UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Bauru

Faculdade de Ciências

Disciplina: Estruturas de Dados I – **ED1**
Professora: Simone das Graças Domingues Prado
e-mail: simone.prado@unesp.br

Apostila 03 - Árvores Genéricas e Binárias

Objetivos:

⇒ Trabalhar com árvores estáticas
⇒ Trabalhar com árvores dinâmicas
⇒ Trabalhar com árvores genéricas
⇒ Trabalhar com árvores binárias
⇒ Trabalhar com árvores binárias de busca

Conteúdo:

Departamento de Computação
Av. Engº Luiz Edmundo Carrijo Coube 14-01 Vargem Limpa Bauru-SP CEP: 17033-360
Fone: (14) 3103-6034 Fax: (14) 3103-6079 email: dcogeral@fc.unesp.br

1. Introdução

Deve-se inicialmente definir:

a) Árvore:
- É uma lista na qual cada elemento possui dois ou mais sucessores, porém, todos os elementos possuem apenas um antecessor (Forbellone & Eberspacher, 2000, p167)
- Uma estrutura de árvore, como tipo básico T, pode ser definida recursivamente conforme uma das duas situações abaixo:

  (1) A estrutura vazia;

  (2) Um nó do tipo T, associado a um número finito de estruturas disjuntas de árvores de mesmo tipo base T, denominada subárvores (Wirth, 1986, p165)
- Uma árvore enraizada T, ou simplesmente árvore, é um conjunto finito de elementos denominados nós ou vértices tais que:

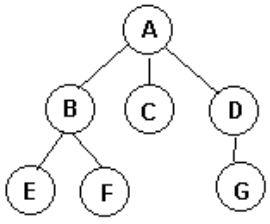
  (1) T = 0, e a árvore é dita vazia, ou

  (2) existe um nó especial, r, chamado raiz de T; os restantes constituem um único conjunto vazio ou são divididos em m >=1 conjuntos disjuntos não vazios, as subárvores de r, ou simplesmente subárvores, cada qual por sua vez uma árvore (Szwarcfiter & Markenzo, 1994, p.62)

b) **Raiz**: primeiro elemento, que dá origem aos demais.
c) **Nó**: qualquer elemento da árvore
d) **Altura de uma árvore**: quantidade de níveis a partir do nó-raiz até o nó mais distante (a raiz está no nível zero)
e) **Grau de uma árvore**: número máximo de ramificações da árvore
f) **Grau de um nó**: número máximo de ramificações a partir desse nó
g) **Filho**: sucessor de um determinado nó
h) **Pai**: único antecessor de um dado elemento
i) **Folha**: elemento final (sem descendentes, sem filhos)
j) **Nível de um nó**: é distância que um nó tem em relação ao nó raiz (a raiz está no nível 0)

Exemplo: Seja a árvore da Figura 01

Figura 01. Um exemplo de árvore



raiz: A
nós: A, B, C, D, E, F, G.
altura da árvore: 2
grau da árvore: 3
folhas: E, F e G

grau do nó-B: 2
filhos do nó-B: E e F
pai do nó-B: A
nível do nó-B: 1

grau do nó-C: 0
filhos do nó-C: 0
pai do nó-C: A
nível do nó-C: 1

grau do nó-D: 1
filhos do nó-D: G
pai do nó-D: A
nível do nó-D: 1

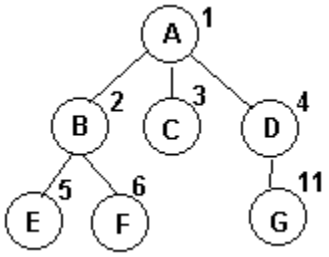
2. Representações:

Pode-se implementar uma árvore usando estruturas estáticas ou dinâmicas.

2.1. Representação Estática

Primeira representação: Cada nó é enumerado a partir da raiz (raiz = 1), considerando o grau da árvore. Assim, a árvore da Figura 1 fica com a enumeração da Figura 2.

Figura 2. A árvore com seus nós enumerados



Note que é prevista a possibilidade de cada nó ter três filhos, obedecendo o grau da árvore.

A construção do vetor pode ser feita como:

```
typedef tipo_no def_arvore [Max];
```

onde o valor do nó fica armazenado na sua posição enumerada a partir da raiz.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
A	B	C	D	E	F					G		

```
void le_arvore (def_arvore arvore){
   int posicao=1,i,pai=0,j,qt;
   char filho, tem_filhos;

   printf("Raiz: ");arvore[pai]=getche();
   for (j=1; j<Max; j++){
       if (arvore[pai]!= '-'){
           printf("\n\nO no %c tem filhos(s/n)?", arvore[pai]);
           tem_filhos=getche();
           if(tem_filhos =='s' || tem_filhos == 'S'){
              printf("\nLeitura dos filhos de: %c ", arvore[pai]);
              printf("\nQuantos filhos: "); qt=getche()-48;
              for(i=1; i<=qt; i++){
                  printf("\nFilho = ");filho=getche();
```

```
                    if (filho!='0')arvore[posicao]=filho;
                    posicao++;}
                  if(Grau>qt) posicao+=(Grau-qt);
                  }
               else posicao+=3;
               }
         pai++;}
}
```

Segunda representação: são armazenados o valor do nó e o número de filhos que ele possui.

Pode-se construir a árvore da seguinte forma:

```
struct elemento{
            tipo_no valor;
            int nro_filhos;}
typedef struct elemento def_arvore [Max]
```

Olhando a Figura 1 tem-se:

A	3	B	2	C	0	D	1	E	0	F	1	G	0

```
void le_arvore(def_arvore arvore){
   int posicao=1,i,pai=0,qt;
   char filho, tem_filhos;

   printf("Raiz: ");arvore[pai].info=getche();
   while(arvore[pai].info!='-'){
         printf("\n\nO no %c tem filhos(s/n)?",arvore[pai].info);
         tem_filhos=getche();
         if(tem_filhos =='s' || tem_filhos ='S'){
            printf("\nLeitura dos filhos de: %c", arvore[pai].info);
            printf("\nQuantos filhos: ");qt=getche()-48;
            arvore[pai].nro_filhos = qt;
            for(i=0; i<qt; i++){
                printf("\nFilho = ");filho=getche();
                if (filho!='0')arvore[posicao].info=filho;
                posicao++;}
              }
           else arvore[pai].nro_filhos=0;
         pai++;}
}
```

Terceira representação: são armazenados o valor do nó e a posição dos filhos dentro da estrutura.

Pode-se construir a árvore como:

```
typedef int vetor_filhos[3]; // Grau=3
struct elemento{
            tipo_no valor;
            vetor_filhos filhos;}
typedef struct elemento def_arvore [Max]
```

Olhando a Figura 1 tem-se:

0				1				2				3				4			
A	1	2	3	B	4	5	0	C	0	0	0	D	6	0	0	E	0	0	0

5				6			
F	0	0	0	G	0	0	0

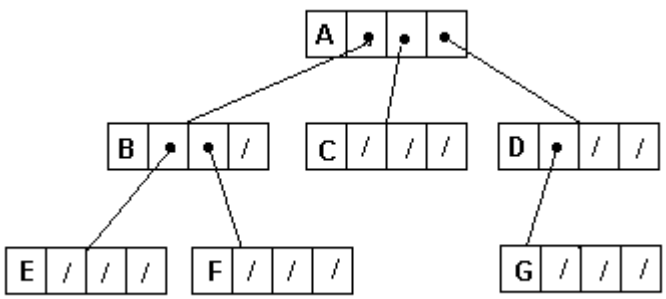
```
void le_arvore(def_arvore arvore){
   int posicao=1,i,j,pai=0,qt;
   char filho, tem_filhos;

   printf("Raiz: ");arvore[pai].info=getche();
   for(j=1;j<Max;j++){
       if(arvore[pai].info!='-'){
         printf("\n\nO no %c tem filhos(s/n)?",arvore[pai].info);
         tem_filhos=getche();
         if(tem_filhos =='s' || tem_filhos =='S'){
            printf("\nLeitura dos filhos de: %c", arvore[pai].info);
            printf("\nQuantos filhos: ");qt=getche()-48;
            for(i=0; i<qt; i++){
                printf("\nFilho = ");filho=getche();
                if (filho!='0'){
                    arvore[posicao].info=filho;
                    arvore[pai].filhos[i]=posicao;}
                posicao++;}
            }}
       pai++;}
}
```

2.2. Representação dinâmica:

A representação dinâmica usa ponteiros para a implementação. Veja uma configuração para uma árvore de grau 3:

Figura 3. Uma representação de implementação de uma árvore



```
typedef struct no{
        tipo_no info;
        struct no* primeiro;
        struct no* segundo;
        struct no* terceiro;
        } *def_arvore;
```

Outras representações podem ser usadas para a implementação. Veja as definições abaixo:

(2) estrutura com um vetor de filhos

```
typedef struct no_arvore {
        tipo_no info;
        struct no_arvore* filhos[Grau_Max];
        } *def_arvore;
```

(3) uso de lista de filhos e irmãos

```
typedef struct no_arvore {
        tipo_no info;
        struct no_arvore* primeiro_filho;
        struct no_arvore* irmão;
        } *def_arvore;
```

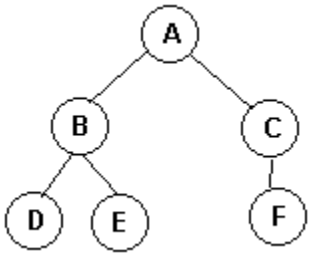
3. Árvores binárias

Definição segundo Tenenbaum et al.(1994, p.303):

> " Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos disjuntos. O primeiro subconjunto contém um único elemento, chamado raiz da árvore. Os outros dois subconjuntos são em si mesmos árvores binárias, chamadas subárvores esquerda e direita da árvore original. Uma subárvore esquerda ou direita pode estar vazia. Cada elemento de uma árvore binária é chamado nó da árvore."

Assim, uma árvore binária é aquela que possui no máximo grau 2 e onde se tem a idéia de filhos à esquerda e filhos à direita de um nó. Veja um exemplo de árvore binária na Figura 4 e ao lado uma definição da estrutura.

Figura 4. Árvore Binária

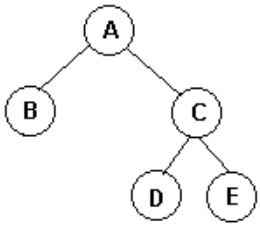



```
typedef struct no{
        tipo_no info;
        struct no* esq;
        struct no* dir;
       } *def_arvore;
```

Outras definições:

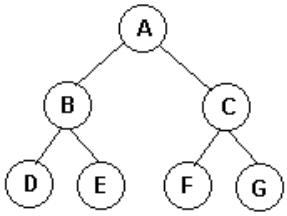
- **Árvore estritamente binária**: se todo nó que não for folha tiver subárvores esquerda e direita não vazias. Veja figura 5.

Figura 5. Uma árvore estritamente binária



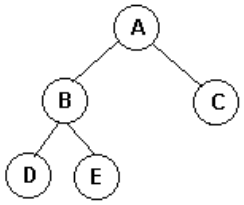
- **Árvore binária completa de profundidade d** é a árvore estritamente binária em que todas as folhas estejam no nível d. Veja figura 6.

Figura 6. Uma árvore binária completa de profundidade 2



- Árvore binária quase completa de profundidade d é uma árvore binária de profundidade d se (1) cada folha na árvore estiver no nível d ou no nível d-1. (2) para cada nó nd na árvore com descendente direito no nível d, todos os descendentes esquerdos de nd que forem folhas estiverem também no nível d. Veja figura 7.

Figura 7. Uma árvore binária quase completa de profundidade 2



3.1 Inserção em Árvore Binária

Na inserção de elementos numa árvore binária é necessário saber se o filho de um nó é à esquerda ou à direita da raiz e fazer a inserção.

3.2. Percurso em Árvore Binária

Uma operação comum em árvores binárias é percorrer cada um de seus nós uma única vez. Pode-se simplesmente imprimir o conteúdo de cada nó ao passar por ele, ou processá-lo de alguma maneira.

Para percorrer a árvore binária, têm-se três possibilidades:

- Pré-ordem
- Em-ordem ou ordem simétrica
- Pós-ordem

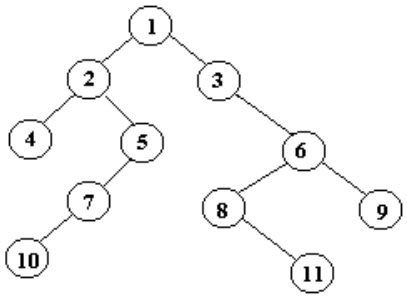
Seja uma árvore binária com três nós: raiz (R), nó à esquerda (E) e nó à direita (D). Então, nos percursos citados acima, tem-se:

- pré-ordem : R, E, D (visita a raiz, a subárvore esquerda e por fim a subárvore direita)
- em-ordem: E, R, D (visita a subárvore esquerda, a raiz e por fim à direita)
- pós-ordem: E, D, R (visita a subárvore esquerda depois à direita e por fim a raiz)

Olhando a árvore da Figura 8, tem-se:

- pré-ordem: 1, 2, 4, 5, 7, 10, 3, 6, 8, 11, 9
- em-ordem: 4, 2, 10, 7, 5, 1, 3, 8, 11, 6, 9
- pós-ordem: 4, 10, 7, 5, 2, 11, 8, 9, 6, 3, 1

Figura 8. Uma árvore binária



3.2.1. Algoritmos recursivos

Pré-ordem:

```
void pre_ordem(def_arvore arvore){
   if (arvore != NULL){
      printf("%d  ",arvore->info);
      pre_ordem(arvore->esq);
      pre_ordem(arvore->dir);}}
```

Em-ordem:

```
void em_ordem(def_arvore arvore){
   if (arvore != NULL){
      em_ordem(arvore->esq);
      printf("%d  ",arvore->info);
      em_ordem(arvore->dir);}}
```

Pós-ordem:

```
void pos_ordem(def_arvore arvore){
   if (arvore != NULL){
      pos_ordem(arvore->esq);
      pos_ordem(arvore->dir);}
      printf("%d  ",arvore->info);}
```

3.2.2. Algoritmos não recursivos

Podemos fazer os percursos sem usar recursão. Para isso usam-se estruturas de pilhas ou filas para percorrer a árvore visitando o nó uma única vez..

A rotina abaixo faz o percurso em-ordem fazendo uso de uma pilha.

```
void mostra_em_ordem (def_arvore arvore){
   def_arvore p=arvore;
   def_pilha pilha=NULL;
   do{
      while (p!=NULL){
         empilha(&pilha,p); p=p->esq;}
      if (!vazia(pilha)){
        desempilha(&pilha,&p); printf("\t%d",p->info);
        p=p->dir;}
   }while (!vazia(pilha) || p!=NULL);
}
```

3.3. Busca em Árvore Binária

Como a árvore não possui nenhum tipo de ordenação, não há como otimizar o processo. Então temos de percorrer todos os elementos para encontrar o que procuramos.

4. Árvore Binária de Busca

Uma árvore binária é dita de busca se suportar operações eficientes de busca, inserção e remoção. Para ser eficiente deve-se ter um critério de ordenação de dados. Assim, uma árvore binária de busca poderá ter como critério que a partir de um nó R os elementos de sua subárvore esquerda tem somente elementos menores que R e a sua subárvore direita tem somente elementos maiores que R. A definição da estrutura da árvore binária não se altera.

4.1. Inserção em Árvore Binária de Busca

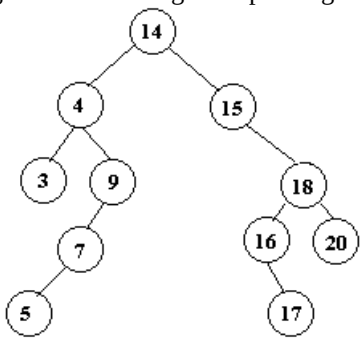
Veja a rotina:

```
void insere_arvore(def_arvore *arvore, int valor){
    def_arvore p;

    if (*arvore!=NULL){
       if ((*arvore)->info > valor) insere_arvore(&((*arvore)->esq),valor);
       else if ((*arvore)->info < valor) insere_arvore(&((*arvore)->dir),valor);
       else printf("O numero ja existe\n");}
    else{
            p=(def_arvore)malloc(sizeof(struct no_arvore));
            p->info = valor;
            p->esq = NULL;
            p->dir = NULL;
            *arvore=p; }
}
```

Assim a árvore construída a partir da entrada de dados = 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5 é a da Figura 9.

Figura 9. A árvore gerada pelo algoritmo



4.2. Busca em Árvores Binárias de Busca

O Problema da busca:

Seja S = {s_1, ..., s_n} o conjunto de chaves satisfazendo $s_1 < ... < s_n$. Seja x um valor dado. O objetivo é verificar se x E S ou não. Em caso positivo, localizar x em S, isto é, determinar o índice j tal que $x = s_j$.

Para resolver o problema usa-se uma árvore binária T de busca que possui as seguintes características:

(i) T possui n nós. Cada nó v corresponde a uma chave distinta sj E S e possui como rótulo o valor r(v) = sj.

(ii) Seja um nó v de T. Seja também v1 pertencente à subárvore esquerda de v. Então r(v1) < r(v). Analogamente, se v2 pertence à subárvore direita de v, r(v2) > r(v).

Veja a rotina:

```
boolean busca (def_arvore arvore, int valor){
    if (arvore==NULL) return False;
    if (arvore->info == valor) return True;
    if (valor < arvore->info) return busca(arvore->esq,valor);
    if (valor > arvore->info) return busca(arvore->dir,valor);
}
```

4.3. Remoção de nós em uma Árvore de Busca Binária

Quando se quer remover um nó tem três situações para estudar:

(a) o nó a ser removido não tem filhos, ou seja, ele é um nó folha
(b) o nó possui um único filho
(c) o nó possui dois filhos

Caso (a): remoção do nó folha (A)
Basta retirar o nó A. Não precisa fazer nenhum ajuste na árvore. Veja Figura 10.

Caso (b): remoção do nó (B) que possui um filho (F)
O filho (F) deverá ser movido para cima para ocupar a posição do nó removido (B). Assim, deve-se fazer com que o pai do nó-B aponte para o filho do nó-B, ou seja o nó-F. Veja Figura 11.

Caso (c): remoção do nó (C) que possui dois filhos
Procura-se um elemento da subárvore à direita do nó que vai ser retirado (C). Um elemento que é uma folha ou um nó que não possua subárvore à esquerda, ou seja, o menor elemento do lado direito do nó a ser removido. Retira esse elemento de sua posição e o coloca na posição do nó-C. Veja Figura 12.

Figura 10. Exemplo de remoção do **caso (a)**

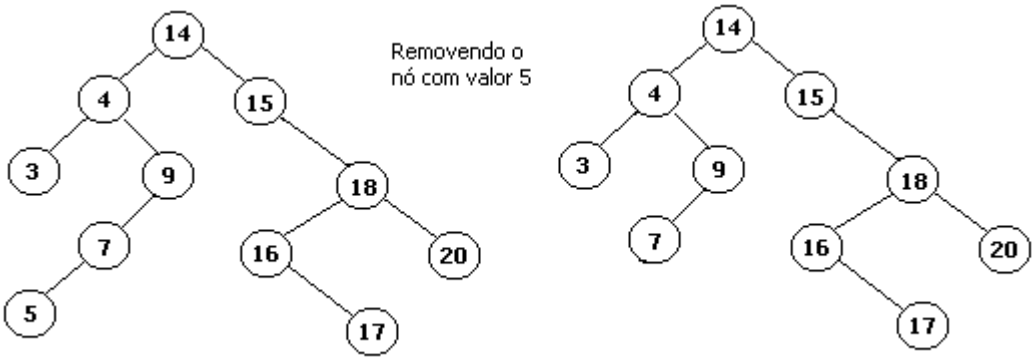


Figura 11. Exemplo de remoção do **caso (b)**

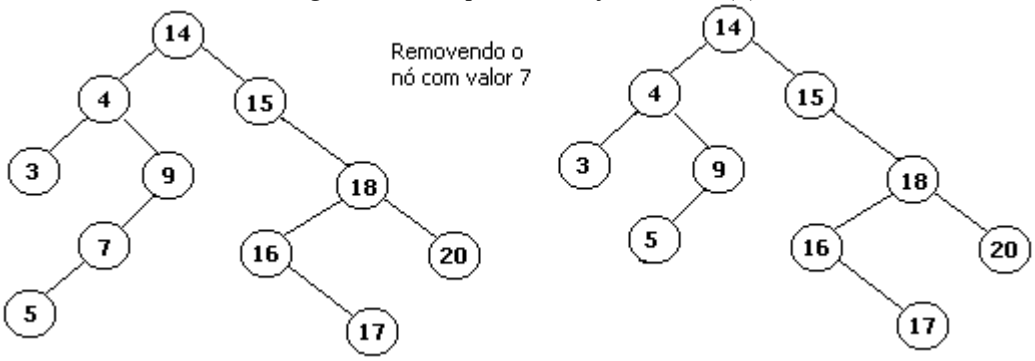
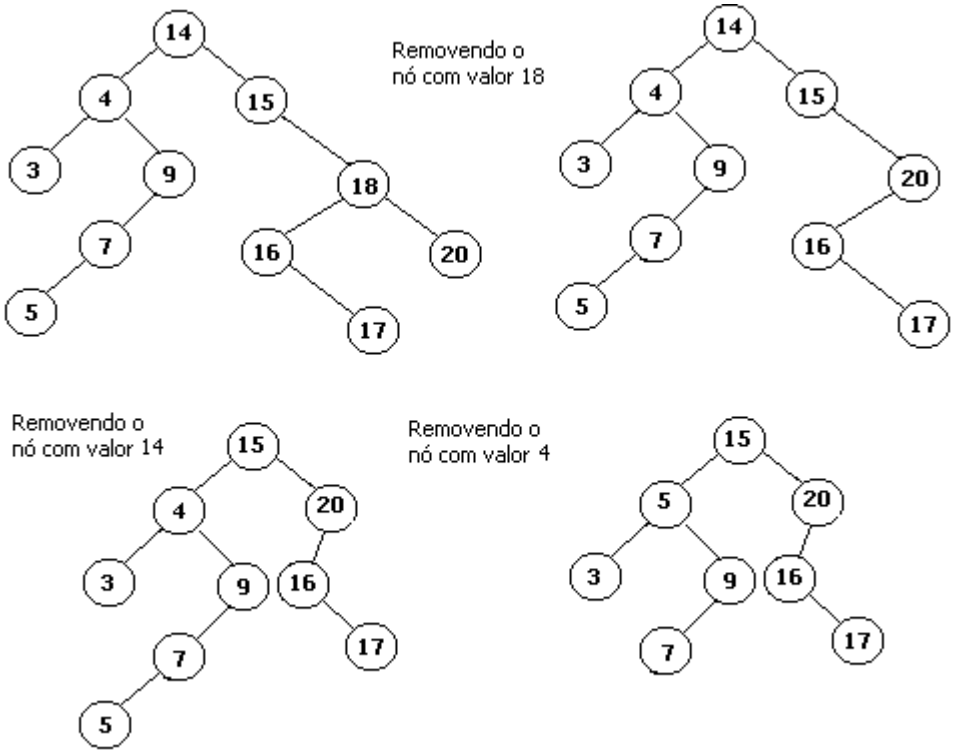


Figura 12. Exemplo de remoção do **caso (c)**



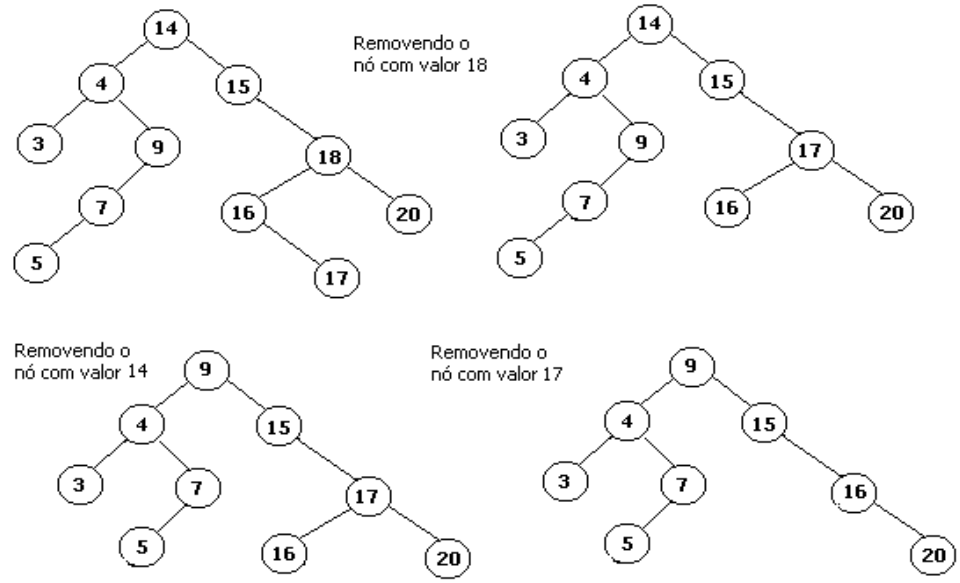
A rotina poderia ser implementada como:

```
void remocao (def_arvore *arvore, int dado){
    def_arvore p,q,rp,f,s;
    p = *arvore;
    q = NULL;
    while (p!=NULL && p->info!=dado){      // Encontra o número
        q = p;
        if (dado < p->info) p=p->esq;
        else p = p->dir;}
    if (p != NULL){                                  // O número existe
      if (p->esq == NULL)  rp=p->dir;              // Possui só um filho à direita
      else if (p->dir == NULL)  rp = p->esq;    // Possui só um filho à esquerda
            else {                                        // Possui dois filhos
                f = p;
                rp= p->dir;
                s= rp->esq;
                while (s != NULL){
                    f=rp;
                    rp=s;
                    s=rp->esq;}
                if (f != p) {
                    f->esq = rp->dir;
                    rp->dir = p->dir;}
                rp->esq = p->esq;}
      if (q == NULL)  *arvore = rp;
      else{
             if (p == q->esq) q->esq=rp;
             else q->dir=rp; }
      free(p);
      }
}
```

OBS:
Esse tipo de remoção de nós de uma árvore de busca binária baseou-se no algoritmo dado no livro do Tenenbaum et al. (1995, p.514), mas pode-se implementar a remoção de outras formas.

A remoção de nós nos casos (a) e (b) permanecem inalterados, mas a remoção no caso (c) onde o nó tem dois filhos pode aparecer de forma diferente em outros livros. Poderia pensar que nesse caso (c), o elemento substituto é o maior elemento que está na subárvore esquerda do nó que vai ser removido. Esse elemento pode ser uma folha ou um nó que não tenha subárvore direita. Veja Figura 13.

Figura 13. Exemplo de remoção do **caso (c)**



.

Assim a rotina seria:

```
void remocao (def_arvore *arvore, int dado){
    def_arvore p,q,rp,f,s;
    p = *arvore;
    q = NULL;
    while (p!=NULL && p->info!=dado){
        q = p;
        if (dado < p->info) p=p->esq;
        else p = p->dir;}
    if (p != NULL){
      if (p->esq == NULL)  rp=p->dir;
      else if (p->dir == NULL)  rp = p->esq;
          else {  f = p;  rp= p->esq;  s= rp->dir;
                  while (s != NULL){
                      f=rp;  rp=s;
                      s=rp->dir;}
                  if (f != p) {
                      f->dir = rp->esq;
                      rp->esq = p->esq;}
                  rp->dir = p->dir;}
      if (q == NULL)  *arvore = rp;
      else{     if (p == q->esq) q->esq=rp;
                else q->dir=rp; }
      free(p);}
}
```

5. Exercícios

1. Usando o conceito de árvore dinâmica implemente uma rotina que faça a remoção de nós em uma árvore de grau qualquer (Grau <= 3). Se o nó não for uma folha, exiba uma mensagem perguntando se querem remover todos os filhos daquele nó e nesse caso ainda permita que se escolha, se o usuário desejar, um filho que ocupe o lugar do nó a ser removido.

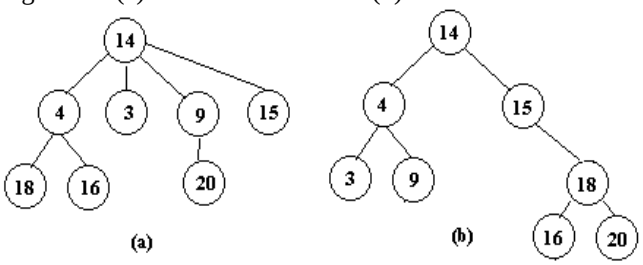
2. Crie uma estrutura de árvore genérica dinâmica e a implemente com rotinas de inserção, busca e impressão.

3. Usando árvore genérica de Grau N, crie as rotinas:
a) encontre o menor elemento de uma árvore
b) encontre o maior elemento de uma árvore
c) encontre a soma de todos os elementos de uma árvore
d) descubra o total de nós de uma árvore
e) descubra a altura da árvore
f) descubra a quantidade de nós que tem números pares em uma árvore
g) descubra quantos nós possuem valores maiores que um valor x
h) determine a quantidade de nós folhas de uma árvore
i) determine a quantidade de nós de uma árvore que possuem somente um filho
j) determine a quantidade de nós de uma árvore que possuem pelo menos um filho
k) determine a quantidade de nós de uma árvore que possuem todos os filhos
l) dado um elemento pertence a árvore, descubra quem é seu pai e seus irmãos

4. Pense num sistema de gerenciamento de pastas de um Sistema Operacional. A cada criação de pasta você escolhe uma pasta que abrigará a sub-pasta a ser criada (evidentemente a raiz é o drive A, C ou D). A remoção de uma pasta consiste em escolher a pasta e apagá-la (aqui, se houver arquivos e/ou pastas dentro da pasta apagada, tudo será removido junto). Implemente essas ações seguindo as diretrizes abaixo:
(a) Defina uma estrutura que comporte o sistema de pastas do SO
(b) Faça a rotina de criação de pasta neste sistema, onde é passada a árvore inteira, a pasta "mãe" e o nome da sub-pasta.

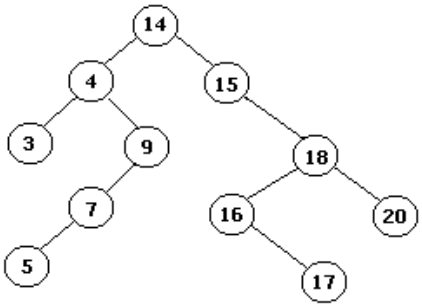
5. É possível transformar uma árvore de N filhos em uma árvore de busca binária. Para isso tem-se que percorrer a primeira árvore segundo algum critério (em largura ou profundidade) e inseri-los na árvore binária. Veja figura 14. Implemente essa conversão usando percurso em largura.

Figura 14. (a) Árvore de 4 filhos e (b) árvore de busca binária



6. Implemente uma rotina que mostre os elementos de uma árvore por níveis de forma não recursiva. Por exemplo, na árvore binária abaixo a sequência seria: 14; 4; 15; 3; 9; 18; 7; 16; 20;5; 17.

Figura 15. Arvore de busca binária



7. Podemos fazer várias formas de busca. Uma delas é buscar pelo valor do elemento e retornar verdadeiro ou falso, como já feito. Outra forma é buscar pelo valor, mas retornar o ponteiro do nó que armazena o valor ou retornar NULL se o valor não existir.

def_arvore busca_no(def_arvore arvore, tipo_no valor)

8. Usando árvores binárias de busca que guardam números em seus nós, faça rotinas que:

a) elimine o menor elemento da árvore
b) elimine o maior elemento da árvore

9. Desenvolva a função Copia(Arvore: TArv): TArv, que duplica uma árvore binária e retorna o endereço da cópia.

10. Duas árvores binárias são "iguais" se são ambas vazias, ou então se armazenam valores "iguais" em suas raízes, suas subárvores esquerdas são iguais e suas subárvores direitas também são "iguais". Desenvolva a função lógica Igual(Arv1, Arv2) que determina se duas árvores binária Arv1 e Arv2 são "iguais".

11. Seja um percurso em uma árvore binária definido pelas seguintes operações:

Ordem A: visitar a raiz;
percorrer a subárvore esquerda na ordem A;
percorrer a subárvore direita na ordem B

Ordem B: percorrer a subárvore esquerda na ordem B;
visitar a raiz;
percorrer a subárvore direita na ordem A

Supondo que o processo se inicie pela raiz, em ordem A, implemente esse percurso numa rotina.

12. Crie uma rotina que verifique se uma árvore binária é estritamente binária.

13. Conte quantas vezes uma palavra aparece num texto. Para isso considere uma árvore binária de busca. O problema consiste em, dada uma sequência de palavras vinda de um arquivo, determinar o número de ocorrências de cada uma delas. Esse número deverá ficar guardado junto com a palavra de forma que se a palavra for encontrada, o respectivo contador de ocorrências é incrementado, caso contrário ela é inserida e caracterizada como uma nova palavra (contador = 1).

14. Considere um labirinto que conste de corredores e saletas. A saleta pode permitir a escolha entre dois outros corredores ou te levar a outra saleta (um único corredor) ou não ter saída. Cada saleta tem um número identificando-a.

(a) Defina a estrutura que comporte esse labirinto
(b) Crie uma rotina para dado o número de uma saleta seja possível
- retornar o caminho feito até a chegada a essa saleta
- retornar se essa saleta tem dois, um ou nenhum corredor à frente

15. Uma árvore binária de busca pode ser usada para trabalhar com ordenação de números por possuir características especiais para inserção e remoção de elementos. Crie uma rotina que

remova o maior elemento e (retorne o número pela chamada da rotina)
retorne quem era seu pai e (como passagem de parâmetro)
se tiver, quem era seu irmão. (como passagem de parâmetro)

16. Ao fazer uma busca por um valor numa árvore binária de busca retorna-se Verdadeiro ou Falso e em algumas situações a referência do nó onde o valor está armazenado. A idéia, agora, é retornar o caminho feito até encontrar o valor nessa árvore.

(a) Use a estrutura que tem sido usada em sala de aula ou redefina-a.
(b) Crie uma (única) rotina de busca que:
   a. retorne a referência do nó e
   b. retorne, também, o caminho onde ele se encontra (não é para fazer a impressão do caminho dentro da rotina)

17. Considere o uso de uma árvore binária de busca

(a) Construa, de forma gráfica, a árvore binária de busca a partir da sequencia S = 69,77,75, 50, 55, 88, 60, 54, 58, 85;
(b) Se na sequencia S for alterada a ordem de algum de seus números, esse fato alterará a árvore construída? Explique sua resposta?
(c) Mostre a sequencia obtida por um percurso pré-ordem;
(d) Retire os seguintes elementos, nesta ordem: 50, 85, 55 e 69 .

18. Duas árvores binárias são espelho-similares (mirror similar) se elas são vazias ou as subárvores esquerdas de cada uma são espelho-similares as subárvores direita da outra. Escreva um programa para determinar se duas árvores binárias são espelho similares.

19. Escreva um programa que receba uma expressão matemática (composta por operandos compostos por um inteiro, operações de +, -, * e / e parênteses) representada por um string:
a) crie uma árvore binária representando esta expressão
b) mostre os percursos in-ordem, pos-ordem e pré-ordem (certifique-se dos parênteses)
c) calcule o valor da expressão