UNIVERSIDADE SÃO JUDAS TADEU
ENGENHARIA DE COMPUTAÇÃO / BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Professor: H. Senger - Outubro de 1999

Notas de Aula
ÁRVORES BINÁRIAS

UNIVERSIDADE SÃO JUDAS TADEU
ENGENHARIA DE COMPUTAÇÃO / BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Professor: H. Senger - Outubro de 1999

Notas de Aula

ÁRVORES BINÁRIAS

1 Árvore Binária de Busca

Para que uma árvore binária seja adequada a operações de busca, é preciso haver um critério de armazenamento. O critério que vamos adotar é o seguinte:

- Todo dado com valor menor que o valor contido no nó raiz deve ser armazenado na sua subárvore esquerda,
- Todo dado com valor maior que o valor contido no nó raiz deve ser armazenado na sua subárvore direita,
- O mesmo critério deve ser aplicado para ambas as subárvores da raiz, e para as subárvores das subárvores, etc., até o nível das folhas.

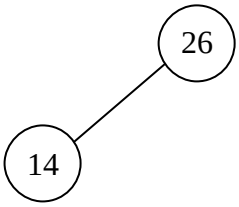
Obs. Poderia ser o contrário, ou seja, os maiores à esquerda e os menores à direita.

Assim, vejamos como construir uma árvore binária a partir da seqüência abaixo:

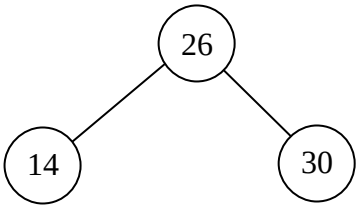
26, 14, 30, 36, 23, 8, 35, 18

Após inserir o 26:

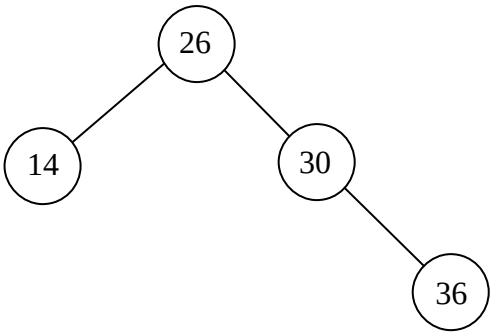
Após inserir o 14:



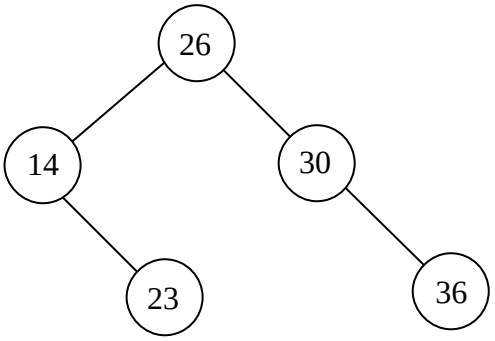
Após inserir 30:



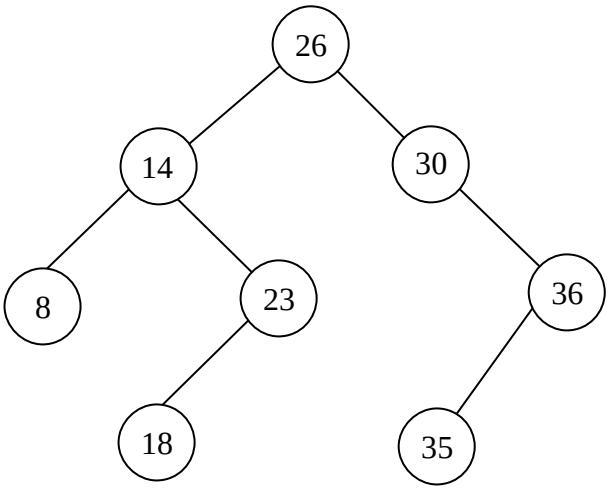
Após inserir 36:



Após inserir 23:

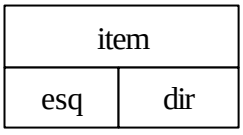


E após inserir 8, 35 e 18:



1.1 Implementação de uma Árvore Binária de Busca

Cada nó da árvore binária pode ser assim implementado :



Cada nó contém os campos :

- **item** : guarda um item de dado
- **esq** : ponteiro para sub-árvore esquerda
- **dir** : ponteiro para sub-árvore direita

Vejamos como representar essa estrutura de dados em linguagem C ou C++:

- Inicialmente, vamos declarar o tipo de dado que vai no interior de cada nó :

```
novotipo inteiro TipoItem;
```

- Neste caso, declaramos que TipoItem eqüivale ao tipo **int**, pois queremos que nossa árvore binária seja capaz de guardar números inteiros.

- Em seguida, vamos construir um nó da árvore :

```
estrutura NO {
   TipoItem          item;
   estrutura NO      *esq,   *dir;
};
```

- A declaração acima determina como é a organização de um nó da árvore. Mas como é que podemos representar uma árvore inteira, e não apenas um nó ? Uma forma de fazer isso é criarmos um ponteiro capaz de guardar o endereço do nó raiz de uma árvore.
- Assim, inicialmente declaramos um tipo especial chamado PONT, capaz de guardar o endereço de um nó, e em seguida, uma variável do tipo PONT :

```
novotipo estrutura  NO * PONT;

...

PONT  Raiz;    /* A variável Raiz aponta para o nó raiz */
```

- Em seguida, poderíamos construir a árvore, utilizando uma rotina capaz de adicionar um único nó à árvore. Bastaria ler um punhado de dados, um por vez, chamando essa rotina para fazer a inclusão de cada dado na árvore :

```
   ...
   Inicializar(Raiz);
   faça { /* loop de construção da árvore */
         exibir ("Dado (ou -1 p/finalizar): ");
         ler_teclado ("%d", &X);
         se (X!=-1)  Inserir(Raiz, X);
   } enquanto (X>-1);
. . .
```

- A inicialização da árvore é feita simplesmente atribuindo o valor **NULL** para o ponteiro **Raiz**. Assim, a rotina possui apenas uma atribuição em seu interior, algo parecido com o trecho abaixo:

```
...
Raiz  = NULL;
...
```

1.1.1 Inclusão de um dado

- A inclusão de um novo dado na árvore pode utilizar um algoritmo recursivo, ou não. Em se tratando de árvores, porém, os algoritmos recursivos em geral são mais simples do que os equivalentes não-recursivos.
- Um algoritmo recursivo para incluir um novo dado em uma árvore binária poderia proceder da seguinte forma :
  - ◊ Se a árvore recebida está vazia, então inclua o novo nó neste ponto
  - ◊ Senão, faça o seguinte
    - Se o novo dado for menor que o dado da raiz, inclua-o na subárvore ESQUERDA
    - Se o novo dado for maior que o dado da raiz, inclua-o na subárvore DIREITA

```
INSERIR_ABB (Raiz, Dado)
se Raiz = NULL, então
      Raiz = alocar_novo_no ( );
      Raiz®item  = Dado;
      Raiz®esq = NULO;
      Raiz®dir = NULO;
   senão
      se Dado < Raiz®item  então
                  INSERIR_ABB(Raiz®esq, Dado)
      senão  se Dado > Raiz®item  então
                  INSERIR_ABB(Raiz®dir, Dado)
            senão "Erro – Dado duplicado"
```

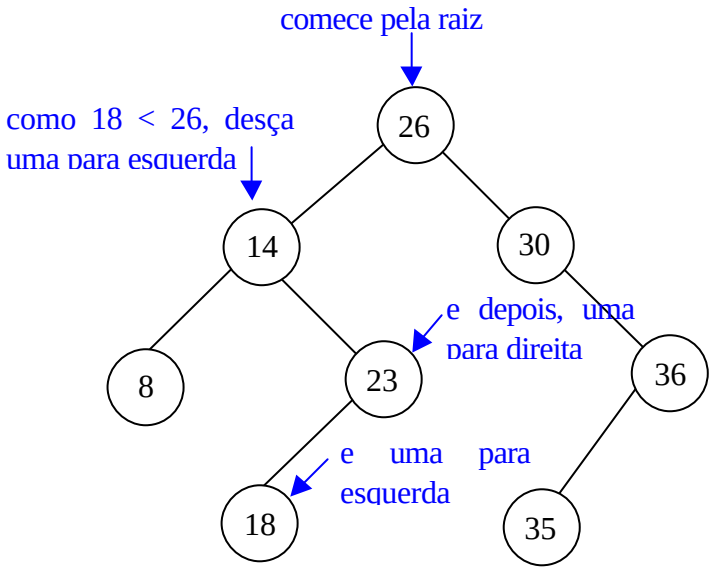
Obs. Você percebeu que estamos escrevendo "**=**" para indicar uma atribuição ao invés de usar o símbolo "←" ?.
É para evitar a confusão com o "→", usado com variáveis do tipo ponteiro (operador de "indireção" da linguagem C/C++).

1.1.2 Busca de um dado

Qualquer operação de busca deve começar pela raiz da árvore, e ir "descendo", ora para esquerda, ora para direita, até encontrar o dado.

Ocorre que, às vezes o dado quer procuramos não está na árvore. É fácil perceber isso, porque em um certo momento, ao tentar descer mais um nó, simplesmente chegamos ao final da árvore.

Vejamos como localizar o valor 18 na árvore abaixo :



Se estivéssemos procurando o valor 20 (que não existe nessa árvore), tentaríamos descer ainda mais um nó, abaixo e à direita do 18. Como não dá para descer mais, eu descubro que 20 não existe nessa árvore.

```
BUSCA_ENDEREÇO (Raiz, Dado)
  se Raiz = NULO então
     devolva NULO  /* se não achou, devolve NULO */
  senão  se Dado = Raiz→item  então
                devolva Raiz
          senão
                se Dado > Raiz→item  então
                   devolva BUSCA_ENDEREÇO(Raiz®dir, Dado)
                senão
                   devolva BUSCA_ENDEREÇO(Raiz®esq, Dado)
```

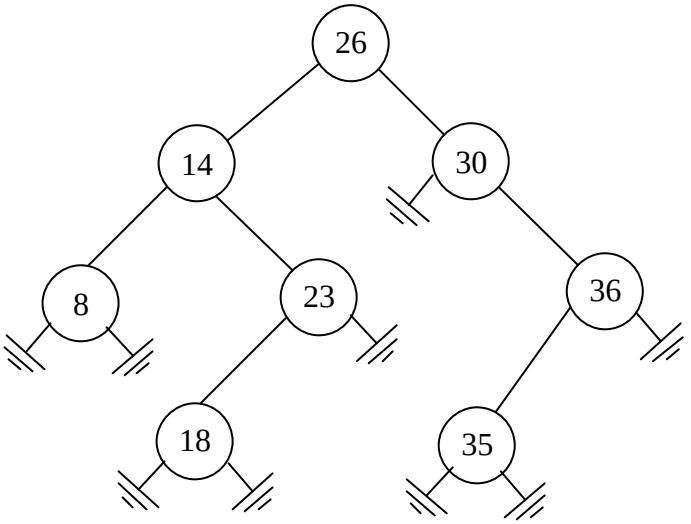
1.1.3 Remoção de um dado

Ao se retirar um dado da árvore, 3 situações podem ocorrer :

a) O nó removido é um nó folha (não tem filhos) : então basta removê-lo, colocando um NULO em seu lugar. Na árvore abaixo, 8, 18 e 35 são exemplos de nó folha.

b) O nó removido tem um filho (que pode ser à esquerda, ou à direita) : nesse caso, o filho toma o lugar do nó removido. Na árvore abaixo, se o nó com valor 30 for removido, o 36 tomará seu lugar.

c) O nó removido tem dois filhos: Se isso ocorrer, outro nó deve ser escolhido para substituir o nó removido. Existem duas soluções igualmente corretas e viáveis : podemos pegar o nó mais à direita na subárvore esquerda de quem foi removido, para substituí-lo; ou pegar o nó mais à esquerda na subárvore direita do nó removido para substituí-lo. Tanto faz. No exemplo abaixo, se tivermos de remover o 26, tanto o 23 como o 30 poderiam substituí-lo.



- Vamos implementar uma rotina para remover o dado. Porém, o que aconteceria se alguém pedisse para apagar um dado da árvore, sendo que esse dado não existe ? A rotina de remover deve fazer uma **tentativa** de remoção, avisando se conseguiu remover ou não.

```
. . .
Status = REMOVER (Raiz, Dado);
se Status ≠ FRACASSO
  então "DADO NÃO EXISTE, E NÃO PODE SER REMOVIDO"
senão
    "OK – DADO FOI REMOVIDO"
. . .
```

- A variável ***Status*** serve para observar se a resposta devolvida pela rotina indica se a tentativa de remover teve sucesso ou não.

- A rotina REMOVER é dada abaixo :

```
REMOVER (PONT& R, TipoItem Dado)
  PONT p, q;
  /* Se a arvore eh vazia, entao o dado procurado nao existe ...*/
  se (R = NULO)   então devolva FRACASSO;
  senão
   se ( Dado > R→item)
        então devolva REMOVER (R→dir, Dado);
   senão
    se (Dado < R®item) devolva REMOVER (R®esq,Dado);
    senão      /* encontrou o dado */
      se ((R®esq = NULO) E (R®dir =NULO)) { /* se nao tem filhos ...*/
                desaloque ( R );
                  R = NULO;
      senão  se (R®esq = NULO) {/* só tem filho a direita */
                p = R®dir;
                desaloque (R);
                R = p;
           senão  se (R®dir = NULO ) {/*se só tem filho a esquerda */
                     p = R®esq;
                     desaloque (R);
                     R = p;
                senão { /* tem dois filhos */
                     p = R®esq; /* procura o maior da subarv. esq*/
                     q = NULO;
                     enquanto (p®dir != NULO)
                          q = p;
                          p = p®dir;
                     se ( q = NULO ) { /* se p e q não se moveram */
                          R®esq = p®esq;
                          R®item = p®item;
                          desaloque (p);
                     senão
                            q®dir = p®esq;
                            R®item = p®item;
                            desaloque (p);
```

1.2 Implementação

Vejamos agora, como tudo pode ser ficaria implementado na prática. Não se esqueça :

```
#include <stdio.h>

#define SUCESSO 1    /* usado no retorno de algumas funções */
#define FRACASSO 0
```

Inicialmente, é preciso declaras as estruturas de dados. Vamos declarar o tipo que define o conteúdo de cada item de dado.

```
typedef int TipoItem;
struct NO {
   TipoItem item;
   struct NO *esq, *dir;
};
typedef struct NO *PONT;
```

```
/* ------------------------------------------------
            Dec. das Estruturas de dados
   ------------------------------------------------ */
typedef int TipoItem;
struct NO {
    TipoItem item;
    struct NO *esq, *dir;
};
typedef struct NO *PONT;

/* ------------------------------------------------------
           Funcoes de tratamento da árvore
   ------------------------------------------------------ */
void Inicializar(PONT& R)
{
      R = NULL;   /* Inicializa como vazia */
}

int Inserir (PONT&  R, TipoItem Novo)
{
  if (R == NULL)/* Se árv. recebida estiver vazia ...*/
  {
      /* então acrescenta novo nó na raiz */
      if ((R=(PONT)malloc(sizeof(struct NO)))==NULL) {
         printf ("ERRO em malloc\n");
         exit(0);
      }
      R->item = Novo;
      R->esq = NULL;
      R->dir = NULL;
      return SUCESSO;
   }
   else if (Novo > R->item)
            return Inserir(R->dir, Novo);
        else if (Novo < R->item)
                 return Inserir(R->esq, Novo);
             else return FRACASSO;
}

void Exibir(PONT R, int H)
{
    int i;
    if (R!=NULL) {
       Exibir(R->dir, H+1);
       for(i=0; i<H; i++)
```

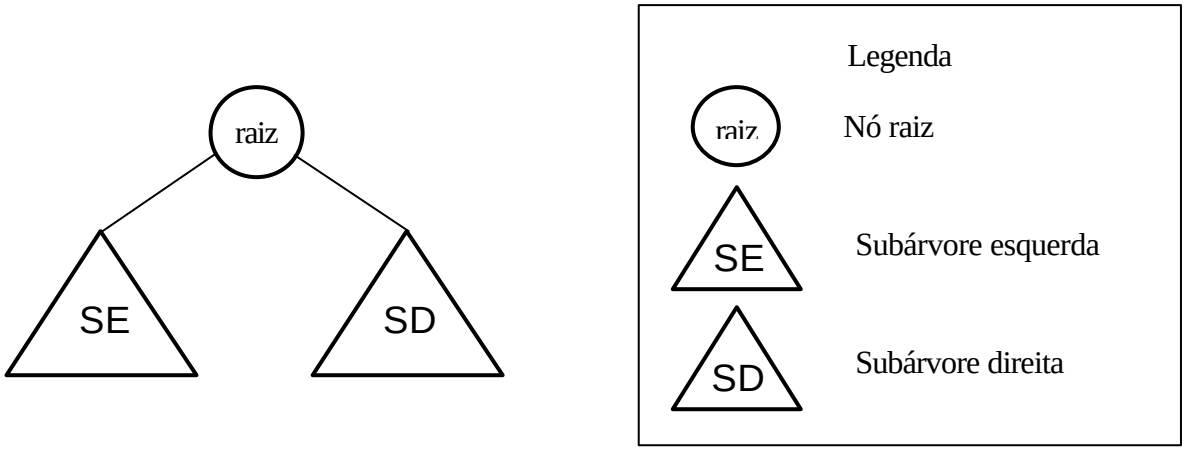
Aqui você tem uma rotina capaz de inserir um novo dado na árvore (Inserir). Para criar uma árvore completa, basta chamá-la diversas vezes, inserindo vários nós.
Há também uma rotina para exibir o conteúdo de uma árvore. Com ela você pode testar se a rotina anterior criou a árvore corretamente.

```
void main ()
{
      PONT Raiz, end;
      TipoItem X;
      int opcao, status;

      Inicializar(Raiz);
      do {
            clrscr();
            printf("%s %s %s %s %s %s %s %s %s %s",
                  " Menu :\n",
                  "1 - Inicializar \n", "2 - Construir\n",
                  "3 - Exibir \n",      "4 - Procurar\n",
                  "5 - Remover\n",      "6 - Pre-Ordem\n",
                  "7 - In-Ordem\n",     "8 - Pos-Ordem\n",
                  "0 - FIM\n");
            printf("        Sua opcao :");
            scanf("%d",&opcao);
            switch(opcao) {
               case 1: Inicializar(Raiz);
                     break;
               case 2: do {
                        printf(" Forneca um dado ou -1 para finalizar:");
                        scanf("%d",&X);
                        if (X>-1) {
                              status = Inserir(Raiz,X);
                              if (status == FRACASSO)
                                 printf(" \n Problemas na insercao.\n");
                              else printf(" Ok\n");
                        }
                     } while (X>-1);
                     break;
               case 3: printf("\n\n\n     Exibicao da arvore ...\n");
                     Exibir(Raiz,0);
                     break;
               case 4: printf(" Opcao ainda nao implementada ...");
                     break;
               case 5: printf(" Opcao ainda nao implementada ...");
                     break;
               case 6: printf(" Opcao ainda nao implementada ...");
                     break;
               case 7: printf(" Opcao ainda nao implementada ...");
                     break;
               case 8: printf(" Opcao ainda nao implementada ...");
                     break;
               case 0: printf(" Bye ...");
                     exit(0);
                     break;
            }
            getchar();
            printf("\n\n           Tecle enter ...");
            getchar();
```

1.3 Percurso em árvores

Depois que uma árvore binária foi criada, existem diversas maneiras de percorrê-la, exibindo seu conteúdo. Dependendo da forma com que uma árvore binária é percorrida, seus dados serão exibidos em uma ordem diferente. Imagine uma árvore binária genérica, com este formato :



Vamos ver algumas as principais formas de se percorrer uma árvore binária, visitando seus nós e exibindo seu conteúdo.

1.3.1 Pré-ordem

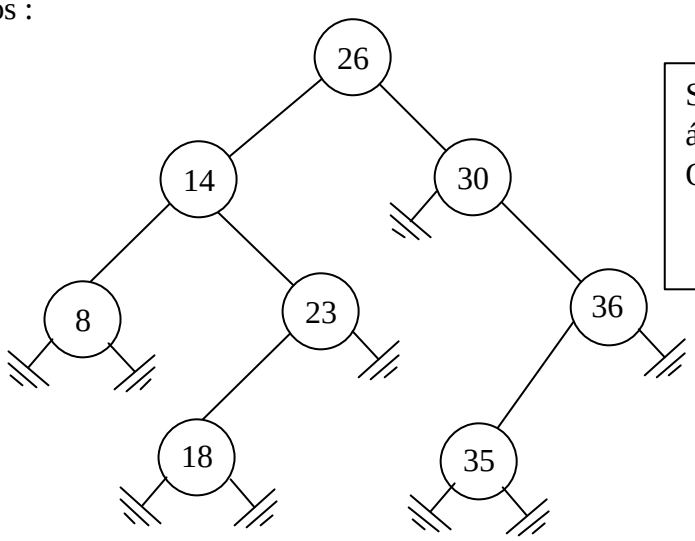
O algoritmo da pré-ordem é o seguinte :

- Se a árvore recebida for vazia, não faça nada e a PRE-ORDEM termina
- caso contrário
  - ◊ visite o nó raiz, exibindo seu conteúdo 

  - ◊ percorra a subárvore esquerda em PRE-ORDEM 

  - ◊ percorra a subárvore direita em PRE-ORDEM 

Veja a árvore binária abaixo. Se a percorrermos em PRE-ORDEM, teremos a seguinte seqüência dados :



Seqüência obtida, quando a árvore é percorrida em Pré-Ordem :

26,14,8,23,18,30,36,35

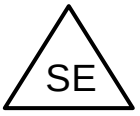
Veja o algoritmo da Pré-ordem :

```
PREORDEM (Raiz)
  se Raiz ≠ NULO então
       exibir_conteúdo (Raiz)
       PREORDEM (Raiz→esq);
       PREORDEM (Raiz→dir);
```

1.3.2 In-ordem

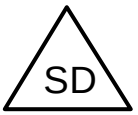
O algoritmo da IN-ORDEM é o seguinte :

- Se a árvore recebida for vazia, não faça nada e a IN-ORDEM termina
- caso contrário
  - ◊ percorra a subárvore esquerda em IN-ORDEM

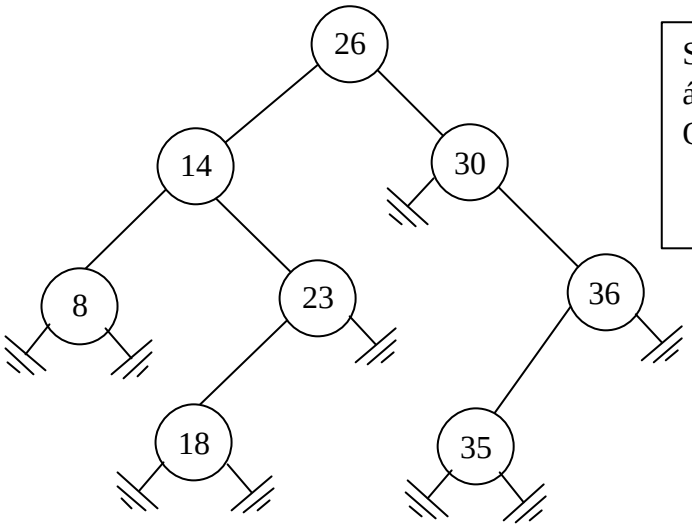


  - ◊ visite o nó raiz, exibindo seu conteúdo

  - ◊ percorra a subárvore direita em IN-ORDEM



Veja a árvore binária abaixo. Se a percorrermos em IN-ORDEM, teremos a seguinte seqüência dados :



Seqüência obtida, quando a árvore é percorrida em In-Ordem :
8,14,18,23,26,30,35,36

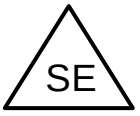
Veja o algoritmo da In-ordem :

```
INORDEM (Raiz)
  se Raiz ≠ NULO então
       INORDEM (Raiz→esq);
       exibir_conteúdo (Raiz)
       INORDEM (Raiz→dir);
```

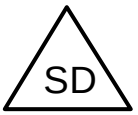
1.3.3 Pós -ordem

O algoritmo da PÓS-ORDEM é o seguinte :

- Se a árvore recebida for vazia, não faça nada e a PÓS-ORDEM termina
- caso contrário :
  - ◊ percorra a subárvore esquerda em PÖS-ORDEM

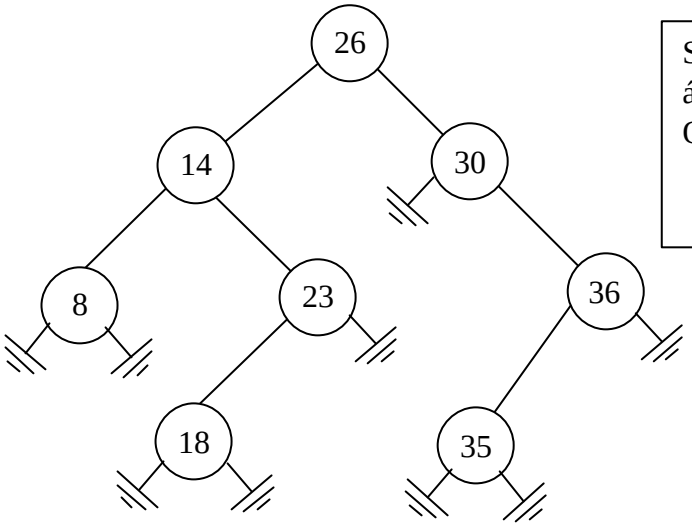


  - ◊ percorra a subárvore direita em PÓS-ORDEM



  - ◊ visite o nó raiz, exibindo seu conteúdo

Veja a árvore binária abaixo. Se a percorrermos em PÓS-ORDEM, teremos a seguinte seqüência dados :



Seqüência obtida, quando a árvore é percorrida em Pós-Ordem :
8,18,23,14,35,36,30,26

Veja o algoritmo da Pós-ordem :

```
PÓSORDEM (Raiz)
  se Raiz ≠ NULO então
      POSORDEM (Raiz→esq);
      POSORDEM (Raiz→dir);
      exibir_conteúdo (Raiz);
```

1.4 Análise de eficiência dos algoritmos

Pergunta : Quanto tempo leva para fazer uma busca na árvore binária de busca (ABB) ?

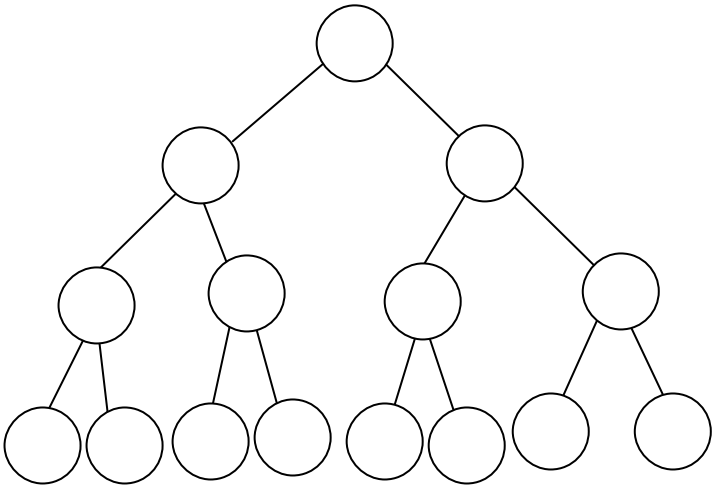
Resposta : Bem, isso depende. Depende da particular árvore na qual eu vou fazer a busca. Vamos supor que a minha árvore tenha *n* dados, inseridos aleatoriamente, ou seja, alguém digitou ao acaso.

- **No melhor caso**, o dado que eu procuro é o próprio que está na raiz. Assim, o tempo de busca nessa árvore de *n* nós é **T(n)=1**.
- **No pior caso**, o dado que eu quero está em um nó folha, ou então não existe, o que significa que terei de chegar até um ponteiro NULO :
  - ◊ Nesse caso, vou supor duas situações extremas. Na primeira delas, a árvore binária gerada é **perfeitamente balanceada**.
    - Vamos supor que *h* seja a altura da árvore gerada.
    - Em cada nível *i* da árvore temos exatamente 2^i nós. Por exemplo, no nível ZERO (o mesmo nível da raiz) existem $2^0 = 1$ nó. No nível UM existem $2^1 = 2$ nós, e assim por diante.
    - Assim, para a nossa árvore de altura *h* teremos um total de *n* nós, onde :

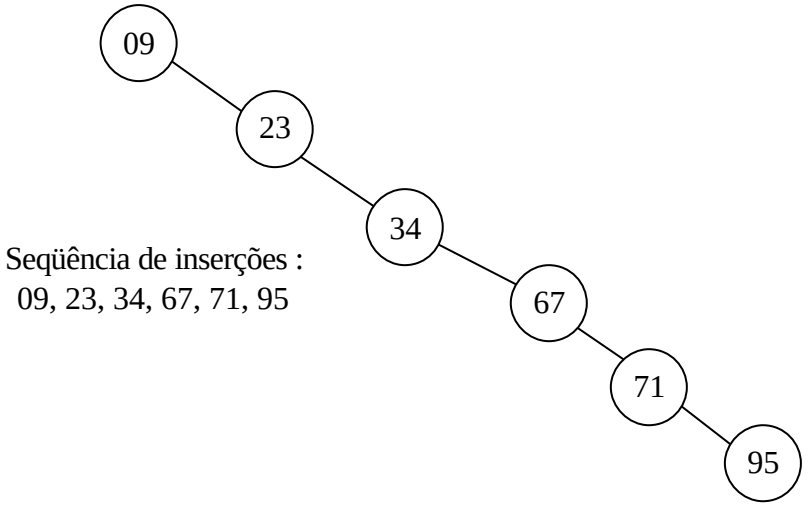
$$n = \sum_{i=0}^{h-1} 2^i$$

$$\therefore n = 2^h - 1$$

    - Como o número de comparações para o pior caso é a própria altura da árvore, ou seja, *h*, temos : $h = \lg^{n+1}$. Assim, T(*n*)=*O*(lg *n*).



- ◊ Na segunda, a árvore é **degenerada** :
  - Nesse caso, cada nó tem apenas um filho, até que o último deles é um folha. Assim, a altura da árvore é dada pela quantidade de nós, ou seja, $h = n$. *Assim,* $T(n) = n$.



- ◊ Assim, podemos concluir que, **no pior caso, o tempo de busca T(*n*) varia entre *O*(*n*) e *O* ($\log_2 n$)**.
- ◊ É verdade que essa árvore degenerada pode ocorrer, mas para isso é preciso que a seqüência de inserções siga uma tendência crescente, ou decrescente.
- ◊ Na sua opinião, qual é a probabilidade de alguém digitar uma seqüência de *n* valores em ordem crescente ou decrescente (considerando um *n* grande), sendo que os números foram sorteados ao acaso ?
- ◊ Eu acho que isso é muito difícil de ocorrer, você não acha ?

- Mas e quanto ao **caso médio** ? Qual é o tempo médio de busca de um dado na árvore desse tipo ?
  - ◊ Para fazer essa estimativa, nós precisamos supor duas coisas :
    - Que todos os valores da árvore tem a mesma probabilidade de acesso.
    - E mesmo aqueles valores que não existem na árvore (mas que alguém poderia tentar procurar) também tem a mesma probabilidade de acesso.
  - ◊ Como um pouco de trabalho, e a matemática adequada, podemos verificar que o tempo médio de busca nessa árvore é de $\mathbf{1.386 \log_2^n}$, ou seja, que, **em média, a árvore de busca está apenas cerca de 39% desbalanceada**.[1]

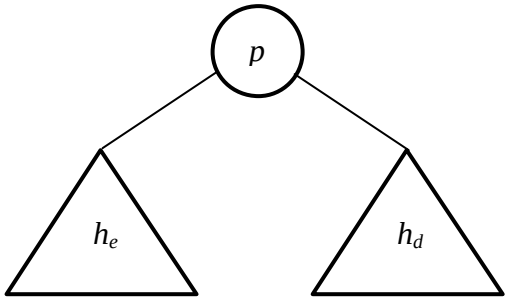
Na prática, árvores degeneradas como as do exemplo acima são muito raras de acontecer, e por isso a busca nesse tipo de árvore tem um desempenho bastante aceitável.

[1] Se você quiser ver como se chega a esse resultado, há dois livros que posso recomendar :
- Tenenbaum, A.M., et.al. "Estruturas de dados usando C", Makron, 1995. pg.517-520.
- Knuth, D.E., "The art of computer programming", Vol. 3 "Sorting and searching", 1973. pg. 427.

1.5 Árvores de busca com balanceamento - AVL

Essa árvore é chamada de AVL, em homenagem aos seus criadores, G.M. **A**del'son-**V**el'skii e M.E. Landis (1962).

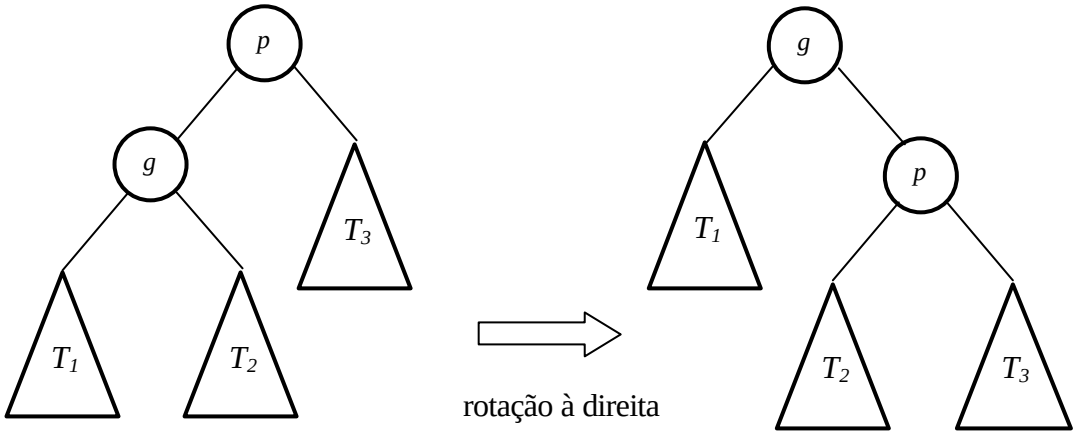
Definição: Uma árvore AVL obedece sempre a seguinte propriedade : a diferença de altura suas duas subárvores só pode ser 1, 0 ou –1.

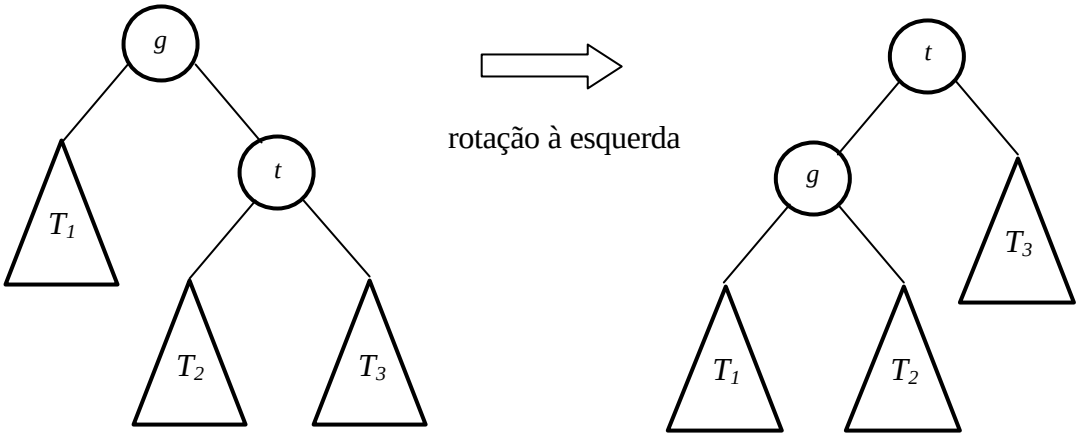


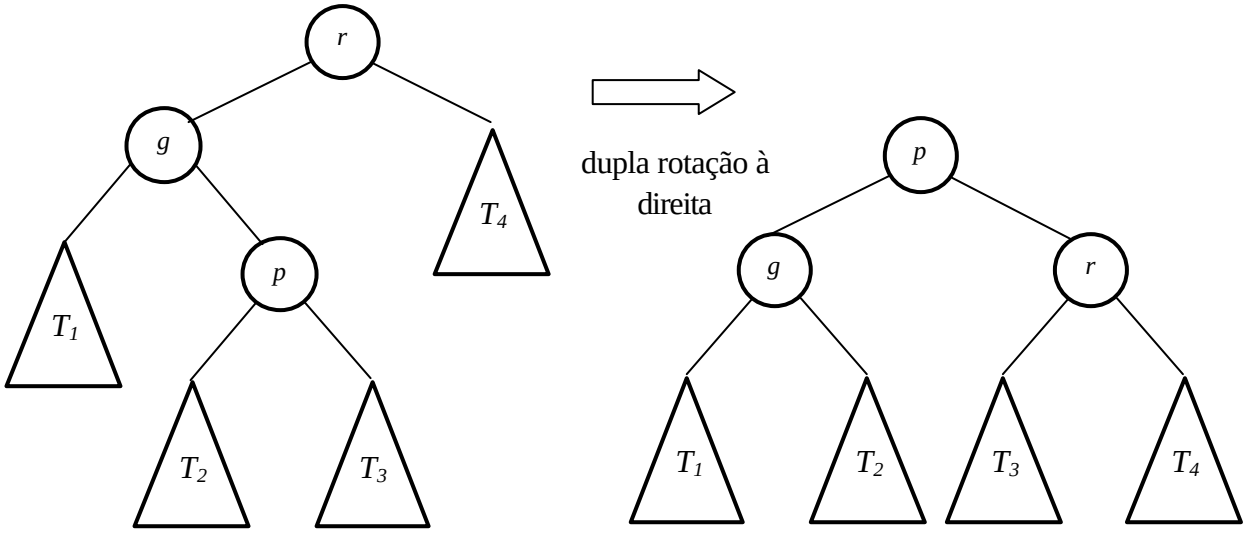
Propriedade p/árvore AVL:

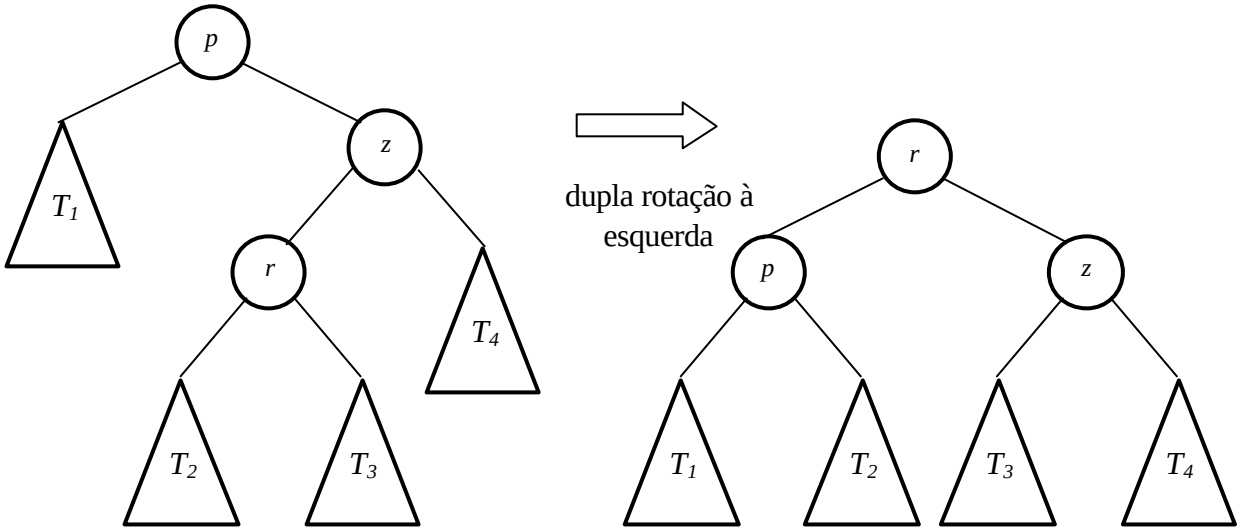
$|h_e - h_d| \leq 1$

Imagine que a árvore acima está balanceada, isto é, segue a propriedade das árvores AVL. Quando inserimos ou removemos um nó da árvore, é aí que pode ocorrer o desequilíbrio entre as duas subárvores. Assim, a inclusão e remoção são efetuadas de forma bastante parecida com a ABB sem balanceamento que vimos em na seção 1, e a árvore em seguida é "consertada", baseada nas 4 transformações ilustradas a seguir.









Knuth [2] provou que a altura ***h*** dessa árvore obedece a seguinte relação :

$$\log_2^{(n+1)} < \boldsymbol{h} < 1{,}4404 \log_2^{(n+2)} - 0{,}328$$

Isto significa que, **no pior caso**, a árvore AVL está 44% desbalanceada.
Além disso, a árvore AVL possui um tempo de busca, **no pior caso**, de :

$$T(n) = \boldsymbol{h} = O(\lg n)$$

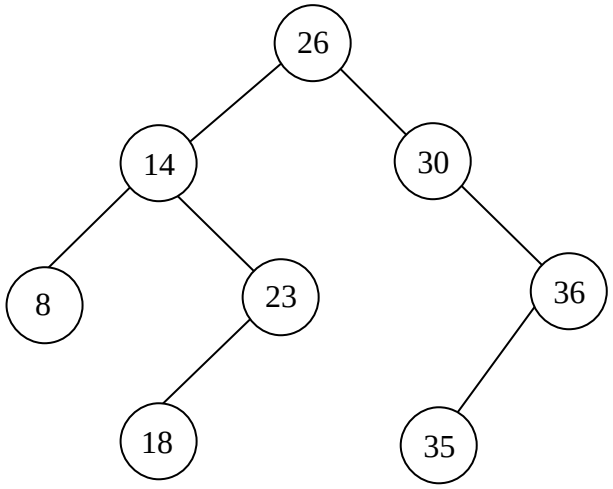
[2] Knuth, D.E., "The art of computer programming", Vol. 3 "Sorting and searching", 1973. pg. 427.

1.6 Árvore de busca otimizada

Mas afinal, qual é a melhor árvore de busca que se conhece ? Existe alguma árvore realmente **ótima** ?

Existe uma árvore binária que leva esse nome, e que tenta otimizar a árvore, ou seja, minimizar o tempo de busca. Essa árvore é apresentada em [3]. Vamos apenas falar um pouquinho sobre ela, somente por curiosidade e informalmente.

Essa árvore só serve quando eu conheço a **freqüência relativa de acesso de cada nó**. Para entender melhor, vamos ver a árvore abaixo :



Imagine que os dados mais procurados sejam :

Valor	fr_i (freqüência relativa)
30	19%
14	17%
36	14%
18	10%
35	8%
23	7%
8	6%
26	3%

Um algoritmo poderia reorganizar a árvore, deixando os dados mais freqüentemente acessados mais próximos da raiz.

Existem um problema com esse tipo de árvore :

- Os algoritmos de organização da árvore são caros ($O(n^2)$), o que pode encarecer a operação de reorganização, ou mesmo inviabilizá-la para valores de n muito grande.

[3] Knuth , D.E. *The Art of Computer Programming – Vol. 3: Sorting and Searching*, Addison-Wesley Publishing, Co., 1973, pg. 435

O estudo de estruturas de dados, seus algoritmos, sua eficiência e aplicações vem sendo desenvolvido desde os anos 50, e tiveram seus principais resultados publicados nas décadas de 60 e 70. Nos anos 80 e 90 ainda acontecem avanços na área, mas em pontos muito específicos.

2 Roteiro de Estudo

Este material tem por objetivo, fornecer um roteiro de estudo sobre árvores binárias. Seu conteúdo inclui exercícios propostos e exercícios resolvidos, intercalados de forma que o aluno possa evoluir através dele.

Este material foi elaborado tendo em vista o ESTUDO INDIVIDUAL, de modo que você poderá aproveitá-lo melhor se tentar não depender da ajuda de colegas. Isto não significa que você está proibido de pedir ajuda ao colega, mas faça isso com moderação. Tente evoluir por conta própria, sem exagerar no pedido de ajuda.

O Professor agradece pelos comentários que você queira fazer sobre a organização deste material; correções, melhorias, críticas, etc., através do e-mail hermes@pcs.usp.br – BOM ESTUDO !!!

Exercício 1 - Programa de teste

Digite, ou tente obter por download o programa ilustrado na seção 1.2. Se você tiver muita dificuldade no assunto, digite-o, pois isso facilita o seu entendimento.

Exercício 2 - Busca (resolvido)

Veja o algoritmo proposto em 1.1.2, para procurar um dado dentro da árvore, devolvendo como resposta o endereço do nó onde o dado foi encontrado. Caso não encontre, a rotina deve devolver um ponteiro NULO como resposta. Note que é preciso receber como parâmetro, o dado e o ponteiro para a raiz da árvore, pois sem eles não dá para fazer busca nenhuma.

Solução:

```
1 -   PONT Busca_Endereco (PONT R, TipoItem Dado)
2 -   {
3 -   if (R==NULL)
4 -           return NULL;
5 -   else if (Dado == R->item)
6 -             return R;
7 -         else if (Dado > R->item)
8 -                   return Busca_Endereco (R->dir, Dado);
9 -               else
10 -                  return Busca_Endereco (R->esq,Dado);
11 -  }
```

Obs.: Note o uso da recursividade nas linhas 8 e 10. Na linha 8, o algoritmo tenta efetuar uma nova busca na subárvore esquerda. Essa chamada recursiva irá retornar algum valor, no futuro. Quando isso ocorrer, a função **pega o mesmo valor que lhe foi devolvido, e o devolve** a que a chamou. Na linha 10 o mesmo é feito para o lado direito.

Exercício 3 - Teste da rotina (quase resolvido)

Inclua no programa principal uma nova opção no *switch....case...*, para testar a rotina que você fez no exercício anterior.

O trecho que você deve inserir, deve ser parecido com este :

```
. . .
case 4: printf(" Dado a ser procurado:");
        scanf("%d",&X);
        end = Busca_Endereco (Raiz,X);
        if (end == NULL)
             printf(" \n %d nao foi encontrado.\n",X);
        else printf(" \nOk = %d\n",end->item);
        break;
. . .
```

Digite e teste a rotina de Busca.

Exercício 4 - Remoção de dados : Remover (...)

Implemente um algoritmo para remover um dado de uma árvore binária, conforme o algoritmo descrito em 1.1.3.
Essa rotina recebe como parâmetro **o valor do nó** que deve ser removido.

Exercício 5 - Teste da rotina de remoção

Faça um trecho no programa principal, que chama a rotina de busca. Assim você poderá testar a sua rotina.
Dica : Não se esqueça de que ela deve retornar algo que diz se conseguiu remover ou não.

Exercício 6 - Mini-Projeto I (opcional)

Estimativa de trabalho – entre 5 e 10 horas-homem/mulher.
O objetivo deste projeto é construir um pequeno banco que será armazenado em memória principal, baseado em árvores binárias. Para isso, vamos admitir algumas condições :

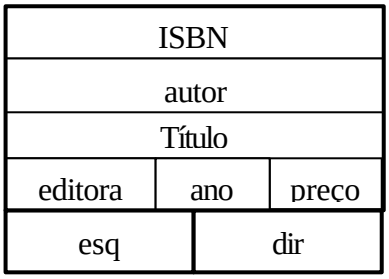
- Cada nó contém um registro de dados, e cada registro contém os seguintes campos :
  - ◊ Código ISBN do livro : numérico, de 10 dígitos (**Campo chave**)
  - ◊ Nome do autor ou autores : alfabético, de 50 posições
  - ◊ Título do livro : alfabético, de 50 posições
  - ◊ Editora : alfabético, de 15 posições
  - ◊ Ano de publicação : numérico
  - ◊ Preço de venda : real

- A estruturas de dados devem estar assim declaradas :

```
struct RegLivro {
      long int ISBN;
      char autores[50];
      char titulo[50];
      char editora[15];
      int ano;
      float preco;
};

typedef struct ReegLivro TipoItem;
struct NO {
    TipoItem item; /*o item contem um registro completo de um livro*/
    struct NO *esq, *dir;
};
typedef struct NO *PONT;
```

- Veja como seria o leiaute de cada nó :



- O ISBN é utilizado como campo chave, ou seja, a árvore binária é organizada segundo o número de ISBN (menores à esquerda, maiores à direita).
- Implemente as seguintes rotinas :
  - ◊ **CadastrarLivro (*Raiz, Livro*)** : Inclui os dados contidos no registro ***Livro*** na árvore binária cujo nó raiz é apontado pela variável ***Raiz***.
    Devolve : SUCESSO ou FRACASSO, indicando o resultado da operação. FRACASSO pode ocorrer, tanto na impossibilidade de alocar mais memória, quanto na tentativa de cadastrar um livro com ISBN repetido (não se deve permitir dois livros cadastrados com mesmo ISBN).
  - ◊ **ProcurarLivro (*Raiz, IsbnProcurado*)** : Procura um livro cujo ISBN é igual a ***IsbnProcurado***, na árvore binária cujo nó raiz é apontado pelo argumento ***Raiz***.
    Devolve : um ponteiro para o nó onde foi encontrado o registro do livro que contém esse ISBN, ou um ponteiro nulo (NULL) caso não encontre um livro com tal ISBN.
  - ◊ **ExibirLivro (*Endereco*)** : Recebe o ponteiro ***Endereco*** que aponta para um nó da árvore, e mostra todos os campos desse nó na tela.
    Obs.: Para procurar um livro na árvore e mostrar seus dados, primeiro você deve chamar a rotina **ProcurarLivro (*Raiz, IsbnProcurado*)**, que irá lhe devolver o endereço do nó que contém o livro que você procura. Aí, você pega esse endereço e envia para a função **ExibirLivro(...)**, que vai mostrar os dados na tela.

- ◊ **RemoverLivro (*Raiz, Dado*)** : Remove o nó cujo valor é indicado por ***Dado***, da árvore binária cujo nó raiz é apontado por ***Raiz***.
  Devolve : nada.
- ◊ **ListarAcervo (*Raiz*)** : Percorre a árvore apontada por ***Raiz***. Você pode, por exemplo, adotar a forma INORDEM (Ver item 1.3.2) para fazer isso, de modo que os livros sejam exibidos por número de ISBN em ordem crescente.

Exercício 7 - Ordenação por título.

No exercício anterior, a rotina **ListarAcervo ( )** provavelmente irá exibir os livros por ordem crescente de ISBN. Discuta alternativas para exibir os livros por ordem alfabética de autor ou de título. Seria necessário utilizar outra estrutura de dados para nos auxiliar ? Como percorrê-la(s) para cumprir o objetivo ?
Não é preciso implementar, mas apenas discuta o problema, apresentando uma ou mais sugestões de como fazer isso. Esboce/desenhe como funcionaria sua solução.

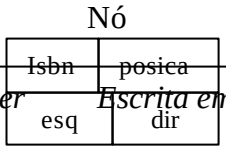
Exercício 8 - Mini-Projeto II (opcional)

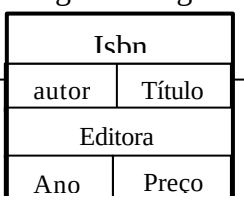
Estimativa de trabalho – entre 10 e 15 horas-homem/mulher.
Considere um problema parecido com o do Exercício 6, do cadastro de livros. Eu gostaria de separar os dados em duas estruturas de dados :

- **Um índice** : que seria formado pela árvore binária com o ISBN de cada livro cadastrado. Os outros campos de cada livro não estariam dentro da árvore binária, mas dentro de uma tabela de dados (um vetor de registros). Cada nó teria o ISBN de um livro e uma indicação sobre onde estão os dados desse livro na tabela de dados.
- **A tabela de dados** : seria um vetor, onde cada elemento é uma *struct* ***RegLivro*** tal como descrito abaixo.

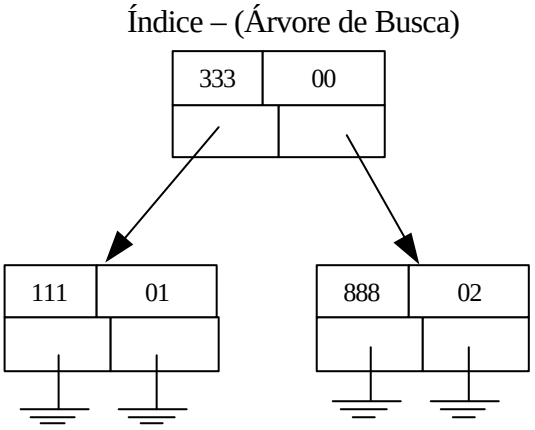
```
struct RegLivro {
      long int ISBN;
      char autores[50];
      char titulo[50];
      char editora[15];
      int ano;
      float preco;
};
struct NO {
    long int ISBN;  /* indicação do livro */
    int  posicao;   /* posição da tabela, onde esse livro está */
    struct NO *esq, *dir;
};
typedef struct NO *PONT;
struct RegLivro Tabela[100]; /* Tabela com capacidade para armazenar
                                até 100 livros */
int QuantLivros;  /* contador, que indica a quantidade real de livros
                     cadastrados na tabela –lembre-se que 100 é o máximo*/
```

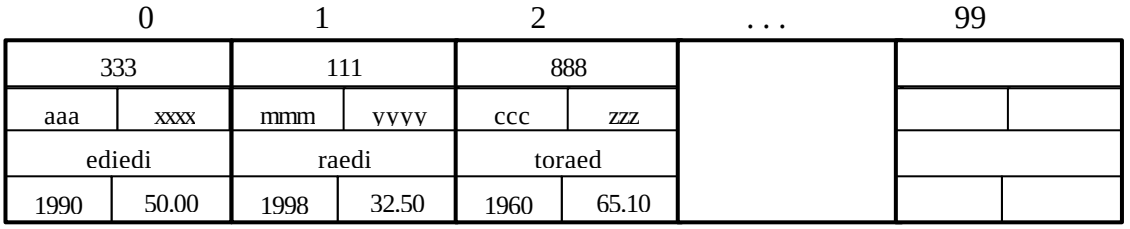
- Veja um esboço das estruturas de dados declaradas :





- E veja um exemplo de como essas estruturas trabalhariam :





- Implemente as seguintes funções :
  - ◊ **CadastrarLivro (*Raiz,Tabela,Livro*)** : Inclui os dados contidos no registro ***Livro*** na ***Tabela*** de dados. Cada vez que um registro é cadastrado em ***Tabela***, é preciso atualizar o índice, ou seja, incluir um novo nó na árvore binária cujo nó raiz é apontado pela variável ***Raiz***. Esse novo nó deve conter o valor campo chave do livro (o ISBN) e a posição que ele ocupa no vetor de dados.

    Devolve : SUCESSO ou FRACASSO, indicando o resultado da operação. FRACASSO pode ocorrer, tanto na impossibilidade de alocar mais memória (se a ***Tabela*** já estiver cheia), ou na tentativa de cadastrar um livro com ISBN repetido (não se deve permitir dois livros cadastrados com mesmo ISBN).
  - ◊ **ProcurarLivro (*Raiz,Tabela, IsbnProcurado*)** : Procura um livro cujo ISBN é igual a ***IsbnProcurado***, na árvore binária apontada por ***Raiz***. Quando encontrar, lembre-se que os dados do livro não estão dentro da árvore, mas sim dentro do vetor ***Tabela***. Dentro do nó da árvore você irá encontrar apenas o número da posição do vetor, onde buscar os dados do livro.

    Devolve : um ponteiro para o nó onde foi encontrado o registro do livro que contém esse ISBN, ou um ponteiro nulo (NULL) caso não encontre um livro com tal ISBN.
  - ◊ **ExibirLivro (*Endereco*)** : Recebe o ponteiro ***Endereco*** que aponta para um nó da árvore. Lembre-se esse nó não guarda os dados do livro, mas guarda um número (campo ***posicao***) que indica onde estão os dados do livro na ***Tabela***.

Obs.: Para procurar um livro na árvore e mostrar seus dados, primeiro você deve chamar a rotina **ProcurarLivro (*Raiz, IsbnProcurado*)**, que irá lhe devolver um endereço. Aí, você pega esse endereço e envia para a função **ExibirLivro(...)**, que vai mostrar os dados na tela.

- ◊ **RemoverLivro (*Raiz, Dado*)** : Essa rotina deve procurar o ***Dado*** na árvore cuja raiz é apontada pela variável ***Raiz***, e fazer o seguinte :
  - Descobrir qual a posição de ***Tabela*** na qual o livro se encontra;
  - ir lá no vetor e apagar os dados; e
  - apagar o nó da árvore onde ***Dado*** foi encontrado.
  - **Devolver** : nada.
- ◊ **ListarAcervo (*Raiz*)** : Percorre a árvore apontada por ***Raiz***. Você pode, por exemplo, adotar a forma INORDEM (Ver item 1.3.2) para fazer isso, de modo que os livros sejam exibidos por número de ISBN em ordem crescente.

Exercício 9 - Mini-Projeto III (opcional)

Estimativa de trabalho – entre 15 e 20 horas-homem/mulher.

No Exercício 8 cada registro de um livro consome aproximadamente 125 bytes. Isso porque não quisemos guardar outras informações adicionais sobre cada livro. Se quisermos armazenar 1.000 livros, por exemplo, o nosso vetor ***Tabela*** deveria consumir aproximadamente 125 Kbytes de memória.

Isso poderia piorar muito, se aumentarmos o tamanho do registro, ou a quantidade de registros, o que, na prática ocorre muito freqüentemente.

Que tal se nós quiséssemos montar uma livraria on-line, para vender através da Internet ? Imagine que eu queira cadastrar novos campos para cada livro, como por exemplo:

- o assunto ou assuntos com os quais o livro está relacionado;
- o link da página do(s) autor(es);
- o tempo que a livraria eletrônica leva para despachar esse livro para a casa do cliente;
- outros livros comprados pelas pessoas que também compraram este livro;
- etc. etc.

Qual seria o tamanho do meu registro ? 500 bytes ? Mais ? Menos ?

Não sei, mas acho uma péssima idéia guardar tudo isso em um vetor de dados. Imagine se a quantidade de livros aumentasse para 10 mil. O meu vetor poderia ter 10.000 x 500 = 50.000.000 bytes, ou seja aproximadamente 50 Megabytes.

A solução seria deixar os dados em um arquivo em disco.

- Crie um arquivo em disco, para armazenar os registros de livros. O arquivo deve ter as seguintes características:
  - ◊ Ter tamanho fixo de registro : pode ser conveniente usar as funções **fopen** (modo binário)**, fread e fwrite** para manipular esse arquivo.
  - ◊ Tratar o arquivo de modo análogo a um vetor, lendo e gravando registros através da posição que ele ocupa no arquivo. Ver função **fseek**.
- O índice, ou seja, a árvore de busca irá registrar a posição que cada registro ocupa dentro do arquivo.
- Implemente as funções:

- ◊ **CriarArquivo (...)** : Cria um novo arquivo e respectivo índice. Apaga possíveis dados que já existissem nesse arquivo.
- ◊ **GravaRegistro(...)** : Inclui um registro no arquivo, registrando-o no índice para busca rápida no futuro.
- ◊ **LeRegistro (...)** : Procura um determinado registro no arquivo, fazendo a busca rápida no índice, através do campo chave (ISBN).
- ◊ **ApagaRegistro (...)** : Remove um registro do arquivo e do índice, tomando o cuidado de deixar livre a área do disco que foi liberada. Outro registro que venha a ser incluído no futuro poderá ocupar essa posição que foi agora liberada.
- ◊ **AbreArquivo (...)** : Abre o arquivo para leitura e/ou expansão de dados. Note que, diferentemente da função **CriaArquivo( )**, esta função não deve destruir dados que pudessem previamente existir. Outro ponto importante é, no momento da abertura, é preciso reconstruir o índice corretamente. Há duas maneiras de fazer isso :
  - Ler seqüencialmente o arquivo de dados, e, a cada registro lido, registrar no índice (árvore binária) cada campo chave lido, bem como seu endereço físico no arquivo de dados. Isso pode tornar a operação **AbreArquivo(...)** muito lenta, conforme o tamanho do arquivo de dados cresce.
  - Outra alternativa é gerar um arquivo em separado, só para o índice : assim, cada vez que o arquivo é aberto, é preciso ler o conteúdo do arquivo de índice, reconstituindo a árvore binária. Ao fechar o arquivo, é preciso baixar o índice no disco, ou seja, gravar a árvore binária que serve como índice no arquivo, para recuperação futura. Como esse arquivo de índice só contém o campo chave e seu endereço no arquivo de dados, a função **AbreArquivo(...)** pode ficar mais eficiente. Porém, como teremos de gerar um arquivo de índice a cada fechamento de arquivo, para uso futuro, a função **FechaArquivo(...)** vai ficar mais lenta.
- ◊ **FechaArquivo (...)** : Fecha o arquivo de dados, atualiza e fecha também o arquivo de índice (caso ele exista).

3 LISTA DE EXERCÍCIOS - I

Tente resolver os exercícios contidos nesta lista. Muitos deles foram exercício de prova em anos anteriores.
Por aqui, você pode ter uma idéia do que pode ser cobrado em avaliações sobre o assunto.

Exercício 1 - Construção de uma ABB.

Construa uma árvore binária de busca com os dados abaixo, obedecendo a seqüência dada:

33, 25, 14, 29, 46, 37, 78, 24, 26, 2, 98, 11, 18, 27, 32, 57, 85, 40, 52

Obs.: Faça um novo desenho a cada 3 ou 4 inserções.

Exercício 2 - Remoção de dados de uma ABB

Depois de construída, remova os dados:

14, 37, 98, 27, 25 ,11e 57

Obs. Desenhe novamente a árvore, no máximo a cada duas remoções.

Exercício 3 - Algoritmos de Percursos

A partir da árvore que você construiu no Exercício 1, dê sua PRE-ORDEM, IN-ORDEM e PÓS-ORDEM.

Exercício 4 - Aplicações de percurso

Observe a resposta do Exercício 3. Você notou algo especial em uma das três ordens ? Explique .

Exercício 5 - Algoritmo de percurso

Construa uma rotina para percorrer árvores binárias em PRE-ORDEM inversa, ou seja, visite sempre a subárvore direita antes de visitar a esquerda.
Faça o mesmo com a IN-ORDEM e POS-ORDEM.

Exercício 6 - Reconstrução da árvore

Ao se percorrer uma certa árvore temos :

PRE-ORDEM $\Rightarrow$ **A B D G H E C F I J K**
IN-ORDEM $\Rightarrow$ **G D H B E A I F K J C**

Desenhe como seria essa árvore.

Exercício 7 - Propriedades de árvores

Qual o número máximo de comparações para se localizar um dado em uma árvore binária perfeitamente balanceada com aproximadamente :

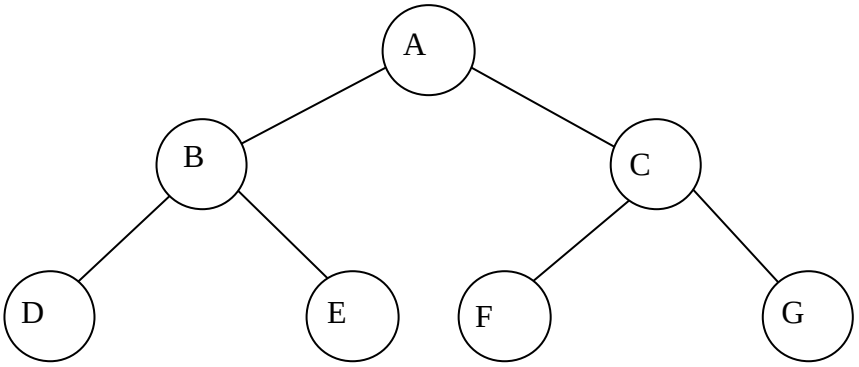
a) 500 nós
b) 2000 nós
c) 1500 nós

Exercício 8 - Propriedade de árvores

Quantos níveis são necessários para armazenar 8000 nós em uma árvore binária.

Exercício 9 - Algoritmo

Considere a árvore binária abaixo :



Simule a rotina abaixo, mostrando a saída em tela gerada.

```
void XYZ ( PONT R, int H)
{
    int I;
    if ( R != NULL)
      {
         XYZ ( R->ESQ, H+1);
         for ( I = 1, i <= H; i++) printf (“  “);
         printf(“%c \n”,R->ITEM);
         XYZ ( R->DIR, H+1);
      }
}
```

Exercício 10 - Algoritmo

Faça uma rotina de inserção recursiva para uma árvore binária de busca, onde cada nó possui os seguintes campos :

INFO	informação contida no nó
QUANT	quantidade de vezes que a informação foi inserida

ESQ, DIR ponteiros para as duas sub-árvores

A rotina deve inserir um novo dado X na árvore. Caso esse dado já exista, simplesmente armazene a informação de que houve uma nova inserção. Caso X esteja sendo inserido pela primeira vez, isto é, ainda não existia na árvore, então crie um novo nó indicando que existe uma unidade do item X.

Exercício 11 -

É possível utilizar árvores binárias para ordenar um conjunto de dados de forma crescente ? Explique.

Exercício 12 -

Construa uma rotina em C/C++ para comparar duas árvores binárias. A rotina deve receber como parâmetros de entrada dois ponteiros (que apontam para a raiz de cada uma das árvores). Em seguida, deve comparar se o conteúdo das duas árvores são exatamente iguais. Devolver o valor TRUE caso sejam iguais e FALSE caso não sejam (em linguagem C pode utilizar 0 ou 1).

Exercício 13 -

Implemente uma rotina que verifica se uma árvore binária é arvore binária de busca ou não. Lembre-se da definição de Arv. Bin. de Busca :

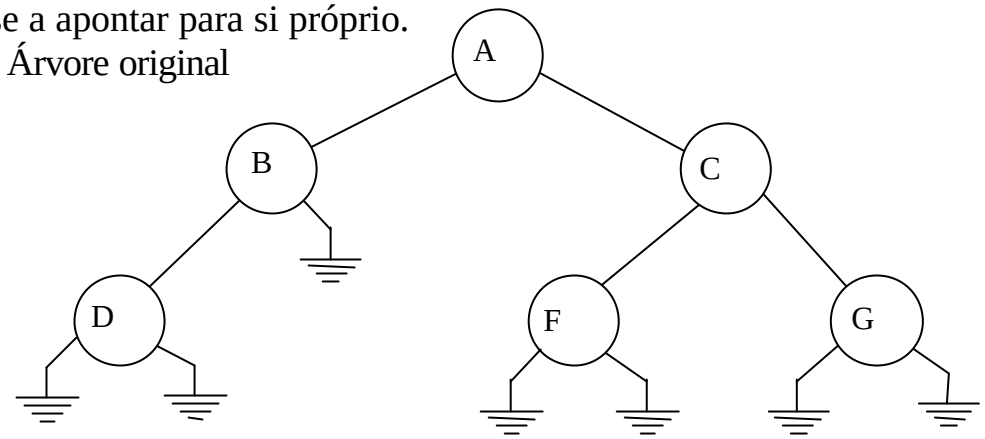
- Os dados com valor menor que a informação da raiz estarão à sua ESQUERDA
- Os dados com valor maior que a informação da raiz estarão à sua DIREITA.

Exercício 14 -

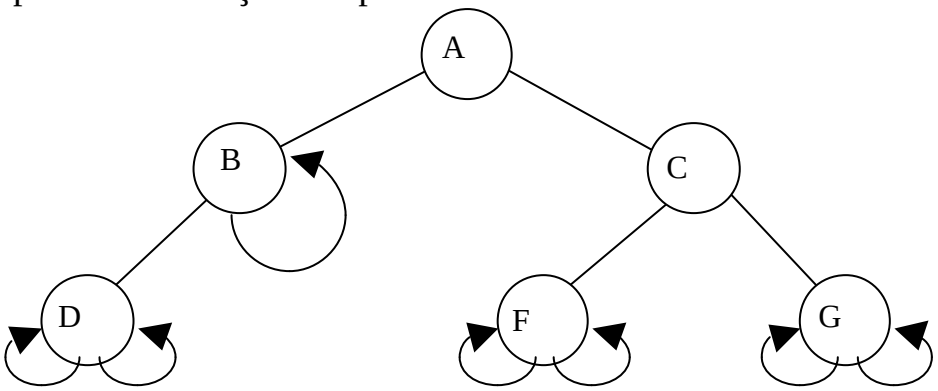
Implemente uma rotina que recebe um ponteiro que aponta para a raiz de uma árvore binária. A rotina deverá contar quantos nós existem nessa árvore, e devolver esse resultado (a quantidade) como um número inteiro.

Exercício 15 -

Implemente uma rotina que recebe um ponteiro para o nó raiz de uma árvore. A rotina deverá fazer com que todo nó que tiver um ponteiro com o valor NULL passe a apontar para si próprio.
Ex : Árvore original



Após a modificação dos ponteiros ela deverá ficar assim :



4 LISTA DE EXERCÍCIOS – II (Exclusiva para disciplina de Análise de Algoritmos)

Exercício 1 - (Resolvido)

Implemente um algoritmo que lê uma seqüência de dados digitados de forma aleatória, e constrói uma lista ligada em ordem crescente.

Solução:

Para resolver esse problema, eu vou fazer a coisa em duas etapas :

- *Primeiro, faço o loop de leitura de dados, mandando inserir cada dado na lista.*

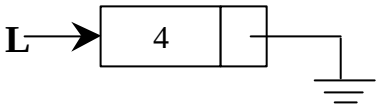
```
...
L = NULO;
faça
    ler_teclado(X);
    INSERIR_ORDENANDO(L,X);
enquanto ( ... );
```

Imagine que seja digitada uma seqüência aleatória de valores, como a abaixo :

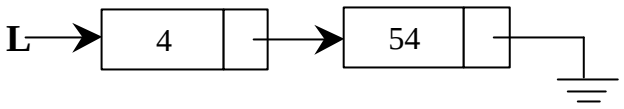
4, 54, 17, 62, 174,...

Depois eu crio uma função, que faz o seguinte :

- *Na primeira vez que for chamada : a lista estará vazia* (*L* = NULO) *e* ***X*** *valerá* ***4****. A função simplesmente inclui o valor na lista, que ficará assim*:



- *Na segunda vez* : *a lista terá um nó, e* ***X*** *valerá* ***54****. A função simplesmente inclui o novo valor depois do nó existente, por que* ***54*** *é maior que* ***4****:*



- *Nas próxima vez, o* ***17*** *deve entrar entre os valores* ***4*** *e* ***54****, e assim por diante.*

Então vamos generalizar o processo. A função INSERIR_ORDENANDO(*L,X*) irá *:*

- *Receber como parâmetros de entrada uma lista **L**, e um dado **X**.*
- *Criar um novo nó para guardar o valor **X**.*
- *Se a lista **L** recebida estiver vazia, simplesmente inclui **X** como sendo o único elemento.*
- *Se a lista **L** não estiver vazia, então há duas possibilidades :*
  - ◊ *Se o novo dado **X** é menor do que o primeiro dado da lista (e portanto, também é menor do que todos os outros), então eu acrescento esse dado no primeiro lugar da lista.*
  - ◊ *Se **X** não é menor que o primeiro da lista, então eu devo percorrer a lista, e achar o ponto exato onde **X** deve ser incluído. Considere que a lista já tenha recebido **n** valores $x_1, x_2, ..., x_n$, eu preciso encontrar o lugar **i** (**0** < **i** < **n**) da lista, tais que $x_i < X < x_{i+1}$.*

Finalmente, eis aqui o algoritmo da função:

```
INSERIR_ORDENANDO(L,X)
   novo = alocar_memória ( ... );
   novo→item = X;
   se ( L = NULO )
      então L = novo;
   senão
      se ( X < L®item )
         então /* se X é menor que o primeiro ... */
            novo→proximo = L;
            L = novo;
         senão
            p = L ;
            q = NULO;
            enquanto ( p→proximo ≠ NULO ) E ( p→item < X)
               q = p;
               p = p ®proximo;
            novo→proximo = p;
            q→proximo = novo;
```

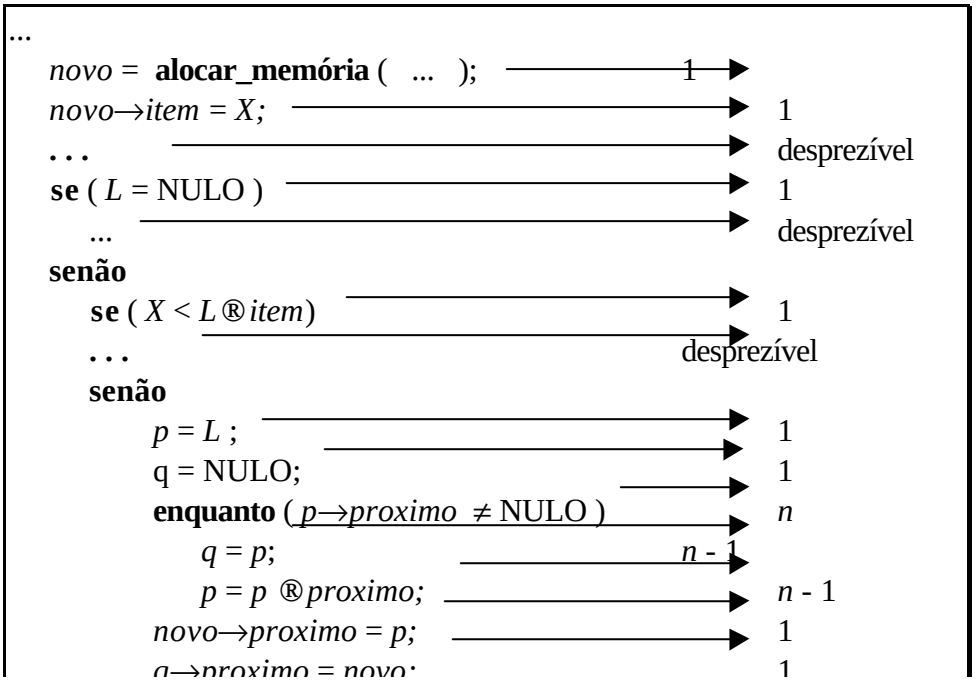
Exercício 2 - (resolvido)

Qual é o tempo de execução dessa função ?

Solução:

- Considerando que a função recebe uma lista ***L***, e um dado ***X***, vamos calcular o tempo de execução pessimista, pelos motivos que foram apresentados no primeiro semestre.
- Ser pessimista significa considerar que a lista ***L*** contém ***n*** registros $\{l_1, l_2, \ldots, l_n\}$, e no pior caso, $X > l_n$ e portanto, deve ser incluído na última posição de ***L***.

- Portanto, vou desprezar os casos onde ***L*** está vazia, ou que $X < l_1$.



Assim, $T(n) = 1 + 1 + 1 + 1 + 1 + 1 + n + (n\text{-}1) + (n\text{-}1) + 1 + 1$

$\therefore\ T(n) = 3n + 6$

Exercício 3 -

A função INSERIR_ORDENANDO (...) é apenas uma parte do algoritmo do Exercício 1, e seu tempo de execução pessimista foi estimado no Exercício 2. Calcule o tempo de execução total, para o algoritmo completo, ou seja, considere o tempo gasto no loop principal de ordenação.

Dica:

Considere que ***S*** é uma seqüência $s_1, s_2, \ldots, s_m$, de comprimento ***m***, composta de dados sorteados ao acaso, e digitados como entrada para o algoritmo em questão.

Calcule $T(m)$, sabendo que o tempo de execução da linha que contém a chamada para a função INSERIR_ORDENANDO (...) é $3n+6$, onde n é o comprimento da lista naquele momento.

Exercício 4 -

Prove ou desprove que :

a) $T(n)$ calculado no Exercício 3 é $O(n^2)$
b) $T(n)$ calculado no Exercício 3 é $O(n)$. Se provar que isto é verdade, das duas uma : ou você errou, ou pode ficar rico (ou pelo menos famoso), pois descobriu e implementou um algoritmo de ordenação de dados com tempo de execução pessimista **linear**.
c) $T(n)$ calculado no Exercício 3 é $\Omega(n^2)$

Exercício 5 -

Você percebeu que acabou de construir um algoritmo de ordenação de dados baseado em listas ligadas ? Pois é, além dos tradicionais algoritmos como **BubbleSort**, ..., **MergeSort, ShellSort, QuickSort**, existe uma infinidade de outros. Que tal batizá-lo de *ListSort ?*

Tente melhorar um pouco a eficiência desse algoritmo. O loop que busca o lugar correto para a inserção de ***X*** na lista pode ser melhorado. Dá para utilizar um único ponteiro para percorrê-la, ao invés de dois (*p* e *q*).

Para quanto foi a complexidade (O novo $T(n)$) ? Você acha que valeu a pena quebrar a cabeça para obter essa nova solução ? Por que ?

Exercício 6 - (Resolvido)

No item 1.4 nós analisamos a eficiência do algoritmo de busca em uma ABB. Analise novamente, porém, com mais detalhes.

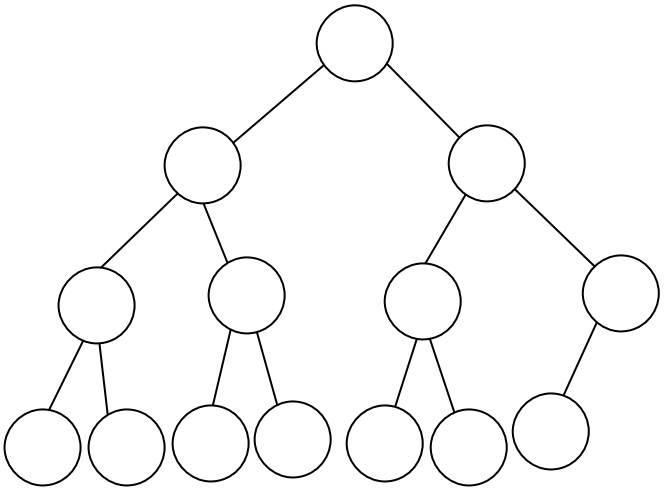
Essa questão é um tanto complexa, mas vamos tentar facilitar um pouco o trabalho. Considere uma certa árvore binária "cheia", a menos do último nível que pode estar incompleto em um nó, pois do contrário só poderíamos ter valores de ***n*** ímpar.

Uma árvore está "cheia", se :

- todos os nós internos (não folha) possuem dois filhos, e
- todos os nós folha estão no mesmo nível.

Solução :

Antes vamos entender como é essa árvore. Este é um exemplo :



Vejamos o algoritmo recursivo, de busca na árvore. Para a análise do pior caso, imagine que ele nunca entra no primeiro **se**, *e ao contrário, entra no último. Assim, todas as comparações consomem tempo igual a 1, e que só precisamos determinar a quantidade de chamadas recursivas que serão feitas.*

```
1 - BUSCA_ENDEREÇO (Raiz, Dado)
2 -   se (Raiz = NULO)  ------------------------------------------> 1
3 -     então devolva NULO
4 -     senão se (Dado = Raiz→item)  -------------------------------> 1
5 -               então devolva Raiz
6 -               senão  se (Dado > Raiz→item) -------------------> 1
7 -                         então devolva PROCURA_ABB (Raiz®dir, Dado)
8 -                         senão devolva PROCURA_ABB (Raiz®esq, Dado)----> T(n/2)
```

Note que, na linha 8 há uma chamada recursiva, e eu não sei quantas vezes isso vai se repetir, até que o algoritmo encontre o dado. Eu só sei que essa chamada recursiva irá reduzir o tamanho do problema, de ***n*** *para* ***n/2****, e portanto, o tempo requerido pelo computador para executar essa chamada é* **T(*n*/2)**. *Temos aí uma "recorrência" :*

$$T(n) = 3 + T(n/2)$$

Nesse caso, vou supor que para um certo k inteiro e positivo :

$$n = 2^k$$

Agora, vou tentar "desenrolar" a recorrência :

$$\begin{aligned} T(n) = T(2^k) &= 3 + T(2^k/2) &&= 3 + T(2^{k-1}) \\ &= 3 + [\,3 + T(2^{k-1}/2)\,] &&= 3*2 + T(2^{k-2}) \\ & &&= 3*3 + T(2^{k-3}) \\ & &&= 3*4 + T(2^{k-4}) \\ & &&\quad ... \\ & &&= 3*k + T(2^{k-k}) \\ & &\therefore\ T(n) &= 3k \end{aligned}$$

Mas eu não quero T(*n*) *em função de k, e sim em função de n.*
Se $n = 2^k \Rightarrow k = \log_2 n$,
\ *Assim, substituindo k teremos* $T(n) = 3 \log_2 n$,
onde *n é a quantidade de nós da árvore.*

Exercício 7 -

Prove ou desprove que :

a) $T(n)$ calculado no Exercício 6 é $O(n^2)$
b) $T(n)$ calculado no Exercício 6 é $O(n)$.
c) $T(n)$ calculado no Exercício 6 é $\Omega(n^2)$

Exercício 8 -

Faça o mesmo para :

a) O algoritmo de Inserção, dado em 1.1.1

Dica :

O tempo para de inserção de um dado é o mesmo tempo requerido para a busca malsucedida (de um dado que não existe), mais uma comparação (com uma subárvore esquerda ou direita que é nula), e algumas linhas comuns para inserir o dado.

b) O algoritmo de Remoção, dado em 1.1.3

Exercício 9 -

a) Desenhe uma árvore binária de busca com os dados abaixo. Vá colocando os dados na mesma ordem que aparecem na seqüência abaixo :

33, 25, 14, 29, 46, 37, 78, 24, 26, 2, 98, 11, 18, 27, 32, 57, 85, 40, 52

b) Mostre os dados dessa árvore, de acordo com a IN-ORDEM, dada em 1.3.2.Você notou que os dados aparecem em ordem crescente ?

c) Construa um algoritmo para ler uma seqüência aleatória de dados, ordenando-os, utilizando uma ABB.

Dica: Vá lendo os dados um a um pelo teclado e inserindo na árvore. Depois de pronta, chame a função que percorre a árvore de acordo com a IN-ORDEM, dada em 1.3.2. Essa rotina irá exibir os dados em ordem crescente.

d) Qual o tempo de execução $T(n)$ desse seu algoritmo ? Não se esqueça de contar o tempo de construção da árvore, mais o tempo para percorrê-la.

Exercício 10 -

Prove ou desprove as seguintes afirmações :

a) $T(n) = O(n)$

b) $T(n) = O(n \log_2 n)$

c) $T(n) = O(n^2)$

Exercício 11 -

Nesta lista de exercícios você viu dois algoritmos de ordenação de dados : o algoritmo dado no Exercício 1 e analisado no Exercício 3, e este último dado e analisado no Exercício 9.

Considere que o seu computador realiza uma operação a cada microssegundo ($1\mu s = 10^{-3}$ segundos), e compare a eficiência dos dois algoritmos, dizendo quanto tempo cada um leva para ordenar um conjunto de n dados.:

Anote aqui o valor de $T(n)$ que você achou para cada algoritmo, e depois complete a tabela

		Quantidade de dados para ordenar (n)					
Algoritmo	$T(n)$	10	100	1.000	100.000	1 milhão	10 milhões
c/ listas							
c/ árvores							

Exercício 12 -

Suponha que existam 5 algoritmos de busca, cada qual com seu tempo de execução $T(n)$. Suponha que o meu computador realiza uma operação a cada microssegundo ($1\mu s = 10^{-3}$ segundos), e calcule quanto tempo o algoritmo deverá levar para encontrar um dado em meio a uma quantidade n de registros.

		Quantidade de dados (n)					
Algoritmo	$T(n)$	10	100	1.000	100.000	1 milhão	10 milhões
Árv. binária	$log_2\, n$						
árv. ternária	$log_3\, n$						
árvore-B (grau 5)	$log_5\, n$						
árvore-B (grau 20)	$log_{20}\, n$						
tabela de espalhamento *hashing*	$\frac{1}{2}[1+1/(1-\alpha)]$*						

* O símbolo α representa algo parecido com "o fator de carga de uma tabela *hashing* ...", e portanto, para nós não possui significado prático nenhum. Apenas para concluir o exercício, considere que α vale 0,5 (meio).