Estruturas de Dados e Algoritmos

Introdução

- O que é um algoritmo?
  - Processo sistemático para computar um resultado a partir de dados de entrada
- O que são estruturas de dados?
  - Maneira de organizar dados e operar sobre eles
- Algoritmos + estruturas de dados = programas
  - Um programa é a expressão em linguagem formal (inteligível por um computador) de um algoritmo.

Projeto de Algoritmos

- Entender a entrada
- Entender o que se espera na saída
- Repetir:
  - Bolar um método,
  - Se o método é correto, então
    - Analisar a complexidade do método,
    - Se complexidade é aceitável, terminar.
- Implementar (programar)

Projeto de Estruturas de Dados

- Uma modelagem abstrata dos objetos a serem manipulados e das operações sobre eles
  - Tipo de Dados Abstrato ("*Abstract Data Type*")
  - Ex.: Uma "pilha", com operações "push", "pop" etc.
- Uma modelagem concreta do TDA, isto é, como armazenar o TDA em memória/disco e que algoritmos devem ser usados para implementar as operações
  - Ex.: Pilha armazenada como lista encadeada ou vetor
  - ...

Projeto versus Implementação

- Um bom projeto leva a uma boa implementação
  - Todas as idéias principais já foram estudadas
  - A tradução do projeto em programa é *quase* mecânica
- (ou não)
  - Programar é uma *arte*
  - Um algoritmo inferior bem programado pode ser mais útil que um algoritmo eficiente mal programado
  - Há considerações práticas quase tão importantes quanto um bom projeto, como por exemplo,
    - Interfaces
    - Manutenibilidade
    - Documentação

Algoritmos e Complexidade

- Eficiência de um algoritmo
  - Complexidade de tempo: quanto “tempo” é necessário para computar o resultado para uma instância do problema de tamanho *n*
    - Pior caso: Considera-se a instância que faz o algoritmo funcionar mais lentamente
    - Caso médio: Considera-se todas as possíveis instâncias e mede-se o tempo médio
- Eficiência de uma estrutura de dados
  - Complexidade de espaço: quanto “espaço de memória/disco” é preciso para armazenar a estrutura (pior caso e caso médio)
- Complexidade de espaço e tempo estão freqüentemente relacionadas

Algoritmos e Complexidade

- Eficiência medida objetivamente depende de:
  - Como o programador implementou o algoritmo/ED
  - Características do computador usado para fazer experimentos:
    - Velocidade da CPU
    - Capacidade e velocidade de acesso à memória primária / secundária
    - Etc
  - Linguagem / Compilador / Sistema Operacional / etc
- Portanto, a medição formal de complexidade tem que ser subjetiva, porém matematicamente consistente

  $\Rightarrow$ Complexidade assintótica

Complexidade Assintótica

- Tempo / espaço medidos em número de “passos” do algoritmo / “palavras” de memória ao invés de segundos ou bytes
- Análise do algoritmo / e.d. permite estimar uma função que depende do tamanho da entrada / número de dados armazenados (n).
  - Ex.: $T(n) = 13n^3 + 2n^2 + 6n \log n$
- Percebe-se que à medida que n aumenta, o termo cúbico começa a dominar
- A constante que multiplica o termo cúbico tem relativamente a mesma importância que a velocidade da CPU / memória
- Diz-se que $T(n) \in O(n^3)$

Complexidade Assintótica

- Definição:

  $T(n) \in O(f(n))$ se existem constantes c e n_0 tais que $T(n) \leq c\, f(n)$ para todo $n \geq n_0$

- Alternativamente,

  $T(n) \in O(f(n))$ se $\lim_{n\to\infty} T(n) / f(n)$ é constante (mas não infinito)

- Exemplo:

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{13n^3 + 2n^2 + 6n\log n}{n^3}$$

$$= \lim_{n\to\infty} \left(13 + \frac{2}{n} + \frac{\log n}{n^2} \right)$$

$$= 13$$

Limite Superior e Limite Inferior

- Notação *O* é usada para estabelecer limites superiores de complexidade
- Para estabelecer limites inferiores de complexidade usa-se a notação Ω
- Definição

  $T(n) \in \Omega(f(n))$ se existem constantes c e n_0 tais que $c\,f(n) \leq T(n)$ para todo $n \geq n_0$

- Alternativamente,

  $T(n) \in \Omega(f(n))$ se $\lim_{n\to\infty} T(n) / f(n) > 0$ para todo $n \geq n_0$ (pode ser, inclusive, infinito)

Limites Justos

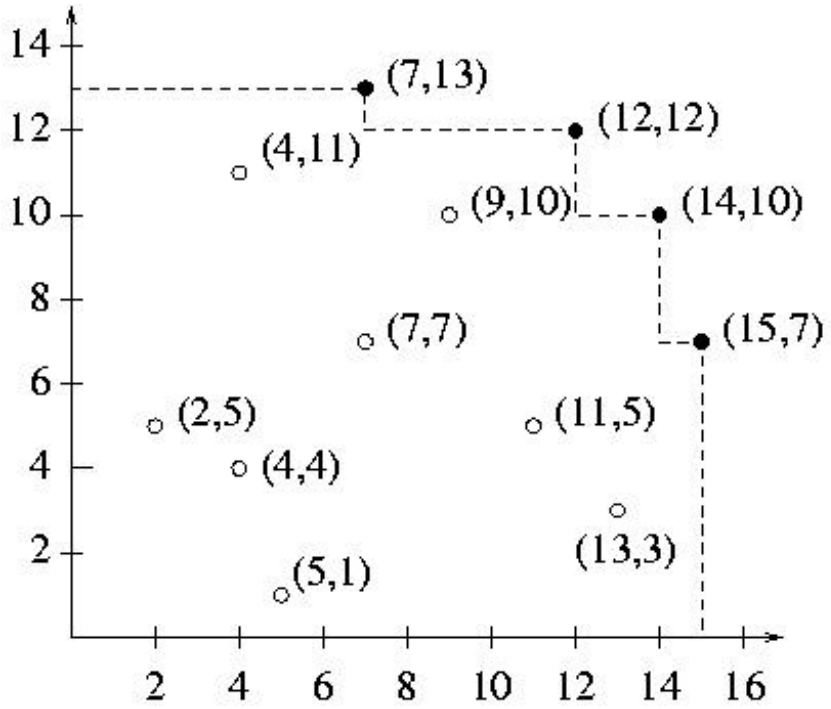
- Observe que se $T(n) \in O(n^3)$ então, $T(n) \in O(n^4)$, $T(n) \in O(n^5)$, etc
- Analogamente, se $T(n) \in \Omega(n^3)$ então, $T(n) \in \Omega(n^2)$, $T(n) \in \Omega(n)$, etc
- Se uma função $T(n)$ tem como limites superior e inferior a mesma função $f(n)$, então diz-se que $f(n)$ é um limite justo de $T(n)$, ou $T(n) \in \Theta(f(n))$
- Definição
  - $T(n) \in \Theta(f(n)) \Leftrightarrow T(n) \in O(f(n)) \wedge T(n) \in \Omega(f(n))$

Inventário de funções de complexidade

- $T(n) \in O(1)$: constante – mais rápido, impossível
- $T(n) \in O(\log \log n)$: super-rápido
- $T(n) \in O(\log n)$: logarítmico – muito bom
- $T(n) \in O(n)$: linear – é o melhor que se pode esperar se algo não pode ser determinado sem examinar toda a entrada
- $T(n) \in O(n \log n)$: limite de muitos problemas práticos, ex.: ordenar uma coleção de números
- $T(n) \in O(n^2)$: quadrático
- $T(n) \in O(n^k)$: polinomial – ok para n pequeno
- $T(n) \in O(k^n)$, $O(n!)$, $O(n^n)$: exponencial – evite!

Exemplo: Pontos máximos em 2D

- Um ponto máximo de uma coleção é um que não é *dominado* por nenhum outro (da coleção)
- Diz-se que um ponto (x_1, y_1) domina um ponto (x_2, y_2) se $x_1 \geq x_2$ e $y_1 \geq y_2$



Exemplo – Algoritmo Força Bruta

- Basta testar todos os pontos e verificar se são máximos:

```
proc maximos (Inteiro n, Ponto p [1..n]) {
    para i desde 1 até n fazer {
        dominado ← falso
        j ← 1
        enquanto ¬dominado ∧ j≤n fazer {
            se i≠j ∧ domina (p[j],p[i]) então
                dominado ← verdadeiro
            j ← j + 1
        }
        se ¬dominado então reportar (p[i])
    }
}
```

Observações sobre pseudo-código

- É uma descrição do algoritmo para humanos
- Não precisa conter detalhes desnecessários
- Ex.: Assumimos que p não contém pontos duplicados, mas este pode ser um detalhe importante para o implementador
- Precisa ser inteligível
- Se o algoritmo usa outro algoritmo, este deve ser óbvio ou deve ser explicitado.
- Ex.: função *domina* deve ser explicitada?

proc *domina* (Ponto *p*, Ponto *q*) {

  retornar $p.x \geq q.x \ \wedge \ p.y \geq q.y$

}

Correção do algoritmo

- Se o algoritmo não é óbvio, deve-se dar uma justificativa de sua correção
- Em particular:
  - Enumere restrições sobre a entrada admitida pelo algoritmo
  - Diga porque cada resultado computado satisfaz o que é pedido
  - Diga porque nenhum resultado correto é omitido

Análise de complexidade (pior caso)

- O pior caso acontece quando todos os pontos são máximos
- O interior do laço mais interno tem complexidade constante, digamos 2 (o comando “se” e a atribuição a “j”)
- O laço mais interno tem complexidade $\sum_{j=1}^{n} 2$
- O interior do laço mais externo tem complexidade $3+\sum_{i=1}^{n} 2$
- O algoritmo tem complexidade

$$\sum_{i=1}^{n}\left(3+\sum_{j=1}^{n} 2\right)=n(3+2n)=3n+2n^2$$

Somatórios

- Propriedades

$$\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Alguns somatórios notáveis

Série aritmética : para $n \geq 0$

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

Série geométrica : seja x uma constante $\neq 1$

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} = \begin{cases} \Theta(1), \text{se } 0 < x < 1 \\ \Theta(x^n), \text{se } x > 1 \end{cases}$$

Série Harmônica :

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n = \Theta(\ln n)$$

Resolvendo somatórios

- O que faríamos se não soubéssemos que

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

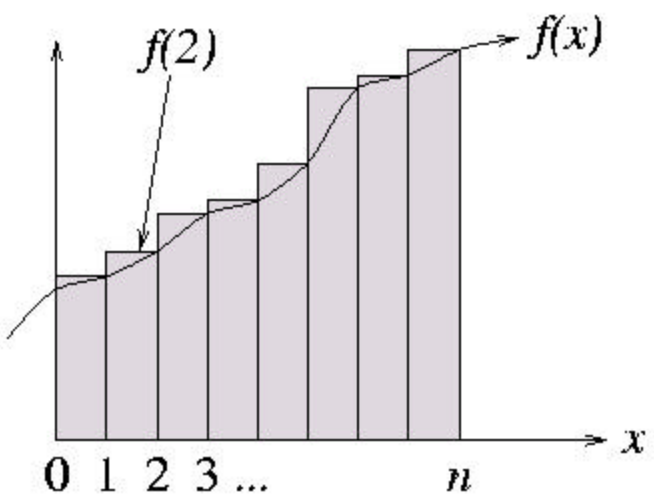
- Usar limites aproximados

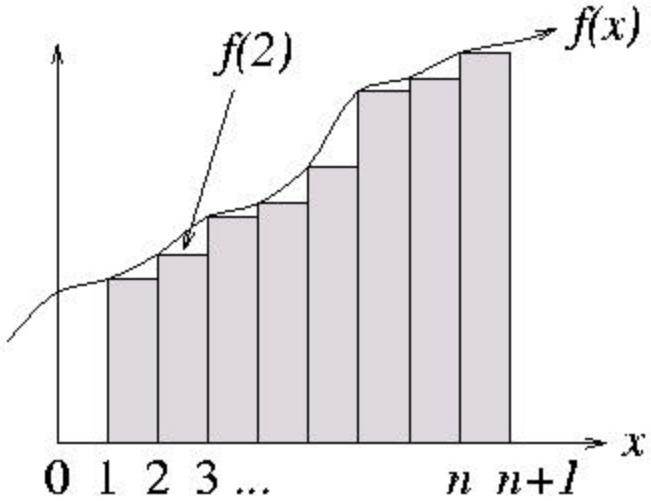
$$\sum_{i=1}^{n} i^2 \leq \sum_{i=1}^{n} n^2 = n^3 = \Theta(n^3)$$

- Aproximar por integrais

$$\sum_{i=1}^{n} i^2 \leq \int_{1}^{n+1} x^2 dx = \frac{x^3}{3}\Big|_{x=1}^{n+1} = \frac{(n+1)^3}{3} - \frac{1}{3} = \frac{n^3 + 3n^2 + 3n}{3}$$

Justificando a aproximação por integral





Resolvendo somatórios por indução

- Formula-se um palpite e tenta-se prová-lo. Ex.:

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d$$

- Prova do caso base:
  - Para $n = 0$, o somatório é 0
  - Trivialmente verdadeiro se admitirmos $d = 0$
- Prova do caso genérico

$$\sum_{i=1}^{n} i^2 = \left(\sum_{i=1}^{n-1} i^2 \right) + n^2$$

$$= a(n-1)^3 + b(n-1)^2 + c(n-1) + n^2$$

$$= an^3 + (-3a + b + 1)n^2 + (3a - 2b + c)n + (-a + b - c)$$

Resolvendo somatórios por indução

- Prova do caso genérico:
  - Coeficientes de potências iguais têm “bater”

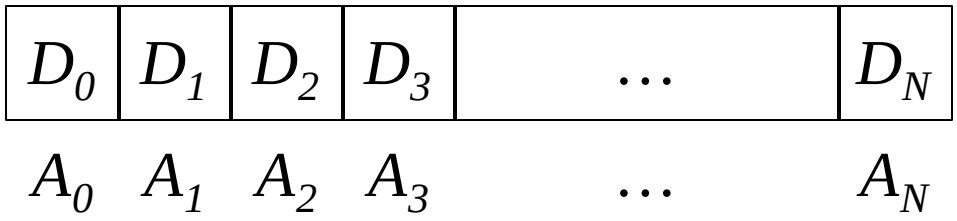
$$a = a, \quad b = -3a + b + 1, \quad c = 3a - 2b + c \quad -a + b - c = 0$$

  - Resolvendo temos $a = 1/3$, $b = 1/2$ e $c = 1/6$

$$\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{2n^3 + 3n^2 + n}{6}$$

Array (vetores, matrizes)

- Organiza dados de mesma natureza (mesmo tamanho) em posições sucessivas da memória



- Cada dado é identificado por um índice
- Dado um índice *i* é possível computar o endereço de memória correspondente em tempo constante
  - Se o array é alocado a partir do endereço A_0 e cada dado ocupa *k* posições, então o *i*-ésimo elemento está no endereço $A_i = A_0 + i.k$
- Matrizes são construídas analogamente como vetores de vetores

Array

- Estrutura de dados fundamental
  - Diversas outras estruturas são implementadas usando arrays
  - Em última análise, a própria memória é um array
- Problemas:
  - Alocação de memória
    - Quantas posições deve ter o array para uma dada aplicação?
    - O que fazer se precisarmos mais?
  - Como inserir um novo dado entre o k-ésimo e o $(k+1)$-ésimo elemento?
  - Como remover o k-ésimo elemento?

Busca em Arrays

- Dado um array *A* contendo *n* valores nas posições *A*[0] .. *A*[*n*-1] e um valor *v*, descobrir:
  - Se *v* pertence ao array (problema + simples)
  - Qual ou quais posições de *A* contêm *v* (normalmente assume-se que todos os dados em *A* são distintos)
- Algoritmo trivial: busca sequencial

```
proc buscasequencial (v, n, A[0..n-1]) {
    achei ← falso
    i ← 0
    enquanto i < n e não achei fazer {
        se A[i] = v então achei ← verdadeiro
        senão i ← i + 1
    }
    se achei então reportar (i) senão reportar(-1)
}
```

Busca em Arrays

- Busca seqüencial simples testa três condições dentro do laço.
- É possível alterar o algoritmo para empregar apenas um teste no laço de repetição
- Busca com *Sentinela*:
  - Usa-se uma posição a mais no final do array ($A[n]$) que é carregada com uma cópia do dado sendo buscado (v)
  - Como é garantido que v será encontrado, não é preciso se precaver contra o acesso de uma posição i não existente

Busca Seqüencial com Sentinela

```
proc busca_com_sentinela (v, n, A[0..n]) {
    A[n] ← v
    i ← 0
    enquanto A [i] ≠ v fazer {
        i ← i + 1
    }
    se i < n então
        reportar (i)            % encontrado
    senão
        reportar(-1)            % não encontrado
}
```

Busca Seqüencial – Análise

- A análise de pior caso de ambos os algoritmos para busca seqüencial são obviamente $O(n)$, embora a busca com sentinela seja mais rápida
- A análise de caso médio requer que estipulemos um modelo probabilístico para as entradas. Sejam:
  - E_0 , E_1 , … E_{n-1} as entradas v correspondentes às situações onde $v=A[0]$, $v=A[1]$, … $v=A[n-1]$
  - E_n entradas v tais que v não pertence ao array A
  - $p(E_i)$ a probabilidade da entrada E_i ocorrer
  - $t(E_i)$ a complexidade do algoritmo quando recebe a entrada E_i
- Assumimos:
  - $p(E_i) = q/n$ para $i < n$
  - $p(E_n) = 1-q$

Busca Seqüencial – Análise de Caso Médio

- Se admitirmos $t(E_i) = i+1$, então temos como complexidade média:

$$\sum_{i=0}^{n} p(E_i)\, t(E_i) = (n+1)(1-q) + \frac{q}{n}\left(\sum_{i=0}^{n-1} i + 1 \right)$$

$$= (n+1)(1-q) + \frac{q}{n}\frac{n(n+1)}{2}$$

$$= \frac{(n+1)(2-q)}{2}$$

- Para $q=1/2$, temos complexidade média $\approx 3n/4$
- Para $q=0$, temos complexidade média $\approx n$
- Para $q=1$, temos complexidade média $\approx n/2$

Arrays Ordenados

- Se os dados se encontram ordenados (em ordem crescente ou decrescente), a busca pode ser feita mais eficientemente
- Ordenação toma tempo $\Theta(n \log n)$
- Útil se a coleção não é alterada ou se é alterada pouco freqüentemente
- Busca seqüencial ordenada tem complexidade média = $n/2$
- Busca binária tem complexidade pior caso $O(\log n)$

Busca Seqüencial Ordenada

```
proc busca_ordenada (v, n, A [0 .. n]) {
    A[n] ← v
    i ← 0
    enquanto A [i] < v fazer i ← i + 1
    se i < n e A [i] = v então
        reportar (i)            % encontrado
    senão
        reportar (-1)           % não encontrado
}
```

Busca Binária

```
proc busca_binária (v, n, A [0 .. n–1]) {
    inf ← 0          % limite inferior
    sup ← n-1        % limite superior
    enquanto inf ≤ sup fazer {
        meio ← (inf + sup) div 2
        se A [meio] < v então
            inf ← meio + 1
        senão se A[meio] > v então
            sup ← meio – 1
        senão
            retornar (meio)      % Valor encontrado
    }
    retornar (-1)                % Valor não encontrado
}
```

Busca Binária - Análise de Complexidade

- O algoritmo funciona examinando as posições $A[inf], A[inf+1], \ldots A[sup]$
- Cada iteração do laço elimina aproximadamente metade das posições ainda não examinadas. No pior caso:
  - Inicialmente: n
  - Após a 1ª iteração: $\sim n/2$
  - Após a 2ª iteração: $\sim n/4$

    …
  - Após a k-ésima iteração: $\sim n/2^k = 1$
- Logo, no pior caso, o algoritmo faz $\sim \log_2 n$ iterações, ou seja, o algoritmo tem complexidade $O(\log n)$

Arrays - Inserção e Remoção de Elementos

- É preciso empregar algoritmos de busca se:
  - A posição do elemento a ser removido não é conhecida
  - O array não pode conter elementos repetidos
- Se o array é ordenado, deseja-se preservar a ordem
  - Deslocar elementos para criar / fechar posições
- Se o array não é ordenado,
  - Inserção: Adicionar elemento no final do array
  - Remoção: Utilizar o elemento do final do array para ocupar a posição removida
- Se todas as posições estão preenchidas, inserção ocasiona *overflow*
  - Realocar o array
  - Reportar erro

Exemplo: Inserção em Array Ordenado

- Assume-se que o array *A* pode conter elementos iguais
- Expressões lógicas são avaliadas em curto-circuito

```
proc inserção_ordenada (v, n, max, A [0 .. max – 1]) {
    se n < max então {
        i ← n
        enquanto i > 0 e A [i–1] > v fazer {
            A [i] ← A [i–1]
            i ← i–1
        }
        A [i] ← v
        n ← n + 1
    }
    senão reportar ("Overflow")
}
```

Exemplo: Remoção em Array Ordenado

- Algoritmo remove o elemento $A[i]$
- Pressupõe-se que i foi obtido por uma operação de busca

```
proc remoção_ordenada (i, n, A [0 .. n–1]) {
    se i < n então {
        n ← n–1
        enquanto i < n fazer {
            A [i] ← A [i+1]
            i ← i+1
        }
    }
    senão reportar ("Erro")
}
```

Complexidade de Inserção e Remoção

- Os dois algoritmos para arrays ordenados têm complexidade de pior caso $O(n)$
- É possível realizar inserção e remoção em $O(1)$ se não for necessário preservar ordem entre os elementos
- Observe que:
  - Array ordenado
    - Busca (binária) = $O(\log n)$
    - Inserção/Remoção = $O(n)$
  - Array não ordenado
    - Busca (seqüencial) = $O(n)$
    - Inserção/Remoção = $O(1)$

Pilhas, Filas e Deques

- Arrays, assim como listas*, são freqüentemente usados para implementar coleções seqüenciais de dados onde as alterações (inserção/remoção) são efetuadas apenas no início ou no final da seqüência:
  - Pilha: inserção e remoção na mesma extremidade
  - Fila: inserção numa extremidade e remoção na outra
  - Deque (*double-ended queue*): inserção e remoção em ambas extremidades

* OBS.: Lembre que estamos empregando o termo "*array*" para denotar coleções de dados de mesmo tamanho armazenados contiguamente em memória. Falaremos de *listas* mais tarde.

Pilhas

- Dada uma pilha *P*, podemos definir as seguintes operações:
  - *Empilha* (dado *v*, pilha *P*): acrescenta o dado *v* no topo da pilha. Pode ocasionar *overflow*
  - *Desempilha* (pilha *P*): descarta o dado mais recentemente empilhado (no topo da pilha). Pode ocasionar *underflow*
  - *Altura* (pilha *P*): retorna o número de elementos de *P*
  - *Topo* (pilha *P*): retorna o dado mais recentemente empilhado. Definida apenas se *Altura* (*P*) > 0
- A política de inserção e remoção à maneira de uma pilha é também conheciada como "*LIFO*": *Last In, First Out*

Implementando Pilhas com Arrays

- Assumimos que uma pilha *P* tem os seguintes campos/componentes:
  - *P*.*max* = número máximo de dados comportado pela pilha
  - *P*.*A* [0 .. *P*.*max* – 1] = array com *P*.*max* elementos
  - *P*.*n* = número de elementos presentes na pilha (inicialmente 0)
- Nossa implementação armazena os dados na pilha em *P*.*A* [0 .. *P*.*n* – 1], na mesma ordem em que foram empilhados:
  - *P*.*A* [0] é o dado mais antigo
  - *P*.*A* [*P*.*n* – 1] é o dado mais recente

Implementando Pilhas com Arrays

```
proc empilha (dado v, pilha P) {
    se P.n < P.max então {
        P.A [P.n] ← v
        P.n ← P.n + 1
    }
    senão reportar ("Overflow")
}
```

```
proc desempilha (pilha P) {
    se P.n > 0 então P.n ← P.n – 1
    senão reportar ("Underflow")
}
```

```
proc altura (pilha P) {
    retornar (P.n)
}
```

```
proc topo (pilha P) {
    retornar (P.A [P.n – 1])
}
```

Complexidade da Implementação de Pilha

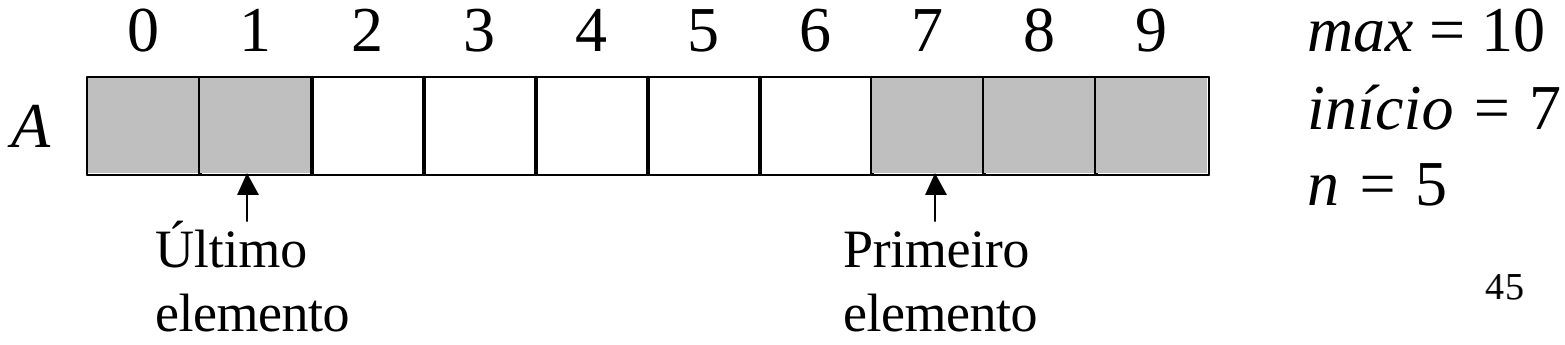
- Todas as operações são $O(1)$
- Se for necessário tratar *overflow* com realocação, inserção pode ter complexidade de pior caso $O(n)$
  - Um novo array de comprimento maior (ex.: 2 *max*) é alocado
  - Todos os elementos são copiados para o novo array

Filas

- São comumente definidas as seguintes operações sobre filas:
  - *Enfileira* (dado *v*, fila *F*) : o dado *v* é posto na fila *F*
  - *Desenfileira* (fila *F*) : descarta o dado mais antigo (menos recentemente enfileirado) da fila *F*
  - *Comprimento* (fila *F*) : retorna o número de elementos na fila *F*
  - *Próximo* (fila *F*) : retorna o dado mais antigo da fila *F*
- A política de inserção e remoção de dados à maneira de uma fila é conhecida como "*FIFO*" – *First In First Out*

Filas Implementadas com Arrays

- Uma fila *F* pode ser implementada usando uma estrutura com os seguintes campos
  - *F.max* = número máximo de dados
  - *F.A* [0 .. *F.max*–1] = array onde os dados são postos
  - *F.início* = índice do 1º elemento da fila (inicialmente 0)
  - *F.n* = número de elementos da fila
- Os elementos da fila são armazenados consecutivamente a partir de *F.A* [*F.início*] podendo "dar a volta" e continuar a partir de *F.A* [0]. Exemplo:



Filas Implementadas com Arrays

```
proc enfileira (dado v, fila F) {
    se F.n < F.max então {
        F.A [(F.início + F.n) mod F.max] ← v
        F.n ← F.n + 1
    }
    senão reportar ("Overflow")
}
```

```
proc desenfileira (fila F) {
    se F.n > 0 então {
        F.início ← (F.início + 1) mod F.max
        F.n ← F.n – 1
    }
    senão reportar ("Underflow")
}
```

```
proc comprimento (fila F) {
    retornar F.n
}
```

```
proc próximo (fila F) {
    retornar F.A [F.início]
}
```

Ordenação de Arrays

- Operação de grande importância teórica e prática
- Muitos algoritmos conhecidos com complexidade $O(n \log n)$:
  - *HeapSort*
  - *QuickSort*
  - *MergeSort*
- Freqüentemente, algoritmos com complexidade assintótica pior – tipicamente $O(n^2)$ – acabam sendo mais eficientes para n pequeno:
  - *BubbleSort*
  - *InsertionSort*

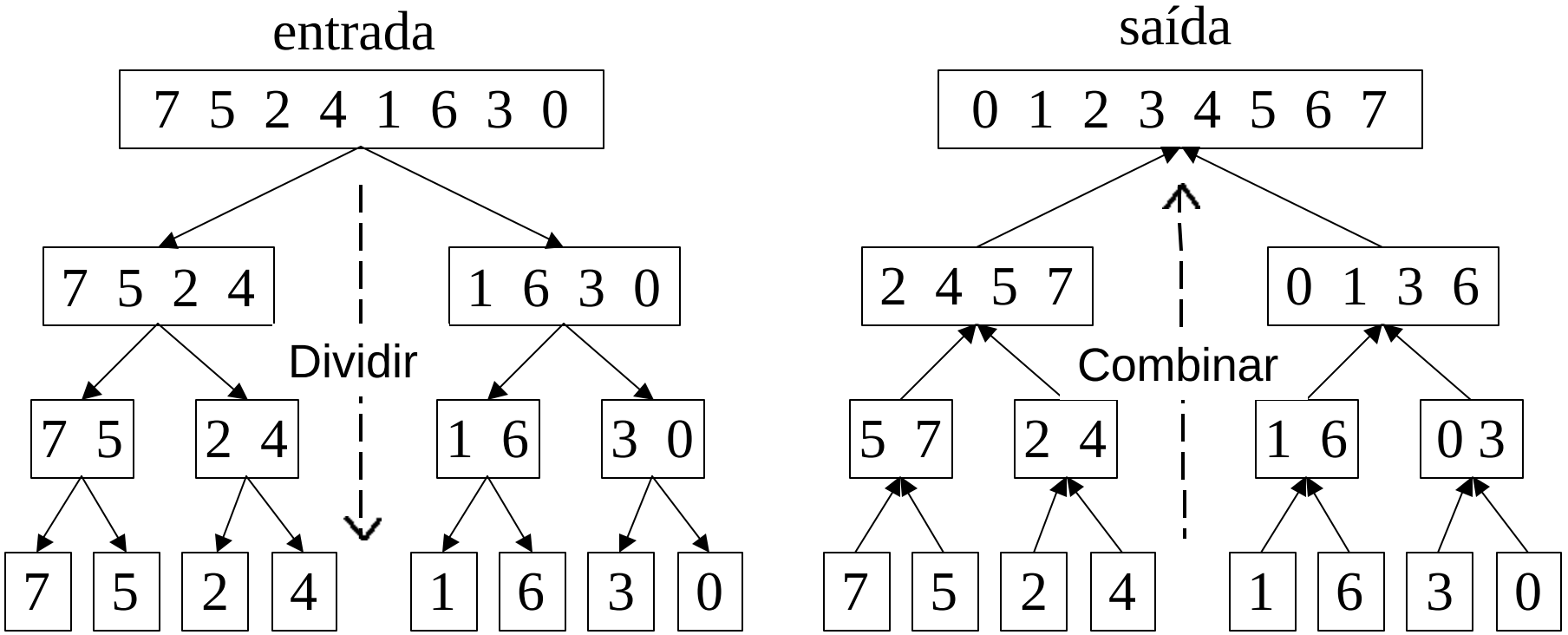
Dividir para Conquistar

- Vamos estudar o *MergeSort*, um algoritmo sob o princípio "Dividir para Conquistar" ou "*Divide and Conquer*"
- Consiste de:
  - Dividir a tarefa em pequenas subtarefas
  - "Conquistar" – resolver cada subtarefa aplicando o algoritmo recursivamente a cada uma
  - Combinar as soluções das subtarefas construindo assim a solução do problema como um todo
- Tipicamente, algoritmos do tipo "Dividir para Conquistar" são recursivos
- Na análise de algoritmos recursivos os limites de complexidade precisam ser determinados resolvendo recorrências

MergeSort

- Considere um array $A[1..n]$. O algoritmo consiste das seguintes fases
  - **Dividir** A em 2 sub-coleções de tamanho $\approx n/2$
  - **Conquistar**: ordenar cada sub-coleção chamando *MergeSort* recursivamente
  - **Combinar** as sub-coleções ordenadas formando uma única coleção ordenada
- Se uma sub-coleção tem apenas um elemento, ela já está ordenada e configura o caso base do algoritmo

MergeSort


entrada
7 5 2 4 1 6 3 0
7 5 2 4
1 6 3 0
Dividir
7 5
2 4
1 6
3 0
7
5
2
4
1
6
3
0
saída
0 1 2 3 4 5 6 7
2 4 5 7
0 1 3 6
Combinar
5 7
2 4
1 6
0 3
7
5
2
4
1
6
3
0

MergeSort

- A rotina *MergeSort* abaixo ordena as posições i, i+1, … i+n–1 do array A[]
- A rotina *Merge* é dada a seguir

```
proc MergeSort (A [], i, n) {
    se n > 1 então {
        m ← n div 2
        MergeSort (A, i, m)
        MergeSort (A, i + m, n – m)
        Merge (A, i, i + m, n)
    }
}
```

MergeSort

```
proc Merge (A [], i1, i2, n) {
    array B []
    i, j, k ← i1, i2, i1
    enquanto i < i2 e j < i1 + n fazer {
        se A [i] ≤ A[j] então {
            B [k] ← A [i]
            k, i ← k + 1, i + 1
        } senão {
            B [k] ← A [j]
            k, j ← k + 1, j + 1
        }
    }
    enquanto i < i2 fazer {
        B [k] ← A [i]
        i, k ← i + 1, k + 1
    }
    para i desde i1 até j–1 fazer {
      A [i] ← B [i]
    }
}
```

MergeSort - Considerações

- MergeSort é um algoritmo de ordenação estável
  - Se dois elementos são iguais eles nunca são trocados de ordem
  - Importante, por exemplo, se os elementos já estão ordenados segundo alguma chave secundária
- O array auxiliar *B* não pode ser evitado sem piorar a performance do algoritmo
- O "overhead" da recursão pode ser aliviado empregando-se um algoritmo mais simples quando os arrays forem pequenos (algumas dezenas)

MergeSort - Considerações

- Pode-se aliviar a situação evitando a cópia dos elementos de *B* de volta para *A*
  - Usa-se 2 arrays *A* e *B* de mesmo comprimento
  - Em níveis pares da recursão, *Merge* opera em *A* usando *B* para armazenar o resultado
  - Em níveis ímpares a situação é invertida
  - Ao final pode ser necessário copiar de *B* para *A* novamente (se o número de níveis de recursão for ímpar)

MergeSort - Análise

- Análise da rotina *Merge*:
  - Cada chamada mescla um total de n elementos
  - Há três laços de repetição, sendo que cada iteração executa um número fixo de operações (que não depende de n)
  - O total de iterações dos 2 primeiros laços não pode exceder n, já que cada iteração copia exatamente um elemento de A para B
  - O total de iterações do terceiro laço não pode exceder n, já que cada iteração copia um elemento de B para A
  - Vemos então que no máximo $2n$ iterações são executadas no total e portanto o algoritmo é $O(n)$

MergeSort - Análise

- Análise da rotina *MergeSort*
  - Admitamos que $T(n)$ represente a complexidade (número de passos, comparações, etc) de *MergeSort*
  - Para $n = 1$, o tempo é constante. Como estamos desprezando fatores constantes, digamos que $T(1) = 1$
  - Para $n > 1$, a rotina chama:
    - a si mesma, recursivamente:
      - » Uma vez c/ n valendo $\lfloor n/2 \rfloor$
      - » Outra vez c/ n valendo $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$
    - *Merge*, que executa n operações
  - Portanto, para $n > 1$, $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$

Resolvendo Recorrências

- Vimos que a análise do algoritmo *MergeSort* resultou numa fórmula recorrente:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{se } n > 1. \end{cases}$$

- Para resolver tais problemas, pode-se empregar muitas técnicas. Vamos ver apenas algumas:
  - "Chute" + verificação por indução
  - Iteração

Percebendo padrões

- Vejamos como $T(n)$ se comporta para alguns valores de n

$T(1) = 1$

$T(2) = T(1) + T(1) + 2 = 1 + 1 + 2 = 4$

$T(3) = T(2) + T(1) + 3 = 4 + 1 + 3 = 8$

$T(4) = T(2) + T(2) + 4 = 4 + 4 + 4 = 12$

…

$T(8) = T(4) + T(4) + 8 = 12 + 12 + 8 = 32$

…

$T(16) = T(8) + T(8) + 16 = 32 + 32 + 16 = 80$

…

$T(32) = T(16) + T(16) + 32 = 80 + 80 + 32 = 192$

Percebendo Padrões

- Podemos vislumbrar que como o algoritmo opera dividindo os intervalos sempre por 2, um padrão pode emergir para valores de n iguais a potências de 2
- De fato observamos o seguinte quando consideramos o valor de $T(n) / n$:

  $T(1) / 1 = 1$

  $T(2) / 2 = 2$

  $T(4) / 4 = 3$

  $T(8) / 8 = 4$

  $T(16) / 16 = 5$

  …

  $T(2^k) / 2^k = k+1$

- Ou seja, para potências de 2, $T(n) / n = (\log_2 n) + 1$, ou $T(n) = (n \log_2 n) + n$

Provando o Palpite por Indução

- Primeiro vamos nos livrar dos arredondamentos $T(\lfloor n/2 \rfloor)$ e $T(\lceil n/2 \rceil)$ provando o teorema apenas para valores de n iguais a potências de 2
- Esta hipótese simplificadora se justifica pois o algoritmo não se comporta de maneira significativamente diferente quando n não é uma potência de 2
- Portanto, temos

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T(n/2) + n & \text{se } n > 1. \end{cases}$$

- Vamos provar por indução que, para $n = 1$ ou qualquer valor par de n maior que 1, $T(n) = (n \log_2 n) + n$

Provando o Palpite por Indução

- Caso base: $n = 1$
  - $T(1) = (1 \log_2 1) + 1 = 0 + 1 = 1$
- Caso geral: como n é uma potência de 2, n/2 também é e podemos admitir que a hipótese é verdadeira para qualquer potência de 2 $n' < n$
  - $T(n) = 2\,T\,(n / 2) + n$
    $= 2\,((n / 2) \log_2 (n /2) + n / 2) + n$
    $= (n \log_2 (n / 2) + n) + n$
    $= n \log_2 (n / 2) + 2n$
    $= n (\log_2 n - \log_2 2) + 2n$
    $= n (\log_2 n - 1) + 2n$
    $= n \log_2 n + n$

Método da Iteração

- Nem sempre temos intuição suficiente sobre o funcionamento do algoritmo para dar um palpite correto
- O método da iteração permite que se reconheça um padrão sem necessidade de chutar
- Quando funciona, a solução do problema da recorrência é obtida resolvendo-se um somatório
- O método consiste esquematicamente de:
  - Algumas iterações do caso geral são expandidas até se encontrar uma lei de formação
  - O somatório resultante é resolvido substituindo-se os termos recorrentes por fórmulas envolvendo apenas o(s) caso(s) base

Resolvendo o *MergeSort* por Iteração

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\
&= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n \\
&= \ldots \\
&= 2^k T(n/(2^k)) + kn
\end{aligned}
$$

- Lembramos que, no limite, temos que chegar no caso base da recursão, ou seja, $T(1)$
- Para termos a fórmula acima em termos de $T(1)$, $n / (2^k)$ tem que convergir para 1, e isso só acontece se $2^k = 1$, ou seja, $k = \log_2 n$

Resolvendo o *MergeSort* por Iteração

- Temos então

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(n/(2^{\log_2 n})) + (\log_2 n)n \\ &= n^{\log_2 2} T(1) + n\log_2 n \\ &= n + n\log_2 n \end{aligned}$$

- Chegamos à mesma fórmula, mas agora sem “chutar”
- Lembre-se dessas úteis relações envolvendo logaritmos:
  - $\log(ab) = \log a + \log b$
  - $\log a^b = b\log a$
  - $\log_a x = \dfrac{\log_b x}{\log_b a}$
  - $a^{\log_b x} = x^{\log_b a}$

Exemplo Mais Complexo de Iteração

- Vamos tentar resolver a seguinte recorrência por iteração:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 3T(n/4) + n & \text{se } n > 1. \end{cases}$$

- Temos

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= 3(3T(n/16) + n/4) + n = 9T(n/16) + 3n/4 + n \\ &= 9(3T(n/64) + n/16) + 3n/4 + n = 27T(n/64) + 9n/16 + 3n/4 + n \\ &= \ldots \\ &= 3^k T(n/4^k) + 3^{k-1}(n/4^{k-1}) + \ldots + 3^2(n/4^2) + 3(n/4) + n \\ &= 3^k T\left(\frac{n}{4^k}\right) + n\sum_{i=0}^{k-1}\left(\frac{3}{4}\right)^i \end{aligned}$$

Exemplo Mais Complexo de Iteração

- No limite, temos $n/4^k = 1$ e portanto, $k = \log_4 n$. Usando este valor na equação, temos

$$T(n) = 3^{\log_4 n} T(n/4^{\log_4 n}) + n \sum_{i=0}^{(\log_4 n)-1} (3/4)^i$$

$$= n^{\log_4 3} + n \sum_{i=0}^{(\log_4 n)-1} (3/4)^i$$

- Lembrando a fórmula para a soma de termos de uma PG:

$$\sum_{i=0}^{m} x^i = \frac{x^{m+1} - 1}{x - 1}$$

Exemplo Mais Complexo de Iteração

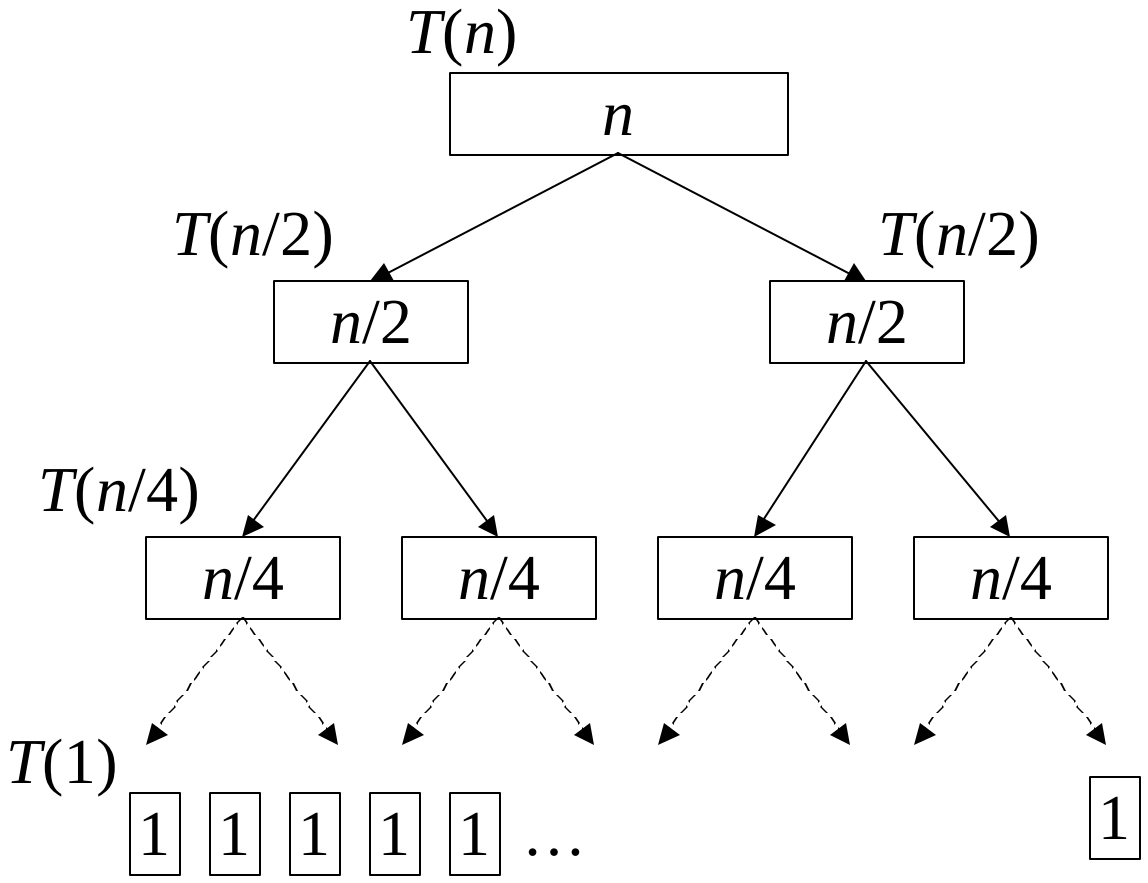
- Temos finalmente

$$
\begin{aligned}
T(n) &= n^{\log_4 3} + n\frac{(3/4)^{\log_4 n} - 1}{-1/4} \\
&= n^{\log_4 3} + n\frac{n^{\log_4 (3/4)} - 1}{-1/4} \\
&= n^{\log_4 3} - 4n(n^{(\log_4 3)-1} - 1) \\
&= n^{\log_4 3} - 4n^{\log_4 3} + 4n \\
&= 4n - 3n^{\log_4 3} \\
&= 4n - 3n^{0.79} \in \Theta(n)
\end{aligned}
$$

Árvores de Recursão

- Maneira gráfica de visualizar a estrutura de chamadas recursivas do algoritmo
- Cada nó da árvore é uma instância (chamada recursiva)
- Se uma instância chama outras, estas são representadas como nós-filhos
- Cada nó é rotulado com o tempo gasto apenas nas operações locais (sem contar as chamadas recursivas)
- Exemplo: *MergeSort*

Árvore de Recursão do *MergeSort*



$= n$

$+$

$2(n/2) = n$

$+$

$4(n/4) = n$

$+$

$n\,(n/n) = n$

$n\,(\log_2 n + 1)$

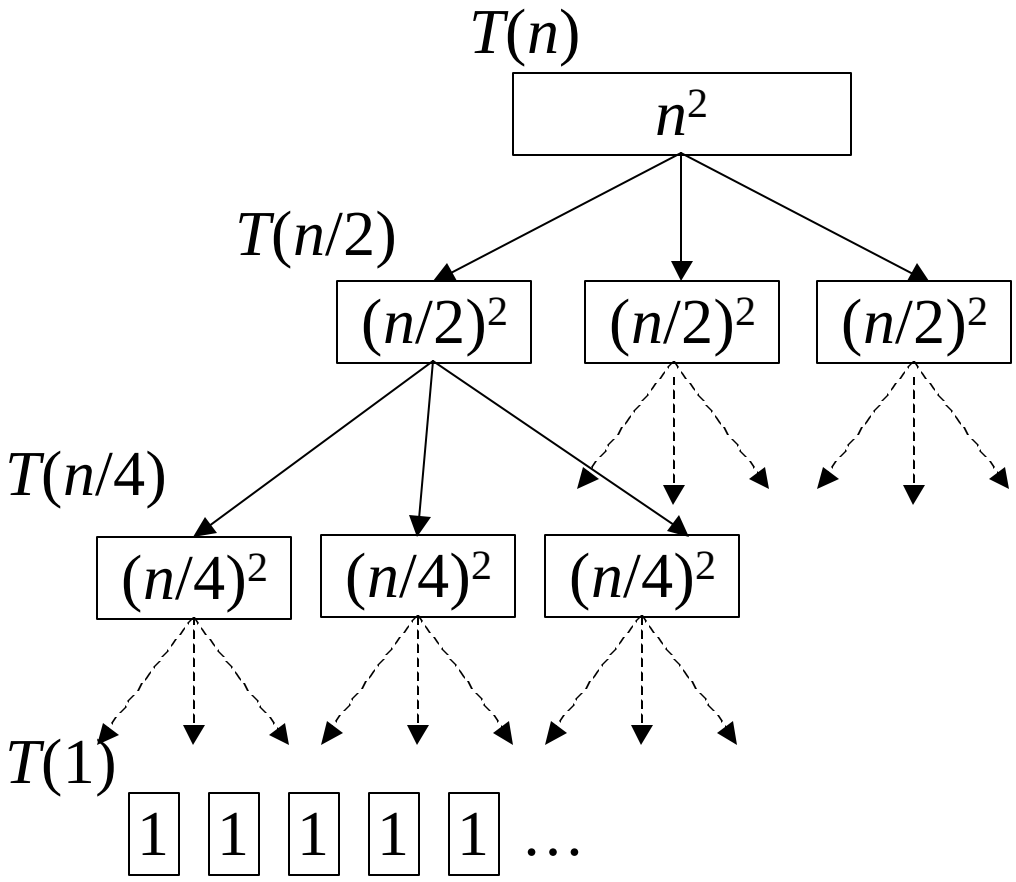
Árvore de Recursão do *MergeSort*

- Observamos que cada nível de recursão efetua no total n passos
- Como há $\log_2 n + 1$ níveis de recursão, o tempo total é dado por $n \log_2 n + n$, o mesmo que encontramos na solução por iteração

- Outro exemplo: considere a recorrência dada por

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 3T(n/2) + n^2 & \text{se } n > 1. \end{cases}$$

Árvore de Recursão – Outro exemplo



n^2

$+$

$3(n/2)^2 = n^2(3/4)$

$+$

$9(n/4)^2 = n^2(3/4)^2$

$+$

$\dots$

$n^2(3/4)^k$

Árvore de Recursão – Outro exemplo

- Vemos que a soma das complexidades locais pelos $k+1$ níveis da recursão nos dá

$$T(n) = n^2 + \sum_{i=0}^{k} (3/4)^i$$

- Na verdade, a complexidade assintótica do algoritmo já pode ser estabelecida aqui como sendo $\Theta(n^2)$ uma vez que sabemos que o somatório acima converge para uma constante finita mesmo que k tenda a infinito
- Se quisermos uma fórmula mais exata, podemos observar (pelo mesmo raciocínio usado na recursão do *MergeSort*) que $k=\log_2 n$. Aplicando a fórmula da soma de termos de uma PG obtemos

$$T(n) = 4n^2 - 3n^{\log_2 3}$$

O Teorema Mestre (Simplificado)

- Podemos observar que as fórmulas de recorrência provenientes de algoritmos do tipo *Dividir-para-Conquistar* são muito semelhantes
- Tais algoritmos tendem a dividir o problema em *a* partes iguais, cada uma de tamanho *b* vezes menor que o problema original
- Quando, além disso, o trabalho executado em cada instância da recursão é uma potência de *n*, existe um teorema que nos dá diretamente a complexidade assintótica do algoritmo
- Em outras palavras, o *Teorema Mestre* pode resolver recorrências cujo caso geral é da forma $T(n)=a\ T\ (n/b)\ +\ n^k$

Teorema Mestre Simplificado

- Dadas as constantes $a \geq 1$ e $b \geq 1$ e uma recorrência da forma $T(n)=a\ T\ (n/b) + n^k$, então,
  - **Caso 1:** se $a > b^k$ então $T(n) \in \Theta(n^{\log_b a})$
  - **Caso 2:** se $a = b^k$ então $T(n) \in \Theta(n^k \log n)$
  - **Caso 3:** se $a < b^k$ então $T(n) \in \Theta(n^k)$
- (Como antes, assumimos que n é uma potência de b e que o caso base $T(1)$ tem complexidade constante)

Teorema Mestre Simplificado

- Exemplos:
  - MergeSort: $T(n)=2T(n/2)+ n$
    - $a=2$, $b=2$, $k=1$.
    - **caso 2** se aplica e $T(n) \in \Theta(n \log n)$
  - $T(n)=3T(n/2)+ n^2$
    - $a=3$, $b=2$, $k=2$
    - **caso 3** se aplica ($3<2^2$) e $T(n) \in \Theta(n^2)$
  - $T(n)=2T(n/2)+ n \log n$
    - Teorema mestre (simplificado ou completo) não se aplica
    - Pode ser resolvida por iteração

Teorema Mestre e Árvores de Recursão

- Os três casos do teorema mestre podem ser entendidos como as três maneiras de distribuir o trabalho na árvore de recursão:
  - Caso 1: O trabalho é concentrado nas folhas (ex.: $T(n)=4T(n/3)+ n$)
  - Caso 2: O trabalho é dividido igualmente entre todos os níveis (ex.: MergeSort)
  - Caso 3: O trabalho é concentrado nas raízes (ex.: $3T(n/2)+ n^2$)

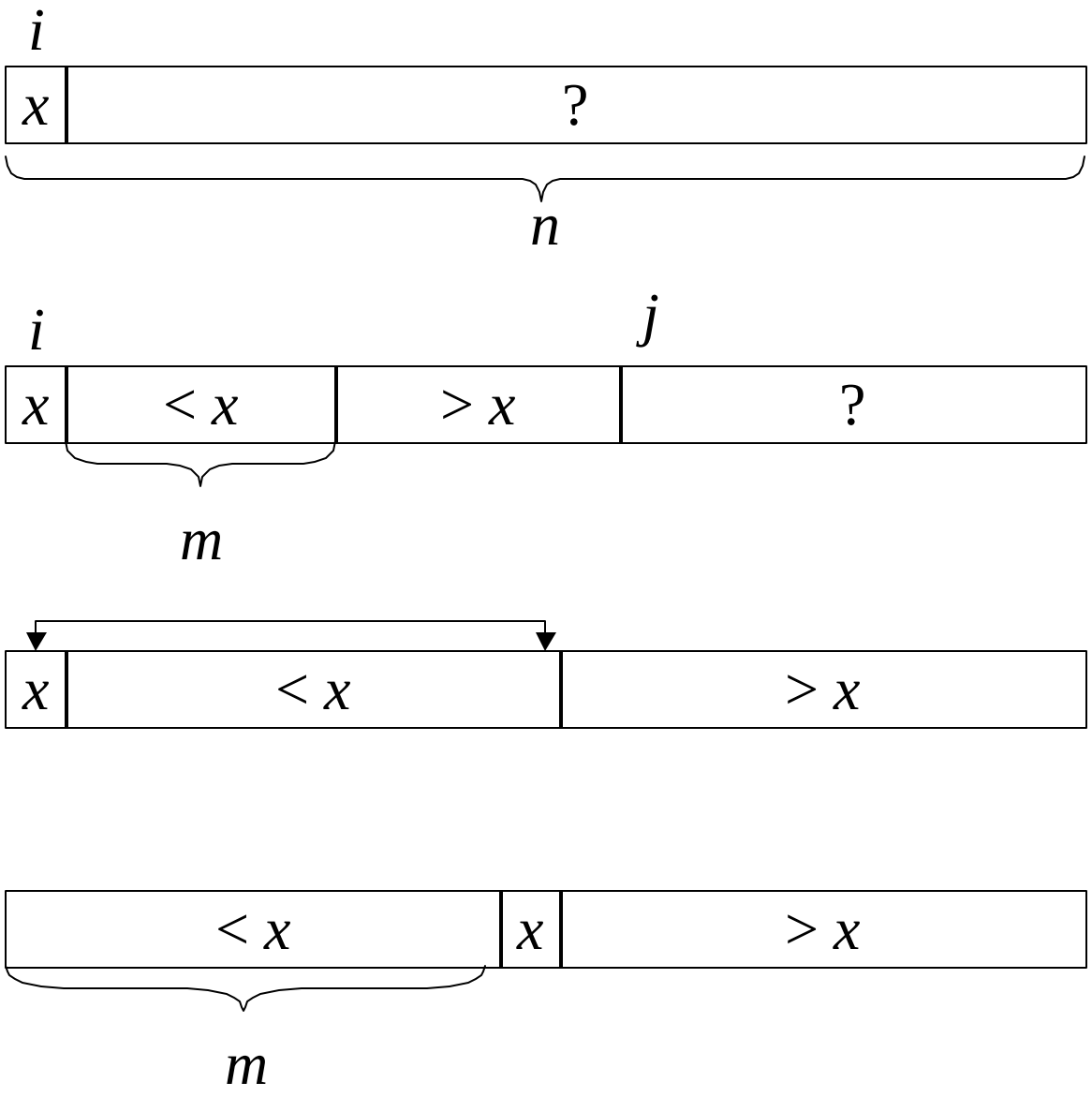
QuickSort

- O QuickSort é provavelmente o algoritmo mais usado na prática para ordenar arrays
- Sua complexidade de caso médio é $\Theta(n \log n)$
- Sua complexidade de pior caso é $\Theta(n^2)$ mas a probabilidade do pior caso acontecer fica menor à medida que n cresce (caindo para 0 à medida que n tende a infinito)
- O passo crucial do algoritmo é escolher um elemento do array para servir de pivô
- O QuickSort pode se tornar um algoritmo de complexidade de pior caso $\Theta(n \log n)$ se escolhermos sempre a mediana como pivô. Usando um algoritmo que seleciona a mediana dos elementos em $\Theta(n)$, mas na prática o algoritmo acaba tendo um desempenho ruim

QuickSort: Partição

- O QuickSort utiliza como base um algoritmo de *particionamento*
- Particionar um array em torno de um pivô *x* significa dividi-lo em três segmentos contíguos contendo, respectivamente, todos os elementos menores que *x*, iguais a *x*, e maiores que *x*.
- Vamos definir uma função *partição* que divide um array *A* de *n* elementos indexados a partir do índice *i*, isto é, $A[i], A[i+1] \ldots A[i + n - 1]$
- Como pivô, escolheremos *x* = valor inicial de $A[i]$
- A rotina retorna *m*, o número de elementos menores que *x*
- Assumimos também que o array tem todos os seus elementos distintos, logo, ao final da chamada, os segmentos são:
  - $A[i] \ldots A[i + m - 1]$
  - $A[i + m]$
  - $A[i + m + 1] \ldots A[i + n - 1]$

QuickSort : Partição



QuickSort: Partição

```
proc partição (i, n, A [i .. i + n – 1]) {
    m ← 0
    para j desde i+1 até i + n – 1 fazer {
        se A [j] < A [i] então {
            m ← m + 1
            trocar A [j] com A [i+m]
        }
    }
    trocar A [i] com A [i+m]
    retornar m
}
```

QuickSort

- É fácil ver que o algoritmo *partição* tem complexidade $\Theta(n)$
- Se quisermos escolher outro elemento como pivô, k digamos, basta trocar $A[i]$ com $A[k]$ antes de chamar *partição*
- O QuickSort funciona particionando o array recursivamente até que todos os segmentos tenham tamanho <= 1
- Na prática, utiliza-se um pivô randômico. Prova-se que isso garante complexidade de caso médio $\Theta(n \log n)$ para o QuickSort
- Diz-se que o quicksort é um *algoritmo de Las Vegas*, isto é, sua complexidade é uma variável randômica. A complexidade desses algoritmos é determinada fazendo-se uma média de todas as escolhas possíveis

QuickSort

```
proc QuickSort (i, n, A [i .. i + n – 1]) {
    se n > 1 então {
        escolha um valor k entre i e i + n – 1
        trocar A [k] com A [i]
        m ← particao (i, n, A)
        QuickSort (i, m, A)
        QuickSort (i+m+1, n–m–1, A)
    }
}
```

Análise do QuickSort

- A análise de pior caso do algorimo resulta na recorrência

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1, \\ \max\limits_{m=0..n-1} \left(T(m) + T(n-m-1) + n \right) & \text{se } n > 1. \end{cases}$$

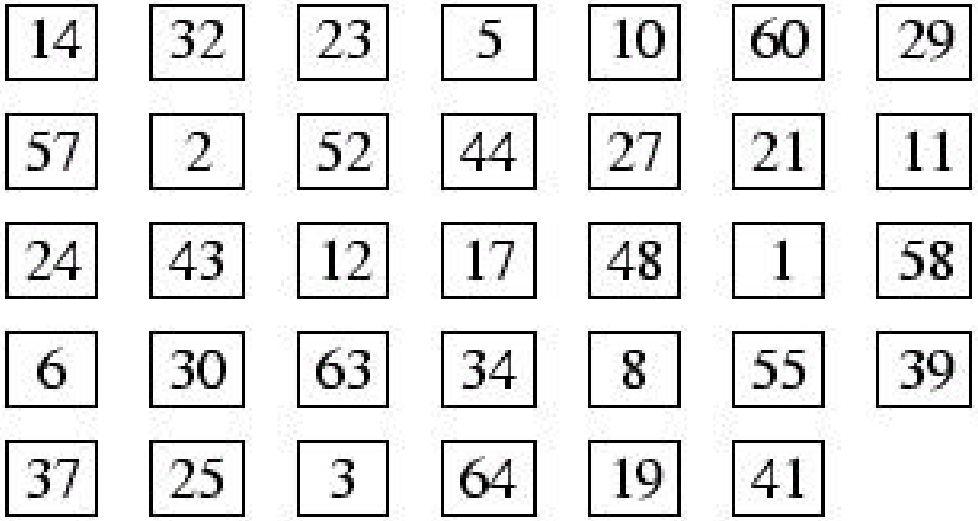
- A pergunta da vez é: qual valor de *m* maximiza *T*(n)?
  - Nesse caso, a resposta é: valores de *m* iguais a 0 ou $n - 1$ fazem $T(n) \in \Theta(n^2)$
  - A argumentação formal é complicada, mas podemos observar que se ao invés de valores extremos, escolhermos um valor médio de *m* minimizaremos T(n)
  - Em particular, se pudermos sempre obter $m = n/2$, teremos a fórmula de recorrência do MergeSort que resulta em $T(n) \in \Theta(n \log n)$

Análise do QuickSort – Usando a Mediana

- Escolher sempre um pivô que resulta em $m=n/2$ implica em termos um algoritmo para calcular a mediana
- Obviamente, só podemos gastar tempo $O(n)$ para fazer essa escolha
- De fato, existe um algoritmo que permite calcular a mediana de um array (ou, na verdade, o k-ésimo maior elemento do array) em $O(n)$
- Este algoritmo também utiliza a técnica do pivô
- Na verdade, prova-se que esse algoritmo tem complexidade $O(n)$ se pudermos sempre escolher um pivô de forma a garantir que $cn \leq m \leq (1–c)\ n$ para alguma constante c
- Existe um algoritmo (bastante enrolado) para fazer essa escolha do pivô que garante $n/4 \leq m \leq 3n/4$

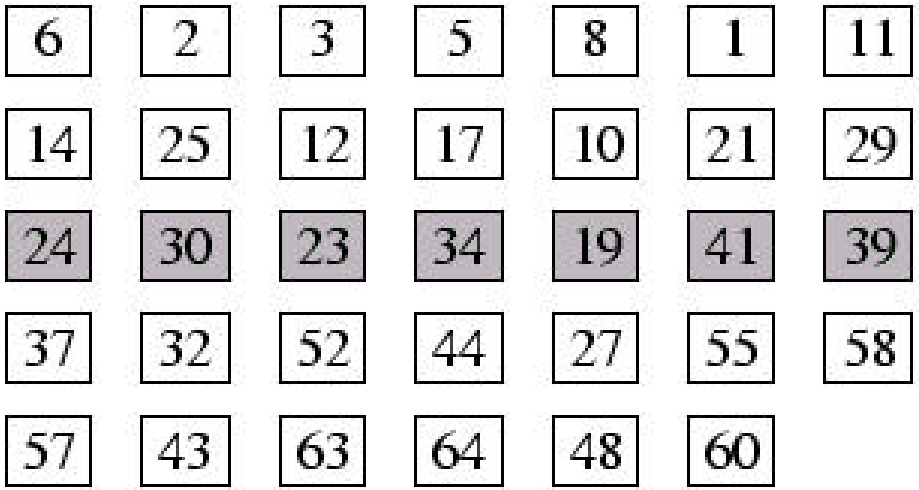
Mediana – Algoritmo para escolher Pivô

- 1º passo: Dividir o array em grupos de 5



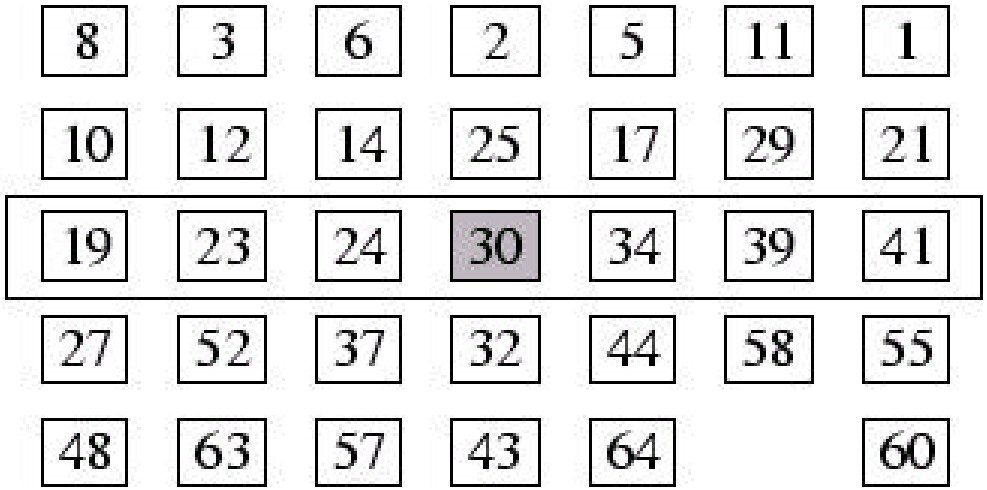
Mediana – Algoritmo para escolher Pivô

- 2º passo: Computar as medianas dos grupos de 5
  - Cada mediana pode ser computada em $O(1)$



Mediana – Algoritmo para escolher Pivô

- 3º passo: Computar a mediana das medianas
  - OBS.: Para tanto, chamamos recursivamente o algoritmo das medianas, que roda em $O(n)$



QuickSort – Considerações Finais

- É comum limitar a recursão do QuickSort empregando um algoritmo mais simples para *n* pequeno
- As implementações de QuickSort com pivô randômico são as mais usadas na prática pois as arquiteturas modernas de computadores executam mais rápido códigos que têm localidade de referência
  - As comparações são sempre entre o pivô (que pode ser guardado num registrador) e posições consecutivas do array,
- O MergeSort também tem boa localidade de referência, mas precisa de um array auxiliar
- O HeapSort não precisa de um array auxiliar mas tem péssima localidade de referência

Listas

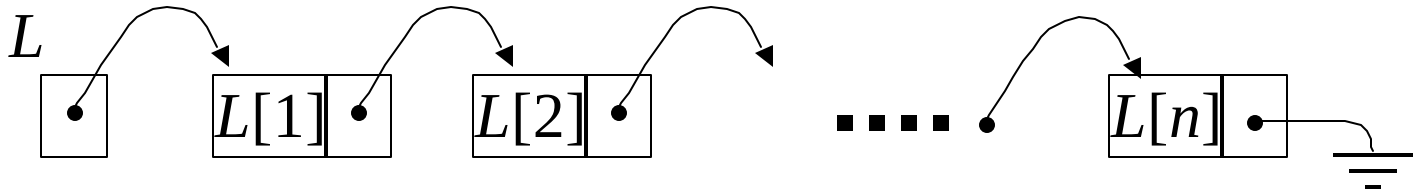
- Uma lista é um arranjo seqüencial de elementos
- Dada uma lista *L*, as seguintes operações são típicas
  - Encontrar o primeiro elemento de *L*
  - Dado um elemento de *L*, encontrar o próximo ou o anterior
  - Encontrar o k-ésimo elemento da lista
- Arrays, são listas onde os elementos são de mesmo tamanho e armazenados contiguamente na memória do computador
- Vamos agora considerar listas *encadeadas*, que não têm as principais restrições dos arrays:
  - É possível inserir ou remover elementos em $O(1)$
  - O problema de overflow não existe, ou melhor, acontece apenas quando a memória do computador se esgota

Listas Encadeadas

- Listas encadeadas também têm desvantagens:
  - Utilizam mais memória que arrays
  - Acesso direto - i.e., $O(1)$ - ao k-ésimo elemento não é possível
- Existem muitas maneiras de implementar listas encadeadas
  - Listas *a la* LISP
  - Listas com ponteiros
- Um assunto muito intimamente ligado a estruturas encadeadas é o problema de alocação de memória

Listas Encadeadas

- Elementos são armazenados na memória em endereços arbitrários
- Ordem seqüencial entre os elementos é armazenada explícitamente (elos, ponteiros, *links*)
- Deve ser possível determinar o elo correspondente a cada elemento (armazenamento consecutivo / 2 elos)
- A lista propriamente só pode ser acessada sabendo-se o endereço do seu primeiro elemento
- Deve haver alguma maneira de determinar o comprimento da lista (elo nulo, comprimento armazenado)



Listas Encadeadas

- Vamos assumir inicialmente:
  - Cada nó (par elemento/elo) é armazenado em posições contíguas de memória
  - Usamos um elo nulo para indicar o fim da lista
  - Uma lista é referida por um elo que leva ao primeiro nó da lista
- Sendo assim, vamos propositalmente confundir o conceito de elo e lista e definir lista da seguinte forma:
  - Uma *lista* é
    - Uma *lista nula* ou
    - Um par *elemento/lista*

Busca Seqüencial em Listas Encadeadas

- Vamos usar a seguinte notação:
  - *Nulo* : elo (lista) nulo
  - *L^* : denota primeiro nó da lista *L*. Definido apenas se *L* não é nula
  - *No.Elemento*: denota o elemento armazenado em *No*
  - *No.Elo*: denota o elo armazenado em No

```
proc Busca (Lista L, Valor v) {
    se L = Nulo então
        retornar falso
    senão
        se L^.Elemento = v então
            retornar verdadeiro
        senão
            retornar Busca (L^.Elo, v)
}
```

Busca Seqüencial em Listas

- (Versão não recursiva)

```
proc Busca (Lista L, Valor v) {
    tmp ← L
    enquanto tmp ≠ Nulo fazer {
        se tmp^.Elemento = v então
            retornar verdadeiro
        senão
            tmp ← tmp^.Elo
    }
    retornar falso
}
```

Alocação de Memória

- É preciso haver algum mecanismo que permita gerenciar a memória livre
- Quando se quer alocar um nó, requisita-se uma área contígua de memória livre suficientemente grande
  - *Aloca* (*Tipo*) retorna um elo para uma área de memória grande suficiente para guardar uma estrutura de dados do tipo *Tipo*
  - Ex.: *Aloca* (NoLista) retorna um elo para um nó de lista, isto é, uma *Lista*
- Quando uma área de memória está mais em uso, ela é retornada ao gerenciador para ser reutilizada posteriormente
  - *Libera*(*Elo*) retorna a área de memória contígua apontada por *Elo*

Criando uma Lista

- Para criar uma lista com um único elemento igual a *v*

proc *CriaListaUnária* (*Valor v*) {
    L ← *Aloca* (*NoLista*)
    L^.*Elemento* ← *v*
    L^.*Elo* ← *Nulo*
    retornar *L*
}

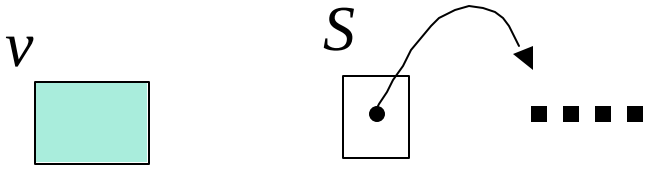
- Para criar uma lista com um elemento igual a *v* à frente de uma lista *S*

proc *CriaNoLista* (*Valor v*, *Lista S*) {
    L ← *Aloca* (*NoLista*)
    L^.*Elemento* ← *v*
    L^.*Elo* ← *S*
    retornar *L*
}

Criando Listas

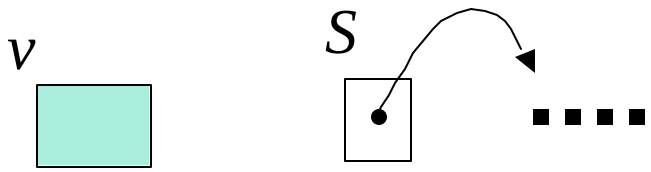
Criando Listas

CriaNoLista (*v*, *S*)

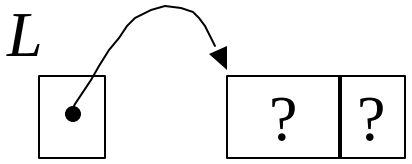


Criando Listas

CriaNoLista (*v*, *S*)

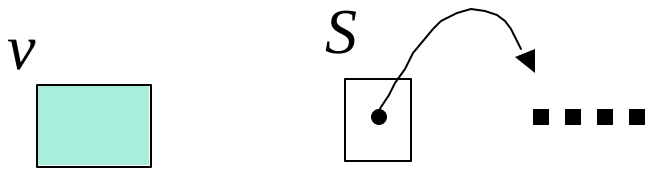


$L \leftarrow$ *Aloca* (*NoLista*)

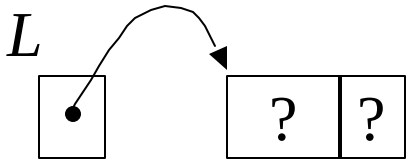


Criando Listas

CriaNoLista (*v*, *S*)

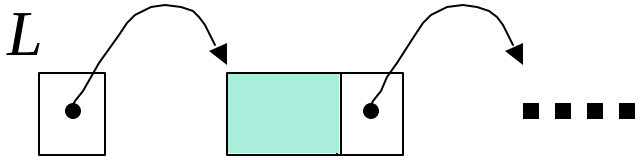


L ← *Aloca* (*NoLista*)



L^.*Elemento* ← *v*

L^.*Elo* ← *S*



Destruindo Listas

- Para destruir o primeiro nó de uma lista

```
proc DestroiNoLista (var Lista L ) {
    tmp ← L
    L ← L^.Elo
    Libera (tmp)
}
```

- Para destruir uma lista inteira

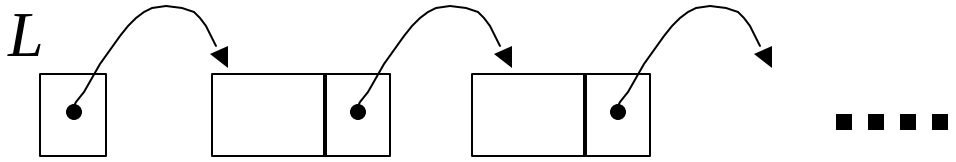
```
proc DestroiLista (var Lista L) {
    enquanto L ≠ Nulo fazer
        DestroiNoLista (L)
}
```

- (Note que em ambas rotinas, *L* é um parâmetro variável, isto é, passado por referência)

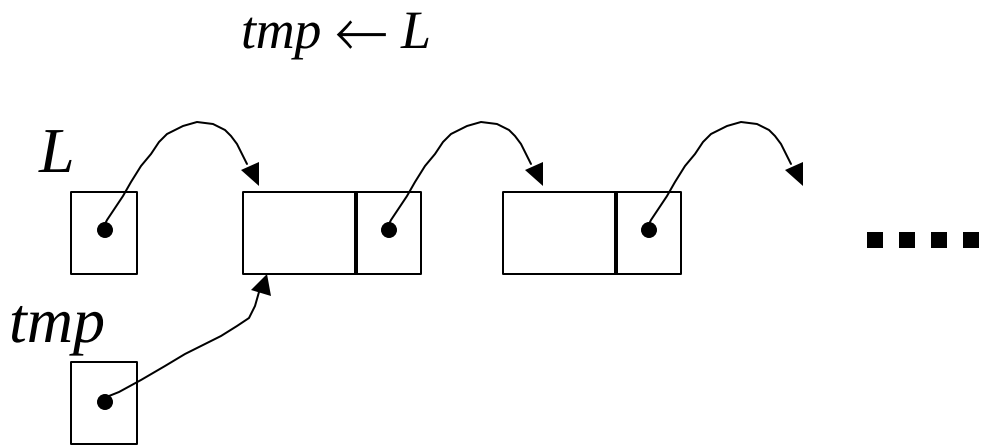
Destruindo Listas

Destruindo Listas

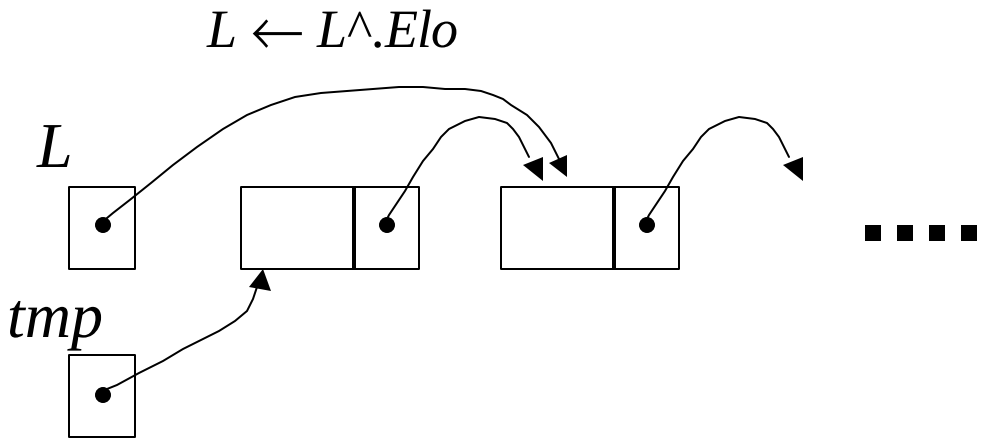
DestroiNoLista (*L*)



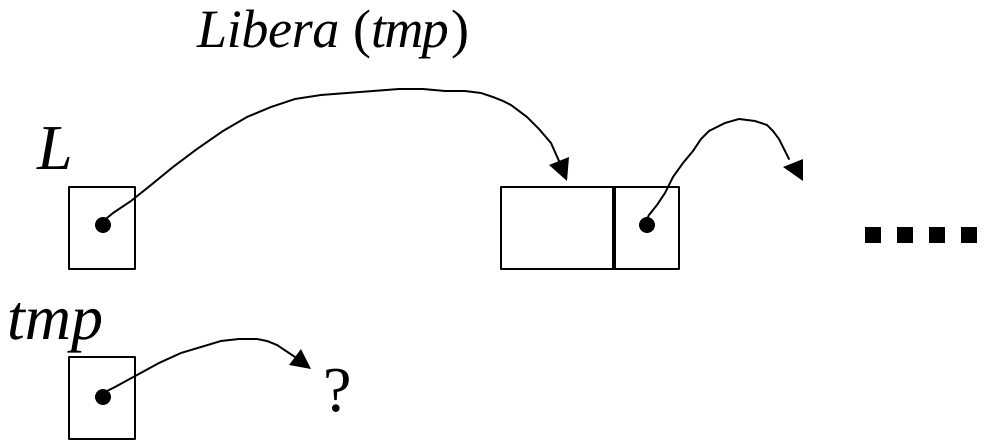
Destruindo Listas



Destruindo Listas



Destruindo Listas



Inserção e Remoção

- Todos os procedimentos de inserção e remoção de elementos de uma lista encadeada podem ser escritos com o auxílio das rotinas *CriaNoLista* e *DestroiNoLista*
- Rotina para inserir um elemento *v* numa lista ordenada *L*

```
proc InsereListaOrdenada (Valor v, var Lista L) {
    se L = Nulo então
        L ← CriaListaUnária (v)
    senão
        se L^.Elemento > v então
            L ← CriaNoLista (v, L)
        senão
            InsereListaOrdenada (v, L^.Elo)
}
```

Inserção e Remoção

- Rotina para remover um elemento igual a *v* de uma lista ordenada *L*

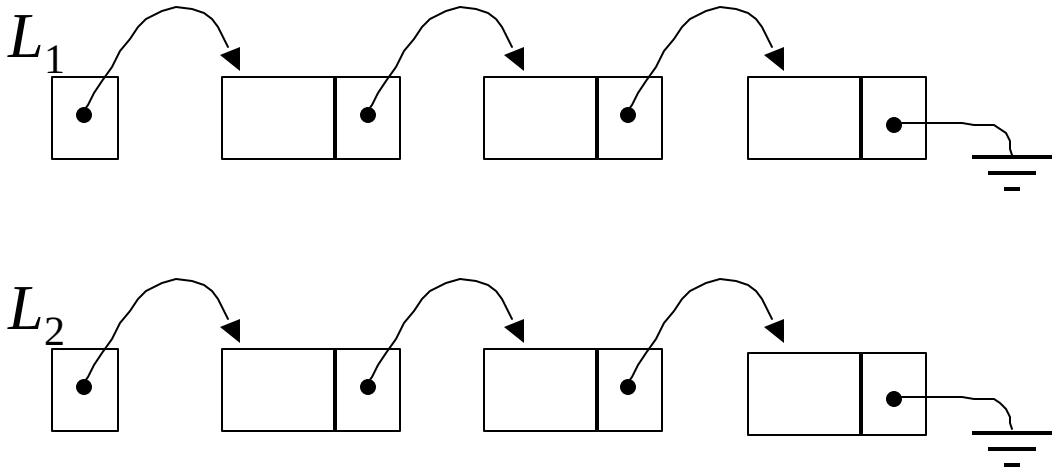
```
proc RemoveListaOrdenada (Valor v, var Lista L) {
    se L ≠ Nulo então
        se L^.Elemento = v então
            DestroiNoLista (L)
        senão
            RemoveListaOrdenada (v, L^.Elo)
}
```

Trabalhando com Ponteiros

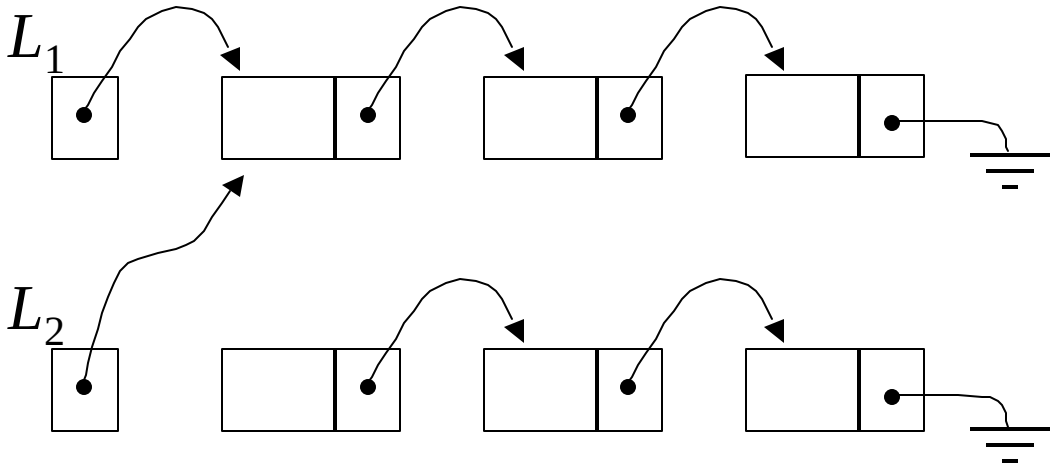
- A utilização de ponteiros para designar elos de uma estrutura encadeada pode levar a uma série de ambigüidades
- Por exemplo, sejam L_1 e L_2 duas listas conforme definidas anteriormente. O que significa a instrução "$L_2 \leftarrow L_1$" ?
  - Duas interpretações são razoáveis:
    1. A lista L_2 recebe uma cópia da lista L_1
    2. L_1 e L_2 passam a designar a mesma lista
  - No entanto, nenhuma das duas é verdadeira na maioria das linguagens de programação!

Trabalhando com Ponteiros

Trabalhando com Ponteiros

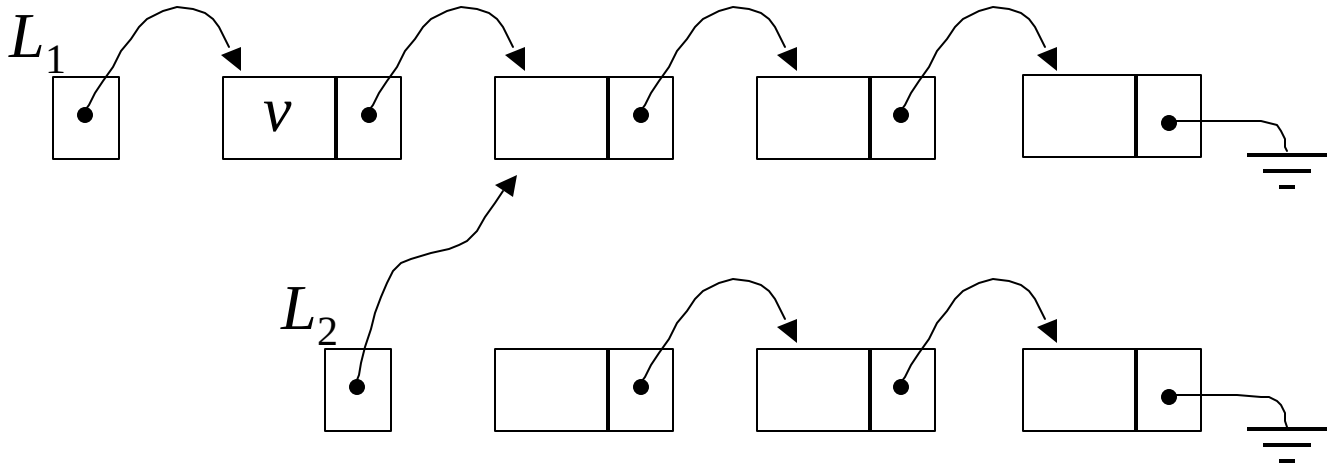


Trabalhando com Ponteiros



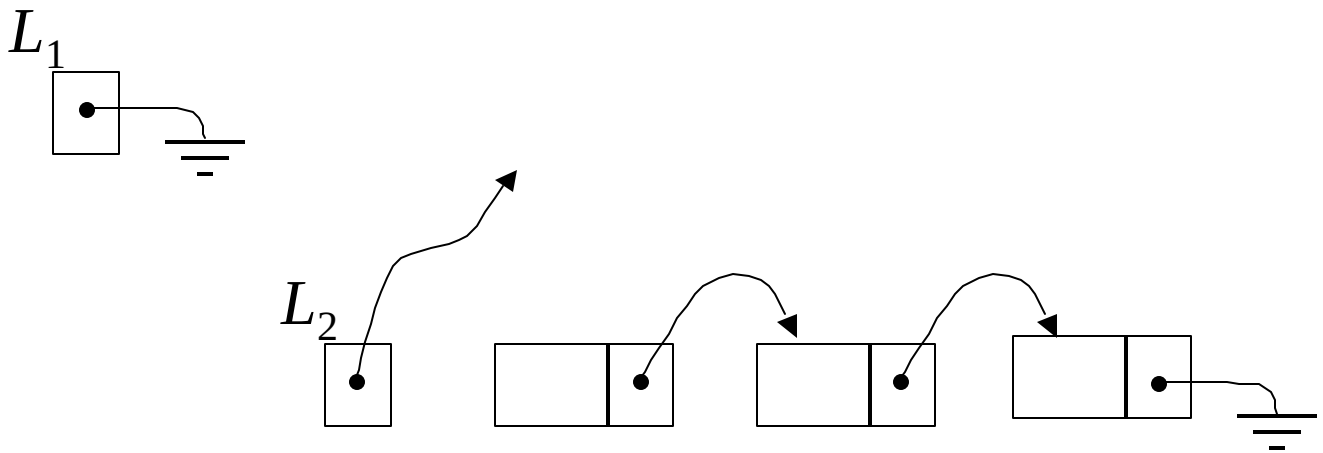
$L_2 \leftarrow L_1$

Trabalhando com Ponteiros



$L_1 \leftarrow CriaNoLista\ (v, L_1)$

Trabalhando com Ponteiros



DestroiLista (L_1)

Problemas Comuns com Ponteiros

- Ponteiro no vazio (*dangling pointer*)
  - Um ponteiro acaba apontando para uma área de memória não mais sob controle do programa, ou
  - Um ponteiro acaba apontando para uma área qualquer de memória do programa sem que o programador se dê conta
- Vazamento de memória (*memory leak*)
  - Uma área de memória alocada para o programa é “esquecida” por um programa
  - Em alguns casos, se o procedimento é repetido muitas vezes, o programa acaba falhando por falta de memória

Coleta de Lixo

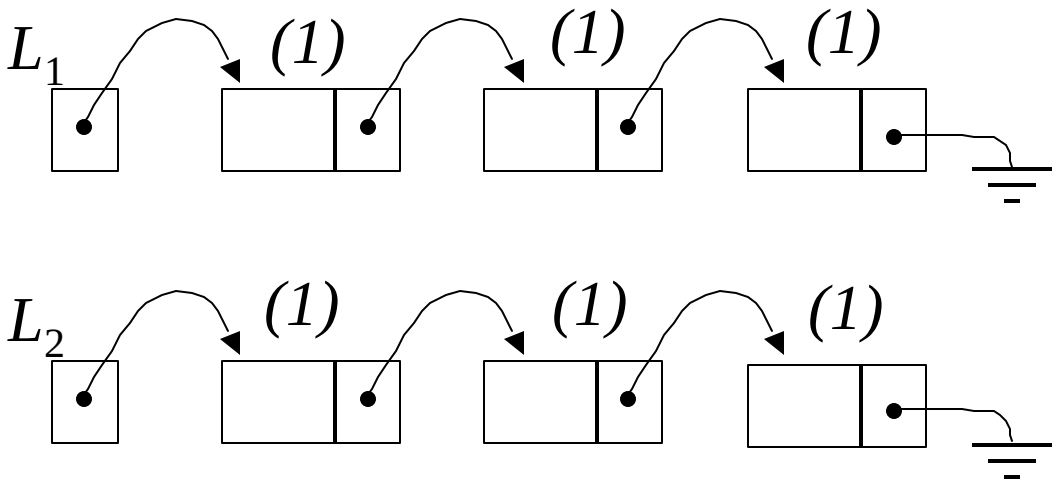
- Muitas linguagens de programação (e.g., Java, LISP, Modula-3) aliviam esses problemas adotando o princípio da “Coleta de Lixo” (*Garbage Collection*)
  - O programador aloca memória explicitamente mas não a libera explicitamente
  - Variáveis que não são mais necessárias (ex. saem de escopo), são entregues a um procedimento automático (coletor de lixo) que se encarrega de devolvê-las ao banco de memória livre
  - O coletor de lixo só devolve a variável ao banco de memória livre se ela não pode mais ser acessada, isto é, se nenhum ponteiro do programa aponta para ela
  - O esquema mais usado para se implementar coletores de lixo é o do contador de referências

Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre

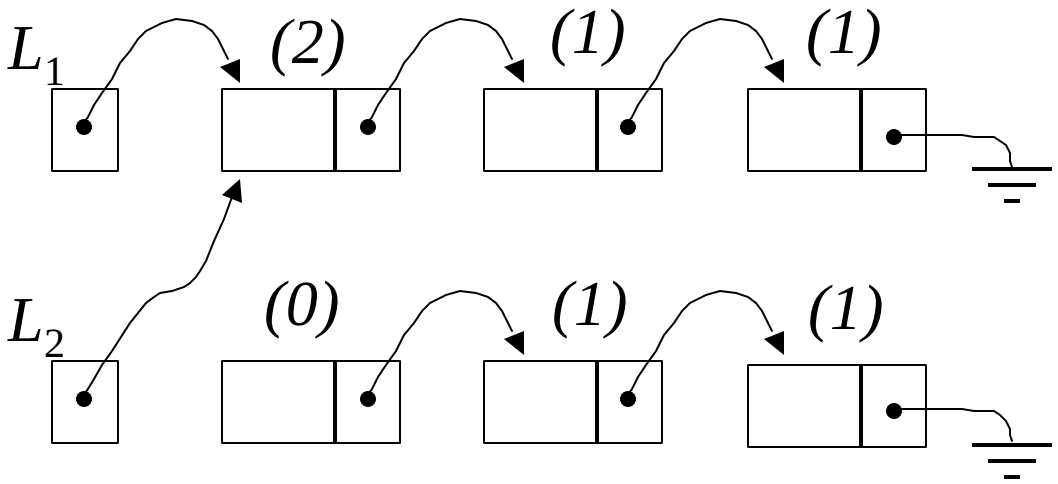
Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



Contador de Referências

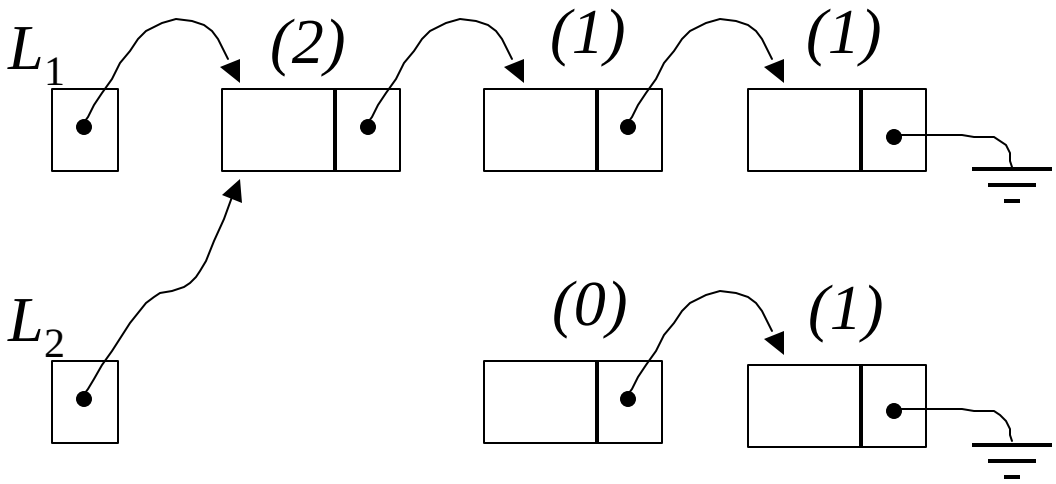
- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



$L_2 \leftarrow L_1$

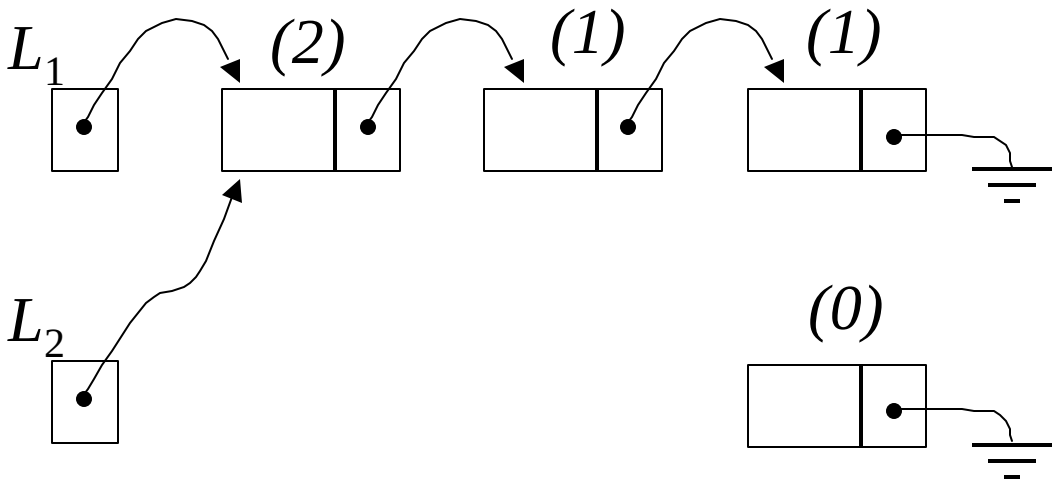
Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



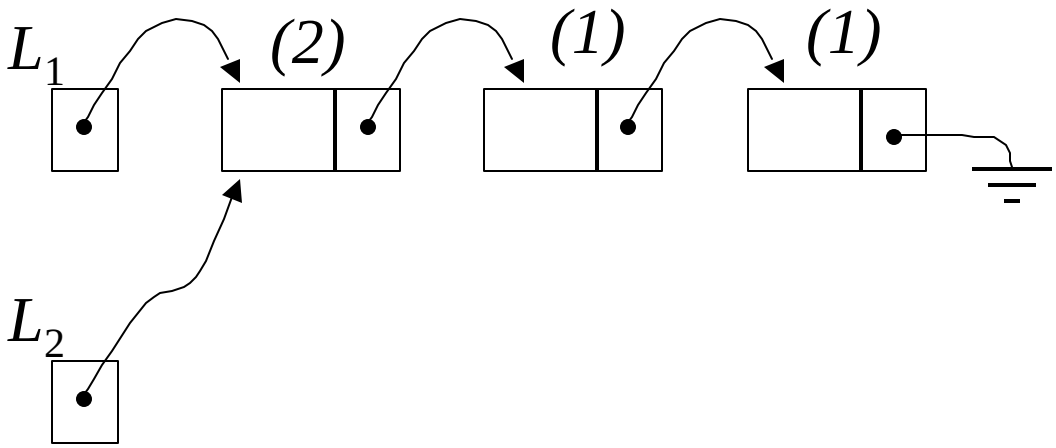
Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



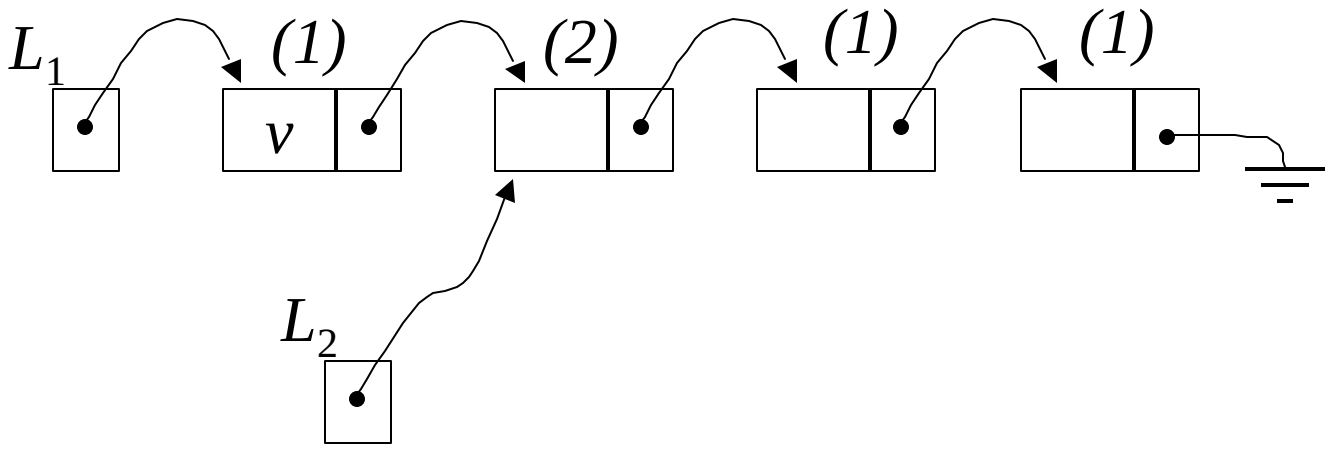
Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



Contador de Referências

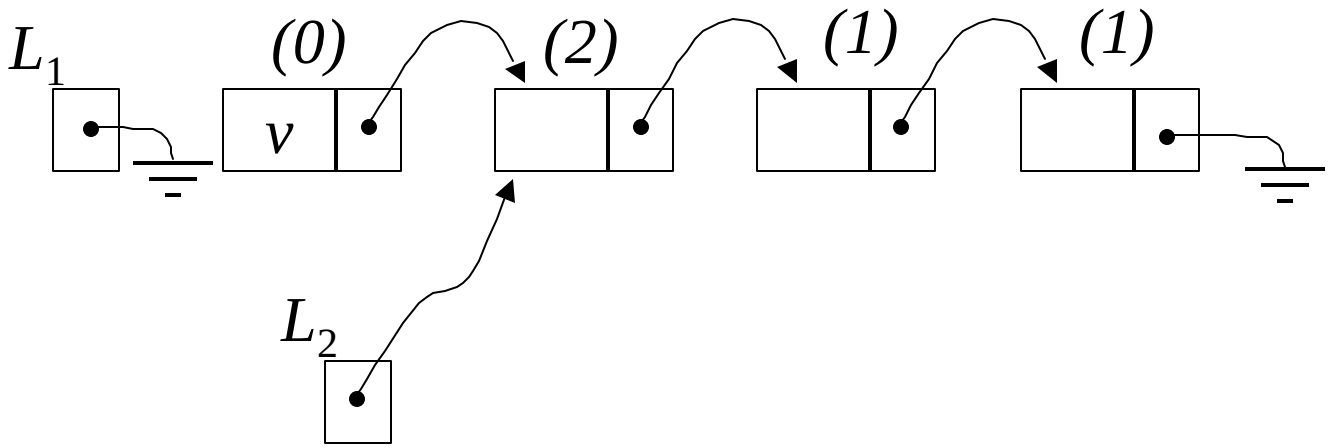
- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



$L_1 \leftarrow CriaNoLista\ (v, L_1)$

Contador de Referências

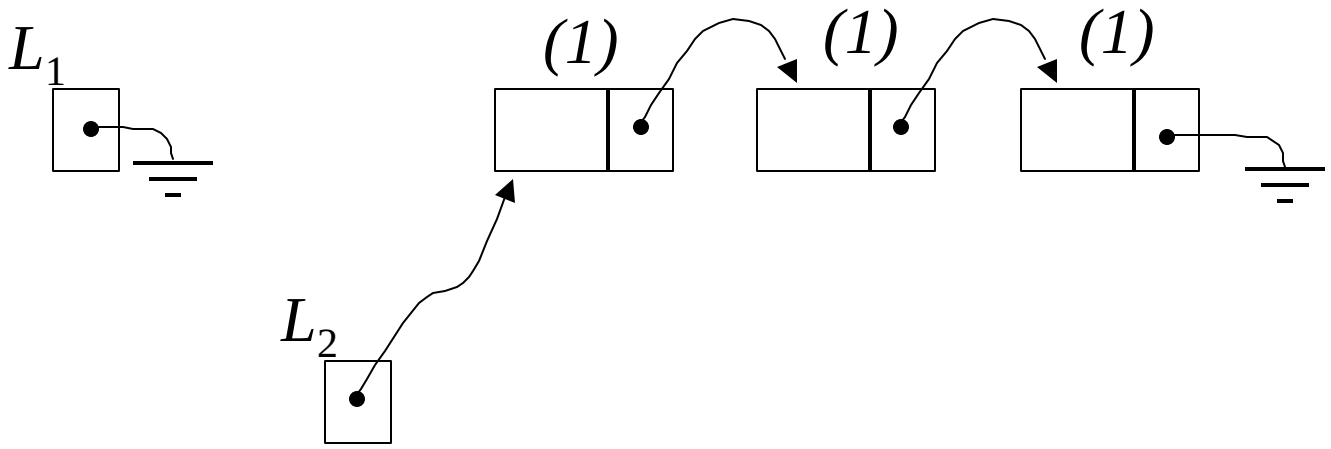
- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



$L_1 \leftarrow Nulo$

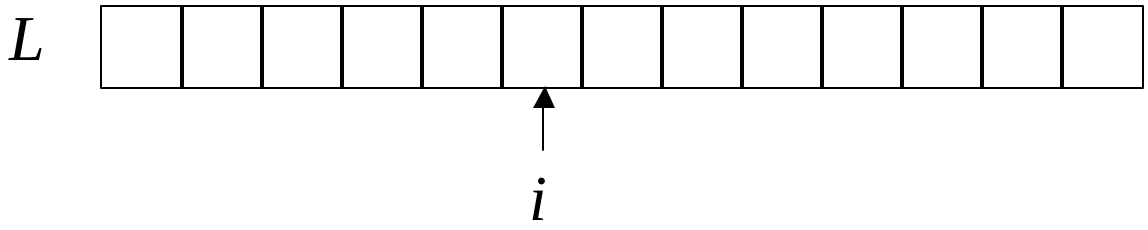
Contador de Referências

- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



Iteradores de Listas

- Quando coleta de lixo não está disponível ou se torna excessivamente custosa, estruturas com ponteiros podem ser manipuladas com menos chance de erro distinguindo-se os seguintes conceitos:
  - variável do tipo lista (ou seja, listas propriamente ditas)
  - ponteiros para nós de lista (ou seja, iteradores de listas)
- (Pense num array!)

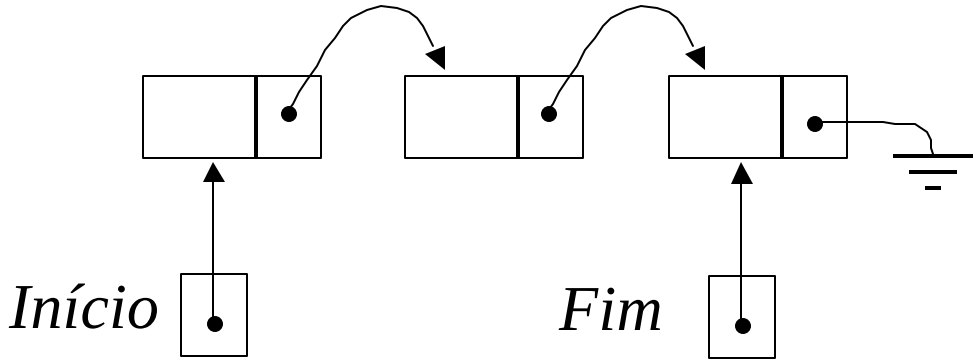


Pilhas, Filas e Deques

- Uma lista encadeada é ideal para ser usada na implentação de pilhas já que inserção e remoção podem ser feitas com naturalidade em $O(1)$ no início da lista
- Para implementar uma fila é necessário manter dois ponteiros: um para o início e outro para o fim da fila

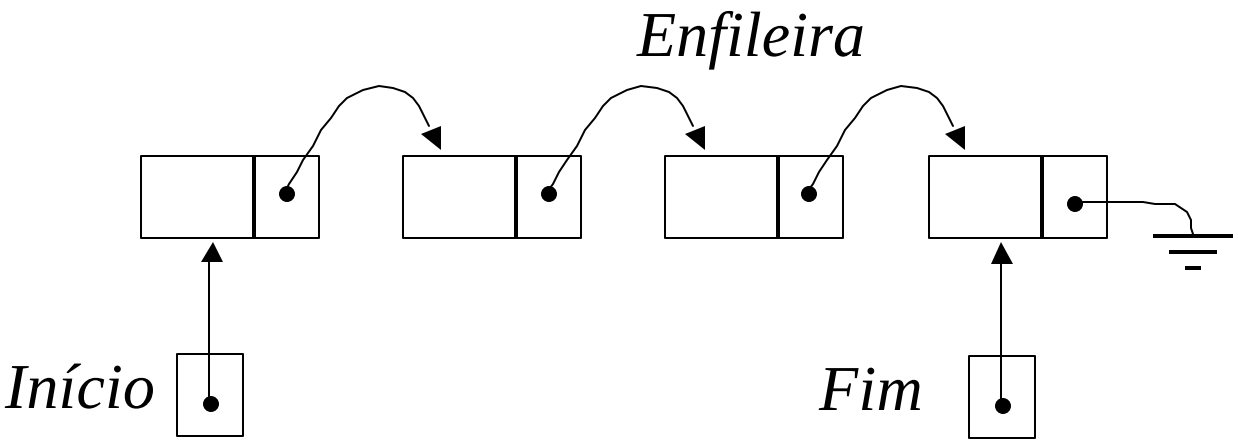
Pilhas, Filas e Deques

- Uma lista encadeada é ideal para ser usada na implentação de pilhas já que inserção e remoção podem ser feitas com naturalidade em $O(1)$ no início da lista
- Para implementar uma fila é necessário manter dois ponteiros: um para o início e outro para o fim da fila



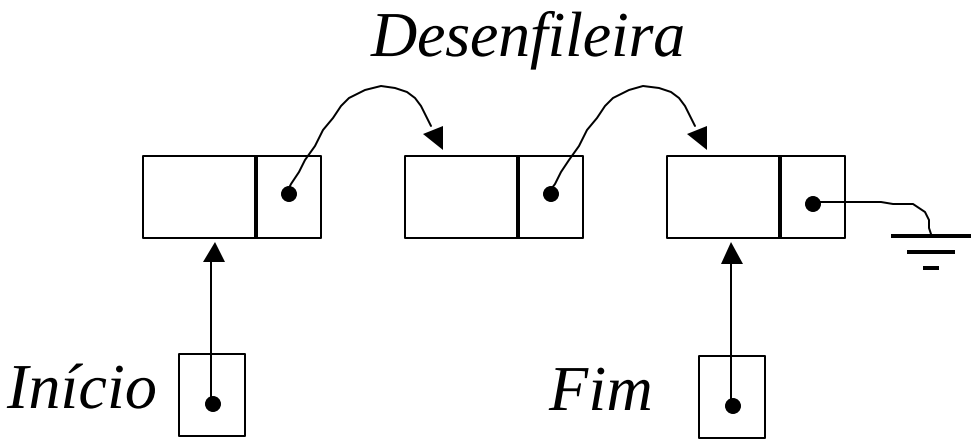
Pilhas, Filas e Deques

- Uma lista encadeada é ideal para ser usada na implentação de pilhas já que inserção e remoção podem ser feitas com naturalidade em $O(1)$ no início da lista
- Para implementar uma fila é necessário manter dois ponteiros: um para o início e outro para o fim da fila



Pilhas, Filas e Deques

- Uma lista encadeada é ideal para ser usada na implentação de pilhas já que inserção e remoção podem ser feitas com naturalidade em $O(1)$ no início da lista
- Para implementar uma fila é necessário manter dois ponteiros: um para o início e outro para o fim da fila

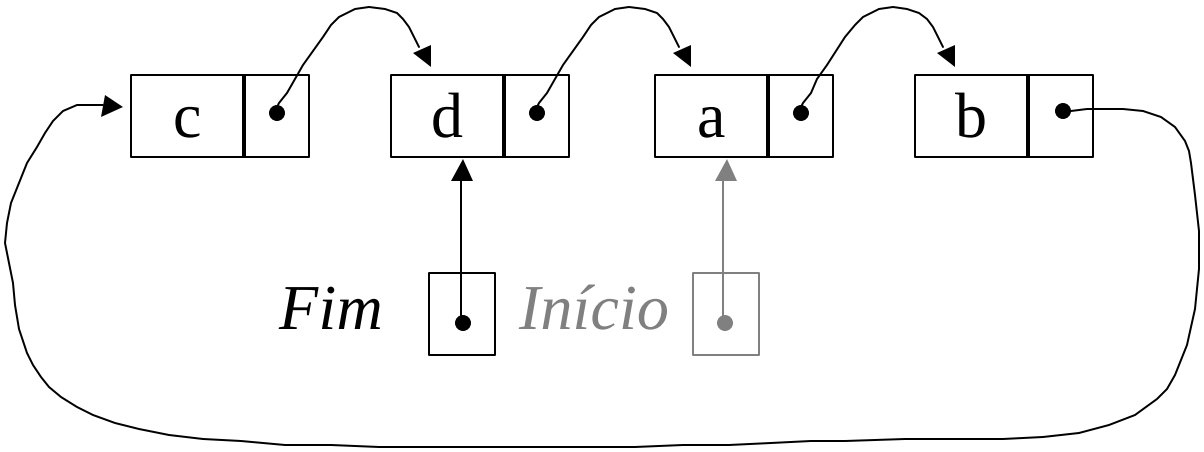


Listas Circulares

- O uso de dois ponteiros tem que ser cuidadoso para contemplar os casos especiais onde a lista tem 0 ou 1 elemento
- Uma solução mais elegante é usar listas circulares
- Neste caso, utiliza-se apenas um ponteiro para o fim da fila e fica implícito que o início da fila é o nó seguinte

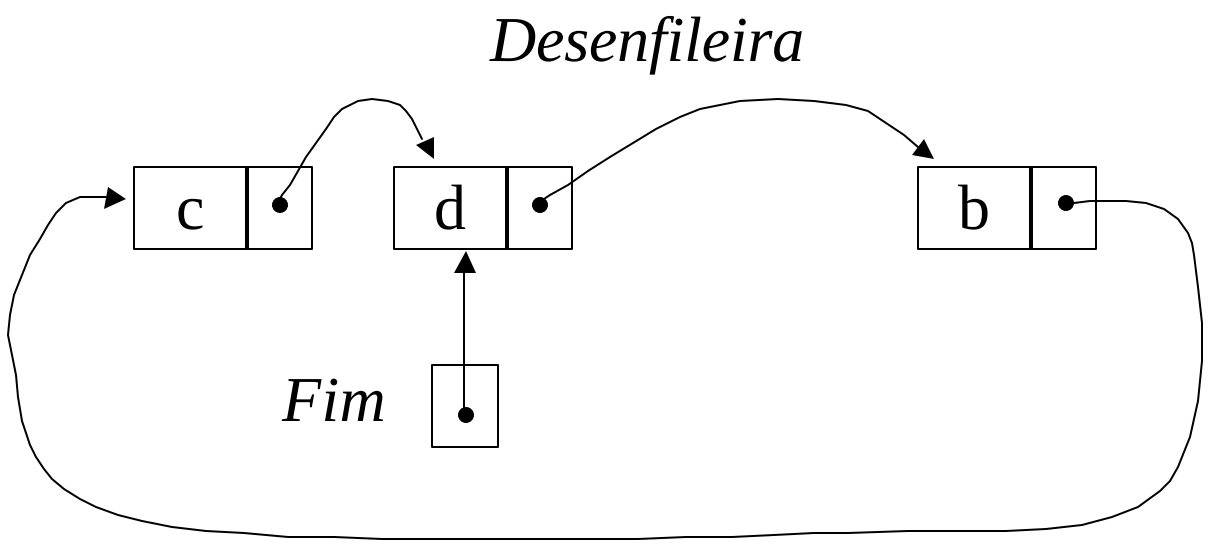
Listas Circulares

- O uso de dois ponteiros tem que ser cuidadoso para contemplar os casos especiais onde a lista tem 0 ou 1 elemento
- Uma solução mais elegante é usar listas circulares
- Neste caso, utiliza-se apenas um ponteiro para o fim da fila e fica implícito que o início da fila é o nó seguinte



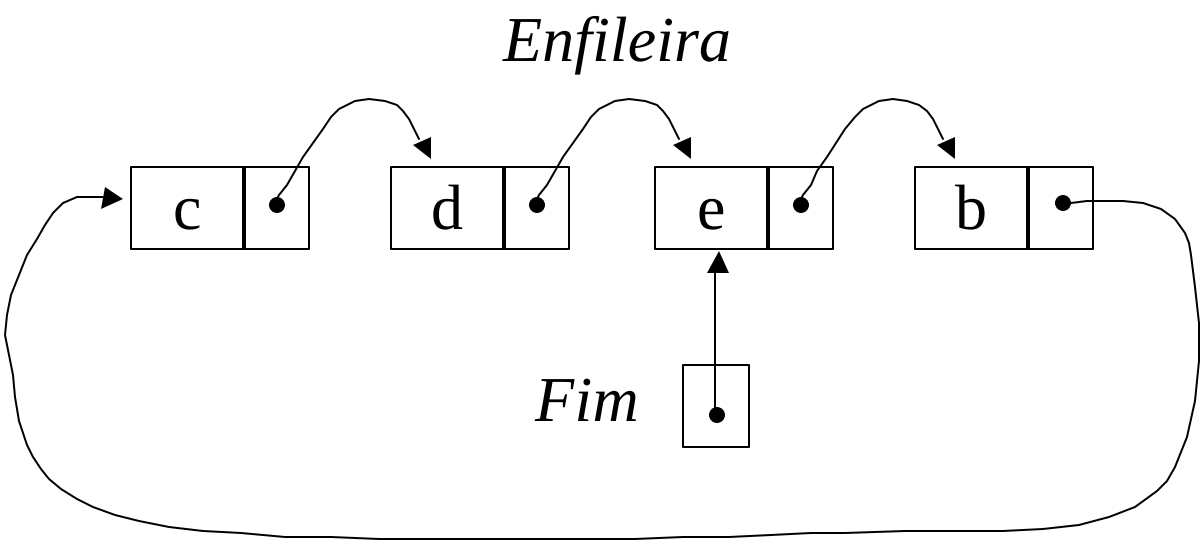
Listas Circulares

- O uso de dois ponteiros tem que ser cuidadoso para contemplar os casos especiais onde a lista tem 0 ou 1 elemento
- Uma solução mais elegante é usar listas circulares
- Neste caso, utiliza-se apenas um ponteiro para o fim da fila e fica implícito que o início da fila é o nó seguinte



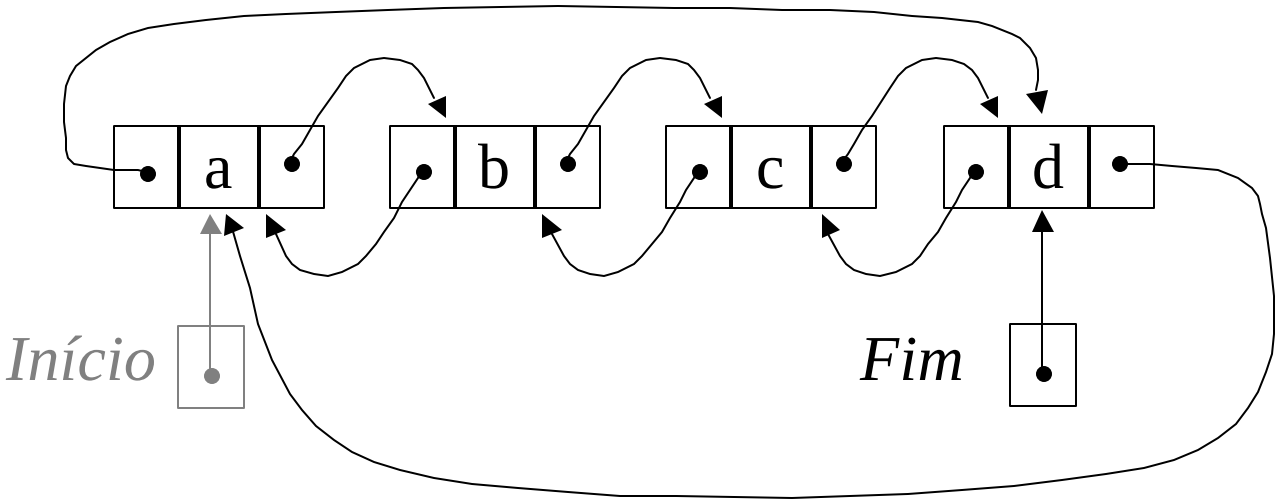
Listas Circulares

- O uso de dois ponteiros tem que ser cuidadoso para contemplar os casos especiais onde a lista tem 0 ou 1 elemento
- Uma solução mais elegante é usar listas circulares
- Neste caso, utiliza-se apenas um ponteiro para o fim da fila e fica implícito que o início da fila é o nó seguinte



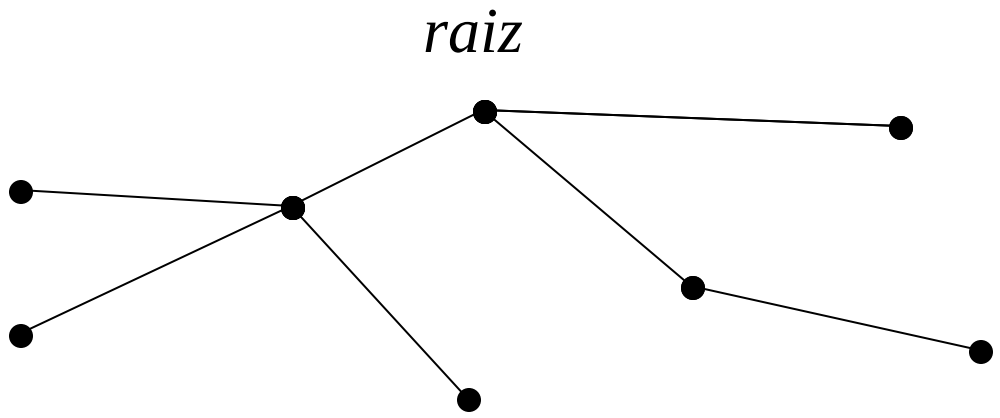
Listas Duplamente Encadeadas

- Para implementar deques, precisamos ser capazes de seguir a seqüência de nós em ambos os sentidos
- Para tanto, utiliza-se listas duplamente encadeadas
- Cada nó possui dois elos, um apontando para o nó seguinte e outro para o nó anterior
- Também neste caso podemos denotar o início e o fim da cadeia explicitamente ou utilizando listas circulares



Árvores

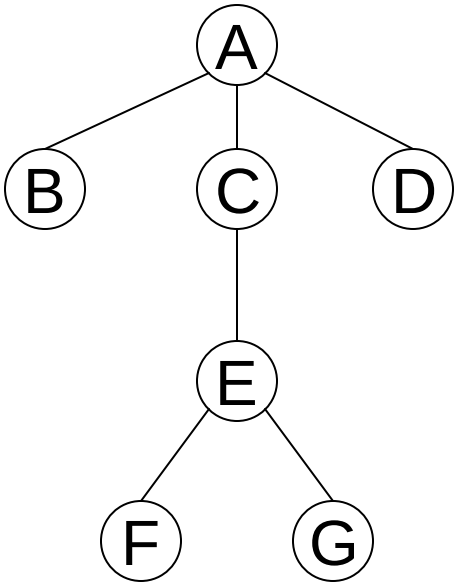
- Árvores são estruturas das mais usadas em computação
- Árvores são usadas para representar hierarquias
- Uma árvore pode ser entendida como um grafo acíclico conexo onde um dos vértices – chamado *raiz da árvore* – é diferenciado dos demais



Árvores

- Uma maneira mais útil de se definir árvores é a seguinte:
  - Uma árvore *T* é um conjunto finito de nós (ou vértices) tal que
    - $T = \varnothing$, isto é, uma árvore vazia
    - Um nó raiz e um conjunto de árvores não vazias, chamadas de subárvores do nó raiz
- 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

Árvores



Representação Gráfica

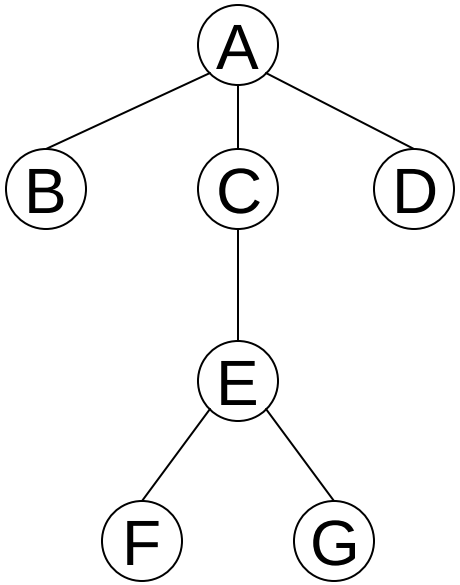
```
A
  B
  C
    E
      F
      G
  D
```

Representação Indentada

(A(B)(C(E(F)(G)))(D))

Representação com Parênteses

Árvores – Nomenclatura



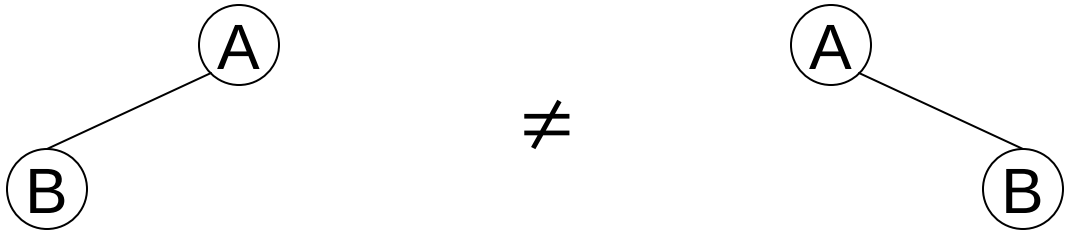
- "A" é o pai de "B", "C" e "D"
- "B", "C" e "D" são filhos de "A"
- "B", "C" e "D" são irmãos
- "A" é um ancestral de "G"
- "G" é um descendente de "A"
- "B", "D", "F" e "G" são nós folhas
- "A", "C" e "E" são nós internos
- O grau do nó "A" é 3
- O comprimento do caminho entre "C" e "G" é 2
- O nível de "A" é 1 e o de "G" é 4
- A altura da árvore é 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

Árvores Binárias

- Uma *árvore binária* é
  - Uma árvore vazia ou
  - Um nó *raiz* e duas subárvores binárias denominadas subárvore *direita* e subárvore *esquerda*
- Observe que uma árvore binária não é propriamente uma árvore já que os filhos de cada nó têm nomes (esquerdo e direito)



Número de Subárvores Vazias

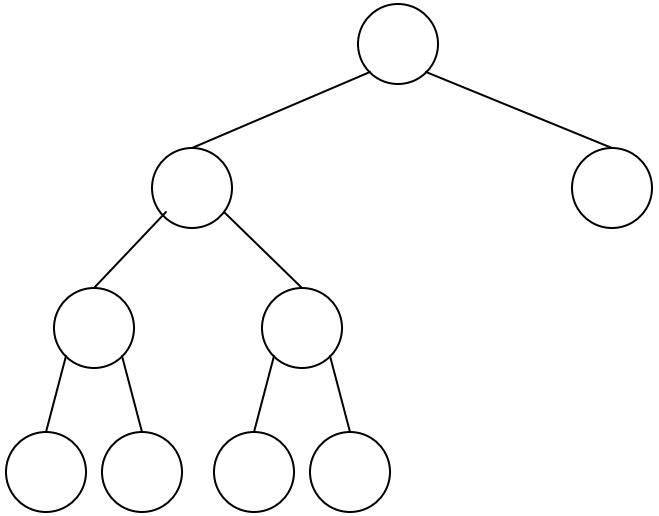
- Se uma árvore tem $n > 0$ nós, então ela possui $n+1$ subárvores vazias
- Para ver isso, observe que
  - Uma árvore com um só nó tem 2 subárvores vazias
  - Sempre que “penduramos” um novo nó numa árvore, o número de nós cresce de 1 e o de subárvores vazias também cresce de 1

Tipos Especiais de Árvores Binárias

- Uma árvore binária é estritamente binária sse 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

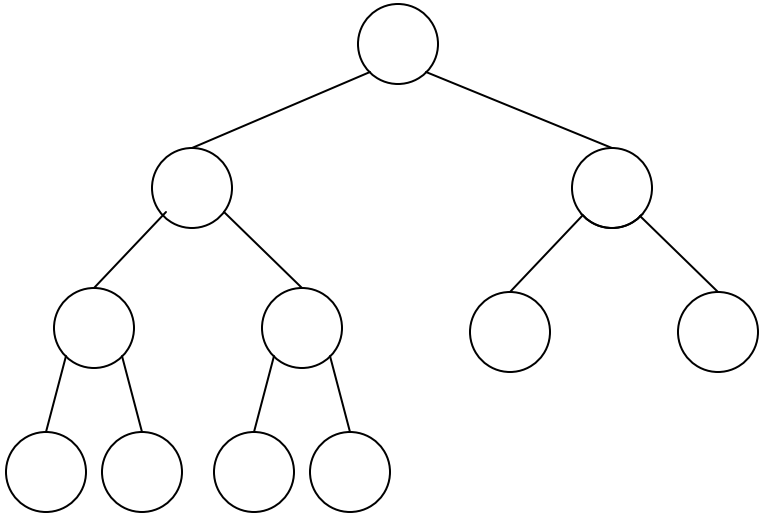
Tipos Especiais de Árvores Binárias

- Uma árvore binária é estritamente binária sse 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



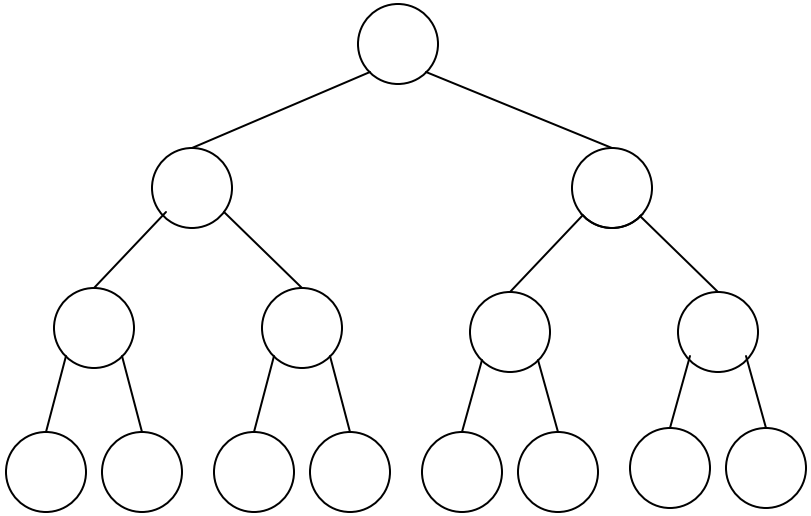
Tipos Especiais de Árvores Binárias

- Uma árvore binária é estritamente binária sse 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



Tipos Especiais de Árvores Binárias

- Uma árvore binária é estritamente binária sse 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

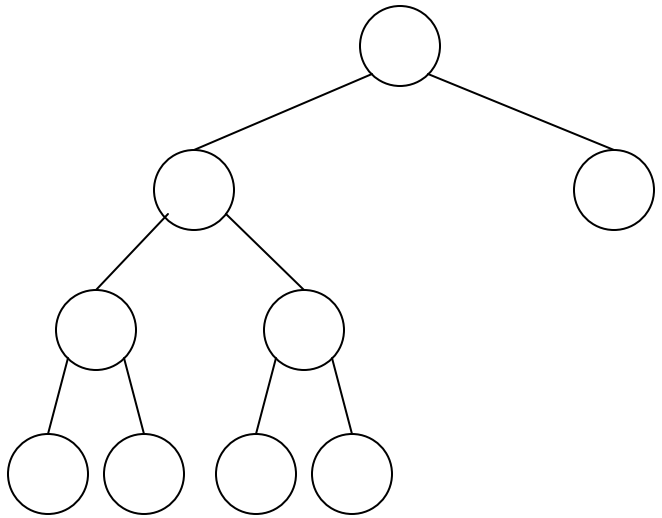


Altura de Árvores Binárias

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

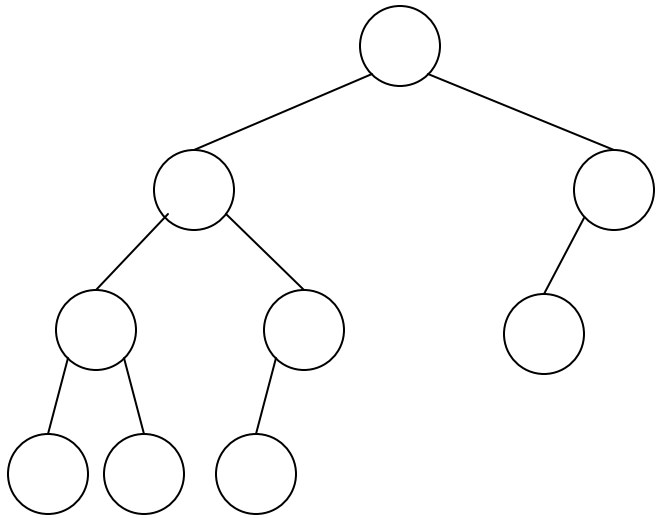
Altura de Árvores Binárias

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



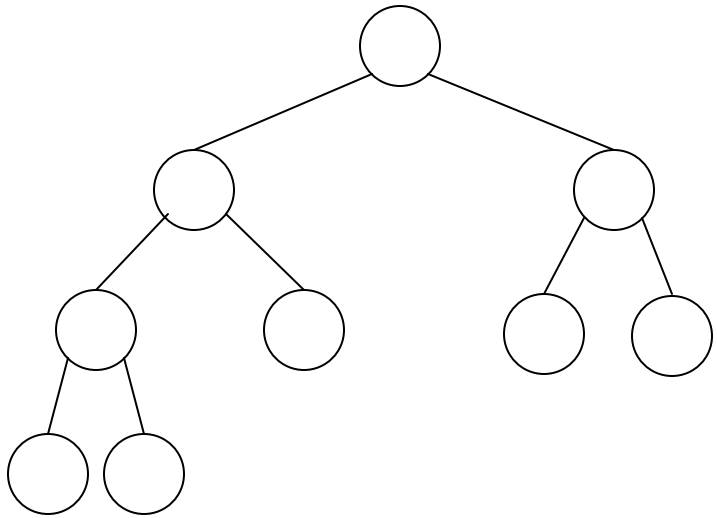
Altura de Árvores Binárias

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



Altura de Árvores Binárias

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

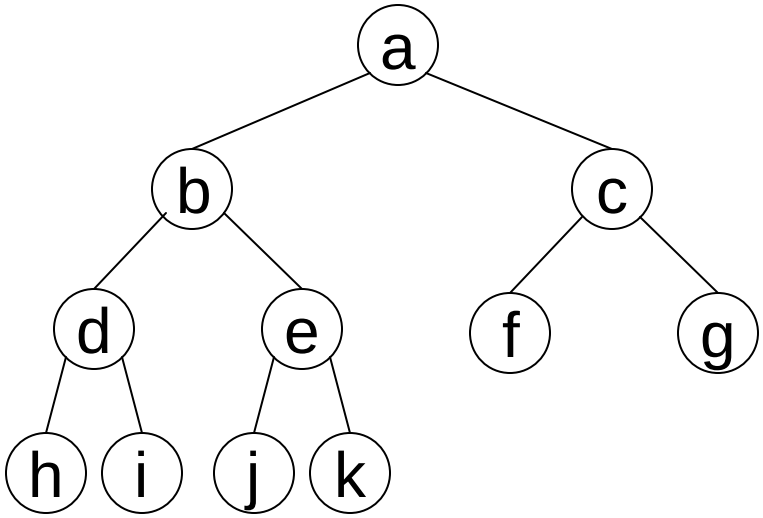


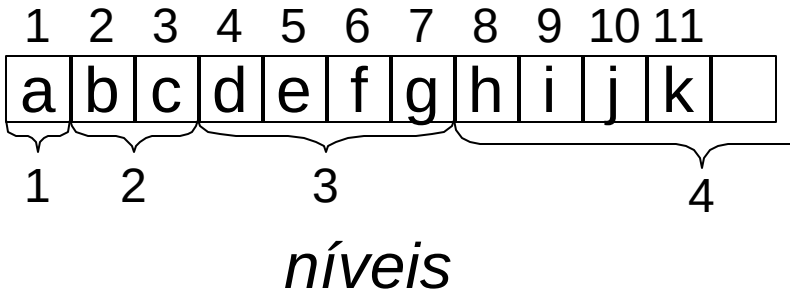
Altura de Árvores Binárias

- A altura mínima de uma árvore binária com $n > 0$ nós é $h = 1 + \lfloor \log_2 n \rfloor$
- Prova-se por indução. Seja T uma árvore completa de altura h
  - Vale para o caso base (n=1)
  - Seja T' uma árvore cheia obtida a partir de T pela remoção de k folhas do último nível
    - Então T' tem $n' = n - k$ nós
    - Como T' é uma árvore cheia, $n' = 1 + 2 + \ldots + 2^{h-2} = 2^{h-1} - 1$ e $h = 1 + \log_2 (n'+1)$
    - Sabemos que $1 \leq k \leq n'+1$ e portanto $\log_2 (n'+1) = \lfloor \log_2 (n' + k) \rfloor = \lfloor \log_2 n \rfloor$

Implementando Árvores Binárias com Arrays

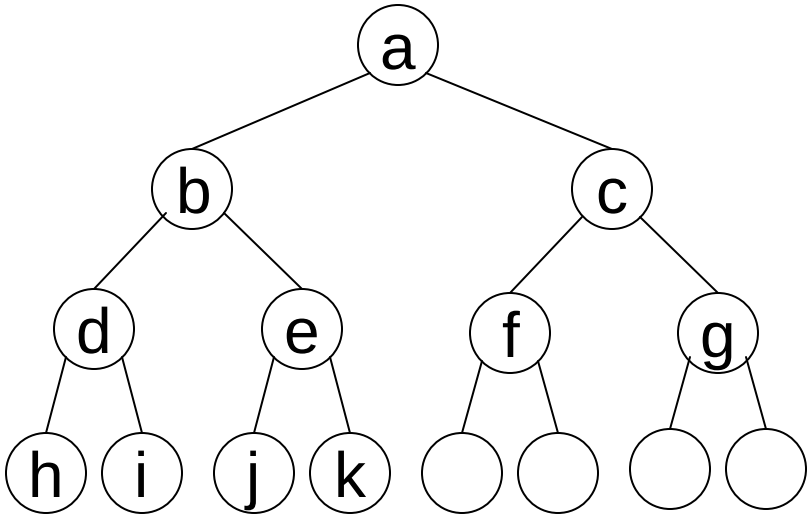
- Assim como listas, árvores binárias podem ser implementadas utilizando-se o armazenamento contíguo proporcionado por arrays
- A idéia é armazenar níveis sucessivos da árvore seqüencialmente no array

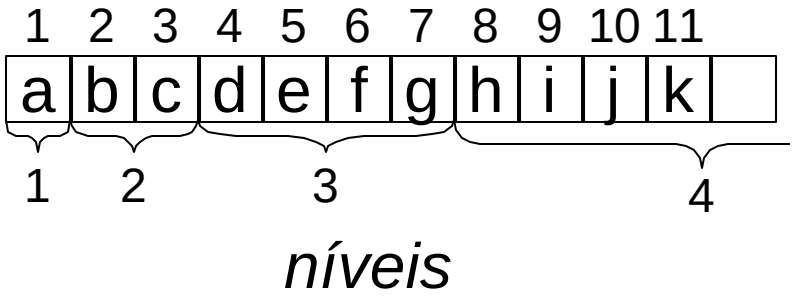




Implementando Árvores Binárias com Arrays

- Assim como listas, árvores binárias podem ser implementadas utilizando-se o armazenamento contíguo proporcionado por arrays
- A idéia é armazenar níveis sucessivos da árvore seqüencialmente no array



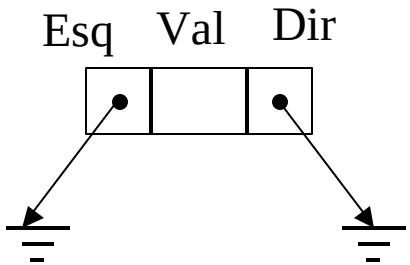


Implementando Árvores Binárias com Arrays

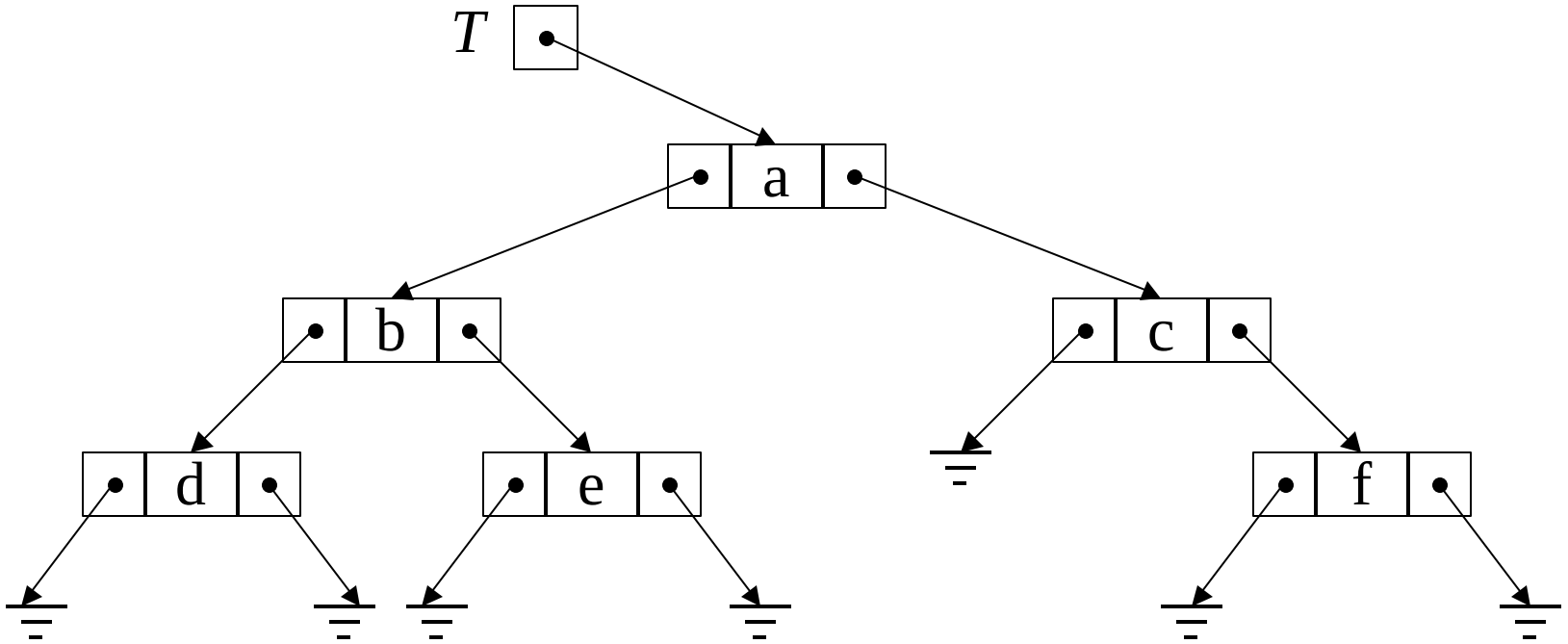
- Dado um nó armazenado no índice *i*, é possível computar o índice
  - do nó filho esquerdo de *i* : 2 *i*
  - do nó filho direito de *i* : 2 *i* + 1
  - do nó pai de *i* : *i* div 2
- Para armazenar uma árvore de altura *h* precisamos de um array de $2^h - 1$ (número de nós de uma árvore cheia de altura *h*)
- Nós correspondentes a subárvores vazias precisam ser marcados com um valor especial diferente de qualquer valor armazenado na árvore
- A cada índice computado é preciso se certificar que está dentro do intervalo permitido
  - Ex.: O nó raiz é armazenado no índice 1 e o índice computado para o seu pai é 0

Implementando Árvores Binárias com Ponteiros

- A implementação com arrays é simples porém tende a desperdiçar memória, e é pouco flexível quando se quer alterar a árvore (inserção e deleção de nós)
- 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
  - Uma árvore é representada por um ponteiro para seu nó raiz

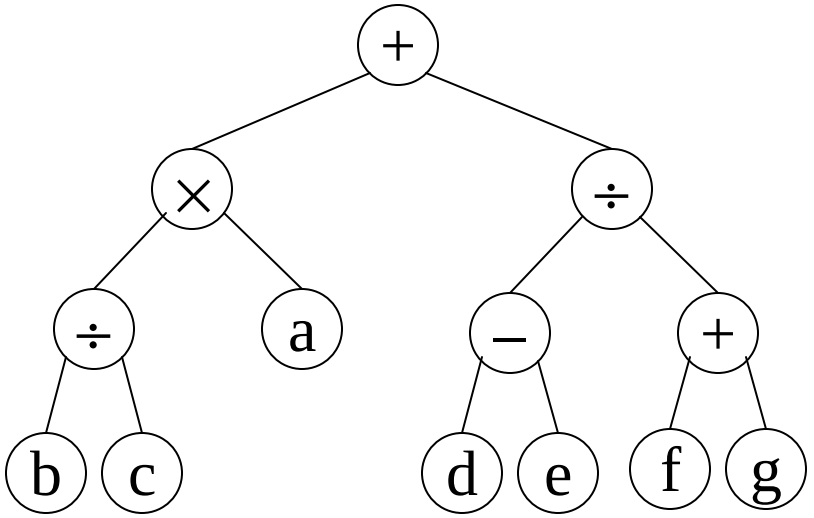


Implementando Árvores Binárias com Ponteiros



Aplicação: Expressões

- Uma aplicação bastante corriqueira de árvores binárias é na representação e processamento de expressões algébricas, booleanas, etc



(((b/c) * a)+((d-e)/(f+g)))

Avaliando uma Expressão

- Se uma expressão é codificada sob a forma de uma árvore, sua avaliação pode ser feita percorrendo os nós da árvore

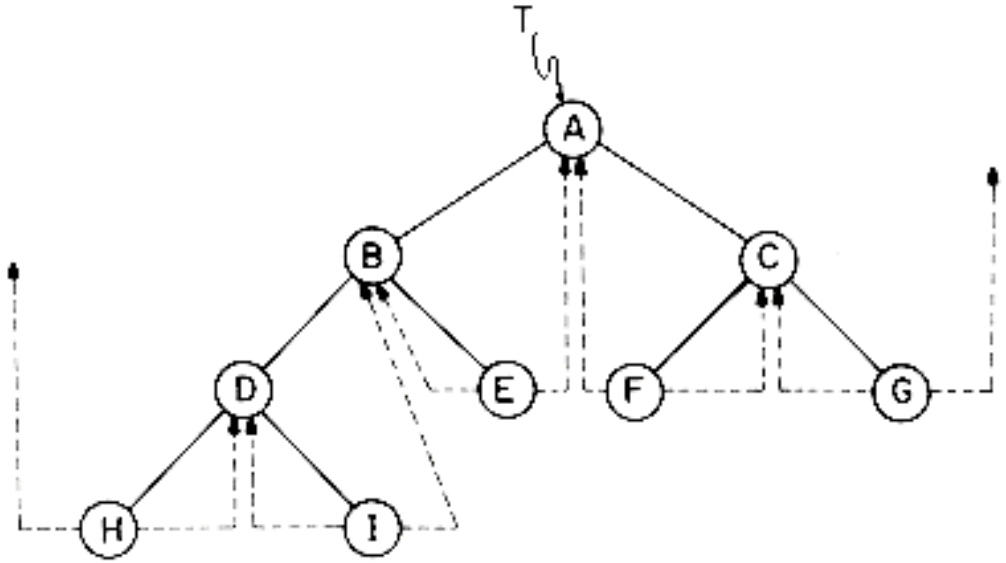
```
proc Avalia (Arvore T) {
    se T^.Val é uma constante ou uma variável então
        retornar o valor de T^.Val
    senão {
        operando1 ← Avalia (T^.Esq)
        operando2 ← Avalia (T^.Dir)
        se T^.Val = "+" então
            retornar operando1 + operando2
        senão se T^.Val = "–" então
            retornar operando1 – operando2
        senão se T^.Val = "*" então
            retornar operando1 * operando2
        senão se T^.Val = "/" então
            retornar operando1 / operando2
    }
}
```

Percurso de Árvores Binárias

- Existem essencialmente 3 ordens “naturais” de se percorrer os nós de uma árvore
  - Pré-ordem: raiz, esquerda, direita
  - Pós-ordem: esquerda, direita, raiz
  - In-ordem: esquerda, raiz, direita
- Por exemplo:
  - percorrendo uma árvore que representa uma expressão em in-ordem, obtém-se a expressão em sua forma usual (infixa);
  - um percurso em pós-ordem produz a ordem usada em calculadoras “HP”;
  - um percurso em pré-ordem retorna a expressão em forma infixa, como usado em LISP

Árvores Costuradas

- Notamos que o percurso de árvores pode ser feito usando uma rotina recursiva em $O(n)$, onde n é o número de nós
- Algumas vezes é interessante se poder percorrer árvores binárias sem usar rotinas recursivas ou pilhas
- Uma idéia consiste em usar os ponteiros esquerdo e direito dos nós folhas para apontar para os nós anterior e posterior do percurso em in-ordem

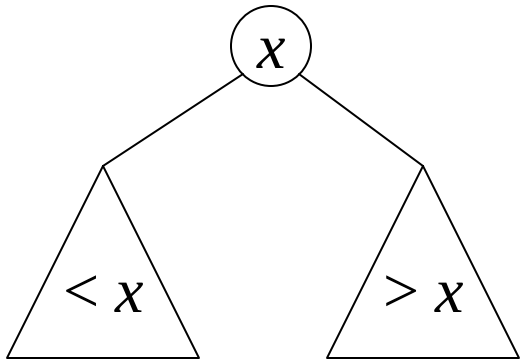


Dicionários

- A operação de busca é fundamental em diversos contextos da computação
- Por exemplo, um *dicionário* é uma estrutura de dados que reúne uma coleção de chaves sobre a qual são definidas as seguintes operações :
  - *Inserir* (*x*, *T*) : inserir chave *x* no dicionário *T*
  - *Remover* (*x*, *T*) : remover chave *x* do dicionário *T*
  - *Buscar* (*x*, *T*) : verdadeiro apenas se *x* pertence a *T*
- Outras operações são comuns em alguns casos:
  - Encontrar chave pertencente a *T* que sucede ou precede *x*
  - Listar todas as chaves entre *x*1 e *x*2

Árvores Binárias de Busca

- Uma maneira simples e popular de implementar dicionários é uma estrutura de dados conhecida como árvore binária de busca
- Numa árvore binária de busca, todos os nós na subárvore à esquerda de um nó contendo uma chave *x* são menores que *x* e todos os nós da subárvore à direita são maiores que *x*

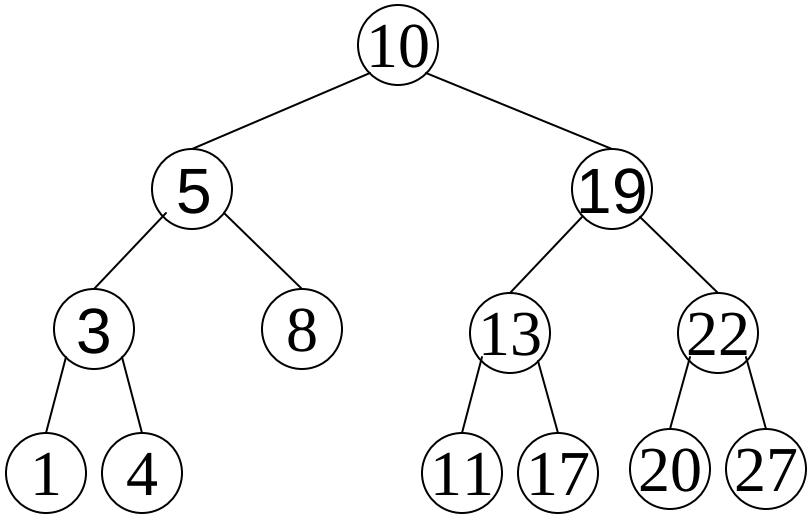


Busca e Inserção em Árvores Binárias de Busca

```
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)
    }
}
```

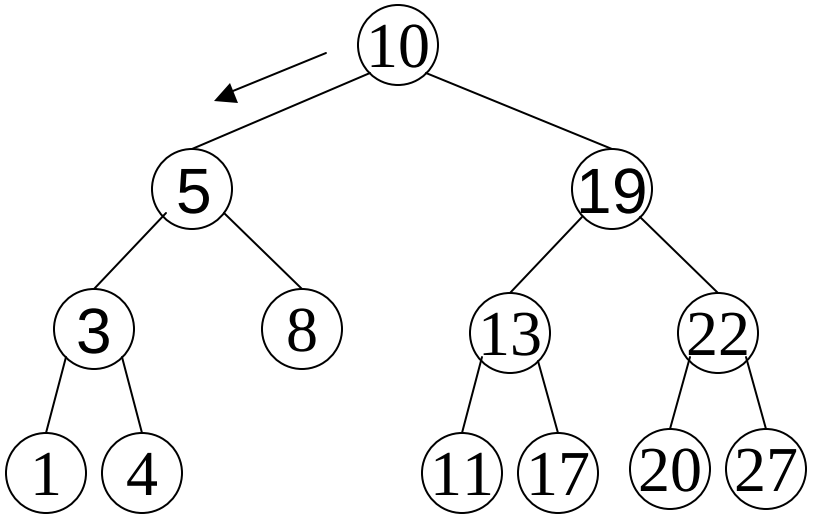
Inserção em Árvores Binárias de Busca

Inserir (9, *T*)



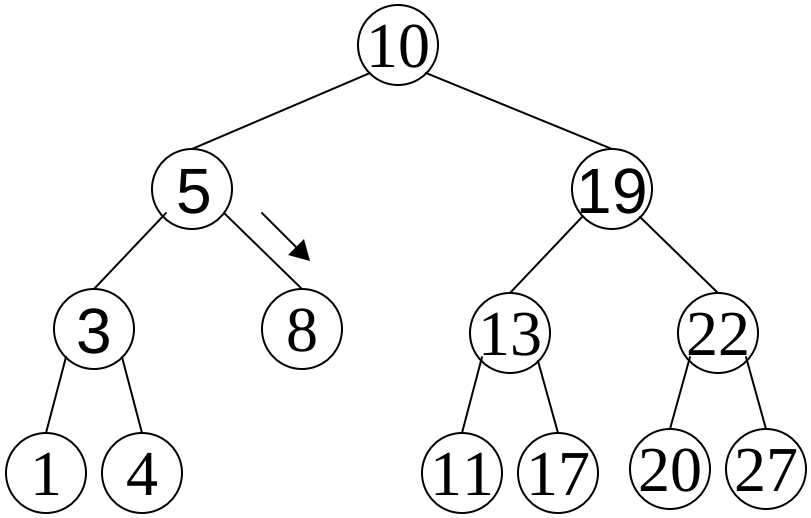
Inserção em Árvores Binárias de Busca

Inserir (9, *T*)



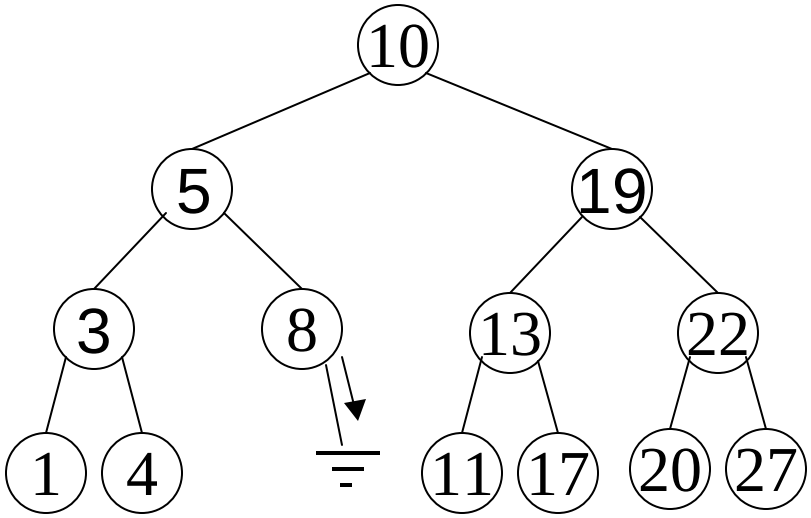
Inserção em Árvores Binárias de Busca

Inserir (9, *T*)



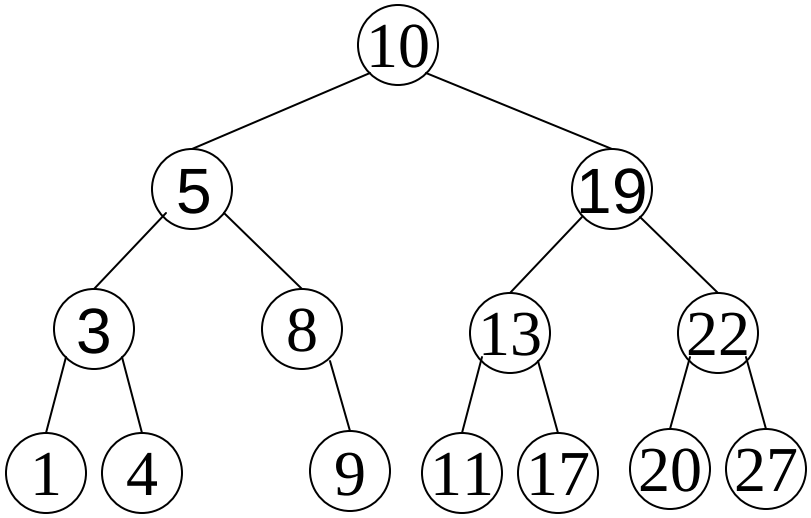
Inserção em Árvores Binárias de Busca

Inserir (9, *T*)



Inserção em Árvores Binárias de Busca

Inserir (9, *T*)



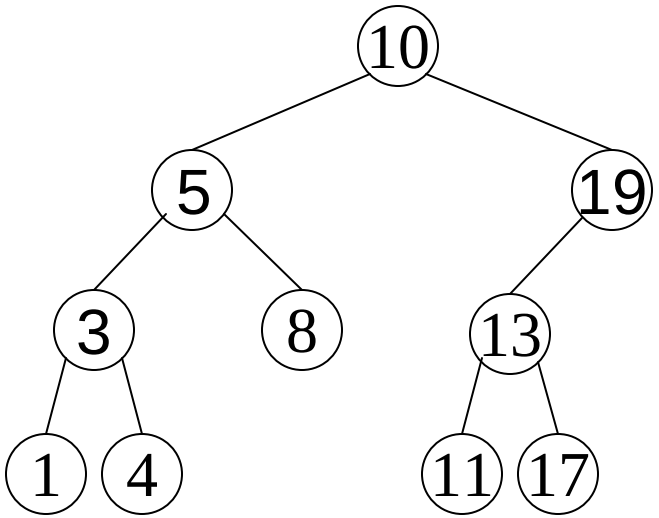
Remoção em Árvores Binárias de Busca

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

Remover (19, *T*)

Remoção em Árvores Binárias de Busca

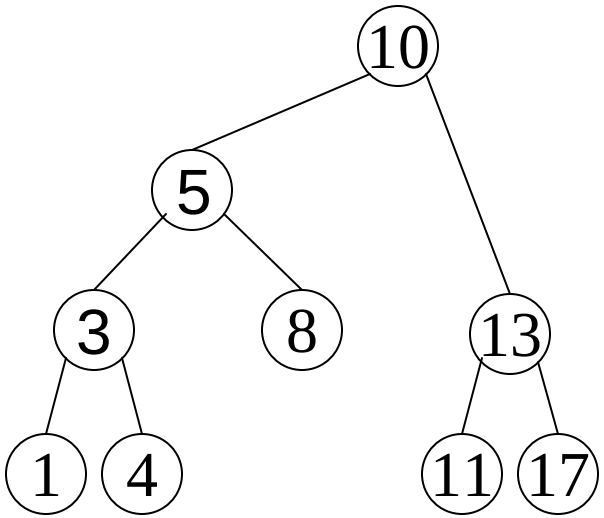
- Para remover uma chave *x* de uma árvore *T* temos que distinguir os seguintes casos
  - *x* está numa folha de *T :* neste caso, a folha pode ser simplesmente removida
  - *x* está num nó que tem sua subárvore esquerda ou direita vazia: neste caso o nó é removido substituído pela subárvore não nula



Remover (19, *T*)

Remoção em Árvores Binárias de Busca

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



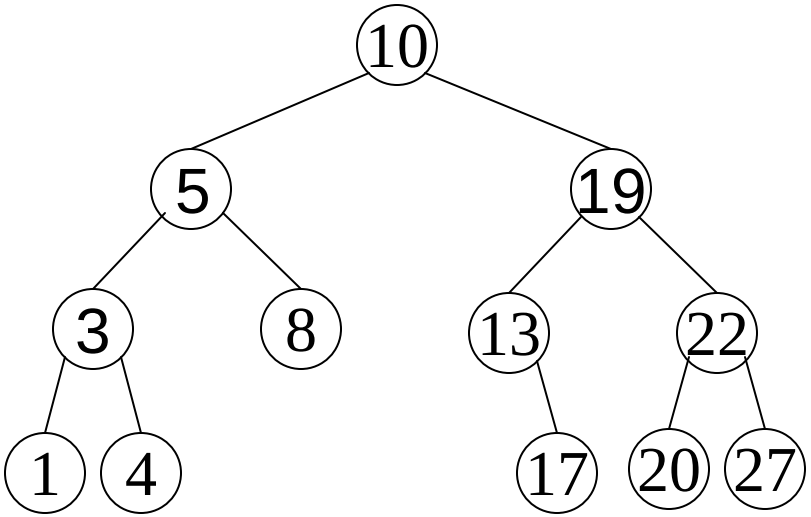
Remover (19, *T*)

Remoção em Árvores Binárias de Busca

- 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:
  - A menor das chaves maiores que *x* ou
  - A maior das chaves menores que x

Remoção em Árvores Binárias de Busca

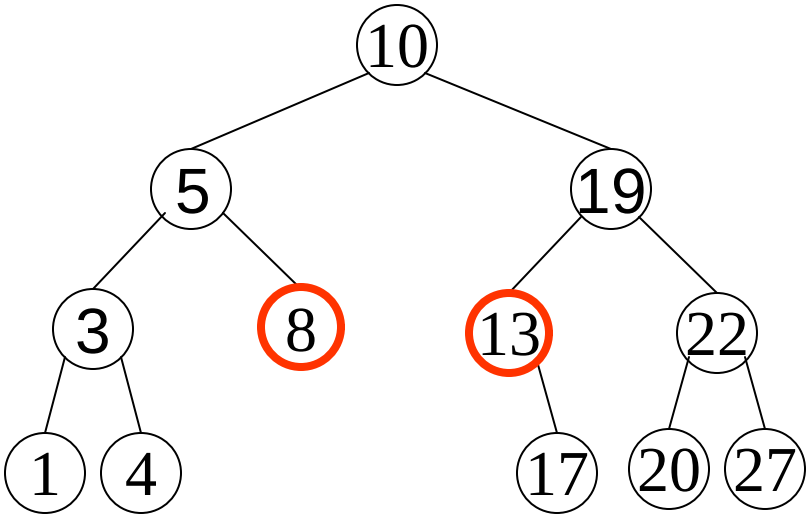
- Se *x* está num nó em que ambas subárvores são não nulas, é preciso encontrar uma chave *y* que a possa substituir. Há duas chaves candidatas naturais:
  - A menor das chaves maiores que *x* ou
  - A maior das chaves menores que x



Remover (10, *T*)

Remoção em Árvores Binárias de Busca

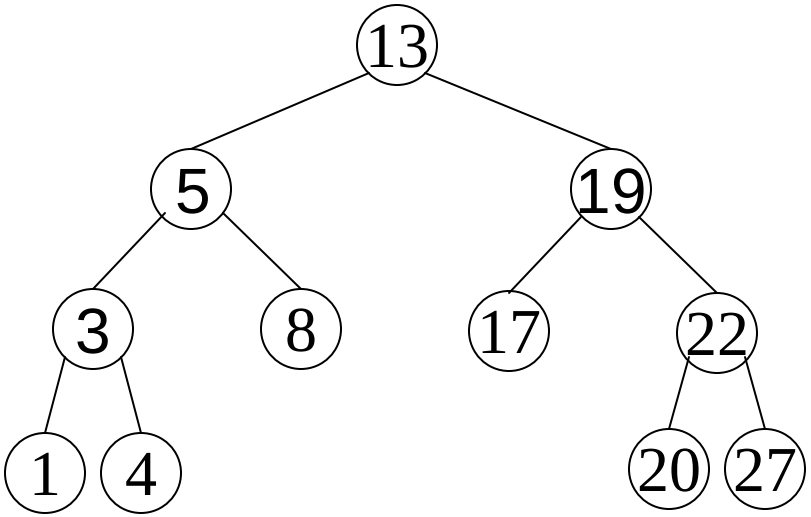
- Se *x* está num nó em que ambas subárvores são não nulas, é preciso encontrar uma chave *y* que a possa substituir. Há duas chaves candidatas naturais:
  - A menor das chaves maiores que *x* ou
  - A maior das chaves menores que x



Remover (10, *T*)

Remoção em Árvores Binárias de Busca

- 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:
  - A menor das chaves maiores que *x* ou
  - A maior das chaves menores que x



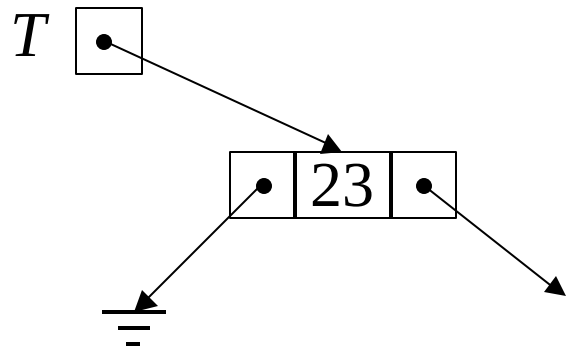
Remover (10, *T*)

Remoção em Árvores Binárias de Busca

```
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)
}
```

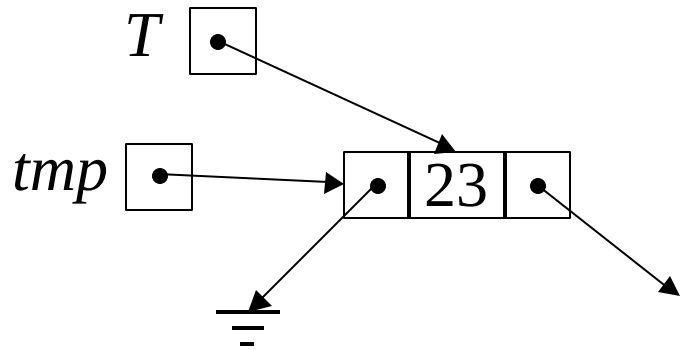
Remoção em Árvores Binárias de Busca

```
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)
}
```



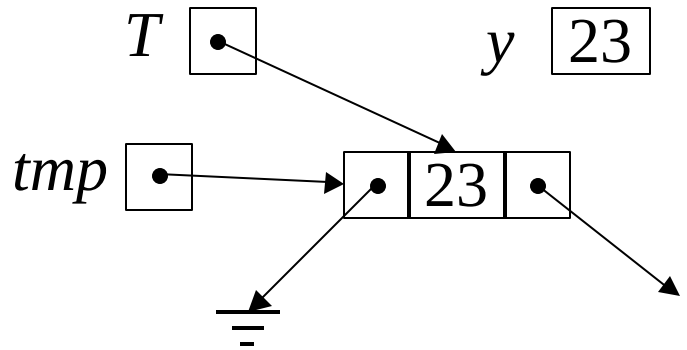
Remoção em Árvores Binárias de Busca

```
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)
}
```



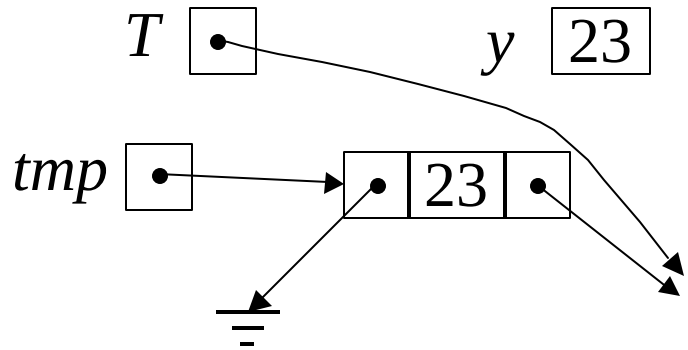
Remoção em Árvores Binárias de Busca

```
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)
}
```



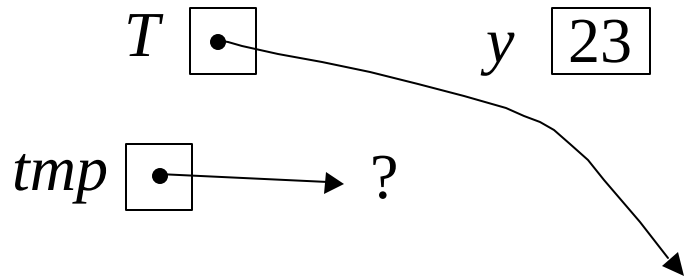
Remoção em Árvores Binárias de Busca

```
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)
}
```



Remoção em Árvores Binárias de Busca

```
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)
}
```



Remoção em Árvores Binárias de Busca

```
proc Remover (Chave x, 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 {
                tmp ← T
                T ← T^.Dir
                Liberar (tmp)
            }
            senão se T^.Dir = Nulo então {
                tmp ← T
                T ← T^.Esq
                Liberar (tmp)
            }
            senão T^.Val ← RemoverMenor (T^.Dir)
}
```

Árvores Binárias de Busca - Complexidade

- A busca em uma árvore binária tem complexidade $O(h)$
- A altura de uma árvore é,
  - no pior caso, n
  - no melhor caso, $\lfloor \log_2 n \rfloor + 1$ (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

Árvores de Busca de Altura Ótima

- É fácil ver que podemos garantir uma árvore de altura ótima para uma coleção de chaves se toda vez que temos que escolher uma chave para inserir, optamos pela mediana:

```
proc InserirTodos (i, n, A [i .. i+n–1], var Árvore T) {
    se n = 1 então Inserir (A [i], T)
    senão {
        j ← Mediana (i, n, A)
        trocar  A[i] com A [j]
        m ← Particao (i, n, A)
        Inserir (A [i+m], T)
        InserirTodos (i, m, A, T^.Esq)
        InserirTodos (i+m+1, n–m–1, A, T^.Dir)
    }
}
```

Árvores de Busca de Altura Ótima

- Sabendo que tanto *Mediana* quanto *Particao* podem ser feitos em $O(n)$ o algoritmo *InsereTodos* executa em tempo $O(n \log n)$
- O algoritmo pode ser reescrito de forma mais sucinta se admitimos que o array *A* se encontra ordenado
- Nem sempre, entretanto, podemos garantir que conhecemos todas as chaves de antemão
- O que esperar em geral?
  - Percebemos que a altura da árvore final depende da ordem de inserção
  - Temos $n!$ possíveis ordens de inserção
  - Se todas as ordens de inserção são igualmente prováveis, no caso médio teremos uma árvore de altura $\approx 1 + 2.4 \log n$

Altura de Árvores Binárias de Busca – Caso Médio

- Eis uma explicação intuitiva para o fato de que no caso médio, a altura de uma árvore binária de busca é $O(\log n)$
- Considere uma coleção ordenada de n chaves
  - Vemos que se escolhemos a k'ésima chave para inserir primeiro, teremos à esquerda do nó raiz uma subárvore com $k - 1$ nós e à direita uma subárvore com $n - k$ nós
  - No melhor caso $k = n / 2$
  - No pior caso, $k = 1$ ou $k = n$
  - Admitamos que o caso médio corresponde a $k = n/4$ ou $k=3n/4$ (estamos ignorando tetos, pisos, etc)

Altura de Árvores Binárias de Busca – Caso Médio

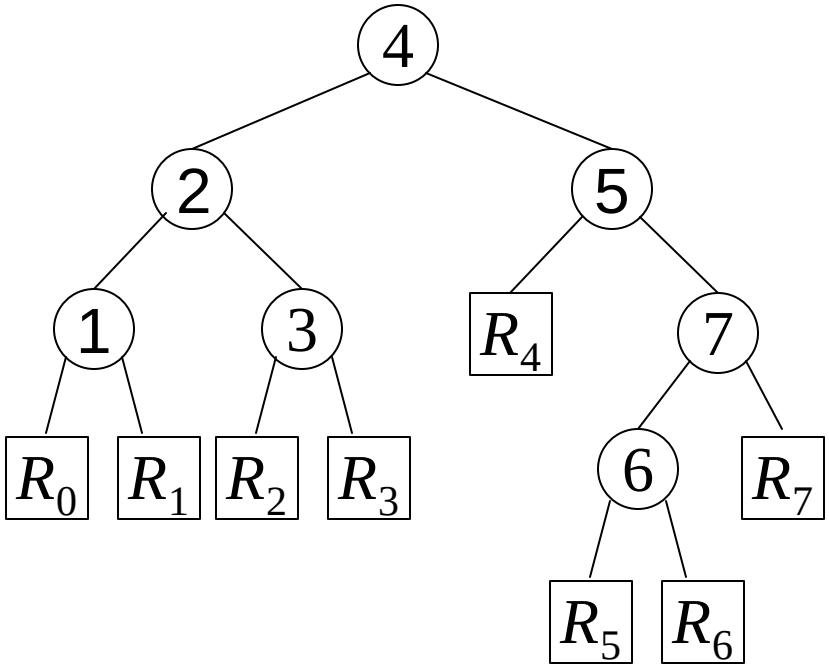
- Em qualquer caso, a subárvore com 3n/4 nós vai dominar a altura da árvore como um todo
- Resolvendo a recursão (novamente ignorando o fato que 3*n*/4 nem sempre é um inteiro), temos

$$
\begin{aligned}
H(1) &= 1 \\
H(n) &= 1 + H(3n/4) \\
&= 2 + H(9n/16) \\
&= 3 + H(27n/64) \\
&= \ldots \\
&= m + H((3/4)^m n) \\
&\therefore m = \frac{\log_2 n}{\log_2 4/3} \approx 2.4 \log_2 n \\
H(n) &= 1 + 2.4 \log_2 n
\end{aligned}
$$

Árvores Binárias de Busca Ótimas

- Dada uma árvore binária de busca, um dado importante é o número total de comparações que precisamos fazer durante a busca de uma chave
  - 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
  - Se a chave x sendo buscada não pertence à árvore, o número de comparações corresponde à subárvore vazia (também chamada de *nó externo*) que encontramos durante o processo de busca
    - Cada subárvore nula R_i corresponde 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
    - Os casos extremos R_0 e R_n correspondem a $x < s_1$ e $s_n < x$
    - O número de comparações para encontrar x, portanto, é o nível dessa subárvore nula (a que chamaremos de l'_k) menos 1

Árvore de Busca 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, $E(T)=I(T)+n$
- Árvores completas minimizam tanto $E(T)$ quanto $I(T)$

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- Admitindo probabilidade uniforme no acesso de quaisquer chaves, as árvores completas são ótimas
- Entretanto, se a distribuição não é uniforme, precisamos empregar um algoritmo mais elaborado
- Sejam
  - f_k a freqüência de acesso à *k*'ésima menor chave, armazenada em T no nível l_k,
  - f'_k a freqüência de acesso a chaves que serão buscadas nos nós externos R_k, armazenados em T no nível l'_k,
- Então, o custo médio de acesso é dado por

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

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- O algoritmo para construção de árvores ótimas baseia-se no fato de que subárvores de árvores ótimas são também ótimas
  - se assim não fosse, poderíamos substituir uma subárvore não ótima por uma ótima e diminuir o custo da árvore ótima, o que é um absurdo
- O algoritmo consiste de testar todas as *n* chaves como raízes da árvore e escolher aquela que leva ao custo mínimo
  - As subárvores são construídas de forma recursivamente idêntica

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- Seja $T(i, j)$ a árvore ótima para as chaves $\{s_{i+1}, s_{i+2}, \ldots s_j\}$
- Seja $F(i, j)$ a soma de todas as freqüências relacionadas com $T(i, j)$, isto é,

$$F(i, j) = \sum_{i<k\leq j} f_k + \sum_{i\leq k\leq j} f'_k$$

- Assumindo que $T(i, j)$ foi construída escolhendo s_k como raiz, então prova-se que

$$c(T(i, j)) = c(T(i, k-1)) + c(T(k, j)) + F(i, j)$$

- Portanto, para encontrar o valor de k apropriado, basta escolher aquele que minimiza a expressão acima

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- Seria possível em computar recursivamente $c(T(0,n))$ usando como caso base $c(T(i,i))=0$
- No entanto, esse algoritmo iria computar cada $c(T(i,j))$ e $F(i,j)$ múltiplas vezes
- Para evitar isso, valores já computados são armazenados em duas matrizes: $c[0 .. n, 0 .. n]$ e $F[0 .. n, 0 .. n]$
- Podemos também dispensar a construção recursiva e computar iterativamente o custo de todas as $n(n+1)/2$ árvores envolvidas no processo
  - As árvores com d nós depende apenas do custo de árvores com 0, 1, 2 ... e até $d - 1$ nós
  - Computar $F(i,j)$ também não oferece dificuldade

Árvore de Busca Ótima para Freqüências de Acesso Dadas

```
proc CustoArvoreOtima (n, f [1 .. n], f '[0 .. n]) {
    array c [0 .. n, 0 .. n], F [0 .. n, 0 .. n]
    para j desde 0 até n fazer {
        c [j, j] ← 0
        F [j, j] ← f ' [j]
    }
    para d desde 1 até n fazer
        para i desde 0 até n – d fazer {
            j ← i + d
            F [i, j] ← F [i, j – 1] + f [j] + f ' [j]
            tmp ← inf
            para k desde i + 1 até j fazer
                tmp ← min(tmp, c [i, k – 1] + c [k, j])
            c [i, j] ← tmp + F [i, j]
        }
}
```

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- O algoritmo para computar o custo da árvore ótima tem complexidade $O(n^3)$
- O algoritmo para criar a árvore ótima é trivial, bastando para isso usar como raízes os nós de índice k que minimizam o custo de cada subárvore (pode-se armazenar esses índices numa terceira matriz)
- É possível obter um algoritmo de complexidade $O(n^2)$ utilizando a propriedade de monotonicidade das árvores binárias de busca
  - Se s_k é a raiz da árvore ótima para o conjunto $\{s_i ... s_j\}$ então a raiz da árvore ótima para o conjunto $\{s_i ... s_j, s_{j+1}\}$ é s_q para algum $q \geq k$
  - (Analogamente, $q \leq k$ para $\{s_{i-1}, s_i ... s_j\}$)

Árvores Balanceadas

- Vimos que árvores completas garantem buscas utilizando não mais que $\lfloor \log_2 n \rfloor + 1$ comparações
- Mais importante, vimos que árvores binárias de busca, se construídas por inserção aleatória de elementos têm altura logarítmica (em *n*) na média
- Entretanto, não podemos assegurar que árvores binárias de busca construídas segundo qualquer ordem de inserção sempre têm altura logarítmica
- A idéia então é modificar os algoritmos de inserção e remoção de forma a assegurar que a árvore resultante é sempre de altura logarítmica
- Duas variantes mais conhecidas:
  - Árvores AVL
  - Árvores Graduadas

Árvores AVL

- 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
- O critério sugerido por Adelson-Velskii e Landis é o de garantir a seguinte invariante:
  - Para cada nó da árvore, a altura de sua subárvore esquerda e de sua subárvore direita diferem de no máximo 1
- Para manter essa invariante depois de alguma inserção ou remoção que desbalanceie a árvore, utiliza-se operações de custo $O(1)$ chamadas *rotações*

Árvores AVL

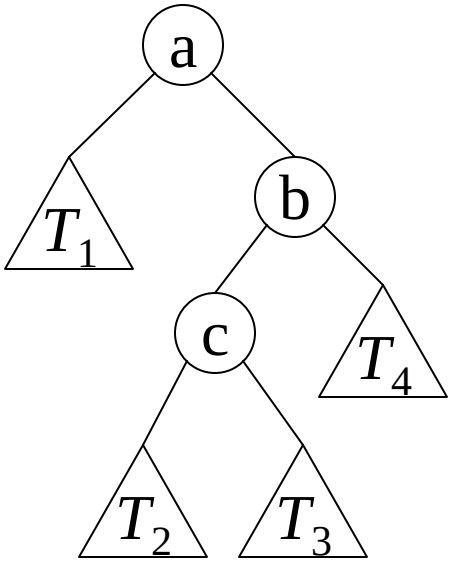
- Uma árvore AVL tem altura logarítmica?
  - Seja $N(h)$ o número mínimo de nós de uma árvore AVL de altura h
  - Claramente, $N(1) = 1$ e $N(2) = 2$
  - Em geral, $N(h) = N(h-1) + N(h-2) + 1$
  - Essa recorrência é semelhante à recorrência obtida para a série de Fibonacci
  - Sua solução resulta aproximadamente em

$$N(h) \approx \left(\frac{1+\sqrt{5}}{2} \right)^h \approx 1.618^h$$

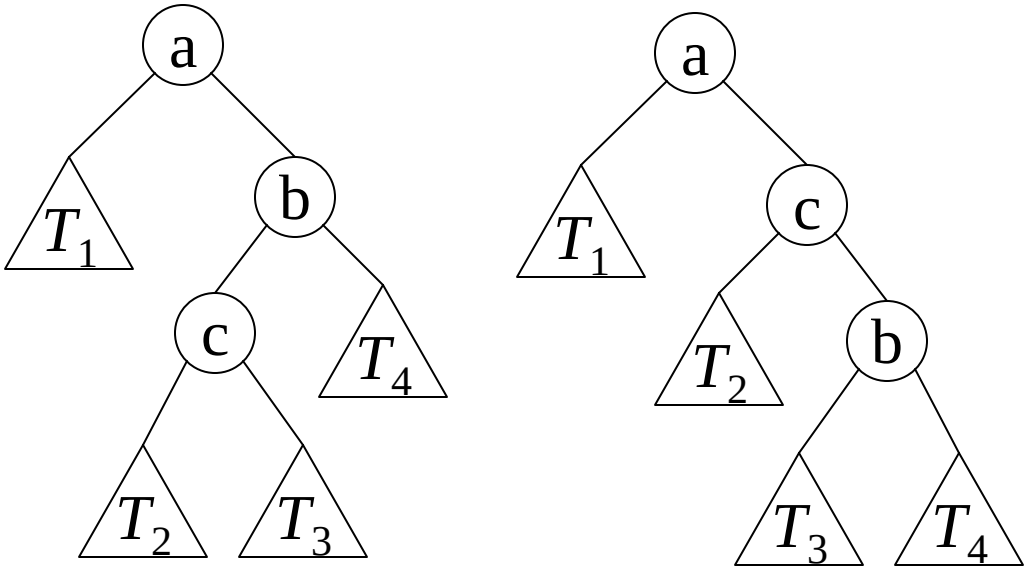
Rotações em Árvores Binárias de Busca

Rotações em Árvores Binárias de Busca

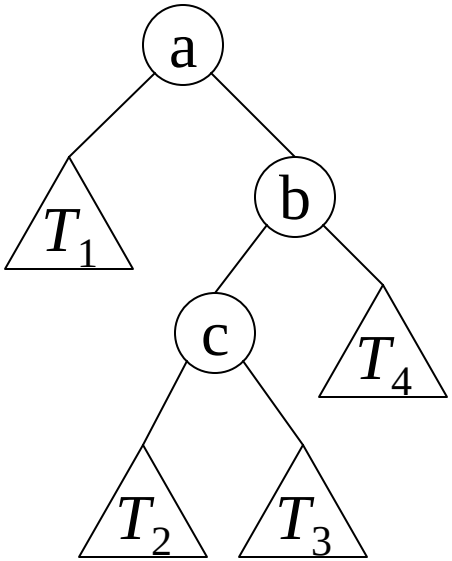
Rotações em Árvores de Busca

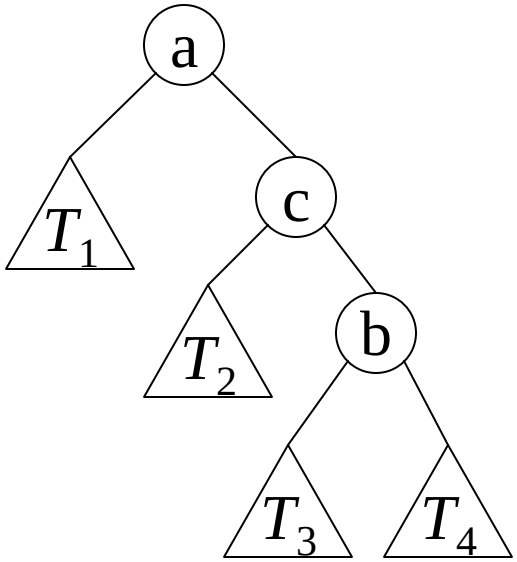


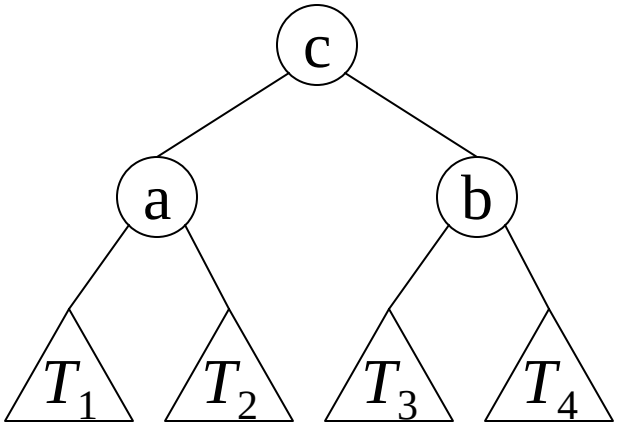
Rotações em Árvores de Busca



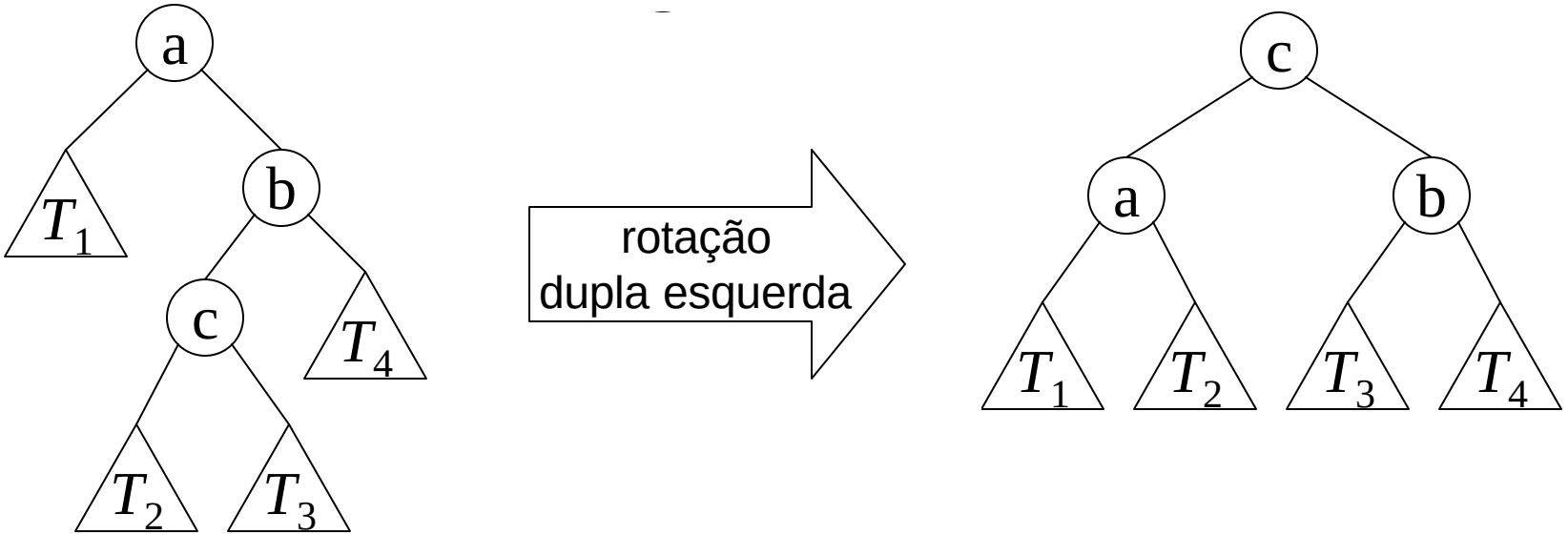
Rotações em Árvores de Busca



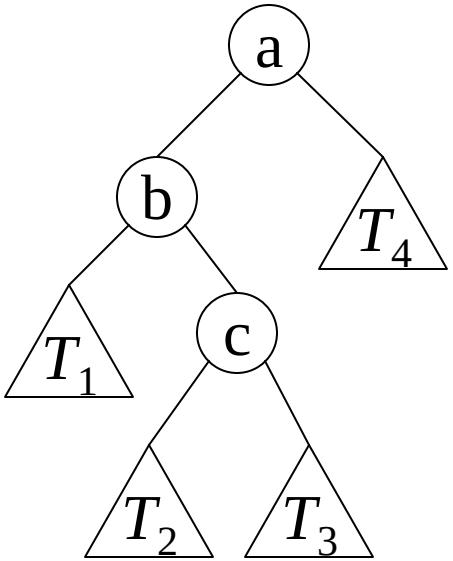




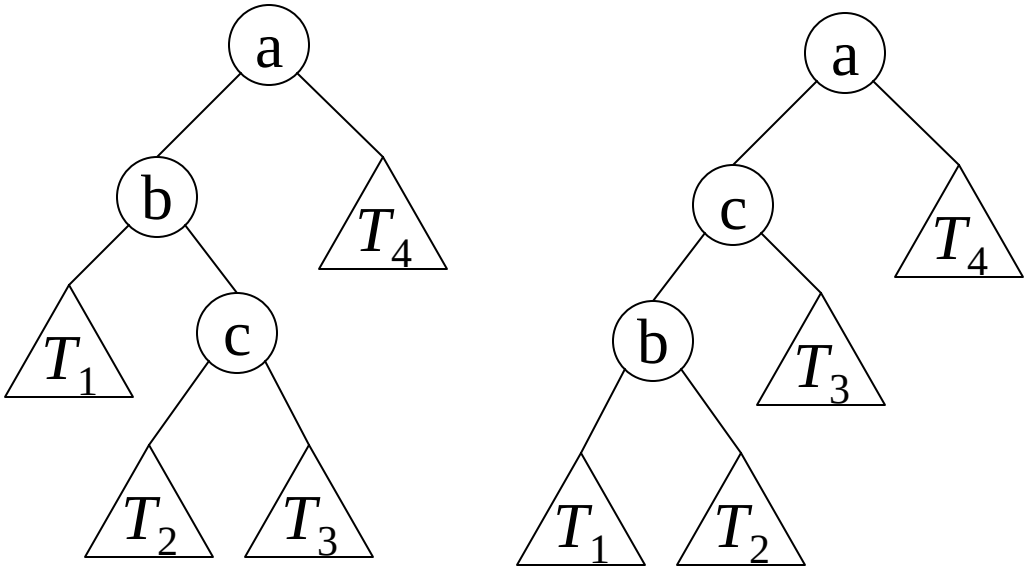
Rotações em Árvores de Busca



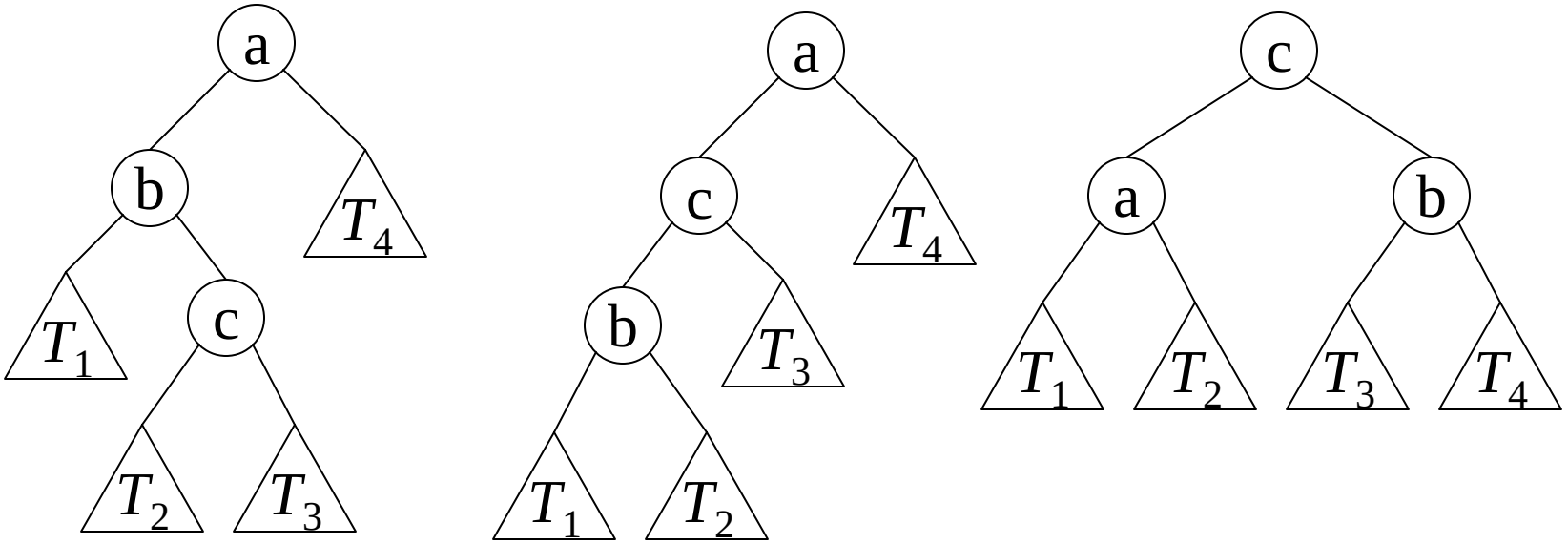
Rotações em Árvores de Busca



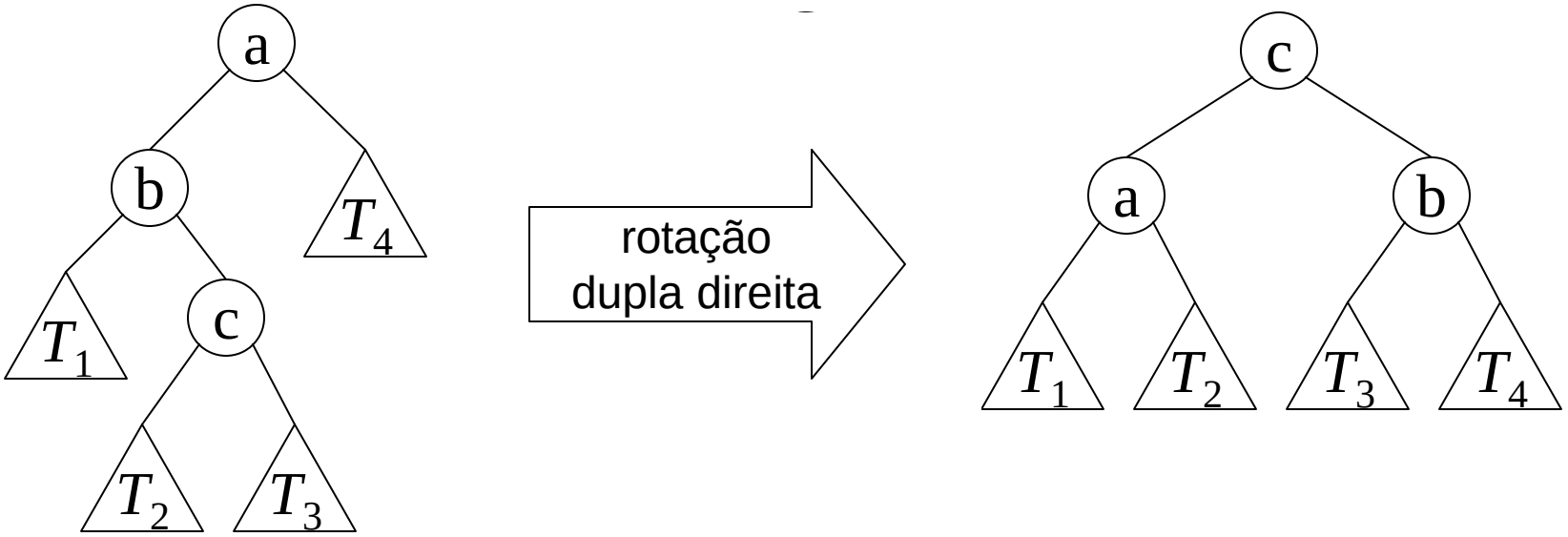
Rotações em Árvores de Busca



Rotações em Árvores de Busca



Rotações em Árvores de Busca



Inserção em árvores AVL

- Precisamos manter em cada nó um campo extra chamado *alt* que vai registrar a altura da árvore ali enraizada
  - Na verdade, apenas a diferença de altura entre a subárvore esquerda e direita precisa ser mantida (2 bits)
- Vamos precisar das seguintes rotinas para acessar 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
}
```

Inserção em árvores AVL

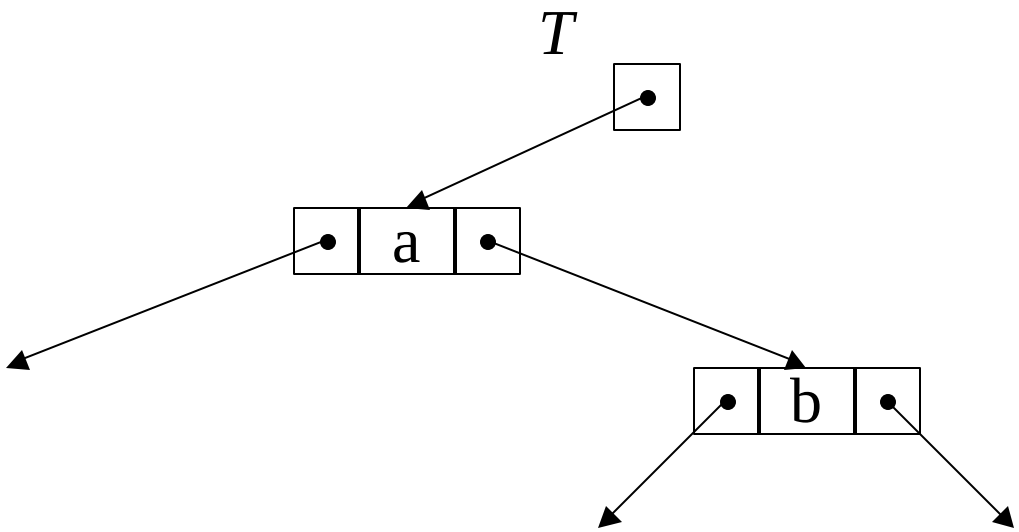
proc *RotacaoEsquerda* (var *Arvore T*) {

T, *T*^.*Dir*, *T*^.*Dir*^.*Esq* ← *T*^.*Dir*, *T*^.*Dir*^.*Esq*, *T*

AtualizaAltura (*T*^.*Esq*)

AtualizaAltura (*T*)

}



Inserção em árvores AVL

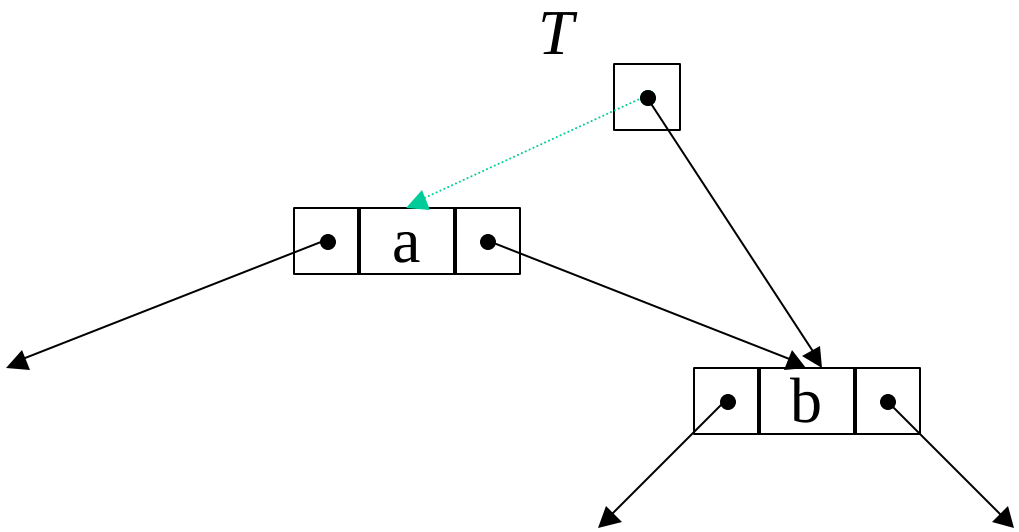
proc *RotacaoEsquerda* (var *Arvore T*) {

T, *T*^.*Dir*, *T*^.*Dir*^.*Esq* ← *T*^.*Dir*, *T*^.*Dir*^.*Esq*, *T*

AtualizaAltura (*T*^.*Esq*)

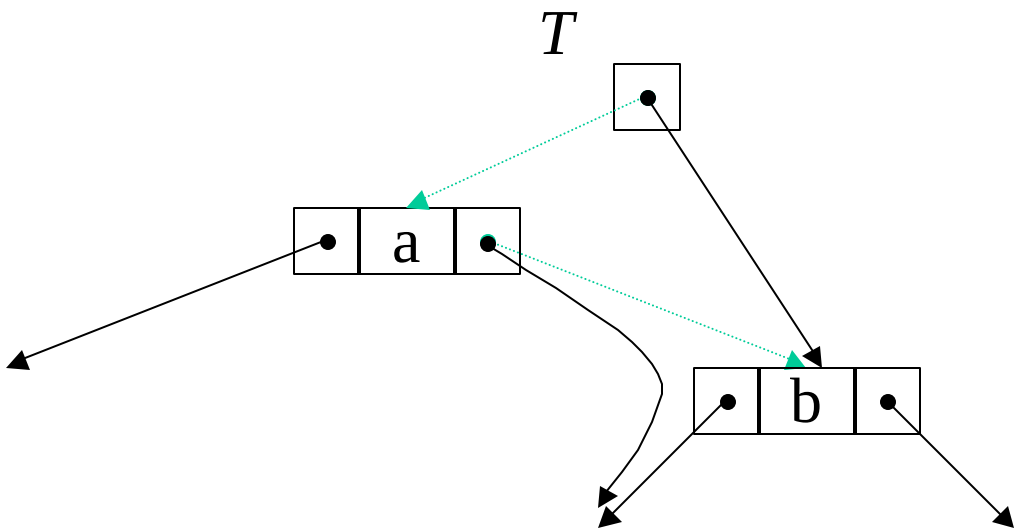
AtualizaAltura (*T*)

}



Inserção em árvores AVL

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



Inserção em árvores AVL

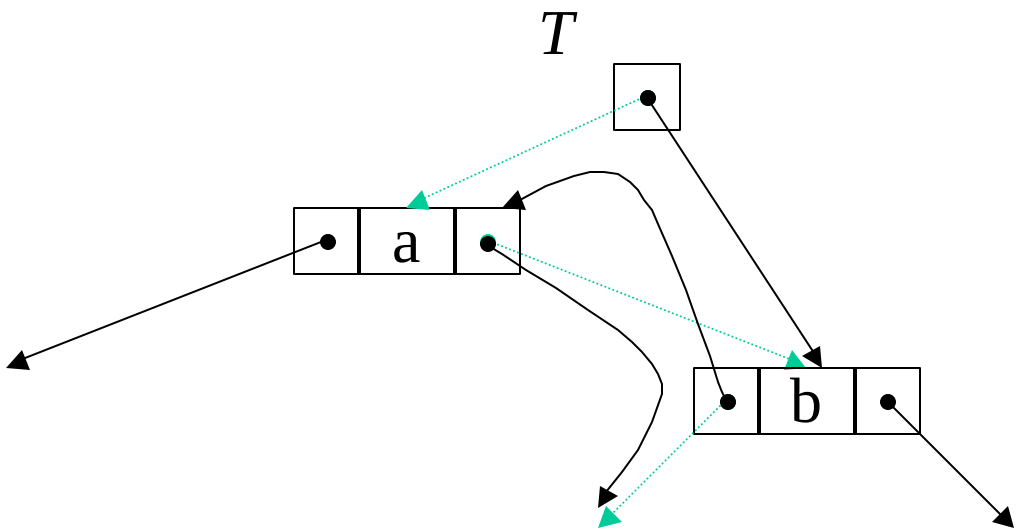
proc *RotacaoEsquerda* (var *Arvore T*) {

T, *T*^.*Dir*, *T*^.*Dir*^.*Esq* ← *T*^.*Dir*, *T*^.*Dir*^.*Esq*, *T*

AtualizaAltura (*T*^.*Esq*)

AtualizaAltura (*T*)

}



Inserção em árvores AVL

```
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)
}
```

Inserção em árvores AVL

```
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)
      }
}
```

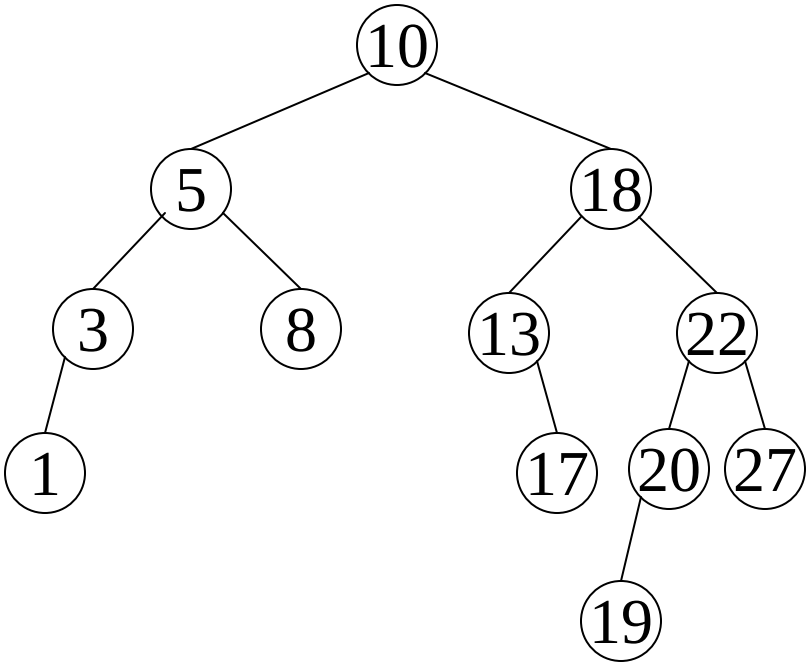
Análise do Algoritmo de Inserção em AVLs

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

Remoção em Árvores AVL

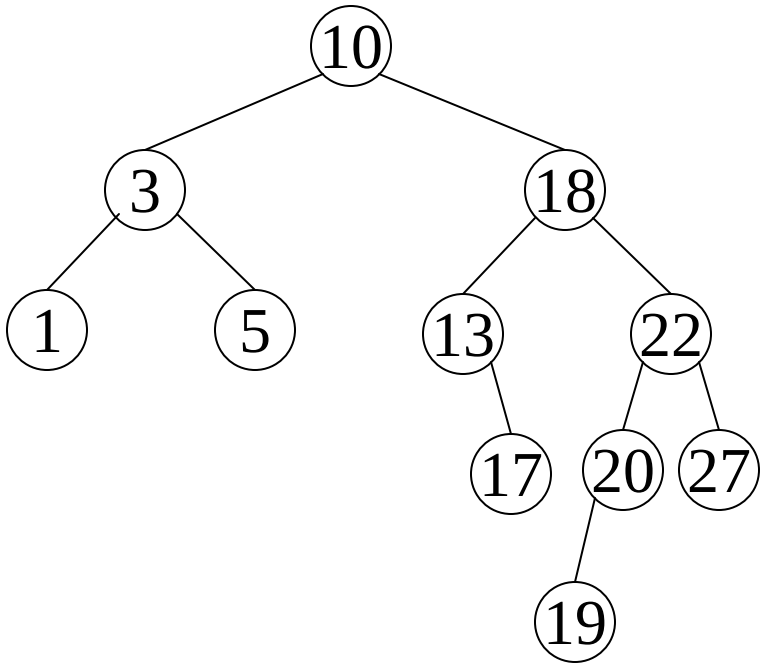
- Segue as mesmas linhas do algoritmo de inserção
  - Faz-se a remoção do nó como uma árvore de busca comum
  - Analisa-se a informação de balanceamento aplicando a rotação apropriada se for necessário
- Diferentemente da inserção, pode ser necessário realizar mais do que uma rotação
  - Na verdade, até log *n* rotações
  - Não afeta a complexidade do algoritmo de remoção que continua *O* (log *n*)

Remoção em Árvores AVL



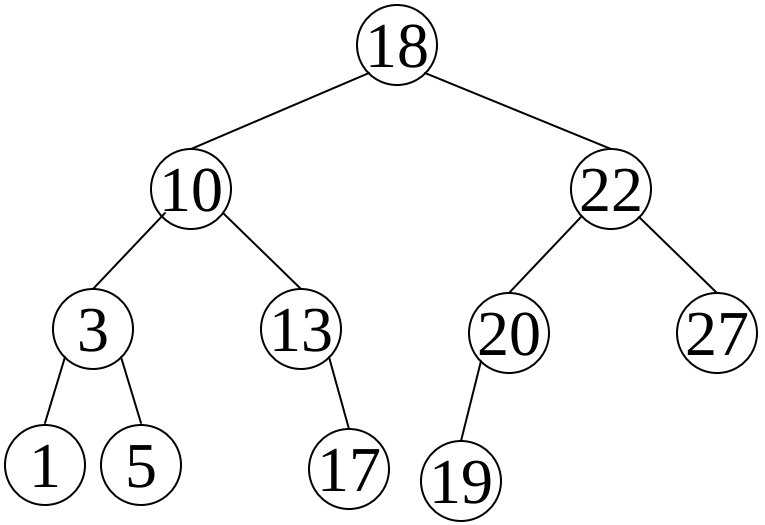
RemoverAVL (8, *T*)

Remoção em Árvores AVL



RemoverAVL (8, *T*)

Remoção em Árvores AVL



RemoverAVL (8, *T*)

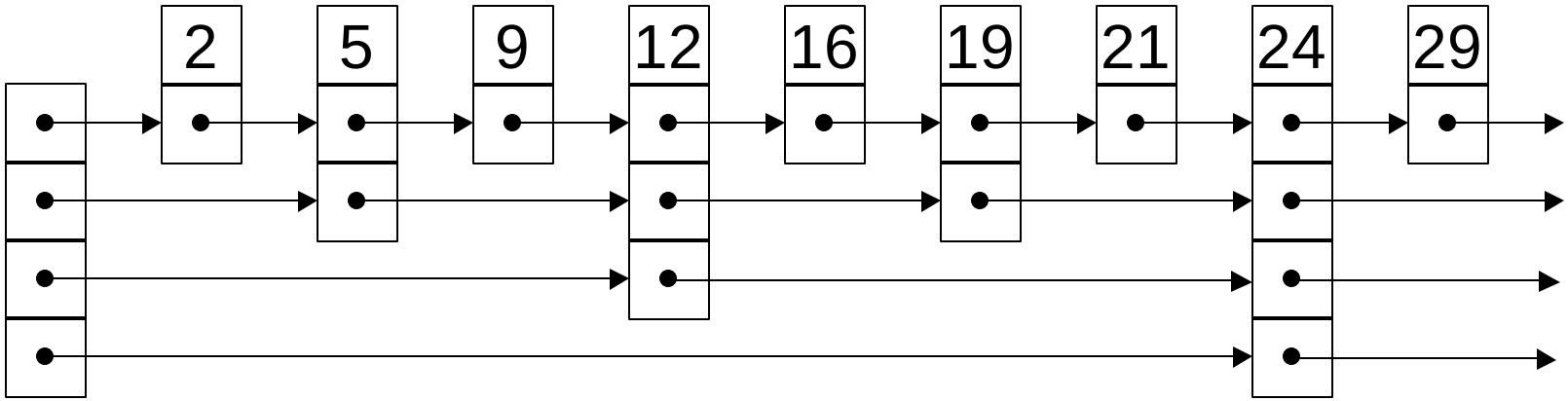
Skip Lists

- Uma das principais motivações para as árvores balanceadas como a AVL é que não se pode garantir que a inserção de nós se faz de maneira aleatória, o que garantiria altura logarítmica
- Em particular, pode-se imaginar que se a ordem de inserção é estipulada por um adversário, sua árvore sempre resultará balanceada. Ex.: Os nós são sempre dados em ordem crescente
- Uma estrutura de dados elegante que resolve esse problema sem apelar para algoritmos complicados é a Skip List
- A idéia é usar randomização de forma a impedir que o adversário de selecionar ordens de inserção ruins
- Na verdade, a probabilidade com que as Skip Lists acabam funcionando mal é MUITO pequena e diminui à medida que a lista aumenta

Skip Lists Determinísticas

- Uma Skip List é, até certo ponto, parecida com uma lista ordenada comum, entretanto, cada nó pode conter, além de 1 ponteiro para o próximo nó, vários outros ponteiros que o ligam a nós subseqüentes mais distantes
- Usando esses ponteiros adicionais, pode-se “pular” vários nós de forma a encontrar a chave procurada mais rápidamente
- Numa skip list determinística, existe uma cadeia de ponteiros ligando cada segundo nó da lista, cada quarto nó, cada oitavo nó, etc.
- A maneira de se implementar essa estrutura é usar em cada nó um array de ponteiros de tamanho variável

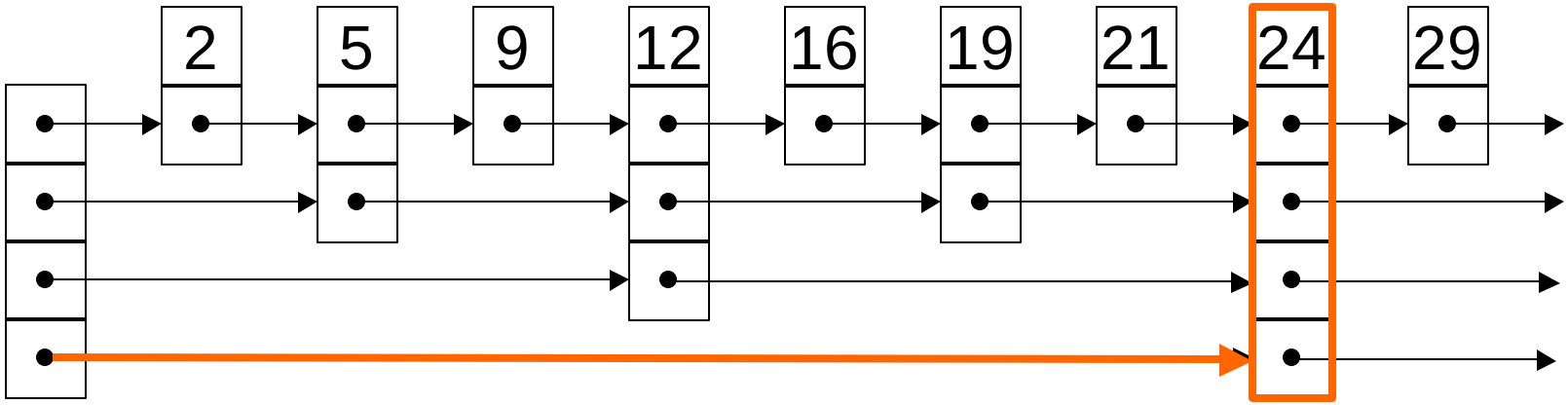
Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

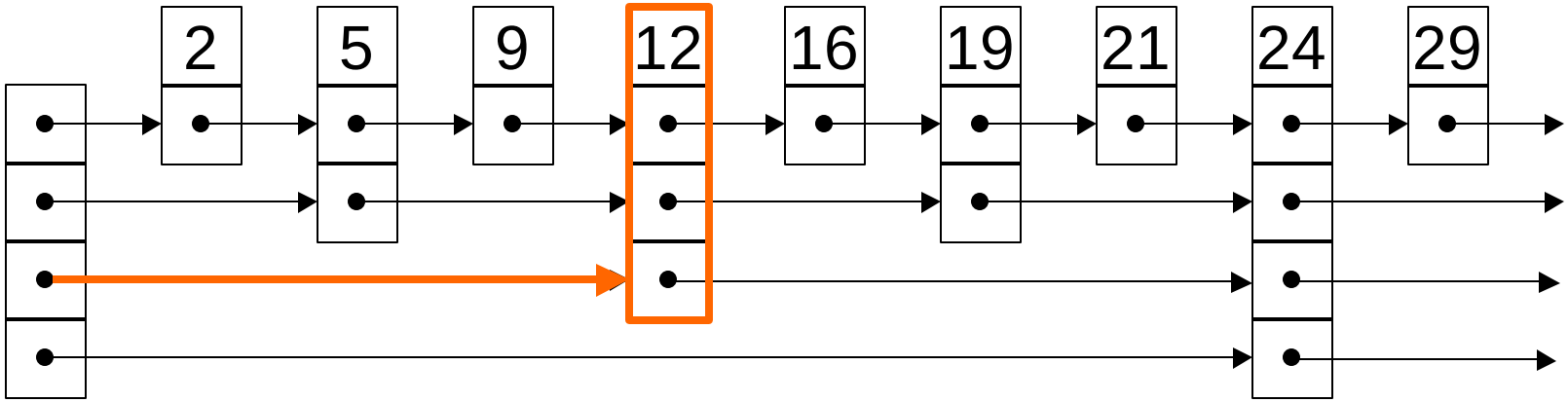
Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

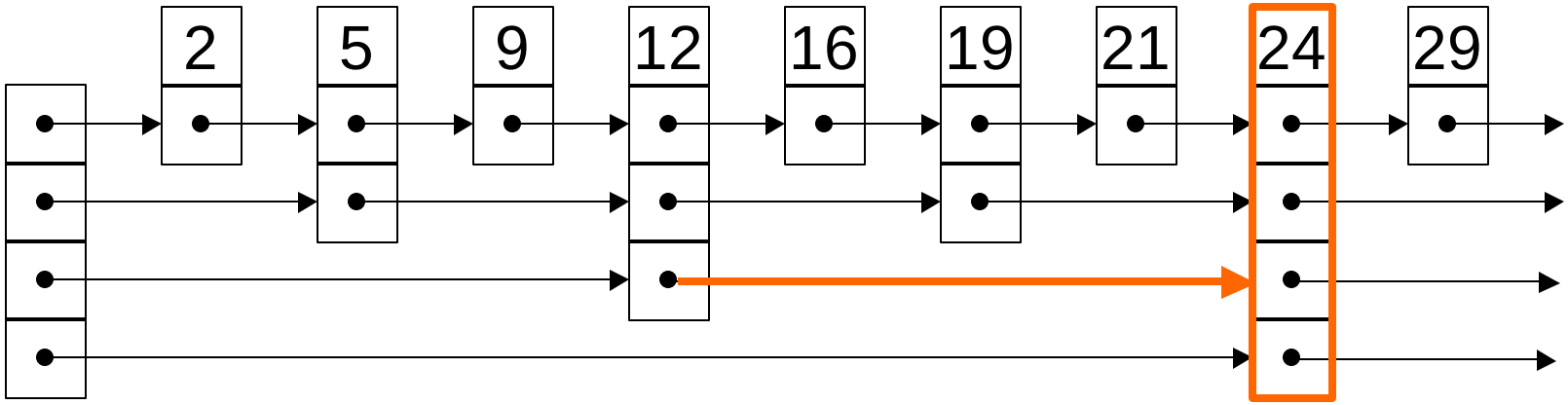
Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

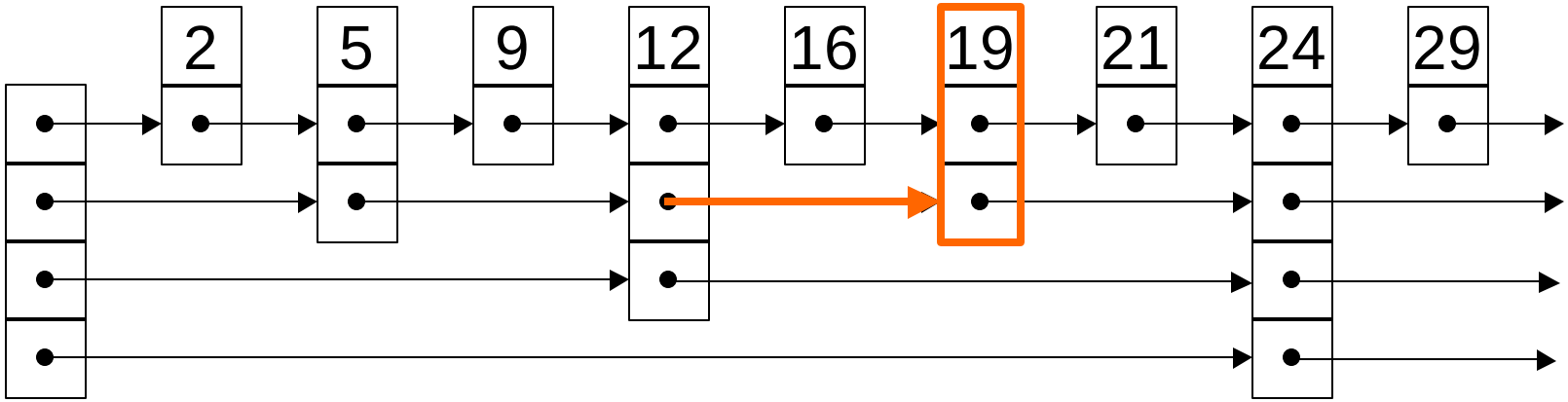
Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

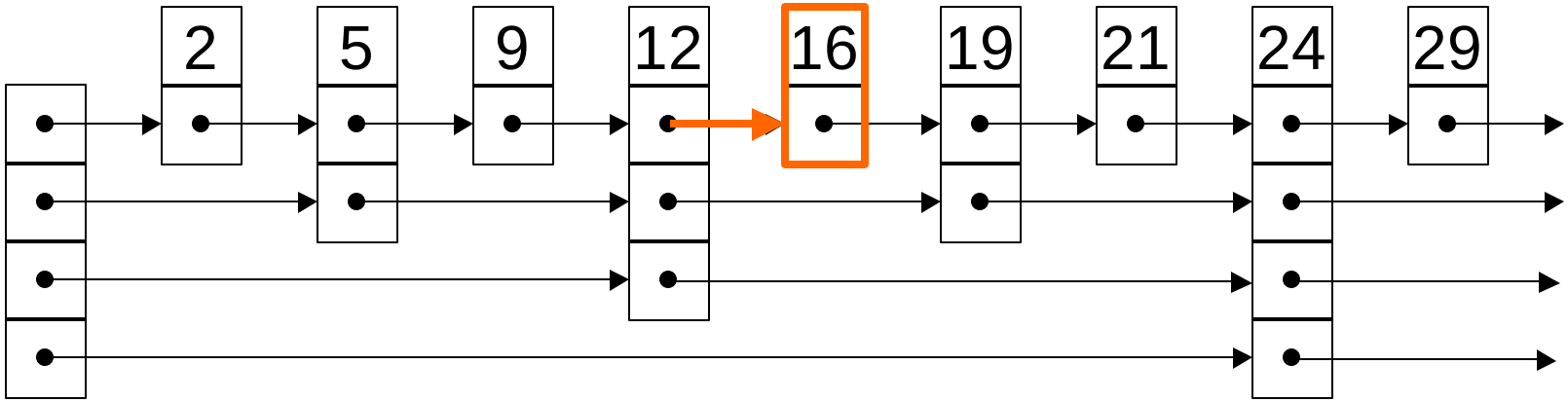
Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

Skip Lists Determinísticas



Cabeça
da Lista

Busca (16, *L*)

Skip Lists Randômicas

- Skip Lists determinísticas não são práticas já que a inserção ou remoção de um nó pode forçar diversos nós a mudarem de “altura”
- A solução é
  - Nunca mudar a altura de um nó já inserido
  - Ao inserir um novo nó, determinar sua altura randomicamente:
    - Fazer altura = 0
    - Repetir
      - » Incrementar altura
      - » Jogar uma moeda
    - Até a moda dar “cara”

Skip Lists Randômicas

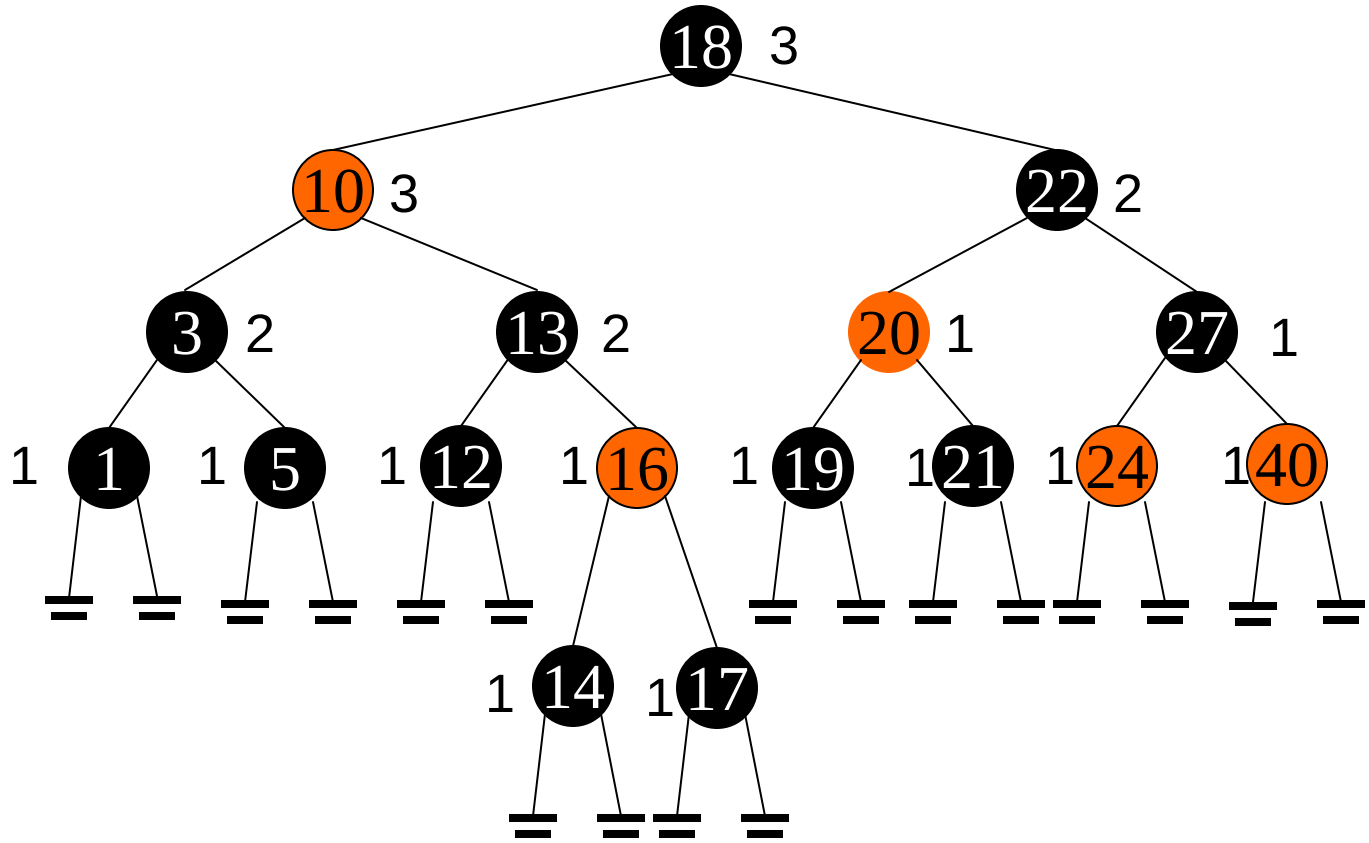
- Observe que, devido a termos usado uma moeda, dos n nós inseridos na lista, podemos esperar que
  - A metade terá altura 2
  - Um quarto terá altura 3
  - ...
  - No máximo 1 nó terá altura $\log_2 n$
- Ao realizar a busca, temos que comparar
  - 1 nó de altura $\log_2 n$
  - 2 nós de altura $(\log_2 n) - 1$
  - 2 nós de altura $(\log_2 n) - 2$
  - ...
  - 2 nós de altura 1
- Tempo esperado da busca é $O(\log_2 n)$

Árvores Rubro-Negras

- São árvores balanceadas segundo um critério ligeiramente diferente do usado em árvores AVL
- A todos os nós é associada uma cor que pode ser vermelha ou negra de tal forma que:
  1. Nós externos (folhas ou nulos) 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
- Observe que as propriedades acima asseguram que o maior caminho desde a raiz uma folha é no máximo duas vezes maior que o de o qualquer outro caminho até outra folha e portanto a árvore é aproximadamente balanceada

Árvores Rubro-Negras

- Altura negra de um nó = número de nós negros encontrados até qualquer nó folha descendente



Árvores Rubro-Negras

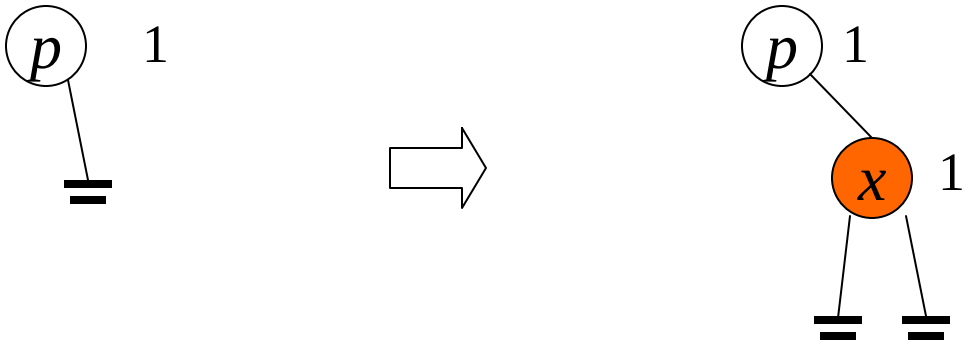
- Lema 1: Um nó x de uma árvore rubronegra tem no mínimo $2^{an(x)} - 1$ nós internos, onde an(x) é a altura negra de x
- Prova por indução
  - Caso base: Um nó de altura 0 (i.e., nó-folha) tem $0 = 2^0 - 1$ nós internos
  - Caso genérico: Um nó x de altura $h > 0$ tem 2 filhos com altura negra an(x) ou an(x) – 1, conforme x seja vermelho ou negro. No pior caso, x é negro e as subárvores enraizadas em seus 2 filhos têm $2^{an(x)-1} - 1$ nós internos cada e x tem $2(2^{an(x)-1} - 1)+1 = 2^{an(x)} - 1$ nós internos

Árvores Rubro-Negras

- Lema 2: Uma árvore rubro-negra com n nós tem no máximo altura $2 \log_2 (n+1)$
  - Prova: Se uma árvore tem altura h , a altura negra de sua raiz será no mínimo $h/2$ (pelo critério 3 de construção) e a árvore terá $n \geq 2^{h/2} - 1$ nós internos (Lema 1)
- Como conseqüência, a árvore tem altura $O(\log n)$ e as operações de busca, inserção e remoção podem ser feitas em $O(\log n)$

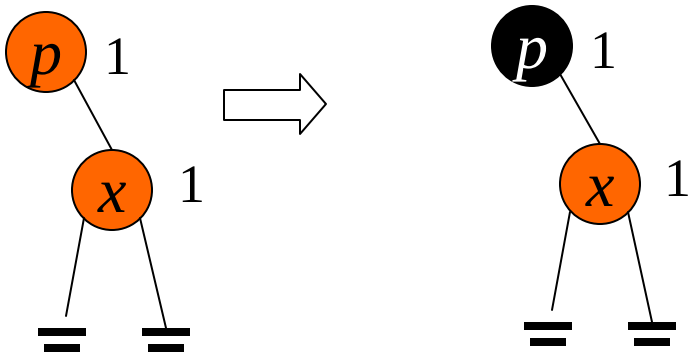
Inserção em Árvore Rubro-Negra

- Ao contrário da árvore AVL, agora temos agora vários critérios para ajustar simultaneamente
- Ao inserir um nó *x* numa posição vazia da árvore (isto é, no lugar de um nó nulo) este é pintado de vermelho. Isto garante a manutenção do critério (2), já que um nó vermelho não contribui para a altura negra



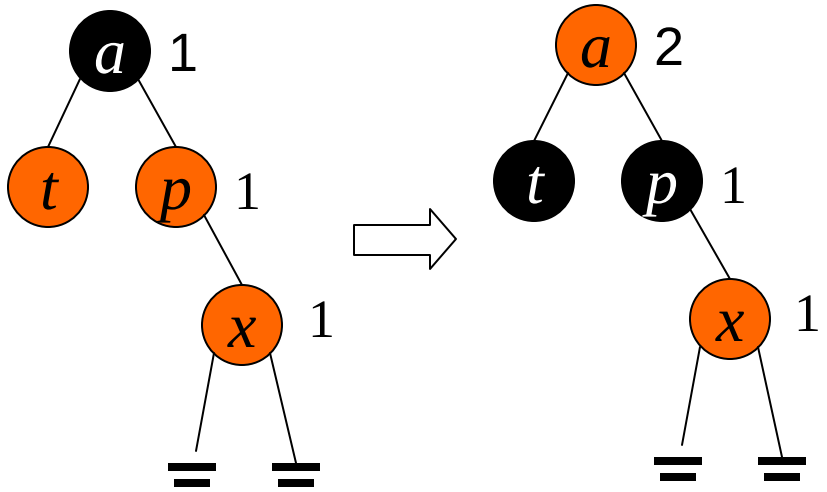
Inserção em Árvore Rubro-Negra

- [Caso 0] Se *x* não tem pai ou se *p*, o pai de *x*, é negro, nada mais precisa ser feito já que o critério (3) também foi mantido
- [Caso 1] Suponha agora que *p* é vermelho. Então, se *p* não tem pai, então *p* é a raiz da árvore e basta trocar a cor de *p* para negro



Inserção em Árvore Rubro-Negra

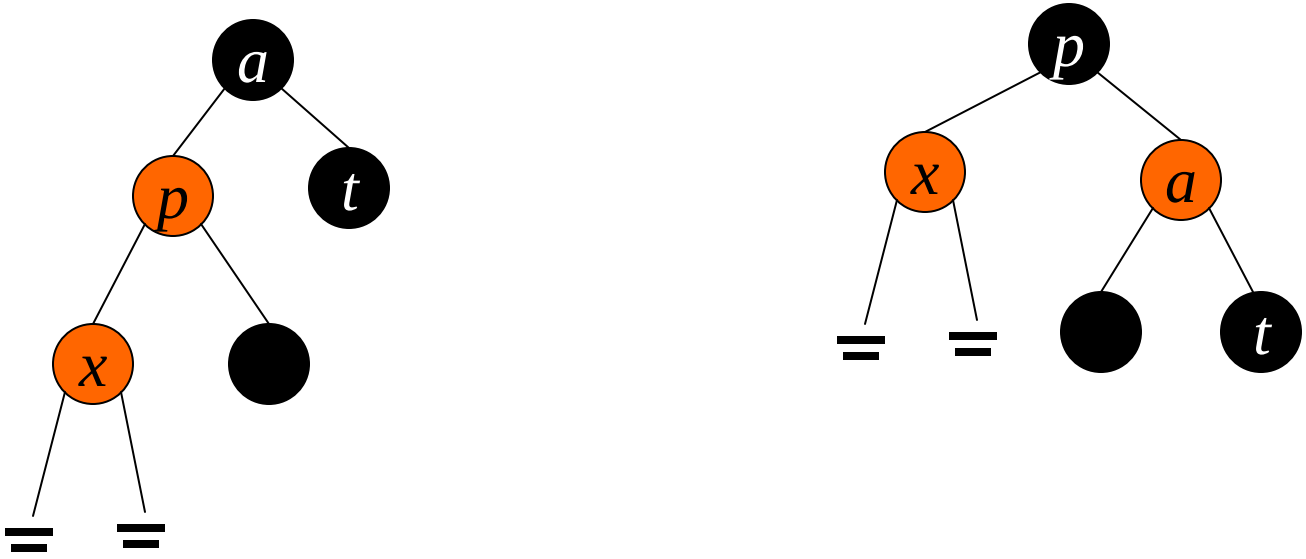
- [Caso 2] Suponha agora que *p* é vermelho e *a*, o pai de *p* (e avô de *x*) é preto. Se *t*, o irmão de *p* (tio de *x*) é vermelho, ainda é possível manter o critério (3) apenas fazendo a recoloração de *a*, *t* e *p*



Obs.: Se o pai de *a* é vermelho, o rebalanceamento tem que ser feito novamente

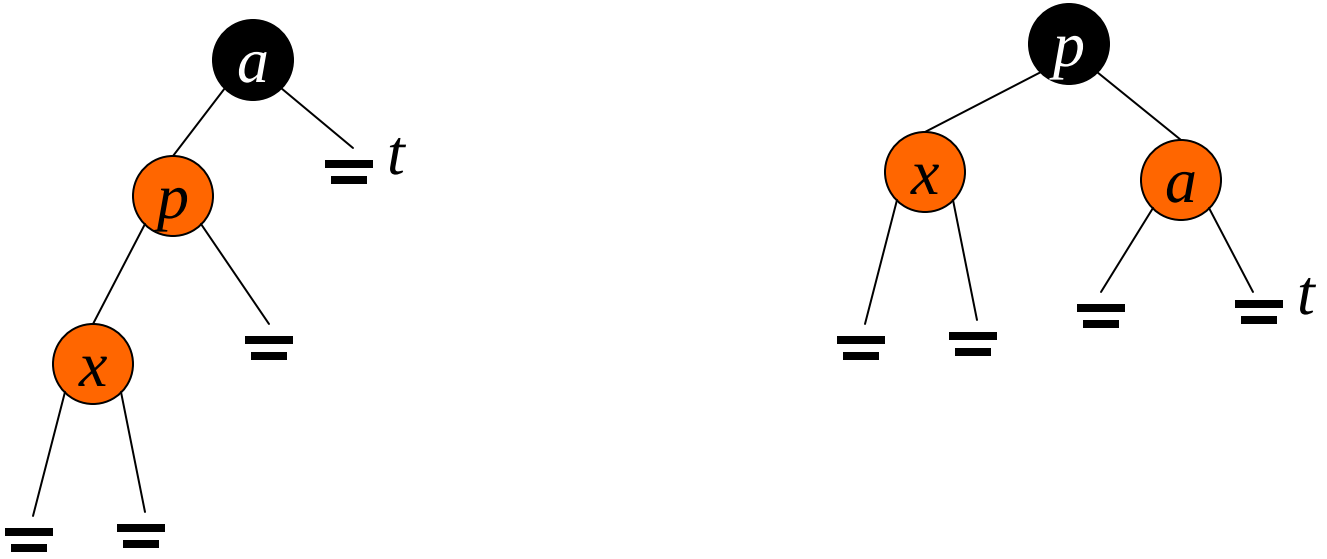
Inserção em Árvore Rubro-Negra

- [Caso 3] Finalmente, suponha que *p* é vermelho, seu pai *a* é preto e seu irmão *t* é preto. Neste caso, para manter o critério (3) é preciso fazer rotações envolvendo *a*, *t*, *p* e *x*. Há 4 subcasos que correspondem às 4 rotações possíveis:
  - [Caso 3a] Rotação Direita



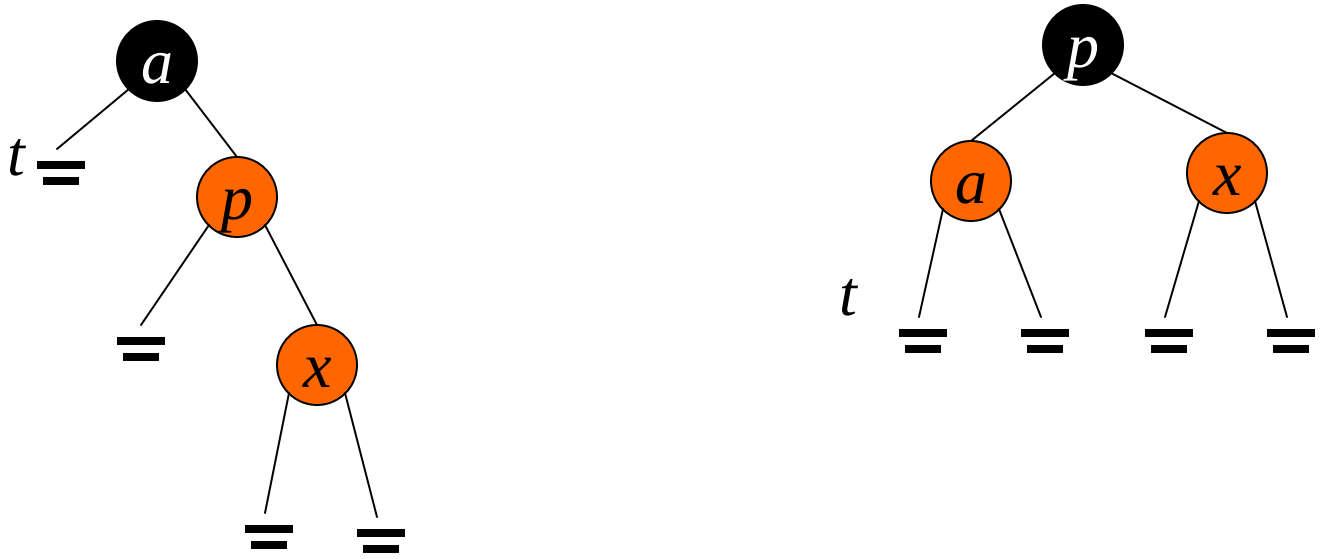
Inserção em Árvore Rubro-Negra

- [Caso 3] Finalmente, suponha que *p* é vermelho, seu pai *a* é preto e seu irmão *t* é preto. Neste caso, para manter o critério (3) é preciso fazer rotações envolvendo *a*, *t*, *p* e *x*. Há 4 subcasos que correspondem às 4 rotações possíveis:
  - [Caso 3a] Rotação Direita



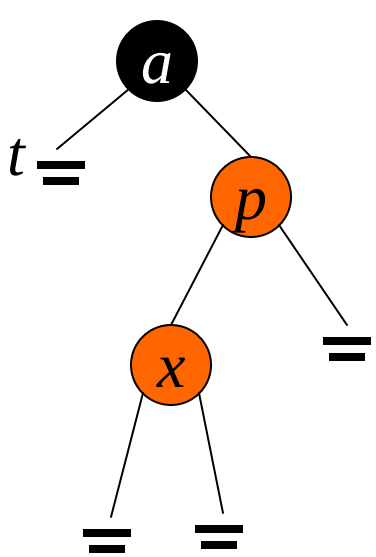
Inserção em Árvore Rubro-Negra

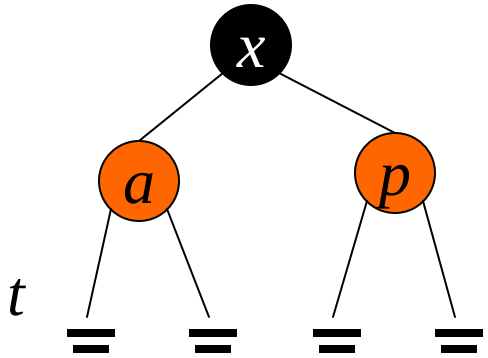
- [Caso 3b] Rotação Esquerda



Inserção em Árvore Rubro-Negra

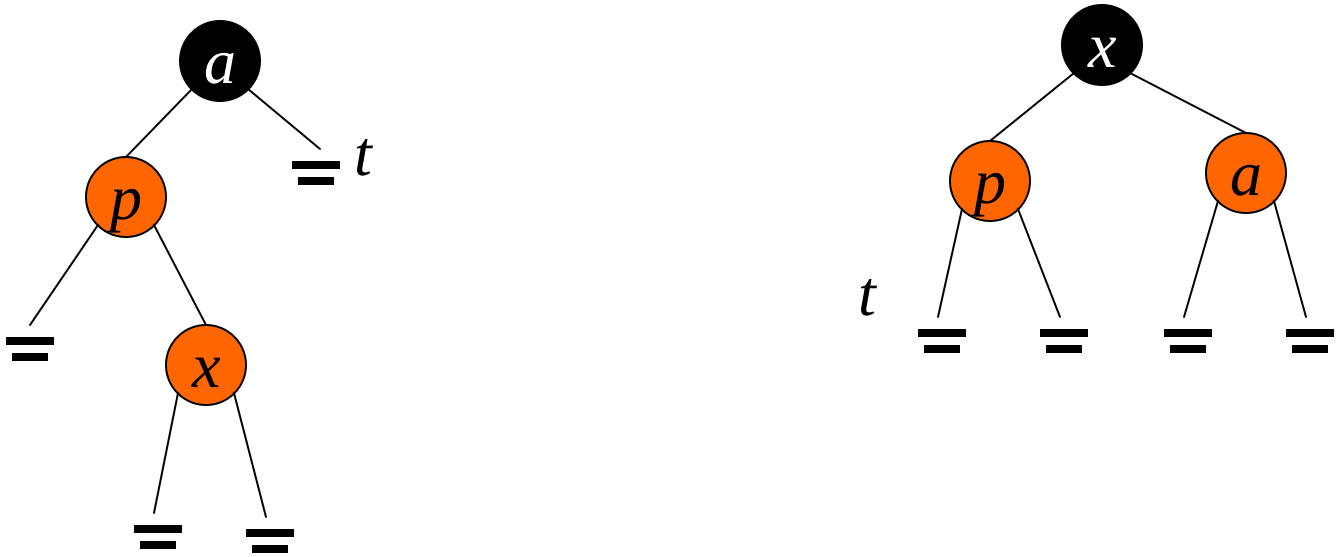
- [Caso 3c] Rotação Dupla Esquerda





Inserção em Árvore Rubro-Negra

- [Caso 3d] Rotação Dupla Direita



Inserção em Árvore Rubro-Negra

```
proc InsereRN (Chave v, var ArvoreRN a, p, x) {
    se x = Nulo então {
        x ← Aloca (NoArvoreRN)
        x^.Esq, x^.Dir, x^.Val, x^.Cor ← Nulo, Nulo, v, Vermelho
    } senão {
        se v < x^.val então
            InsereRN (v, p, x, x^.Esq)
        senão se v > x^.val então
            InsereRN (v, p, x, x^.Dir)
    }
    Rebalanceia (a, p, x)
}
```

Inserção em Árvore Rubro-Negra

```
proc Rebalanceia (var ArvoreRN a, p, x) {
    se x^.Cor = Vermelho e p ≠ Nulo então
        se p^.Cor = Vermelho então
            se a = Nulo então                                   % Caso 1
                p^.cor := Negro
            senão se a^.Cor = Negro então
                se a^.Esq ≠ Nulo e a^.Dir ≠ Nulo e a^.Esq^.Cor = Vermelho
                    e a^.Dir^.Cor = Vermelho então              % Caso 2
                    a^.Cor, a^.Esq^.Cor, a^.Dir^.Cor ← Vermelho, Negro, Negro
                senão
                    se p = a^.Esq
                        se x = p^.Esq então RotacaoDireita (a)  % Caso 3a
                        senão RotacaoDuplaDireita (a)           % Caso 3d
                    senão
                        se x = p^.Dir então RotacaoEsquerda (a) % Caso 3b
                        senão RotacaoDuplaEsquerda (a)          % Caso 3c
```

Inserção em Árvore Rubro-Negra

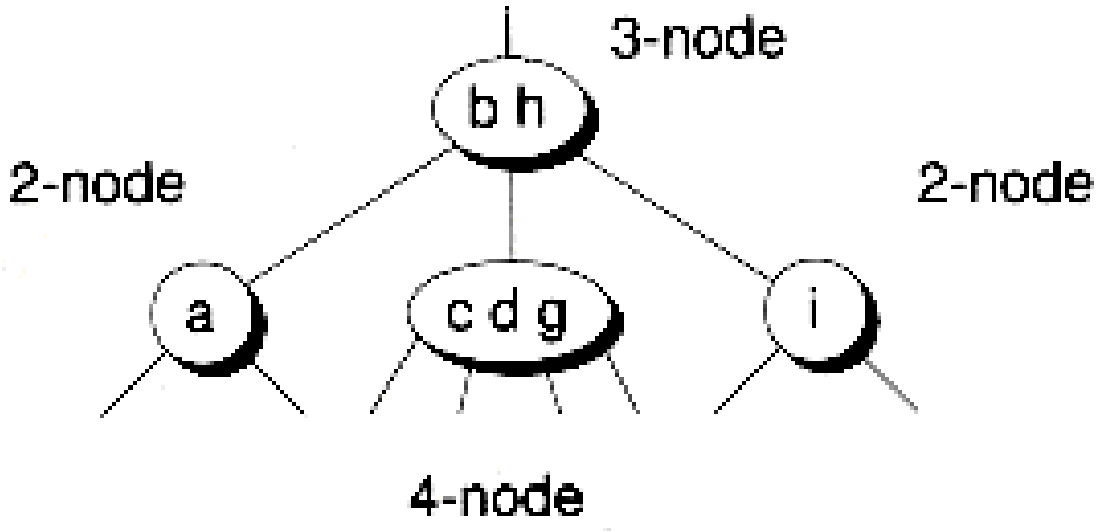
```
proc RotacaoDireita (var ArvoreRN T) {
    T, T^.Esq, T^.Esq^.Dir ← T^.Esq, T^.Esq^.Dir, T
    T^.Cor, T^.Dir^.Cor ←  T^.Dir^.Cor, T^.Cor
}
proc RotacaoEsquerda (var ArvoreRN T) {
    T, T^.Dir, T^.Dir^.Esq ← T^.Dir, T^.Dir^.Esq, T
    T^.Cor, T^.Esq^.Cor ←  T^.Esq^.Cor, T^.Cor
}
proc RotacaoDuplaDireita (var ArvoreRN T) {
    RotacaoEsquerda (T^.Esq)
    RotacaoDireita (T)
}
proc RotacaoDuplaEsquerda (var ArvoreRN T) {
    RotacaoDireita (T^.Dir)
    RotacaoEsquerda (T)
}
```

Complexidade da Inserção em Árvore Rubro-Negra

- *Rebalanceia* tem custo $O(1)$
- *RotacaoXXX* têm custo $O(1)$
- *InsereRN* tem custo $O(\log n)$

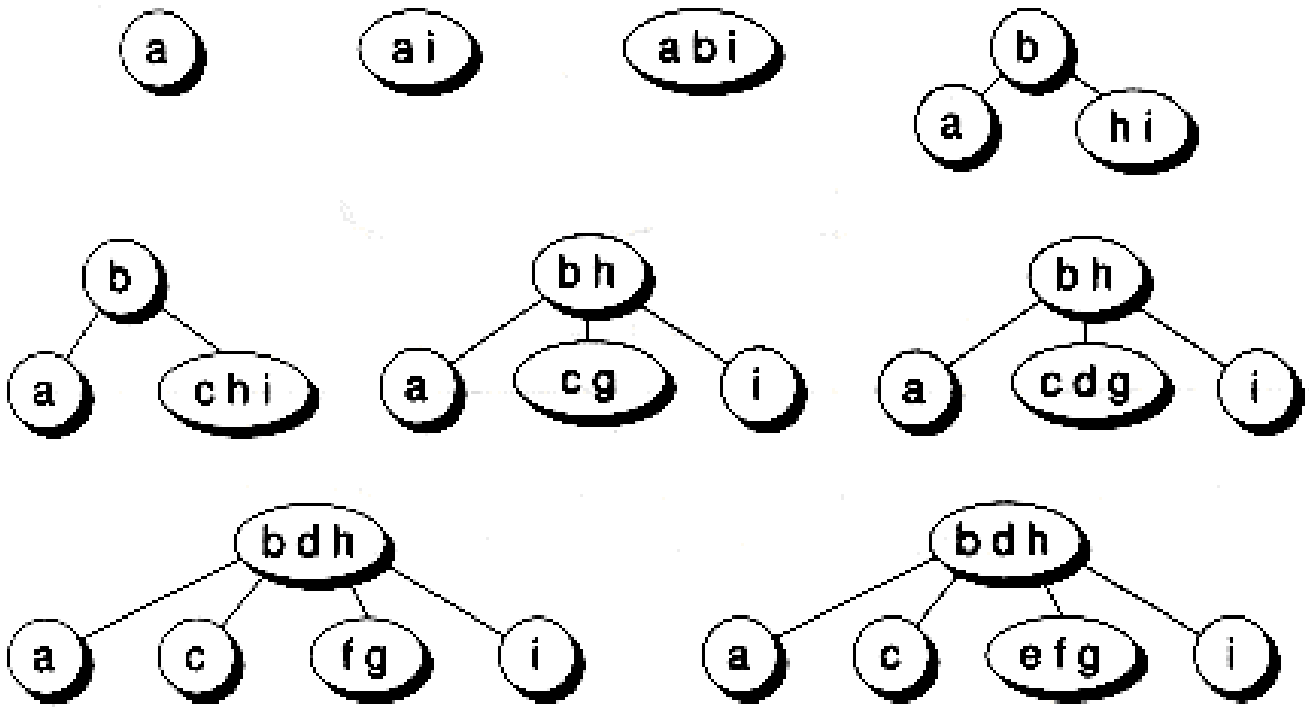
Árvores 2-3-4

- É uma árvore onde cada nó pode ter mais do que uma chave
- Na verdade, uma árvore 2-3-4 é uma árvore B onde a capacidade de cada nó é de até 3 chaves (4 ponteiros)



Árvores 2-3-4

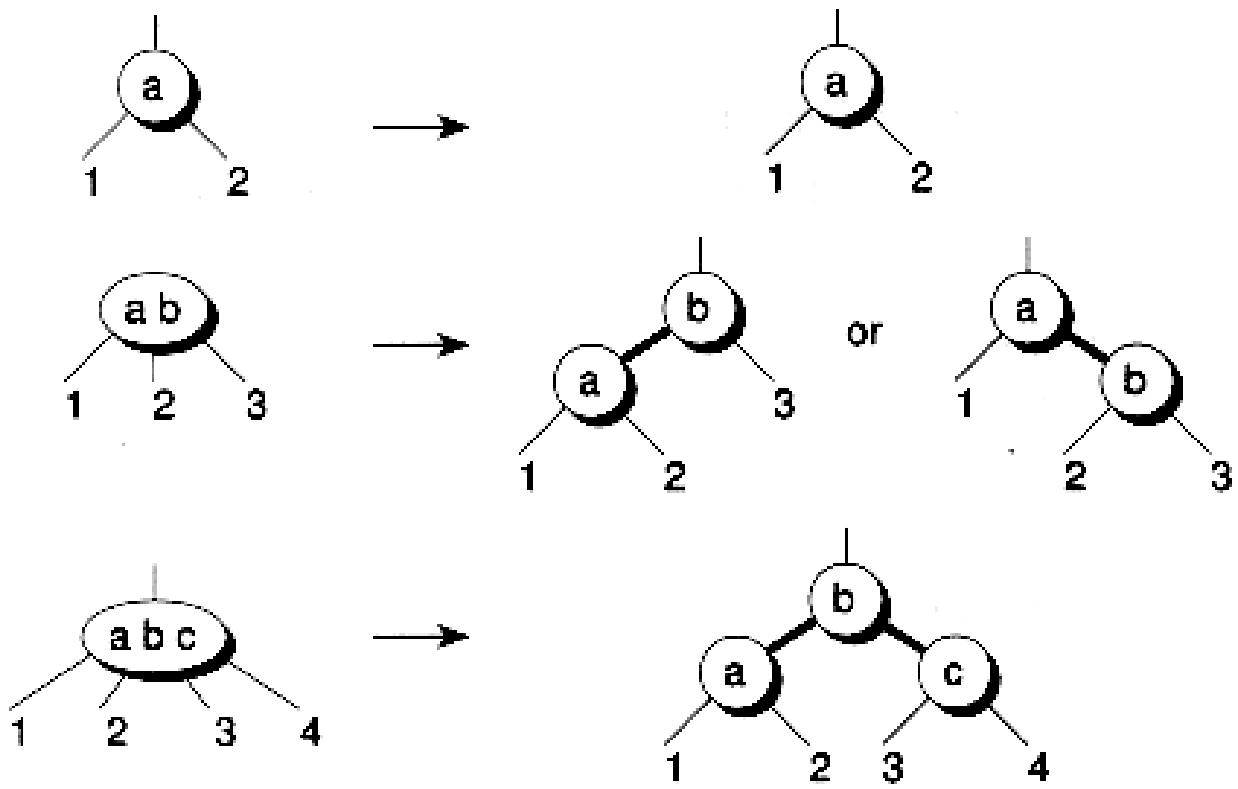
- Exemplo de inserção em árvores 2-3-4



Insertion sequence: a-i-b-h-c-g-d-f-e

Árvores 2-3-4

- Na verdade, uma árvore 2-3-4 pode ser implementada como uma árvore binária Rubro-Negra



Árvores de Difusão (Splay)

- Vimos:
  - Árvores de Busca: garantem inserção / remoção/busca em tempo logarítmico no caso médio mas não no pior caso
  - Árvores AVL / Rubro-Negra: garantem inserção / remoção/busca em tempo logarítmico no pior caso
    - Precisamos guardar informação adicional em cada nó (altura da árvore/cor)
  - Skip Lists: garantem inserção / remoção/busca em tempo logarítmico no caso médio e que o pior caso nunca pode ser adivinhado por um adversário, Probabilisticamente, o pior caso fica menos provável à medida que *n* aumenta
- Árvores de Difusão: Garantem boa performance *amortizada*

Árvores de Difusão

- São árvores binárias de busca cujo desempenho *amortizado* é ótimo
  - Começando com uma árvore vazia, o tempo total para realizar qualquer seqüência de m operações de inserção/remoção/busca é $O(m \log n)$, onde n é o maior número de nós alcançado pela árvore
- Custo (desempenho) amortizado não é o mesmo que custo médio
  - Custo médio leva em conta todas as possíveis seqüências de operações e o limite é obtido na média
  - Custo amortizado é obtido para qualquer seqüência de operações (o adversário pode obrigar uma ou outra operação a ter má performance, mas não todas)

Árvores de Difusão

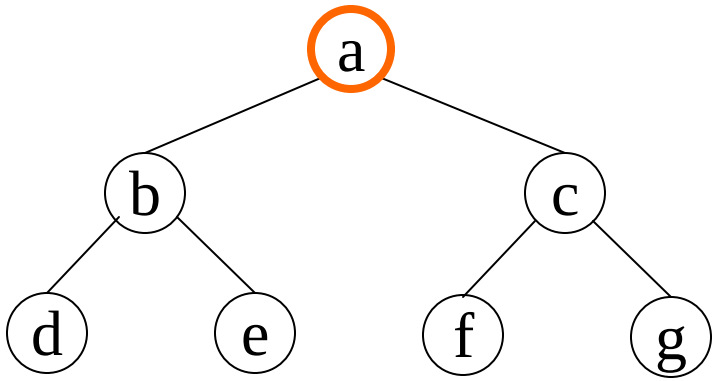
- Uma árvore de difusão não guarda informações sobre o balanceamento das subárvores
- É uma estrutura auto-ajustável, isto é, cada operação que é executada com mau desempenho “rearruma” a estrutura de forma a garantir que a mesma operação, se repetida, seja executada eficientemente
- Na árvore de difusão, 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*)

Árvores de Difusão

- Em nossa implementação, para simplificar, assumiremos que *splay* (*x*, *T*) só é chamada quando se pode garantir que *x* pertence à árvore
  - Após uma busca bem sucedida de *x* em *T*
  - Após uma inserção de *x* em *T*
  - Como toda remoção é precedida de uma busca, se esta for bem sucedida, chama-se *splay* (*x*, *T*) antes de remover *x* de *T*
  - Alternativamente, quando *x* é removido de *T*, chama-se *splay* (*y*, *T*), onde *y* é o pai de *x* (caso haja)
- O efeito de chamar *splay* (*x*, *T*) é mover o nó *x* para a raiz da árvore *T* por meio de uma série de rotações efetuadas de baixo para cima na árvore (das folhas para a raiz)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

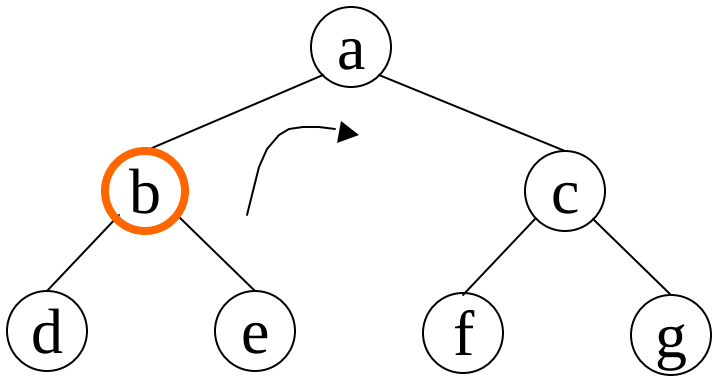


Caso 1: *x* = a (raiz)

Nada a fazer

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

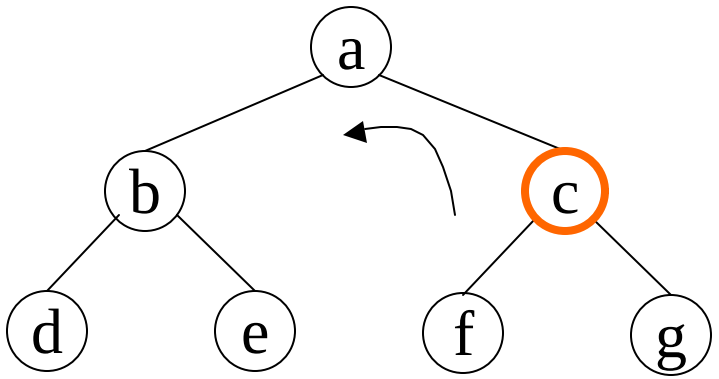


Caso 2.1: *x* = b (filho esquerdo)

RotaçãoDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

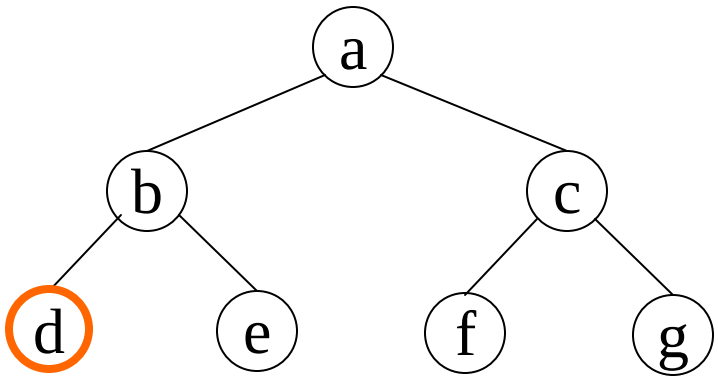


Caso 2.2: *x* = c (filho direito)

RotaçãoEsquerda (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

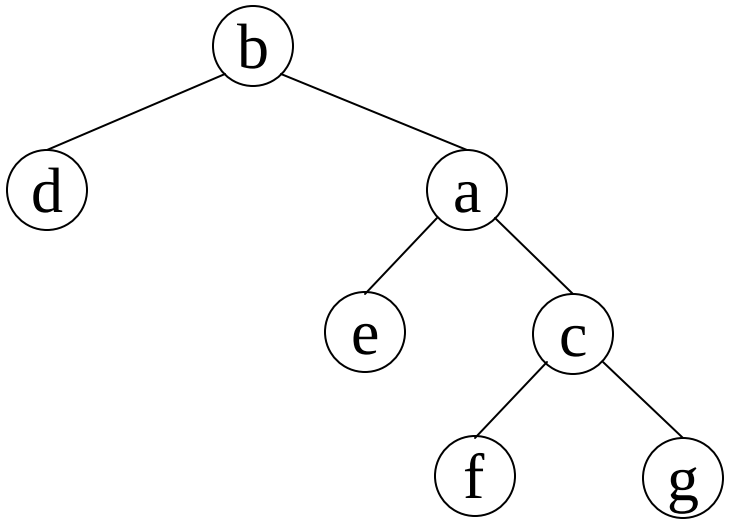


Caso 3.1: *x* = d
(neto esquerdo / esquerdo)

RotaçãoDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

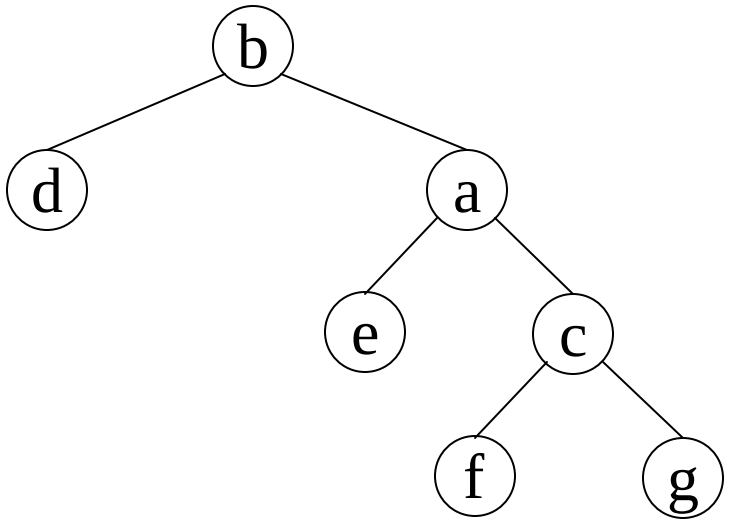


Caso 3.1: *x* = d
(neto esquerdo / esquerdo)

RotaçãoDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis



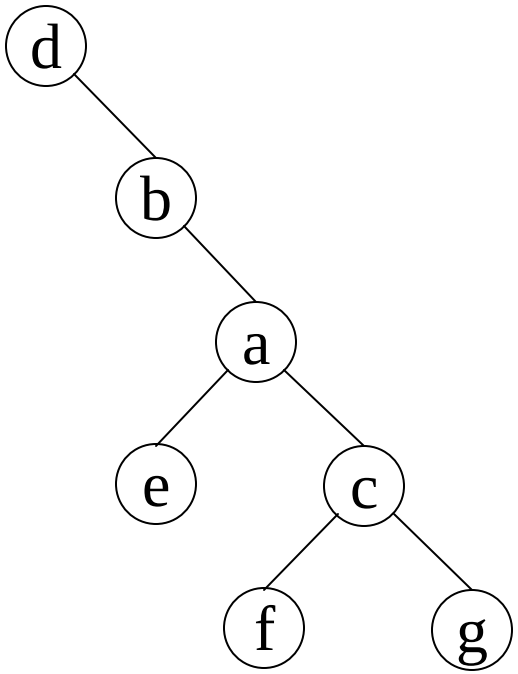
Caso 3.1: *x* = d
(neto esquerdo / esquerdo)

RotaçãoDireita (*T*)

RotaçãoDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis



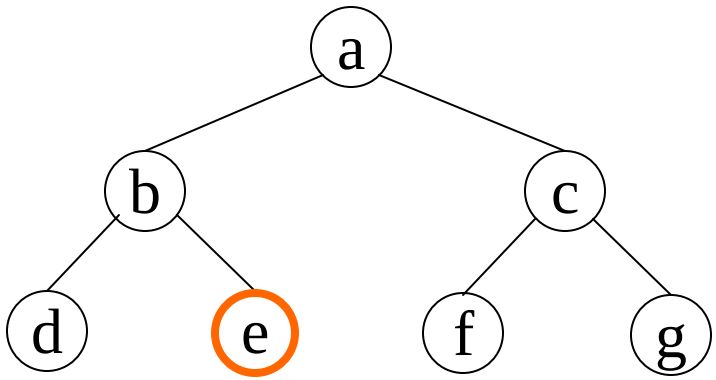
Caso 3.1: *x* = d
(neto esquerdo / esquerdo)

RotaçãoDireita (*T*)

RotaçãoDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

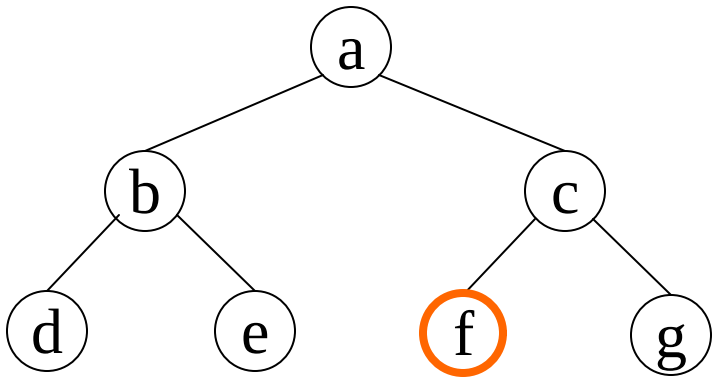


Caso 3.2: *x* = e
(neto esquerdo / direito)

RotaçãoDuplaDireita (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

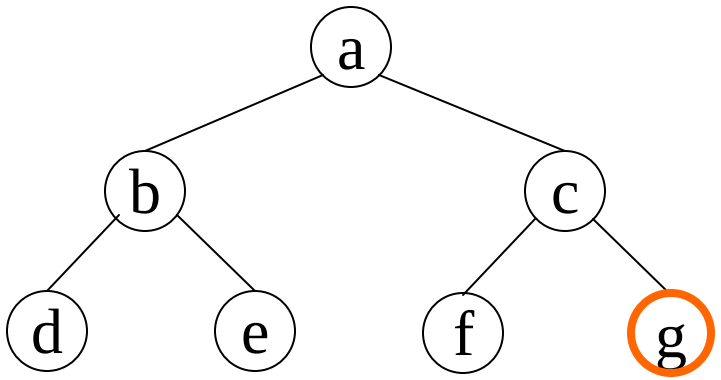


Caso 3.3: *x* = f
(neto direito / esquerdo)

RotaçãoDuplaEsquerda (*T*)

Árvores de Difusão

- Antes de vermos o procedimento *splay* (*x*, *T*) vamos analisar um procedimento mais simples considerando que a árvore *T* tem apenas três níveis

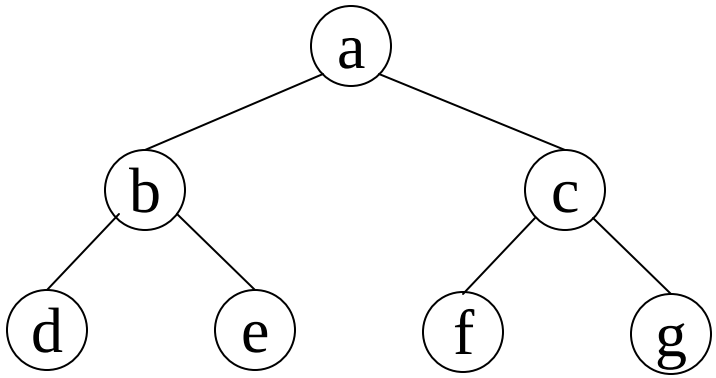


Caso 3.4: *x* = g
(neto direito / direito)

RotaçãoEsquerda (*T*)
RotaçãoEsquerda (*T*)

Árvores de Difusão

- Se nenhum desses nós é *x*, então *x* descende de d, e, f, ou g
  - Após aplicarmos o procedimento recursivamente à subárvore enraizada em d, e, f ou g, *x* passará a ser a raiz daquela subárvore
  - Basta então aplicar as rotações como vimos anteriormente



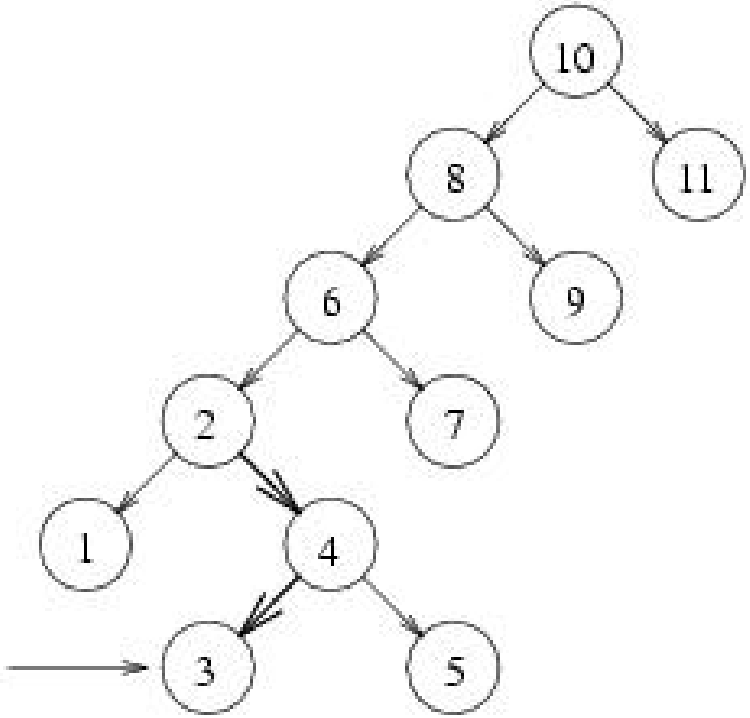
Árvores de Difusão

```
proc Splay (Chave x, var T Arvore) {
    se T^.Val = x então retornar
    se x < T^.Val então {
        se x = T^.Esq^.Val então RotacaoDireita (T)
        senão se x < T^.Esq^.Val
            então {
                se x ≠ T^.Esq^.Esq^.Val então Splay (x, T^.Esq^.Esq)
                RotacaoDireitaDireita (T)
            } senão {
                se x ≠ T^.Esq^.Dir^.Val então Splay (x, T^.Esq^.Dir)
                RotacaoDuplaDireita (T)
            }
    }  senão {  % x > T^.Val
        ...
    }
}
```

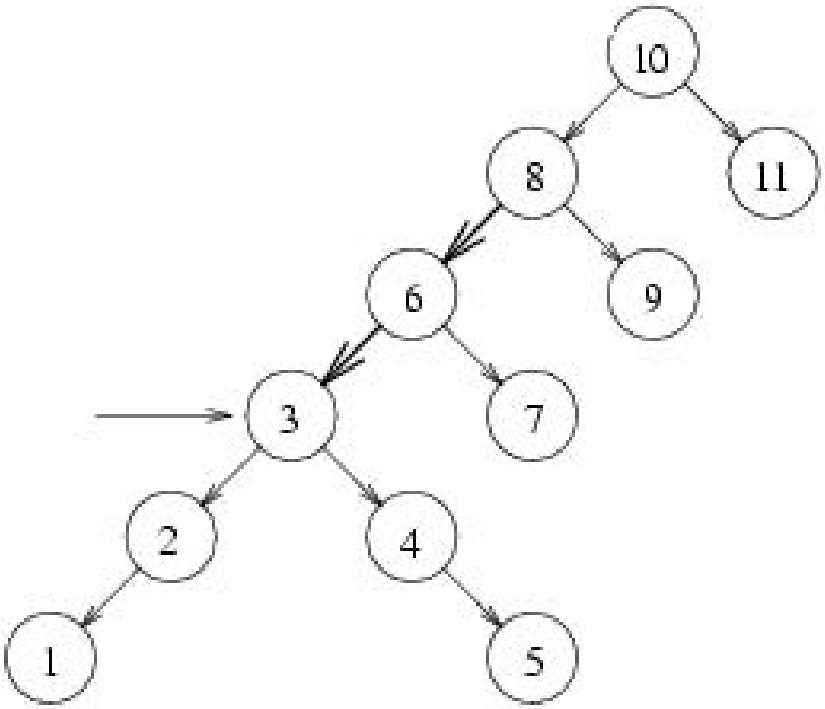
Árvores de Difusão

```
proc Splay (Chave x, var T Arvore) {
    se T^.Val = x então retornar
    se x < T^.Val então {
      . . .
    }  senão {      % x > T^.Val
        se x = T^.Dir^.Val then RotacaoEsquerda (T)
        senão se x > T^.Dir^.Val
            então {
                se x ≠ T^.Dir^.Dir^.Val então Splay (x, T^.Dir^.Dir)
                RotacaoEsquerdaEsquerda(T)
            } senão {
                se x ≠ T^.Dir^.Esq^.Val então Splay (x, T^.Dir^.Esq)
                RotacaoDuplaEsquerda (T)
            }
      }
}
```

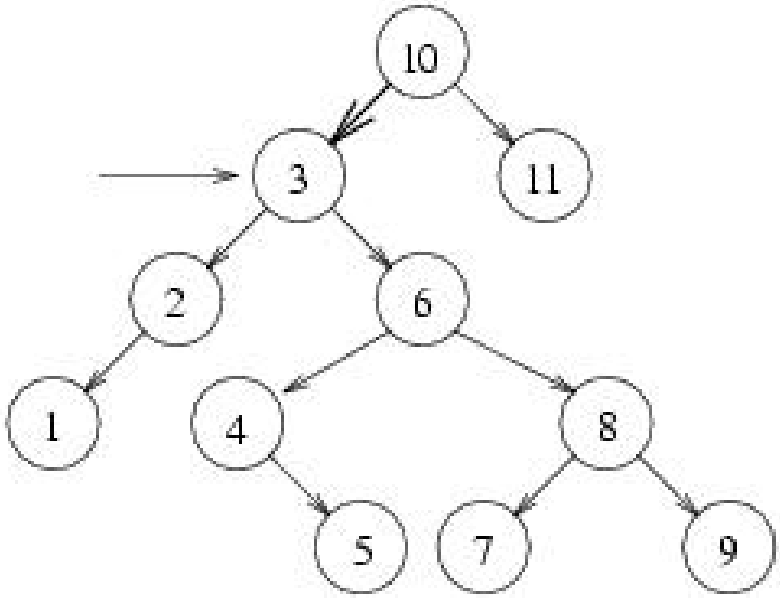
Exemplo de Difusão



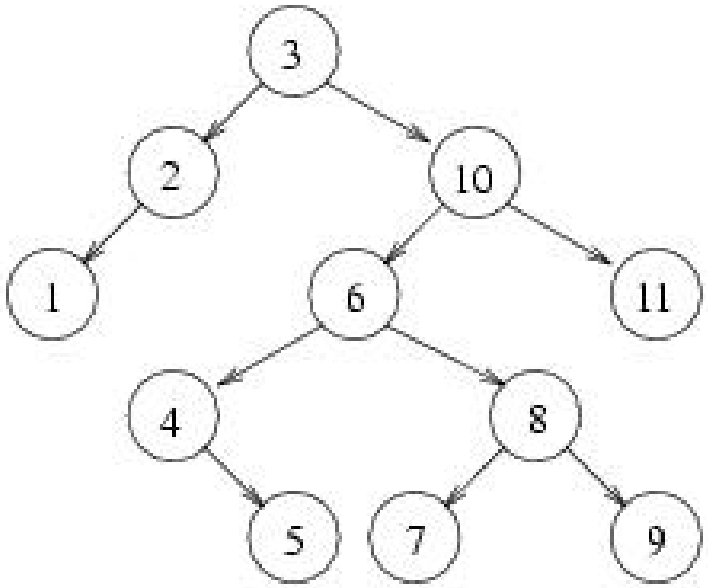
Exemplo de Difusão



Exemplo de Difusão



Exemplo de Difusão

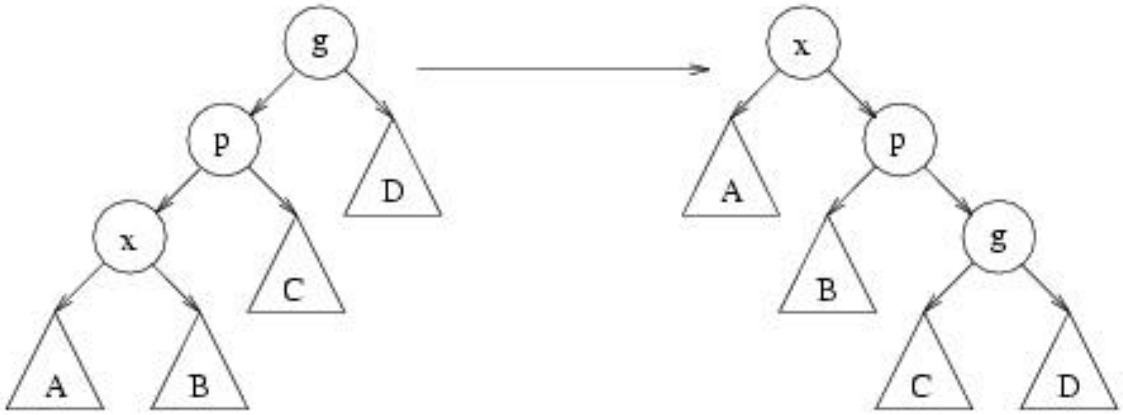


Complexidade de Árvores de Difusão

- Não faremos uma análise completa da complexidade amortizada de árvores de difusão (veja livro texto)
- É possível entretanto entender intuitivamente porque o custo amortizado é de $O(m \log n)$
- De forma geral, há dois fatores a considerar:
  - Custo real: tempo que a função *Splay* leva para completar seu trabalho. É proporcional ao nível de x na árvore
  - Melhora no balanceamento: O quanto a aplicação da função *Splay* melhora o balanceamento da árvore

Complexidade de Árvores de Difusão

- Considere o caso em que o elemento sendo buscado se encontra em um nível muito profundo da árvore (na subárvore A ou B abaixo, por exemplo)
  - Então, gastou-se muito tempo para fazer o splay
  - Entretanto, a subárvore A ou B, que necessáriamente contém muitos nós, agora se encontra mais próxima da raiz e a árvore agora está mais balanceada



Complexidade de Árvores de Difusão

- Considere o caso em que o elemento sendo buscado se encontra em um nível raso da árvore.
  - Então, gastou-se pouco tempo para fazer o splay
  - Se isso acontece sempre, a árvore está balanceada
- De forma geral, podemos ver que ou o splay não tem custo alto ou então ele contribui para melhorar o desempenho da árvore em futuras buscas

Listas de Prioridades

- Em muitas aplicações, dados de uma coleção são acessados por ordem de *prioridade*
- A prioridade associada a um dado pode ser qualquer coisa: tempo, custo, etc, mas precisa ser um escalar
- Nesse contexto, as operações que se costuma querer implementar eficientemente são
  - Seleção do elemento com maior (ou menor) prioridade
  - Remoção do elemento de maior (ou menor) prioridade
  - Inserção de um novo elemento

Listas de Prioridade

- Implementação

Operação	**Lista**	**Lista Ordenada**	**Árvore Balanceada**
Seleção	$O(n)$	$O(1)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$
Remoção (do menor)	$O(n)$	$O(1)$	$O(\log n)$
Alteração (de prioridade)	$O(n)$	$O(n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n \log n)$

Listas de Prioridade

- Implementação

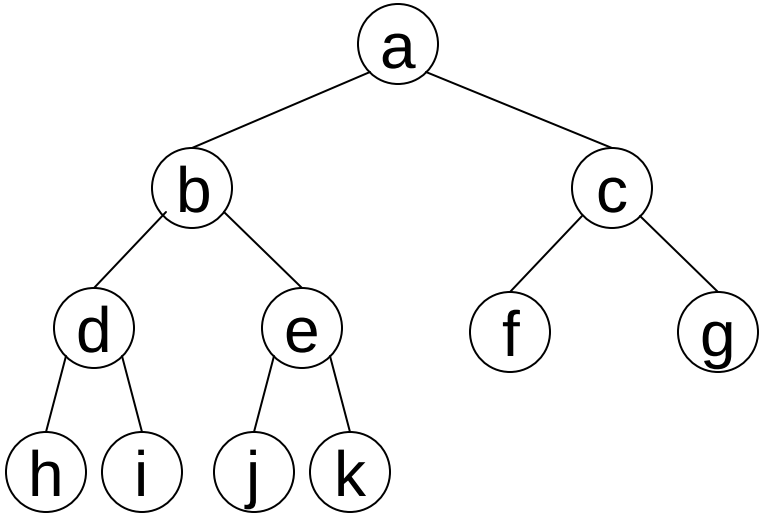
Operação	**Lista**	**Lista Ordenada**	**Árvore Balanceada**	**Heap**
Seleção	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Remoção (do menor)	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Alteração (de prioridade)	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

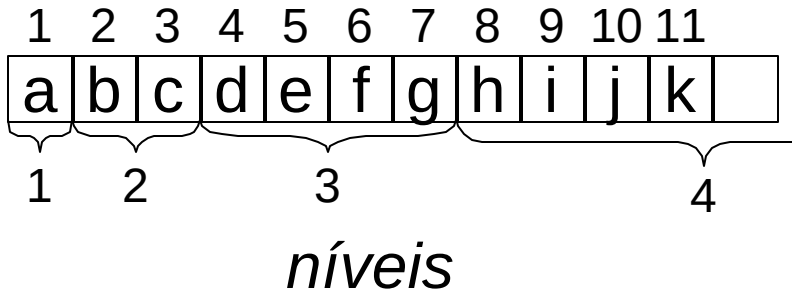
Heaps

- Estruturas próprias para implementação de listas de prioridade
- Podem ser pensadas como árvores binárias (mas não de busca) onde todos os nós possuem as propriedades
  - chave do nó ≤ chave do nó à esquerda (se houver)
  - chave do nó ≤ chave do nó à direita (se houver)
- Como essas propriedades valem para toda a árvore, a raiz contém a chave (prioridade) de menor valor
- Diferentemente de árvores binárias de busca, heaps são implementados usando *arrays*

Implementando Árvores Binárias com Arrays

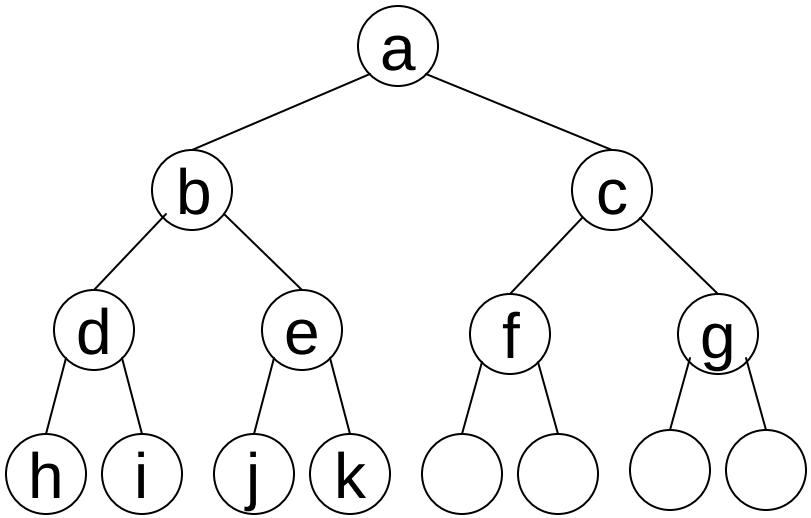
- Dado um nó armazenado no índice *i*, é possível computar o índice
  - do nó filho esquerdo de *i* : 2 *i*
  - do nó filho direito de *i* : 2 *i* + 1
  - do nó pai de *i* : *i* div 2
- Para armazenar uma árvore de altura *h* precisamos de um array de $2^h - 1$ (número de nós de uma árvore cheia de altura *h*)

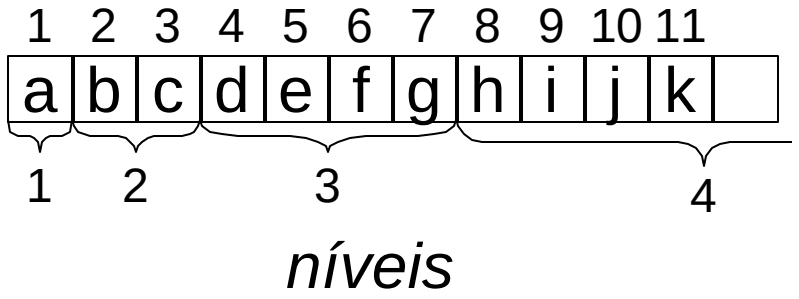




Implementando Árvores Binárias com Arrays

- Dado um nó armazenado no índice *i*, é possível computar o índice
  - do nó filho esquerdo de *i* : 2 *i*
  - do nó filho direito de *i* : 2 *i* + 1
  - do nó pai de *i* : *i* div 2
- Para armazenar uma árvore de altura *h* precisamos de um array de $2^h - 1$ (número de nós de uma árvore cheia de altura *h*)





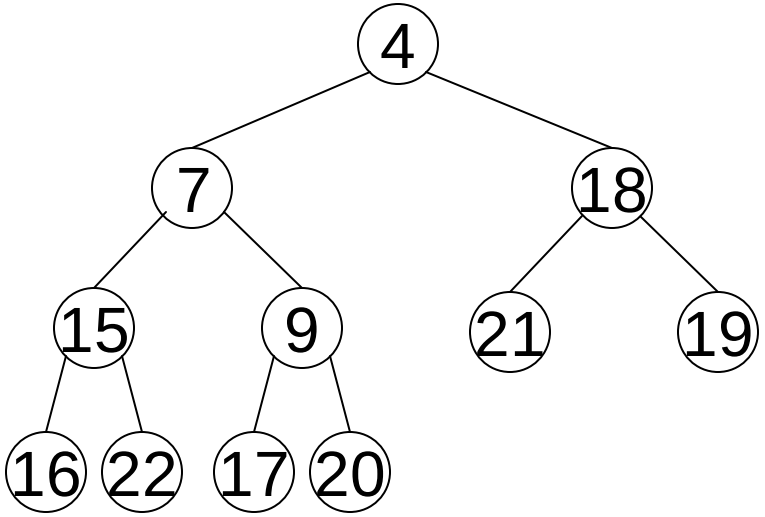
Alterando a Prioridade em Heaps

- Se um nó tem seu valor alterado, a manutenção das propriedades do Heap pode requerer que nó migre na árvore
  - para cima (se ele diminuir de valor)
  - para baixo (se ele aumentar de valor)
- Para cada uma dessas situações utiliza-se um algoritmo de migração:
  - *subir* (*i*, *n*, *H*) migra o nó *i* para cima no heap *H*
  - *descer* (*i*,*n*,*H*) migra o nó *i* para baixo no heap *H* (sendo *n* o número total de nós da árvore/heap)
- OBS.: Num heap os algoritmos de inserção e remoção mantêm os *n* nós da árvore nas *n* primeiras posições do array.

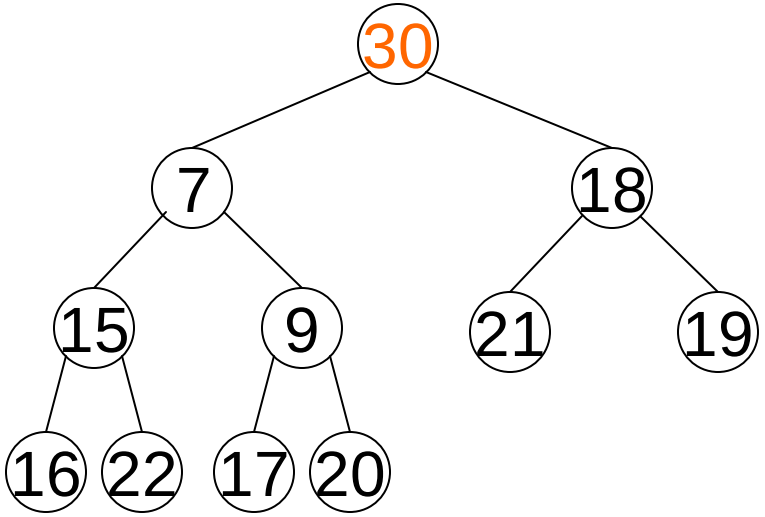
Migração de valores num Heap

```
proc Pai(i) { retornar i div 2 }
proc Esq(i) { retornar i * 2 }
proc Dir(i) { retornar i * 2 + 1 }
proc Subir (i, n, H [1 .. n]) {
    se i > 1 e H [Pai(i)] > H [i] então {
        H [i], H [Pai(i)] ← H [Pai(i)], H [i]
        Subir (Pai(i), n, H)
    }
}
proc Descer (i, n, H [1 .. n]) {
    se Dir(i) ≤ n e H [Dir(i)] < H [Esq(i)]
        então filho ← Dir(i)
        senão filho ← Esq(i)
    se filho ≤ n  e H [filho] < H [i] então {
        H [i], H [filho] ← H [filho], H [i]
        Descer (filho, n, H)
    }
}
```

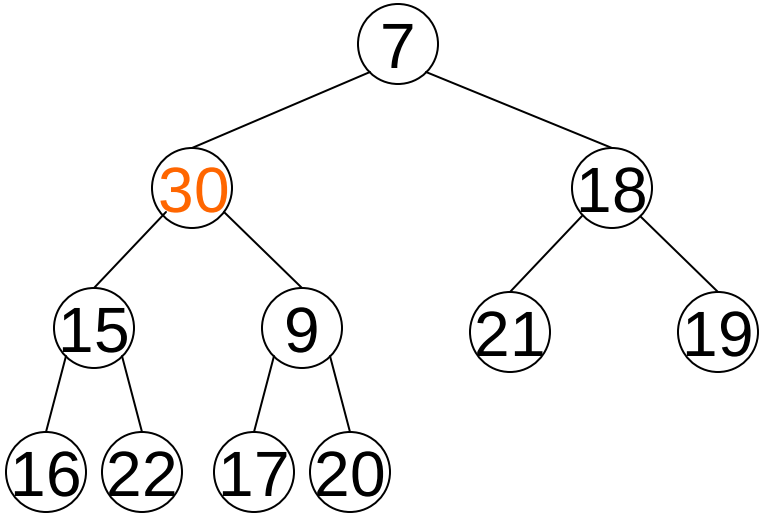
Migração de valores num Heap



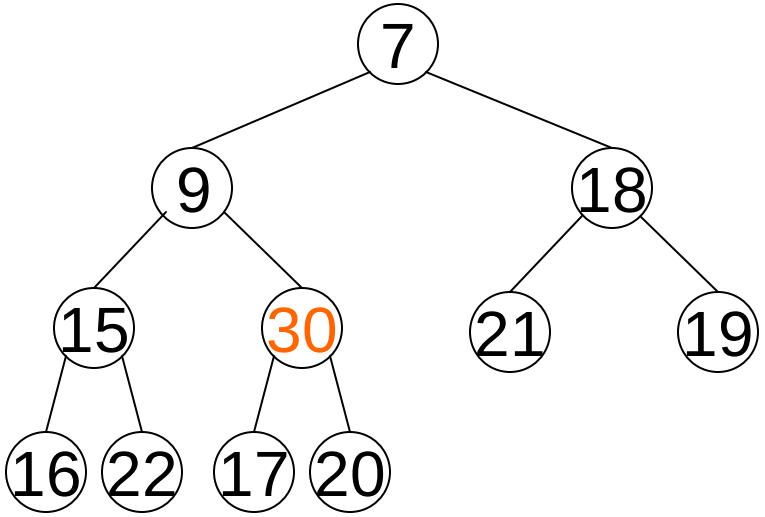
Migração de valores num Heap



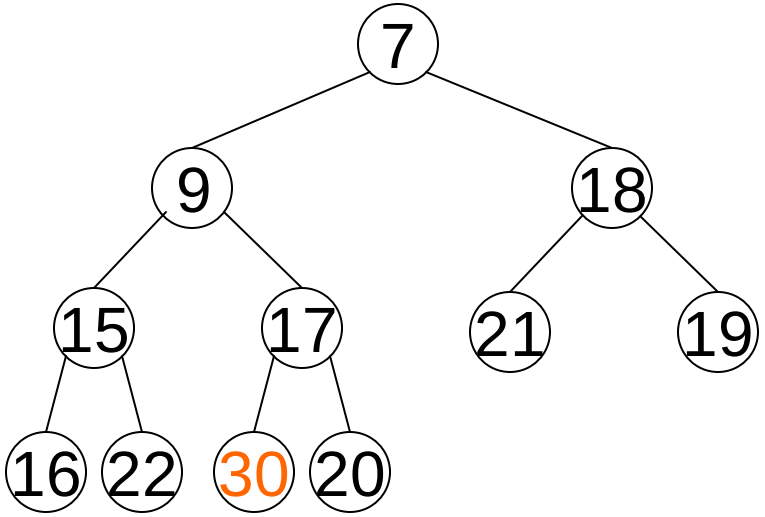
Migração de valores num Heap



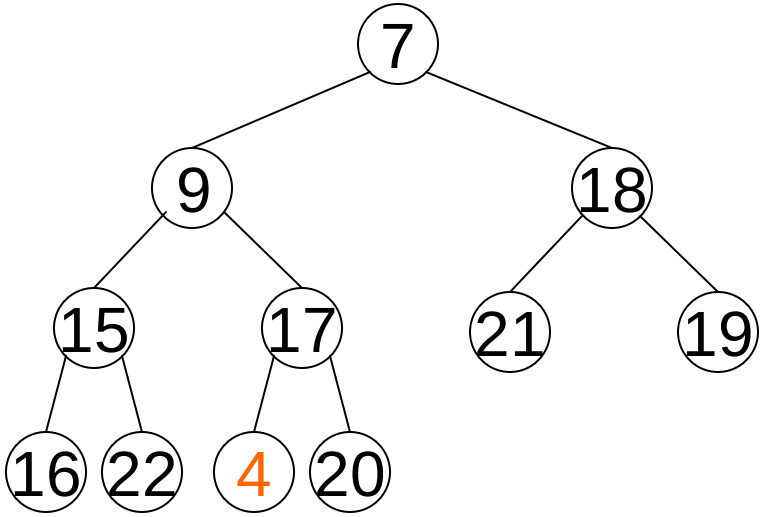
Migração de valores num Heap



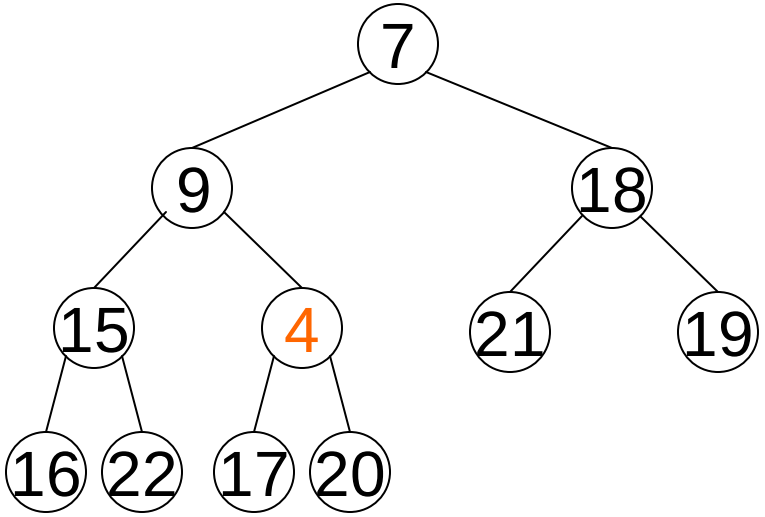
Migração de valores num Heap



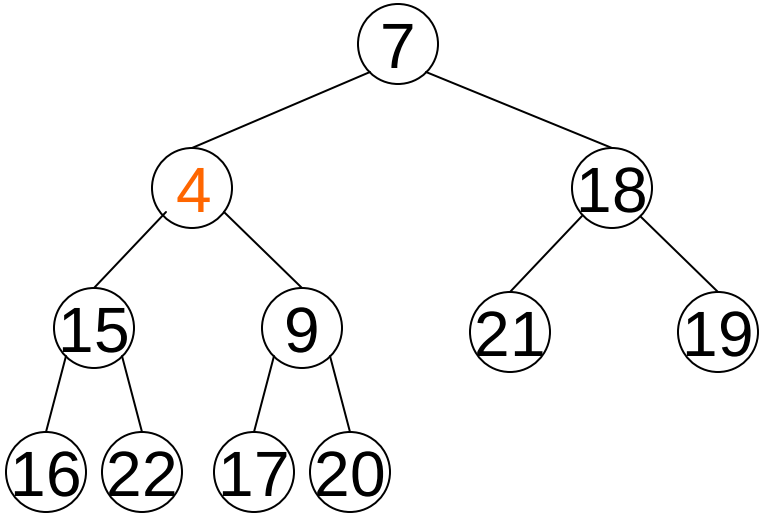
Migração de valores num Heap



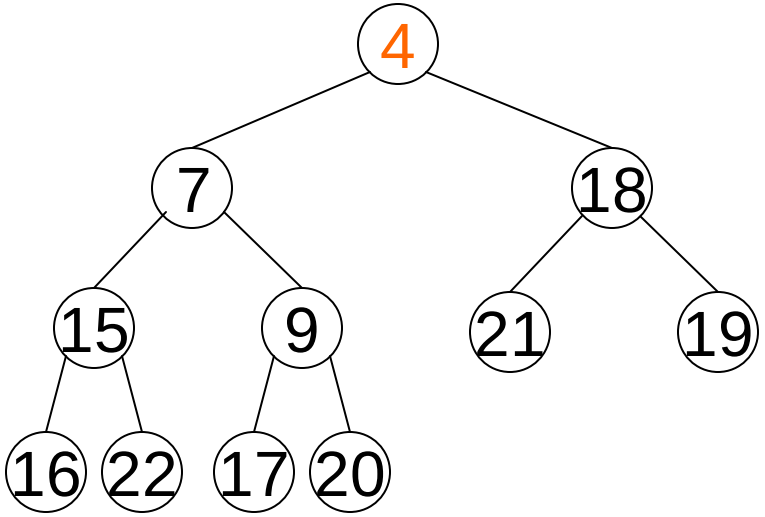
Migração de valores num Heap



Migração de valores num Heap



Migração de valores num Heap



Inserção e Remoção num Heap

- Claramente, *Subir* e *Descer* têm complexidade $O(\log n)$ já que percorrem no máximo um caminho igual à altura da árvore que é completa
- Para inserir, basta colocar o novo valor na posição $n + 1$ do heap e chamar *Subir*
- Para remover o menor valor, basta substituir a raiz ($H[1]$) por H (n) e chamar *Descer*

```
proc Inserir (x, n, H [1 .. n + 1]) {
    n ← n + 1
    H [n] ← x
    Subir (n, n, H)
}
proc RemoverMinimo (n, H [1 .. n]) {
    H [1] ← H [n]
    n ← n – 1
    Descer (1, n, H)
}
```

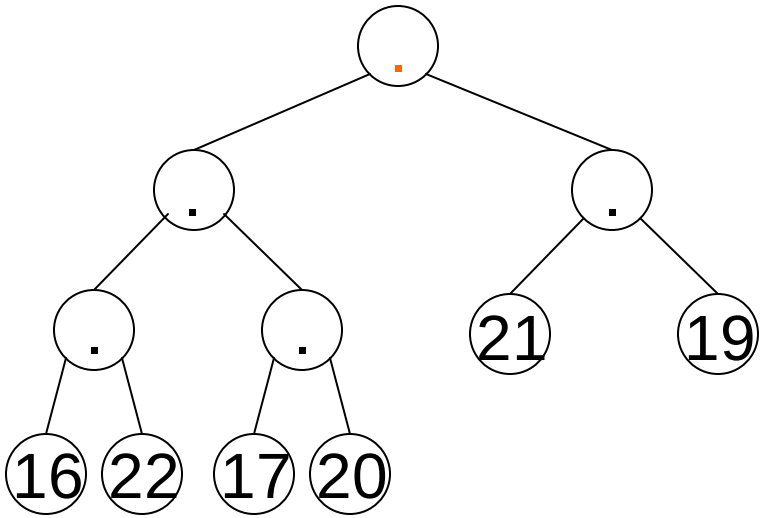
Construção de Heaps

- Se queremos construir um Heap com *n* elementos, podemos recorrer a um algoritmo ingênuo, isto é, inserir os *n* elementos um a um num heap inicialmente vazio.
- Mais simplesmente, podemos colocar os *n* elementos no array *H* e "subí-los" um a um

  para *i* **desde** 2 **até** *n* **fazer** *Subir* (*i*, *i*, *H*)
- Este algoritmo ingênuo tem complexidade $O(n \log n)$
- Entretanto, pode-se ordenar o heap em $O(n)$ se observarmos:
  - As folhas da árvore (elementos $H[n \text{ div } 2 + 1 .. n]$) não têm descendentes e portanto já estão ordenadas em relação a eles
  - Se acertarmos todos os nós internos (elementos $H[1 .. n \text{ div } 2]$) em relação a seus descendentes (rotina *Descer*), o heap estará pronto
  - É preciso trabalhar de trás para frente desde $n \text{ div } 2$ até 1 pois as propriedades da heap são observadas apenas nos níveis mais baixos

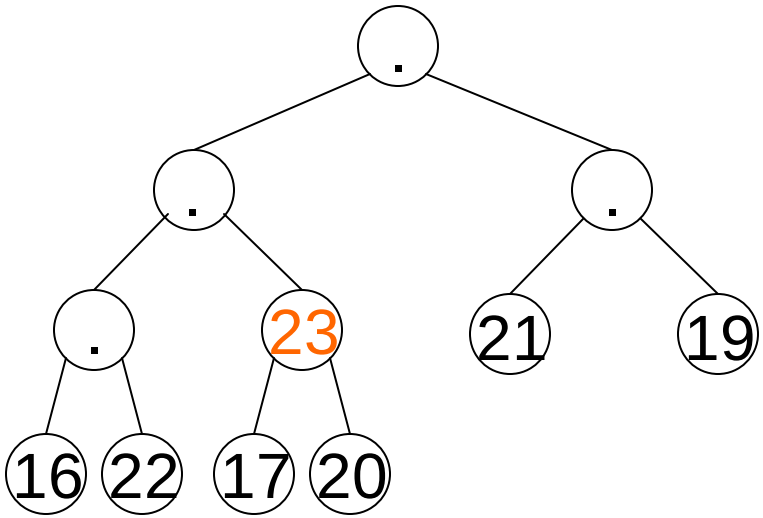
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



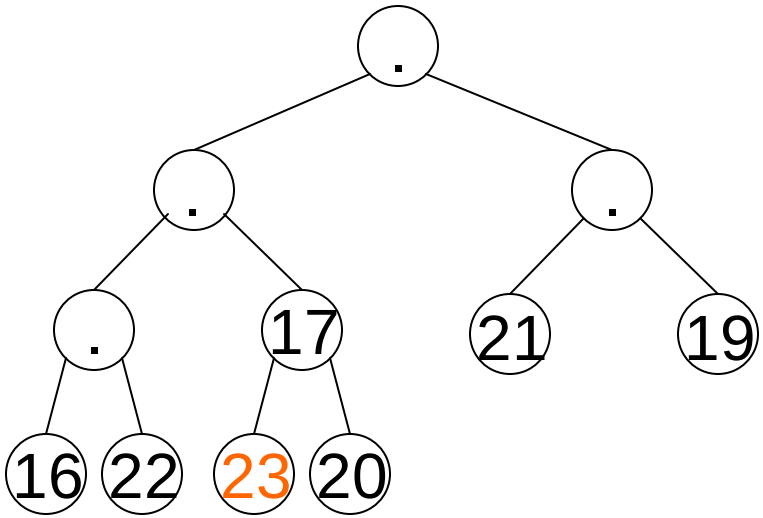
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



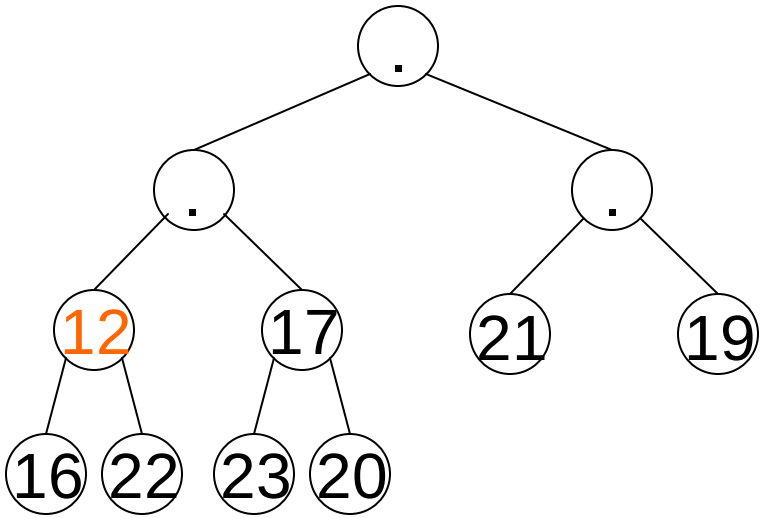
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



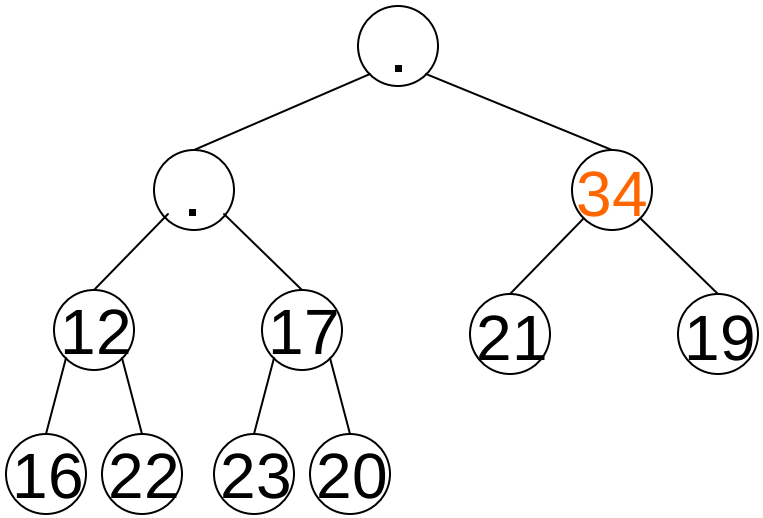
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



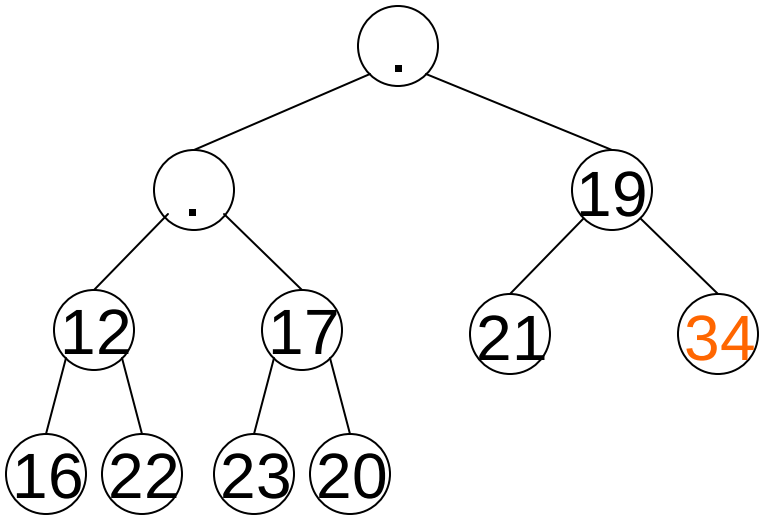
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



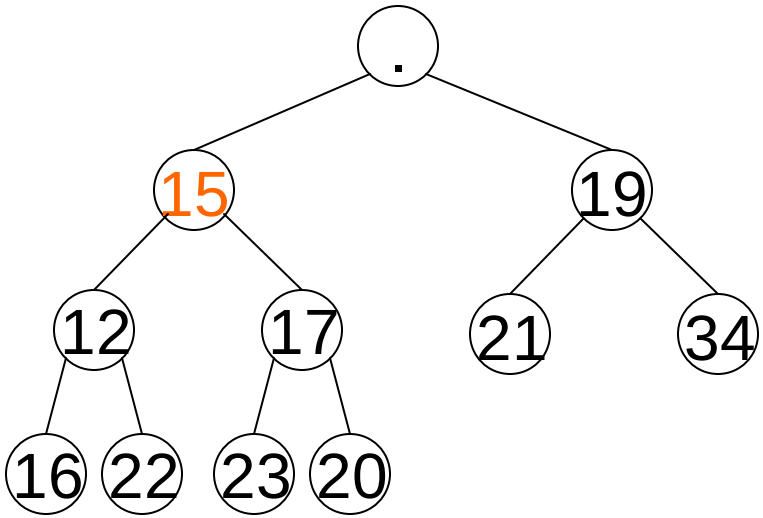
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



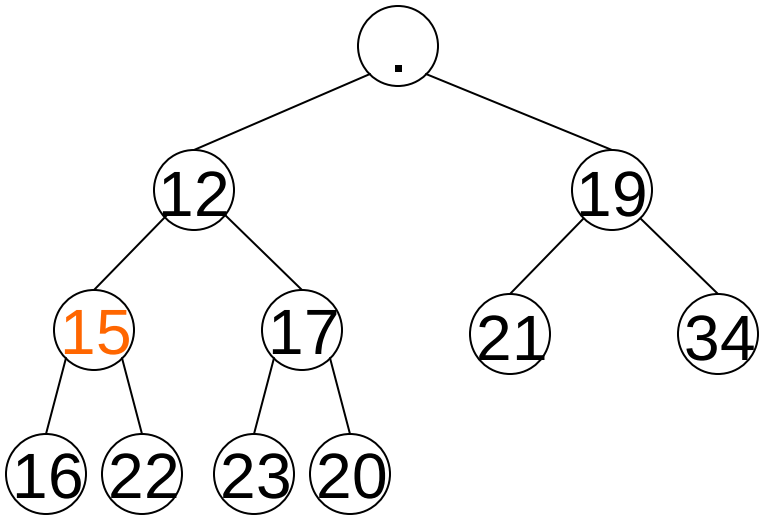
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



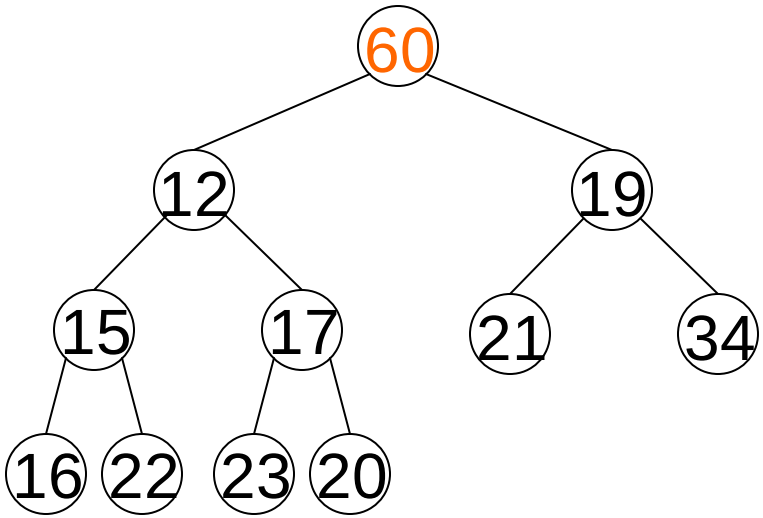
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



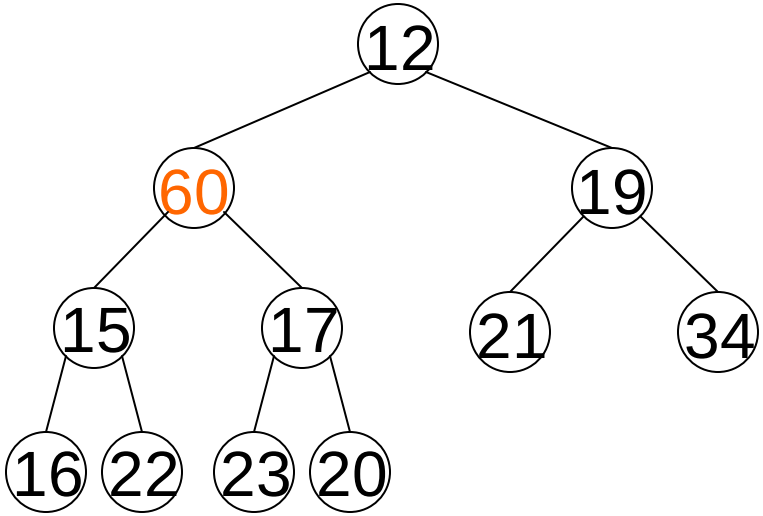
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



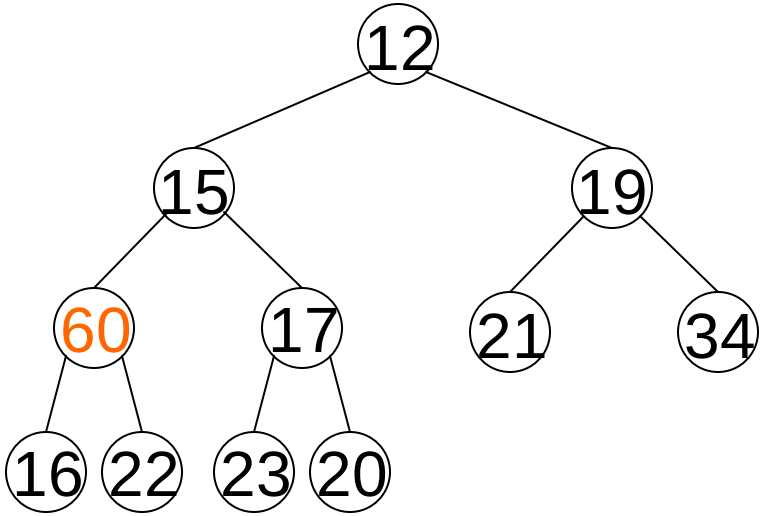
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



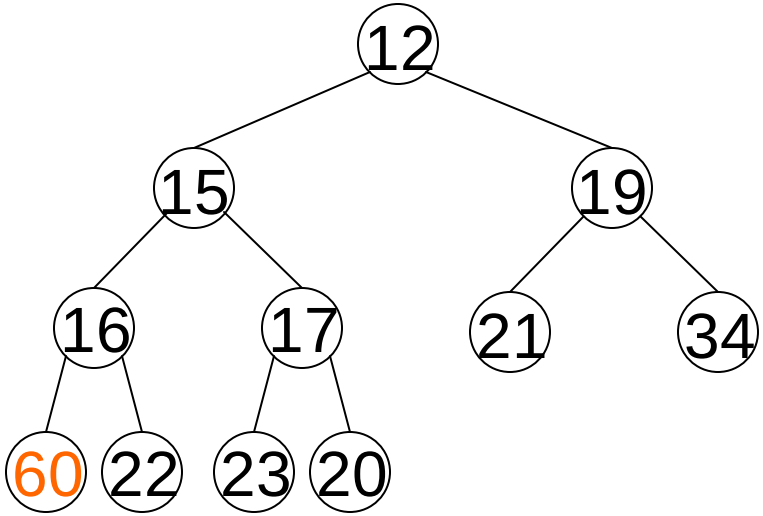
Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



Construção de Heap

para *i* **desde** *n* div 2 **decrementando até** 1 **fazer** *Descer* (*i*, *n*, *H*)



Complexidade do algoritmo de construção de Heap

- Suponhamos que a árvore seja cheia. Então,
  - $n = 2^h - 1$, onde h é a altura
  - desses, apenas $2^{h-1} - 1$ são nós internos
  - A raiz da árvore pode descer no máximo $h - 1$ níveis
  - Os dois nós de nível 2 podem descer $h - 2$ níveis
  - ...
  - Os 2^{h-2} nós de nível $h - 1$ podem descer 1 nível
  - Logo, no total temos

$$S = 1(h-1) + 2(h-2) + 2^2(h-3) + \ldots + 2^{h-2}(1)$$

Complexidade do algoritmo de construção de Heap

- Para resolver esse somatório, dobramos *S* e a seguir subtraímos *S* agrupando os termos com mesma potência de 2

$$S = 1(h-1) + 2(h-2) + 2^2(h-3) + \ldots + 2^{h-2}(1)$$

$$2S = 0 + 2(h-1) + 2^2(h-2) + \ldots + 2^{h-2}(2) + 2^{h-1}(1)$$

$$2S - S = (-h+1) + 2(h-1-h+2) + \ldots + 2^{h-2}(2-1) + 2^{h-1}(1)$$

$$= 1 - h + 2 + 2^2 + \ldots + 2^{h-2} + 2^{h-1}$$

$$= -h + \sum_{i=0}^{h-1} 2^i = -h + 2^h - 1 = -h + n = O(n)$$

HeapSort

- Uma vez que dispomos dos algoritmos para operar sobre um heap é possível usá-los para ordenar um array:
  - Construir o heap usando o método explicado anteriormente
  - Repetidamente remover o menor elemento e movê-lo para o fim do array, acertando o heap:

    $m \leftarrow n$
    enquanto $m > 1$ **fazer** {
    $H[1], H[m] \leftarrow H[m], H[1]$
    Descer $(1, m, H)$
    }
  - Ao final, o array está ordenado decrescentemente
  - Para obter ordem crescente, ou inverte-se a ordem do array ($O(n)$) ou utiliza-se um heap onde a raiz é o maior de todos os elementos

Tabelas de Dispersão (Hash Tables)

- São tabelas (arrays) cujos índices são de alguma forma relacionadas com os conteúdos das posições respectivas
  - O relacionamento é estabelecido por uma função $h:N \rightarrow M$, onde o domínio N é o espaço de chaves e M é o espaço de índices
  - Exemplo:
    - N é o conjunto de todas as cadeias alfabéticas
    - M é um inteiro entre 65 e 91
    - h é o código ASCII do primeiro caractere da cadeia
      - h(‘AMORA’) = 65
      - h(‘ZEBRA’) = 91
- A idéia é obter um método para implementação de dicionários onde o acesso a uma dada chave pode ser feito em $O(1)$ na média.
  - No pior caso, entretanto, o acesso pode ter custo $O(n)$

Funções de Dispersão

- Idealmente, funções de dispersão deveriam ser injetivas, isto é, todas as chaves deveriam ser mapeadas por h em um índice distinto
  - Isto só é possível se $|N| \leq |M|$
  - Mesmo assim, pode não ser trivial construir a função de dispersão
    - Por exemplo, considere o conjunto S de nomes dos alunos deste curso. Então
      - » $|S| < 100$ mas é impossível escrever uma função injetiva para esse domínio que não leve $O(|S|)$ para ser avaliada
      - » Precisamos considerar como o domínio de h o conjunto de todas as cadeias alfabéticas (de até 40 caracteres, digamos)
  - Um caso trivial é $h(x) = x$. Em programação comercial isto é conhecido como *acesso direto*
  - Em geral, $|N| >> |M|$
- Quando a função de dispersão não é injetiva, pode-se ter duas chaves x e y, $x \neq y$ tais que $h(x) = h(y)$. Se se tentar inserir ambas as chaves na tabela tem-se o que é conhecido como *colisão*

Funções de Dispersão

- Em geral, uma boa função de dispersão deve reunir as seguintes qualidades
  - produzir poucas colisões
    - Depende de se conhecer algo sobre a distribuição das chaves sendo acessadas
    - Ex.: se as chaves são códigos alfanuméricos que começam sempre por ‘A’ ou ‘B’, usar o primeiro caractere das chaves pode levar a muitas colisões
  - ser fácil de computar
    - Típicamente, funções contendo poucas operações aritméticas
  - ser uniforme
    - Idealmente, o número máximo de chaves que são mapeadas num mesmo índice deve ser $|N|/|M|$

Funções de Dispersão – Método da Divisão

- Assumindo $N = \{0 .. n - 1\}$ e $M = \{0 .. m - 1\}$, a função de dispersão é dada por

$$h(x) = x \bmod m$$

- Qual deve ser o valor de m?
  - não deve ser uma potência de 2
    - se $m = 2^k$, $h(x) = k$ bits menos significativos de x
  - não deve ser um número par,
    - se m é par então $h(x)$ é par $\Leftrightarrow$ x é par
  - na prática, bons resultados são obtidos com:
    - m = número primo não próximo a uma potência de 2
    - m = número sem divisores primos menores que 20
- Não é bom que chaves sucessivas sejam mapeadas em índices sucessivos. Por isso, comumente se multiplica a chave por uma constante k antes de se fazer a divisão (m e k devem ser primos entre si):

$$h(x) = x\,k \bmod m$$

Funções de Dispersão – Método da Multiplicação

- Assume-se $m = 2^k$.
- A idéia é multiplicar a chave por ela mesma ou por alguma constante c. Se o resultado cabe numa palavra com b bits, toma-se os k bits do meio da palavra descartando os $(b - k)/2$ bits mais e menos significativos

$$h(x) = (x^2 \text{ div } 2^{(b-k)/2}) \bmod 2^k$$

ou

$$h(x) = (x\,c \text{ div } 2^{(b-k)/2}) \bmod 2^k$$

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

5	8	7	3	2	1

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

7	3	2	1
8	5		

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

	7	3	2	1
+			8	5
=			3	8

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

3	8	2	1

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

2	1

8	3

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

	2	1	
+	8	3	
=	0	4	

Funções de Dispersão – Método da Dobra

- Suponha que a chave seja dada por uma seqüência de dígitos escritos numa folha de papel. O método consiste em dobrar sucessivamente a folha de papel após o j-ésimo dígito somando os dígitos que se superpoem (sem fazer o "vai um")

0	4

Funções de Dispersão – Método da Dobra

- Uma outra variação consiste em fazer a dobra de *k* em *k* bits, ou seja, considerando os "dígitos" 0 e 1 da representação binária do número. O resultado é um índice entre 0 e $2^k - 1$
- Nesse caso, ao invés de somar os bits, utiliza-se uma operação de ou-exclusivo entre os bits
  - Não se usa "e" ("ou") pois estes produzem resultados menores (maiores) que os operandos
- Exemplo: Suponha $k = 5$

  $71 = 0001000111_2$

  $h(71) = 00010_2$ *xor* $00111_2 = 00101_2 = 5$

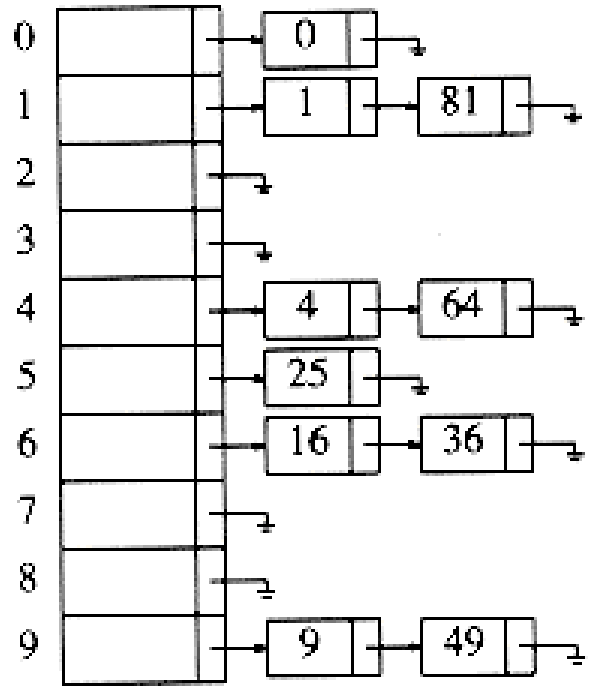
Funções de Dispersão – Método da Análise dos Dígitos

- Usado em casos especialíssimos
- É preciso conhecer todos os valores de antemão
- Ver Livro do Jayme
- Se alguma aplicação precisar de uma função de hash ajustada para uma coleção particular de chaves, deve usar um dos métodos para computar funções de hash perfeitas
- e.g.: *gperf* [Schmidt 90]

Tratamento de Colisões – Encadeamento Exterior

- Mesmo com boas funções de dispersão, à medida que o fator de carga α (número de chaves armazenadas / número de índices) aumenta, a probabilidade de haver colisões aumenta
- De maneira geral qualquer tabela de espalhamento precisa prever algum esquema para tratamento de colisões
- Uma das maneiras mais empregadas para lidar com colisões é permitir que cada posição da tabela seja ocupada por mais de uma chave
  - Em vez de guardar uma chave, guarda-se uma lista de chaves
  - Na verdade, pode-se usar qualquer estrutura – uma árvore, por exemplo – mas como a ocorrência de colisões deve ser relativamente rara, uma lista ordenada ou não costuma ser suficiente

Tratamento de Colisões – Encadeamento Exterior



Tratamento de Colisões – Encadeamento Exterior

- Quantas comparações podemos esperar em média para um acesso a chaves não presentes (buscas sem sucesso)?
  - Supomos que *h* é uma função uniforme, que o fator de carga da tabela é α e que as listas são não ordenadas
  - Então a probabilidade de *h* computar cada índice *i* é uniforme e igual a 1/*m*
  - O número de comparações feitas ao se acessar a entrada *i* da tabela é o comprimento da lista L_i
  - Então,

$$\text{Custo Médio} = \frac{1}{m}\sum_{i=0}^{m-1}\left|L_i\right| = \frac{n}{m} = a = \text{Fator de Carga}$$

Tratamento de Colisões – Encadeamento Exterior

- Quantas comparações podemos esperar em média para um acesso a chaves presentes (buscas bem-sucedidas)?
  - Para achar uma chave *x*, pesquisa-se uma lista L_i
  - Além da comparação bem sucedida com a chave armazenada em L_i, o número de comparações mal-sucedidas é o comprimento da lista L_i *no momento em que x foi originalmente inserido na tabela*
  - Se *x* foi a (*j*+1)-ésima chave a ser incluída, então o comprimento médio de L_i é j/m
  - Então o custo médio é dado por

$$\mathrm{CM} = \frac{1}{n}\sum_{j=0}^{n-1}\left(1+\frac{j}{m}\right) = \frac{1}{n}\left(n+\frac{1}{m}\sum_{j=0}^{n-1} j\right) = 1+\frac{n(n-1)}{2nm} = 1+\frac{a}{2}-\frac{1}{2m}$$

Tratamento de Colisões – Encadeamento Exterior

- Portanto, se mantemos o fator de carga baixo (menor que uma constante α), temos que a complexidade média da busca é O(1)
- A única desvantagem do encadeamento exterior é que ele requer o uso de estruturas externas e com isso o uso de alocação dinâmica de memória e o “overhead” correspondente
- Para contornar isso pode-se usar encadeamento interior ou endereçamento externo

Tratamento de Colisões – Encadeamento Interior

- A idéia é usar como nós das listas as próprias entradas da tabela
- Há duas variantes
- Na primeira, a tabela de *m* entradas é dividida em duas porções:
  - A função de dispersão *h* retorna apenas índices na primeira porção – de 0 a *p* –1, por exemplo
  - A segunda porção – índices de *p* a *m*-1é usada como área comum para overflow
  - Pode acontecer que a área de overflow seja toda tomada sem que todas as entradas da tabela tenham sido usadas
  - Pode-se aumentar a área de overflow diminuindo-se *p*, mas isso também é ineficiente. No limite, *p* = 1 e a tabela resume-se a uma lista encadeada

Tratamento de Colisões – Encadeamento Interior

- Na segunda variante, todo o espaço de endereçamento é usado
- O maior problema dessa abordagem é que pode haver colisões secundárias, isto é colisões entre chaves não sinônimas ($h(x) \neq h(y)$)
- Quando ocorre uma colisão, a chave é armazenada na primeira posição livre após $h(x)$, a posição d, digamos
- Se agora incluirmos y tal que $h(y)=d$, teremos a fusão das listas correspondentes a $h(x)$ e $h(y)$, diminuindo a eficiência do esquema

Tratamento de Colisões – Encadeamento Interior

- Um outro problema refere-se às dificuldades introduzidas no processo de exclusão
  - Não se pode simplesmente retirar o elemento da cadeia
  - Além do valor de chave especial que indica “posição vazia”, é preciso criar um valor de chave especial que indica “elemento removido”
  - Uma inserção posterior pode reaproveitar posições marcadas com “elemento removido”
- Na verdade, encadeamento interior com espaço de endereçamento único não é uma boa idéia, já que os problemas são os mesmos encontrados no tratamento de colisões por endereçamento aberto, sendo que nesse último temos a vantagem de não precisar de ponteiros

Tratamento de Colisões – Endereçamento Aberto

- Ao invés de usar ponteiros, utiliza-se uma outra função de dispersão que indica o próximo índice a ser tentado. Em geral temos a função de dispersão $h(x, k)$ onde
  - x *é* a chave
  - $k = 0, 1, 2$, etc. é o número da tentativa
- $h(x, k)$ tem que ser desenhada de tal forma a visitar todos os m endereços em m tentativas
- No pior caso, m tentativas são feitas

Tratamento de Colisões – Endereçamento Aberto

- Tentativa linear:
  - $h\,(x, k) = (h'\,(x) + k) \bmod m$
  - Tem a desvantagem de agrupar tentativas consecutivas
- Tentativa quadrática
  - $h\,(x, k) = (h'\,(x) + c_1\,k + c_2\,k^2) \bmod m$
  - Resolve o problema do agrupamento primário
  - Problema do agrupamento secundário
    - chaves x e y tais que $h'\,(x) = h'\,(y)$ geram a mesma seqüência de tentativas
- Dispersão dupla
  - $h\,(x, k) = (h'\,(x) + k\,h''\,(x)) \bmod m$