Algoritmos e Estruturas de Dados II

MCCC002-23

Árvores

Prof. Carlo Kleber
carlo.kleber@ufabc.edu.br

Sumário

1. Introdução
2. Definição Básica
3. Nomeclatura
4. Árvores Ordenadas
5. Árvores Binárias
6. Árvores Binárias de Busca
7. Árvores Balanceadas
8. Árvores de Difusão
9. Árvores B
10. Árvores Digitais
11. Síntese Final

Parte Prática e Referências

1. Introdução

- Neste módulo analisamos tópicos relacionados a **Árvores**

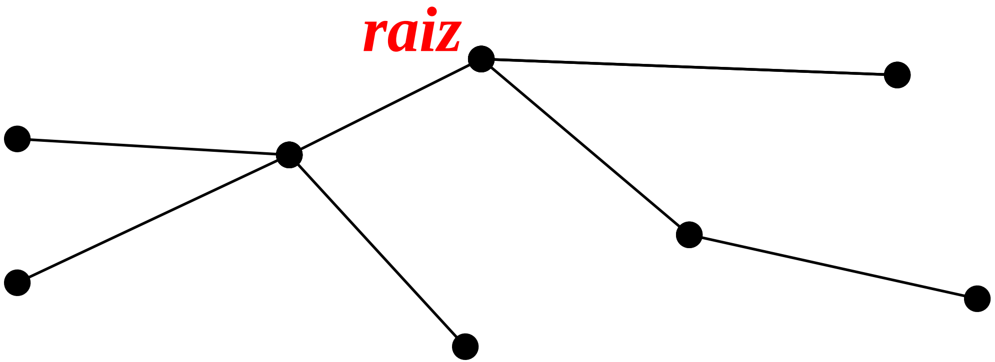
- Em **diversas aplicações** necessita-se de estruturas mais complexas do que as puramente **sequenciais**.

- Entre essas, destacam-se as **árvores**, por existirem inúmeros problemas práticos que podem ser modelados por meio delas. Além disso, em geral, admitem um tratamento computacional **relativamente** **simples e eficiente**.

2. Definição Básica

Árvores são estruturas das mais usadas em computação. São usadas para representar **hierarquias.**

Uma **árvore** pode ser entendida como um **grafo acíclico conexo** onde um dos **vértices ou nós**, chamado **raiz da árvore**, é **diferenciado** dos demais.



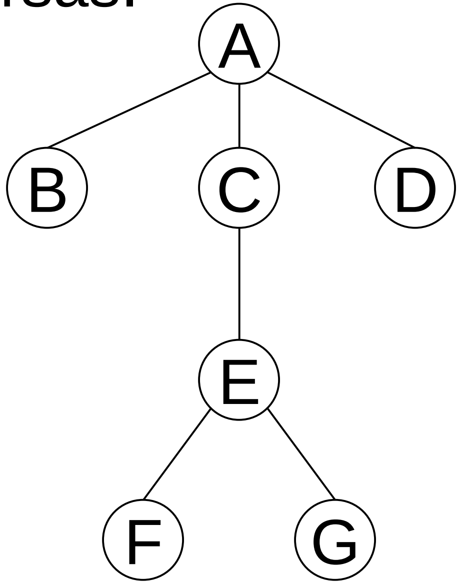
2. Definição Básica

Uma **árvore** enraizada ***T***, ou simplesmente **árvore**, é um conjunto finito de elementos denominados **vértices** ou **nós** tal que:

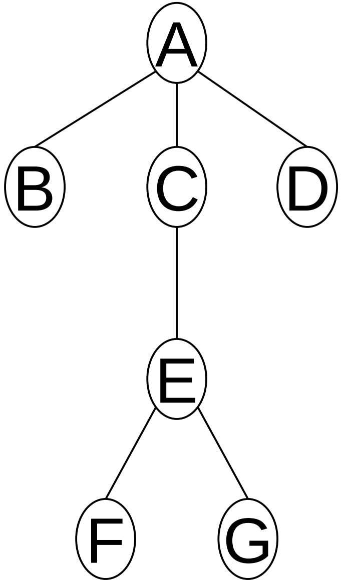
- ***T*** **= ∅**, e a árvore é dita **vazia** ou
- Existe um **nó** especial ***r***, chamado **raiz** de ***T***; os nós restantes constituem **um único conjunto vazio** ou são **divididos** em ***m*** **≥ 1** conjuntos disjuntos **não vazios**, as **subárvores** de ***r***, ou simplesmente **subárvores**, cada qual por sua vez **uma árvore**.

2. Definição Básica

Uma **floresta** é um **conjunto** de **árvores.** É comum associar-se **rótulos** aos **nós das árvores** para que possamos nos referir a eles. **Na prática**, os **nós** são usados para **guardar informações** diversas.



3. Nomeclatura

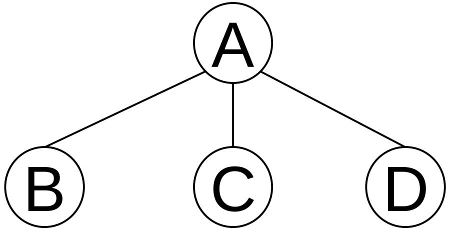


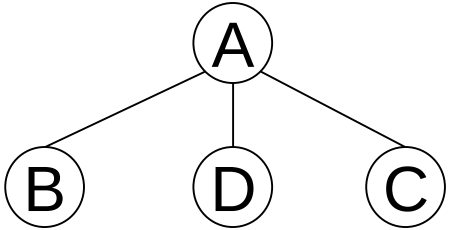
- A é o **pai** de B, C e D.
- B, C e D são **filhos** de A.
- B, C e D são **irmãos.**
- A é um **ancestral** de G.
- G é um **descendente** de A.
- B, D, F e G são **nós folhas** (nós que não têm filhos).
- A, C e E são nós **interiores** (nós que têm filhos).
- Nós **internos** são os nós que contém as chaves.
- Nós **externos** são nós que não pertencem à árvore, são representados pelas **subárvores vazias** ou **nós nulos.**
- O **grau** do nó A é **3.**
- O **comprimento do caminho** entre C e G é **2**, que corresponde ao **número de arestas** entre os nós.
- O **nível** de A (**raiz**) é **1** e o de G é **4.**
- A **altura** da árvore é **4** (nível do nó mais profundo).

4. Árvores Ordenadas

- Se é considerada a **ordem entre os filhos** de cada nó, a árvore é chamada de **ordenada.**

- Pode-se definir o conceito de **árvores isomorfas** quando elas têm a **mesma relação de incidência** entre nós mas são **desenhadas** de forma diferente, isto é, são **distintas** quando consideradas como **árvores ordenadas.**





Exemplo de **árvores isomorfas.**

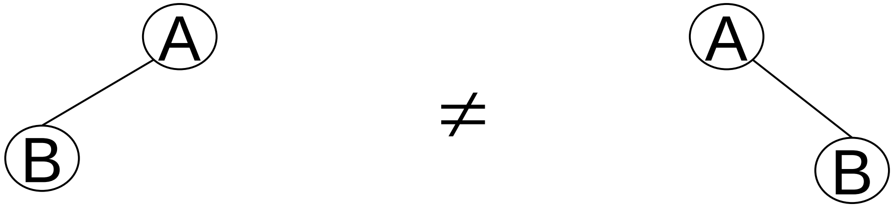
5. Árvores Binárias

5. Árvores Binárias

Uma **árvore binária** é:

- Uma **árvore vazia** ou
- Um **nó raiz** e **duas subárvores binárias** denominadas **subárvore direita** e **subárvore esquerda,** sendo que **estas** podem ser **vazias (uma ou as duas)** ou **não.**

Observe que em uma **árvore binária** os filhos de cada nó têm nomes (**filho esquerdo** e **filho direito**)



Exemplo: Essas árvores binárias são **diferentes** entre si.

5.1 Subárvores Vazias

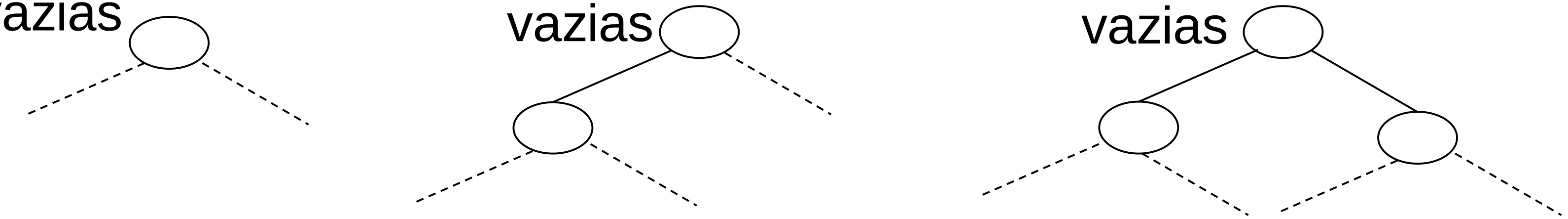
Se uma árvore **binária** tem ***n* > 0 nós**, então ela possui ***n*+1 subárvores vazias.** Para ver isso, observe que:

- Uma árvore binária com **um só nó** tem **2 subárvores vazias**
- Sempre que **"penduramos"** um **novo nó** numa árvore binária, o **número de nós cresce de 1** e o de **subárvores vazias** também **cresce de 1**

1 nó, **2** subárvores vazias

2 nós, **3** subárvores vazias

3 nós, **4** subárvores vazias



5.1 Tipos

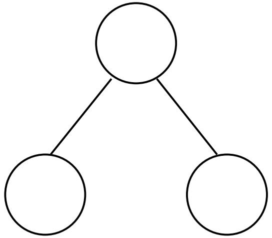
- Uma árvore binária é **estritamente binária se e somente se todos** os seus **nós** têm **0** ou **2 filhos.**

- Uma **árvore binária completa** é aquela em que **todas as subárvores vazias** são **filhas de nós** do **último ou penúltimo nível.**

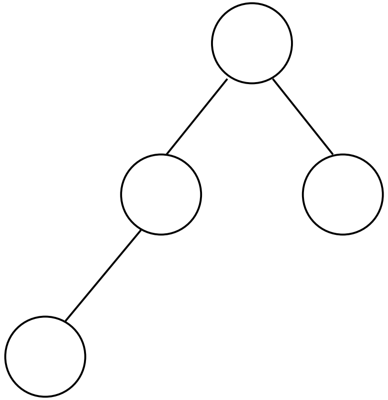
- Uma **árvore binária cheia** é aquela em que **todas as subárvores vazias** são **filhas de nós** do **último nível.**

5.1 Tipos

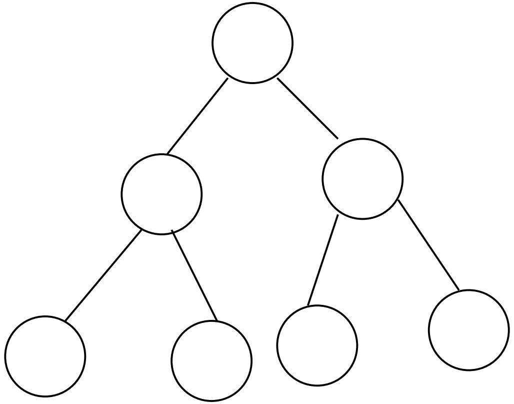
Estritamente binária



Completa



Cheia

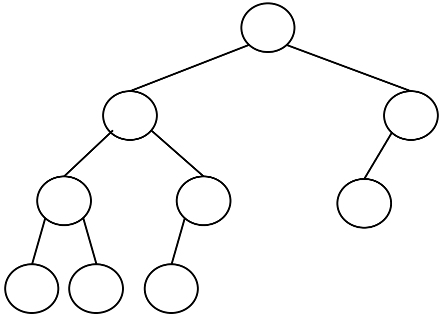


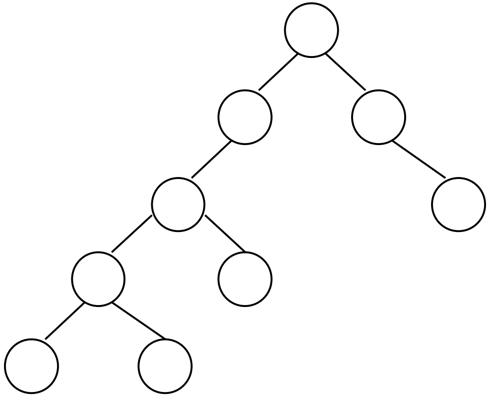
5.2 Altura

- O **processo de busca** em árvores é normalmente feito **a partir da raiz** **na direção** de **alguma de suas folhas.** Naturalmente, são de especial interesse as árvores com a **menor altura possível**
- A **altura mínima** de uma **árvore binária** com $n > 0$ **nós** é $h = 1 + \lfloor \log_2 n \rfloor$

Ex.: se n = 9 então
h = 1 + 3 = 4

n = 9, tem **altura mínima**

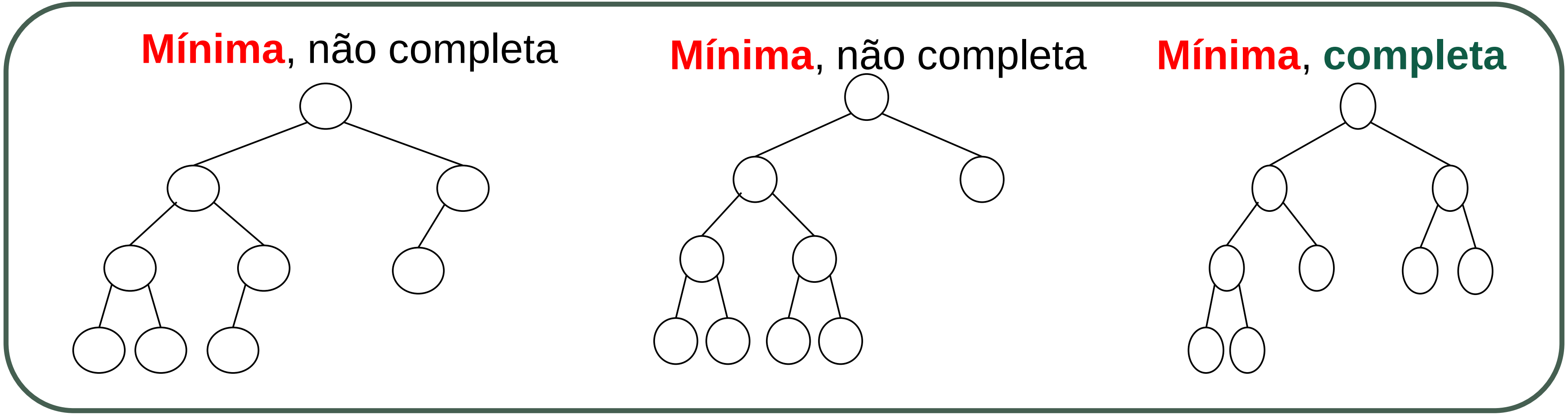




n = 9, **não** tem **altura mínima**

5.2 Altura

- Se uma **árvore binária *T*** com ***n* > 0** nós é **completa,** então ela **tem altura mínima.**
- Para ver isso, observe que mesmo que uma árvore de **altura mínima não seja completa** é possível **torná-la completa** movendo folhas para **níveis mais altos.**

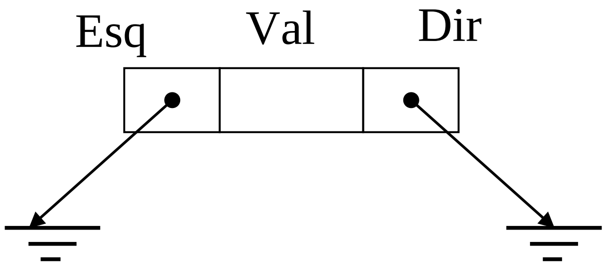


5.3 Implementação

- **Via-de-regra**, árvores são implementadas **com ponteiros.**

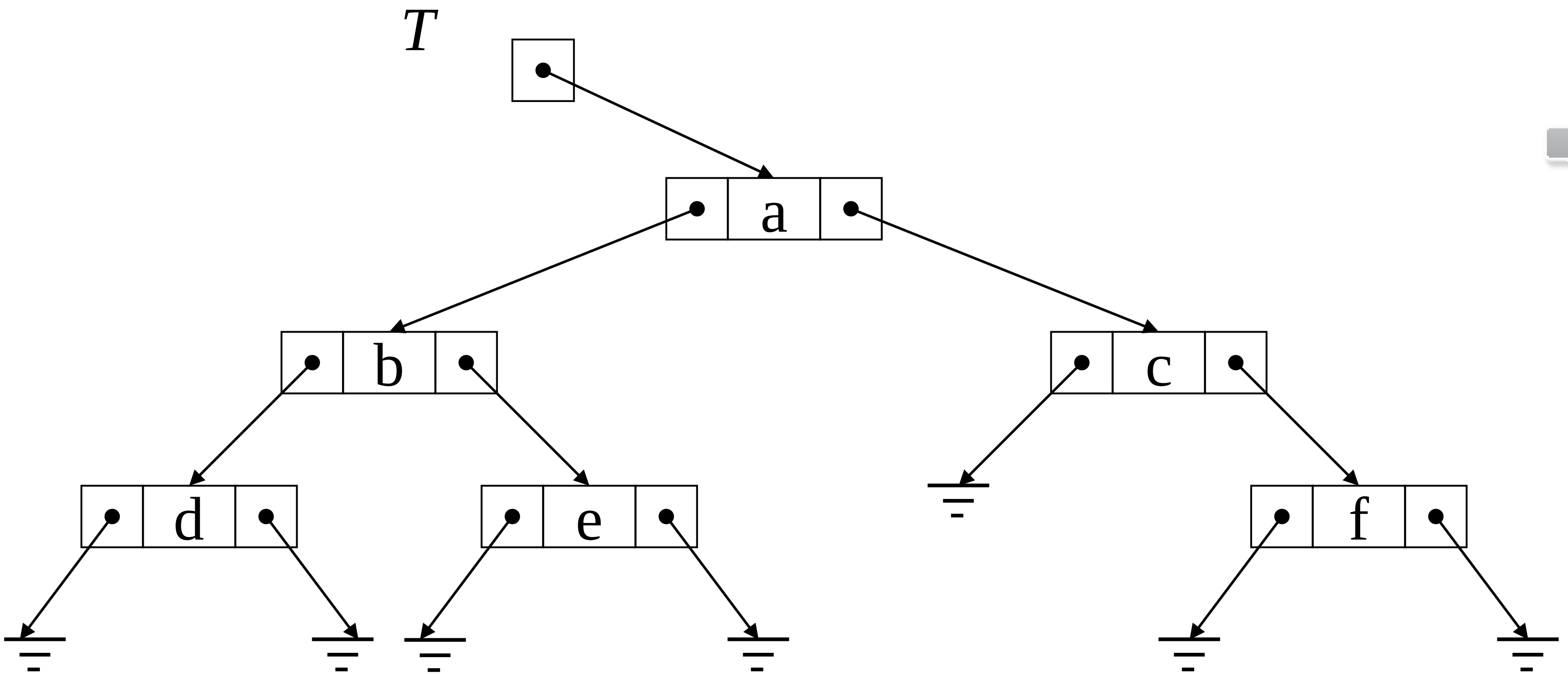
- Cada **nó *X*** contém **3 campos**:

  X.Val : valor armazenado no nó.
  X.Esq : ponteiro p/ árvore esquerda.
  X.Dir : ponteiro p/ árvore direita.



5.3 Implementação

Uma **árvore** é representada por um **ponteiro** para seu **nó raiz**

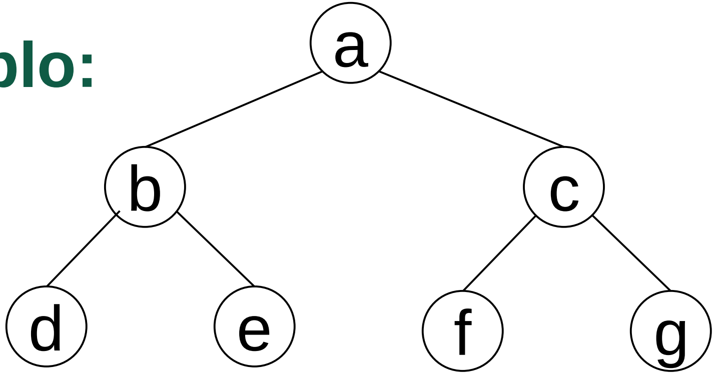


5.4 Percurso

Existem essencialmente **03 (três) ordens** de se percorrer os nós de uma **árvore binária.**

- **Pré-ordem:** raiz, esquerda, direita.
- **Pós-ordem:** esquerda, direita, raiz.
- **Ordem simétrica (In-ordem):** esquerda, raiz, direita.

Por exemplo:

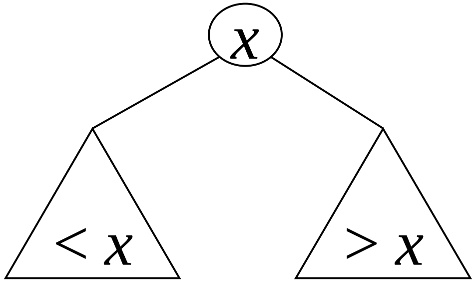


Pré-ordem: a, b, d, e, c, f, g
Pós-ordem: d, e, b, f, g, c, a
Simétrica: d, b, e, a, f, c, g

6. Árvores Binárias de Busca (ABB)

6. Árvores Binárias de Busca (ABB)

- Uma **maneira simples** e popular de **implementar dicionários** é uma **estrutura de dados** conhecida como **árvore binária de busca.**
- Numa **árvore binária de busca**, todas as chaves **dos nós** da **subárvore à esquerda** de um nó **contendo uma chave *x*** são **menores que *x*** e, ainda, todas as chaves dos nós da **subárvore à direita** são **maiores que *x*.**
- **Supõe-se** que **não** existem **chaves de valores iguais.**

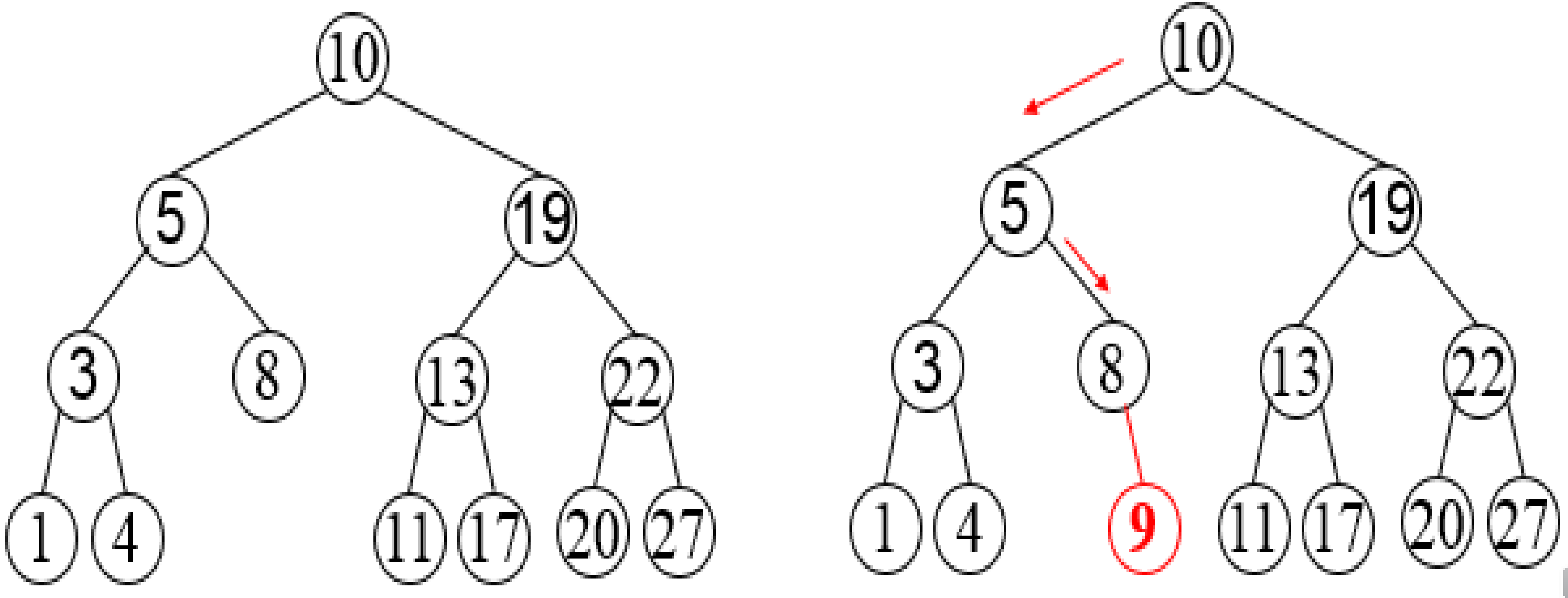


6.1 Busca e Inserção em ABB

```
proc Buscar (Chave x, Árvore T)
{
    se T = Nulo então retornar falso
    se x = T^.Val então retornar verdadeiro
    se x < T^.Val então retornar Buscar (x, T^.Esq)
    retornar Buscar (x, T^.Dir)
}
proc Inserir (Chave x, var Árvore T)
{
    se T = Nulo então {
        T ← Alocar (NoArvore)
        T^.Val, T^.Esq, T^.Dir ← x, Nulo, Nulo
    }
    senão {
        se x < T^.Val então Inserir (x, T^.Esq)
        se x > T^.Val então Inserir (x, T^.Dir)
    }
}
```

6.1 Busca e Inserção em ABB

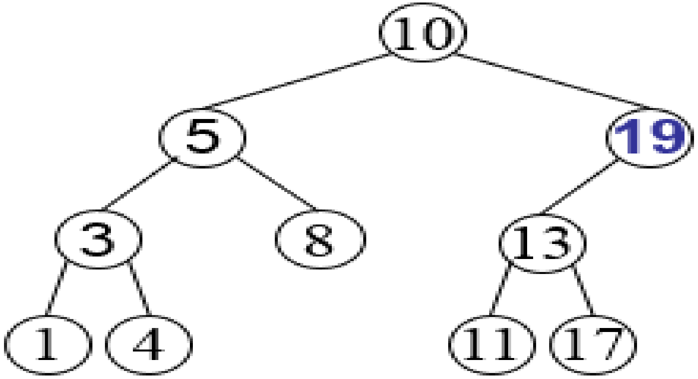
Exemplo: *Inserir* (9, *T*)

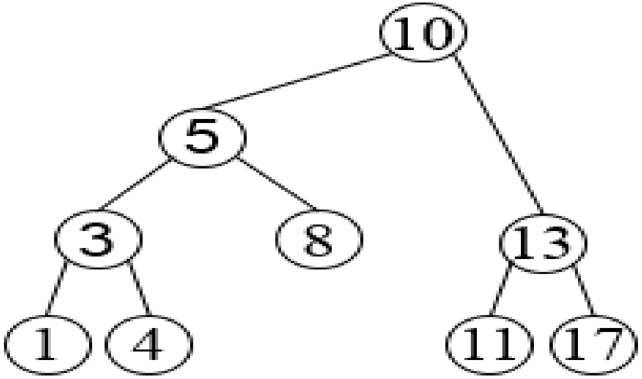


6.2 Busca e Remoção em ABB

- Para **remover uma chave *x*** de uma **árvore *T*** temos que distinguir os seguintes casos:
  - **Caso 1) *x* está numa folha de *T*** : neste caso, a folha pode ser simplesmente removida
  - **Caso 2) está num nó que tem sua subárvore esquerda ou direita vazia**: neste caso o nó é removido e substituído pela subárvore não nula

Exemplo: *Remover* (**19**, *T*)



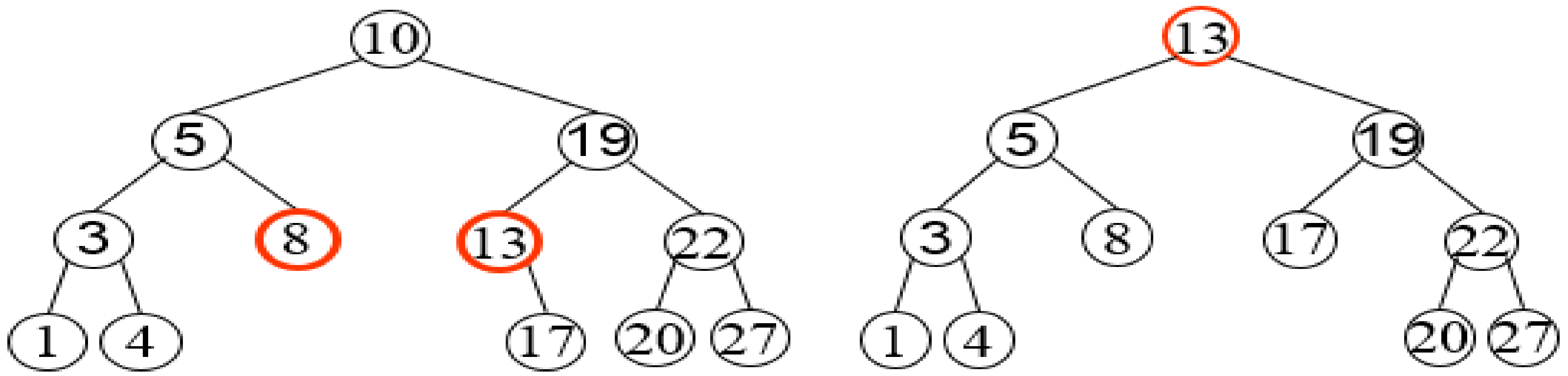


6.2 Busca e Remoção em ABB

- **3) Se *x* está num nó em que ambas subárvores são não nulas:** é preciso encontrar uma chave *y* que a possa substituir.
  - Há duas chaves candidatas naturais:
    - **Caso 3.1)** A **menor** das chaves maiores que *x* ou
    - **Caso 3.2)** A **maior** das chaves menores que x

Exemplo: *Remover* (10, *T*)

Neste exemplo, optou-se por **3.1**



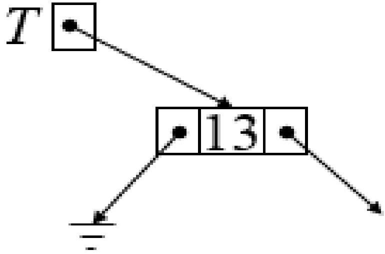
6.2 Busca e Remoção em ABB

- **Caso 3.1: A rotina** considera a situação em que **se desce na árvore** até encontrar um **filho esquerdo** que **não tem** mais **filho esquerdo**, apenas **filho direito.** A **chave retornada** é justamente **esse filho esquerdo** e a **raiz da árvore** passa a ser o **filho direito.**

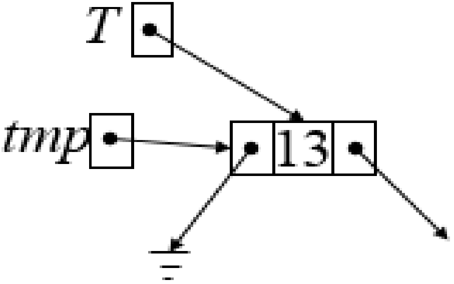
```
proc RemoverMenor (var Árvore T)
{
    se T^.Esq = Nulo então
   {
        tmp ← T
        y ← T^.Val
        T ← T^.Dir
        Liberar (tmp)
        retornar y
   }
    senão
        retornar RemoverMenor (T^.Esq)
}
```

6.2 Busca e Remoção em ABB

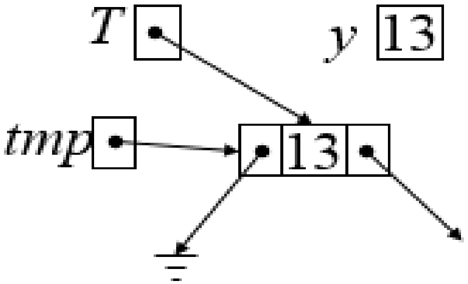
1 Árvore sem **filho esquerdo**, apenas **direito.**



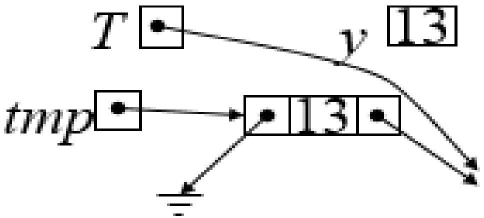
2



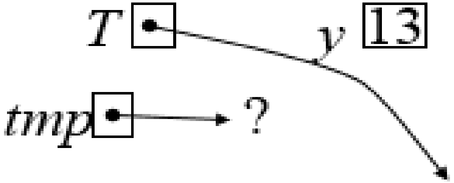
3



4



5 Árvore passa a ser o então filho **direito.**



6.2 Busca e Remoção em ABB

```
proc Remover (Chave x, var Árvore T)
{
    se T ≠ Nulo então
        se x < T^.val então Remover (x, T^.Esq)
        senão se x > T^.val então Remover (x, T^.Dir)
        senão
            se T^.Esq = Nulo então   % Caso 1 %
            {
                tmp ← T
                T ← T^.Dir
                Liberar (tmp)
            }
            senão se T^.Dir = Nulo então  % Caso 2 %
            {
                tmp ← T
                T ← T^.Esq
                Liberar (tmp)
            }
            senão T^.Val ← RemoverMenor (T^.Dir)    % Caso 3.1 %
}
```

6.3 Complexidade em ABB

- A **busca em uma árvore binária** tem complexidade ***O*(*h*),** onde *h* é a **altura da árvore.**

- A **altura** de uma árvore é, no **pior** caso, ***n*,** e, no **melhor** caso, $\lfloor \log_2 n \rfloor + 1$ (altura de árvore **completa**).

- **Inserção** e **remoção** também têm complexidade de **pior caso *O*(*h*)** e, portanto, a **inserção** ou a **remoção** de *n* chaves toma tempo $O(n^2)$, no **pior caso, ou** $O(n \log n)$, se pudermos **garantir** que árvore tem **altura logarítmica.**

6.4 Altura Logarítmica

- Pode-se **garantir** a **construção** de uma **ABB de altura logarítmica** para uma dada **coleção de chaves** se, toda vez que temos de escolher uma chave para inserir, optamos pela **MEDIANA.**
- Neste caso, a construção da **ABB** leva **O(n log n).**

6.4 Altura Logarítmica

Nem sempre, **entretanto**, podemos garantir que **conhecemos todas as chaves de antemão**. O que esperar em geral?

- A **altura** da ABB final **depende** da **ordem de inserção.**
- Temos **n!** possíveis **ordens de inserção.**
- Porém, se **todas as ordens são igualmente prováveis**, no **caso médio**, teremos uma árvore de altura ≈ **1 + 2,4 log n**

Ou seja, **no caso médio**, a **altura de uma ABB é O(log n).**

6.5 ABB Ótima

- A **ABB** **ótima** é aquela árvore que **minimiza** o **número total de comparações** que precisamos fazer durante a **busca** de uma **chave**.
- Como calcular o **número de comparações**?

- **EXTRACLASSE: Szwarcfiter, J. L.; Markenzon, L. Estruturas de dados e seus algoritmos. Editora LTC, 2ª edição, 1994.**

6.5 ABB Ótima

- Como calcular o **número de comparações**?

- Se a **chave buscada** é uma chave s_k pertencente à árvore, o **número de comparações** é o **nível** da chave na árvore, isto é, l_k (comprimento de caminho interno).

- Se a **chave buscada** s_k não pertence à árvore, o **número de comparações** é o **nível** da **subárvore vazia ou nula** (*nó externo*) **menos 1**, que encontramos durante o processo de busca, isto é, $l'_k - 1$ (comprimento de caminho externo).

6.5 ABB Ótima

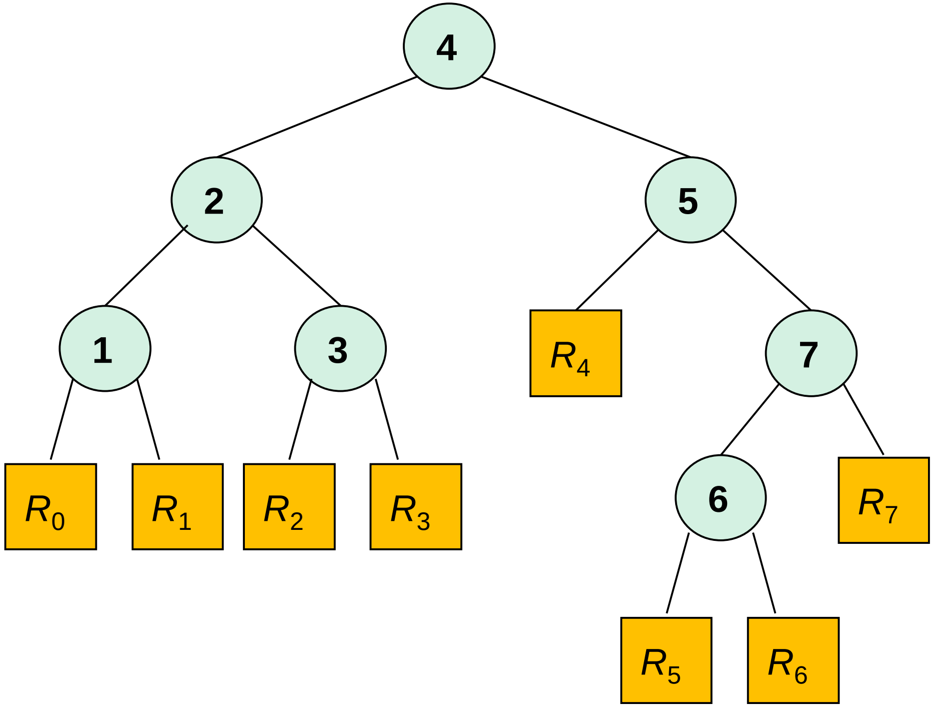
- **Explica-se** que:

- Cada **subárvore nula** R_i corresponde na verdade **a um intervalo entre duas chaves** da árvore, digamos s_i e s_{i+1}, isto é, $s_i < x < s_{i+1}$ para algum i entre 1 e n-1.

- Os **casos extremos** R_0 e R_n correspondem a $x < s_1$ e $s_n < x$.

- O **número de comparações** para encontrar x neste caso, é o nível dessa **subárvore nula menos** 1, isto é, $l'_k - 1$.

6.5 ABB Ótima



- Comprimento de Caminho Interno: $I(T) = \Sigma_{1 \le i \le n}\, l_i$
- Comprimento de Caminho Externo $E(T) = \Sigma_{0 \le i \le n}\, (l'_i - 1)$
- No exemplo,
  $I(T)$=1+2*2+3*3+4=18
  $E(T)$=2+5*3+2*4=25
- **Em geral**, $E(T)=I(T)+n$

6.5 ABB Ótima

- Veja que neste caso estamos **admitindo** que as **frequências de acesso** às diferentes chaves **são todas idênticas.** Ou seja, admitimos uma **probabilidade uniforme de acesso.**

- **ATENÇÃO:** Neste cenário, as **árvores completas** minimizam tanto **E(T)** quanto **I(T)** e são, portanto, **ABB ótimas.**

- **ENTRETANTO**, se a distribuição de probabilidade **não é uniforme**, precisamos de uma **modelagem mais elaborada**.

6.5 ABB Ótima

Sejam:

- ***f_k*** a frequência de acesso à ***k*-ésima** chave, em ***T*** no nível ***l_k***;
- ***f'_k*** a frequência de acesso a chaves que serão buscadas nos **nós externos *R_k***, em ***T*** no nível ***l'_k***

Então, o **custo médio de acesso** é dado por:

$$\boxed{c(T) = \sum_{1 \le k \le n} f_k l_k + \sum_{0 \le k \le n} f'_k (l'_k - 1)}$$

6.5 ABB Ótima

Uma OBSERVAÇÃO sobre o custo de acesso e o custo de construção da ABB Ótima no slide a seguir...

6.5 ABB Ótima

Para **calcular o custo** de acesso da **ABB ótima**, bem como **construir** a mesma, é possível obter um **algoritmo** de complexidade $O(n^2)$, utilizando a propriedade de **monotonicidade** da ABB, assim descrita:

- Se s_k é a **raiz da árvore** ótima para o conjunto $\{s_i \ldots s_j\}$, então a **raiz da árvore** ótima para o conjunto $\{s_i \ldots s_j, s_{j+1}\}$ é s_q para algum $q \geq k$.

- Analogamente, $q \leq k$ para $\{s_{i-1}, s_i \ldots s_j\}$.

Szwarcfiter e Markenzon, 1994 (p. 104)

6.5 ABB Ótima

A **ideia** do algoritmo para **construção da ABB ótima**, se a distribuição de probabilidade de acesso **NÃO é uniforme**, é conseguir:

- *Colocar as **chaves de maior frequência** de acesso **próximas à raiz**, e as de **menor frequência** de acesso **próximas às folhas.***

Por fim, é importante dizer que a **ABB ótima NÃO** é necessariamente **única.**

FIM 1ª. Parte

7. Árvores Balanceadas

7. Árvores Balanceadas

- Um aspecto fundamental do **estudo de árvores de busca** é, naturalmente, o **custo de acesso** a uma chave desejada.

- Vimos que **ABBs completas** garantem buscas utilizando **não mais que $\lfloor \log_2 n \rfloor + 1$ comparações.**

- Mais importante, sabemos que **ABBs**, se **construídas** por **inserção aleatória** de elementos, têm **altura logarítmica** (em *n*) **na média**.

7. Árvores Balanceadas

- Entretanto, **não** podemos **assegurar** que **ABBs** construídas segundo **qualquer ordem** de inserção sempre **têm altura logarítmica.**

- Ou seja, as ABBs se restringem mais a **aplicações estáticas**, pois **inserções** e **remoções** podem **eliminar** a desejável condição de **altura logarítmica.**

7. Árvores Balanceadas

- A **idéia** então é **modificar** os algoritmos **de inserção e remoção** de forma a **assegurar** que a **árvore resultante** é sempre de **altura logarítmica**.

- Este é o conceito de **árvores balanceadas.**

- Há **duas** variantes bem conhecidas:
  - **Árvores AVL**
  - **Árvores Rubro-Negras**

7. Árvores Balanceadas

- **Em geral**, **rebalancear** uma árvore quando **ela deixa de ser completa** (devido a uma **inserção** ou **remoção**, por exemplo) pode ser muito custoso **(até *n* operações)**

- Uma **idéia** é estabelecer **um critério mais fraco** que, não obstante, garanta altura **logarítmica.**

7.1 Árvores AVL

7.1 Árvores AVL

O critério sugerido por Adelson-Velskii e Landis é o de **garantir** o seguinte:

- Para **cada nó** da árvore, **a altura** de sua subárvore **esquerda** e **a altura** de sua subárvore **direita** diferem de **no máximo 1.**

Para manter essa condição, depois de alguma **inserção ou remoção** que **desbalanceie** a árvore, utilizam-se operações de **custo *O*(1),** chamadas **rotações.**

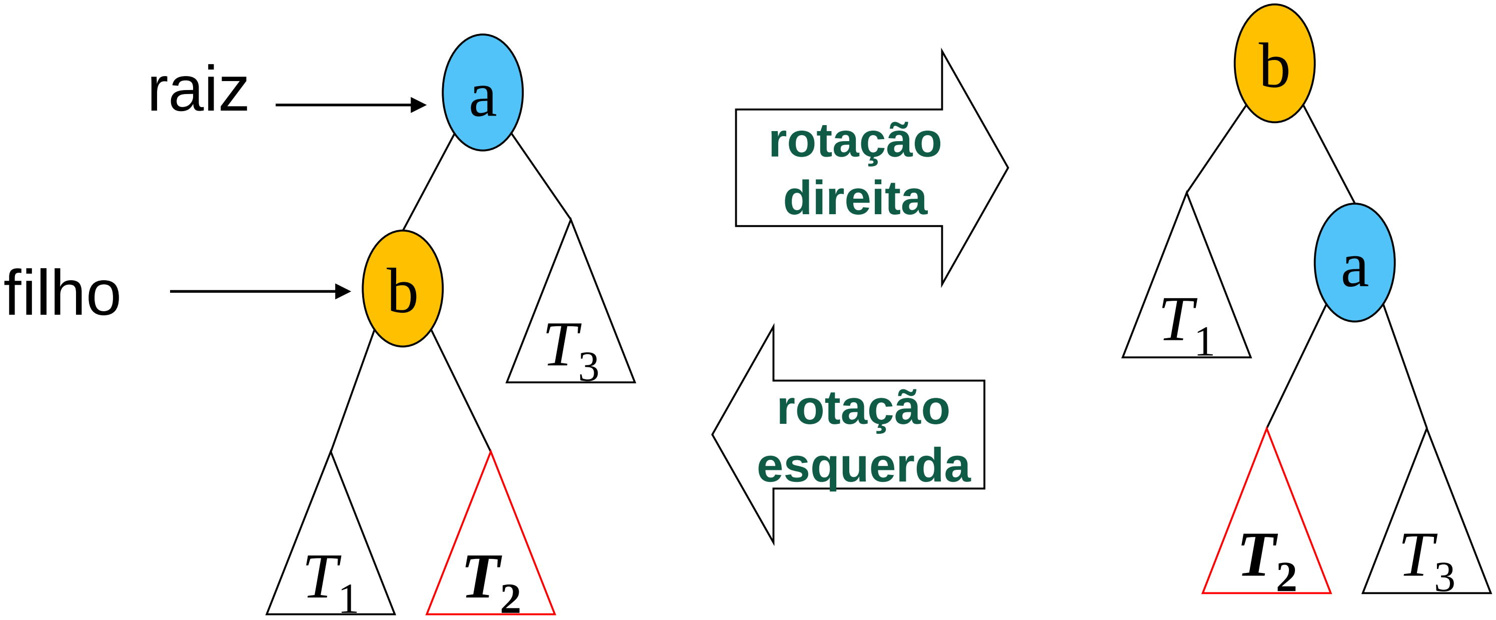
7.1 Árvores AVL

- A **AVL** **não** necessariamente é uma **árvore completa,** mas tem **altura logarítmica** (i.e., $h = O(\log n)$).

- Em particular, se a **diferença** entre a altura da subárvore **esquerda** e altura da subárvore direita **é zero**, então a **AVL é completa.**

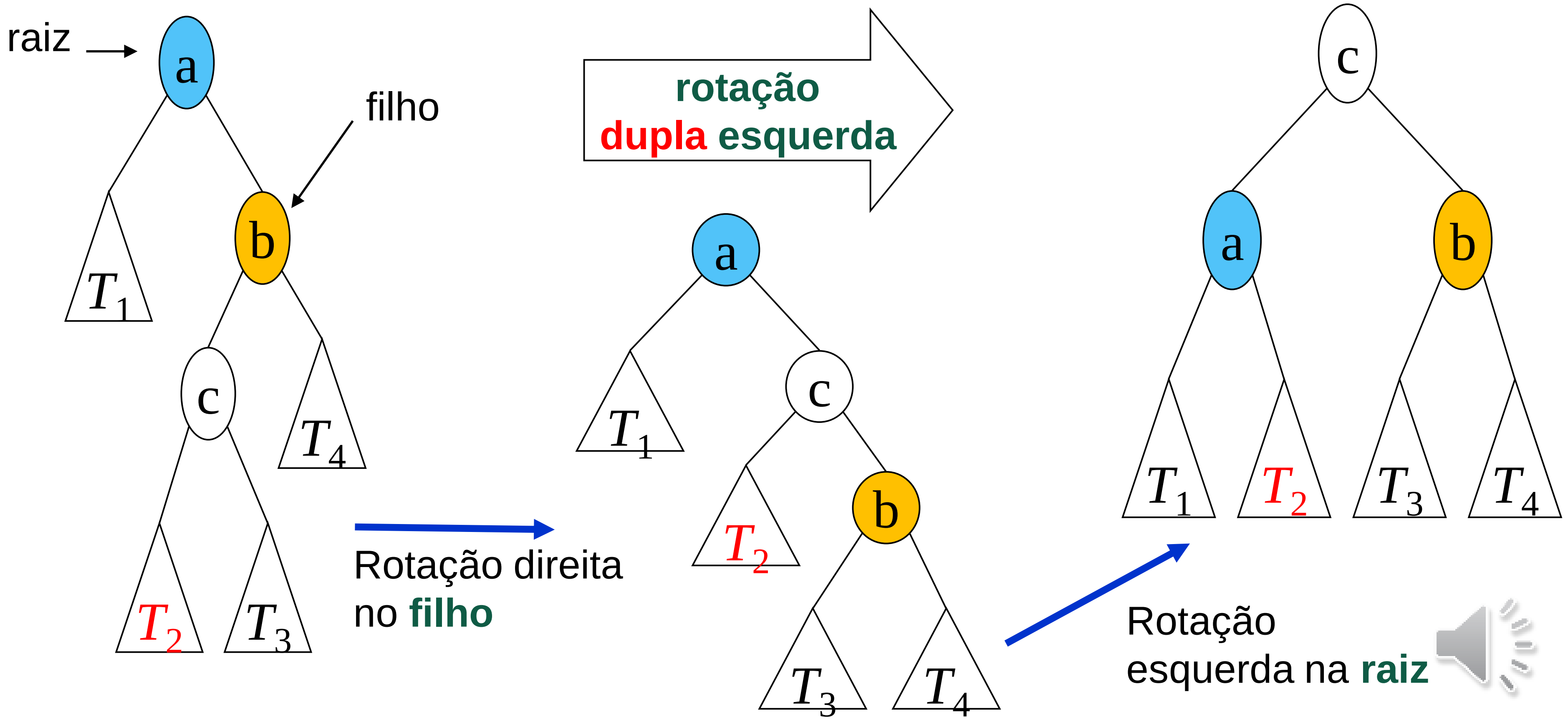
- Por fim, toda árvore **completa** é **AVL**, mas **nem** toda **AVL** é **completa.**

7.1.1 Rotações

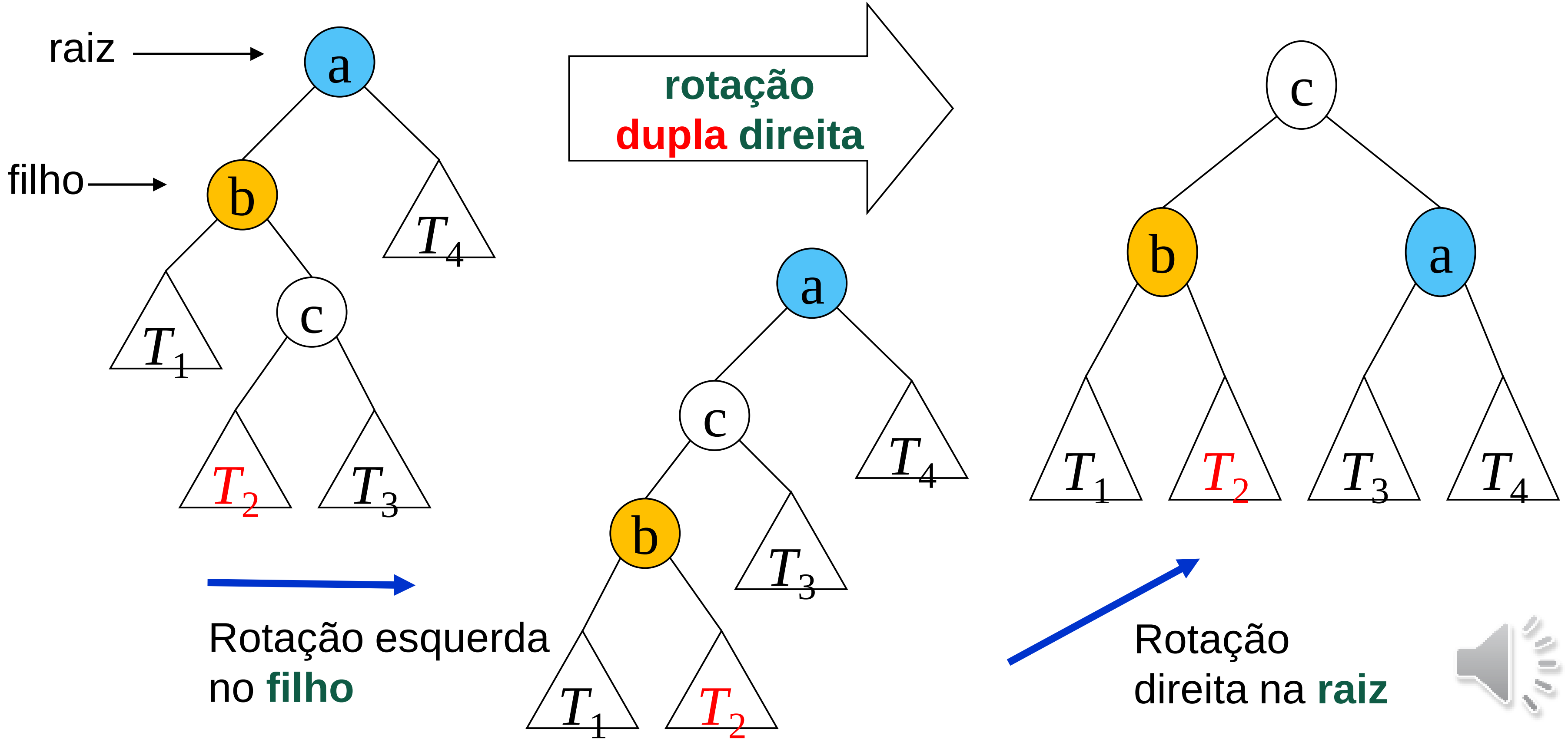


Note que T_2 corresponde ao intervalo de valores (b, a)

7.1.1 Rotações



7.1.1 Rotações



7.1.2 Inserção

- Precisamos **manter em cada nó** um **campo extra** chamado ***alt*** , que vai registrar a **altura da árvore ali enraizada**.

- Uma alternativa (em vez manter a **altura em cada nó**) seria manter apenas **a diferença de altura** entre as subárvores **esquerda e direita.**

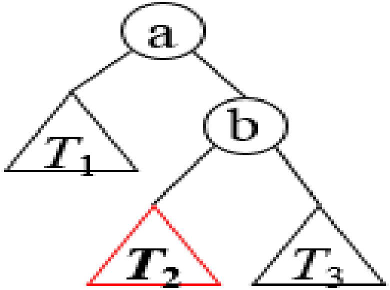
7.1.2 Inserção: rotinas para altura

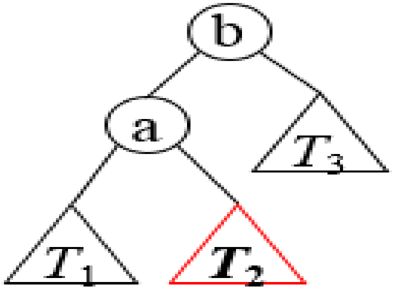
- Para **acessar (saber)** e **atualizar** as **alturas** das árvores:

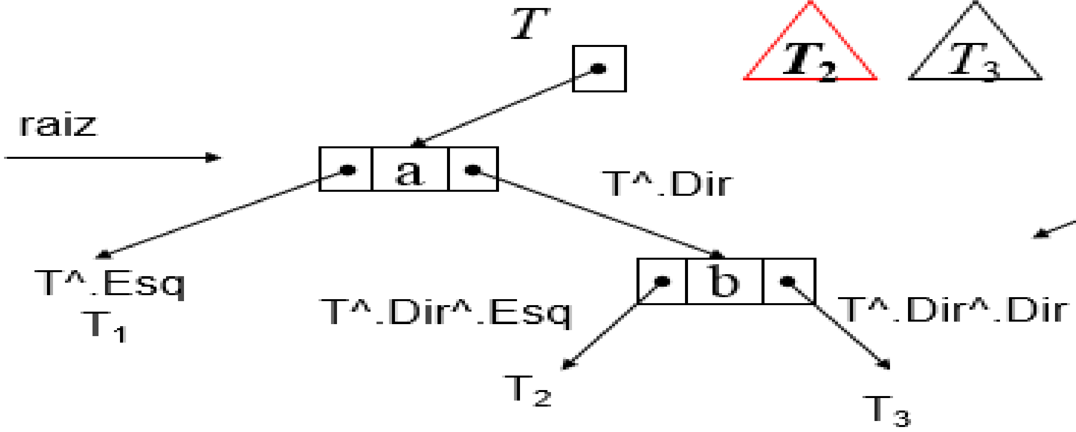
```
proc Altura (Arvore T)
{
      se T = Nulo então retornar 0
       senão retornar T^.Alt
}

proc AtualizaAltura (Arvore T)
{
      se T ≠ Nulo então
          T^.Alt ← max (Altura (T^.Esq), Altura (T^.Dir)) + 1
}
```

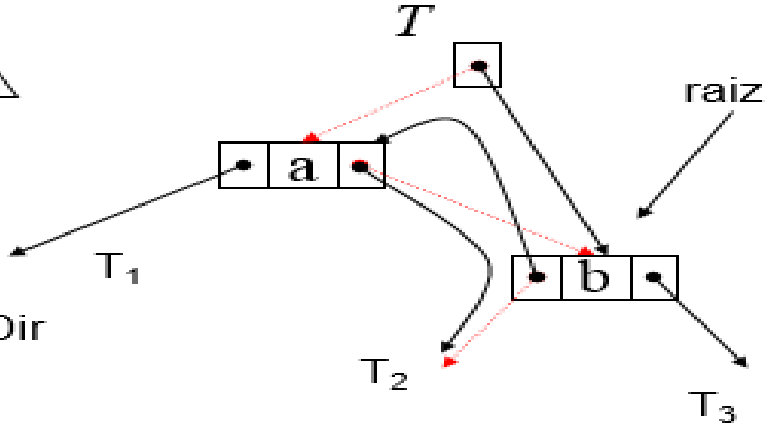
7.1.2 Inserção: rotinas rotação

```
proc RotacaoEsquerda (var Arvore T)
{
    T, T^.Dir, T^.Dir^.Esq ← T^.Dir, T^.Dir^.Esq, T
    AtualizaAltura (T^.Esq)
    AtualizaAltura (T)
}
```









7.1.2 Inserção: rotinas rotação

```
proc RotacaoDireita (var Arvore T)
{
    T, T^.Esq, T^.Esq.Dir ← T^.Esq, T^.Esq^.Dir, T
    AtualizaAltura (T^.Dir)
    AtualizaAltura (T)
}
```

```
proc RotacaoDuplaEsquerda (var Arvore T)
{
    RotacaoDireita (T^.Dir)
    RotacaoEsquerda (T)
}
```

```
proc RotacaoDuplaDireita (var Arvore T)
{
    RotacaoEsquerda (T^.Esq)
    RotacaoDireita (T)
}
```

7.1.2 Inserção: algoritmo

```
proc InserirAVL (Chave x, var Arvore T)
{
    se T = Nulo então {
        T ← Alocar (NoArvore)
        T^.Val, T^.Esq, T^.Dir, T^.Alt ← x, Nulo, Nulo, 1 }
    senão {
            se x < T^.Val então {
                InserirAVL (x, T^.Esq)
                se Altura (T^.Esq) – Altura (T^.Dir) = 2 então
                    se x < T^.Esq^.Val então RotacaoDireita (T)
                    senão RotacaoDuplaDireita (T) }
            senão {
                    InserirAVL (x, T^.Dir)
                    se Altura (T^.Dir) – Altura (T^.Esq) = 2 então
                    se x > T^.Dir^.Val então RotacaoEsquerda (T)
                    senão RotacaoDuplaEsquerda (T)
                    }

            AtualizaAltura (T)
        }
}
```

Obs:
1) a inserção ocorre sempre nas **folhas.**
2) Quando a inserção ocorre nas folhas dos extremos (esq ou dir), ocorre **rotação dupla.**
3) Quando a inserção ocorre nas folhas que não são extremos, ocorre **rotação simples.**

7.1.2 Inserção: complexidade

- Pode-se ver que **apenas uma rotação** (dupla ou simples) no máximo é necessária (**$O(1)$**)

- A **atualização do campo altura** (**$O(1)$**) pode ter de ser feita mais do que uma vez: na verdade, tantas vezes quantos forem os nós no caminho até a folha inserida, i.e., no total **$O(\log n)$**

- No mais, o **algoritmo é idêntico** ao da inserção em ABBs, e portanto a complexidade total é **$O(\log n)$**

7.1.3 Remoção: complexidade

- Segue **as mesmas linhas** do **algoritmo de inserção.**

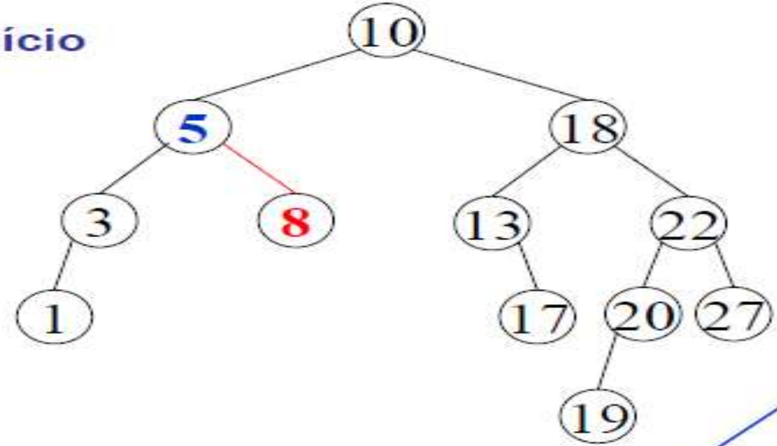
1. Faz-se a **remoção do nó** como uma **árvore de busca comum;**
2. Analisa-se **a informação de balanceamento,** aplicando a **rotação apropriada** se for necessário.

- Mas, **diferentemente da inserção**, pode ser necessário realizar **mais do que uma rotação** (até **$O(\log n)$**) rotações).
- Isso não afeta a **complexidade** do **algoritmo de remoção,** que resulta sendo **$O(\log n)$.**

7.1.3 Remoção: exemplo

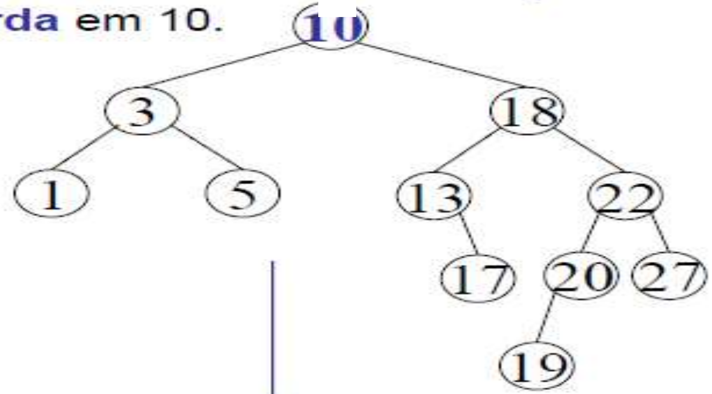
Ex: *RemoverAVL* (8, *T*)

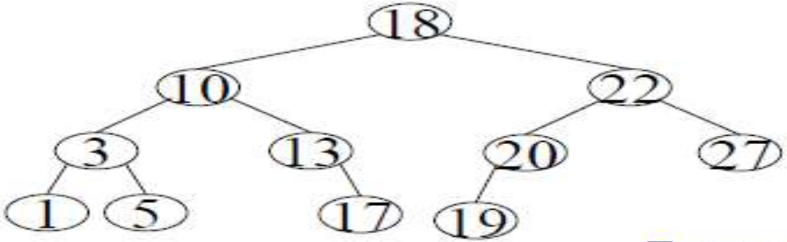
Início



Ao remover-se 8, será preciso aplicar uma **rotação simples à direita** em 5 para balancear a estrutura, pois *h* da subárvore esquerda de 5 é 3 e *h* da subárvore direita de 5 passa a ser 1 → **3 – 1 = 2.**

h da subárvore esq de 10 é 3 e ***h*** da subárvore dir de 10 é 5 → **rotação à esquerda** em 10.





Estrutura balanceada (fim)

7.2 Árvores Rubro-Negras

7.2 Árvores Rubro-Negras

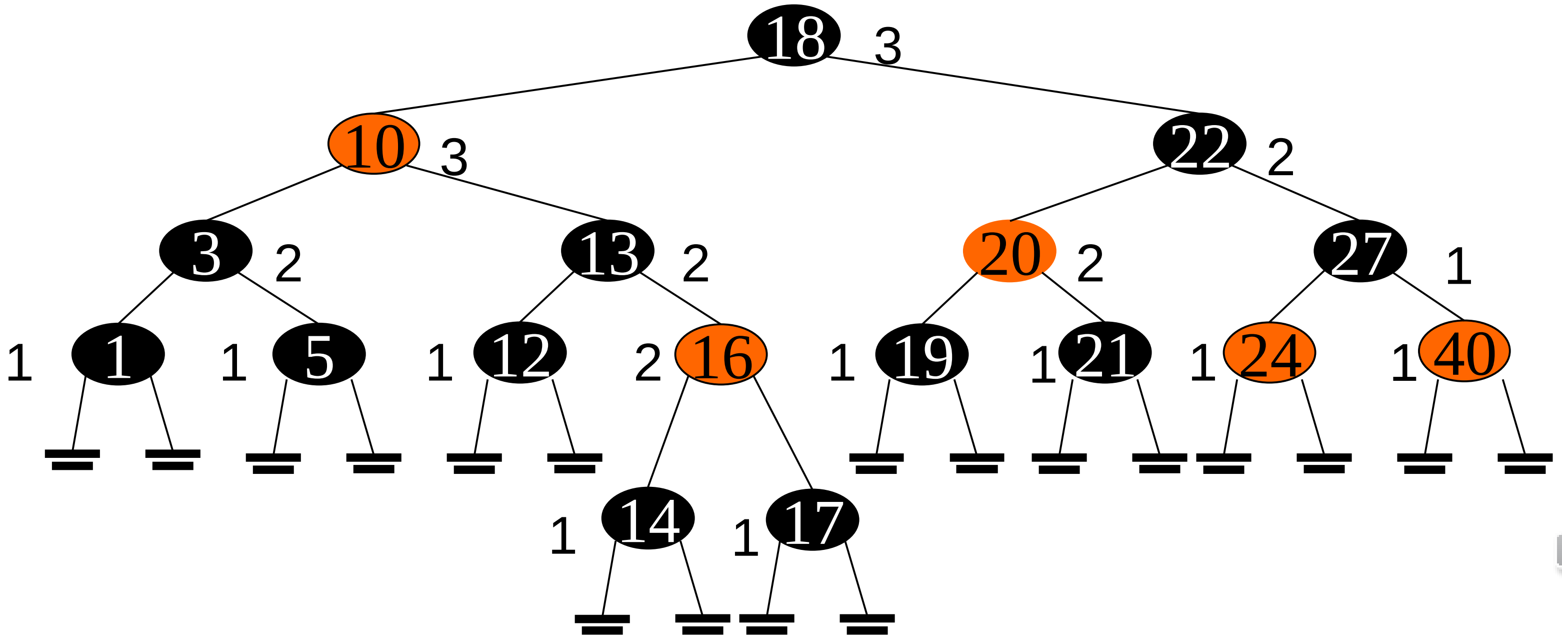
- Também **Vermelho e Preto**. São **ABBs**. São **balanceadas**.
- A todos os nós é associada **uma cor** que pode ser **vermelha** ou **negra** de tal forma que:

1. Nós **externos** (nulos ou subárvores vazias) são **negros**;
2. Todos os caminhos entre **um nó** e qualquer de **seus nós externos descendentes** percorre um número **idêntico** de nós **negros**;
3. Se um nó é **vermelho** (e não é a raiz), seu pai é **negro.**

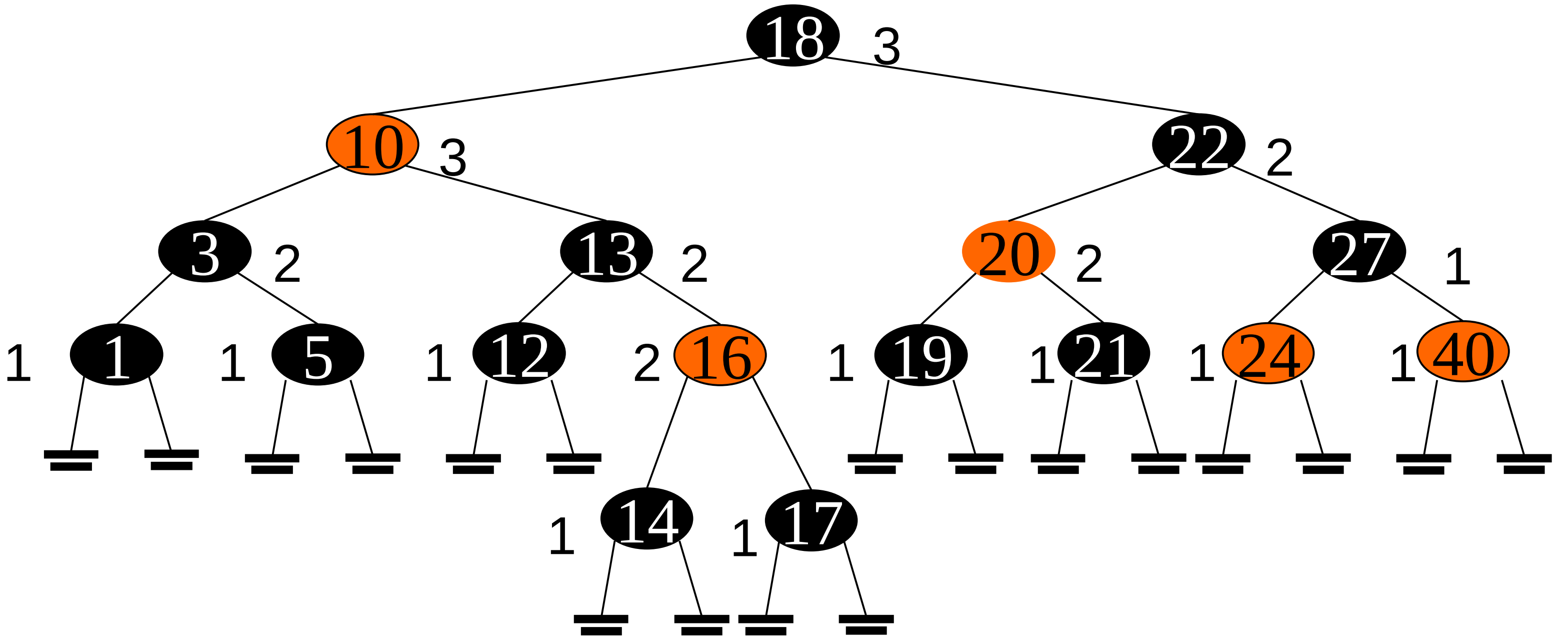
- As **propriedades** (critérios) acima garantem que o **maior caminho** da **raiz** até um **nó externo** é no máximo **duas vezes** maior que o de qualquer **outro caminho** até outro nó externo.

7.2 Árvores Rubro-Negras

Altura negra de um nó = **número de nós negros** encontrados até qualquer **nó externo descendente.** Nesta contagem **exclui-se** o **nó** a partir do qual inicia-se a contagem e **inclui-se** o nó nulo, que é **negro.**



7.2 Árvores Rubro-Negras



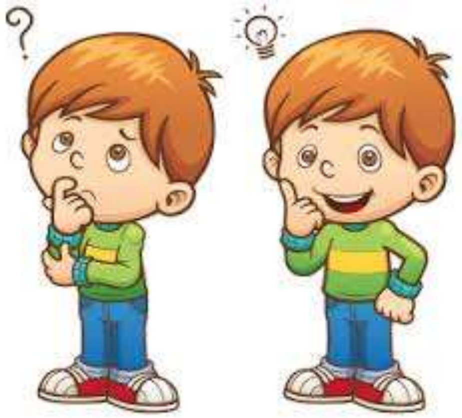
O ESTUDO DETALHADO DESSE TIPO DE ÁRVORE É DEIXADO COMO ATIVIDADE EXTRACLASSE.

FIM 2ª. Parte

8. Árvores de Difusão (Splay Trees)

8.1 Árvores de Difusão: discussão

O **bom funcionamento** de uma estrutura de dados em forma de **árvore binária** depende, essencialmente, da **eficiência** com que o **problema de BUSCA** pode ser resolvido.



8.1 Árvores de Difusão: discussão

- **As ABBs**: garantem inserção/remoção/busca em **tempo logarítmico** no caso médio, mas **não no pior caso.**

- As **ABBs balanceadas** (**AVL/Rubro-Negra**): garantem inserção/remoção/busca em tempo **logarítmico** no **pior caso.**
  Mas precisamos guardar informação adicional em cada nó (**altura da árvore/cor**)

- **Árvores de Difusão** (**Splay Tree**): É uma **ABB autoajustável**. Garante boa performance ***amortizada.***

8.1 Árvores de Difusão: discussão

- A **árvore de difusão não** garante **necessariamente** um **pior caso de O(log *n*)** para **acessar** algum nó desejado.

- **Contudo**, ela assegura **complexidade AMORTIZADA de O(log *n*),** no pior caso.

- Para alcançar esse objetivo, são realizadas **automaticamente operações de ajustes,** após **cada acesso a algum nó procurado**. Essas operações se processam ao longo do caminho da raiz até esse nó.

8.1 Árvores de Difusão: discussão

Custo **amortizado NÃO** é o **mesmo** que **custo médio.**

- **Custo médio** leva em conta **todas as possíveis sequências** de operações e o limite é obtido **na média**.

- **Custo amortizado** é obtido para **qualquer sequência** de operações. O custo de cada execução depende das execuções anteriores.

A chamada visão **do banqueiro** ajuda a entender **intuitivamente** o conceito de complexidade amortizada. Modela-se o algoritmo como um sistema de avaliação de **créditos e débitos**. Mostra-se que os **créditos compensam os débitos.**

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/amortized.html

8.1 Árvores de Difusão: discussão

- A ideia da **árvore de difusão** é que as chaves **mais** **RECENTEMENTE** **acessadas** estejam mais **próximas da raiz.**

- No caso das **ABB ótimas**, as chaves **mais** **FREQUENTEMENTE** **acessadas** é que ficam **próximas da raiz.**

8.1 Árvores de Difusão: discussão

- Uma **precondição natural das árvores de difusão** é que seu processo **de ajuste** seja **mais simples** que aquele executado na **ABB balanceada**.

- **ATENÇÃO**: em vez de ter **n = 1.000.000** de chaves em uma **ABB balanceada** sabendo que gasto **O(log *n*)**, no pior caso, para acessar qualquer chave, mas sabendo também que acesso **99% das vezes** as mesmas **100 chaves**, **é melhor ter** uma **árvore de difusão** de **n = 1.000.000** de chaves.

8.1 Árvores de Difusão: discussão

- Isso porque a **árvore de difusão** vai garantir que aquelas **100 chaves** estarão **bem próximas da raiz**, resultando em um **custo de acesso na prática** baixo (**amortizado** O(log *n*), no pior caso).

- E se depois o conjunto de chaves mais recentemente acessadas mudar, **automaticamente** a árvore **de difusão** se ajusta para continuar a **ter a mesma eficiência**.

8.2 Árvores de Difusão: definição

- São **ABBs** cujo desempenho **amortizado** é **ótimo**.

- O **tempo total amortizado** para realizar **qualquer sequência** de ***m*** operações de **inserção/remoção/busca** em uma **árvore de difusão**, **incialmente vazia**, é ***O*(*m* log *n*),** no pior caso, onde ***n*** é o maior número de nós alcançados pela árvore (i.e., que foram incluídos).

8.2 Árvores de Difusão: definição

- Uma **árvore de difusão** **não** guarda informações sobre o balanceamento das subárvores.

- É uma estrutura **autoajustável**, isto é, cada operação que é executada **rearruma** a estrutura.

- Caso a operação tenha **mau desempenho**, a rearrumação garante que a mesma operação, se repetida, será **executada eficientemente.**

8.2 Árvores de Difusão: definição

- Na árvore **de difusão**, a **rearrumação** é chamada **splaying** que **significa difundir, espalhar, misturar**.

- Todos acessos para **inserir/remover/buscar** uma **chave *x*** em uma **árvore de difusão *T*** são precedidos/sucedidos por uma chamada à função ***splay* (*x*, *T*)**, que é a **difusão completa** de ***x*** (detalhada mais adiante).

8.3 Árvores de Difusão: operações

- As **operações** (rearrumações) se iniciam em um certo nó **q** e se propagam até a raiz da árvore **T**, por meio do caminho de **q** a **r**.

- A **operação** relativa a **q** denomina-se **difusão** de **q** e emprega as **rotações** das **árvores AVL**.

- A **difusão** do nó **q** depende da posição de **q** em relação a seu **pai** e **avô** e implica a realização de até **duas rotações**.

8.3 Árvores de Difusão: operações

- **Os passos** para efetuar a **difusão** do nó **q** numa árvore de raiz **r,** bem como o funcionamento de uma árvore de difusão, são detalhados a seguir:
- **Caso 1: q = r**

  Nada a efetuar.

- **Caso 2: q é filho de v = r**

  Caso 2.1: q é filho esquerdo

  Rotação <u>direita</u> em v

  Caso 2.2: q é filho direito

  Rotação <u>esquerda</u> em v

8.3 Árvores de Difusão: operações

- Caso 3: q é filho de v ≠ r (seja z o pai de v)
  - Caso 3.1: q, v são ambos filhos esquerdos
    - Rotação direita em z e, em seguida, em v.
  - Caso 3.2: q, v são ambos filhos direitos
    - Rotação esquerda em z e, em seguida, em v.
  - Caso 3.3: q é filho direito e v esquerdo
    - Rotação dupla esquerda em z.
  - Caso 3.4: q é filho esquerdo e v direito
    - Rotação dupla direita em z.

8.3 Árvores de Difusão: operações

- A **difusão de q** deve ser repetida **enquanto** **q** for diferente da raiz **r**, sendo assim **sintetizada** (**difusão completa**):

Enquanto **q ≠ r** efetuar **difusão** do nó **q**

- As **árvores de difusão** empregam essas operações em seu funcionamento da seguinte maneira:

i) após **acessar nó q** → efetuar **difusão completa** de **q**.
ii) após **incluir** um novo nó **q** → efetuar **difusão completa** de **q.**
iii) após **excluir** o nó **q** → efetuar **difusão completa** do nó que era o pai de **q.**

FIM 3ª. Parte

9. Árvores B

9.1 Árvores B: discussão

- As **ABBs** são estruturas de dados **muito eficientes** quando se deseja trabalhar com **tabelas** que caibam **inteiramente na memória principal**.

- Considere, porém, o problema de **recuperar informação de grandes arquivos de dados** que estejam armazenados em **memória secundária,** e.g., no disco magnético.

9.1 Árvores B: discussão

- Uma **forma simplista** de resolver esse problema utilizando **ABBs** é armazenar os **nós da árvore no disco**, e as **referências (ponteiros)** à esquerda e à direita de cada nó se tornam **endereços de disco**, em vez de endereços de memória principal.

- Se a **ABB for balanceada**, a pesquisa por **um registro** necessitará de **O(log *n*) acessos** ao disco. Isso significa que, e.g., um arquivo com **n = 10e+06** registros necessitará aproximadamente de **20 acessos** (buscas) **ao disco,** o que pode não ser **tolerável.**

9.1 Árvores B: discussão

- Para **diminuir o número de acessos** ao disco, tornando a busca **mais eficiente**, pode-se pensar da seguinte forma:

"***Agrupar*** *os* ***nós da árvore em páginas*** *e, a cada acesso, trazer para memória principal não mais* ***um único registro****, mas sim* ***uma página*** *contendo um* ***conjunto de registros****.*"

- **Tecnicamente**: **diminuímos** o nº de ponteiros e a altura da árvore, pois agrupamos os nós em páginas.

9.1 Árvores B: discussão

- Uma forma **equivalente** de ter esta **mesma discussão** é **questionarmos** sobre o tipo **mais eficiente** de **arquivo de índices** para **recuperar informação de grandes arquivos de dados** armazenados em **memória secundária.**

9.1 Árvores B: discussão

- Arquivo de índices **PLANO.**

Arquivo de Índice

	CHAVE	PONTEIRO
0	3	4
1	5	2
2	10	1
3	15	3
4	16	5
5	21	6
6	23	0

Arquivo de Dados

	COD	NOME
0	23	JOSE
1	10	MARIO
2	5	ANA
3	15	MARCIA
4	3	JULIO
5	16	BEATRIZ
6	21	CAMILA

9.1 Árvores B: discussão

Se percorrermos o **arquivo de índices** **sequencialmente** para encontrar uma chave, o **índice** não será de **muita utilidade.**

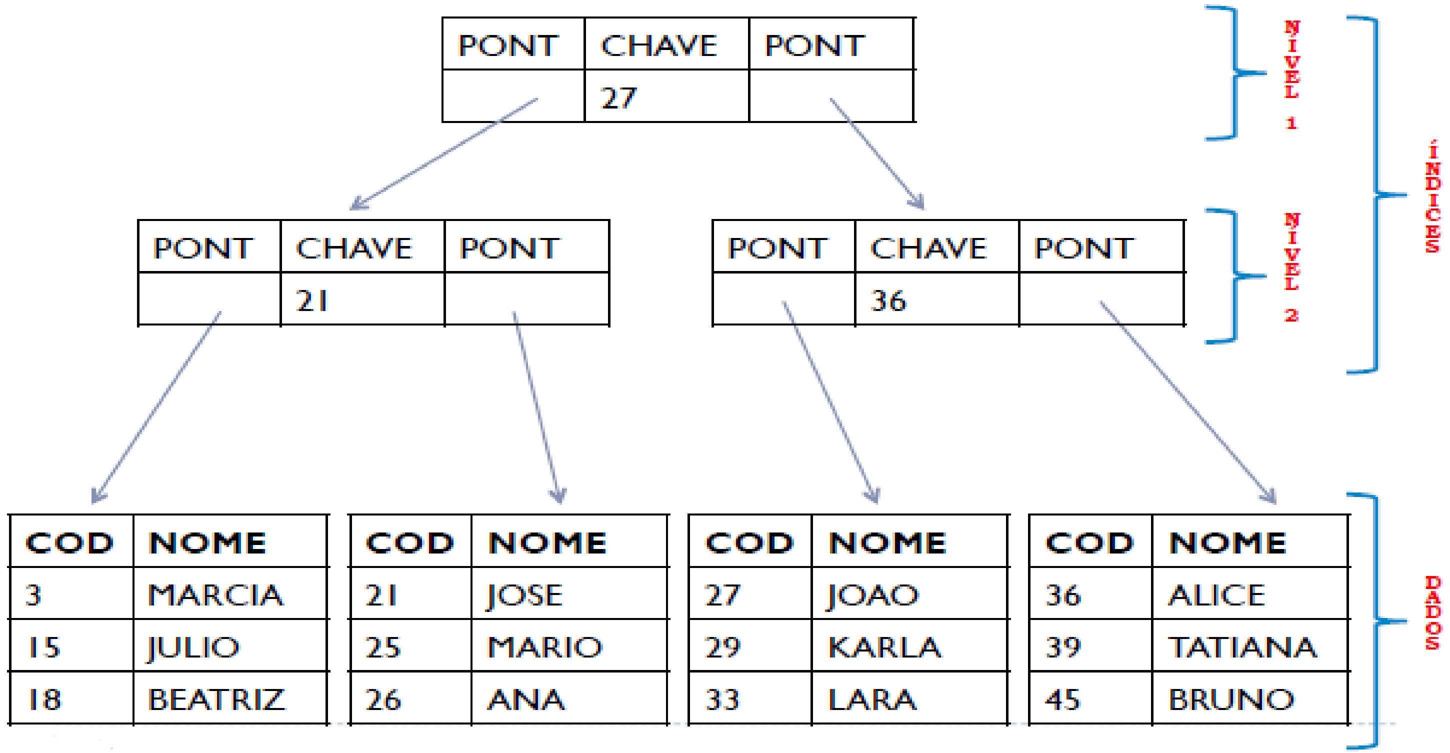
- Pode-se fazer busca **um pouco mais eficiente** (e.g., **busca binária**), se o arquivo de índices estiver **ordenado**. Mas, mesmo assim, isso não seria o **ideal.**

Daí, alternativamente:

- Os **arquivos de índices** **NÃO** seriam **estruturas sequenciais**, e sim **hierárquicas**. Os **índices** **NÃO** apontam para um registro específico, mas para uma **página de registros** (e dentro da página é feita **busca sequencial**) – exige-se que os registros dentro de uma página estejam ordenados.

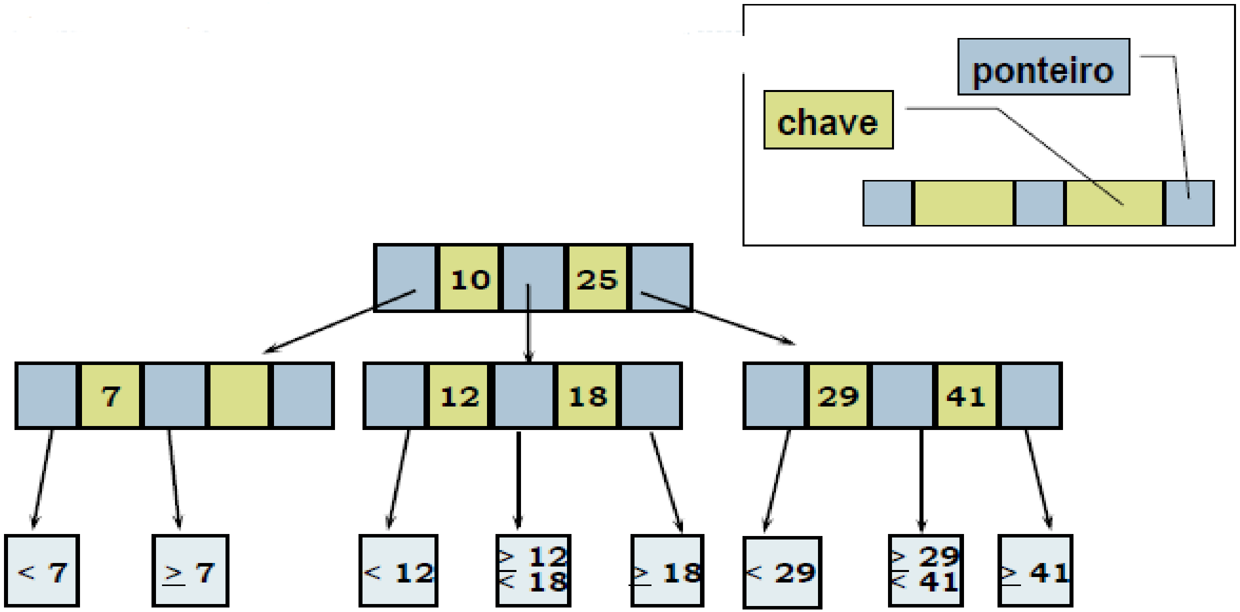
9.1 Árvores B: discussão

- Arquivo de índices HIERÁRQUICO ou MULTINÍVEL



9.1 Árvores B: discussão

A **maioria** das estruturas de **arquivos de índice** é implementada por **árvores de busca**. Note que as chaves são ordenadas e funcionam como **separadores** para os **ponteiros** para os filhos.



9.2 Árvores B: definição

Ante a **discussão anterior**, temos então:

- Devemos empregar **árvores de busca n-árias** **balanceadas**.

- **Cada nó** (página) deve ser mantido **cheio**. Isso para não desperdiçar operações de leitura (acessos ao disco).

- O **tempo de busca** a uma chave deve ser **logarítmico.**

- Uma estrutura de dados que tem essas qualidades é a ***B-tree*** (**Árvore B**), que nada mais é do que uma **árvore de busca *n*-ária** à qual se impõe **critérios adicionais**.

9.2 Árvores B: definição

Cada **nó**, exceto o **nó raiz**, tem entre $\lfloor n/2 \rfloor$ e ***n* chaves.**

- O **nó raiz** tem entre **1** e ***n*** chaves (ou nenhuma, se a árvore estiver vazia).
- As **folhas** estão sempre no mesmo nível.
- A **ordem m** da árvore é assim calculada: $m = n/2$.

Cada **nó *A*** é da forma $P_0C_1P_1C_2P_2 \ldots C_kP_k$, onde P_i é:

- Um **ponteiro** para **outro nó** da árvore contendo **apenas chaves** cujos valores são **maiores ou iguais** a C_i e menores que C_{i+1} (se o nó ***A*** é **interior**); ou
- Um ponteiro nulo (se o nó ***A*** é **folha).**

9.2 Árvores B: definição

Bayer e McCreight, Comer, dentre outros, definem a **ordem m** como sendo o **número mínimo de chaves** que uma página pode conter, ou seja, com **exceção da raiz,** todas podem conter **esse número mínimo de chaves.**

Essa definição pode causar **ambiguidades** quando se quer armazenar um **número máximo n ímpar** de **chaves**. Ex: se quisermos **no máximo n = 7 chaves** em cada página, qual será a **ordem da árvore? m = n/2 = 3,5?**

Já **Knuth** propôs que a **ordem** fosse o **número máximo de páginas filhas** que toda página pode conter. Dessa forma, o **número máximo de chaves por página** ficou estabelecido como a **ordem menos um.**

Por **simplicidade**, vamos aqui sempre **ADMITIR**: *m* = *n*/2, **onde *n* é o número máximo de chaves em cada página, e *n* é sempre par.**

9.3 Árvores B: algoritmo de busca

Seja X o valor buscado:

1. Nó raiz é lido
2. Faz-se uma busca <u>binária</u> (*array*) nas ***k*** chaves do nó.

    *2.1. **<u>Se</u>** **X** for encontrado, o algoritmo termina retornando <u>Verdadeiro</u>*

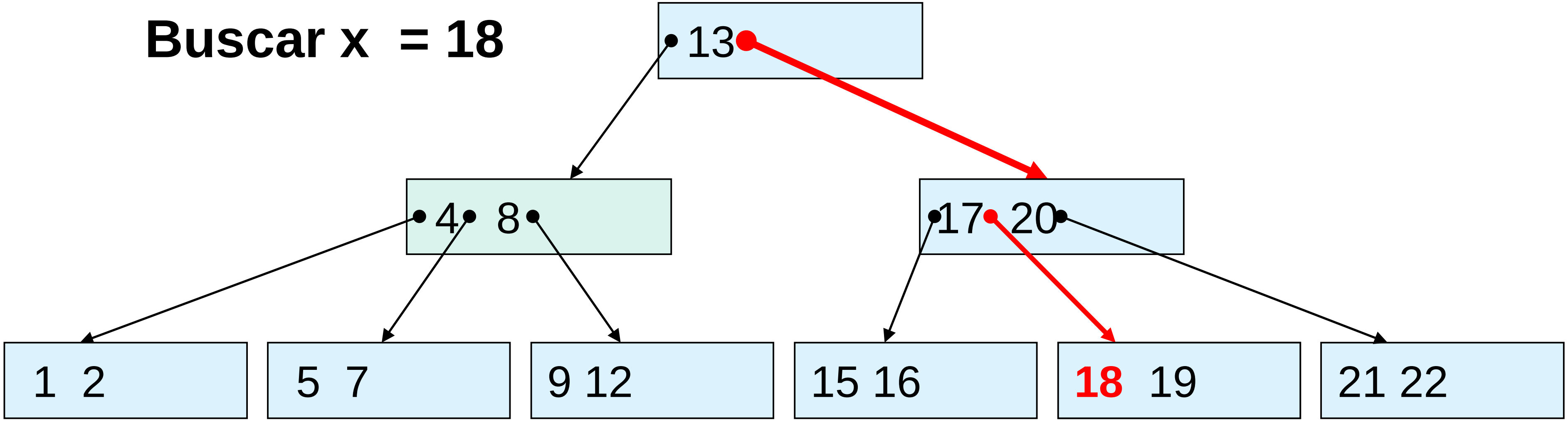
    *2.2. **<u>Senão</u>**, **X** corresponde ao intervalo de algum P_i .*

    *2.2.1 **<u>Se</u>** P_i é <u>nulo</u>, o algoritmo termina retornando <u>Falso</u>*

    *2.2.2 **<u>Senão</u>**, o nó apontado por P_i é lido e retorna-se ao <u>passo 2</u>*

9.3 Árvores B: algoritmo de busca

Buscar x = 18



9.4 Árvores B: algoritmo de inserção

1. **Buscar** o nó folha onde o valor **X** deve ser inserido.
2. **Se** o nó ainda tem lugar, **insere** **X** e algoritmo **termina.**
3. **Senão**, dividir as **n+1 chaves** em **três** grupos: **% Overflow**

    [1] as $\lfloor n/2 \rfloor$ menores chaves,

    [2] a chave mediana; e

    [3] as $\lceil n/2 \rceil$ maiores chaves.

    Grupo [1] permanece no nó folha

    Grupo [3] vai formar um **novo** nó folha

    Grupo [2] (chave mediana), é inserida (recursivamente se necessário) no nó pai. **Se** não existe nó pai, uma nova raiz é criada.

Lembrete: Mediana

Mediana: **valor central** do conjunto que divide a sequência de valores em duas partes iguais (mesmo número de valores abaixo e acima do valor central).

- Os valores devem estar **ordenados.**
- Depois de ordenados os valores, por ordem crescente ou decrescente, a **mediana** é:

❑ O valor que ocupa a **posição central**, se a quantidade desses valores for **ímpar;**

❑ A **média dos dois valores centrais**, se a quantidade desses valores for **par.**

9.4 Árvores B: algoritmo de inserção

Exemplo: ***n*** **= 4; Ordem de Inserção:**

8 5 15 12 7 16 13 9 1 2 3 17 19 18 21 22 20

5 8 12 15

5 7 8 12 15

Overflow: não há espaço para inserir 7 (limite é *n* = 4)

8

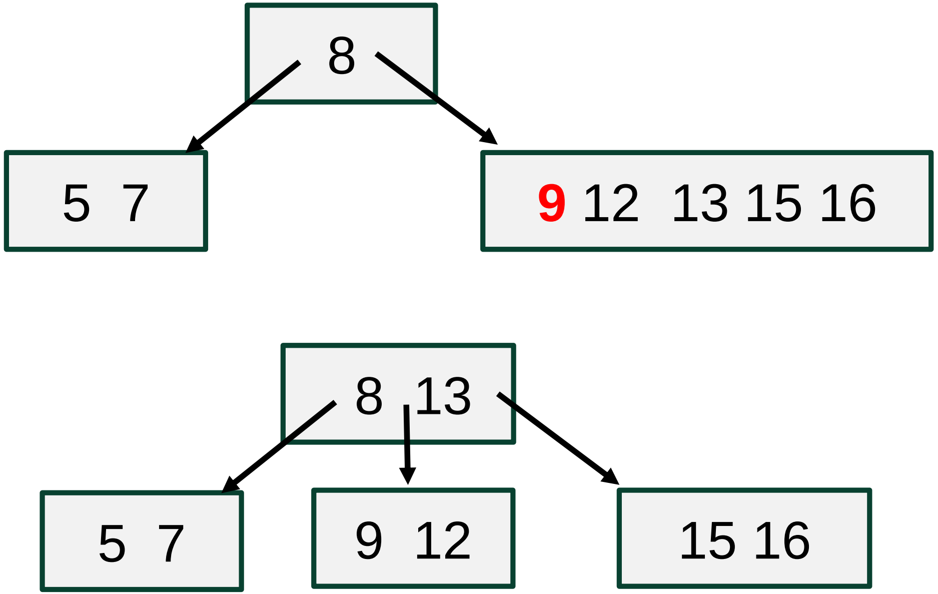
5 7

12 15

9.4 Árvores B: algoritmo de inserção

Exemplo: ***n*** **= 4; Ordem de Inserção**:

8 5 15 12 7 16 13 **9** 1 2 3 17 19 18 21 22 20

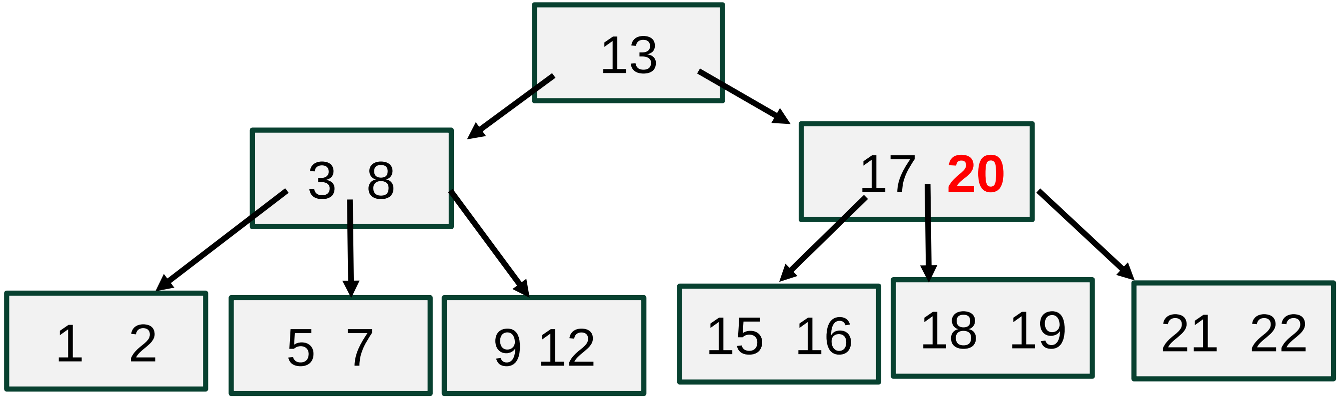


Overflow

9.4 Árvores B: algoritmo de inserção

Exemplo*:* *n* = 4; Ordem de Inserção:

8 5 15 12 7 16 13 9 1 2 3 17 19 18 21 22 **20**



***Estrutura final.**

9.5 Árvores B: algoritmo de remoção

- O algoritmo de **exclusão (remoção)** de uma chave é **um pouco mais complexo** do que o de **inserção.**

- Há **vários casos** a serem considerados, dado que:
   - Nó que contém a chave é **folha** ou **interior**?
   - Há risco de **underflow** (nó fica com chaves de menos) **ou não**?

9.5 Árvores B: remoção no NÓ FOLHA

1. **Se a chave está em um NÓ FOLHA**

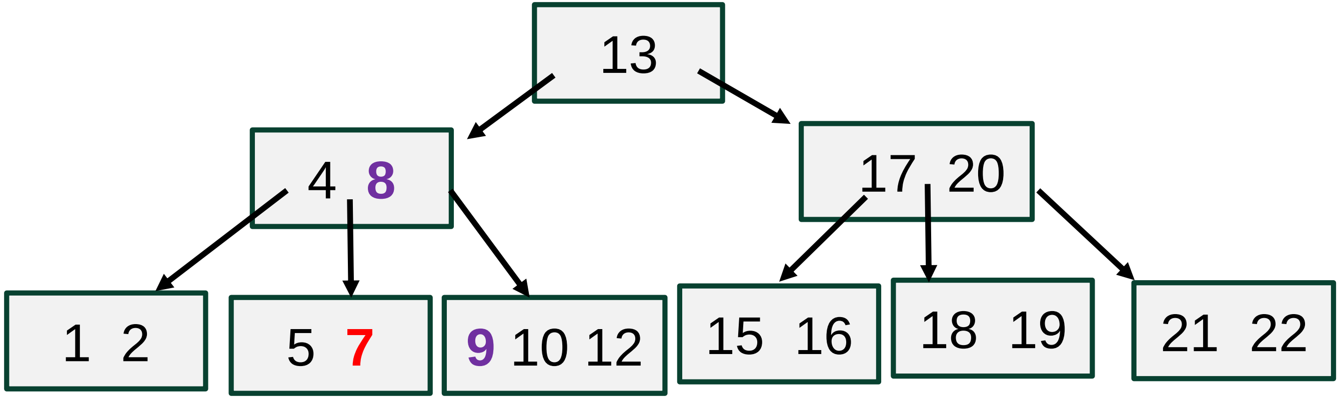
- **Se** o nó **não** corre risco de **underflow**, a chave é **removida** e o algoritmo **termina**
- **Senão**, **duas** situações são possíveis:

a) O nó **vizinho** à esquerda ou à direita têm uma chave para **emprestar**, isto é, não corre o risco de **underflow**. Neste caso fazer operação de **empréstimo.**

b) Ambos os **vizinhos** têm o número **mínimo** de chaves. Neste caso fazer a operação de **combinação** do nó com **um** de seus vizinhos.

9.5 Árvores B: remoção no NÓ FOLHA

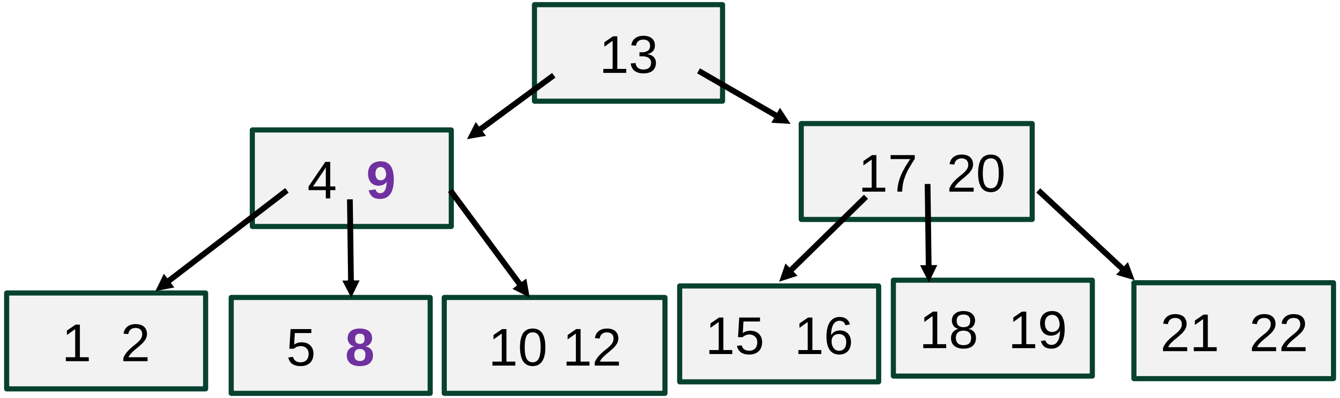
Exemplo 1: Remoção da chave 7, n = 4 (mínimo **n/2 = 2**)



1. a) O nó **vizinho** à esquerda ou à direita têm uma chave para emprestar, isto é, não corre o risco de **underflow**. Neste caso fazer operação de **empréstimo.**

9.5 Árvores B: remoção no NÓ FOLHA

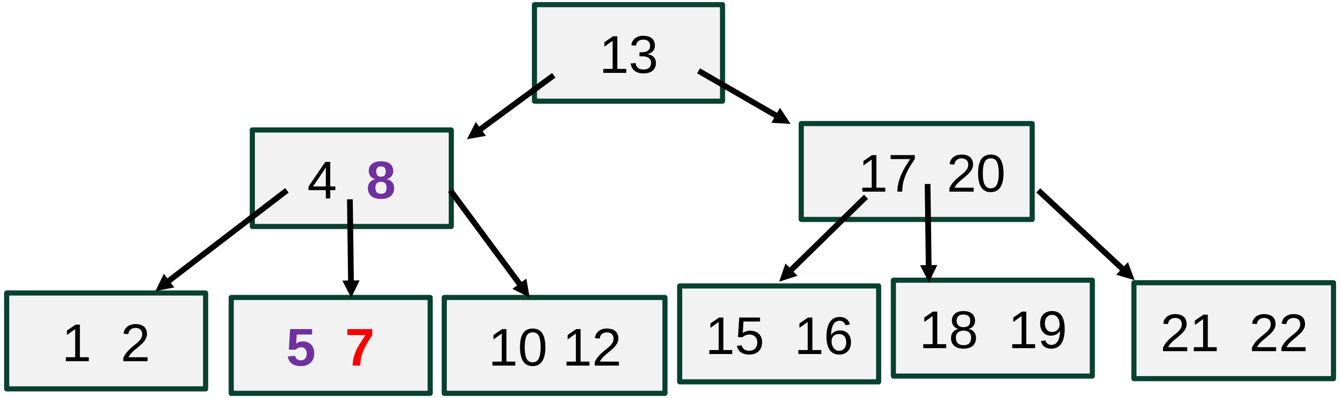
Exemplo 1: Remoção da chave 7, n = 4 (mínimo **n/2 = 2**)



1. a) O nó **vizinho** à esquerda ou à direita têm uma chave para emprestar, isto é, não corre o risco de **underflow**. Neste caso fazer operação de **empréstimo.** (Mas precisarei **descer** 8 e **subir** 9)

9.5 Árvores B: remoção no NÓ FOLHA

Exemplo 2: Remoção da chave 7, n = 4 (mínimo **n/2 = 2**)



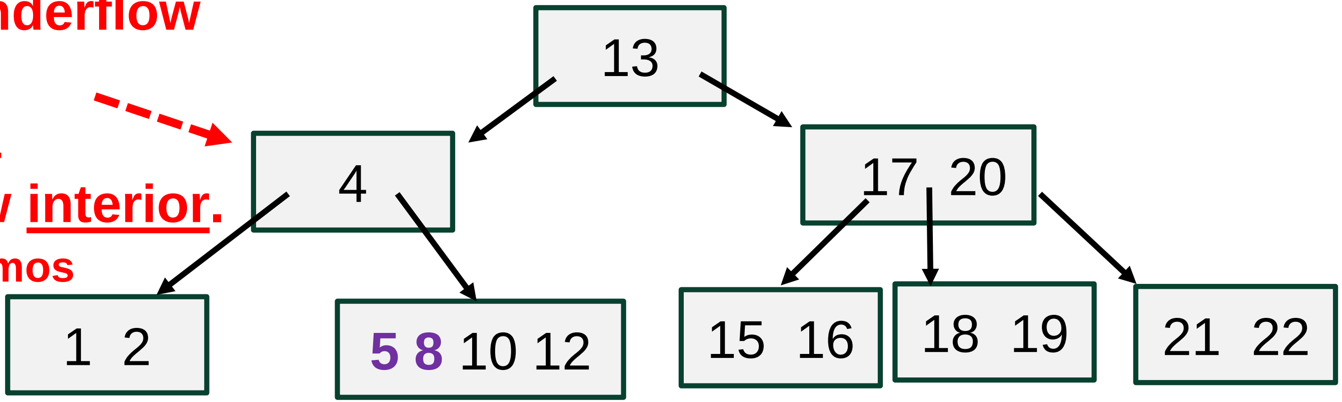
1. b) Ambos os **vizinhos** têm o número **mínimo de chaves**. Neste caso fazer a operação de **<u>combinação</u>** do nó com um de seus vizinhos. (**Ex.**: **combinar** com o **nó [10 12]**, mas precisarei **descer** a chave **8**)

9.5 Árvores B: remoção no NÓ FOLHA

Exemplo 2: Remoção da chave 7, n = 4 (mínimo **n/2 = 2**)

Ocorre underflow de novo.
Desta vez underflow <u>interior</u>.

*** Vide próximos Slides ***



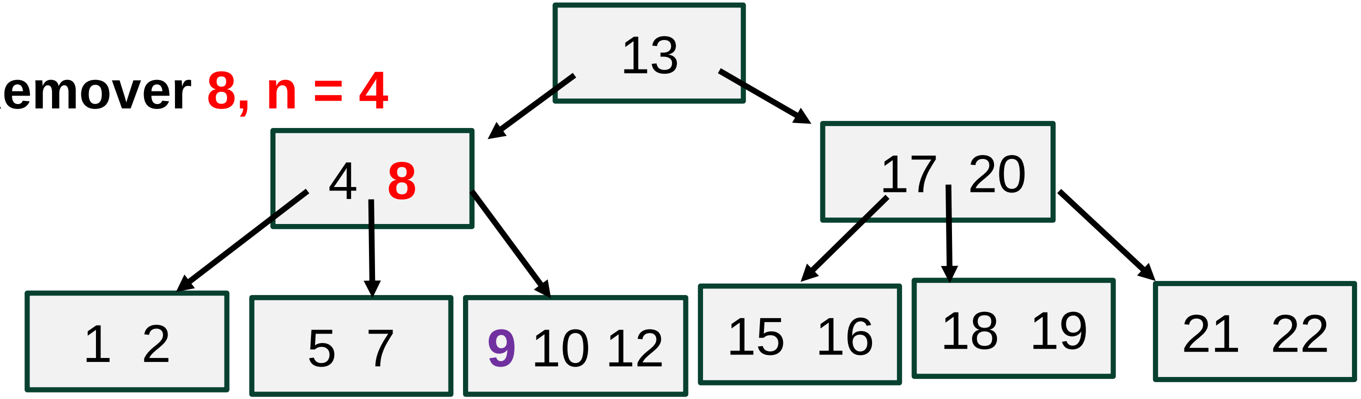
1. b) Ambos os **vizinhos** têm o número **mínimo** de chaves. Neste caso fazer a operação de **<u>combinação</u>** do nó com um de seus vizinhos.

9.5 Árvores B: remoção no NÓ INTERIOR

2. Se a chave está em um NÓ INTERIOR

a) Se é possível fazer uma operação de **empréstimo** entre os **nós filhos** à esquerda e à direita, então fazê-la e **remover** a chave do filho que a recebeu.

Ex.: Remover 8, n = 4

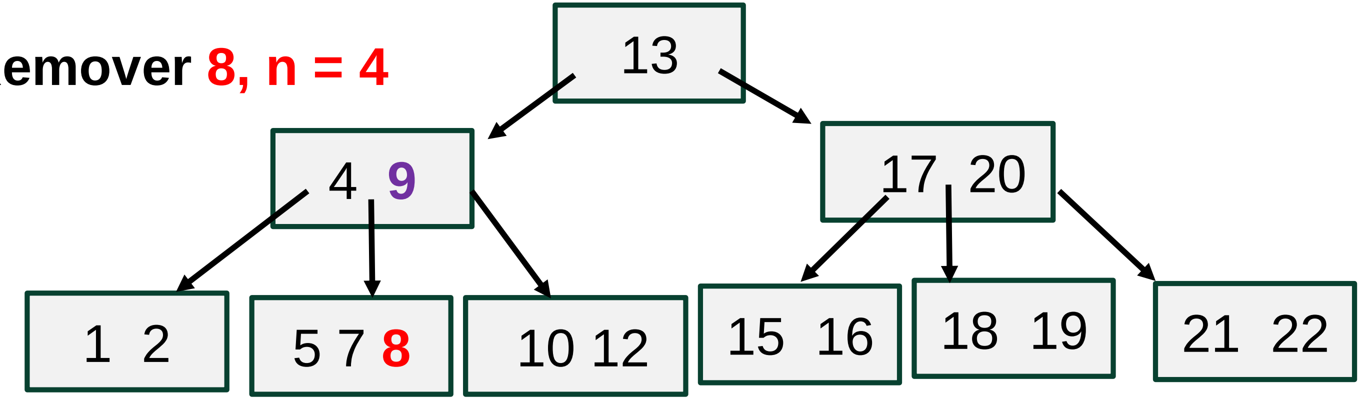


9.5 Árvores B: remoção no NÓ INTERIOR

2. Se a chave está em um NÓ INTERIOR

a) Se é possível fazer uma operação de **empréstimo** entre os **nós filhos** à esquerda e à direita, então fazê-la e **remover** a chave do filho que a recebeu.

Ex.: Remover 8, n = 4

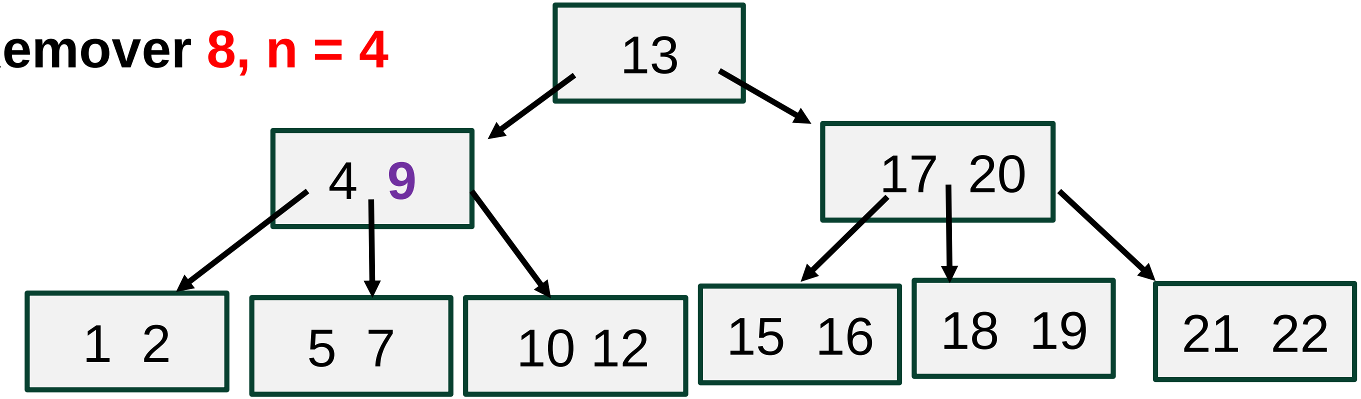


9.5 Árvores B: remoção no NÓ INTERIOR

2. Se a chave está em um NÓ INTERIOR

a) Se é possível fazer uma operação de empréstimo entre os nós filhos à esquerda e à direita, então fazê-la e remover a chave do filho que a recebeu.

Ex.: Remover 8, n = 4



9.5 Árvores B: remoção no NÓ INTERIOR

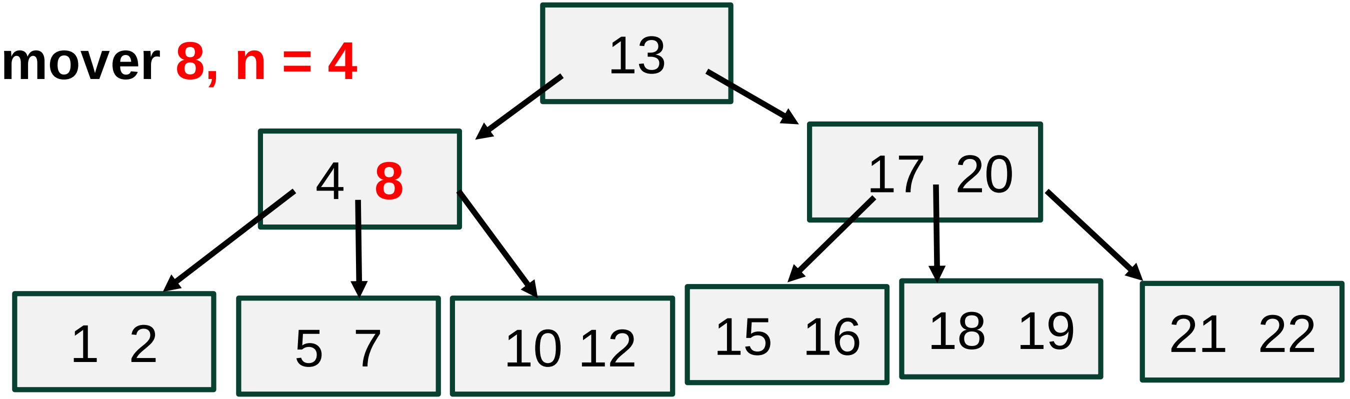
2. Se a chave está em um NÓ INTERIOR

b) Se **NÃO** **é possível** fazer uma operação de **empréstimo** entre os **nós filhos** à esquerda e à direita:

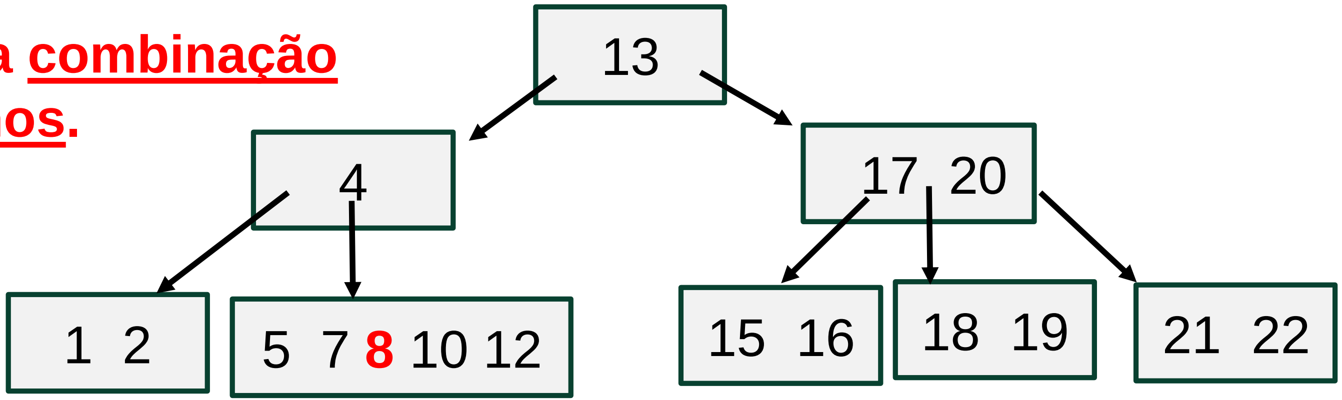
- **Combine-os** (junto com a chave) e **remova** a chave do nó filho resultante
- **Se o nó interior resultante** estiver com **underflow,** aplicar um **empréstimo** com um **nó vizinho** se possível, ou então uma **combinação**. (Isso pode **propagar-se** até a raiz).

9.5 Árvores B: remoção no NÓ INTERIOR

Ex.: Remover 8, n = 4

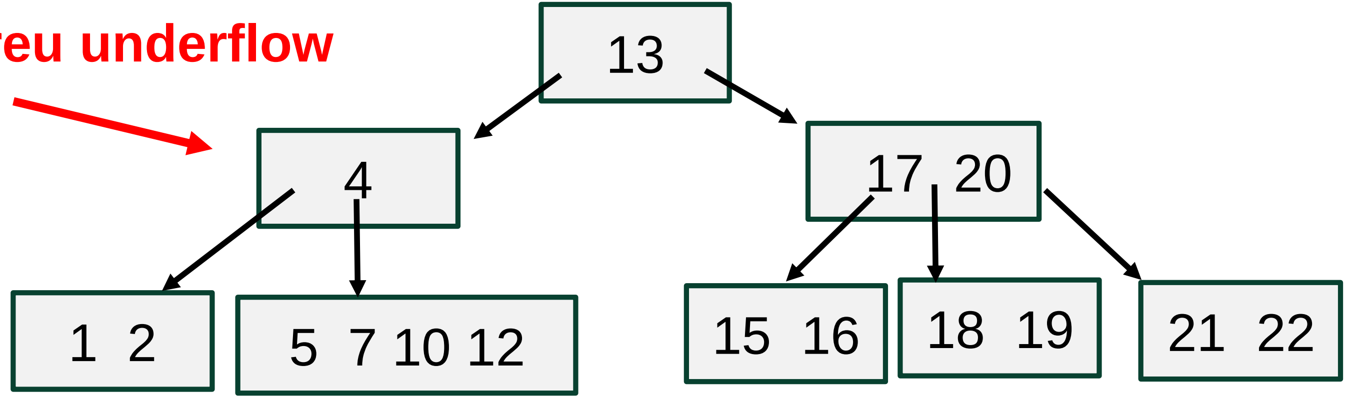


Fez uma combinação com filhos.



9.5 Árvores B: remoção no NÓ INTERIOR

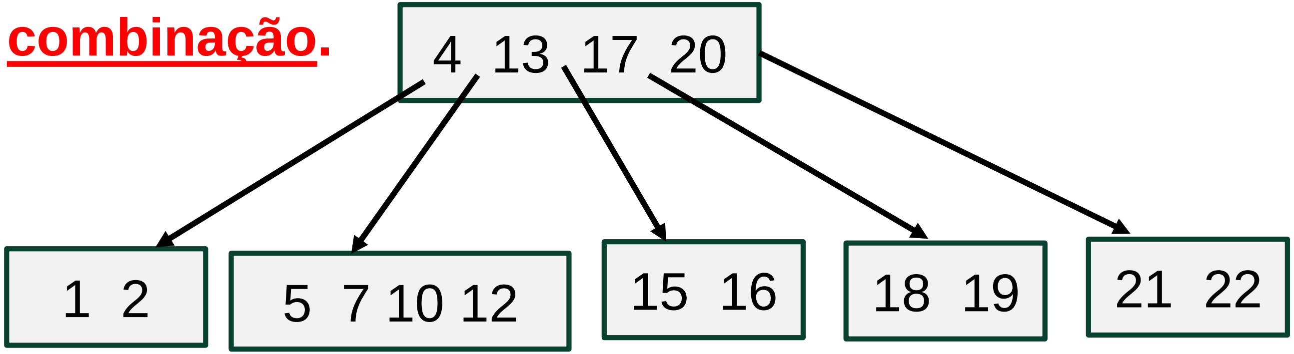
Mas ocorreu underflow **interior**.



- **Se o nó interior resultante** estiver com **underflow**, aplicar um **empréstimo** com um **nó vizinho** se possível, ou então uma **combinação**. (**Neste exemplo** precisarei de uma **combinação**).

9.5 Árvores B: remoção no NÓ INTERIOR

Fez uma combinação.



- **Se o nó interior resultante** estiver com **underflow**, aplicar um **empréstimo** com **um nó vizinho** se possível, ou então uma **combinação**. (Isso pode **propagar-se** até a raiz).

9.6 Árvores B: peculiaridades

- A **árvore B** é **simples**, de **fácil manutenção**, **eficiente** e **versátil**.

- O **custo** para recuperar (buscar), inserir e retirar (remover) registros do arquivo é **logarítmico**. Essas operações sempre deixam a árvore **balanceada**.

- O **espaço utilizado** pelos dados é, no mínimo, **50%** do espaço **reservado** para o arquivo (i.e., poucos ponteiros).

- A **altura esperada** de uma **árvore B** **não** é conhecida **analiticamente.**

9.6 Árvores B: peculiaridades

- O **caminho mais longo** em uma árvore **B** de ordem **m** com **N** registros contém **no máximo cerca** de:

$$\boxed{\text{Nº de páginas} = \log_{2m+1} N}$$

- Foi provado que os limites para as **alturas (*h*) máxima e mínima** de uma árvore **B,** de ordem **m** contendo **N** registros, são:

$$\boxed{\log_{2m+1}(N+1) \leq h \leq \log_{m+1}\left(\frac{N+1}{2}\right)}$$

- Consegue armazenar **índice e dados** na **mesma estrutura** (mesmo arquivo físico).

9.7 Árvores B: variantes

9.7 Árvores B: variantes

- Existem **diversas variantes** para implementação da **árvore B** original. Dentre elas, citamos: **Árvores B*** e **Árvores B+.**

- O “**Asterisco ***” é usualmente lido como “**Estrela**”.

- Menciona-se que **há** obras na literatura em que **B+** é chamado de **B*** e **vice-versa**.

9.8 Árvores B: variante B*

- Todos os nós, **exceto a raiz**, precisam estar **2/3 cheios**, em contraste com **1/2** exigido pela árvore **B original.**
- **De forma geral**, como aspecto **diferenciador e peculiar**, os nós **não são divididos** logo que ocorre **overflow** em uma inserção.
  - **Em vez disso**, ocorre a **redistribuição** de chaves se houver espaço no nó vizinho (irmão), até que ocorra **overflow** em ambos. Neste momento, os **dois** nós são então divididos em **três** nós.
- Na prática, **NÃO é muito utilizada**.

9.9 Árvores B: variante B+

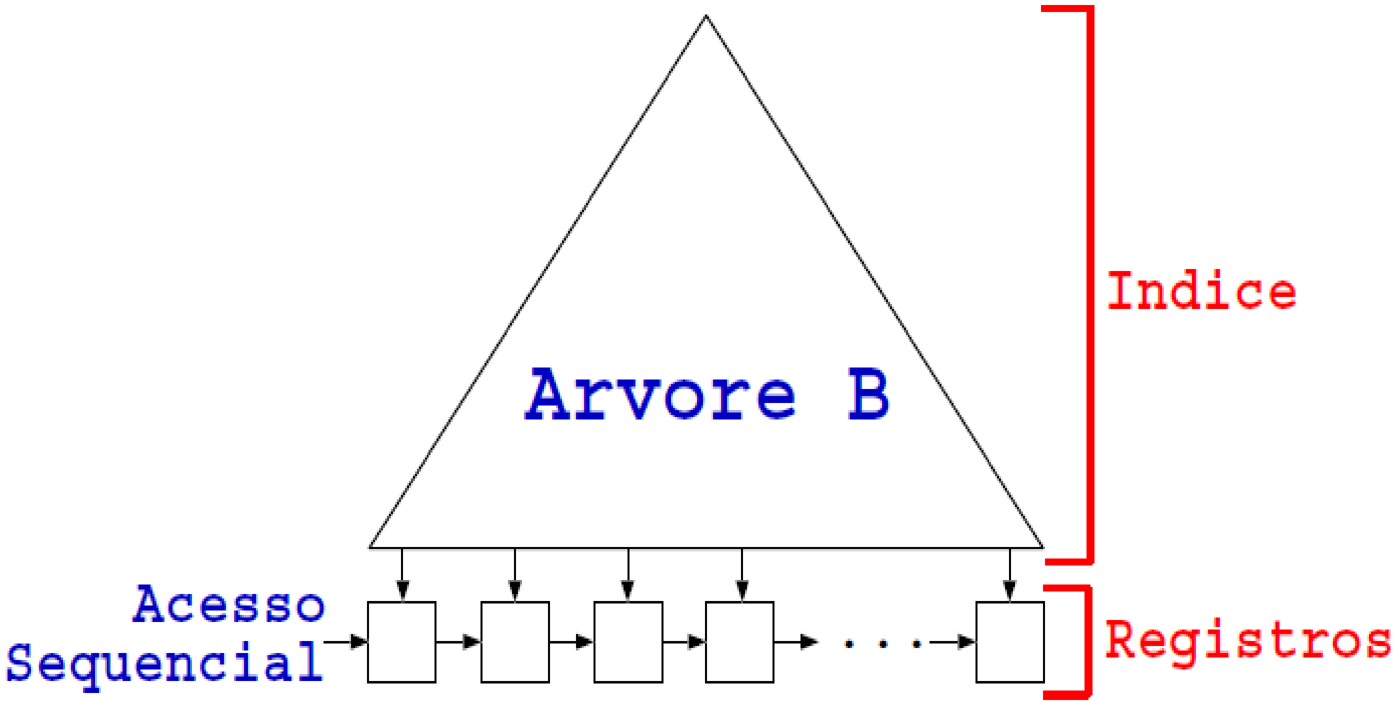
- Os **dados** (registros) são armazenados nas **folhas.**
- Os níveis **acima do último nível** (folhas) constituem um **índice**, cuja organização é a organização de uma **árvore B**.
- No **índice só** aparecem **chaves** (i.e., **registros nas folhas**).
- As **folhas** são estruturadas como **uma lista ENCADEADA** da esquerda para direita.
- Permite-se o armazenamento dos **dados em um arquivo**, e do **índice em outro arquivo separado**.

9.9 Árvores B: variante B+

- Para **procurar (buscar)** um registro, o processo de pesquisa inicia-se na **raiz** da árvore e continua até uma **página folha**.

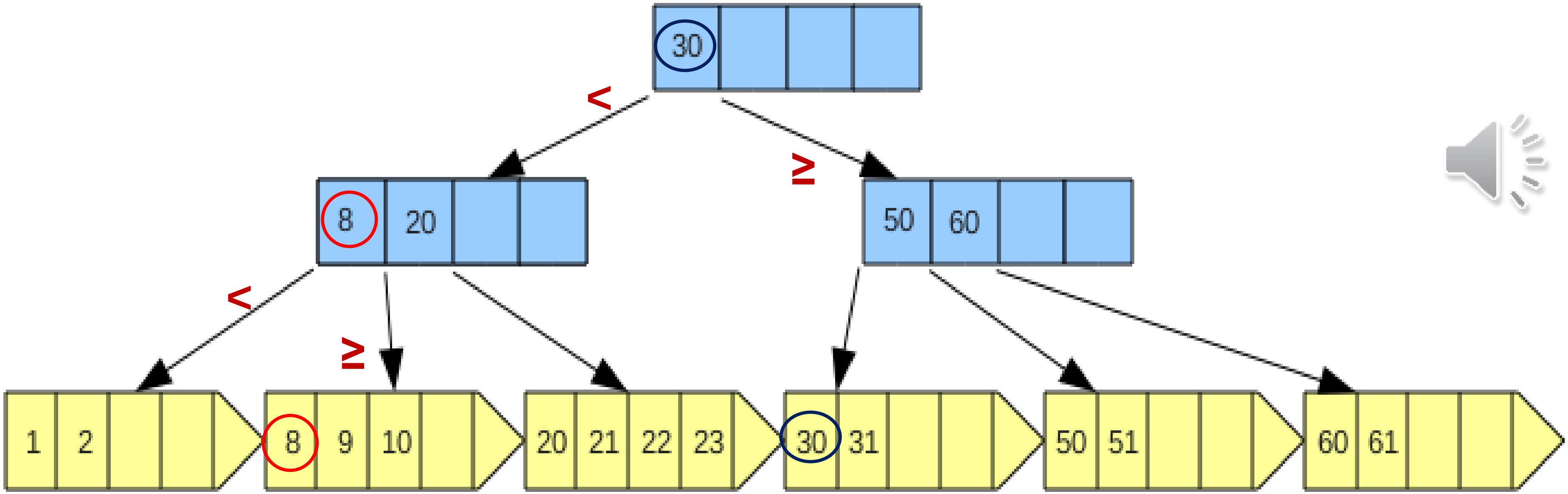
- Como **todos os registros** residem nas **folhas**, a pesquisa **não para** se a **chave** (do registro procurado) **for encontrada em uma página do índice.** Nesse caso, a referência à subárvore direita é seguida até que uma **página folha seja encontrada.**

9.9 Árvores B: variante B+



9.9 Árvores B: variante B+

EXEMPLO: **Índice em azul**: estrutura de árvore B; **navegação**: **<** à esquerda, **≥** à direita; **Registros em amarelo**, nas folhas; **navegação**: da esquerda para direita, para acesso sequencial aos registros.



9.10 Árvores B: Síntese

- Deixa-se **maior detalhamento** como atividade **extraclasse.**
- Árvores **B** e **B+** são **muito importantes** por sua **eficiência.**
- **Exemplos** de aplicações:

Sistemas de arquivos: NTFS, ReiserFS, NSS, XFS, JFS, ReFS, HFS+ e HFS (Apple), AIX, sistemas Linux como brtfs e Ext4, etc;

Bancos de Dados Relacionais: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, SQLite, etc;

Sistemas de gerência de dados como CouchDB, Tokyo Cabinet e Tokyo Tyrant.

https://en.wikipedia.org/wiki/B-tree

FIM 4ª. Parte

10. Árvores Digitais

- Também chamadas de **Tries.**

- **Edward Fredkin**, o inventor, usa o termo **trie** (do inglês, "re**trie**val") (recuperação), porque essa estrutura é basicamente usada na **recuperação de dados**.

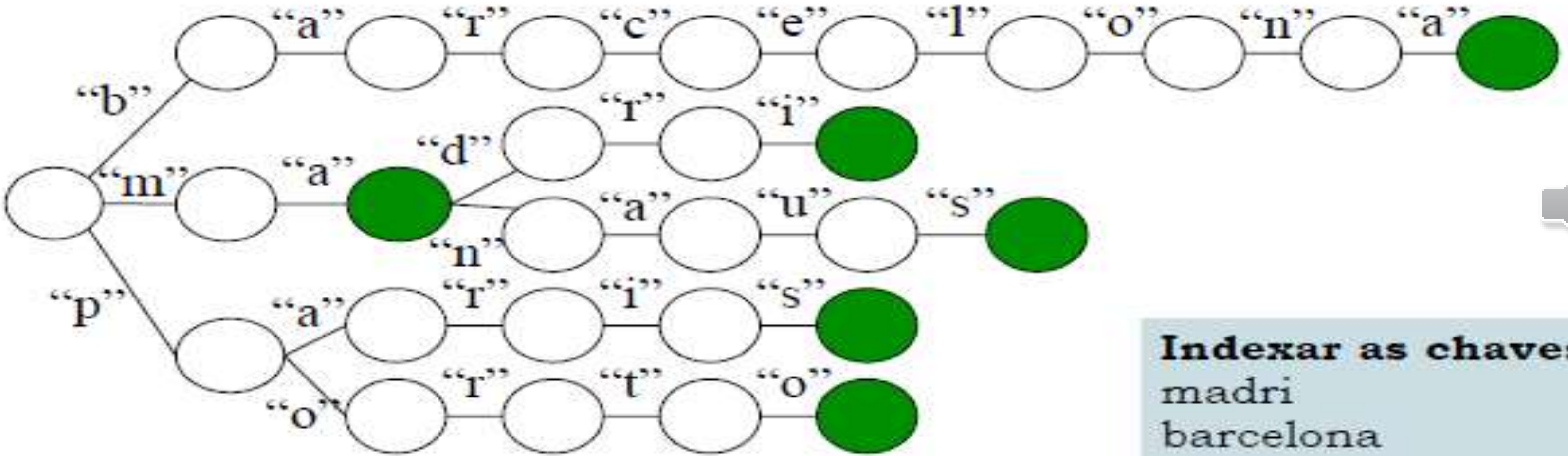
- **De acordo** com essa etimologia ele é pronunciado [**tri**] ("tree"), embora alguns encorajem o uso de [**traɪ**] ("try") de modo a diferenciá-lo do termo mais geral **tree.**

10. Árvores Digitais

- A **diferença básica** entre a **busca digital** e aquela examinada anteriormente é que a **chave**, na **busca digital**, **não** é tratada como um **elemento único, indivisível**.
- Isto é, assume-se que **cada chave** é constituída de um **conjunto de caracteres ou dígitos,** definidos em um **alfabeto** apropriado.
- **Em vez** de se comparar a **chave** procurada com as **chaves** do conjunto armazenado, a **comparação** é efetuada, **individualmente** entre os dígitos (ou caracteres) que compõem as chaves, **dígito a dígito** (ou caractere a caractere).

10.1 Árvores Digitais: exemplo

Ideia: nós **verdes apontam** para os **registros correspondentes às chaves,** e nós **brancos** apontam para **nulo**.



Indexar as chaves:
madri
barcelona
ma
manaus
paris
porto

10.2 Árvores Digitais: definição

- **$S = \{s1, .., sn\}$** é o conjunto de **chaves** a serem **indexadas.**
- Cada chave ***si*** é formada por uma **sequência de elementos *dj*** denominados **dígitos.**
- Supõe-se que existe, em ***S***, um total de ***m*** **dígitos** distintos, que compõe o **alfabeto** de ***S*.**
- Os **dígitos** do alfabeto admitem ordenação, tal que:

$$d1 < \ldots < dm$$

- Os ***p*** primeiros **dígitos** de **uma chave** compõe o **prefixo de tamanho *p*** da **chave**.

10.2 Árvores Digitais: definição

Formalmente, uma **árvore digital** para ***S*** é uma árvore ***m*-ária *T***, não vazia, tal que:

1. Se um **nó *v*** é o ***j*-ésimo** filho de seu pai, então ***v*** corresponde ao dígito ***dj*** do alfabeto de ***S***.
2. Para cada **nó *v***, a **sequência** de **dígitos** definida pelo **caminho desde a raiz de *T* até *v*** corresponde a um **prefixo** de alguma chave de ***S*.**

10.3 Árvores Digitais: síntese geral

- A **árvore digital** se constitui em uma **árvore de ponteiros**.

- **Nós** que correspondem ao **último dígito** de uma chave válida **apontam para o registro** correspondente da **chave.**

- **Nós** que **não correspondem** ao último dígito de uma chave válida contém um **ponteiro nulo.**

10.4 Árvores Digitais: algoritmos

Nos **algoritmos** a seguir, supõe-se que a **árvore digital m-ária T** se encontra armazenada da seguinte maneira.

Cada nó **v** apontado por **pt**, pt ≠ Nulo, possui **m** filhos ordenados, apontados por **pt^.pont[1], pt^.pont[2], ..., pt^.pont[m]**, respectivamente. Se algum **i-ésimo** filho deste nó está **ausente**, então **pt^.pont[i] = Nulo**.

Se nó **v** for **nó terminal** (que contém **último dígito** da chave) de alguma chave, então **pt^.info = terminal**; caso contrário **pt^.info = não terminal**.

10.5 Árvores Digitais: buscar x

A **chave** **x** a ser procurada (buscada) possui **k** **dígitos**, denotados por **d[1], d[2], d[3], ..., d[k].**

O parâmetro **pt** indica o **nó corrente da árvore**, em exame, enquanto **l** e **a** indicam o **resultado da busca**. O valor retornado por **l** é o **tamanho do maior prefixo de x** que coincide com **um prefixo** de alguma **chave.**

Esse **prefixo** é obtido percorrendo-se a árvore desde a sua **raiz** até o nó **w** apontado por **pt**, ao **final** do processo.

10.5 Árvores Digitais: buscar x

- Se **a = 1**, a **chave** foi encontrada no nó **w**; caso contrário, **a = 0.**

- A variável **ptraiz** armazena um **ponteiro para a raiz da árvore.**

- A **chamada externa** é **buscadig(x, pt, l, a), com:**

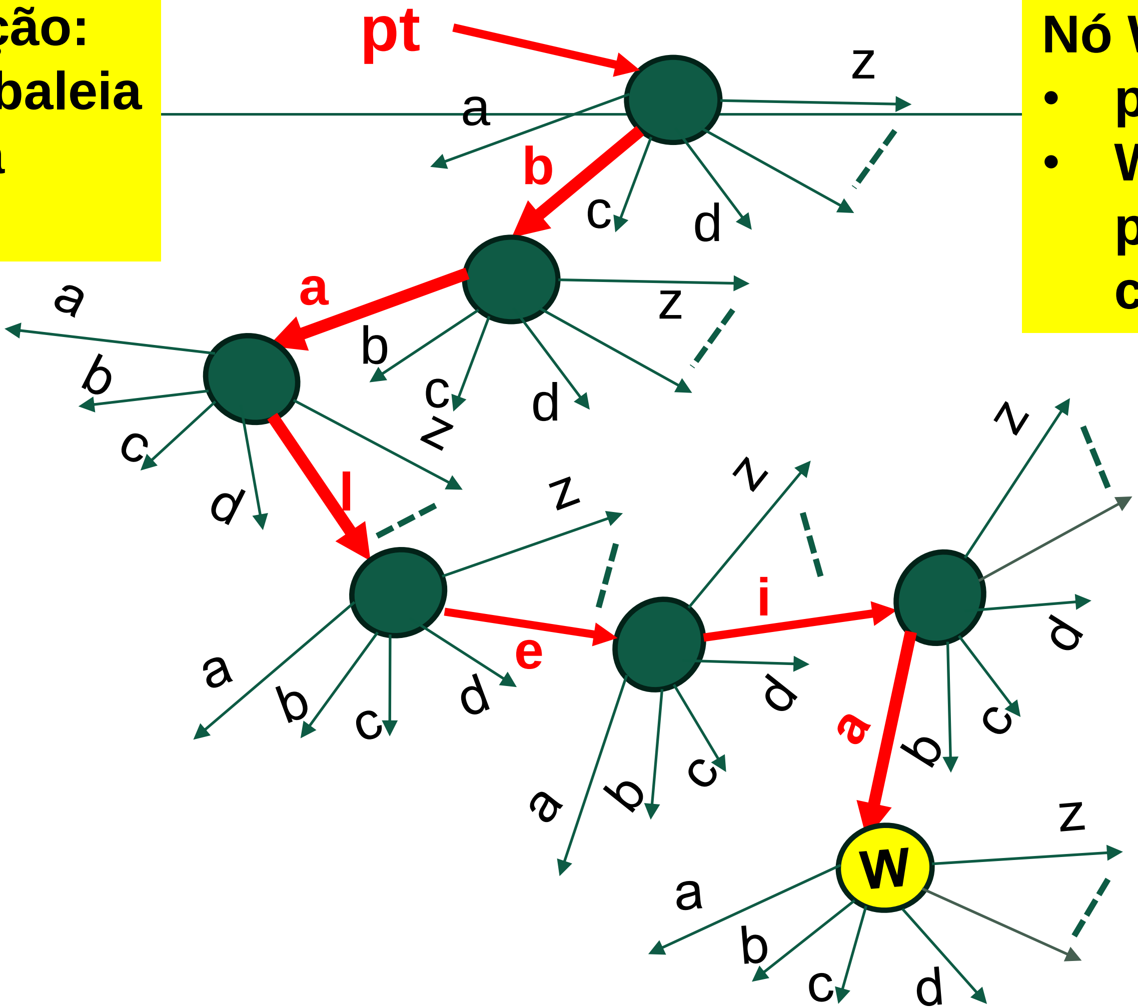
pt = ptraiz, l = a = 0

10.5 Árvores Digitais: buscar x

% pt = ptraiz; l = a = 0; x = baleia; k = 6; d[1] = b; d[2] = a, ...
% k = nº de dígitos da chave x; pt^.info = terminal ou não terminal

Procedimento buscadig(x, pt, l, a)
Se l < k então
Seja j a posição de d[l + 1] na ordenação do alfabeto.
% Ex: x = baleia; S = { a, b,...}, d[1] = b e j = 2.
Se pt^.pont[j] ≠ Nulo então
pt := pt^.pont[j]; l := l + 1
buscadig(x, pt, l, a)
Senão **se** pt^.info = terminal então a:=1 % encontrei x

Ilustração:
Se x = baleia está na árvore!



Nó W:

- **pt^.info = terminal**
- **W possui ponteiro para o registro de chave x = baleia**

10.6 Árvores Digitais: incluir x

Para **incluir** uma nova chave **x = d[1],...,d[k]** em uma árvore **m-ária T,** utiliza-se o algoritmo **buscadig**.

Edward Fredkin (*1934 +2023, 88 anos) foi um físico estadunidense.

https://en.wikipedia.org/wiki/Edward_Fredkin

10.6 Árvores Digitais: incluir x

- Se **x** foi localizado na árvore, então a **inclusão é inválida.**

- **Caso contrário**, a busca determina o nó **w** da árvore, apontado por **pt**, tal que o **caminho da raiz até w** corresponde ao **maior prefixo** que coincide com **x.** O tamanho **l** desse **prefixo** também é conhecido após a busca.

- Seja **j** a posição de **d[l + 1]** na ordenação de dígitos. Então um **novo nó** é incluído em **T**, sendo o **j-ésimo** filho de **w**. O processo se repete até que **todos dígitos** de **x** sejam incluídos.

10.7 Árvores Digitais: complexidade

A **construção da árvore digital** considera a **inclusão** de cada chave **individualmente**, iniciando por uma árvore contendo unicamente a raiz. É **possível mostrar** que:

A **complexidade da busca** de **x** é **O(k(log m + 1)),** onde m é o nº de dígitos distintos do alfabeto e **k** é o tamanho da chave **x.**

A **complexidade da inclusão** é **O(k1 log m + k2 m)**, onde **k1** é o tamanho do **maior prefixo comum** a **x** e a alguma **outra chave** da árvore, e o valor de **k2** é o **número de nós** a serem incluídos na árvore devido a **inclusão** de **x**. Isto é: **k1 + k2 = k.**

10.7 Árvores Digitais: complexidade

- Veja que a **complexidade da busca** **independe** do **número total** de **chaves** (e do **tamanho do arquivo**).

- Outra vantagem é que as **chaves** podem possuir **tamanho arbitrário e variável**.

Árvore Digital Binária

10.8 Árvore Digital Binária

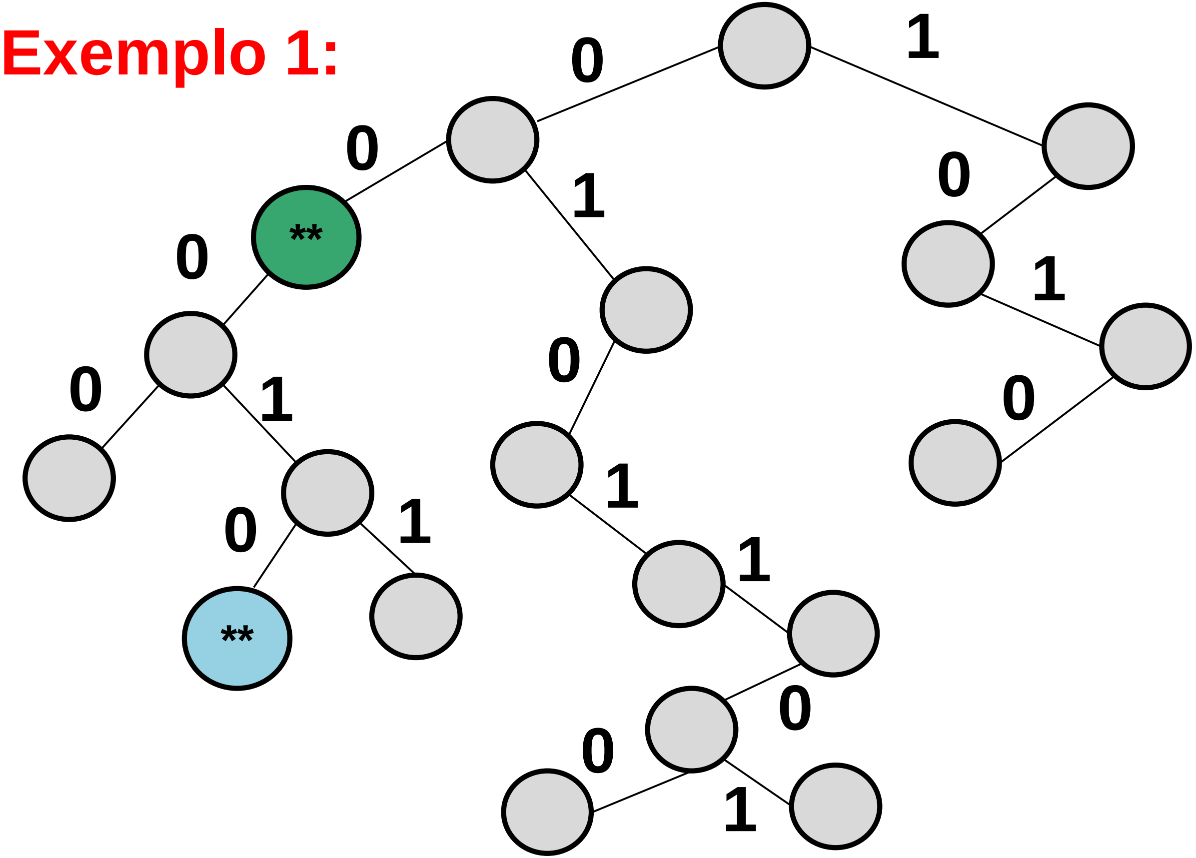
Uma **árvore digital binária** é, simplesmente, o **caso binário** da **árvore digital m-ária** com **m = 2**. Nesse caso, o alfabeto de **S = {0, 1}.**

Assim sendo, esta árvore é uma **árvore binária** e cada **chave** é uma **sequência binária**. A **seleção** do filho esquerdo de um nó é interpretada como o **dígito 0**, e o direito como dígito **1.**

A **maior utilização** das **árvores digitais** se dá, possivelmente, nesse caso **binário.**

10.8 Árvore Digital Binária

Exemplo 1:



Chaves:

00
0000
00010
00001
0101100
0101101
10
101
1010

10.8 Árvore Digital Binária

Observando-se o **Exemplo 1**, nota-se que:

- Há **chaves** binárias que são **prefixos** de outras da coleção;

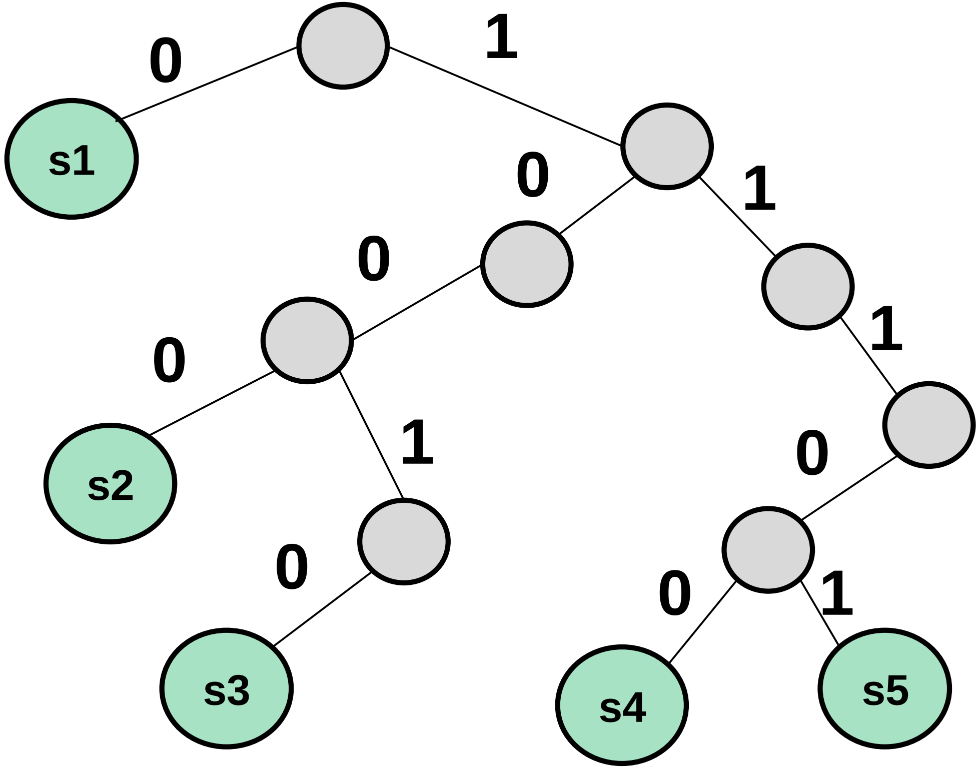
- **Exemplo**: a chave **00** é **prefixo** da chave **00**010, i.e., o caminho da raiz até o **nó (**)**, correspondente à chave **00**, é parte daquele até o **nó (**)**, que corresponde a 00**010**;

- Em especial, a **árvore binária de PREFIXO** é uma **árvore binária digital** tal que **nenhum código** (chave) é **prefixo de outro**.

10.9 Árvore Binária de Prefixo

Exemplo 2:

Árvore Binária de Prefixo



Chaves:

s0=0
s1=1000
s3=10010
s4=11100
s5=11101

10.9 Árvore Binária de Prefixo

- Em **uma árvore binária de PREFIXO**, há uma correspondência entre o **conjunto das chaves** e o das **folhas da árvore**. Isto é, **cada chave** é unicamente representada por **uma folha** e a **codificação binária** dessa chave corresponde ao **caminho da raiz até essa folha**.

- Árvore **Patricia** (Practical Algorithm to Retrieve Information Coded in Alphanumeric) é uma **implementação** da **árvore digital binária**. A **árvore Patricia** é **estritamente binária** (i.e., nós têm 0 ou 2 filhos), não possuindo, portanto, **zigue-zagues** (subárvores parciais cujos nós possuem um único filho).

11. Síntese Final

Este módulo apresentou o tema **Árvores.**
O tema foi desenvolvido sob o enfoque geral de **Algoritmos e Estruturas de Dados.**

Os slides desta aula estão em
https://sites.google.com/site/carlokleber/disciplinas-ministradas/aed-2

A **lista de exercícios**, disponível no site acima, propicia a oportunidade de praticar, complementar e fixar o conteúdo ministrado.

Prática e Estudo Individual

Acesse o MOODLE tempestivamente, acesse a aba atividades e realize a atividade prevista.

Referências

- Cormen, T. H et al., Algoritmos: Teoria e Prática. Rio de Janeiro: Editora Campus, 2ª edição, 2002
- Ziviani, N. Projeto de Algoritmos com implementação em Java e C++.São Paulo: Editora Thomson, 1ª edição, 2007
- Szwarcfiter, J. L.; Markenzon, L. Estruturas de dados e seus algoritmos. Editora LTC, 3ª edição, 2010.
- João Arthur Brunet, 2019. Estruturas de Dados e Algoritmos, Computação @ UFCG, http://joaoarthurbm.github.io/eda.
- Ferraz, I. N. Programação com Arquivos. Editora Manole Ltda., 2003.

Fim

Prof. Carlo Kleber da Silva Rodrigues
carlo.kleber@ufabc.edu.br