ÁRVORES BINÁRIAS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

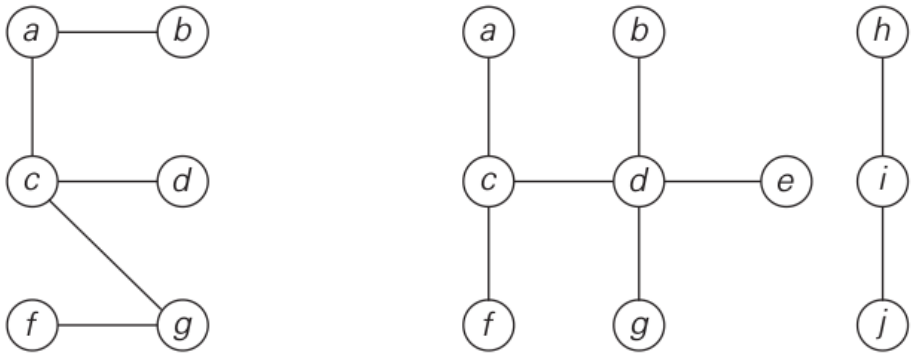
Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Agenda

Introdução

Árvore livre: grafo (V, E), $V \neq \emptyset$, não dirigido, acíclico e conectado

- Motivação: eficiência temporal para inserção e remoção
- Floresta: grafo não dirigido, acíclico, mas não conectado



[1]

Propriedade importante: $| E | = | V | - 1$

- Condição necessária, mas não suficiente
- Se o grafo for não dirigido e conectado, é suficiente

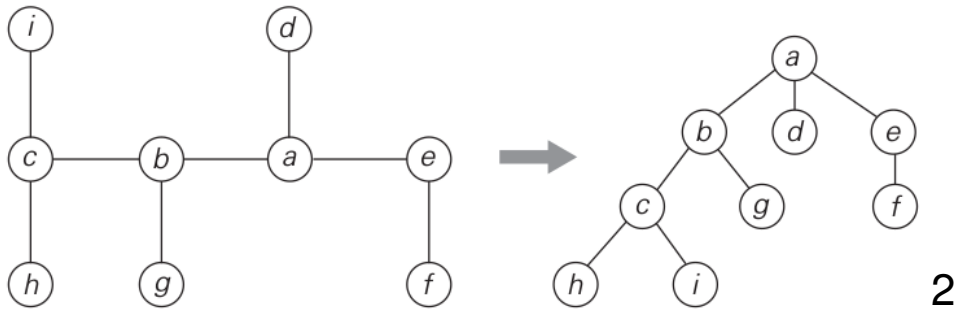
[1] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Para cada dois vértices de uma árvore, sempre existe exatamente um caminho simples entre estes vértices.

- Caminho de tamanho $n \in \mathbb{Z}^+$ de u para v: sequência de arestas $e_1, \cdots, e_n$ tal que $e_1 = \{x_0, x_1\}, \cdots, e_n = \{x_{n-1}, x_n\}$, onde $x_0 = u$ e $x_n = v$.
  - No caso de árvores: sequência de vértices, pois toda árvore é um grafo simples (sem direção, sem arestas paralelas, sem laços)
  - Tamanho de um caminho: quantidade de vértices - 1

Árvore enraizada (muitas vezes, somente árvores): raiz está no nível 0



[2]

[2] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Aplicações: representar hierarquias, implementar dicionários, análise de falhas, base para técnicas como *backtracking* e *branch-and-bound*

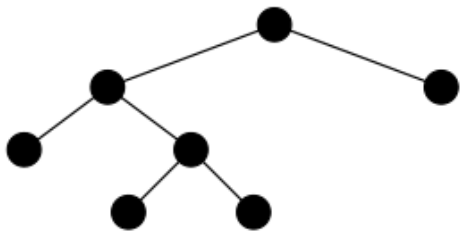
Terminologoia

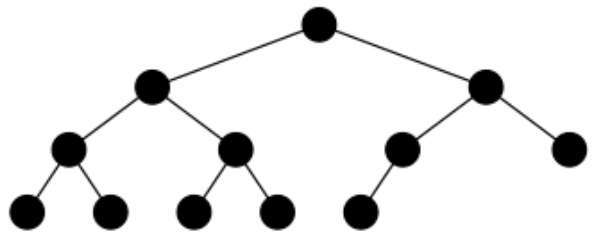
- Ancestrais/Descendentes (próprios)
- Irmãos: vértices com mesmo pai
- Folha: vértice sem filho
- Vértice interno: pelo menos um filho:
- Sub-árvore
- Nível de v: tamanho do caminho simples da raiz para v
- Altura: tamanho do maior caminho simples

Introdução

Terminologia:

- Uma árvore enraizada é m-ária se todo nó interno não possui mais do que m filhos.
  - Árvore m-ária com $m = 2$: árvore binária
- Uma árvore enraizada é m-ária cheia se todo nó interno possui exatamente m filhos.
- Uma árvore enraizada é m-ária completa se seus níveis são preenchidos da esquerda para a direita.





3

[3] Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Introdução

Propriedades de uma árvore m-ária cheia:

- n nós,
  tem $i = (n-1)/m$ nós internos e $l = ((m-1)n+1)/m$ folhas
- i nós internos,
  tem $n = mi+1$ nós e $l = (m-1)i+1$ folhas
- l folhas,
  tem $n = (ml-1)/(m-1)$ nós e $i = (l-1)/(m-1)$ nós internos

Propriedade de uma árvore m-ária:

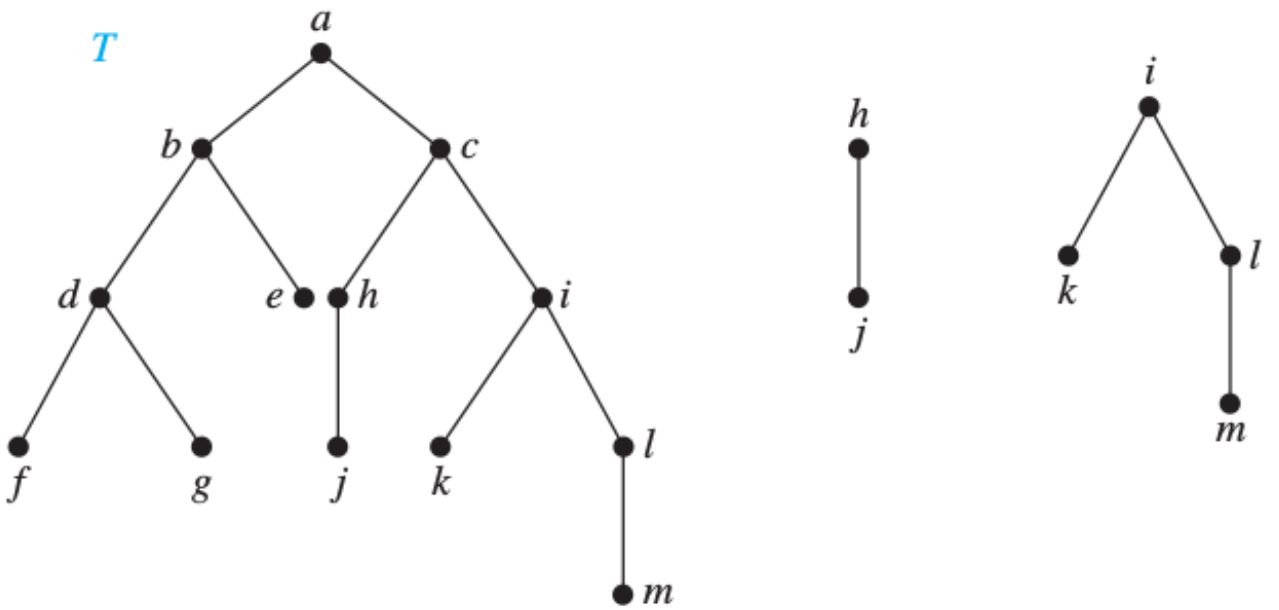
- Existem no máximo m^h folhas em uma árvore m-ária de altura h.

Introdução

Uma árvore enraizada m-ária é ordenada se os filhos de todos os vértices internos estão ordenados.

- Nas binárias: filho (subárvore) à esquerda e à direita

Exemplo: subárvore à esquerda e à direita de *c*

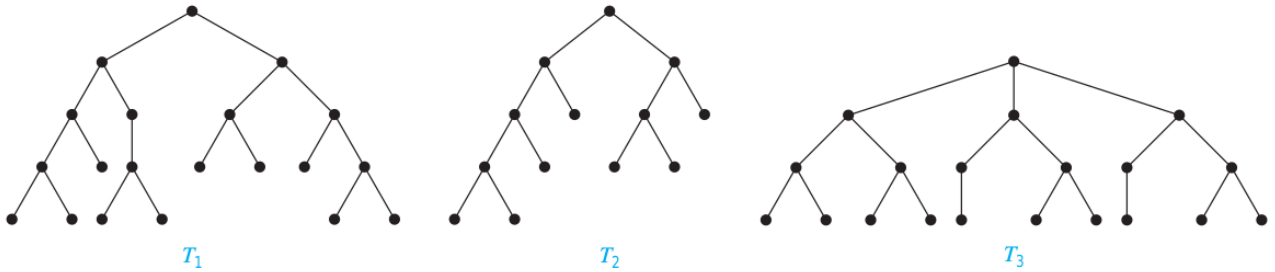


4

[4] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Introdução

Uma árvore enraizada m-ária de altura h é balanceada se todas as folhas estão no nível h ou $h-1$.



[5]

Propriedades:

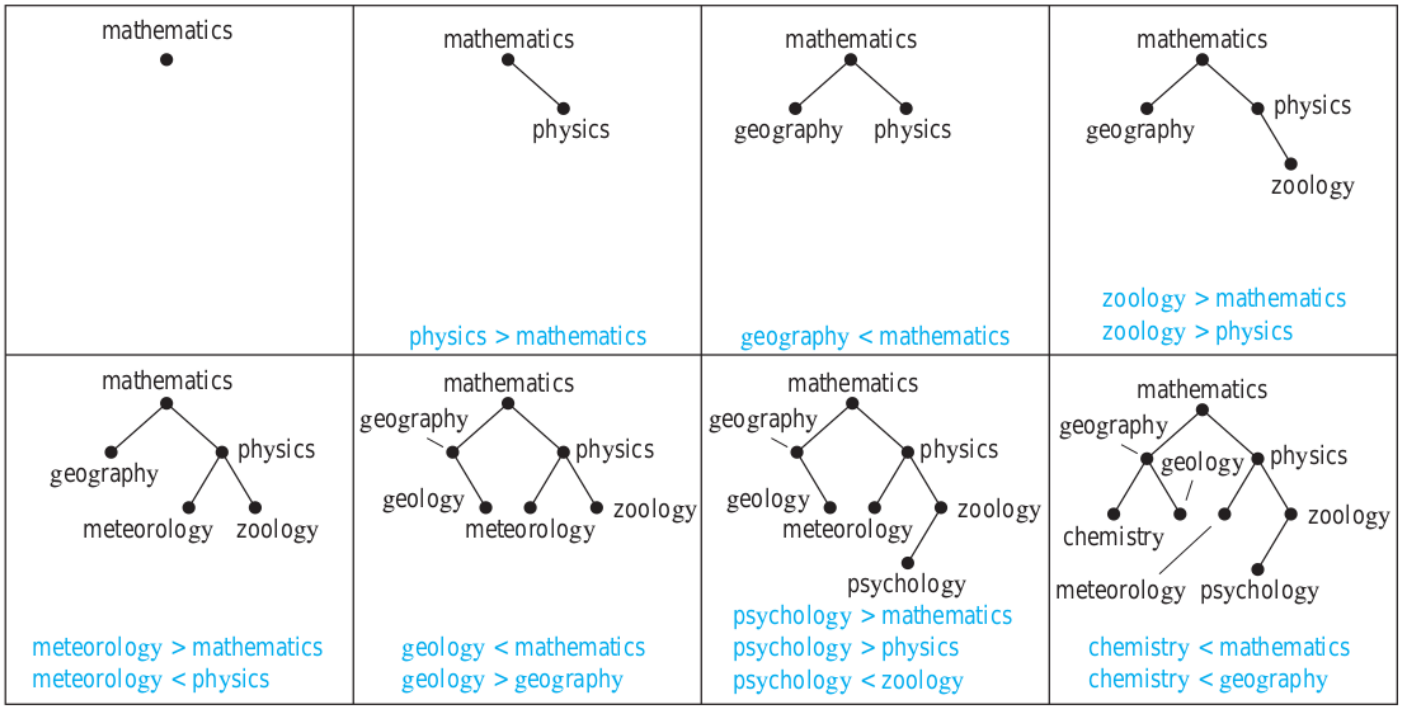
- Em uma árvore m-ária, $h \geq \lceil log_m l \rceil$, $l =$ qtd. de folhas
- Em uma árvore m-ária cheia e balanceada, $h = \lceil log_m l \rceil$

[5] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Introdução

Árvore binária de busca

- Cada vértice é rotulado por uma chave de forma que esta é maior do que as chaves de todos os nós da subárvore esquerda e menor do que as chaves dos nós da subárvore direita.



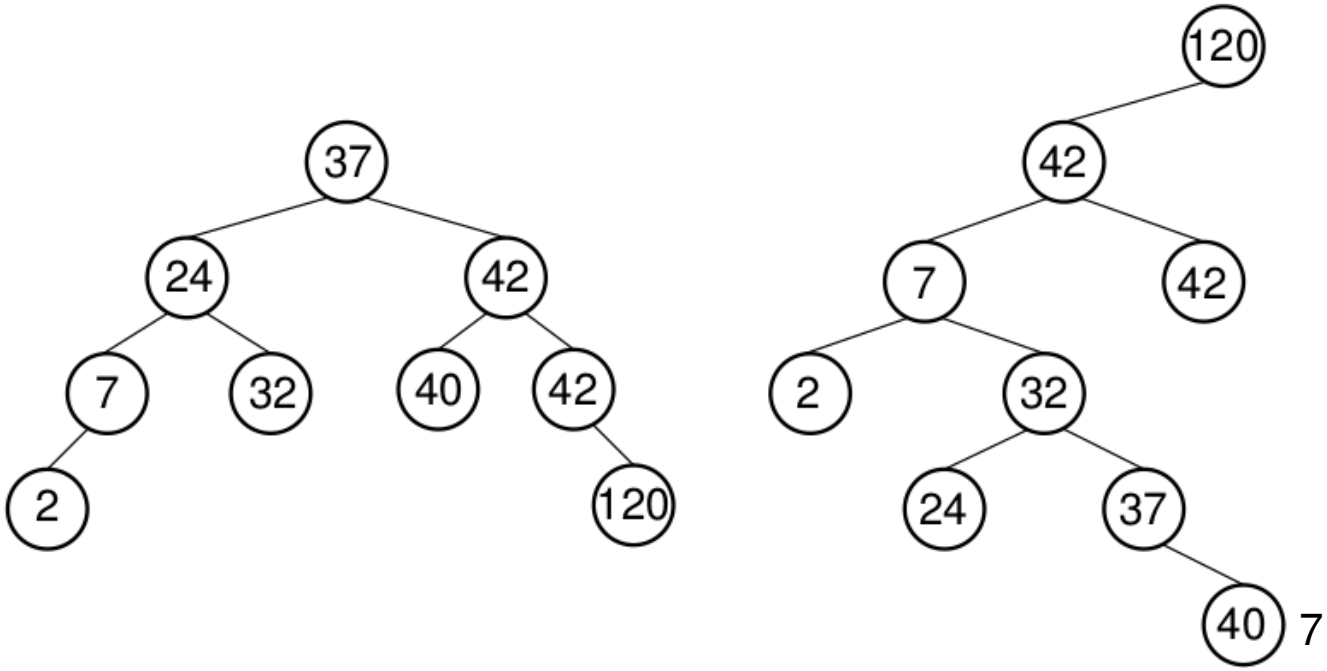
[6]

[6] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Introdução

Árvore binária de busca (BST – *binary search tree*)

- Esquerda: 37, 24, 42, 7, 2, 40, 42, 32, 120
- Direita: 120, 42, 42, 7, 2, 32, 37, 24, 40

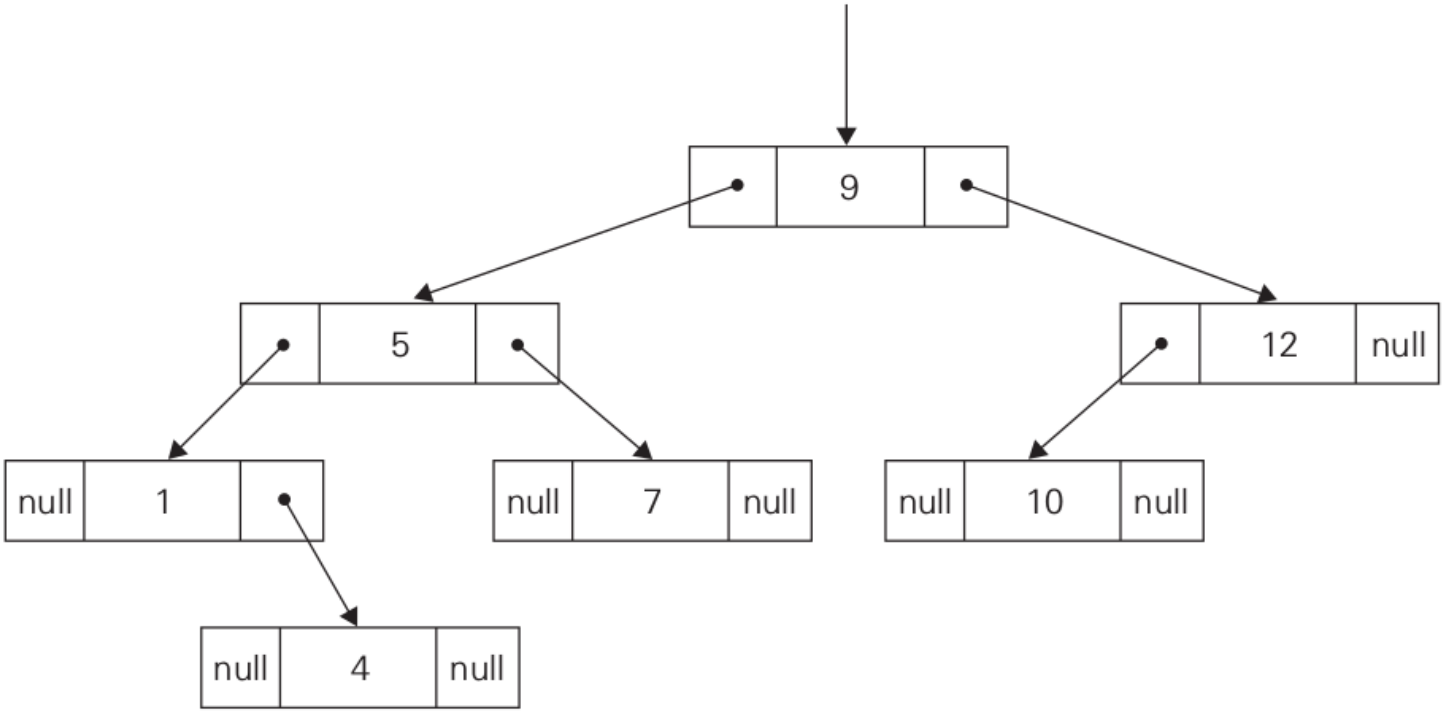


[7] Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Agenda

Implementação de BST

Implementação mais comum: baseada em referências (ponteiros)



8

[8] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Implementação de BST

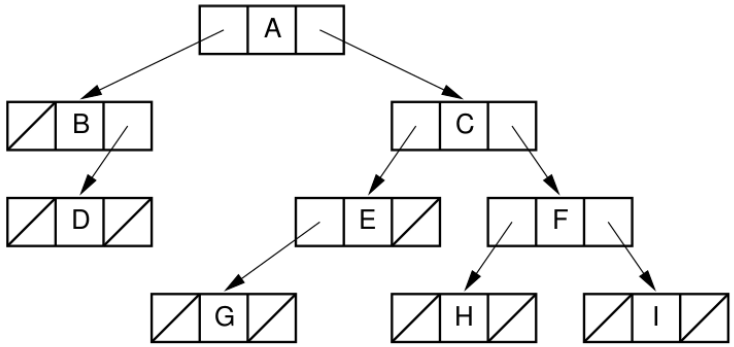
Estrutura de dados (BSTNode):

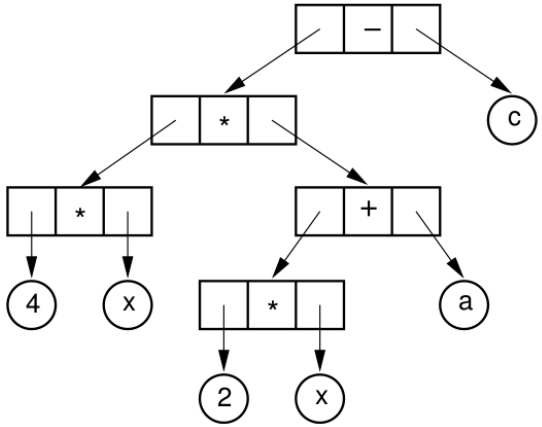
```
1  Key key ;                          // chave
2  E element ;                        // elemento
3  BSTNode left ;                     // filho à esquerda
4  BSTNode right ;                    // filho à direita
```

Em geral, não há necessidade dos filhos apontarem para o pai

Implementação de BST

Decisão importante: definição (não) uniforme de nós internos e folhas[9]





10

[9] C. Shaffer. Data Structures and Algorithm Analysis. Seção 5.3. 2013.

[10] Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Implementação de BST

Estrutura de dados (BST):

```
1  BSTNode root ;                                  // raiz
2  int nodecount ;                    // quantidade de elementos
```

Algoritmo: BST create_bst()

```
1  bst.root ← NULL;
2  bst.nodecount ← 0;
3  return bst;
```

Algoritmo: void clear(BST bst)

```
1  bst.root ← NULL;
2  bst.nodecount ← 0;
```

Implementação de BST

Algoritmo: int size(BST bst)

```
1  return bst.nodecount;
```

Algoritmo: E find(BST bst, Key k)

```
1  return findhelp(bst.root, k);
```

Implementação de BST

Algoritmo: E findhelp(BSTNode rt, Key k)

```
1  if rt = NULL then return NULL ;
2  if rt.key > k then
3  |    return findhelp(rt.left, k);
4  else if rt = k then
5  |    return rt.element;
6  else
7  |    return findhelp(rt.right, k);
```

Implementação de BST

Algoritmo: void insert(BST bst, Key k, E e)

```
1  bst.root ← inserthelp(bst.root, k, e);
2  bst.nodecount++;
```

Algoritmo: BSTNode inserthelp(BSTNode rt, Key k, E e)

```
1  if rt = NULL then return create_bstnode(k, e) ;
2  if rt.key > k then
3  |    rt.left ← inserthelp(rt.left, k, e);
4  else
5  |    rt.right ← inserthelp(rt.right, k, e);
6  return rt;
```

Importante: na presença de chaves repetidas, sub-árvore direita!

Implementação de BST

Algoritmo: E remove(BST bst, Key k)

```
1  E temp ← findhelp(bst.root, k);
2  if temp ≠ NULL then
3      bst.root ← removehelp(bst.root, k);
4      bst.nodecount--;
5  return temp;
```

Algoritmo: E removeAny(BST bst)

```
1  if bst.root = NULL then return NULL;
2  E temp ← bst.root.element;
3  bst.root ← removehelp(bst.root, bst.root.key);
4  bst.nodecount--;
5  return temp;
```

Implementação de BST

Algoritmo: BSTNode removehelp(BSTNode rt, Key k)

```
1  if rt = NULL then return NULL;
2  if rt.key > k then
3      rt.left ← removehelp(rt.left, k);
4  else if rt.key < k then
5      rt.right ← removehelp(rt.right, k);
6  else
7      if rt.left = NULL then  return rt.right ;
8      else if rt.right = NULL then  return rt.left ;
9      else
10         BSTNode temp ← getmin(rt.right);
11         rt.element ← temp.element;
12         rt.key ← temp.key;
13         rt.right ← deletemin(rt.right);
14 return rt;
```

Implementação de BST

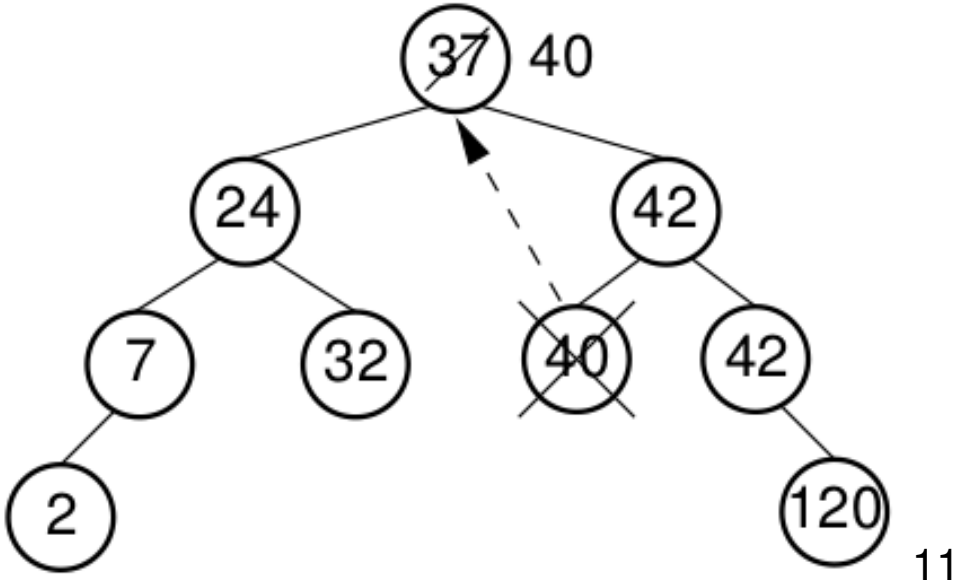
Algoritmo: BSTNode getmin(BSTNode rt)

```
1  if rt.left = NULL then return rt;
2  return getmin(rt.left);
```

Algoritmo: BSTNode deletemin(BSTNode rt)

```
1  if rt.left = NULL then return rt.right;
2  rt.left ← deletemin(rt.left);
3  return rt;
```

Implementação de BST



[11]

[11] Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Custo computacional

Relembrando (caso médio = considerando BST balanceada):

Data Structure	Time Complexity								Space Complexit
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

12

[12] http://bigocheatsheet.com/

Verificando se é BST

Algoritmo: boolean checkBST(BinNode rt, int low, int high)

```
1  if rt = NULL then return true;
2  int rootkey ← rt.element;
3  if rootkey < low || rootkey > high then
4      return false;                          // Fora dos limites
5  if !checkBST(rt.left, low, rootkey) then
6      return false;                          // Lado esquerdo falhou
7  return checkBST(rt.right, rootkey, high);
```

Para uma BST inteira *bst*, chamar inicialmente:
checkBST(*bst.root*, *MIN_INT*, *MAX_INT*).

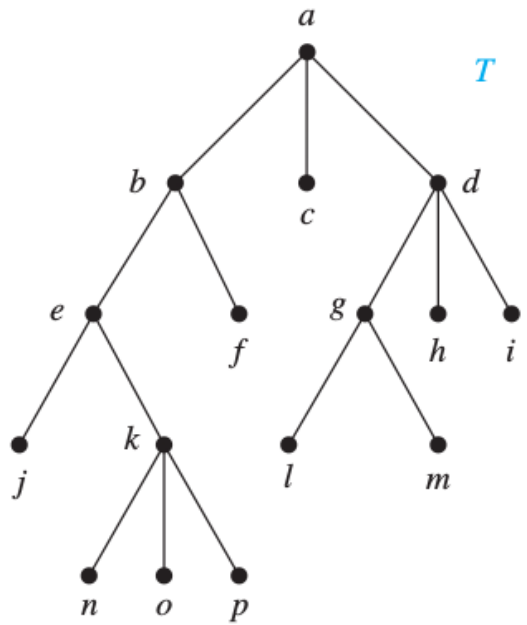
Agenda

Caminhando em árvores

Seja T uma árvore enraizada e ordenada com raiz r. Se T possui apenas r, então o caminhamento em pré-ordem de T é r. Caso contrário, sejam $T_1, T_2, ..., T_n$ as subárvores de r da esquerda para a direita. O caminhamento em pré-ordem começa visitando r e continua fazendo um caminhamento em pré-ordem em $T_1, T_2, ..., T_n$.

Caminhando em árvores

Encontre o caminhamento pré-ordem da árvore abaixo.

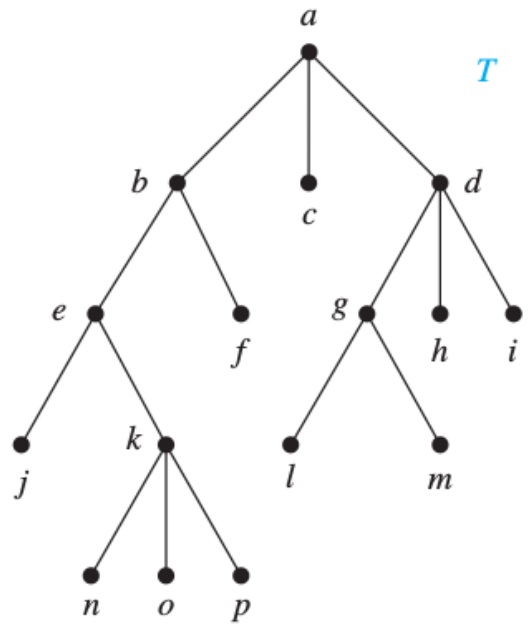


13

[13] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Caminhando em árvores

Encontre o caminhamento pré-ordem da árvore abaixo.



13

Resposta: $a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i$

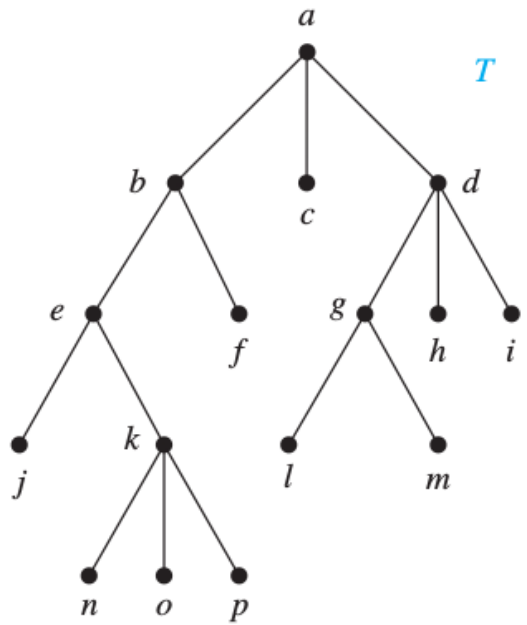
[13] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Caminhando em árvores

Seja T uma árvore enraizada e ordenada com raiz r. Se T possui apenas r, então o caminhamento em ordem de T é r. Caso contrário, sejam $T_1, T_2, ..., T_n$ as subárvores de r da esquerda para a direita. O caminhamento em ordem começa percorrendo em ordem T_1, em seguida visita r, e continua fazendo um caminhamento em ordem em $T_2, T_3, ..., T_n$.

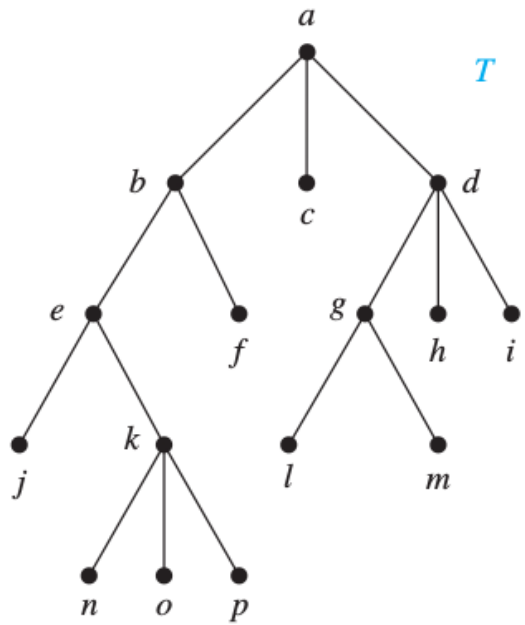
Caminhando em árvores

Encontre o caminhamento em ordem da árvore abaixo.



Caminhando em árvores

Encontre o caminhamento em ordem da árvore abaixo.



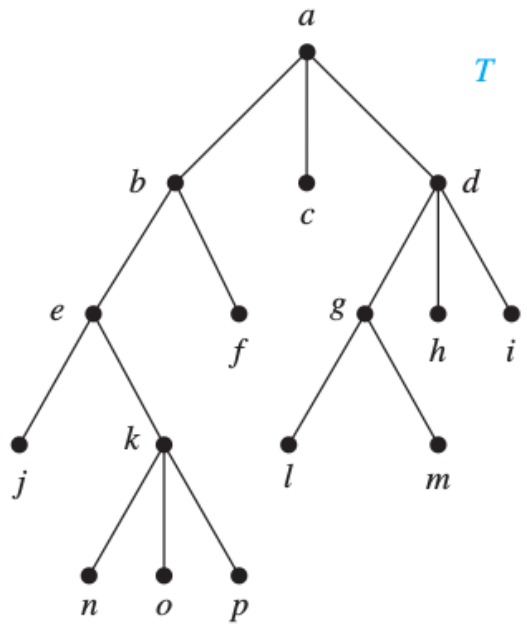
Resposta: $j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i$

Caminhando em árvores

Seja T uma árvore enraizada e ordenada com raiz r. Se T possui apenas r, então o caminhamento em pós-ordem de T é r. Caso contrário, sejam $T_1, T_2, ..., T_n$ as subárvores de r da esquerda para a direita. O caminhamento em pós-ordem começa percorrendo em pós-ordem $T_1, T_2, ..., T_n$, e finaliza visitando r.

Caminhando em árvores

Encontre o caminhamento pós-ordem da árvore abaixo.

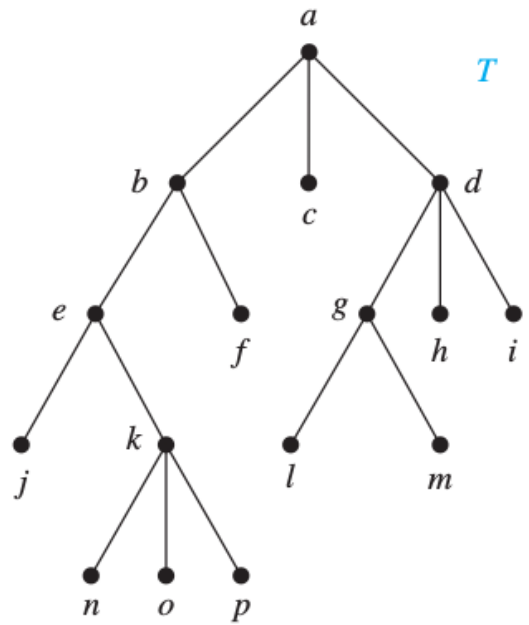


14

[14] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Caminhando em árvores

Encontre o caminhamento pós-ordem da árvore abaixo.



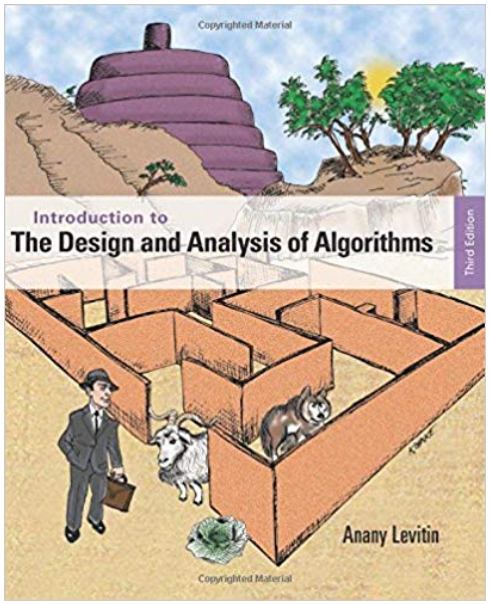
14

Resposta: $j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a$

[14] Fonte: K. Rosen. Discrete Mathematics and Its Applications. 2011.

Agenda

Bibliografia + leitura recomendada



Capítulo 1 (pp. 31–35)
Capítulo 4 (pp. 163–164)
Capítulo 5 (pp. 182–185)
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3a edição. Pearson. 2011.

Capítulo 5 (pp. 145–170)
Clifford Shaffer.
Data Structures and Algorithm Analysis.
Dover, 2013.

ÁRVORES BINÁRIAS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil