ordenação por Inserção

- Assumindo que todas as permutações de n são igualmente prováveis no caso médio, temos:
  - melhor caso : $C(n) = (1 + 1 + ... + 1) = n - 1$
  - pior caso : $C(n) = (2 + 3 + ... + n) =$
    $n^2/2 + n/2 + 1$
  - caso medio : $C(n) = \frac{1}{2} (3 + 4 + ... + n + 1) =$
    $n^2/4 + 3n/4 - 1$

Ordenação por Inserção

- **Análise**
  - Seja M(n) a função que conta o número de movimentações de registros.
  - O número de movimentações na i-ésima iteração é:

    $M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$
  - Logo, o número de movimentos é:
    - melhor caso : $M(n) = (3 + 3 + ... + 3) = 3(n-1)$
    - pior caso : $M(n) = (4 + 5 + ... + n + 2) = n^2/2 + 5n/2 - 3$
    - caso medio : $M(n) = ½ (5 + 6 + \quad + n + 3) = n^2/4 + 11n/4 - 3$

Ordenação por Inserção

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é **estável.**

ShellSort

- Proposto por Shell em 1959.
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas n - 1 comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

ShellSort

- Os itens separados de *h* posições são rearranjados.

- Todo *h*-ésimo item leva a uma seqüência ordenada.

- Tal seqüência é dita estar *h*-ordenada.

ShellSort

- Exemplo de utilização:

	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*

- Quando $h = 1$, Shellsort corresponde ao algoritmo de inserção.

ShellSort

- Como escolher o valor de *h*:
  - Seqüência para h:

$$h(s) = 3h(s - 1) + 1, \quad \text{para } s > 1$$
$$h(s) = 1, \quad \text{para } s = 1.$$

  - Knuth (1973, p. 95) mostrou experimentalmente que esta seqüência é difícil de ser batida por mais de 20% em eficiência.
  - A seqüência para h corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, ...

ShellSort

```
void Shellsort (Item *A, Indice *n)
{ int i, j; int h = 1;
  Item x;
  do h = h * 3 + 1; while (h < *n);
  do
  { h /= 3;
    for (i = h + 1; i <= *n; i++)
    { x = A[i]; j = i;
      while (A[j - h].Chave > x.Chave)
      { A[j] = A[j - h]; j -= h;
        if (j <= h) goto L999;
      }
      L999: A[j] = x;
    }
  } while (h != 1);
}
```

ShellSort

- A implementação do Shellsort não utiliza registros **sentinelas.**

- Seriam necessários *h* registros sentinelas, uma para cada *h*-ordenação.

ShellSort

- **Análise**
  - A razão da eficiência do algoritmo ainda não é conhecida.
  - Ninguém ainda foi capaz de analisar o algoritmo.
  - A sua análise contém alguns problemas matemáticos muito difíceis.
  - A começar pela própria seqüência de incrementos.
  - O que se sabe é que cada incremento não deve ser múltiplo do anterior.
  - Conjecturas referente ao número de comparações para a seqüência de Knuth:

    Conjetura 1 : $C(n) = O(n^{1,25})$

    Conjetura 2 : $C(n) = O(n(\ln n)^2)$

ShellSort

- Vantagens:
  - Shellsort é uma ótima opção para arquivos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.

- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - **O método não é estável,**

Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- 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.
- Os resultados são combinados para produzir a solução final.

Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor A[Esq..Dir] é rearranjado por meio da escolha arbitrária de um **pivô** x.
- O vetor A é particionado em duas partes:
  - A parte esquerda com chaves menores ou iguais a x.
  - A parte direita com chaves maiores ou iguais a x.

Quicksort

- Algoritmo para o particionamento:
  1. Escolha arbitrariamente um **pivô** x.
  2. Percorra o vetor a partir da esquerda até que A[i] ≥ x.
  3. Percorra o vetor a partir da direita até que A[j] ≤ x.
  4. Troque A[i] com A[j].
  5. Continue este processo até os apontadores i e j se cruzarem.
- Ao final, o vetor A[Esq..Dir] está particionado de tal forma que:
  - Os itens em A[Esq], A[Esq + 1], ..., A[j] são menores ou iguais a *x*;
  - Os itens em A[i], A[i + 1], ..., A[Dir] são maiores ou iguais a *x*.

Quicksort

- Ilustração do 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*

- O pivô x é escolhido como sendo A[(i + j) / 2].
- Como inicialmente i = 1 e j = 6, então x = A[3] = D.
- Ao final do processo de partição *i* e *j* se cruzam em i = 3 e j = 2.

Quicksort

- Função Partição:

```
void Particao(Indice Esq, Indice Dir,
               Indice *i, Indice *j, Item *A)
{ Item x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j)/2]; /* obtem o pivo x */
  do
  { while (x.Chave > A[*i].Chave) (*i)++;
    while (x.Chave < A[*j].Chave) (*j)--;
    if ((*i) ≤ (*j))
    { w = A[*i]; A[*i] = A[*j]; A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}
```

Quicksort

- O anel interno do procedimento Particao é extremamente simples.

- Razão pela qual o algoritmo Quicksort é tão rápido.

Quicksort

- Função Quicksort

```
/* Entra aqui o procedimento Particao */
void Ordena(Indice Esq, Indice Dir, Item *A)
{ Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, Indice *n)
{ Ordena(1, *n, A); }
```

Quicksort

- Exemplo do estado do vetor em cada chamada recursiva do procedimento Ordena:

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*

Quicksort

- **Análise**
  - Seja C(n) a função que conta o número de comparações.
  - Pior caso: $C(n) = O(n^2)$
  - O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
  - Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.

Quicksort

- **Análise**
  - Melhor caso:

    $C(n) = 2C(n/2) + n = O(n \log n)$
  - Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
  - Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

    $C(n) \approx 1{,}386n \log n - 0{,}846n$,
  - Isso significa que em média o tempo de execução do Quicksort é $O(n \log n)$.

Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - 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.
  - O método não é **estável**.

Quicksort - Otimizacoes

Quicksort - Otimizacoes

- O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
  - Para isso basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô.
  - Mais genericamente: mediana de k elementos
- Interromper as particoes para vetores pequenos, utilizando um dos algoritmos basicos para ordena-los
  - Selecao ou insercao
- Remover a recursao: Quicksort nao recursivo
- Planejar ordem em que subvetores sao processados
  - Qual ordem leva a maior eficiencia: quando e por que?

Quicksort Nao Recursivo (uma possivel implementacao)

```
typedef struct {
          Indice Esq, Dir;
}TipoItem;

void QuickSort_NaoRec (Item *A, Indice *n)
{
  TipoPilha Pilha;
  Indice Esq, Dir, i, j;
  TipoItem Item;

  FPvazia(Pilha);
  Esq = 1;   Dir = *n;
  Item.Dir = Dir; Item.Esq = Esq;
  Empilha(Item,Pilha);
```

Quicksort Nao Recursivo (uma possivel implementacao)

```
repeat
    if (Dir > Esq) {
      Particao(Esq, Dir, &i, &j, A);
      Item.Dir = Dir; /* adia processamento do lado direito*/
      Item.Esq = i;
      Empilha(Item, Pilha);
      Dir = j;
    }
    else {
     Desempilha(Pilha, Item);
      Dir = Item.Dir;
      Esq = Item.Esq;
    }
} while (!Vazia(Pilha));

}
```

Heapsort

- 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 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.

Heapsort

- **Filas de Prioridades**
  - É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.
  - Aplicações:
    - SOs usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
    - Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
    - Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Heapsort

- **Filas de Prioridades - Tipo Abstrato de Dados**
  - Operações:
    1. Constrói uma fila de prioridades a partir de um conjunto com n itens.
    2. Informa qual é o maior item do conjunto.
    3. Retira o item com maior chave.
    4. Insere um novo item.
    5. Aumenta o valor da chave do item i para um novo valor que é maior que o valor atual da chave.
    6. Substitui o maior item por um novo item, a não ser que o novo item seja maior.
    7. Altera a prioridade de um item.
    8. Remove um item qualquer.
    9. Ajunta duas filas de prioridades em uma única.

Heapsort

- **Filas de Prioridades – Representação**
  - Representação através de uma lista linear ordenada:
    - Neste caso, Constrói leva tempo O(n log n).
    - Insere é O(n).
    - Retira é O(1).
    - Ajunta é O(n).
  - Representação através de uma lista linear não ordenada:
    - Neste caso, Constrói tem custo linear.
    - Insere é O(1).
    - Retira é O(n).
    - Ajunta é O(1) para apontadores e O(n) para arranjos.

Heapsort

- **Filas de Prioridades – Representação**
  - A melhor representação é através de uma estruturas de dados chamada *heap:*
    - Neste caso, Constrói é O(nlogn).
    - Insere, Retira, Substitui e Altera são O(log n).

  - **Observação:**
    - Para implementar a operação Ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações Insere, Retira, Substitui e Altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

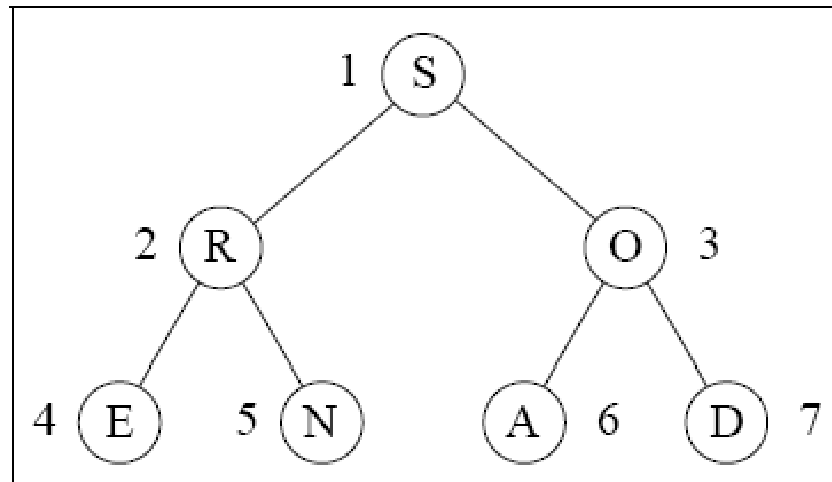
Heapsort

- **Filas de Prioridades - Algoritmos de Ordenação**
  - As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
  - Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
  - Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
  - O uso de listas lineares não ordenadas corresponde ao método da seleção.
  - O uso de listas lineares ordenadas corresponde ao método da inserção.
  - O uso de *heaps corresponde ao método* Heapsort.

Heapsort

- **Heaps**
  - É uma 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:



Heapsort

- Heaps
  - 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 <= n/2.

Heapsort

- **Heaps**
  - 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*

Heapsort

- Heaps
  - 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.
  - O 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.

Heapsort

- Heaps
  - O algoritmo não necessita de nenhuma memória auxiliar.
  - Dado um vetor A[1], A[2], ..., A[n].
  - Os itens A[n/2 + 1], A[n/2 + 2], ..., A[n] formam um *heap:*
    - Neste intervalo não existem dois índices i e j tais que j = 2i ou j = 2i + 1.

Heapsort

- **Heaps**
  - Algoritmo:

	1	2	3	4	5	6	7
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*	*S*
Esq = 3	*O*	*R*	**S**	*E*	*N*	*A*	**D**
Esq = 2	*O*	*R*	*S*	*E*	*N*	*A*	*D*
Esq = 1	**S**	*R*	**O**	*E*	*N*	*A*	*D*

Heapsort

- Heaps
  - Os itens de A[4] a A[7] formam um *heap.*
  - O *heap é estendido para a esquerda* (Esq = 3), englobando o item A[3], pai dos itens A[6] e A[7].
  - A condição de *heap é violada:*
    - O *heap é refeito trocando os itens D e S.*
  - O item R é incluindo no *heap (Esq = 2), o que* não viola a condição de *heap.*
  - O item O é incluindo no *heap (Esq = 1).*
  - A Condição de *heap violada:*
    - O *heap é refeito trocando os itens O e S,* encerrando o processo.

Heapsort

- **Heaps**
  - O Programa que implementa a operação que informa o item com maior chave:

```
Item Max(Item *A)
{ return (A[1]); }
```

Heapsort

- Programa para refazer a condição de *heap:*

```
void  Refaz(Indice Esq, Indice Dir, Item *A)
{ Indice i = Esq;
  int j;
  Item x;
  j = i * 2;
  x = A[i];
  while (j <= Dir)
  { if (j < Dir)
    { if (A[j].Chave < A[j+1].Chave) j++;
    }
    if (x.Chave >= A[j].Chave) goto L999;
    A[i] = A[j];
    i = j; j = i *2;
  }
  L999: A[i] = x;
}
```

Heapsort

- Programa para construir o *heap:*

```
void Constroi(Item *A, Indice n)
{ Indice Esq;
  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```

Heapsort

- Programa que implementa a operação de retirar o item com maior chave:

```
Item RetiraMax(Item *A, Indice *n)
{ Item Maximo;
  if (*n < 1)
    printf("Erro: heap vazio\n");
  else {
    Maximo = A[1];
    A[1] = A[*n];
    (*n)--;
    Refaz(1, *n, A);
  }
  return Maximo;
}
```

Heapsort

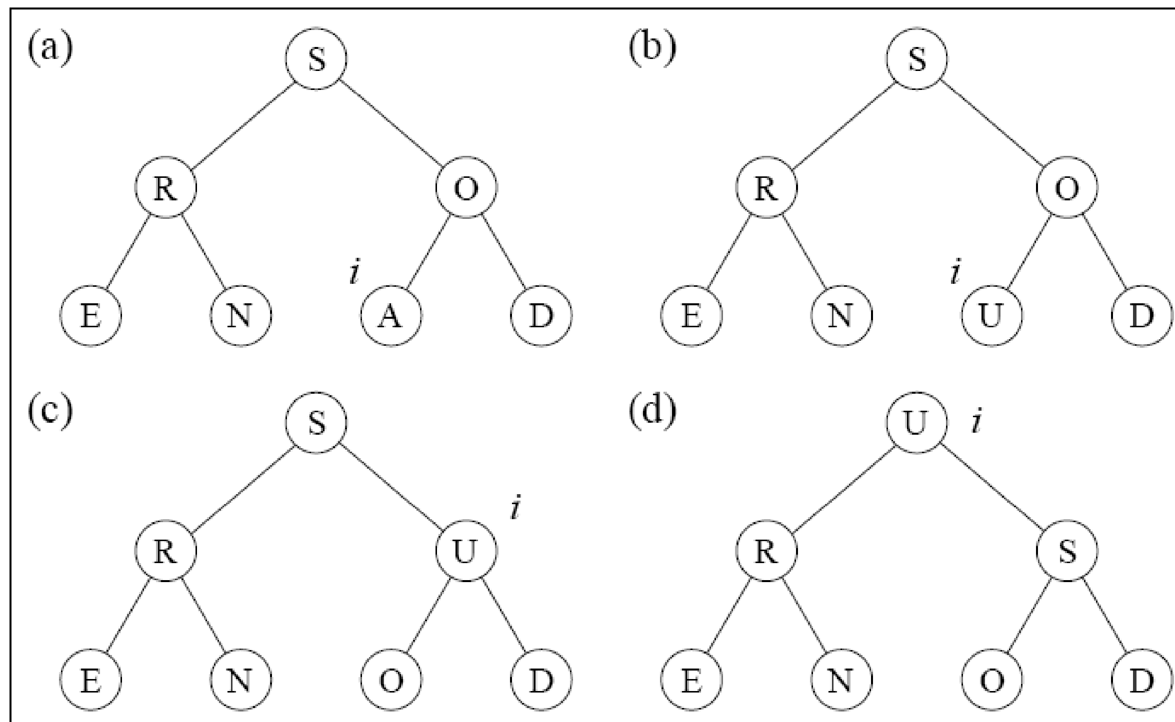
- Programa que implementa a operação de aumentar o valor da chave do item i:

```
void AumentaChave(Indice i, ChaveTipo ChaveNova, Item *A)
{ Item x;

     if (ChaveNova < A[i].Chave)
     {
       printf("Erro: ChaveNova menor que a chave atual\n");
       return;
     }
     A[i].Chave = ChaveNova;
     while (i > 1 && A[i/2].Chave < A[i].Chave) {
        x = A[i/2];
        A[i/2] = A[i];
        A[i] = x;
        i /= 2;
     }
}
```

Heapsort

- Heaps
  - Exemplo da operação de aumentar o valor da chave do item na posição i:



O tempo de execução do procedimento AumentaChave em um item do *heap é* O(log n).

Heapsort

- **Heaps**
  - Programa que implementa a operação de inserir um novo item no *heap:*

```
void Insere(Item *x, Item *A, Indice *n)
{
  (*n)++;
  A[*n] = *x;
  A[*n].Chave = INT_MIN;
  AumentaChave(*n, x->Chave, A);
}
```

Heapsort

- 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* A[1], A[2], ..., A[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.

Heapsort

- Exemplo de aplicação do Heapsort:

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*					

Heapsort

- O caminho seguido pelo procedimento Refaz para reconstituir a condição do *heap está em* negrito.
- Por exemplo, após a troca dos itens S e D na segunda linha da Figura, o item D volta para a posição 5, após passar pelas posições 1 e 2.

Heapsort

- Programa que mostra a implementação do Heapsort:

```
/* -- Entra aqui a função Refaz -- */
/* -- Entra aqui a função Constroi -- */
void Heapsort(Item *A, Indice *n)
{ Indice Esq, Dir;
  Item x;
  Constroi(A, n); /* constroi o heap */
  Esq = 1; Dir = *n;
  while (Dir > 1)
  { /* ordena o vetor */
    x = A[1];
    A[1] = A[Dir];
    A[Dir] = x;
    Dir--;
    Refaz(Esq, Dir, A);
  }
}
```

Heapsort

- **Análise**
  - O procedimento Refaz gasta cerca de log n operações, no pior caso.
  - Logo, Heapsort gasta um tempo de execução proporcional a n log n, no pior caso.

Heapsort

- 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.
  - O Heapsort não é **estável**.
- 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.*

Comparação entre os Métodos

- **Complexidade:**

	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heapsort	$O(n \log n)$

  - Apesar de não se conhecer analiticamente o comportamento do Shellsort, ele é considerado um método eficiente.

Comparação entre os Métodos

- **Tempo de execução:**
  - Observação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
  - Registros na ordem aleatória:

	5.00	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

Comparação entre os Métodos

- Tempo de Execução
  - Registros na ordem ascendente:

	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

Comparação entre os Métodos

- Tempo de Execução
  - Registros na ordem descendente:

	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

Comparação entre os Métodos

- **Observações sobre os métodos:**

1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza.
2. O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação Heapsort/Quicksort se mantém constante para todos os tamanhos.
4. A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta.

Comparação entre os Métodos

- **Observações sobre os métodos:**

5. Para arquivos 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. O Inserção é o mais rápido para qualquer tamanho se os elementos estão ordenados.
8. O 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.

Comparação entre os Métodos

- **Influência da ordem inicial do registros:**

	Shellsort			Quicksort			Heapsort		
	5.000	10.000	30.000	5.000	10.000	30.000	5.000	10.000	30.000
Asc	1	1	1	1	1	1	1,1	1,1	1,1
Des	1,5	1,6	1,5	1,1	1,1	1,1	1	1	1
Ale	2,9	3,1	3,7	1,9	2,0	2,0	1,1	1	1

Comparação entre os Métodos

- **Influência da ordem inicial do registros:**
  1. O Shellsort é bastante sensível à ordenação ascendente ou descendente da entrada.
  2. Em arquivos do mesmo tamanho, o Shellsort executa mais rápido para arquivos ordenados.
  3. O Quicksort é sensível à ordenação ascendente ou descendente da entrada.
  4. Em arquivos do mesmo tamanho, o Quicksort executa mais rápido para arquivos ordenados.
  5. O Quicksort é o mais rápido para qualquer tamanho para arquivos na ordem ascendente.
  6. O Heapsort praticamente não é sensível à ordenação da entrada.

Comparação entre os Métodos

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

Comparação entre os Métodos

- **Método da Seleção:**
  - É vantajoso quanto ao número de movimentos de registros, que é O(n).
  - Deve ser usado para arquivos com registros muito grandes, desde que o tamanho do arquivo não exceda 1.000 elementos.

Comparação entre os Métodos

- **Shellsort:**
  - É o método a ser escolhido para a maioria das aplicações por ser muito eficiente para arquivos de tamanho moderado.
  - Mesmo para arquivos 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 arquivo parcialmente ordenado trabalha menos.

Comparação entre os Métodos

- **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 arquivo está total ou parcialmente ordenado.
  - O pior caso tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios.

Comparação entre os Métodos

- **Quicksort:**
  - 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 subarquivos.
  - Para isto, basta chamar um método de ordenação simples nos arquivos pequenos.
  - A melhoria no desempenho é significativa, podendo chegar a 20% para a maioria das aplicações (Sedgewick, 1988).

Comparação entre os Métodos

- **Heapsort:**
  - É um método de ordenação elegante e eficiente.
  - 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.

Comparação entre os Métodos

- **Considerações finais:**
  - Para registros muito grandes é desejável que o método de ordenação realize apenas n movimentos dos registros.
  - Com o uso de uma **ordenação indireta é** possível se conseguir isso.
  - Suponha que o arquivo A contenha os seguintes registros: A[1], A[2], ..., A[n].
  - Seja P um arranjo P[1], P[2], ..., P[n] de apontadores.
  - Os registros somente são acessados para fins de comparações e toda movimentação é realizada sobre os apontadores.
  - Ao final, P[1] contém o índice do menor elemento de A, P[2] o índice do segundo menor e assim sucessivamente.
  - Essa estratégia pode ser utilizada para qualquer dos métodos de ordenação interna.

MergeSort
Método da Intercalação

- Princípio da Divisão e Conquista:
  - Divida o vetor A em dois subconjuntos A1 e A2
  - Solucione os sub-problemas associados a A1 e A2. Em outras palavras, ordene cada subconjunto separadamente (chamada recursiva)
    - Recursão pára quando atinge sub-problemas de tamanho 1
  - Combine as soluções de A1 e A2 em uma solução para A: intercale os dois sub-vetores A1 e A2 e obtenha o vetor ordenado
- Operação chave: intercalação.

MergeSort
Método da Intercalação

- Procedimento intercala elementos nas posições p a q-1 com os elementos nas posições q a r, do vetor A

```
void Intercala(int p, int q, int r, Vetor A)
{   int i,j,k;
    Vetor temp;
    i = p; j = q; k= 0;
    while (i <= (q-1)) &&  (j <= r) {
      if (A[i] <= A[j])  {
           temp[k] = A[i]; i = i+1;
      }
      else {
           temp[k] = A[j]; j = j+1;
      }
      k= k+1;
  }
```

MergeSort
Método da Intercalação

```
    while (i < q) {
        temp[k] = A[i];
        i = i+1; k= k+1;
    }
    while (j <= r) {
        temp[k] = A[j];
        j= j+1; k= k+1;
   }
    k = 0;
    while (k < (r-p+1)){
          A[k+p] = temp[k];
          k++;
    }
}
```

MergeSort
Método da Intercalação

Algoritmo Intercala:

- Ordem de complexidade linear O(N)
- Precisa de vetor auxiliar.
- Para o caso de intercalação de listas lineares, o procedimento pode ser realizado com a mesma complexidade e sem necessidade de memória auxiliar, bastando a manipulação de apontadores.

MergeSort
Método da Intercalação

- Procedimento MergeSort

```
void mSort (int p, int q, Vetor A)
{ int i;

  if (p < q) {
      i := (p + q) / 2;
      mSort(p,i,A);
      mSort(i+1,q,A);
      intercala(p,i+1,q,A);
}
void MergeSort(Vetor A, int  n)
{
     mSort(1,n,A);
}
```

MergeSort
Método da Intercalação

- Procedimento MergeSort – ordem de complexidade

$$T(n) = 2\,T(n/2) + n$$

$$O(n \log n)$$

Método da Contagem : *Distribution Counting*

- Problema: Ordene um arquivo de N registros cujas chaves são inteiros distintos entre 1 e N
  - Você poderia utilizar qualquer um dos métodos utilizados, mas deve tirar proveito das características específicas do arquivo (chaves distintas entre 1 e N)
  - O que fazer?

Método da Contagem : *Distribution Counting*

- Solução:

```
for i:= 1 to N do t[A[i]] := A[i];
for i:= 1 to N do a[i] := t[i];
```

- Utiliza vetor auxiliar t de tamanho N.
- Ordem de complexidade linear O(N)

Método da Contagem : *Distribution Counting*

- Problema 2 : Ordene um arquivo de N registros, cujas chaves são inteiros entre 0 e M-1

Método da Contagem : *Distribution Counting*

- Problema 2 : Ordene um arquivo de N registros, cujas chaves são inteiros entre 0 e M-1

- Idéia

– Conte o número de chaves com cada valor

– Acumule os valores para determinar o número de chaves com valores menor ou igual a um determinado valor

– Utilize estes contadores para mover os registros para as posições corretas, em um segundo passo

Método da Contagem : *Distribution Counting*

- Solução:

```
for (j=0, j <= (M-1); j++) count[j] = 0;
for (i=0; i < N; i++) count[A[i]] = count[A[i]] + 1;
for (j=0; j<= (M-1); j++) count[j] = count[j] + count[j-1];
for (i=(N-1); i >=0; i--)  {
      t[count[A[i]] = A[i];
      count[A[i]] = count[A[i]] - 1;
}
for (i=0; i < N; i++) do A[i]= t[i];
```

- Algoritmo eficiente se M não for muito grande.
- Complexidade linear O(max(N,M))
- Base para algoritmos *radix sorting*

Radix Sorting

- Os registros a serem ordenados podem ter chaves bastante complexas: sequências de caracteres (lista telefônica)
  - Ordenação via comparação de chaves
- Várias aplicações têm chaves que são *inteiros*, definidos dentro de um intervalo
- Radix sorting: métodos que tiram proveito das propriedades digitais destes números
  - Chaves tratadas como números na base M (raiz ou radix)
  - Processamento e comparação de *pedaços* (bits) das chaves
  - Ex: ordenar faturas de cartão de crédito considerando pilhas com o mesmo dígito em cada posição. Neste caso M = 10

Radix Sorting

- **Operação fundamental**: dada um chave representada por um número binário *a*, extrair um conjunto contíguo de bits
  - Extrair os dois bits mais significativos de um número de 10 bits
  - Desloque (shift) os bits para a direita 8 posições, e faça um “and” bit-a-bit com a máscara 0000000011
  - O *shift* elimina os bits à direita dos desejados e o *and* zera os bits à esquerda deles.
  - Logo o resultado contém apenas os bits desejados.

- Em C:
  - shift : >>
  - And bit-a-bit: &

Radix Sorting

- *Radix sorting*: idéia geral
  - Examine os bits das chaves um a um (ou em grupos), movendo as chaves com bits 0 para antes das chaves com bit 1
  - Dois algoritmos: *radix exchange* e *straight radix*

- *Radix exchange sort*: examina bits da esquerda para a direita
  - Ordem relativa de duas chaves depende apenas do valor dos bits na posição mais significativa na qual eles diferem (primeira posição diferente da esquerda para a direita)
- *Straight radix sort:* examina bits da direita para a esquerda

Radix Sorting

- *Radix Exchange Sorting:*
  - Rearranje os registros do arquivo tal que todas as chaves com o bit mais significativo igual a 0 aparecem antes das chaves com o bit 1.
  - Repita este processo para o 2o bit mais significativo, o 3o bit mais significativo ….
  - Em cada passo, particiona o vetor em função do valor do bit sendo considerado: estratégia parecida com Quicksort
  - Operação básica: extrair o (B-k)-esimo bit mais significativo, onde B é o número de bits da representação (j = 1 abaixo)

```
int bits(int x, int k, int j) {
    return (x >> k) & j;
}
```

Radix Sorting

```
void radixexchange (int esq, int dir, int  b, Vetor A)
{   int t, i, j;

   if ((dir > esq ) && (b >= 0))  {
      i= esq;j= dir;
      do {
          while ((bits(A[i],b,1) == 0) &&  (i < j)) i++;
          while ((bits(A[j],b,1) == 1) && (i < j )) j--;
          t = A[i]; A[i] = A[j]; A[j] = t;
      } while (j != i);
      if (bits(A[dir],b,1)== 0) j++;
      radixexchange(esq, j-1, b-1,A);
      radixexchange(j, dir, b-1,A);
   }
}
```

Radix Sorting

- Se A[1..N] contém inteiros positivos menores que 2^{32}, então eles podem ser representados como números binários de 31 bits.
  - Bit mais à esquerda é o bit 30, bit mais à direita é o bit 0.
  - Para ordená-los, basta chamar **radixexchange(1,N,30)**
- Note que somente os bits que diferem uma chave das demais são investigados
  - Uma vez que se atinge um prefixo que diferencia a chave das demais, os demais bits (menos significativos) daquela chave não são mais investigados.
  - Em contrapartida, todos os bits de chaves iguais são investigados : não funciona bem se arquivo contém muitas chaves iguais
- Problema em potencial: partições degeneradas (todas chaves com o mesmo bit)
  - Radix exchange ligeiramente melhor que Quicksort se chaves verdadeiramente aleatórias, mas Quicksort se adapta melhor para situações menos aleatórias.

Radix Sorting

- *Straight Radix Sorting:*
  - Examina os bits da direita para a esquerda, *m* bits por vez.
    - W = Número de bits de uma chave; múltiplo de *m*
  - A cada passo reordene as chaves utilizando *Distribution Counting* considerando os *m* bits.
    - Neste caso, o vetor de contadores tem tamanho M = 2^m
    - Cada passo executa em tempo linear.
    - Mask = máscara com m bits iguais a 1 (mask = 2^m -1)
  - Número de passos: W / m
  - Tem-se um compromisso entre espaço e tempo
    - Quanto maior m, mais rápido o algoritmo roda, mas mais espaço é necessário para o vetor de contadores.
  - Além do vetor de contadores, precisa também de um vetor auxiliar de tamanho N

Radix Sorting

```
void straightradix(Vetor A; int n)
{ int i, j, k, pass;
  int count[M];
  Vetor B;
  for (pass=0; pass <= (W/m)-1; pass++) {
      for (j=0; j < M; j++) count[j] = 0;
      for (i=0; i< N; i++) {
          k= bits(A[i], pass*m, mask);  count[k] = count[k] + 1;
      }
      for (j=1; j< M; j++) count[j]= count[j-1] + count[j];
      for (i=(N-1); i>=0; i--) {
          k= bits(A[i], pass*m, mask);
          b[count[k]] = A[i]; count[k] = count[k] -1;
      }
      for (i=0; i <N; i++) A[i] = b[i];
   }
}
```

Radix Sorting

- Análise de complexidade de Radix Exchange:
  - Em cada passo, em média metade dos bits são 0 e metade são 1:
  - $C(N) = C(N/2) + N$
  - Algoritmo faz em média N log N comparações de bits
- Radix Exchange e Straight Radix fazem menos que *Nb* comparações de bits para ordenar N chaves de b bits
  - Linear no número de bits
- Straight radix: ordena N chaves de b bits em b/m passos, cada um com complexidade O(N), usando um espaço extra para 2^m contadores (e um vetor extra).
  - Fazendo m = b/4: algoritmo tem número constante (4) de passos: O(N)
- Embora tenha custo linear, Straight Radix realiza número grande de operações no loop interno (constante grande)
  - Superior a Quicksort somente para arquivos muito grandes
- Principal desvantagem do Straight Radix: espaço extra

Ordenação Parcial

- Consiste em obter os k primeiros elementos de um arranjo ordenado com n elementos.
- Quando k = 1, o problema se reduz a encontrar o mínimo (ou o máximo) de um conjunto de elementos.
- Quando k = n caímos no problema clássico de ordenação.

Ordenação Parcial

- **Aplicações:**
  - Facilitar a busca de informação na Web com as **máquinas de busca:**
    - É comum uma consulta na Web retornar centenas de milhares de documentos relacionados com a consulta.
    - O usuário está interessado apenas nos k documentos mais relevantes.
    - Em geral k é menor do que 200 documentos.
    - Normalmente são consultados apenas os dez primeiros.
    - Assim, são necessários algoritmos eficientes de ordenação parcial.

Ordenação Parcial

Algoritmos considerados:

- Seleção parcial.
- Inserção parcial.
- Heapsort parcial.
- Quicksort parcial.

Seleção Parcial

- Um dos algoritmos mais simples.
- Princípio de funcionamento:
  - Selecione o menor item do vetor.
  - Troque-o com o item que está na primeira posição do vetor.
  - Repita estas duas operações com os itens n – 1, n – 2, ... n – k.

Seleção Parcial

```
void SelecaoParcial(Vetor A, Indice *n,
  Indice *k)
{ Indice i, j, Min; Item x;
  for (i = 1; i <= *k; i++)
  { Min = 1;
    for (j = i + 1; j <= *n; j++)
      if (A[j].Chave < A[Min].Chave)
        Min = j;
    x = A[Min]; A[Min] = A[i]; A[i] = x;
  }
}
```

Seleção Parcial

- **Análise:**
  - Comparações entre chaves e movimentações de registros:

    $C(n) = kn - k^2/2 - k/2$

    $M(n) = 3k$

Seleção Parcial

- É muito simples de ser obtido a partir da implementação do algoritmo de ordenação por seleção.
- Possui um comportamento espetacular quanto ao número de movimentos de registros:
  - Tempo de execução é linear no tamanho de k.

Inserção Parcial

- Pode ser obtido a partir do algoritmo de ordenação por Inserção por meio de uma modificação simples:
  - Tendo sido ordenados os primeiros k itens, o item da k-ésima posição funciona como um pivô.
  - Quando um item entre os restantes é menor do que o pivô, ele é inserido na posição correta entre os k itens de acordo com o algoritmo original.

Inserção Parcial

```
void InsercaoParcial(Vetor A, Indice *n,
  Indice *k)
{ /* -- Não preserva o restante do vetor --* /
  Indice i, j; Item x;
  for (i = 2; i <= *n; i++)
  { x = A[i];
    if (i > *k) j = *k; else j = i - 1;
    A[0] = x; /* sentinela */
    while (x.Chave < A[j].Chave)
    { A[j+1] = A[j];
      j--;
    }
    A[j+1] = x;
  }
}
```

Inserção Parcial

- **Obs:**
  1. A modificação realizada verifica o momento em que i se torna maior do que k e então passa a considerar o valor de j igual a k a partir deste ponto.
  2. O algoritmo não preserva o restante do vetor não.

Inserção Parcial

Algoritmo de Inserção Parcial que preserva o restante do vetor:

```
void InsercaoParcial2(Vetor A, Indice *n, Indice *k)
{ /* -- Preserva o restante do vetor -- */
  Indice i, j; Item x;
  for (i = 2; i <= *n; i++)
  { x = A[i];
    if (i > *k)
    { j = *k;
      if (x.Chave < A[*k].Chave A[i] = A[*k];
    }
    else j = i – 1;
    A[0] = x; /* sentinela */
    while (x.Chave < A[j].Chave)
    { if (j < *k) { A[j+1] = A[j]; }
      j--;
    }
    if (j < *k) A[j+1] = x;
  }
}
```

Inserção Parcial

- **Análise:**
  - No anel mais interno, na i-ésima iteração o valor de C_i é:

    melhor caso : $C_i(n) = 1$

    pior caso : $C_i(n) = i$

    caso medio : $C_i(n) = 1/i\ (1 + 2 + ... + i) = (i+1)/2$

  - Assumindo que todas as permutações de n são igualmente prováveis, o número de comparações é:

    melhor caso : $C(n) = (1 + 1 + ... + 1) = n – 1$

    pior caso : $C(n) = (2 + 3 + ... + k + (k + 1)(n - k))$
    $= kn + n – k2/2 – k/2 – 1$

    caso medio : $C(n) = ½\ (3 + 4 + ... + k + 1 + (k + 1)(n - k))$
    $= kn/2 + n/2 – k2/4 + k/4 – 1$

Inserção Parcial

- **Análise:**
  - O número de movimentações na i-ésima iteração é:

    $M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$
  - Logo, o número de movimentos é:

    melhor caso : $M(n) = (3 + 3 + ... + 3) = 3(n - 1)$

    pior caso : $M(n) = (4 + 5 + ... + k + 2 + (k + 1)(n - k))$
    $= kn + n - k^2/2 + 3k/2 - 3$

    caso medio : $M(n) = ½ (5 + 6 + ... + k + 3 + (k + 1)(n - k))$
    $= kn/2 + n/2 - k2/4 + 5k/4 - 2$
  - O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
  - O número máximo ocorre quando os itens estão originalmente na ordem reversa.

Heapsort Parcial

- Utiliza um tipo abstrato de dados *heap para* informar o menor item do conjunto.
- Na primeira iteração, o menor item que está em a[1] (raiz do *heap) é trocado com o item* que está em A[n].
- Em seguida o *heap é refeito.*
- Novamente, o menor está em A[1], troque-o com A[n-1].
- Repita as duas últimas operações até que o k-ésimo menor seja trocado com A[n – k].
- Ao final, os k menores estão nas k últimas posições do vetor A.

Heapsort Parcial

```
/* -- Entram aqui as funções Refaz e Constroi -- */
/* -- Coloca menor em A[n], segundo menor em A[n-1],
   ..., -- */
/* -- k-ésimo em A[n-k] -- */
void HeapsortParcial(Item *A, Indice *n, Indice *k)
{ Indice Esq = 1; Indice Dir;
  Item x; long Aux = 0;
  Constroi(A, n); /* -- Constroi o heap -- */
  Dir = *n;
  while (Aux < *k)
  { /* ordena o vetor */
    x = A[1]; A[1] = A[*n - Aux]; A[*n - Aux] = x;
    Dir--; Aux++;
    Refaz(Esq, Dir, A);
  }
}
```

Heapsort Parcial

- **Análise:**
  - O HeapsortParcial deve construir um *heap a* um custo O(n).
  - O procedimento Refaz tem custo O(log n).
  - O procedimento HeapsortParcial chama o procedimento Refaz k vezes.
  - Logo, o algoritmo apresenta a complexidade:

$$O(n + k \log n) = \begin{cases} O(n) & \text{se } k \le n/\log n \\ O(k \log n) & \text{se } k > n/\log n \end{cases}$$

Quicksort Parcial

- Assim como o Quicksort, o Quicksort Parcial é o algoritmo de ordenação parcial mais rápido em várias situações.
- A alteração no algoritmo para que ele ordene apenas os k primeiros itens dentre n itens é muito simples.
- Basta abandonar a partição à direita toda vez que a partição à esquerda contiver k ou mais itens.
- Assim, a única alteração necessária no Quicksort é evitar a chamada recursiva Ordena(i,Dir).

Quicksort Parcial

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*

- Considere k = 3 e D o pivô para gerar as linhas 2 e 3.
- A partição à esquerda contém dois itens e a partição à direita contém quatro itens.
- A partição à esquerda contém menos do que k itens.
- Logo, a partição direita não pode ser abandonada.
- Considere E o pivô na linha 3.
- A partição à esquerda contém três itens e a partição à direita também.
- Assim, a partição à direita pode ser abandonada.

Quicksort Parcial

```
/* -- Entra aqui a função Partição -- */
void Ordena(Vetor A, Indice Esq, Indice Dir,
  Indice k)
{ Indice i, j;
  Particao(A, Esq, Dir, &i, &j);
  if (j - Esq >= k - 1)
  { if (Esq < j) Ordena(A, Esq, j, k);
    return;
  }
  if (Esq < j) Ordena(A, Esq, j, k);
  if (i < Dir) Ordena(A, i, Dir, k);
}

void QuickSortParcial(Vetor A, Indice *n,
  Indice *k)
{ Ordena(A, 1, *n, *k); }
```

Quicksort Parcial

- **Análise:**
  - A análise do Quicksort é difícil.
  - O comportamento é muito sensível à escolha do pivô.
  - Podendo cair no melhor caso O(k log k).
  - Ou em algum valor entre o melhor caso e O(n log n).

Comparação entre os Métodos de Ordenação Parcial

n, k	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n: 10^1 \quad k: 10^0$	1	2,5	1	1,2	1,7
$n: 10^1 \quad k: 10^1$	1,2	2,8	1	1,1	2,8
$n: 10^2 \quad k: 10^0$	1	3	1,1	1,4	4,5
$n: 10^2 \quad k: 10^1$	1,9	2,4	1	1,2	3
$n: 10^2 \quad k: 10^2$	3	1,7	1	1,1	2,3
$n: 10^3 \quad k: 10^0$	1	3,7	1,4	1,6	9,1
$n: 10^3 \quad k: 10^1$	4,6	2,9	1	1,2	6,4
$n: 10^3 \quad k: 10^2$	11,2	1,3	1	1,4	1,9
$n: 10^3 \quad k: 10^3$	15,1	1	3,9	4,2	1,6
$n: 10^5 \quad k: 10^0$	1	2,4	1,1	1,1	5,3
$n: 10^5 \quad k: 10^1$	5,9	2,2	1	1	4,9
$n: 10^5 \quad k: 10^2$	67	2,1	1	1,1	4,8
$n: 10^5 \quad k: 10^3$	304	1	1,1	1,3	2,3
$n: 10^5 \quad k: 10^4$	1445	1	33,1	43,3	1,7
$n: 10^5 \quad k: 10^5$	∞	1	∞	∞	1,9

Comparação entre os Métodos de Ordenação Parcial

n, k	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n : 10^6 \quad k : 10^0$	1	3,9	1,2	1,3	8,1
$n : 10^6 \quad k : 10^1$	6,6	2,7	1	1	7,3
$n : 10^6 \quad k : 10^2$	83,1	3,2	1	1,1	6,6
$n : 10^6 \quad k : 10^3$	690	2,2	1	1,1	5,7
$n : 10^6 \quad k : 10^4$	∞	1	5	6,4	1,9
$n : 10^6 \quad k : 10^5$	∞	1	∞	∞	1,7
$n : 10^6 \quad k : 10^6$	∞	1	∞	∞	1,8
$n : 10^7 \quad k : 10^0$	1	3,4	1,1	1,1	7,4
$n : 10^7 \quad k : 10^1$	8,6	2,6	1	1,1	6,7
$n : 10^7 \quad k : 10^2$	82,1	2,6	1	1,1	6,8
$n : 10^7 \quad k : 10^3$	∞	3,1	1	1,1	6,6
$n : 10^7 \quad k : 10^4$	∞	1,1	1	1,2	2,6
$n : 10^7 \quad k : 10^5$	∞	1	∞	∞	2,2
$n : 10^7 \quad k : 10^6$	∞	1	∞	∞	1,2
$n : 10^7 \quad k : 10^7$	∞	1	∞	∞	1,7

Comparação entre os Métodos de Ordenação Parcial

1. Para valores de k até 1.000, o método da InserçãoParcial é imbatível.
2. O QuicksortParcial nunca ficar muito longe da InserçãoParcial.
3. Na medida em que o k cresce,o QuicksortParcial é a melhor opção.
4. Para valores grandes de k, o método da InserçãoParcial se torna ruim.
5. Um método indicado para qualquer situação é o QuicksortParcial.
6. O HeapsortParcial tem comportamento parecido com o do QuicksortParcial.
7. No entano, o HeapsortParcial é mais lento.

Ordenação Externa

- A ordenação externa consiste em ordenar arquivos de tamanho maior que a memória interna disponível.
- Os métodos de ordenação externa são muito diferentes dos de ordenação interna.
- Na ordenação externa os algoritmos devem diminuir o número de acesso as unidades de memória externa.
- Nas memórias externas, os dados são armazenados como um arquivo seqüencial.
- Apenas um registro pode ser acessado em um dado momento.
- Esta é uma restrição forte se comparada com as possibilidades de acesso em um vetor.
- Logo, os métodos de ordenação interna são inadequados para ordenação externa.
- Técnicas de ordenação completamente diferentes devem ser utilizadas.

Ordenação Externa

- Fatores que determinam as diferenças das técnicas de ordenação externa:
  1. Custo para acessar um item é algumas ordens de grandeza maior.
  2. O custo principal na ordenação externa é relacionado a transferência de dados entre a memória interna e externa.
  3. Existem restrições severas de acesso aos dados.
  4. O desenvolvimento de métodos de ordenação externa é muito dependente do estado atual da tecnologia.
  5. A variedade de tipos de unidades de memória externa torna os métodos dependentes de vários parâmetros.
  6. Assim, apenas métodos gerais serão apresentados.

Ordenação Externa

- O método mais importante é o de ordenação por intercalação.
- Intercalar significa combinar dois ou mais blocos ordenados em um único bloco ordenado.
- A intercalação é utilizada como uma operação auxiliar na ordenação.
- Estratégia geral dos métodos de ordenação externa:
  1. Quebre o arquivo em blocos do tamanho da memória interna disponível.
  2. Ordene cada bloco na memória interna.
  3. Intercale os blocos ordenados, fazendo várias passadas sobre o arquivo.
  4. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Ordenação Externa

- Os algoritmos para ordenação externa devem reduzir o número de passadas sobre o arquivo.
- Uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória auxiliar.
- Os bons métodos de ordenação geralmente envolvem no total menos do que dez passadas sobre o arquivo.

Intercalação Balanceada de Vários Caminhos

- Considere um arquivo armazenado em uma fita de entrada:

I N T E R C A L A C A O B A L A N C E A D A

- Objetivo:
  - Ordenar os 22 registros e colocá-los em uma fita de saída.
- Os registros são lidos um após o outro.
- Considere uma memória interna com capacidade para para três registros.
- Considere que esteja disponível seis unidades de fita magnética.

Intercalação Balanceada de Vários Caminhos

- Fase de criação dos blocos ordenados:

fita 1:	*I N T*	*A C O*	*A D E*
fita 2:	*C E R*	*A B L*	*A*
fita 3:	*A A L*	*A C N*	

Intercalação Balanceada de Vários Caminhos

- Fase de intercalação - Primeira passada:
  1. O primeiro registro de cada fita é lido.
  2. Retire o registro contendo a menor chave.
  3. Armazene-o em uma fita de saída.
  4. Leia um novo registro da fita de onde o registro retirado é proveniente.
  5. Ao ler o terceiro registro de um dos blocos, sua fita fica inativa.
  6. A fita é reativada quando o terceiro registro das outras fitas forem lidos.
  7. Neste instante um bloco de nove registros ordenados foi formado na fita de saída.
  8. Repita o processo para os blocos restantes.

Intercalação Balanceada de Vários Caminhos

- Resultado da primeira passada da segunda etapa:

fita 4: *A A C E I L N R T*

fita 5: *A A A B C C L N O*

fita 6: *A A D E*

Intercalação Balanceada de Vários Caminhos

- Quantas passadas são necessárias para ordenar um arquivo de tamanho arbitrário?
  - Seja n, o número de registros do arquivo.
  - Suponha que cada registro ocupa m palavras na memória interna.
  - A primeira etapa produz n=m blocos ordenados.
  - Seja P(n) o número de passadas para a fase de intercalação.
  - Seja f o número de fitas utilizadas em cada passada.
  - Assim:

$$P(n) = \log_f n/m.$$

No exemplo acima, n=22, m=3 e f=3 temos:

$$P(n) = \log_3 22/3 = 2:$$

Intercalação Balanceada de Vários Caminhos

- No exemplo foram utilizadas 2f fitas para uma intercalação-de-f-caminhos.
- É possível usar apenas f + 1 fitas:
  - Encaminhe todos os blocos para uma única fita.
  - Redistribuia estes blocos entre as fitas de onde eles foram lidos.
  - O custo envolvido é uma passada a mais em cada intercalação.
- No caso do exemplo de 22 registros, apenas quatro fitas seriam suficientes:
  - A intercalação dos blocos a partir das fitas 1, 2 e 3 seria toda dirigida para a fita 4.
  - Ao final, o segundo e o terceiro blocos ordenados de nove registros seriam transferidos de volta para as fitas 1 e 2.

Implementação por meio de Seleção por Substituição

- A implementação do método de intercalação balanceada pode ser realizada utilizando filas de prioridades.
- As duas fases do método podem ser implementadas de forma eficiente e elegante.
- Operações básicas para formar blocos ordenados:
  - Obter o menor dentre os registros presentes na memória interna.
  - Substituí-lo pelo próximo registro da fita de entrada.
- Estrutura ideal para implementar as operações: *heap.*
- Operação de substituição:
  - Retirar o menor item da fila de prioridades.
  - Colocar um novo item no seu lugar.
  - Reconstituir a propriedade do *heap.*

Implementação por meio de Seleção por Substituição

- Algoritmo:
  1. Inserir m elementos do arquivo na fila de prioridades.
  2. Substituir o menor item da fila de prioridades pelo próximo item do arquivo.
  3. Se o próximo item é menor do que o que saiu, então:
     - Considere-o membro do próximo bloco.
     - Trate-o como sendo maior do que todos os itens do bloco corrente.
  4. Se um item marcado vai para o topo da fila de prioridades então:
     - O bloco corrente é encerrado.
     - Um novo bloco ordenado é iniciado.

Implementação por meio de Seleção por Substituição

- Primeira passada sobre o arquivo exemplo:

Entra	1	2	3
E	I	N	T
R	N	E*	T
C	R	E*	T
A	T	E*	C*
L	A*	E*	C*
A	C*	E*	L*
C	E*	A	L*
A	L*	A	C
O	A	A	C
B	A	O	C

Implementação por meio de Seleção por Substituição

- Primeira passada sobre o arquivo exemplo: (cont.)

A	B	O	C
L	C	O	A*
A	L	O	A*
N	O	A*	A*
C	A*	N*	A*
E	A*	N*	C*
A	C*	N*	E*
D	E*	N*	A
A	N*	D	A
	A	D	A
	A	D	
	D		

- Os asteriscos indicam quais chaves pertencem a blocos diferentes.

Implementação por meio de Seleção por Substituição

- Tamanho dos blocos produzidas para chaves randômicas:
  - Os blocos ordenados são cerca de duas vezes o tamanho dos blocos criados pela ordenação interna.
- Isso pode salvar uma passada na fase de intercalação.
- Se houver alguma ordem nas chaves, os blocos ordenados podem ser ainda maiores.
- Se nenhuma chave possui mais do que m chaves maiores do que ela, o arquivo é ordenado em um passo.

Implementação por meio de Seleção por Substituição

- Exemplo para as chaves RAPAZ:

Entra	1	2	3
A	*A*	*R*	*P*
Z	*A*	*R*	*P*
	P	*R*	*Z*
	R	*Z*	
	Z		

Implementação por meio de Seleção por Substituição

- Fase de intercalação dos blocos ordenados obtidos na primeira fase:
  - Operação básica: obter o menor item dentre os ainda não retirados dos f blocos a serem intercalados.
- Algoritmo:
  - Monte uma fila de prioridades de tamanho f.
  - A partir de cada uma das f entradas:
    - **Substitua o item no topo da fila de** prioridades pelo próximo item do mesmo bloco do item que está sendo substituído.
    - **Imprima em outra fita o elemento** substituído.

Implementação por meio de Seleção por Substituição

- Exemplo:

Entra	1	2	3
A	*A*	*C*	*I*
L	*A*	*C*	*I*
E	*C*	*L*	*I*
R	*E*	*L*	*I*
N	*I*	*L*	*R*
	L	*N*	*R*
T	*N*	*R*	
	R	*T*	
	T		

Implementação por meio de Seleção por Substituição

- Para f pequeno não é vantajoso utilizar seleção por substituição para intercalar blocos:
  - Obtém-se o menor item fazendo f – 1 comparações.
- Quando f é 8 ou mais, o método é adequado:
  - Obtém-se o menor item fazendo $\log_2 f$ comparações.

Considerações Práticas

- As operações de entrada e saída de dados devem ser implementadas eficientemente.
- Deve-se procurar realizar a leitura, a escrita e o processamento interno dos dados de forma simultânea.
- Os computadores de maior porte possuem uma ou mais unidades independentes para processamento de entrada e saída.
- Assim, pode-se realizar processamento e operações de E/S simultaneamente.

Considerações Práticas

- Técnica para obter superposição de E/S e processamento interno:
  - Utilize 2f áreas de entrada e 2f de saída.
  - Para cada unidade de entrada ou saída, utiliza-se duas áreas de armazenamento:
    1. Uma para uso do processador central
    2. Outra para uso do processador de entrada ou saída.
  - Para entrada, o processador central usa uma das duas áreas enquanto a unidade de entrada está preenchendo a outra área.
  - Depois a utilização das áreas é invertida entre o processador de entrada e o processador central.
  - Para saída, a mesma técnica é utilizada.

Considerações Práticas

- Problemas com a técnica:
  - Apenas metade da memória disponível é utilizada.
  - Isso pode levar a uma ineficiência se o número de áreas for grande.

    Ex: Intercalação-de-f-caminhos para f grande.
  - Todas as f áreas de entrada em uma intercalação-de-f-caminhos se esvaziando aproximadamente ao mesmo tempo.

Considerações Práticas

- Solução para os problemas:
  - Técnica de previsão:
    - Requer a utilização de uma única área extra de armazenamento durante a intercalação.
    - Superpõe a entrada da próxima área que precisa ser preenchida com a parte de processamento interno do algoritmo.
    - É fácil saber qual área ficará vazia primeiro.
    - Basta olhar para o último registro de cada área.
    - A área cujo último registro é o menor, será a primeira a se esvaziar.

Considerações Práticas

- Escolha da ordem de intercalação f:
  - **Para fitas magnéticas:**
    - f deve ser igual ao número de unidades de fita disponíveis menos um.
    - A fase de intercalação usa f fitas de entrada e uma fita de saída.
    - O número de fitas de entrada deve ser no mínimo dois.
  - **Para discos magnéticos:**
    - O mesmo raciocínio acima é válido.
    - O acesso seqüencial é mais eficiente.
  - Sedegwick (1988) sugere considerar f grande o suficiente para completar a ordenação em poucos passos.
  - Porém, a melhor escolha para f depende de vários parâmetros relacionados com o sistema de computação disponível.

Intercalação Polifásica

- Problema com a intercalação balanceada de vários caminhos:
  - Necessita de um grande número de fitas.
  - Faz várias leituras e escritas entre as fitas envolvidas.
  - Para uma intercalação balanceada de f caminhos são necessárias 2f fitas.
  - Alternativamente, pode-se copiar o arquivo quase todo de uma única fita de saída para f fitas de entrada.
  - Isso reduz o número de fitas para f + 1.
  - Porém, há um custo de uma cópia adicional do arquivo.
- Solução:
  - **Intercalação polifásica.**

Intercalação Polifásica

- Os blocos ordenados são distribuídos de forma desigual entre as fitas disponíveis.
- Uma fita é deixada livre.
- Em seguida, a intercalação de blocos ordenados é executada até que uma das fitas esvazie.
- Neste ponto, uma das fitas de saída troca de papel com a fita de entrada.

Intercalação Polifásica

- Exemplo:
  - Blocos ordenados obtidos por meio de seleção por substituição:

fita 1:	*I N R T*	*A C E L*	*A A B C L O*
fita 2:	*A A C E N*	*A A D*	
fita 3:			

Intercalação Polifásica

- Exemplo:
  - Configuração após uma intercalação-de-2-caminhos das fitas 1 e 2 para a fita 3:

fita 1: *A A B C L O*

fita 2:

fita 3: *A A C E I N N R T* *A A A C D E L*

Intercalação Polifásica

- Exemplo:
  - Depois da intercalação-de-2-caminhos das fitas 1 e 3 para a fita 2:

fita 1:

fita 2: *A A A A B C C E I L N N O R T*

fita 3: *A A A C D E L*

Intercalação Polifásica

- Exemplo:
  - Finalmente:

fita 1:	*A A A A A A A B C C C D E E I L L N N O R T*
fita 2:	
fita 3:	

- A intercalação é realizada em muitas fases.
- As fases não envolvem todos os blocos.
- Nenhuma cópia direta entre fitas é realizada.

Intercalação Polifásica

- A implementação da intercalação polifásica é simples.
- A parte mais delicada está na distribuição inicial dos blocos ordenados entre as fitas.
- Distribuição dos blocos nas diversas etapas do exemplo:

fita 1	fita 2	fita 3	Total
3	2	0	5
1	0	2	3
0	1	1	2
1	0	0	1

Intercalação Polifásica

- **Análise:**
  - A análise da intercalação polifásica é complicada.
  - O que se sabe é que ela é ligeiramente melhor do que a intercalação balanceada para valores pequenos de f.
  - Para valores de f > 8, a intercalação balanceada pode ser mais rápida.

Quicksort Externo

- Foi proposto por Monard em 1980.
- Utiliza o paradigma de **divisão e conquista.**
- O algoritmo ordena ***in situ*** *um arquivo* A = {R1, ..., R_n} de n registros.
- Os registros estão armazenados consecutivamente em memória secundária de acesso randômico.
- O algoritmo utiliza somente O(log n) unidades de memória interna e não é necessária nenhuma memória externa adicional.

Quicksort Externo

- Seja R_i, 1 <= i <= n, o registro que se encontra na *i-ésima* posição de A.
- Algoritmo:

1. Particionar A da seguinte forma:

   $\{R1, .., R_i\} <= R_{i+1} <= R_{i+2} <= ... <= R_{j-2} <= R_{j-2} <= \{R_j, ..., R_n\}$

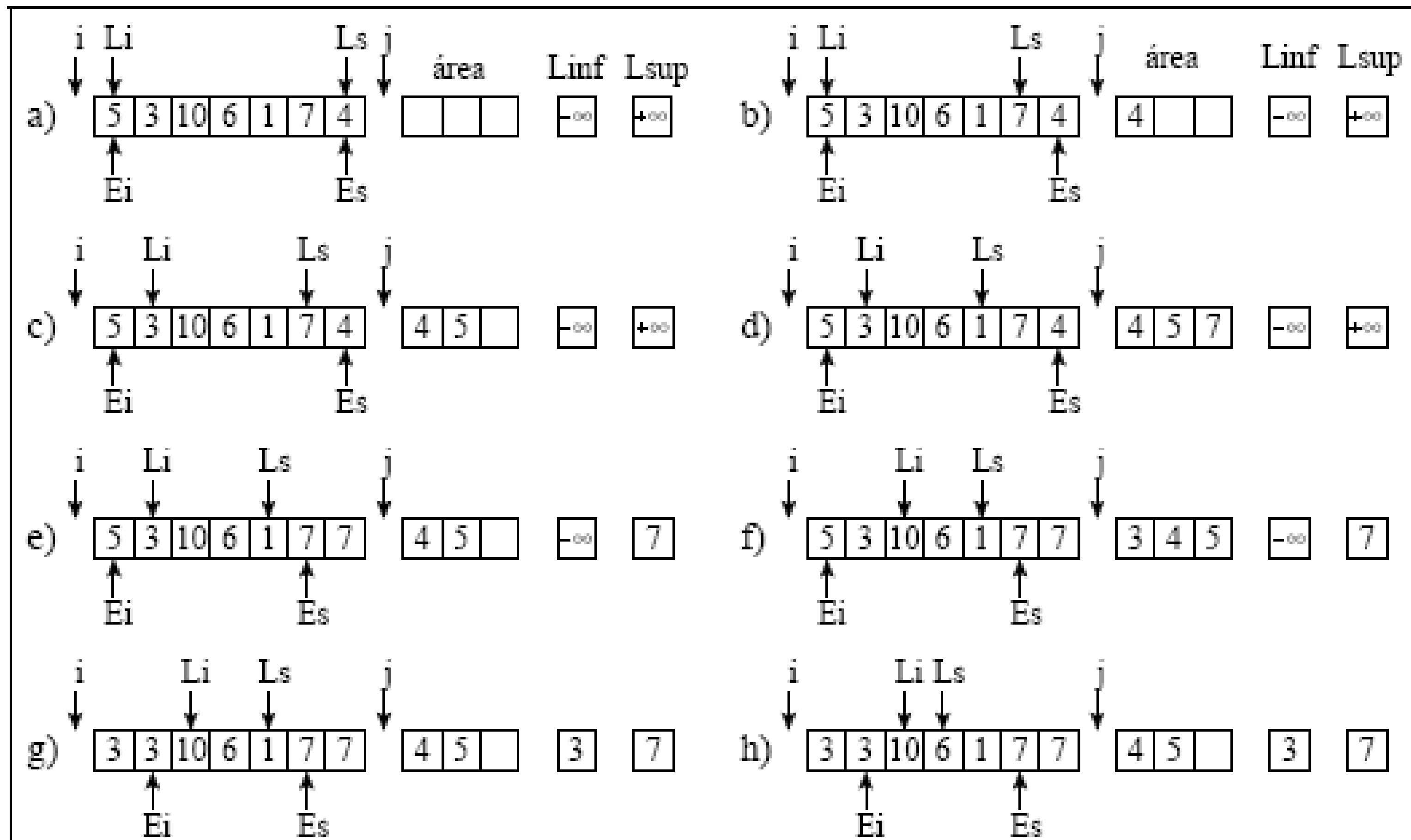
2. chamar recursivamente o algoritmo em cada um dos subarquivos

   $A1 = \{R_1, ..., R_i\}$ e $A2 = \{R_j, ..., R_n\}$.

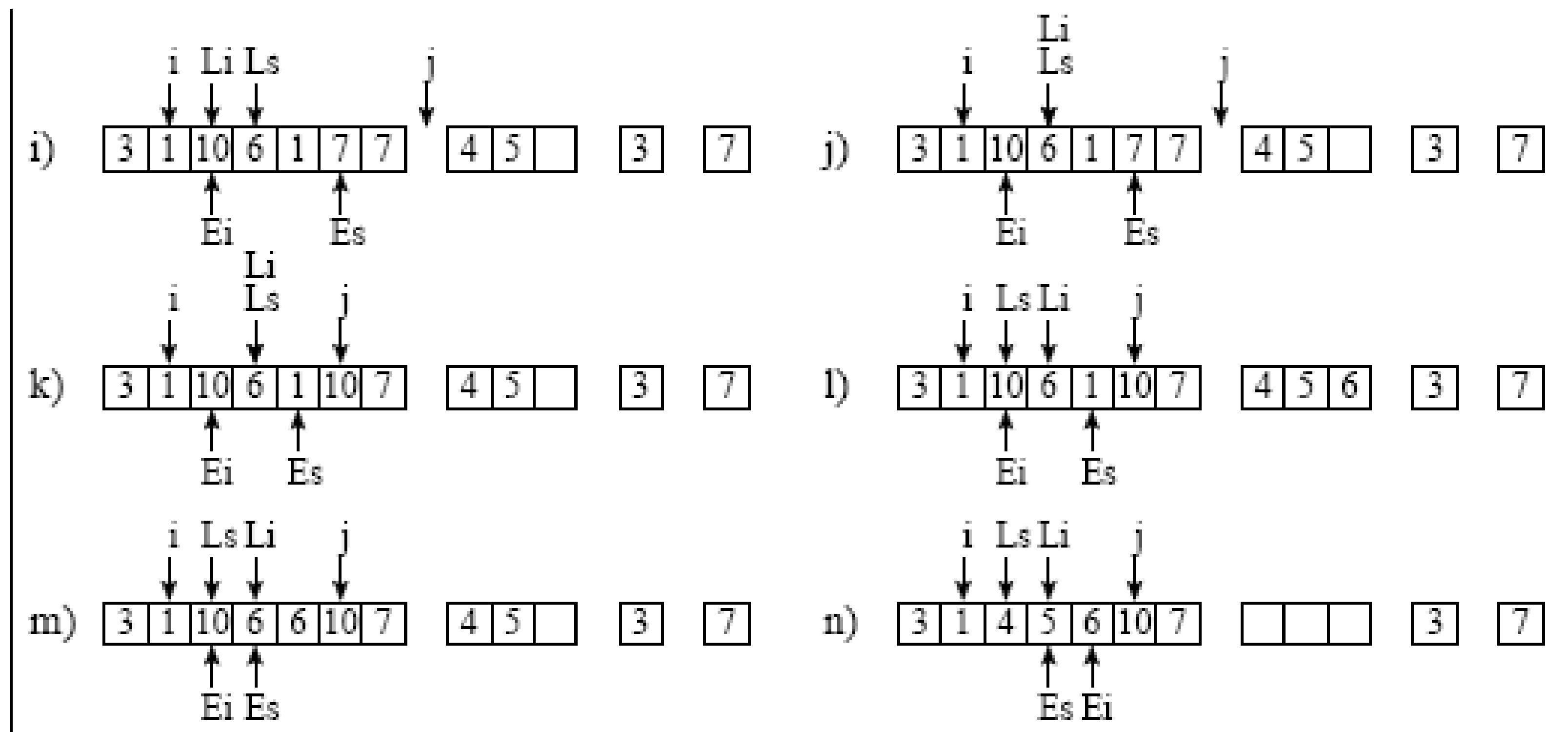
Quicksort Externo

- Para o particionamento é utilizada uma área de armazenamento na memória interna.
- Tamanho da área: TamArea = j – i – 1, com TamArea >= 3.
- Nas chamadas recusivas deve-se considerar que:
  - Primeiro deve ser ordenado o subarquivo de menor tamanho.
  - Condição para que, na média, O(log n) subarquivos tenham o processamento adiado.
  - Subarquivos vazios ou com um único registro são ignorados.
  - Caso o arquivo de entrada A possua no máximo TamArea registros, ele é ordenado em um único passo.

Quicksort Externo



Quicksort Externo



Quicksort Externo

```
void QuicksortExterno(FILE **ArqLi, FILE **ArqEi,
   FILE **ArqLEs, int Esq, int Dir)
{ int i, j;
  TipoArea Area; /* Area de armazenamento interna */
  if (Dir - Esq < 1) return;
  FAVazia(&Area);
  Particao(ArqLi, ArqEi, ArqLEs, Area, Esq, Dir, &i,
   &j);
  if (i - Esq < Dir - j)
  { /* ordene primeiro o subarquivo menor */
    QuicksortExterno(ArqLi, ArqEi, ArqLEs, Esq, i);
    QuicksortExterno(ArqLi, ArqEi, ArqLEs, j, Dir);
  }
  else
  { QuicksortExterno(ArqLi, ArqEi, ArqLEs, j, Dir);
    QuicksortExterno(ArqLi, ArqEi, ArqLEs, Esq, i);
  }
}
```

Quicksort Externo

- Procedimentos auxiliares utilizados pelo procedimento Particao:

```
void LeSup(FILE **ArqLEs, TipoRegistro *UltLido, int *Ls, short
   *OndeLer)
{ fseek(*ArqLEs, (*Ls - 1) * sizeof(TipoRegistro), SEEK_SET);
  fread(UltLido, sizeof(TipoRegistro), 1, *ArqLEs);
  (*Ls)--; *OndeLer = FALSE;
}

void LeInf(FILE **ArqLi, TipoRegistro *UltLido, int *Li, short
   *OndeLer)
{ fread(UltLido, sizeof(TipoRegistro), 1, *ArqLi);
  (*Li)++; *OndeLer = TRUE;
}

void InserirArea(TipoArea *Area, TipoRegistro *UltLido, int *NRArea)
{ /* Insere UltLido de forma ordenada na Area */
  InsereItem(*UltLido, Area); *NRArea = ObterNumCelOcupadas(Area);
}
```

Quicksort Externo

```
void EscreveMax(FILE **ArqLEs, TipoRegistro R, int *Es)
{ fseek(*ArqLEs, (*Es - 1) * sizeof(TipoRegistro), SEEK_SET);
  fwrite(&R, sizeof(TipoRegistro), 1, *ArqLEs); (*Es)--;
}

void EscreveMin(FILE **ArqEi, TipoRegistro R, int *Ei)
{ fwrite(&R, sizeof(TipoRegistro), 1, *ArqEi); (*Ei)++; }

void RetiraMax(TipoArea *Area, TipoRegistro *R, int *NRArea)
{ RetiraUltimo(Area, R);
  *NRArea = ObterNumCelOcupadas(Area);
}

void RetiraMin(TipoArea *Area, TipoRegistro *R, int *NRArea)
{ RetiraPrimeiro(Area, R);
  *NRArea = ObterNumCelOcupadas(Area);
}
```

Quicksort Externo

- Procedimento Partição

```
void Particao(FILE **ArqLi, FILE **ArqEi, FILE **ArqLEs,
   TipoArea Area, int Esq, int Dir, int *i, int *j)
{ int Ls = Dir, Es = Dir, Li = Esq, Ei = Esq, NRArea = 0,
   Linf = INT_MIN, Lsup = INT_MAX;
  short OndeLer = TRUE; TipoRegistro UltLido, R;
  fseek(*ArqLi, (Li - 1)* sizeof(TipoRegistro), SEEK_SET);
  fseek(*ArqEi, (Li - 1)* sizeof(TipoRegistro), SEEK_SET);
  *i = Esq - 1; *j = Dir + 1;
  while (Ls >= Li)
  { if (NRArea < TamArea - 1)
    { if (OndeLer)
        LeSup(ArqLEs, &UltLido, &Ls, &OndeLer);
      else LeInf(ArqLi, &UltLido, &Li, &OndeLer);
      InserirArea(&Area, &UltLido, &NRArea);
      continue;
    }
```

Quicksort Externo

- Procedimento Partição (cont.)

```
if (Ls == Es)
  LeSup(ArqLEs, &UltLido, &Ls, &OndeLer);
else if (Li == Ei) LeInf(ArqLi, &UltLido, &Li, &OndeLer);
  else
    if (OndeLer)
      LeSup(ArqLEs, &UltLido, &Ls, &OndeLer);
    else LeInf(ArqLi, &UltLido, &Li, &OndeLer);
if (UltLido.Chave > Lsup)
{ *j = Es;
  EscreveMax(ArqLEs, UltLido, &Es);
  continue;
}
if (UltLido.Chave < Linf)
{ *i = Ei;
  EscreveMin(ArqEi, UltLido, &Ei);
  continue;
}
```

Quicksort Externo

- Procedimento Partição (cont.)

```
      InserirArea(&Area, &UltLido, &NRArea);
      if (Ei - Esq < Dir - Es)
      { RetiraMin(&Area, &R, &NRArea);
        EscreveMin(ArqEi, R, &Ei);
        Linf = R.Chave;
      }
      else
      { RetiraMax(&Area, &R, &NRArea);
        EscreveMax(ArqLEs, R, &Es);
        Lsup = R.Chave;
      }
    }
    while (Ei <= Es)
    { RetiraMin(&Area, &R, &NRArea);
      EscreveMin(ArqEi, R, &Ei);
    }
  }
```

Quicksort Externo

- Programa Teste:

- (FALTANDO)

Quicksort Externo

- **Análise:**
  - Seja n o número de registros a serem ordenados.
  - Seja e b o tamanho do bloco de leitura ou gravação do Sistema operacional.
  - Melhor caso: O(n/b)
    - Por exemplo, ocorre quando o arquivo de entrada já está ordenado.
  - Pior caso: O(n^2/TamArea)
    - ocorre quando um dos arquivos retornados pelo procedimento Particao tem o maior tamanho possível e o outro é vazio.
    - A medida que n cresce, a probabilidade de ocorrência do pior caso tende a zero.
  - Caso Médio: O(n/b log(n/TamArea))
    - É o que tem a maior probabilidade de ocorrer.