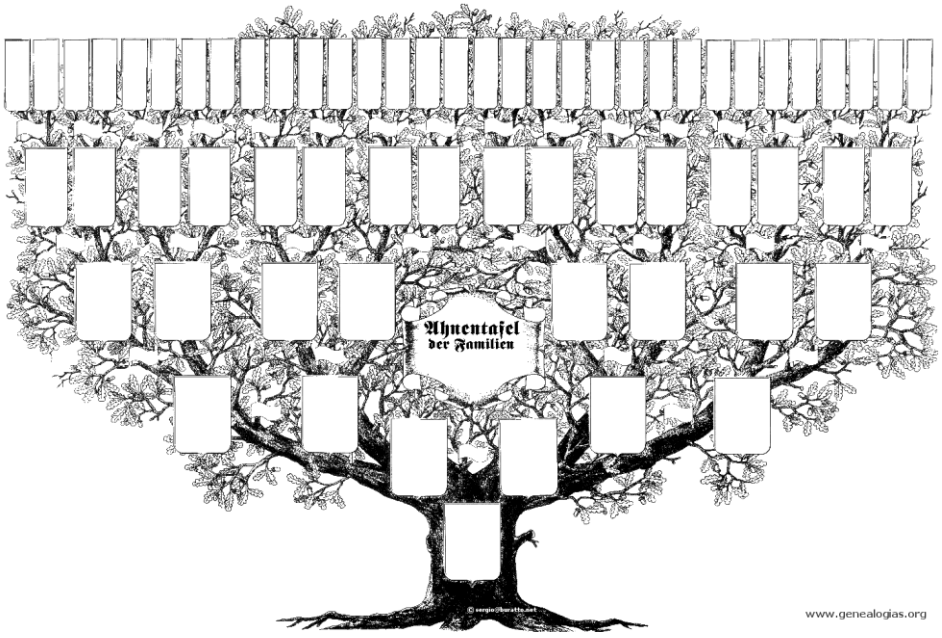
Capítulo

10 Estrutura Árvore



Árvore

As estruturas de dados Vetor, Lista, Fila e Pilha, são estruturas de dados unidimensionais ou lineares, e existem aplicações que necessitamos de outras estruturas mais complexas, como por exemplo, estruturas do tipo hierárquica, isto é, um determinado nó possui, um ou mais nós hierarquicamente inferiores. Podemos destacar vários exemplos de aplicações de estruturas hierárquicas bem como: as pastas, sub-pastas e documentos de um computador, índice remissivo de um determinado livro, a árvore filogenética da vida etc.

A figura 10.1, ilustra as pastas, sub-pastas e documentos de um diretório utilizando o sistema operacional Microsoft Windows Vista. Pode-se destacar, a árvore da raiz, pasta “Desktop”, e algumas sub-pastas, como por exemplo “Public”, e alguns exemplos de documentos, do tipo clip, na sub-pasta “Sample Media”, “Apollo 13”, “Jewels of Caribbean” e “Vertigo”.

.

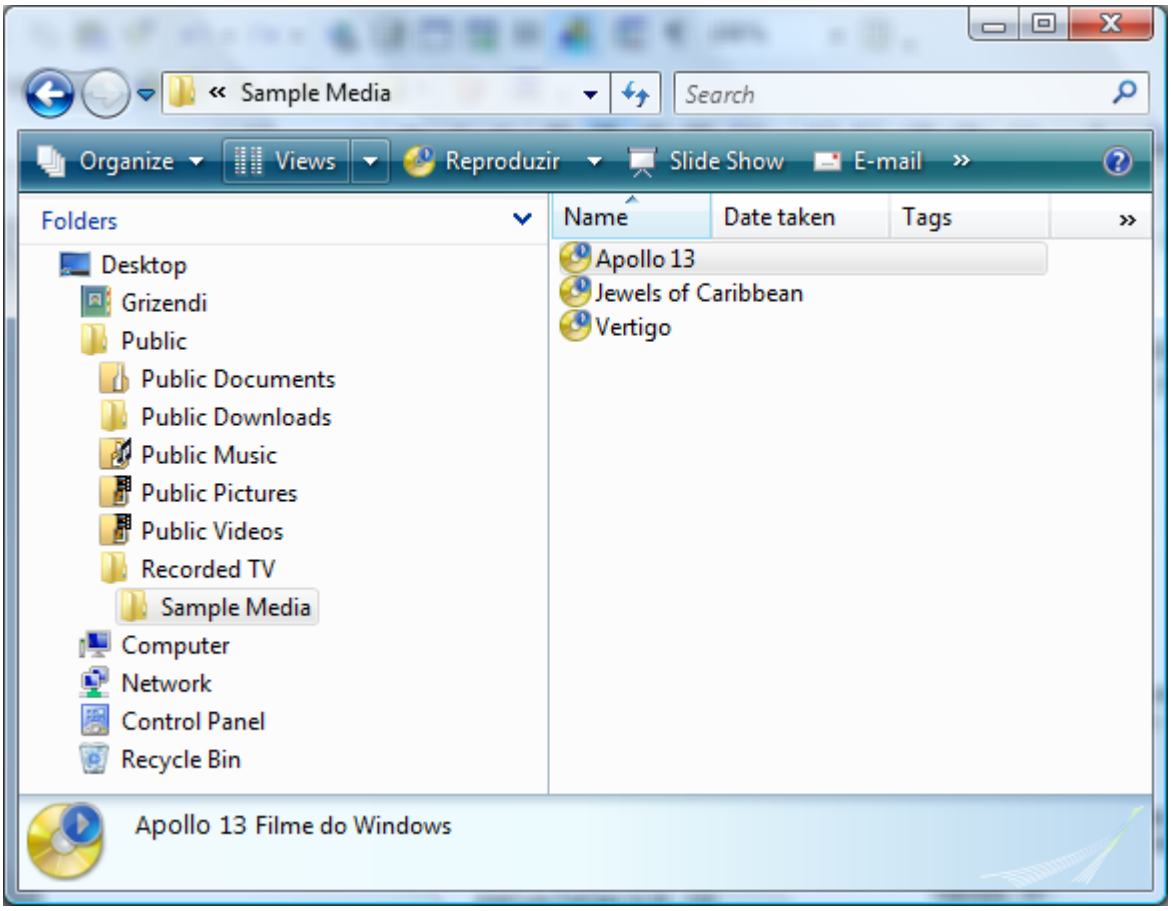


Figura 10.1 – Exemplo de pastas, sub-pastas e documentos de um diretório utilizando o sistema operacional Microsoft Windows Vista.

A figura 10.2, ilustra a árvore filogenética da vida.

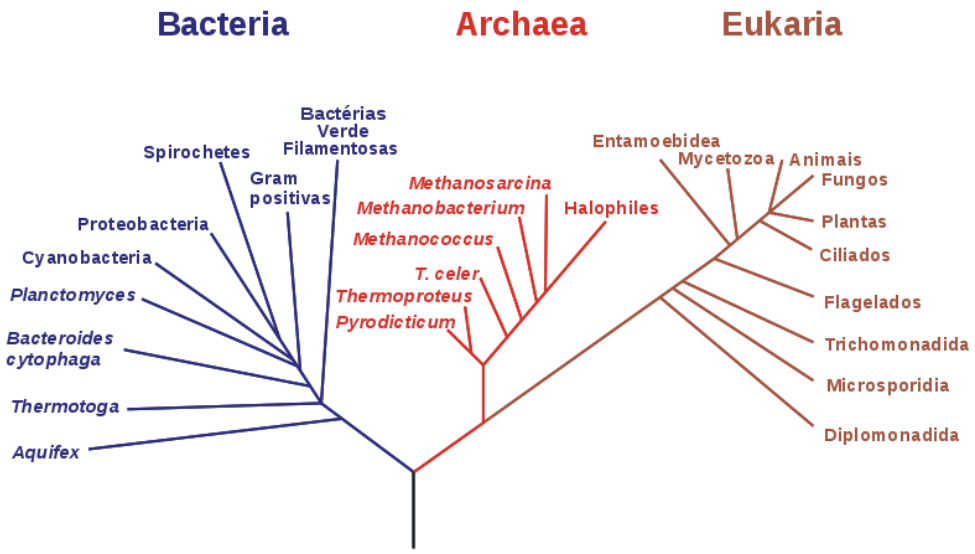


Figura 10.2 – Arvore filogenética da vida.

Existem diversas maneiras de representar árvores. Uma representação que reflete a idéia de árvores como conjuntos aninhados é mostrado na figura arvconj abaixo.

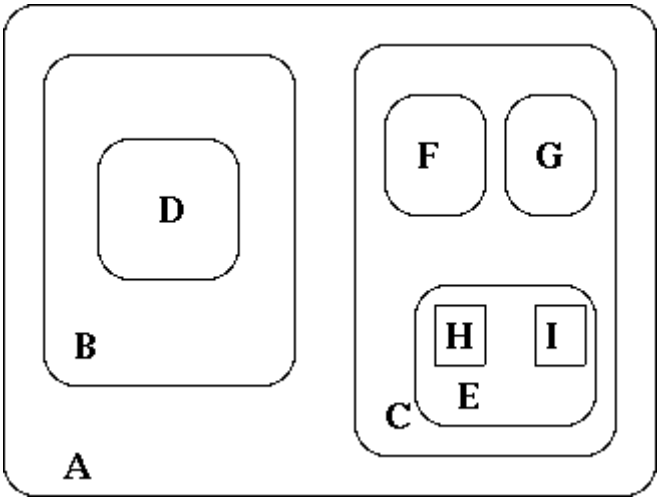


Figura 10.3 – Arvore representada como um conjunto aninhados.

Uma outra notação que encontramos a toda hora, e que está representada na figura 10.4, é a forma indentada ou de diagrama de barras. Nota-se que esta representação lembra um sumário de livro. Os sumários dos livros são representações da árvore do conteúdo do livro.

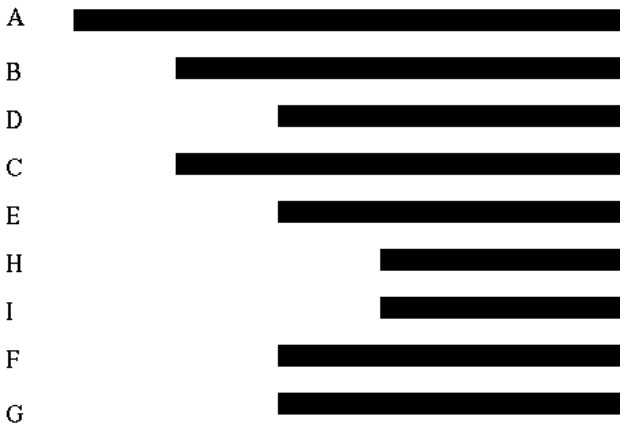


Figura 10.4 – Árvore e sua representação por barras

A seguir um exemplo de uma simples página wev escrita usando HTMH.

```
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=utf-8" />
<title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
<li>List item one</li>
<li>List item two</li>
</ul>
<h2><a href="https://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

A figura 10.5 ilustra a árvore correspondente onde os nós da árvore contêm as tags utilizadas para a criação da página.

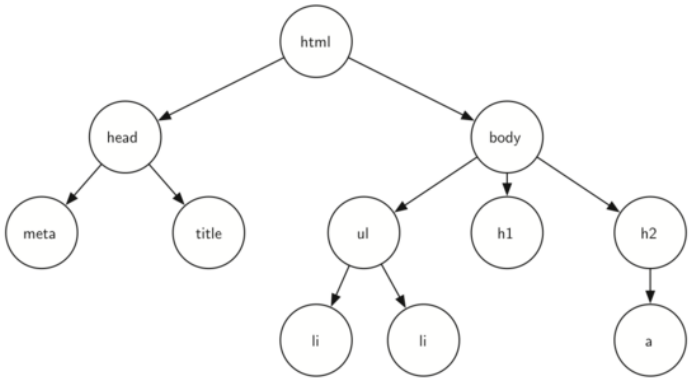


Figura 10.5 – Representação na forma de uma árvore com os tags de criação de uma página em HTML.

Árvore Binária (Binary Tree)

A árvore binária é uma estrutura hierárquica de dados, onde cada nó tem no máximo 2 filhos.

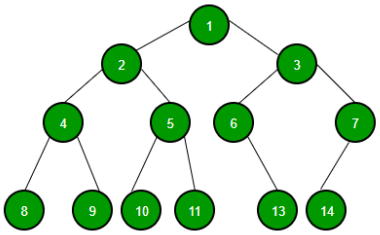


Figura 10.6 – Exemplo de uma árvore binária

Anatomia de uma Binary Tree

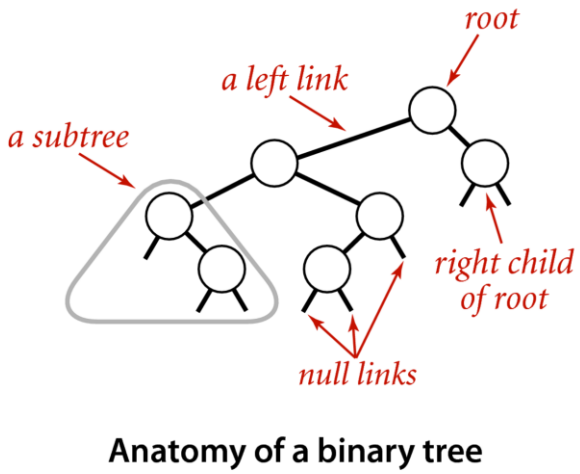


Figura 10.7 – Anatomia de uma árvore binária

- **Nó (node) :** são todos os itens guardados na árvore.
- **Raiz** (**root**) é o item do topo da árvore (neste caso o número 50).
- **Filho (child)** são os itens, hierarquicamente inferior, ligados à um mesmo nó, denominado de **Pai** (**5arente**).
- **Filho esquerdo (left child)** é o filho que está à esquerda de um nó pai
- **Filho direito (right child**) é o filho que está à direita de um nó pai
- **Folha** é um nó que não possui filho.
- **Sub árvore (subtree)** de um nó é uma árvore que tem como raiz um determinado filho

A classe BinaryTreeNode

A classe binaryTreeNode é a classe que implementa cada nó de uma árvore. O uso de nós e referências à esse s nós é que construirão a árvore propriamente dito.

O nó da árvore binária contém as seguintes informações:

1. element (element0 ou dado)
2. left (Ponteiro para o filho esquerdo)
3. right (Ponteiro para o filho direito)

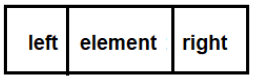


Figura 10.8 – Representação gráfica da classe Node

Usando os nós (Nodes) e as referências podemos construir uma árvore binária como mostrada na figura 10.9.

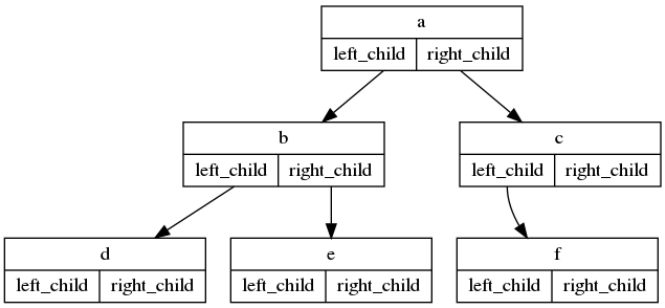


Figura 10.9 – Representação gráfica dos nós de uma árvore e suas referências

A figura 10.10 ilustra uma binary tree com os elementos (element) em cada nós e, as referências (ponteiros left e right). O nível de uma árvore, corresponde ao nível hierárquico da raiz até uma folha. A raiz é considera como nível 0, o nível 1, são os filhos do nó raiz e, assim seguidamente.

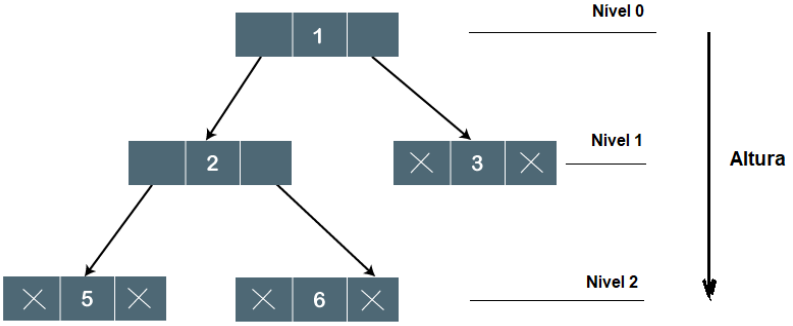


Figura 10.10 – Representação gráfica de uma árvore binária com os nível e a altura

O fragmento de código 10.1 ilustra a classe binaryTreeNode e o seu constructor.

```
class binaryTreeNode:
      def __init__(self,element):
         self.element = element
         self.left = None
         self.right = None

      def __str__(self):
           return str(self.element)
```

Fragmento de Código 10.1 – Implementação da Classe binaryTreeNode e o seu constructor.

A figura 10.11 ilustra as sub árvores do nó raiz de uma árvore. O nó raiz, elemento 50, tem as sub árvores cujos elementos são: 30 (sub árvore esquerda) e 70 (sub árvore direita).

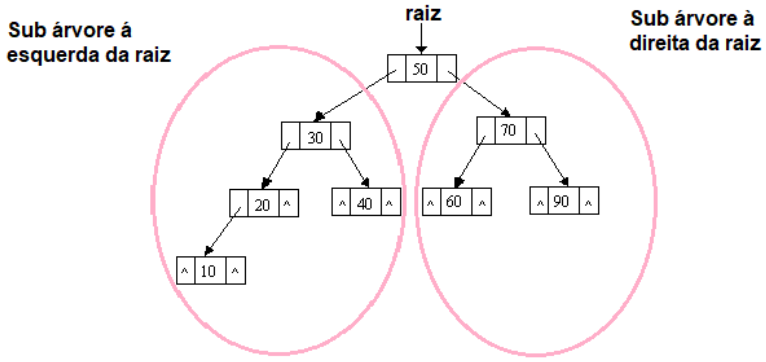
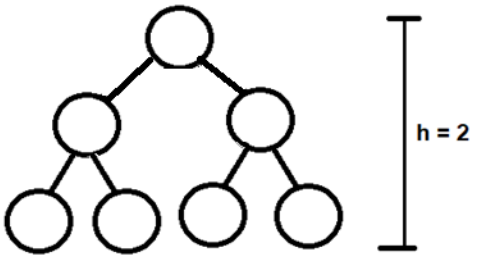


Figura 10.11 – Representação das sub árvores do nó raiz de uma árvore binária

Propiedades da Binary Tree

- Cada nível i, o número máximo de nodes é 2^i.
- A **altura de um nó** é o número de arestas no maior caminho desde o nó até um de seus descendentes.
- A **altura de árvore** é a altura do nó raiz
- O número máximo de nós de uma árvore de altura 2



N = 7 (número de nós)

Figura 10.12 (a) – Representação de uma árvore de altura 2 com o máximo de nós

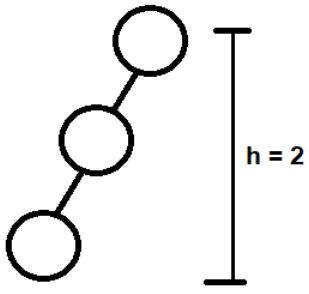
$$N = (1+2+4) = 7$$

- Genericamente o número máximo de nós de uma árvore de altura h é dada por:

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

A demonstração é feita pela soma de uma PG onde o primeiro termo vale 2^0 (1) e, o último termo 2^h

- O número mínimo de nós de uma árvore de altura h é igual a h + 1.



N = 3 (número de nós)

Figura 10.12 (b) – Representação de uma árvore de altura 2 com o mínimo de nós

- Se o número de nós for mínimo, a altura da árvore será máxima. Por outro lado, se o número de nós for máximo, a altura da árvore será mínima.

Seja N o número de nós de uma binary tree de altura h.

A menor altura pode ser calculada da forma:

O número N tem que ser menor ou igual ao número máximo de nós, logo:

$$N \leq 2^{h+1} - 1$$

Vamos isolar h na expressão anterior

$$N + 1 \leq 2^{h+1}$$

Aplicando logaritmo

$$log_2(N+1) \leq log_2(2^{h+1})$$
$$log_2(N+1) \leq h + 1$$

$$\rightarrow h \geq log_2(N+1) - 1$$

A maior altura pode ser calculada da forma:

O número N tem que ser maior ou igual ao número mínimo de nós, logo:

$$N \geq h + 1$$
$$\rightarrow h \leq N - 1$$

Logo, podemos concluir que:

$$log_2(N+1) - 1 \leq h \leq N - 1$$

Tipos de Binary Tree

- **Full (strict) Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

Full/ proper/ strict Binary tree

A full binary tree também conhecida como strict binary tree é uma árvore binária em que um pai tem 0 ou 2 filhos

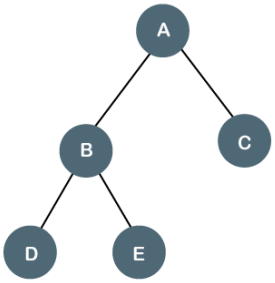


Figura 10.13 – Representação gráfica de uma árvore strict Binary Tree

Complete Binary Tree

A complete binary tree é uma árvore binária no qual todos os nós possuem dois filhos. Também são consideradas complete binary tree aquelas em que o ltimo nível as folhas estão ajustadas o mais à esquerda possível.

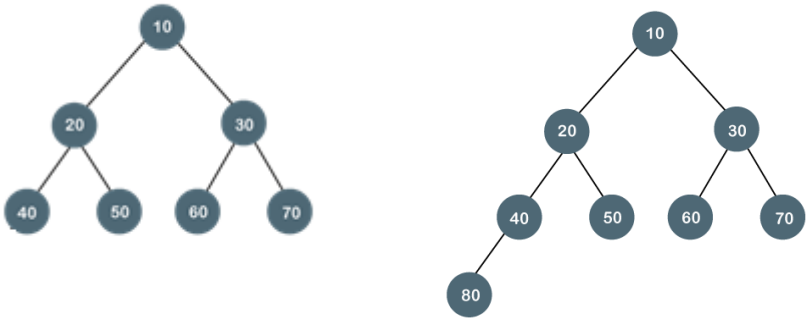


Figura 10.13 – Representações gráficas de uma árvore complete Binary Tree

Perfect Binary Tree

Um árvpre é denominada de perfect binary tree se todos os nós internos (que têm filhos) possuem sempre dois filhos em todos os níveis.

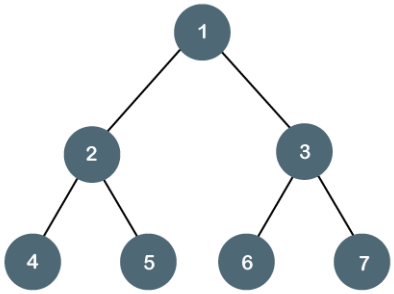


Figura 10.14 – Representação gráfica de uma árvore perfect Binary Tree

Degenerate Binary Tree

A árvore conhecida como degenerate binary tree é uma árvore no qual todos os nós internos possuem somente um filho.

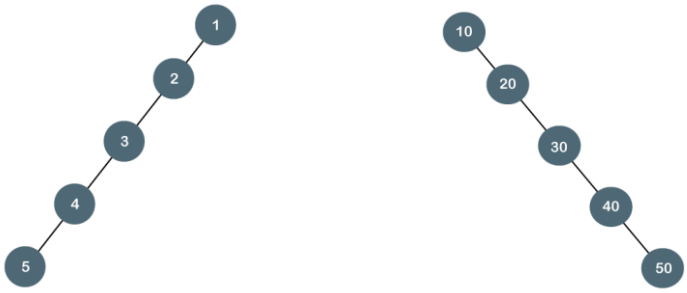


Figura 10.15 – Representações gráficas de uma árvore degenerate Binary Tree

Balanced Binary Tree

A balanced binary tree é uma árvore em que as sub árvores de qualquer nó tem alturas que diferem no máximo em uma unidade. Como exemplos de balanced binary tree temos a ***AVL*** e ***Red-Black trees***.

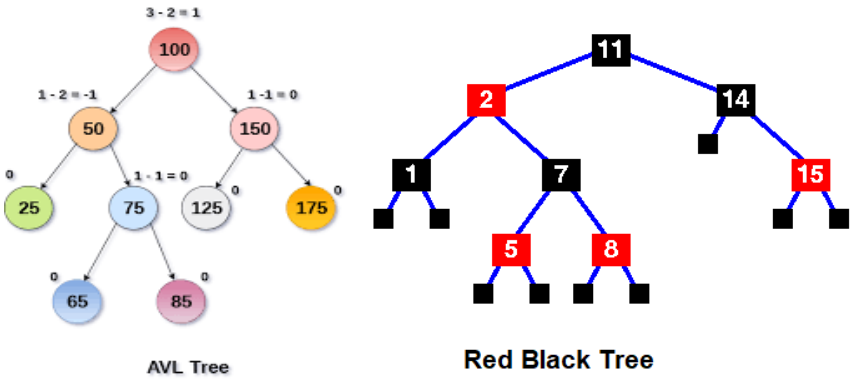


Figura 10.16 – Representações gráficas de uma árvore balanced Binary Tree

Implementação da classe Binary Tree

A binary tree é implementada com o auxílio de ponteiros. Cada célula da árvore é um nó (Node) que possui os três elementos básicos para a implementação: os ponteiros left, que aponta para o nó correspondente ao filho esquerdo e right, que aponta para o nó correspondente ao filho esquerdo e, o campo element, que guarda o elemento a ser inserido.

O primeiro nó da árvore é representado pelo ponteiro root (raiz). Primeiramente, ao criarmos uma binary tree, precisamos inicializar a variável root com o valor None.

O fragmento de código 10.2 ilustra a classe binaryTree e o seu constructor.

```
from BinaryTreeNode import binaryTreeNode
class binaryTree:
    def __init__(self):
        self.root = None
```

Fragmento de Código 10.2 – Implementação da Classe binaryTree e o seu constructor.

Inserção à esquerda

Inserção da raiz

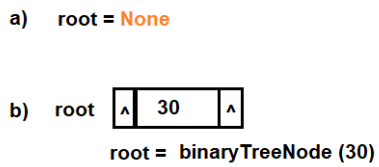


Figura 10.17 – Representação gráfica da inserção da raiz em uma árvore binária vazia (a) antes da inserção (b) depois da inserção

Inserção à esquerda da raiz

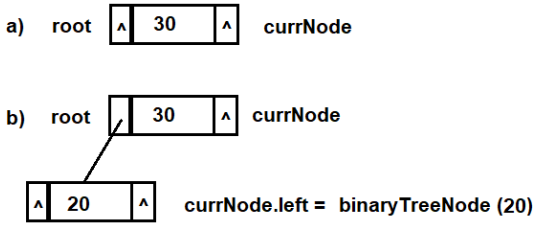


Figura 10.18 – Representação gráfica da inserção à esquerda da raiz em uma árvore binária (a) antes da inserção (b) depois da inserção

O fragmento de código 10.3 ilustra o método insertLeft em uma binaryTree.

```
def insertLeft(self,currNode,element):
    if currNode == None:
        self.root = binaryTreeNode(element)
    else:
        currNode.left = binaryTreeNode(element)
```

Fragmento de Código 10.3 – Implementação do método insertLeft da classe binaryTree.

Inserção à direita

Inserção à direita da raiz

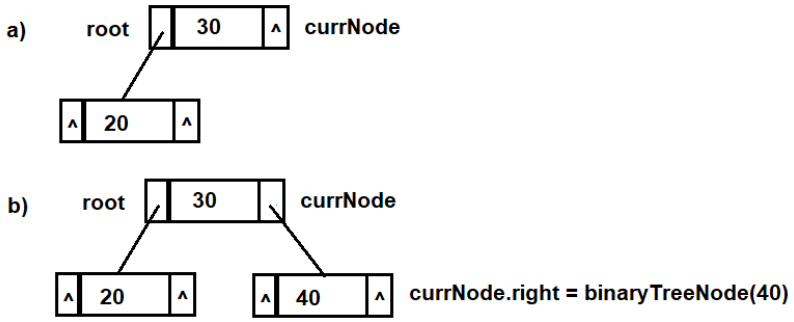


Figura 10.19 – Representação gráfica da inserção à direita da raiz em uma árvore binária (a) antes da inserção (b) depois da inserção

O fragmento de código 10.4 ilustra o método insertLeft em uma binaryTree.

```
def insertRight(self,currNode,element):
    if currNode==None:
        self.root = binaryTreeNode(element)
    else:
        currNode.right = binaryTreeNode(element)
```

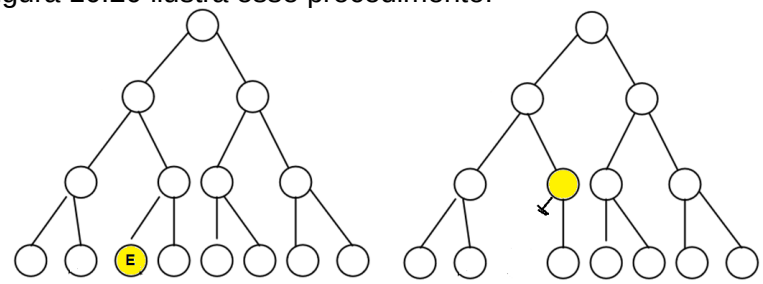
Fragmento de Código 10.4 – Implementação do método insertRight da classe binaryTree.

Remoção de nós

Existem três casos distintos a serem tratados: nó a ser removido tem zero, um ou dois filhos: o nodo a ser removido não possui filhos, o nodo a ser removido possui um e somente um filho e, o nodo a ser removido possui dois filhos.

Caso 1

O nodo a ser removido não possui filhos, isto é, é um nodo folha. Neste caso simplesmente removemos o nodo e atualizamos o ponteiro correspondente de seu pai. A figura 10.20 ilustra esse procedimento.

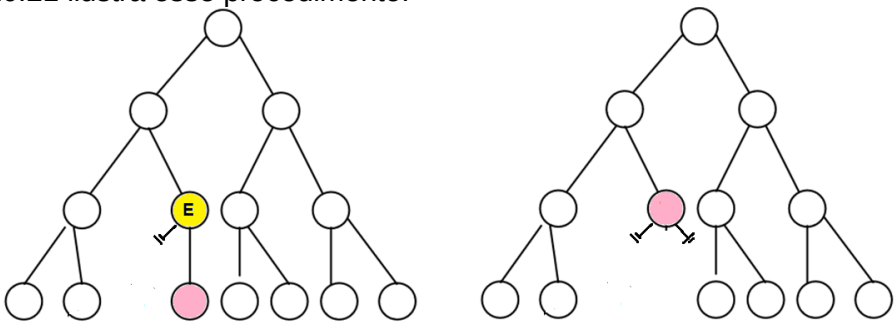


(a) (b)

Figura 10.20 – Remoção do nó folha em uma árvore binária. (a) antes da remoção (b) após a remoção

Caso 2

O nodo a ser removido possui um e somente um filho. Neste caso simplesmente substituímos o campo element desse nó pelo campo element do seu único filho. A figura 10.21 ilustra esse procedimento.

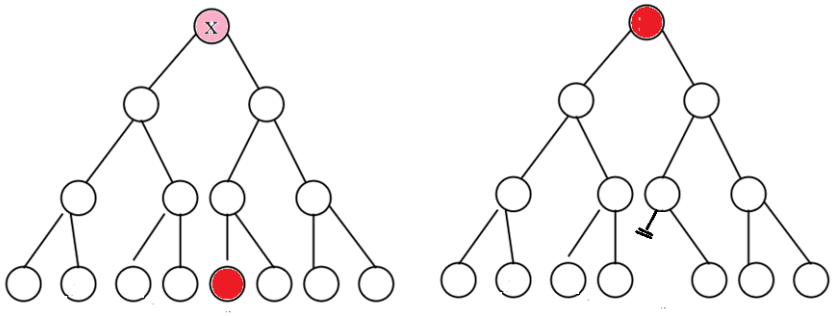


(a) (b)

Figura 10.321– Remoção do nó com um filho em uma árvore binária. (a) antes da remoção (b) após a remoção

Caso 3

O nodo a ser removido possui dois filhos. Neste caso simplesmente substituímos o campo element desse nó pelo campo element do seu nodo sucessor e depois, e depois, eliminamos o nodo sucessor.. A figura 10.22 ilustra esse procedimento.



(a) (b)

Figura 10.22– Remoção do nó com dois filhos em uma árvore binária. (a) antes da remoção (b) após a remoção

O fragmento de código 10.5 ilustra o métodos remove e, o método auxiliar _delete para remoção um elemento em uma binaryTree.

```
 def remove(self,element):
     self.root=self._delete(self.root,element)

def _delete(self, root, element):
    # Step 1 - Perform normal Binary Tree
    if not root:
        return root
    elif element != root.element:
        root.left = self._delete(root.left, element)
        root.right = self._delete(root.right, element)
    else:
        # cases 1 or 2 : node is leaf or has one child
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        # caso 3 - node with two children
        temp = self._getMinValueNode(root.right)
        root.element = temp.element
        root.right = self._delete(root.right,
                                  temp.element)
    return root
```

Fragmento de Código 10.5 – Implementação dos métodos remove e o seu método auxiliar _delete de um elemento da classe binaryTree.

Atravessamento na Árvore

Dada uma árvore qualquer podemos atravessar esta árvore, isto é, visitar os seus nós utilizando-se de um procedimento fixo qualquer de visita aos nós e os próximos. Os algoritmos de atravessamentos que veremos são : Em-Ordem, Pós-Ordem e finalmente Pré Ordem.

Considere a árvore da figura 10.20 com apenas três elementos.

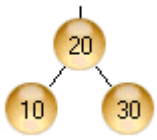


Figura 10.20 – Representação gráfica de uma árvore com 3 elementos

Em-Ordem

O atravessamento Em-Ordem ou atravessamento IN_fixado, consiste em se visitar os nós da seguinte forma :

- Imprimir o FE
- Imprimir a Raiz
- Imprimir o FD

Logo na figura anterior a ordem da árvore utilizando-se da técnica Em-Ordem seria : 10, 20 e 30

Pré-Ordem

O atravessamento Pré-Ordem ou atravessamento Pré-fixado, consiste em se visitar os nós da seguinte forma :

- Imprimir a Raiz
- Imprimir o FE
- Imprimir o FD

Logo na figura anterior a ordem da árvore utilizando-se da técnica Pré-Ordem seria : 20, 10 e 30

Pós-Ordem

O atravessamento Pós-Ordem ou atravessamento Pós-fixado, consiste em se visitar os nós da seguinte forma :

- Imprimir o FE
- Imprimir o FD
- Imprimir a Raiz

Logo na figura anterior a ordem da árvore utilizando-se da técnica Pós-Ordem seria : 10, 30 e 20

Considere agora, a figura 10.21 correspondente a uma árvore binária qualquer. Qual seria o procedimento para o atravessamento Em-Ordem, Pós-Ordem e Pré-Ordem ?

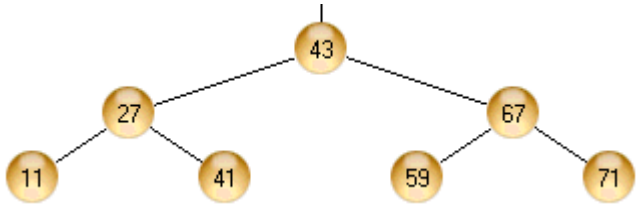


Figura 10.21 – Representação gráfica de uma árvore com 7 elementos

Em-Ordem

O atravessamento Em-Ordem, quando na árvore a raiz possui sub Árvores, consiste em se visitar os nós da seguinte forma :

- Aplicar o procedimento Em-Ordem na Sub-Árvore à Esquerda do nó raiz
- Imprimir a Raiz
- Aplicar o procedimento Em-Ordem na Sub-Árvore à Direita do nó raiz

Logo na figura anterior a ordem da árvore utilizando-se da técnica Em-Ordem seria :

((11)27(41)) 43 ((59) 67(71))

Pós-Ordem

O atravessamento Pós-Ordem, quando na árvore a raiz possui sub Árvores, consiste em se visitar os nós da seguinte forma :

- Aplicar o procedimento Pós-Ordem na Sub-Árvore à Esquerda do nó raiz
- Aplicar o procedimento Pós-Ordem na Sub-Árvore à Direita do nó raiz
- Imprimir a Raiz

Logo na figura anterior a ordem da árvore utilizando-se da técnica Pós-Ordem seria :

((1)(41)27) ((59)(71)67) 43

Pré-Ordem

O atravessamento Pré-Ordem, quando na árvore a raiz possui sub Árvores, consiste em se visitar os nós da seguinte forma :

- Imprimir a Raiz
- Aplicar o procedimento Pré-Ordem na Sub-Árvore à Esquerda do nó raiz
- Aplicar o procedimento Pré-Ordem na Sub-Árvore à Direita do nó raiz

Logo na figura anterior a ordem da árvore utilizando-se da técnica Pré-Ordem seria :

43(27(11)(41))(67(59)(71))

O fragmento de código 10.5 ilustra as implementações dos métodos preOrder, inOrder e posOrder uma binaryTree.

```
def preOrder(self):
    tr = []
    def bodyPreOrder(root):
        if root:
            tr.append(root.element)
            bodyPreOrder(root.left)
            bodyPreOrder(root.right)
        return tr

    return bodyPreOrder(self.root)


def inOrder(self):
    tr = []
    def bodyInOrder(root):
        if root:
            bodyInOrder(root.left)
            tr.append(root.element)
            bodyInOrder(root.right)
        return tr

    return bodyInOrder(self.root)


def posOrder(self):
    tr = []
    def bodyPosOrder(root):
        if root:
            bodyPosOrder(root.left)
            bodyPosOrder(root.right)
            tr.append(root.element)
        return tr

    return bodyPosOrder(self.root)
```

Fragmento de Código 10.5 – Implementação dos métodos preOrder, inOrder e posOrder

Parse Tree

As arvores do tipo Parse tree são árvores usadas geralmente para representar exemplos reais de construção de sentenças ou expressões matemáticas. A figura 10.22 ilustra uma Parse Tree genérica de uma sentença.

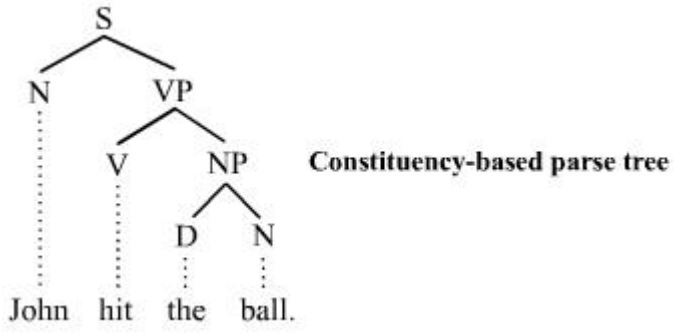


Figura 10.22 – Uma Parse Tree para uma simples sentença

A figura 10.23 ilustra uma Parse Tree genérica de uma expressão matemática.

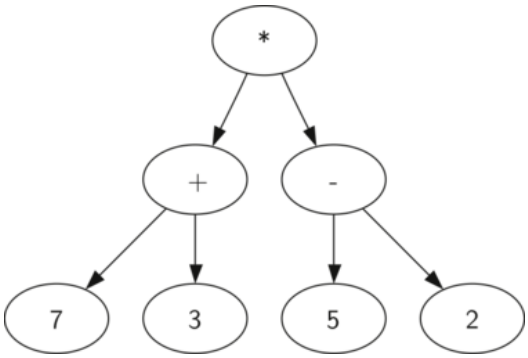


Figura 10.23 – Uma Parse Tree para ((7+3)∗(5−2))((7+3)∗(5−2))

Regras da construção de uma Parse Tree de expressão matemática:

1. Se o corrente token é um `"("`, adicione um novo nó como filho esquerdo ao nó corrente, e então visite o filho esquerdo.
2. Se o corrente token está na lista `["+", "-", "/", "*"]`, armazene esse símbolo no campo elemento da raiz do nó corrente. Adicione um novo nó como filho direito da raiz do nó corrente e visite esse filho direito.
3. Se o corrente token é um número, armazene esse valor no campo elemento da raiz e retorne ao pai do nó corrente.
4. Se o corrente token é um `")"`, vá para o nó pai do nó corrente.

Exemplo

Faça um trace na implementação da classe ParseTree e construa a árvore seguindo as regras de construção para a expressão;

$$(3+(4*5))(3+(4*5))$$

A expressão será uma lista de caracteres com os tokens e os números como a lista a seguir:

```
["(", "3", "+", "(", "4", "*", "5", ")", ")"]
```

A sequência de inserção é mostrada na figura 10.24.

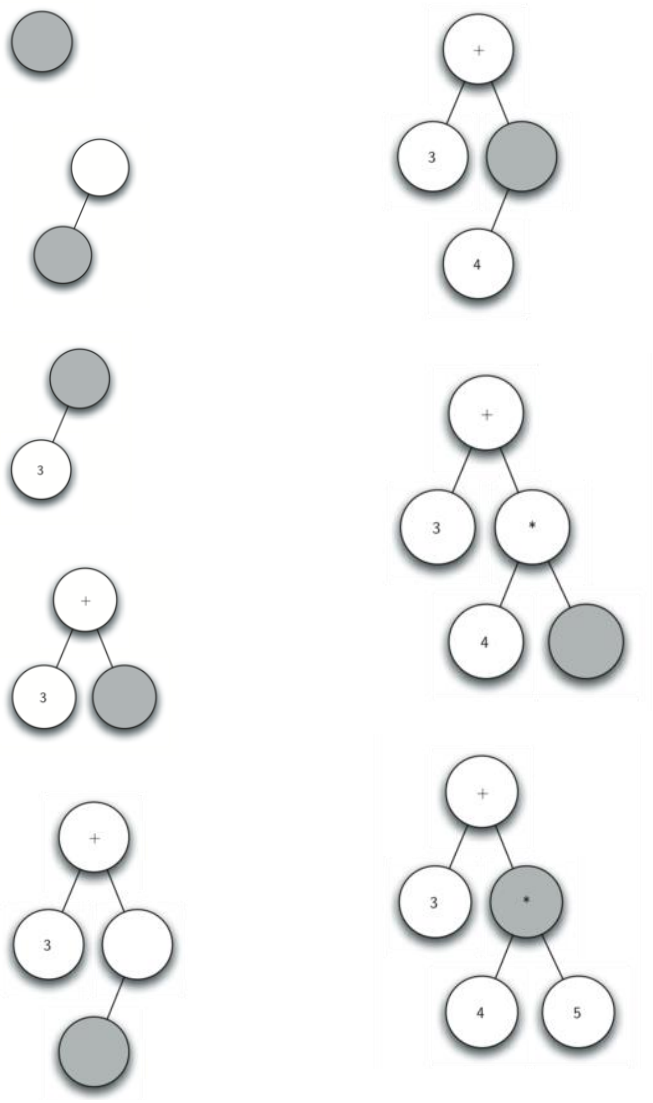


Figura 10.24 – Passo a passo a construção de uma Parse Tree.

Árvore Binária de Pesquisa - Binary Search Tree (BST)

Em ciência da computação, a árvore de busca binária ou árvore de pesquisa binária é uma árvore binária onde todos os nós são valores, todo nó a esquerda contêm uma sub-árvore com os valores menores ao nó raiz da sub árvore e todos os nós da sub árvore a direita contêm somente valores maiores ao nó raiz. (Esta é a forma padrão, podendo ser invertida as sub árvores dependendo da aplicação). Os valores são relevantes na árvore de busca binária. O objetivo desta árvore é estruturar os dados de forma flexível permitindo pesquisa binária

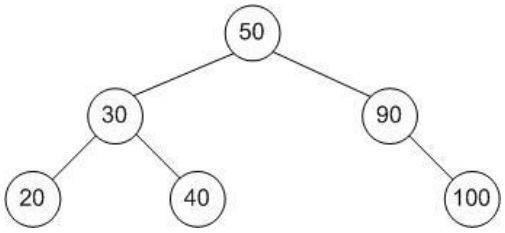


Figura 10.25 – Exemplo de uma árvore binária de pesquisa (BST)

A figura 10.26 ilustra a anatomia de uma uma Binary Search Tree (BST).

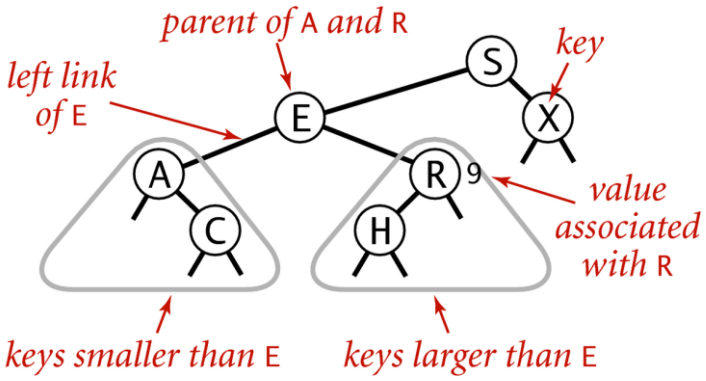


Figura 10.26 – Anatomia de uma árvore binária de pesquisa (BST)

A BST é um caso particular do TAD árvore, onde os registros são armazenados em células denominadas de ***nós***. Os ***nós*** são estruturados hierarquicamente. Cada ***nó*** tem um ponteiro para o ***nó*** inferior à esquerda e o ***nó*** inferior à direita. O primeiro ***nó*** desta estrutura é denominado e ***raiz***, e os demais de ***filhos***. Cada nó tem no máximo 2 filhos, por isso esta estrutura é denominada Binária. A diferença básica é que o armazenamento é baseado em comparação com a chave. Se a chave for menor que a chave do nó corrente, visite o filho esquerdo, caso contrário, visite o filho direito. A figura 10.27 ilustra este conceito.

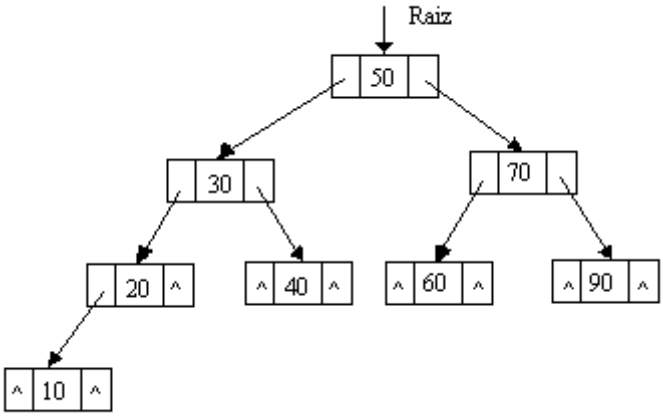


Figura 10.27 - Exemplo de uma Estrutura Árvore Binária de Pesquisa (BST).

A Figura 10.28 ilustra exemplos de árvores não balanceadas. No caso da Figura 10.28(a), temos *h* = 3 e *N* = 4, mas pela relação anterior se $N = 4 \rightarrow \boldsymbol{h \geq 1,32}$.. logo, a árvore, deveria ter altura igual a 2 no máximo, mas observe que a altura é igual a 3.

No segundo caso da figura 10.28(b), temos *h* = 3 e *N* = 5, mas pela relação anterior se $N = 5 \rightarrow \boldsymbol{h \geq 1,58}$.. , logo, a árvore, deveria ter altura igual a 2 no máximo, mas observe que a altura é igual a 3.

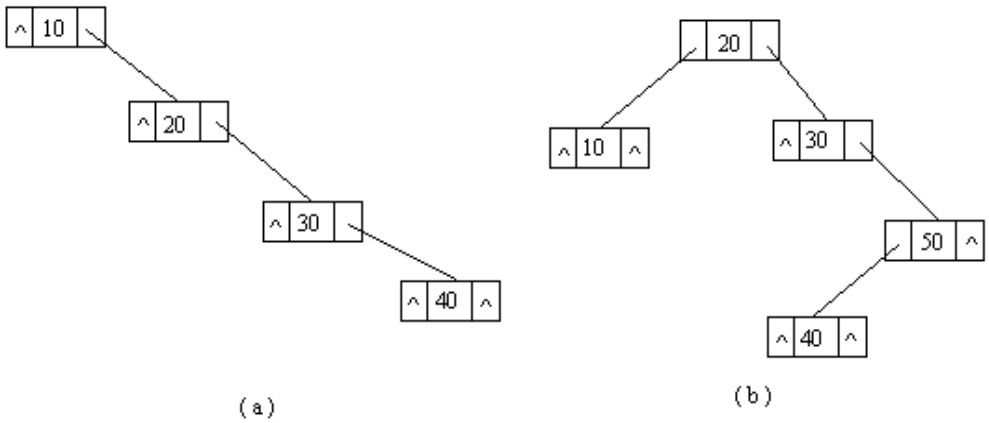
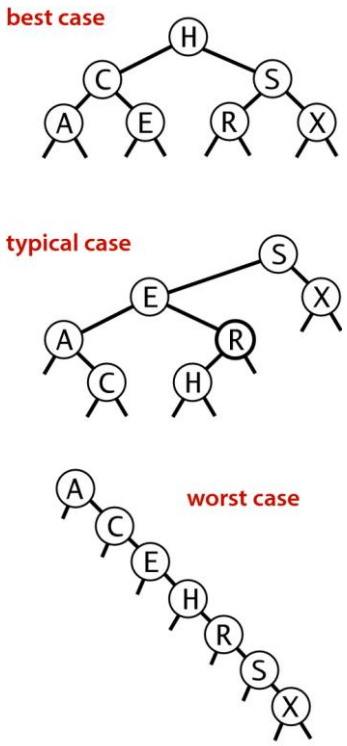


Figura 10.28 - Exemplos de árvores desbalanceadas.

A Figura 10.29 ilustra o melhor, o típico e o pior caso de balanceamento em um BST.



BST possibilities

Figura 10.29 – BST possibilidades

O par key Value

Todo BST é baseada no par (key, value), onde key é a chave de pesquisa e, vale, é o valor armazenado. A figura 10.30 ilustra a ideia da representação do par (key,value).

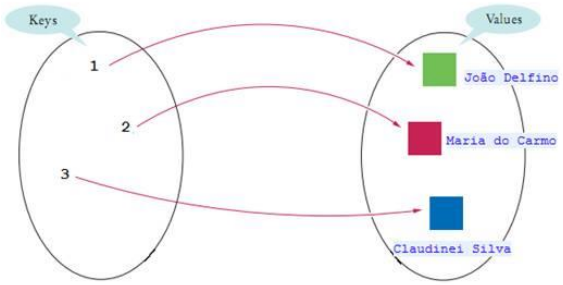


Figura 10.30 – Representação do par (key, value)

O campo value, pode conter um tipo simples de dados, um objeto ou uma referência. Por exemplo, nas tabelas de índices, o campo value, contém as referências à área de dados. A figura 10.31 ilustra esse conceito.

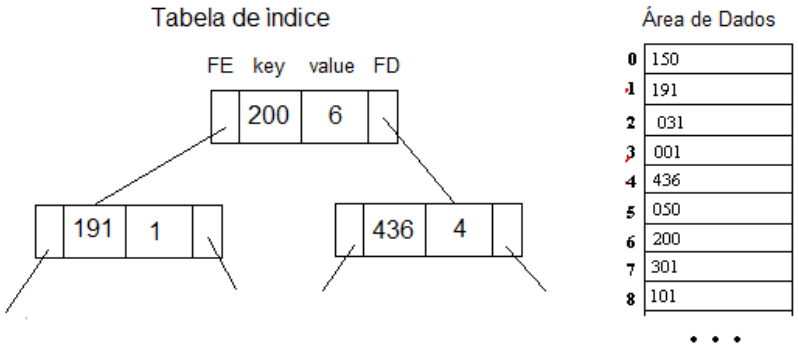


Figura 10.31 – Representação esquemática de uma estrutura de índice na forma de uma BST

Pesquisando em uma BST

A figura 10.32 ilustra, a pesquisa da chave R, como um sucesso, em uma BST.

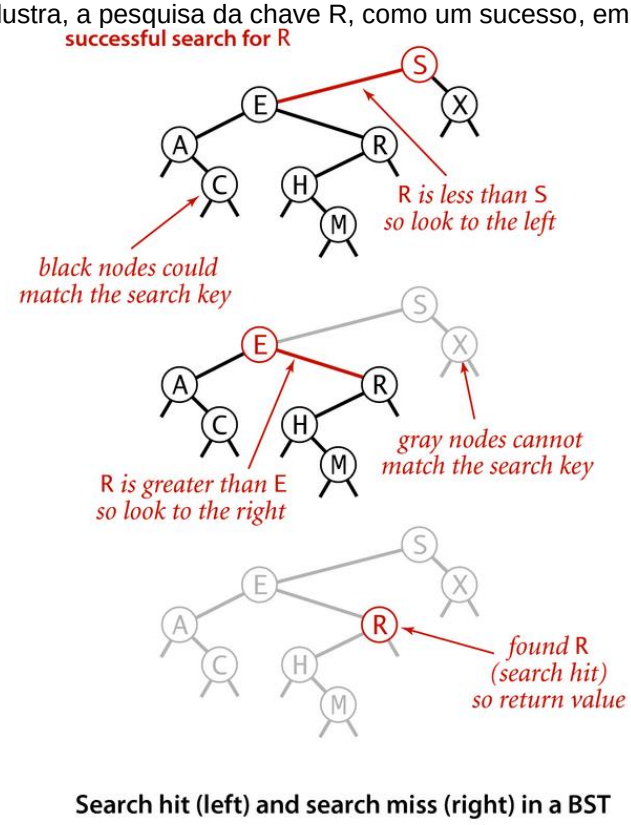


Figura 10.32 – Pesquisa da chave R em um BST

A figura 10.33 ilustra, a pesquisa da chave T, como um insucesso em uma BST.

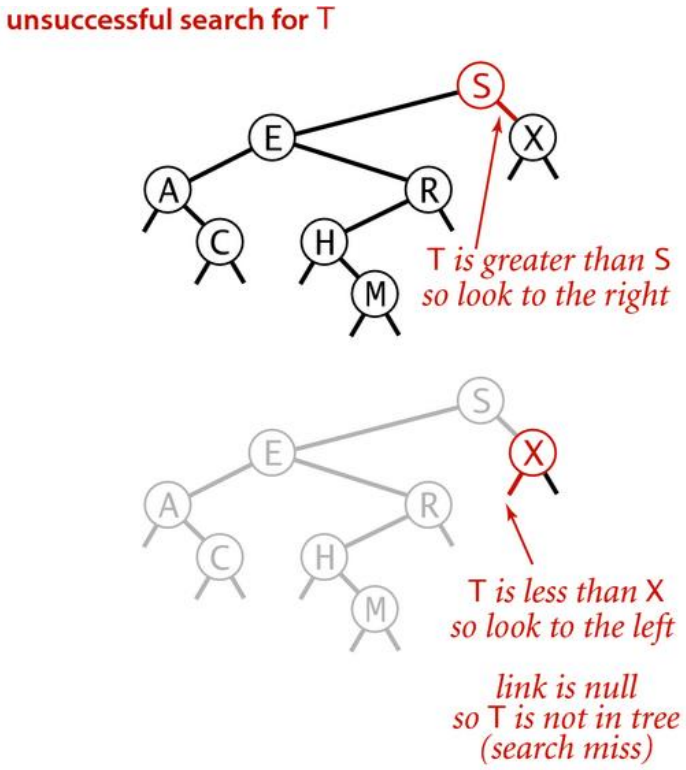


Figura 10.33 – Pesquisa da chave T em um BST

Inserir um elemento e uma BST

A figura 10.34 ilustra a inserção da chave L na BST.

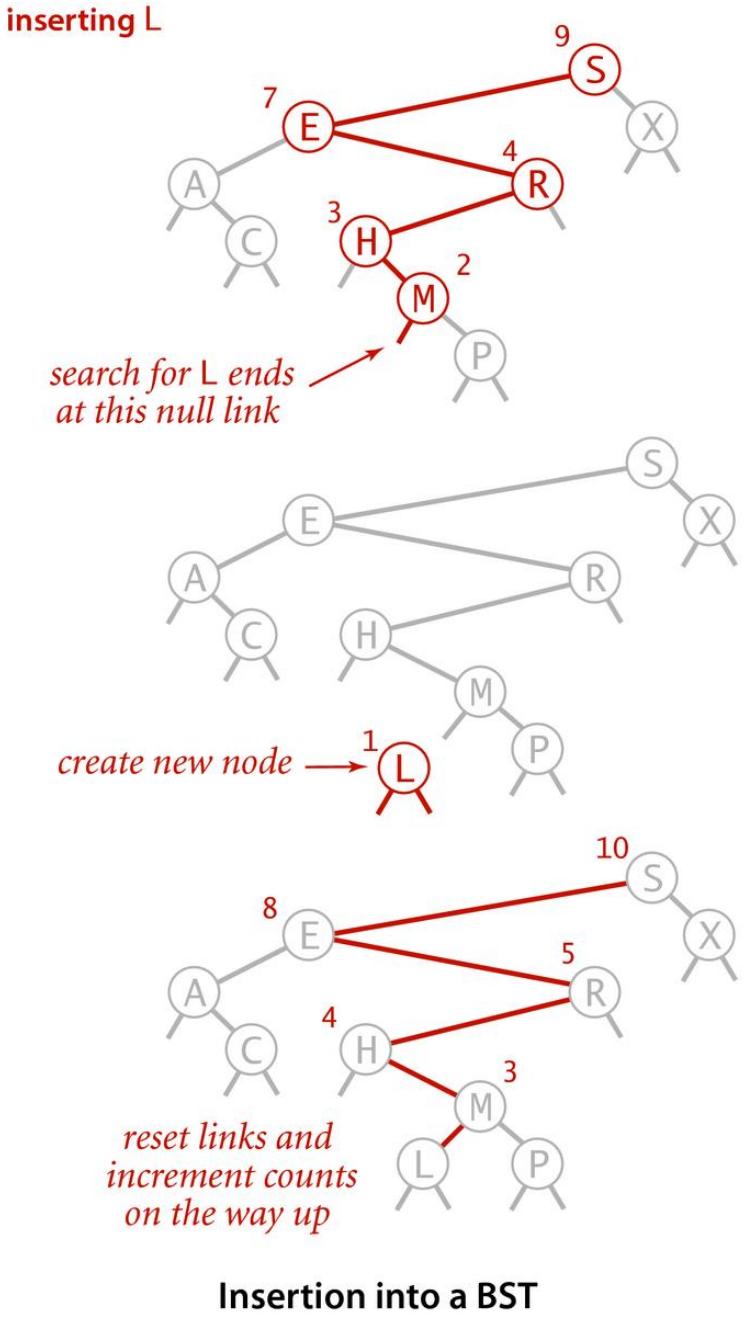


Figura 10.34 – Inserção da chave L em um BST

Link do applet de BST

https://www.cs.usfca.edu/~galles/visualization/BST.html

Criação de uma Árvore Binária de Pesquisa

Criar uma árvore de pesquisa com as chaves 10, 20, 30, 25, 33, 11 e 60.

A primeira célula será sempre a raiz. Logo a raiz será a célula de chave 10. A figura 10.35 (a) ilustra este procedimento. A segunda chave a ser inserida é a chave 20, maior que a chave da raiz, então insere como filho direito da raiz. . A figura 10.35 (b) ilustra este procedimento. A terceira chave a ser inserida é a chave 30. Pode-se observar na figura 10.35(c) que a chave 30 deve ser inserida como filho direito da chave 20.

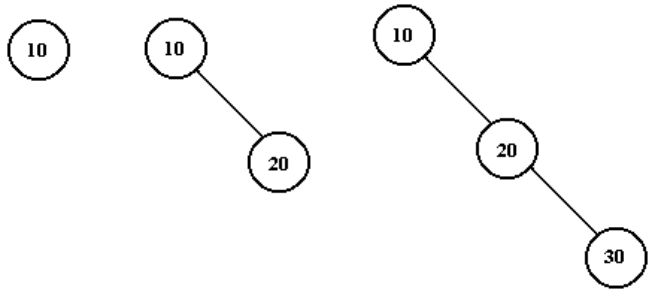


Figura 10.35 – Inserção das chaves 10, 20 e 30.

A próxima chave é a chave 25 que na hierarquia da árvore anterior deverá ser inserida como filho esquerdo da chave 25. A figura 10.35 (d) ilustra a inserção da chave 25. A próxima chave a ser inserida é a chave 33, que na hierarquia da árvore anterior deverá ser inserida como filho direito da chave 30. A figura 10.35(e) ilustra a inserção da chave 33.

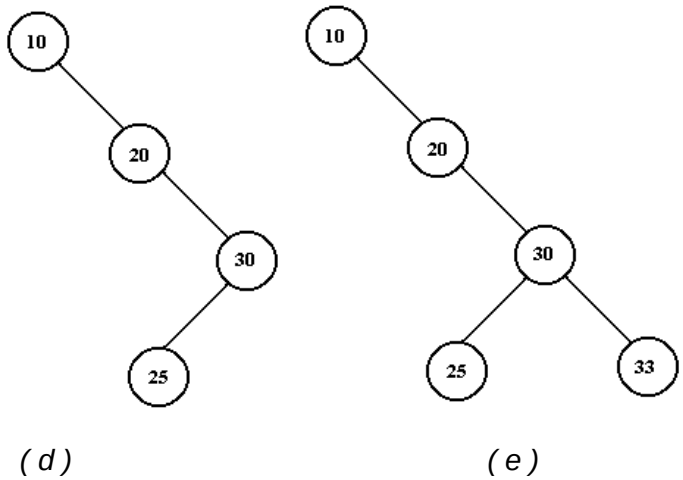
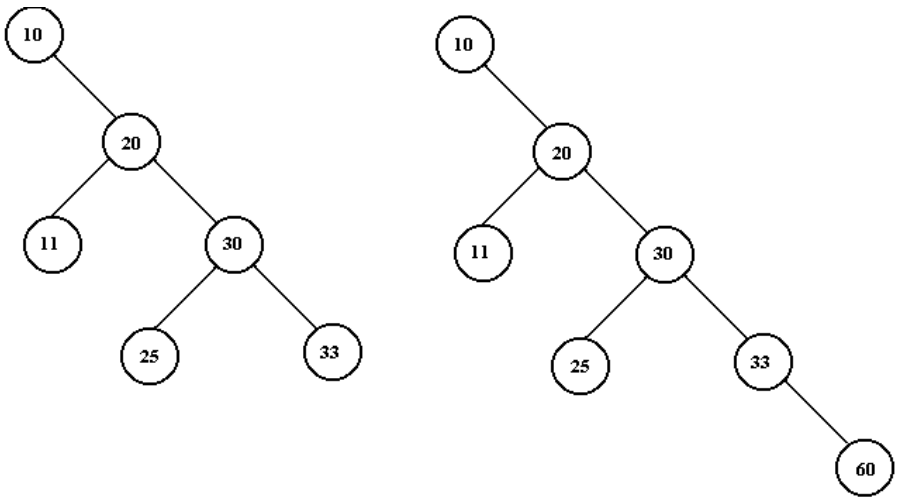


Figura 10.35 – Inserção das chaves 25 e 33 .

A próxima chave é a chave 11 que na hierarquia da árvore anterior deverá ser inserida como filho esquerdo da raiz, chave 10. A figura 10.35(f) ilustra a inserção da chave 11. A última chave a ser inserida é a chave 60, que na hierarquia da árvore anterior deverá ser inserida como filho direito da chave 33. A figura 10.35(g) ilustra a inserção da chave 60.



(f) (g)

Figura 10.35 –Inserção das chaves 11 e 60 .

Sucessor e Predecessor (antecessor)

Seja uma BST T, qualquer, com chaves distintas. Denominamos de nó sucessor a um nó x, a:

- O sucessor de um nodo x é um outro nodo y, tal que a chave[y] é o menor valor maior que a chave[x]
- O antecessor de um nodo x é um outro nodo y, tal que a chave[y] é o maior valor maior que a chave[x]

A figura 10.36 ilustra o nó sucessor de de um nó X. Nesse exemplo , a chave do nó X é 10 , e a chave, de seu socessor, nó Y, é 11.

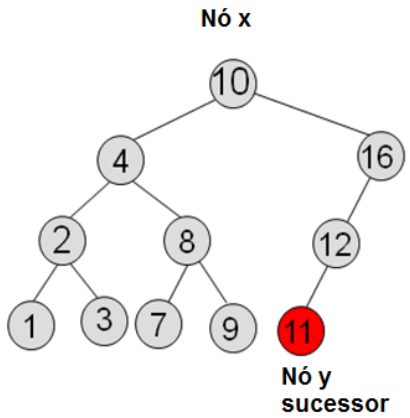
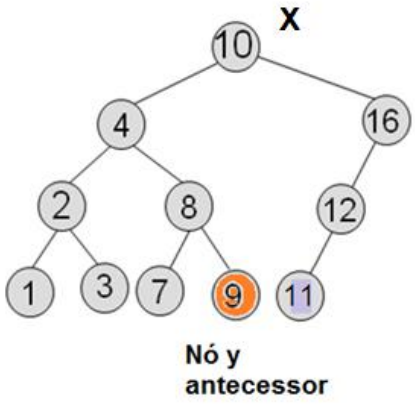


Figura 10.36 – Ilustração do nó sucessor do nó X

A figura 10.37 ilustra o nó antecessor de de um nó X. Nesse exemplo, a chave do nó X é 10 , e a chave, de seu antecessor, nó Y, é 9.



.

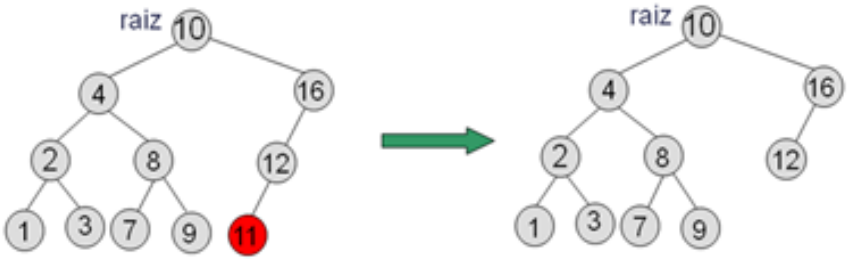
Figura 10.37– Ilustração do nó antecessor do nó X

Remoção de nós

Existem três casos distintos a serem tratados: nó a ser removido tem zero, um ou dois filhos: o nodo a ser removido não possui filhos, o nodo a ser removido possui um e somente um filho e, o nodo a ser removido possui dois filhos.

Caso 1

O nodo a ser removido não possui filhos, isto é, é um nodo folha. Neste caso simplesmente removemos o nodo e atualizamos o ponteiro correspondente de seu pai. A figura 10.38 ilustra esse procedimento.

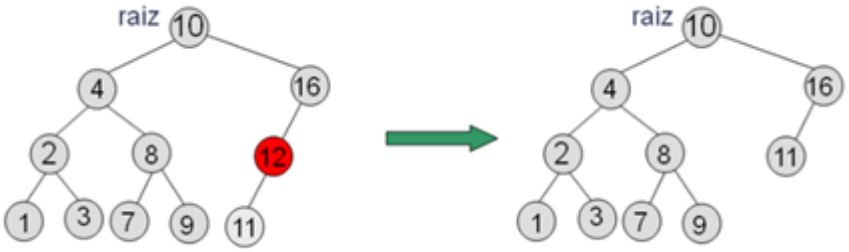


(b) (b)

Figura 10.38– Remoção do nó folha em uma BST. (a) antes da remoção (b) após a remoção

Caso 2

O nodo a ser removido possui um e somente um filho. Neste caso simplesmente substituímos o par (key,value) desse nó pelo par (key,value) do seu único filho. A figura 10.39 ilustra esse procedimento.

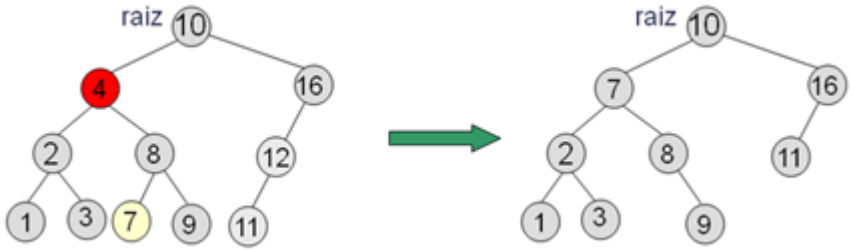


(b) (b)

Figura 10.39– Remoção do nó com um filho em uma BST. (a) antes da remoção (b) após a remoção

Caso 2

O nodo a ser removido possui dois filhos. Neste caso simplesmente substituímos o par (key,value) desse nó pelo par (key,value) do seu nodo sucessor. A figura 10.39 ilustra esse procedimento.



(b) (b)

Figura 10.39– Remoção do nó com dois filhos em uma BST. (a) antes da remoção (b) após a remoção

Aplicações das Árvores Binárias de Pesquisa

- Ordenação
- Encontrar repetições em um vetor
- Representação de Expressões
- Estruturas de índices para arquivos