Algoritmos e Estruturas de Dados II

Pesquisa em Memória Principal

Introdução

Leonardo Jose Silvestre

lsilvestre@ufsj.edu.br

Pesquisa

- Como recuperar informação a partir de uma grande massa de informação previamente armazenada?
- Informação dividida em **registros**: cada registro possui uma chave para pesquisa
- Objetivo da pesquisa: encontrar uma ou mais ocorrências de registros com chaves iguais à **chave de pesquisa**
- **Pesquisa com sucesso**; **Pesquisa sem sucesso**
- Conjunto de registros: **Tabela** ou **Arquivo**

Métodos de Pesquisa

- Vários métodos de pesquisa existentes

- Escolha do mais adequado depende:
  – Da quantidade de dados envolvidos
  – De o arquivo estar sujeito a inserções e retiradas frequentes ou não

Algoritmos de Pesquisa

- Considerados TADs, já que possuem um conjunto de operações associados a uma estrutura de dados

- Operações mais comuns:
  - Inicializar a estrutura de dados
  - Pesquisar um ou mais registros com determinada chave
  - Inserir novo registro
  - Retirar um registro específico
  - Ordenar um arquivo para obter todos os registros em ordem (de acordo com a chave)
  - Ajuntar dois arquivos para formar um arquivo maior

Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa

- É um TAD com as operações *inicializar*, *pesquisar*, *inserir* e *retirar*

- Analogia com dicionário comum: chaves são as palavras; registros são as entradas associadas com cada palavra

- Nem todos os métodos de pesquisa que veremos terão todas as operações implementadas

Pesquisa Sequencial

- Método de pesquisa mais simples
- A partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada; então pare.
- Aspectos e convenções:
  - Forma possível para armazenar os registros: arranjo
  - Item contem chave e outros componentes

Pesquisa Sequencial - Estrutura

```
#define Maxn                10
typedef long TipoChave;
typedef struct Registro {
  TipoChave Chave;
  /* outros componentes */
} Registro;
typedef int Indice;
typedef struct Tabela {
  Registro Item[Maxn + 1];
  Indice n;
} Tabela;
```

Pesquisa Sequencial - Operações

```
void Inicializa(Tabela *T){
  T->n = 0;
}

Indice Pesquisa(TipoChave x, Tabela *T) {
  int i;
  T->Item[0].Chave = x; /* Sentinela: caso o retorno seja 0,
  significa que não encontrou*/
  i = T->n + 1;
  do {i--;}  while (T->Item[i].Chave != x);
  return i;
}
```

Pesquisa Sequencial - Operações

```
void Insere(Registro Reg, Tabela *T) {
  if (T->n == Maxn)
    printf("Erro : tabela cheia\n");
  else {
    T->n++;
    T->Item[T->n] = Reg;
  }
}
```

Pesquisa Sequencial - Análise

- Conforme já vimos:
  - melhor caso: $C(n) = 1$
  - pior caso: $C(n) = n$
  - caso médio: $C(n) = (n + 1)/2$

- Pesquisa sem sucesso: $C'(n) = n + 1$

- Técnica usando sentinela é conhecida como pesquisa sequencial rápida (laço interno é muito simples)

- Melhor solução para pesquisa em tabelas com 25 ou menos registros

Pesquisa Binária

- Pesquisa pode ser mais eficiente?

Pesquisa Binária

- Pesquisa pode ser mais eficiente?
  - Sim, desde que os registros sejam mantidos em ordem

- Procedimento:
  - Compare a chave com o registro que está na posição do meio
  - Se a chave é maior, repita o processo na primeira metade
  - Senão, repita o processo na segunda metade
  - O processo é repetido até que a chave seja encontrada ou fique apenas um registro de chave diferente da procurada

Pesquisa Binária - encontrar G

	1	2	3	4	5	6	7	8
Chaves iniciais:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
	A	*B*	*C*	**D**	*E*	*F*	*G*	*H*
					E	**F**	*G*	*H*
							G	*H*

Pesquisa Binária - encontrar G

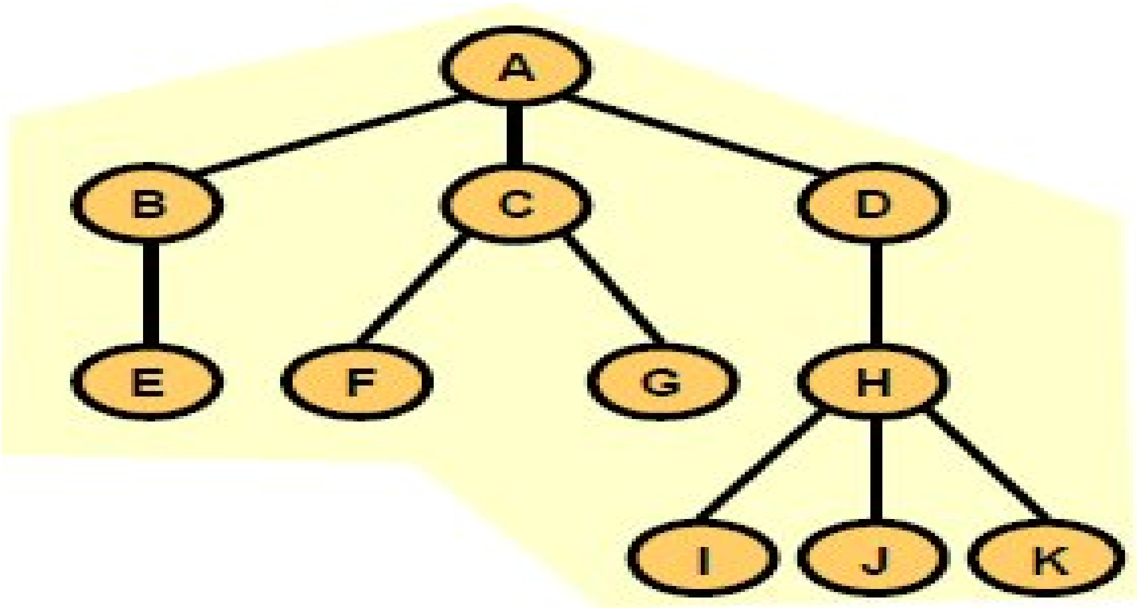
```
    Indice Binaria(TipoChave x, Tabela *T){
    Indice i, Esq, Dir;
    if (T->n == 0) return 0;
    else {
      Esq = 1;
      Dir = T->n;
      do {
        i = (Esq + Dir) / 2;
        if (x > T->Item[i].Chave)  Esq = i + 1;
        else
            Dir = i - 1;
      } while (x != T->Item[i].Chave && Esq <= Dir);
      if (x == T->Item[i].Chave)
        return i;
      else
        return 0;
    }
}
```

Pesquisa Binária - Análise

- A cada iteração, o tamanho da tabela é dividido ao meio
- Número de vezes que ocorre a divisão: $\log n$
- Custo para manter a tabela ordenada é alto (deslocamento de registros)
- Uso: aplicações não muito dinâmicas

Árvores

- Estruturas fundamentais em Computação

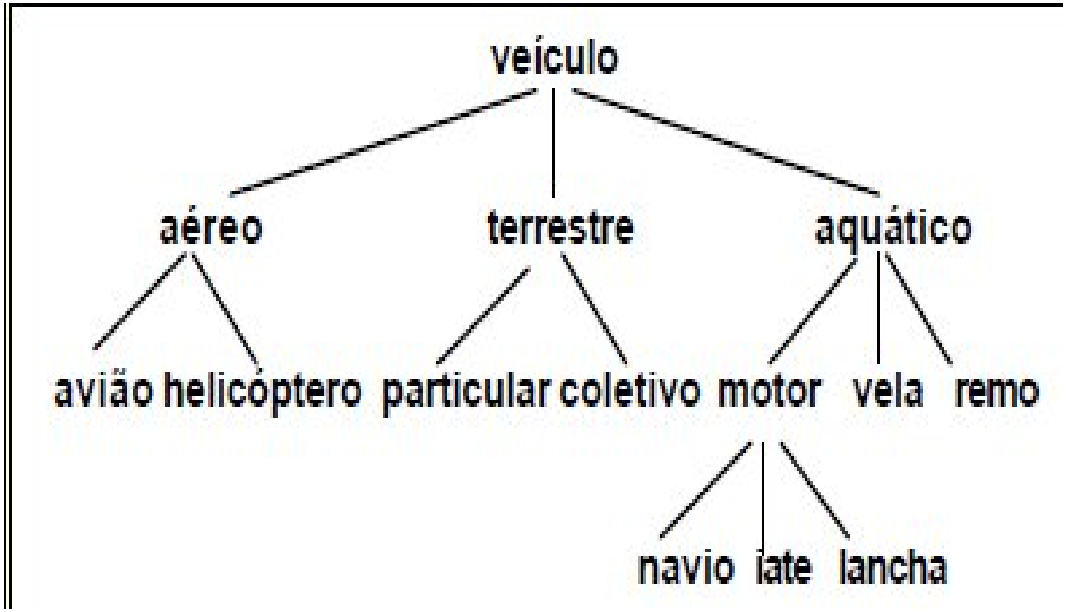


Árvores

- Compostas de raiz + sub-árvores
- Quando a raiz é retirada, devem sobrar árvores distintas
- Ou seja, uma árvore não pode ter ciclos

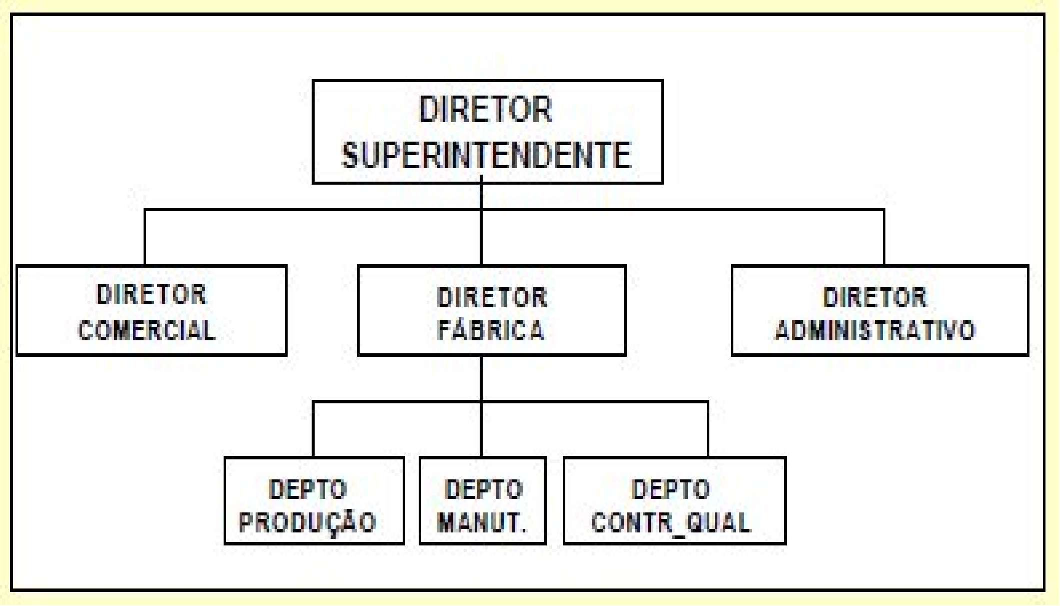
Árvores - Aplicações

- Hierarquia (classes e subclasses)



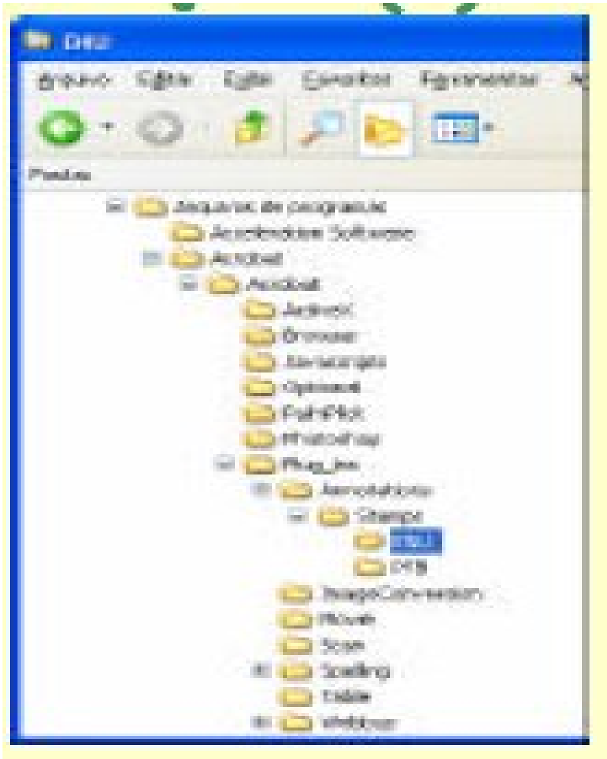
Árvores - Aplicações

- Organograma



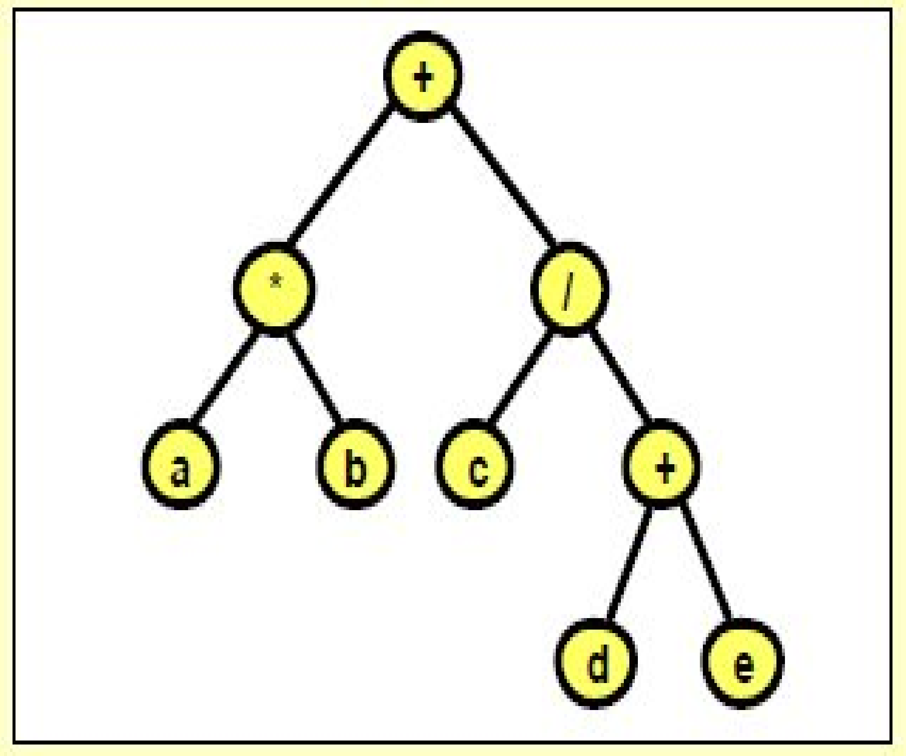
Árvores - Aplicações

- Estrutura de Diretórios



Árvores - Aplicações

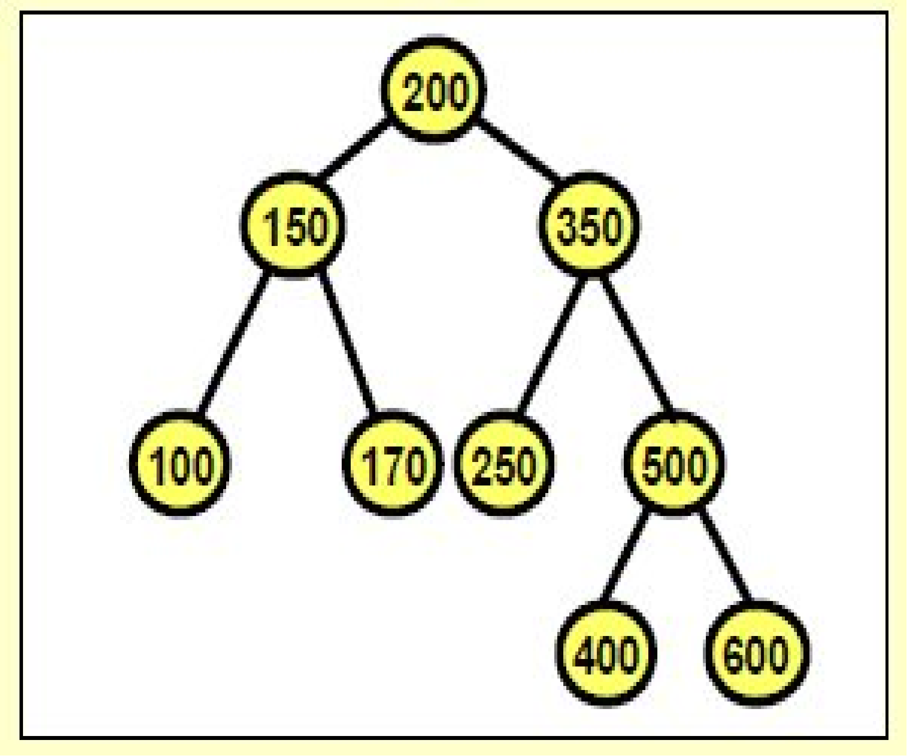
- Árvore de derivação - Compilador



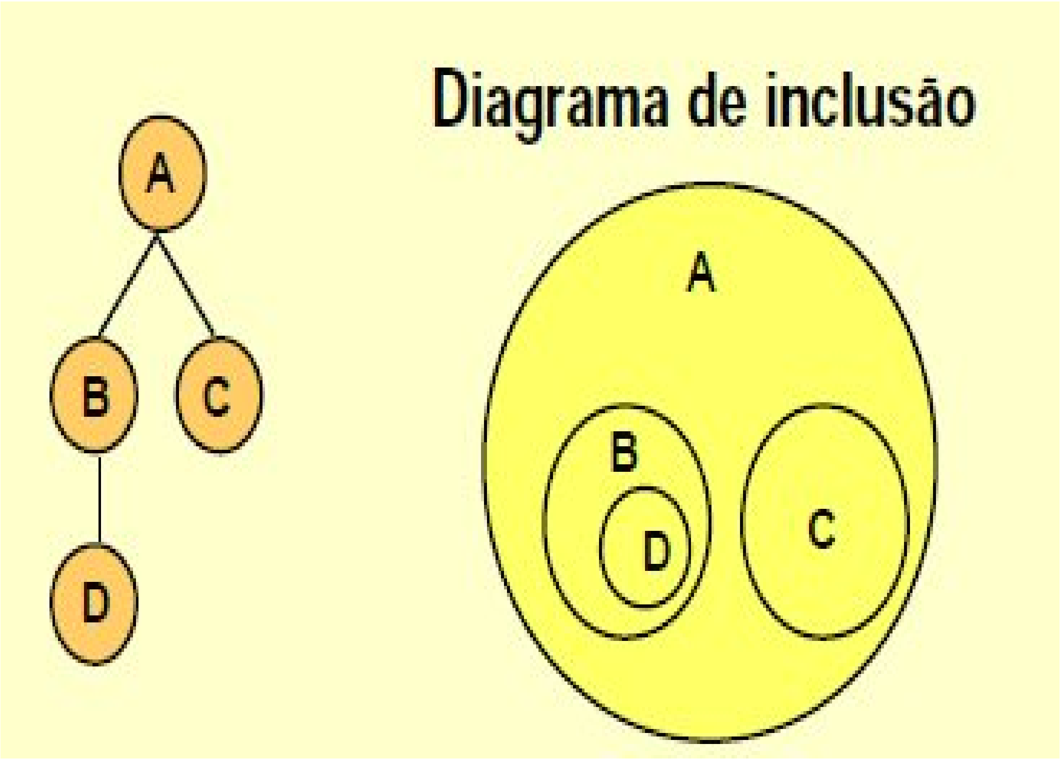
- Expressão aritmética: $(a * b) + (c/(d + e))$

Árvores - Aplicações

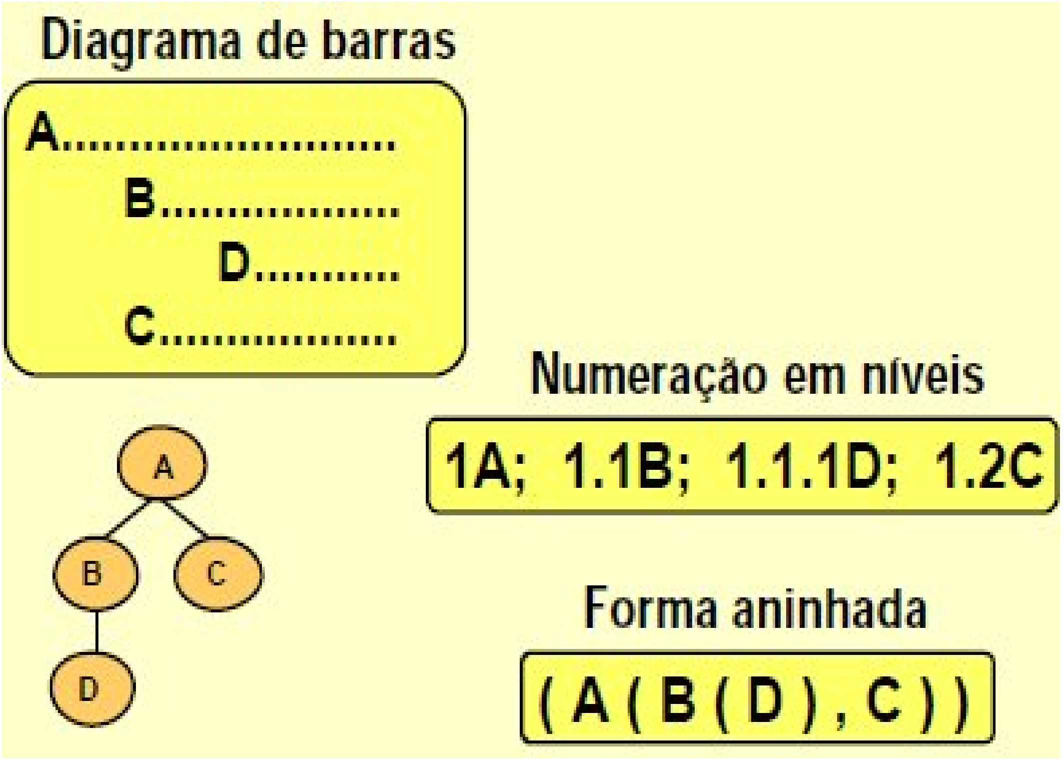
- Ordenação de valores



Árvores - Outras Formas de Representação



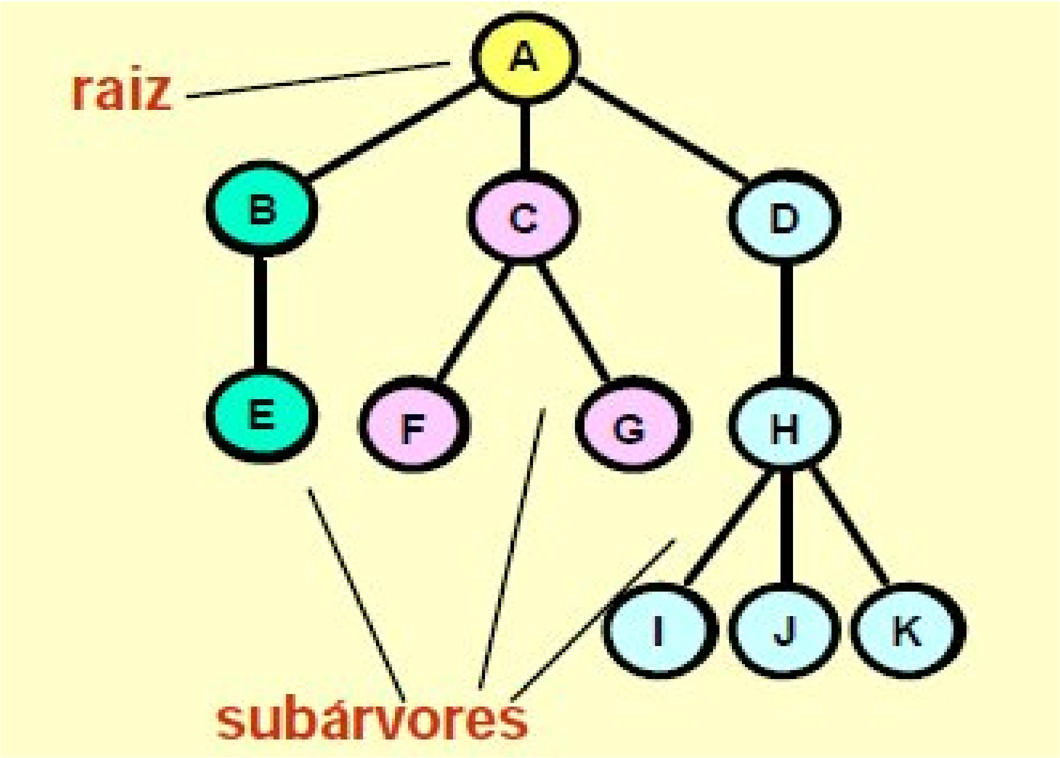
Árvores - Outras Formas de Representação



Árvores - Definição Formal

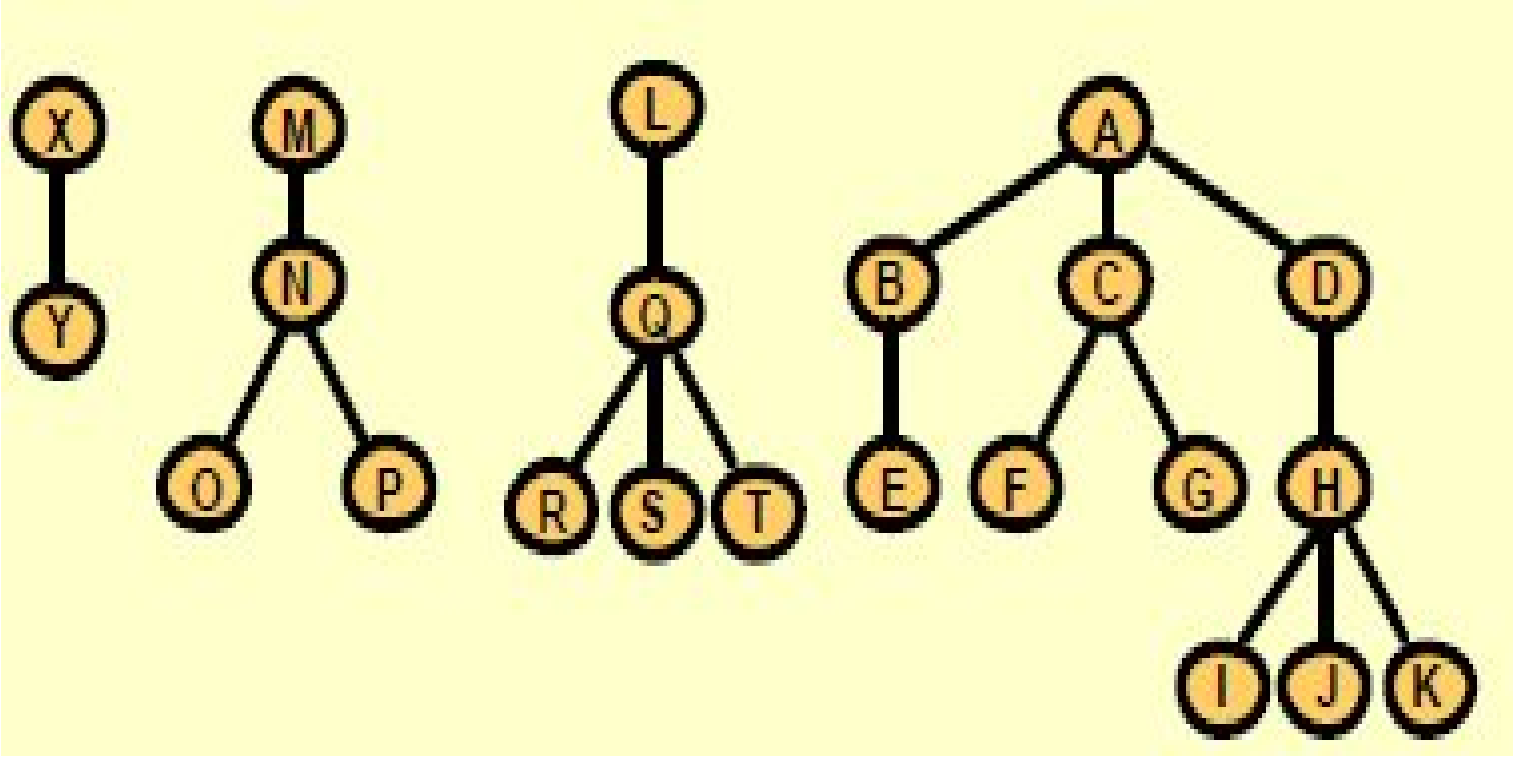
- Uma árvore é um conjunto finito A de zero ou mais nodos, tal que:
  - (1) Caso o número de nodos seja maior que zero
    - $\rightarrow$ existe um nodo denominado raiz da árvore
    - $\rightarrow$ os demais nodos formam m > 0 conjuntos disjuntos $S_1, S_2, \ldots, S_m$, onde cada um destes é uma árvore (as S_i's são denominadas subárvores)
  - (2) Número de nodos zero: árvore vazia

Árvores - Definição



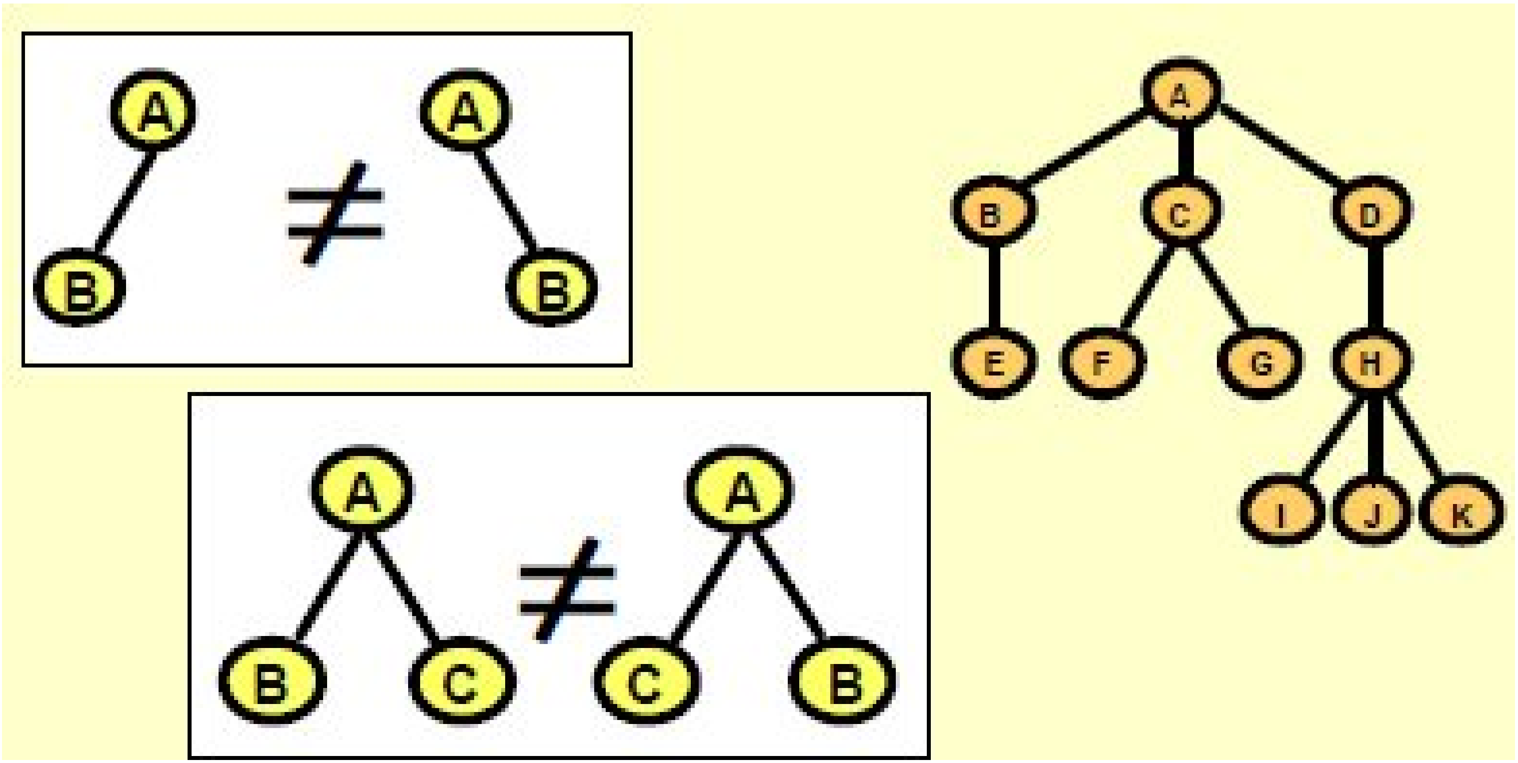
Árvores - Terminologia

- **Floresta**: conjunto de zero ou mais árvores distintas



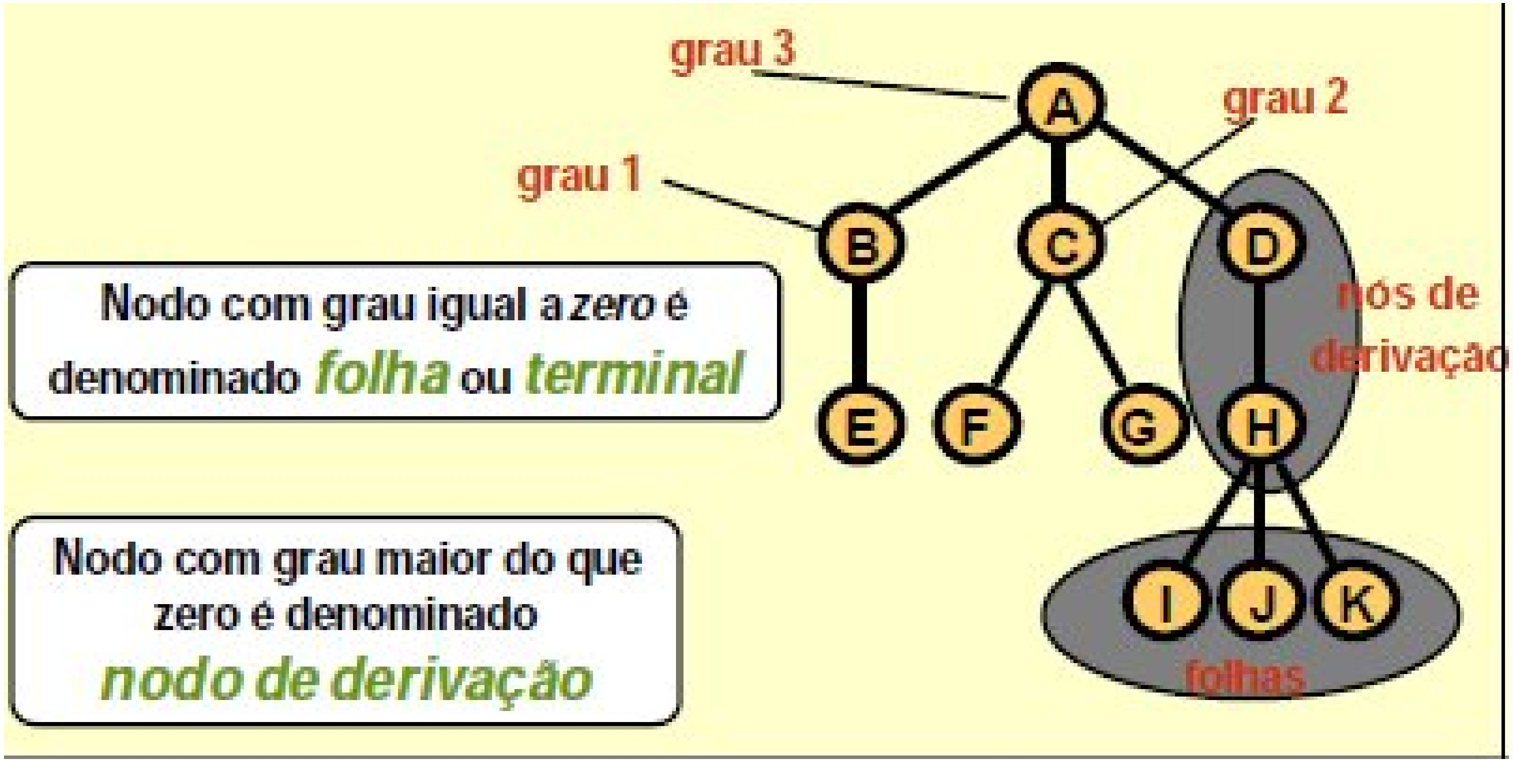
Árvores - Terminologia

- Árvore **ordenada**: ordem de suas sub-árvores é relevante



Árvores - Terminologia

- **Grau** de um nodo: conjunto sub-árvores do mesmo

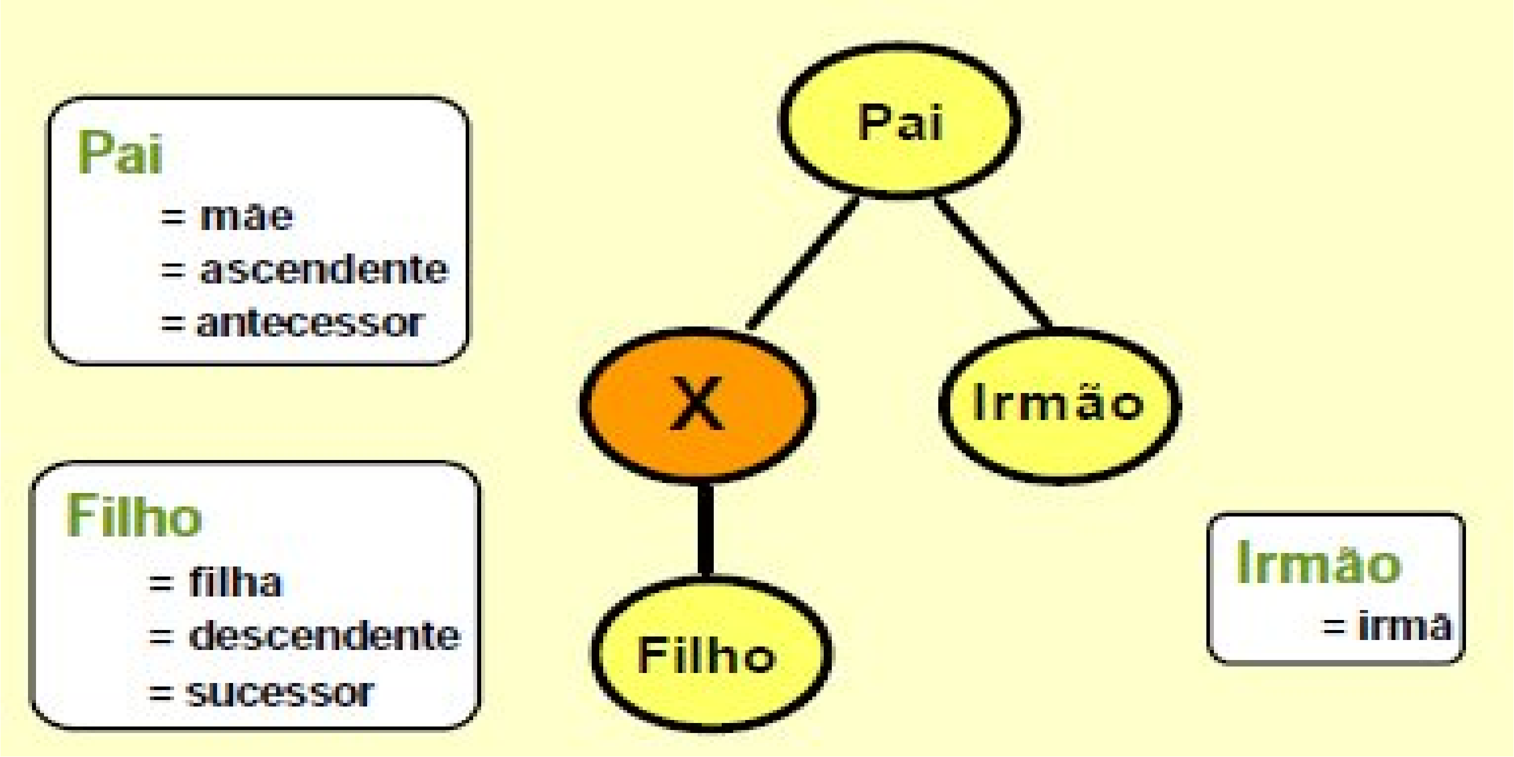


Árvores - Terminologia

- **Nível** de um nodo: número de linhas entre ele e a raiz, acrescido de uma unidade

Árvores - Terminologia

- Denominação relativa de um nodo:

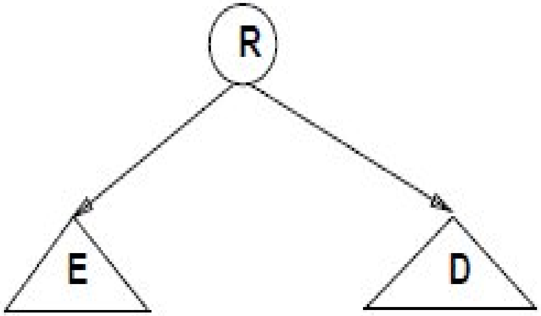


Árvores Binárias de Pesquisa

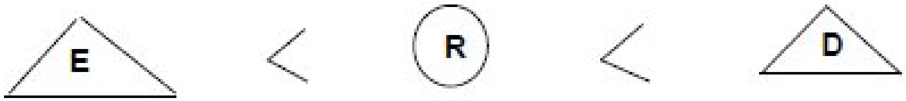
- BST - *Binary Search Tree*
- Estrutura de dados muito eficiente para armazenar informação
- Adequada quando existe necessidade de considerar todos ou alguma combinação de:
  - Acesso direto e seqüencial eficientes
  - Facilidade de inserção e retirada de registros
  - Boa taxa de utilização de memória
  - Utilização de memória primária e secundária

BST sem Balanceamento

- Para qualquer nó que contenha um registro



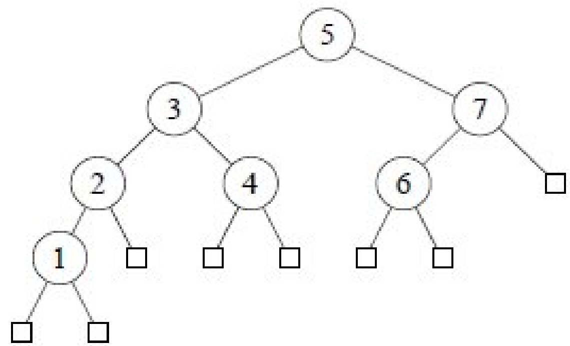
- temos a relação invariante:



  – Todos os registros com chaves menores estão na subárvore à esquerda
  – Todos os registros com chaves maiores estão na subárvore à direita

BST sem Balanceamento

- Exemplo



- O **nível** do nó raiz é 0
- Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$
- 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

Implementação do TAD Dicionário usando a Estrutura de Dados BST

- Exercício: definir o TAD Dicionário
- Applet

Implementação do TAD Dicionário usando a Estrutura de Dados BST

```
typedef long Chave;
typedef struct Registro {
  Chave chave;
  /* outros componentes */
} Registro;
typedef struct No * Apontador;
typedef struct No {
  Registro registro;
  Apontador esq, dir;
} No;
typedef Apontador Dicionario;
```

Procedimento para Pesquisar na BST

- 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 BST

```
void pesquisar(Registro *x, Apontador *p);
```

Procedimento para Pesquisar na BST

```
void pesquisar(Registro *x, Apontador *p) {
  if (*p == NULL) {
    printf("Erro : Registro nao esta presente na arvore\n");
    return;
  }
  if (x->chave < (*p)->registro.chave) {
    pesquisar(x, &(*p)->esq);
    return;
  }
  if (x->chave > (*p)->registro.chave)
    pesquisar(x, &(*p)->dir);
  else
    *x = (*p)->registro;
}
```

Procedimento para Inserir na BST

- Para inserir um registro com uma chave x:
  - Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso
  - O apontador nulo atingido é o ponto de inserção
  - A BST não permite chaves repetidas

Procedimento para Inserir na BST

```
void inserir(Registro x, Apontador *p);
```

Procedimento para Inserir na BST

```
void inserir(Registro x, Apontador *p) {
  if (*p == NULL) {
    *p = (Apontador)malloc(sizeof(No));
    (*p)->registro = x; (*p)->esq = NULL; (*p)->dir = NULL;
    return;
  }
  if (x.chave < (*p)->registro.chave) {
    inserir(x, &(*p)->esq); return;
  }
  if (x.chave > (*p)->registro.chave)
    inserir(x, &(*p)->dir);
  else
    printf("Erro : Registro ja existe na arvore\n");
  }
```

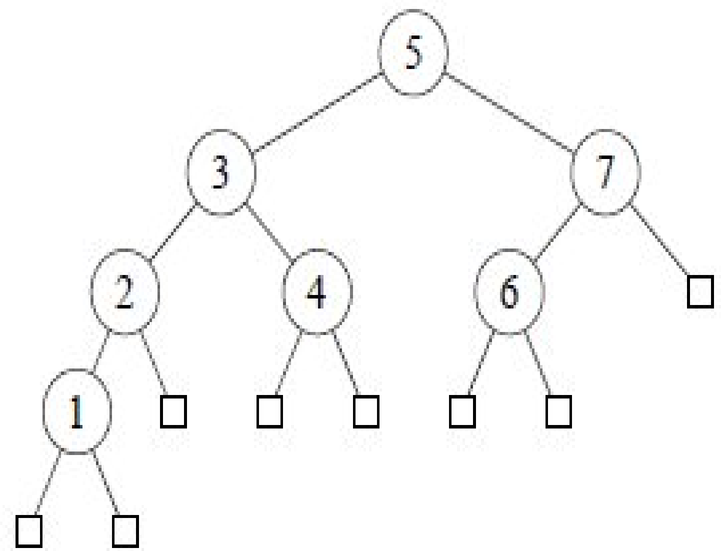
Procedimento para Inicializar a BST

```
void inicializar(Apontador *dicionario) {
  *dicionario = NULL;
}
```

Procedimento para Retirar x da BST

- Não é tão simples quanto a inserção
- Se o nó que contém o registro a ser retirado possui no máximo um descendente, a operação é simples.
- 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 de Retirada de um Registro da BST



- para retirar o registro com chave 5 na 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.

Procedimento para Retirar x da BST

```
void retirar(Registro x, Apontador *p);
```

Procedimento para Retirar x da BST

```
void retirar(Registro x, Apontador *p) {
  Apontador aux;
  if (*p == NULL) {
    printf("Erro : Registro nao esta na arvore\n");
    return;
  }
  if (x.chave < (*p)->registro.chave) {
    retirar(x, &(*p)->esq); return;
  }
  if (x.chave > (*p)->registro.chave) {
    retirar(x, &(*p)->dir); return;
  }
```

```
  if ((*p)->dir == NULL) {
    aux = *p;
    *p = (*p)->esq;
    free(aux);
    return;
  }
  if ((*p)->esq != NULL) {
    antecessor(*p, &(*p)->esq);
    return;
  }
  aux = *p;
  *p = (*p)->dir;
  free(aux);
}
```

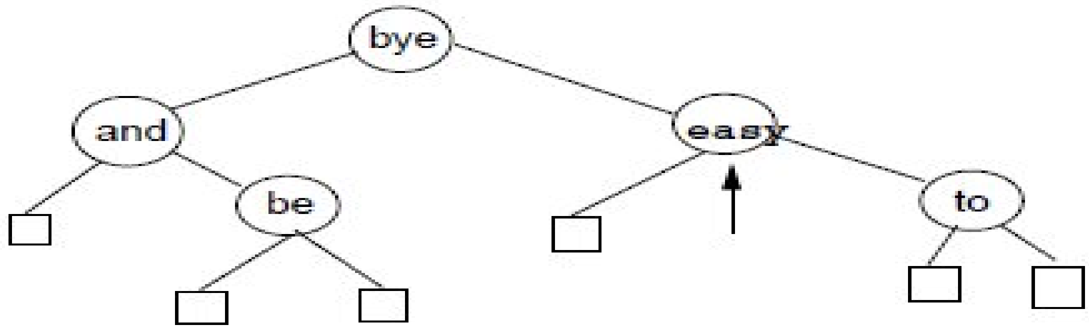
Procedimento Antecessor

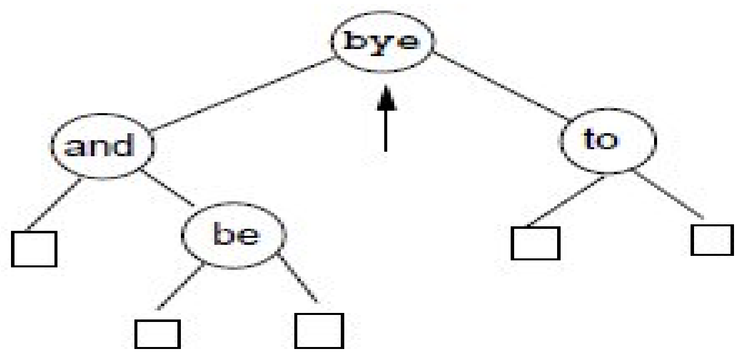
```
void antecessor(Apontador q, Apontador *r);
```

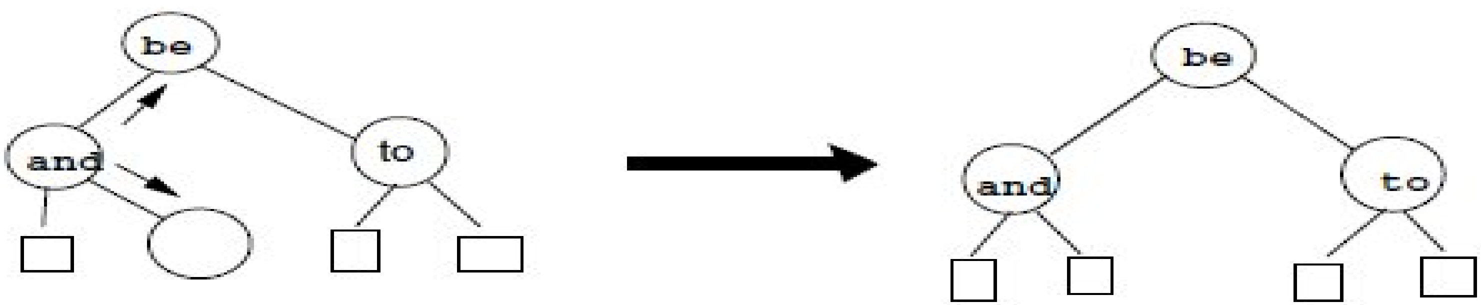
Procedimento Antecessor

```
void antecessor(Apontador q, Apontador *r) {
  if ((*r)->dir != NULL) {
    antecessor(q, &(*r)->dir);
    return;
  }
  q->registro = (*r)->registro;
  q = *r;
  *r = (*r)->esq;
  free(q);
}
```

Outro Exemplo de Retirada de um Registro da BST







Caminhamento Central

- Como percorrer todos os registros da árvore?

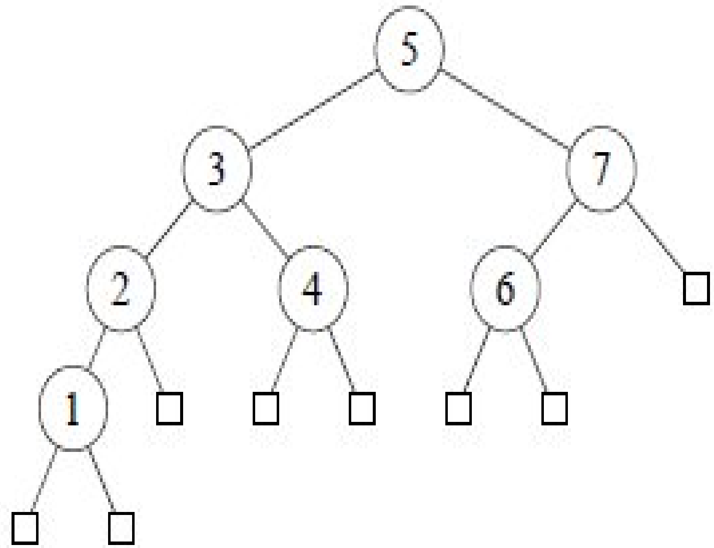
- Mais de uma ordem de caminhamento em árvores. Mais útil:*ordem de caminhamento central*

- Mais bem expresso em termos recursivos:
  - caminha na subárvore esquerda na ordem central
  - visita a raiz
  - caminha na subárvore direita na ordem central

- Característica importante do caminhamento central: 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

```
void central(Apontador p);
```

Caminhamento Central

```
void central(Apontador p) {
  if (p == NULL) {
    return;
  }
  central(p->esq);
  printf("%ld\n", p->registro.chave);
  central(p->dir);
}
```

Análise

- Número de comparações em uma pesquisa com sucesso
  melhor caso: $C(n) = O(1)$
  pior caso: $C(n) = O(n)$
  caso médio: $C(n) = O(\log n)$

- Tempo de execução depende muito do formato da árvore

Análise

- Pior caso: chaves inseridas em ordem crescente ou decrescente (árvore se “degenera” em lista linear)

- **Árvore de pesquisa randômica**: número esperado de comparações para recuperar um registro qualquer é de cerca de $1,39 \log n$, 39% pior que para uma árvore balanceada (melhor caso da BST)

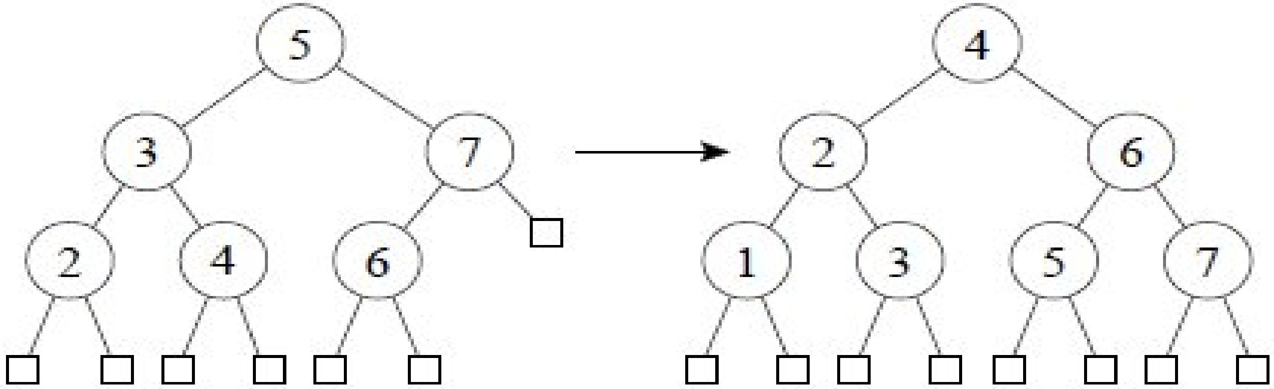
- Uma árvore de pesquisa randômica com n chaves é uma árvore constituída através de n inserções randômicas em uma árvore inicialmente vazia

BSTs com Balanceamento

- Árvore completamente balanceada: nós externos aparecem em, no máximo, dois níveis adjacentes
- Minimiza tempo médio de pesquisa para uma distribuição uniforme das chaves, onde cada chave é igualmente provável de ser usada em uma pesquisa
- Contudo, custo para manter a árvore completamente balanceada após cada inserção é muito alto

BSTs com Balanceamento

- Para inserir a chave 1 na árvore do exemplo à esquerda e obter a árvore à direita do mesmo exemplo é necessário movimentar todos os nós da árvore original



BSTs com Balanceamento

- Solução: manter a árvore “quase-balanceada”, em vez de tentar manter a árvore completamente balanceada

- Objetivo: Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.

- Heurísticas: existem várias heurísticas baseadas no princípio acima

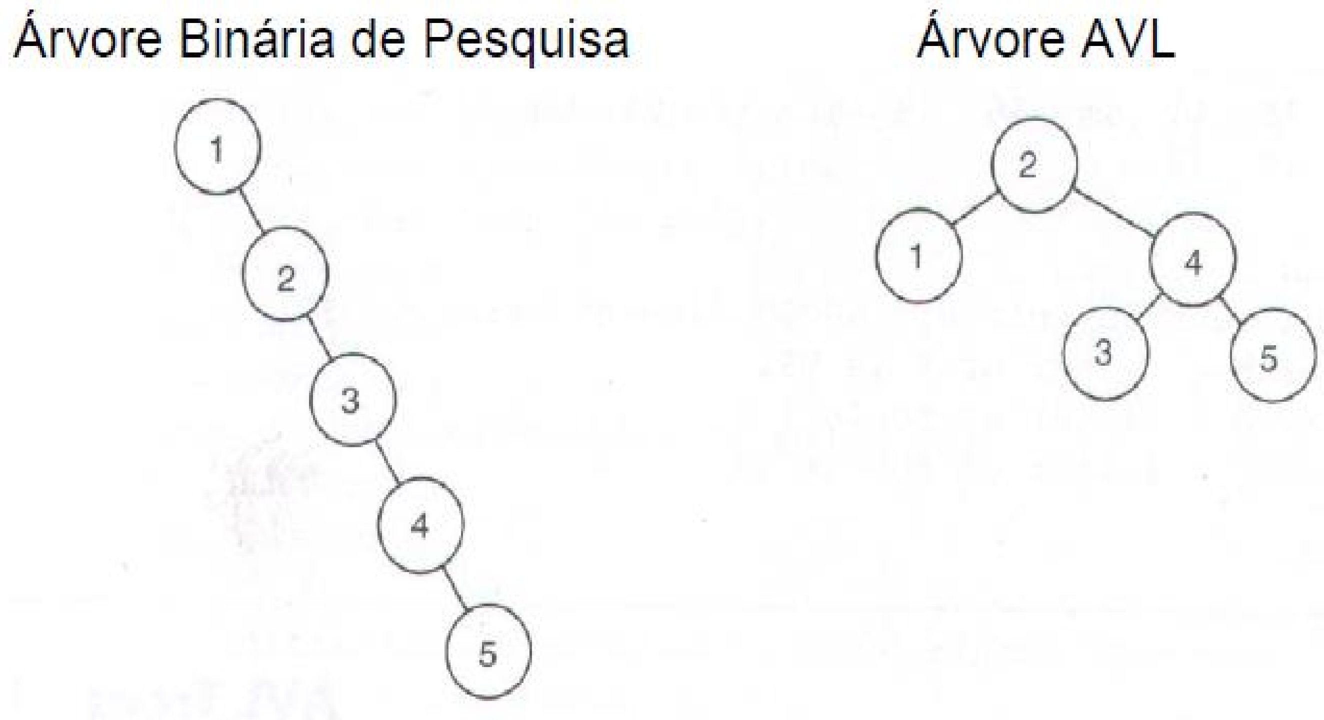
- Critérios de balanceamento:
  - na diferença das alturas de subárvores de cada nó da árvore
  - na redução do comprimento do caminho interno
  - todos os nós externos apareçam no mesmo nível

Árvore AVL

- BST Balanceada
- Altura das duas sub-árvores a partir de cada nó difere, no máximo, em uma unidade
- Balanceamento é mantido nas inserções e remoções (custo: $O(\log n)$)
- AVL: *Adelson Velsky* e *Landis* - criadores da AVL (1962)

AVL - Exemplo

- Supondo a inserção das chaves 1, 2, 3, 4, 5, nesta ordem, teríamos:



Inserção na AVL

- O que pode acontecer quando um novo nó é inserido em uma árvore balanceada?

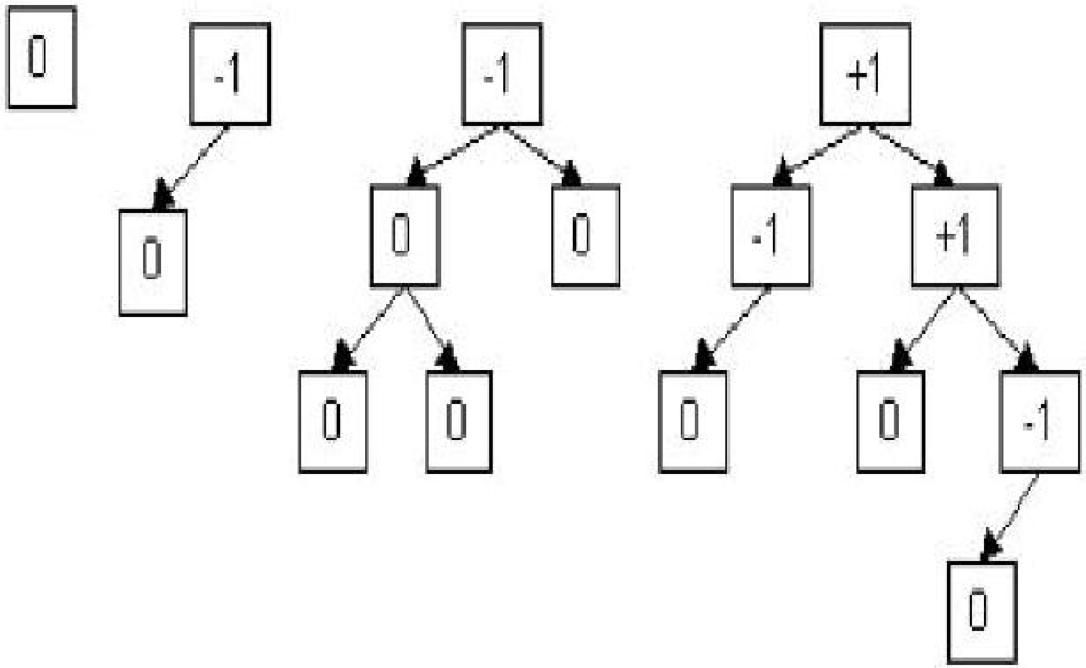
- Dada uma raiz r com subárvores L (*Left*) e R (*Right*), e supondo que a inserção seja feita na sub-árvore da esquerda. Podemos distriguir 3 casos:
  - Se hL = hR, então L e R ficam com alturas diferentes mas continuam balanceadas
  - Se hL < hR, então L e R ficam com alturas iguais e balanceamento foi melhorado
  - Se hL > hR, então L fica ainda maior e balanceamento foi violado

AVL - Fator de Balanceamento

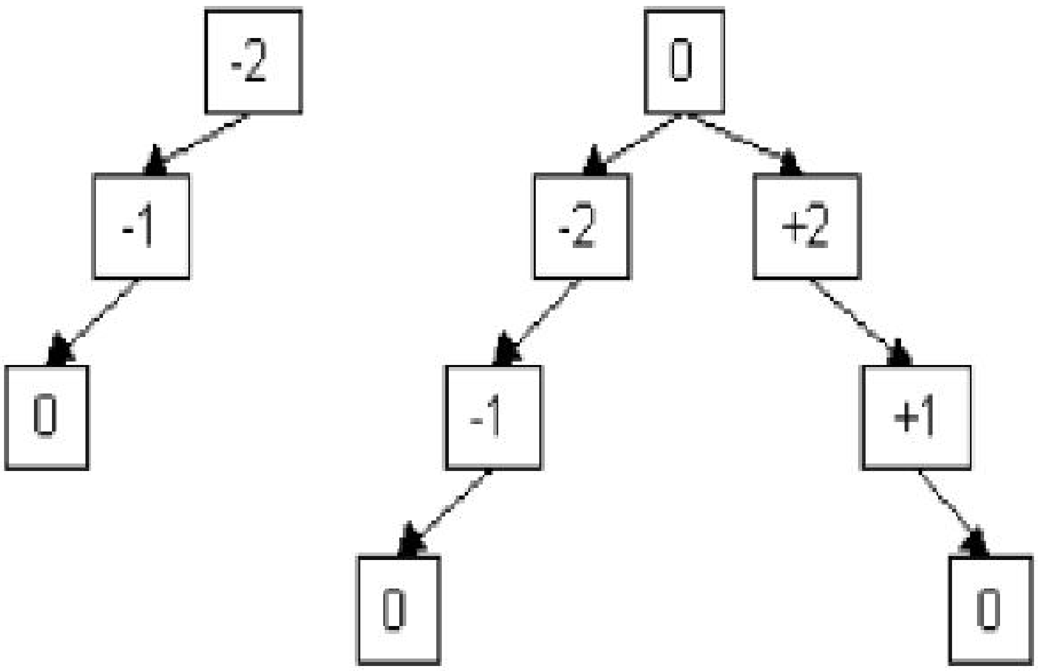
- **Fator de Balanceamento** (FB) de um nó: altura da sub-árvore direita do nó menos a altura da sub-árvore esquerda do nó

- Em uma árvore AVL, temos que $|FB| \leq 1$:
  - Se $FB = 0$, as duas sub-árvores tem a mesma altura
  - Se $FB = -1$, a sub-árvore esquerda é mais alta que a direita em 1
  - Se $FB = +1$, a sub-árvore direita é mais alta que a esquerda em 1

Exemplos de Árvores AVL

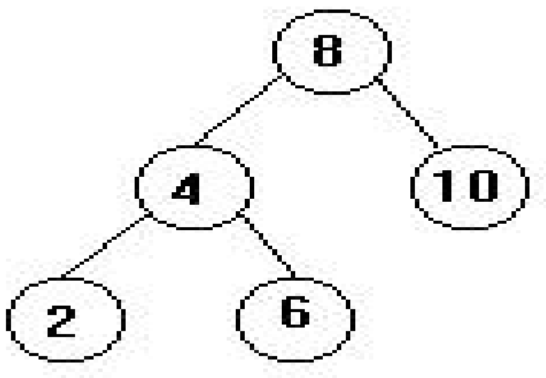


Exemplos de Árvores não-AVL



Inserção na AVL

- Na árvore abaixo:



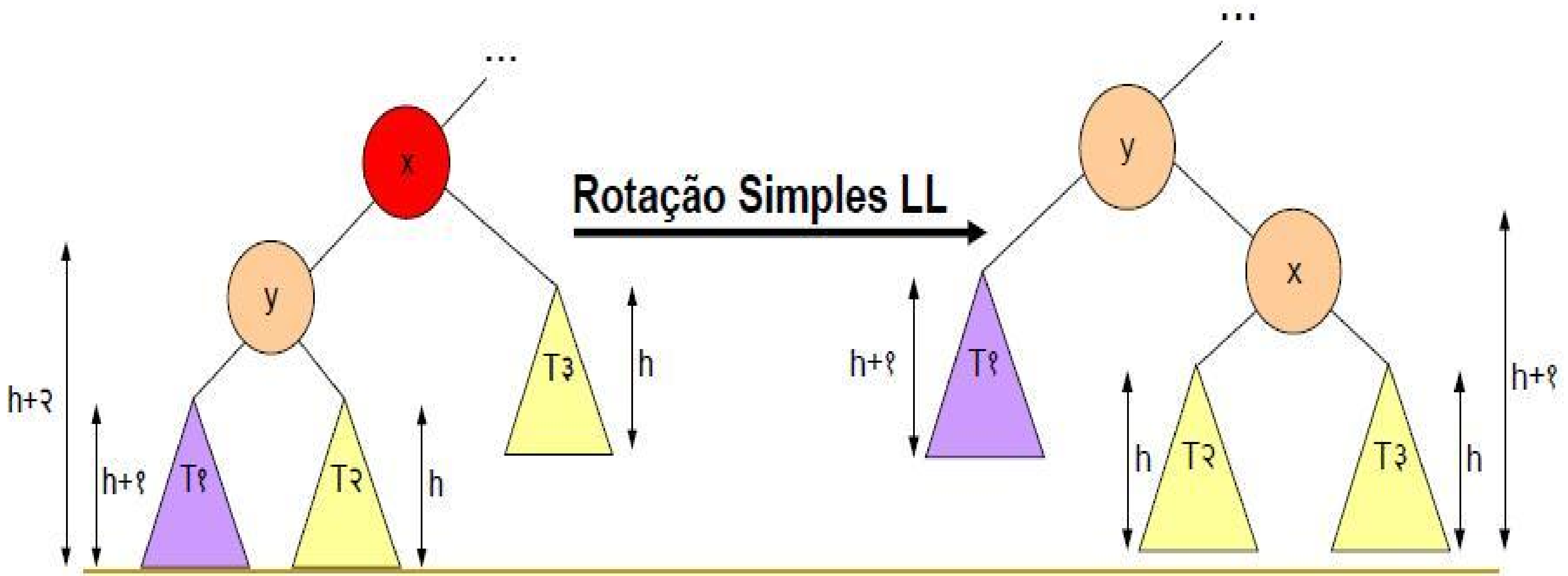
- Nós 9 ou 11 podem ser inseridos sem balanaceamento

- Inserção dos nós 3, 5 ou 7 requerem que a árvore seja rebalanceada!

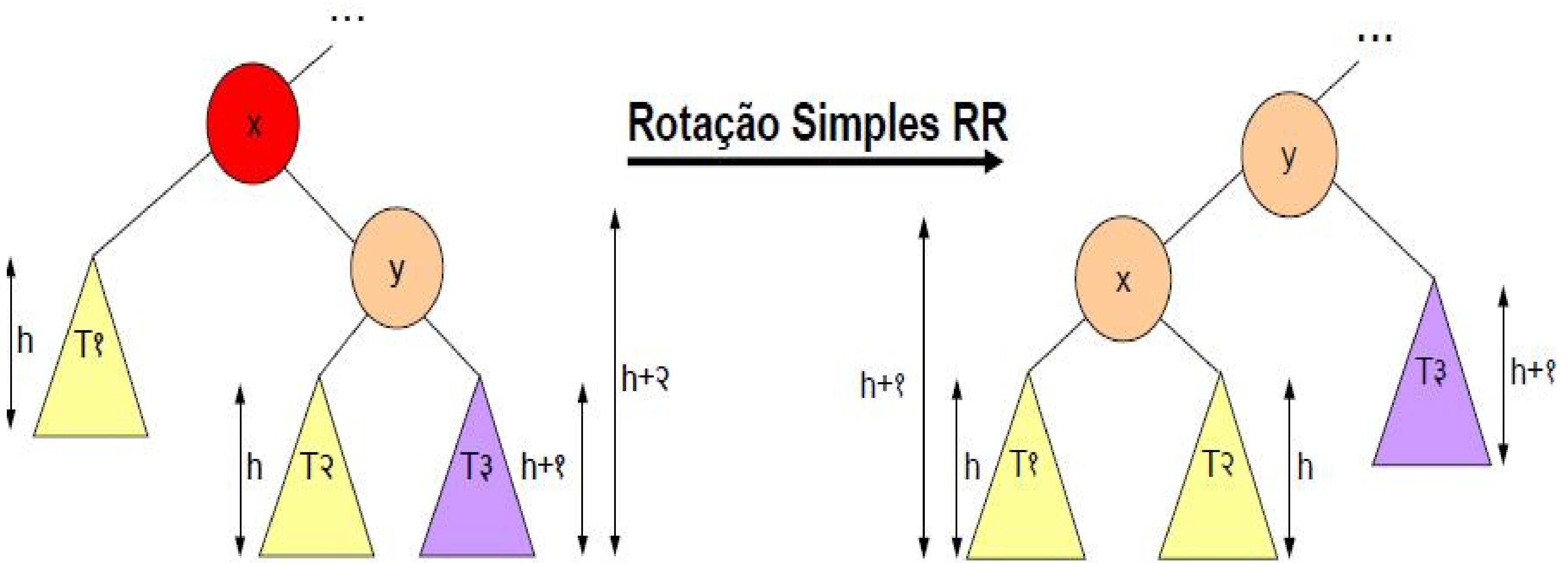
AVL - Balanceamento

- Como manter uma árvore AVL balanceada?
  - Inicialmente insere-se um novo nó na árvore normalmente
  - Essa inserção pode ou não violar a propriedade de balanceamento
  - Caso não viole a propriedade de balanceamento pode-se então continuar com a inserção de novos nós
  - Caso contrário deve-se restaurar o balanço da árvore
  - A restauração deste balanço é efetuada através de **rotações** na árvore
  - Mantemos o FB de cada nó armazenado para facilitar a verificação de necessidade de balanceamento

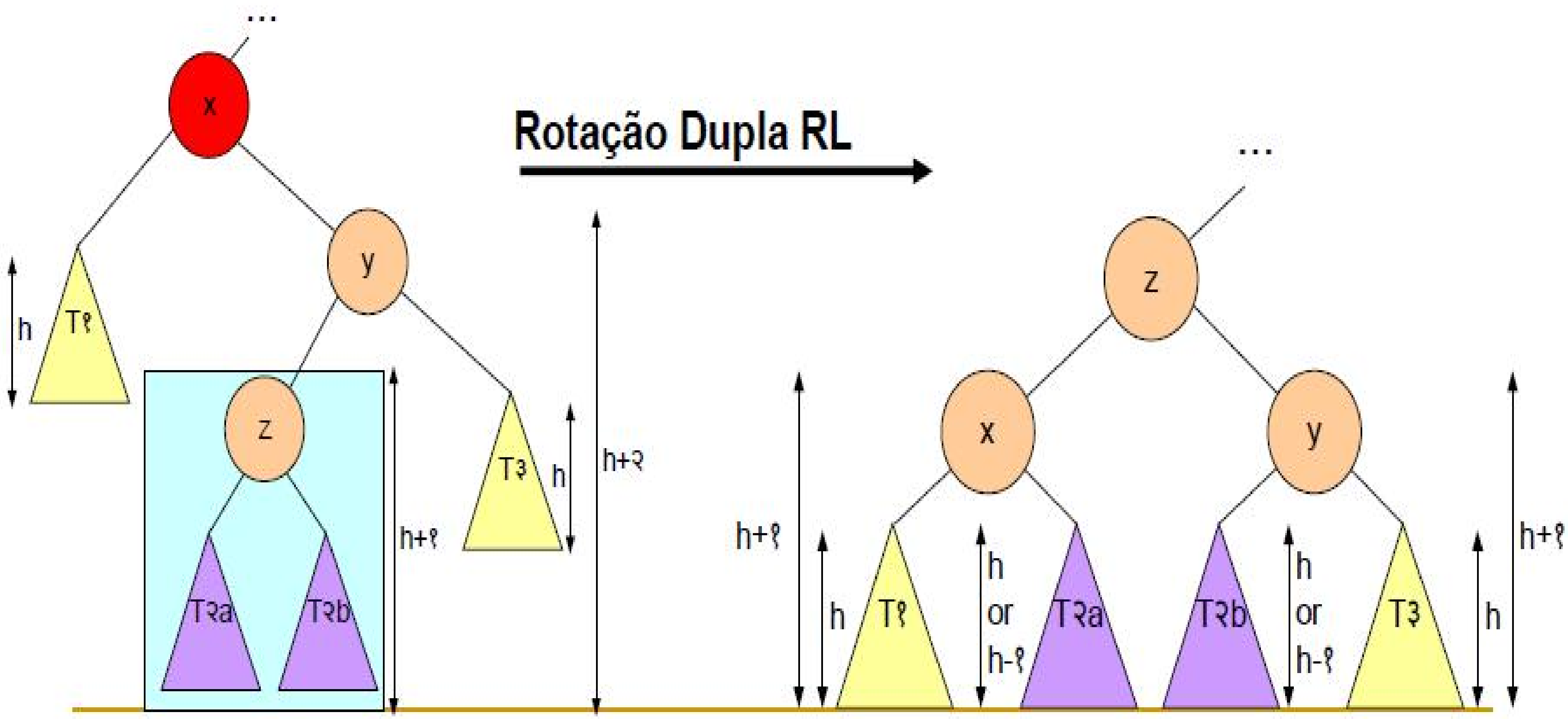
AVL - Rotação Simples LL



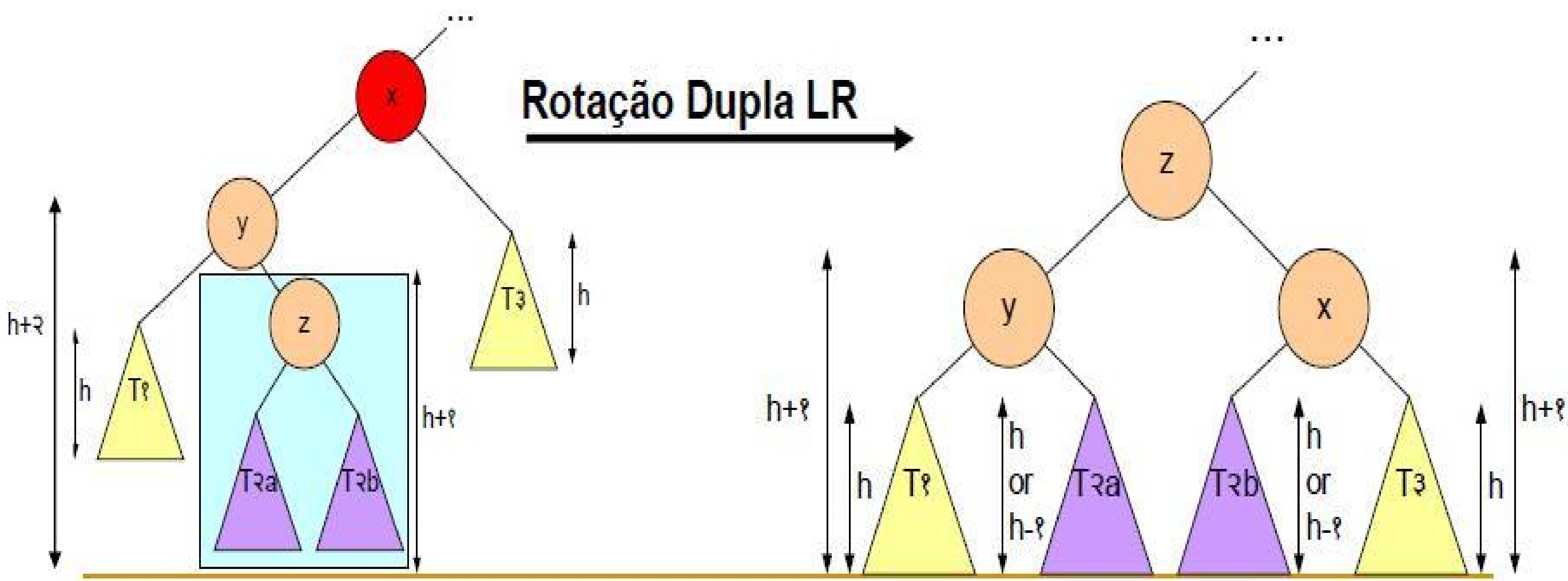
AVL - Rotação Simples RR



AVL - Rotação Dupla RL



AVL - Rotação Dupla LR



AVL - Inserção

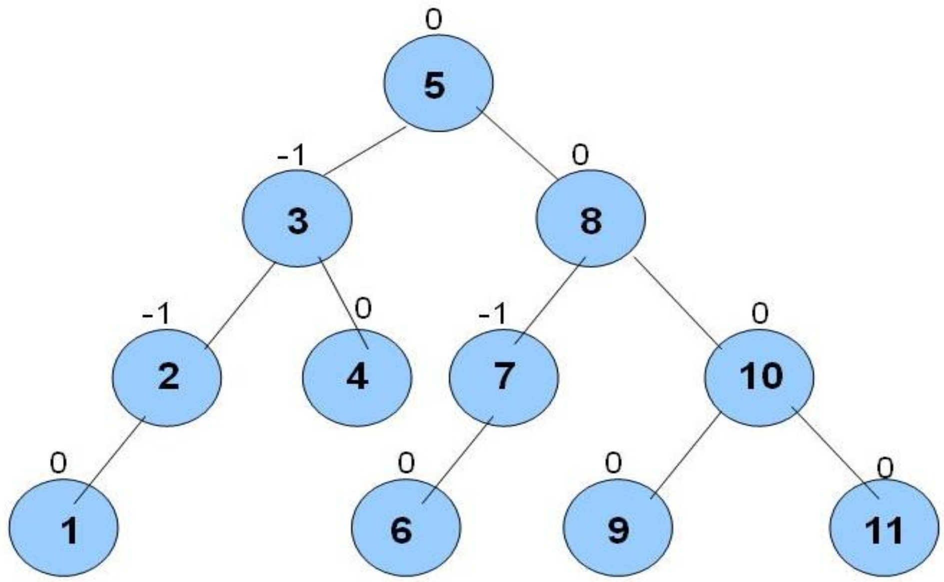
- Construir uma árvore AVL com as seguintes chaves:

S, T, X, E, B, M, G, N, J, Q, Z

AVL - Remoção

- Dada a árvore abaixo, remover as seguintes chaves:

4, 8, 6, 5, 2, 1, 7



AVL - Análise

- Altura maxima: $\lfloor \log n \rfloor$
- Número máximo de rotações na remoção: proporcional a $logn$ (devemos sempre verificar até a raiz, mesmo que já tenha sido feita uma rotação)
- Inserção, Remoção e Pesquisa: $O(\log n)$ $(1,44 \log n)$
- Implementação bastante complexa

Outras BSTs Balanceadas

- Árvore Red-Black
- Árvore SBB
- Árvore de Fibonacci (F_h, F_{h-1}, F_{h-2})
- Árvore B: mais utilizada para pesquisa em memória secundária, mas também pode ser utilizada para memoria principal (veremos adiante)

Pesquisa Digital

- Baseada na representação das chaves como uma seqüência de caracteres ou de dígitos
- Os métodos de pesquisa digital são particularmente vantajosos quando as chaves são grandes e de tamanho variável
- Um aspecto interessante quanto aos métodos de pesquisa digital é a possibilidade de localizar todas as ocorrências de uma determinada cadeia em um texto, com tempo de resposta logarítmico em relação ao tamanho do texto
- Árvores Digitais:
  - Trie
  - Patrícia

Trie

- Árvore M-ária cujos nós são vetores de M componentes com campos correspondentes aos dígitos ou caracteres que formam as chaves

- Cada nó no nível i representa o conjunto de todas as chaves que começam com a mesma seqüência de i dígitos ou caracteres

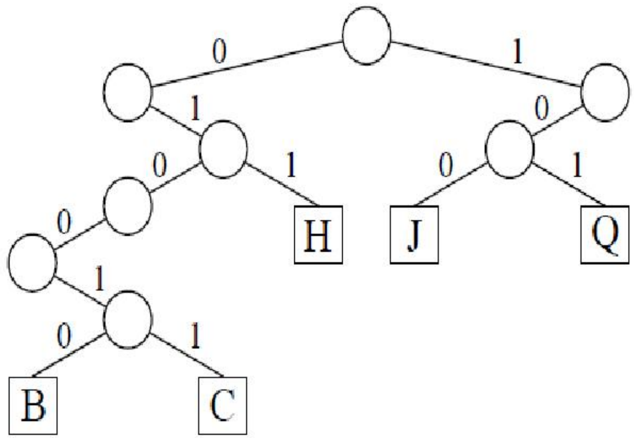
- Este nó especifica uma ramificação com M caminhos dependendo do $(i+1)$-ésimo dígito ou caractere de uma chave

- Considerando as chaves como seqüência de bits (isto é, $M = 2$), o algoritmo de pesquisa digital é semelhante ao de pesquisa em BST, exceto que, em vez de se caminhar na árvore de acordo com o resultado de comparação entre chaves, caminha-se de acordo com os bits de chave

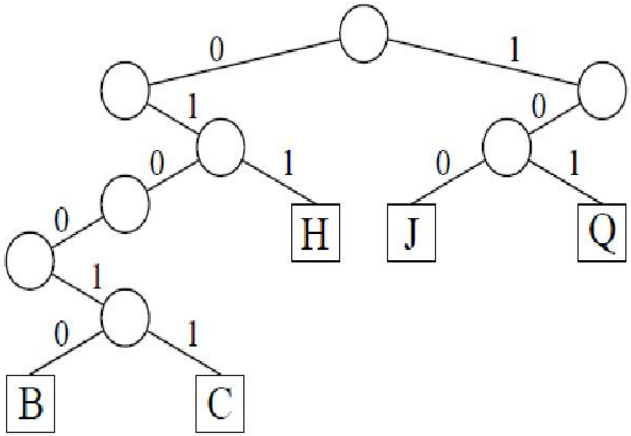
Trie Binaria - Exemplo

- Dadas as chaves de 6 bits:

```
B = 010010
C = 010011
H = 011000
J = 100001
M = 101000
```

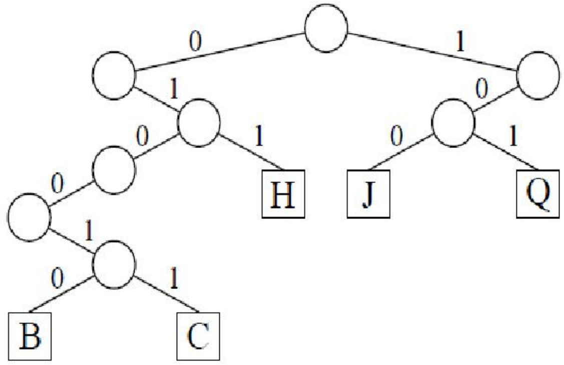


Trie Binária - Inserção de W e K

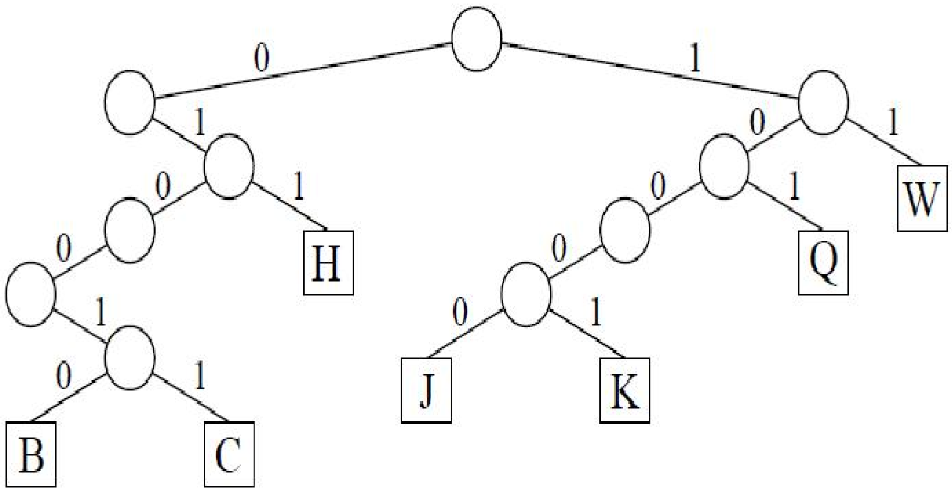


- Pesquisa-se na árvore, com a chave a ser inserida
- Se o nó externo em que a pesquisa terminar for vazio: cria-se um novo nó externo nesse ponto contendo a nova chave, exemplo: a inserção da chave `W = 110110`
- Se o nó externo contiver uma chave: cria-se um ou mais nós internos cujos descendentes conterão a chave já existente e a nova chave. exemplo: inserção da chave `K = 100010`

Trie Binária - Inserção de W e K



- `W = 110110` e `K = 100010`



Tries - Considerações

- Formato das tries não depende da ordem em que as chaves são inseridas, mas da estrutura das chaves através da distribuição de seus bits
- **Desvantagem**: formação de caminhos de uma só direção para chaves com um grande número de bits em comum
  - Exemplo: Se duas chaves diferirem somente no último bit, elas formarão um caminho cujo comprimento é igual ao tamanho delas, não importando quantas chaves existem na árvore
  - Caminho gerado pelas chaves B e C

Patricia

- *Practical Algorithm To Retrieve Information Coded In Alphanumeric*
- Criado por Morrison D. R. 1968 para aplicação em recuperação de informação em arquivos de grande porte
- Knuth D. E. (1973): novo tratamento (algoritmo)
- Reapresentou-o de forma mais clara como um caso particular de pesquisa digital, essencialmente, um caso de árvore trie binária
- Sedgewick R. (1988) apresentou novos algoritmos de pesquisa e de inserção baseados nos algoritmos propostos por Knuth
- Gonnet, G.H e Baeza-Yates R. (1991) propuseram também outros algoritmos

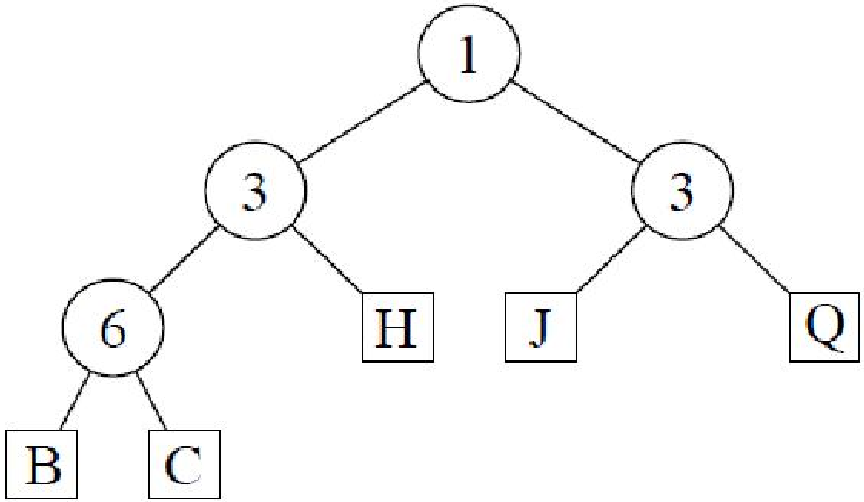
Patricia

- Algoritmo para construção da árvore Patricia: baseado no método de pesquisa digital, mas sem apresentar o inconveniente citado para o caso das tries
- Problema de caminhos de uma só direção eliminado por meio de uma solução simples e elegante: cada nó interno da árvore contém o índice do bit a ser testado para decidir qual ramo tomar

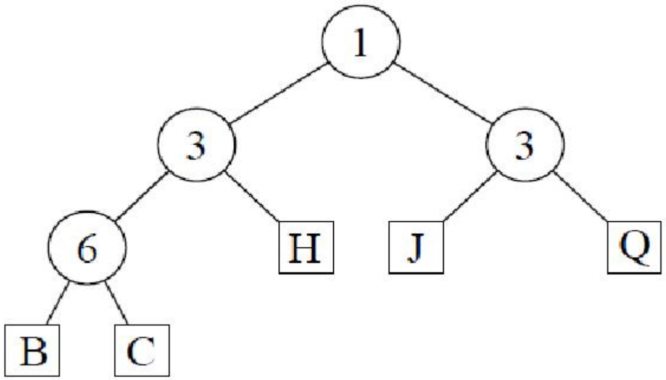
Patricia

- Exemplo:
- Dadas as chaves de 6 bits:

```
B = 010010
C = 010011
H = 011000
J = 100001
M = 101000
```



Patricia - Inserção de K

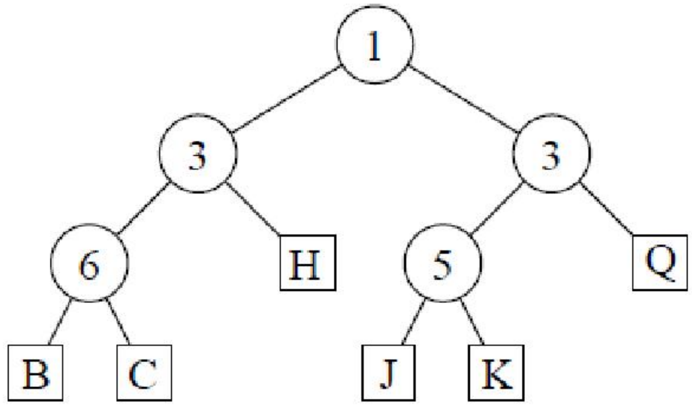


- Inserir a chave `K = 100010`: pesquisa inicia pela raiz e termina quando se chega ao nó externo contendo J
- Os índices dos bits nas chaves estão ordenados da esquerda para a direita. Bit de índice 1 de K é 1 $\rightarrow$ a subárvore direita. Bit de índice 3 $\rightarrow$ subárvore esquerda que, neste caso, é um nó externo
- Chaves J e K mantêm o padrão de bits `1x0xxx`, assim como qualquer outra chave que seguir este caminho de pesquisa

Patricia - Inserção de K

- Novo nó interno repõe o nó J, e este com nó K serão os nós externos descendentes

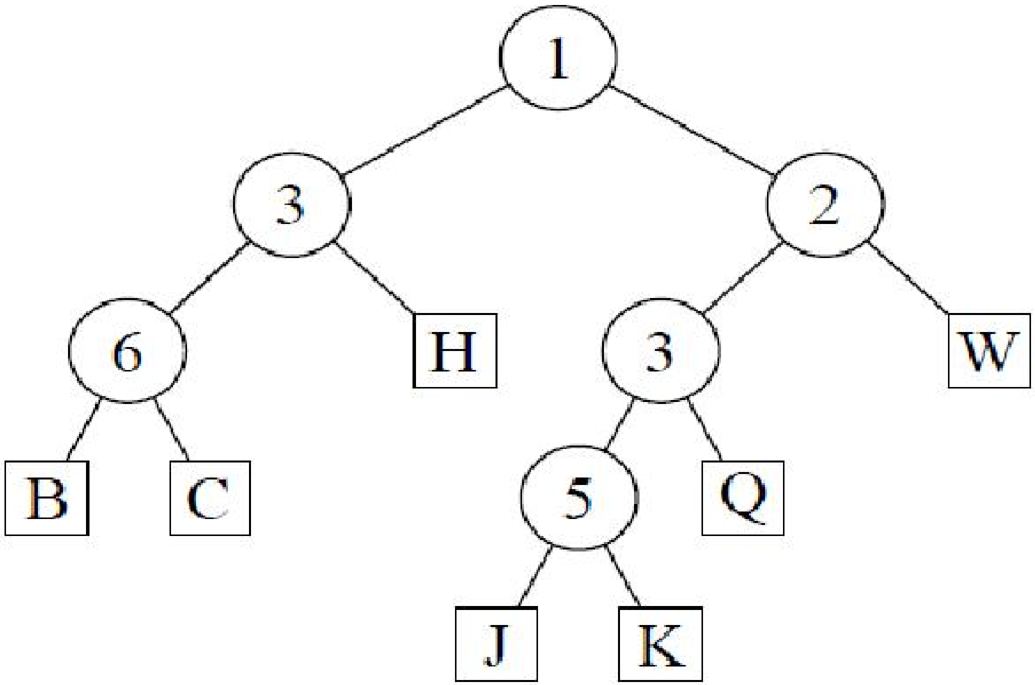
- O índice do novo nó interno é dado pelo 1^o bit diferente das 2 chaves em questão, que é o bit de índice 5. Para determinar qual será o descendente esquerdo e o direito, verifique o valor do bit 5 de ambas as chaves.



Patricia - Inserção de W

- A inserção de `W = 110110` ilustra outro aspecto
- Os bits das chaves K e W são comparados a partir do primeiro para determinar em qual índice eles diferem, sendo, neste caso, os de índice 2
- Ponto de inserção agora será no caminho de pesquisa entre os nós internos de índice 1 e 3
- Cria-se aí um novo nó interno de índice 2, cujo descendente direito é um nó externo contendo W e cujo descendente esquerdo é a subárvore de raiz de índice 3

Patricia - Inserção de W



Transformação de Chave (*Hashing*)

- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa
- Hash significa:
  - Fazer picadinho de carne e vegetais para cozinhar
  - Fazer uma bagunça. (*Webster's New World Dictionary*)

Hashing

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:

  1. Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela

  2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para lidar com **colisões**

- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma

- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões

Hashing

- **Paradoxo do aniversário** (Feller,1968): em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia

- Se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:
$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} = \prod_{i=1}^{n} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$

Hashing

- Alguns valores de p para diferentes valores de N,onde $M = 365$

N	p
10	0,883
22	0,524
23	0,493
30	0,303

- Para N pequeno a probabilidade p pode ser aproximada por $p \approx \frac{N(N-1)}{730}$. Por exemplo, para $N = 10$, então $p \approx 87,7\%$

Funções de Transformação

- Função de Transformação: deve mapear chaves em inteiros dentro do intervalo $[0..M-1]$, onde M é o tamanho da tabela

- Função de Transformação ideal:
  - Simples de ser computada
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer

Funções de Transformação - Método mais usado

- Resto da divisão por M

$$h(K) = K mod M$$

onde K é um inteiro correspondente à chave

- Cuidado na escolha do valor de M

- M deve ser um número primo, mas não qualquer primo: devem ser evitados os números primos obtidos a partir de

$$b^i \pm j$$

  – onde b é a base do conjunto de caracteres (geralmente $b = 64$ para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros

Transformação de Chaves Não-Numéricas

- Devem ser transformadas em números
$$K = \sum_{i=1}^{n} Chave[i] \times p[i]$$

- n é o número de caracteres da chave

- $Chave[i]$ corresponde à representação ASCII do i-ésimo caractere da chave

- $p[i]$ é um inteiro de um conjunto de pesos gerados randomicamente para $1 \leq i \leq n$

- Vantagem de se usar pesos: Dois conjuntos diferentes de pesos $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, levam a duas funções de transformação $h_1(K)$ e $h_2(K)$ diferentes

Transformação de Chaves Não-Numéricas

- Função para gerar um peso para cada caractere de uma chave constituída de n caracteres:

```
void gerarPesos(TipoPesos p) {
  /* -Gera valores randomicos entre 1 e 10.000- */
  int i;
  struct timeval semente;
  /* Utilizar o tempo como semente para a funcao srand() */
  gettimeofday(&semente,NULL);
  srand((int)(semente.tv_sec + 1000000*semente.tv_usec));

  for (i = 0; i < n; i++){
     p[i] =  1+(int) (10000.0*rand()/(RAND_MAX+1.0));
  }
}
```

Transformação de Chaves Não-Numéricas

- Função de transformação:

```
Indice h(Chave chave, Peso p) {
  int i;
  unsigned int soma = 0;
  int comp = strlen(chave);
  for (i = 0; i < comp; i++) {
    soma += (unsigned int)chave[i] * p[i];
  }
  return (soma % M);
}
```

Hash - Solução de Colisões

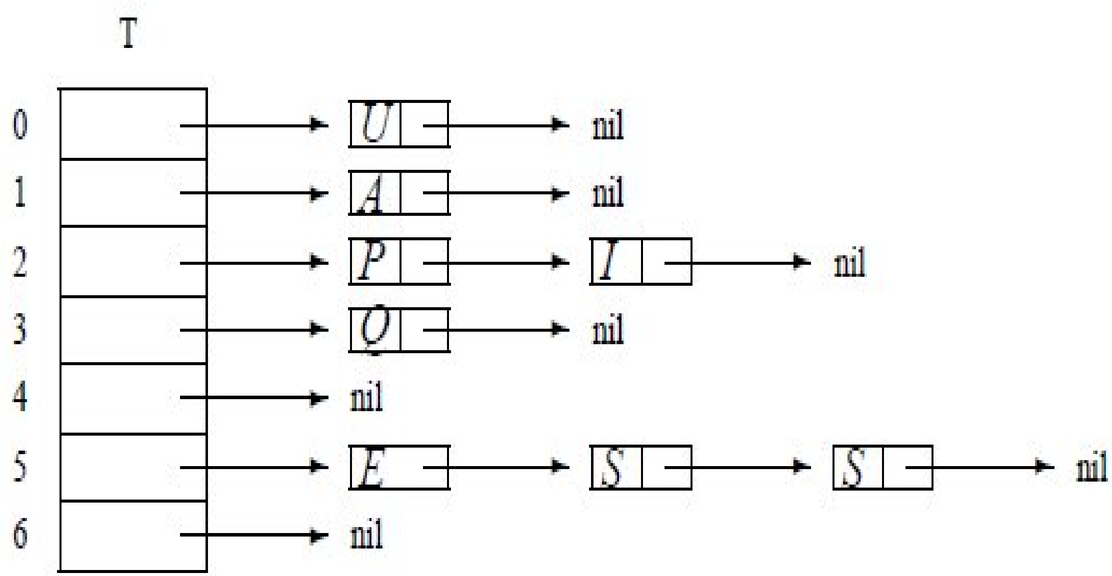
- Como resolver colisões?

Hash - Solução de Colisões

- Usando Listas Lineares!
- Cada endereço da tabela possui uma lista encadeada
- Todas as chaves com um mesmo endereço são encadeadas na respectiva lista
- **Exemplo**: Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(chave) = Chave$ mod M é utilizada para $M = 7$, qual é o resultado da inserção das chaves *P E S Q U I S A* na tabela?
- Considere: $h(A) = h(1) = 1, h(E) = h(5) = 5, h(S) = h(19) = 5$, e assim por diante

Hash - Solução de Colisões

- *P E S Q U I S A*



Hash - Estrutura de Dados

```
typedef char Chave[n];
typedef unsigned int Peso[n];

typedef struct Item {
  /* outros componentes */
  Chave chave;
} Item;
typedef unsigned int Indice;
typedef struct Celula* Apontador;
typedef struct Celula {
  Item item;
  Apontador prox;
} Celula;

typedef struct Lista {
  Celula *primeiro, *ultimo;
} Lista;
```

```
typedef Lista Dicionario[M];
Dicionario tabela;
Item elemento;
Peso p;
Apontador i;
FILE *arq;
```

Hash - Operações

```
void inicializar(Dicionario t) {
  int i;
  for (i = 0; i < M; i++)
    criarLista(&t[i]);
}

Apontador pesquisar(Chave ch, Peso p, Dicionario t);
```

Hash - Operações

```
Apontador pesquisar(Chave ch, Peso p, Dicionario t) {
   /* -- Obs.: Apontador de retorno aponta para
         o item anterior da lista -- */
  Indice i;  Apontador ap;  i = h(ch, p);
  if (ehVazia(t[i]))  return NULL;   /* pesquisa
  else {                                sem sucesso */
    ap = t[i].primeiro;
    while (ap->prox->prox != NULL &&
           strncmp(ch, ap->prox->item.chave, sizeof(Chave))){
       ap = ap->prox;
    }
    if (!strncmp(ch, ap->prox->item.chave, sizeof(Chave)))
       return ap;
    else
      return NULL;   /* pesquisa sem sucesso */
  }
}
```

```
/*strncmp: < 0 se str1 é menor do que str2;
           > 0 se str1  é maior do que str2,
           e 0 se forem iguais.
          (terceiro parâmetro é o número de caracteres
          que devem ser comparados. Se qualquer
          uma das strings é menor do que len,
          então o tamanho desta string será usado
          para comparação*/
```

Hash - Operações

```
void inserir(Item x, Peso p, Dicionario t);
```

```
void retirar(Item x, Peso p, Dicionario t) {
```

Hash - Operações

```
void inserir(Item x, Peso p, Dicionario t) {
  if (pesquisar(x.chave, p, t) == NULL)
    ins(x, &t[h(x.chave, p)]);
  else
    printf(" Registro ja  esta  presente\n");
}
void retirar(Item x, Peso p, Dicionario t) {
  Apontador ap;
  ap = pesquisar(x.chave, p, t);
  if (ap == NULL)
    printf(" Registro nao esta  presente\n");
  else
    ret(ap, &t[h(x.chave, p)], &x);
}
```

Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada é N/M, onde N representa o número de registros na tabela e M o tamanho da tabela.

- Operações pesquisar, inserir e retirar custam $O(1 + N/M)$ operações em média (constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista). Para valores de M próximos de N, o tempo se torna constante, isto é, independente de N

Endereçamento Aberto

- Se o número de registros pode ser estimado, não há necessidade de usar apontadores para armazenar os registros

- Existem vários métodos para armazenar N registros em uma tabela de tamanho $M > N$: utilizam os lugares vazios na própria tabela para resolver as colisões. (Knuth, 1973)

- **Endereçamento aberto**: chaves são armazenadas na própria tabela, sem o uso de apontadores explícitos

- Várias propostas para a escolha de localizações alternativas. Mais simples: **hashing linear**. Posição h_j na tabela é dada por: $hj = (h(x) + j) mod M; para 1 \leq j \leq M - 1$

Endereçamento Aberto - Exemplo

- i-ésima letra do alfabeto: representada pelo número i; função de transformação $h(Chave) = Chave \bmod M$ utilizada para $M = 7$

- Resultado da inserção das chaves $LUNES$ na tabela, usando hashing linear para resolver colisões:

- $h(L) = h(12) = 5, h(U) = h(21) = 0, h(N) = h(14) = 0, h(E) = h(5) = 5, e h(S) = h(19) = 5$

	T
0	U
1	N
2	S
3	
4	
5	L
6	E

Endereçamento Aberto - Estrutura

```
#define VAZIO           "!!!!!!!!!!"
#define RETIRADO        "**********"
#define M               7
#define n               11   /* Tamanho da chave */
typedef unsigned int Apontador;
typedef char Chave[n];
typedef unsigned Peso[n];
typedef struct Item {
  /* outros componentes */
  Chave chave;
} Item;
typedef unsigned int Indice;
typedef Item Dicionario[M];
Dicionario Tabela;
Peso p;
Item Elemento;
FILE *arq;
int j, i;
```

Endereçamento Aberto - Operações

```
void inicializar(Dicionario t) {
 int i;
 for (i = 0; i < M; i++)
    memcpy(t[i].chave, VAZIO, n);
}
```

Endereçamento Aberto - Operações

```
Apontador pesquisar(Chave chave, Peso p, Dicionario t) {
  unsigned int  i = 0;
  unsigned int  inicial;

  inicial = h(chave, p);
  while (strcmp(t[(inicial + i) % M].chave,VAZIO) != 0 &&
         strcmp(t[(inicial + i) % M].chave, chave) != 0 &&
         i < M)
    i++;
  if (strcmp(t[(inicial + i) % M].chave, chave) == 0)
    return ((inicial + i) % M);
  else
    return M;   /* pesquisa sem sucesso */
}
```

Endereçamento Aberto - Operações

```
void inserir(Item x, Peso p, Dicionario t) {
  unsigned int i = 0;
  unsigned int inicial;
  if (pesquisar(x.chave, p, t) < M) {
    printf("Elemento ja esta presente\n");
    return;
  }
  inicial = h(x.chave, p);
  while ( strcmp ( t[(inicial + i) % M].chave,VAZIO) != 0 &&
  strcmp ( t[(inicial + i) % M].chave, RETIRADO) != 0 &&
  i < M)
     i++;
  if (i < M) {
    strcpy (t[(inicial + i) % M].chave, x.chave);
    /* Copiar os demais campos de x, se existirem */
  }
  else
```

```
    printf(" Tabela cheia\n");
}
```

Endereçamento Aberto - Operações

```
void retirar(Chave chave, Peso p, Dicionario t) {
  Indice i;
  i = pesquisar(chave, p, t);
  if (i < M)
    memcpy(t[i].chave, RETIRADO, n);
  else
    printf("Registro nao esta presente\n");
}
```

Endereçamento Aberto - Análise

- Seja $\alpha = N/M$ o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é $C(n) = \frac{1}{2}(1 + \frac{1}{1-\alpha})$

- O *hashing linear* sofre de um mal chamado **agrupamento**(**clustering**) (Knuth, 1973)

- Ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas

- Apesar de o hashing linear ser um método relativamente pobre para resolver colisões os resultados apresentados são bons

- O melhor caso, assim como o caso médio, é $O(1)$

Transformação da Chave: Vantagens e Desvantagens

- Vantagens:
  - Alta eficiência no custo de pesquisa, que é $O(1)$ para o caso médio
  - Simplicidade de implementação
- Desvantagens:
  - Custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo
  - Pior caso é $O(N)$