

Índice

1.	PROGRAMAÇÃO BÁSICA EM PASCAL	1-1
1.1.	Tipos primitivos de dados.....	1-1
1.2.	Variáveis	1-1
1.3.	Comandos	1-1
1.3.1.	Operadores	1-1
1.3.2.	Comandos básicos	1-1
1.3.3.	Estruturas de controle.....	1-1
2.	MODULARIZAÇÃO.....	2-1
2.1.	Procedure	2-1
2.1.1.	Tipos de parâmetros	2-1
2.1.2.	Passagem de parâmetros.....	2-1
2.1.3.	Tipos de variáveis.....	2-1
2.2.	Function	2-2
2.3.	Recursividade.....	2-2
3.	ESTRUTURAS DE DADOS BÁSICAS.....	3-1
3.1.	Matrizes	3-1
3.1.1.	Matrizes Unidimensionais (vetores)	3-1
3.1.2.	Matrizes Multidimensionais	3-1
3.2.	Registros	3-1
3.3.	Tipos definidos pelo usuário.....	3-2
3.4.	Ponteiros (Referência)	3-2
3.4.1.	Variáveis estáticas	3-2
3.4.2.	Variáveis dinâmicas	3-2
3.4.3.	Ponteiro ou apontador.....	3-2
3.4.4.	Constante NIL.....	3-2
3.4.5.	Geração: procedimento new()	3-3
3.4.6.	Remoção: procedimento dispose()	3-3
3.4.7.	Exemplos	3-3
4.	ORGANIZAÇÕES BÁSICAS DE ARQUIVOS.....	4-1
4.1.	Conceitos.....	4-1
4.2.	Estruturas de Arquivos	4-1
4.2.1.	Arquivo seqüencial	4-1
4.2.2.	Arquivo seqüencial-indexado.....	4-2
4.2.3.	Arquivo indexado	4-3
4.2.4.	Arquivo direto	4-4
4.3.	Arquivos em Pascal	4-4
4.3.1.	Tipo Text.....	4-4
4.3.2.	Tipo File	4-4
4.3.3.	Rotinas de arquivos	4-4
4.3.4.	Exemplos de programas	4-6
5.	MÉTODOS DE CLASSIFICAÇÃO	5-1
5.1.	Conceitos.....	5-1
5.2.	Classificação com Inserção Direta.....	5-1
5.3.	Classificação por Seleção	5-2
5.4.	Classificação por Troca (BubbleSort).....	5-2
5.5.	Classificação Rápida por troca (QuickSort).....	5-3
5.6.	Intercalação	5-3

6.	MÉTODOS DE PESQUISA	6-1
6.1.	Conceitos.....	6-1
6.2.	Pesquisa seqüencial	6-1
6.3.	Pesquisa binária.....	6-2

1. PROGRAMAÇÃO BÁSICA EM PASCAL

1.1. Tipos primitivos de dados

- **Integer:** números inteiros (-32768 a 32767)
- **Real:** números reais
- **Char:** caracteres (geralmente tabela ASCII)
- **String:** conjunto de caracteres
- **Boolean:** valores lógicos (True/False)

1.2. Variáveis

São objetos que armazenam em memória o valor dos dados, durante a execução do programa. Possuem apenas um valor, em dado instante, que pode ser alterado pelos comandos. São identificadas por um *nome* e devem ser declaradas antes do início do programa.

```
Var
    Nota, Media : real;
    Indice : integer;
```

1.3. Comandos

1.3.1. Operadores

- **Aritméticos:** + - / *
- **Relacionais:** = <> > < >= <=
- **Lógicos:** and or not

1.3.2. Comandos básicos

- **Atribuição:** usado para armazenar um valor em uma variável de memória

```
Media := Soma /30;
```

- **Entrada de dados:** usado para armazenar em uma variável o valor informado pelo usuário a partir da unidade de entrada

```
read|readln(<lista de variáveis>);
```

- **Saída de Dados:** usado para mostrar um valor na unidade de saída

```
write|writeln(<lista de objetos>);
```

1.3.3. Estruturas de controle

Seqüência

Indica a execução de um bloco de comandos. Os blocos de comandos são delimitados por **begin** e **end**. Caso o bloco seja constituído por apenas um comando, os delimitadores são opcionais.

```
begin
    <comandos>;
end;
```

Decisão

Usada na execução condicional de um bloco de comandos. Uma expressão lógica é avaliada, retornando um valor **true** ou **false**. Com base neste resultado é definido qual bloco será executado.

- Executa opcionalmente um bloco dependendo do valor de *condição*.

```
if <condicao>
  then <bloco>;
```

- Executa o *bloco-1* se a *condição* for **true** ou o *bloco-2* se a *condição* for **false**.

```
if <condicao>
  then <bloco-1>
  else <bloco-2>;
```

- Executa um *bloco* dependendo do valor da *variável* (que deve ser um tipo escalar). Caso o valor da *variável* não coincida com nenhum dos valores listados, é executado o *bloco* em *else*.

```
case <variável> of
  <valor1> : <bloco1>;
  <valor2> : <bloco2>;
  <valor3> : <bloco3>;
  ...
  <valor-n> : <bloco-n>;
  ...
  else <bloco>;
end;
```

Repetição

Usada quando os comandos de um bloco devem ser executados repetidamente, baseado em uma determinada condição ou variável de controle.

- **Teste no início do loop:** o *bloco* é executado enquanto a *condição* permanecer **true**. Caso seja inicialmente **false**, o *bloco* não é executado nenhuma vez.

```
while <condicao> do
  <bloco>;
```

- **Teste no fim do loop:** o *bloco* é executado até que a *condição* avaliada se torne **true**. O *bloco* é executado pelo menos uma vez, antes da *condição* ser avaliada.

```
repeat
  <comandos>;
until <condicao>;
```

- **Repetição controlada:** a quantidade de vezes que o bloco será repetido é controlada pela *var_controle*, que varia de um valor inicial a um valor final, em incrementos de 1.

```
for <var_controle> := <val_inicial> to <val_final> do <bloco>;
```

2. MODULARIZAÇÃO

Modularização é o processo de quebrar a solução de um problema complexo em vários subprogramas mais simples, a serem executados a partir de um *programa principal*.

O processo de criação de módulos é fundamental na tarefa de programação por permitir a reutilização, ocultar a complexidade e facilitar a manutenção do programa.

2.1. Procedure

Procedures (procedimentos) são blocos de comandos que possuem um cabeçalho, podendo ser referenciados pelo seu nome. Podem receber parâmetros, especificados no cabeçalho.

2.1.1. Tipos de parâmetros

- **Parâmetros formais:** são as variáveis declaradas no cabeçalho da *procedure*, cuja função é receber os valores que serão passados através da chamada à *procedure* pelo programa principal ou por outra *procedure*.
- **Parâmetros efetivos ou atuais:** são os valores reais, informados pelo programa principal na chamada à *procedure*.

2.1.2. Passagem de parâmetros

- **Por valor:** os valores dos parâmetros efetivos são copiados para os parâmetros formais; as alterações nos parâmetros formais não são refletidas nos parâmetros efetivos; os parâmetros efetivos podem ser constantes, expressões ou variáveis.
- **Por referência:** toda alteração nos parâmetros formais é refletida nos parâmetros efetivos; os parâmetros efetivos devem ser variáveis; declarados com a palavra reservada **var** no cabeçalho da *procedure*.

2.1.3. Tipos de variáveis

- **Variáveis globais:** variáveis declaradas no programa principal, cujo escopo é todo o programa.
- **Variáveis locais:** variáveis declaradas dentro de uma *procedure*, só existindo enquanto a *procedure* estiver sendo executado (escopo local).

Ex 1:

```
Program Troca_Valor;
Var
    A, B, Temp : integer;

Procedure Troca;
Begin
    Temp := A;
    A := B;
    B := Temp;
End;

Begin
    Readln(A, B);
    Writeln(A, B);
    Troca;
    Writeln(A, B);
End.
```

Ex 2:

```
Program Troca_Valor;
Var
    A, B : integer;

Procedure Troca( var X, Y : integer);
Var
    Temp : integer;
Begin
    Temp := X;
    X := Y;
    Y := Temp;
End;

Begin
    Readln(A, B);
    Writeln(A, B);
    Troca(A,B);
    Writeln(A, B);
End.
```

2.2. Function

Uma *function* (função) é um tipo de procedimento que deve retornar um valor ou resultado para o programa principal. Deve-se declarar o *tipo da função*, que é o tipo do resultado que ela retorna.

Ex: Contar o número de vogais em um nome

```
Program Conta_Vogais;
Var
  Nome : string;
  N : integer;

Function Num_Vogais( S : string) : integer;
Var
  N, Cont, i : integer;
  Letra : char;
Begin
  S := upper(S);
  N := length(S);
  Cont := 0;
  For i := 1 to N do
  Begin
    Letra := S[i];
    Case Letra of
      'A','E','I','O','U' : cont := cont + 1;
    end;
  End;
  Num_Vogais := cont;
End;

Begin
  Readln(Nome);
  N := num_vogais(Nome);
  Writeln(Nome, ` tem `, n, ` vogais`);
End.
```

2.3. Recursividade

Uma *procedure* ou *function* que contém no seu bloco de comandos uma chamada a ela própria é chamada de *procedimento recursivo*. A recursividade é um poderoso instrumento de programação que permite a definição de estruturas infinitas com um conjunto finito de sentenças.

Um exemplo comum de uma definição recursiva é o da função fatorial de "n". A função fatorial é denotada matematicamente por n! e expressa o valor do produto de todos os números inteiros até (e inclusive) "n". Assim, o fatorial de "n" é definido como:

$$\left\{ \begin{array}{l} n! = 1, \text{ se } n = 0 \\ n * (n - 1)!, \text{ se } n > 0 \end{array} \right.$$

Esta função pode ser escrita facilmente utilizando-se a recursividade:

```
function Fatorial (N : integer) : integer;
begin
  if (N > 0)
  then Fatorial := N * Fatorial(N - 1)
  else Fatorial := 1;
end;
```

Embora não seja usada nenhuma estrutura de repetição, está sendo realizado um produto de "n" termos.

3. ESTRUTURAS DE DADOS BÁSICAS

3.1. Matrizes

As matrizes (também denominadas *variáveis compostas homogêneas*, *arranjos*, *arrays* ou "variáveis indexadas") correspondem a posições contíguas de memória, identificadas por um mesmo nome (uma mesma variável).

Nas matrizes, cada posição é referenciada por um ou mais índices. Todos os componentes da estrutura devem ser de um mesmo tipo, chamado *tipo-base*.

3.1.1. Matrizes Unidimensionais (vetores)

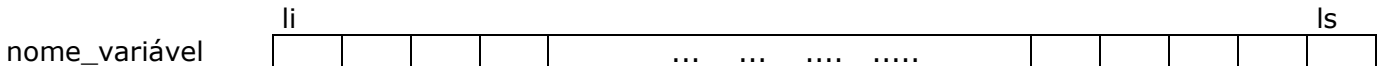
Os vetores correspondem a posições seqüenciais na memória, onde cada posição armazena um valor do *tipo-base*, e é acessada através do índice.

Declaração:

```
var
  <nome_variável> : array [li..ls] of <tipo_base>;
```

li : número inteiro indicando o limite inferior (índice inferior)

ls : número inteiro indicando o limite superior (índice superior)



3.1.2. Matrizes Multidimensionais

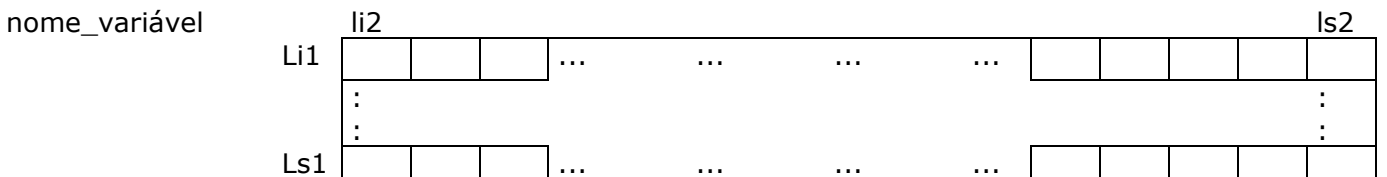
Declaração:

```
var
  <nome_variável> : array[li1..ls1,li2..ls2,...,lin..lsn] of <tipo_base>;
```

li : número inteiro indicando o limite inferior (índice inferior) da dimensao

ls : número inteiro indicando o limite superior (índice superior) da dimensao

Ex: Matriz bidimensional



3.2. Registros

São conjuntos de dados, que têm alguma relação lógica entre si, mas que são de tipos diferentes (inteiro, real, caracter, string, lógico). São também denominados *variáveis compostas heterogêneas*. Os registros são constituídos por componentes chamados **campos**, onde cada campo tem um tipo.

Declaração:

```
var
  <nome_variável> : record
    <campo_1> : <tipo_1>;
    <campo_2> : <tipo_2>;
    :
    <campo_n> : <tipo_n>;
  end;
```

3.3. Tipos definidos pelo usuário

Em Pascal é possível a criação de novos tipos de dados, baseados nos tipos primitivos da linguagem. Isto é feito através da declaração **type**.

```
type
  reg_func = record
    numero : integer;
    nome : string;
    salario : real;
  end;
  tipo_vetor = array[1..100] of integer;

var
  Func : reg_func;
  Vetor : tipo_vetor;
```

3.4. Ponteiros (Referência)

3.4.1. Variáveis estáticas

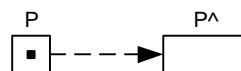
- São definidas em tempo de compilação com um *tamanho* fixo
- Possuem um *nome* que indica uma posição na memória
- Possuem um *valor* que é o conteúdo armazenado nesta posição de memória

3.4.2. Variáveis dinâmicas

- Área de memória que não tem um *nome* (identificador) próprio
- São referenciadas e acessadas através de outra variável
- A memória necessária para armazenar o valor é alocada durante a execução do programa

3.4.3. Ponteiro ou apontador

O *pointer* (ponteiro ou apontador) é uma variável cujo valor é o endereço de uma posição de memória.



Declaração:

```
var
  P : ^T;
```

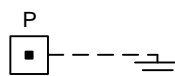
Onde P é um ponteiro para o tipo T

Ex:

```
pont : ^integer;
```

3.4.4. Constante NIL

Constante pré-definida que indica que o ponteiro não aponta para objeto algum. Um valor NIL é diferente de um *valor indefinido*.



```
P := NIL;
```

3.4.5. Geração: procedimento new()

Aloca a memória necessária para armazenar um valor do tipo indicado pelo ponteiro. Antes da alocação o ponteiro é considerado *indefinido*.

```
new(P);
```

3.4.6. Remoção: procedimento dispose()

Libera a memória alocada ao ponteiro por **new()**, tornando o ponteiro indefinido.

```
dispose(P);
```

3.4.7. Exemplos

```
program Exemplo1;
var
  P : ^integer;
begin
  new(p);
  P^ := 10;
  writeln( p^ );
  dispose(P);
end.
```

```
program Exemplo2;
var
  P : ^integer;
  A : integer;
begin
  A := 10;
  new(p);
  P^ := A;
  writeln( p^ );
  dispose(P);
end.
```

```
program Exemplo3;
type
  pReg = ^reg;
  reg = record
    numero: integer;
    nome: string;
  end;
var
  P : pReg;
  R : reg;
  A : integer;
  S : string;
begin
  A := 10;
  S := 'Smith';
  new(P);
  P^.numero := A;
  P^.nome := S;
  R := P^;
  writeln(R.numero);
  writeln(R.nome);
  dispose(P);
end.
```

4. ORGANIZAÇÕES BÁSICAS DE ARQUIVOS

4.1. Conceitos

- **Arquivo:** coleção de registros lógicos, cada um deles representando um objeto ou entidade
- **Registro lógico (registro) :** seqüência de itens, cada item sendo chamado de campo ou atributo, correspondendo a uma característica do objeto representado. Os registros podem ser de tamanho fixo ou de tamanho variável.
- **Campo:** item de dados do registro, com um nome e um tipo associados
- **Bloco:** unidade de armazenamento do arquivo em disco, também denominado registro físico. Um registro físico normalmente é composto por vários registros lógicos. Cada bloco armazena um número inteiro de registros.
- **Chave:** é uma seqüência de um ou mais campos em um arquivo
- **Chave primária:** é uma chave que apresenta um valor diferente para cada registro do arquivo. É usada para identificar, de forma única, cada registro.
- **Chave secundária:** é uma chave que pode possuir o mesmo valor em registro distintos. É normalmente usada para identificar um conjunto de registros.
- **Chave de acesso:** é uma chave usada para identificar o(s) registro(s) desejado(s) em uma operação de acesso ao arquivo.

4.2. Estruturas de Arquivos

4.2.1. Arquivo seqüencial

Nos arquivos seqüenciais a ordem lógica e física dos registros armazenados é a mesma. Os registros podem estar dispostos seguindo a seqüência determinada por uma chave primária (chamada chave de ordenação), ou podem estar dispostos aleatoriamente.

#	Numero	Nome	Idade	Salario
0	1000	ADEMAR	25	600
1	1050	AFONSO	27	700
2	1075	CARLOS	28	500
3	1100	CESAR	30	1000
4	1150	DARCI	23	1500
5	1180	EBER	22	2000
6	1250	ENIO	27	750
7	1270	FLAVIO	28	600
8	1300	IVAN	30	700
9	1325	MIGUEL	34	1000
10	1340	MARIA	35	1500
11	1360	RAMON	32	2000
12	1400	SANDRA	29	700
13	1450	TATIANA	30	500

Acesso a um registro

Podemos considerar dois tipos de acesso: seqüencial ou aleatório.

O **acesso seqüencial** consiste em acessar os registros na ordem em que estão armazenados, ou seja, o registro obtido é sempre o posterior ao último acessado. Como os registros são armazenados em sucessão contínua, acessar o registro "n" de um arquivo requer a leitura dos "n-1" registros anteriores.

O **acesso aleatório** se caracteriza pela utilização de um "argumento de pesquisa" (chave de acesso), que indica qual o registro desejado. Neste caso, a ordem em que os registros são acessados pode ser diferente da ordem em que eles estão armazenados fisicamente. Se o arquivo

está ordenado e a chave de acesso coincide com a chave de ordenação, podemos utilizar a pesquisa binária. Caso contrário, deve ser realizada uma pesquisa seqüencial no arquivo.

Inserção de um registro

Se o arquivo não está ordenado, o registro pode ser simplesmente inserido após o último registro armazenado.

Se o arquivo está ordenado, normalmente é adotado o seguinte procedimento:

Dado um *arquivo base* B, é construído um *arquivo de transações* T, que contem os registros a serem inseridos, ordenado pela mesma chave que o arquivo B. Os arquivos B e T são então intercalados, gerando o arquivo A, que é a versão atualizada de B.

Arquivo B			
#	Num	Nome	Idade
0	1000	ADEMAR	25
1	1050	AFONSO	27
2	1075	CARLOS	28
3	1100	CESAR	30
4	1150	DARCI	23
5	1180	EBER	22
6	1250	ENIO	27
7	1270	FLAVIO	28
8	1300	IVAN	30
9	1325	MIGUEL	34
10	1340	MARIA	35
11	1360	RAMON	32
12	1400	SANDRA	29
13	1450	TATIANA	30

Arquivo T			
#	Num	Nome	Idade
0	1070	ANGELA	25
1	1120	CLAUDIA	27
2	1280	IARA	28
3	1310	LUIS	30
4	1420	SONIA	23

Arquivo A			
#	Num	Nome	Idade
0	1000	ADEMAR	25
1	1050	AFONSO	27
2	1070	ANGELA	25
3	1075	CARLOS	28
4	1100	CESAR	30
5	1120	CLAUDIA	27
6	1150	DARCI	23
7	1180	EBER	22
8	1250	ENIO	27
9	1270	FLAVIO	28
10	1280	IARA	28
11	1300	IVAN	30
12	1310	LUIS	30
13	1325	MIGUEL	34
14	1340	MARIA	35
15	1360	RAMON	32
16	1400	SANDRA	29
17	1420	SONIA	23
18	1450	TATIANA	30

Exclusão de um registro

Normalmente é implementada como a inserção, com a criação de um arquivo de transações que contém os registros a serem excluídos, que é processado posteriormente.

Pode ainda ser implementada através de um campo adicional no arquivo que indique o estado (*status*) de cada registro. Na exclusão, o valor deste campo seria alterado para "excluído". Posteriormente, é feita a leitura seqüencial de todos os registros, sendo que os registros que não estiverem marcados como "excluídos" são copiados para um novo arquivo.

Alteração de um registro

Consiste na modificação do valor de um ou mais atributos de um registro. O registro deve ser localizado, lido e os campos alterados, sendo gravado novamente, na mesma posição.

A alteração é feita sem problemas, desde que ela não altere o tamanho do registro nem modifique o valor de um campo usado como chave de ordenação.

4.2.2. Arquivo seqüencial-indexado

Quando o volume de acessos aleatórios em um arquivo seqüencial torna-se muito grande, é necessário utilizar uma estrutura de acesso que ofereça maior eficiência na localização de um registro com base em uma chave de acesso.

O arquivo seqüencial-indexado é um arquivo seqüencial acrescido de uma estrutura de acesso (*índice*). Um **índice** é formado por uma coleção de pares, associando um valor da chave de acesso a um endereço de registro. Deve existir um índice específico para cada chave de acesso.

Índice Primário		
#	Num	End.
0	1300	0
1	1605	3
2	**	6

** Maior valor que a chave pode assumir

Índice Secundário		
#	Num	End.
0	1070	0
1	1200	3
2	1300	6
3	1430	9
4	1520	12
5	1605	15
6	1710	18
7	1745	21
8	**	24

** Maior valor que a chave pode assumir

Arquivo			
#	Num	Nome	Idade
0	1000	ADEMAR	25
1	1050	AFONSO	27
2	1070	ANGELA	25
3	1075	CARLOS	28
4	1100	CESAR	30
5	1200	CLAUDIA	25
6	1250	CRISTIE	26
7	1275	DARCI	29
8	1300	DIOGO	25
9	1310	ELBER	27
10	1400	EDISON	25
11	1430	EDMUNDO	28
12	1470	ENIO	30
13	1510	FLAVIO	25
14	1520	GENARO	26
15	1530	GERSON	29
16	1590	HELENA	25
17	1605	IARA	27
18	1650	IVAN	25
19	1700	LUIS	28
20	1710	MARIA	30
21	1730	MIGUEL	25
22	1740	RAMON	26
23	1745	SANDRA	29
24	1800	SONIA	32
25	1905	TATIANA	34
26	2010	WILSON	20

4.2.3. Arquivo indexado

O arquivo indexado é aquele em que os registros são acessados através de um ou mais índices, não havendo qualquer compromisso com a ordem em que os registros estão armazenados.

Podem existir tantos índices quantas forem as chaves de acesso aos registros. As entradas no índice são ordenadas pelo valor das chaves de acesso, sendo cada uma delas constituída por um par (chave do registro, endereço do registro).

Índice	
Num	End.
1000	1
1070	2
1075	12
1200	3
1250	8
1310	10
1400	4
1470	16
1510	5
1520	9
1530	14
1590	6
1605	11
1650	7
1700	13
1710	17
1745	15
1800	18
1905	19
2010	0

Arquivo				
#	Num	Nome	Idade	Salário
0	2010	WILSON	26	1000
1	1000	ADEMAR	32	250
2	1070	ANGELA	28	300
3	1200	CLAUDIA	25	750
4	1400	EDISON	22	1500
5	1510	FLAVIO	30	250
6	1590	HELENA	26	300
7	1650	IVAN	32	750
8	1250	CRISTIE	25	1500
9	1520	GENARO	24	750
10	1310	ELBER	26	250
11	1605	IARA	32	300
12	1075	CARLOS	28	750
13	1700	LUIS	32	1500
14	1530	GERSON	26	400
15	1745	SANDRA	32	400
16	1470	ENIO	25	750
17	1710	MARIA	24	400
18	1800	SONIA	22	750
19	1905	TATIANA	30	400

4.2.4. Arquivo direto

A idéia básica de um arquivo direto é o armazenamento dos registros em endereços determinados com base no valor de uma chave primária, de modo que se tenha acesso rápido aos registros especificados por argumentos de pesquisa, sem que haja necessidade de percorrer uma estrutura de índice.

Em um arquivo direto ao invés de um índice é usada uma função que calcula um endereço de registro a partir do argumento de pesquisa.

Arquivo				
#	Num	Nome	Idade	Salário
0	2010	WILSON	26	1000
1				
2	1070	ANGELA	28	300
3	1200	CLAUDIA	25	750
4	1300	DIOGO	24	400
5				
6				
7				
8	1650	IVAN	32	750
9	1730	MIGUEL	28	400
10	1250	CRISTIE	25	1500
11				
12				
13	1050	AFONSO	30	1500
14				
15	1800	SONIA	22	750
...				

C='CLAUDIA' → E=F(C) → E=3

O principal problema associado com os arquivos diretos é o da determinação de uma função F , que transforme o valor C da chave de um registro no endereço E , que lhe corresponde no arquivo.

Geralmente são usadas "funções probabilísticas" que geram, para cada valor da chave, um endereço "tão único quanto possível", podendo gerar, para valores distintos da chave, o mesmo endereço. Este fato é denominado "colisão", e devem ser estabelecidos procedimentos para tratá-lo.

4.3. Arquivos em Pascal

4.3.1. Tipo Text

Os arquivos do tipo **Text** (*text files*) são utilizados para armazenamento de *strings* em disco. São arquivos que só permitem o acesso seqüencial, sendo que as inclusões são feitas sempre no final do arquivo (*append*).

4.3.2. Tipo File

Os arquivos do tipo **File** (*typed files*) são utilizados para armazenamento de estruturas de dados genéricas. São também denominados "*file of records*" ou arquivos de registros, já que usualmente a estrutura a ser armazenada é do tipo "*record*" (registro). Estão enquadrados na organização seqüencial, porém permitem acesso aleatório, através do número do registro.

4.3.3. Rotinas de arquivos

`assign(var_arq, nome_arq)`

Procedure que associa o nome do arquivo em disco (*nome_arq*) à variável *var_arq* (do tipo Text ou File) declarada no programa. Deve ser a primeira rotina a ser usada na manipulação de arquivos em disco.

rewrite(var_arq)

Procedure que cria e abre um novo arquivo em disco, associado à variável *var_arq*. Arquivos do tipo Text são abertos apenas para escrita. Arquivos do tipo File são posicionados no registro #0.

reset(var_arq)

Procedure que abre um arquivo já existente, associado à variável *var_arq*. Arquivos do tipo Text são abertos apenas para leitura. Arquivos do tipo File são posicionados no registro #0.

append(var_arqtxt)

Procedure utilizada com arquivos do tipo Text. Abre um arquivo já existente, posicionando o ponteiro no final do arquivo. O arquivo está aberto apenas para escrita.

close(var_arq)

Procedure que fecha um arquivo externo, gravando antes as informações do buffer.

eof(var_arq)

Função lógica que retorna *True* se o ponteiro estiver no fim do arquivo.

eoln(var_arq)

Função lógica que retorna *True* se o ponteiro estiver no fim de uma linha (CR+LF) ou no fim do arquivo.

erase(var_arq)

Procedure que remove um arquivo do disco.

FilePos(var_arq)

Função inteira que retorna a posição do ponteiro no arquivo associado a *var_arq*. O primeiro registro está na posição #0. Não é válida para arquivos do tipo Text.

FileSize(var_arq)

Função inteira que retorna a posição do último registro no arquivo associado a *var_arq*. O número de registros no arquivo é igual a FileSize+1. Não é válida para arquivos do tipo Text.

flush(var_arqtxt)

Procedure que força a gravação dos dados contidos no buffer de um arquivo do tipo Text.

seek(var_arq, num)

Procedure que posiciona o ponteiro do arquivo *var_arq* no registro indicado por *num*. O primeiro registro está na posição #0. Não é válida para arquivos do tipo Text.

read(var_arq, var)

Procedure utilizada para leitura de dados a partir do arquivo indicado por *var_arq*. Os dados lidos são armazenados na variável *var* (geralmente do tipo *record*). Se *var_arq* não for indicada, os dados são lidos a partir do dispositivo padrão de entrada (teclado). Após a leitura o ponteiro do arquivo avança para a posição seguinte.

readLn(var_arq, var)

Semelhante a Read, porém os dados são lidos até ser encontrada uma seqüência de fim-de-linha (CR+LF).

write(var_arq, var)

Procedure utilizada para escrever os dados armazenados na variável *var* (geralmente do tipo *record*) no arquivo indicado por *var_arq*. Os dados são escritos na posição atual do arquivo. Se *var_arq* não for indicada, os dados são escritos no dispositivo padrão de saída (monitor de vídeo). Após a leitura o ponteiro do arquivo avança para a posição seguinte.

writeLn(var_arq, var)

Semelhante a Write, porém é escrita uma seqüência de fim-de-linha (CR+LF) após escrever a variável.

4.3.4. Exemplos de programas

Text Files

- **Criação de um arquivo Texto:**

```
program Criar_Txt;
var
  ArqTxt : Text;
  S : string;
  i : integer;
begin
  assign(ArqTxt, 'Text01.txt');
  rewrite(ArqTxt);
  S := 'Linha de tamanho variavel';
  for i := 1 to 13 do begin
    writeln(ArqTxt, 'Linha #', i, ' ', copy(S,1,i*2));
  end;
  close(ArqTxt);
end.
```

- **Leitura de um arquivo Texto:**

```
program Ler_Txt;
var
  ArqTxt : Text;
  S : string;
begin
  assign(ArqTxt, 'Text01.txt');
  reset(ArqTxt);
  while not eof(ArqTxt) do begin
    readln(ArqTxt, S);
    writeln(S);
  end;
  close(ArqTxt);
end.
```

- **Inserção de linhas em um arquivo Texto:**

```
program Inserir_Txt;
var
  ArqTxt : Text;
  S : string;
  i : integer;
begin
  assign(ArqTxt, 'Text01.txt');
  append(ArqTxt);
  S := 'Linha acrescentada #';
  for i := 1 to 5 do begin
    writeln(ArqTxt, S, i);
  end;
  close(ArqTxt);
end.
```

Typed Files

▪ Criação de um arquivo de dados a partir de um arquivo texto:

```
program Cria_Dat;
type
  reg_func = record
    status   : char;
    codigo   : string[4];
    nome     : string[10];
    idade    : integer;
    salario  : real;
  end;
var
  ArqTxt : Text;
  ArqFunc : file of reg_func;
  Linha : string;
  func : reg_func;
  code : integer;
begin
  assign(ArqTxt, 'Func.txt');
  assign(ArqFunc, 'Func.dat');
  reset(ArqTxt);
  rewrite(ArqFunc);
  while not eof(ArqTxt) do begin
    readln(ArqTxt, linha);
    func.status := ' ';
    func.codigo := copy(linha, 1, 4);
    func.nome := copy(linha, 5, 10);
    val(copy(linha, 15, 2), func.idade, code);
    val(copy(linha, 17, 7), func.salario, code);
    write(ArqFunc, func);
  end;
  close(ArqTxt);
  close(ArqFunc);
end.
```

▪ Leitura de um arquivo de dados, com exibição na tela:

```
program Ler_Dat;
type
  reg_func = record
    status   : char;
    codigo   : string[4];
    nome     : string[10];
    idade    : integer;
    salario  : real;
  end;
var
  ArqFunc : file of reg_func;
  func : reg_func;
  pos : longint;
begin
  assign(ArqFunc, 'Func.dat');
  reset(ArqFunc);
  while not eof(ArqFunc) do begin
    pos := filepos(ArqFunc);
    read(ArqFunc, func);
    with func do
      writeln('#', pos, ' ', status, ' ', codigo, ' ', nome, ' ', idade, ' ',
        salario:7:2);
    end;
  end;
  close(ArqFunc);
end.
```

▪ **Inserção em um arquivo de dados, com uso de um arquivo de transações:**

```
program Insere_Dat;
type
  reg_func = record
      status   : char;
      codigo   : string[4];
      nome     : string[10];
      idade    : integer;
      salario  : real;
  end;
var
  ArqFunc, ArqIns, ArqNovo : file of reg_func;
  Func, FuncIns : reg_func;

procedure Ler_Func;
begin
  if not eof(ArqFunc)
  then read(ArqFunc,Func)
  else Func.Codigo := '9999';
end;

procedure Ler_FuncIns;
begin
  if not eof(ArqIns)
  then read(ArqIns,FuncIns)
  else FuncIns.Codigo := '9999';
end;

function Fim : boolean;
begin
  Fim := (Func.Codigo='9999') and (FuncIns.Codigo='9999');
end;

begin
  assign(ArqFunc, 'Func.dat');
  assign(ArqIns, 'FuncIns.dat');
  assign(ArqNovo, 'FuncNovo.dat');
  reset(ArqFunc);
  reset(ArqIns);
  rewrite(ArqNovo);
  Ler_Func;
  Ler_FuncIns;
  while not Fim do begin
    while (Func.Codigo < FuncIns.Codigo) do begin
      write(ArqNovo,Func);
      Ler_Func;
    end;
    while (FuncIns.Codigo < Func.Codigo) do begin
      write(ArqNovo,FuncIns);
      Ler_FuncIns;
    end;
  end;
  close(ArqIns);
  close(ArqFunc);
  close(ArqNovo);
end.
```

▪ **Exclusão de um registro em um arquivo de dados:**

```
program Exclui_Dat;
type
  reg_func = record
    status   : char;
    codigo   : string[4];
    nome     : string[10];
    idade    : integer;
    salario  : real;
  end;

var
  ArqFunc : file of reg_func;
  Func    : reg_func;
  ultimo, nReg : longint;

begin
  assign(ArqFunc, 'Func.dat');
  reset(ArqFunc);
  if not eof(ArqFunc) then begin
    ultimo := filesize(ArqFunc);
    writeln('Arquivo possui ',ultimo+1,' registros.');
```

write('Informe qual registro deseja excluir [0..' ,ultimo,'] : ');

readln(nReg);

if (nReg >= 0) and (nReg <= ultimo) then begin

seek(ArqFunc, nReg);

read(ArqFunc, Func);

writeln('Excluindo funcionario ',func.codigo,'-',func.nome,'...');

Func.Status := '*';

seek(ArqFunc, nReg);

write(ArqFunc, Func);

end else writeln('Registro fora da faixa valida!');

end;

close(ArqFunc);

end.

5. MÉTODOS DE CLASSIFICAÇÃO

5.1. Conceitos

Classificação ou *ordenação* de dados constitui uma das tarefas mais freqüentes e importante em processamento de dados.

Classificação de dados é o processo pelo qual é determinada a ordem em que devem se apresentar as entradas de uma tabela ou os registros de um arquivo, baseado no valor de um ou mais campos. Estes campos são chamados de *chaves de classificação* (ou ordenação).

Existem basicamente dois tipos de ordenação:

- **Interna:** onde a ordenação é realizada totalmente na memória principal;
- **Externa:** onde é utilizada a memória secundária, geralmente caracterizada por um disco.

Existem diversos métodos de classificação interna: por inserção, por seleção, por troca, por distribuição e por intercalação.

5.2. Classificação com Inserção Direta

Consiste em realizar a ordenação da lista de valores pela inserção de cada um dos elementos em sua posição correta dentro da lista, segundo a chave de ordenação. É usado apenas com pequenos conjuntos de dados.

Neste método, o vetor *V* é dividido em dois segmentos. Inicialmente o primeiro segmento possui apenas o primeiro elemento, estando portanto classificado, enquanto o segundo segmento contém os elementos *V*[2] a *V*[*N*]. Através de iterações sucessivas é feita a inserção de todos os elementos do segundo segmento no primeiro, de forma ordenada. Após cada inserção, o primeiro segmento possuirá mais um elemento e o segundo, menos um.

Considerando:

- *N* : número de elementos no vetor
- *V* : vetor com *N* posições (1 a *N*) contendo as chaves
- *PosElemRet* : Posição do elemento a ser retirado do 2º. segmento
- *PosElemIns* : Posição correta do elemento no 1º. segmento
- *PosLim* : Limite do 1º. segmento em determinado instante
- *Chave* : Chave do 2º. segmento a ser inserida no 1º.

```
Procedure OrdenacaoInsercao;  
Var  
    PosElemRet, PosElemIns, PosLim, Chave: integer;  
Begin  
    For PosElemRet := 2 to N do begin  
        PosElemIns := 1;  
        PosLim := PosElemRet - 1;  
        Chave := V[PosElemRet];  
        While (PosLim >= 1) and (PosElemIns = 1) do begin  
            If (Chave < V[PosLim]) then begin  
                V[PosLim + 1] := V[PosLim];  
                PosLim := PosLim - 1;  
            End  
            else PosElemIns := PosLim - 1;  
        End;  
        V[PosElemIns] := Chave;  
    End;  
End;
```

5.3. Classificação por Seleção

Consiste em fazer seleções sucessivas da menor chave de ordenação e colocar o elemento correspondente na sua posição definitiva dentro da lista.

Considerando:

- N : número de elementos do vetor
- V : vetor com N posições (1 a N) contendo as chaves
- PosLimInf : limite inferior do vetor em dado instante
- PosMenorChave : posição da menor chave
- MenorChave : valor da menor chave selecionada em uma passada
- PosAux : variável auxiliar

```
Procedure OrdenacaoSelecao;  
Var  
    PosLimInf, PosMenorChave, MenorChave, PosAux, Aux: integer;  
Begin  
    For PosLimInf := 1 to N do begin  
        MenorChave := V[PosLimInf];  
        PosMenorChave := PosLimInf;  
        For PosAux := (PosLimInf + 1) to N do begin  
            If (V[PosAux] < MenorChave) then begin  
                MenorChave := V[PosAux];  
                PosMenorChave := PosAux;  
            End;  
        End;  
        If PosMenorChave <> PosLimInf then begin  
            Aux := V[PosMenorChave];  
            V[PosMenorChave] := V[PosLimInf];  
            V[PosLimInf] := Aux;  
        End;  
    End;  
End;
```

5.4. Classificação por Troca (BubbleSort)

Também conhecido como "método da bolha", consiste na comparação de pares de chaves de ordenação, trocando os elementos correspondentes, caso estejam fora de ordem.

Considerando:

- N : número de elementos do vetor
- V : vetor com N posições (1 a N) contendo as chaves
- HouveTroca : variável booleana que indica se houve troca ou não
- PosLimInf : limite inferior do vetor em dado instante
- PosTroca : Posição onde ocorreu a última troca
- PosAux : variável auxiliar

```
Procedure BublbleSort;  
Var  
    PosLimInf, PosTroca, PosAux, Aux: integer;  
    HouveTroca : boolean;  
Begin  
    PosLimInf := N - 1;  
    HouveTroca := True;  
    While HouveTroca do begin  
        HouveTroca := False;  
        For PosAux := 1 to PosLimInf do  
            If V[PosAux] > V[PosAux+1] then begin  
                Aux := V[PosAux];  
                V[PosAux] := V[PosAux + 1];  
                V[PosAux + 1] := Aux;  
            End;  
        End;  
        PosLimInf := PosTroca;  
        HouveTroca := (PosTroca < PosLimInf);  
    End;  
End;
```

```

        PosTroca := PosAux;
        HouveTroca := True;
    End;
    PosLimInf := PosTroca;
End;
End;

```

5.5. Classificação Rápida por troca (QuickSort)

Em 1962, Hoare desenvolveu um dos algoritmos mais eficientes de ordenação. O algoritmo particiona o vetor a ser ordenado em dois subconjuntos, um à esquerda e outro à direita, de tal forma que todo elemento do subconjunto à esquerda seja menor que qualquer elemento do subconjunto à direita. Cada subconjunto é reparticionado segundo este mesmo critério, e assim sucessivamente. Isto nos leva a uma solução recursiva.

O particionamento ocorre a partir de um elemento que será o pivô. É a comparação entre este pivô e as demais chaves que determina o correto posicionamento dos elementos para um novo particionamento. Neste exemplo, o pivô será o elemento central de cada partição.

Considerando:

- V : vetor com N posições (1 a N) contendo as chaves
- PosEsquerda : posição inicial da partição
- PosDireita : posição final da partição
- Pivo : elemento do meio da partição

```

Procedure QuickSort (PosEsquerda, PosDireita : integer);
Begin
    PosEsquerdaAux := PosEsquerda;
    PosDireitaAux := PosDireita;
    Pivo := V[trunc((PosEsquerda+PosDireita)/2)];
    While (PosEsquerdaAux <= PosDireitaAux) do begin
        While V[PosEsquerdaAux] < Pivo do PosEsquerdaAux := PosEsquerdaAux + 1;
        While V[PosDireitaAux] > Pivo do PosDireitaAux := PosDireitaAux - 1;
        If PosEsquerdaAux <= PosDireitaAux then begin
            Aux := V[PosEsquerdaAux];
            V[PosEsquerdaAux] := V[PosDireitaAux];
            V[PosDireitaAux] := Aux;
            PosEsquerdaAux := PosEsquerdaAux + 1;
            PosDireitaAux := PosDireitaAux - 1;
        End;
    End;
    If PosEsquerda < PosDireitaAux then QuickSort(PosEsquerda,PosDireitaAux);
    If PosDireita > PosEsquerdaAux then QuickSort(PosEsquerdaAux,PosDireita);
End;

```

5.6. Intercalação

A intercalação consiste na obtenção de um vetor ordenado, a partir de dois outros vetores já ordenados. Na intercalação é feita uma varredura simultânea dos dois vetores, sendo a cada vez movido o menor de dois elementos para o vetor resultante.

Considerando:

- V1 : vetor com N posições (1 a N) contendo as chaves já ordenadas
- V2 : vetor com M posições (1 a M) contendo as chaves já ordenadas
- V : vetor resultante da intercalação com M+N elementos
- IndV1 : índice do vetor V1
- IndV2 : índice do vetor V2
- IndV : índice do vetor V
- N : posição do último elemento do vetor V1

- M : posição do último elemento do vetor V2
- Ch1 : elemento do vetor V1
- Ch2 : elemento do vetor V2
- MaxValor : maior valor possível no domínio das chaves

```

program Insercao;
uses Crt;

type
  vet = array [1..60] of integer;
var
  v1,v2,v : vet;
  n,m,maxvalor,ch1,ch2,indv1,indv2,indv,i : integer;

procedure mostra_v;
begin
  for i := 1 to n+m do write(v[i]:3,' ');
  writeln; readkey;
end;

Procedure LeV1;
Begin
  IndV1 := IndV1 + 1;
  if IndV1 <= N
    then Ch1 := V1[IndV1]
    else Ch1 := MaxValor;
end;

Procedure LeV2;
Begin
  IndV2 := IndV2 + 1;
  if IndV2 <= M
    then Ch2 := V2[IndV2]
    else Ch2 := MaxValor;
end;

procedure Atribui_1;
begin
  IndV := IndV + 1;
  V[IndV] := Ch1;
  LeV1;
end;

procedure Atribui_2;
begin
  IndV := IndV + 1;
  V[IndV] := Ch2;
  LeV2;
end;

Function Fim : boolean;
Begin
  Fim := (Ch1 = MaxValor) and (Ch2 = MaxValor);
End;

Procedure Intercala;
Begin
  IndV1 := 0;
  IndV2 := 0;
  IndV := 0;
  LeV1;
  LeV2;
  While not Fim do begin
    if (Ch1 < Ch2)

```

```

        then Atribui_1
        else if (Ch2 < Ch1)
            then Atribui_2
            else if not fim then begin
                Atribui_1;
                Atribui_2;
            end;
        end;
    End;
begin
    clrscr;
    maxvalor := 9999;
    n := 10;
    for i := 1 to n do begin
        v1[i] := i*2;
    end;
    m := 15;
    for i := 1 to m do begin
        v2[i] := (i*2)+1;
    end;
    intercala;
    mostra_v;
end.

```

6. MÉTODOS DE PESQUISA

6.1. Conceitos

A pesquisa (ou busca) de chaves em uma estrutura de dados é uma operação que se encontra praticamente em todos os sistemas. A escolha da técnica de pesquisa a ser utilizada é muito importante, pois vai influenciar diretamente o desempenho do sistema.

Os dois tipos de pesquisa mais utilizados em estruturas estáticas são a pesquisa seqüencial e a pesquisa binária.

6.2. Pesquisa seqüencial

Consiste em percorrer a estrutura do início ao fim, ou até a posição onde se encontra a chave procurada.

Considerando:

- N : número de elementos em um vetor
- V : vetor com N elementos
- Chave : valor sendo pesquisado

```
Procedure PesquisaSequencial;  
Var Ind : integer;  
Begin  
  Ind := 1;  
  While (ind <= N) and (V[Ind] <> Chave) do Ind := ind + 1;  
  If (Ind > N)  
    Then writeln('Chave não encontrada')  
    Else writeln('Chave encontrada na posicao ', Ind);  
End;
```

A pesquisa pode ser agilizada incluindo um elemento após a última posição do vetor com a chave procurada. Assim precisamos apenas de uma comparação durante o laço.

Considerando:

- N : número de elementos originais em um vetor
- V : vetor com N+1 elementos
- Chave : valor sendo pesquisado

```
Procedure PesquisaSequencialRapida;  
Var Ind : integer;  
Begin  
  V[N+1] := Chave;  
  Ind := 1;  
  While (V[Ind] <> Chave) do Ind := ind + 1;  
  If (Ind = N+1)  
    Then writeln('Chave não encontrada')  
    Else writeln('Chave encontrada na posicao ', Ind);  
End;
```

Se os elementos estiverem ordenados no vetor, o algoritmo pode ser modificado.

Considerando:

- N : número de elementos originais em um vetor
- V : vetor com N+1 elementos
- Chave : valor sendo pesquisado
- MaxValor : Maior valor que a chave pode assumir;

```

Procedure PesquisaSequencialOrdenada;
Var Ind : integer;
Begin
  V[N+1] := MaxValor;
  Ind := 1;
  While (V[Ind] < Chave) do Ind := ind + 1;
  If (V[Ind] > Chave)
    Then writeln('Chave não encontrada')
    Else writeln('Chave encontrada na posicao ', Ind);
End;

```

6.3. Pesquisa binária

Esta é a técnica mais utilizada para pesquisa em um conjunto **ordenado** de valores. Consiste em obter o elemento do meio da estrutura e comparar este valor com a chave procurada. Como resultado desta comparação podemos ter:

- Se Chave é menor que o valor do meio, ela deve ser procurada no intervalo entre o início do vetor até o meio.
- Se Chave é maior que o valor do meio, ela deve ser procurada no intervalo entre o meio do vetor e seu final.
- Se Chave é igual ao valor do meio, a pesquisa é encerrada.

Considerando:

- N : número de elementos em um vetor
- V : vetor com N elementos
- Chave : valor sendo pesquisado
- Ini : limite inicial para pesquisa
- Fim : limite final para pesquisa
- Meio : posicao no meio do vetor

```

Procedure PesquisaBinaria;
Begin
  Ini := 1;
  Fim := N;
  While (Ini <= Fim) do begin
    Meio := Trunc((Ini+Fim)/2) + 1;
    If Chave < V[Meio]
      Then Fim := Meio - 1
      Else If Chave > V[Meio]
        Then Ini := Meio + 1
        Else Ini := Fim + 1;
  End;
  If Chave <> V[Meio]
    Then writeln('Chave não encontrada')
    Else writeln('Chave encontrada na posicao ', Meio);
End;

```