ISSN - 0103-2585

Técnicas avançadas de programação Prolog para tratamento de árvores

MARIA CAROLINA M0NARD

MARIA DO CARMO NICOLETTI

N° 008

NOTAS DIDATICAS DO ICMSC

São Carlos

fev. / 1993

Técnicas Avançadas de Programação Prolog para Tratamento de Arvores

Maria Carolina Monard[1]
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

Fevereiro 1993 - Versão 2.0

trabalho realizado com o auxílio do CNPq/ILTC.

Conteúdo

1 Prefácio

A linguagem de programação lógica Prolog surgiu na década de 70 e ganhou popularidade nos últimos anos através, principalmente, de seu uso em aplicações de computação simbólica.

Existem vários livros que abordam os fundamentos teóricos de programação lógica; não é difícil, entretanto, encontrar livros específicos da linguagem Prolog, onde a linguagem é introduzida através de problemas e de suas soluções em Prolog.

O aumento significativo de publicações que tratam de Prolog justifica-se também pelo aumento do número de cursos que utilizam esta linguagem, tanto a nível de graduação, quanto a nível de pós-graduação.

Lembrando que uma linguagem pode ser classificada como procedural, funcional e/ou lógica, ao analisarmos o curriculum de nossos cursos de computação, verificamos que primeiro são ensinadas várias linguagens procedimentais, tais como Pascal, Cobol, Fortran, etc., e depois linguagens do tipo Apl, Lisp, Prolog, etc.

Temos observado, após vários anos de experiência com ensino em cursos de computação, que o estilo procedimental de programação, talvez por ser o primeiro ensinado, interfere, de certa forma, na correta aprendizagem da linguagem Prolog por parte dos estudantes.

Constatamos, por outro lado, que tal interferência pode ser minimizada se durante o aprendizado de Prolog, os estudantes forem submetidos à resolução de conjuntos de problemas semelhantes. Motivados por tal constatação, decidimos preparar uma série de notas em programação Prolog, cada uma delas tratando classes de problemas semelhantes, e desenvolvendo para cada um deles a resolução em Prolog.

Já foram publicadas pelo Instituto de Lógica Filosofia e Teoria da Ciência - ILTC - as notas dedicadas a processamento de listas [Monard 88] e árvores [Monard 89]. Devido à boa receptividade dessas notas, resolvemos reestruturá-las [Monard 93]. Este fascículo é uma reorganização da Nota *Programas Prolog para Processamento de Arvores*. Deve ser ressaltado que os problemas aqui apresentados se encontram espalhados nos vários livros citados nas referências. O que pretendemos nesta Nota é reunir e discutir alguns problemas típicos, apresentando para cada um deles uma solução.

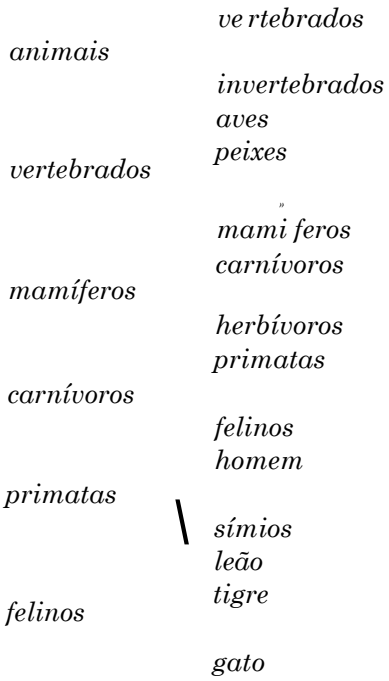
A fim de resolver os problemas aqui tratados, o leitor deverá ter um conhecimento básico da linguagem Prolog — ver livros textos listados nas Referências. E importante também que tente resolver os problemas e executar seus programas antes de consultar as soluções propostas nesta Nota.

A sintaxe do Prolog utilizada é a sintaxe de Edinburgh, especificamente utilizamos o Arity-Prolog [Arity 90]. Há implementações de Prolog que usam outras sintaxes. Como a sintaxe da linguagem é muito simples, torna-se fácil a expressão de um mesmo programa em diferentes sintaxes.

2 Introdução

Árvore *é* uma estrutura de dados extremamente importante na qual, da mesma forma que na estrutura de dados lista, existe uma relação entre os dados que a compõe. No caso da estrutura de árvore a relação que existe entre os dados, chamados nos, é uma relação de hierarquia, onde um conjunto de nós é hierarquicamente subordinado a outro.

Por exemplo, seja uma classificação (incompleta) de animais que obedece à seguinte hierarquia:



Estas relações hierárquicas podem ser representadas como mostra a Figura 1, pg.3, que exibe mais claramente a estrutura sob forma de uma árvore de classificação de animais.

2.1 Definição e Terminologia Básica

Formalmente, uma árvore é um conjunto finito de um ou mais nós onde:

1. existe um nó especial denominado *raiz da árvore;*
2. os demais nós formam n conjuntos disjuntos (n > 0), onde cada um destes conjuntos T,, (1 < *i* < n) é, por sua vez, uma árvore. As árvores T, são chamadas de *subárvores da raiz.*

A definição de árvore é uma definição recursiva. A raiz da árvore representada na Figura 1 é o nó rotulado *animais.* Esta árvore tem duas subárvores, cujas raízes são os

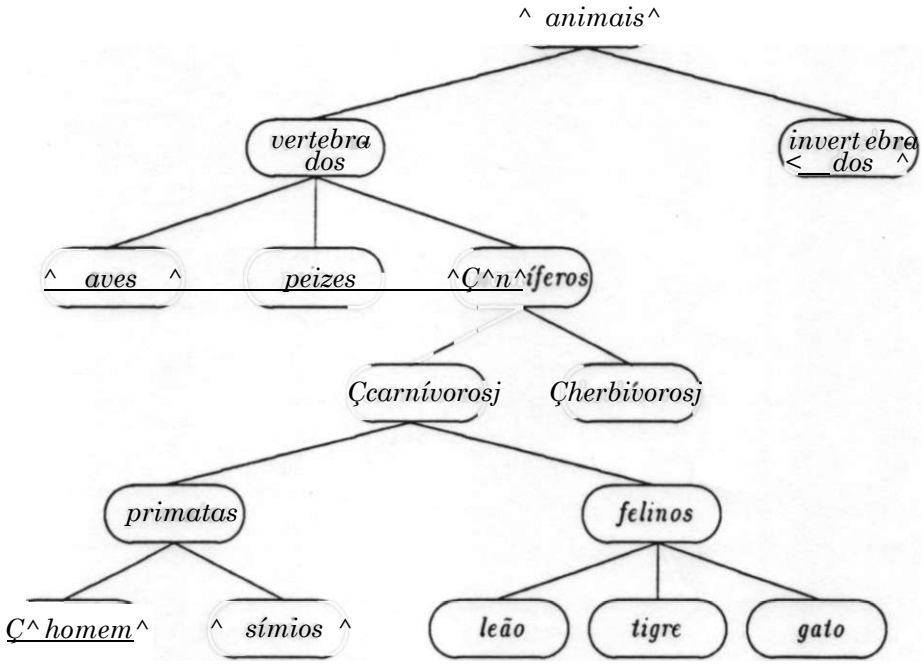


Figura 1: Representação de hierarquia

nós rotulados *vertebrados* e *invertebrados*. A condição de que os conjuntos T,(1 < *i* < *n)* devem ser disjuntos proíbe a interligação entre subárvores.

A seguir é definida a terminologia comumente adotada para a estrutura de árvore, utilizando para exemplificar, a Figura 1.

- *grau* de um nó é o número de subárvores deste nó.

  0 grau do nó rotulado *animais é* dois e o grau do nó rotulado *vertebrados* é três.

- *grau de uma árvore é* o máximo grau de seus nós.

  0 grau da árvore representada na Figura 1 é três.

- *folha* ou *nó terminal é* um nó de grau igual a zero.

  Os nós rotulados *aves, peixes*, Aomem, *símios, leão*, onça, *gato, herbívoros* e *invertebrados* são nós terminais.

- yiZ/ios de um nó são as raízes das subárvores deste nó. Este nó é denominado *pai* desses filhos.

  0 nó rotulado *vertebrados* possui três subárvores, cujas raízes são os nós rotulados *aves, peixes* e *mamíferos*. Estes três nós são, portanto, filhos do nó rotulado *vertebrados* o qual, por sua vez, é o pai dos três nós.

- *irmãos* são os nós filhos do mesmo pai.

  Os nós rotulados *homem* e *símios*— filhos do nó rotulado *primatas* — são irmãos.

- *antepassados* de um nó são todos os nós que pertencem ao caminho desde a raiz até o nó considerado.

  Os antepassados do nó *herbívoros* são os nós rotulados *mamíferos, vertebrados* e *animais.*

- na definição do *nível de um nó* assume-se que o nível da raiz é um. Se o nível de um nó é fc, os seus filhos estão no nível *k + 1.*

  No exemplo considerado, a raiz — *animais* —, está no nível um. Seus filhos — *vertebrados* e *invertebrados* — estão no nível dois. Os filhos do nó rotulado *vertebrados* estão no nível três e assim por diante.

- *altura* ou *profundidade* de uma árvore é o nível máximo entre os nós da árvore.

  No exemplo, a altura da árvore é seis.

- *comprimento de caminho de um nó y* é o número de ramos da árvore que devem ser percorridos a fim de ir da raiz ao nó *y* . A raiz tem comprimento de caminho zero, seus filhos comprimento de caminho um e assim por diante. Os nós que estão no nível *i* têm comprimento de caminho 7 - 1 .

- *comprimento de caminho* ou *comprimento de caminho interno* de uma árvore é a soma dos comprimentos de caminho de todos os nós da árvore.

  0 comprimento de caminho interno da árvore considerada é:

$$2 \times 1 + 3 \times 2 + 2 \times 3 + 2 \times 4 + 5 \times 5 = 4\ 7$$

2.2 Representação

Uma das formas de representar uma árvore em Prolog é como uma lista. Por exemplo, a árvore da Figura 2, pg.5 pode ser representada pela lista:

[a, [b, [f] , [g]] , [c, [h]] , [d] , [e, [i]]]

onde o nó raiz corresponde à cabeça da lista — a — e os elementos da cauda correspondem às subárvores da raiz, que são:

[b,[f],[g]]
[c,[h]]
[d]
[e,[i]]

e possuem estrutura semelhante. Os nós terminais estão representados por listas de um só elemento.

Figura 2: Representação em árvore da lista **[a,** [b, [f] , [g]] , [c, [h]] , [d] , [e, [i]]]

A maioria dos livros dedicados à estrutura de dados concentra-se em um tipo especial de árvore chamada *árvore binária*. A razão disto se deve ao fato de toda árvore poder ser representada como uma árvore binária. Os procedimentos para a obtenção da árvore binária associada a uma árvore podem ser vistos, por exemplo, em [Horowitz 84]. Este tipo de representação de uma árvore *é* conveniente pois uniformiza o seu tratamento.

3 Árvore Binária

Uma *árvore binária* pode ser uma árvore vazia ou consistir de:

- uma *raiz,*
- uma *subárvore esquerda* - Se - e
- uma *subárvore direita* - Sd.

onde as subárvores são, também, árvores binárias. A Figura 3 mostra a estrutura geral de uma árvore binária.

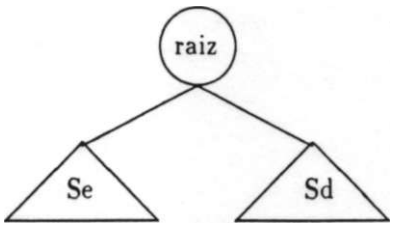


Figura 3: Arvore binária

E importante observar que a estrutura de árvore binária não é um caso especial de árvore, isto é, uma árvore binária não pode ser vista simplesmente como uma árvore de grau dois. A definição de árvore não impõe nenhuma restrição com relação à ordem das subárvores na árvore, enquanto que a de árvore binária impõe. As árvores representadas nas Figuras 4 e 5 são iguais se forem consideradas como árvores, enquanto que isto não acontece se forem consideradas como árvores binárias.

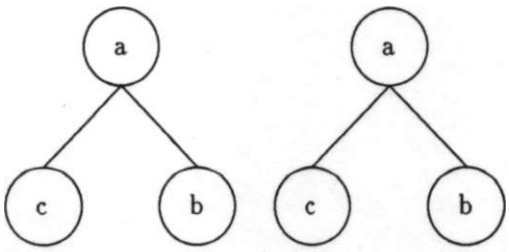


Figura 4: Duas árvores binárias iguais

No caso da Figura 5 a subárvore esquerda (direita) da primeira árvore não é igual à subárvore direita (esquerda) da segunda.

As definições introduzidas para árvores, tais como grau, nível, altura, etc. aplicam-se também às árvores binárias. Arvores binárias possuem várias propriedades e algumas delas são apresentadas a seguir. As verificações destas propriedades são simples e podem ser encontradas em [Knuth 78a].

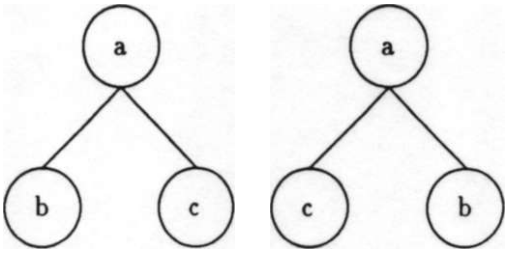


Figura 5: Duas árvores binárias diferentes

- o número máximo de nós no nível i de uma árvore binária é:

$$2^{i-1} \quad i > 1$$

- o número máximo de nós de uma árvore binária de altura h é:

$$2^h - 1 \quad h > 1$$

- para qualquer árvore binária não vazia, se no é o número de nós terminais e o número de nós de grau 2, então:

$$= n_2 + 1$$

3.1 Representação

São várias as possíveis representações de uma árvore binária. Em Prolog, uma representação conveniente é feita através do uso de uma estrutura. Uma estrutura é expressa por um funtor e seus argumentos. Em geral, estruturas são representadas na notação prefixada, isto é, o funtor em primeiro lugar seguido de seus argumentos. Na representação de árvores binárias, será usada a estrutura:

arv(Se,Raiz,Sd)

onde arv é o funtor e Se, Raiz e Sd são seus argumentos — Figura 3, pg.6. Nesta estrutura:

- Se - representa a subárvore esquerda da árvore;
- Raiz - representa a raiz da árvore;
- Sd - representa a subárvore direita da árvore.

isto é, Se e Sd são também estruturas do tipo arv. Como uma árvore binária pode ser uma árvore vazia, será usado um átomo especial nil para indicar este fato. Nesta notação, a árvore binária que consiste só da raiz, rotulada a, será representada por:

arv(nil,a,nil)

As Figuras 6, 7, 8 e 9 mostram uma sequência de inserções de nós em uma árvore binária, inicialmente vazia.

arv(nil,a,nil)

Figura 6: Arvore após a inserção do nó a

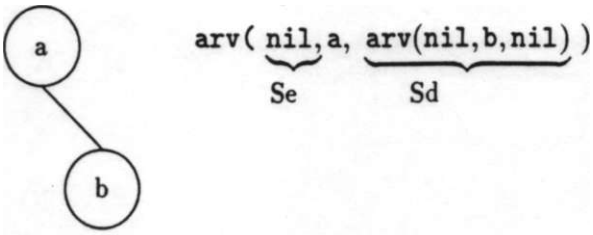


Figura 7: Árvore após a inserção do nó b

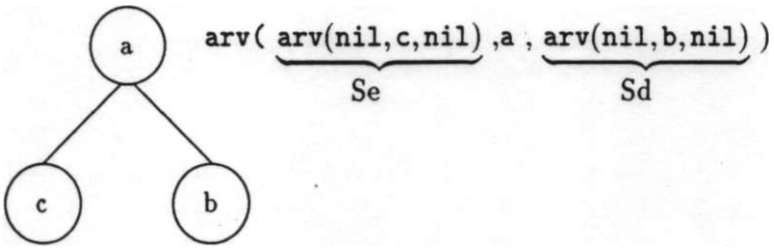


Figura 8: Árvore após a inserção do nó c

3.2 Programas

São apresentados a seguir vários programas Prolog que manipulam ávores binárias. A fim de documentar as formas de interrogação dos programas, será usada a notação definida em [Monard 93], isto é,

modo(<arg-l>,<arg-2>,...,<arg-n>)

onde:

<arg-i> = + se <arg-i> deve estar instanciado
<arg-i> = - se <arg-i> deve ser uma variável livre
<arg-i> = ? se <arg-i> pode ser qualquer um dos casos acima

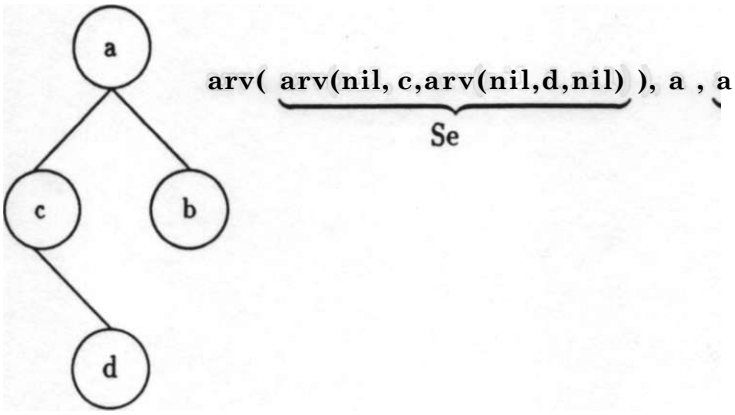


Figura 9: Árvore após a inserção do nó d

3.2.1 Exibir uma Árvore

Uma árvore binária T pode ser exibida simplesmente através do uso do predicado Prolog pré-definido `vrite(T)`. Este predicado não exibe graficamente a estrutura da árvore.

Conseguir visualizar a estrutura de uma árvore a partir do termo Prolog que a representa exige, às vezes, muito esforço. Por esta razão, serão apresentados a seguir dois programas para exibir graficamente uma árvore binária. 0 primeiro programa `mostrah/1`, mostra a árvore disposta horizontalmente; o segundo programa `mostrav/1`, mostra a árvore disposta verticalmente.

0 modo de chamada para os dois programas é o mesmo e é dado por:

```
modo(+)

mostrah(<arg-l>),  mostrav(<arg-l>)
<arg-l>  :  arvore
```

Para mostrar uma árvore binária horizontalmente:

1. mostrar a subárvore direita deslocada de uma distância D à direita,
2. mostrar a raiz da árvore,
3. mostrar a subárvore esquerda deslocada de uma distância D à direita.

A distância D de deslocamento é um parâmetro adicional na exibição da árvore; é o segundo argumento de `mostra2(Arvore,D)`.

Para o caso de D = 0 o programa `mostrah/1` é:

```
mostrah(Arvore)mostra2(Arvore,0).

mostra2(nil,_).

mostra2(arv(Se,Raiz,Sd),Indent)
                    Ind2 is Indent • 4,
                    mostra2(Sd,Ind2),
                    tab(Indent),write(Raiz),nl,
                    mostra2(Se,Ind2).
```

A fim de exemplificar o uso de **mostrah/**1 e **mostrav**/1, será utilizado o programa **instancie/**1 que simplesmente instancia seu argumento com uma estrutura de árvore binária previamente definida. Este programa pode ser definido por:

```
instancie(X)pegue.arvore(X).
```

onde pegue.arvore(X) é uma asserção da forma

```
pegue.arvore(arv(Se,Raiz,Sd)).
```

Segue um exemplo de execução de **mostrah/1** usando como argumento a árvore da Figura 9, pg.9.

Exemplo 3.2.1.1 modo(+)

```
?- instancie(Y),mostrah(Y).


        d
    c

X = arv(arv(nil,c,arv(nil.d.nil)> ,a.arv(nil,b,nil))
yes
```

O programa para a exibição de uma árvore na vertical — **mostrav**/1 exibe os nós da árvore nível por nível, como segue:

1. se todos os níveis foram mostrados, então parar, caso contrário,
2. mostrar por nível, a partir do primeiro, todos os nós da árvore.

0 programa caminhe/4 é responsável por mostrar os nós da árvore binária que estão no mesmo nível. Os passos são:

1. se a árvore for a árvore vazia, nada é mostrado, caso contrário,

2. mostrar todos os nós da subárvore esquerda que estão no nível considerado e depois os nós da subárvore direita que estão naquele mesmo nível.

```
mostrav(Arvore)mostranivel(Arvore,0,mais).

mostranivel(Arvore,Nivel,todos)!.

mostranivel(Arvore,Nivel,mais):-
                caminhe(Arvore,Nivel,0 »Continue),nl,nl,
                Proxnivel is Nivel • 1,
                mostranivel(Arvore,Proxnivel,Continue).

caminhe(nil,.

caminhe(arv(Se,Raiz,Sd),Nivel,Xfundo,Continue):-
                Prox.abaixo is Xfundo • 1,
                caminhe (Se, Nivel, Prox.abaixo, Continue),
                (Nivel = Xfundo, !,
                write(Raiz)»Continue = mais; writeC O),
                caminhe (Sd, Nivel, Prox.abaixo, Continue).
```

Segue um exemplo de execução de mostrav/1 usando como argumento a árvore da Figura 9, pg.9.

Exemplo 3.2.1.2 modo(+)

```
?- instancie(Y), mostrav(Y).

    a

c     b

  d

X = arv(arv(nil,c,arv(nil,d,nil)),a,arv(nil,b,nil))
yes
```

3.2.2 Determinar se um Elemento Pertence a uma Árvore Binária

Dada uma árvore binária A, representada pela estrutura arv(Sd,Raiz,Se), um elemento Elem pertence a A se:

1. Elem for a raiz de A, ou

2. **Elem pertencer à subárvore esquerda de A, ou**

3. **Elem pertencer à subárvore direita de A.**

modo(?,+)

pertence(<arg-l>,<arg-2>)
<arg-l> : elemento
<arg-2> : arvore

```
pertence(Elem,arv(.,Elem,_))

pertence(Elem,arv(Se,_,_))    pertence(Elem,Se).

pertence(Elem,arv(.,_,Sd)): - pertence(Elem,Sd).
```

Deve ser observado que se a árvore for uma árvore vazia, o programa sempre falha, isto é:

```
?- pertence(Elem,nil).
no
```

Exemplo 3.2.2.1 modo(-,+)

```
?- instancie(A)»pertence(Elem, A).

A = arv(arv(nil,b,nil),a,arv(nil,c,nil))
Elem = a ->;

A « arv(arv(nil,b,nil),a,arv(nil,c,nil))
Elem = b ->;

A = arv(arv(nil,b,nil),a,arv(nil,c,nil))
Elem = c ->;
no
```

A cláusula acima dá, à medida que forem sendo solicitados através do ; todos os nós que pertencem à árvore.

0 problema com este programa é a sua ineficiência. Se o elemento procurado for a raiz da árvore, o programa é bem sucedido na primeira tentativa. Entretanto, se o elemento procurado não pertencer à árvore, o programa vai percorrer todos os nós da árvore antes de falhar, o que evidencia uma busca ineficiente. A busca pode ser melhorada se existir uma relação de ordem entre os elementos da árvore, de tal forma que, dado um elemento, pode-se determinar se ele pertence à subárvore esquerda ou direita da

árvore binária. Esta estrutura de árvore binária, com uma relação de ordem entre seus elementos, chamada *árvore binária ordenada* ou *dicionário binário*, será vista em detalhes na seção 4.

Ainda que seja possível distinguir a subárvore esquerda da direita em uma árvore binária, para muitas aplicações esta distinção não é relevante, como pode ser visto a seguir.

3.2.3 Determinar se Duas Árvores Binárias são Isomorfas

Isomorfismo entre árvores é uma relação de equivalência com a seguinte definição recursiva:

Duas árvores binárias Al e A2 são isomorfas se:

1. Al e e A2 são árvore vazias, ou
2. raiz de Al igual raiz de A2 e
   (a) subárvore direita de Al e subárvore direita de A2 são isomorfas, e subárvore esquerda de Al e subárvore esquerda de A2 são isomorfas, ou
   (b) subárvore direita de Al e subárvore esquerda de A2 são isomorfas, e subárvore esquerda de Al e subárvore direita de A2 são isomorfas.

definição que pode ser traduzida diretamente para as seguintes cláusulas Prolog:

```
modo(+,+)

iso(<arg-l>,<arg-2>)
<arg-l> : primeira arvore
<arg-2> : segunda arvore
```

```
iso(nil,nil).

iso(arv(Se,Raiz,Sd),arv(Sei,Raiz,Sdl))    iso(Se,Sel),
                                           iso(Sd,Sdl),!.

iso(arv(Se,Raiz,Sd),arv(Sei,Raiz,Sdl))    iso(Se,Sdl),
                                           iso(Sd,Sel).
```

Exemplo 3.2.3.1 modo(+,+)

```
?- instancie(A),instancie(B),A \=B,iso(A,B),mostrav(A),mostrav(B).

  b

a  c


  b

c  a

A = arv(arv(nil,a,nil),b,arv(nil,c,nil))
B = arv(arv(nil,c,nil),b,arv(nil,a,nil))
yes
```

3.2.4 Substituir Todas as Ocorrências de um Elemento em uma Árvore Binária

Uma outra operação frequente em árvores binárias *é* a substituição de seus elementos. Dada uma árvore binária, é comum ser necessário substituir todas as ocorrências de um determinado elemento na árvore por outro elemento qualquer, obtendo-se uma nova árvore binária.

As regras para a substituição de todas as ocorrências do elemento El da árvore binária Al por um elemento E2, para obter a árvore binária A2, são:

1. se Al for vazia então A2 é vazia.
2. se El for a raiz da árvore Al então substituir a raiz por E2 e substituir El por E2 na subárvore esquerda de Al e na subárvore direita de Al.
3. se El não for a raiz da árvore Al então substituir El por E2 na subárvore esquerda de Al e na subárvore direita de Al.

```
modo (•,?,•.?),(?,

substitui(<arg-1>,<arg-2>,<arg-3>,<arg-4>)
<arg-l> : elemento a ser substituido
<arg-2> : novo elemento
<arg-3> : arvore original
<arg-4> : nova arvore
```

```
substitui(X,Y,nil,nil).

substitui(X,Y,arv(Se,X,Sd),arv(Sel,Y,Sdl))
                                    substitui(X,Y,Se,Sel),
                                    subst itui(X,Y,Sd,Sd1).

substitui(X,Y,arv(Se,Z,Sd),arv(Se1,Z,Sd1)):-
                                    X \== Z,
                                    substitui(X,Y,Se,Sel),
                                    substitui(X,Y>Sd>Sdl).
```

Exemplo 3.2.4.1 modo(+,+,+

```
?- instancie(X)tsubstitui(a,x,X,Y),mostrav(X),mostrav(Y).

        n

    m       f

b       a m a

  a


        n

    m       f

b    x  m  x

  x

X = arv(arv(arv(nil,b,arv(nil,a,nil)),m,arv(nil,a,nil)),n,
    arv(arv(nil,m,nil),f,arv(nil,a,nil)))
Y = arv(arv(arv(nil,b,arv(nil,x,nil)) ,m,arv(nil,x,nil)) ,n,
    arv(arv(nil,m,nil) ,f ,arv(nil,x,nil)))  ->;
no
```

3.2.5 Encontrar a Altura de uma Árvore Binária

A altura de uma árvore binária é:

1. um, se a árvore possuir apenas a raiz, caso contrário,
2. é o máximo entre as alturas de suas subárvores (direita e esquerda).

```
modo(+,?)

altura(<arg-l>,<arg-2>)
<arg-l>  :  arvore
<arg-2>  :  altura da arvore
```

```
altura(nil,0).

altura(arv(Se,Raiz,Sd),X)      altura(Se,Y),
                               altura(Sd,Z),
                               max(Y,Z,Max),
                               X is Max
```

onde o programa **max/3,** que encontra o maior entre dois elementos numéricos é definido por:

```
max(X,Y,X):- X >= Y,!.

max(X,Y,Y).
```

Exemplo 3.2.5.1 modo(+,-)

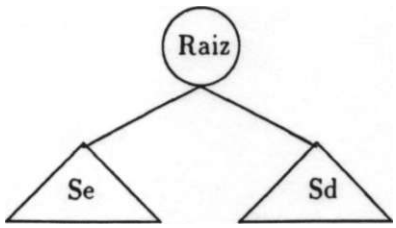
```
?-  instancie(X),mostrav(X),altura(X,A).

                  a

      m                 o

b          a          s    m

  a  v        1            h  l

           o  c  m            k  a


X = arv(arv(arv(nil,b,arv(nil,a,nil)) ,m,arv(arv(nil,v,arv(nil,o,nil)) ,
    a,arv(arv(nil,c,nil) ,l,arv(nil,m,nil)))) ,a,arv(arv(nil,s,nil) ,o,
    arv(arv(nil,h,nil)  ,m,arv(arv(nil,k,nil),l,arv(nil,a,nil)))))
A = 5
yes
```

3.2.6 Percorrer Árvores Binárias

Existem muitas operações que podem ser executadas na estrutura de árvores binárias. É comum executar uma determinada operação *P* sobre cada nó da árvore apenas uma vez.

Se a operação *P* for considerada como um processo seqüencial simples, então os nós da árvore são visitados em uma ordem específica e podem ser considerados como se estivessem linearmente dispostos. De fato, a descrição de alguns algoritmos torna-se consideravelmente mais fácil se for possível fazer referência ao próximo elemento da árvore, considerando uma ordem pré-estabelecida.

Existem três maneiras principais de visitar os elementos de uma árvore binária, que emergem naturalmente de sua estrutura. Na árvore binária mostrada a seguir:



onde Raiz denota a raiz, Se e Sd as subárvores esquerda e direita respectivamente, as três maneiras principais de percorrê-la são:

1. Pré-ordem: Raiz, Se, Sd
2. In-ordem: Se, Raiz, Sd
3. Pós-ordem: Se, Sd, Raiz

Isto é:

1. para visitar uma árvore binária em *pré-ordem* visitar primeiro a raiz, a seguir a subárvore esquerda e finalmente a subárvore direita.
2. para visitar uma árvore binária em *in-ordem* visitar primeiro a subárvore esquerda, a seguir a raiz, e finalmente a subárvore direita.
3. para visitar uma árvore binária em *pós-ordem* visitar primeiro a subárvore esquerda, a seguir a subárvore direita e finalmente a raiz.

A árvore da Figura 10 - pg.18 - pode ser considerada como representação da expressão aritmética

(a • b/c) * (d - e*f)

(E importante lembrai' que para utilizar os operadores de multiplicação, divisão, soma e subtração em cláusulas Prolog como constantes e não como funtores, é necessário que eles estejam entre apóstrofos '.)

Figura 10: Representação da expressão aritmética (a + *b/c)* * *(d* - e * /)

A fim de exemplificar as interrogações dos programas, o programa instancie(X) será usado para instanciar a variável X com a estrutura de árvore binária mostrada na Figura 10, e o programa mostrav/1 para imprimi-la.

Exemplo 3.2.6.1 modo(+)

```
?- instancie(X),mostrav(X).
             *


    +

 a      /      d      *

       b   c         e   f


 X = arv(arv(arv(nil,a,nil),+,arv(arv(nil,b,nil),/,arv(nil,c,nil)))
     arv(arv(nil,d,nil),-,arv(arv(nil,e,nil),*,arv(nil,f,nil))))
 yes
```

A seguir são apresentados os programas Prolog que realizam o percurso em uma árvore binária em pré-ordem, in-ordem e pós-ordem. Para exemplificar, será usada como operação *P* o predicado Prolog pré-definido write(Termo).

O modo de chamada dos três programas é o mesmo:

modo(+)

pre_ordem(<arg-l>)
in.ordem(<arg-1>)
pos.ordem(<arg-l>)
<arg-l> : arvore

3.2.7 Pré-ordem

Para percorrer uma árvore binária em pré-ordem:

1. visitar a raiz,
   percorrer a subárvore esquerda em pré-ordem,
   percorrer a subárvore direita em pré-ordem.
2. árvore vazia náo pode ser mais percorrida.

```
pre_ordem(arv(Se,Raiz,Sd))    write(Raiz),
                              pre.ordem(Se),
                              pre.ordem(Sd).

pre.ordem(nil).
```

Exemplo 3.2.7.1 modo(+)

```
?- instancie(X),mostrav(X),pre_ordem(X).
          *


  +

a    /    d    *

   b  c      e  f


*+a/bc-d*ef
X = arv(arv(arv(nil,a,nil),+,arv(arv(nil,b,nil),/,arv(nil,c,nil))),
    *,arv(arv(nil,d,nil),-,arv(arv(nil,e,nil)    ,arv(nil,f,nil))))
yes
```

3.2.8 In-ordem

Para percorrer uma árvore binária em in-ordem:

1. percorrer a subárvore esquerda em in-ordem,
   visitar a raiz,
   percorrer a subárvore direita em in-ordem.

2. árvore vazia não pode ser mais percorrida.

```
in.ordem(arv(Se,Raiz,Sd)):- in.ordem(Se),
                            vrite(Raiz),
                            in.ordem(Sd).
in.ordem(nil).
```

Exemplo 3.2.8.1 modo(+)

```
?- instancie(X),mostrav(X),in_ordem(X).
        *


   .

a    /    d    *

   b  c      e  f


a+b/c*d-e*f
X = arv(arv(arv(nil,a,nil),+,arv(arv(nil,b,nil),/,arv(nil,c,nil))),*,
    arv(arv(nil,d,nil),-,arv(arv(nil,e,nil),*,arv(nil,f,nil))))
yes
```

3.2.9 Pós-ordem

Para percorrer uma árvore em pós-ordem:

1. percorrer a subárvore esquerda em pós-ordem,
   percorrer a subárvore direita em pós-ordem,
   visitar a raiz.

2. árvore vazia não pode ser mais percorrida.

```
pos_ordem(arv(Se,Raiz,Sd)):- pos.ordem(Se),
                             pos.ordem(Sd),
                             write(Raiz).
pos.ordem(nil).
```

Exemplo 3.2.9.1 modo(+)

```
?- instancie(A),mostrav(A),pos_ordem(A).

          *

   -f

a     /     d     *

    b   c       e   f


abc/+def*-*
X = arv(arv(arv(nil,a,nil),+,arv(arv(nil,b,nil),/,arv(nil,c,nil)))
    arv(arv(nil,d,nil),-,arv(arv(nil,e,nil),*,arv(nil,f,nil))))
yes
```

3.2.10 Linearizar uma Árvore Binária

Este programa obtém uma lista que tem como elementos os nós de uma árvore binária. Esta lista é tal que:

1. se a árvore for vazia, a lista é vazia,
2. se a árvore for não vazia, a lista tem como cabeça a raiz da árvore e a sua cauda é a concatenação das listas que contém, respectivamente, os nós das subárvores esquerda e direita.

```
modo(+,?)

linearize(<arg-l>,<arg-2>)
<arg-l> : arvore
<arg-2> : lista de nos da arvore


linearize(nil, [] ).

linearize(arv(Se,Raiz,Sd),[RaizlLista] )
                              linearize(Se,Listl),
                              linearize(Sd,List2),
                              concatenar(Listl,List2,Lista).
```

onde o programa **concatenar/3** simplesmente concatena **Listl** e **List2** em **Lista** [Monard 93].

Exemplo 3.2.10.1 modo(+,-)

```
?- instancie(X),linearize(X,Y).

X = arv(arv(arv(nil,ana,arv(nil,maria,nil)),pedro,nil) ,jose,
 arv(nil,caria,arv(nil ,flavio,nil)))

Y = [jose,pedro,ana,maria,caria,flavio]
yes
```

4 Dicionário Binário

y4.1 Introdução

Foi comentado na seção 3.2.2 que o programa **pertence (Elem, Arvore)** é ineficiente pois em algumas situações realiza uma busca exaustiva na estrutura. Como já foi adiantado, um melhoramento de tal programa pode ser conseguido se existir uma relação de ordem entre os nós da árvore.

Uma árvore binária não vazia arv(Se,Raiz,Sd) está ordenada da esquerda para a direita se:

1. todos os nós da subárvore esquerda — Se — são menores que Raiz,
2. todos os nós da subárvore direita — Sd — são maiores que a Raiz,
3. a subárvore esquerda e direita também estão ordenadas.

Esta estrutura é também chamada de *dicionário binário*. A Figura 11 mostra um exemplo de árvore e de dicionário binários.

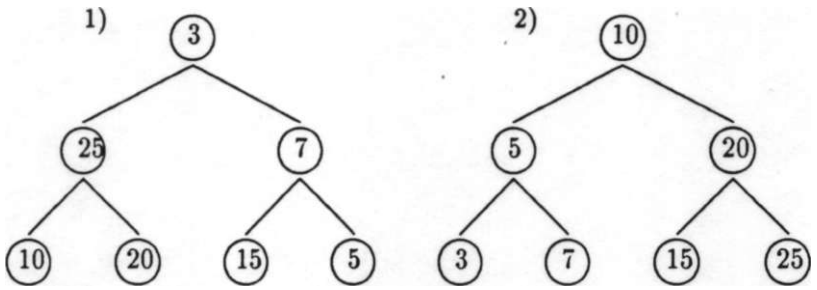


Figura 11: 1) Árvore binária 2) Dicionário binário

Uma das vantagens do dicionário binário sobre a árvore binária é que para procurar por um elemento no dicionário é necessário procurá-lo em apenas uma de suas subárvores. Após comparar o elemento procurado com a raiz do dicionário, se eles não forem iguais, a busca prossegue em apenas uma de suas subárvores. Se o elemento for menor que a raiz, a busca continua na subárvore esquerda, caso contrário, na subárvore direita.

É importante comparar o número de nós visitados numa busca bem sucedida, em uma árvore e no dicionário binário, ambos contendo os mesmos elementos. Esta contagem dá uma medida da maior eficiência de uma busca, quando esta é feita em um dicionário binário.

Este fato pode ser visualizado na tabela mostrada a seguir que exibe, para o exemplo da Figura 11, o número de nós percorridos da árvore e do dicionário, quando da busca de um determinado elemento.

Elemento Procurado	Nós Visitados			
	Arvore Binária	Total	Dicionário Binário	Total
3	3	1	10„5,3	3
5	3,25,10,20,7,15,5	7	10,5	2
7	3,25,10,20.7	5	10,5,7	3
10	3,25,10	3	10	1
15	3,25,10,20.7,15	6	10,20,15	3
20	3,25,10,20	4	10,20	2
25	3,25	2	10,20,25	3
Total		28		17

É importante observar que o dicionário considerado é muito especial no sentido que ele está perfeitamente balanceado, uma vez que todos os nós terminais estão no mesmo nível. No caso extremo do dicionário estar totalmente desbalanceado, como mostrado na Figura 12, tem-se o pior caso de busca nesta estrutura. 0 pior caso torna a busca em um dicionário semelhante à busca numa lista linear ordenada.

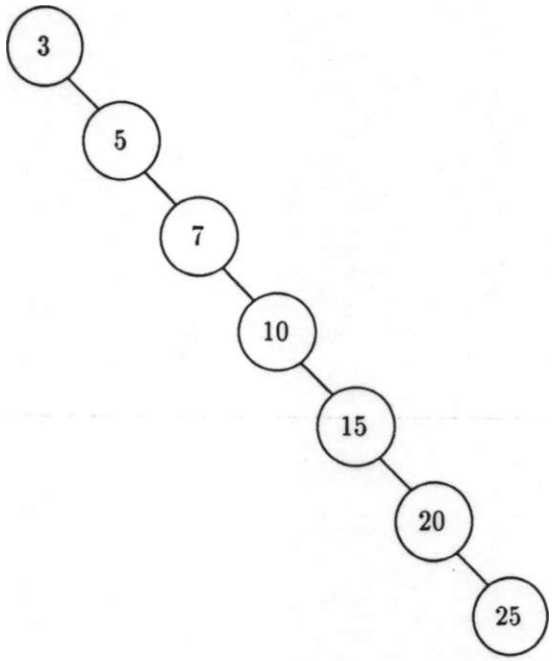


Figura 12: Árvore degenerada

4.2 Programas

Seguem vários programas para o tratamento de um dicionário binário. Como serão usados vários dos programas definidos na seção anterior, aquela mesma estrutura de árvore binária será mantida para a representação do dicionário binário, ou seja:

arv(Se,Raiz,Sd)

4.2.1 Procurar um Elemento em um Dicionário Binário e Construir o Dicionário

0 método geral para procurar um elemento Elem em um dicionário binário Dic é o seguinte:

1. se a raiz de Dic for Elem, então Elem foi encontrado,
2. se Elem for menor que a raiz de Dic a busca continua na subárvore esquerda, caso contrário,
3. a busca continua na subárvore direita.

```
modo(+,?),(?,•)

<arg-l>  :  elemento
<arg-2>  :  dicionário binário
```

```
em(Elem,arv(_,Elem,_)):-!.

em(Elem,arv(Se,Raiz,Sd)):-  menor(Elem,Raiz),
                            em(Elem,Se).

em(Elem,arv(Se,Raiz,Sd))    menor(Raiz,Elem),
                            em(Elem,Sd).
```

A definição do predicado menor/2 vai depender do tipo de dado contido no dicionário. No caso do dicionário conter somente valores numéricos, pode ser definido por:

```
menor(X,Y)  :-  X  <  Y.
```

Se o dicionário contiver valores alfanuméricos, menor/2 pode ser definido por:

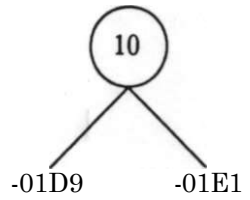
```
menor(X,Y)      X •< Y.
```

Exemplo 4.2.1.1 modo(+,+)

```
?-  em(5,arv(arv(nil,5,nil),10,arv(nil,15,nil))).
yes

?-  em(7,arv(arv(nil,5,nil),10,arv(nil,15,nil))).
no
```

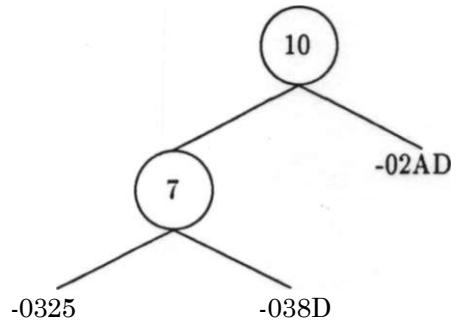
O mesmo programa em/2 pode ser usado para construir um dicionário binário. A seguir são mostradas três interrogações Prolog e as respectivas representações gráficas das respostas obtidas.

```
?- em(10,A).
A = arv(.01D9,10,.01El)
yes
```



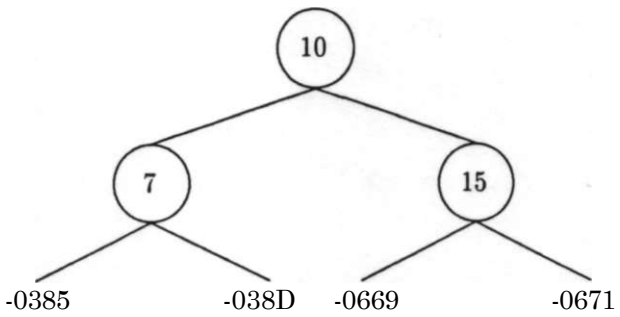
as variáveis .01D9 e .01E1 representam duas subárvores não especificadas.

```
?- em(10,A),em(7,A).
A = arv(arv(.0385,7,.038D),10,.02AD)->;
no
```



Neste caso as variáveis .0385, .038D e .02AD representam três subárvores não especificadas.

```
?- em(10,A),em(7,A),em(15,A).
A = arv(arv(.057D,7,.0585),10,arv(.0669,15,.0671))->;
no
```



Neste caso tem-se quatro subárvores nâo especificadas. Estruturas que possuem variáveis livres são chamadas *estruturas incompletas.*

0 programa em/2 pode ser usado para procurar por um elemento em um dicionário (se o dicionário não for uma estrutura incompleta, a busca pode ser mal sucedida), bem como pode ser usado para criar um dicionário. E importante observar que a ordem em que os nós são inseridos determina o balanceamento do dicionário.

Após a criação do dicionário como uma estrutura incompleta, o programa `freez/2` pode ser usado para completar a estrutura, substituindo as variáveis livres, que correspondem às subárvores não especificadas, pelo átomo `nil`.

```
modo(+,?)

<arg-l> : dicionário binário incompleto
<arg-2> : dicionário binário completo
```

```
freez(arv(_,A,_),nil)var(A),!.

freez(arv(Se,Raiz,Sd),arv(Sel,Raiz,Sdl))    freez(Se,Sel),
                                            freez(Sd,Sdl).
```

Exemplo:

```
?- em(10,A),freez(A,B).

A = arv(_01D9,10,_01El)
B = arv(nil,10,nil)
yes

?- em(10,A),em(5,A),em(15,A),em(7,A),freez(A,B).

A = arv(arv(.0421,5,arv(.0705,7,.070D)),10,arv(.0539,15,.0541))
B = arv(arv( nil,5,arv( nil,7, nil)),10,arv( nil,15, nil)) ->;
no
```

Após ter completado a estrutura, como inserir novos elementos no dicionário? Uma possível solução pode ser fazendo o caminho inverso, ou seja:

1. substituir o átomo `nil` por uma variável livre,
2. usar o programa em/2 para inserir os novos elementos e, logo após,
3. usar o programa `freez/2` para completar o novo dicionário.

Entretanto este processo é muito custoso uma vez que a estrutura do dicionário é percorrida várias vezes. 0 método geral para inserir sem repetições um elemento Elem em um dicionário binário Dic é o seguinte:

1. se a raiz de Dic for Elem, então Elem já está na árvore,
2. se a raiz de Dic for nil, inserir Elem nesta posição,
3. se Elem for menor que a raiz de Dic, inserir na subárvore esquerda, caso contrário
4. inserir na subárvore direita.

```
modo(+,?,?)

<arg-l>: elemento
<arg-2>: dicionário binário
<arg-3>: dicionário binário com o elemento

inserir(Elem,arv(Se,Elem,Sd),arv(Se,Elem,Sd)):-!.

inserir(Elem,nil,arv(nil,Elem,nil)):-!.

inserir(Elem,arv(Se,Raiz,Sd),arv(Se1,Raiz,Sd)):-
                                        menor(Elem,Raiz),
                                        inserir(Elem,Se,Sei).

inserir(Elem,arv(Se,Raiz,Sd),arv(Se,Raiz,Sdl)):-
                                        menor(Raiz,Elem),
                                        inserir(Elem,Sd,Sdl).
```

Exemplo 4.2.1.2 modo(+,+,-)

```
?- inserir(l,arv(nil,2,nil),X).

X = arv(arv(nil,1,nil),2,nil) ->;
no

?- inserir(1,arv(nil,2,nil),W),inserir(3,W,Y),inserir(7,Y,X).

W = arv(arv(nil,1,nil),2,nil)
Y = arv(arv(nil,1,nil),2,arv(nil,3,nil))
X = arv(arv(nil,1,nil),2,arv(nil,3,arv(nil,7,nil))) ->;
no
```

0 programa inserir/3 pode também ser usado para verificar se um elemento está ou não no dicionário. Para isto, basta exigir que o dicionário de entrada e saída sejam iguais.

4.2.2 Verificar se uma Estrutura é um Dicionário Binário

Uma estrutura é um dicionário binário se:

1. for uma estrutura vazia, ou
2. tem a forma arv(Se,Raiz,Sd) e
   (a) Raiz for maior que todos elementos da subárvore esquerda Se,
   (b) Raiz for menor que todos os elementos da subárvore direita Sd e
   (c) Se e Sd também forem dicionários binários.

modo(+)

<arg-l>: estrutura

```
dic(nil):-!.

dic(arv(Se,Raiz,Sd)): - raiz.maior(Raiz.Se),
                        raiz.menor(Raiz,Sd),
                        dic(Se),
                        dic(Sd).

raiz.maior(_,nil):-!.

raiz.maior(Raiz,arv(. ,RI,.)):- menor(RI,Raiz).

raiz.menor(.,nil):-!.

raiz.menor(Raiz,arv(. ,R1,-)):- menor(Raiz,RI).
```

Exemplo 4.2.2.1 modo(+)

```
?- dic(arv(arv(nil,5,nil),10,nil)).
yes

?- dic(arv(arv(nil,5,nil),10,arv(nil,15,nil))).
yes

?- dic(arv(arv(nil, 15,nil), 10, arv(nil, 15 ,nil))).
no
```

4.2.3 Elemento Máximo e Mínimo de um Dicionário Binário

0 elemento máximo de um dicionário binário é o nó que está mais à direita na subárvore direita do dicionário, ou seja, é a raiz de uma subárvore da forma:

arv(_,Max,nil)

enquanto que o elemento mínimo é o nó que está mais à esquerda na subárvore esquerda do dicionário binário, ou seja, é a raiz de uma subárvore da forma:

arv(nil,Min,_)

0 modo de chamada para os programas que encontram o elemento máximo e mínimo de um dicionário binário, é:

modo(?,+)

<arg-l> : elemento (máximo, minimo)
<arg-2> : dicionário binário

e os programas são, respectivamente:

```
máximo(Max,arv(_,Max,nil)).

máximo (Max, arv (Se, Raiz, Sd))    máximo (Max ,Sd) .


minimo (Min, arv (nil, Min, _)).

minimo (Min, arv (Se, Raiz, Sd)):-  minimo (Min, Se).
```

Exemplo 4.2.3.1 modo(-,+)

```
?-  instancie(A),maximo(Elem,A).

A = arv(arv(nil,5,arv(nil,7,nil)),10,arv(nil,15,nil))
Elem = 15 ->;
no

?-  instancie(A),minimo(Elem,A).

A = arv(arv(nil,5,arv(nil,7,nil)),10,arv(nil,15,nil))
Elem = 5 ->;
no
```

4.2.4 Deleção de um Elemento em um Dicionário Binário

A deleção de um elemento de um dicionário binário deve considerar a posição em que se encontra o nó correspondente àquele elemento dentro do dicionário binário. 0 nó pode tanto ser um nó terminal quanto um nó interno ao dicionário binário.

A remoção de um nó terminal (folha), é feita de forma trivial, isto é, apenas substitui-se o elemento a ser removido pelo `nil`.

Para a remoção de um nó interno, é necessário encontrar um nó do dicionário binário que possa ser transferido para a posição daquele nó interno, mantendo-se as características do dicionário binário. Isto pode ser feito através do seguinte esquema:

1. se o nó a ser deletado é tal que uma de suas subárvores é vazia, então o resultado é a subárvore não vazia - Figura 13.

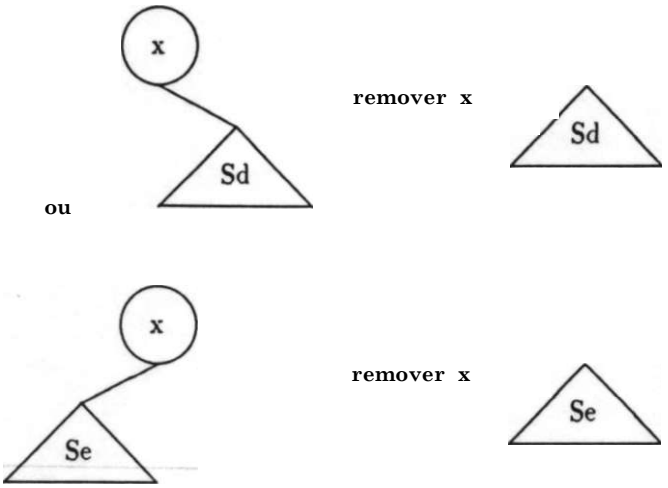


Figura 13: Remoção de um nó interno quando uma das subárvores é vazia

2. no caso de ambas as subárvores não serem vazias, deve ser escolhido um elemento **v,** para substituir o elemento **x** a ser removido — Figura 14.

Pode-se escolher **v** como o menor elemento da subárvore direita. Em um dicionário binário este elemento está no nó mais à esquerda da subárvore direita. Um outro elemento que poderia ser utilizado, e que satisfaria da mesma forma a definição de dicionário binário, é o maior elemento da subárvore esquerda, que se encontra no nó mais à direita da subárvore esquerda.

Em ambos os casos o procedimento consiste na remoção do elemento **v** do nó terminal, e na sua inserção no nó que contém o elemento **x.**

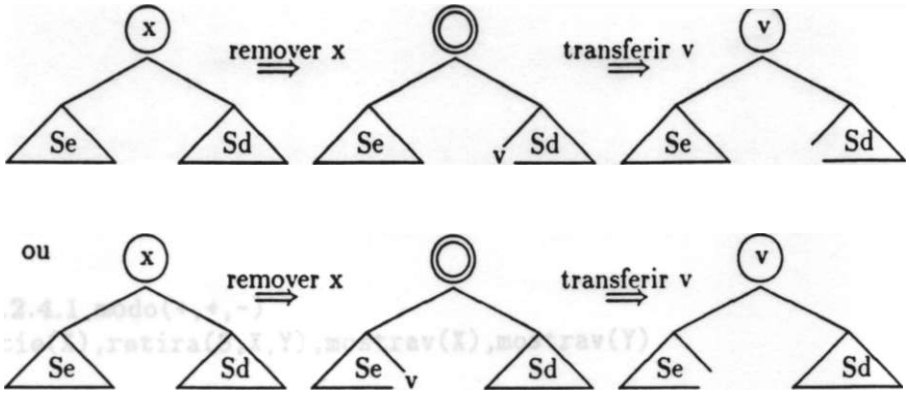


Figura 14: Remoção de um nó interno quando ambas subárvores são não vazias.

```
modo(?,+,?)

retira(<arg-1>,<arg-2>,<arg-3>)
<arg-l>: elemento
<arg-2>: dicionário binário
<arg-3>: novo dicionário binário


retira(X,arv(nil,X,Sd),Sd):-!.

retira(X,arv(Se,X,nil),Se):-!.

ret ira(X,arv(Se,X,Sd),arv(Se,Y,Sdl))
                        retiramin(Y,Sd,Sdl).

retira(X,arv(Se,Raiz,Sd),arv(Se1,Raiz,Sd)):-
                        maior(Raiz,X),
                        retira(X,Se,Sel).

retira(X,arv(Se,Raiz,Sd),arv(Se,Raiz,Sdl)):-
                        menor(Raiz,X),
                        retira(X,Sd,Sdl).

retiramin(Y,arv(nil,Y,R),R).

retiramin(Y,arv(Se,Raiz,Sd),arv(Sei,Raiz,Sd)):-
                        retiramin(Y,Se,Sel).
```

Exemplo 4.2.4.1 modo(+,+,-)

```
?- instancie(X),retira(8,X,Y),mostrav(X).mostrav(Y).

        20

     10   30

8          40

  9


     20

  10   30

9        40

X = arv(arv(arv(nil,8,arv(nil,9,nil)),10,nil),20,
                                        arv(nil,30,arv(nil,40,nil)))
Y « arv(arv(arv(nil,9,nil),10,nil),20,arv(nil,30,arv(nil,40,nil))) ->;
no

?- instancie(X),retira(20,X,Y),mostrav(X),mostrav(Y).

        20

     10   30

8          40

  9


        30

     10   40

8

  9

X = arv(arv(arv(nil,8,arv(nil,9,nil)),10,nil),20,
                                      arv(nil,30,arv(nil,40,nil)))
Y = arv(arv(arv(nil,8,arv(nil,9,nil)) ,10,nil),30,arv(nil,40,nil)) ->;
no
```

4.2.5 Árvores AVL

Foi comentado anteriormente que é importante manter o balanceamento de uma árvore, uma vez que as operações realizadas em árvores balanceadas são mais eficientes.

Uma árvore está perfeitamente balanceada se, para cada nó, o número de nós nas suas subárvores esquerda e direita diferem, no máximo, em um. Portanto, após uma insersão ou deleção em uma árvore perfeitamente balanceada, quase sempre é necessário rebalancear a árvore. 0 restabelecimento do balanceamento de uma árvore é, via de regra, uma operação complicada e com alto consumo de tempo.

Uma solução para facilitar tal operação é a da não exigência de um balanceamento perfeito e do uso de outro critério menos restrito que. ainda assim, permita o desenvolvimento de algoritmos eficientes.

Uma das definições de balanceamento não perfeito foi dada por Adelson-Velskii e Landis [Horowitz 84], [Wirth 76], e as árvores que verificam esta definição são chamadas *árvores AVL.*

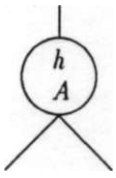
Seja **arv**(Se**,Raiz**,Sd) a estrutura que representa uma árvore binária, e h_c e *hj* a altura das subárvores esquerda (Se) e direita (Sd) respectivamente. Ela é uma árvore AVL se:

1. for vazia, ou
2. Se e Sd forem tais que as suas alturas verificam \h$_t$ - < 1.

As operações para localizar, inserir ou deletar um nó de uma árvore AYL podem ser realizadas em tempo proporcional a 0(log(n)), no pior caso.

Adelson-Velskii e Landis provaram que uma árvore AVL, no pior caso, tem como altura não mais que a altura de sua correspondente perfeitamente balanceada, acrescida de 45%.

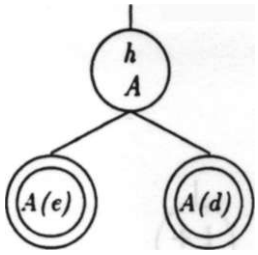
Graficamente um nó de uma árvore AVL será representado por:



onde:

- *A* é o rótulo do nó e
- $h = h_e - hd$ (*h* podendo assumir os valores +1, 0 ou —1)

As subárvores esquerda .4(e) e direita *A{d)* de um nó rotulado *A* serão representadas graficamente por:



Quando são realizadas operações de insersão e deleção de elementos, às vezes é necessário realizar rotações na árvore para mantê-la balanceada. Essencialmente, para reequilibrar a árvore, podem ser realizados quatro tipos de rotações chamadas: LL, RR, LR e RL, simétricas duas a duas. Isto é, LL com RR e LR com RL.

Quando um nó A' for inserido numa árvore AVL e o fator de equilíbrio de seu ancestral mais próximo permanecer dentro dos valores permitidos (+1, 0 ou -1), não é necessário rebalancear a árvore resultante, uma vez que esta é também uma árvore AVL. No caso do fator de equibíbrio do ancestral mais próximo passar a ser +2 ou -2, é necessário realizar uma rotação para manter a estrutura de árvore AVL.

Considerando que *X* é o nó inserido e que *A* é seu ancestral mais próximo, podem ser realizados os seguintes tipos de rotação para reequilibrar a árvore:

1. LL - -Y é inserido na subárvore esquerda da subárvore esquerda de A.
2. RR - A' é inserido na subárvore direita da subárvore direita de *A*.
3. LR - A" é inserido na subárvore direita da subárvore esquerda de A.
4. RL - A' é inserido na subárvore esquerda da subárvore direita de *A*.

A seguir são mostrados graficamente estes quatro esquemas de rotação.

1. Rotação LL: A* é inserido na subárvore esquerda da subárvore esquerda de *A* — Figura 15.

   Seja *X* um novo elemento que, quando inserido em i?(e), desbalança a árvore. Para rebalanceá-la, faz-se uma rotação de tal forma que *B* passa a ser a raiz da subárvore e *B(d)* é concatenada à *A*, que passa a ser a raiz da subárvore direita de como mostrado na Figura 16.

2. Rotação RR: A" é inserido na subárvore direita da subárvore direita de *A* — Figura 17.

   Seja A' um novo elemento que, quando inserido em £?(</), desbalança a árvore. Para rebalanceá-la, faz-se uma rotação de tal forma que *B* passa a ser a raiz da subárvore e *B(e*) é concatenada à *A*, que passa a ser a raiz da que era subárvore esquerda de B. como mostrado na Figura 18.

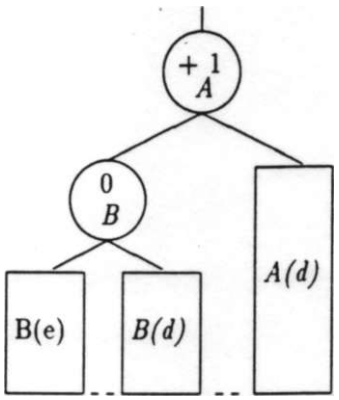


Figura 15: Subárvore balanceada antes da inserção do elemento A'para ilustrar rotação LL

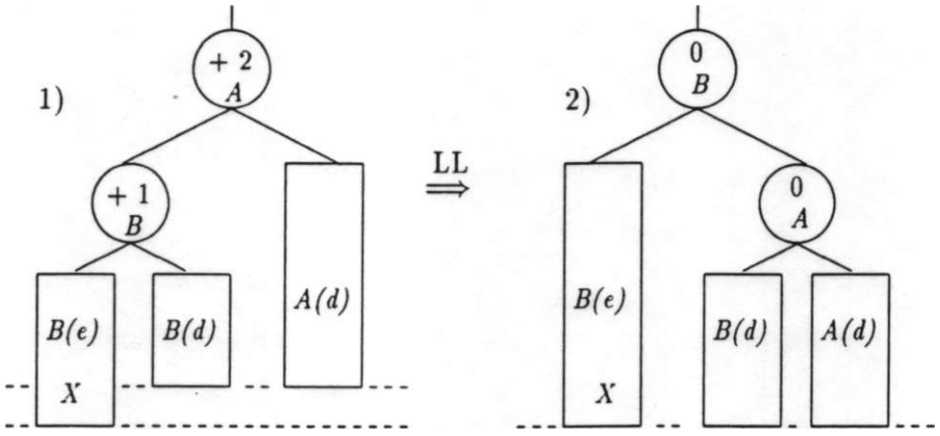


Figura 16: 1) Altura de B(e) 2) Subárvore rebalanceada

3. **Rotação LR:** *X* é inserido na subárvore direita da subárvore esquerda — Figura 19.

   Seja A" um novo elemento que, quando inserido em C(e), desbalança a árvore. Para rebalanceá-la. faz-se uma rotação de tal forma que *C* passa a ser a raiz da subárvore, distribuindo C(e) e *C(d)* nas subárvores esquerda e direita de C, como mostrado em Figura 20 e Figura 21.

4. **Rotação RL:** A' é inserido na subárvore esquerda da subárvore direita de A.

   Este caso é simétrico do anterior, onde *A(d)* apareceria como a subárvore esquerda de *A* e *B* apareceria como a raiz da subárvore direita de *A*.

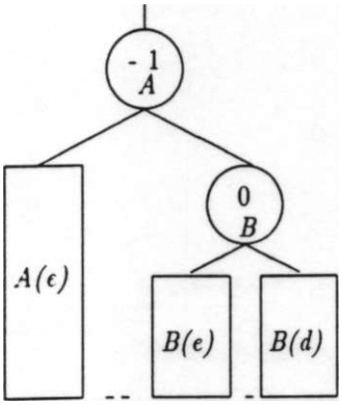


Figura 17: Subárvore balanceada antes da inserção do elemento A' para ilustrar rotação RR

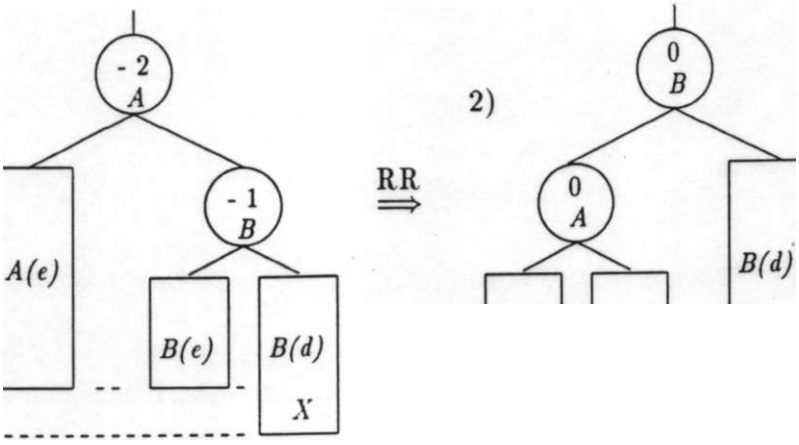


Figura 18: 1) Aumenta a altura de B(d) 2) Subárvore rebalanceada

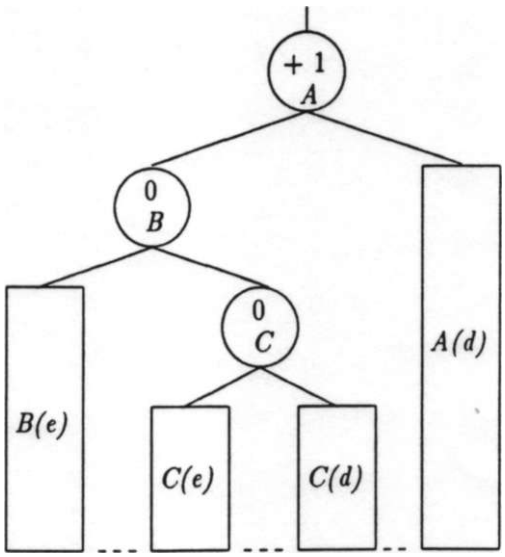


Figura 19: Subárvore balanceada antes da inserção do elemento A' para ilustrar rotação LR

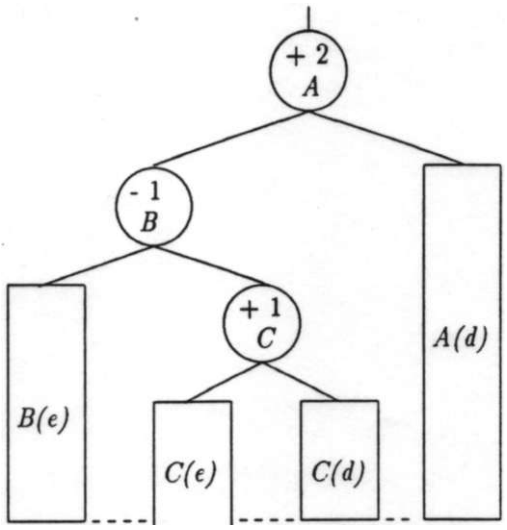


Figura 20: Aumenta a altura de C(E)

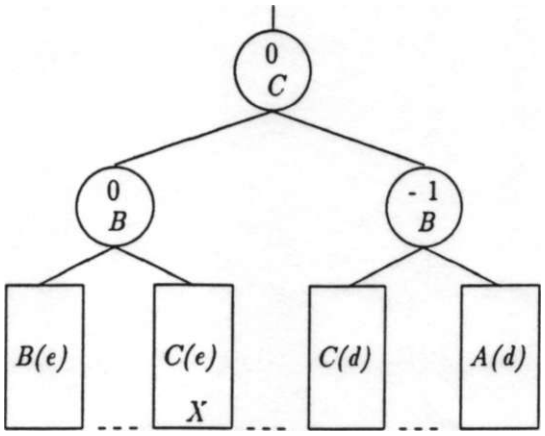


Figura 21: Resultado da Rotação LR

As operações realizadas em uma árvore AVL são semelhantes às realizadas em um dicionário binário. Entretanto, é necessário que se guarde informações extras, a fim de ser possível rebalanceá-la, quando for o caso, após uma operação de insersão ou deleção.

A seguir é apresentado um programa que adiciona um elemento em uma árvore AVL, e mantém a sua estrutura balanceada [Bratko 90], [van Emden 81]. Ele é definido pelo predicado:

ins _avl(Avl,Elem,NovaAvl)

onde Avl é uma árvore AVL e NovaAvl é a árvore AVL resultante da inserção do elemento Elem na árvore Avl.

As árvores AVL serão representadas por termos da forma:

arv(Se,Raiz,Se)/H

onde H representa a altura da árvore. A árvore vazia é representada por nil.

modo(?,+,?)

<arg-l>: arvore AVL
<arg-2>: elemento
<arg-3>: arvore AVL com elemento

```
ins_avl(nil/0,X,arv(nil/0,X,nil/0)/l).

ins.avl(arv(Se,Y,Sd)/Hy,X,NovaArv):-
                    maior(Y,X),
                    ins_avl(Se,X,arv(Sel,Z,Se2)/_),
                    combine(Se1,Z,Se2,Y,Sd,NovaArv).

ins.avl(arv(Se,Y,Sd)/Hy,X,NovaArv) :-
                    maior(X,Y),
                    ins_avl(Sd,X,arv(Sdl,Z,Sd2)/_),
                    combine(Se,Y,Sdl,Z,Sd2.NovaArv).

combine(Tl/H1,A,arv(T21,B,T22)/H2,C,T3/H3,
    arv(arv(Tl/Hl,A,T21)/Ha,B,arv(T22,C,T3/H3)/Hc)/Hb)
                    H2 > Hl,
                    H2 > H3,
                    Ha is Hl+1,
                    Hc is H3+1,
                    Hb is Ha+1.

combine(Tl/Hl,A.T2/H2,C,T3/H3,
    arv (Tl/Hl,A,arv(T2/H2,C.T3/H3)/Hc)/Ha):-
                    Hl >« H2,
                    Hl >= H3,
                    maxl(H2,H3,Hc),
                    maxl(Hl,Hc,Ha).

combine(Tl/Hl, A,T2/H2,C,T3/H3,
    arv(arv(Tl/Hl,A,T2/H2)/Ha,C,T3/H3)/Hc):-
                    H3 >= H2,
                    H3 >= Hl,
                    maxl(Hl,H2,Ha),
                    maxl(Ha,H3,Hc).

maxl(U.V.M):-
                    U > V, !, M is U+l;
                    M is V + 1.
```

onde o programa maior/2 pode ser definido de maneira semelhante ao programa menor/2 definido na seção 4.2.1. pg.25.

5 Generalização de Dicionário Binário

Na seção anterior foi apresentada a estrutura de dicionário binário, foram discutidos vários programas que lidam com esta estrutura e foi enfatizada a importância de se manter o balanceamento do dicionário.

A estrutura do dicionário, entretanto, lida apenas com chaves únicas, isto é, cada nó do dicionário é rotulado com uma informação considerada indivisível.

Porém, para a resolução de muitos problemas é necessário o processamento de mais do que uma chave de informação. Um exemplo típico é o do tratamento do nome de uma pessoa numa situação em que o nome e o sobrenome devem, ambos, ser tratados como chaves.

Há várias estruturas adequadas para manipulação de chaves múltiplas [Scarpelli 84]. Nesta seção será discutida uma destas estruturas — chamada árvore *Quad* — que é uma generalização de dicionário binário.

5.1 Arvore Quad

No dicionário binário, cada nó subdivide a coleção de nós que estão na subárvore da qual ele é raiz, em duas subcoleções, determinando, com base no valor da sua chave, quais nós ficam na sua subárvore esquerda e quais na direita. Em uma árvore Quad de k dimensões, onde k é o número de chaves, cada nó subdivide a coleção de nós que estão na subárvore da qual ele é raiz, em 2^k subárvores, determinando, com base nos valores de suas k chaves, quais nós ficam em cada uma das suas 2^k subárvores. Assim, cada nó terá no máximo 2^k filhos. Nos procedimentos de inserção e busca, as comparações são feitas com base nos k valores das chaves de cada nó, para decidir qual subárvore deve ser percorrida.

Na estrutura de árvore Quad de dimensão (k = 2), os conceitos e procedimentos básicos que serão apresentados, podem ser facilmente estendidos para mais dimensões.

Na árvore Quad de dimensão dois, cada nó representa duas chaves e tem no máximo quatro filhos. Considerando estas duas chaves como pontos no plano e o ponto (xi, j/i) como a raiz da árvore Quad, esta raiz divide o espaço onde se localizam os outros nós em quatro quadrantes denominados de quadrantes 1, 2, 3, 4 respectivamente, como mostra a Figura 22:

Cada filho do nó com chaves (xi, j/i) pertence a um destes quadrandes e é a raiz da subárvore que representa este quadrante. Os pontos que se localizam no quadrante 1 ficarão na primeira subárvore deste nó. Os pontos do quadrante 2 na segunda subárvore e assim por diante, como mostra a Figura 23.

De maneira análoga, cada filho do nó (xi,j/i) terá, no máximo, quatro filhos e assim sucessivamente, ou seja, cada filho é raiz de uma árvore Quad de ordem 2.

No caso de pontos que se localizam sobre a linha que sai de um nó, adota-se a convenção dos quadrantes 1 e 3 serem fechados e quadrantes 2 e 4 serem abertos.

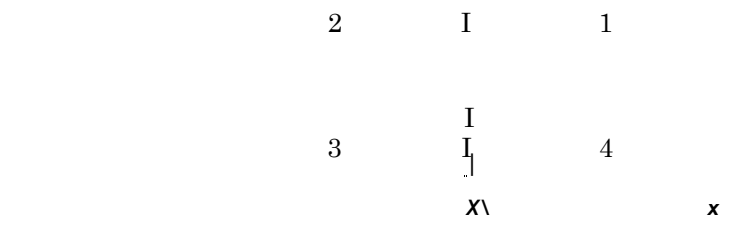


Figura 22: Quadrantes de uma árvore Quad de ordem 2

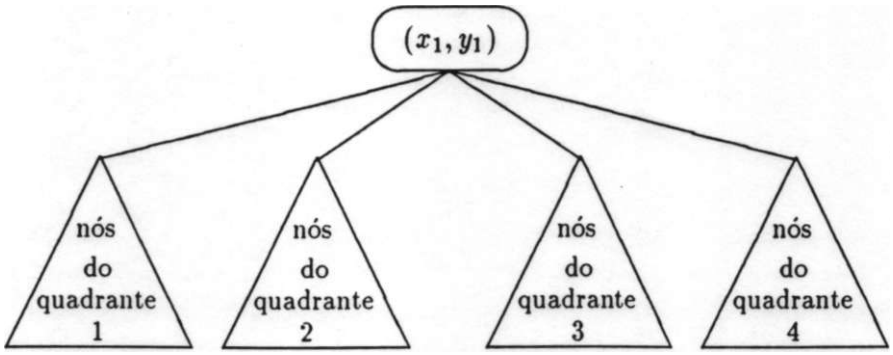


Figura 23: Subárvores de uma árvore Quad de ordem 2

Resumindo, se o nó com chaves (xi,yi) é a raiz da árvore Quad, a relação de ordem para identificar o quadrante de um nó genérico (x,y) é mostrata da Tabela 1.

Para representar árvores Quad será usada a seguinte estrutura:

quad(Raiz,Q1,Q2,Q3,Q4)

onde Ql, Q2, Q3 e Q4 — que também são árvores Quad —, representam os quadrantes da árvore Quad que tem Raiz como raiz. A raiz será representada por uma lista de dois elementos.

Os programas que lidam com esta estrutura são semelhantes aos da seção anterior, já que a árvore Quad é uma generalização de dicionário binário. Serão usados, portanto, os mesmos nomes de relações que os programas análogos de dicionário binário.

5.2 Procurar um Elemento em uma Árvore Quad e Construção da Arvore Quad

0 método geral para procurar um elemento Elem em uma árvore Quad de ordem 2 é o seguinte:

Quadrante	Relaqão
1	*X* > *X*\ *y* > *y i*
2	*X* < *X*\ *y* > *y* i
3	*X* < *Xj* *y* < *y* i
4	*X* > *Xi* *y* < *y* i

Tabela 1: Relação da5 coordenadas de um ponto (x, *y) versus* Quadrante

1. se a raiz da árvore Quad for Elem, então El em foi encontrado;
2. continuar a busca no quadrante 1, 2, 3 ou 4 dependendo da relação entre Elem e a raiz da árvore.

0 programa para verificar em que quadrante se encontra o elemento procurado é dado por:

quadrante(N,Raiz,Elem)

que dá o quadrante N onde Elem está situado relativo à Raiz. No caso da árvore Quad conter valores numéricos esta relação é:

```
modo(?,+,+)

quadrante(<arg-1>,<arg-2>,<arg-3>)
<arg-l>: numero do quadrante
<arg-2>: raiz da subarvore
<arg-3>: elemento a ser localizado
```

```
quadranteÜ, [XI,Y1] , [X,Y])     X >= XI,
                                 Y >= Yl.

quadrante(2, [XI,Yl],[X,Y]):-    X < XI,
                                 Y > Yl.

quadrante(3,[XI,Y1],[X,Y])       X =< XI,
                                 Y =< Yl.

 quadrante(4, [XI,Y1] , [X.Y]) :•- X > XI,
                                   Y < Yl.
```

e o programa para procurar por um elemento em uma árvore Quad é:

```
modo(?,*),  (•,?)

em(<arg-l>,<arg-2>)
<arg-l>:  elemento
<arg-2>:  arvore Quad
```

```
em(Elem,quad(Elem,.,_,_,_)):-! .

em(Elem,quad(Raiz,  Q1,_,_,_)):-  quadrante(l,Raiz,Elem),
                                  em(Elem,Ql).

em(Elem,quad(Raiz,.,Q2,.,.)):-  quadrante(2,Raiz,Elem),
                                em(Elem,Q2).

em(Elem,quad(Raiz,.,.,Q3,.)):-  quadrante(3,Raiz,Elem),
                                em(Elem,Q3).

em(Elem,quad(RaizQ4)):-  quadrante(4,Raiz,Elem),
                         em(Elem,Q4).
```

Como foi visto anteriormente, este programa pode também ser usado para construir uma árvore Quad como uma estrutura incompleta.

A seguir é mostrado o programa que insere um elemento em uma árvore Quad.

```
modo(?,+,?),  (+,?,?)

inserir(<arg-l>,<arg-2>,<arg-3>)
 <arg-l>:  Elemento a ser inserido
 <arg-2>:  Arvore Quad
 <arg-3>:  Arvore Quad com o elemento
```

```
inserir(X,nil,quad(X,nil,nil,nil,nil))      !.

inserir(X,quad(X,Q1,Q2,Q3,Q4),quad(X,Ql,Q2,Q3,Q4))!.

inserir(Elem,quad(Raiz,Q1,Q2,Q3,Q4),quad(Raiz,Q11,Q2,Q3,Q4)):-
                    quadrante(1,Elem,Raiz),
                    inserir(Elem,Ql,Q11).

inserir(Elem,quad(Raiz,Q1 ,Q2,Q3,Q4),quad(Raiz, Q1,Q22,Q3,Q4)):-
                    quadrante(2,Elem,Raiz),
                    inserir(Elem,Q2,Q22).

inserir(Elem,quad(Raiz,Q1,Q2,Q3,Q4),quad(Raiz,Q1,Q2,Q33,Q4)):-
                    quadrante(3,Elem,Raiz),
                    inserir(Elem,Q3,Q33).

inserir(Elem,quad(Raiz,Q1,Q2,Q3,Q4),quad(Raiz ,Q1,Q2,Q3,Q44)):-
                    quadrante(4,Elem,Raiz),
                    inserir(Elem,Q4,Q44).
```

5.3 Árvore Quad Ótima

Foi visto que o balanceamento de uma árvore depende da ordem de insersão de seus elementos. Em alguns casos, pode ser conveniente investir um certo tempo na construção de uma árvore otimizada, de forma a garantir um melhor tempo médio de busca.

A árvore Quad otimizada para $k = 2$ tem a seguinte propriedade:

> para todo nó z, nenhuma subárvore de z tem mais da metade dos nós da árvore cuja raiz é z.

Para construir esta árvore, deve-se ordenar lexicograficamente os nós pela primeira chave e depois pela segunda. A raiz da árvore é escolhida como o elemento médio desta sequência ordenada, isto é, o elemento que divide a sequência em duas sequências tal que o comprimento das mesmas difere, quanto muito, em um elemento. Logo após, esta sequência é subdividida em quatro subcoleções, para formar os descendentes da raiz. Para cada subcoleção aplica-se o processo recursivamente. Com este método, os nós que antecedem o elemento médio da sequência ficarão nos quadrantes 2 e 3 e os nós que o sucedem ficarão nos quadrantes 1 e 4. Portanto, nenhuma subárvore do nó terá mais da metade dos nós.

Para se construir uma árvore Quad ótima, a partir de uma lista de nós, deve-se:

1. ordenar a lista de nós,
2. criar a lista de entrada ótima,

3. inserir, um a um, os elementos da lista de entrada ótima na árvore Quad.

```
modo(+,?)

crie.quad.otima(<arg-l>,<arg-2>)
<arg-l>: Lista de nos a serem inseridos
<arg-2>: Quad ótima que contem estes nos

crie_quad_otima(ListaNos,QuadOtima)
                    sort(ListaNos.ListaOrdenada),
                    entrada_otima(ListaOrdenada,ListaOtima),
                    quad.ótima(ListaOtima,QuadQtima).
```

O programa **sort/2** é um programa pré-definido que ordena a lista **ListaNos** em **ListaOrdenada** suprimindo, quando houverem, elementos repetidos. Quase todas as implementações de Prolog oferecem algum programa pré-definido deste tipo. Se não for o caso, este programa deve ser implementado usando algum dos algoritmos de classificação. Em [Bratko **90]** é mostrado em detalhes a implementação dos algoritmos de classificação Quicksort, Bubblesort e Insertionsort. Estas implementações também podem ser encontradas em [Monard **93].**

O programa entrada.otima/2 constrói, a partir da lista que contém os nós ordenados, uma lista que contém estes mesmos nós em uma ordem tal que, ao inseri-los na árvore Quad obtém-se a estrutura ótima. O processo a ser seguido é o seguinte:

1. se a lista de entrada estiver vazia, a lista ótima é a lista vazia, caso contrário,
2. a lista ótima tem como cabeça o elemento **Elem** que é o elemento médio da lista ordenada e como cauda, a concatenação das listas obtidas aplicando, recursivamente, este processo às listas ordenadas **Men** e Mai, que contém, respectivamente, os elementos menores e maiores que **Elem.**

```
entrada_otima( •,[]).

entrada_otima(ListaOrdenada, [ElemC]):
                    divida(ListaOrdenada,Men,Mai,Elem),
                    entrada_otima(Men,Menl),
                    entrada_otima(Mai,Mail),
                    append(Menl,Mail,C) .
```

O programa divida/4 deve encontrar o elemento médio da lista ordenada — ListaOrdenada — bem como retornar as listas dos elementos maiores e menores que ele, sem alterar a ordem em que aparacem na lista ordenada. O programa é:

```
divida([El] , •, • ,E1):-!.

divida(ListaOrdenada,Men,Mai,Elem)
                    no.elementos(ListaOrdenada,N),
                    NI is N // 2.
                    div1 (N1,ListaOrdenada,Men,Mai,Elem).

divl(0,[ElemiMai] , • ,Mai,Elem):-!.

div1(N, [EIC] , [EIMen],Mai,Elem):-
                    NI is N - 1,
                    divl(Nl,C,Men,Mai,Elem).
```

onde no.elementos/2 dá o número de elementos de uma lista.

```
no.elementos( • ,0).

no.elementos([C|Cau],N):-
                    no.elementos(Cau,NI),
                    N is Nl+1.
```

Finalmente, o programa quad.otima/2 deve inserir os elementos desta lista na árvore Quad.

```
quad.otima(ListaOtima,QuadOtima):-
                    inslist(ListaOtima,nil,QuadOtima).

inslist( • ,Quad,Quad).

inslist([No|L],Q,Quad)inserir(No,Q,Q1),
                              inslist(L,Q1,Quad) .
```

onde o programa inserir/3 é o definido na seção 5.2.

A fim de exemplificar todos os passos do procedimento, a seguir é mostrado a execução do corpo do programa crie.quad.otima/2.

Quando o Prolog é interrogado com:

```
?- crie.quad.otima([[2,2],[1,1]],Quad).
```

a lista de entrada [[2,2] , [1,1]] é ordenada, através do programa sort/2 na lista [[1,1] , [2,2]]. A partir desta lista ordenada uma outra lista é obtida, de tal forma que quando seus elementos forem inseridos na árvore Quad, obtém-se a estrutura ótima. No exemplo apresentado, a lista ordenada [[1,1] , [2,2]] é transformada em [[2,2] , [1,1]].

Os elementos desta lista são então inseridos na árvore Quad, obtendo-se a árvore Quad representada pela estrutura Prolog:

```
quad([2,2],
      quad([1,1],
            nil,
            nil,
            nil,
            nil),
      nil,
      nil,
      nil)
```

cuja representação gráfica é mostrada na Figura 24.

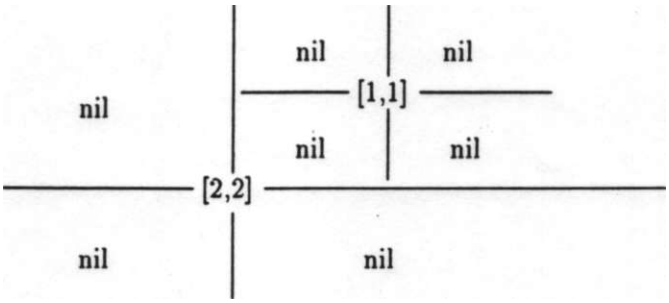


Figura 24: Árvore Quad ótima com 2 nós

Analogamente, se Prolog for interrogado com:

```
?- crie.quad([[2,2] , [-1,-1] , [1,1]] ,Quad).
```

a lista de entrada [[2,2] , [-1,-1] , [1,1]] é ordenada, através do programa sort/2 na lista [[-1,-1] , [1,1] , [2,2]].

A seguir, é obtida a lista ótima de entrada a partir da lista ordenada, de tal forma que ao inserir os seus elementos na árvore Quad, obtém-se a estrutura Quad ótima. No exemplo, a lista ótima de entrada obtida é: [[1,1] , [-1,-1] , [2,2]].

Os elementos desta lista são, então inseridos na árvore Quad, obtendo-se a árvore Quad representada pela estrutura Prolog:

```
quad([1,1] ,
      quad( [-1,-1],
            nil,
            nil,
            nil,
            nil),
      nil,
      quad ([2,2],
```

```
        nil,
        nil,
        nil,
        nil),
    nil)
```

cuja representação gráfica é mostrada na Figura 25

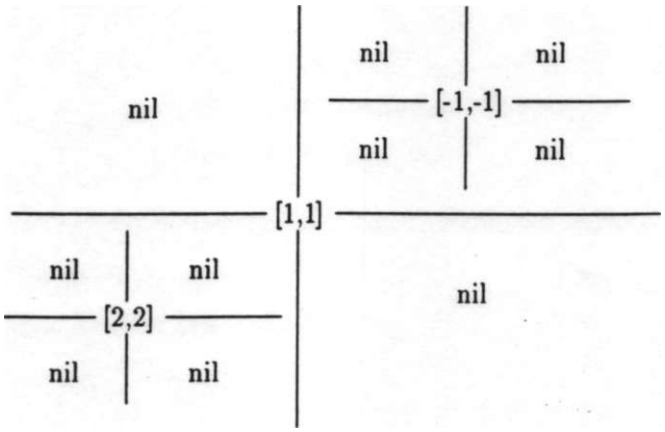


Figura 25: Árvore Quad Ótima com 3 nós

6 Conclusões

Neste documento foram introduzidos alguns conceitos fundamentais da estrutura de dados árvore, bem como apresentadas e discutidas as implementações Prolog de diversos programas para o processamento desta estrutura.

Foi enfatizada a idéia de manter a árvore balanceada, ou aproximadamente balanceada, de maneira a prevenir que a árvore se degenere em uma lista. 0 fato da árvore estar balanceada (ou aproximadamente balanceada) garante um tempo de acesso a dados relativamente rápido, mesmo no pior caso. Diversos esquemas de balanceamento de árvores foram discutidos

O conceito de dicionário binário foi estendido para situações de múltiplas chaves, através da estrutura de árvore Quad.

Agradecimentos: a Roberto de Lima Vidal e Solange Rezende Rodrigues pelo auxílio na depuração de vários dos programas apresentados; a Carmen Lúcia Pagadigorria pela ajuda na elaboração da parte gráfica contida neste documento.

Referências

[Aho 83] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Araribóia 89] Araribóia, G. *Inteligência Artificial: Um Enfoque Prático*. LTC, 1989.

[Arity 90] Arity Corporation. *The Arity/Prolog Programming Language*. 1990.

[Balbin 84] Balbin, I.; Lecot, K. *Logic Programming: A Classified Bibliography*. Wilgrass Books Pty Ltd, Australia, 1984.

[Bharath 86] Bharath, R. *An Introduction to Prolog*. TAB Books Inc., 1986.

[Bharath 89] Bharath, R. *Prolog: Sophisticated Applications in Artificial Intelligence*. Windcrest Books, 1989.

[Bratko 90] Bratko, I. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, (2nd Ed.), 1990.

[Burnham 85] Burnham, W.D.; Hall, A.R. *Prolog Programming and Applications*. MacMillan Publishing Company, 1985.

[Casanova 87] Casanova, M.A.; Giorno, F.A.C.; Furtado, A.L. *Programação em Lógica e a Linguagem Prolog*. Editora Blücher Ltda, São Paulo, 1987.

[Clark 84] Clark, K.L.; McCabe, K.G. *Micro-PROLOG: Programming in Logic*. (2nd Ed.) Prentice-Hall, 1984.

[Clocksin 87] Clocksin, W.F.; Mellish, C.S. *Programming in Prolog*. (3rd Ed.) Springer-Verlag, 1987.

[Coelho 88] Coelho, H.; Cotta, J.C. *Prolog by Examples*. Springer-Verlag, 1988.

[Garavaglia 87] Garavaglia, S. *Prolog: Programming Techniques and Applications*. Harper & Row, 1987.

[Gonnet 84] Gonnet, G.H. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

[Hogger 84] Hogger, C.J. *Introduction to Logic Programming*. Academic Press, 1984.

[Horowitz 84] Horwitz, E.; Sahni, S. *Fundamentos de Estruturas de Dados*. Editora Campus Ltda., 1984.

[Kluzniak 85] Kluzniak, F.; Szpakowicz, S. *Prolog for Programmers*. Academic Press, 1985.

[Knuth 78a] Knuth, D.E. *The Art of Computer Programming, Vol I: Fundamental Algorithms*. (2nd Ed.) Addison-Wesley, 1978.

[Knuth 78b] Knuth, D.E. *The Art of Computer Programming, Vol III: Sorting and Searching*. (2nd Ed.) Addison-Wesley, 1978.

[Kowalski 79] Kowalski, R.A. *Logic for Problem Solving*. Artificial Intelligence Series, Vol 7, Elsevier-North Holland, 1979.

[Lloyds 84] Lloyds, J.W. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[Malpas 87] Malpas, J. *Prolog: A Relational Language and its Applications*. Prentice-Hall, 1987.

[Marcus 86] Marcus, C. *Prolog Programming: Aplications for Database System, Expert Systems and Natural Languages System*. Addison-Wesley, 1986.

[Maier 88] Maier, D; Warren, D.S. —em Computing with Logic: Logic Programming with Prolog. The Benjamin/Cummings Publishing Company, 1988.

[Monard 88] Monard, M.C.; Lima Vidal, R. *Programas Prolog para Processamento de Listas*. ILTC, 1988, 52 pg.

[Monard 89] Monard, M.C.; Lima Vidal, R.; Nicoletti, M.C. *Programas Prolog para Processamento de Arvores*. ILTC, 1988, 74 pg.

[Monard 93] Monard, M.C.; Nicoletti, M.C. *Programas Prolog para Processamento de Listas e Aplicações*. Notas Didáticas do ICMSC Nº 7, 1993, 71 Pg.

[Rogers 86] Rogers, J.B. *A Prolog Primer*. Addison-Wesley, 1986.

[Rowe 88] Rowe, N.C. *Artificial Intelligence Through Prolog*. Prentice Hall, 1988.

[Scarpelli 84] Scarpelli, H.A.C. *Estrutura de Dados para Busca por Intervalo*. Dissertação de Mestrado, ICMSC-USP-São Carlos, 1984.

[Sterling 86] Sterling, L.; Shapiro, E. *The Art of Prolog*. The MIT Press, 1986.

[van Emden 81] van Emden, M.H. *A Runnable Specification of AVL-Tree Insertion*. Research Report CS-81-14, Department of Computer Science, Univer sity of Waterloo, Canada, 1981.

[Veloso 83] Veloso, P.; Santos, C.; Azeredo, P.; Furtado, A. *Estrutura de Dados*. Editora Campus, 1983.

[Walker 87] Walker, A.(Ed); McCord, M.; Souza, J.F.; Wilson, W.G. *Knowledge Systems and Prolog*. Addison-Wesley, 1987.

[Wirth 76] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice- Hall, 1976.

[Wirth 86] Wirth, N. *Algorithms and Data Structure*. Prentice-Hall, 1986.