ED – esquema para tutorial 04

Capítulo 4 – Árvores

[página de abertura]

Ao concluir este capítulo, você deverá ser capaz de:

- compreender o que são as árvores
- conhecer os algoritmos básicos que utilizam as árvores binárias
- representar árvores genéricas por meio de árvores binárias
- utilizar árvores binárias de pesquisa
- fazer balanceamento de árvores binárias de pesquisa

4.1 Conceitos

Uma árvore é uma estrutura de dados que se caracteriza por uma relação de hierarquia entre os elementos que a compõem. Exemplos de estruturas em forma de árvore são:

- o organograma de uma empresa;
- a divisão de um livro em capítulos, seções, tópicos;
- a árvore genealógica de uma pessoa.

De um modo um pouco mais formal, podemos dizer que uma árvore é um conjunto finito de um ou mais nós, tais que:

a) existe um nó denominado raiz;

b) os demais nós formam m>=0 conjuntos disjuntos s1, s2, ... sm, tais que cada um desses conjuntos também é uma árvore (denominada sub-árvore).

As árvores podem ser representadas de diversos modos, como por exemplo:

a) representação hierárquica

{inserir desenho tirado, ou do livro, ou do CBT}

Neste exemplo, **A** é a raiz da árvore. Os nós **B** e **C** são "filhos" de **A,** e dão origem a duas sub-árvores, nas quais **D** e **E** são "filhos" de **B,** e **F** é "filho" de **C**.

b) representação por conjuntos

{inserir desenho tirado, ou do livro, ou do CBT}

A árvore aqui representada é a mesma do desenho anterior. Neste caso, cada nó é representado por um conjunto formado por seus "filhos".

c) representação por expressão parentetizada

A mesma árvore pode ser representada pela expressão:

(A (B (D ( ) E ( )) C (F ( ))))

Cada conjunto de parêntesis correspondentes contém um nó e seus filhos. Se um nó não tem filhos, ele é seguido por um par de parêntesis sem conteúdo.

d) representação por expressão não parentetizada

Ainda utilizando a mesma árvore, podemos usar a representação abaixo:

A 2 B 2 D 0 E 0 C 1 F 0

Cada nó é seguido por um número que indica a quantidade de filhos desse nó, e em seguida por esses filhos, representados do mesmo modo.

Pode-se representar uma árvore de muitos outros modos, mas é interessante notar que, dentre os exemplos apresentados, a representação **a)** é a que permite uma melhor visualização, e que será utilizada a partir deste ponto. As representações **c)** e **d)** não permitem boa visualização da estrutura, mas podem ser úteis para guardar em arquivos os dados de uma árvore.

Como, por definição, os sub-conjuntos s1, s2, ... sm são disjuntos, cada nó só pode ter um "pai". Assim, o desenho abaixo, por exemplo, não representa uma árvore:

{inserir desenho tirado, ou do livro, ou do CBT}

Vejamos agora alguns termos utilizados para identificar algumas características de uma árvore ou de alguma parte dela.

{ver o esquema montado no CBT para este caso, tente fazer algo igual ou parecido}

- raiz – nó de origem da árvore
- folhas – nós que não têm filhos
- grau de um nó – número de filhos de cada nó
- nível de um nó – por definição, é zero para a raiz, e, para os demais nós, é o número de "linhas" que ligam o nó à raiz
- altura da árvore – é o nível mais alto da árvore
- floresta – conjunto de árvores disjuntas (eliminando-se a raiz de uma árvore, obtém-se uma floresta

4.2 Árvores Binárias

Árvores binárias são árvores em que todos os nós têm grau menor ou igual a dois. Desse modo, cada nó pode ter no máximo dois filhos, que são identificados como filho à esquerda e filho à direita.

4.2.1 Caminhamento em Árvores Binárias

Dada uma árvore binária, é possível estabelecer várias formas para percorrer todos os seus nós, sem repetir nenhum e sem deixar de passar por nenhum. Esta operação é denominada **caminhamento**, que é usada, por exemplo, para consultar ou alterar as informações contidas nos nós.

As três maneiras mais usuais para percorrer os nós são:

- caminhamento pré-fixado:
    1. visita a raiz
    2. percorre a sub-árvore da esquerda
    3. percorre a sub-árvore da direita

- caminhamento in-fixado:
    1. percorre a sub-árvore da esquerda
    2. visita a raiz
    3. percorre a sub-árvore da direita

- caminhamento pós-fixado:
    1. percorre a sub-árvore da esquerda
    2. percorre a sub-árvore da direita
    3. visita a raiz

Tomando como exemplo a árvore abaixo, temos, para cada tipo de caminhamento, a correspondente seqüência de nós:

{veja a animação do CBT, acho que podemos fazer uma animação como um applet}

caminhamento pré-fixado: A B D E G C F H I

caminhamento in-fixado; D B G E A C H F I

caminhamento pós-fixado: D G E B H I F C A

4.2.2 Implementação de Árvores Binárias

Na implementação das árvores binárias, utilizaremos para cada nó um registro contendo um campo para dados (*dado*), um ponteiro que aponta a sub-árvore da esquerda (*esq*) e um ponteiro que aponta a sub-árvore da direita (*dir*). O acesso à árvore é feito através de um ponteiro que aponta para a raiz (*ainicio*).

```
typedef struct nodo {
  char dado;
  struct nodo *esq, *dir;
} arvore ;

arvore *ainicio;
```

Pela própria definição, pode-se perceber que as árvores são estruturas adequadas para o uso de algoritmos recursivos. A própria **definição** formalizada acima para uma árvore é uma definição recursiva.

[
De um modo um pouco mais formal, podemos dizer que uma árvore é um conjunto finito de um ou mais nós, tais que:

a) existe um nó denominado raiz;

b) os demais nós formam m>=0 conjuntos disjuntos s1, s2, ... sm, tais que cada um desses conjuntos também é uma árvore (denominada sub-árvore).]

4.2.3 Algoritmos Básicos em Árvores Binárias

A maior parte dos algoritmos apresentados aqui são recursivos. Para efeitos didáticos, em alguns casos se apresenta também o algoritmo que realiza a operação equivalente, fazendo uso de uma pilha de ponteiros tipo *void*, isto é, ponteiros que podem apontar para qualquer elemento. No caso, são utilizados apontando para os nós da árvore. Os algoritmos recursivos, em geral, são mais simples.

Muitas vezes, o acesso à árvore pode se modificar durante a execução de uma função. Neste caso, será passado como parâmetro o endereço do ponteiro de acesso à árvore, denominado *eainicio*.

As definições dessas variáveis ficam sendo então:

```
arvore *ainicio;
arvore **eainicio;
```

Por exemplo:

Se uma função é definida como:

*void nome_da_funcao (arvore **eainicio)*

Então a chamada a essa função será efetuada da seguinte forma:

nome_da_funcao (&aini)

onde aini é um ponteiro para *arvore,* ou seja, fora da função foi definido que:

*arvore *aini;*

Abaixo são examinados alguns dos algoritmos básicos que utilizam árvores binárias:

a) Algoritmo de caminhamento em ordem pré-fixada utilizando pilha – versão 1

```
//--- definição do tipo de pilha de uso geral
typedef struct tipopilhageral {
  void *dado;
  struct tipopilhageral *prox;
} pilhageral;

//-------------------------------------------
void CaminhaPreComPilha1 (arvore *ainicio) {
  arvore *a1;
  pilhageral *p1;
  char c;

  a1 = ainicio;
  InicializaPilha (&p1);
  InserePilha (&p1, a1);
  while (!PilhaVazia (p1)) {
    a1 = RetiraPilha (&p1);
    if (a1 != NULL) {
      printf ("%c", a1->dado);
      InserePilha (&p1, a1->dir);
      InserePilha (&p1, a1->esq);
    }
  }
}
```

Tira Teima

Observação: Nesta versão, a função de caminhamento insere na pilha os ponteiros NULL encontrados no caminhamento. Consequentemente, ela verifica se ainicio é NULL em cada retirada da pilha.

b) Algoritmo de caminhamento em ordem pré-fixada utilizando pilha – versão 2

```
//--- definição do tipo de pilha de uso geral
typedef struct tipopilhageral {
  void *dado;
  struct tipopilhageral *prox;
} pilhageral;

//-------------------------------------------
void CaminhaPreComPilha2 (arvore *ainicio) {
  arvore *a1;
  pilhageral *p1;
  char c;

  a1 = ainicio;
  InicializaPilha (&p1);
```

```
  if (a1 != NULL)
    InserePilha (&p1, a1);
  while (!PilhaVazia (p1)) {
    a1 = RetiraPilha (&p1);
    printf ("%c", a1->dado);
    if (a1->dir != NULL)
      InserePilha (&p1, a1->dir);
    if (a1->esq != NULL)
      InserePilha (&p1, a1->esq);
  }
}
```

Tira Teima

Observação: Nesta versão, a função de caminhamento **não** insere na pilha os ponteiros NULL encontrados no caminhamento. Consequentemente, ela não verifica se ainicio é NULL em cada retirada da pilha. Na primeira inserção na pilha, deve ser feito um teste para verificar se a árvore é vazia Comparando-se as duas versões, tem-se que a versão 1 executa muito mais inserções e retiradas de elementos nas pilhas (aproximadamente metade dos ponteiros de uma árvore binária são NULL).

c) Algoritmo de caminhamento em ordem in-fixada utilizando pilha – versão 1

Para o caminhamento em ordem in-fixada, cada elemento da árvore entra na pilha duas vezes (acompanhe o funcionamento no TiraTeima). Para discernir se o elemento está entrando pela primeira ou pela segunda vez, criou-se um novo tipo de pilha, com um elemento **int**, que recebe o valor 0 quando a entrada é feita na primeira vez, e recebe o valor 1 quando a entrada é feita pela segunda vez. Para facilitar a compreensão, foi desenvolvida a função Empilha, interna à função de caminhamento, que procede à inserção na pilha.

A versão 1 processa o caso de ainicio ser igual a NULL.

```
void CaminhaInComPilha1 (arvore *ainicio) {

  pilhageral *pilha;
  void *paux1;
  struct reg {
    arvore *arv;
    int pri;
  };
  struct reg *paux2;
  arvore *a1;

//-------------------------------------
void Empilha (arvore *a, int n){
 paux2 = malloc (sizeof (struct reg));
 paux2->arv = a;
 paux2->pri = n;
 InserePilha (&pilha, paux2);
}
//-------------------------------------
```

Tira Teima

```
  a1 = ainicio;
  InicializaPilha(&pilha);
  Empilha(a1, 1);
  while (!PilhaVazia (pilha)) {
    paux2 = RetiraPilha (&pilha);
    a1 = paux2->arv;
    if (a1 != NULL) {
      if (!paux2->pri) printf ("%c", paux2->arv->dado);
      else {
        Empilha (a1->dir, 1);
        Empilha (a1, 0);
        Empilha (a1->esq, 1);
      }
    }
  }
}
```

d) Algoritmo de caminhamento em ordem in-fixada utilizando pilha – versão 2

Neste caso a pilha utilizada é igual à da versão 1. A função de caminhamento não insere nem retira da pilha os ponteiros iguais a NULL. Valem as mesmas observações feitas nos itens **a)** e **b).**

```
void CaminhaInComPilha2 (arvore *ainicio) {

  pilhageral *pilha;
  void *paux1;
  struct reg {
    arvore *arv;
    int pri;
  };
  struct reg *paux2;
  arvore *a1;

//-------------------------------------
void Empilha (arvore *a, int n){
 paux2 = malloc (sizeof (struct reg));
 paux2->arv = a;
 paux2->pri = n;
 InserePilha (&pilha, paux2);
}
//-------------------------------------

  a1 = ainicio;
  InicializaPilha(&pilha);
  if (a1 != NULL)
    Empilha(a1, 1);
  while (!PilhaVazia (pilha)) {
```

Tira Teima

```
   paux2 = RetiraPilha (&pilha);
   a1 = paux2->arv;
   if (!paux2->pri) printf ("%c", paux2->arv->dado);
   else {
    if (a1->dir != NULL)
      Empilha (a1->dir, 1);
    Empilha (a1, 0);
    if (a1->esq != NULL)
      Empilha (a1->esq, 1);
  }
 }
}
```

O caminhamento em ordem pós-fixada utilizando pilhas pode ser feito de forma muito parecida ao caminhamento em ordem in-fixada, utilizando a pilha com o campo **int**, que serve para indicar se o elemento está entrando da pilha pela primeira ou pela segunda vez.

e) Algoritmo recursivo de caminhamento em ordem pré-fixada – versão 1

Como se nota, o algoritmo recursivo é muito mais simples que aquele que utiliza pilha. Nesta versão, a função de caminhamento processa os casos em que ainicio vale NULL.

```
void CaminhaPre1(arvore *ainicio) {
 if (ainicio != NULL) {
   printf("%c", ainicio->dado);
   CaminhaPre1 (ainicio->esq);
   CaminhaPre1 (ainicio->dir);
 }
}

//-----------------------------------------
main ()
{
//-- construção da árvore
 CaminhaPre1 (aini);
}
```

Tira Teima

f) Algoritmo recursivo de caminhamento em ordem pré-fixada – versão 2

```
void CaminhaPre2(arvore *ainicio) {
 printf("%c", ainicio->dado);
 if (ainicio->esq != NULL)
   CaminhaPre2 (ainicio->esq);
 if (ainicio->dir != NULL)
   CaminhaPre2 (ainicio->dir);
}

//-----------------------------------------
main ()
```

Tira Teima

```
{
//-- construção da árvore
  if (aini != NULL) CaminhaPre1 (aini);
}
```

Nesta versão, a função de caminhamento **não** processa os casos em que ainicio vale NULL. Há muito menos chamadas recursivas, mas a função perde versatilidade: para ser chamada é necessário testar se a árvore é vazia, caso que a função não resolve.

Em todas as funções apresentadas até aqui, com duas versões, a escolhe entre uma delas é uma opção do programador. O importante é que a escolha seja respeitada e não haja mistura de lógicas dentro de cada função. A partir deste ponto, optaremos sempre por funções que resolvam o caso de o ponteiro ainicio ser NULL.

g) Algoritmo recursivo de caminhamento em ordem in-fixada

```
void CaminhaIn(arvore *ainicio) {
  if (ainicio != NULL) {
    CaminhaIn (ainicio->esq);
    printf("%c", ainicio->dado);
    CaminhaIn (ainicio->dir);
  }
}
```

Tira Teima

h) Algoritmo recursivo de caminhamento em ordem pós-fixada

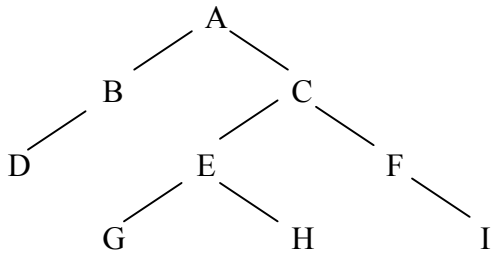
```
void CaminhaPos(arvore *ainicio) {
  if (ainicio != NULL) {
    CaminhaPos (ainicio->esq);
    CaminhaPos (ainicio->dir);
    printf("%c", ainicio->dado);
  }
}
```

Tira Teima

i) Algoritmo de construção de uma árvore

Esta função constrói uma árvore binária a partir dos dados contidos em um arquivo que contém os elementos da árvore em ordem pré-fixada. O nó de cada sub-árvore é seguido pela sua sub-árvore da esquerda e depois pela sua sub-árvore da direita. Quando o ponteiro *esq* ou *dir* de algum nó for NULL, no arquivo aparecerá um ponto. Abaixo se apresenta um exemplo de um arquivo com a árvore correspondente

A B D . . . C E G . . H . . F . I . .



A partir deste ponto, será adotada sempre esta notação para armazenar uma árvore em um arquivo, por ser mais prática.

```
void Constroi(arvore **eainicio) {
  char c;

  c = getc (arq);
  if (c == '.')
    *eainicio = NULL;
  else {
    *eainicio = malloc (sizeof (arvore));
    (*eainicio)->dado = c;
    Constroi (&((*eainicio)->esq));
    Constroi (&((*eainicio)->dir));
  }
}
```

Tira Teima

j) Algoritmo para ler uma árvore, gerando a seqüência dos elementos utilizada para armazenagem em arquivos

Esta função faz o inverso da anterior. A partir de uma árvore dada em memória, ela gera e escreve na tela a seqüência de caracteres e pontos que representam a árvore, de acordo com a notação vista no item anterior. Em lugar de escrever na tela, a função poderia gravar a seqüência em um arquivo, armazenando a árvore.

```
void LeArv(arvore *ainicio) {
  if (ainicio == NULL)
    printf (".");
  else {
    printf ("%c" , ainicio->dado);
    LeArv (ainicio->esq);
    LeArv (ainicio->dir);
  }
}
```

Tira Teima

k) Algoritmo para remover todos os elementos de uma árvore

A função de remoção dos nós de uma árvore apresentada aqui é recursiva e elimina toda a árvore. Como o ponteiro de acesso à árvore é passado por referência, depois da execução da função seu valor fica sendo NULL. Se o objetivo for remover apenas uma sub-árvore,

basta chamar a função passando como parâmetro o ponteiro que aponta para a raiz da sub-árvore que se deseja remover, como será visto adiante.

```
void Remove(arvore **eainicio) {
  if (*eainicio != NULL) {
    Remove (&((*eainicio)->esq));
    Remove (&((*eainicio)->dir));
    free (*eainicio);
    *eainicio = NULL;
  }
}
```

Tira Teima

l) Algoritmo para procurar um elemento em uma árvore

Os elementos estão distribuídos na árvore sem nenhuma estratégia (a organização dos elementos vai ser feita nas árvores binárias de pesquisa, no próximo tópico). Portanto, a tarefa de procura consiste em fazer um caminhamento pelos nós da árvore verificando se algum deles contém a chave procurada.

Na função abaixo, o valor retornado é um ponteiro que aponta para o nó da árvore que contém o elemento procurado. Se o elemento procurado não estiver presente, a função retorna NULL.

A forma de caminhamento da árvore é pré-fixada, mas com a ajuda da variável auxiliar *a1*. Se o elemento não está em um determinado nó, o algoritmo aciona a procura na sub-árvore da esquerda, devolvendo a resposta em *a1*. Somente se *a1* retorna NULL na primeira chamada recursiva (ou seja, o elemento não foi encontrado na sub-árvore da esquerda), é acionada a procura na sub-árvore da direita. Desse modo, se o elemento é encontrado, a procura se encerra, sem necessidade de se percorrer toda a árvore. Se o elemento não estiver presente, toda a árvore será percorrida e a função retorna NULL.

```
arvore *Procura(arvore *ainicio, char chave) {
  arvore *a1;
  a1 = NULL;
  if (ainicio != NULL) {
    if (ainicio->dado == chave)
      a1 = ainicio;
    if (a1 == NULL)
      a1 = Procura1 (ainicio->esq, chave);
    if (a1 == NULL)
      a1 = Procura1 (ainicio->dir, chave);
  }
  return a1;
}
```

Tira Teima

m) Algoritmo para procurar um elemento em uma árvore identificando o pai do elemento procurado

Esta função também retorna um ponteiro que aponta para o elemento encontrado, como a anterior. Além disso, o parâmetro *epai* sai da função guardando o endereço do nó pai do

elemento procurado. Se o elemento procurado é a raiz da árvore, ou se ele estiver ausente, o conteúdo do elemento apontado por *epai* é NULL.

```
arvore *ProcuraComAnt(arvore *ainicio, char chave, arvore **epai) {
  arvore *a1;
  a1 = NULL;
  if (ainicio != NULL) {
    if (ainicio->dado == chave)
      a1 = ainicio;
    if (a1 == NULL) {
      *epai = ainicio;
      a1 = ProcuraComAnt (ainicio->esq, chave, epai);
    }
    if (a1 == NULL){
      *epai = ainicio;
      a1 = ProcuraComAnt (ainicio->dir, chave, epai);
    }
  }
  if (a1 == NULL)
    *epai = NULL;
  return a1;
}
```


n) Algoritmo para remover uma sub-árvore

Esta função procura na árvore um nó com valor *chave* e, se encontrar, remove da árvore original a sub-árvore que tem como raiz o nó com a *chave*. Para manter a integridade da árvore que resta depois da remoção, esta função utiliza a procura com a localização do pai, vista no item anterior.

```
void RemoveSub(arvore **eainicio, char chave) {
  arvore *a1, *aant;
  aant = NULL;
  a1 = ProcuraComAnt (*eainicio, chave, &aant);
  if (a1 == *eainicio)
    *eainicio = NULL;
  else
    if (a1 != NULL)
      if (a1 == aant->esq)
        aant->esq = NULL;
      else
        if (a1 == aant->dir)
          aant->dir = NULL;
  Remove (&a1);
}
```


4.2.4 Exemplos de aplicação de árvores binárias

a) Determinação do número de vezes que um elemento dado está presente em uma árvore binária

```
int Ocorrencias (arvore *ainicio, int chave) {
  int ocdir, ocesq;

  if (ainicio == NULL)
    return 0;
  else {
    ocdir = Ocorrencias (ainicio->dir, chave);
    ocesq = Ocorrencias (ainicio->esq, chave);
    if (ainicio->dado == chave)
      return (1  + ocdir + ocesq);
    else
      return (ocdir + ocesq);
  }
}
```


b) Determinação da altura de uma árvore binária

```
int Altura (arvore *ainicio) {
  int adir, aesq;

  if (ainicio == NULL)
    return -1;
  else {
    adir = Altura (ainicio->dir);
    aesq = Altura (ainicio->esq);
    if (adir > aesq)
      return (1 + adir);
    else
      return (1 + aesq);
  }
}
```


c) Construção de uma árvore equivalente a outra árvore dada

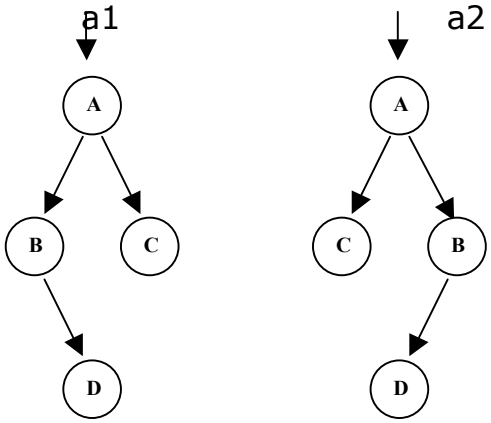
Uma árvore é dita equivalente a outra se ambas têm exatamente o mesmo número de nós, distribuídos da mesma maneira e com o mesmo conteúdo nos nós correspondentes.

```
arvore *ConstroiEquivalente(arvore *ainicio) {
  arvore *a1;

  if (ainicio == NULL)
    return NULL;
  else {
    a1 = malloc (sizeof (arvore));
    a1->dado = ainicio->dado;
    a1->esq = ConstroiEquivalente (ainicio->esq);
```

```
    a1->dir = ConstroiEquivalente (ainicio->dir);
    return a1;
  }
}
```

d) Construção de uma árvore espelhada em relação a outra árvore dada

Uma árvore é dita espelhada em relação a outra se ambas têm exatamente o mesmo número de nós, distribuídos de forma simetria, uma em relação a outra, com os conteúdos dos nós também distribuídos simetricamente. Como exemplo, as duas árvores desenhadas a seguir são espelhadas.




```
arvore *ConstroiEspelhada(arvore *ainicio) {
  arvore *a1;

  if (ainicio == NULL)
    return NULL;
  else {
    a1 = malloc (sizeof (arvore));
    a1->dado = ainicio->dado;
    a1->esq = ConstroiEspelhada (ainicio->dir);
    a1->dir = ConstroiEspelhada (ainicio->esq);
    return a1;
  }
}
```

Tira Teima

e) Verificação se duas árvores dadas são espelhadas – versão 1

```
int VerificaEspelhadas1(arvore *ainicio1, arvore *ainicio2) {
  int ok;
  char c1, c2;

  ok = 0;
  if ((ainicio1 == NULL) && (ainicio2 == NULL))
     ok = 1;
  else
```

Tira Teima

```
   if ((ainicio1 != NULL) && (ainicio2 != NULL) && (ainicio1->dado == ainicio2->dado)
&&
      (VerificaEspelhadas1 (ainicio1->esq, ainicio2->dir)) &&
      (VerificaEspelhadas1 (ainicio1->dir, ainicio2->esq)))
     ok = 1;
  return ok;
}
```

f) Verificação se duas árvores dadas são espelhadas – versão 2

A diferença entre esta versão e a anterior é que, nesta versão, todas as condições que correspondem a árvores espelhadas estão agrupadas em uma única expressão lógica.

```
int VerificaEspelhadas2(arvore *ainicio1, arvore *ainicio2) {
  int ok;
  char c1, c2;

  ok = 0;
  if (((ainicio1 == NULL) && (ainicio2 == NULL)) ||
    ((ainicio1 != NULL) && (ainicio2 != NULL) && (ainicio1->dado == ainicio2->dado) &&
    (VerificaEspelhadas2 (ainicio1->esq, ainicio2->dir)) &&
    (VerificaEspelhadas2 (ainicio1->dir, ainicio2->esq))))
      ok = 1;
  return ok;
}
```


g) Verificação de quais elementos de uma árvore estão presentes em outra

Neste caso duas funções recursivas são aninhadas. A função *ProcuraDuasArvores* percorre a primeira árvore e, para cada elemento, chama a função *Verifica*. A função *Verifica* percorre a segunda árvore até encontrar o elemento que está sendo procurado. Ela pára a procura se encontrar uma ocorrência.

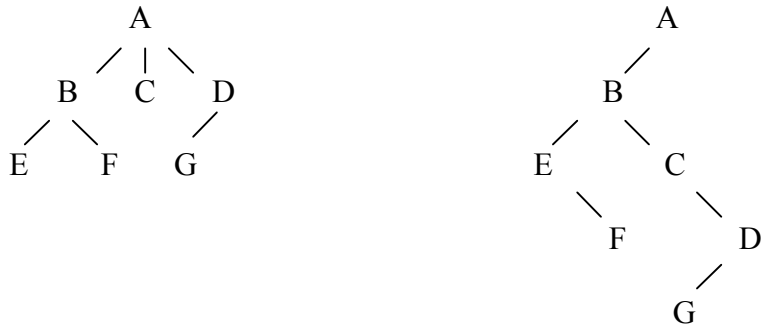
```
//-----------------------------------------
int Verifica(arvore *ainicio, char chave) {
  int achou;

  achou = 0;
  if (ainicio != NULL) {
    if (ainicio->dado == chave)
      achou = 1;
    if (!achou)
      achou = Verifica (ainicio->esq, chave);
    if (!achou)
      achou = Verifica (ainicio->dir, chave);
  }
  return achou;
}

//-----------------------------------------
```

```
void ProcuraDuasArvores(arvore *ainicio1, arvore *ainicio2) {
  if ((ainicio1 != NULL) && (ainicio2 != NULL)) {
    if (Verifica (ainicio2, ainicio1->dado))
      printf("achou: %3c \n", ainicio1->dado);
    ProcuraDuasArvores (ainicio1->esq, ainicio2);
    ProcuraDuasArvores (ainicio1->dir, ainicio2);
  }
}
```

4.3 Árvores Genéricas representadas como Árvores Binárias

Nas árvores genéricas (não binárias) cada nó pode ter qualquer quantidade de filhos. Este fato inviabiliza a implementação de uma árvore genérica da forma feita para árvores binárias, pois não se sabe quantos ponteiros seriam necessários em cada nó.

A melhor solução é implementar a árvore genérica através de uma árvore binária na qual os ponteiros da esquerda e da direita têm significados diferentes. Ao ponteiro da esquerda atribuímos o significado de "filho" e ao da direita, o significado de "irmão". Como exemplo, a árvore genérica da figura abaixo à esquerda fica representada como no desenho abaixo à direita.



A estrutura de dados para a implementação dessa árvore pode ser definida da seguinte forma:

```
typedef struct nodo {
  char dado;
  struct nodo *filho, *irmao;
} arvoreG ;
```

Os dados de uma árvore genérica podem ser guardados em arquivo de várias formas diversas. Para os exemplos que seguem, utilizaremos o seguinte padrão: cada linha do arquivo inicia com o dado de um nó, seguido de um número inteiro que indica quantos filhos esse nó tem, e em seguida a relação de seus. A primeira linha do arquivo contém os dados da raiz da árvore. As linhas seguintes contêm dados de nós que já apareceram anteriormente como filhos de algum outro nó.

A árvore utilizada como exemplo anteriormente ficaria armazenada em um arquivo da seguinte forma:

A

B

E C

F

```
      if (a1->filho == NULL) {
        a1->filho = malloc (sizeof (arvoreG));
        a2 = a1->filho;
      }
      else {
        a2->irmao = malloc (sizeof (arvoreG));
        a2 = a2->irmao;
      }
      a2->dado = c;
      a2->filho = NULL;
      a2->irmao = NULL;
      c = getc (arq);
    }
  }
 }
}
```

b) Leitura de uma árvore, gerando a seqüência dos elementos utilizada para armazenagem em arquivos

Esta função faz o inverso da anterior. A partir de uma árvore dada em memória, ela gera e escreve na tela a representação da árvore, de acordo com a notação vista no item anterior. Em lugar de escrever na tela, a função poderia gravar a seqüência em um arquivo, armazenando a árvore.

Observe que, como o algoritmo utilizado é recursivo, o arquivo montado não necessariamente tem a mesma forma do arquivo original, mas mantém as mesmas regras de construção. Dessa forma, se o algoritmo de construção for aplicado a este arquivo, a árvore construída será a mesma.

```
void LeG(arvoreG **eainicio) {
 arvoreG *a1;
 int n;

 if (*eainicio != NULL) {
   printf("%c %c", (*eainicio)->dado, ' ');
   a1 = (*eainicio)->filho;
   n = 0;
   while (a1 != NULL) {
     n = n + 1;
     a1 = a1->irmao;
   }
   printf ("%d", n);
   a1 = (*eainicio)->filho;
   while (a1 != NULL) {
     printf ("%c %c", ' ',a1->dado);
     a1 = a1->irmao;
   }
   printf ("\n");
```

```
  LeG (&((*eainicio)->irmao));
  LeG (&((*eainicio)->filho));
  }
}
```

c) Remoção de uma sub-árvore composta por um nó e seus filhos

Esta função remove da árvore um determinado nó e todos os seus descendentes (filhos, netos, etc.). Ela utiliza a função *ProcuraComAntG,* idêntica à função de procura que marca o nó antecessor do nó procurado, vista anteriormente (*ProcuraComAnt*).

```
void RemoveNoEFilhosG(arvoreG **eainicio, char dado) {
  arvoreG *a1, *a2;

  a2 = NULL;
  a1 = ProcuraComAntG (eainicio, dado, &a2);
  if (a1 == NULL)
    printf ("no inexistente\n");
  else {
    if (a1 == *eainicio)
      *eainicio = NULL;
    else {
      if (a2->filho == a1)
        a2->filho = a1->irmao;
      else
        a2->irmao = a1->irmao;
      a1->irmao = NULL;
    }
  RemoveG (&a1);
  }
}
```


d) Inserção de um nó em uma árvore genérica

A função *InsereG* faz a inserção de um nó na árvore genérica. Para isso, ela deve receber como parâmetros o nó a ser inserido e o nó que vai ser o pai do nó inserido.

```
void InsereG(arvoreG *ainicio, char dadonovo, char dadopai) {
  arvoreG *a1;

  a1 = ProcuraG (&ainicio, dadopai);
  if (a1 == NULL)
    printf ("pai nao encontrado\n");
  else {
    if (a1->filho == NULL) {
      a1->filho = malloc (sizeof (arvoreG));
      a1 = a1->filho;
    }
    else {
```

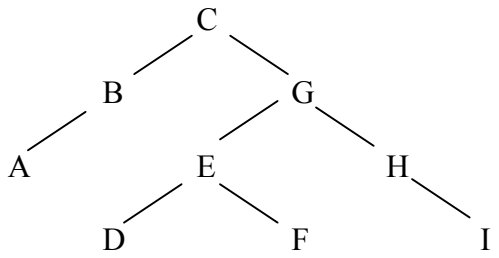
```
      a1 = a1->filho;
      while (a1->irmao != NULL)
        a1 = a1->irmao;
      a1->irmao = malloc (sizeof (arvoreG));
      a1 = a1->irmao;
    }
    a1->dado = dadonovo;
    a1->filho = NULL;
    a1->irmao = NULL;
  }
}
```

4.4 Árvores Binárias de Pesquisa

Árvores binárias de pesquisa são árvores binárias nas quais os dados são distribuídos de forma a facilitar a pesquisa de um determinado elemento.

4.4.1 Conceitos Básicos

Para tornar a pesquisa mais eficiente, na árvore binária de pesquisa adota-se a seguinte estratégia para a distribuição dos dados: em qualquer sub-árvore, todos os elementos situados à esquerda da raiz são menores que o elemento da raiz, e todos os situados à direita da raiz são maiores que o da raiz.



Com essa distribuição, a pesquisa de um elemento qualquer fica bem mais eficiente. Em qualquer nível, verifica-se a raiz: se o elemento for menor que o da raiz procura-se na sub-árvore da esquerda, se for maior, procura-se na sub-árvore da direita. Mas nunca é necessário procurar nas duas sub-árvores, como foi feito no algoritmo de procura em árvores binárias comuns, visto anteriormente. Para que esta estratégia seja eficiente é necessário que, em cada nível, a altura da sub-árvore da esquerda não seja muito diferente da altura da sub-árvore da direita. Mas este é um assunto que será tratado no próximo tópico, balanceamento de árvores binárias.

4.4.2 Implementação

Na implementação das árvores binárias de pesquisa utilizaremos a mesma estrutura de dados utilizada para árvores binárias comuns, vista anteriormente:

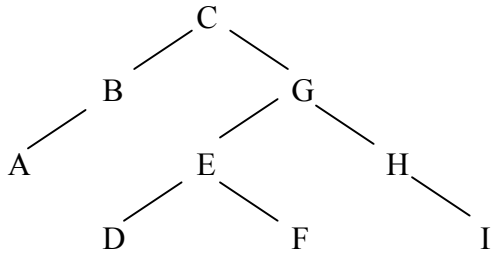
```
typedef struct nodo {
  char dado;
  struct nodo *esq, *dir;
} arvore ;
```

4.4.3 Algoritmos Básicos

Abaixo são examinados alguns dos algoritmos básicos que utilizam árvores binárias de pesquisa.

a) Construção de uma árvore binária de pesquisa

A construção de uma árvore binária de pesquisa pode ser feita utilizando o mesmo algoritmo visto para a construção de árvores binárias comuns, desde que o arquivo seja previamente preparado. Por exemplo, para a árvore binária utilizada como exemplo anteriormente, o arquivo deveria ser formado do seguinte modo:



C B A . . . G E D . . F . . H . I . .

```
void Constroi(arvore **eainicio) {
  char c;

  c = getc (arq);
  if (c == '.')
    *eainicio = NULL;
  else {
    *eainicio = malloc (sizeof (arvore));
    (*eainicio)->dado = c;
    Constroi (&((*eainicio)->esq));
    Constroi (&((*eainicio)->dir));
  }
}
```

Tira Teima

b) Procura de um elemento em uma árvore binária de pesquisa

Dada a estratégia de distribuição dos elementos na árvore binária de pesquisa, não é necessário que a procura de um elemento seja recursiva. O ponteiro local *a1* não percorre todos os ramos da árvore, mas desce diretamente pelo ramo onde o elemento pode estar. Se estiver presente, a função retorna um ponteiro apontando para ele. Se estiver ausente, a função retorna NULL.

```
arvore *Procura(arvore *ainicio, char chave) {
  arvore *a1;

  a1 = ainicio;
  while ((a1 != NULL) && (a1->dado != chave))
    if (a1->dado != chave)
      if (chave < a1->dado)
        a1 = a1->esq;
      else
        a1 = a1->dir;
  return a1;
}
```

Tira Teima

c) Inserção de um elemento em uma árvore binária de pesquisa

Esta função insere o elemento *dadonovo* na árvore, mantendo a disposição dos dados de forma que a árvore continue a ser binária de pesquisa.

Há uma procura pelo elemento *dadonovo* na árvore, através da função *ProcuraComAnt2*, que é uma nova versão da *ProcuraComAnt* vista anteriormente. Nesta versão, se o dado procurado estiver ausente, o ponteiro **eant* não é feito NULL, mas aponta para o nó que seria o pai do nó procurado, se ele estivesse presente. Ou seja, os ponteiros *a1* e *ant* vão descendo pela árvore, sem uso de recursividade, à procura do elemento *dadonovo.* Se *dadonovo* não está presente, a função retorna NULL, mas o ponteiro *ant* fica parado no último nó pesquisado, anterior ao local onde deve ser inserido o *dadonovo.*

```
arvore *ProcuraComAnt2(arvore **eainicio, char chave, arvore **eant) {
  arvore *a1;

  *eant = NULL;
  a1 = *eainicio;
  while ((a1 != NULL) && (a1->dado != chave))
    if (a1->dado != chave)
      if (chave < a1->dado) {
        *eant = a1;
        a1 = a1->esq;
      }
      else {
        *eant = a1;
        a1 = a1->dir;
      }
  return a1;
}

//-----------------------------------------
void Insere(arvore **eainicio, char dadonovo) {
  arvore *a1, *ant, *aux;

  ant = NULL;
  a1 = ProcuraComAnt2 (eainicio, dadonovo, &ant);
  if (a1 != NULL)
    printf ("dado ja existe\n");
  else {
    aux = malloc (sizeof (arvore));
    aux->dado = dadonovo;
    aux->esq = NULL;
    aux->dir = NULL;
    if (*eainicio == NULL)
      *eainicio = aux;
    else
    if (dadonovo < ant->dado)
      ant->esq = aux;
```

```
    else
      ant->dir = aux;
  }
}
```

d) Remoção de um elemento de uma árvore binária de pesquisa

A remoção de um nó de uma árvore binária de pesquisa deve ser efetuada com cuidado, para que os nós que permanecem na árvore continuem distribuídos de acordo com a mesma estratégia que permite a pesquisa binária. Para isso, devem ser analisados três casos:

a) O nó a ser retirado é uma folha, isto é, não tem filhos. Neste caso o nó é simplesmente retirado, como por exemplo, o nó G da figura abaixo.

b) O nó a ser retirado tem apenas um filho. Neste caso o nó é retirado e em seu lugar é colocada a raiz da sua única sub-árvore. Na figura abaixo, o nó retirado é o E, cujo lugar vai ser ocupado pelo nó C.

c) O nó a ser retirado tem dois filhos. Neste caso, não se retira o nó desejado, mas altera-se o seu conteúdo. Por exemplo, na figura, se o nó desejado é o B, altera-se o seu conteúdo para o nó seguinte na ordem, que é o C, e em seguida remove-se o nó que continha originalmente o valor C. Para determinar o nó seguinte a outro, caminha-se para a sub-árvore da direita do nó original, e em seguida desce-se sempre à esquerda, até encontrar o nó que não tenha filhos à esquerda.

{ANIMAÇÃO COM FIGURA "REMOCAO", PARA OS 3 CASOS – copiar o exemplo e a animação do CBT feito em Pascal}

A função *RetiraNo* utiliza a função *ProcuraComAnt2*, que retorna um ponteiro para o nó procurado e informa qual é o nó pai do nó procurado.

A função *RetiraNo* verifica se o nó procurado tem alguma sub-árvore vazia (essa situação corresponde aos casos a ou b vistos anteriormente, e são resolvidas diretamente). Caso o nó procurado tenha dois filhos, a função *RetiraNo* executa a operação prevista no caso c, e é usada recursivamente para eliminar o nó correspondente.

```
void RetiraNo (arvore **eainicio, char dado) {
  arvore *a1, *a2, *ant;
  char aux;

  ant = NULL;
  a1 = ProcuraComAnt2 (eainicio, dado, &ant);
  if (a1 == NULL)
    printf ("no inexistente\n");
  else
    if ((a1->dir == NULL) && (a1->esq == NULL))
      if (ant == NULL){
        free (a1);
        *eainicio = NULL;
```

Tira Teima
UTILIZAR O ALGORITMO COM A ARVORE ACIMA, PARA 4 CASOS: NOS G, E , C , e B

```
    }
    else {
      if (ant->dir == a1)
        ant->dir = NULL;
      else
        ant->esq = NULL;
      free (a1);
    }
  else
    if ((a1->dir == NULL) || (a1->esq == NULL)){
      if (a1 == *eainicio)
        if (a1->dir == NULL)
          *eainicio = a1->esq;
        else
          *eainicio = a1->dir;
      else
        if (a1->dir == NULL)
          if (ant->dir = a1)
            ant->dir = a1->esq;
          else
            ant->esq = a1->esq;
        else
          if (ant->dir = a1)
            ant->dir = a1->dir;
          else
            ant->esq = a1->dir;
      free (a1);
    }
    else {
      a2 = a1->dir;
      while (a2->esq != NULL)
        a2 = a2->esq;
      aux = a2->dado;
      RetiraNo (&((*eainicio)->dir), a2->dado);
      a1->dado = aux;
    }
}
```

4.5 Balanceamento de Árvores Binárias

As árvores binárias de pesquisa são construídas de modo a tornar mais eficiente a tarefa de procura de um elemento na árvore: em cada sub-árvore o elemento é procurado, ou à esquerda, ou à direita, mas não em ambos. Para que essa tarefa seja realmente eficiente é necessário que, em cada sub-árvore, o lado esquerdo tenha aproximadamente a mesma altura que o lado direito, ou seja, que a árvore esteja balanceada. Enquanto em uma lista o tempo de procura é proporcional a n (número de elementos), em uma árvore binária balanceada esse tempo cai para aproximadamente $\log_2 n$.

Formalmente, se diz que uma árvore está balanceada quando, para cada nó, a altura da sub-árvore da esquerda não difere da altura da sub-árvore da direita em mais de uma unidade. Abaixo apresentamos alguns exemplos de árvores balanceadas e de árvores não balanceadas.

{fazer aqui os exemplos do CBT, com as animações correspondentes}

Se uma árvore binária de pesquisa for utilizada apenas para a procura de um elemento, basta construí-la balanceada, usando adequadamente o arquivo em que estão armazenados os dados. Mas na maior parte dos casos, a árvore pode sofrer alterações durante seu uso, pela inserção ou retirada de elementos, que acabam alterando também o balanceamento da árvore. Para manter a árvore balanceada, podem-se adotar duas estratégias diferentes:

a) balanceamento estático: depois de certo tempo de uso, a árvore é destruída, guardando-se os dados em alguma outra estrutura, e depois reconstruída de forma balanceada;

b) balanceamento dinâmico: cada vez que um elemento é retirado ou inserido na árvore, verifica-se se houve desbalanceamento, e neste caso alteram-se as posições de alguns nós, de modo a restabelecer o balanceamento.

4.5.1 Balanceamento Estático

O balanceamento estático consiste em, depois de certo tempo de uso, destruir a sua estrutura, guardando as informações em alguma outra estrutura, e depois reconstruí-la de forma balanceada. Veremos aqui dois algoritmos:

a) o primeiro percorre a árvore, armazenando os dados em um vetor. Em vez de vetor, na prática pode-se usar uma estrutura dinâmica como uma lista, para que a solução seja mais genérica, ou um arquivo, se a quantidade de dados é grande. Escolhemos aqui um vetor para dar mais clareza à essência do método. Nosso algoritmo apenas constrói o vetor: a destruição da árvore pode ser feita como já foi visto em algoritmos anteriores.

A função abaixo percorre a árvore em ordem infixada e gera o vetor v, que contém em cada elemento uma variável tipo char. O índice i, que percorre o vetor, deve ser inicializado fora da função, com o valor 0.

```
void GeraVetor(arvore **eainicio) {
  if (*eainicio != NULL) {
    GeraVetor (&((*eainicio)->esq));
    i = i + 1;
```

Tira Teima

```
    v [i] = (*eainicio)->dado;
    GeraVetor (&((*eainicio)->dir));
  }
}
```

b)o segundo algoritmo consiste em reconstruir a árvore a partir do vetor originado pelo primeiro algoritmo. A função *GeraArvore* reconstrói a árvore a partir do vetor gerado pela função anterior. O nó situado na posição média do vetor é feito raiz da árvore e, em seguida, cada uma das metades do vetor recebe tratamento análogo, recursivamente. Na primeira chamada da função, os valores de *pos1* e *pos2* correspondem aos índices da primeira e da última posição do vetor.

```
void GeraArvore(arvore **eainicio, int pos1, int pos2) {
  int posm;

  if (pos1 > pos2)
    *eainicio = NULL;
  else {
    posm = (pos1 + pos2) / 2;
    *eainicio = malloc (sizeof (arvore));
    (*eainicio)->dado = v [posm];
    GeraArvore ((&((*eainicio)->esq)), pos1, posm - 1);
    GeraArvore ((&((*eainicio)->dir)), posm + 1, pos2);
  }
}
```

Tira Teima

4.5.2 Balanceamento Dinâmico

O balanceamento dinâmico consiste em reajustar os nós de uma árvore sempre que uma inserção ou remoção provoque seu desbalanceamento. Para exemplificar essa estratégia usaremos a árvore AVL (Adelson-Velskii e Landis).

Para cada nó, define-se um fator de balanceamento (fatbal), que é um número inteiro que pode assumir os seguintes valores, para uma árvore balanceada:

fatbal = -1 quando a sub-árvore da esquerda é uma unidade mais alta que a da direita
fatbal = 0 quando as duas sub-árvores têm a mesma altura
fatbal = 1 quando a sub-árvore da direita é uma unidade mais alta que a da esquerda

Se o fatbal sair do intervalo (-1, + 1), isto significa que a árvore ficou desbalanceada, e deve ser "regulada" (balanceada novamente).

{ colocar aqui a figura do livro – ou do CBT – com o fatbal de cada nó}

Será analisado agora o processo de regulagem de uma árvore que ficou desbalanceada pela inserção ou remoção de um novo nó. Há dois nós especialmente importantes nesse processo:

P – é o nó ancestral mais próximo do nó inserido, cujo fatbal fica fora do intervalo (-1,+1) depois da inserção

Q – é o filho de P na sub-árvore onde ocorreu a inserção

Se o desbalanceamento for provocado pela remoção de um nó, a única diferença é a definição do nó Q:

Q – é o filho de P na sub-árvore oposta a aquela onde ocorreu a inserção

A regulagem pode se dar por uma das quatro seguintes operações:

a) rotação simples à esquerda
b) rotação simples à direita
c) rotação dupla à esquerda
d) rotação dupla à direita

Nos esquemas abaixo, P é o nó que contem o valor X, e Q é o nó que contém o valor Y. Os triângulos numerados representam sub-árvores que eventualmente podem estar ligadas aos nós P e Q.

Rotação simples à esquerda

{inserir animação parecida com a do CBT}

Neste caso o nó Q sobe para a posição anteriormente ocupada pelo nó P. A sub-árvore 2, que estava à esquerda de Q e continha todos os elementos imediatamente menores que Y, passa a ocupar a posição à direita de P (contém todos os elementos imediatamente maiores que X, e desse modo a ordem dos elementos não se altera).

Rotação simples à esquerda

{inserir animação parecida com a do CBT}

Neste caso o nó também o nó Q sobe para a posição anteriormente ocupada pelo nó P. A sub-árvore 2, que estava à direita de Q e continha todos os elementos imediatamente maiores que Y, passa a ocupar a posição à esquerda de P (contém todos os elementos imediatamente maiores que X, e desse modo a ordem dos elementos não se altera).

Rotação dupla à esquerda

{inserir animação parecida com a do CBT}

A rotação dupla à esquerda é constituída por duas rotações simples: a primeira, à direita, em torno do nó Q, e segunda, à esquerda, em torno do nó P.

Rotação dupla à direita

{inserir animação parecida com a do CBT}

A rotação dupla à direita é constituída por duas rotações simples: a primeira, à esquerda, em torno do nó Q, e segunda, à direita, em torno do nó P.

Os critérios para a escolha da rotação a ser utilizada são:

a) se o fatbal do nó P for positivo, a rotação deve ser feita para a esquerda, se for negativo, para a direita;

b) se o fatbal de P e o fatbal de Q tiverem o mesmo sinal, a rotação deve ser simples, se tiverem sinais contrários, a rotação deve ser dupla.

Como exemplo, consideremos uma árvore que vai sendo construída através da inserção de uma série de elementos:

{inserir animação parecida com a do CBT}

Apresentamos a seguir o algoritmo completo para inserir um elemento novo em uma árvore AVL. O algoritmo para retirar um elemento pode ser construído de forma análoga. O tipo árvore continua sendo o mesmo utilizado até agora, contando apenas com mais um campo em cada nó, que é o *fatbal* (fator de balanceamento, inteiro).

```
typedef struct nodo {
  char dado;
  int fatbal;
  struct nodo *esq, *dir;
} arvoreAVL;
```

Outra modificação na estrutura da árvore é a criação de um nó descritor (conteúdo não utilizado), cujo ponteiro *dir* aponta para a raiz da árvore. O nó descritor facilita a função dos ponteiros rastreadores, que são ponteiros que seguem o rastro dos ponteiros que percorrem a árvore. Por exemplo, *paux* é um ponteiro que percorre a árvore procurando a posição do nó a ser inserido; *pant* é seu rastreador: guarda sempre a posição do pai de *paux* na árvore. Os ponteiros *pP* e *pQ* ficarão apontando para os nós P e Q vistos anteriormente, que são os pivôs das rotações. O ponteiro *pantP* é o rastreador de *pP*. A inicialização da árvore AVL é feita pela função *InicializaAVL*.

```
void InicializaAVL (arvoreAVL **eainicio){
  *eainicio = malloc (sizeof (arvoreAVL));
  (*eainicio)->esq = NULL;
  (*eainicio)->dir = NULL;
  }
```

A variável *poschave* serve para indicar o lado em que houve a inserção (+1 para a direita e –1 para a esquerda).

A primeira parte da função *InsereAVL* verifica se a chave já está presente na árvore, através da variável lógica *achou*. Se não estiver presente, insere o nó na posição correspondente, de forma análoga à inserção vista em outros tópicos. A novidade nesta parte do algoritmo é o posicionamento do ponteiro *pP*, que deve ficar apontando para o ancestral mais próximo do nó inserido que já tivesse fatbal <> 0 antes da inserção. O rastreador *pantP* também é posicionado nesta fase.

Depois de inserido o novo nó, e posicionados *pP* e *pantP*, a função *InsereAVL* chama as funções *AjustaFatoresAVL* e *BalanceiaAVL*, que serão examinados separadamente.

```
void InsereAVL (arvoreAVL *adesc, char dadonovo) {
  arvoreAVL *paux, *pant, *pP, *pQ, *pantP, *pnovo;
  int poschave;
  int achou;

  paux = adesc->dir;
  pP = paux;
  pant = adesc;
  pantP = adesc;
  achou = 0;
  while ((!achou) && (paux != NULL)) {
    if (paux->fatbal != 0) {
      pP = paux;
      pantP = pant;
    }
    pant = paux;
    if (dadonovo == paux->dado)
      achou = 1;
    else
      if (dadonovo < paux->dado)
        paux = paux->esq;
      else
        paux = paux->dir;
  }
  if (achou)
    printf("este dado ja esta presente na arvore\n");
  else {
    pnovo = malloc (sizeof (arvoreAVL));
    pnovo->dado = dadonovo;
    pnovo->esq = NULL;
    pnovo->dir = NULL;
    pnovo->fatbal = 0;
    if (adesc->dir == NULL)
      adesc->dir = pnovo;
    else {
      if (dadonovo < pant->dado)
        pant->esq = pnovo;
      else
        pant->dir = pnovo;
      AjustaFatoresAVL ();
      BalanceiaAVL ();
    }
  }
}
```

A função *AjustaFatoresAVL* inicialmente posiciona *pQ* (aponta para o filho do nó P na direção da inserção) e em seguida ajusta os fatbal's dos nós situados entre *pQ* e *pant*. O nó P é o ancestral mais próximo do novo nó, com fatbal <> 0 antes da inserção, ou seja, depois da inserção ficaria com fatbal fora do intervalo permitido. Com a correção da rotação, o fatbal de P volta ao valor anterior. O mesmo acontece com os nós antecessores de P cujo fatbal tivesse sido alterado pela inserção. Assim sendo, não é necessário ajustar os fatbal's desses nós, já que a rotação fará com que automaticamente voltem aos valores anteriores. Por esse motivo o ajuste do fatbal só é efetuado entre Q e o nó apontado por *pant*.

```
void AjustaFatoresAVL (){
  if (dadonovo < pP->dado){
    pQ = pP->esq;
    paux = pP->esq;
  }
  else {
    pQ = pP->dir;
    paux = pP->dir;
  }
  while (paux->dado != dadonovo)
    if (dadonovo < paux->dado) {
      paux->fatbal = paux->fatbal - 1;
      paux = paux->esq;
    }
    else {
      paux->fatbal = paux->fatbal + 1;
      paux = paux->dir;
    }
}
```

A função *BalanceiaAVL* identifica o lado em que houve inserção, e atualiza a variável poschave (+1 para direita, -1 para esquerda). A seguir verifica o *fatbal* de P antes da inserção, podendo ocorrer três situações:

a) fatbal = 0 : significa que não havia nó ancestral do inserido, com *fatbal* <> 0 antes da inserção, e portanto a árvore não desbalanceou. Nesse caso deve-se corrigir o *fatbal* de P, que passa a ter o mesmo valor de *poschave*;

b) fatbal = - poschave : significa que P estava pendendo para o lado contrário ao lado da inserção, e portanto a inserção corrigiu essa inclinação. Nesse caso o *fatbal* de P deve ser feito igual a zero;

c) fatbal = poschave : significa que P estava pendendo para o mesmo lado da inserção, e portanto a inserção desbalanceou a árvore. Nesse caso deve-se aplicar uma rotação para corrigir a árvore. Se Q pende para o mesmo lado de P, a rotação é simples, se Q pende para o lado contrário ao de P, a rotação deve ser dupla. As funções *RotacaoSimples* e *RotacaoDupla* são vistas separadamente.

Ainda dentro da *BalanceiaAVL* é feito o ajuste do ponteiro que aponta inicialmente para P, e que deve, depois da rotação, apontar para a raiz da sub-árvore resultante da rotação.

```
void BalanceiaAVL (){
  if (dadonovo < pP->dado)
    poschave = -1;
```

```
  else
    poschave = 1;
  if (pP->fatbal == 0)
    pP->fatbal = poschave;
  else
    if (pP->fatbal == -poschave)
      pP->fatbal = 0;
    else {
      if (pQ->fatbal * poschave > 0)
        RotacaoSimples ();
      else
        RotacaoDupla ();
      if (pantP->dir == pP)
        pantP->dir = paux;
      else
        pantP->esq = paux;
    }
}
```

A função *RotacaoSimples* executa a rotação simples, para a direita ou para a esquerda, de acordo com o critério visto anteriormente. No final da rotação ela ajusta os *fatbal*'s dos nós P e Q.

```
void RotacaoSimples (){
  if (pP->fatbal == 1) {
    pP->dir = pQ->esq;
    pQ->esq = pP;
  }
  else {
    pP->esq = pQ->dir;
    pQ->dir = pP;
  }
  paux = pQ;
  pP->fatbal = 0;
  pQ->fatbal = 0;
}
```

A função *RotacaoDupla* executa a rotação dupla, para a direita ou para a esquerda, de acordo com o critério visto anteriormente. No final da rotação ela ajusta os *fatbal*'s dos nós P e Q. Para este ajuste é necessário levar em conta 3 possibilidades para o valor de *fatbal* do nó que fica sendo a raiz da sub-árvore depois da rotação: pode ser igual a + *poschave*, igual a zero, ou igual a – *poschave*.

```
void RotacaoDupla (){
  if (pP->fatbal == 1) {
    paux = pQ->esq;
    pQ->esq = paux->dir;
    paux->dir = pQ;
    pP->dir = paux->esq;
    paux->esq = pP;
  }
  else {
    paux = pQ->dir;
    pQ->dir = paux->esq;
    paux->esq = pQ;
    pP->esq = paux->dir;
    paux->dir = pP;
  }
```

```
  if (paux->fatbal == -poschave){
    pP->fatbal = 0;
    pQ->fatbal = poschave;
  }
  else
    if (paux->fatbal == 0) {
      pP->fatbal = 0;
      pQ->fatbal = 0;
    }
    else {
      pP->fatbal = -poschave;
      pQ->fatbal = 0;
    }
  paux->fatbal = 0;
}
```