ESTRUTURAS DE DADOS PARA A INTERFACE DE BANCO DE DADOS LOBAN

Antônio Carlos dos Santos

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COM PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.)

Prof. Jano Moreira de Souza

Prof. José Lucas Mourão Rangel Netto

Prof. Estevam Gilberto De Simone

Prof. Gernot Richter

Rio de Janeiro, RJ - Brasil

Abril de 1981

SANTOS, ANTONIO CARLOS DOS

Estruturas de Dados para a Interface de Banco de Dados LOBAN |Rio de Janeiro| 1981.

IX, 208. 29,7 cm (COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1981).

Tese - Univ. Fed. Rio de Janeiro - Fac. Engenharia

1. Banco de Dados I. COPPE/UFRJ II. Estruturas de Dados para a Interface de Banco de Dados LOBAN.

Aos meus pais.

AGRADECIMENTOS

Ao professor Jano Moreira de Souza pela orientação, dedicação e incentivo.

Aos amigos de república Wagner Germano Arnays Destro, Luiz Bertelli Neto e Sérgio de Mello Schneider pela convivência diária, apoio e incentivo.

Aos amigos e colegas do Projeto MINIBAN/COPPE, Antônio Cláudio Gomes de Souza, Beatriz Zakimi Miyasato, Gernot Richter, Jorge Silva Dantas, Paulo Renato Bastos Pinto e Vera Lúcia D'Albuquerque.

Aos amigos e colegas da COPPE, Ludmila Campfield Pereira, Magali Andrade Loureiro, Fernando Bessa Seibel e Miguel Argolo.

À Universidade Federal de São Carlos e especialmente ao Departamento de Computação e Estatística.

Ao Programa Institucional de Capacitação de Docente, PICD-CAPES pelo financiamento.

Ao Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ pela oportunidade a mim confiada, bem como aos seus Professores e Funcionários.

À Denise Schwartz Cupolillo pelo trabalho datilográfico da primeira versão desta tese e a Maria da Graça do Prado Juliano e a Maria José Gualtieri da Costa pelo trabalho final.

RESUMO

Este trabalho apresenta uma série de sugestões de estruturas de dados, para a implementação da Interface de Banco de Dados LOBAN. As estruturas sugeridas estão baseadas em definições e conceitos aqui apresentados.

O capítulo 2 apresenta estas definições e conceitos através de diversos algorítmos, que manipulam informações desde estruturas simples, até complexas e sofisticadas. O capítulo 2 tem a pretensão de ser didático com possível utilização para cursos de graduação em Ciência da Computação.

Para caracterizar as estruturas de dados discutidas, são mostradas as estruturas lógicas da interface LOBAN que é projetada sobre o sistema conceitual de IMC (*I*nformation *Ma*nagement *C*oncepts), desenvolvido por R. Durchholz e G. Richter.

As estruturas lógicas de LOBAN são rearranjadas para refletir as características do banco de dados e de seu uso.

As estruturas de dados apresentadas se referem a um nível de abstração, onde características de hardware não entram em consideração. Nem são incluídas considerações de software dependentes do equipamento a ser escolhido para a implementação do sistema.

ABSTRACT

We present here a series of suggested data structures meant for the implementation of the LOBAN data base interface. The definitions and concepts which were the basis for the choice of those structures are also presented.

Chapter 2 presents those concepts and definitions by means of several algorithms. They handle data structures ranging from very simple to complex and sophisticated ones. Chapter 2 is intended to be clear enough for use in courses meant for Computer Sciences undergraduate students.

In order to characterize the data structures to be analyzed we have also included the logical structures of the LOBAN interface, which was developed under the framework of the IMC (*I*nformation *M*anagement *C*oncepts), of R. Durchholz and G. Richter.

We have rearranged the LOBAN logical structures; they now reflect the characteristics of the database interface and its use.

The data structures form only a general view of the whole system since hardware characteristics were not considered in this work, together with some software characteristics dependent on the equipment to be chosen for each particular implementation.

ÍNDICE

1. INTRODUÇÃO

Neste trabalho discutimos as descrições dos dados e dos seus relacionamentos em dois níveis de abstração que chamamos de nível "lógico" e de nível "físico".

Descrições lógicas dos dados referem-se à forma na qual os dados se apresentam para o usuário do dado. Descrições físicas dos dados referem-se à forma como os dados são gravados no computador. As palavras lógico e físico são usadas para descrever vários aspectos dos dados: lógico refere-se ao modo como o usuário os vê, e físico refere-se ao modo como os dados são gravados num meio de armazenamento.

1.1. Apresentação do Problema

Um sistema de banco de dados, envolvendo armazenamento, recuperação e atualização de informações, tem como fator da sua operação principal, o número médio de acessos ao equipamento de acesso direto que contém os dados para completar uma solicitação.

Dois fatores que afetam o número médio de acessos são: as estruturas para armazenar os dados do usuário e as estruturas internas para armazenar os dados do sistema que serão manipulados através de algorítmos específicos para cada caso.

Uma aproximação mais pragmática do problema é determinar rapidamente: se uma pessoa está cadastrada como usuário do sistema de banco de dados; se um usuário está autorizado a ler, alterar, excluir e ou escrever informações no sistema; se es

tas operações de manipulações na base de dados estão corretas do ponto de vista da informação apresentada com relação à sua entrada; se a segurança e a privacidade estão mantidas no sistema como um todo; se as definições e restrições estabelecidas para a base de dados do usuário são observadas; ... etc.

Assim o que se pretende apresentar é um conjunto de sugestões que satisfaça as questões assinaladas acima em função de estruturas internas de representação com discussões de vantagens e desvantagens, incluindo estruturas para representação dos dados da base de dados do usuário, bem como a estrutura para suportar as definições (*esquema*) da sua aplicação.

1.2. Objetivos do Trabalho

O principal objetivo deste trabalho é apresentar um conjunto de estruturas de dados para suportar as estruturas lógicas da interface de banco de dados LOBAN.

As estruturas de dados apresentadas *somente* vão apresentar uma visão geral das estruturas de representação da informação na base de dados, pois a visão detalhada dependerá em muitos aspectos da escolha de um sistema portador ("hardware" + "software"), o que não é o objetivo do trabalho. Este método faz com que a especificação independa relativamente do computador a ser escolhido para a sua realização física.

É desejável e extremamente importante que as estruturas tenham uma alta eficiência no seu processamento, no uso de memória e na recuperação de informações. Uma organização ineficiente das estruturas de dados do banco de dados pode gerar um grande número de acessos a disco e resultar em tempos impraticáveis de processamento. O número de acessos a disco é fortemente conectado às estruturas de dados e aos algorítmos de sua manipulação.

Por isso o trabalho visa sugerir um conjunto de estruturas de dados, baseado na consideração de vantagens e desvantagens para cada uma delas.

1.3. Desenvolvimento do Trabalho

O trabalho aqui apresentado é uma contribuição ao projeto de pesquisa MINIBAN/COPPE.

A Interface LOBAN foi definida como uma nova interface de comunicação entre usuários e um sistema de gerência de base de dados que permite a definição, manipulação e controle da base de dados em uma única linguagem auto-contida, tendo como principal característica que sua definição parte de requisitos funcionais do usuário, tentando separar nitidamente as fases de definição e projeto da interface (já terminada) e de realização ou implementação da mesma em um sistema portador específico (em andamento).

A interface é definida completamente em (SANTOS[29]) e tem como referências anteriores (RICHTER[8,9]).

1.4. Organização da Tese

A tese foi dividida em capítulos que tratam de assuntos relativamente estanques, mas evolutivos no sentido de apresentarem um relacionamento de pré-requisitos de um para o seguinte.

O capítulo 2 é um resumo didático sobre estruturas de dados, sua organização e sobre os algoritmos que operam sobre elas. Apresenta estruturas elementares como pilhas e filas e suas implementações sequenciais e encadeadas. Estruturas em árvores e suas várias implementações. Algorítmos para criar e manipular estruturas de informação de acesso rápido como pesquisa em árvore binária, árvore n-ária e de "hashing", Estruturas para recuperação por chaves múltiplas como arquivos invertidos. Muitas definições e explicações neste capítulo são adaptações e modificações daqueles nas referências (CATTO[28]), (WIRTH[3]), (SOUZA[5,27]), (KNUTH[1,2]). Outras referências são citadas ao longo do texto.

O capítulo 3 apresenta a interface de banco de dados LOBAN e suas construções lógicas. Vários conceitos básicos são apresentados para compreender a sua terminologia. As construções lógicas apresentadas dão todas as características estruturais da interface. As referências básicas para este capítulo são (RICHTER[8]), (SANTOS[29]).

O capítulo 4 apresenta as sugestões de estruturas de dados para a implementação das estruturas lógicas da interface de banco de dados LOBAN. Tais sugestões são fundamentadas nas estruturas apresentadas no capítulo 2 e nas considerações de vantagens e desvantagens que cada uma delas apresenta.

O capítulo 5 apresenta a conclusão do trabalho e a sua pertinência ao projeto MINIBAN.

2. REVISÃO DA LITERATURA

2.1. Enfoque Adotado

No desenvolvimento de algorítmos, estamos habituados a criar variáveis para representar certas quantidades cujo conhecimento é vital para a solução do problema. Uma abstração nos permite associar a cada valor que uma variável possa assumir um certo significado, que chamamos informação.

Na maioria das vezes, as quantidades com que nossos algorítmos operam não se comportam como elementos independentes, mas guardam entre si importantes relações estruturais. É muito importante que uma representação dessas relações exista também entre as variáveis que representam essas quantidades.

O desenvolvimento de técnicas para a representação estruturada da informação, bem como da sua manipulação, constituem o objetivo deste capítulo.

Os algorítmos necessários são apresentados segundo as técnicas de programação estruturada, sendo apresentados em ALGOL. Na maioria dos casos as declarações das variáveis foram omitidas, por se julgar que seu tipo e significado já estão perfeitamente determinados.

2.2. *Terminologia Geral Adotada*

As estruturas de dados receberão a designação geral de listas. Quando as propriedades estruturais dos elementos de uma lista decorrem essencialmente de sua posição relativa ao longo de uma dimensão, a *lista* será dita *linear*. *Pilhas* e *filas* constituem as listas lineares mais comuns. Árvores constituem o exemplo mais importante de listas não lineares.

O fator que determina o arranjo de um conjunto de elementos numa estrutura é a existência de uma ou mais qualidades ou atributos comuns. Por outro lado, esses elementos poderão possuir outros atributos que interessem ao problema e que sejam comuns. Por exemplo, numa fila de teatro todos têm em comum a intenção de assistir à uma peça; mas o nome, a idade, a altura, o peso, etc., variam de pessoa para pessoa na fila.

Cada elemento que participar de uma estrutura será representado na lista por um nó ou registro. Esse elemento terá uma série de atributos que interessam ao problema e que devem ser guardados e outros que não. Por exemplo, para se calcular a idade média das pessoas que estão na fila do teatro, basta anotar a idade de cada uma; não é necessário perguntar o nome, o endereço, o peso, etc.

Cada informação necessária sobre um elemento da lista será representada num campo do registro correspondente, dessa forma cada registro da lista terá um ou mais campos de dados.

Eventualmente usaremos um ou mais campos de um registro para referenciar (ligar) com outros registros da lista e, dessa forma, conseguir representar estruturas mais complexas. Os dados contidos nesses campos de referência (ligação) serão

denominados *ponteiros*.

Resta saber como serão indicados o começo e o fim da lista. A solução natural será criar uma variável que aponte para o primeiro registro. Este normalmente terá um ponteiro indicando o segundo, que terá um ponteiro indicando o terceiro e assim sucessivamente. O ponteiro do último registro não poderá indicar ninguém. Isso será conseguido fazendo-se com que esse ponteiro assuma um valor pré-estabelecido (que não seja um endereço válido), para indicar a ausência de ligação. Essa ausência de ligação vazia aparecerá nos diagramas como ⏚ (ligação com a terra) e no texto como Λ. Na ausência de registros PONT tem o valor Λ.

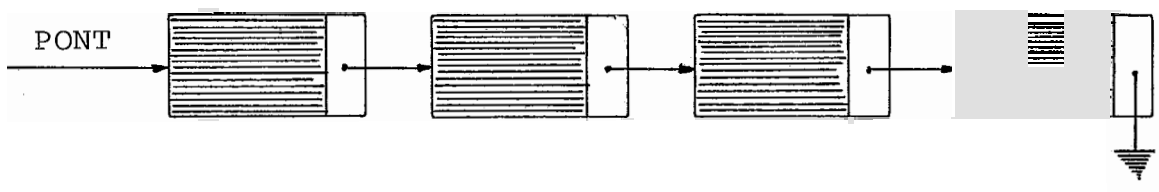


Fig. 2.1. - Representação Esquemática de uma Lista Linear

Na figura 2.1, PONT é o ponteiro externo indicando o começo da lista, cada retangulo representa um registro (a parte hachurada representa os campos de dados e a parte clara o campo de ponteiro), e o último registro está ligado à terra, indicando o fim da lista.

Um ponteiro tem a sua representação realizada internamente por um conjunto de bits.

2.3. *Listas Lineares*

2.3.1. *Introdução*

Ao escolher a forma de representação dos elementos de uma certa estrutura de informação em um computador deve-se considerar dois aspectos principais: as qualidades ou atributos relevantes desses elementos e o tipo de operações que deverão ser realizadas sobre a estrutura, os atributos relevantes determinarão o número e a constituição dos campos de dados necessários. O tipo de operação permitirá a escolha adequada dos campos de ligação.

Uma lista linear ocupará um vetor de registros de dimensão $[1..M]$, onde M será escolhido de forma compatível com o comprimento previsto para a lista e os registros terão seus campos adequadamente definidos.

Deverão ser desenvolvidos algorítmos capazes de efetuar algumas operações básicas sobre essas listas, como por exemplo:

a) Incluir um novo registro em determinada posição;
b) Excluir um determinado registro;
c) Ordenar os registros de uma lista pelo conteúdo de um ou mais de seus campos;
d) Selecionar dentre os registros de uma lista aqueles que satisfazem a determinadas condições quanto ao conteúdo de um ou mais campos, etc.

Uma das propriedades estruturais mais relevantes de

uma lista é a forma de acesso para inclusão ou exclusão de registros. Por isso as listas lineares em que essas operações são possíveis (somente nas extremidades), recebem tratamento e designação particulares:

a) Uma pilha é uma lista linear em que a inclusão e a exclusão de registros somente podem ocorrer em uma das extremidades;

b) Uma fila é uma lista linear em que a inclusão de registros é feita numa extremidade e a exclusão na outra;

c) Uma fila de duas pontas é uma lista linear em que se permite a inclusão ou exclusão de registros em qualquer uma das extremidades. Uma fila de duas pontas pode ainda ser de entrada restrita, quando se permite apenas numa extremidade e a exclusão em ambas, ou de saída restrita quando se permite a inclusão em ambas as extremidades e a exclusão somente em uma delas.

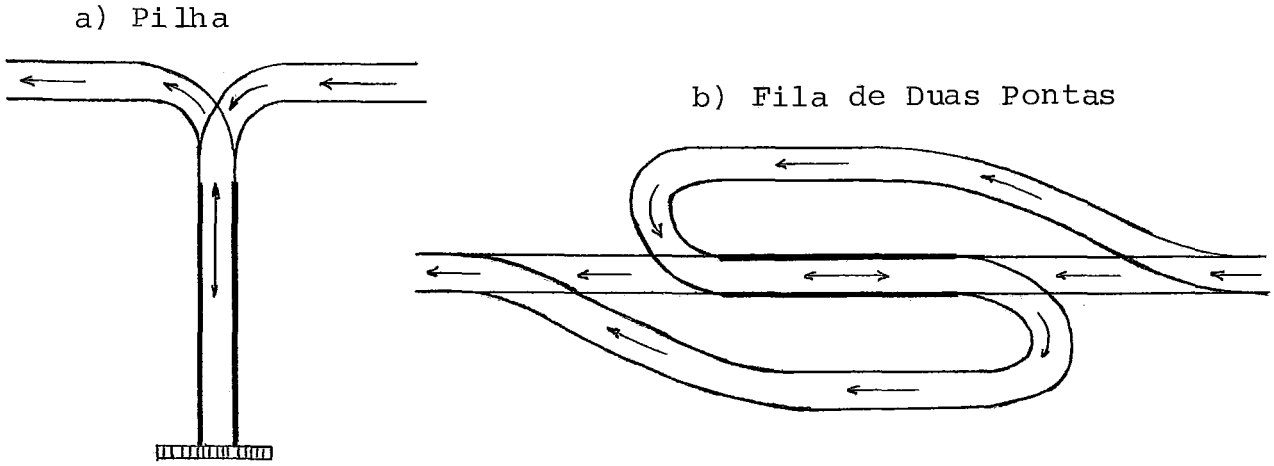


Fig. 2.2. - Analogia Proposta por DIJKSTRA

Dijkstra propôs uma analogia com os trilhos de uma estrada de ferro para tornar mais claras essas definições. A partir da figura 2.2 (b) pode-se obter uma fila de duas pontas de entrada restrita suprimindo-se o trilho superior, e de saída restrita suprimindo-se o trilho inferior.

Em geral as extremidades ativas dessas listas também recebem nomes específicos, compatíveis com as analogias feitas. Desse modo o primeiro registro de uma pilha (que é o único a que se tem acesso) é o *topo* da pilha e o último a *base*; o primeiro registro é o *começo* da fila e o último o *fim*; numa fila de duas pontas a idéia de primeiro e último perde o significado e fala-se então em *extremidade direita* e *extremidade esquerda* da fila.

Quando for o caso, nas pilhas e filas a ligação entre os registros será estabelecida por um único ponteiro orientado do começo para o fim da lista. Nas filas de duas pontas, no entanto, a eficiência das operações exigirá a existência de dois ponteiros, um orientando da esquerda para direita e outro no sentido oposto.

2.3.2. Alocação Sequencial

A maneira mais cômoda para se representar uma lista linear na memória é ocupando sequencialmente o vetor $L\ [1..M]$ reservado para ela.

De uma maneira geral, uma lista será delimitada por dois ponteiros *P1* e *P2*, tais que:

(A) *P1* aponta o índice que antecede o início da lista em L;

(B) *P2* aponta o índice do último registro da lista em L.

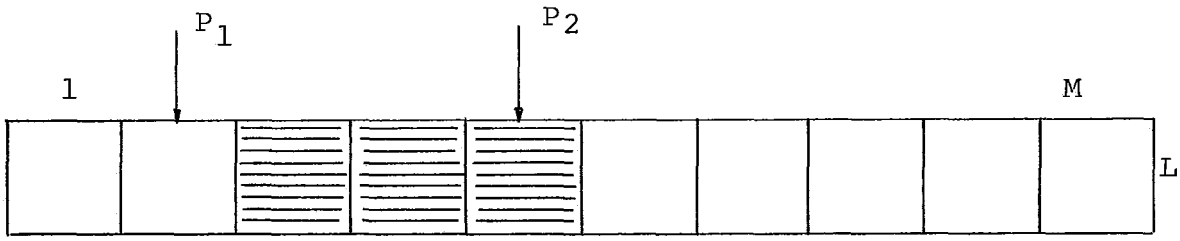


Figura 2.3. - Lista Alocada Sequencialmente

Nessa representação a situação de lista vazia é conseguida fazendo-se $P1 = P2$.

Esta forma de alocação de espaço é muito conveniente para a representação de pilhas. Nesse caso, admitindo-se a base da pilha à esquerda (em *P1*), e o topo à direita (em *P2*), faremos $P1 = \emptyset$, e chamaremos *P2* de T, o ponteiro do topo da pilha. Inicialmente, com a pilha vazia, teremos $P2 = P1$, isto é,

$T = \emptyset$.

A inclusão de um novo resgistro da pilha será conseguida com:

$$T := T + 1; \qquad L[T] := X \tag{1}$$

A exclusão do registro que se encontra no topo da pilha será conseguida com:

$$X := L[T]; \qquad T := T - 1 \tag{2}$$

No caso de uma fila, chamaremos $P1$ de C, o começo da fila, e $P2$ de F, o fim da fila. No início teremos $P1 = P2 = \emptyset$, isto é, $C = F = \emptyset$.

A inclusão de um novo registro no fim da fila será conseguida com:

$$F := F + 1; \qquad L[F] := X \tag{3}$$

E a exclusão do registro no começo da fila será feita por:

$$C := C + 1; \qquad X := L[C] \tag{4}$$

No algoritmo (4), observe que devido ao seu posicionamento, C deve ser incrementado antes de ocorrer a eliminação do registro.

Ao passo que a pilha anteriormente estudada tem sua base permanente presa ao índice $\emptyset$ do vetor L, enquanto o topo

percorre o vetor para cima e para baixo (nas inclusões e exclusões, respectivamente), no caso de uma fila ambos os ponteiros se movimentam, e sempre para cima, de acordo com (3) e (4).

Considere a sequência *IEIEIEIE ...*, onde *I* significa "incluir um registro na fila" e *E* significa "excluir um registro da fila". A fila correspondente a esse código de operação terá sempre um ou nenhum registro. No entanto, os ponteiros *C* e *F* percorrerão o vetor *L* sempre acima, logo provocando um problema de *estouro*.

Esse estouro pode ser evitado nesse caso, se observarmos que sempre que a fila se esvazia os ponteiros *C* e *F* podem ser reinicializados. Incorporando essa melhoria (4) se transforma em:

```
C:= C + 1; X:= L[C]; if C = F  then begin
                                      C:= ∅;
                                      F:= ∅
                                      end                    (4')
```

Essa solução no entanto não resolve o problema criado por *IIEIEIEIE ...*, que continua provocando o estouro do vetor *L*, apesar de ter sempre somente um ou dois elementos na lista. Uma análise um pouco mais elaborada nos sugere considerar o vetor *L* como um anel, de modo a franquear ao ponteiro *F* o espaço liberado pelo avanço de *C*.

Assim supondo a adjacência dos índices *M* e *1*, e que inicialmente tenhamos *C = F = M*, a inclusão de um registro passará a ser feita por:

```
if  F = M  then  F:= 1
           else  F:= F + 1;
L[F]:= X                                    (5)
```

E a exclusão será feita por:

```
if  C = N  then  C:= 1
           else  C:= C + 1;
X:=[L]C                                     (6)
```

Observamos que no momento em que *L* passa a ser considerado como um anel desaparece o interesse pela reinicialização de *C* e *F*, como se fazia em (4'). Assim, os algorítmos desenvolvidos para pilhas, (1) e (2), e os desenvolvidos para filas, (5) e (6), supoem que:

(A) No caso de inclusão ainda existe espaço disponível em *L*;

(B) No caso de exclusão ainda existe pelo menos um registro na lista.

Evidentemente essas situações não serão sempre encontradas, a não satisfação de (A) causará a ocorrência de OVERFLOW *(estouro por excesso)* do vetor *L*, e a não satisfação de (B) causará UNDERFLOW *(estouro por falta)*.

Prevendo esses problemas criaremos dois procedimentos, um para OVERFLOW e o outro para UNDERFLOW, que se encarregarão de processar essas situações e incluiremos nos algorítmos desenvolvidos, as chamadas necessárias.

Assim os algorítmo mencionados, (1), (2), (5) e (6),

transformar-se-ão em:

```
T:= T + 1; if  T > M then OVERFLOW
                     else L[T]:= X                    (1')

if T = ∅  then UNDERFLOW
          else begin  X:= L[T];
                      T:= T - 1
               end                                    (2')

if  F = M  then  F:= 1 else F:= F + 1;
if  F = C  then  OVERFLOW else  L[F]:= X              (5')

if C = F then UNDERFLOW
         else begin
              if C = M then C:= 1
                       else C:= C + 1;
              X:= L[C]
              end                                     (6')
```

O procedimento de UNDERFLOW será chamado quando se tentar excluir da lista um registro inexistente, o que geralmente é um recurso usado pelo programador para controlar o fluxo de um programa, por exemplo, excluir registro por registro de uma pilha até que ela acabe, isto é, até que ocorra UNDERFLOW. Por outro lado, OVERFLOW geralmente representa uma situação de erro: A tentativa de incluir um novo registro onde já não há mais espaço. Como consequência, na maioria dos casos, UNDERFLOW simplesmente retornará um indicador que possa

ser testado pelo programa, enquanto OVERFLOW forçará o término da execução do programa, eventualmente acompanhado de uma mensagem de erro.

Na maior parte das vezes estaremos trabalhando não só com uma, mas com várias listas simultâneas, e aí, nosso programa deve permitir que o espaço disponível seja livremente utilizado por qualquer uma delas, na medida de suas necessidades de crescimento. Essa política visa o retardamento da ocorrência de OVERFLOW, por permitir que uma lista que se desenvolva mais rapidamente utilize o espaço disponível em outra.

Um caso simples de transferência de espaço ocorre quando temos duas pilhas. Aí em vez de declararmos dois vetores *L1[1..M1]* e *L2[1..M2]*, para representá-las, podemos declarar um único vetor *L[1..M]*, *M >= M1 + M2*, e fazer a base da primeira pilha coincidir com o índice Ø, a base da segunda coincidir com *M + 1*, e fazê-las crescerem em sentido contrários.

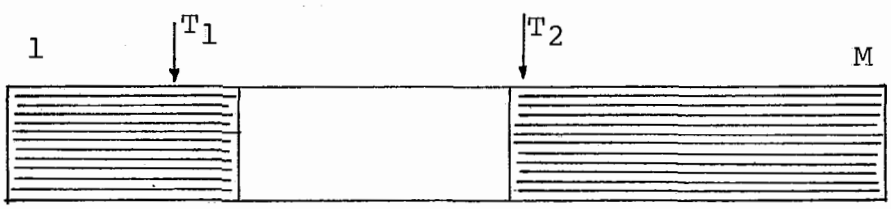


Figura 2.4. - Duas Pilhas Alocadas sobre o mesmo vetor

Observando a figura 2.4 podemos concluir que o espaço

disponível em uma das pilhas é franqueado ao crescimento da outra e que a exclusão de um registro em qualquer uma delas retorna o espaço ao bolo comum.

Esse processo garante:

(A) OVERFLOW só ocorre se o número total de registros nas duas pilhas se tornar maior que M;

(B) A base de cada pilha permanece fixa numa posição determinada do vetor.

Verificamos que essa situação favorável deixa de existir no caso de serem três ou mais pilhas.

Nessa situação, como (A) é a condição que realmente interessa satisfazer, (B) deverá ser relaxada. Isto quer dizer que as bases das pilhas não mais serão fixas, mas poderão mudar de posição dentro do vetor.

Vamos analisar a situação no caso em que o vetor $L[1..M]$ deverá conter N pilhas. Será suposta a existência de dois vetores internos TOPO e BASE que contenham em TOPO$[I]$ e BASE$[I]$ a posição atual do topo e da base da I-ésima pilha, respectivamente.

O algorítmo para a inclusão de um registro X na pilha I seria:

```
TOPO[I]:= TOPO[I] + 1;
if TOPO[I] > BASE[I + 1] then OVERFLOW
                         else L[TOPO[I]]:= X        (7)
```

E o algorítmo para a exclusão do registro no topo da pilha I seria:

```
if TOPO[I] = BASE[I]
   then UNDERFLOW
   else begin
        X:= [L[TOPO[I]];
        TOPO[I]:= TOPO[I] - 1
        end                                    (8)
```

Com relação ao algorítmo (7) cabem as seguintes observações:

(A) A condição de pilha I vazia agora se obtem quando $TOPO[I] = BASE[I]$;

(B) É necessário que para cada pilha I exista uma pilha $I + 1$. Isto quer dizer que teremos que providenciar uma pilha fictícia $N + 1$, tal que $BASE[N + 1] = M$, de modo a garantir o funcionamento do algorítmo para $I = N$;

(C) A indexação de L, que será feita diretamente através do ponteiro T, agora deverá ser feita indiretamente através do vetor $TOPO[I]$, tornando a execução do algorítmo um pouco mais demorada;

(D) Sempre que o topo da pilha I alcançar a base da pilha $I + 1$, ocorre OVERFLOW da pilha I. Esse OVERFLOW não é uma situação tão grave agora, porque o espaço ocioso que existir em outra pilha poderá ser cedido à pilha I. Nosso problema passa a ser então de escrever um algorítmo capaz de localizar um espaço ocioso em L e cedê-lo à pilha I, eliminando a sua condição de OVERFLOW.

Esse raciocínio sugere a substituição da chamada de OVERFLOW em (7) pela de um outro procedimento que realoque as pilhas se for possível ou chame OVERFLOW em caso contrário. Assim teremos:

```
TOPO[I]:= TOPO[I] + 1;
if TOPO[I] > BASE[I + 1] then REALOCAR;
L[TOPO[I]]:= X                                    (7)
```

Observamos que a operação de inclusão do elemento *X* não mais está condicionada ao resultado do teste, mas ao retorno do controle do procedimento de realocação.

O algoritmo proposto abaixo como ponto de partida, supõe que os vetores *BASE* e *TOPO* tenham sido inicializados da seguinte maneira:

```
for I:= 1 to M
    do begin
       BASE[I]:= ∅;
       TOPO[I]:= ∅
       end;
BASE[N + 1]:= M
```

Observamos que, no início, a inclusão de um elemento em qualquer pilha *I*, $I < N$, provocará OVERFLOW.

Para contornar essa situação tentaremos pela ordem as três possibilidades apresentadas a seguir:

(A) Procurar o menor *K*, $I < K \leq N$, tal que $TOPO[K] < BASE[K + 1]$, e mover todos os registros um índice acima:

```
for J:= TOPO[K] step - 1 until BASE[I + 1]+ 1
    do  L[J + 1]:= L[J];
for J:= I + 1 to  K
    do begin
       BASE[J]:= BASE[J + 1] + 1;
       TOPO[J]:= TOPO[J] + 1
       end
```

É interessante notar que se $K = I + 1$ e se essa pilha estiver vazia, haverá somente uma atualização dos ponteiros, sem movimentação alguma de registros.

(B) Se não existir K algum que satisfaça (s), procurar o maior K, $1 < K \leq I$, tal que $BASE[K] > TOPO[K - 1]$, mover todos os registros um índice abaixo:

```
for J:= BASE[K] + 1 to TOPO[I] - 1
    do L[J - 1]:= L J ;
for J:= K to I
    do begin
       BASE[J]:= BASE[J] - 1;
       TOPO[J]:= TOPO[J] - 1;
       end
```

(C) Se não encontrar K que satisfaça (A) ou (B), isto é, se $TOPO[K] = BASE[K + 1]$, para todo $K \neq 1$, então o vetor L já estará totalmente ocupado e não haverá outro recurso senão chamar OVERFLOW e desistir.

Evidentemente, esse método de realocação exige substancial manipulação de dados, em virtude do grande número de

estouros de pilhas que podem ocorrer. Essa desvantagem poderá ser atenuada por:

(A) Uma alocação inicial das bases mais compatível com o comportamento esperado para cada pilha, reservando algum espaço que evite os estouros iniciais;

(B) Na realocação das pilhas, quando ocorrer 1 OVERFLOW, mover os registros algumas posições ao invés de apenas uma, abrindo lugar para possíveis inclusões futuras.

Como exemplo, no caso de as pilhas terem aproximadamente a mesma probabilidade de crescimento, a primeira opção poderia ser realizada por:

```
DELTA:= M/N;
for I:= 1 to N
    do begin
       J:= [I - 1]* DELTA + 1;
       BASE[I]:= J;
       TOPO[I]:= J
       end                                          (9)
```

No caso do item (B), rearrumar quando ocorrer 1 overflow podemos utilizar o algorítmo a seguir cuja forma original foi proposta por JAN GARWICK: ele usa um vetor TOPOANTERIOR[*I*] para reter a posição do topo da pilha *I* quando da última realocação e emprega essa informação para fazer uma distribuição ponderada do espaço ocioso entre as pilhas.

Essencialmente, o algorítmo substituirá a chamada de realocar em (7); é composto dos seguintes passos:

```
/* ALGORÍTMO DE GARWICK */
begin
Calcular o espaço disponível em L;
if Há espaço disponível em L
   then begin
        Calcular fatores para distribuição do espaço;
        Reposicionar as listas e atualizar os ponteiros
        end
   else OVERFLOW
end
```

```
/* Calcular o espaço disponível em L */
begin
/* Espaço = espaço total disponível
   Crescimento = crescimento total das pilhas
                 desde a última chamada */
ESPAÇO:= M; CRESCIMENTO:= Ø;
for J:= 1 to N
    do begin
       ESPAÇO:= ESPAÇO - (TOPO[J] - BASE[J]);
       DELTA[J]:= TOPO[J] - TOPOANTERIOR[J];
       if DELTA[J] < Ø then DELTA[J]:= Ø;
       CRESCIMENTO:= CRESCIMENTO + DELTA[J]
       end
end;
```

```
/* Há espaço disponível em L */
begin
 ESPAÇO = ∅
end


/* Calcular fatores para distribuição do espaço */
begin
/* 10% do espaço ocioso será distribuido uniformemente
   e 90% proporcionalmente ao crescimento de cada pilha */
UNIF:= .1* ESPAÇO/N;
PROP:= .9* ESPAÇO/CRESCIMENTO
end


/* Reposicionar as listas e atualizar os ponteiros */
begin
/* Atualizar as bases */
BASENOVA [1]:= BASE[1];
for j:= 2  to N
   do BASENOVA[J]:= BASENOVA[J-1] + TOPO[J-1] - BASE[J-1]
                    + UNIF + PROP* DELTA [J-1];
```

```
/* Reposicionar as listas */
J:= 2;
while J <= N
do
if BASENOVA[J] <  BASE[J]
 then begin /* MOVER a lista para baixo */
      DESLOC:= BASE[J] - BASENOVA[J];
      for K:= BASE[J] + 1  to TOPO[J]
        do L[K - DESLOC]:= L[K];
      BASE[J]:= BASENOVA[J]; TOPO[J]:= TOPO[J]- DESLOC;
      J:= J + 1
      end
 else if BASENOVA[J] >  BASE[J]
         then begin /* localizar o topo do deslocamento */
              T:= J;
              while BASENOVA[T + 1]> BASE[T + 1] do T:= T + 1;
              S:= T; /* Mover as listas para cima */
              while  T >= J
              do begin
                 DESLOC:= BASENOVA[T] - BASE[T];
                 for K:= TOPO[T] step - 1 until BASE[T] + 1
                   do L[K + DESLOC]:= L[K];
                 BASE[T]:= BASENOVA[T];
                 TOPO[T]:= TOPO[T] + DESLOC;
                 T:= T - 1
                 end;
              J:= S + 1
              end;
/* Guardar a posição dos topos para a próxima */
```

```
for J:= 1 to N do TOPOANTERIOR[J]:= TOPO[J]
end;
```

Os algorítmos desenvolvidos para pilhas poderão ser adaptados para outras estruturas lineares como filas cujos registros estejam contidos entre $BASE[I]$ e $TOPO[I]$.

Segundo KNUTH[01], o algoritmo de GARWICK mostrou ser eficiente para ocupação parcial do vetor L. À medida que essa ocupação tende a ser total o tempo gasto com o reposicionamento das listas para cima e para baixo começa a onerar muito a execução do programa, devido aos frequentes OVERFLOWS.

Uma alternativa seria não esperar pela ocupação total de L e considerar OVERFLOW sempre que o espaço ocioso ficar abaixo de um certo limite mínimo. Essa idéia encontra suporte no fato de que, geralmente, um programa que chega muito próximo do estouro realmente acaba estourando logo depois.

2.3.3. Alocação Encadeada

A sequencialização dos registros de uma lista, que é implícita na alocação sequencial, pode ser indicada explicitamente pelo uso de ponteiros. Uma lista em que a sequencialização é explicitamente indicada é chamada de encadeada.

Para processar uma lista em alocação *encadeada*, necessitaremos de um ponteiro externo que indique o primeiro registro da lista. Este indicará o segundo, que indicará o terceiro, etc., até que um ponteiro vazio no último sinalize o fim da lista. Veja figura 1.

Fazendo-se uma comparação entre alocação sequencial e a encadeado, podemos concluir entre outras coisas, que:

(A) A alocação encadeada requer memória adicional para os ponteiros. Embora este fato possa ser decisivo em certos casos, em geral ele é compensado pela possibilidade de superposição de tabelas partilhando registros comuns, e pela memória que deve ser deixada livre de qualquer modo, para o bom funcionamento dos algorítmos de reposicionamento de listas alocadas sequencialmente.

(B) As operações sobre listas encadeadas são muito mais simples: a exclusão de um registro, por exemplo, requer somente a alteração de um ponteiro, e a inclusão de um novo registro requer a alteração de dois ponteiros. Essas operações seriam bem mais demoradas no caso de listas sequenciais.

(C) O tempo de acesso ao k-ésimo registro de uma lista sequencial é uma constante, mas é proporcional a K numa lista encadeada. Como em geral as listas serão processadas de modo sequencial, e não aleatório, esse problema se atenuará. Quando o acesso aleatório for frequente ou desejável, deverão ser providenciados ponteiros intermediários que reduzam o número de buscas necessárias à localização do registro desejado.

(D) A alocação encadeada torna mais simples a reunião de várias listas numa única ou a partição de uma lista em várias outras.

(E) O esquema encadeado permite a representação de estruturas mais complexas: pode-se ter um número variável de listas de comprimento variável e cada registro de uma lista pode ser o ponto de partida para outra, os mesmos registros podem ser ligados em várias ordens correspondendo a diferentes listas, etc.

(F) Devido ao acesso direto, o percurso numa linha sequencial é geralmente mais rápido do que numa lista encadeada.

Exceto onde ficar explícita outra coisa, admitiremos que os registros do vetor $L[1..M]$, que conterá a lista, possuam dois campos: um deles - INFO - conterá as informações necessárias e o outro - LIG - fará a ligação com o próximo registro da lista.

Como a ocupação do vetor *L* será aleatória (isto é, após algumas inclusões e exclusões de registros desaparecerá qualquer sequencialização eventual), deveremos ter uma forma de identificar, quais posições do vetor estão ocupadas e quais estão livres, de uma forma rápida quando precisarmos reutilizar uma posição. Em geral reunem-se as posições livres numa outra lista especialmente criada, chamada lista ou pilha do espaço disponível (DISPO). A designação pilha aplica-se porque DISPO será geralmente tratada desta forma.

Como as posições disponíveis estão todas encadeadas, e como a variável DISPO indica a primeira delas, a obtenção do índice da próxima posição utilizará *J* do vetor *L* poderá ser conseguida por:

```
/* requisita um registro disponível */
J:= DISPO;
/* o topo da pilha DISPO abaixa */
DISPO:= L[DISPO]. LIG                                    (1)
```

Observamos que o que se fez nada mais foi do que excluir o primeiro elemento de uma pilha, cujo topo é indicado por *DISPO*. Como (1) ocorre muito frequentemente nós o indicaremos por:

X:= DISPO

Quando um registro na posição *J* for excluido, esse índice deve ser devolvido à pilha de espaço disponível. Assim faremos:

```
/* o topo da pilha sobe */
 L[J]. LIG:= DISPO;
/* DISPO recebe um novo registro */
 DISPO:= J                                          (2)
```

Dado a frequência que (2) ocorre nós o indicaremos por:

$$DISPO::= J$$

Até o momento não se disse como criar inicialmente a pilha *DISPO*, nem se cogitou da inexistência de elementos em *DISPO* quando (1) é executado.

O problema de se criar inicialmente a pilha *DISPO* pode ser resolvido pelo seguinte algoritmo, que reune na pilha *DISPO* todo o espaço disponível em *L*:

```
DISPO:= 1;
for I:= 1 to M - 1 do L[I]. LIG:= I + 1
L[M]. LIG:= Λ                                       (3)
```

A requisição de um registro de *DISPO* quando a pilha estiver vazia, exigirá uma redefinição de (1), uma vez que o esvaziamento da pilha DISPO significará ausência de espaço disponível, e, consequentemente, o estouro (OVERFLOW) do vetor *L*. Levando em conta essa possibilidade teremos que *J::= DISPO* na realidade corresponderá à:

```
if DISPO = Λ then OVERFLOW
                 else begin
                      J:= DISPO; DISPO:=L[DISPO]. LIG
                      end
```

O OVERFLOW mais uma vez poderá ser tratado de duas formas:

(A) Encerrando o programa;

(B) Chamando uma rotina de "COLETA DE RESTOS" (GARBAGE COLECTION) para tentar conseguir mais espaço.

Na realidade, nem sempre podemos determinar quanto do vetor *L* poderá ser destinado à pilha *DISPO*, considerando que *L* poderá conter mais do que uma lista ao mesmo tempo.

O raciocínio desenvolvido abaixo elimina o tempo gasto com a inicialização em (3):

(A) Inicialmente fazemos:

DISPO:= Λ; MAXDISPO:= J_0 - 1

onde MAXDISPO representa o maior índice já atingido pela pilha DISPO e que deve estar entre J_0 e *JMAX*.

(B) A operação *J:= DISPO*

```
if DISPO ≠
   then begin
        J:= DISPO; DISPO:= L[DISPO]. LIG
        end
   else begin
```

```
MAXDISPO:= MAXDISPO + 1
if MAXDISPO    JMAX then OVERFLOW
                    else J:= MAXDISPO
end                                          (5)
```

(C) Mantem-se *DISPO:;= J* como em (2).

Ficando clara a viabilidade de um eficiente controle sobre a memória disponível, passaremos a discutir a representação de algumas estruturas simples.

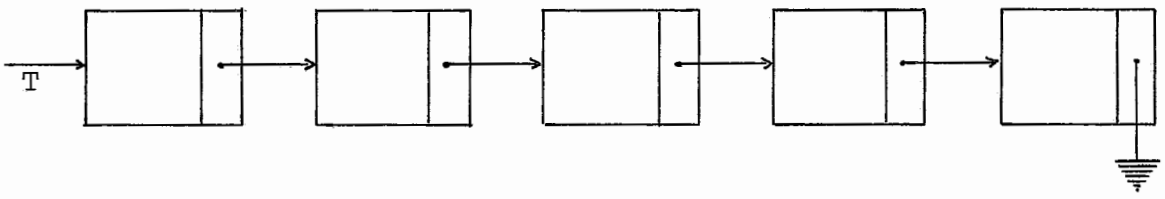


Figura 2.5.- Pilha Encadeada

A figura 2.5 mostra uma pilha, com um ponteiro *T* indicando o seu topo. Quando a pilha estiver vazia, esse ponteiro terá o valor Λ.

Para incluir um novo registro *X* no topo da pilha precisaremos de uma variável auxiliar *TEMP:*

TEMP:= DISPO; L[TEMP]. INFO:= X; L[TEMP]. LIG:= T; T:=[TEMP]

(6)

E para excluir o registro do topo da pilha faremos:

```
if T = Λ
    then UNDERFLOW
    else begin
         TEMP:= T; T:=[TEMP].LIG; X:= L[TEMP].INFO;
                DISPO:= TEMP
         end                                          (7)
```

A alocação encadeada é particularmente conveniente à representação de filas, onde se assemelha ao caso da pilha, exceto que agora teremos dois ponteiros *C* e *F* para indicarem o começo e o fim da fila, respectivamente.

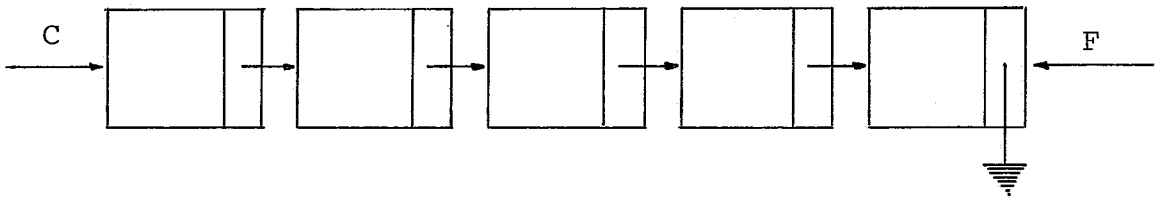


Figura 2.6. - Fila Encadeada

Sempre que projetamos as operações sobre uma lista deveremos ter um cuidado especial com o caso de lista vazia. O processamento inadequado dessa condição e a incorreta manipulação das ligações são os erros mais frequentes nos programas que trabalham com alocação encadeada. Para evitar o primeiro, examinaremos cuidadosamente as condições externas e para evitar o segundo, desenharemos diagramas das situações "antes" e

"depois" que evidenciem as ligações que necessitam ser modificadas.

A figura 2.7. mostra a fila representada na figura 2.6. após a inclusão de um registro *X*.

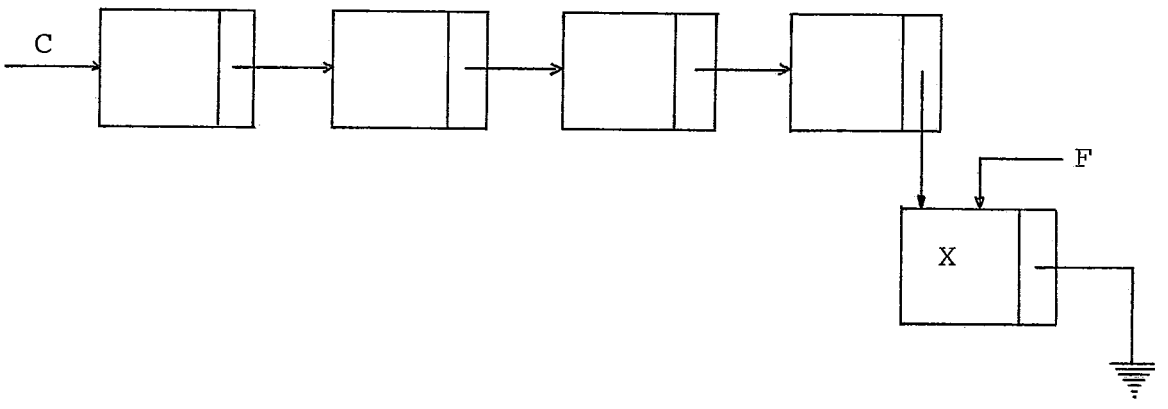


Figura 2.7. - Inclusão de Registro em Fila Encadeada

Comparando as figuras 2.6 e 2.7 temos o necessário ao desenvolvimento do seguinte algorítmo:

$$TEMP := DISPO;\ L[TEMP].LIG := \Lambda;\ L[F].LIG := TEMP;\ F := TEMP;$$
$$L[TEMP] := INFO \qquad (8)$$

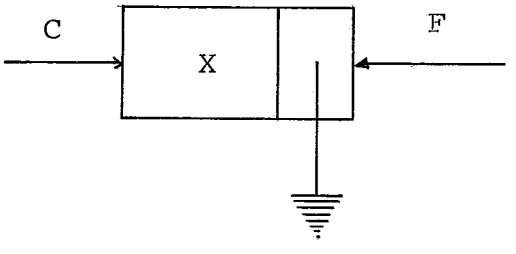


Figura 2.8. - Fila Encadeada com um Registro

Considerando a situação $C = F = \Lambda$ como o indicador de pilha vazia, notamos que a atribuição $L[F].\ LIG := TEMP$ não pode ser executada neste caso, e em seu lugar devemos ajustar adequadamente o valor de C.

Assim teremos:

```
TEMP:= DISPO; L[TEMP]. LIG:= Λ
if  F = Λ  then  C:= TEMP
           else  L[F]. LIG:= TEMP;
F:= TEMP;
L[TEMP]. INFO:= X;                                    (8')
```

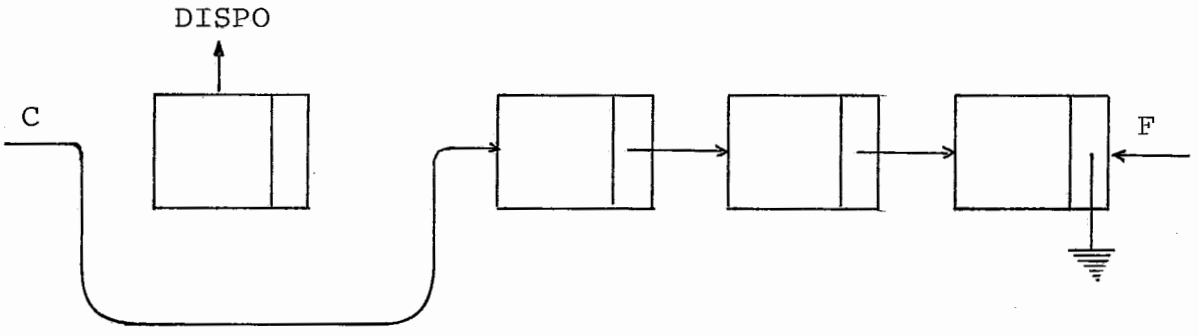


Figura 2.9. - Fila após a Exclusão do Registro do Começo

A operação de eliminação do registro que se encontra no começo da fila será conseguida pelo algorítmo abaixo, já levando em consideração a possibilidade de UNDERFLOW.

```
if  C = Λ then UNDERFLOW
          else begin
               TEMP:= C;
               C:= L[TEMP]. LIG; X:= L[TEMP]. INFO;
               DISPO:= TEMP;
               if  C = Λ then F:= Λ
               end                                    (9)
```

Deve-se ressaltar que os algoritmos apresentados aqui não representam a única maneira de se automatizar o processamento dessa lista. Outras formas alternativas serão apresentadas em outras seções.

2.3.4. *Listas Circulares*

As listas circulares são listas encadeadas cuja propriedade é que ao invês do último registro apontar para a terra (⏚) ele aponta para o primeiro registro à esquerda. Desta forma é possível acessar todos os registros da lista, iniciando por qualquer ponto dado. Doravante chamaremos registro de nó.

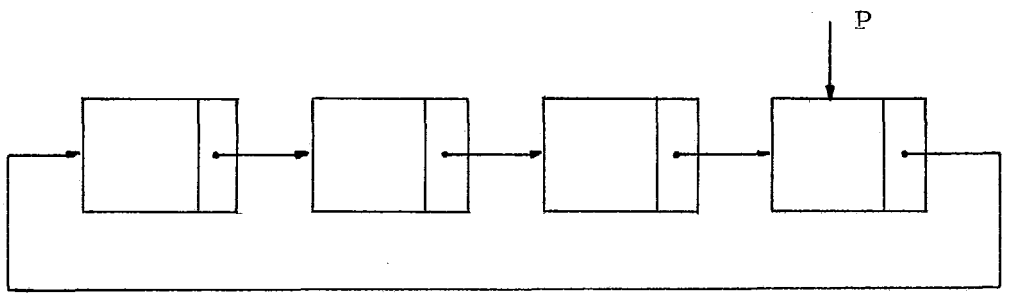


Figura 2.9. - Lista Circular (espaço linear)

Assumindo que os nós tenham dois campos, *INFO* e *LIG*, como na seção anterior. Há um ponteiro *P* que indica o nó mais à direita da lista, e *L*[*P*]. *LIG* indica o nó mais à esquerda.As operações mais importantes são.

(A) Inclusão no fim:

```
TEMP::= DISPO;
L[TEMP]. LIG:= L[P]. LIG;
L[P]. LIG:= TEMP;
L[TEMP]. INFO:= X                    (1)
```

(B) Inclusão no começo:

inclusão no fim;

P:= TEMP (2)

(C) Exclusão no fim:

TEMP:= L[P]. LIG;

L[P]. LIG:= L[TEMP]. LIG;

X:= L[TEMP]. INFO;

DISPO::= TEMP (3)

Devemos lembrar que não foi considerado o caso de listas vazias. Se por exemplo a operação (C) for aplicada 4 vezes na figura 1, nós teremos o ponteiro *P* apontado para um nó da lista DISPO, e isso pode trazer problemas.

Se fizermos *P* igual a Λ no caso de lista vazia, nós podemos solucionar as operações anteriores com a adição de novas instruções.

(A') Inclusão no fim:

```
TEMP::= DISPO;
if p ≠ Λ then begin
                  L[TEMP]. LIG:= L[P]. LIG;
                  L[P]. LIG:= TEMP
                  end
              else begin
                  L[TEMP]. LIG:= TEMP;
                  P:= TEMP
                  end;
L[TEMP]. INFO:= X                                    (1')
```

(C') Exclusão no fim:

```
if P = Λ then
        else begin
             TEMP:= L[P]. LIG;
             if TEMP ≠ P
                then L[P]. LIG:= L[TEMP]. LIG
                else P:= ;
             X:= L[TEMP]INFOR;
             DISPO::= TEMP
             end                                    (3')
```

Os algorítmos (1'), (2) e (3') podem ser usadas para manipular as estruturas vistas anteriormente, donde tiramos o seguinte quadro:

1	*fila de 2 pontas com saída restrita*	*(1'), (2) e (3')*
2	*fila*	*(2) e (3')*
3	*pilha*	*(1') e (3')*

Figura 2.10. - Relação de Estrutura com Algorítmo

Outras operações tornam-se eficientes com listas circulares. Por exemplo, apagar uma lista, isto é, colocar a lista circular de uma vez na pilha DISPO:

```
if P ≠ Λ then begin
                TEMP:= DISPO;
                DISPO:= L[P]. LIG;
                L[P]. LIG:= TEMP
                end                                    (4)
```

A operação (4) é válida mesmo que o ponteiro *P* esteja em qualquer lugar da lista circular.

Usando uma técnica similar, nós podemos incluir uma lista *L2* à direita de uma lista *L1*, onde *P1* e *P2* são respectivamente os ponteiros das listas:

```
if P2 ≠ Λ then begin
                if P1 ≠ Λ then begin
                                TEMP:= L2[P2]. LIG;
                                L2[P2]. LIG:=L1[P1].LIG;
                                L1[P1]. LIG:= TEMP
                                end;
                P1:= P2;
                P2:= Λ
                end                                    (5)
```

Dividir uma lista circular em duas, de várias maneiras, é outra operação que pode ser efetuada. Estas duas últimas operações correspondem a concatenação e desconcatenação de cadeias.

Uma questão que pode aparecer em decorrência das características da lista circular seria: Como encontrar o fim de uma volta na lista, em vista da simetria circular? A res

posta é encontrada se percorrermos a lista de um nó para outro e pararmos quando o próximo é o nó de onde partimos.

Outra solução alternativa para o problema é colocar um nó de reconhecimento em cada lista circular que determine a posição de parada. Este nó especial que chamaremos *CABL* deve ser único na lista circular. Uma vantagem desta solução é que a lista circular nunca estará vazia, facilitando seu manuseio. A figura 2.1 agora torna-se:

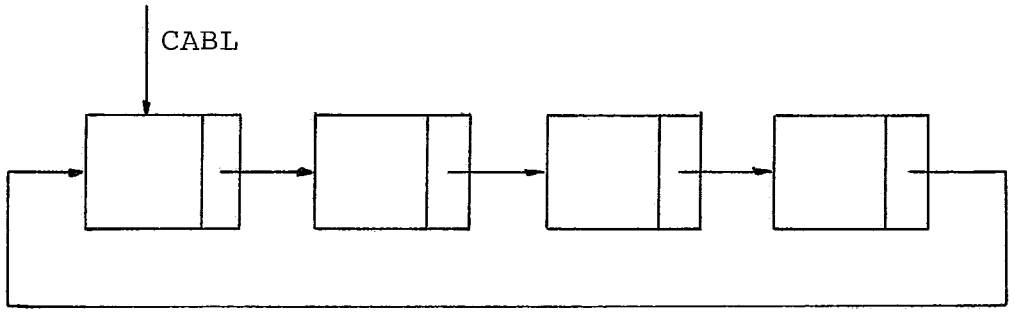


Figura 2.11. - Lista Circular com Cabeça de Lista

Em vez de um ponteiro na extremidade direita da lista, referências à listas do tipo da figura 2.11 são normalmente feitas através de CABL.

A principal diferença entre a lista circular da figura 2.11, com a lista encadeada da figura 2.5 da seção anterior, é que a primeira permite chegar a qualquer ponto da lista tendo partido de um ponto dado.

2.3.5. *Listas Duplamente Encadeadas*

Para melhorar a flexibilidade na manipulação de listas lineares, podemos utilizar dois campos de ponteiros em cada nó, apontado o seu sucessor e o seu antecessor.

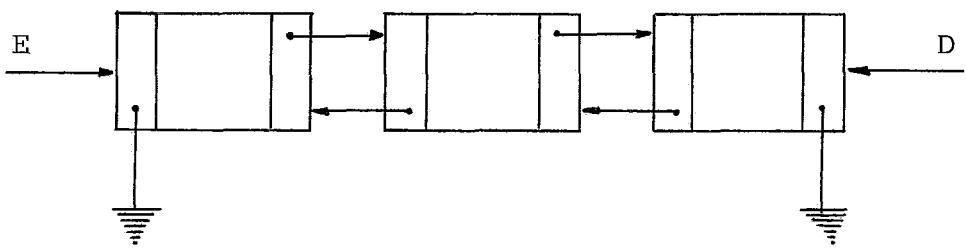


Figura 2.12, - Lista Duplamente Encadeada

Os ponteiros variáveis *E* e *D* apontam a esquerda e a direita da lista. Cada nó da lista tem dois ponteiros que chamaremos ligações esquerda (LIGE) e direita (LIGD). Com estas informações já podemos apresentar os algorítmos que manipulam informações nas extremidades da lista.

(A) Incluir *X* à esquerda:

```
TEMP:= DISPO;
LIGE[TEMP]:= Λ;
LIGD[TEMP]:= E;
if E ≠ Λ then LIGE[E]:= TEMP
              D:= TEMP;
INFO[TEMP]:= X;
E:= TEMP                                   (1)
```

Observação: Para abreviarmos os nossos algorítmos fizemos a seguinte simplificação, onde há equivalência entre as duas estruturas:

$$LIGE[TEMP] = L[TEMP] \,.\, LIGE$$
$$LIGD[TEMP] = L[TEMP] \,.\, LIGD \quad \text{e}$$
$$INFO[TEMP] = L[TEMP] \,.\, INFO$$

(B) Excluir à esquerda:

```
if E = Λ then UNDERFLOW
TEMP:= E;
E:= LIGD[TEMP];
if E = Λ then D:= Λ
         else LIGE[E]:= Λ;
DISPO:= TEMP                                    (2)
```

Da mesma forma poderão ser escritos os algorítmos para o lado direito da lista. Entretanto, manipulações de listas duplamente encadeadas quase sempre se tornam muito fácil se um nó cabeça de lista *(CABL)* é parte de cada lista, como descrito na seção anterior. Quando uma cabeça de lista está presente, teremos o seguinte diagrama da lista duplamente encadeada:

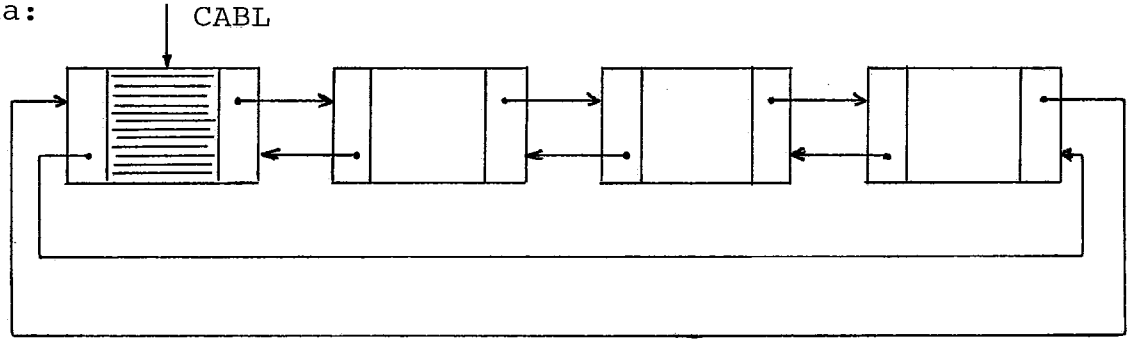


Figura 2.13 - Lista Duplamente Encadeada com Cabeça de Lista

Os campos *LIGE* e *LIGD* do nó *CABL* tomam o lugar dos ponteiros *E* e *D* da figura 2.12.

A seguinte condição é satisfeita claramente pela figura 2.13.

$$LIGD[LIGE[X]] = LIGE[LIGD[X]] = X \qquad (3)$$

Se *X* é a posição de qualquer nó da lista (incluindo o cabeça).

Uma lista duplamente encadeada normalmente ocupa mais espaço de memória do que uma encadeada (veja seção anterior). Mas tem a vantagem de permitir percursos para ambos os lados da lista. Uma outra vantagem é que podemos excluir um nó *J* só com o conhecimento de sua localização. A figura 2.15 mostra as situações antes e depois da exclusão do nó *J*.

(A) Excluir o nó *J*

```
LIGD[LIGE[J]] := LIGD[J];
LIGE[LIGD[J]] := LIGE[J];
DISPO:= J                                   (4)
```

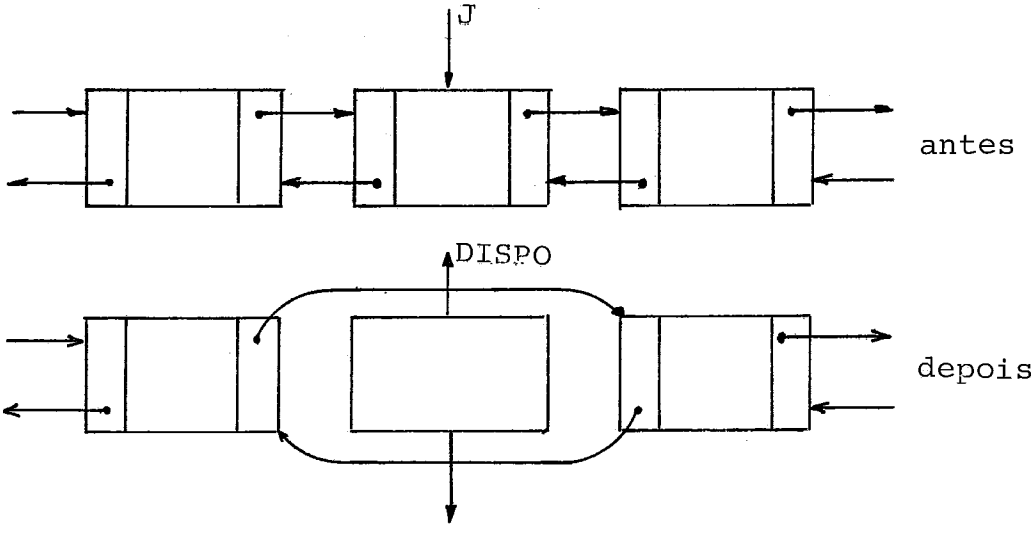


Figura 2.14. - Exclusão em uma Lista Duplamente Encadeada

Do mesmo modo, uma lista duplamente encadeada permite a inclusão de um nó adjacente ao nó *J*, tanto à esquerda como à direita.

(B) Inclusão à direita do nó *J*:

```
TEMP:= DISPO;
LIGE[TEMP]:= J
LIGD[TEMP]:= LIGD[J];
LIGE[LIGD[J]]:= TEMP;
LIGD[J]:= TEMP;
INFO[TEMP]:= X                                    (5)
```

Da mesma forma pode-se escrever uma algorítmo para a inclusão à esquerda do nó *J*, e para isto basta trocar no algorítmo anterior *LIGE* por *LIGD* e vice-versa.

Mostraremos a seguir uma forma de representar listas *duplamente encadeadas* com economia de espaço da ordem de um ponteiro por nó. A idéia é fazer operações lógicas para determinar os nós onde queremos atuar. O campo de ponteiro de um nó resulta da operação lógica: (nó antecessor ou *EXCLUSIVO (XOR)* nó sucessor) desta maneira temos a figura 2.15 que mostra as combinações de nós e encadeamentos. Nó antecessor e nó sucessor são representados fisicamente por um conjunto de bits na forma $2^0 + 2^1 + 2^2 + \ldots$, que para simplificar foi representado em decimal.

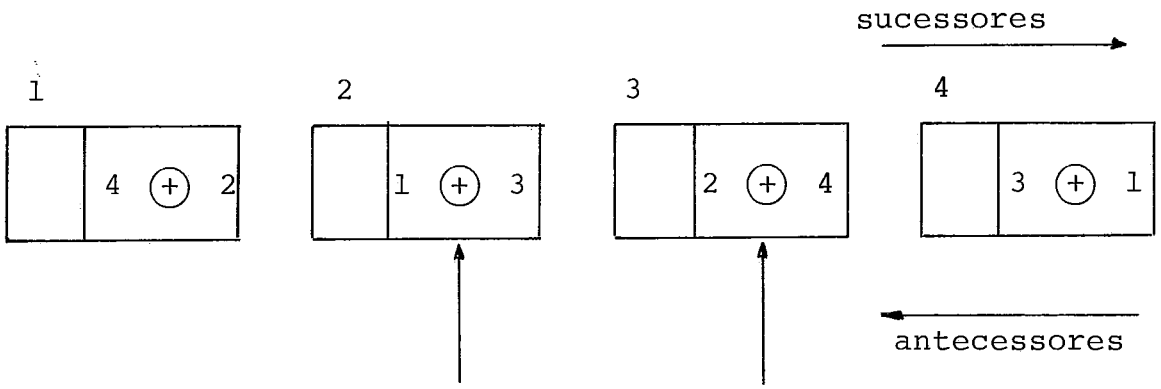


Figura 2.15. - Representação de Lista Duplamente Encadeada com Operação Lógica de Ponteiros

Para determinarmos o sucessor do nó *J*, necessitamos o endereço do nó de onde viemos, ou seja *I*, veja figura 2,15. O endereço é obtido da seguinte maneira:

$$((2 \oplus 4) \oplus 2) = 4$$

Se viermos no sentido contrário e desejarmos encontrar o antecessor de *I*, precisamos do sucessor de *I*, ou seja *J*, para podermos calcular o endereço. Temos:

$$((1 \oplus 3) \oplus 3) = 1$$

A desvantagem neste caso é a necessidade de operações lógicas, e dos dois ponteiros sucessivos para acompanhamento.

Mas esta desvantagem é o custo para a economia de uma ligação, no caso de haver restrições para o espaço de memória.

O campo de ligação precisa de um espaço suficiente para conter o valor do número total de nós que são encadeados, sem levar em consideração o tamanho do nó. A mesma quantidade de nós, de dois tipos diferentes, com cadeias grandes ou pequenas no número de bits, precisará de espaços iguais para conter o ponteiro, pois esta só depende do número de nós.

2.4. Árvores

2.4.1. Estruturas de Árvore

Uma árvore é definida formalmente como um conjunto finito T de um ou mais nós tal que:

a) existe um nó chamado raiz (T)

b) a raiz está ligada a $T_1, T_2, \ldots, T_m$ $(m \geq \phi)$

conjuntos disjuntos, onde cada conjunto é uma árvore. As árvores T_1, T_2,..., T_m são chamadas *sub-árvores* da raiz.

Segue desta definição que todo nó de uma árvore é a raiz de alguma sub-árvore contida naquela árvore. Um nó de grau zero (chamamos grau do nó o número de sub-árvores do nó) é chamado um "nó terminal" ou algumas vezes uma "folha". Um nó não terminal é também chamado de um "nó de ramificação". Dizemos que o nó raiz tem nível 1, e um nó com respeito a sub-árvore da raiz T_j, a qual contém ele, tem um nível maior que o nó raiz.

A figura 2.16 ilustra uma árvore e os conceitos colocados anteriormente

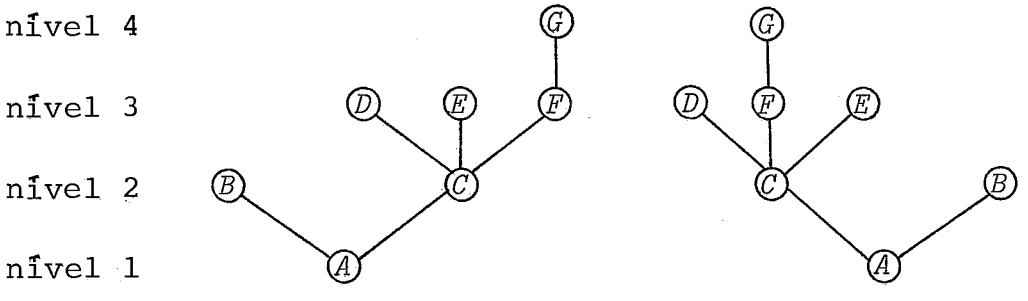


Figura 2.16. - Dois Exemplos de Árvores

Se a ordem relativa das sub-árvores $T_1, T_2, \ldots, T_m$ na definição (b) é importante, nós dizemos que a árvore é uma *árvore ordenada*; quando $m \geq 2$ em uma árvore ordenada, T_2 é chamada de "segunda sub-árvore" da raiz, etc.. Assumiremos que todas as árvores discutidas aqui são ordenadas, à menos que se especifique que não o são.

Uma *floresta* é um conjunto de zero ou mais árvores disjuntas. Outra maneira de escrever a definição (b) seria dizer que "os nós de uma árvore excluindo a raiz formam uma floresta".

Há uma pequena diferença entre florestas abstratas e árvores; se nós excluirmos a raiz de uma árvore, nós temos uma floresta, e, caso contrário, se nós incluirmos um nó como raiz em qualquer floresta nós temos uma árvore.

Há muitas maneiras de desenhar árvores. Nós sempre apresentaremos os desenhos das árvores, conforme a figura 2.17 com a raiz no topo e as folhas na base.

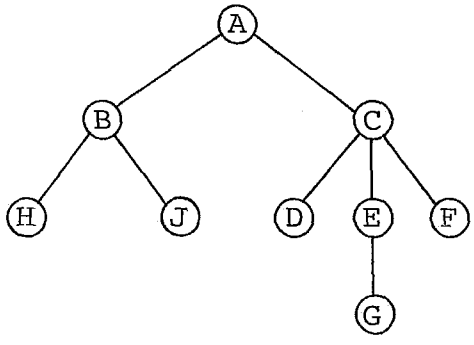


Figura 2.17. - Apresentação Convencional de Árvore

Cada raiz é chamada de "pai" das raízes de suas sub árvores, e estas raízes são chamadas de "irmãos", e eles são "filhos" de seu "pai". A raiz de toda árvore não tem pai. Por

exemplo, na figura 2.17, *C* tem três filhos (*D*, *E* e *F*); *E* é pai de *G*; *B* e *C* são irmãos.

Outro tipo importante de estrutura de árvore é a *"árvore binária"* que apresentamos na figura 2.18.

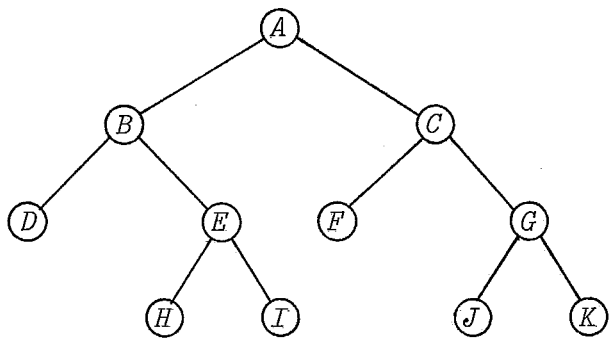


Figura 2.18. - Árvore Binária

Em uma árvore binária cada nó tem no máximo duas sub árvores, e quando há somente uma sub-árvore presente, nós distinguimos entre a sub-árvore esquerda e a sub-árvore direita. Uma definição mais formal de árvore binária é: "um conjunto finito de nós o qual ou é vazio, ou consiste de uma raiz e duas árvores binárias disjuntas" chamadas de sub-árvores direita e esquerda da raiz.

A árvore binária não é um caso especial de uma árvore; é um conceito completo. Por exemplo, as árvores binárias da figura 2.19 são diferentes (a raiz tem uma sub-árvore direita

Figura 2.19. - Árvores Binárias com uma única Subárvore

vazia em um caso e uma sub-árvore esquerda vazia no outro), entretanto como árvores elas serão iguais. Por isso nós sempre devemos tomar cuidado em usar a palavra "binário" para distinguir entre árvores binárias e árvores comuns.

Expressões algébricas podem ser representadas por uma estrutura em árvore. A figura 2.20 mostra uma árvore correspondente a expressão aritmética:

$$A - B\ (C/D + E/F)$$

Em adição à árvores, florestas, e árvores binárias, há um quarto tipo de estrutura, comumente chamado uma "estrutura em lista". A palavra "lista" está sendo usada aqui no seu sentido técnico, e para distinguir seu uso da palavra nós sempre a apresentaremos em maiúsculo, *"LISTA"*. Uma *"LISTA"* é definida (recursivamente) como uma "sequência finita de zero ou

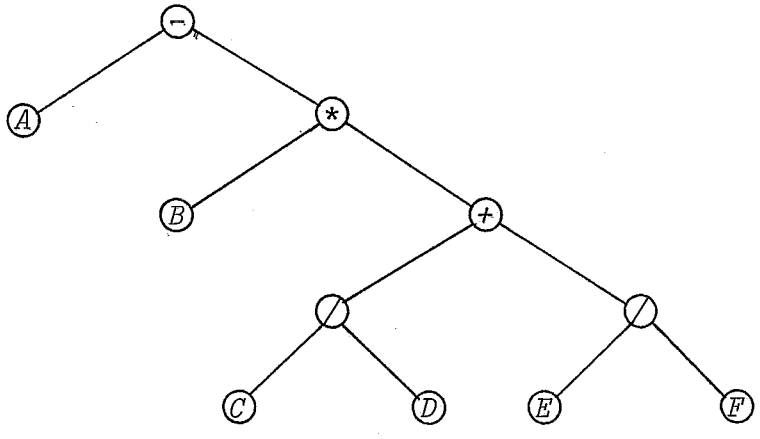


Figura 2.20. - Árvore Representando Expressão Algébrica

mais átomos ou *LISTAS*". Aqui "átomo" é um conceito indefinido referindo-se a elementos de qualquer universo de objetos que

pode ser desejado.

Por meio de uma notação envolvendo vírgulas e parentes ses, nós podemos distinguir entre átomos e LISTAS e exibir convenientemente a ordem dentro de uma LISTA. Vamos considerar a seguinte expressão

$$L = (a,(b,a,b),(\ ),c,(((2)))) \qquad (2)$$

a qual é uma *LISTA* com cinco elementos: primeiro o átomo *a*, a *LISTA (b,a,b)*, a *LISTA* vazia ( ), o átomo *c*, e a *LISTA(((2)))*. A última *LISTA* consiste da *LISTA ((2))*, a qual consiste da *LISTA (2)*, a qual consiste do átomo 2. A figura 2.21 mostra a árvore desta expressão.

O asterisco no diagrama indica a definição e o aparecimento de uma *LISTA*.

LISTAS, são na sua essência, uma generalização de listas lineares tal como já vimos anteriormente, com a adicional

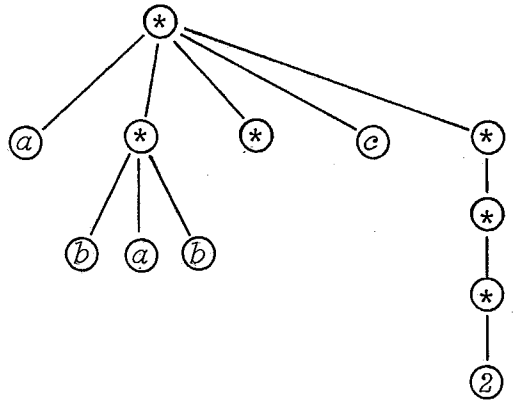


Figura 2.21. - Representação de LISTA em Árvore

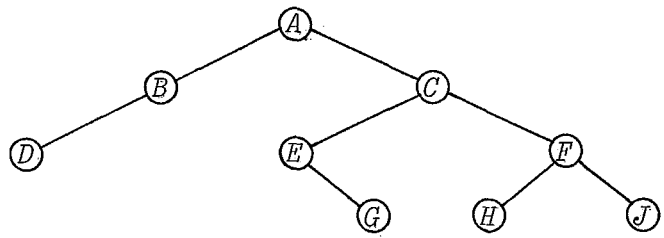
observação que os elementos de uma lista linear podem ter ponteiros as quais apontam para outras listas lineares (e possi

velmente elas mesmas).

Os quatro tipos de estruturas de informação expostas anteriormente (árvores, florestas, árvores binárias, LISTAS) resultam de muitas fontes, e elas são muito importantes em algorítmos utilizados em computadores os quais veremos nas seções seguintes.

2.4.2. Percurso em Árvores Binárias

Árvore binária foi definida como um conjunto finito de nós que ou é vazio, ou consiste de uma raiz ligando no máximo duas sub-árvores (também árvores binárias). Associado a cada árvore nós temos uma ligação variável T, que é ponteiro da raiz da árvore. Se a árvore é vazia, $T = \Lambda$; caso contrário T é o endereço da raiz da árvore, e $LIGE[T]$, $LIGD[T]$ são ponteiros respectivamente para as sub-árvores esquerda e direita da raiz. Cada nó da árvore também tem o seu *LIGE* e *LIGD*. Essas regras definem a representação de memória de qualquer árvore binária; por exemplo, a seguinte árvore binária



é representada por:

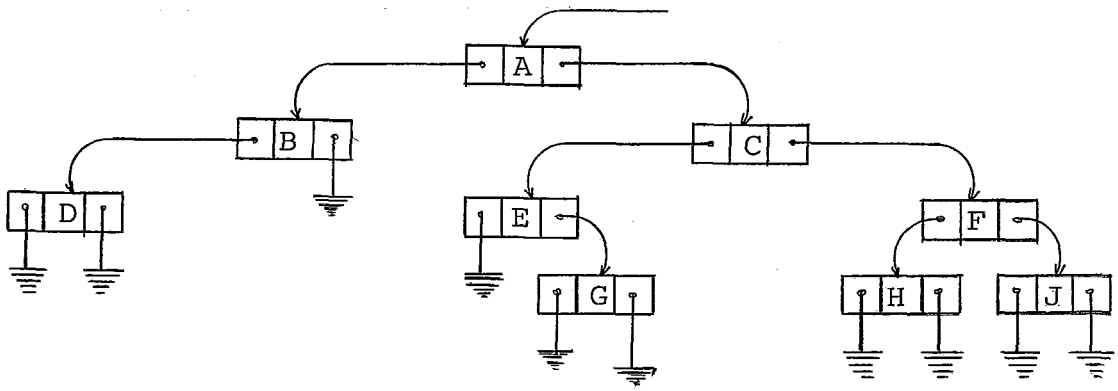


Figura 2.22. - Representação de Memória de Árvore Binária

Há muitos algoritmos para manipulação de estruturas em árvores binárias, e a idéia que ocorre nestes algorítmtos é a noção de percorrer ou andar através da árvore.

Existem três principais métodos para se percorrer uma árvore binária (KNUTH[1]), e são definidos respectivamente; quando a árvore binária é vazia não se faz nada, e caso contrário o percurso procede em três passos. A seguir veremos como funciona cada um dos métodos.

(A) Anterior (Preorder)

"Visitar a raiz;
Percorrer a subárvore esquerda;
Percorrer a subárvore direita"

(B) Intermediária, Natural ou Simétrica (Inorder)

"Percorrer a subárvore esquerda;
Visitar a raiz;
Percorrer a subárvore direita"

(C) Posterior (Postorder)

"Percorrer a subárvore esquerda;
Percorrer a subárvore direita;
Visitar a raiz"

Exemplos: (relativos à figura 2.22)

(A) *A B D C E G F H J*

(B) *D B A E G C H F J*

(C) *D B G E H J F C A*

As definições declaradas recursivamente para estes três tipos de ordem de percurso, precisam ser trabalhadas para uma implementação em computador. Apresentamos a seguir um algoritmo para percorrer árvore binária em ordem natural.

A idéia do algorítmo é que toda vez que um nó tiver sub-árvore esquerda, nós o empilhamos para facilitar a visita ao nó, e também possibilitar que possamos percorrer a sua sub árvore direita.

As seguintes condições iniciais são colocadas para o funcionamento do algoritmo:

ESPERA ≡ *pilha de espera inicialmente vazia*

P ≡ *na saída do algorítmo contém o registro a ser visitado na ordem natural.*

PROXCAND ≡ *na entrada contém o próximo candidato à visita.*

ALGORITMO Para Percorrer Árvore Binária em Ordem Natural.

```
PROXCAND:= T;
repeat
  P:= PROXCAND;
  while P ≠ Λ do begin
                  ESPERA ← P;
                  P:= LIGE[P]
                  end;
  P ← ESPERA;          /* VISITA O NÓ */
  PROXCAND:= LIGD[P]
  until ESPERA = Λ and PROXCAND = Λ                    (1)
```

Um algorítmo quase idêntico pode ser formulado para percorrer árvores binárias em ordem anterior. Já o algorítmo para percorrer em ordem posterior é mais complicado.

É conveniente definir uma nova notação para os nós sucessores e predecessores para os três tipo de ordem. Se P apontar para um nó de uma árvore binária, temos:

$P*$ sucessor de P (anterior)

$*P$ antecessor de P (anterior)

$P\$$ sucessor de P (natural)

$\$P$ antecessor de P (natural)

$P\neq$ sucessor de P (posterior)

$\neq P$ antecessor de P (posterior)

Entre parenteses é citado o método de percurso na árvore.

Segundo esta notação temos:

$$* \ (P*) = (*P)* = P$$

$$\$ \ (P\$) = (\$P)\$ = P$$

$$\neq \ (P\neq) = (\neq P)\neq = P$$

Se P aponta para a raiz nós temos:

$INFO\ (P) = A$

$INFO\ (P*) = B$

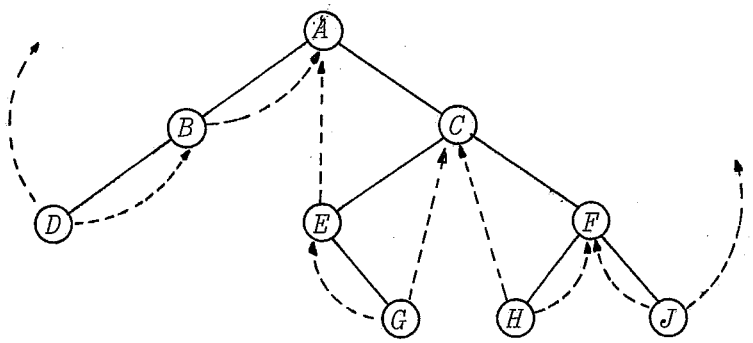
$INFO\ (P\$) = E$

$INFO\ (\$P) = B$

$INFO\ (\neq P) = C$

Podemos notar que na árvore da figura 2.22 há mais ponteiros nulos que outros ponteiros na árvore, na verdade isto acontece para qualquer árvore binária representada por este método.

Uma idéia para melhorar este espaço extra de memória foi sugerido por A.J. Perlis e C. Thornton, que inventaram e chamaram de árvore alinhavada. Neste método, as ligações terminais são substituidas por alinhavos à outras partes da árvore, como uma ajuda para percorrê-la. A árvore alinhavada equivalente a (ϕ) é



(2)

Agora para este tipo de árvore é necessário distinguir os nós que têm alinhavo dos que não têm. A sugestão que se apresenta para os nós que têm alinhavo é atribuir um sinal negativo à ligação. Assim teremos:

Representação Normal	Representação Alinhavada
$LIGE\ (P) = \Lambda$	$LIGE\ (P) = -\ \$P$
$LIGE\ (P) = Q \neq \Lambda$	$LIGE\ (P) = Q$
$LIGD\ (P) = \Lambda$	$LIGD\ (P) = -\ P\$$
$LIGD\ (P) = Q$	$LIGD\ (P) = Q$

De acordo com a definição acima, o alinhavo aponta di

retamente o antecessor ou sucessor do nó em questão, em ordem natural. A figura 2.23. ilustra a orientação geral dos alinhavos em qualquer árvore binária.

O algorítmo seguinte busca o sucessor de um nó numa árvore binária alinhavada em ordem natural.

```
ALGORITMO Para Buscar Sucessor de um Nó

begin
  Q:= LIGD[P];
  if Q < ϕ then Q:= -Q
            else while LIGE[Q] > ϕ
                    do Q:= LIGE[Q]
end                                                    (3)
```

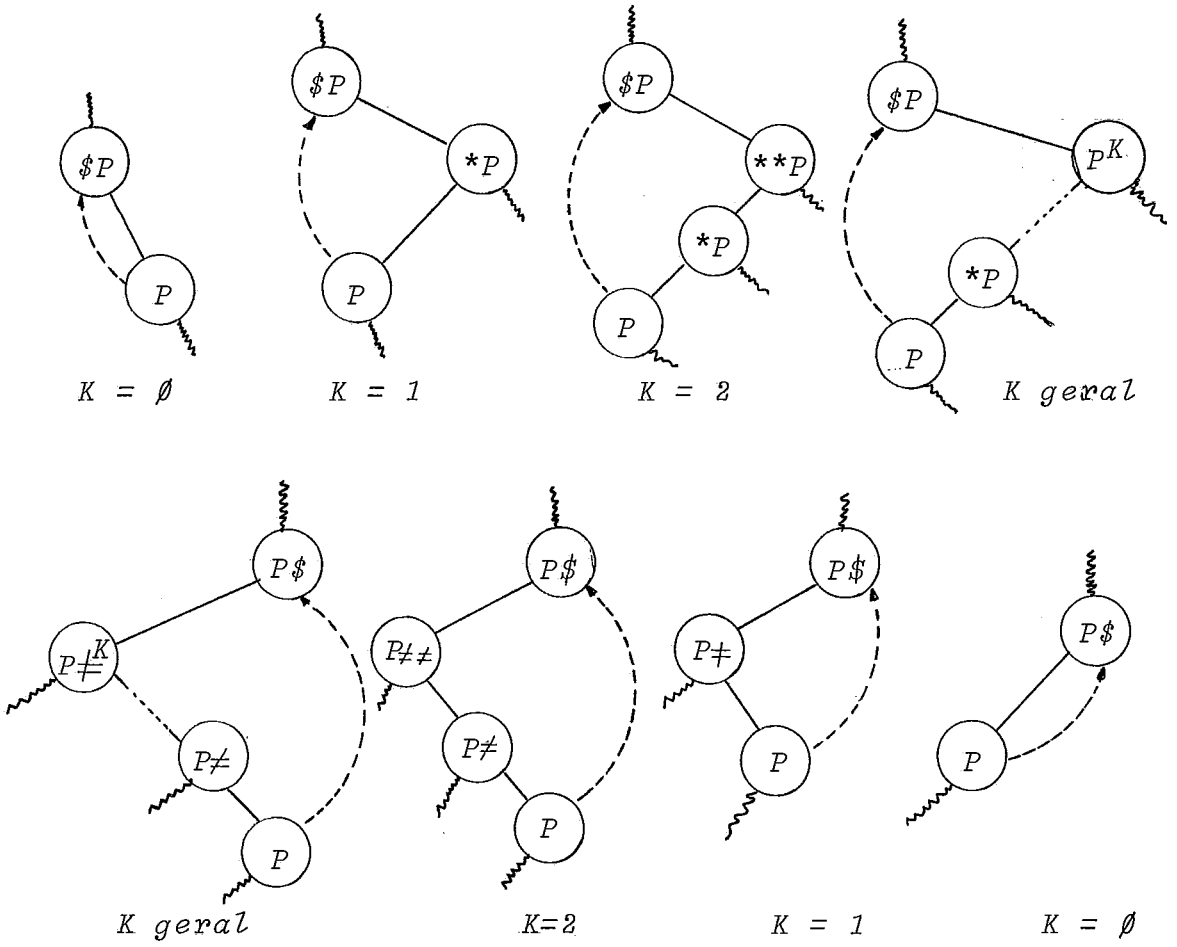


Figura 2.23. - Representação Geral de Alinhavos para Árvores Binárias

Neste algorítmo não é necessário uma pilha para ir guardando informações pois os ponteiros permitem um percurso sobre toda árvore.

Voltando à árvore apresentada em (2), podemos verificar que os alinhavos dos nós mais a esquerda *(D)*, e mais à direita *(J)* não estão completamente definidos. Para completar a representação de uma árvore binária alinhavada é necessário adicionarmos um nó cabeça da árvore (CABA) que tem as seguintes características:

$$LIGE[CABA] = T$$
$$LIGD[CABA] = CABA$$

no caso de árvore não vazia e as seguintes no caso de árvore vazia:

$$LIGE[CABA] = \neg\ CABA$$
$$LIGD[CABA] = \quad CABA$$

De acordo com estas convenções, a representação para uma árvore alinhavada é

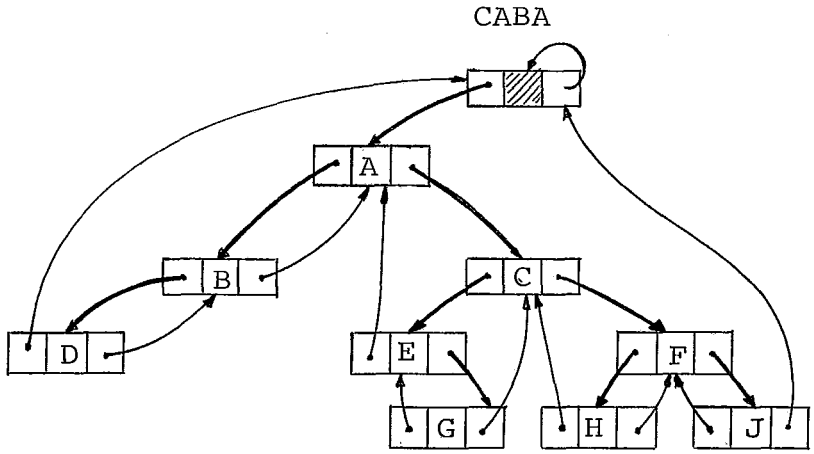


Figura 2.24. - Representação Lógica da Árvore Binária Alinhavada

Como aplicação para árvore binária alinhavada veremos a seguir um algoritmo que faz inserção na mesma. Este algoritmo inclui um nó *Q*, como sub-árvore direita do nó *P*, se a sub-árvore direita é vazia, e insere o nó *Q* entre o nó *P* e o nó *LIGD*[*P*] caso contrário. A árvore binária em questão tem a forma da apresentada na figura 2.24.

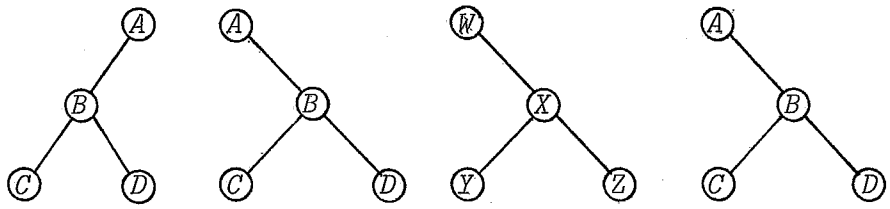
```
ALGORITMO de Inclusão em uma Árvore Binária Alinhavada
begin
  Q:= DISPO;
  INFO[Q]:= X;
  LIGD[Q]:= LIGD [P];
```

```
LIGD[P] := Q;
LIGE[Q] := -P;
if LIGD[Q] > ϕ then begin
                        R:= LIGD[Q];
                        while LIGE [R] > ϕ
                              do R:= LIGE[R];
                        LIGE[R]:= - Q
                        end
end
```

Vamos considerar agora um conceito importante sobre árvores binárias, e sua conexão com percursos. Duas árvores binárias T e T' são chamadas *"similares"* se elas tem a mesma estrutura; isto significa que ou elas são ambas vazias, ou elas são ambas não vazias e suas sub-árvores esquerda e direita tanto de T como de T', tem a mesma representação, ou seja, há uma correspondência um a um entre os nós de T e T'.

As árvores binárias T e T' são chamadas *"equivalentes"* se elas são similares e, além disso, os nós correspondentes tem a mesma informação. As árvores são equivalentes se e somente se são ambas vazias, ou elas são ambas não vazias e $INFO[RAIZ[T]] = INFO[RAIZ[T']]$ e suas sub-árvores esquerda e direita são respectivamente equivalentes.

Para exemplificar estas definições vamos apresentar as seguintes árvores:



(5)

Nestes exemplos as duas primeiras não são similares;a segunda, terceira e quarta são similares; a segunda e a quarta são equivalentes.

Vamos apresentar agora um algoritmo que copia uma árvore binária em diferentes posições de memória. As seguintes condições são colocadas: seja *H* o ponteiro para a cabeça da árvore binária *T* (*T* é a sub-árvore esquerda de *H; LIGE(H)* é um ponteiro para a árvore). Seja o nó *HC* com uma sub-árvore esquerda vazia. Este algoritmo faz uma cópia de *T* que se torna sub-árvore esquerda do nó *HC*. Em particular, se o nó *HC* é a cabeça da lista de uma árvore binária vazia, este algorítmo troca a árvore vazia por uma cópia de *T*.

```
ALGORITMO para copiar uma árvore binária
begin
  inicializar HC;
  if tem árvore para copiar
     then begin
             reservar espaço para a raiz;
             inicializar ponteiros;
             while tem nó para copiar
                do begin
                   copiar o nó
                   avançar ponteiros
                   end
          end
end
```

(6)

inicializar HC

HC:= DISPO;

LIGE[HC]:= - HC;

LIGD[HC]:= HC;

tem árvore para copiar

LIGE[H] > ϕ

reservar espaço para raiz

inserir esq (HC)

inicializar ponteiros

P:= LIGE[H]

Q:= LIGE[HC]

tem nó para copiar

$P \neq H$

copiar o nó

INFO[Q]:= INFO[P];

if *LIGE[P]* > ϕ

then *inserir esq (Q);*

if *LIGD[P]* > ϕ

then *inserir dir (Q)*

avançar ponteiros

suc preorder (P);

suc preorder (Q)

```
procedure suc preorder (PONT: INTEGER);
    if LIGE[PONT]> ϕ
       then pont:= LIGE[PONT]
       else begin
            while LIGD[PONT] < ϕ
                      PONT:= LIGD[PONT];
            PONT:= LIGD[PONT]
            end;


procedure inseriresq (PONT: INTEGER);
    begin
      X:= DISPO;
      LIGE[X]:= LIGE[PONT];
      LIGD[X]:= - PONT;
      LIGE[PONT]:= X
    end


procedure inserirdir (PONT: INTEGER);
    begin
      X:= DISPO;
      LIGD[X]:= LIGD[PONT];
      LIGE[X]:= - PONT;
      LIGD[PONT]:= X
    end
```

Este algorítmo simples mostra uma aplicação típica de percurso em árvore.

2.4.3. *Representação de Árvores como Árvores Binárias*

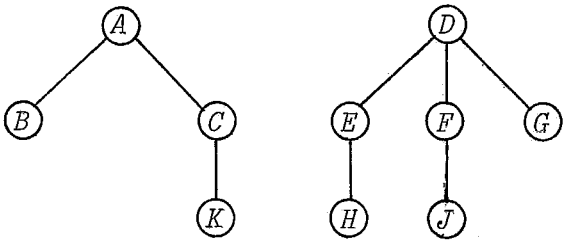
Vamos recordar as diferenças básicas entre árvores e árvores binárias:

1. Uma árvore nunca é vazia, isto é, ela sempre tem no mínimo um nó; e cada nó da árvore pode ter $\emptyset, 1, 2, 3, \ldots,$ filhos.

2. Uma árvore binária pode ser vazia, e cada um dos seus nós pode ter $\emptyset$, 1, ou 2 filhos; nós distinguimos entre um filho à "esquerda" e um filho à "direita".

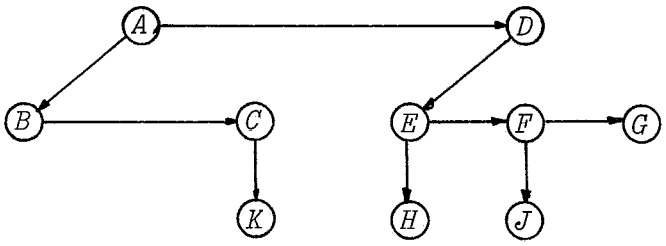
Recordamos também que uma *floresta* é um conjunto ordenado de zero ou mais nós. As sub-árvores imediatamente seguintes a qualquer nó de uma árvore *formam uma floresta*.

Há uma forma natural de representar qualquer floresta como uma árvore binária. Consideremos a seguinte floresta de duas árvores:



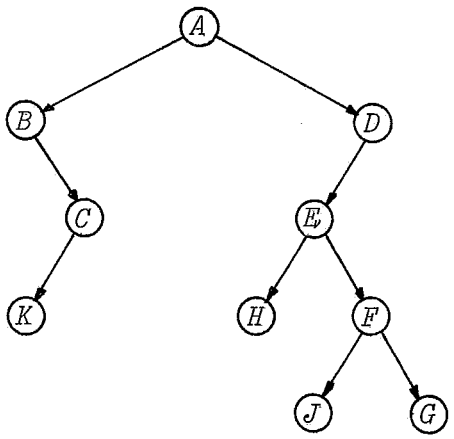
(1)

A árvore binária correspondente é obtida, ligando-se os filhos de mesmo nível de cada família e removendo as linhas verticais exceto para o pai do seu primeiro filho:



(2)

Em seguida, girando posicionalmente o diagrama nós temos uma árvore binária:



(3)

É facil ver que qualquer árvore binária corresponde a uma única floresta de árvores pela reversão do processo.

A transformação acima é muito importante; ela é chamada de "correspondência natural" entre árvores binárias.

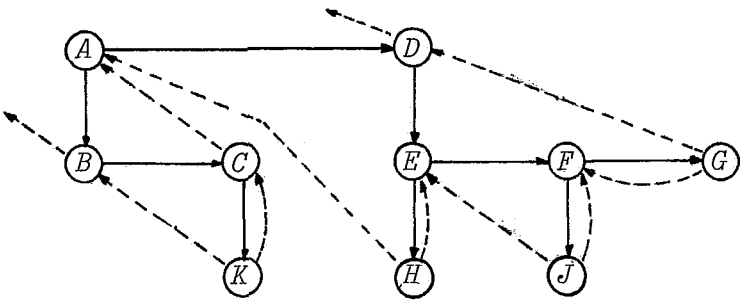
Seja $F = (T_1, T_2, \ldots, T_n)$ uma floresta de árvores. A árvore binária $B(F)$ correspondendo a F pode ser definida rigorosamente como:

a) Se $n = \emptyset$, então $B(F)$ é vazia

b) Se $n > \emptyset$, então a raiz de $B(F)$ é a raiz de T_1; a sub-árvore esquerda de $B(F)$ é $B(T_{11}, T_{12}, \ldots, T_{1m})$, onde $T_{11}, T_{12}, \ldots, T_{1m}$ são as sub-árvores da raiz (T_1); e a sub-árvore direita de $B(F)$ é $B(T_2, \ldots, T_n)$.

A especificação destas regras ajustam-se às transformações ocorridas nas árvores nos passos de (1) a (3).

A árvore binária alinhavada correspondendo a (1) é

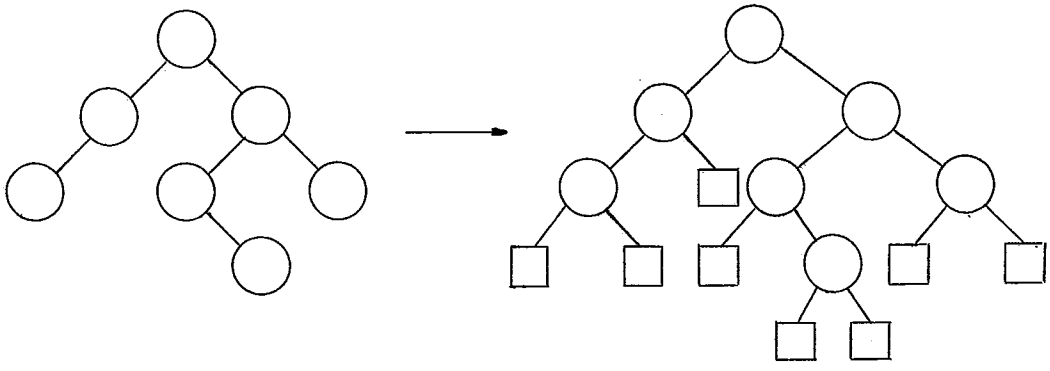


(4)

Os algoritmos desenvolvidos anteriormente (nos capítulos anteriores) podem ser usados sem necessidade de adaptações. Desta forma o inconveniente de se trabalhar com árvores quaisquer (não binárias),deixa de existir fazendo-se a conversão para árvore binária.

2.4.3. Comprimento de Trajetórias em Árvores

Nas discussões que se seguem, vamos extender o diagrama de árvores binárias com adições de nós especiais, sempre que uma sub-árvore nula esteja presente na árvore original. Assim teremos:



(1)

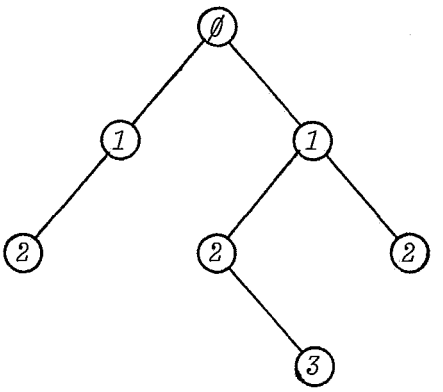
O número de ramificações ou arestas os quais tem que ser percorridos para se deslocar da raiz a um nó X é chamado de *comprimento da trajetória* de X.

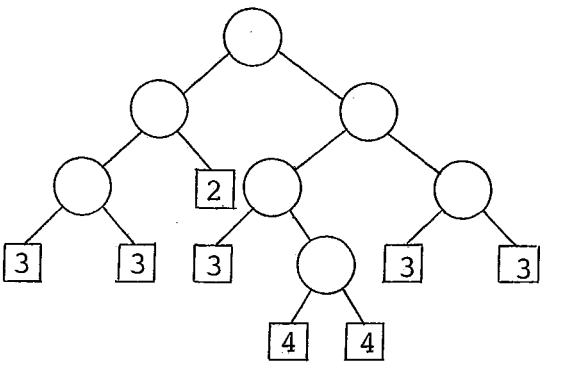
Na extensão feita no diagrama (1) fica claro que todo nó circular tem dois filhos e todo nó quadrado não tem nenhum. Se existem n nós circulares e s nós quadrados, teremos $n + s-1$ arestas, e expressando pelo número de filhos vemos que existem $2n$ arestas e que $s = n + 1$ ou seja, o número de nós externos adicionados é um a mais que o número de nós originais.

O *comprimento interno da trajetória* de uma árvore é definido como a soma dos comprimentos das trajetórias sobre os

nós internos.

O *comprimento externo da trajetória* é a mesma quantidade somada sobre os nós externos.





Em (2) o comprimento da trajetória interna é $CI = \emptyset + 1 + 1 + 2 + 2 + 2 + 3 = 11$, e o comprimento da trajetória externa é $CE = 2 + 3 + 3 + 3 + 3 + 3 + 4 + 4 = 25$. A partir destes dados podemos tirar a seguinte expressão:

$$CE = CI + 2n$$

onde n é o número de nós internos.

de campo *chave*, aquele que identifica unicamente um registro entre outros registros do mesmo *tipo*, como por exemplo o número de alunos em registros que contenham informações cadastrais sobre todos os alunos de uma universidade. O campo chave é às vezes chamado de *chave principal* ou *chave primária*.

2.5. *Operações de Busca por Chave sobre Estruturas de Informação*

Esta seção apresenta características de muita importância no tratamento de informações, pois a necessidade de informações pode se traduzir em necessidades onde a rapidez com que ela é trazida é determinante na tomada de decisões, e portanto necessitamos de algorítmos ágeis e uma estrutura de armazenamento de acesso fácil, talvez muito onerosa, mas necessária, ou necessidades onde o tempo é elemento secundário e portanto podemos ter algorítmos e estruturas menos sofisticadas e talvez com custos mais baixos.

Essas necessidades irão determinar se as estruturas físicas são internas (em memória principal) ou externas (em memória secundária) e consequentemente busca interna e busca externa. Ou as operações de busca podem ser divididas em métodos de busca estáticos (onde as informações praticamente não mudam) ou em métodos de busca dinâmicos (onde as informações sofrem frequentes alterações, inclusões e exclusões).

Levando em consideração estes aspectos apresentaremos, uma série de casos onde estas características estarão mais, ou menos evidentes.

Nesta seção e seguintes iremos manipular registros que estarão armazenados contiguamente numa área de memória, a qual chamaremos de *tabela* e poderá ter o mesmo comportamento já visto para pilhas e filas.

Os registros são compostos de vários campos de informação *(atributos)* e campo(s) de ponteiro(s) (quando for o caso serão indicados explicitamente); dentre estes campos, chamamos

2.5.1. Busca Sequencial

A busca sequencial se caracteriza pela localização de uma chave *Ch* em uma tabela de registros *R1, R2,..., Rn* cujas chaves respectivamente são *Ch1, Ch2,..., Chn*, iniciando a pesquisa na tabela pelo registro *R1* até a localização, ou não localização da chave *Ch* na tabela.

Apresentamos a seguir um algoritmo que executa a tarefa descrita anteriormente.

ALGORITMO DE BUSCA SEQUENCIAL

```
É assumido que o tamanho N da tabela seja ≥ 1
I:= 1;
Achou:= falso;
repeat
  if Ch = Ch[I] then Achou:= verdade
                else I:= I + 1
until I > N or Achou
```

Se a chave *Ch* procurada foi encontrada o algoritmo termina com a variável *Achou = verdade* e a sua localização no registro R_I e caso a chave não foi encontrada a variável *Achou* é igual a *falso*.

Um outro algoritmo, mais rápido que este por exigir menos comparações é apresentado.

ALGORITMO RÁPIDO DE BUSCA SEQUENCIAL

Este algorítmo exige um registro a mais, que para diminuir o número de comparações recebe o valor da chave procurada.

```
I:= 1;
Ch[N+1]:= Ch;
Achou:= falso;
while Ch ≠ Ch[I] do I:= I + 1 ;
if  I ≤ N then Achou:= verdade                    (2)
```

Este último algorítmo (2), mostra uma filosofia de otimização; pois tinhamos uma estrutura repetitiva com dois testes, um dependendo do tamanho da tabela e outro de comparação de chaves. A introdução da chave na tabela do algorítmo (2) eliminou o teste de tamanho da tabela.

Uma pequena mudança na estrutura do algorítmo (2) pode ainda torná-lo mais rápido.

ALGORITMO RAPIDISSIMO DE BUSCA SEQUENCIAL

```
Achou:= falso;
Ch[N+1]:= Ch;
I:= 1;
repeat
  I:= I + 2;
  if Ch = Ch[I] then Achou:= verdade
                else if Ch = Ch[I + 1] then begin
                                            Achou:=verdade;
                                            I:= I + 1
                                            end
until ACHOU;
if I >N then ACHOU:= falso
```

Com a duplicação na estrutura interna, há uma sensível melhora pois cerca da metade das instruções *I:= I + 1* serão evitadas.

Se soubessemos que as chaves estão em ordem crescente, uma pequena mudança no algorítmo (2) gera um outro melhoramento.

ALGORITMO DE BUSCA SEQUENCIAL EM TABELA ORDENADA

```
Achou:= falso;
Ch[N+1]:=  ∞ ;
I:= 1;
while Ch > Ch[I] do I:= I + 1;
if Ch = Ch[I] then Achou:= verdade                    (4)
```

Fazendo a suposição que a probabilidade para todas as chaves são iguais este algorítmo apresenta um tempo semelhante ao (2) para pesquisa com sucesso. Mas pesquisas sem sucesso são descobertas mais rapidadamente, pela ausência da chave na ordenação.

2.5.2. Busca em Tabela Ordenada

Nesta seção iremos mostrar métodos apropriados para pesquisar tabelas cujas chaves estão em ordem, como por exemplo: $Ch1 < Ch2 < Ch3 < \ldots < ChN$.

Se fizermos um acesso aleatório à tabela teremos após a comparação das chaves Ch com $Ch1$:

$Ch < ChI$ [os Registros de R_I, R_{I+1},..., R_N são eliminados da pesquisa]

$Ch > ChI$ [os Registros de R_1, R_2, ...,R_I são eliminados da pesquisa]

$Ch = ChI$ [a chave foi encontrada]

2.5.2.1. Pesquisa Binária

Este método inicia a pesquisa pelo meio da tabela, dividindo-a em 2 metades, sendo uma eliminada pela composição das chaves, uma nova pesquisa no meio da meia tabela que ainda pode conter a chave é feita e assim sucessivamente até chegarmos a uma tabela de tamanho 1.

Este processo também é conhecido por pesquisa logarítmica ou bisseção. Uma das formas de elaborar o algorítmo é atribuir dois ponteiros (inic, fim) para respectivamente início e fim da tabela e que no processo do algorítmo indicarão os limites de pesquisa na tabela.

ALGORITMO DE BUSCA BINÁRIA

```
begin
Achou:= falso;
inic:= 1;
fim:= N;
repeat
  I:= (inic + fim) \ 2; /* divisão inteira */
  if Ch < Ch[I] then fim:= I - 1
                else inic:= I + 1
until fim < começo or Ch = Ch[I];
if Ch = Ch[I] then Achou:= verdade
end
```

As variáveis *INIC*, *FIM* e *I*, usadas no algorítmo, são essencialmente calculadas; somente o esquema de alocação sequencial pode ser usado sem adaptações.

Devido ao fato de se manter a tabela alocada sequencialmente e ordenada, inserções tornam-se trabalhosas quando se utiliza pesquisa binária.

Se ao invés de usarmos os ponteiros *INIC* e *FIM* para limitar a tabela e mais o *I* para apontarmos o meio, podemos usar somente dois, o *I* e um deslocamento δ para a esquerda ou para a direita. O algorítmo seguinte usa esta idéia.

ALGORÍTMO DE BUSCA BINÁRIA UNIFORME

```
begin
Ch[∅] = - ∞;
I:= ⌈N/2⌉;
Achou:= falso;
M:= N;
repeat
  M:= ⌊M/2⌋;
  if  Ch < Ch[I] then I:= I - ⌈m/2⌉
                 else if Ch > Ch[I] then I:= I + ⌊m/2⌋
                                    else achou:=verdade
  M:= ⌊M/2⌋
until  M = ∅ or achou
end
```

A principal vantagem do algoritmo (2) é que ele pode ser alterado para utilizar uma tabela contendo os deslocamentos $\delta(M)$. Assim teremos um novo algorítmo.

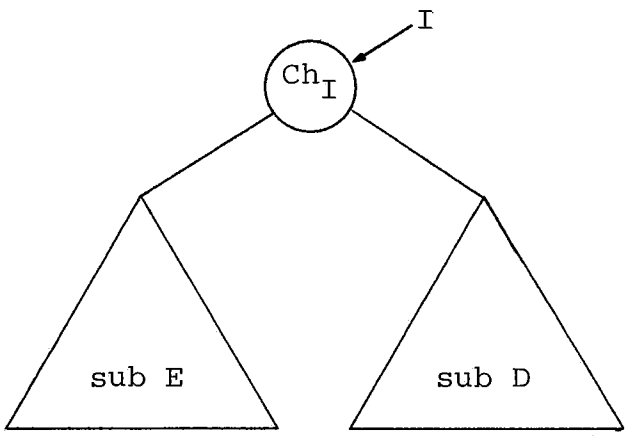
ALGORÍTMO DE BUSCA BINÁRIA UNIFORME

```
Ch[Ø]:= - ∞;
Achou:= falso;
/* O comando abaixo é usado para inicialização da tabela de deslocamento, devendo ser armazenada junto
   com as chaves; só deve ser executado no início ou
   com a mudança de N */
for J:=1 to ⌊log2 N⌋ + 2 do δ[J]:= ⌊(N+2^(J-1))⌋ / 2^J⌋;
J:= 1; I:= δ[1];
repeat
J:= J + 1;
if Ch < Ch[I] then I:= I - δ[J]
              else if Ch > Ch[I] then I:= I + δ[J]
                                 else Achou:= verdade
until δ[J] = Ø or Achou
```

2.5.3. Busca em Árvore Binária

Um método interessante de busca é obtida colocando-se os registros de uma tabela e as suas respectivas chaves em uma árvore binária, onde todo nó da árvore de busca possui chave maior que qualquer nós da subárvore à esquerda *(subE)*, e menor que qualquer nó da subárvore à direita *(subD)*.

Esquematicamente temos:



onde *I* é um ponteiro para um nó qualquer na folha da árvore.

E a relação:

$$sub\ E < Ch_I < sub\ D$$

Onde *subE* e *subD* podem ser substituidos respectivamente, por qualquer chave de *subE* e por qualquer chave de *subD*.

Relações como esta são chamadas invariantes da estrutura. Devido a esta invariância, busca, inserção e remoção de nós são obtidos lexicograficamente ordenadas em ordem simétrica; temos desta forma não só um método de busca como também de ordenação.

A construção de uma árvore é feita com a "chegada"

dos registros, e diferentes permutações de entrada dos registros dão origem à árvores diferentes. Para se obter árvores com razoável distribuição é preferível randomizar as chaves através de uma função de espalhamento. (KNUTH[2]).

Apresentamos a seguir uma descrição informal de um algorítmo de busca e inclusão em árvore binária:

a) Compara-se a chave *Ch* com a raiz da árvore;

b) Se for menor, repete-se o passo *a* para a subárvore esquerda. Se não encontrar requisita um nó disponível e inclue a chave *Ch* na árvore;

c) Se for maior, faz-se a mesma coisa para a subárvore direita.

```
ALGORITMO DE BUSCA E INCLUSÃO EM ÁRVORE BINÁRIA
PONT:= RAIZ;
Achou:= falso; ESQ:= falso;
repeat
  if Ch < Ch[PONT] then if LIGE[PONT] ≠ Λ
                          then PONT:= LIGE[PONT]
                          else begin
                               ESQ:= verdade;
                               inclui chave
                               end
                  else if Ch > Ch[PONT]
                          then if LIGD[PONT] ≠ Λ
                                  then PONT:=LIGD[PONT]
                                  else inclui chave
                          else Achou:= verdade
  until Achou
```

```
INCLUI CHAVE
AUX::= DISPO;                    /* obtém um nó disponível */
Ch[AUX]:= Ch;
LIGE[AUX] := LIGD[AUX]:= Λ;
if ESQ then begin LIGE[PONT]:= AUX; ESQ:= falso end
       else LIGD[PONT]:= AUX;
Achou:= verdade
```

O processo inverso do algorítmo anterior é a retirada de um nó da árvore. Será fácil retirar um nó cujas subárvores esquerda e direita são vazias, mas o processo se complicará um pouco caso o nó a ser retirado tenha as duas subárvores pois precisamos manter a ordem lexicográfica dos nós, com a retirada de qualquer um deles.

Outro problema que poderá acontecer com a retirada de um nó é a árvore se tornar desbalanceada.

O algorítmo seguinte elimina um nó, mantendo a árvore aleatória e razoavelmente balanceada. Ele supõe que o nó a ser removido seja apontado por *Q*, que é filho à esquerda *(ESQ = verdade)* ou à direita *(ESQ = falso)* do nó apontado por *P*.

ALGORITMO DE EXCLUSÃO DE UM NÓ EM UMA ÁRVORE BINÁRIA

```
begin
if LIGD[Q] = Λ
   then if ESQ then LIGE[P]:=LIGE[Q] else LIGD[P]:=LIGE[Q]
   else if LIGE[Q] = Λ then if ESQ then LIGE[P]:=LIGD[Q]   (a)
                                   else LIGD[P]:= LIGD[Q]
                        else
   begin
   R:= LIGD[Q];
   if LIGE[R]= Λ then begin
                      if ESQ then LIGE[P]:= R
                                                            (b)
                             else LIGD[P]:= R;
                      LIGE[R]:= LIGE[Q]
                      end
                 else begin
                      S:= LIGE[R];
                      while LIGE[S] ≠ Λ
                           do begin R:= S; S:= LIGE[S] end;
                      LIGE[S]:= LIGE[Q];
                                                            (c)
                      LIGE[R]:= LIGD[S];
                      LIGD[S]:= LIGD[Q];
                      if ESQ then LIGE[P]:= S
                             else LIGD[P]:= S
                      end
     end;
   DISPO::=Q;
   end
```

(2)

Este algorítmo resolve bem 3 tipos de remoção ou seja:

a) se a subárvore à esquerda ou à direita de *Q* for Vazia;

b) se não ocorrer *a* e a subárvore esquerda do nó *R* filho à direita de *Q* for vazia;

c) se não ocorrer nem *a*, nem *b*.

Cada um destes tipos é anotado no algorítmo.

Temos visto até agora pelos algorítmos que apresentamos que os nós estão ordenados lexicograficamente em ordem simétrica, bem como também *assumem* que as chaves tem igual probabilidade de busca, o que não ocorre na maioria das aplicações. Essas considerações são naturais e questionam a possibilidade de se desenvolver um algorítmo que *construa* uma árvore ótima para buscar chaves com frequências dadas.

O algorítmo seguinte constrói a árvore ótima de busca binária sendo dadas as probabilidades *P1*, *P2*,..., *Pn* e *q0*, *q1*,...,*qn* onde:

Pi = probabilidade que a chave Ch_I seja o argumento de busca.

qi = probabilidade que o argumento de busca esteja entre as chaves Ch_I e Ch_{I+1}.

Por convenção, *q0* é a probabilidade que o argumento de busca seja menor que a chave *Ch1* e *qn* é a probabilidade que o argumento de busca seja maior que a chave Ch_n, então temos *P1 +P2 + ... + Pn + q0 + q1 + ... qn = 1*.

Dadas as probabilidades acima, o algorítmo constrói

árvores binárias $A(i,j)$ as quais tem custo mínimo para as probabilidades $(P_{i+1},\ldots,P_j,q_i,\ldots,q_j)$. São calculadas também as seguintes matrizes:

CUSTO $[i,j]$, para $\emptyset \leq i \leq j \leq n$, o custo da árvore $A[i,j]$

RAIZ $[i,j]$, para $\emptyset \leq i \leq j \leq n$, a raiz da árvore $A[i,j]$

PESO $[i,j]$, para $\emptyset \leq i \leq j \leq n$, o peso total da árvore $A[i,j]$

ALGORITMO QUE CALCULA AS RAÍZES DE UMA ÁRVORE BINÁRIA PARA BUSCA ÓTIMA

```
begin
for I:= ∅  to N do begin
                    CUSTO[I,I]:= ∅; PESO[I,I]:= Q [I];
                    for J:= I + 1 to N do
                                PESO[I,J]:=PESO[I,J-1]+P[J]+Q[J]
                    end;
for J:=1 to N do begin
                  CUSTO[J-1,J]:= PESO[J-1,J];
                  RAIZ[J-1,J]:= J
                  end;                                          (3)
for D:= 2 to N do
    for J:= D to N do
        begin
        I:= J-D; MIN:= ∞;
        K:= RAIZ[I,J-1]; SALVAK:=K;
        LIM:= RAIZ[I+1,J];
        for L:= K to LIM
            do begin
               PARC:= CUSTO[I,L-1] + CUSTO[L,J];
```

```
            if PARC < MIN then begin
                                MIN:= PARC;
                                SALVAK:=K
                                end
            end;
        CUSTO[I,J]:= PESO[I,J] + MIN;
        RAIZ[I,J]:= SALVAK
        end
end
```

O algorítmo anterior calcula as raízes da árvore ótima e agora devemos apresentar um algoritmo que dadas as raízes constrói a árvore ótima. A chamada do algorítmo é *CONSTROIARVORE (Ø,N)* que é apresentado na forma de um procedimento recursivo.

```
CONSTROI ÁRVORE (I,J)
begin
TEMP:= DISPO;
SALVARAIZ:= R[I,J]; /*R[I,J] tem o índice da chave */
/*Aqui deve ser chamado um algoritmo que inclui a
  chave numa árvore */;
if I < SALVARAIZ-1 then CONSTROIARVORE (I,SALVARAIZ-1);
if SALVARAIZ < J then CONSTROIARVORE (SALVARAIZ,J)
end
```

2.5.4. Árvores Balanceadas

O algorítmo de inclusão em árvore (1) (veja seção anterior) produz boas árvores de busca, quando a entrada dos dados é aleatória, mas nem sempre depois que novos nós foram incluidos ou excluidos a árvore permanecerá aleatória. Iremos apresentar um algorítmo que mantem a árvore balanceada todo o tempo. O algorítmo pesquisa para uma da chave *Ch*. Se a chave *Ch* não estiver na tabela, um novo nó contendo *Ch* é incluido na árvore no local apropriado e a árvore é rebalanceada se necessário.

Os nós da árvore contêm os campos *CHAVE*, *LIGE*, *LIGD* e tem também um campo $B[P]$ que é o fator de balanço do *NÓ(P)* que é a altura da subárvore direita menos a altura da subárvore esquerda; este campo sempre contêm os valores $+1$, $\emptyset$ ou -1. Um nó cabeça especial também aparece no topo da árvore; o valor de $LIGD[CAB]$ é um ponteiro para a raiz da árvore, e $LIGE[CAB]$ é usado para guardar a altura da árvore (o conhecimento da altura não é realmente necessário para este algorítmo, mas ele é útil no procedimento de concatenação discutido a seguir). Nós assumimos que a árvore é não vazia, ou seja, que $LIGD[CAB] \neq \Lambda$.

Para conveniência na descrição, o algorítmo usa a notação $LIG[a,P]$ como sinônimo para $LIGE[P]$ se $a = -1$, e para $LIGD[P]$ se $a = +1$.

ALGORITMO DE INCLUSÃO E BUSCA EM ÁRVORE BALANCEADA

```
begin
T:= CAB;
S:= P:= LIGD[CAB];
ACHOU:= falso,
repeat
if Ch = Ch[P]
   then ACHOU:= verdade
   else begin
        if Ch > Ch[P]
                   then Q:= LIGD[P]
                   else Q:= LIGE[P];
        if Q ≠Λ then begin
                   if B[Q]≠ Ø then begin
                                   T:= P; S:= Q
                                   end;
                   P:= Q
                   end
              else begin
                   Q::= DISPO;
                   if Ch > Ch[P] then LIGD[P]:= Q
                                 else LIGE[P]:= Q;
                      Ch[Q]:= Ch;
                   LIGE[Q]:= LIGD[Q]:= Λ; B[Q]:= Ø;
                   incluirebalancear; ACHOU:= verdade
                   end
        end
until ACHOU:= verdade
end
```

INCLUIR E BALANCEAR

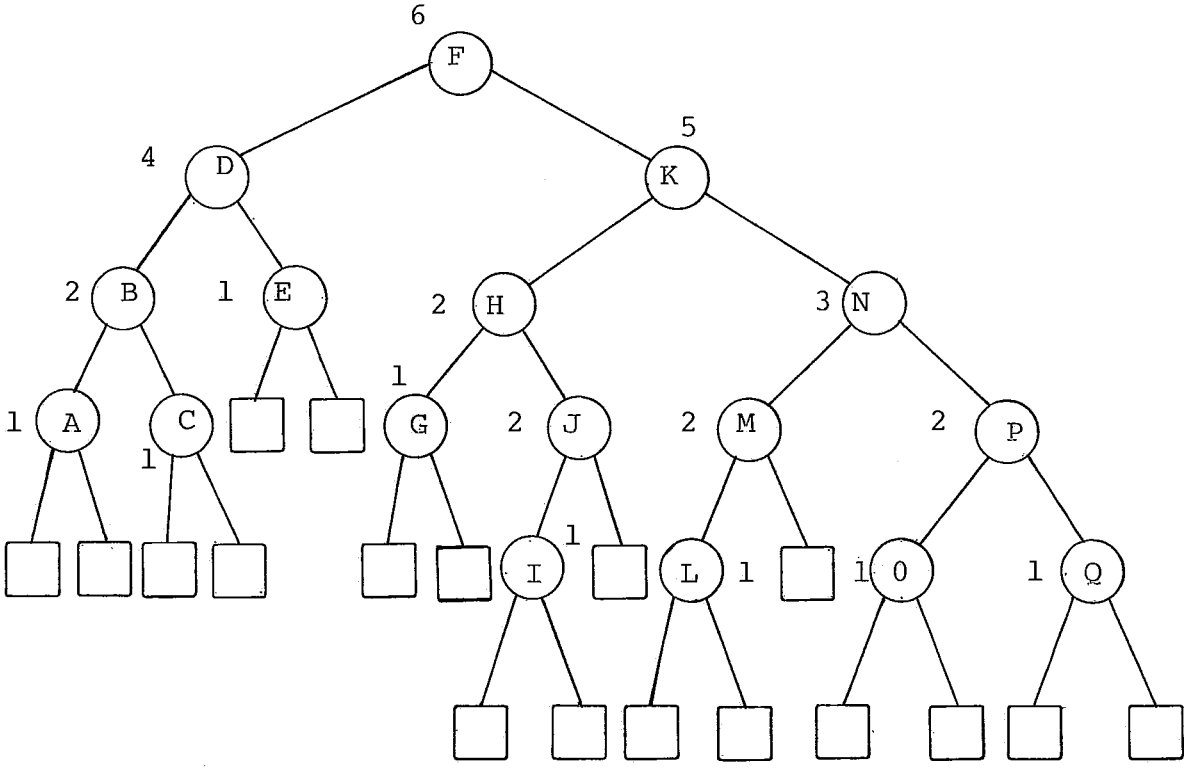
```
begin
if Ch > Ch[S] then begin R:= P:= LIGD[S]; a:= + 1 end
              else begin R:= P:= LIGE[S]; a:= - 1 end;
while P ≠ Q
  do if Ch > Ch[P]
        then begin B[P]:= + 1; P:= LIGD[P] end
        else begin B[P]:= - 1; P:= LIGE[P] end;
if B[S] = ∅
   then begin B[S]:= a; LIGE[CAB]:= LIGE[CAB]+ 1 end
   else if B[S] = -a
   then B[S]:= ∅
   else begin
        if B[R] = a
           then begin
                P:= R;
                if a = 1 then begin
                              LIGD[S]:= LIGE[R];
                              LIGE[R]:= S
                              end
                         else begin
                              LIGE[S]:= LIGD[R];
                              LIGD[R]:= S
                              end
                B[S]:= B[R]:= ∅
                end
```

```
        else begin
             if a = 1 then begin
                           P:= LIGE[R];LIGE[R]:= LIGD[P];
                           LIGD[P]:= R; LIGD[S]:= LIGE[P];
                           LIGE[P]:= S
                           end
                      else begin
                           P:= LIGD[P]; LIGD[R]:= LIGE[P];
                           LIGE[P]:= R; LIGE[S]:= LIGD[P];
                           LIGD[P]:= S
                           end;
             if B[P] = a
                then begin B[S]:= -a; B[R]:= ∅ end
                else if B[P] = ∅  then B[S]:= B[R]:= ∅
                                  else begin
                                       B[S]:= ∅;
                                       B[R]:= a
                                       end;
             end;
        if S = LIGD[T] then LIGD[T]:= P
                       else LIGE[T]:= P
        end
end
```

Para facilitar implementações, árvores balanceadas podem ser usadas para representar listas lineares de tal modo que podemos inserir itens rapidamente (sobrepondo-se à dificuldade de alocação sequencial), assim nós podemos também fazer acessos aleatórios a itens da lista (sobrepondo-se à dificuldade

da alocação encadeada).

A idéia é introduzir um novo campo em cada nó, chamado de campo *POS*, que indica a posição relativa daquele nó em sua subárvore, ou seja, é um valor numérico maior que 1 que dá a posição lexicográfica segundo a ordem simétrica dentro de cada subárvore. Para ficar mais claro apresentamos a figura abaixo.



Para exemplificar o nó *K*, é o 5º nó na sua subárvore o nó *E*, é o 1º nó na sua subárvore.

Nós podemos eliminar o campo *CHAVE*, ou se desejarmos podemos manter ambos os campos *CHAVE* e *POS*, desta forma é possível recuperar itens ou por seu valor chave ou por sua posição relativa na lista.

Usando o campo *POS*, vamos apresentar um algoritmo que

recupere informações sobre os nós desejados. Dado *K* o algorítmo encontra o K-ésimo elemento na lista. Assumimos que a árvore binária tenha campos *LIGE* e *LIGD* e *CAB*, o valor de *LIGD* [*CAB*] é um ponteiro para a raiz da árvore.

ALGORITMO DE BUSCA EM ÁRVORE POR POSIÇÃO

```
begin
M:= K;
P:= LIGD[CAB];
Achou:= falso;
while not Achou and P ≠ Λ
  do if M < POS[P]
        then P:= LIGE[P]
        else if M > POS[P]  then begin
                                 M:= M - POS[P];
                                 P:= LIGD[P]
                                 end
                            else Achou:= verdade
end
```

Tendo sido apresentado este algorítmo que localiza nós, em uma lista linear representada como uma árvore binária balanceada é necessário apresentar também um algorítmo que inclua novos nós na lista. Este algorítmo insere um novo nó justamente antes do K-ésimo elemento da lista, dados *K* e um ponteiro *Q* para o novo nó. Se *K = N + 1*, o novo nó é inserido justamente depois do último elemento da lista. Assumimos que a árvore não esteja vazia e tenha *LIGE*, *LIGD*, o campo *B* (balanço),

a cabeça da árvore *CAB*, mais o campo *POS*. Este algorítmo é bastante parecido com o algorítmo (1).

ALGORITMO DE INCLUSÃO E BUSCA EM ÁRVORE BALANCEADA POR POSIÇÃO

```
begin
T:= CAB;
S:= P:= LIGD[CAB]; U:= M:= K; ACHARPOS := falso;
repeat
   if M ≤ POS[P] then begin POS[P]:=POS[P]+1;  R:=LIGE[P] end
                 else begin M:= M-POS[P]; R:= LIGD[P] end;
   if R ≠ Λ then begin
                 if B[R] ≠ ∅ then begin
                                  T:= P; S:= R; U:= M
                                  end;
                 P:= R
                 end
            else begin
                 if M > POS[P] then LIGD[P]←Q
                               LIGE[P]←Q;
                 POS[Q]:= 1;
                 LIGE[Q]:= LIGD[Q]:= Λ;
                 B[Q]:= ∅;
                 BALANCEAR;
                 ACHARPOS:= verdade
                 end
until ACHARPOS
end
```

```
BALANCEAR

begin
M:= U;
if M < POS[S]
   then  R:= P:= LIGE[S]
   else begin R:= P:= LIGD[S]; M:= M - POS[S] end;
repeat
if M < POS[P]
   then begin
   else if M > POS[P] then begin
                              B[P]:= + 1;
                              M:= M - POS[P];
                              P:= LIGD[P]
                              end
until P = Q;
if U < POS[S] then  a:= -1
              else  a:= +1;
if B[S] = ∅
   then begin
        B[S]:= a; LIGE[CAB]:= LIGE[CAB] + 1
        end
   else if  B[S] = - a
   then  B[S]:= ∅
```

```
else if  B[R] ≠ α
         then begin
              if α = 1 then begin
                            P:= LIGE[R];
                            LIGE[R]:= LGD[P];
                            LIGD[P]:= R;
                            LIGD[S]:= LIGE[P];
                            LIGE[P]:= S
                            end
                       else begin
                            P:= LIGD[R];
                            LIGD[R]:= LIGE[P];
                            LIGE[P]:= R;
                            LIGE[S]:= LIGD[P];
                            LIGD[P]:= S
                            end;
              if B[P]= α
                 then begin B[S]:= -α; B[R]:= ∅ end
                 else if B[P] = ∅
                 then begin B[S]:= ∅; B[R]:= ∅ end
                 else begin B[S]:=  ; B[R|:= α end;
              if α = +1 then begin
                            POS[R]:=*-POS[P];POS[P]:=*+POS[P]
                            end
                        then begin
                            POS[P]:= +POS[R];POS[S]:=*-POS[P]
                            end
              end
```

```
        else begin
             if α=1 then begin
                         LIGD[S]:= LIGE[R]; LIGE[R]:= S
                         end
                    else begin
                         LIGE[S]:= LIGD[R]; LIGD[R]:= S
                         end;
             P:= R;
             B[S]:= B[R]:= ∅;
             if α = 1 then POS[R]:= POS[R] + POS[S]
                      else POS[S]:= POS[S] - POS[R]
             end;
if S = LIGD[T] then  LIGD[T]:= P
               else  LIGE[T]:= P
end
```

2.5.5. *Árvores n-árias*

Até agora, vimos em nossa apresentação, árvores nas quais todo nó tinha no máximo 2 descendentes, ou seja, árvores binárias. Os métodos de busca apresentados foram desenvolvidos principalmente para pesquisa interna, ou seja, contidos inteiramente na memória interna de computadores.

Assumindo agora que os nós de uma árvore devam ser armazenados em uma memória secundária, tal como discos, a principal inovação será que os ponteiros serão representados por endereços no disco ao invés de endereços na memória principal.

Armazenando uma árvore binária, de um milhão de itens, o acesso médio aos nós exigirá cerca de 20 acessos ao disco para localizar o nó procurado. Desde que cada acesso a um nó envolve um acesso a disco, é desejável usar uma organização de memória que realize poucos acessos. As árvores de caminho múltiplo são uma solução para este problema.

Sugere-se que a árvore seja subdividida em subárvores e que as subárvores sejam representadas em unidades as quais sejam acessadas todas juntas. Nós chamamos essas subárvores de *páginas*. A figura 2.25 mostra uma árvore binária subdividida em páginas, cada página constituida de 7 nós. Se um item localizado no disco é acessado, um grupo inteiro de itens pode também ser acessado sem muito custo adicional.

A economia no número de acessos a disco pode ser considerada, pois cada acesso a uma página, agora significa um acesso a disco. Assumindo que uma página contenha 100 nós; então a busca em uma árvore com um milhão de itens requer somente $log_{100}\ 10^6 = 3$ acessos à páginas ao invés de 20.

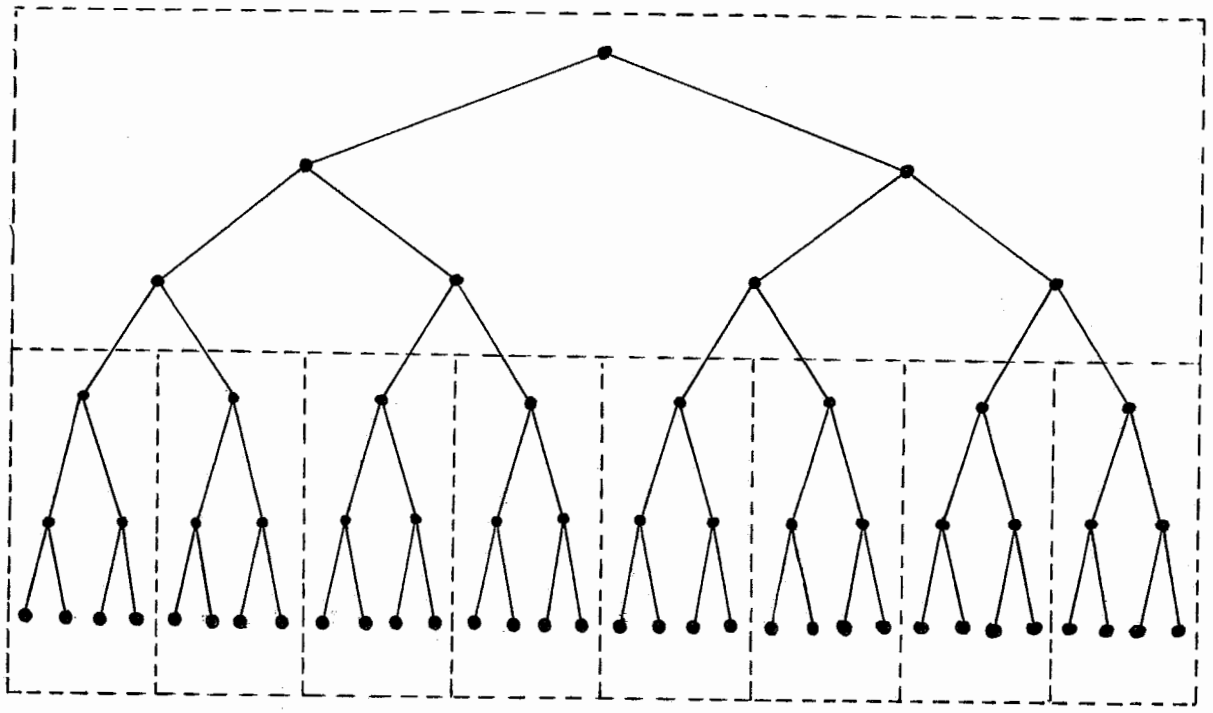


Figura 2.25.

Naturalmente não vamos escolher páginas arbitrariamente grandes, desde que o tamanho da memória interna é limitada e também pode se levar um tempo longo para ler uma página grande.

ÁRVORES-B

Um enfoque para busca em memória externa foi proposta por BAYER[16]. Sua idéia baseada em uma versátil e nova estrutura de dados, chamada árvore-*B*, torna possível buscar e atualizar um grande arquivo com eficiência garantida, e no pior caso, usando algoritmo comparativamente simples.

Uma árvore *B* de ordem *d*, tem as seguintes características (BAYER[16]):

1. Todo nó (página) tem no máximo $2\ d$ chaves e $2d + 1$ filhos;
2. Todo nó (página), exceto o nó (página) raiz deve ter d chaves e $d + 1$ filhos;
3. Todo nó (página), exceto o nó (página) raiz, contém no mínimo d chaves;
4. Todo nó (página) folha, não têm nenhum filho e aparecem no mesmo nível;
5. O nó (página) raiz tem no mínimo 2 filhos (à menos que ele seja uma folha).

A figura 2.26 apresenta uma Árvore-B de ordem 2 com 3 níveis. Todas as páginas contêm 2, 3, ou 4 chaves; a exceção

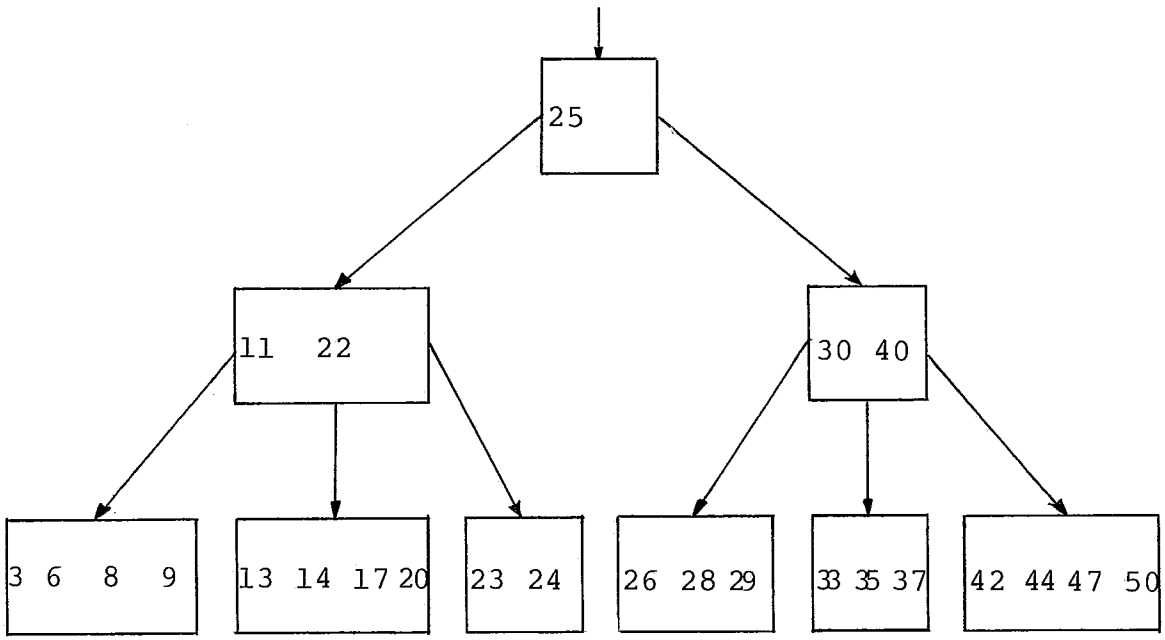


Figura 2.26

é a raiz a qual é permitida conter uma única chave. Todas as folhas aparecem no nível 3. As chaves aparecem em ordem cres

cente da esquerda para a direita.

Uma página com I chaves e $I + 1$ filhos pode ser representada por:

$$E_0, Ch_1, E_1, Ch_2, E_2, Ch_3 \ldots\ldots E_{I-1}, Ch_I, E_i$$

onde $Ch_1 < Ch_2 < \ldots < Ch_I$ são as chaves e $E_0, E_1, \ldots, E_i$ são os endereços das subárvores que contêm outras chaves. Supondo que a página tenha sido movida para a memória principal, e que tenha sido solicitado a busca da chave X, nós podemos usar os métodos de busca convencionais para localizá-la. Se a busca foi sem sucesso, nós podemos ter as seguintes situações:

1. $Ch_J < X < Ch_{J+1}$, para $1 \leq J < I$. Nós continuamos a busca na página apontada por E_J.
2. $Ch_I < X$. A busca continua na página apontada por E_I.
3. $X < Ch_1$. A busca continua na página apontada por E_0.

Se em algum caso o ponteiro for nulo, a pesquisa termina sem sucesso, concluindo-se que a chave não pertence à árvore.

A beleza das árvores-B permanece com os métodos para inserção e exclusão de registros, pois a árvore continua balanceada. O maior caminho em uma árvore-B de n chaves, não contêm mais que $log_d n$ nós, sendo d a ordem da árvore-B.

A operação de busca nunca visita mais que $1 + log_d\ n$

nós em uma árvore-B balanceada de ordem d. Como cada visita requer um acesso, o balanceamento tem grande potencial econômico.

INCLUSÃO

A inclusão de uma nova chave requer um processo de dois passos. Primeiro, a busca segue da raiz para localizar a folha própria para a inclusão. Assim a inclusão é efetuada, e o balanço é restituido por um procedimento que se move da folha para a raiz.

Se o nó onde for incluida a nova chave estiver cheio, o novo nó terá $2d + 1$ chaves. Assim o nó divide-se em dois tal que os d menores ficam no nó da esquerda e os d maiores no da direita e o que sobrou é promovido para o nó pai que da mesma forma se estiver cheio também se dividirá. Se o processo se repetir até a raiz a árvore ganha mais um nível.

Para exemplificar daremos um exemplo:

Incluir as chaves 27 e 5 na árvore-B da figura 2.26.

1º - Inclusão da chave 27, o nó ficará com 4 chaves

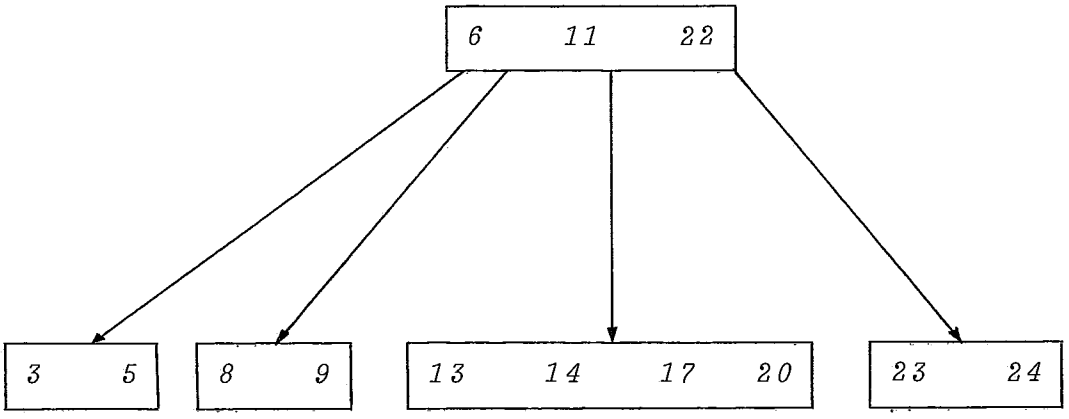
26	27	28	29

2º - Inclusão da chave 5, o nó ficará com 5 chaves

3	5	6	8	9

Só é permitido 4 chaves, portanto o nó se *dividirá* em

dois nós, com a promoção da chave 6 para o nó pai.



Se o nó que recebeu a chave 6 estivesse cheio este também se subdividiria, e assim sucessivamente.

EXCLUSÃO

A exclusão de uma chave também requer a busca que segue do nó raiz até o nó onde se encontra a chave. Aqui pode ocorrer o processo inverso do que ocorreu na inclusão, ou seja, a retirada de uma chave pode deixar o nó com um número não permitido de chaves. então deverá ocorrer uma concatenação de nós vizinhos através do nó pai. Este processo poderá se prolongar até a raiz, se isto ocorrer, a árvore diminui um nível.

Na exclusão de chaves pode ocorrer 2 circunstâncias diferentes:

1. A chave a ser excluida está num nó folha. Assim se a exclusão reduzir a folha para $d-1$ chaves, deve ser feito um ajuste na árvore, de modo a corrigir esta situação. As alternativas para este ajuste são,basicamente:

a) tomar a maior chave do irmão imediatamente anterior, passar para o nó pai, e pegar a chave do pai, para ajustar a folha;

b) tomar a menor chave do irmão imediatamente posterior, fazer o ajuste necessário no nó pai;

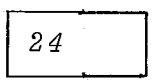
c) caso (a) e (b) não sejam possíveis, porque os irmãos já estão no limite mínimo do número de filhos, fazer a concatenação do nó que sofreu a exclusão, com um dos dois irmãos, ajustando o nó pai, o qual perderá uma chave, podendo portanto, recair no mesmo problema.

2. A chave não está num nó folha; ela deve ser substituída por uma chave de um dos dois nós lexicograficamente adjacentes.

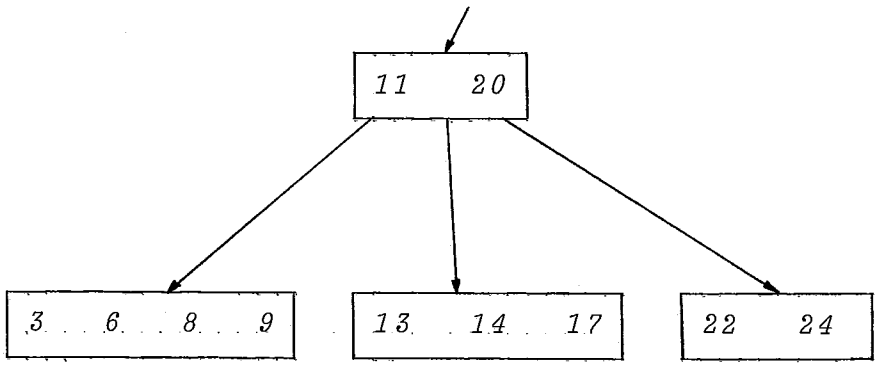
Para exemplificar apresentaremos em exemplo:

Excluir as chaves 23 e 30 da árvore-B da figura 2.26.

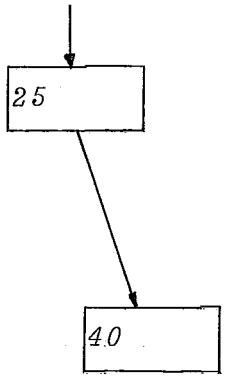
1º - exclusão da chave 29, o nó ficará com 1 chave.



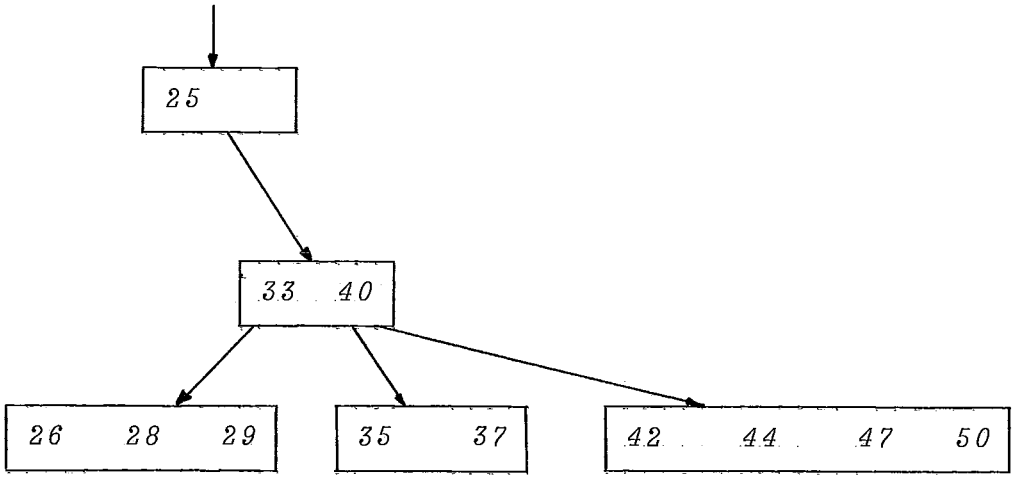
não é permitido 1 chave, portanto teremos:



2º - exclusão da chave 30, o nó ficará com 1 chave.



não é permitido 1 chave, sobe o sucessor, de 30.



Observações: Nota-se que para economia de espaço os nós folhas não necessitam de ponteiros, portanto os nós folhas podem ter formato diferente dos outros nós.

ALGORÍTMO DE RECUPERAÇÃO

Apresentamos a seguir um algorítmo para recuperar um registro com uma chave *Ch*. As variáveis *E*, *R* e *S* são ponteiros que podem assumir o valor Λ (vazio). *R* aponta para a raiz e $R = \Lambda$ se a árvore é vazia. *Pag(E)* é a página que *E* está apontando, e $Ch_1,\ldots, Ch_I$ são as chaves em *PAG (E)* e $E_0,\ldots,E_I$ os

ponteiros na página *Pag(E)*.

```
begin
E:= R;
S:=  Λ;
Achou:= falso;
while E ≠ Λ  and ¬| Achou
   do begin
      S:= E;
      if Ch < Ch_1
         then E:= E_0
         else if ∃i (Ch = Ch_I)
                  then  Achou:= verdade)
                  else if ∃i(Ch_i < Ch_i+1)
                           then E:= E_i
                           else E:= E_l
      end
end
```

ALGORITMO DE INCLUSÃO

O algorítmo apresentado a seguir inclue uma chave *Ch* na árvore-B. A variável *S* (a mesma do algorítmo anterior) é o ponteiro da página que foi atribuido no algorítmo de Recuperação. *S* tem a última página que foi pesquisada ou tem valor Λ se a página da árvore é vazia.

```
begin
Recuperação (Ch, S, Achou);
if  ¬ Achou
   then begin
        if S = Λ
           then cria uma página raiz contendo Ch
           else if Pag(S) = cheio
                   then divide a página e inclue Ch
                   else inclue a entrada (Ch,  Λ) em Pag(S);
        Achou:= verdade
        end
end
```

ALGORITMO DE EXCLUSÃO

Como numa árvore, pode haver necessidade de se excluir chaves da árvore-B, o algoritmo que apresentamos em seguida exclue uma chave Ch da árvore-B e mantém a estrutura.Ele primeiro localiza a chave, por exemplo Ch_I. Para manter a estrutura da árvore, Ch_I é excluida se ela está numa folha, senão ela deve ser substituída pela menor chave da subárvore cuja e $Pag(E_I)$. A menor chave é encontrada seguindo de $Pag(E_I)$ através do ponteiro E_0 até a página folha, digamos L, e tomando a primeira chave em L. Então esta chave, digamos Ch_1, é excluida de L. Como consequência L pode conter menos que d (ordem da árvore) chaves e uma concatenação ou underflow entre L e um irmão adjacente é executado.

```
begin
Recuperação (Ch, Achou);
if ¬ Achou
   then begin
        if Ch está numa página folha
           then exclua Ch da folha
           else begin
                  recuperar as páginas de cima até a folha
                  através dos ponteiros E0;
                  substituir Ch pela primeira chave da
                  página folha;
                  exclua primeiro a chave na folha
                end;
        se necessário, execute concatenação ou overflow;
        Achou:= verdade
        end
end
```

Os algoritmos apresentado aqui foram propostos por BAYER[16] e não foram completamente desenvolvidos. WIRTH[3] apresenta algorítmos semelhantes inteiramente programados.

Para evitar muita divisão de páginas durante a inclusão de chaves, BAYER[16] sugeriu que quando uma página encher ele procure passar chaves suas para uma página vizinha. Desta forma quando as duas páginas se encherem elas se dividirão e formarão uma terceira página agora com ocupação mínima de 66% cada uma das 3 páginas. Também foi proposto que ao invês de só passar chaves para uma página vizinha se passasse para duas páginas vizinhas, assim o overflow só ocorreria quando 3 pági

nas vizinhas, assim o overflow só ocorreria quando 3 páginas estivessem cheias, desta forma a ocupação de cada página passaria para 3/4 ou 75%.

COMER[11], faz um histórico sobre árvores-*B*, colocando as suas características ou mudanças sugeridas por diversos autores. Uma delas chamada *árvore-B*$^+$ tem a característica de ter todas as chaves também nas folhas facilitando o trabalho em determinadas condições, como a possibilidade de se processar todas as chaves de modo sequencial.

2.6. *Processamento de Registros Usando Hashing*

Até agora nós vimos métodos de busca baseados na comparação de um dado argumento *Ch* com as chaves em uma tabela. O tempo requerido para busca nos métodos apresentados cresce com o tamanho *N* da tabela. No melhor destes métodos, a pesquisa binária, o tempo cresce com $log_2 N$, e este é o melhor resultado apresentado para buscas através de comparações de chaves. (KNUTH[2]).

Uma maneira de se evitar a busca da chave por qualquer dos métodos já vistos é fazer um cálculo aritmético com a chave *Ch*, calculando uma função *f(Ch)* a qual dá como resultado a posição de *Ch* e o dado associado em uma tabela. A figura 2.27 apresenta esquematicamente, a aplicação da função *f* sobre várias chaves.

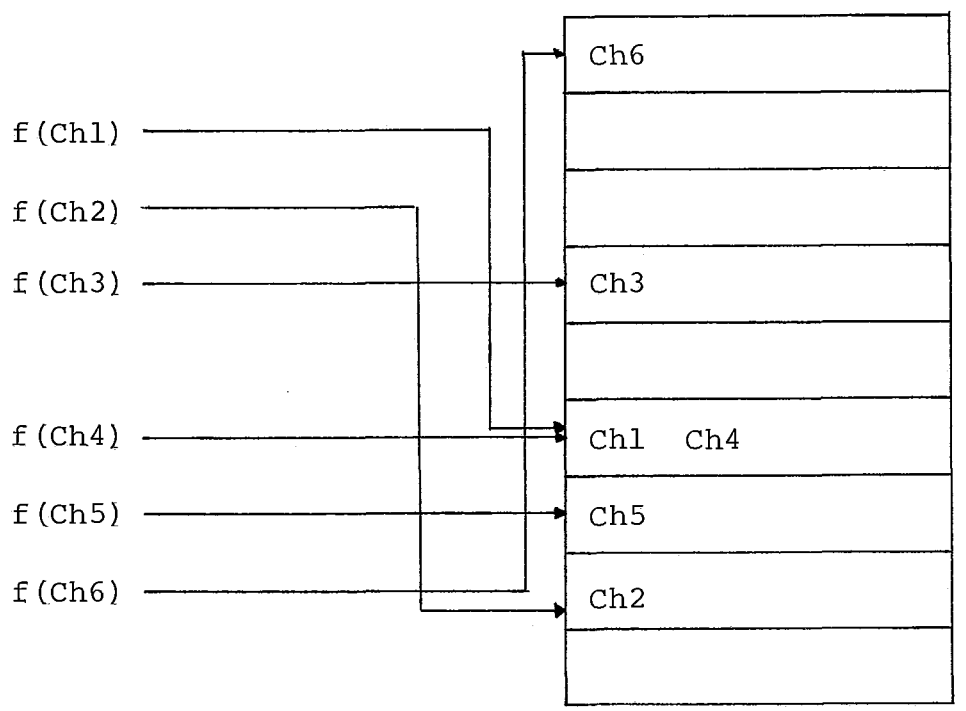


Figura 2.27

Sobre o exemplo apresentado podemos fazer algumas considerações:

1. A tabela deve conter espaço suficiente para conter todas as chaves;
2. A aplicação da função f a chaves diferentes pode gerar o mesmo endereço;
3. No caso de duas ou mais chaves estarem ocupando o mesmo endereço, temos que providenciar maneiras de de se reconhecer as chaves diferentes;
4. Certamente uma nova função, irá gerar diferentes valores de endereço.

Feitas essas considerações, podemos notar quais serão os principais fatores que afetarão a eficiência deste novo método:

a) a função de tranformação de uma chave num endereço;

b) o método de manipulação de esgotamento da tabela;

c) o tamanho do registro que conterá chaves diferentes, mas com mesmo endereço.

2.6.1. Funções de Tranformação

Uma função de transformação ótima deve ter como características:

- minimizar o número de colisões
- ter cálculo rápido

Vamos agora examinar vários métodos comumente utilizados para cálculo de funções de transformação:

1. DIVISÃO

Este método apresenta resultados melhores que o gerador de números aleatórios. A chave é dividida por um número aproximadamente igual ao número de endereços disponíveis, e o resto é tomado como endereço relativo na tabela. Um número primo ou um número que não seja fatorável deve ser usado como divisor.

A razão porque o método da divisão dá pouco overflow com relação ao de aleatorização é que muitos conjunto de chaves terão números consecutivos ao se aplicar a divisão.

2. MEIO DO QUADRADO

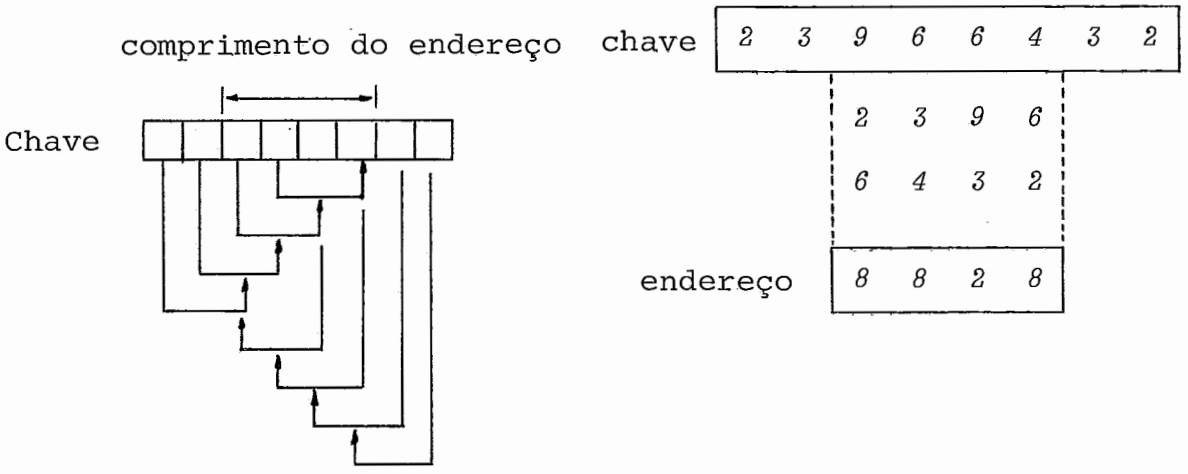
A chave é multiplicada por ela mesma, e os dígitos centrais do quadrado são ajustados para a faixa de endereços.

Se uma chave de 6 dígitos for multiplicado por ela mesma teremos um número com 12 dígitos dos quais retiramos os 4 centrais para serem usados. Este novo número é multiplica

do pelo tamanho da tabela, se o número for 4624 e o tamanho da tabela 5000, teremos 0.5 x 4624 = 23120 o qual é ajustado para 2312 e é usado como endereço na tabela.

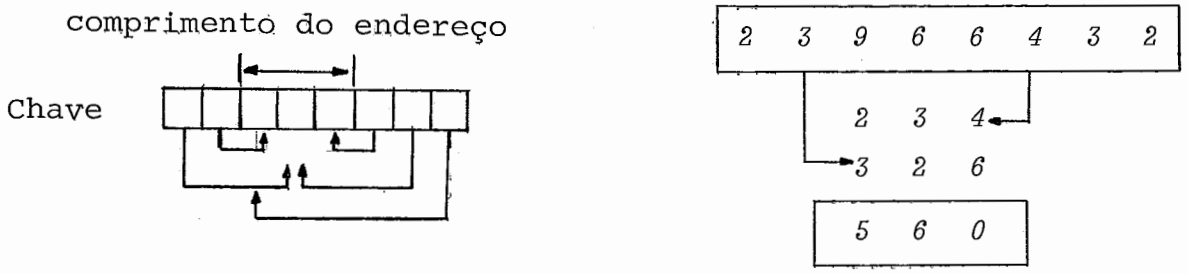
3. DESLOCAMENTO

Os dígitos laterais da chave são deslocados para o centro sobrepondo-se os dígitos para um comprimento igual ao do endereço. Os dígitos são somados, e o resultado é ajustado para o tamanho da tabela.



4. DOBRAMENTO

Os dígitos da chave são dobrados para o centro, como dobrar papel. Os dígitos são somados e ajustados. Dobramento tende a dar bons resultados para chaves grandes.



5. DIVISÃO POLINOMIAL

Cada dígito da chave é tomado como coeficiente de um polinômio. Então a chave 23966432 gera o seguinte polinômio $2x^7 + 3x^6 + 9x^5 + 6x^4 + 6x^3 + 4x^2 + 3x + 2$. Este polinômio obtido é agora dividido por um outro fixo. Os coeficientes obtidos são ajustados e formam um endereço na tabela.

É importante notar que não existe função "ótima" no sentido de que apresente distribuições uniformes para quaisquer dados. A escolha da função depende dos dados que vamos armazenar (chaves), da máquina que se pretende utilizar, da linguagem em que se vai programar e da organização do arquivo que foi escolhido (SOUZA[5]).

Segundo, LUM[20], os métodos que apresentaram melhor resultado foram o do meio quadrado e o da divisão, embora este último possa ser preterido por um outro método, por causa do tempo de execução.

2.6.2. Soluções para Colisões

O tempo gasto com pesquisas em uma tabela utilizando Hash não depende do tamanho da tabela, mas sim do tratamento feito a colisões, provocada pela aplicação de uma função de transformação, em chaves diferentes; provocando como resultado o mesmo endereço na tabela.

A idéia seria construir um algorítmo que pesquisa-se uma tabela de tamanho N, procurando por uma chave Ch. Se Ch não estivesse na tabela e a tabela não estiver cheia, Ch é incluida na tabela.

Os registros da tabela são chamados por $TAB[I]$, onde $\emptyset \leq I \leq N$, e podem estar vazios ou ocupados. Um registro ocupado contém um campo $CHAVE[I]$, um tampo de ligação $LIG[I]$, e possivelmente outros campos.

O algorítmo faz uso de uma função de hash $f(Ch)$. Uma variável auxiliar R é usada, para encontrar espaços vazios na tabela, quando a tabela estiver vazia, nós temos $R = N+1$. Por convenção, $TAB[\emptyset]$ é vazia. Quando houver colisão os espaços vazios são ocupados do fim para o começo.

INCLUSÕES

A inclusão de uma chave de um registro numa tabela é resolvido pelos algoritmos seguintes que apresentam soluções diferentes para o problema de colisões de chaves.

ALGORITMO - DE BUSCA E INCLUSÃO EM TABELA DE ESPALHAMENTO COM ENCADEAMENTO

```
begin
I:= f(Ch) + 1; Achou:=falso; R:= N + 1;
if TAB[I] = Λ
   then begin CHAVE[I]:= Ch; LIG[I]:=Ø; Achou:= verdade end
   else repeat
        if Ch = CHAVE[I]
           then Achou:= verdade
           else if LIG[I] = Ø
                   then begin
                        repeat
                        R:= R - 1
                        until TAB R  =  ;
                        if R = Ø
                           then OVERFLOW
                           else begin
                                LIG[I]:= R;
                                I:= R;
                                CHAVE[I]:= Ch;
                                LIG[I]:= Ø;
                                Achou:= TRUE
                                end
                           end
                   else I:= LIG[I]
           until ACHOU OR OVERFLOW
end
```

Uma das desvantagens deste algorítmo é que ele permite que encadeamentos de endereços diferentes estejam no mesmo laço ou seja ela permite a coalescência de listas diferentes.

Um algorítmo parecido com este sugerido por Lampson in (KNUTH[2]) observou que o espaço necessário para ligações pode ser preservado no método de encadeamento, se forem evitadas a coalescência de listas. A idéia é que além de se usar uma função de hash $f(Ch)$, usa-se uma outra $q(Ch)$, de tal modo que Ch poderia ser determinada se $f(Ch)$ e $q(Ch)$ forem dadas. Por exemplo podíamo ter $f(Ch) = Ch \bmod N$ e $q(Ch) = \lfloor Ch/N \rfloor$. Os registros tem a mesma forma que no algorítmo anterior, com excessão que se inclui um campo de um bit chamado $TAG[I]$. A solução também faz uso de lista circular, com $TAG[I] = 1$ na primeira palavra de cada lista. Se $TAG[I] = 1$ a chave está ocupando o seu lugar, e $TAG[I] = \emptyset$ se está ocupando um registro que antes era vazio.

ALGORITMO - DE BUSCA E INCLUSÃO EM TABELA COM ESPALHAMENTO EN
CADEADO SEM COALESCÊNCIA

```
begin
PROCEDURE INSERE-NOVA-CHAVE;
begin CHAVE[I]:= Q; LIG[I]:= J; Achou:= verdade end;
PROCEDURE PROCURA-NÓ_VAZIO;
begin
repeat R:= R - 1 until TAB[R]= Λ;
if R = ∅ then begin OVERFLOW; STOP end
         else LIG[I]:= R
end;
I:= J:= f(Ch) + 1; Q:= q(Ch); Achou:= falso; R:= N + 1;
if  TAB|I| = Λ
   then begin  TAB[I]:= 1; INSERE-NOVA-CHAVE end
   else if TAB[I] = ∅
            then begin
                 repeat  I:= LIG[I] until LIF[I] = J;
                 PROCURA-NÓ-VAZIO; TAB[R]:= TAB[J]; I:= J;
                 TAB[J]:= 1; INSERE-NOVA-CHAVE
                 end
            else repeat
                 if Q = CHAVE[I]
                    then  Achou:= verdade
                    else if LIG[I] = J then begin          (2)
```

```
                    PROCURA-NÓ-VAZIO; I:= R;
                    TAG[R]:= ∅; INSERE-NOVA-CHAVE
                    end
                else I:= LIG[I]
        until ACHOU
end
```

Outra maneira de se resolver o problema de colisões é eliminar os campos de ligação. A idéia é pesquisar uma tabela de tamanho N, procurando por uma chave Ch. Se Ch não está na tabela e a tabela não está cheia, Ch é incluida. Os registros da tabela são chamados por $TAB[I]$, $\emptyset \leq I \leq N$, e podem estar vazios ou ocupados. Um registro ocupado contêm um campo $CHAVE[I]$, e possivelmente outros campos. O registro N é usado para indicar quantos registros estão ocupados.

ALGORÍTMO - DE BUSCA E INCLUSÃO EM TABELA COM ESPALHAMENTO ABERTO - VISITA LINEAR

```
begin
Achou:= falso;
I:= f(Ch); R:= TAB[N];
repeat
if CHAVE[I] ≠ Ch
   then if TAB[I] = Λ
           then if R ≠ N - 1 then begin
                                  R:= R + 1;
                                  CHAVE[I]:= Ch;
                                  ACHOU:= TRUE
                                  end
                             else begin
                                  OVERFLOW;
                                  STOP
                                  end
           else begin
                I:= I - 1;
                if I < ∅
                   then I:= I + N
                end
   else Achou:= verdade
until ACHOU;
TAB[N]:= R
end                                                    (3)
```

O desempenho dos algorítmos anteriores são razoáveis e são melhores ou piores dependendo do espaço de folga que se deixar na tabela.

Uma tabela com N registros tendo X chaves armazenadas tem um fator de ocupação

$$\alpha = X/N$$

A tabela sendo um pouco maior que o número de chaves que ela irá armazenar, faz com que as chaves sejam melhor distribuidas. Consequentemente o desempenho melhora com a diminuição do fator α, e degrada rapidamente quando X aproxima-se de N.

O algorítmo seguinte é bastante parecido com o anterior, mas ele investiga a tabela de um modo um pouco diferente pois faz uso de 2 funções de hash $f_1(Ch)$ e $f_2(Ch)$. Normalmente $f_1(Ch)$ gera um valor entre $\emptyset$ e $N-1$ inclusive; mas $f_2(Ch)$ deve produzir um valor entre 1 e $N-1$ que é relativamente primo a N. Por exemplo, se N é primo $f_2(Ch)$ pode ser qualquer valor entre 1 e $N-1$ inclusive; ou se $N = 2^n$, $f_2(Ch)$ pode ser qualquer número ímpar entre 1 e $2^n - 1$. Para ficar mais claro vamos dar outro exemplo:

Se N é primo e $f_1(Ch) = Ch \bmod N$ podemos ter:

$$f_2(Ch) = 1 + (Ch \bmod (N-1))$$

mas $N-1$ é par então é melhor usarmos:

$$f_2(Ch) = 1 + \lfloor Ch/N \rfloor \bmod (N-2))$$

Se $N = 2^n$ e estamos usando hash multiplicativo $f_2(Ch)$ pode ser calculada simplesmente pelo deslocamento à esquerda de n bits.

Gary Knott in KNUTH[2] sugeriu que $f_2(Ch)$ depende de $f_1(Ch)$. Se N é primo, podemos ter:

$$f_2(Ch) = \begin{cases} 1 & se \quad f_1(Ch) = \emptyset \\ N - f_1(Ch) & se \quad f_1(Ch) > \emptyset \end{cases}$$

ALGORITMO - DE ESPALHAMENTO DUPLO COM ENDEREÇAMENTO ABERTO

```
begin
Achou:= verdade; I:= f1(Ch); R:= TAB[N];
if TAB[I] = Λ
   then INSERE-Ch-NA-TABELA
   else if  CHAVE[I] ≠ Ch
            then begin
                 C:= f2(Ch);
                 repeat
                 I:= I - C;
                 if I < ∅ then I:= I + N;
                 if TAB[I] = Λ
                    then INSERE-Ch-NA-TABELA
                    else if CHAVE[I] = Ch
                            then Achou:= verdade
                 until Achou
                 end
            else Achou:= verdade;
TAB[N]:= R
end;
```

```
PROCEDURE INSERE-Ch-NA-TABELA
if R = N - 1 then OVERFLOW
             else begin
                  R:= R + 1; CHAVE[I]:= Ch; Achou:= verdade
                  end;
```

Brent in KNUTH[2] propos algumas mudanças no algorítmo anterior. Seu método é baseado no fato que buscas com sucesso são muito mais comuns que inserções, na maior parte das aplicações; assim ele propos que se trabalhe um pouco mais na hora de se incluir o registro, movendo registros, para reduzir o tempo esperado de recuperação. O método é vantajoso sempre que o número médio de buscas à tabela for maior do que X vezes o número de inclusões feitas nesta (que é o próprio número de chaves na tabela), onde X depende da máquina, linguagem e fator de ocupação utilizados. Além do cálculo normal de $f(Ch)$, também é usado o incremento de saltos na tabela, que pode ser calculado no procedimento de Hash.

ALGORITMO DE BUSCA E INCLUSÃO PELO MÉTODO DE BRENT

```
begin
hash (Ch,I,INCR)/*I endereço de Ch na tabela, INCR incremento*/
CONT:= ∅; SE:= I; troca:= falso; achou:=cheio:= falso;
while CHAVE[SE] = Λ or¬Achou or¬cheio
    do if CHAVE[SE] = Ch
          then ACHOU:=verdade
          else if SE=I then CHEIO:= verdade
                       else begin
                            CONT:=CONT+1;
                            SE:=(SE+INCR) mod N
                            end;
if CONT>1 and ⌉ ACHOU and ⌉ Cheio
  then begin
       LIM:= CONT-2; J:= ∅;
       while J<= LIM do begin
                        END:=(I+J*INCR) mod N;
                        hash (CHAVE[END], NI, NINCR);
                        SE2:= (END + NINCR) mod N;
                        CONT2:= 1; LIM2:= LIM -J + 2;
                        while CONT2<LIM2 and CHAVE[SE2]≠ Λ do
                             begin
                             CONT2:=CONT2+1;
                             SE2:= (SE2 + NINCR) mod N;
                             end;
```

```
                    if CONT2<LIM then begin
                                        velho:=SE2;NOVO:=END;
                                        LIM:= CONT2 + J-2;
                                        troca:=verdade
                                        end;
                    J:= J + 1
                    end;
    if troca then begin
                CHAVE[VELHO]:=CHAVE[NOVO];CHAVE[NOVO]:= Ch;
                 end
            else CHAVE[SE]:= Ch
    end
end
```

EXCLUSÕES

A exclusão duma chave de um registro da tabela não é permitida, pois perderíamos o acesso às chaves que colidiram com esta ou com alguma outra em endereçamentos anteriores e foram colocadas em posições posteriores circularmente. Em geral se quisermos fazer deleções, colocamos uma marca de chave excluida, assim um registro teria 3 tipos de marca: vazia, ocupada e excluida. Mas essa solução só funciona se as exclusões forem muito raras, pois do contrário após algum tempo só haveria 2 tipos de marcas na tabela: as ocupadas e as excluidas consequentemente uma busca sem sucesso fará com que se percorra a tabela toda.

Uma alternativa ao processo de marcar a exclusão, é mover as chaves que poderiam ter o seu acesso prejudicado pela

exclusão, para mais perto do seu endereço. O algoritmo a seguir faz uso deste processo e exclui a chave da posição *TAB [I]*. A tabela resultante tem uma chave a menos e sem a degradação de performance comentada anteriormente.

ALGORITMO DE EXCLUSÃO DE CHAVES COM COMPARAÇÃO LINEAR

```
begin
  TAB[I]:= Λ; J:= I; I:=I-1; if I < Ø then I:= I + N;
  while TAB[I] ≠ Λ
     do begin
        R:= f(CHAVE[I]);
        if¬(( I ≤ R or R< J) OR (R<J or J<I) OR (J<I or I≤R))
          then begin
                TAB[J]:= TAB[I]; TAB[I]:= Λ ; J:= I
                end;
        I:= I - 1; if I < Ø then  I:= I+ N
        end
end
```

SOUZA[5], apresenta uma série de algorítmos de Hash, para tratamento de problemas específicos, constituindo uma fonte muito útil de informações, além de fazer um histórico sobre o assunto.

2.7. Busca por Chaves Múltiplas

Até esse momento, todos os nossos algorítmos trataram de busca pela chave principal que especificava unicamente um registro num Arquivo. Mas algumas vezes as solicitações de um usuário estarão baseadas em valores de outros campos do registro além da chave principal; esses campos são chamados de *chaves secundárias* do registro, tornando a organização do arquivo mais complexa para atender ao usuário.

Um exemplo típico de um sistema de informação com o uso de chaves secundárias poderia ser encontrado num sistema com informações pessoais . O usuário poderia solicitar do sistema o nome das pessoas que tem olhos azuis e altura igual a 1,90 metros.

Assim, podemos ter muitos sistemas de informação de uso diário da indústria, no comércio, no governo, etc..., que poderia solicitar informações através de chaves múltiplas que caracterizem as suas aplicações e necessidades.

Um meio de encontrar um registro através de uma chave é usar um índice para esta chave. Um índice baseado num atributo não chave pode ser usado como "chave secundária" é chamado de índice secundário.

Uma primeira solução para este caso é apresentada na figura 2.28, onde os registros contendo a chave secundária tem ponteiros que apontam para informações do mesmo valor em outros registros, além de ter um índice (tabela ou lista) constituido de registros cada um contendo um valor da chave que está sendo indexada.

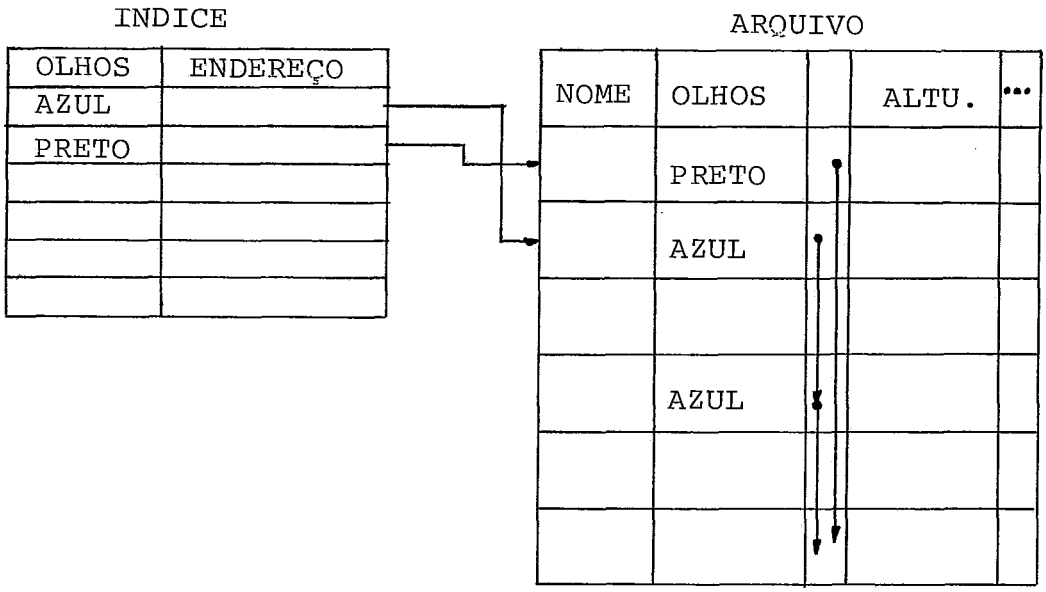


Figura 2.28

Desta forma para cada chave secundária poderíamos construir um índice, associados ao arquivo principal de dados. No caso da questão formulada teríamos dois índices, um para *olhos* e outro para *altura*. Para respondermos a questão bastaria seguir uma das cadeias, por exemplo olhos, e verificarmos se existem pessoas de olhos azuis e altura igual a 1,90 metros.

Uma solução melhorada do caso apresentado seria colocar no índice um contador assinalando o total de ocorrências do mesmo tipo para um determinado valor da chave, desta forma poderíamos percorrer o índice que tenha menor comprimento e acelerar a nossa busca. A figura 2.29, apresenta graficamente esta solução.

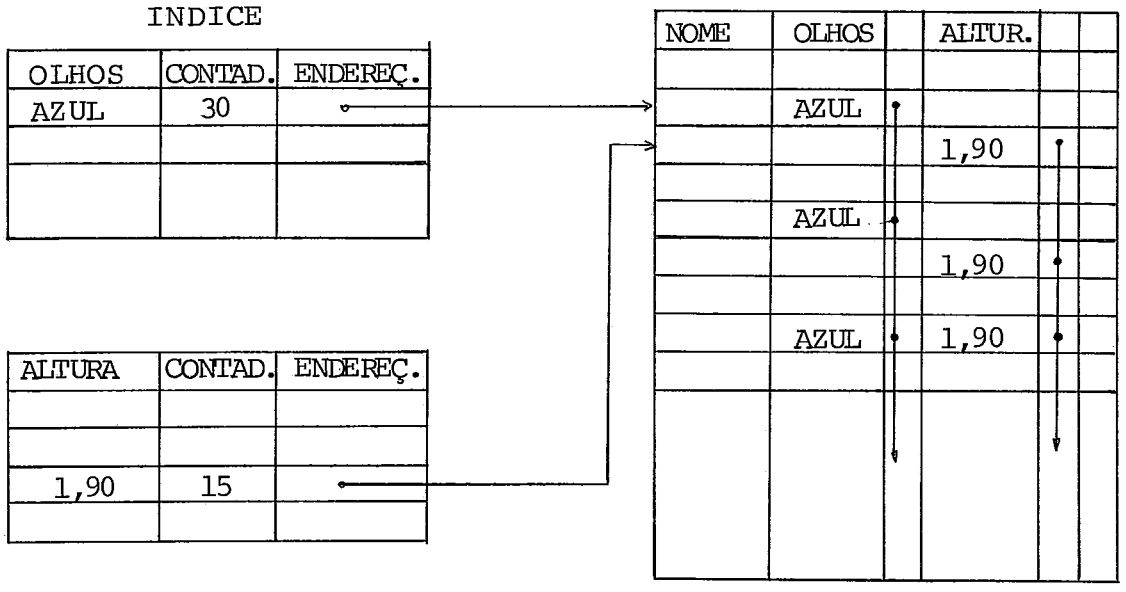


Figura 2.29

Na solução apresentada a busca mais rápida se daria através da cadeia de altura por ser menor e consequentemente apresentar tempo de resposta menor.

Quando o conjunto de registros são organizados da forma como apresentamos anteriormente as cadeias mostradas tem comprimento variável desde zero ponteiros na cadeia até um número que encadeie todas as ocorrências daquele valor. Uma forma de se limitar o comprimento das cadeias seria determinar um comprimento fixo para a cadeia e colocar no índice um novo registro para cada cadeia completa. A figura 2.30 apresenta graficamente esta solução.

OLHOS	CONTADOR	ENDEREÇO
AZUL	4	
	4	
	4	
	2	
PRETO	4	
	4	
	1	

Figura 2.30

Se diminuirmos o tamanho da cadeia que na figura anterior tem comprimento 4 para 1 teremos uma nova estrutura no índice a qual chamamos *índice invertido ou lista invertida*.

Um *Arquivo Invertido* tem como componentes as listas invertidas. Em geral um arquivo invertido deve ser usado junto com o arquivo original, ele providencia duplicata, e informação redundante para recuperar mais rapidamente chaves secundárias.

A figura 2.31 apresenta graficamente um arquivo invertido, juntamente com o arquivo original.

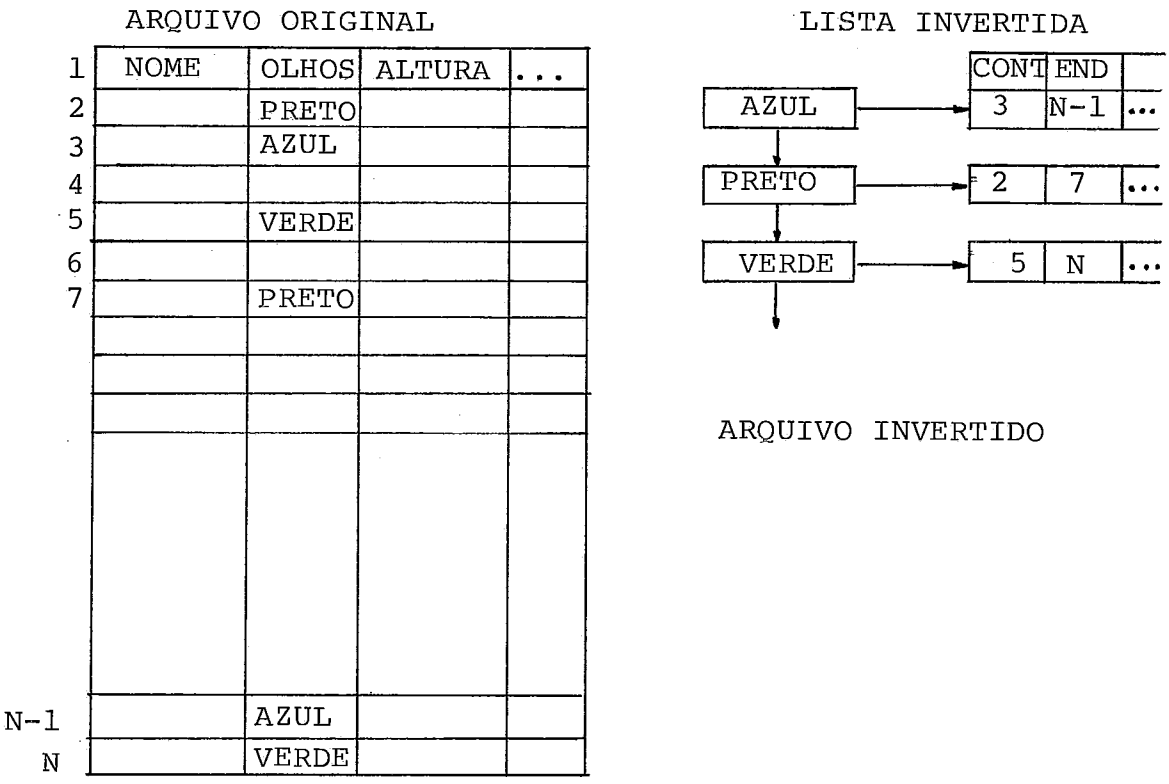


Figura 2.31

KNUTH[2], apresenta uma série de alternativas para se implementar listas invertidas.

HILL[26], apresenta algorítmos para processar registros usando arquivos invertidos e discute principalmente,geração de um arquivo invertido, tempo de construção, algorítmo de recuperação, algorítmo de inclusão, algorítmo de exclusão, tempo de exclusão e necessidades de espaço para armazenamento. A estrutura de armazenamento, é semelhante à apresenta na figura anterior.

A implementação da lista de todos os valores de um determinado item pode usar um método eficiente para o seu armazenamento, e muito mais importante o tempo de busca, poderá usar um dos métodos já vistos como hash, árvore-B ou árvore binária para respostas rápidas.

3. ESTRUTURAS DA INTERFACE DE BANCO DE DADOS LOBAN

A definição da interface, engloba a definição das estruturas de informação e das funções do Banco de Dados, bem como da Linguagem de Operação de Banco de Dados (LOBAN) (SANTOS[29]), mas esta não será tratada neste trabalho.

3.1. Características Estruturais e Funcionais

A interface definida pela Linguagem LOBAN apresenta as seguintes características estruturais e funcionais:

a) autocontida, isto é, cobre todas as funções para operar um banco de dados escrito na própria linguagem sem necessidade de uma linguagem hospedeira ou programas utilitários. Tem funções que descrevem os elementos de dados e seus relacionamentos (esquema), funções para descrever e armazenar a estrutura de uma base de dados, funções para modificar e recuperar informações da base de dados, através de instruções de alto nível pelas quais os usuários definem o que deve ser feito sem se preocupar em como fazer, e funções para manipulação do esquema.

b) predefinição da coerência dos dados, ou seja, da consistência da informação, que define o tipo dos dados, e da sua persistência, que especifica as modificações permitidas.

c) predefinição de formatos de entrada e saída. Estas funções são definidas de uma maneira independente dos aspectos concernentes aos equipamentos periféricos.

d) predefinição e armazenamentos de procedimentos através de textos fonte em LOBAN.

e) predefinição de critérios de classificação de arquivos.

f) controle do acesso a informações, definidos através de mecanismos de proteção (segurança).

g) predefinição de estratégias na utilização de recursos, ou seja, o usuário dirige, nos seus próprios termos, o sistema para algumas políticas a serem seguidas na utilização de recursos do sistema portador.

h) disponibilidade de medidas que permitam a reconstrução da base de dados e reinício de serviços, em caso de erro ou falha de processamento.

i) processamento de lotes, de programas e dados, de forma sequencial.

j) processamento interativo de conversação entre o usuário e o sistema.

k) disponibilidade de características para a utilização de uma linguagem hospedeira para conter a interface LOBAN.

3.2. Estrutura Global

3.2.1. Construções na Base de Dados

Aqui deve-se ressaltar a distinção entre a informação e a sua representação. A informação está no nível de abstração, enquanto a representação está no nível do meio físico. Ao usuário não é necessário o conhecimento da forma de representação das informações na base de dados, isto é, como elas são armazenadas internamente pelo banco de dados. Embora o usuário não deva se preocupar, a representação das informações é o principal assunto deste trabalho. Ao usuário interessam apenas as informações e o enfoque, vista ou organização que ele tem da informação. É sob este último ponto de vista, ísto é, da organização da informação, que a base de dados será descrita a seguir. A cada estrutura de informação referenciável pelo usuário implícita ou explicitamente, chamamos de *construção*.

As construções se compõem de outras construções e assim sucessivamente, até chegar ao nível mais elementar, ou seja, ao nível denominado *atômico*.

Esta forma de composição será descrita a seguir, em suas linhas gerais, deixando-se de mencionar algumas construções para não obscurecer a visão global que se pretende transmitir.

O conteúdo total da base de dados é chamado de *acervo total*. O acervo total não tem um nome pelo qual possa ser referenciado no sistema. O conteúdo da base de dados e o acervo total se confundem.

Dentro do acervo total estão os *acervos setoriais*, os quais são conhecidos do sistema por nomes. Portanto, o usuário pode se referir nominalmente a cada acervo. A divisão em acervos facilita a implantação de aplicações, normalmente correspondendo um acervo setorial a uma aplicação ou sistema de informação.

Não há comunicação de informações entre acervos setoriais. Cada aplicação poderá,no entanto,efetuar operações de leitura em um dado acervo setorial enquanto realiza alterações em um outro acervo setorial, mas nunca operações de alterações em dois acervos setoriais distintos.

A figura 3.1 representa um acervo total, constituído pelos acervos setoriais de nomes *FOLHA-PAGTO*, *CONTR-ESTOQUE*,... ..., *VENDAS*.

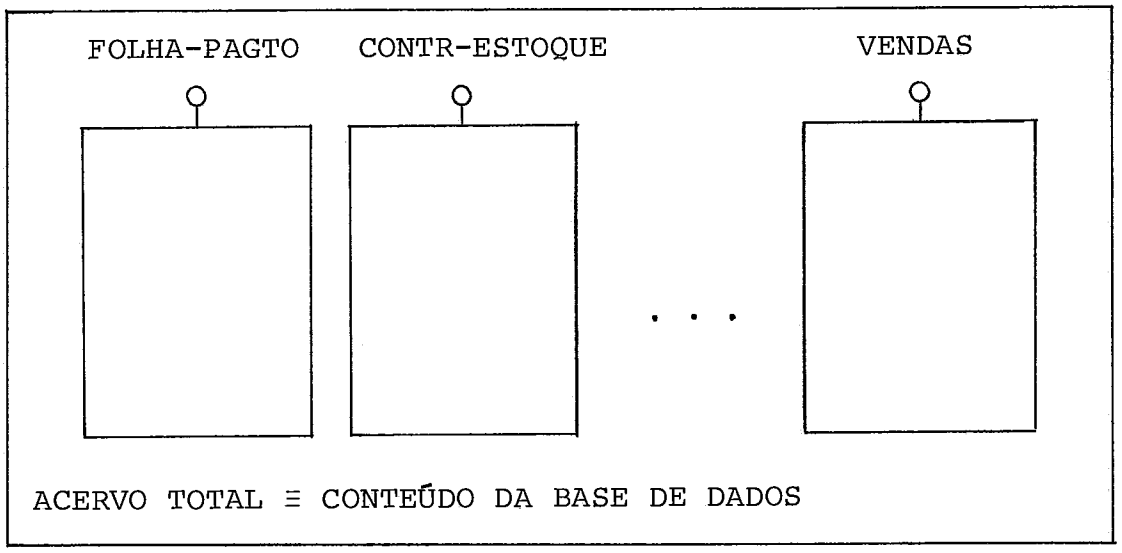


Figura 3.1

Cada acervo setorial, por sua vez é composto de cons

truções de vários tipos, sendo a principal delas, chamada de acervo de trabalho, que aparece em um acervo setorial sob o nome padrão ACTRAB. Cada acervo de trabalho contém também várias construções, onde ressaltamos as construções denominadas por arquivos. Em um acervo de trabalho podem coexistir dois tipos de arquivos, que são os arquivos relacionais e os ligacionais. Aos arquivos são atribuídos nomes e o usuário pode fazer referência nominal a qualquer deles dentro de um acervo de trabalho. A figura 3.2 representa um acervo setorial, cujo acervo de trabalho contém alguns arquivos, cujos nomes são ARQ-1, ARQ-2, ..., ARQ-N..

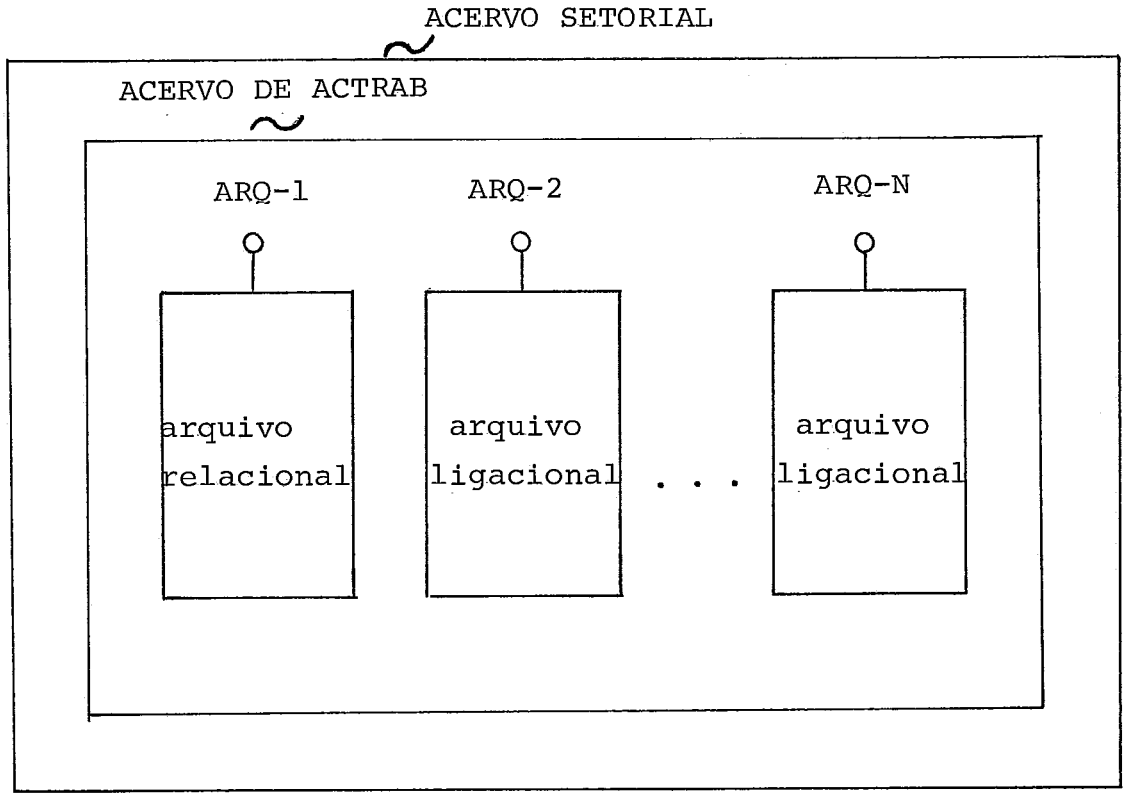


Figura 3.2

Um *arquivo relacional* contém uma tabela, entre outros componentes, que é uma *tabela relacional*, em que cada linha é

uma *tupla*, no sentido da terminologia relacional, ou um registro, na terminologia corrente de processamento de dados. Cada componente de uma tupla corresponde aproximadamente a um campo, na terminologia corrente de processamento de dados. Assim, cada um dos componentes de uma *linha* pode ser identificado por um nome. A representação de uma tupla pode ser vista na figura 3.3,onde os componentes têm os nomes ID, NRO, ..., END.

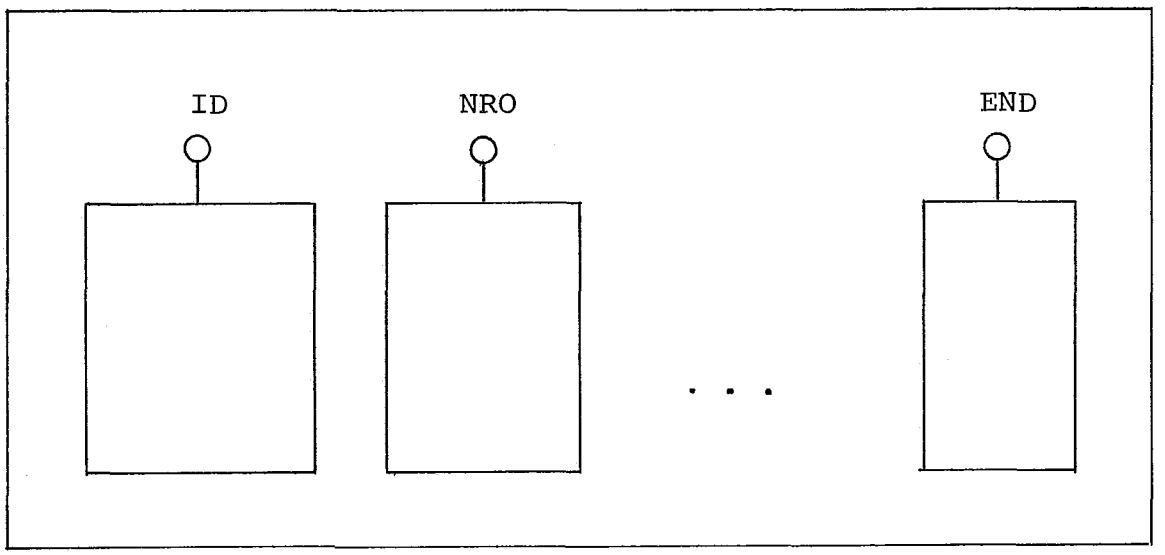


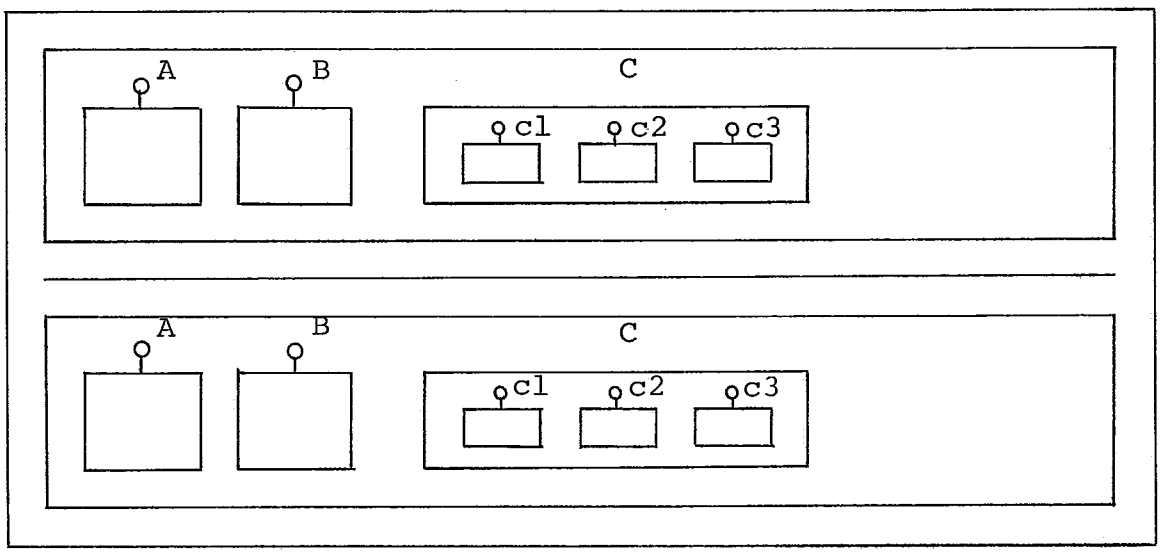
Figura 3.3

Um componente de uma tupla pode ser um átomo ou uma tupla. Um átomo é considerado indivisível constituindo a construção de mais baixo nível. Há diversos tipos de átomos, que contêm construções de naturezas diversas, como por exemplo, numéricos e alfanuméricos.

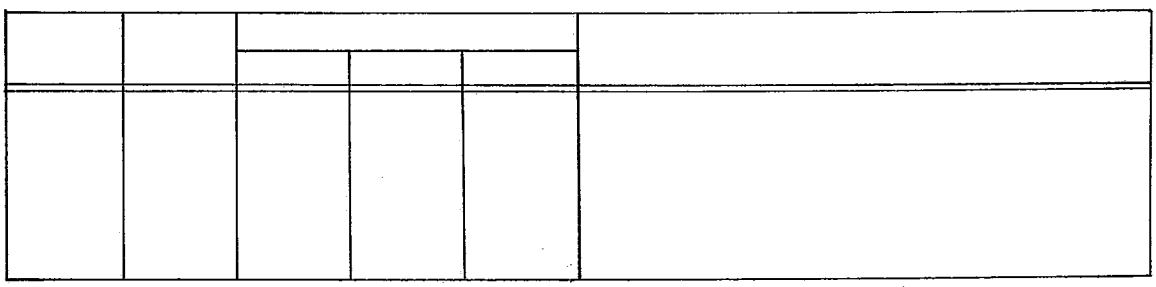
Uma tabela relacional é constituída de tuplas do mes

mo tipo. Pode-se assim aplicar cada nome a todos os componentes de uma mesma coluna da tabela. A referência a uma *linha* pode ser feita pela posição relativa dentro da tabela, se for estabelecida uma ordem sobre as *linhas*, ou por condições a serem satisfeitas pelos seus componentes.

A figura 3.4 mostra duas formas de representação da tabela do arquivo relacional, ou tabela relacional, sendo A, B, C, ..., os nomes dos componentes de cada linha. O componente sob nome C é uma tupla novamente, e seus componentes têm os nomes C1, C2 e C3.



a) Representação por Caixas



b) Representação Tabular

Figura 3.4

Um *arquivo ligacional* contém, entre outros componentes, uma tabela ligacional, que pode ser vista como constituída por registros especiais, chamados de *ligações*.

Cada uma destas ligações é constituída por uma tupla e uma tabela relacional. Uma ligação é representada na figura 3.5, onde, sob nome *L*, se encontra a tupla e, sob nome *T*, a tabela relacional.

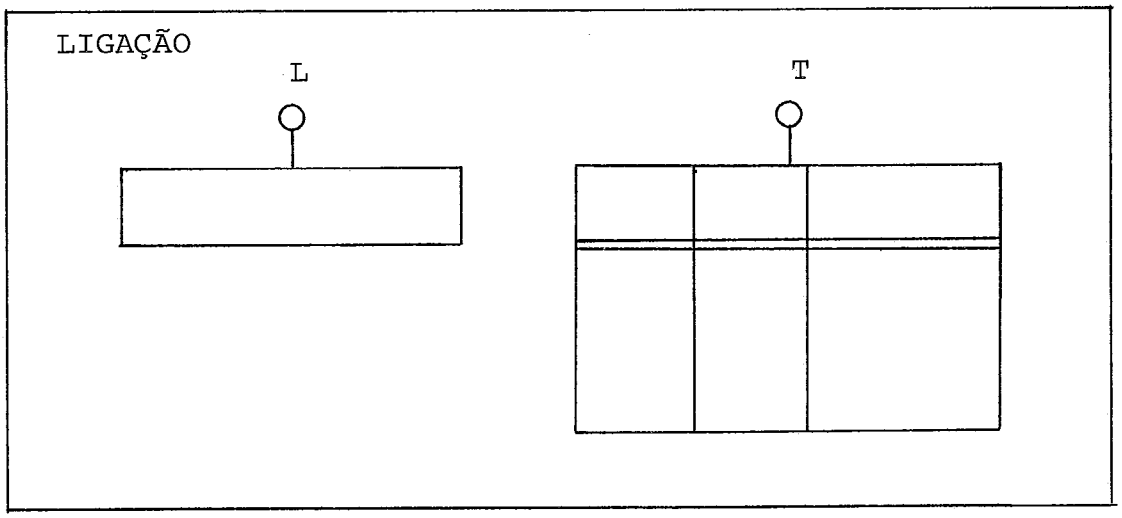


Figura 3.5

Desta maneira é possível estabelecer a ligação entre um registro de informações, o ligante, e um conjunto de registros de uma tabela, chamada tabela ligada, onde cada registro é chamado ligado, que assim ficam ligados ao ligante.

A referência a uma ligação pode ser feita pela sua posição relativa dentro da tabela ligada se for estabelecida uma ordem sobre as ligações, ou por condições a serem satisfeitas pelo ligante. É possível referenciar, dentro de uma ligação, o ligante ou cada um dos ligados.

Portanto, em relação a estes, também é possível referenciar cada um dos seus componentes, átomos e tuplas.

Uma tabela deste tipo, de um arquivo ligacional, é chamada de *tabela ligacional* que é uma "estrutura homogênea" pois contém ligações do mesmo tipo. A figura 3.6 representa uma tabela ligacional, contendo as ligações.

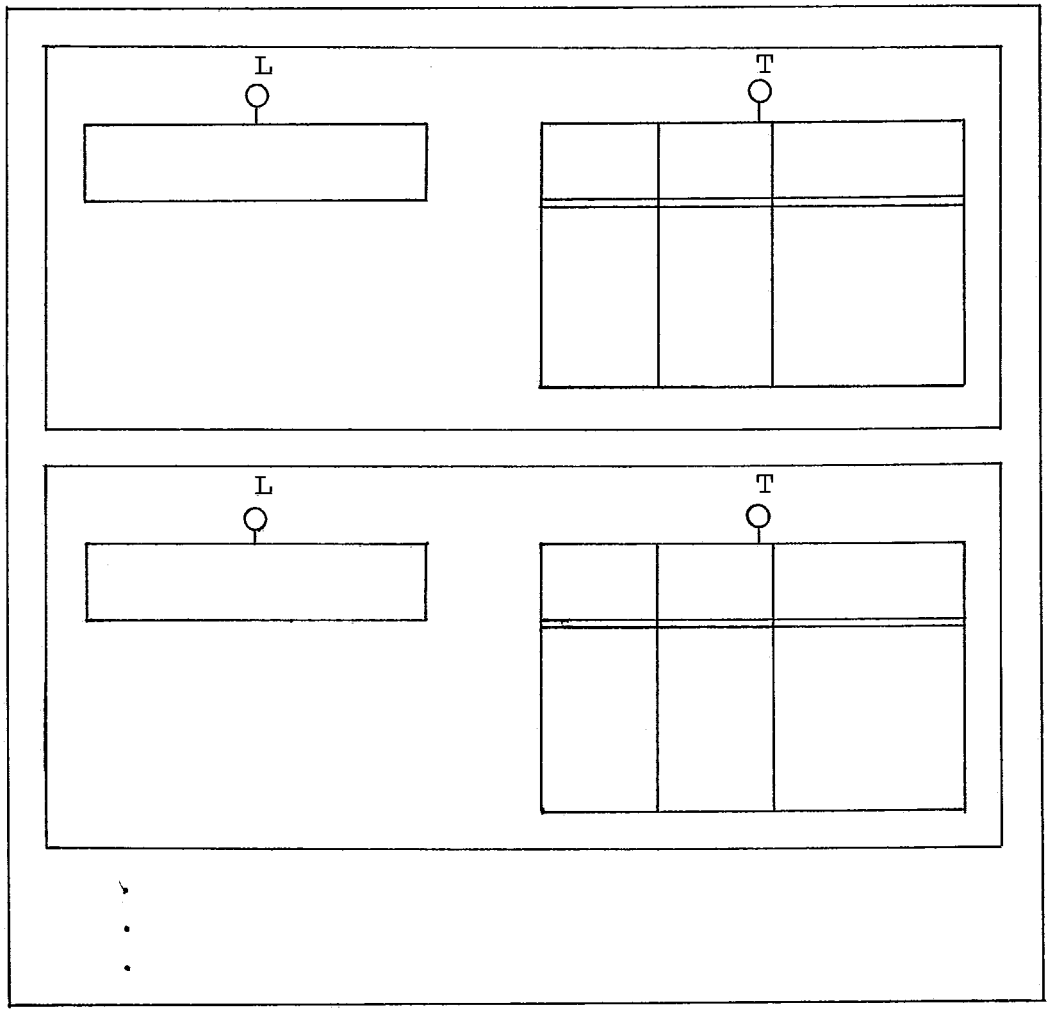


Figura 3.6

Tem-se assim, com estes dois tipos de arquivos, duas maneiras de se guardar registro de informações ou tuplas, bem como de estabelecer ligação entre elas. Desta maneira, o usuá

rio pode estruturar informações conforme as abordagens relacional, hierárquica ou de redes.

A figura 3.7 apresenta graficamente o Acervo Total e suas construções mais importantes.

Dentre estes a *FOLHA* é a construção mais complexa, pois contém a descrição do acervo através de verbetes, ou seja, descreve os elementos de dados e seus relacionamentos.

A coleção de verbetes que descreve o Banco de Dados é chamado de *Esquema* em alguns Sistemas de Gerência de Banco de Dados.

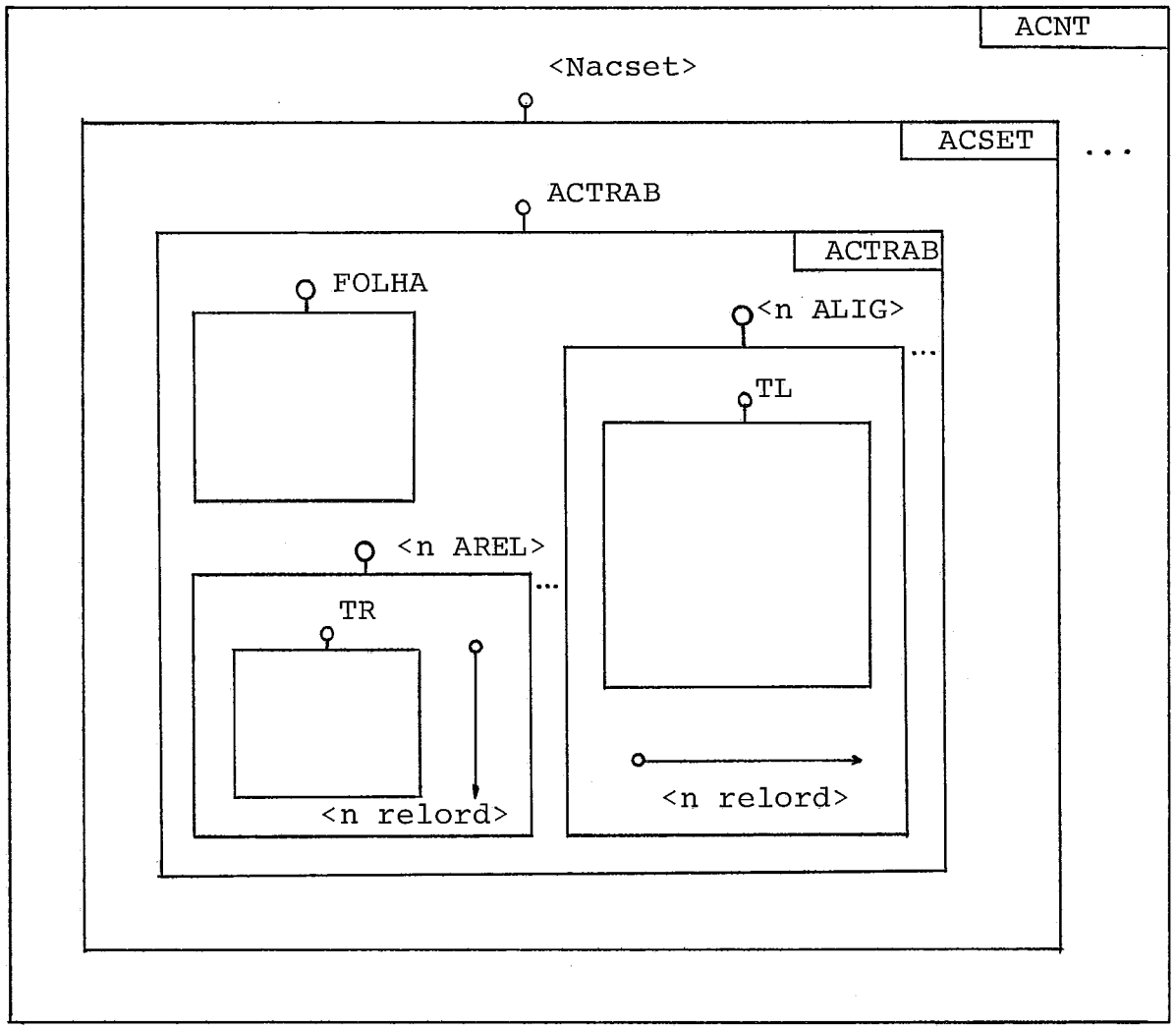


Figura 3.7

3.2.2. O Sistema de Informação

O banco de dados; onde LOBAN é a interface que permite a definição, implantação e operação de sistemas de informação; se constitui das seguintes unidades funcionais:

a) uma base de dados (canal)

b) um canal auxiliar

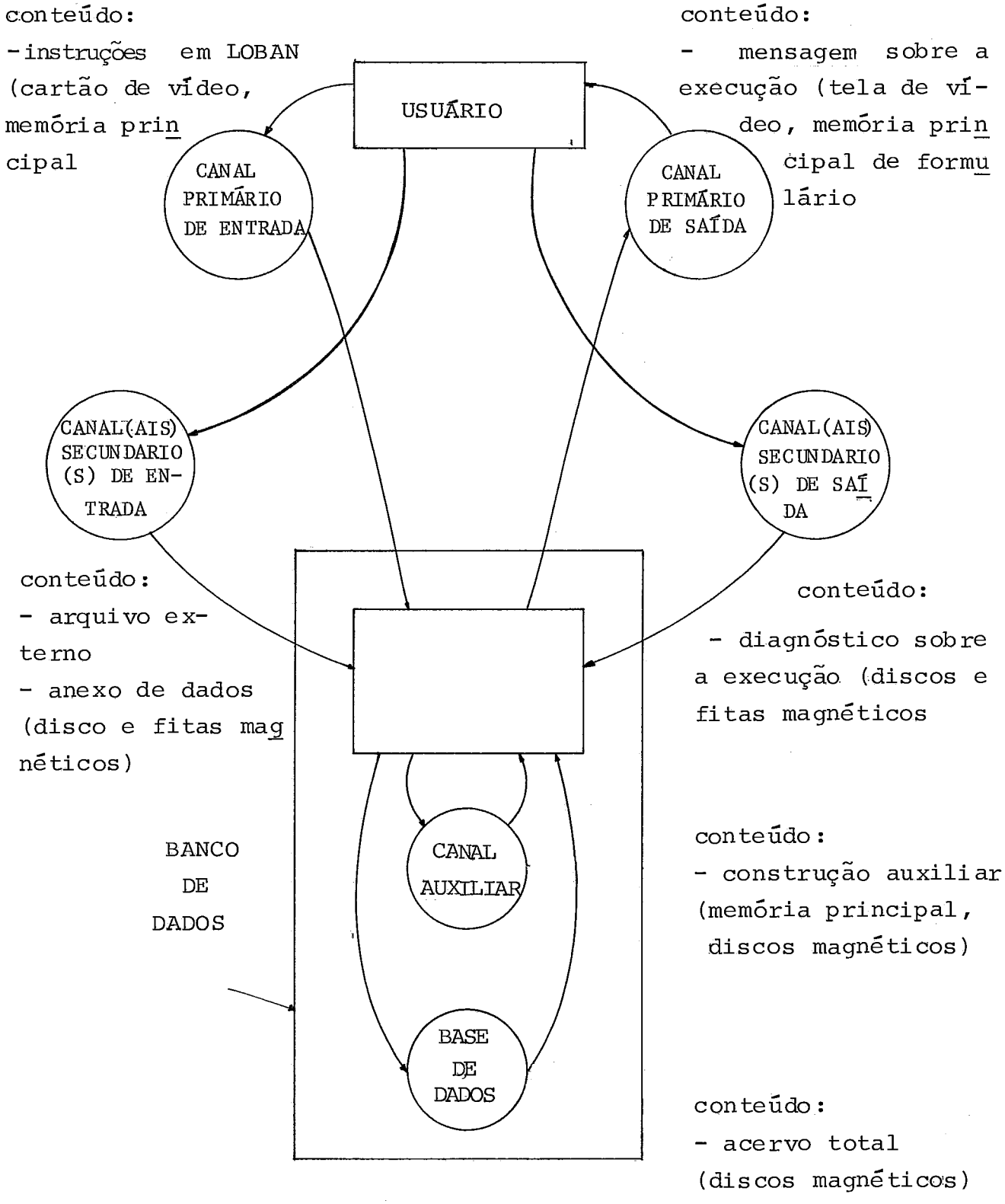
c) um sistema de banco de dados (estação)

A figura 3.8 apresenta graficamente a estrutura funcional do banco de dados e a comunicação com o Usuário.

Um Canal é uma unidade funcional componente do Sistema. Um canal desempenha no sistema o papel da unidade funcional que pode guardar informações, isto é, pode assumir estados que representam informações. A *base de dados* é o canal que armazena as informações segundo uma abordagem determinada, de forma permanente, e onde é possível ao usuário incluir,excluir ou obter informações. Sua realização pode ser feita em unidades de discos magnéticos. O *canal auxiliar* armazena informações temporariamente, durante a execução de um serviço, servindo como "base de dados temporária". Sua realização pode ser feita na memória principal ou em disco magnético, dependendo do sistema portador.

As construções que podem ser geradas no Canal Auxiliar, e também representadas na Base de Dados, são:

- tabela relacional
- ligação
- tabela ligacional
- tupla
- átomo



Obs.: os equipamentos citados entre parênteses são exemplos de realização dos canais.

Figura 3.8

O *Sistema de Banco de Dados* (SBD), é a unidade funcional que faz a gerência da Base de Dados, obedecendo às instruções que lhe forem entregues pela unidade funcional Usuário (realizada através de canais primários). Para tal,está dotado de funções que vão desde a interpretação de instruções, passando pela leitura e armazenamento tanto de entrada de dados como das definições a serem incluídas na própria Base de Dados, até a emissão de relatórios ou diagnósticos sobre as instruções submetidas.

As possibilidades de acesso no Banco de Dados são muitas. Os caminhos da informação permitem transmití-la entre todos os canais. No caso específico do canal da Base de Dados, a informação pode ser movida dentro da própria Base de Dados. A comunicação entre o usuário e o Banco de Dados é realizada através de Canais Primários de entrada e saída. A partir dos Canais Secundários o Banco de Dados pode receber informações (canal secundário de entrada), e fornecer informações (canal secundário de saída).

3.2.3. Conceitos Básicos

Alguns conceitos e termos são fundamentais para o desenvolvimento do presente trabalho.

3.2.3.1. Pretipos de Construção

Um pretipo é um tipo padrão do banco de dados, isto é, pré-estabelecido. Um pretipo e um tipo podem ser compreendidos como conceitos equivalentes ao de conjunto matemático.

Um pretipo é um conjunto de construções, não necessariamente finito nem componente do acervo. É um conjunto ideal, definido por seus predicados. Portanto, todos os seus elementos, isto é, ocorrências, são possuidores dos mesmos predicados comuns ao pretipo. A condição de pertinência de um elemento ao pretipo é satisfazer aos predicados do pretipo.

Os predicados de um pretipo são, todavia, bastante genéricos para permitir a definição de tipos, ou seja, subconjuntos dos pretipos. Estes predicados estão predefinidos, ou embutidos no banco de dados, fazendo parte da interface, como uma das convenções que permitem a comunicação usuário/banco de dados.

Portanto os pretipos são tipos bastante gerais, já previamente definidos (como tipos primitivos), parte integrante das convenções a nível da interface, que permitem ao usuário definir novos tipos, de acordo com suas necessidades. Como exemplo de pretipos referimo-nos àqueles descritos no ítem 3.2.1, a saber acervo total, acervo setorial, tupla, ligação,

arquivo relacional, arquivo ligacional.

3.2.3.2. Tipos de Construções

Um tipo de construção é um conjunto de construções que possuem os mesmos predicados. Um predicado necessariamente comum às construções de um tipo é a sua pertinência ao mesmo pretipo. A definição de um tipo de construção é feita pelo usuário com referência a um pretipo.

Portanto, ao se definir um tipo de construção, isto é, os predicados de um conjunto de construções, será declarado o seu pretipo. Desta maneira, todos os predicados do pretipo serão atribuídos ao tipo. Além destes, serão declarados os demais predicados que permitirão distinguir este subconjunto, o tipo, dentre os demais subconjuntos, outros tipos, pertencentes ao conjunto envolvente, o pretipo. A figura 3.9 mostra alguns tipos definidos dentro de um pretipo, em forma de diagrama.

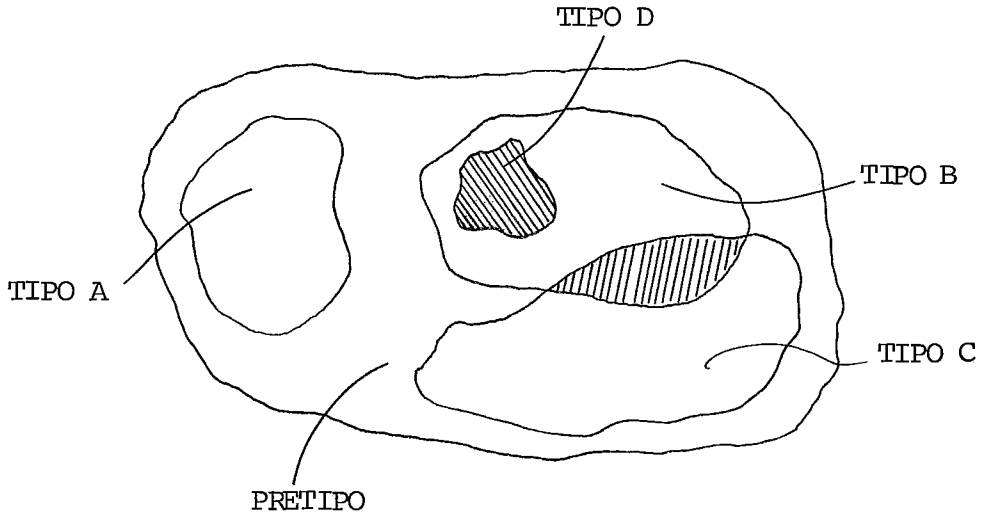


Figura 3.9

No diagrama, a área hachurada, representa as ocorrências dos tipos *B* e *C* ou *B* e *D*. Considerando os tipos *A* e *B* ou *A* e *C*, verificamos que eles constituem conjuntos disjuntos e que não existem ocorrências comuns entre eles. Observamos ainda que o tipo *D* é um subconjunto do tipo *B*, mais abrangente.

3.2.3.3. Consistência

O conceito de consistência está ligado ao conceito de tipo.

Ao apresentarmos a descrição geral do acervo no ítem 3.2.1, ficou evidenciada a maneira de descrever os tipos de construções, através dos tipos dos seus componentes, que também são tipos de construções e assim sucessivamente.

Portanto, cada tipo de construção composta é definida por composição, isto é, pelos seus componentes imediatos, em termos de tipos de construção. A descrição dos tipos de construção mais abrangentes baseia-se na descrição dos componentes de mais baixo nível.

Em cada nível de definição de um tipo de construção (com exceção de átomos), podem ser definidas *conexões*, isto é, relações entre os componentes.

Por exemplo, para um tipo de tupla, pode-se estabelecer que a soma de dois átomos componentes não exceda um determinado valor limite. Chamamos este tipo de relação de *conexão*.

Quando qualquer tipo de construção é definido por composição, a conexão dentro dos componentes imediatos ainda é mantida. Cada componente traz para a construção composta as conexões que foram definidas para os componentes dele.

A consistência é a pertinência ao tipo. Portanto, as condições de consistência são aquelas condições que permitam determinar a pertinência ou não de uma construção considerada ao tipo definido.

3.2.3.4. Coerência

O conceito de coerência está vinculado aos conceitos de consistência, já apresentados,e ao conceito de *persistência.*

A persistência é a qualidade de uma transição da informação representada numa construção, obedecendo a regras para alteração desta.

Cada regra de transição é um par formado por uma construção e um conjunto de construções do mesmo tipo, que é o conjunto de possíveis sucessores da construção. O conjunto de sucessores é um subconjunto de tipo. Em geral, quando não houver outra regra de transição o conjunto de sucessores é o próprio tipo.

Para garantir a persistência, as transições devem ser feitas de acordo com regras que são chamadas regras de transição. As regras de transição permitem fazer alterações nas construções e garantir a sua persistência.

Como exemplo de uma regra de transição é só admitir que um átomo contendo o valor *'CASADO'* possa ser alterado para *'VIÚVO'*, mas não para *'SOLTEIRO'*, porque não é possível a um casado tornar-se solteiro, no caso da informação se referir a uma pessoa.

3.2.3.5. Coleção

Uma coleção é uma construção em que os componentes imediatos são construções sem nome. A representação gráfica de uma coleção está na figura 3.10. A designação padrão para o

pretipo coleção é *COEEÇÃO*.

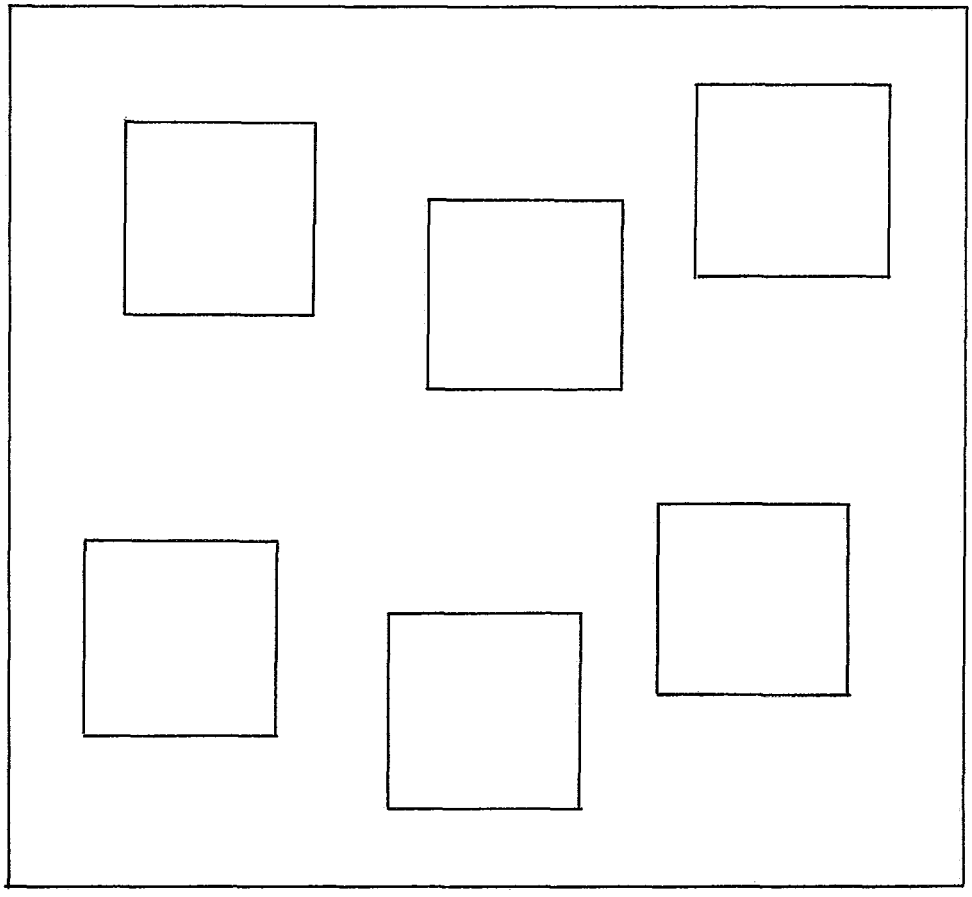


Figura 3.10

3.2.3.6. Nominação

Uma nominação é uma construção em que os componentes imediatos são construções sob nomes. O conceito de nominação corresponde ao de função matemática, isto é, a nominação é definida por partes (nome, componente imediato). Portanto a um dado nome está associado um componente imediato. Em geral, vários nomes podem estar associados a um único componente imediato. Todavia, o inverso não é válido, isto é, não pode haver

vários componentes imediatos associados ao mesmo nome. A representação gráfica de uma nominação está na figura 3.11. A designação padrão para o pretipo nominação é *NOMINAÇÃO*.

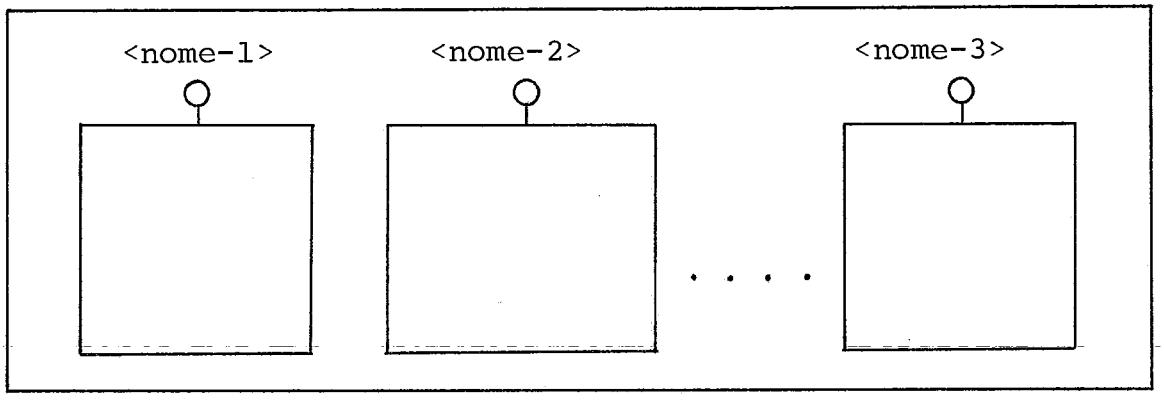


Figura 3.11

3.3. Átomos e Tuplas

3.3.1. Átomos

Átomo é uma designação usada para o conjunto de todas as construções indecomponíveis. A figura 3.12 representa graficamente um átomo. A designação padrão para átomo é *AT* ou *ÁTOMO*.

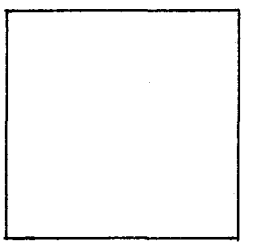


Figura 3.12

3.3.2. Tuplas

Uma tupla é uma nominação de nomes escolhidos pelo usuário, sobre *átomos* ou *tuplas*. É permitida a definição de tuplas, cujos componentes sejam tuplas com quaisquer níveis de embutimento. A designação padrão para tupla é *TUP* ou *TUPLA*. A figura 3.13 apresenta a representação gráfica de tupla.

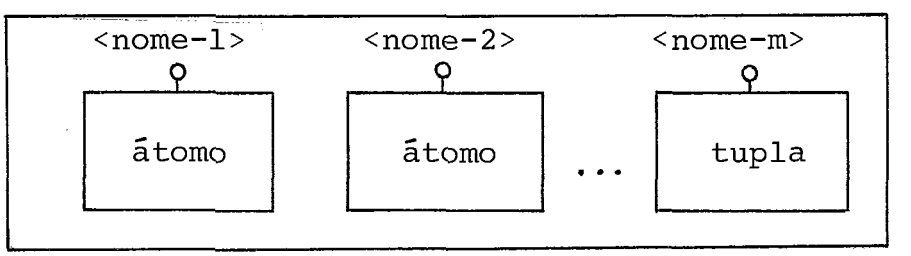


Figura 3.13

3.4. Ligações e Tabelas

3.4.1. Tabelas

Uma tabela é uma *coleção*. Significa que os componentes imediatos são construções sem nome. Todavia uma tabela não é uma coleção qualquer.

Além de serem coleções, seus componentes imediatos, tuplas ou ligações, são *nominações*; segundo, estas nominações têm nomes comuns, ou seja, pertencem ao mesmo conjunto de nomes.

Os nomes dentro dos componentes de uma tabela se aplicam a todas as construções de uma mesma coluna. É possível fatorar estes nomes, isto é, colocá-los em evidência, para aplicá-los a todos os elementos de uma coluna.

A figura 3.14 apresenta graficamente uma tabela, onde

A		B			C	D					...
		B1		B		D1		D2	D3		
A1	A2	B11	B12			D11	D12		D31	D32	

Figura 3.14

todos os nomes estão fatorados sobre a linha dupla. Aparecem, além dos nomes dos componentes imediatos, também os nomes dos

componentes mais internos se forem comuns, em vários níveis, constituindo uma hierarquia.

A uma sequência de nomes obedecendo a uma destas hierarquias, que permitem associar uma coluna ou conjunto de colunas, chama-se de *atributo*.

No exemplo da figura 3.14, são atributos, por exemplo, tanto as sequências de nomes *A.A2* e *D.D3.D32* como as sequências de nomes *A*, *B*, *B1*, *C*.

3.4.2. *Tabela Relacional*

Uma tabela relacional é uma *coleção homogênea de tuplas do mesmo tipo*. A figura 3.15 apresenta um diagrama de uma tabela relacional.

A	B	C	D			E	...	...
			D1	D2	D3			

Figura 3.15

Cada tupla incluída em uma tabela relacional chama-se uma linha da tabela, ou seja, a linha é um ponto onde aparece um componente imediato da tabela relacional. Assim, os atributos são sequência de nomes comuns a todos os elementos de uma coluna, aplicando-se em cada linha. Aplicam-se à tabela relacional todas as observações já feitas para tabelas, em geral. A designação padrão do pretipo tabela relacional é *TAREL* ou *TABELA RELACIONAL*.

3.4.3. Ligação

Uma ligação é uma nominação sobre uma tupla, uma tabela relacional e *uma relação de ordem*. Os nomes da tupla, da tabela relacional e da relação de ordem, são, respectivamente, *L* (de *L*IGANTE) e *T* (de *T*ABELA LIGADA) e <*n relord*>.

A tabela relacional sob o nome *T* é uma tabela ligada ao ligante. Cada linha de uma tabela ligada é chamada de ligado. A designação padrão para ligação é *LIG* ou *LIGAÇÃO*.

A figura 3.16 apresenta graficamente uma ligação.

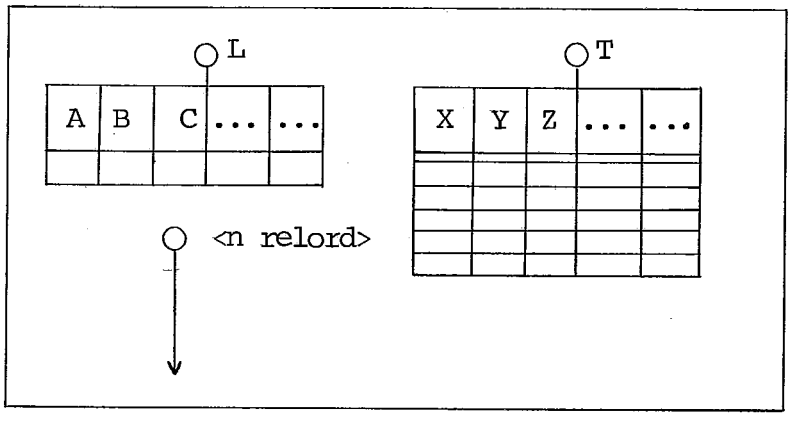


Figura 3.16

3.4.4. Tabela Ligacional

Uma tabela ligacional é uma coleção homogênea de ligações do mesmo tipo.

Aplicam-se à tabela ligacional todas as observações já feitas para tabelas em geral. A designação padrão para ta

bela ligacional é *TALIG* ou *TABELA LIGACIONAL*.

A figura 3.17 apresenta graficamente uma tabela ligacional.

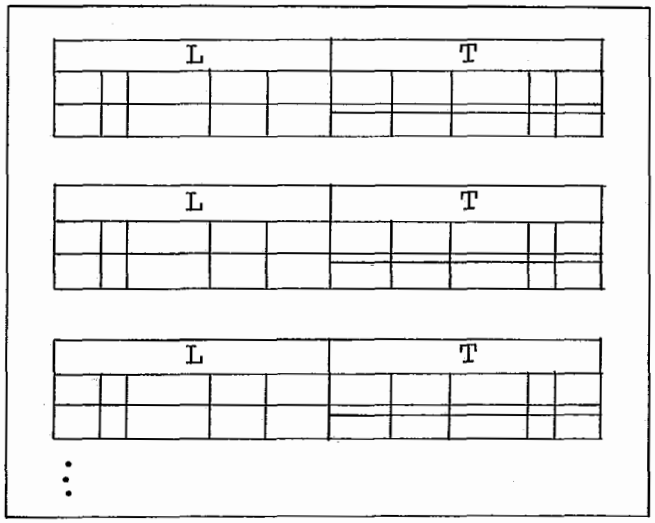


Figura 3.17

3.5. Relação de Ordem

Uma relação de ordem indica a sequência entre tuplas. Estas tuplas podem ser tuplas de uma tabela relacional ou ligantes de uma tabela ligacional. A relação de ordem é apresentada graficamente na figura 3.18, por uma tabela onde cada componente imediato é uma nominação dos nomes *DE* e *PARA* sobre duas tuplas, respectivamente. Estas tuplas são as mesmas sobre as quais se estabelece a ordem. Cada uma dessas nominações indica que a tupla sob nome *DE* antecede imediatamente a tuplas sob nome *PARA* na ordem estabelecida. Como antecessora da primeira tupla e sucessora da última tupla na ordem estabelecida aparece a construção vazia. A designação padrão para relação de ordem é *RELORD* ou *RELAÇÃO DE ORDEM*.

DE	PARA
tupla	tupla
tupla	construção vazia
tupla	tupla
tupla	tupla
construção vazia	tupla

ou

Figura 3.18

3.6. Folha

A definição da *FOLHA* é padrão. Contém a descrição do acervo através de verbetes. Os verbetes da mesma natureza constituem uma coleção de verbetes (CV) e para todos os componentes do pretipo *VERBETE* a designação comum de pretipo é *COMPONENTE DE VERBETE*. A figura 3.19 apresenta graficamente a folha.

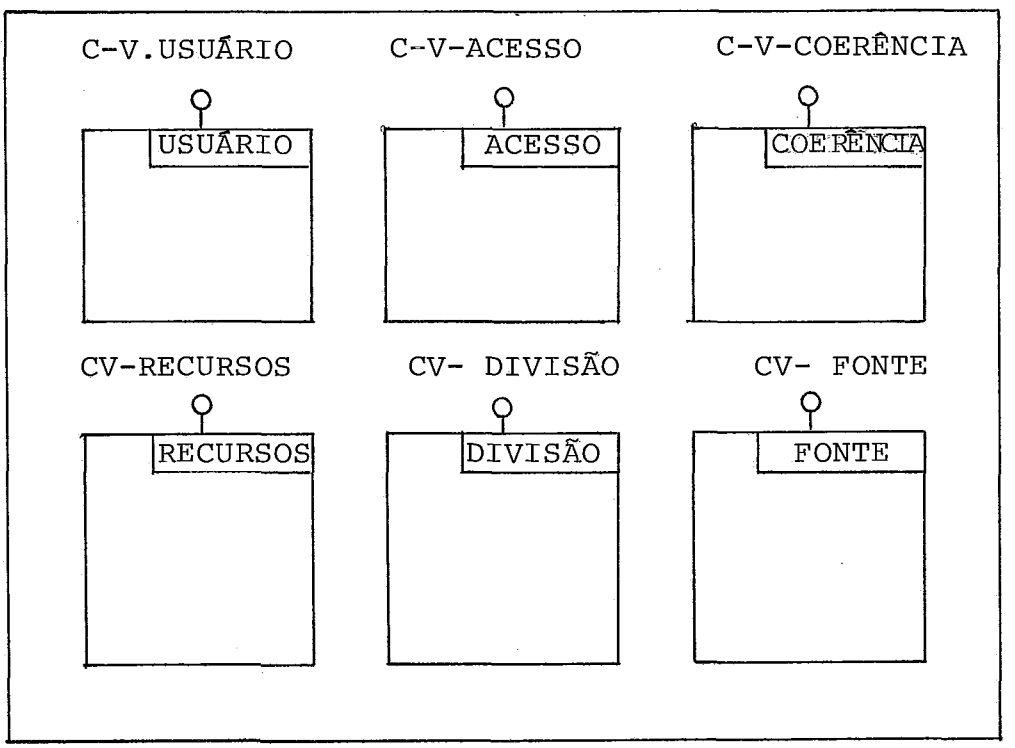


Figura 3.19

A *FOLHA* é constituída de verbetes de Usuário, de Acesso, de Coerência, de Divisão, de Texto-Fonte e de Recursos.

A seguir veremos a composição de cada um destes verbetes, que serão estudados para a sua representação física.

3.6.1. Verbetes de Usuário

Este verbete identifica o usuário para o sistema. Cada usuário tem uma identificação (IDUS) e uma senha de usuário (SENHUS) e poderá ter várias senhas de acesso (SENHAC). A representação gráfica é mostrada na figura 3.20.

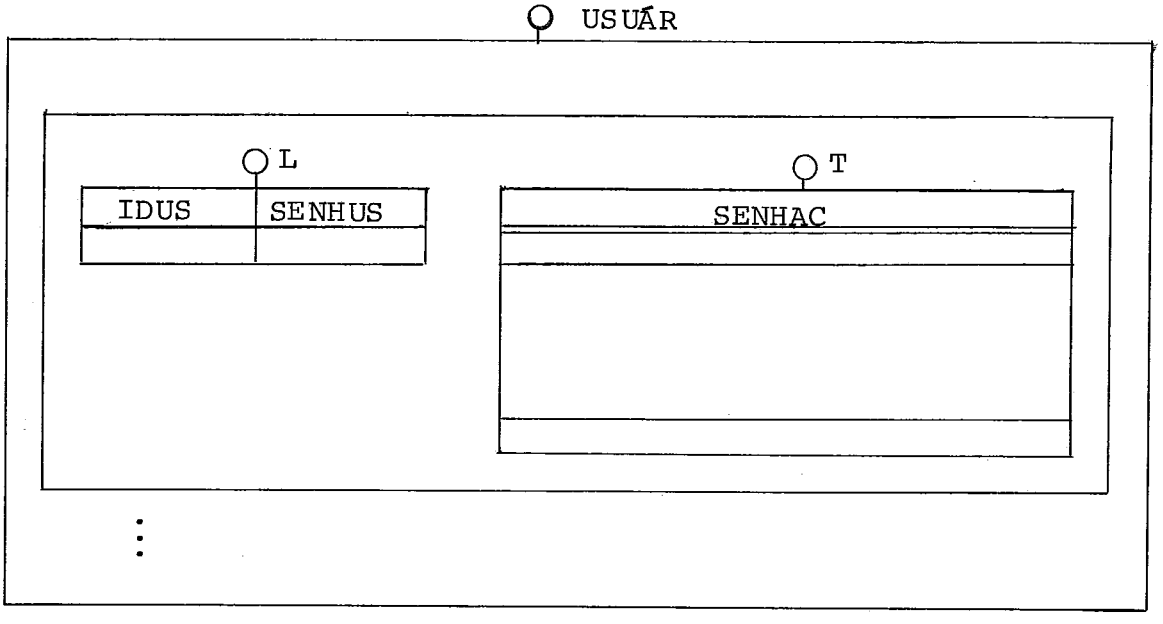


Figura 3.20

3.6.2. Verbetes de Acesso

Este verbete regulamenta o acesso de um Usuário através de sua Senha de Acesso, possibilitando que lhe seja Proibido ou Permitido (VALOR), Gerenciar, Modificar ou Consultar (FUNÇÃO) construções na base dados, mediante uma lista de parâmetros.

A figura 3.21 representa graficamente os verbetes de acesso.

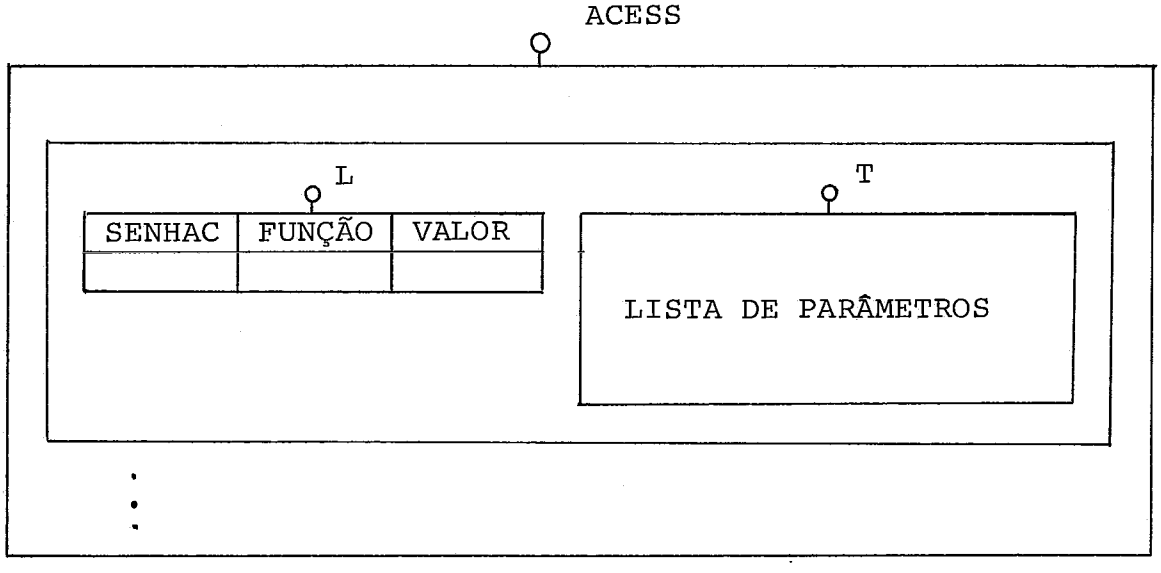


Figura 3.21

3.6.3. Verbetes de Coerência

A coerência define os acervos admitidos na Base de Dados bem como as transições admitidas de um acervo para outro resultando de uma operação de alteração.

Um verbete de coerência define tanto a consistência, isto é, um tipo de construção, quanto a persistência das ocorrências deste tipo, isto é, as transições admitidas em todos os pontos onde elas aparecerão.

O verbete de coerência pode ser ou uma combinação de *verbete de consistência* com *verbete de persistência* ou um *verbete de ordenação*.

O verbete de consistência, permite verificar a pertinência de uma dada construção ao tipo em definição.

O verbete de persistência, é no fundo uma comparação da antiga ocorrência em um dado ponto (PANTES), com a nova ocorrência deste tipo no ponto correspondente a PANTES (PDEPOIS).

Nos verbetes de coerência *DTIPO* é o nome que o usuário chamou a construção que ele está definindo e a lista de parâmetros definem nominações e coleções. A figura 3.22 representa graficamente os verbetes de coerência.

Um verbete de ordenação (figura 3.22) desempenha um papel particular por ser ele simultaneamente um verbete de consistência e de persistência. Uma relação de ordem é aplicada sobre um determinado tipo de tupla ou de ligação onde a lista de parâmetros definem por quais atributos e de que forma será a ordenação.

Como termos de composição para os verbetes de coerên

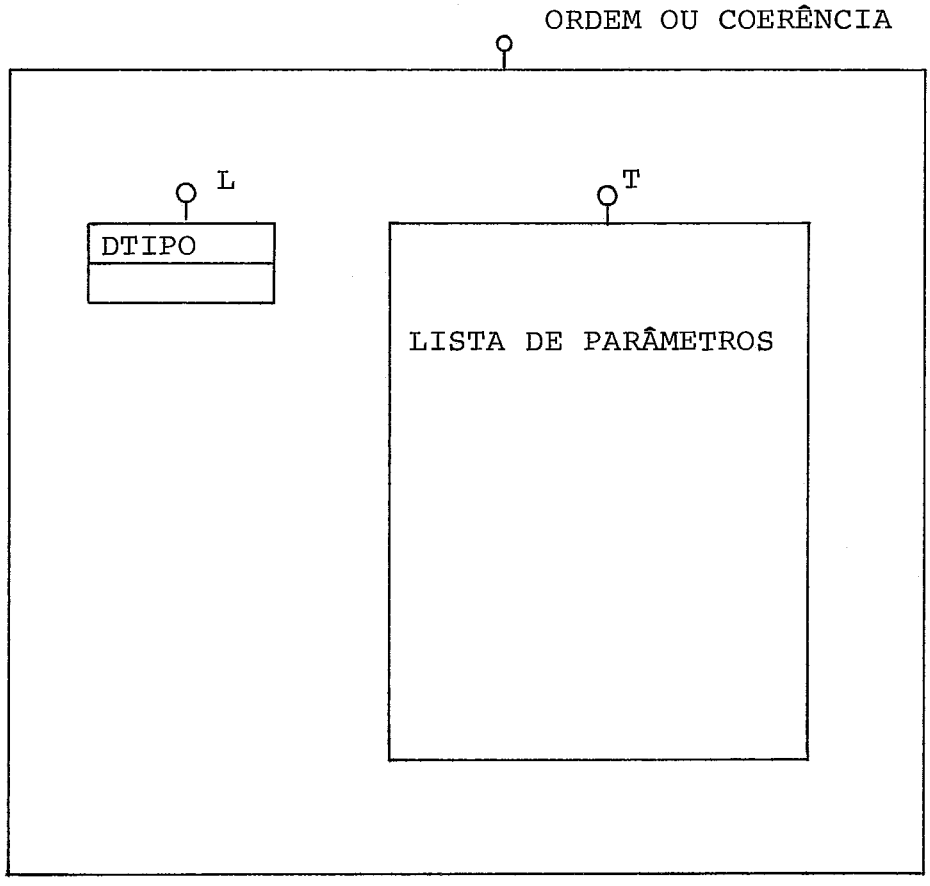


Figura 3.22

cia tem-se as possibilidades:

Para Nominações:

Uma composição para nominações é uma tabela ligacional de verbetes de coerência por nominação (figura 3.22). Da mesma forma a conexão e a persistência irão compor tabelas ligacionais. A lista de parâmetros da composição para nominações especifica as suas construções componentes.

O usuário ao descrever um verbete de coerência pode estabelecer conexões através da lista de parâmetros entre os seus componentes e definir chaves para a tabela, relações de ordem do tipo ordem por enumeração, pedir que o sistema tome

providências para ajustar construções, emitindo ou não mensagens.

O usuário ao descrever o verbete também pode estabelecer uma persistência através da lista de parâmetros que define restrições para uma ocorrência do tipo em definição no que diz respeito à sua primeira aparição, às suas alterações, e ao seu desaparecimento; também pode definir uma comparação entre construções antes de INCLUSÕES, ALTERAÇÕES e EXCLUSÕES em relação às construções que irão substituí-las.

Para Coleções

O usuário ao definir uma coleção no verbete de coerência designa através da lista de parâmetros (figura 3.22) um tipo de construção já definida.

3.6.4. Verbetes de Divisão

Para definir as estruturas nos canais de entrada e saída, o usuário disporá das expressões chamadas de *verbetes de divisão*.

Um verbete de divisão especifica um tipo de máscara, ou seja, um conjunto de máscaras capazes de serem sobrepostas a um campo dado. Por isso é que chama-se de *divisão* um tipo de máscara, significando isso em outras palavras uma "máscara de subdividir um campo em subcampos". Na medida em que há analogia com os tipos de construção (divisão ... tipo de construção, máscara ... ocorrência), a sintaxe dos verbetes de divisão visam refletir essa semelhança.

Um verbete de divisão corresponde a uma construção com a estrutura apresentada na figura 3.23.

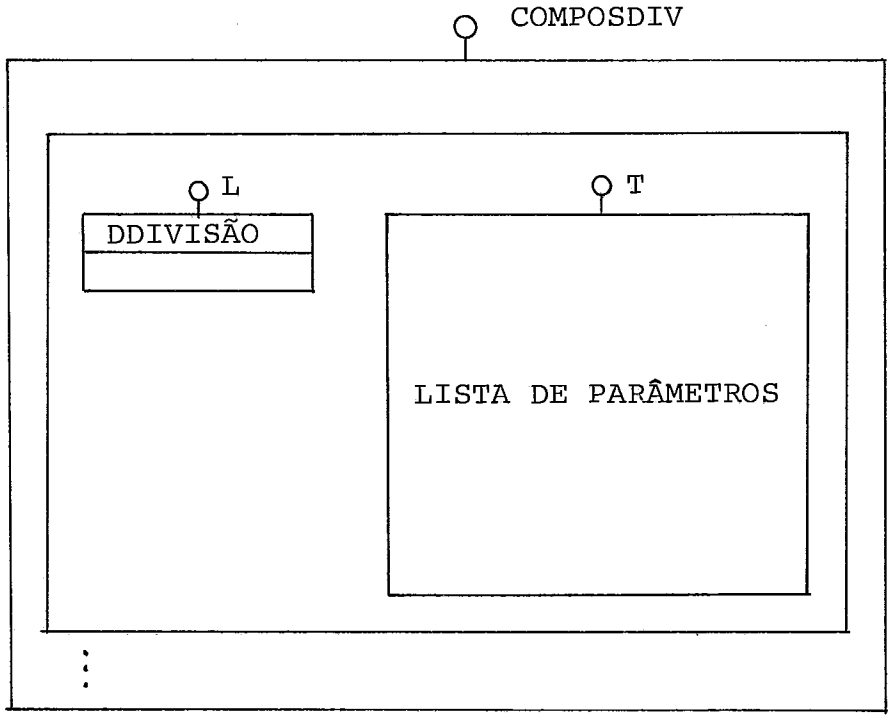


Figura 3.23

O usuário atribui um nome (DDIVISÃO) ao seu formato (máscara) de entrada e saída e através da lista de parâmetros especifica o tipo de máscara e uma ou mais entradas do seguinte conjunto: nome de campo; número de ocorrências do campo em definição no campo abrangente; formato, localização e tipo do campo.

3.6.5. *Verbetes de Texto Fonte*

Por meio de um verbete de texto fonte, o usuário pode armazenar textos fonte de LOBAN na folha, para referência futura a partir de instruções de trabalho.

A estrutura de um verbete de texto fonte, como componente imediato da construção CV-TEXTO da FOLHA, é representada na figura 3.24.

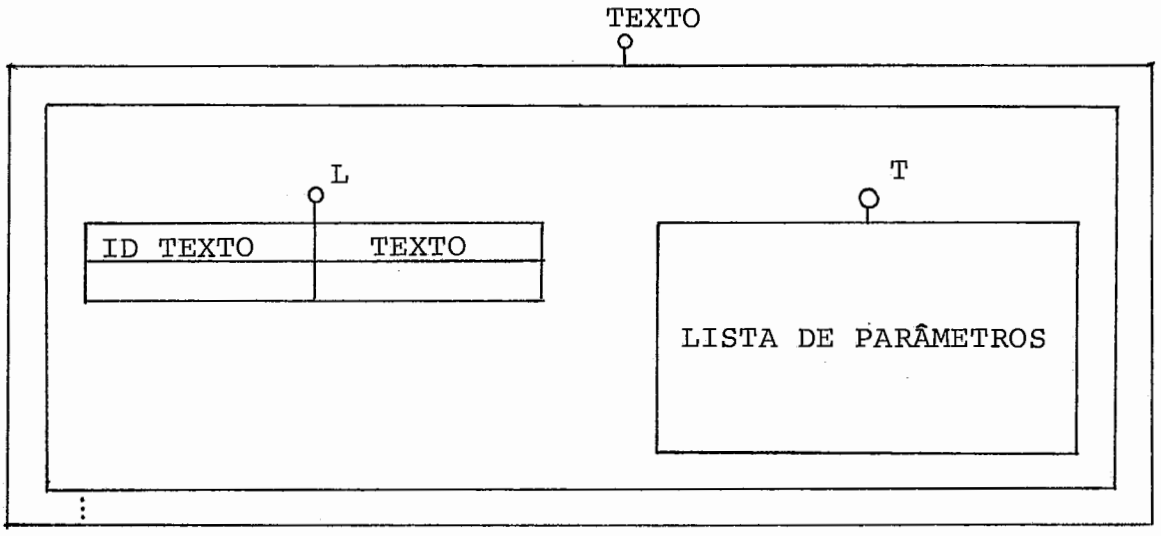


Figura 3.24

Ao definir um verbete de Texto Fonte, o usuário fornece a identificação do texto fonte (IDTEXTO), o texto fonte em LOBAN (TEXTO) e uma lista de parâmetros formais opcionais, onde cada parâmetro tem que aparecer pelo menos uma vez no texto.

3.6.6. *Verbetes de Utilização de Recursos*

Através dos verbetes de utilização de recursos, o usuário orienta o sistema para algumas políticas a serem seguidas na utilização de recursos do sistema portador.

A prestação de um serviço pelo sistema, provoca custos para o usuário, pois o sistema de Banco de Dados (SBD) tem que utilizar recursos (memória, UCP, etc.), utilização essa que contribui aos custos mencionados.

Na medida que o SBD seja capaz de escolher entre várias maneiras de utilização de recursos, para realizar esta escolha é necessário que este seja avisado das preferências e políticas do usuário. Por isso foram previstos os verbetes de utilização de recursos, através dos quais o SBD é avisado das preferências e políticas de utilização desejadas pelo usuário na prestação de serviços.

A representação de um verbete de utilização de recursos, como componente imediato da construção CV-RECURSOS, é mostrada na figura 3.25.

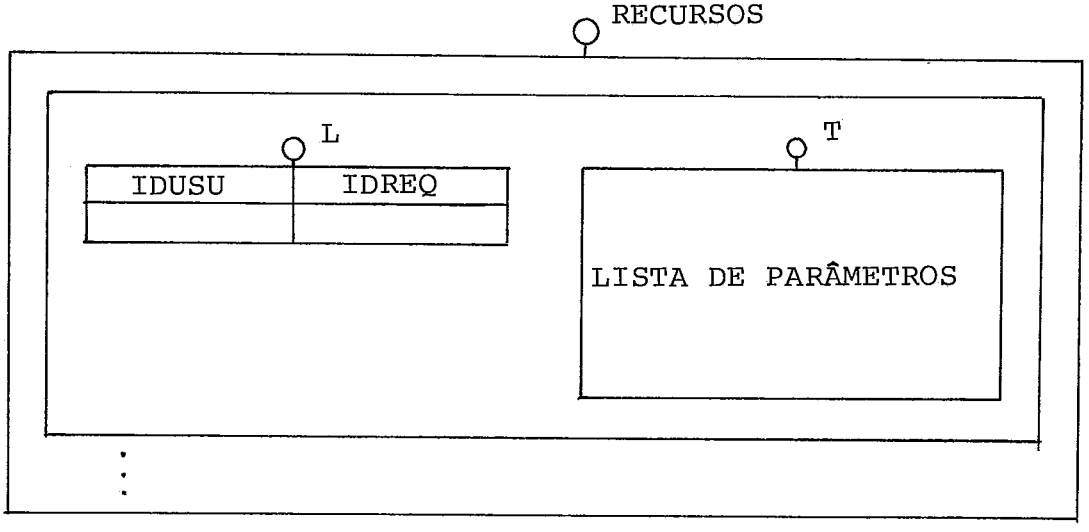


Figura 3.25

Ao definir o verbete de Utilização de Recursos,o usuário fornece: o identificador do verbete que está definindo (IDREQ); a sua identificação (IDUSU), e através da lista de parâmetros especifica a política de armazenamento das construções, ou especifica um conjunto de instruções que deve ser otimizada.

4. ESTRUTURAS DE DADOS PARA A IMPLEMENTAÇÃO DA INTERFACE

Quando se fala sobre banco de dados, deve-se referir como uma coleção de dados mutuamente relacionadas, armazenados em algum meio físico (hardware), para serem consultados e manipulados a partir de aplicações (programas) diversas.

Este capítulo apresentará um conjunto de estruturas de dados para suportar a maior parte das construções exibidas no capítulo anterior.

Na apresentação sumária de cada um dos verbetes que fizemos anteriormente, muitas vezes os elementos que descrevem os verbetes através da lista de parâmetros fazem referências a cadeias de caracteres, textos e endereçamento de construções que caracterizam informações de tamanho variável, assim é sugerido a estruturação para o armazenamento destas informações.

Uma idéia que surge é aplicar uma função de hash em todas as cadeias e gerenciar um espaço de memória para armazená-las.

A figura 4.1 representa graficamente esta idéia.

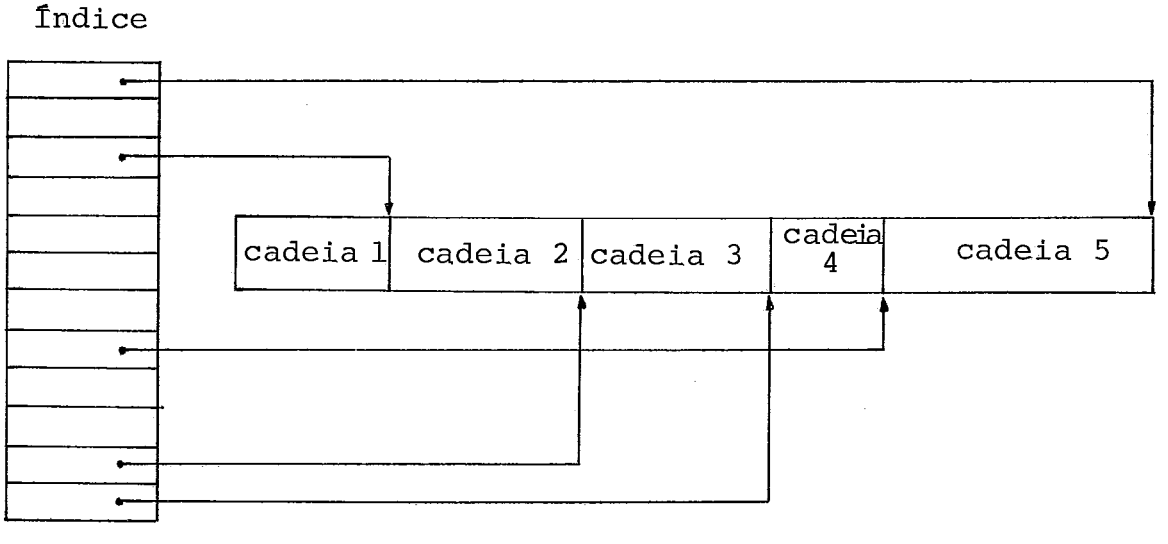


Figura 4.1

Deve haver também um ponteiro do elemento que tem cadeia para a área ou um ponteiro indireto através do índice. Em qualquer dos casos é necessário informar o tamanho da cadeia.

4.1. Representação para Tuplas

As tuplas constituem os registros de dados do usuário.

É sugerido que sejam armazenados contiguamente e para sua recuperação e reconhecimento deve ser criado um descritor que identifica os seus atributos bem como suas posições relativas dentro do registro. A figura 4.2 representa graficamente esta idéia.

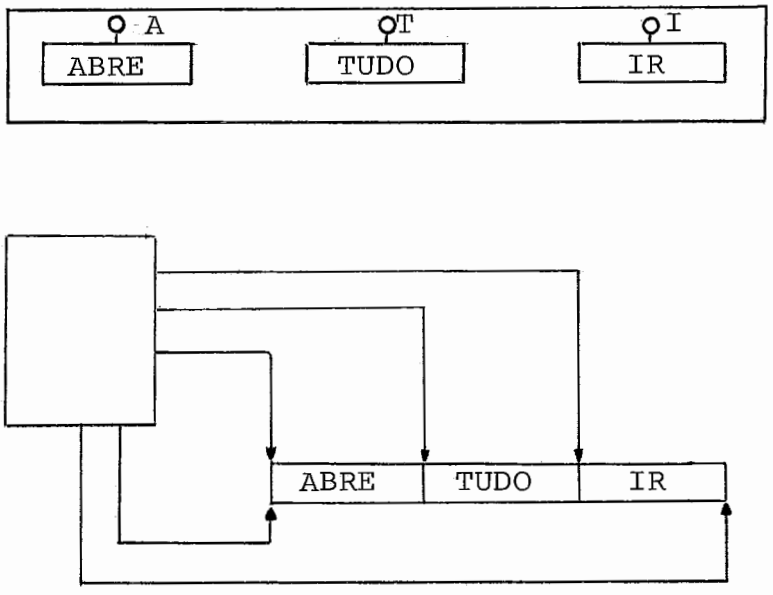


Figura 4.2

4.2. Representação de Tabela Relacional

Uma tabela relacional foi definida no capítulo anterior como sendo uma coleção homogênea de tuplas do mesmo tipo. A sua estrutura de armazenamento pode ser qualquer uma das mencionadas no capítulo 2.

Definida a chave ou chaves da tabela, algumas estruturas como árvore-B e hash apresentam mais vantagens que listas e filas. Outros fatores que influem muito na hora de realizar a implementação em computador, são: hardware portador, o software básico disponível bem como os *padrões de utilização* que serão dadas a estas estruturas.

Os critérios que determinam a escolha da organização física são diferentes daqueles que determinam a organização lógica. A seleção de estruturas físicas deve ser determinada pela necessidade de eficiência operacional, tempos de resposta rápidas e minimização de custos.

Algumas questões se colocam e podemos relacionar uma série de critérios que afetam a seleção da estrutura física. Estes critérios seriam: economia de espaço, minimização de redundância, processamento sequencial ou aleatório?, atividade dos dados do arquivo, necessidades de tempo de respostas, volatilidade dos dados, crescimento da estrutura, recuperação por chaves múltiplas, etc.

Estes critérios devem ser levados em consideração na escolha das estruturas sugeridas para a implementação.

4.2.1. Sugestão 1

O arquivo de dados é criado e as tuplas dele são armazenadas contiguamente.

Cria-se um índice em Árvore-B^+ com a(s) chave(s) da tabela. As folhas da árvore conterão todas as chaves além de um ponteiro para a informação no arquivo sequencial. Os nós da folha poderão estar duplamente encadeados para facilitar a política de ocupação do nó, bem como facilitar algum tipo de pesquisa. Inclusões e alterações são feitas normalmente solicitando-se espaço quando necessário ou superestimando o espaço inicial alocado. Exclusões podem ser feitas fisicamente quando ocorrerem ou anotadas no arquivo para uma posterior reorganização do índice e do arquivo.

A figura 4.3 apresenta graficamente esta sugestão.

A estrutura apresenta as seguintes vantagens e desvantagens:

1. Vantagens

- garante busca sequencial rápida através das folhas;
- inclusões, alterações e exclusões são realizadas através do índice em árvore-B^+;
- não precisa reorganização do índice;
- espaço de memória da ordem do tamanho do nó;
- os registros podem estar nas folhas e desta forma pode-se desprezar o arquivo com os registros.

2. Desvantagens

- necessidade de deslocamentos até a folha para

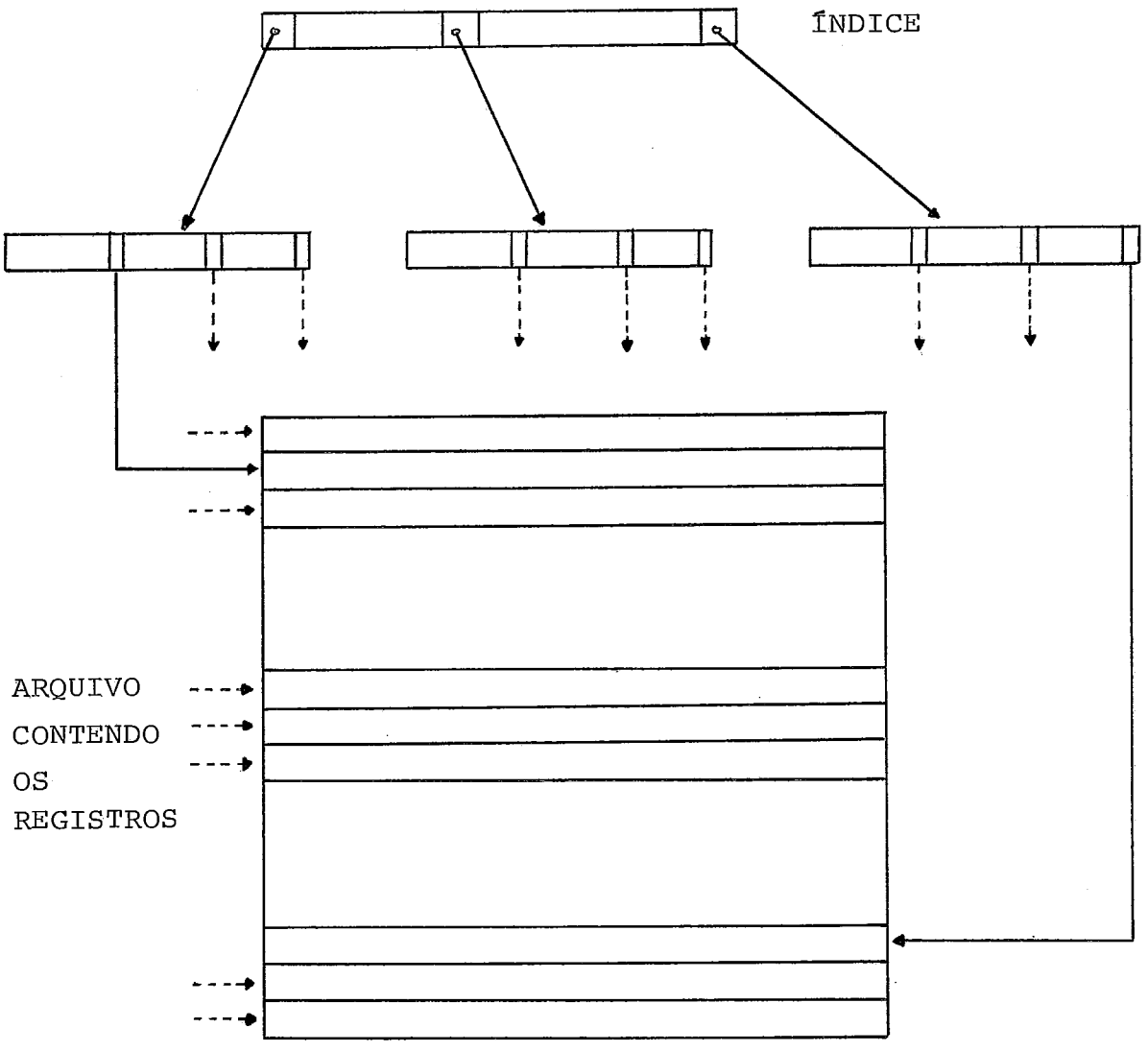


Figura 4.3

buscar informações;

- espaço para chaves e ponteiros;
- além da busca da folha, é necessário uma busca binária em cada folha;
- pesquisa para registros não chaves muito difícil;
- necessidade de dividir os nós quando der over flow ou concatenação quando der underflow.

4.2.2. Sugestão 2

O arquivo de dados é criado e armazenado contiguamente.

Cria-se um índice utilizando-se um algorítmo de hash. Cada registro do índice conterá além da chave um ponteiro para o arquivo sequencial. A política de colisão poderá ser escolhida entre os algorítmos apresentados no capítulo 2. Alterações são feitas normalmente. Inclusões devem seguir a política do algorítmo escolhido, eventualmente poderá ser necessário inicializar o índice novamente por problemas de espaço. Exclusões podem ser executadas seguindo a política do algorítmo e no arquivo de dados podem ser realizadas fisicamente ou anotadas para um algorítmo de acertos. A figura 4.4 apresenta graficamente esta sugestão.

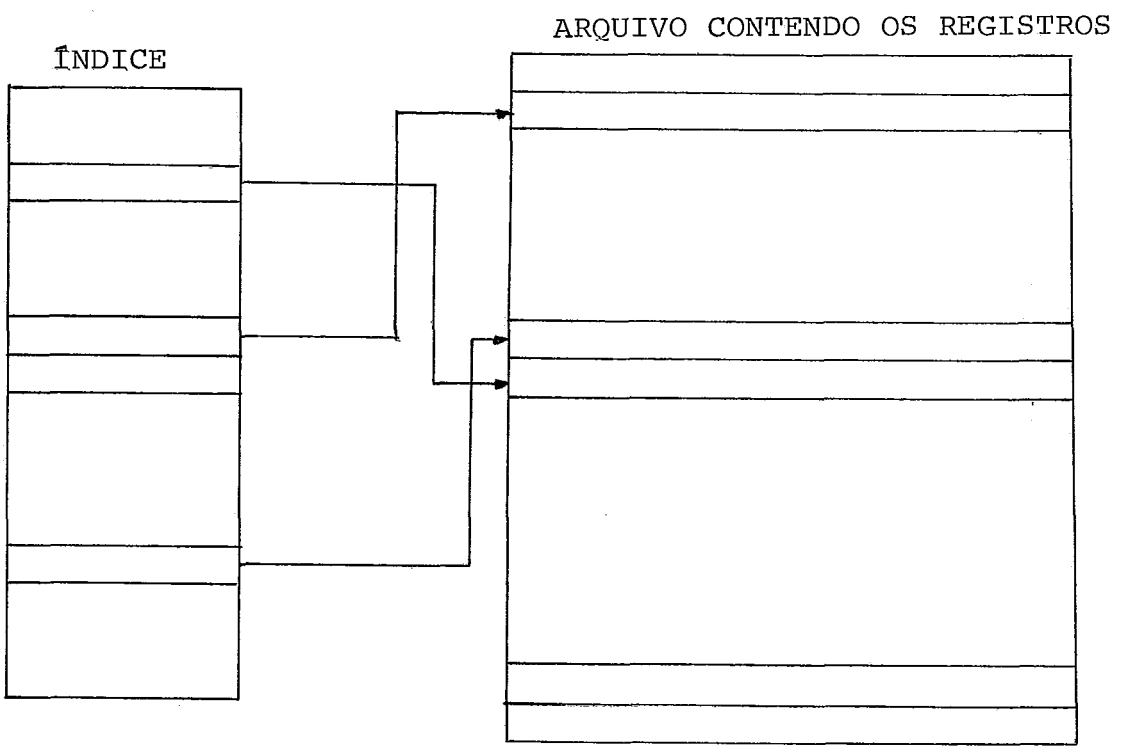


Figura 4.4

A estrutura apresenta as seguintes vantagens e desvantagens:

1. *Vantagens*

   - buscas muito rápidas quando realizadas através da chave;
   - eficiente com buckets de tamanho maior que 20 - HILL[26];
   - inclusões de chave resolvidas rapidamente;

2. *Desvantagens*

   - pode ter baixa ocupação de memória;
   - busca por chaves ordenadas muito demorado;
   - necessidade de reorganização do índice quando da ocupação alta do índice.

4.2.3. Sugestão 3

Outra representação seria construir um índice em Árvore Binária, e os registros de dados armazenados contiguamente. A figura 4.5 apresenta graficamente esta idéia.

A escolha na implementação pode ficar a critério do SBD desde que ele tenha informações para decidir qual o tipo de índice para os dados que serão fornecidos mediante informações do usuário. A implementação pode seguir um único tipo de índice ou nenhum e desta forma as buscas ocorrerão sobre o arquivo de dados.

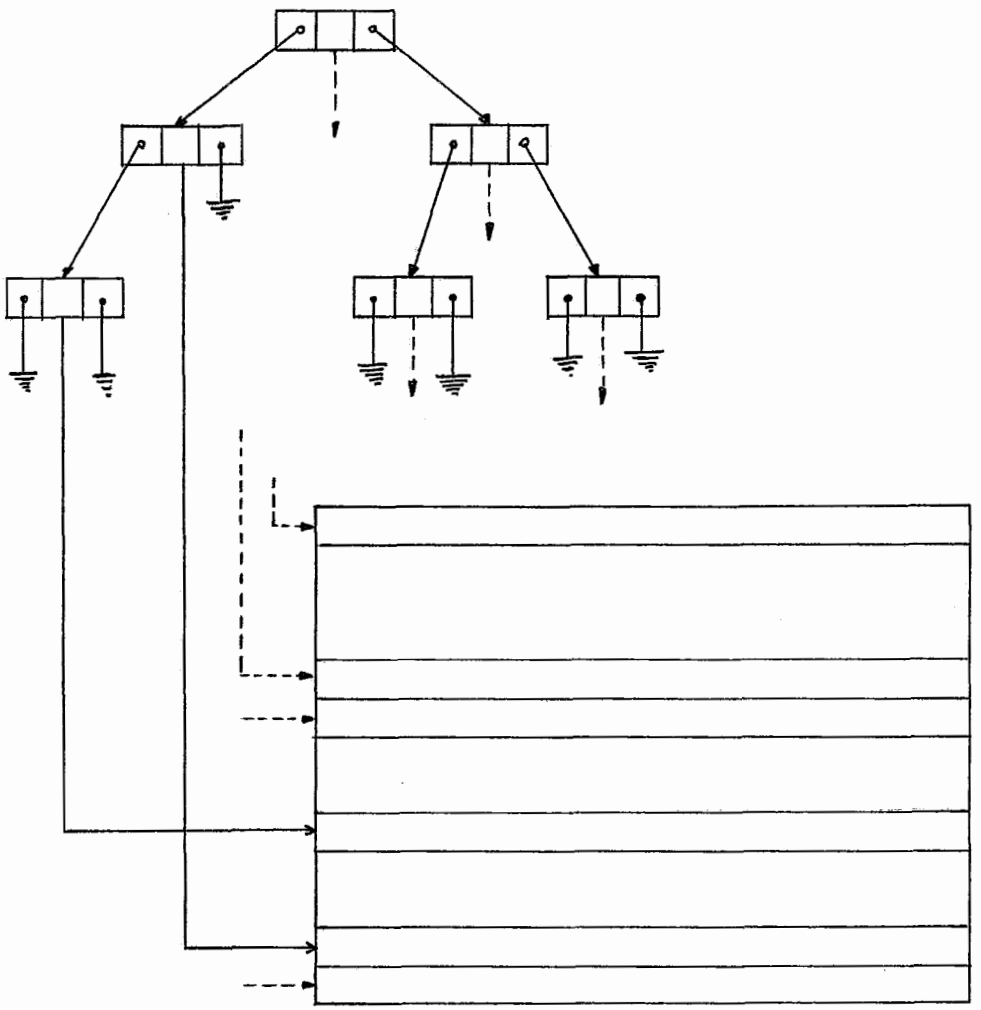


Figura 4.5

A estrutura apresenta as seguintes vantagens e desvantagens:

1. Vantagens

- inclusões e exclusões realizadas rápida e eficientemente;
- é apropriado para a memória e quando não ocupa muito espaço.

2. Desvantagens

- necessidade de ponteiros;
- busca por chaves ordenadas muito demorada;
- muitos acessos a disco se estiver armazenado em disco.

4.3. Representação de Tabela Ligacional

Uma tabela ligacional foi definida no capítulo 3 como sendo uma coleção homogênea de ligações do mesmo tipo. Cada ligação é constituída de uma tupla chamada ligante e uma tabela relacional chamada ligada.

O ligante "contém" uma ou mais chaves. A tabela ligada também "contém" uma ou mais chaves. Estas chaves são definidas através dos verbetes de coerência.

Para o caso de uma tabela ligacional vários casos se apresentam.

4.3.1. Sugestão 1

Transformar a tabela ligacional em uma tabela relacional e escolher uma alternativa para se implementar tabelas relacionais. A transformação consiste em fazer para cada ligação um desagrupamento do ligante com seus ligados. Este desagrupamento consiste em juntar cada tupla ligada com seu ligante, desaparecendo os nomes padrão *L* e *T*. A figura 4.6 mostra esta transformação.

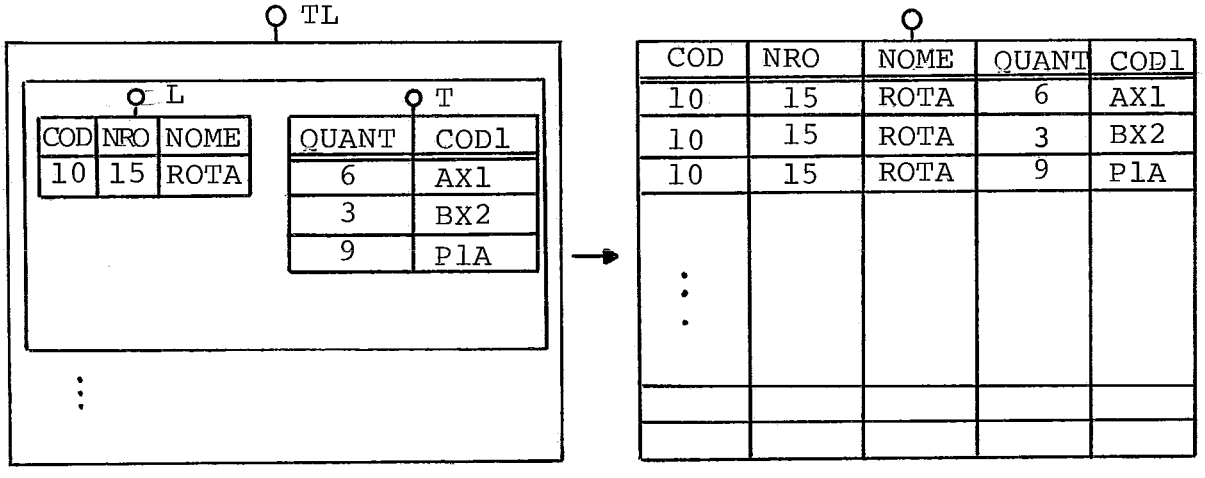


Figura 4.6

4.3.2. Sugestão 2

Ligantes e Ligados armazenados por contiguidade em áreas separadas. Cada ligante deve ter um ponteiro para um Ligado qualquer e estes devem estar encadeados.

É criado um índice para as chaves dos ligantes (árvore-B ou suas variações, hash ou árvore binária). Cada nó além de ter os ponteiros normais do seu tipo de índice tem um ponteiro para o seu ligado. Os ligados devem estar encadeados por ligante e fazer referência a este individualmente ou no final da lista.

Índices sobre ligados não são uma boa idéia, pois ligados podem ocorrer em diferentes tabelas de ligados, e devido a estas ocorrências em duplicatas, o controle se torna muito oneroso.

Inclusões, alterações e exclusões devem ser realizadas através do índice de ligantes com marcação ou não das chaves excluídas, conforme o algorítmo escolhido.

A figura 4.7 apresenta graficamente esta idéia.

As sugestões feitas para se acessar os Ligantes apresentam as mesmas vantagens e desvantagens discutidas anteriormente para a implementação de tabela relacional.

A estrutura proposta para os ligados é característica de uma pilha encadeada. Inclusões são efetuadas rapidamente colocando-se o Ligado no topo da pilha ao qual ele pertence. Percorrer todos os Ligados significa percorrer do topo até a base através dos ponteiros. Exclusões também são realizadas sem problemas.

Para a estrutura proposta apresenta-se outra desvanta

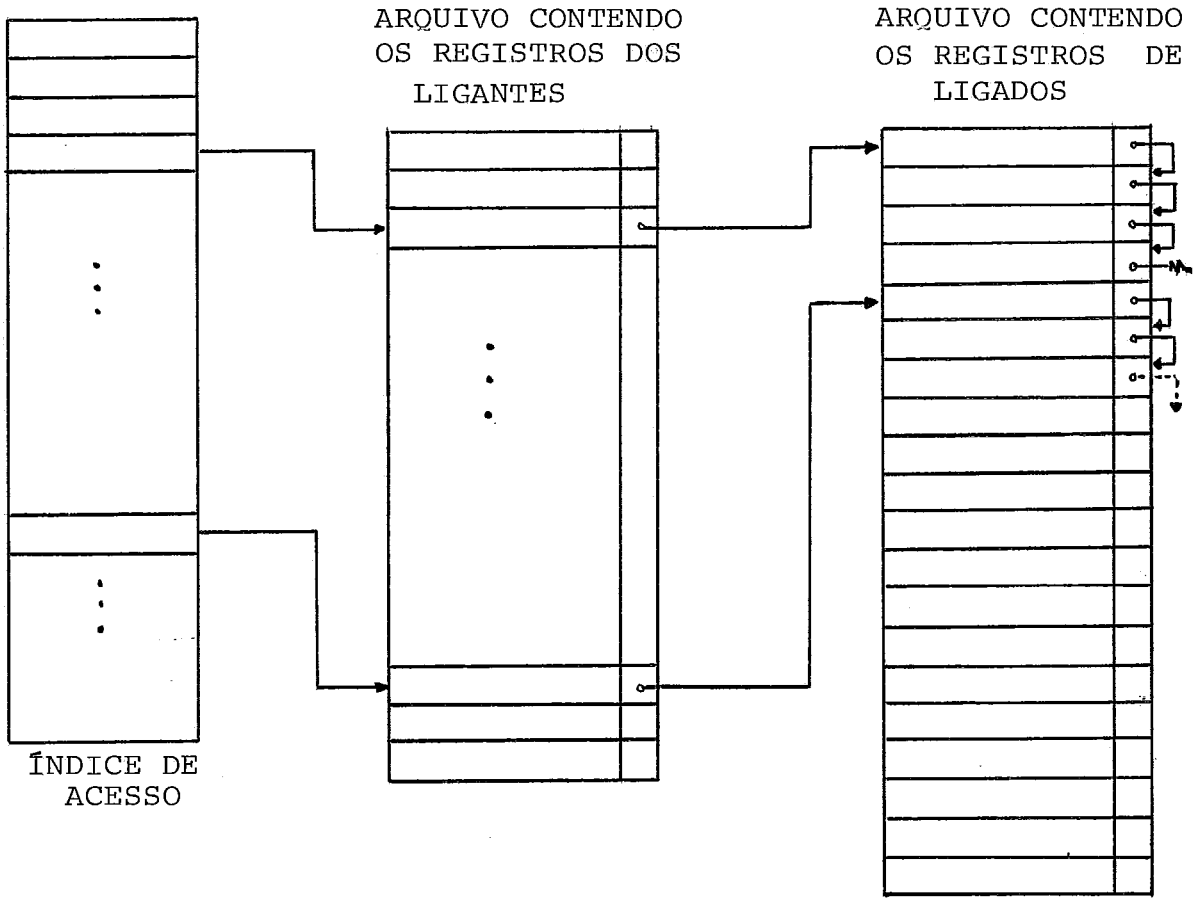


Figura 4.7

gem que é a demora, para se tomar um registro disponível na área reservada para os Ligados, ou devolver o registro. Por outro lado esta desvantagem torna-se uma vantagem, pois permite que Ligados pertencentes a Ligantes diferentes estejam juntos numa mesma área.

4.3.3. Sugestão 3

Ligantes e Ligados são armazenados sequencialmente em áreas separadas.

Da transformação efetuada na sugestão 1 usamos a combinação de chave(s) dos ligantes com a(s) chave(s) dos ligados. É criado um índice em Árvore-B^+, onde todos os nós folhas são duplamente encadeados e contém ponteiros tanto para os ligantes como para os ligados.

Entradas no índice são realizadas através das chaves do ligante e ligados. O caso limite é quando os ligados não têm chave definida e o índice só é definido pela(s) chave(s) dos ligantes.

Inclusões, alterações e exclusões são realizadas através do índice.

Podemos usar a idéia de combinação de chaves com um Algorítmo de Hash, no lugar de uma Árvore-B^+. A figura 4.8 apresenta graficamente esta sugestão.

A estrutura apresenta as seguintes vantagens e desvantagens:

1. Vantagens

- permite acessar a tabela ligacional através das chaves de Ligantes e Ligados;
- permite a exclusão de Ligados rapidamente;
- se o índice for implementado em Árvore-B^+, permite o acesso sequencial por chave para os Ligantes e Ligados.

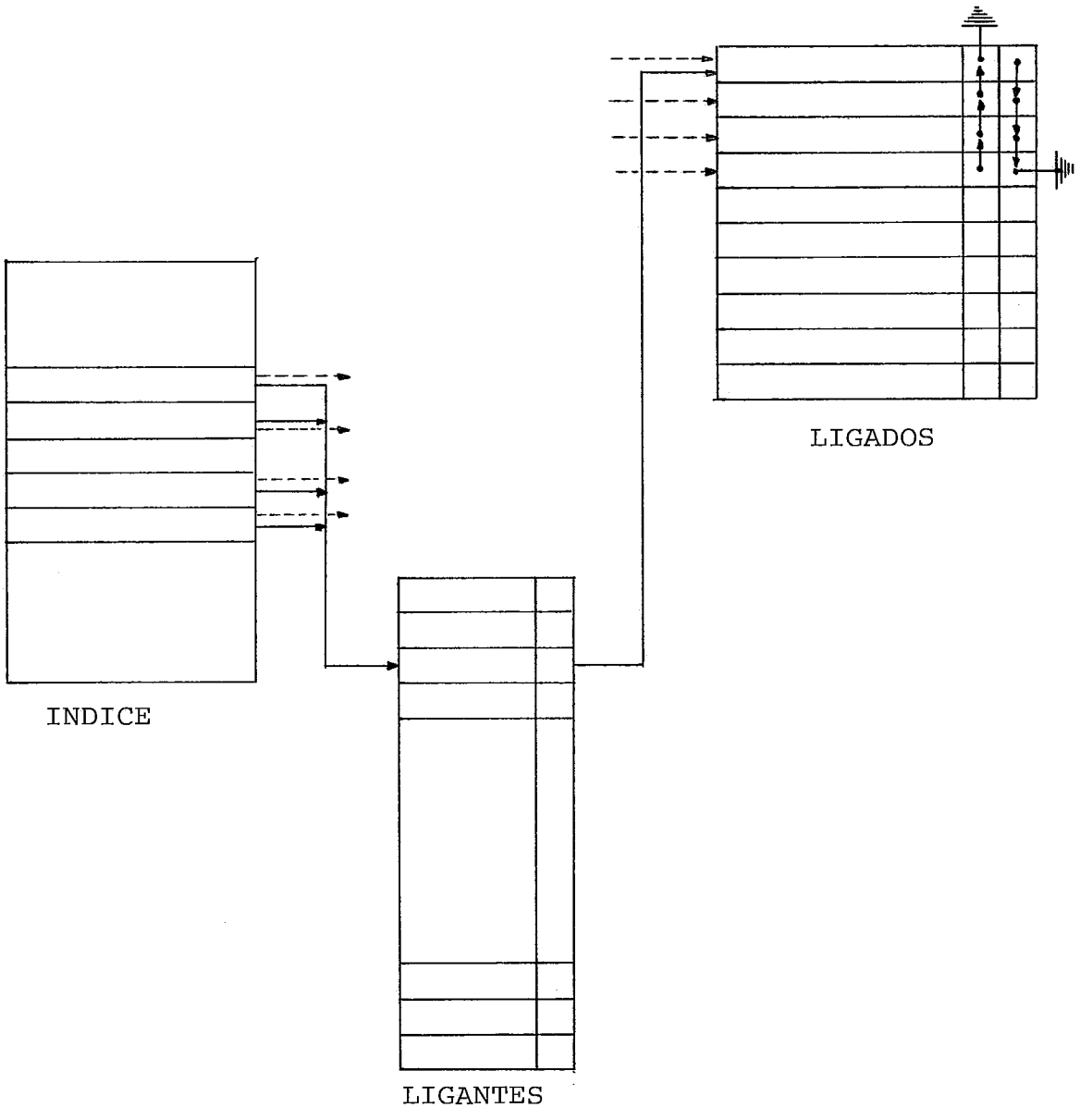


Figura 4.8

2. *Desvantagens*

- espaço para ponteiros no índice;
- com a combinação das chaves há um aumento no tempo de processamento para o índice.

4.4. Representação de Relações de Ordem

Uma relação de ordem foi definida no capítulo anterior como sendo o sequenciamento entre as tuplas de uma tabela relacional ou entre os ligantes de uma tabela ligacional.

O sequenciamento será dado em função da chave de ordenação definida no verbete de relações de ordem e do tipo de ordenação solicitada.

Assim faremos as seguintes sugestões:

4.4.1. Sugestão 1

A relação de ordem poderá não existir fisicamente, e só ser construída assim que, um comando qualquer da linguagem requisitar uma busca com referência a uma ordenação, e ser destruída no momento que o programa em execução não a precisar mais. Esta sugestão faz com que o sistema chame um programa de ordenação para classificar os registros que o usuário necessite, e apresenta as seguintes vantagens e desvantagens:

1. Vantagens

- não ocupa espaço permanente e consequentemente a não necessidade de controlar o arquivo de dados;
- útil para pesquisas sequenciais e ocasionais.

2. Desvantagens

- tempo de ordenação;
- inclusões e exclusões provocam rearranjos se for necessário manter temporariamente a ordenação.

4.4.2. Sugestão 2

Os registros na medida que vão entrando para o Sistema são armazenados contiguamente e já ordenados, novas inclusões provocarão deslocamentos. Esta sugestão apresenta as seguintes vantagens e desvantagens:

1. Vantagens

- registros permanentemente ordenados.

2. Desvantagens

- inclusões provocam deslocamentos dos registros;
- em caso de exclusões físicas dos registros também haverá deslocamentos dos mesmos.

4.4.3. Sugestão 3

Em alguns casos a relação de ordem é percorrida nos dois sentidos, ou seja no sentido crescente das chaves e no decrescente.

Uma das estruturas mais comuns para este tipo de necessidade seria uma lista duplamente encadeada onde inclusões e exclusões são realizadas com grande rapidez.

Para acelerar a busca que é necessária em cada inclusão, esta lista pode ser implementada num índice sequencial em árvore.

Cada registro da lista poderá conter todas as informações, assim o próprio conjunto de dados contém o índice, ou ponteiros para as informações.

Esta sugestão apresenta as mesmas vantagens e desvantagens discutidas para a implementação de tabela relacional. Possui também a vantagem de percorrer os dados na ordem inversa, o que muitas vezes será necessário.

4.4.4. Observações sobre Relações de Ordem

Tabelas relacionais e Ligacionais poderão ter várias relações de ordem sobre respectivamente suas tuplas e suas Ligações.

4.4.4.1. Sugestão 1

Aplicar a sugestão 3 de implementação para relações de ordem.

Tabelas de Ligados também podem ter uma relação de ordem. Neste caso as ordenações ocorrerão em cada tabela de Ligado, independentemente uma das outras.

4.4.4.2. Sugestão 2

Nas sugestões apresentadas para a implementação de tabela Ligacional sugerimos que os Ligados já sejam incluídos de forma ordenada. Para facilitar a implementação sugere-se que seja feita em listas duplamente encadeadas.

4.5. Representação dos Verbetes

Naturalmente a representação física dos verbetes apresentados na seção 3.6, poderá utilizar as sugestões das seções 4.2 e 4.3.

Comparativamente os arquivos das tabelas Relacionais ou Ligacionais, e as estruturas suportes para a definição completa da base de dados, apresentam características de necessidades de espaço bastante diferentes e variam a extremos muito grandes.

A descrição da base de dados que é constituída por seus verbetes, por ser muito consultada durante a interpretação e execução, pois constantemente tem que se verificar a coerência das construções bem como a sua proteção, tem características especiais com relação aos dados da base de dados, uma vez que o fato fortemente predominante é o tempo e não o espaço ocupado.

Assim estruturas simples como área sequencial, listas encadeadas; e uma ou mais estruturas de hash são sugeridas para uma primeira implementação.

4.5.1. Sugestão 1

Durante a inclusão, alteração e exclusão de informações na base de dados, há necessidade de se verificar a coerência dos dados entre o que foi definido e a realidade do momento.

As necessidades de representação diferenciadas são importantes e caracterizam basicamente os seguintes verbetes:

1. Verbetes de Usuário
2. Verbetes de Acesso
3. Verbetes de Coerência para Relação de Ordem
4. Verbetes de Divisão
5. Verbetes de Texto Fonte
6. Verbetes de Utilização de Recursos
7. Verbetes de Coerência para as demais construções da Base de Dados

Basicamente teremos 7 estruturas de entrada, compondo 7 índices aleatórios que indexarão os ligantes das tabelas que representam o verbete.

Assim podemos ter a entrada para os ligantes feitas por busca sequencial ou por hash.

Cada ligante aponta para uma lista de ligados que normalmente será percorrida sequencialmente e armazenada contiguamente.

Na maioria dos casos a tabela de ligados terá elementos que são cadeias de caracteres e serão interpretadas em tempo de execução. Assim sugere-se que se use a mesma estrutura de área para cadeias apresentada na seção 4.

4.6. Representação dos Dados em Arquivos

Na implementação da interface o gerenciamento dos arquivos deve ser realizada através do software básico do sistema portador ou criando áreas próprias de controle fazendo o seu próprio gerenciamento.

Todas as informações da base de dados são guardadas em arquivos. Associado à um arquivo podemos ter um registro na Diretoria (ou Diretório) do sistema portador. Este registro contém informações descrevendo a posição e o formato dos registros compreendendo o arquivo.

Tipos diferentes de organização de arquivos mantém informações diferentes no conteúdo da diretoria de seus arquivos. A maior parte das informações que são mantidas na diretoria são associadas com alocação de memória no *sistema portador* e com os conceitos da Interface. Por esta razão estruturas de registro da diretoria não são discutidos em detalhe neste trabalho como parte da análise de representação dos dados em arquivos.

As informações típicas mantidas em uma diretoria são:

- nome do arquivo
- o criador do arquivo
- o ponto inicial dos dados
- o ponto final dos dados
- a quantidade de espaço alocada
- a quantidade de espaço usada
- formato dos dados
- etc.

É importante observar que uma coleção de registros na

diretoria para um número de arquivos pode por sua vez formar um arquivo. O proprietário do arquivo da diretoria ou é o sistema operacional do sistema portador ou o sistema de gerência de base de dados.

4.7. Gerenciamento dos Dados na Memória

Algumas decisões precisam ser tomadas com relação ao sistema de gerenciamento que será adotado. Alocação contígua de memória, alocação particionada, alocação particionada relocável, alocação paginada, paginação sob demanda, alocação segmentada ou alocação paginada segmentada? A descrição dos verbetes da base de dados será resumida na memória? armazenados nos discos?, combinados e carregados na memória por partes?

Além das construções normais armazenadas em disco podemos ter uma série de índices que podem ser criados através dos verbetes de utilização de recursos que tem o objetivo de informar o SBD das preferências e políticas de utilização desejadas pelo usuário.

Aqui também cabe a pergunta inicial, os índices ficarão na memória interna? ou externa? É importante a escolha porque definirão a política de criação e utilização dos índices. Se for o caso de memória externa usaremos Árvore-B e suas variações, ou tabelas de espalhamento com bucket. Se for em memória interna podemos ter árvores binárias para o índice sequencial, ou tabela de espalhamento com endereçamento aberto, com número limitado de colisões (permitindo retirada rápida e ocupação satisfatória), (SOUZA[5]).

Naturalmente no nosso caso cada diretoria do arquivo tem ponteiros para: a sua tabela relacional (ou ligacional) e relação(ões) de ordem.

(KNUTH[1]), apresenta diversos algorítmos relativamente simples para alocação dinâmica de memória que podem ser utilizados, dentre os quais recomendamos o "buddy system" por ser

mais eficiente.

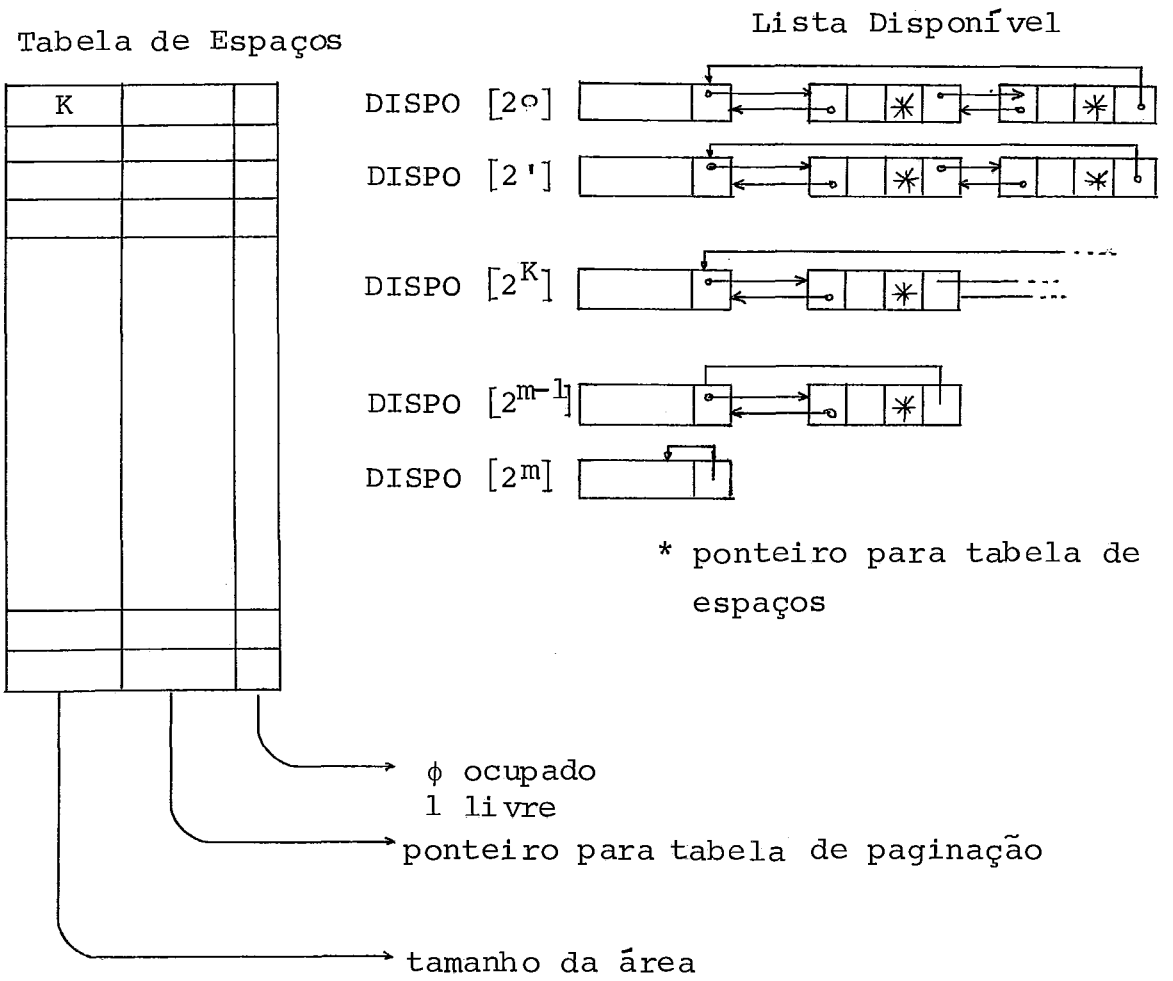


Figura 4.9

mação. Seu principal objetivo é indicar se a informação desejada pelo usuário, está presente ou não na memória. Esta tabela tem um ponteiro para a tabela de paginação. Como já foi descrito anteriormente, esta tabela de hashing pode ser por endereçamento aberto, com número limitado de coli

4.7.1. Sugestão para Gerenciamento

Para o gerenciamento vamos usar um algorítmo de alocação dinâmica de memória e armazenamento em memória auxiliar e um algorítmo de paginação.

São sugeridas as seguintes estruturas fundamentais:

1. Uma área na memória, usada para mapear os espaços livres. É usada pelo algorítmo de alocação dinâmica e tem ponteiros para a tabela de paginação. O mapa de espaços é constituído de duas subáreas; uma que tem uma lista dos espaços disponíveis e outra com as áreas que estão ocupadas ou desocupadas veja na figura 4.9.

2. Uma tabela de paginação onde estão anotadas as áreas da memória que podem ser desalocados quando houver necessidade de novas informações na memória. É usada pelo algorítmo de paginação. Esta tabela tem: um ponteiro para o mapa de espaços (este ponteiro de volta é usado para desalocar a área de memória); um ponteiro para uma tabela de hashing (é um ponteiro de retorno no caso da informação ser desalocada); um ponteiro para área de memória onde está a informação; uma marca que identifica se a informação sofreu ou não alteração; um ponteiro para a informação no seu local de origem. Sobre esta tabela de paginação deve ser aplicada a filosofia do algorítmo de substituição de páginas.

3. Uma tabela de hashing que contém a chave da infor

sões para o tratamento de colisões.

4. Uma área na memória, contendo espaço para armazenar as informações. Esta área estará sob a administração do algorítmo de alocação dinâmica de memória.

A figura 4.10 apresenta graficamente esta idéia.

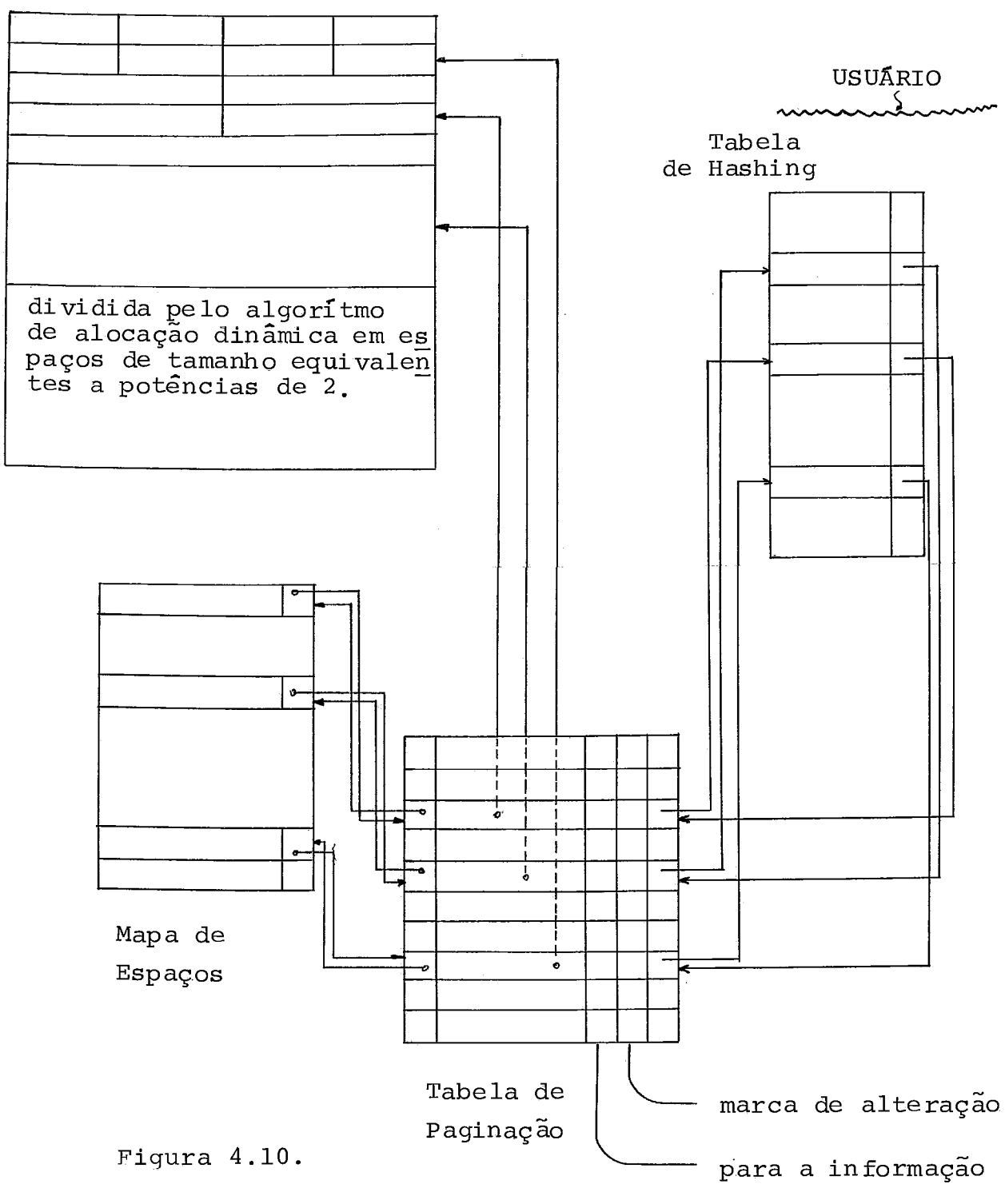


Figura 4.10.

4.8. Esquema Geral

A seguir apresentamos a figura 4.11 que mostra o diagrama geral das Estruturas de Dados.

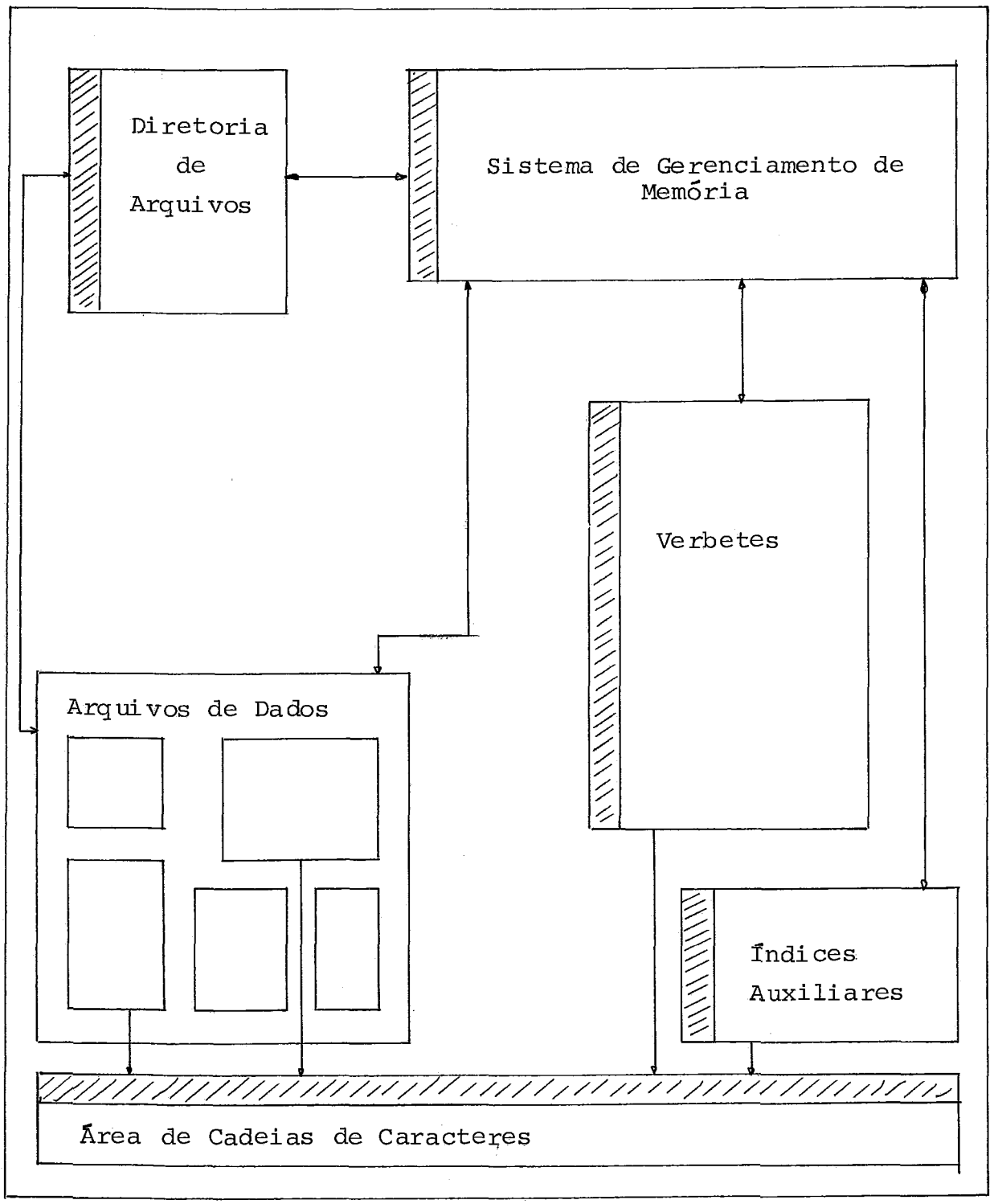


Figura 4.11

As áreas hachuradas da figura 4.11 são índices que devem ser usados para melhorar a performance na busca de informações no SBD.

Assim temos:

1. *Índice para a Diretoria de Arquivos*

   É um índice para os arquivos do usuário. Certamente deve ser uma estrutura de acesso rápido como hash. Localizado o arquivo este por sua vez indicará os seus componentes, ou seja, relações de ordem e tabela relacional ou ligacional (alguns deles poderão ser um novo índice, como por exemplo a(s) relação(ões) de ordem, ou a TR, ou a TL associadas).

2. *Índice para o Sistema de Gerenciamento de Memória*

   Veja exemplificação na seção 4.7.1.

3. *Índice para os Verbetes*

   Poderá ser um único Índice ou um conjunto deles para cada tipo de verbete (foi sugerido neste trabalho). O acesso aos Índices é acelerado através da interpretação de instruções bem como através dum algorítmo de hashing para encontrar facilmente os nomes que o usuário atribuiu para os seus verbetes e que caracterizarão a sua Base de Dados.

4. *Índices Auxiliares*

   Podemos ter uma área para índices auxiliares que podem ter sido caracterizados através dos verbetes de Utilização de Recursos. O objetivo destes verbetes é otimizar o Armazenamento dos dados do Usuá

rio através de algorítmos rápidos, dependendo das características destes dados, bem como criar índices para acelerar buscas baseado nas instruções que o Usuário forneceu no verbete. Neste índice são armazenados também as relações de ordem e arquivos de dados que estejam totalmente contidos em índices.

A área de cadeias de caracteres deve ser controlada por um algorítmo que gerencie o seu espaço alocando novas áreas ou desalocando e rearranjando o espaço.

5. CONCLUSÕES

Diversos métodos foram apresentados e formulados para armazenar e recuperar informações tanto na memória como em equipamento de acesso direto.

Vários autores fazem comparações entre os diversos métodos apresentados. Os principais citados aqui são (KNUTH[1,2]), (WIRTH[3]), (AHO[4]), (SOUZA[5]) e (HILL[26]). Normalmente são apresentados através de gráficos comparativos entre fator de carregamento, número de acesso, e tamanho do bucket, para as diversas estruturas de dados e algorítmos de busca.

Os algorítmos e os conceitos apresentados sobre hashing, listas encadeadas, arquivos invertidos, buscas em árvores binárias e árvores-B foram sugeridos para implementação da interface LOBAN.

Antes de qualquer implementação é necessário uma análise na estrutura lógica do sistema, e depois a sua adaptação às estruturas de dados existentes ou sugestões de novas estruturas.

Naturalmente o sistema portador será fundamental na escolha de determinadas estruturas, bem como o tipo de dado e as operações sobre ele.

Além destas considerações, algumas restrições e simplificações devem ser feitas na interface e linguagem LOBAN, para a sua implementação devida a sua grande complexidade. Uma vez terminada esta fase, a determinação de primitivas para o interpretador e para os algorítmos de manipulação do Sistema de Gerenciamento de Base de Dados seria o próximo passo.

As estruturas sugeridas neste trabalho poderão também

sofrer restrições na implementação, dependendo do porte e suporte do sistema que for escolhido para conter a linguagem.

Procuramos analisar neste trabalho todas as estruturas da interface, à menos de algumas restrições que são feitas no texto, deixamos de apresentar algumas sugestões em alguns casos onde a complexidade do problema por si só justificaria um outro trabalho.

Esperamos que este trabalho seja de ajuda para as pessoas que venham a projetar a implementação da linguagem.

BIBLIOGRAFIA

01. KNUTH, D. - *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Reading-Mass., Addison-Wesley, 1973, 634 p.

02. KNUTH, D. - *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Reading-Mass., Addison-Wesley, 1973, 723 p.

03. WIRTH, N. - *Algorithms + Data Structures = Programs*, Englewoods Cliffs - NJ, Prentice-Hall, 1976, 366 p.

04. AHO, A., HOPCKOFT, J. e ULLMAN, J. - *The Design and Analysis of Computer Algorithms*, Reading-Mass., Addison-Wesley, 1974, 470 p.

05. SOUZA, J.M. - *Algorítmos de Hashing para Problemas Específicos*. Tese de Mestrado - COPPE/UFRJ - 1978.

06. PINTO, P.R.B. - *Aspectos Conceituais sobre Concorrência em Banco de Dados*. Tese de Mestrado - COPPE/UFRJ - 1979.

07. RICHTER, G. - *Documentação de Software por Meio de Funciogramas*. (GMD-CNPq) - MINIBAN 38 - Relatório Técnico - Março de 1979, Rio de Janeiro.

08. RICHTER, G.; PEREIRA FILHO, J.C.; CASTILHO, J.M.V. - *Projeto MINIBAN - Relatório Final da Segunda Etapa - Parte A*. MINIBAN 29 - Relatório Técnico - Abril 78. Rio de Janeiro.

09. RICHTER, G.; CASTILHO, J.M.V. - *Uma Interface para Sistemas de Informação: LOBAN - Linguagem de Operação de Banco de Dados*. Anais do 11º CNPD - SUCESU - 1978, Rio de Janeiro.

10. DURCHHOLZ, R.; RICHTER, G. - *Information Management Concepts (IMC) for use with DBMS Interfaces*. In: Modelling in data base management systems (Proceedings IFIP Working Conf.), Nijssen, G.M(ed), Amsterdam, North-Holland, 1976, pg. 49-72.

11. COMER, D. - The Ubiquitous B. Tree - *Computing Surveys* - 11, 2, (79), 121-137.

12. BAYER, R.; METZGER, J. - On encipherment of search trees and random access files - *ACM Trans Database Syst*. 1,1, (76), 37-52.

13. BAYER, R.; UNTERAUER, K. - Prefix B-Trees - *ACM Trans Database Syst*., 2, 1, (77), 11-26.

14. BAYER, R.; SCHOKOLNICK, M. - Concurrency of operations on B-Trees. *Acta Inf*. - 9, 1, (77), 1-21.

15. McCREIGHT, E. - Pagination of B*-Tress with variable-lengh records - *Comm. ACM*, 20, 9, (77), 670-674.

16. BAYER, R. e McCREIGHT, C. - Organization and Maintenance of Large Ordered Indexes, *Acta Informática*, 1, 3, (72), 173-189.

17. MAUER, W. e LEWIS, T. - Hash table methods - *Comput Surveys*. 7, 1, (75), 5-19.

18. BERLINER, H. - The B*-Tree search algorithm: a best-fist proof procedure - *Tech Rep. CMU-CA-78-112*, Computer Science Dept. Carnegie-Mellon Univ., Pittsburg, 1978.

19. SAMADI, B. - B-Trees in a System with multiple users - *Information Processing Letters*, 5, 4, (76), 107-112.

20. LUM, V.Y.; YUEN, P.S.T.; DODD, M. - Key-to-address transform techniques. A fundamental Performance Study on Large Existing Formatted Files - *Comm. ACM*, 14, 4, (71), 228-239.

21. Additional Results on ... *Comm ACM*, 15, 11, (72), 996-997.

22. WIEDERHOLD, G. - *Database Design*, Tokyo, Kogakusha, 77, 658 p.

23. DATE, C.J. - *An Introduction to Database Systems*, Second Edition, New York, Addison Wesley, 77, 536 p.

24. MARTIN, J. - *Computer Database Organization*, Englewood Cliffs, N.J., Prentice-Hall, 1975, 558 p.

25. DURCHHOLZ, R.; RICHTER, G. - *Concepts for Data Management Systems*. In: Data Base Management, Klimbie, J.W. - Koffemman, K.L(Eds), Amsterdam, North-Holland, 1974, pg. 97 a 121.

26. HILL JR., EDWARD - *A Comparative Study of Very Large Data Bases*, 1ª ed., Berlim, Springer-Verlag, 1978, 140 p., (Lectures Notes in Computer Science).

27. SOUZA, JANO MOREIRA - Notas de aula do curso de Busca em Arquivos da COPPE/UFRJ, 1978.

28. CATTO, ARTHUR JOÃO - Notas de aula do curso de Estruturas de Dados da UFSCar, 1975.

29. SANTOS, A.C. et alii - Projeto MINIBAN/COPPE - Especificação da Interface LOBAN - Relatório Técnico, Outubro 80, Rio de Janeiro.