Árvores de Pesquisa

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.

- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:

  1. Acesso direto e seqüencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

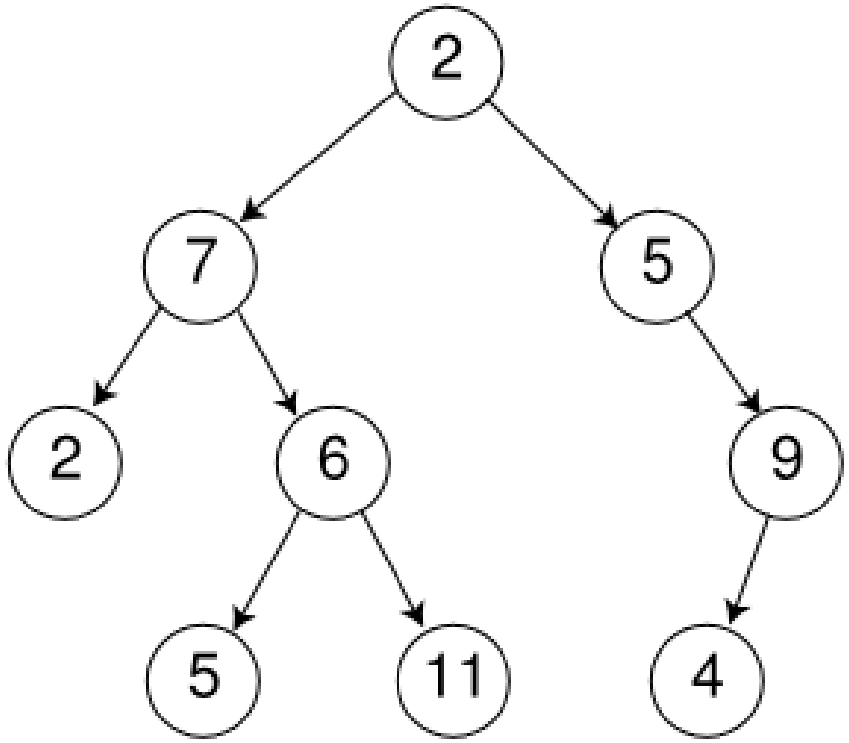
Árvores

- Estrutura hierárquica muito utilizada em ciência da computação para organização de elementos
- Cada elemento é chamado de nó
- Relação: pai – filhos
  - Raiz: primeiro da hierarquia
  - Folhas: nós que não tem filhos
- Recursividade: o filho de um nó é a raiz de uma subárvore

Exemplo

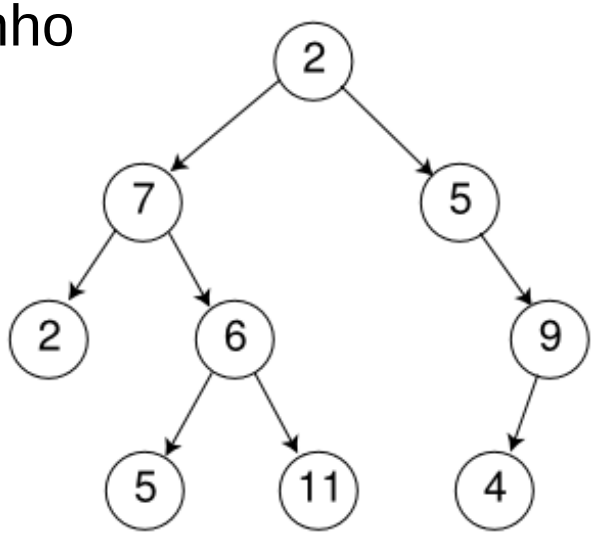
- Árvore Binária
  - Cada nó tem no máximo dois filhos

- Obs: a árvore ao lado não impões nenhuma ordenação em seus nodos



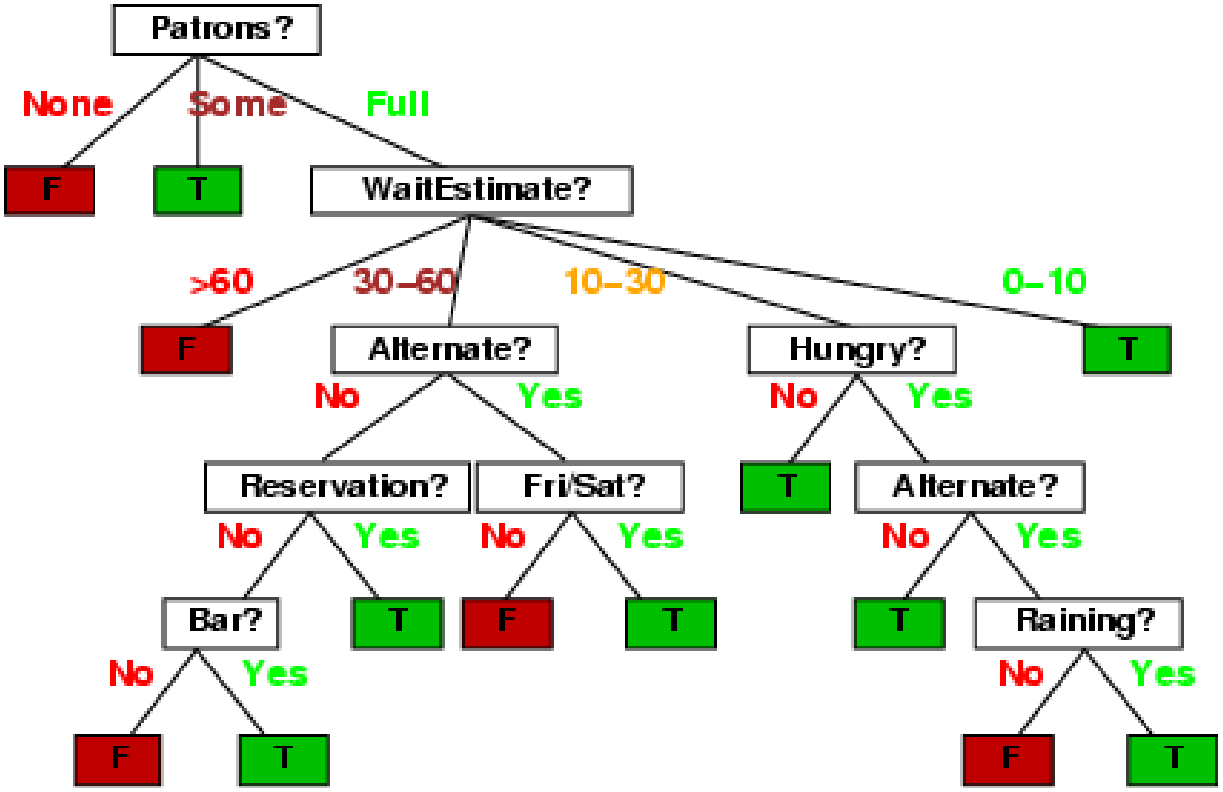
Árvores

- O nível do nó raiz é 0.
- Se um nó está no nível i então a raiz de suas subárvores estão no nível i + 1.
- Caminho: ligação entre quaisquer dois nós
  - Em uma árvore, só existe um caminho entre quaisquer dois nós: grafo sem ciclos!
- Ramo: caminho que termina numa folha
- A altura de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
- A altura de uma árvore é a altura do nó raiz.
- Nós internos x Nós externos



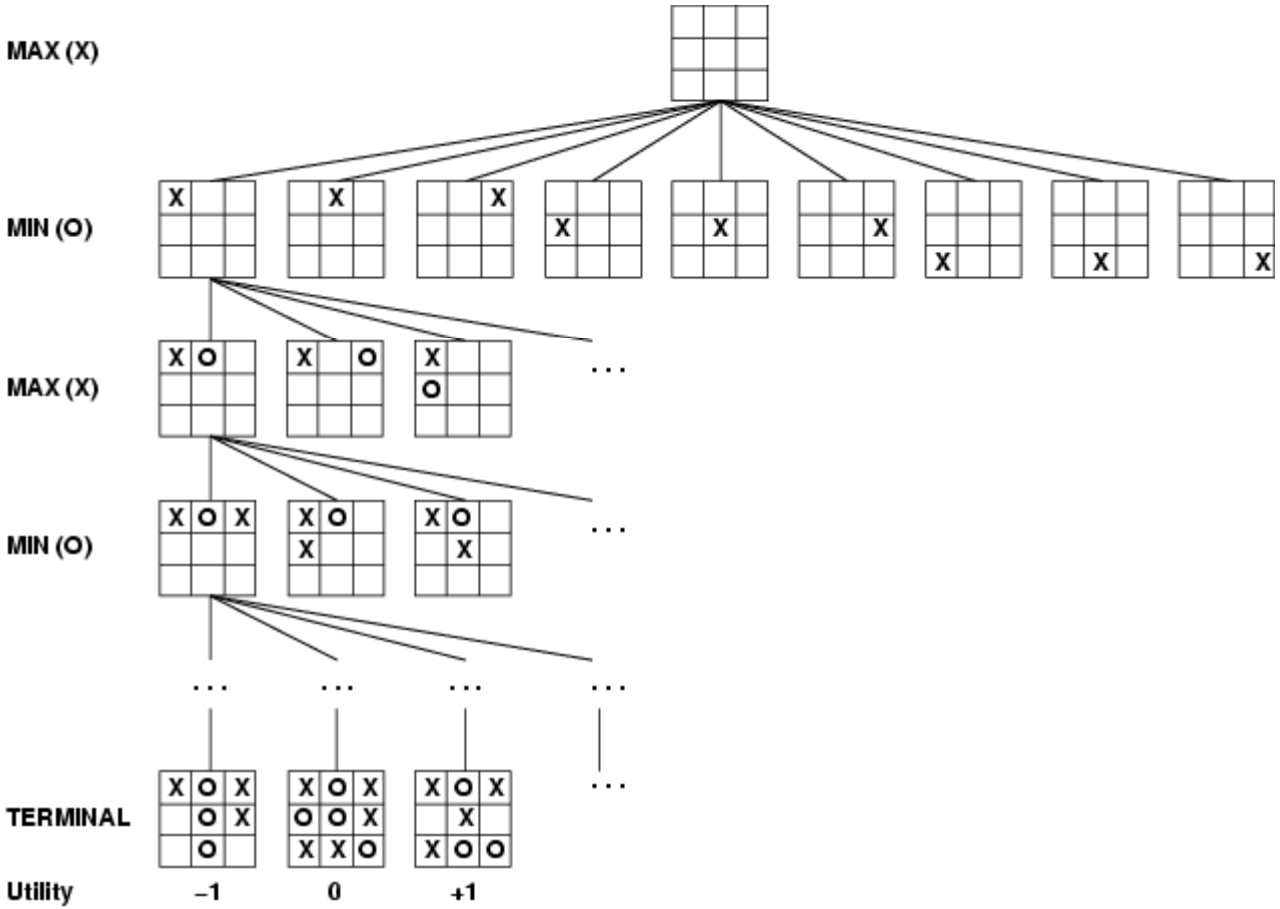
Exemplos

- IA: Árvores de Decisão



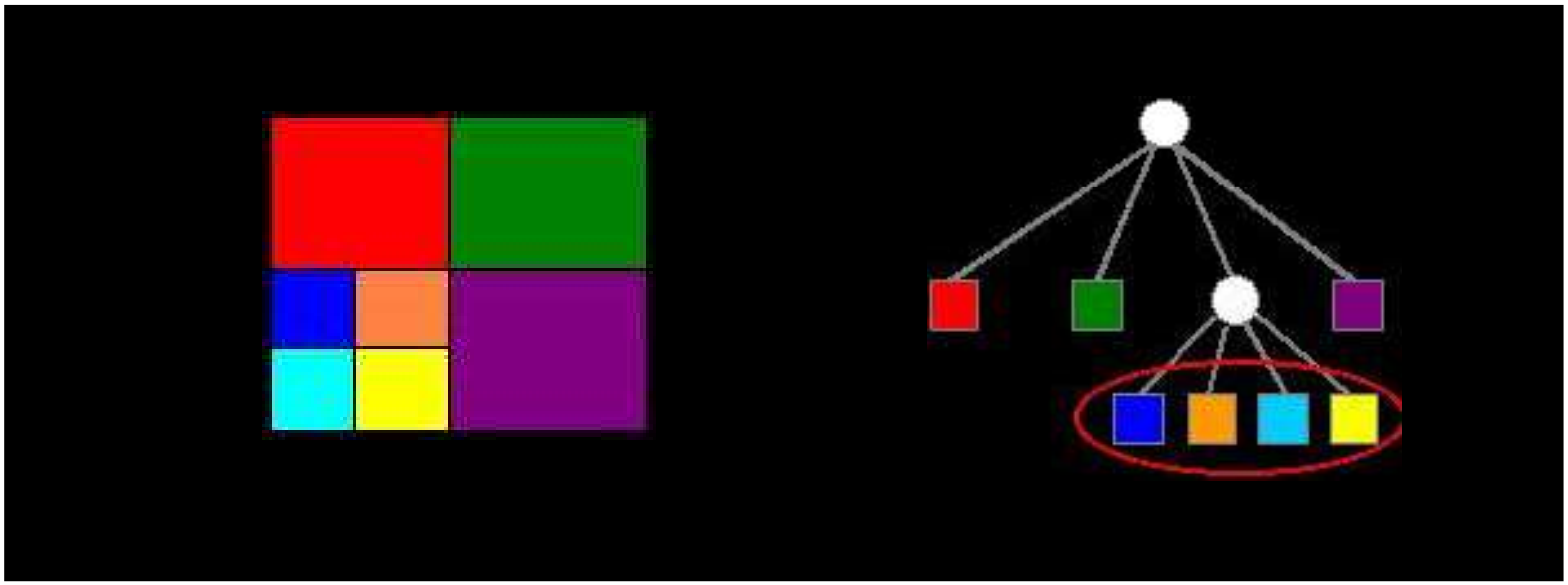
Exemplos

- IA, Jogos: Minimax



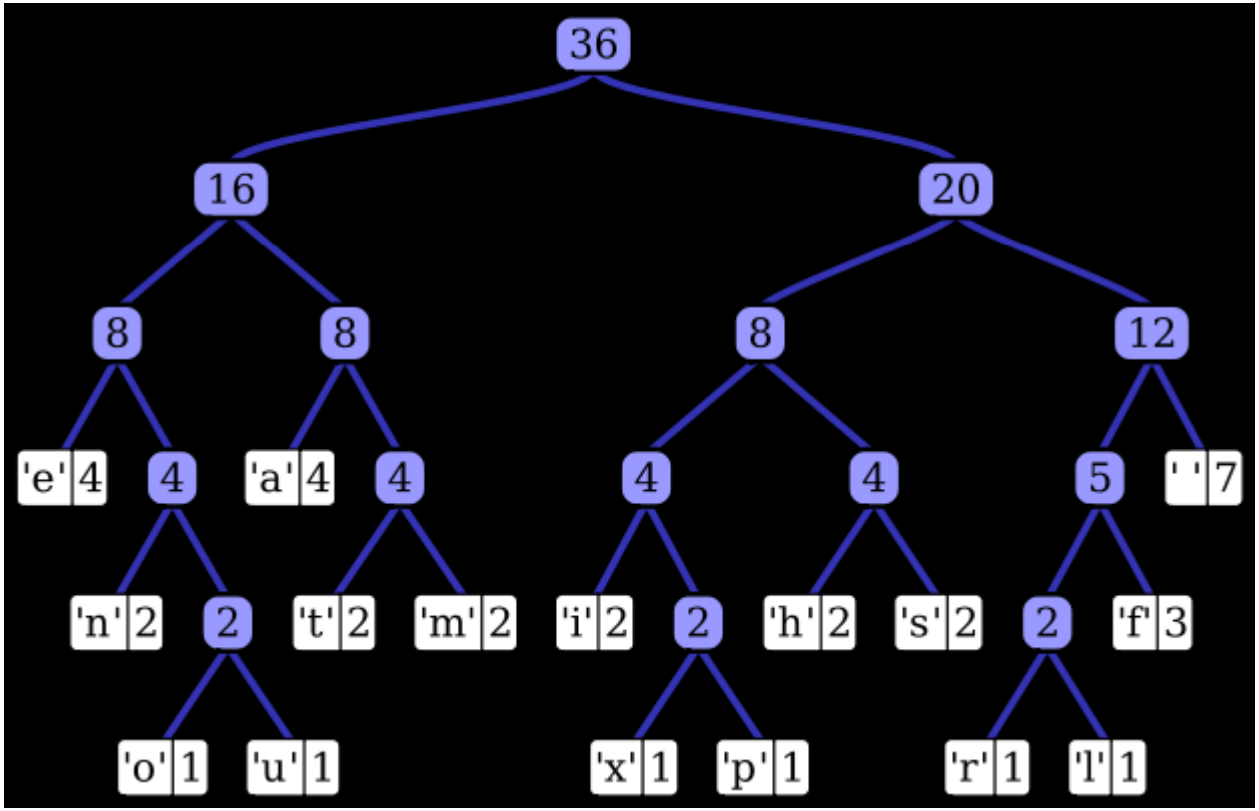
Exemplos

- PDI, Visão: quadtree



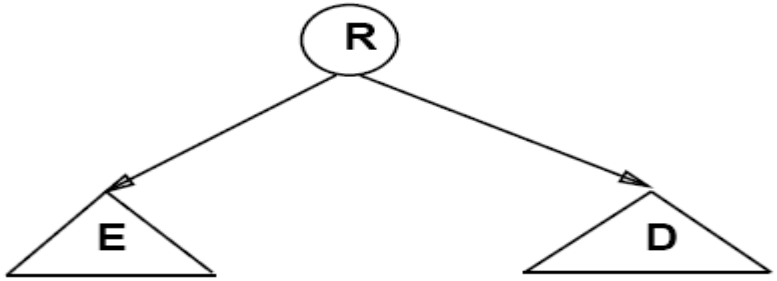
Exemplos

- Compressão de dados: Árvore de Huffman



Árvores Binárias de Pesquisa sem Balanceamento

- Para qualquer nó que contenha um registro



- Temos a relação invariante

1. Todos os registros com chaves menores estão na subárvore à esquerda.
2. Todos os registros com chaves maiores estão na subárvore à direita.

Implementação do Tipo Abstrato de Dados Dicionário usando a Estruturade Dados Árvore Binária de Pesquisa

- Estrutura de dados:

```
typedef long TipoChave;

typedef struct Registro {
    TipoChave Chave;
    /* outros componentes */
} Registro;

typedef struct No * Apontador;
```

```
typedef Struct No {
        Registro Reg;
        Apontador Esq, Dir;
} No;

typedef Apontador TipoDicionario;
```

Procedimentos para Inicializar e Criar a Árvore

```
void Inicializa(Apontador *Dicionario)
{
*Dicionario = NULL;
}
```

Procedimento para Pesquisar na Árvore

- Para encontrar um registro com uma chave x:
  - Compare-a com a chave que está na raiz.
  - Se x é menor, vá para a subárvore esquerda.
  - Se x é maior, vá para a subárvore direita.
  - Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha seja atingido.
  - Se a pesquisa tiver sucesso então o conteúdo do registro retorna no próprio registro x.

Procedimento para Pesquisar na Árvore

```
void Pesquisa(Registro *x, Apontador p) {
    if (p == NULL) {
        printf("Erro : Registro nao esta presente na arvore\n");
   else  if (x->Chave < p->Reg.Chave)
            Pesquisa(x, p->Esq);
   else if (x->Chave > p->Reg.Chave)
            Pesquisa(x, p->Dir);
   else
            *x = p->Reg;
}
```

Procedimento para Inserir na Árvore

- Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
- O apontador nulo atingido é o ponto de inserção.
- Obs. O apontador deve ser passado por referência para poder ligar corretamente o novo nodo à árvore.

Procedimento para Inserir na Árvore

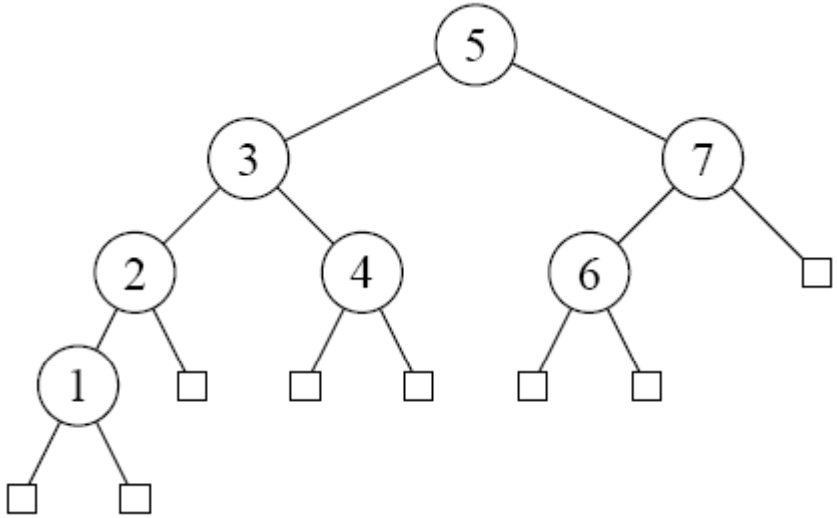
```
void Insere(Registro x, Apontador *p)  {
        if (*p == NULL)  {
                *p = (Apontador) malloc (sizeof(No));
                (*p)->Reg = x;
                (*p)->Esq = NULL;
                (*p)->Dir = NULL;
         }
         else if (x.Chave < (*p)->Reg.Chave)
                  Insere(x, &(*p)->Esq);
         else if (x.Chave > (*p)->Reg.Chave)
                  Insere(x, &(*p)->Dir);
         else
                  printf("Erro : Registro ja existe na arvore\n");
}
```

Procedimento para Retirar x da Árvore

- Alguns comentários:

1. A retirada de um registro não é tão simples quanto a inserção.
2. Se o nó que contém o registro a ser retirado possui no máximo um descendente ⇒ a operação é simples.
3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
   – substituído pelo registro mais à direita na subárvore esquerda;
   – ou pelo registro mais à esquerda na subárvore direita.

Exemplo da Retirada de um Registro da Árvore



- Assim: para retirar o registro com chave 5 da árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

Exemplo da Retirada de um Registro da Árvore

```
void Antecessor(Apontador q, Apontador *r)
{
    if ( (*r)->Dir != NULL)
    {
        Antecessor(q, &(*r)->Dir);
        return;
    }
    q->Reg = (*r)->Reg;
    q = *r;
    *r = (*r)->Esq;
    free(q);
}
```

Exemplo da Retirada de um Registro da Árvore

```
void Retira(Registro x, Apontador *p)
{   Apontador Aux;

    if (*p == NULL) {
         printf("Erro : Registro nao esta na arvore\n");
    }
    else  if (x.Chave < (*p)->Reg.Chave) {
              Retira(x, &(*p)->Esq);
     }
    else  if (x.Chave > (*p)->Reg.Chave){
               Retira(x, &(*p)->Dir);
    }
```

Exemplo da Retirada de um Registro da Árvore

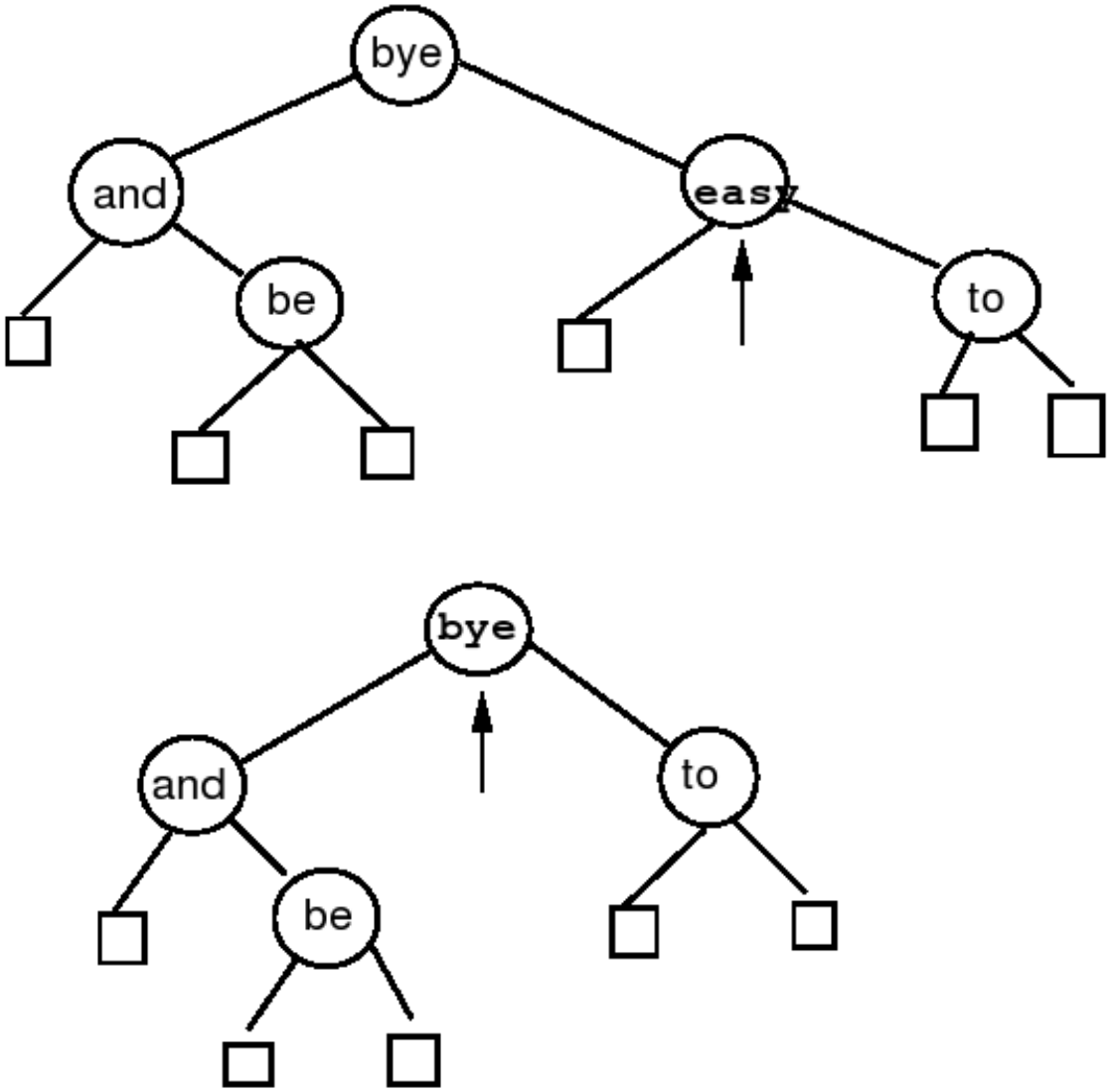
```
 else if ((*p)->Dir == NULL)  {
            Aux = *p;
            *p = (*p)->Esq;
            free(Aux);
}
else if ((*p)->Esq == NULL) {
           Aux = *p;
           *p = (*p)->Dir;
            free(Aux);
}
 else
       Antecessor(*p, &(*p)->Esq);
```

Exemplo da Retirada de um Registro da Árvore

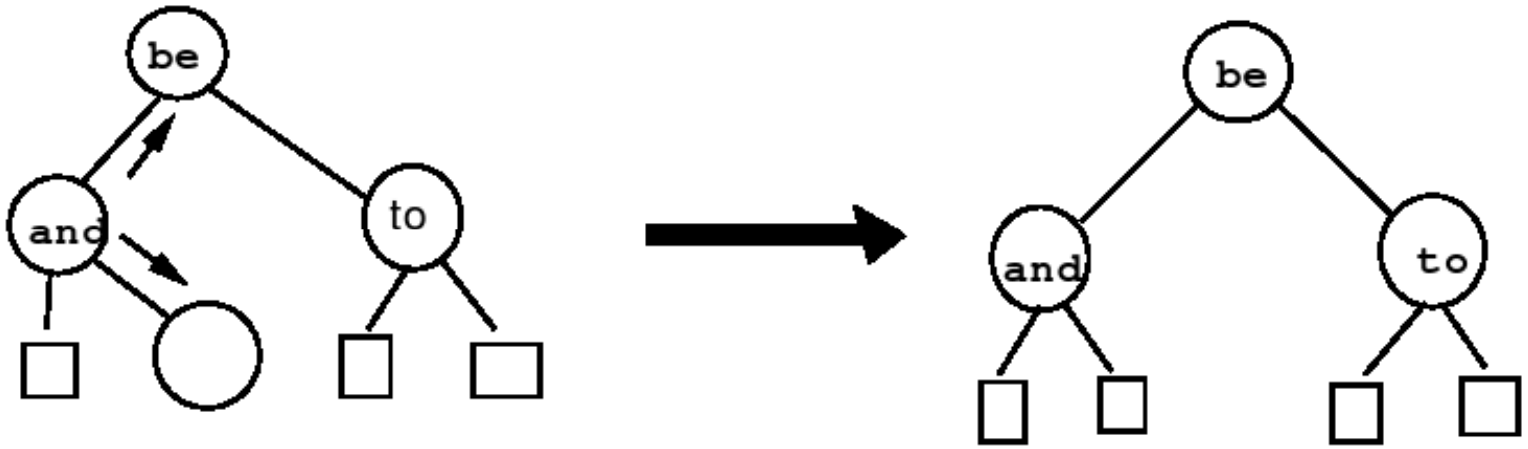
Obs.: proc. recursivo Antecessor só é ativado quando o nó que contém registro a ser retirado possui 2 descendentes.

Solução usada por Wirth, 1976, p.211.

Outro Exemplo de Retirada de Nó



Outro Exemplo de Retirada de Nó

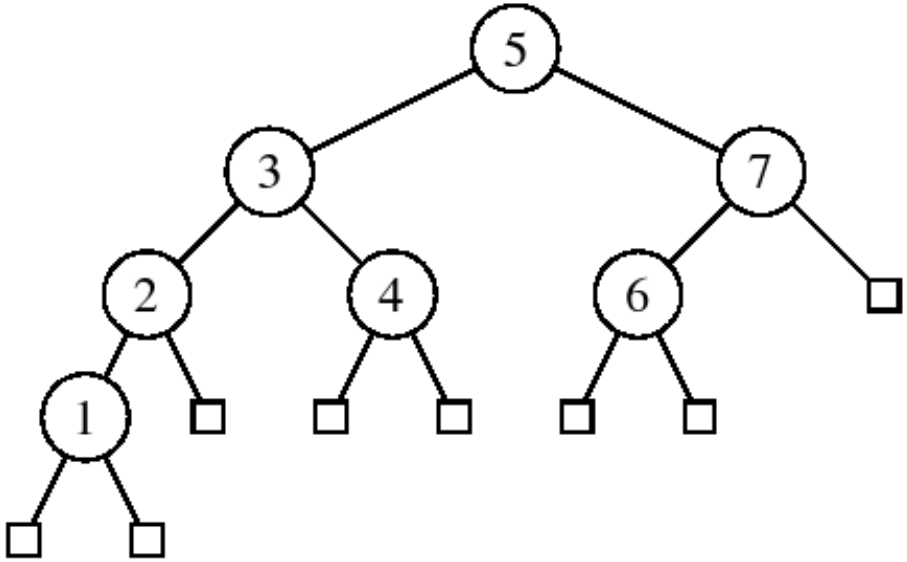


Caminhamento Central

- Após construída a árvore, pode ser necessário percorrer todo os registros que compõem a tabela ou arquivo.
- Existe mais de uma ordem de caminhamento em árvores, mas a mais útil é a chamada ordem de caminhamento central.
- O caminhamento central é mais bem expresso em termos recursivos:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- Uma característica importante do caminhamento central é que os nós são visitados de forma ordenada.

Caminhamento Central

- Percorrer a árvore:



- usando caminhamento central recupera as chaves na ordem 1, 2, 3, 4, 5, 6 e 7.

Caminhamento Central

- O procedimento Central é mostrado abaixo:

```
void Central(Apontador p)
{
    if (p == NULL) return;
    Central(p->Esq);
    printf("%ld\n", p->Reg.Chave);
    Central(p->Dir);
}
```

Análise

- O número de comparações em uma pesquisa com sucesso:

  melhor caso : C(n) = O(1)

  pior caso: C(n) = O(n)

  caso médio : C(n) = O(log n)

- O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores.

Análise

1. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é (n + 1)/2.

2. Para uma árvore de pesquisa aleatoria o número esperado de comparações para recuperar um registro qualquer é cerca de 1,39 log n, apenas 39% pior que a árvore completamente balanceada.