

Notas de Aula de Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.1

Estas notas de aula não devem ser usadas como única fonte de estudo. O aluno deve ler outros livros disponíveis na literatura.

Nenhuma parte destas notas pode ser reproduzida, qualquer que seja a forma ou o meio, sem a permissão dos autores.

Os autores concedem a permissão explícita para a utilização e reprodução deste material no contexto do ensino de disciplinas regulares dos cursos de graduação sob a responsabilidade do Instituto de Computação da UNICAMP.

© Copyright 2000

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas-SP
{fkm,tomasz}@ic.unicamp.br

13 Arquivos

Quando um programa trabalha com dados que estão na memória RAM, estes são apagados sempre que o computador é desligado ou mesmo quando o programa é terminado. Mas muitas vezes queremos que os dados se mantenham em memória para que possam ser processados posteriormente. Neste caso, é necessário que estes dados estejam gravados em uma memória não volátil. Este é o caso de se usar *arquivos*, que permite gravar os dados e recuperá-los sem necessidade de reentrada dos dados. Outra razão para se usar arquivos é que estes são armazenados em memória secundária, discos e fitas magnéticas, que têm em geral uma capacidade muito maior de armazenamento.

Podemos distinguir os arquivos suportados pela linguagem Pascal como sendo de dois tipos: arquivos de texto e arquivos binários.

13.1 Arquivos Texto

Quando trabalhamos com arquivos de texto, supomos que não iremos trabalhar com qualquer tipo de dado, mas com dados que nos representem um texto. Neste caso, vamos supor que os dados de um texto estão organizadas por linhas, onde cada linha contém uma seqüência de caracteres ASCII que podem ser impressos. Exemplo destes caracteres são as letras (maiúsculas/minúsculas/com acento), dígitos e os caracteres especiais (como: \$, %, #, , ?, ...). Além disso, um arquivo texto contém alguns caracteres de controle, como por exemplo os caracteres que indicam o fim de uma linha. A unidade básica em arquivos texto é o caracter.

Um programa em Pascal considera o vídeo e o teclado como arquivos texto especiais. O vídeo como um arquivo onde podemos apenas escrever e o teclado como um arquivo que podemos apenas ler.

Existem algumas diferenças entre um arquivo texto feito para o sistema operacional MS-DOS e um arquivo texto feito para o ambiente UNIX. Um arquivo texto no ambiente MS-DOS usa dois caracteres para especificar que começará a próxima linha, o caracter Carriage Return seguido do caracter Line Feed, que tem os códigos internos 13 e 10, respectivamente. No caso dos arquivos texto em ambiente UNIX, a próxima linha começa apenas depois de um caracter, Line Feed, de código interno 10.

Quando trabalhamos com arquivos, é importante considerar que todo arquivo tem um indicador da posição corrente de leitura e escrita (gravação). Inicialmente este indicador está posicionado no primeiro byte do arquivo. Se for realizada uma operação de leitura ou gravação de n bytes, então o indicador da posição corrente no arquivo se deslocará destes n bytes para frente. Além disso, existe uma posição que será a seguinte do último byte do arquivo (posição de fim de arquivo) e não poderemos ler nenhum byte seguinte, mas poderemos escrever outros bytes a partir desta posição.

A sintaxe para declarar uma variável para trabalhar com arquivos texto é a seguinte:

```
var Lista_de_Identificadores_de_Arquivos: Text;
```

O identificador declarado desta forma será usado para trabalhar com um arquivo que está gravado em disco. O arquivo que está em disco (ou outro meio de armazenamento secundário) tem um nome e a primeira operação para trabalhar com ele é *associar* este nome (*nome externo*) com a variável de arquivo. O comando para fazer esta associação é a seguinte:

```
assign(Variável_de_arquivo, Nome_Arquivo_Externo);
```

Esta operação apenas “diz” para o programa qual o arquivo (que está em disco) que iremos trabalhar pela variável de arquivo. Mas isto não faz com que os dados que estão no arquivo (em disco) já estejam disponíveis para serem manipulados. Para isso, precisamos dar outro comando que de fato irá possibilitar esta manipulação. Informalmente, este comando irá “abrir” o arquivo para ser manipulado. Podemos ter três tipos de comandos para abrir um arquivo texto, um que é somente para leitura dos dados, outro que é para escrita em um arquivo novo e outro que é para escrita a partir do fim do arquivo texto.

Para abrir um arquivo somente para leitura (a partir do início do arquivo), usamos o seguinte comando:

```
reset(Variável_de_arquivo);
```

Note que não precisamos dar o nome externo do arquivo, uma vez que isto já foi feito pelo comando **assign**. Neste caso, o indicador de posição corrente fica apontando para o início do arquivo.

Para abrir um arquivo somente para escrita começando com um arquivo vazio, usamos o seguinte comando:

```
rewrite(Variável_de_arquivo);
```

Se já existir um arquivo com o mesmo nome, o arquivo que existia antes é reinicializado como arquivo vazio. Como neste caso sempre começamos com um arquivo vazio, o indicador da posição corrente começa no início do arquivo.

Por fim, para abrir um arquivo somente para escrita a partir do fim do arquivo (i.e., o conteúdo inicial do arquivo é preservado) usamos o seguinte comando:

```
append(Variável_de_arquivo);
```

Neste caso, o indicador da posição corrente no arquivo fica inicializado como o fim de arquivo.

O comando de abertura de arquivo, em um programa Pascal, usa uma memória auxiliar (em memória RAM) para colocar parte dos dados que são lidos/escritos do/para o arquivo. Isto é porque em geral, qualquer mecanismo de armazenamento secundário (disco rígido, fitas magnéticas, disquetes,...) funcionam através de meios eletro-mecânicos, o que força com que a leitura e escrita de dados usando estes dispositivos seja muito lenta. Assim, sempre que uma operação de leitura ou escrita é feita, ela não é feita diretamente no arquivo, mas sim nesta memória auxiliar (que é extremamente rápida). Caso algum comando faça uso de um dado do arquivo que não está nesta memória auxiliar, o programa usa serviços do sistema operacional para atualizar o arquivo com a memória auxiliar e colocar na memória auxiliar os novos dados que o programa precisa manipular.

Se um programa for finalizado sem nenhum comando especial, os arquivos que ele estiver manipulando poderão não estar atualizados corretamente, já que parte dos dados pode estar na memória auxiliar. Assim, quando não precisarmos mais trabalhar com o arquivo, é necessário usar de um comando que descarrega todos os dados desta memória auxiliar para o arquivo propriamente dito (além de liberar esta memória auxiliar para futuro uso do sistema operacional). Isto é feito com o comando **close** da seguinte maneira:

```
close(Variável_de_arquivo);
```

Uma vez que o arquivo foi aberto, poderemos ler e gravar dados (dependendo do modo como o arquivo foi aberto) no arquivo. Lembrando que para ler do teclado, usamos o comando read/readln, e para escrever no vídeo, usamos o comando write/writeln. Do mesmo jeito, para ler e escrever (gravar) em um arquivo texto, usamos os comandos read, readln, write e writeln. A única diferença é que colocamos um parâmetro a mais nestes comandos, especificando em qual arquivo será feita a operação de leitura ou escrita.

Assim, vamos supor que em determinado momento os dados do arquivo e a posição corrente no arquivo estão como apresentados no quadro seguinte, onde apresentamos as três primeiras linhas do arquivo e a posição corrente está indicado por uma seta.

```
1  2  3  Esta é a primeira linha do arquivo texto
123 456 3.14 999 Esta é a segunda linha
↑
Terceira Linha
```

se dermos o comando:

```
read(Variável_de_arquivo,a, b, c);
```

onde a , b e c são variáveis reais, teremos o mesmo resultado se dêssemos o comando `read(a, b, c);` e entrássemos pelo teclado com os dados `456 3.14 999`. I.e., as variáveis a , b e c ficarão com os valores 456, 3.14 e 999, respectivamente, e o indicador da posição corrente é atualizado de maneira a ficar com a seguinte configuração:

```
1  2  3  Esta é a primeira linha do arquivo texto
123 456 3.14 999 Esta é a segunda linha
↑
Terceira Linha
```

Outra rotina importante é a função para verificar se estamos no fim de arquivo, pois neste caso não poderemos ler mais nenhum dado a partir do arquivo texto. A função para fazer esta verificação é a função booleana **eof**, cuja sintaxe é:

```
eof(Variável_de_arquivo)
```

A função **eof** retorna verdadeiro (**true**) caso o indicador da posição corrente esteja no fim do arquivo e falso (**false**) caso contrário.

Além do comando **read**, também podemos usar o comando **readln**. Neste caso, os dados são lidos como no comando **read**, mas se ainda existir alguns dados na mesma linha corrente do arquivo texto, estes são ignorados e o indicador da posição corrente fica situado no primeiro caracter da próxima linha.

Se a variável a ser lida é uma cadeia de caracteres (**string**), a quantidade de caracteres é definida pelo número de caracteres que ela foi definida. Além disso, se a rotina **read** for aplicada para ler uma cadeia de caracteres e a quantidade de caracteres a ler ultrapassa a linha do arquivo, o indicador da posição corrente fica posicionado no fim desta linha. I.e., se uma próxima leitura com o comando **read** for usado para ler uma string, o indicador de posição corrente se manterá naquele mesmo fim de linha. Neste caso, para começar a ler a próxima linha como string, o comando de leitura anterior deve ser o **readln**. Caso a leitura seja de algum outro tipo (**integer, real, ...**), mesmo que se use o comando **read**, e o dado não estiver na linha corrente, o comando busca o dado a ser lido nas próximas linhas.

Os comandos de impressão (gravação), **write** e **writeln** têm resultado semelhante aos correspondentes comandos para imprimir na tela. A diferença é que o resultado da impressão será feita no arquivo. Além disso, um arquivo texto não tem a limitação de número de colunas, como temos em um vídeo. Lembrando que para ir para a próxima linha, em um arquivo texto MS-DOS são usados os caracteres *Carriage_Return* e *Line_Feed*, enquanto em arquivos Unix, é usado apenas o caracter *Line_Feed*.

Em um arquivo texto, como em um arquivo binário, podemos obter a quantidade de elementos básicos do arquivo através da função *filesize*, cuja sintaxe é:

```
filesize(Variável_de_arquivo)
```

e retorna a quantidade de elementos básicos do arquivo.

Exemplo 13.1 *O seguinte programa lê o nome de um arquivo texto e o lista na tela, especificando o número de cada linha. Por fim, o programa também imprime o número de linhas e o número de bytes do arquivo.*

```
program ListaArquivo(input,output);
type TipoString = string[255];
procedure LeArquivoTexto(NomeArquivo:TipoString);
var ArquivoTexto : text;
    linha      : TipoString;
    nlinha     : integer;
begin
    assign(ArquivoTexto,NomeArquivo); {Associar o nome do arquivo com a variável}
    reset(ArquivoTexto);             {Abrir o arquivo para leitura}
    nlinha:=0;
    while not eof(ArquivoTexto) do begin
        nlinha:=nlinha+1;  readln(ArquivoTexto,linha);  writeln(' [ ',nlinha:3,' ] ',linha);
    end;
    writeln(NomeArquivo,' tem ',nlinha,' linhas e usa ',filesize(ArquivoTexto),' bytes. ');
    close(ArquivoTexto); {Fecha o arquivo}
end; { LeArquivoTexto }
var NomeArq : TipoString;
begin
    write('Entre com o nome do arquivo a listar: ');
    readln(NomeArq);
    LeArquivoTexto(NomeArq);
end.
```

Exemplo 13.2 O programa seguinte elimina os espaços em branco repetidos (seguidos) de um arquivo texto.

```
program Brancos;
var
  linha          : String;
  nomeEntrada, nomeSaida : String[20];
  arqEntrada, arqSaida   : Text;

procedure EliminaBranco(var s: String);
var
  i,desl,n : Integer;
  p        : char;
begin
  desl := 0; n := Length(s); p := 'x';
  for i:=1 to n do
    begin
      if s[i]<>' '
        then s[i-desl] := s[i]
      else if p<>' '
        then s[i-desl] := s[i]
      else inc(desl);
      p := s[i]
    end;
  Setlength(s,n-desl)
end; {EliminaBranco}

begin
  Write('Forneça o nome do arquivo de entrada: ');
  Readln(nomeEntrada);
  Write('Forneça o nome do arquivo de saída: ');
  Readln(nomeSaida);
  AssignFile(arqEntrada,nomeEntrada);
  AssignFile(arqSaida,nomeSaida);
  Reset(arqEntrada);
  Rewrite(arqSaida);
  while not eof(arqEntrada) do
    begin
      Readln(arqEntrada,linha);
      EliminaBranco(linha);
      Writeln(arqSaida,linha)
    end;
  Close(arqSaida)
end.
```

Exercício 13.1 Os professores do curso de Algoritmos e Programação de Computadores desenvolveram uma linguagem assembler, chamada HIP, para um computador hipotético com conjunto de comandos bastante reduzido. A linguagem é voltada para manipulação de inteiros e tem 16 instruções que explicitaremos a seguir: O único tipo de dado existente em HIP é o tipo inteiro. Cada valor inteiro deve ser armazenado em uma variável representada por um identificador que usa apenas uma letra. Além de variáveis, os identificadores podem representar labels que podem ser pontos de desvios do programa.

- **var** <Ident>
Faz a declaração da variável <Ident>, inicializada com valor 0 (zero).
- **ler** <Ident>
Lê o valor da variável <Ident> pelo teclado.
- **atr** <Ident> <Constante>
Atribui a constante <Constante> (número inteiro) para a variável <Ident>.
- **mov** <Ident> <Ident1>
Atribui o valor da variável <Ident1> para a variável <Ident>.
- **sum** <Ident> <Ident1> <Ident2>
Soma os valores das variáveis <Ident1> e <Ident2> e atribui o resultado na variável <Ident>.
- **sub** <Ident> <Ident1> <Ident2>
Subtrai o valor da variável <Ident1> do valor da variável <Ident2> e atribui o resultado na variável <Ident>.
- **mul** <Ident> <Ident1> <Ident2>
Multiplica os valores das variáveis <Ident1> e <Ident2> e atribui o resultado na variável <Ident>.
- **div** <Ident> <Ident1> <Ident2>
Divide o valor da variável <Ident1> por <Ident2> e atribui a parte inteira na variável <Ident>.
- **mod** <Ident> <Ident1> <Ident2>
Este comando atribui o resto da divisão inteira de <Ident1> por <Ident2> na variável <Ident>.
- **lab** <Ident>
Este comando define um label (ponto de desvio do programa) com o nome <Ident>.
- **des** <Ident>
Este comando desvia o fluxo de execução do programa para o comando com label <Ident>.
- **ifz** <Ident> <Ident1>
Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor 0 (zero).
- **ifp** <Ident> <Ident1>
Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor positivo.
- **ifn** <Ident> <Ident1>
Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor negativo.
- **imp** <Ident>
Imprime o valor da variável <Ident> na tela.
- **fim**
Termina a execução do programa.

Vamos exemplificar esta linguagem com um programa feito em HIP para calcular o máximo divisor comum pelo algoritmo de Euclides.

Linha lida	Explicação do comando
<code>var a</code>	Declara a variável <code>a</code> .
<code>var b</code>	Declara a variável <code>b</code> .
<code>var r</code>	Declara a variável <code>r</code> .
<code>ler a</code>	Lê um valor na variável <code>a</code> através do teclado.
<code>ler b</code>	Lê um valor na variável <code>b</code> através do teclado.
<code>lab x</code>	Declara ponto de desvio <code>x</code> {Início da estrutura de repetição}
<code>mod r a b</code>	$r \leftarrow a \bmod b$.
<code>mov a b</code>	$a \leftarrow b$.
<code>mov b r</code>	$b \leftarrow r$.
<code>ifz r y</code>	Até que $r = 0$. Neste caso, desvio o ponto <code>y</code> .
<code>des x</code>	Volta a repetir desde o ponto <code>x</code> .
<code>lab y</code>	Ponto de desvio fora da estrutura de repetição
<code>imp a</code>	Impressão do MDC.
<code>fim</code>	Fim do programa.

Faça um programa que lê um programa em HIP (armazenado como um arquivo texto) e faz a execução do programa. Para facilitar, armazene o programa em um vetor, comando por comando, incluindo as operações **var** e **lab**. A posição que tiver o comando **var** contém um campo chamado `valor` que pode armazenar um valor inteiro.

Exercício adicional: Implemente programas em HIP para: (i) Calcular o fatorial de um número positivo. (ii) Verificar se um número positivo é primo.

Exercício adicional: Modifique o programa para que comandos como **lab** e **var** não sejam mais representados no vetor. As variáveis são representadas em um vetor adicional, chamado `memória` e os comandos de desvios são armazenado com o índice da posição no vetor para onde o fluxo é desviado. Os comandos que fazem manipulação de variáveis devem armazenar seus respectivos índices na memória. Estas modificações agilizam a execução de um programa em HIP, uma vez que não há necessidade de se buscar um identificador dentro do programa.

13.2 Arquivos Binários

Um arquivo binário é um arquivo que não tem necessariamente a restrição de ter apenas texto. A linguagem Pascal trata os arquivos binários como sendo uma seqüência de elementos de arquivo. Os elementos de um mesmo arquivo tem o mesmo tipo, que pode ser um tipo básico da linguagem Pascal, ou outro definido pelo programador.

Os comandos **read** e **write** também podem ser usadas em arquivos binários (as funções `readln` e `writeln` não são válidas para arquivos binários), mas neste caso, usamos exatamente dois parâmetros para estas funções, o arquivo a ler/escrever e a variável (de mesmo tipo que o elemento básico) que iremos ler/escrever.

Um arquivo binário é declarado da forma:

FILE of tipo_elemento_básico;

Exemplo 13.3 A seguir apresentamos alguns exemplos de declaração de arquivos.

`ArqDados: File of TipoAluno;`

`ArqNums: File of Integer;`

`ArqVetCars: File of array [1..100] of char;`

`ArqCars: File of char;`

Obs.: Arquivos texto (**text**) são muito semelhantes com arquivos do tipo **file of char**, mas arquivos do tipo **text** tem operações como `EOLN` (*End of Line*), o que não ocorre com arquivos do tipo **file of char**.

Para associar o nome externo (nome do arquivo em disco) também usamos o comando `assign`, da mesma forma que usada para arquivos texto.

Os comandos de “abertura” de um arquivo binário são o **reset** e o **rewrite**.

Para abrir um arquivo para leitura e gravação de um arquivo já existente, usamos o comando:

`reset(Variável_de_arquivo);`

Note a diferença deste comando para arquivos texto e arquivos binários. Usando este comando para arquivos texto, só podemos ler do arquivo, enquanto para arquivos binários, podemos ler e escrever. Inicialmente os dados já existentes no arquivo são mantidos, podendo ser atualizados por comandos de escrita. Caso o arquivo já esteja aberto, este comando reposiciona o indicador de posição corrente do arquivo para o início do arquivo.

Para abrir um arquivo novo (vazio) para leitura e gravação começando com um arquivo vazio, usamos o comando **rewrite**:

```
rewrite(Variável_de_arquivo);
```

Se já existir um arquivo com o mesmo nome, o arquivo que existia antes é reinicializado como arquivo vazio. Com este comando também podemos ler, escrever/atualizar no arquivo binário.

Tanto usando o comando **reset** como **rewrite**, o indicador de posição corrente é inicializado no início do arquivo.

Para ler um elemento do arquivo (definido pela posição do indicador corrente do arquivo) podemos usar o comando **read**:

```
read(Variável_de_arquivo,Variável_Elemento_Básico);
```

onde Variável_Elemento_Básico é a variável que irá receber o valor do elemento que está indicado no arquivo. Após a leitura deste elemento o indicador de posição corrente passa para o próximo elemento (fica em fim de arquivo, caso não exista o próximo elemento).

Para escrever um elemento no arquivo (na posição indicada pelo indicador de posição corrente do arquivo) usamos o comando **write**:

```
write(Variável_de_arquivo,Variável_Elemento_Básico);
```

onde Variável_Elemento_Básico é a variável que irá receber o valor do elemento que está indicado no arquivo. Após a escrita deste elemento o indicador de posição corrente passa para o próximo elemento (fica em fim de arquivo, caso não exista o próximo elemento).

Quando usamos arquivos binários, podemos usar a função **eof**, que retorna **true** caso estejamos no fim de arquivo e **false** caso contrário e também podemos redefinir a posição do indicador corrente de arquivo, tanto para frente como para trás. O comando que nos permite mudar este indicador é o comando **seek** cuja sintaxe é a seguinte.

```
seek(Variável_de_arquivo,NovaPosição);
```

onde NovaPosição é um inteiro entre 0 e $(\text{filesize}(\text{Variável_de_arquivo})-1)$. Com este comando, o indicador irá ficar posicionado no elemento de arquivo de número *NovaPosição*. Lembrando que o primeiro elemento está na posição 0 (zero) e $\text{filesize}(\text{Variável_de_arquivo})$ retorna o número de elementos de arquivo.

Exemplo 13.4 A universidade UNICOMP mantém um cadastro de alunos, cada aluno do tipo *TipoAluno*, como declarado no exercício 9.2 página 91. Para que os dados sejam armazenados de maneira permanente, em algum tipo de memória secundária (discos flexíveis, discos rígidos, ...) é necessário guardar os dados usando arquivos. Faça mais duas rotinas para o programa do exercício 9.2 de maneira que o programa possa gravar os alunos em um arquivo, lidos a partir de um vetor, e ler os alunos de um arquivo inserindo-os em um vetor. O nome do arquivo deve ser fornecido pelo usuário.

```

program LeituraGravacaoDeAlunos;
type
{ --> Inserir aqui a declaração de TipoAluno <-- }
  mystring      = string[255];
  tipocadastro  = array[1..100] of tipoaluno;

procedure LeArquivo(var v : TipoCadastro; var n:integer; NomeArquivoDisco:mystring);
var Arq: File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
begin
  assign(Arq,NomeArquivoDisco);
  reset(Arq);
  n:=0;
  while not eof(Arq) do begin
    n:=n+1;
    read(Arq,v[n]); { Le o n-ésimo elemento básico do arquivo }
  end; { Quando terminar o loop, n terá a quantidade de elementos }
  close(Arq);
end;

procedure GravaArquivo(var v: TipoCadastro; n:integer; NomeArquivoDisco:mystring);
var i : integer;
  Arq: File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
begin
  assign(Arq,NomeArquivoDisco);
  rewrite(Arq); { Começa um arquivo vazio }
  for i:=1 to n do write(Arq,v[i]); { Grava n elementos básicos (alunos) }
  close(Arq);
end;

var n,i          : integer;
  Cadastro       : tipocadastro;
  NomeArquivoDisco : mystring;
begin
  writeln('Entre com o nome do arquivo contendo os alunos: ');
  readln(NomeArquivoDisco);
  LeArquivo(Cadastro,n,NomeArquivoDisco);
  writeln('Os nomes dos alunos lidos são:');
  for i:=1 to n do writeln(Cadastro[i].nome);
end.

```

Exemplo 13.5 Vamos considerar que temos um sistema que mantém um cadastro de alunos, cada aluno do tipo *TipoAluno*, como no exemplo 13.4. Agora vamos supor que temos uma grande restrição de limitação de memória do computador (e conseqüentemente do programa) usado. I.e., não podemos guardar todos os alunos do cadastro em um vetor. Além disso, vamos supor que necessitamos da seguinte operação: Ordenar os alunos, no próprio arquivo e listá-los na tela. Desenvolva uma rotina de ordenação, por nome de aluno, usando não mais que uma quantidade limitada (constante) de memória para guardar alunos.

```

program OrdenacaoDeAlunos;
uses tipoaluno1;
type
{ --> Inserir aqui a declaração de TipoAluno <-- }
  mystring      = string[255];
  TipoArquivoAluno = File of TipoAluno;
procedure TrocaAlunoArquivo(var Arq :TipoArquivoAluno; i1,i2 : integer);
var Aluno1,Aluno2: TipoAluno;
begin
  seek(Arq,i1); read(Arq,Aluno1); seek(Arq,i2); read(Arq,Aluno2);
  seek(Arq,i2); write(Arq,Aluno1); seek(Arq,i1); write(Arq,Aluno2);
end;
function IndMaximoArquivo(var Arq:TipoArquivoAluno; n : integer) : integer;
var i, Ind : integer;
  Maior,Aux : TipoAluno;
begin
  seek(Arq,0); read(Arq,Maior);
  Ind := 0; { O maior elemento começa com o primeiro elemento do vetor }
  for i:=1 to n do begin { Já começa no segundo elemento (posição 1) }
    read(Arq,Aux); { Le um elemento do arquivo }
    if Maior.nome < Aux.nome then begin
      Ind:=i;      { Se o elemento lido é maior que o que encontramos, }
      Maior:=Aux;  { Atualize o maior e o indice do maior }
    end;
  end;
  IndMaximoArquivo := Ind;
end; { IndMaximoArquivo }
procedure OrdenaAlunosArquivo(NomeArquivoDisco:mystring);{ Idéia do SelectionSort }
var Arq      : File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
  n,m,imax : integer;
begin
  assign(Arq,NomeArquivoDisco); reset(Arq);
  n:=filesize(Arq)-1; { Obtém a posição do último elemento do arquivo }
  for m:=n downto 1 do begin { Modificação devido ao arquivo começar com posição 0 }
    imax := IndMaximoArquivo(Arq,m);
    TrocaAlunoArquivo(Arq,m,imax); { Troca os alunos das posições m e imax }
  end;
  close(Arq);
end; { OrdenaAlunosArquivo }
var aluno:TipoAluno;  n,i:integer;  Arq:TipoArquivoAluno;
begin
  assign(Arq,'Arquivo.dat'); reset(Arq);
  n:=filesize(Arq);
  for i:=1 to n do begin
    read(Arq,Aluno); writeln(Aluno.nome);
  end;
end.

```

Exercício 13.2 A rotina de ordenação de elementos de arquivo, sem uso de vetores (diretamente no arquivo), apresentada no exemplo 13.5, usa a estratégia do SelectionSort. I.e., localiza o elemento de maior valor e coloca na posição correta, e continua com o mesmo processo com os demais elementos. Faça uma rotina de ordenação, nas mesmas condições apresentadas no exemplo 13.5, mas agora usando o algoritmo QuickSort.

Exercício 13.3 A universidade UNICOMP tem milhares de funcionários e todo início de mês, quando deve fazer o pagamento de seus funcionários, deve imprimir uma listagem de funcionários, e para cada funcionário o programa deve imprimir seu nome, cargo e salário. O número de funcionários é grande, mas a quantidade de empregos diferentes é bem pequena. Há apenas 5 tipos de empregos diferentes, conforme a tabela a seguir:

Cargo	Salário
Faxineira	100,00
Pedreiro	110,00
Carpinteiro	120,00
Professor	130,00
Reitor	200,00

Como os recursos da UNICOMP são escassos, ela deve economizar memória (em disco) do computador que fará a listagem. Assim, o programador que decide fazer o sistema resolve colocar a tabela acima em apenas um arquivo, chamado CARGOS. Cada registro deste arquivo é do seguinte tipo:

type

```
tipocargo = record
    nomecargo: string[50];
    salario:real;
end;
```

Outro arquivo a ser manipulado é o arquivo CADASTRO, que contém os funcionários. Este arquivo é da seguinte forma:

Nome	PosCargo
José Carlos	2
Carlos Alberto	1
Alberto Roberto	0
Roberto José	4
⋮	⋮

Cada registro deste arquivo é do seguinte tipo:

type

```
tipofuncionario = record
    nome: string[50];
    poscargo:integer;
end;
```

Desta maneira não é necessário guardar todos os dados do cargo junto com o do funcionário. Basta guardar um indicador de onde está a informação de cargo.

Se um funcionário tem o campo poscargo igual a k , então seu cargo é o cargo que esta na linha $(k + 1)$ do arquivo de cargos. Assim, o funcionário "José Carlos" tem cargo "Carpinteiro" e ganha 120,00 reais; o funcionário "Roberto José" tem cargo "Reitor" e ganha 200,00 reais. Faça um programa (com menu) que:

- Inicialmente gera o arquivo CARGOS, apresentado acima.
- Lê sempre que o usuário quiser, um novo funcionário e o inteiro indicando a posição do cargo dele para adicioná-lo no arquivo CADASTRO (inicialmente vazio).
- Se o usuário quiser lista todos os funcionários, um por linha, indicando o nome, cargo e salário dele.

Sugestão: Durante a execução do programa, mantenha os dois arquivos abertos ao mesmo tempo.

Exercício 13.4 Em um sistema computacional, existem dois arquivos, um chamado *Funcionarios.dat* e outro chamado *Cargos.dat*, ambos arquivos binários. O arquivo *Funcionarios.dat* é um arquivo de elementos do tipo *tipofunc* e o arquivo *Cargos.dat* é um arquivo de elementos do tipo *tipocargo*, definidos a seguir:

type

```
tipofunc = record
    nome:string[50];
    codcargo:string[2];
end;
tipocargo = record
    codcargo:string[2];
    nomecargo:string[50];
    salario:real;
end;
```

A seguir apresentamos um exemplo do arquivo *Funcionarios.dat* e do arquivo *cargos.dat*. Ambos os arquivos podem ter vários elementos.

Nome	Codcargo
José Carlos	Fa
Carlos Alberto	Re
Alberto Roberto	Pr
Roberto José	Pe
⋮	⋮

Codcargo	Cargo	Salário
Fa	Faxineira	100,00
Pe	Pedreiro	110,00
Ca	Carpinteiro	120,00
Pr	Professor	130,00
Re	Reitor	200,00
⋮	⋮	⋮

O campo **codcargo** é um código de cargo composto de duas letras e relaciona o cargo de um funcionário. No exemplo acima, o funcionário Carlos Alberto tem *codcargo* igual a *Re* e procurando no arquivo *Cargos.dat*, vemos que ele tem cargo de *Reitor* e ganha R\$200,00 reais por mês. Faça um procedimento que imprime para cada funcionário (do arquivo *Funcionarios.dat*) seu nome, seu cargo e seu salário.

Exercício 13.5 Suponha que no programa anterior o arquivo *Cargos* tem seus registros sempre ordenados pelo campo *Codcargo*. Faça um procedimento, como no exercício anterior, que imprime para cada funcionário (do arquivo *Funcionarios.dat*) seu nome, seu cargo e seu salário, mas para buscar os dados de um cargo (nome do cargo e o salário) faz uma busca binária no arquivo *Cargos* pelo campo *Codcargo*. Faça rotinas auxiliares para inserção e remoção nestes arquivos.

13.3 Outras funções e procedimentos para manipulação de arquivos

Alguns compiladores oferecem outras rotinas para manipular arquivos. Listamos algumas destas a seguir.

flush(vararq) Flush esvazia o buffer do arquivo (memória auxiliar em RAM) em disco *vararq* e garante que qualquer operação de atualização seja realmente feita no disco. O procedimento Flush nunca deve ser feito em um arquivo fechado.

erase(vararq) É um procedimento que apaga o arquivo em disco associado com *vararq*. Ao dar este comando, o arquivo não pode estar aberto, apenas associado (pelo comando *assign*).

rename(vararq,novonome) O arquivo em disco associado com *vararq* é renomeado para um novo nome dado pela expressão de seqüência de caracteres *novonome*. Ao dar este comando, o arquivo não pode estar aberto, apenas associado (pelo comando *assign*).

filepos(vararq) É uma função inteira que retorna a posição atual do indicador de posição corrente do arquivo. O primeiro componente de um arquivo é 0. Se o arquivo for texto, cada componente é um byte.

Observação: Note que é de responsabilidade do programador garantir a existência do arquivo nomeado por uma string. Se o programa tentar abrir um arquivo para leitura e o mesmo não existir, o programa abortará.

13.4 Exercícios

1. Faça uma rotina que tem como parâmetros dois vetores X e Y , de elementos reais, e um parâmetro inteiro n indicando a quantidade de números em V . Faça uma rotina que escreve os dados de X e Y em um arquivo texto da seguinte maneira: Na primeira linha é impresso o valor n . Em seguida imprima n linhas. Na primeira linha imprima os valores $X[1]$ e $Y[1]$ separados por um espaço em branco. Na segunda linha imprima os valores $X[2]$ e $Y[2]$, assim por diante até os n -ésimos elementos de X e Y .
2. Faça uma rotina que leia um valor n e dois vetores X e Y a partir de um arquivo texto, conforme o exercício anterior.
3. Um grupo de alunos do curso de *Algoritmos e Programação de Computadores* usa um compilador Pascal de linha de comando para compilar seus programas. Entretanto o editor de textos usado por eles não apresenta o número de linhas do programa, ficando difícil para encontrar a linha que contém o erro no programa. Faça um programa para ajudá-los. O programa deve ler o nome de um arquivo texto, *NomeArq* e deve ler o número de uma linha, *nlinha*. O programa deve imprimir na tela as linhas do programa: *nlinha-5, nlinha-4, ..., nlinha, nlinha+1, ..., nlinha+5*. Caso alguma destas linhas não exista, ela não deve ser apresentada. Mostre também o número de cada uma das linhas impressas (com 4 dígitos).

Por exemplo, vamos supor que um compilador de linha tenha dado erro na linha 125 de um programa. Então usamos este programa para imprimir as linhas 120, ..., 130. Como mostrado a seguir:

```
120> var esq,dir,IndicePivo : integer;
121>     Pivo           : real;
122> begin
123>     troca(v[inicio],v[(inicio+fim) div 2]);{escolha de um índice do pivô ini
124>     Pivo := v[Inicio]; {Pivo fica na posição Inicio }
125>     esq=Inicio+1; {primeiro elemento mais a esquerda, pulando o pivô}
126>     dir:=Fim;      {primeiro elemento mais a direita}
127>     while (esq<dir) do begin
128>         while (esq<dir) and (v[esq] <= Pivo) do esq:=esq+1;
129>         while (esq<dir) and (pivo < v[dir]) do dir:=dir-1;
130>         if esq<dir then begin
```

Uma vez que sabemos qual a linha que tem erro (125), podemos dizer que a atribuição `esq=Inicio+1` está errada, e deveríamos consertar para `esq:=Inicio+1`.

Obs.: Note que muitas vezes um compilador diz que há erro em uma linha, quando na verdade o erro ocorreu em alguma linha anterior.

4. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1*, *NomeArq2*, lê o texto que está no arquivo de nome *NomeArq1* e grava em um arquivo de nome *NomeArq2*. O texto gravado deve ser o mesmo texto que foi lido, exceto que em minúsculas e sem acento. Este é um programa útil quando precisamos mandar um texto acentuado por uma rede que traduz os acentos em outros códigos, deixando o texto difícil de ser lido para quem o recebe.
5. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1*, *NomeArq2* e um inteiro k , $k \in \{0, \dots, 25\}$, lê o texto que está no arquivo de nome *NomeArq1* e grava em um arquivo texto de nome *NomeArq2*. O texto gravado deve ser o texto que foi lido criptografado com a cifra de Cesar, com deslocamento k . Considere apenas a criptografia das letras.
6. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1*, *NomeArq2* e um inteiro k , $k \in \{0, \dots, 25\}$, lê o texto que está no arquivo de nome *NomeArq1* e grava em um arquivo texto de nome *NomeArq2*. O texto gravado deve ser o texto que foi lido decriptografado com a cifra de Cesar, com deslocamento k .
7. (Busca de Padrão) Faça um programa que, dado um nome de um arquivo texto: *NomeArq*, e uma string: *padrao*, lê o arquivo texto e imprime (na tela) as linhas que contêm a string *padrao*. Além disso, o programa também

deve imprimir o número da linha impressa (sugestão: imprima o número da linha, a string ' > ' e em seguida a linha do arquivo). No fim do programa, o programa deve imprimir quantas vezes o padrão apareceu no texto (considere repetições em uma mesma linha).

8. Um professor do curso de *Algoritmos e Programação de Computadores* tem um arquivo texto chamado *entrada.txt* contendo as notas dos alunos do seu curso. O arquivo tem várias linhas, uma linha para cada aluno. Cada linha tem o RA do aluno (string de 6 caracteres) e três notas (números reais) separadas por pelo menos um espaço em branco:

RA P_1 P_2 L

A média M é calculada da seguinte maneira: $M \leftarrow (2 \cdot P_1 + 3 \cdot P_2 + 3 \cdot L) / 8$

Se M é tal que $M \geq 5.0$, o aluno está *Aprovado*, caso contrário o aluno está de *Exame*. Faça um procedimento que lê o arquivo *entrada.txt* e gera um outro arquivo texto chamado *saida.txt* contendo uma linha para cada aluno. Cada linha deve ter o seguinte formato:

RA M Resultado

Onde M é a média do aluno e Resultado é a palavra *Aprovado* ou *Exame*, dependendo se o aluno passou ou ficou de exame. Os seguintes dois quadros ilustram o formato destes dois arquivos:

<i>entrada.txt</i>				<i>saida.txt</i>		
991111	7.5	6.5	8.5	991111	7.5	Aprovado
992222	9.0	7.0	9.0	992222	8.25	Aprovado
993333	3.5	4.0	5.0	993333	4.25	Exame
		⋮				⋮

9. **Os exercícios seguintes consideram as declarações feitas na seção 9.4 (exercício 1).** Faça uma rotina, *Grava-CadastroArquivo*, que tem como parâmetros o cadastro e um nome (nome de arquivo). A rotina grava todos os registros do cadastro neste arquivo, que terá o nome especificado no parâmetro.
10. Faça uma rotina, *LeCadastroArquivo*, que tem como parâmetros o cadastro e um nome (nome de arquivo). A rotina lê o cadastro do arquivo com o nome especificado no parâmetro.
11. Faça uma rotina, *InserDeCadastroArquivo*, que insere os funcionários que estão em um arquivo dentro do cadastro. Obs.: Os funcionários anteriores que estão no cadastro são permanecidos.
12. Faça as rotinas que fazem impressão/listagem, inserção, atualização de funcionarios, nos exercícios anteriores, mas agora sem uso do cadastro (vetor). Você deve fazer as operações diretamente no arquivo.