

Análise de Algoritmos e Complexidade da Computação

Estruturas de Dados

Prof. Dr. Osvaldo Luiz de Oliveira

Estas anotações devem ser
complementadas por
apontamentos em aula.

Vetor (*array*)

- É uma estrutura de dados que armazena elementos do mesmo tipo (inteiro, ou real, ou um registro etc.).
- Permite acesso indexado a qualquer de seus elementos.
- Tem sempre um tamanho fixo.

Complexidades

- Espaço: tendo um tamanho fixo, um vetor gasta sempre uma quantidade constante de memória durante a execução do algoritmo. Portanto, a complexidade de espaço devida a um vetor é $O(1)$.
- Tempo para inserir, buscar ou remover um elemento: qualquer elemento de um vetor pode ser pesquisado, inserido ou removido em um tempo constante ($O(1)$).

Limitações

- Um vetor não pode ser usado para armazenar elementos de diferentes tipos.
- O tamanho de um vetor não pode aumentar dinamicamente, i.e., à medida que o algoritmo é executado.

Registro

- É uma E.D. similar a um vetor, exceto pelo fato de que ela pode armazenar elementos de diferentes tipos.

```
registro exemplo {  
  n: inteiro;  
  x: real;  
  v: vetor [1..20] de inteiro;  
  matriz: vetor [1..20, 1..30] de booleano;  
  registro outroRegistro {  
    h: cadeia_de_caracteres;  
    c: caracter;  
    y: vetor [ 1..12] de inteiro;  
  }  
}
```

Acesso

`exemplo.n := 5;`

`exemplo.x := 3.2;`

`y := exemplo.x * 5 + exemplo.n;`

`exemplo.v [2] := 15;`

`Exemplo.matriz [2, 4] := verdadeiro;`

`exemplo.outroRegistro.h := 'a';`

Complexidade

- Espaço: tendo um tamanho fixo, um registro gasta sempre uma quantidade constante de memória durante a execução de um algoritmo. Portanto, a complexidade de espaço é $O(1)$.
- Tempo para buscar, inserir e remover: qualquer campo de um registro pode ser pesquisado, inserido ou removido em um tempo constante ($O(1)$).

Limitação

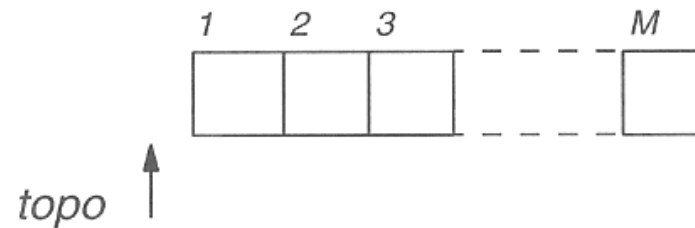
- O tamanho de um registro não pode aumentar dinamicamente.

Pilha

Disciplina de entrada e saída de elementos:
LIFO (*last input, first output*).

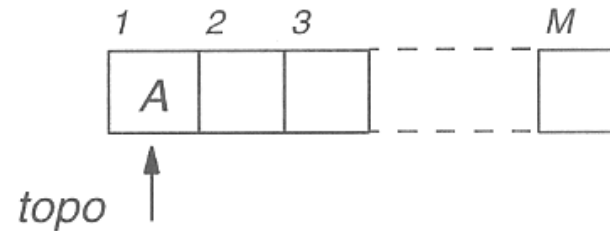
situação 1:

inicial : pilha vazia



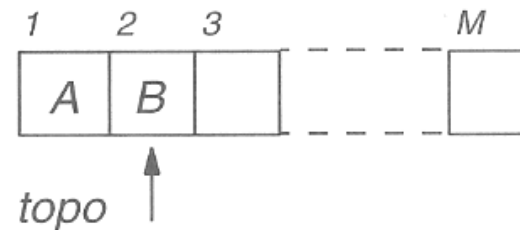
situação 2:

inserir informação A



situação 3:

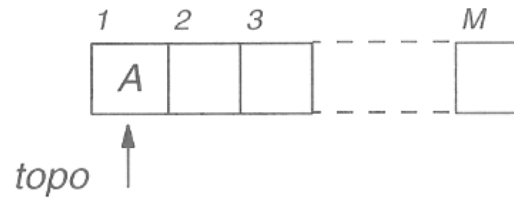
inserir informação B



Pilha

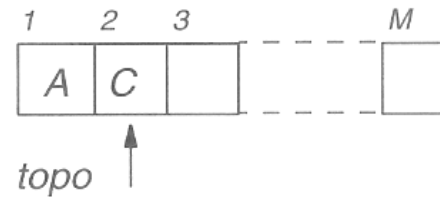
situação 4:

retirar informação (B)



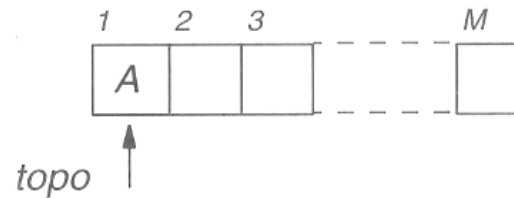
situação 5:

inserir informação C



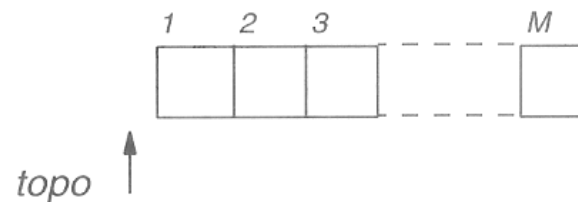
situação 6:

retirar informação (C)



situação 7:

retirar informação (A)



Algoritmos

Algoritmo Empilhar (P, x)

```
{
  se ( topo  $\neq$  M )
  {
    topo := topo + 1;
    P [topo] := x
  }
  senão Escrever ('Escrever: pilha cheia.')
}
```

Algoritmo Desempilhar (P, x)

```
{
  se ( topo  $\neq$  0 )
  {
    x := P [topo]; topo := topo - 1;
    retornar x
  }
  senão Escrever ('Escrever: pilha vazia.')
}
```

Algoritmos

Algoritmo Vazia (P)

```
{  
    se ( topo = 0 ) retornar verdadeiro  
    senão retornar falso  
}
```

Algoritmo Cheia (P)

```
{  
    se ( topo = M ) retornar verdadeiro  
    senão retornar falso  
}
```

Algoritmo ElementoNoTopo (P)

```
{  
    se ( não Vazia (P) ) retornar P [topo]  
    senão Escrever (‘Erro: pilha vazia.’)  
}
```

Complexidades

Empilhar, Desempilhar, Vazia, Cheia, ElementoNoTopo

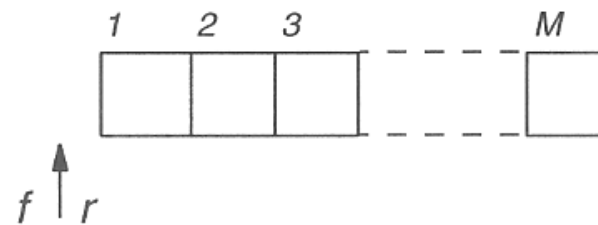
$$T(n) = O(1).$$

Fila

Disciplina de entrada e saída de elementos:
FIFO (*first input, first output*).

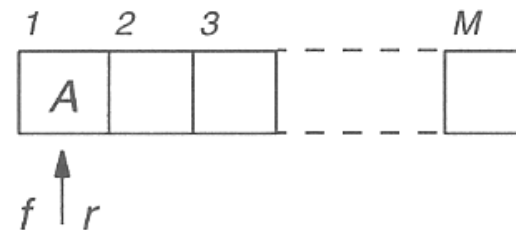
situação 1:

inicial : fila vazia



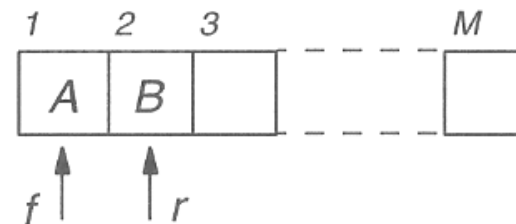
situação 2:

inserir informação A



situação 3:

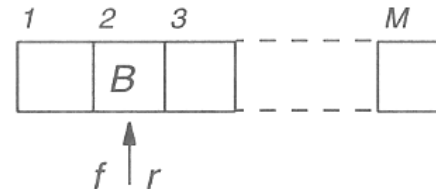
inserir informação B



Fila

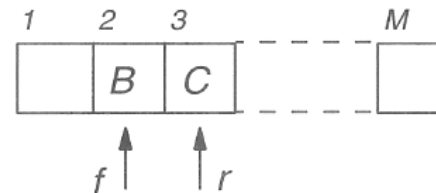
situação 4:

retirar informação (A)



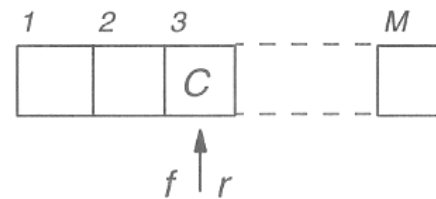
situação 5:

inserir informação C



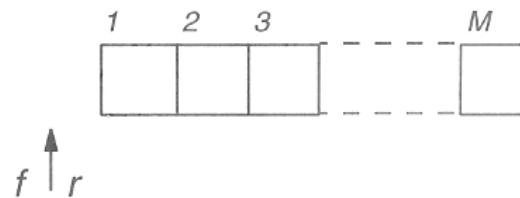
situação 6:

retirar informação (B)



situação 7:

retirar informação (C)



Algoritmos

Algoritmo Inserir (F, x)

```
{  
  prox := r mod M + 1;  
  se ( prox ≠ f )  
  {  
    r := prox; F [prox] := x;  
    se ( f = 0 ) f := 1  
  }  
  senão Escrever ('Fila cheia.')}
```

Algoritmo Retirar (F, x)

```
{  
  se ( f ≠ 0 )  
  {  
    x := F [ f ];  
    se ( f = r ) então f := r := 0  
    senão f := f mod M + 1;  
    retornar x  
  }  
  senão Escrever ('Fila vazia.')}
```


Algoritmos

Algoritmo Vazia (F)

```
{  
  se ( $f = 0$ ) retornar verdadeiro  
  senão retornar falso  
}
```

Algoritmo Cheia (F)

```
{  
  prox :=  $r \bmod M + 1$ ;  
  se ( $f = \text{prox}$ ) retornar verdadeiro  
  senão retornar falso  
}
```

Algoritmo ElementoNaFrente (F)

```
{  
  se ( não Vazia (F) ) retornar  $F[f]$   
  senão Escrever ('Erro: fila vazia.')
```

Complexidades

Inserir, Retirar, Vazia, Cheia, ElementoNaFrente:

$$T(n) = O(1).$$

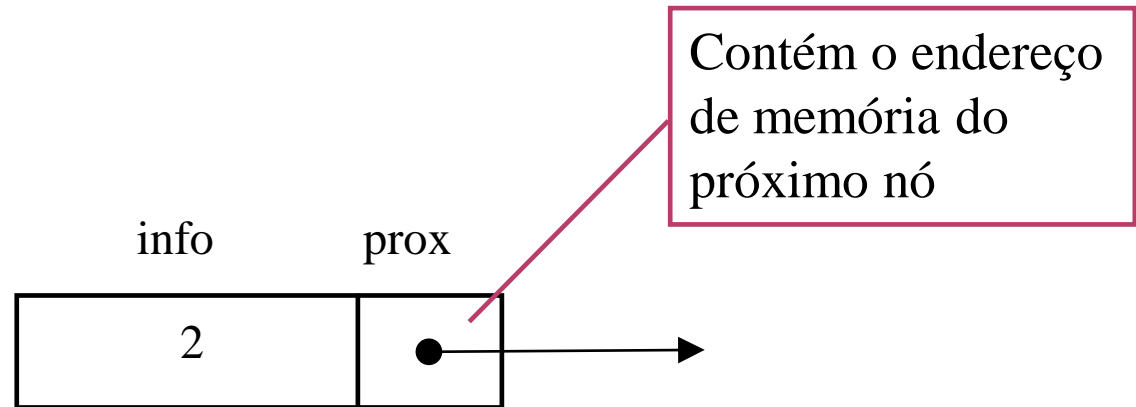
Listas ligadas (ou encadeadas)

- Existem muitas aplicações na qual o número de elementos envolvidos pode variar durante a execução do algoritmo. Pode-se definir um vetor ou um registro de um certo tamanho fixo como estrutura de dados para o algoritmo. No entanto, o tamanho fixo pode não ser suficiente ou pode ser demasiado. Nestes casos, estruturas de dados que podem modificar o seu tamanho dinamicamente podem ser desejáveis. A lista ligada é uma delas.

Listas ligadas simples

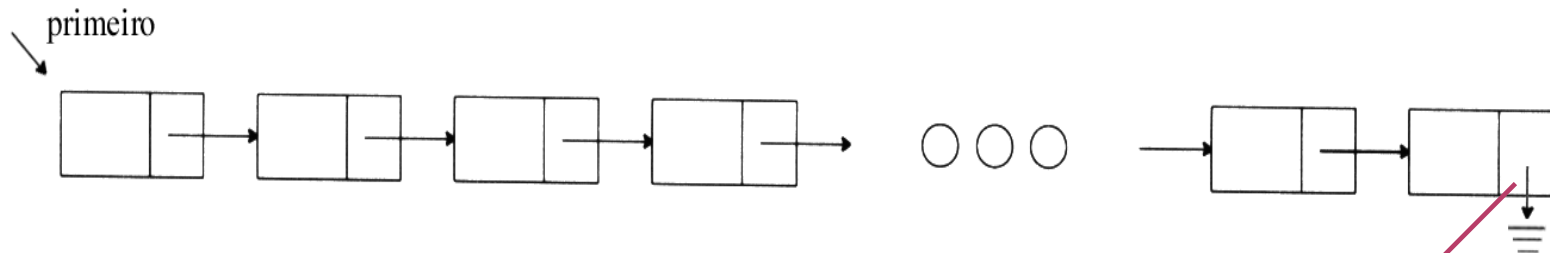
- Listas ligadas são estruturas de dados formadas por nós contendo no mínimo um campo, onde a informação é armazenada, e um campo ponteiro. Em uma lista ligada simples, o campo ponteiro contém o endereço do próximo nó na lista.

```
registro nó  
{  
  info: inteiro;  
  prox: ponteiro;  
}
```



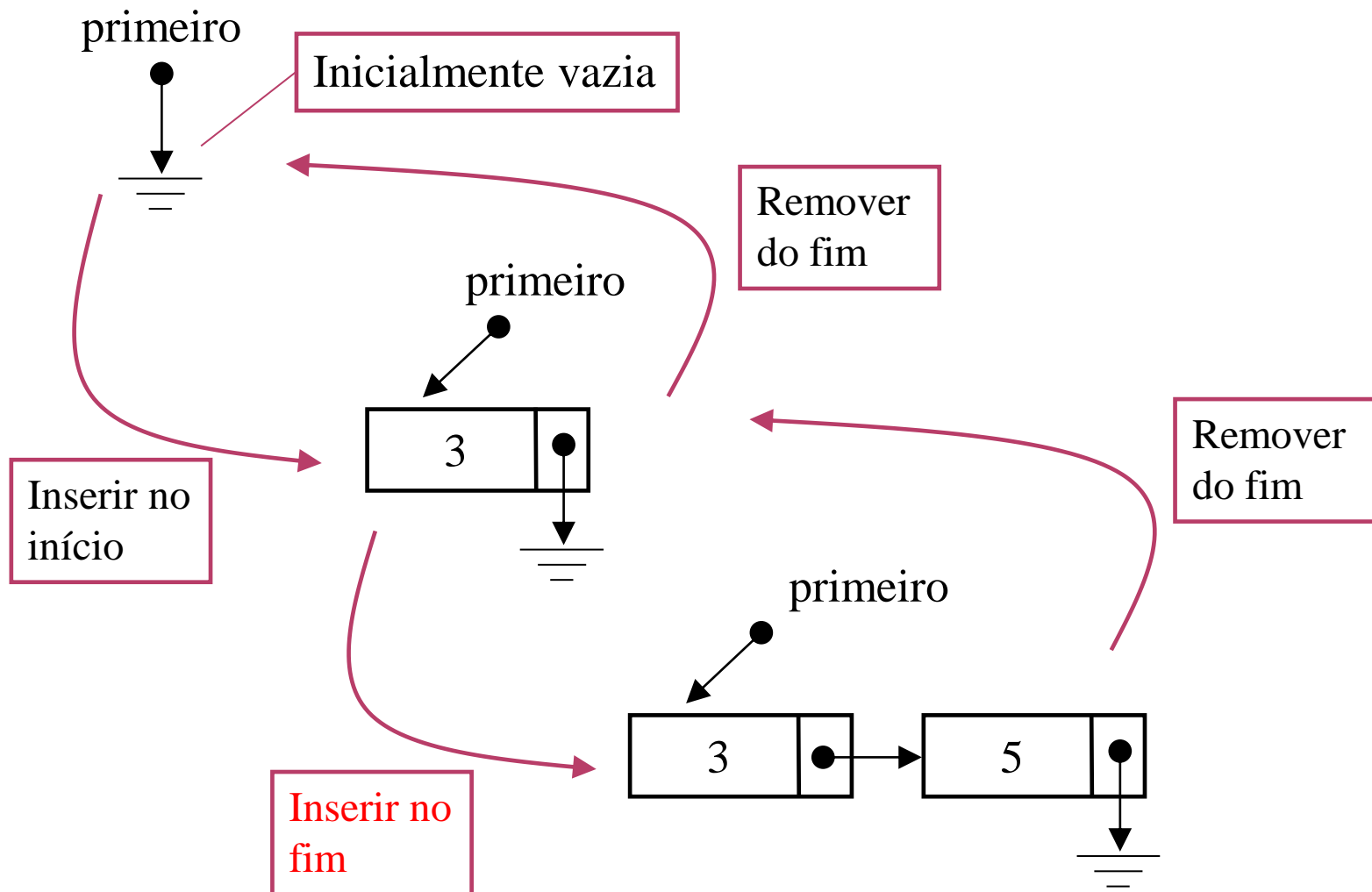
Uma lista ligada simples

- Um ponteiro especial contém o endereço do primeiro nó da lista.



Ponteiro para nil
(ou null): indica o
fim.

Um possível “ciclo de vida”



Busca

Algoritmo Busca (*primeiro*, x)

Entrada: *primeiro*, um ponteiro para uma lista ligada simples e x , um valor.

Saída: um ponteiro para o nó de valor igual a x , ou nil, caso tal nó não exista.

```
{  
    p := primeiro;  
    enquanto ( p ≠ nil e p.info ≠ x )  
        p := p.prox;  
  
    retornar p  
}
```

Complexidade

$$T(n) = O(n).$$

Inserir no início

Algoritmo InserirInício (*primeiro*, *x*)

Entrada: *primeiro*, um ponteiro para uma lista ligada simples e *x*, um elemento a inserir.

Saída: a lista com o elemento *x* inserido no início.

```
{  
① Novo (p); // Cria um novo nó e faz o ponteiro p apontar para ele.  
② p.info := x;  
③ p.prox := primeiro;  
④ primeiro := p  
}
```

Complexidade

$$T(n) = O(1).$$

Obs.: observe que o algoritmo funciona também no caso da lista estar inicialmente vazia.

Remover do fim

Algoritmo RemoveFim (*primeiro*)

Entrada: *primeiro*, um ponteiro para uma lista ligada simples.

Saída: se a lista não está vazia, remove o último elemento e retorna o seu valor.

```

{
  se (primeiro = nil) // A lista está vazia.
    Mensagem ('Não é possível remover')
  senão
    { // A lista tem, pelo menos, um elemento.
      ① p := primeiro; q := p.prox; // q “passeará” na frente de p.
      se (q = nil)
        { // A lista tem apenas um elemento.
          ② x := p.info;
          ③ Devolver (p); // Devolve a região de memória apontada por p.
          ④ primeiro := nil;
        }
    }
}

```

```
senão
```

```
{ // A lista tem, pelo menos, dois elementos.
```

```
    enquanto (q.prox  $\neq$  nil)
```

```
    {
```

```
        ⑤ p := q;
```

```
        ⑥ q := q.prox
```

```
    }
```

```
    // Ao final deste laço p e q apontam, respectivamente, para o penúltimo e último  
    // elemento da lista.
```

```
    ⑦ x := q.info;
```

```
    ⑧ p.prox := nil;
```

```
    ⑨ Devolver (q); // Devolve a região de memória apontada por q.
```

```
}
```

```
retornar x
```

```
}
```

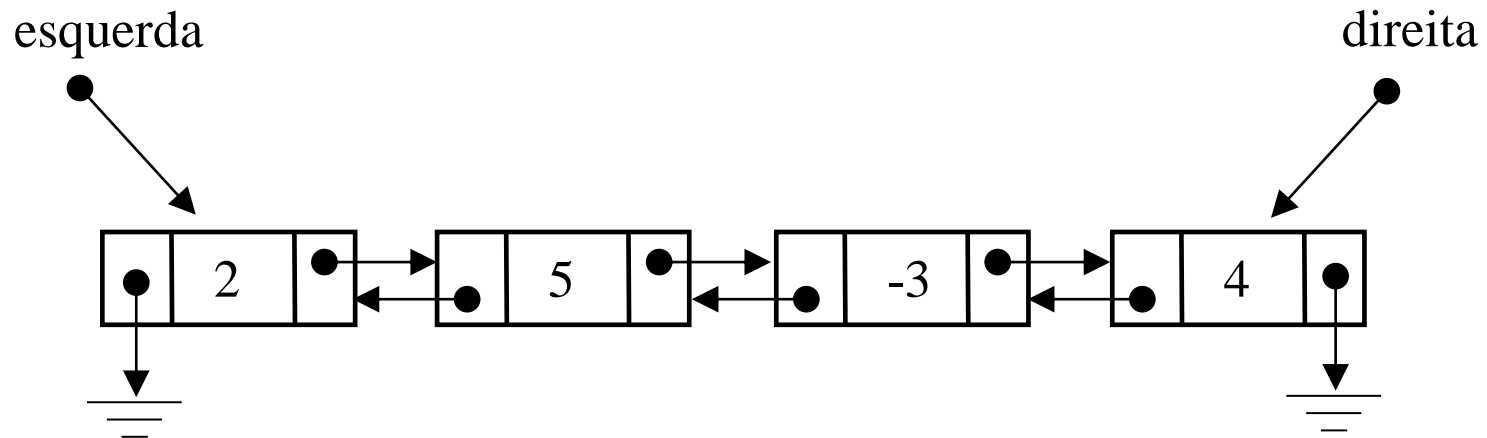
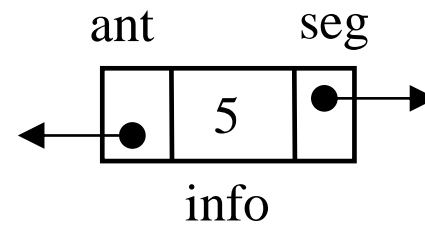
```
}
```

Complexidade

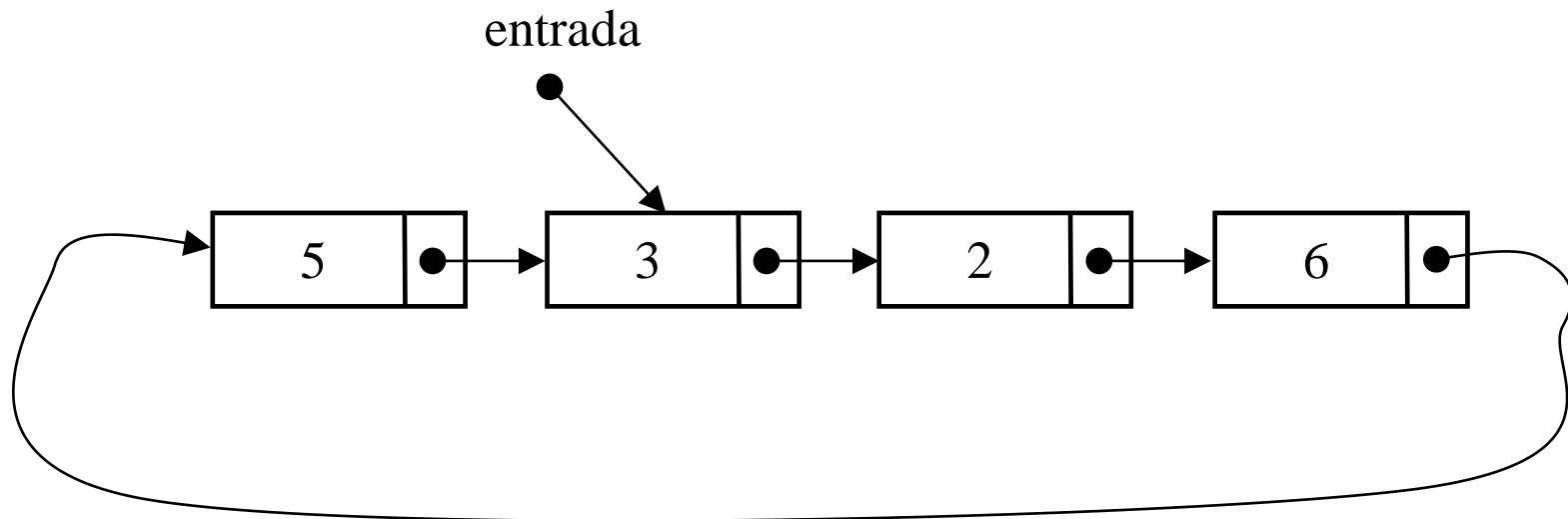
- A complexidade deste algoritmo é proporcional à quantidade de vezes em que o laço “**enquanto** (q.prox \neq nil) ...” será executado. Logo,

$$T(n) = O(n).$$

Lista duplamente ligada



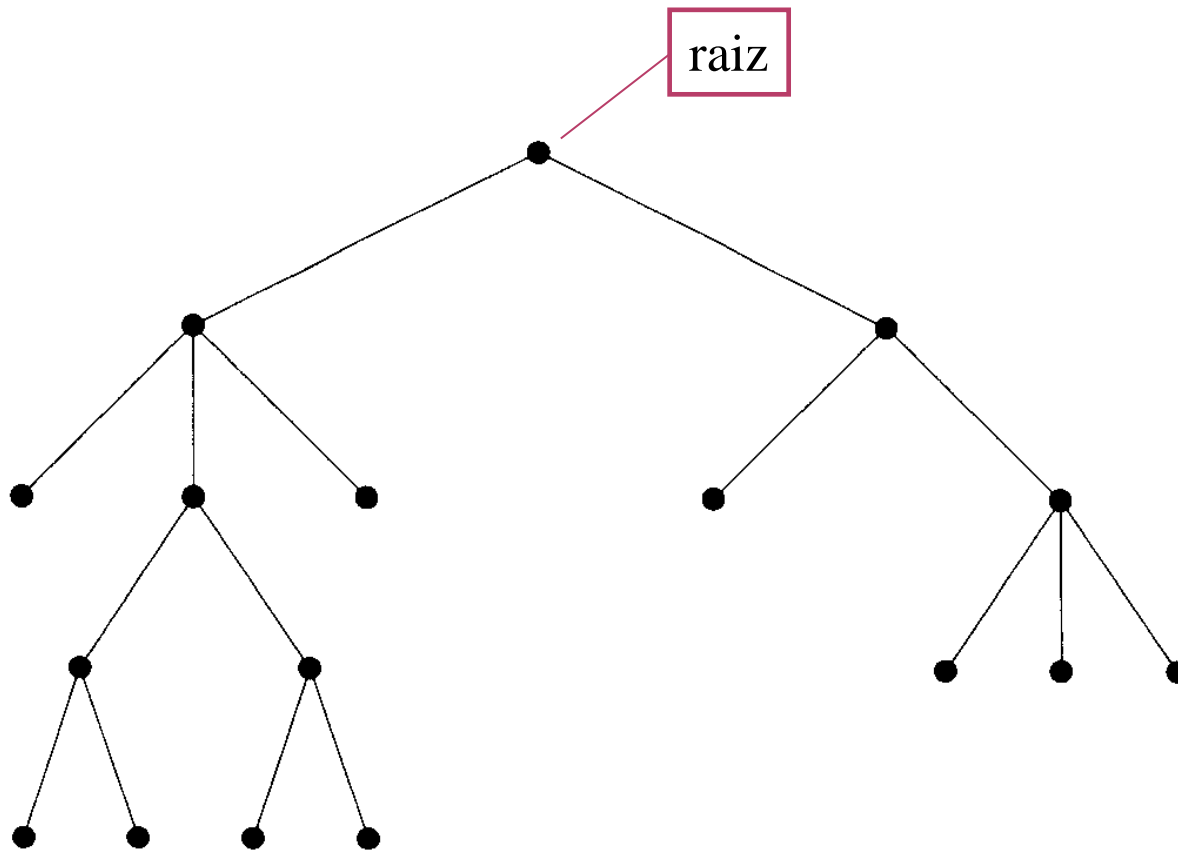
Lista circular simples



Árvores

Árvores são estruturas de dados que suportam bem muitas operações dinâmicas tais como: busca, inserção, remoção, cálculo do máximo, cálculo do mínimo, cálculo do predecessor, cálculo do sucessor etc..

Árvore enraizada



Definições

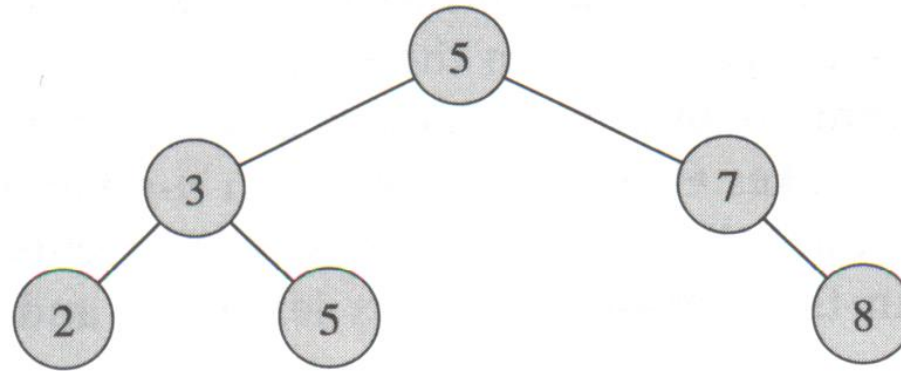
- O nó no topo da árvore é chamado de raiz. A raiz, **dizemos no nível 0**, está conectada a outros nós, os quais estão no nível 1 da hierarquia. Estes nós por sua vez estão conectados a nós no nível 2 e assim por diante. Todas as conexões são entre um nó e o seu único pai. A raiz não tem nenhum pai.
- A principal propriedade de uma árvore é que ela não tem ciclo. Como consequência, existe apenas uma única rota (caminho) ligando quaisquer dois nós na árvore.
- Um nó conectado ao seu superior é dito ser filho deste nó.

Definições

- O maior número de filhos entre todos os nós da árvore é denominado grau da árvore.
- Um nó sem nenhum filho é chamado de folha.
- A altura de uma árvore é o nível máximo da árvore. Em outras palavras, é a distância máxima entre a raiz e as folhas. Por definição a raiz está no nível 0.

Árvore binária

- Árvore cujo grau é 2, isto é, cada nó pode ter no máximo 2 filhos.



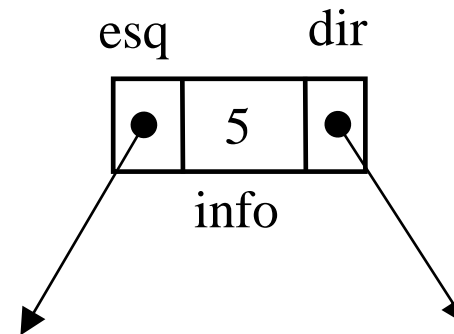
- Tendo um nó no máximo dois filhos eles serão chamados de filho esquerdo e filho direito conforme a sua posição em relação a um observador que olha para a hierarquia.

Representações

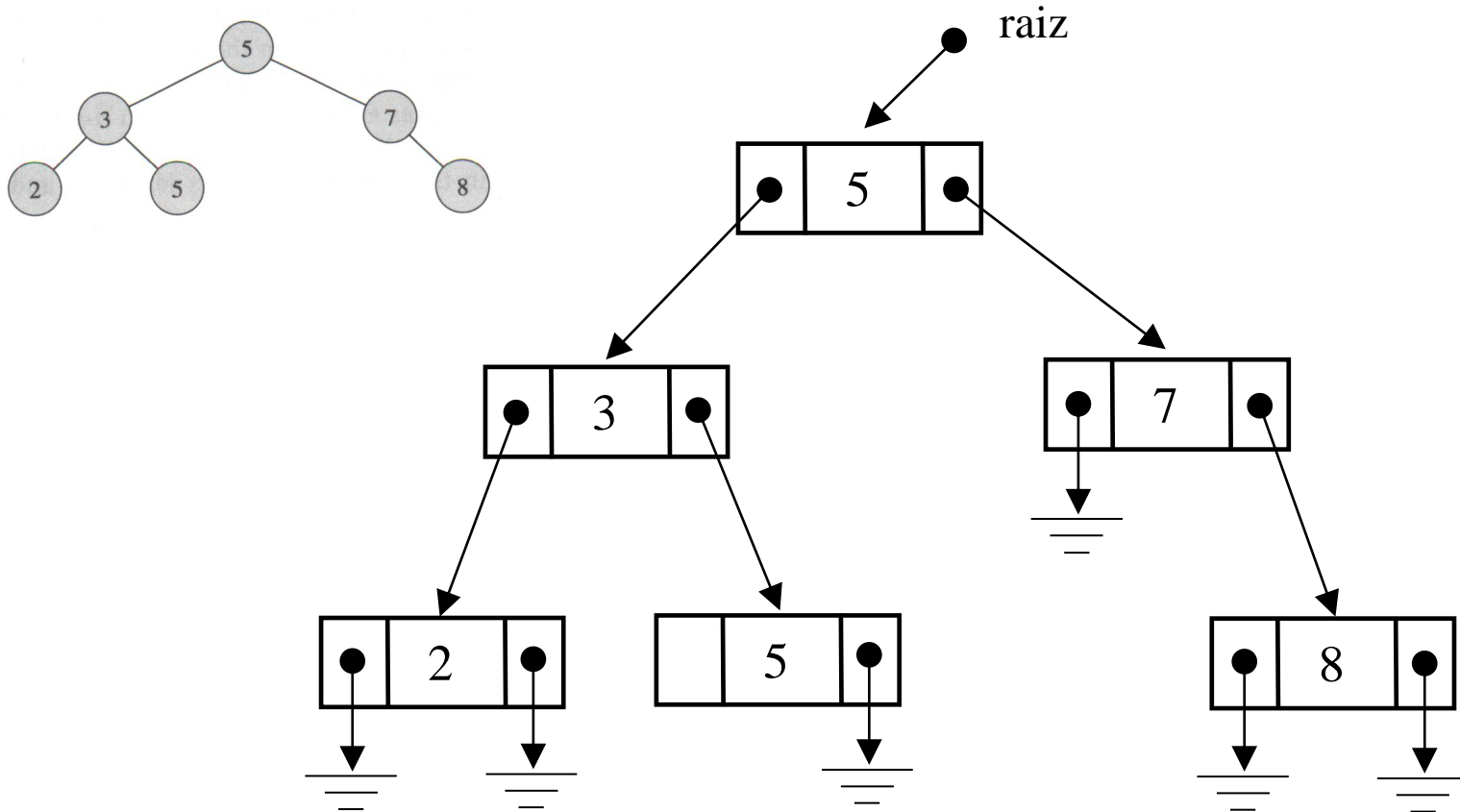
- Por meio de ponteiros;
- Por meio de um vetor;
- Por meio de uma matriz de adjacências;
- Por meio de uma lista de adjacências;
- Outras

Por meio de ponteiros

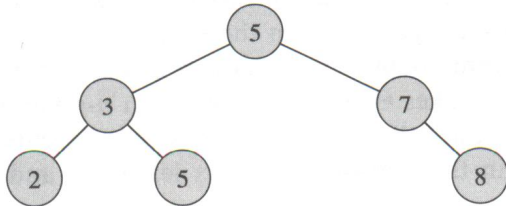
```
registro nó  
{  
  info: inteiro;  
  esq, prox: ponteiro;  
}
```



Por meio de ponteiros



Por meio de um vetor

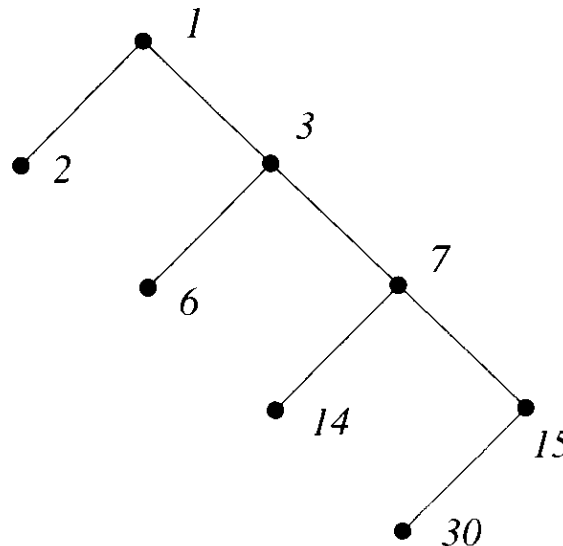


1	2	3	4	5	6	7
5	3	7	2	5		8

- (1) a a raiz é o elemento $A[1]$.
- (2) Os filhos esquerdo e direito do elemento $A[i]$ são, respectivamente, $A[2i]$ e $A[2i + 1]$.

Vantagem e desvantagem da representação por meio de um vetor

- Vantagem: economiza espaço por não usar ponteiros.
- Desvantagem: se a árvore é desbalanceada então muitos nós não existentes são representados. A ilustração abaixo mostra que é necessário um vetor de tamanho igual a 30 para armazenar os 8 nós da árvore representada, gerando desperdício.



Árvore binária de busca

- Uma árvore binária de busca implementa eficientemente as seguintes operações:
 - busca (x): procura o elemento de valor igual a x na estrutura de dados ou determina que x não está presente;
 - inserção (x): insere o elemento x na estrutura de dados.
 - remoção (x): remove o elemento de valor igual a x da estrutura de dados.
- Estruturas de dados que implementam eficientemente estas operações são referenciadas na literatura como sendo dicionários.

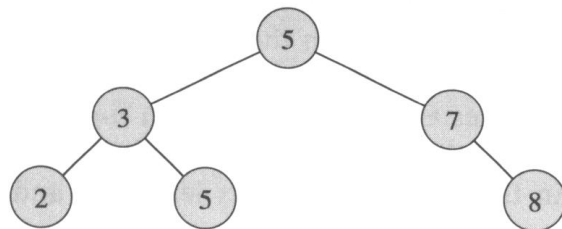
Propriedade

Seja A uma árvore binária cuja raiz é r .

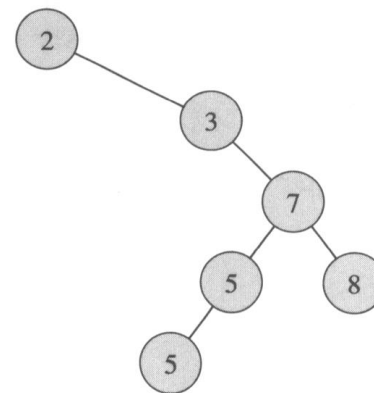
Sejam A_e e A_d as subárvores esquerda e direita de A .

Nós dizemos que A é uma árvore binária de busca se ela satisfizer as seguintes propriedades:

- $x < r < y$ para todo elemento x pertencente a A_e e para todo elemento y pertencente a A_d ;
- A_e e A_d são também árvores binárias de busca.



(a)



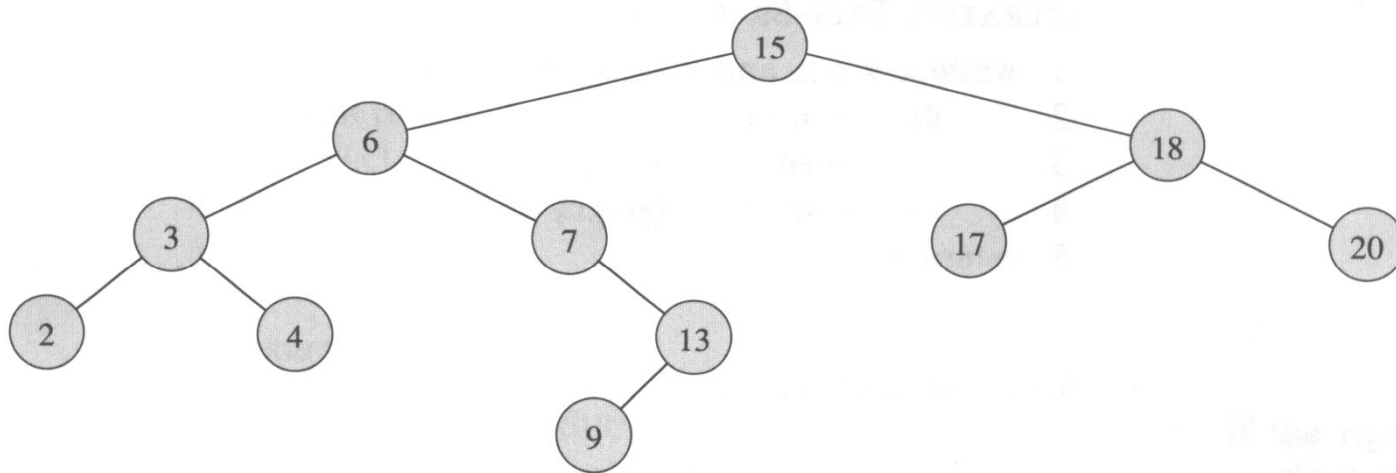
(b)

Percursos

- Inorder
- Preorder
- Posorder
- Por níveis (ou em largura)
- Outros

Inorder

1. Visita a sub-árvore esquerda
Visita a raiz
Visita a sub-árvore direita

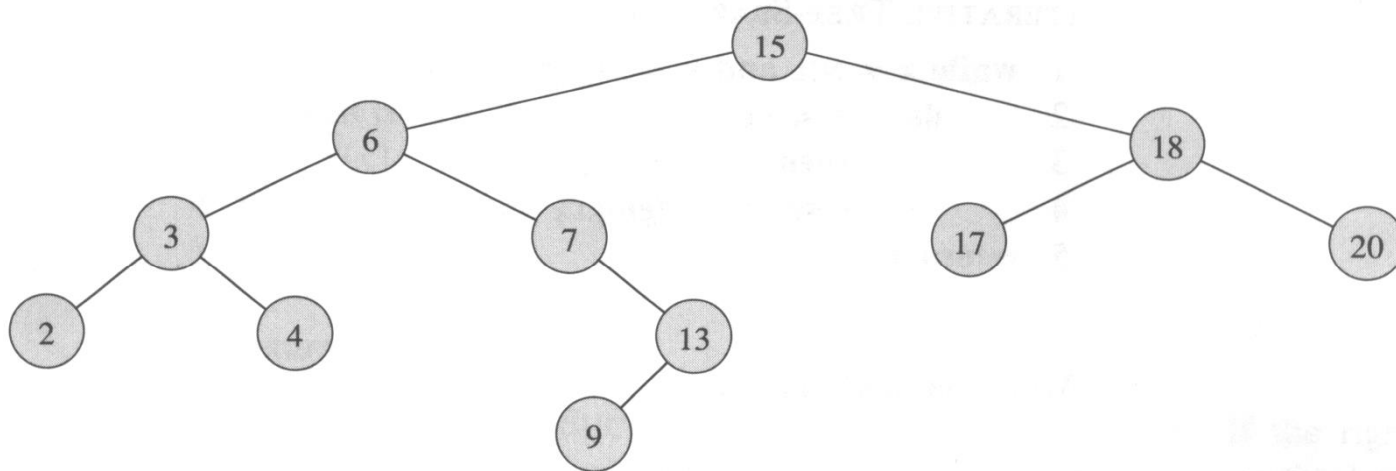


Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

Ordem do percurso: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Preorder

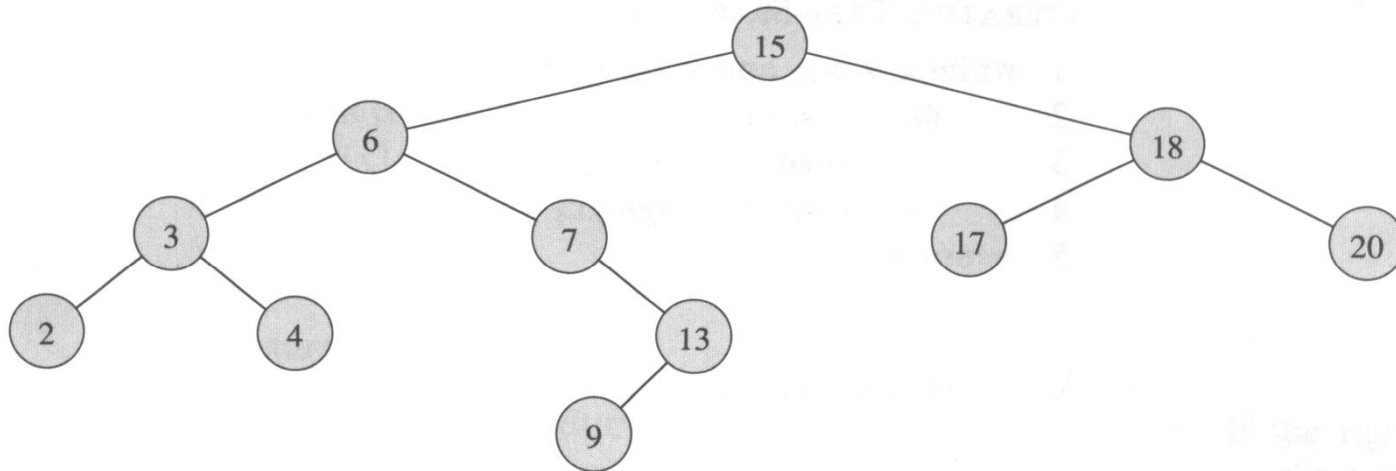
1. Visita a raiz
Visita a sub-árvore esquerda
Visita a sub-árvore direita



Ordem do percurso: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20

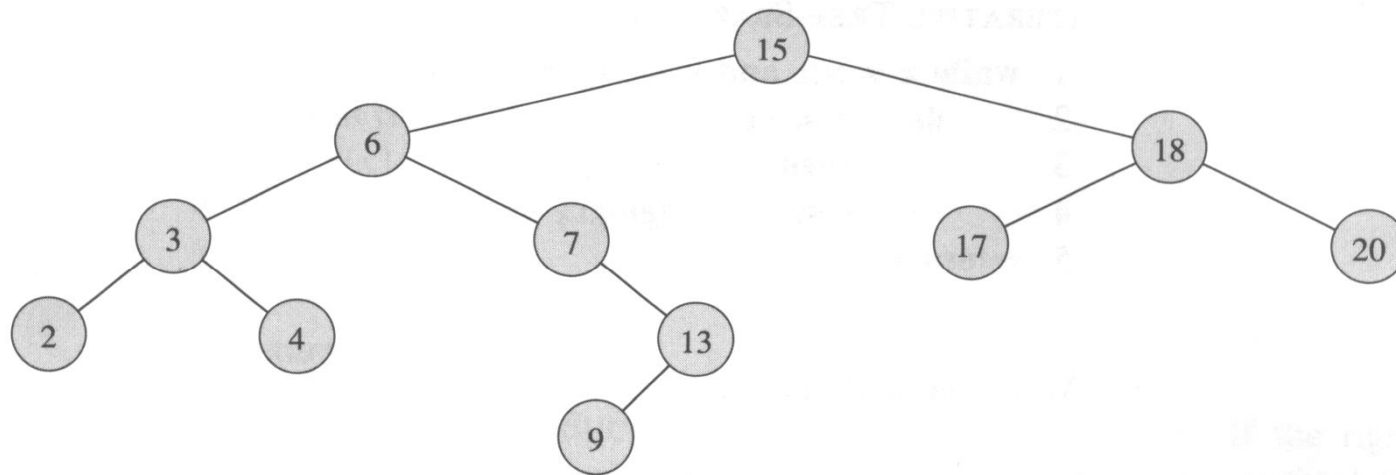
Posorder

1. Visita a sub-árvore esquerda
Visita a sub-árvore direita
Visita a raiz



Ordem do percurso: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

Por níveis (em largura)



Ordem do percurso: 15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

Algoritmos e complexidades

Teorema

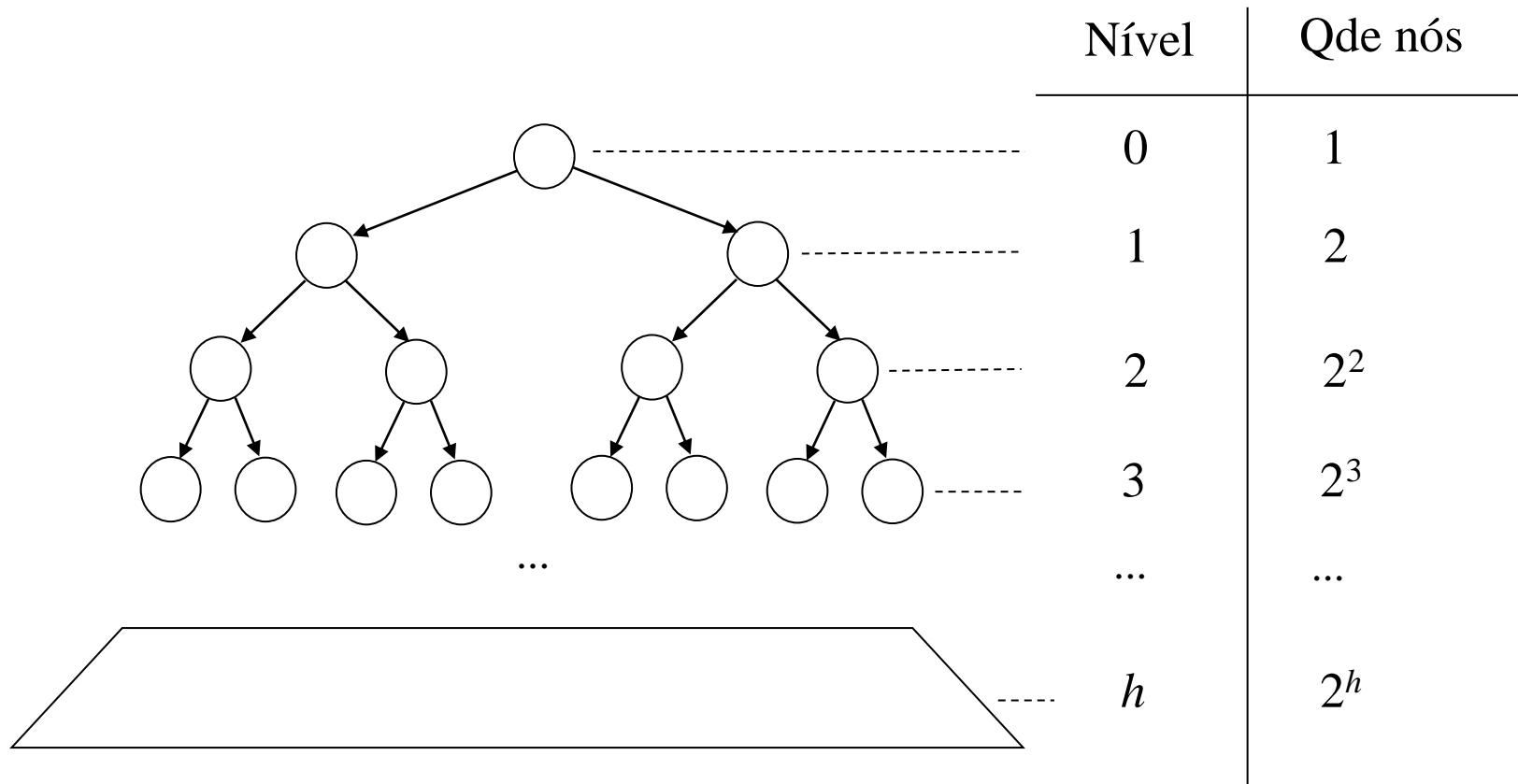
A altura h de uma árvore binária com $n \geq 1$ vértices varia segundo a relação abaixo:

$$\Omega(\log n) \leq h \leq O(n)$$

Mínima: ocorre para a árvore binária é cheia.

Máxima: ocorre quando a árvore binária está estritamente desbalanceada, por exemplo, para a direita.

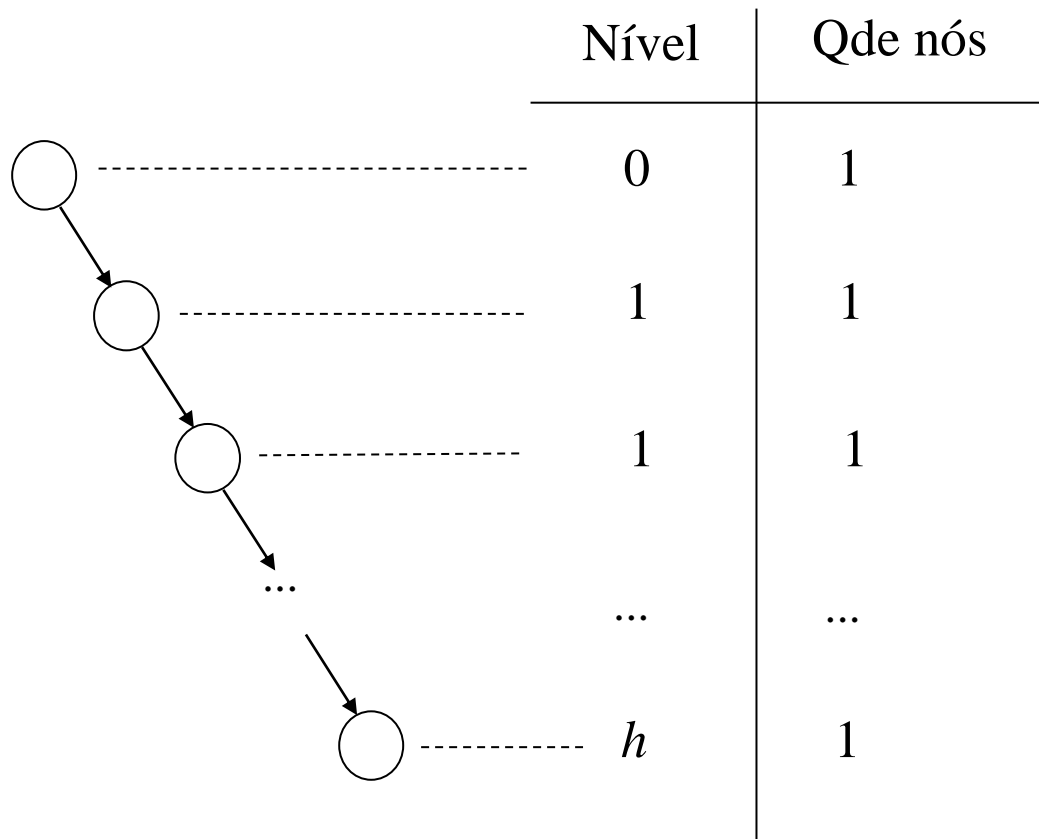
Árvore binária cheia



$$n = 1 + 2 + 2^2 + 2^3 + \dots + 2^h = 2^{h+1} - 1$$

$$h = \log(n + 1) - 1 = \Omega(\log n)$$

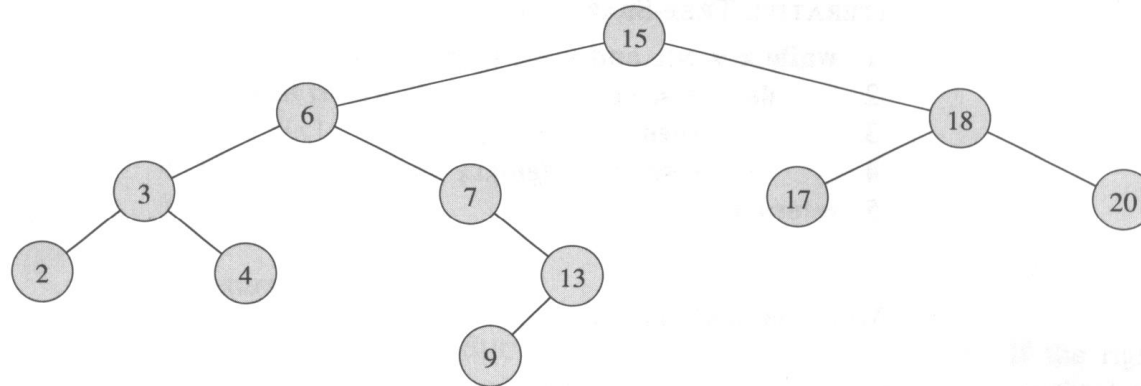
Árvore estritamente desbalanceada à direita



$$n = (h - 0 + 1) \cdot 1 = h + 1$$

$$h = n - 1 = O(n)$$

Busca

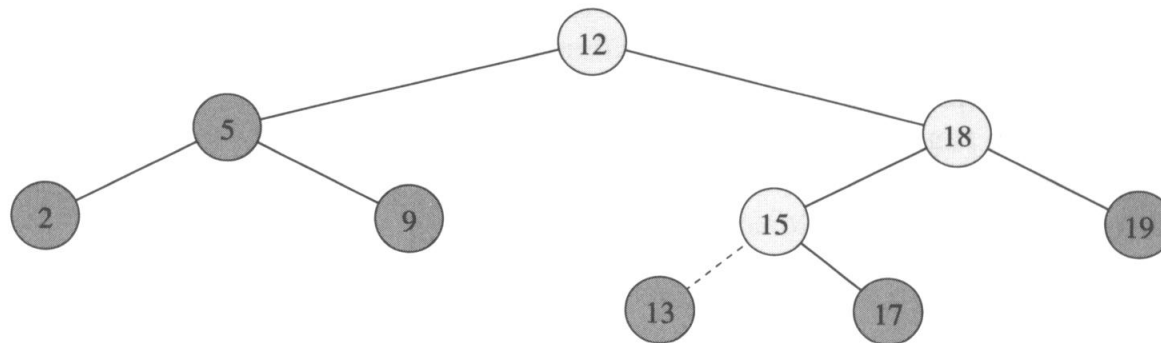


Para achar o elemento 13 na árvore deve-se seguir a rota 15 -> 6 -> 7 -> 13.

Algoritmo e complexidade

Inserção

Inserir o elemento 13.

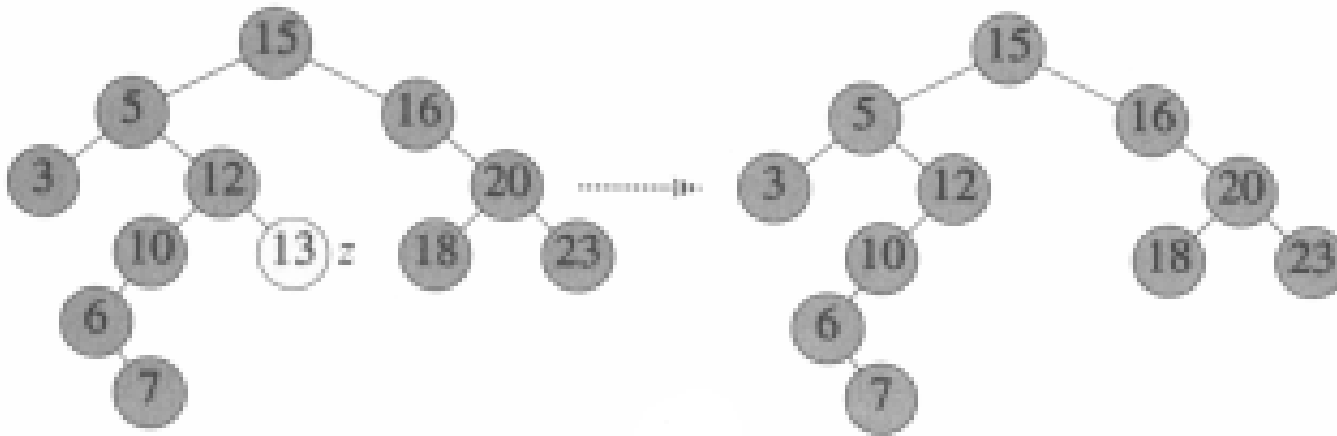


Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

Algoritmo e complexidade

Remoção

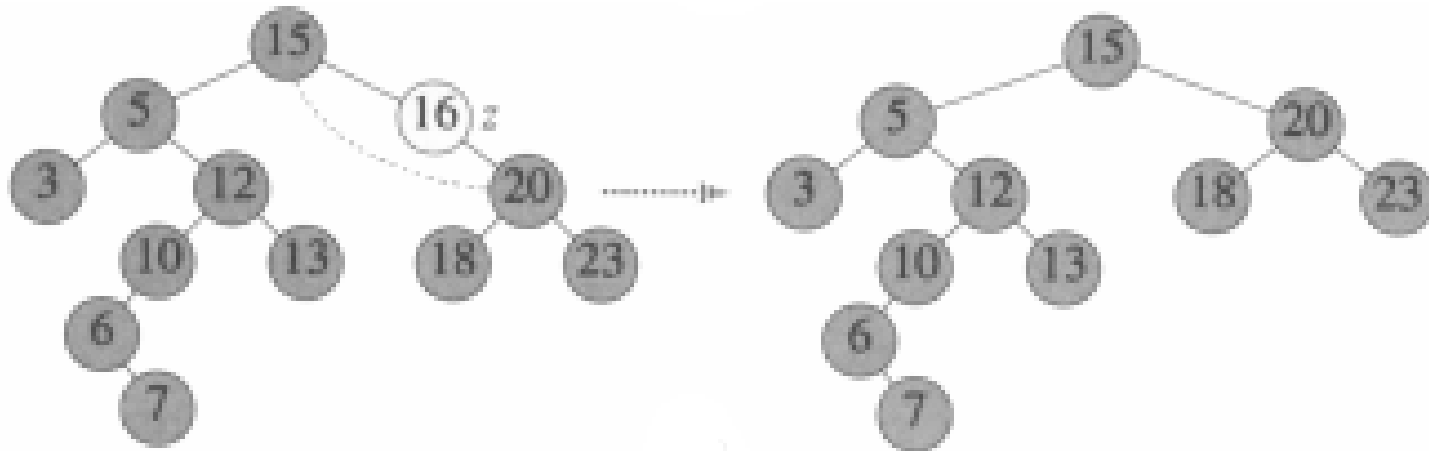
Caso 1: elemento a ser removido não tem filhos.



Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

Remoção

Caso 2: elemento a ser removido tem um filho.

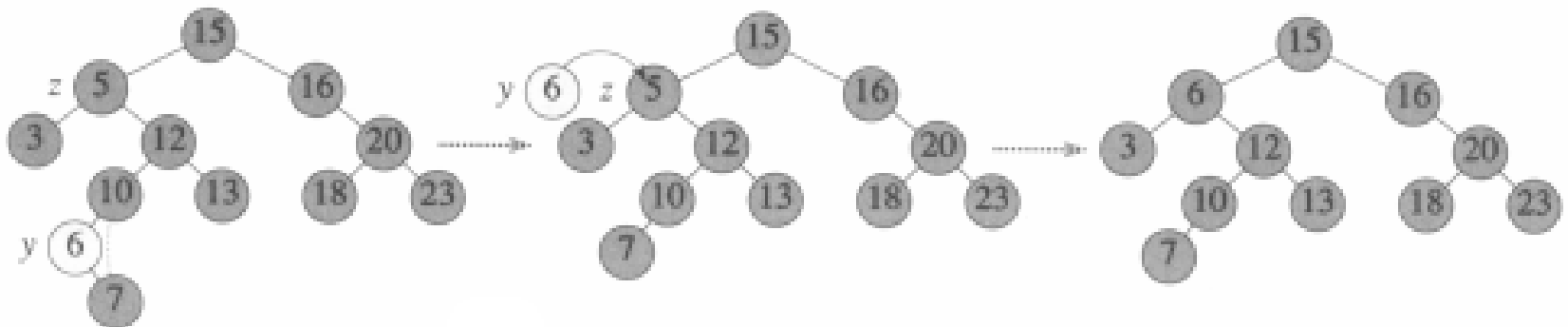


Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

Remoção

Caso 3: elemento a ser removido tem dois filhos.

Desengatar o sucessor de z , o qual tem no máximo um filho (digamos, y), e substituir o conteúdo de z pelo conteúdo de y .



Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

Observações:

- 1) Em vez do sucessor, poderíamos também eleger o predecessor;
- 2) O sucessor de z tem no máximo um filho (veja exercício na lista).

Complexidade

Outros resultados

- Se elementos são inseridos em uma árvore binária de busca segundo uma ordem aleatória, então a altura esperada da árvore é $O(\log n)$.
- O desbalanceamento da árvore pode ser amenizado se, em vez de escolher sempre o nó sucessor para ser removido, nós alternarmos entre o predecessor e o sucessor.
- Remoções podem aumentar a altura da árvore. Se existirem freqüentes remoções seguidas por inserções, a altura da árvore é $O(\sqrt{n})$, mesmo para inserções e remoções aleatórias.

Árvores balanceadas

- 2-3
- AVL
- Graduadas
- Rubro-negras
- B
- Outras

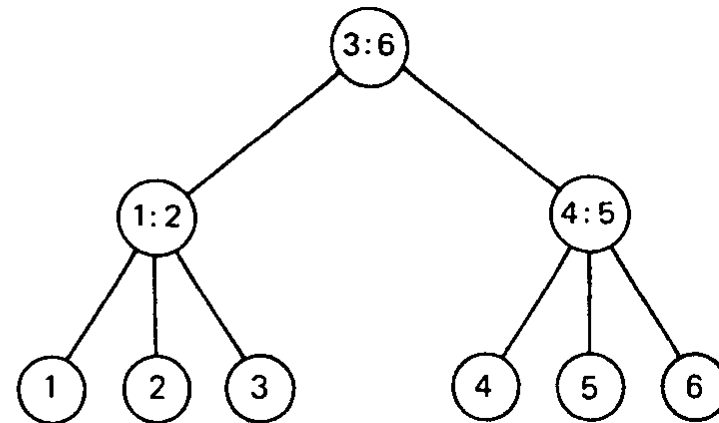
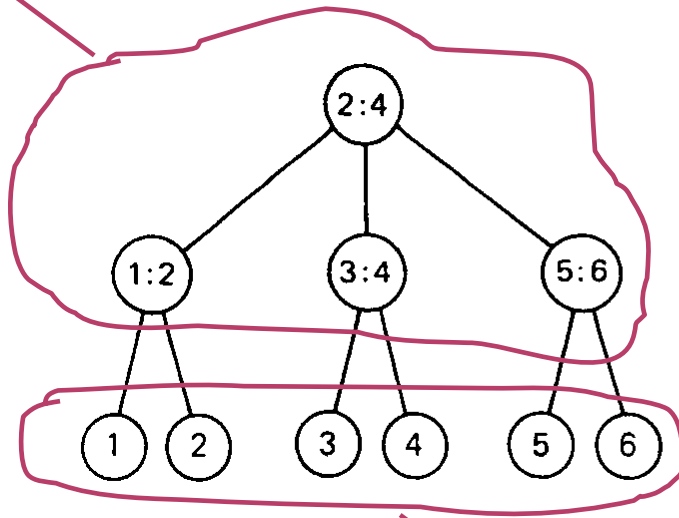
Obs.: Estas árvores oferecem uma solução ao problema de degradação, motivada por possíveis desbalanceamentos, da complexidade de algoritmos que operam sobre árvores binárias de busca.

Árvore 2-3

- Uma árvore 2-3 é uma árvore com as seguintes propriedades:
 - toda folha está na mesma altura;
 - todo nó que não é uma folha (nó interno) possui 2 ou 3 filhos;
 - todo nó interno possui dois valores v_1 e v_2 ;
 - . v_1 é o maior valor encontrado na subárvore esquerda do nó;
 - . v_2 é o maior valor encontrado na subárvore intermediária do nó.

Árvore 2-3

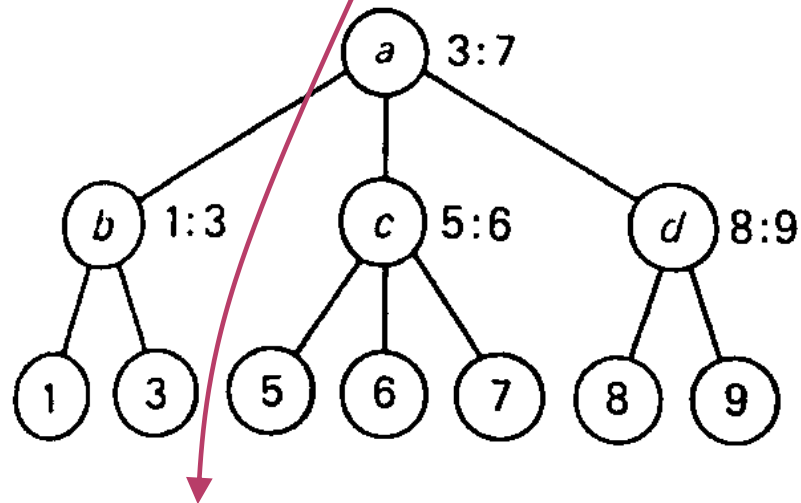
Nós internos: usados
como índice



Folhas: contêm
informação

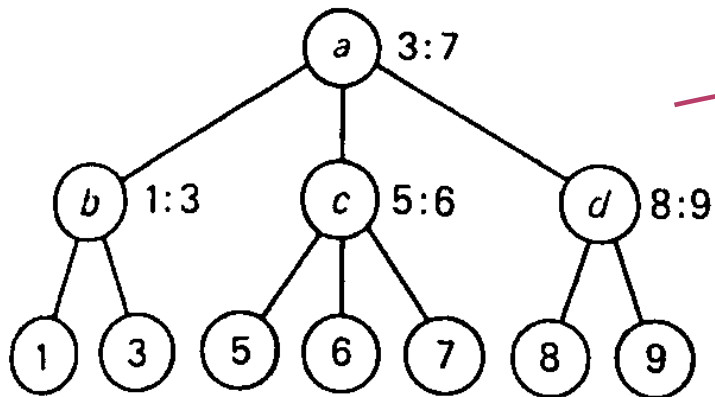
Busca

$$T(n) = O(\log n)$$

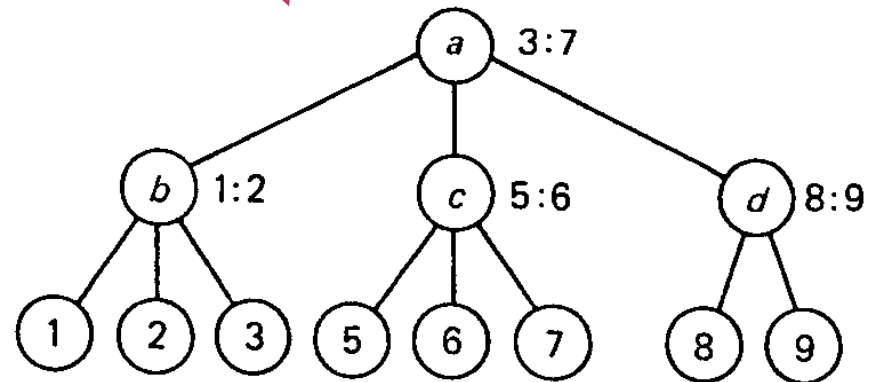


Inserção

Caso 1: inserção sob um pai com dois filhos.

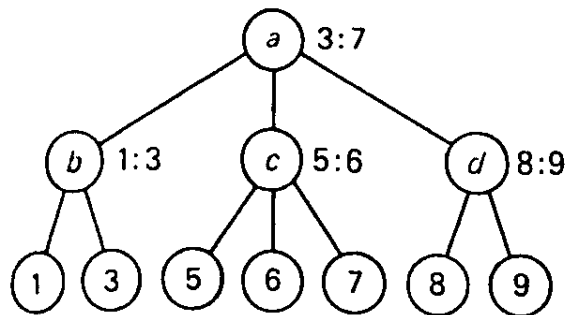


Por exemplo, inserir o elemento 2.

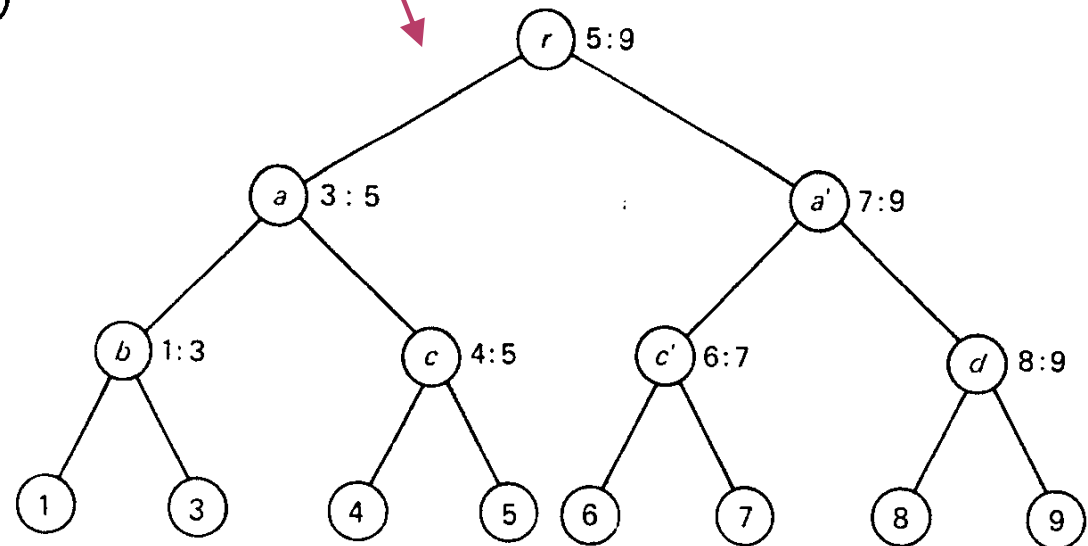


Inserção

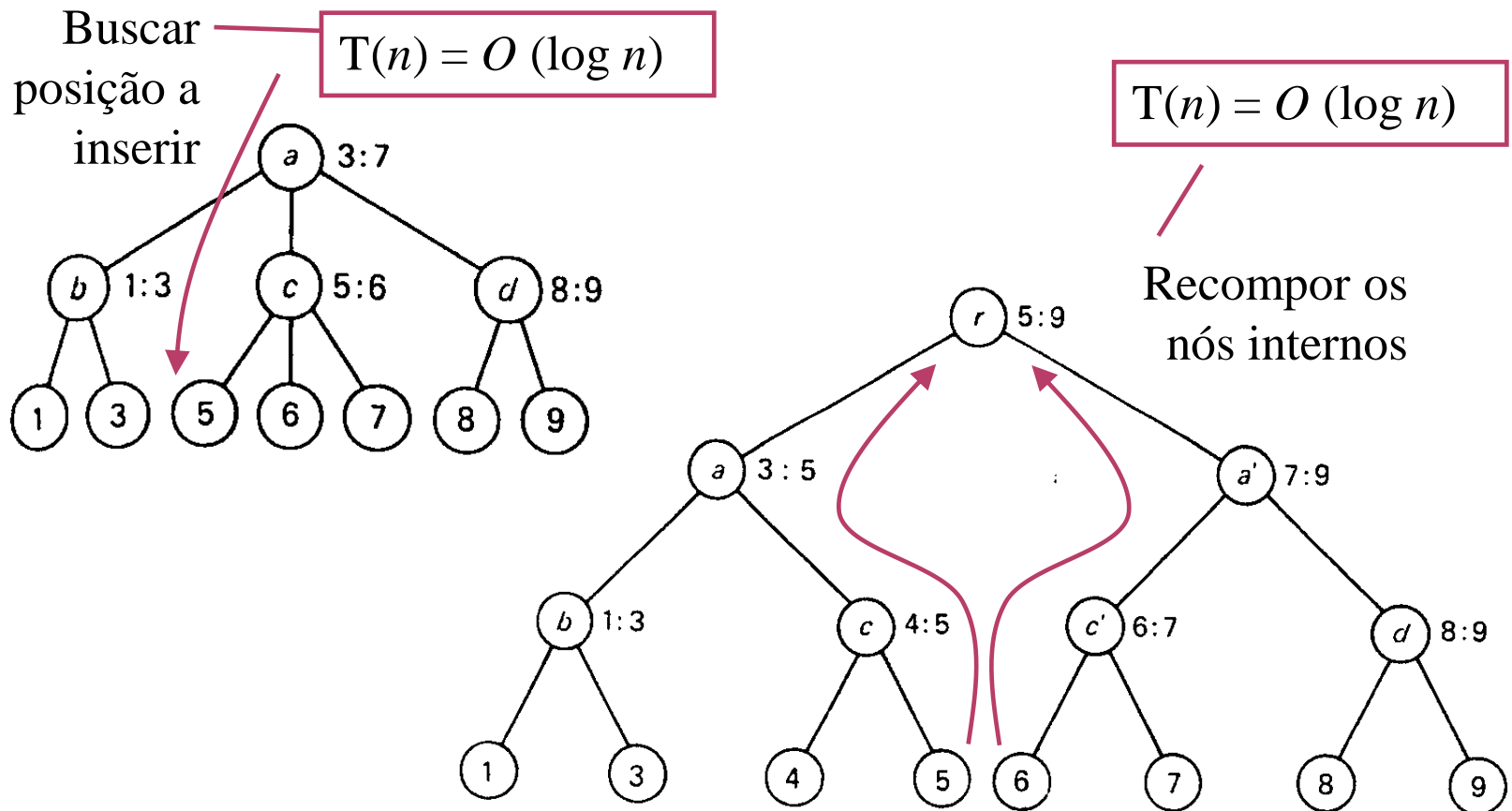
Caso 2: inserção sob um pai com três filhos.



Por exemplo, inserir o elemento 4.



Complexidade

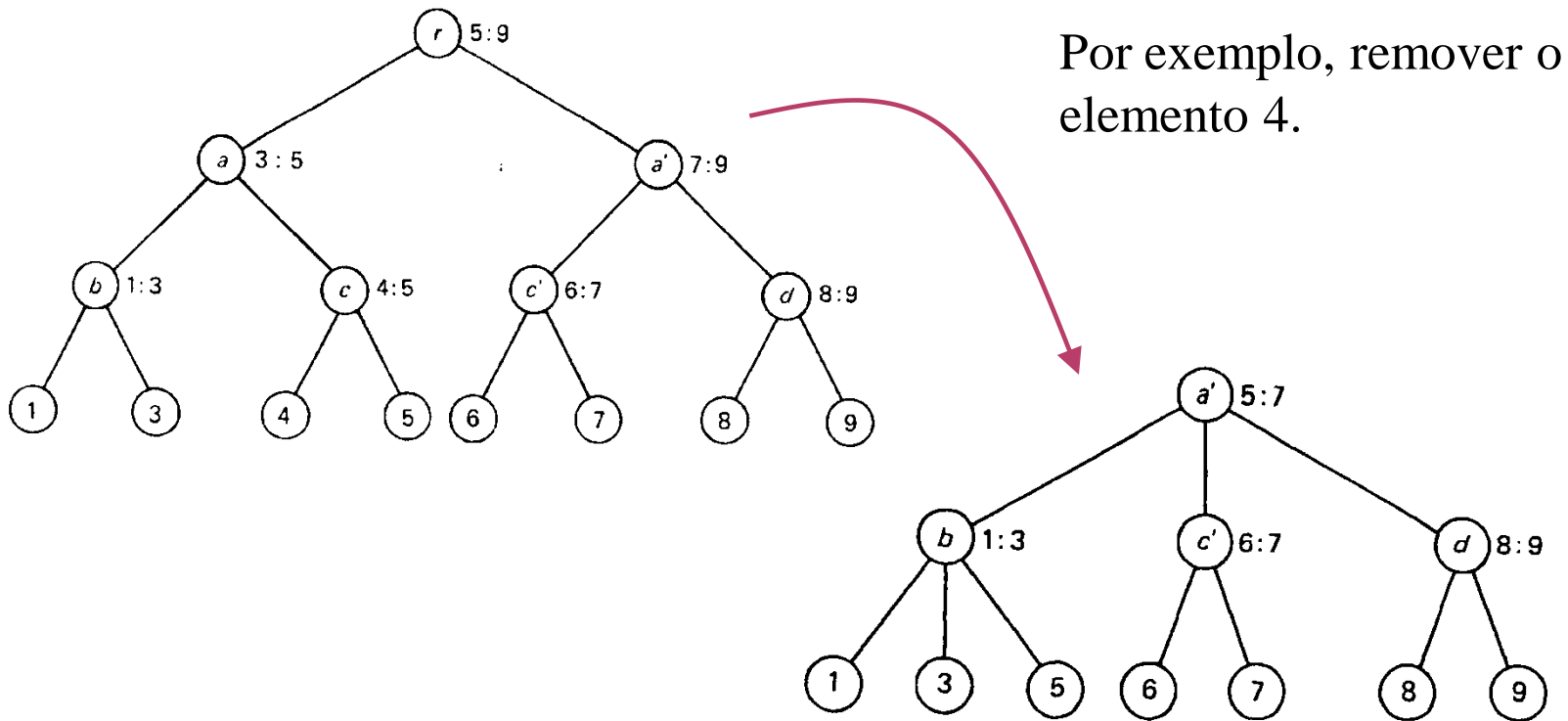


$$T(n) = O(\log n) + O(\log n) = O(2 \log n) = O(\log n)$$

Remoção

Caso 1: remoção sob um pai com três filhos: trivial.

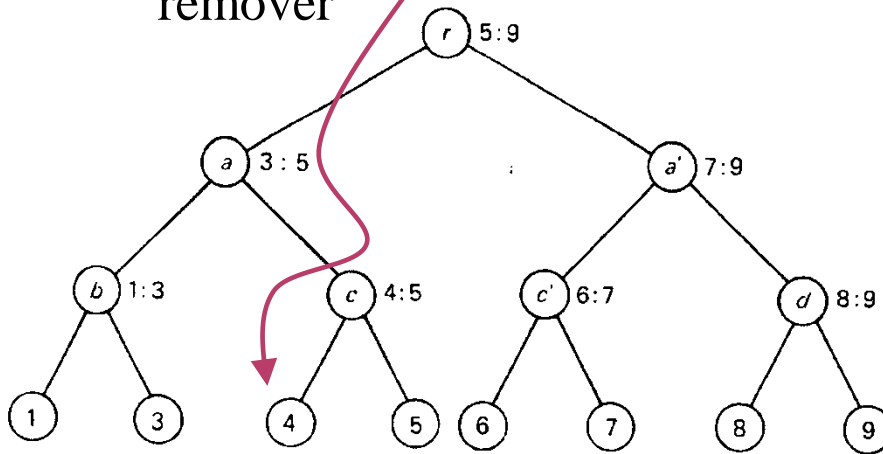
Caso 2: remoção sob um pai com dois filhos



Complexidade

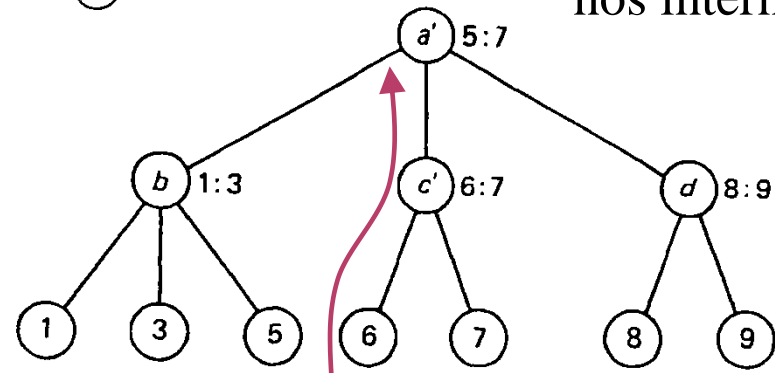
Buscar
elemento a
remover

$$T(n) = O(\log n)$$



$$T(n) = O(\log n)$$

Recompor os
nós internos



$$T(n) = O(\log n) + O(\log n) = O(2 \log n) = O(\log n)$$

Memória consumida

- Nós índice: servem de referências para a informação;
- Nós folha: informação propriamente.

Qual é a relação entre a quantidade de memória gasta por nós índice e a quantidade de memória gasta com nós folha?

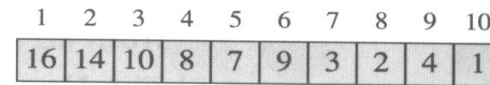
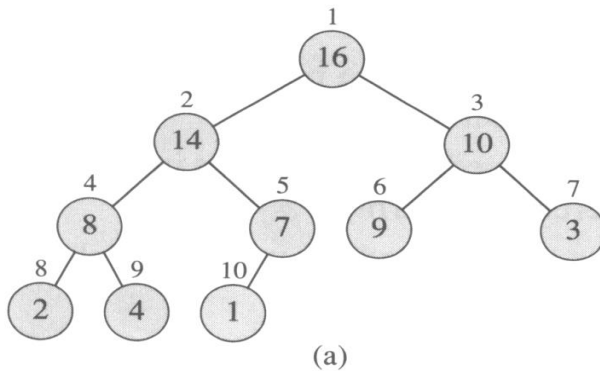
Heap

Seja A uma árvore binária cuja raiz é r .

Sejam A_e e A_d as subárvores esquerda e direita de A , respectivamente.

Nós dizemos que A é um heap se ele satisfizer às seguintes propriedades:

- $r > x$ para todo elemento x pertencente a A_e e a A_d ;
- A_e e A_d são também heaps.



Filas de prioridade

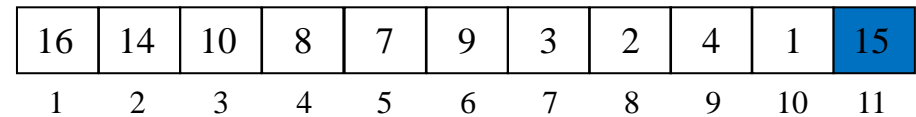
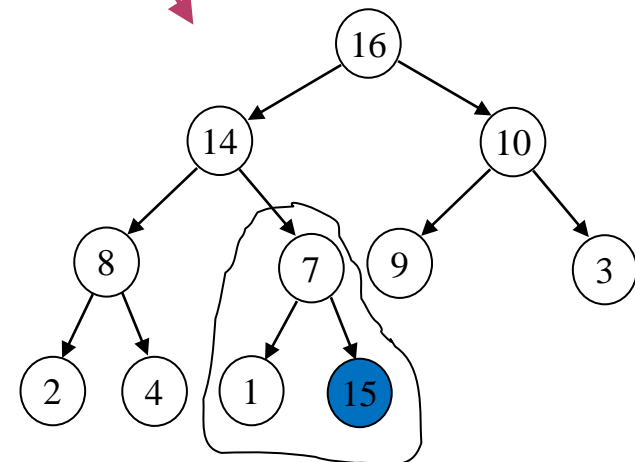
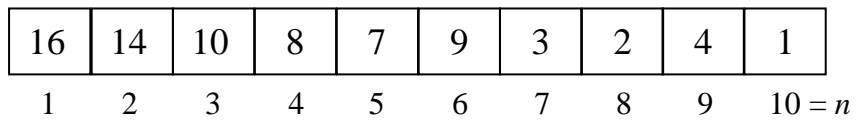
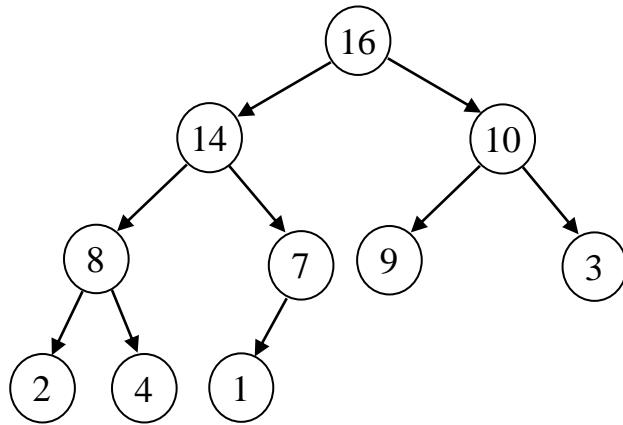
Um heap é uma estrutura de dados útil para implementar filas de prioridade.

Filas de prioridade requerem eficiência na execução das seguintes operações:

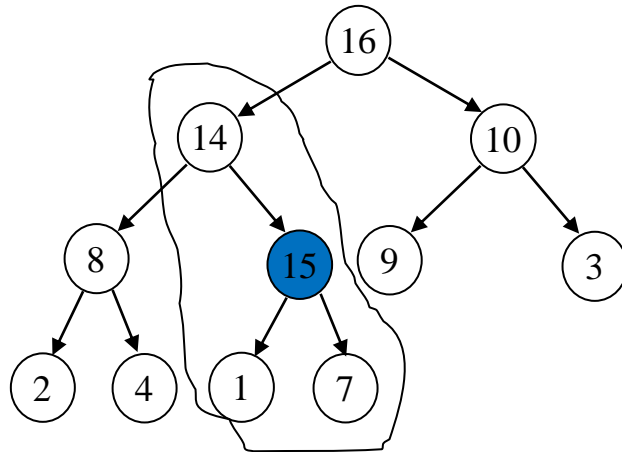
- Inserção (x): insere um elemento x na estrutura de dados;
- Remoção (x): remove o maior elemento da estrutura de dados.

Inserção

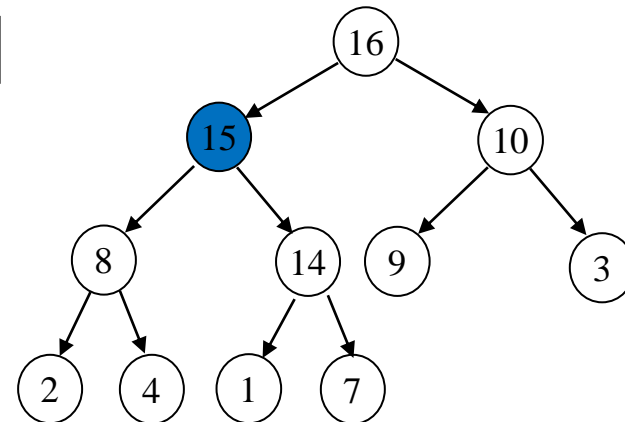
Seja, por exemplo, inserir o número 15



Inserção



16	14	10	8	15	9	3	2	4	1	7
1	2	3	4	5	6	7	8	9	10	11



16	15	10	8	14	9	3	2	4	1	7
1	2	3	4	5	6	7	8	9	10	11

Inserção

Algoritmo InserirHeap (A, n, x)

Entrada: A , um heap de $n \geq 1$ elementos representado como um vetor e x , um elemento a inserir.

Saída: o heap com o elemento x inserido.

```
{  
     $n := n + 1; A[n] := x;$   
  
    // Restabelece a propriedade de heap para toda árvore.  
    filho :=  $n$ ; pai :=  $\lfloor \text{filho} / 2 \rfloor$  ;  
    enquanto ( pai  $\geq 1$  e  $A[\text{filho}] > A[\text{pai}]$  )  
    {  
        troca :=  $A[\text{pai}]$ ;  $A[\text{pai}] := A[\text{filho}]$ ;  $A[\text{filho}] := \text{troca}$ ;  
        filho := pai;  
        pai :=  $\lfloor \text{filho} / 2 \rfloor$   
    }  
}
```

Complexidade

A complexidade deste algoritmo é proporcional à quantidade de vezes que o laço “enquanto” é executado.

A variável “pai” controla o laço e assume valores segundo a P.G. $n, n/2, n/4, \dots, 2, 1$.

A quantidade k de valores assumidos pela variável é igual a quantidade de vezes em que o laço será executado no pior caso.

$$a_k = a_1 \cdot r^{k-1}$$

$$1 = n \cdot (1/2)^{k-1}$$

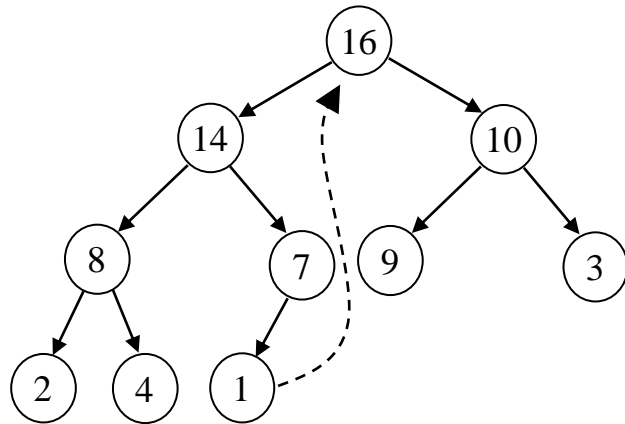
$$2^{k-1} = n$$

$$k - 1 = \log n$$

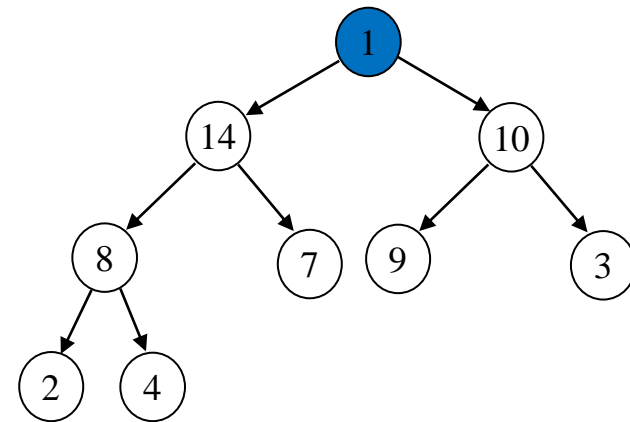
$k = \log(n) + 1$. O laço é executado, no pior caso, $\log(n) + 1$ vezes.

$$T(n) = O(\log n).$$

Remoção

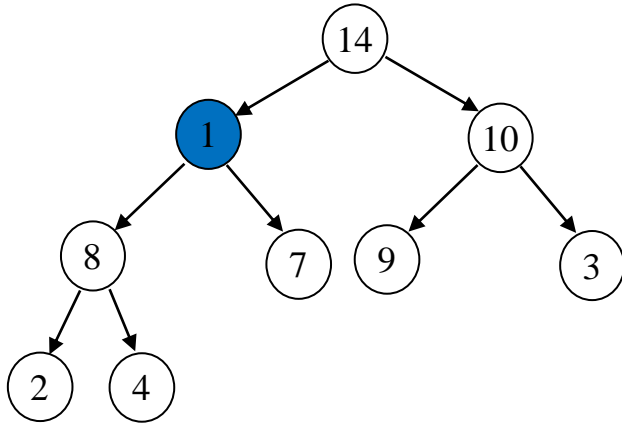


16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10 = n

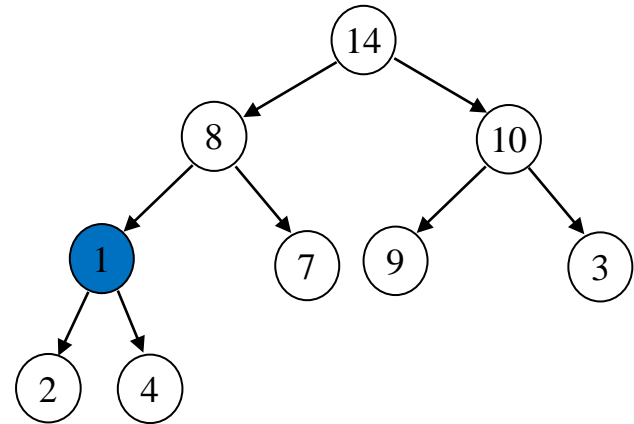


1	14	10	8	7	9	3	2	4
1	2	3	4	5	6	7	8	9

Remoção

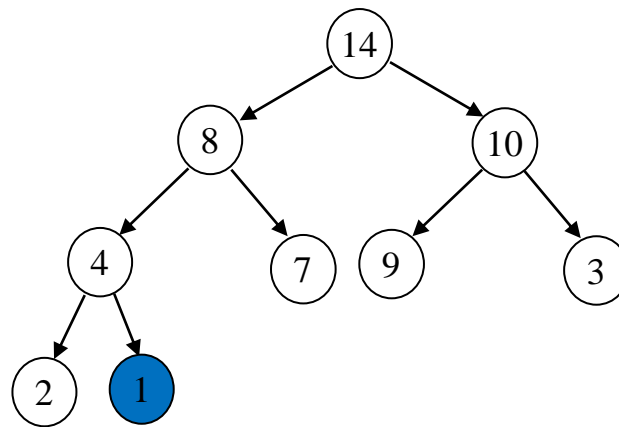


14	1	10	8	7	9	3	2	4
1	2	3	4	5	6	7	8	9



14	8	10	1	7	9	3	2	4
1	2	3	4	5	6	7	8	9

Remoção



14	8	10	4	7	9	3	2	1
1	2	3	4	5	6	7	8	9

Remoção

Escrita do algoritmo será deixada como exercício.

Complexidade:

No pior caso serão realizadas tantas trocas quanto a altura (h) da árvore. Cada troca envolve duas comparações.

$h = O(\log n)$ para árvores binárias balanceadas.

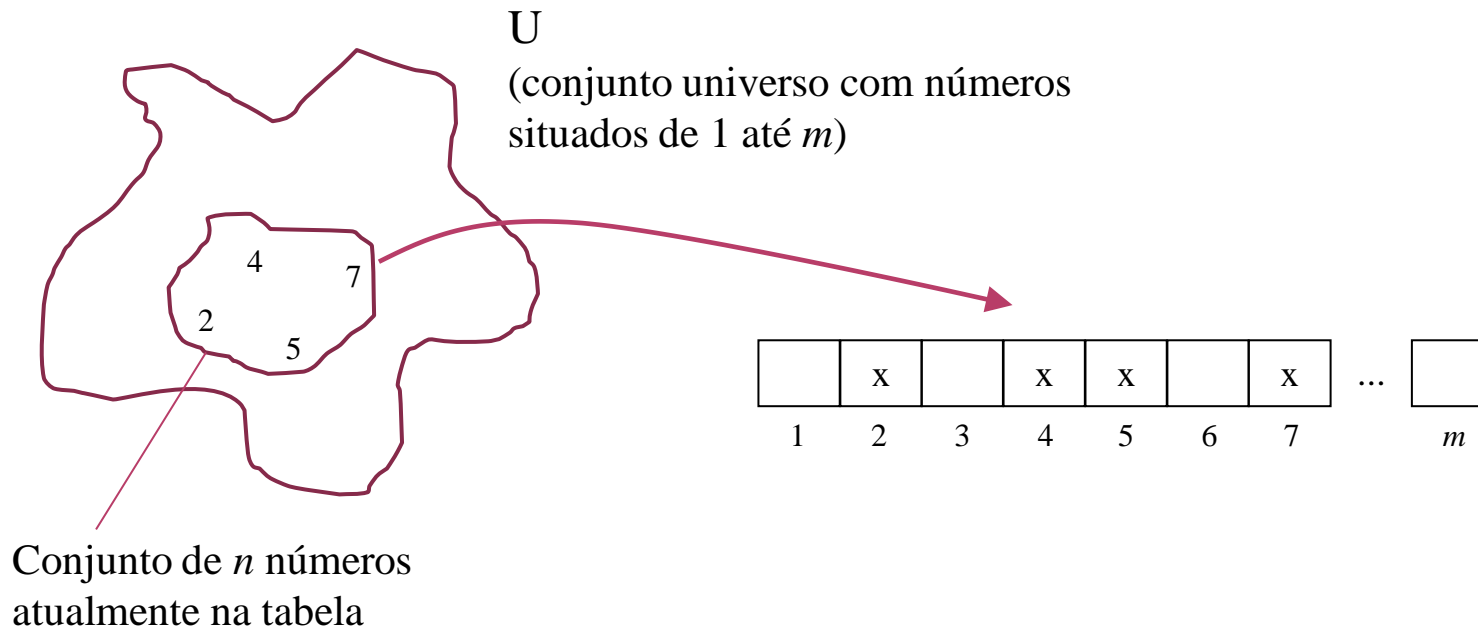
$T(n) = 2h = 2O(\log n) = O(\log n)$.

Tabelas de dispersão (ou espalhamento)

(Hash tables)

Motivação: tabelas de acesso direto

n números inteiros (não repetidos) no intervalo de 1 a m , onde m é pequeno (é possível alocar memória de tamanho igual a m)



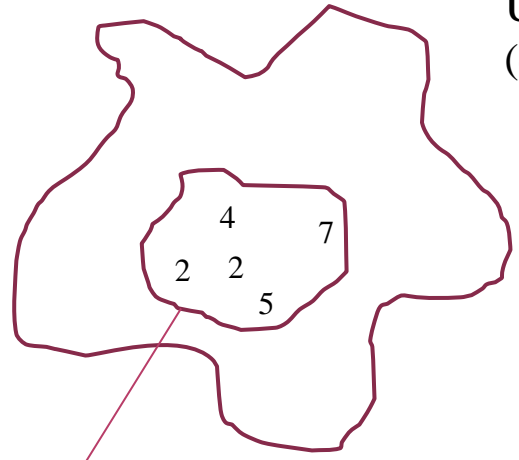
Um problema

E se $n \gggg m$?

Exemplo: 100 milhões de CPFs para serem inseridos em uma tabela indexada de 0 a 999 elementos.

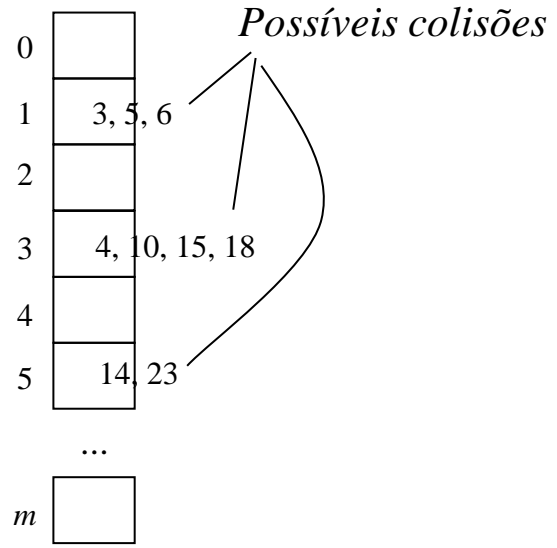
Formalizando

U
(conjunto universo com m números)



n números em um certo instante na tabela

Função de dispersão



Necessidades

- Função de dispersão que mapeia uniformemente;
- Estratégias para resolver as colisões.

Funções de dispersão

- $h(x) = x \bmod m$ (para m primo);
- Se m não puder ser ajustado para um número primo:
 $h(x) = (x \bmod p) \bmod m$ (para algum p primo, $p > m$);
- Outras existem.

Resolvendo as colisões

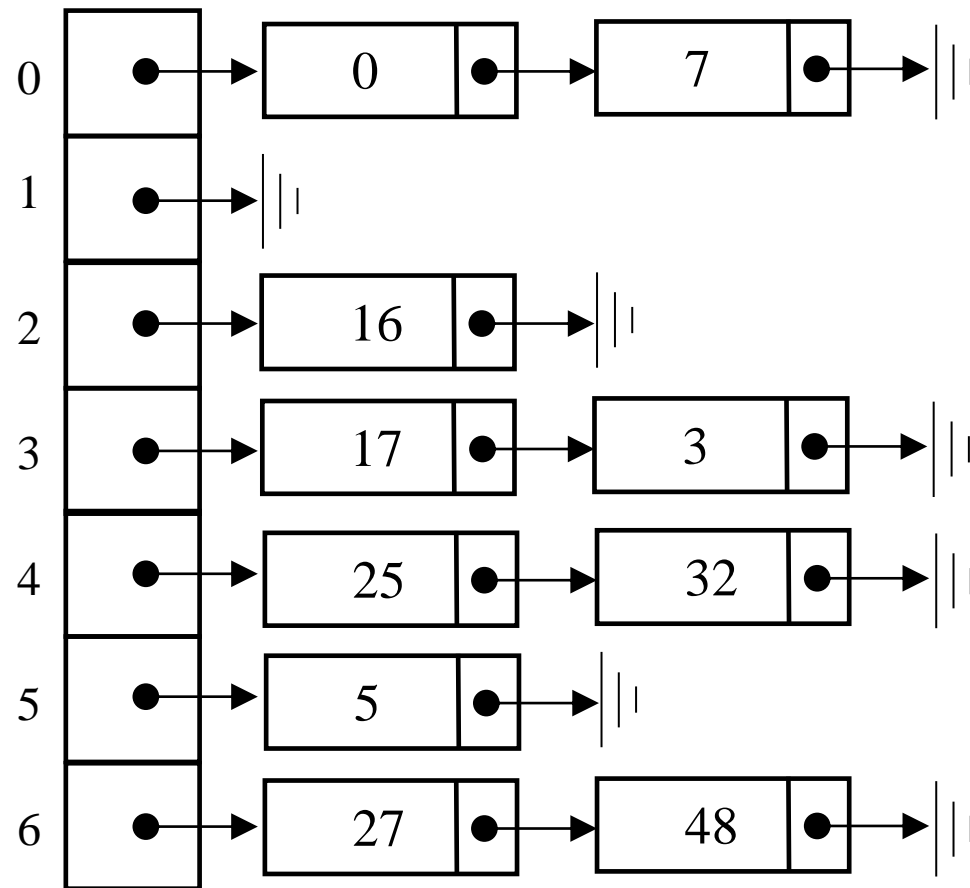
- Por encadeamento exterior;
- Busca linear na tabela;
- Outras estratégias existem.

Encadeamento exterior

Sejam
 $h(x) = x \bmod 7$

E o conjunto de
 elementos:

27, 25, 16, 0, 17, 5, 7, 3,
 48, 32



Busca

Algoritmo Busca (T, x)

Entrada: T , uma tabela de dispersão e x um valor.

Saída: um ponteiro para o elemento da tabela T cujo valor é igual a x ou nil, caso tal elemento não exista.

{

$i := x \bmod m$; // m é tamanho da tabela, indexada de 0 a $m - 1$.

retornar Busca ($T[i], x$) // Busca x na lista apontada por $T[i]$.

}

Complexidade

Supondo que a função mapeia uniformemente, então cada lista terá n/m elementos.

A complexidade do algoritmo é proporcional à busca em uma das listas.

Logo,

$$T(n, m) = O(n/m).$$

Se for possível controlar o valor de m para que $n \leq c m$, para c constante, então $T(n) = O(1)$.

Inserção e remoção

- Algoritmos similares.
- Complexidade
A mesma da busca.

Controlando o fator de carga (tabelas de dimensão dinâmica)

- Fator de carga $f = n / m$.
- Seja $f = 4$ (listas terão tamanho máximo esperado de 4 elementos)
 - 1 - Inserir elementos na tabela.
 - 2 - Quando $f = 4$, dobrar o tamanho da tabela.
 - 3 – Voltar ao passo 1.