Árvores

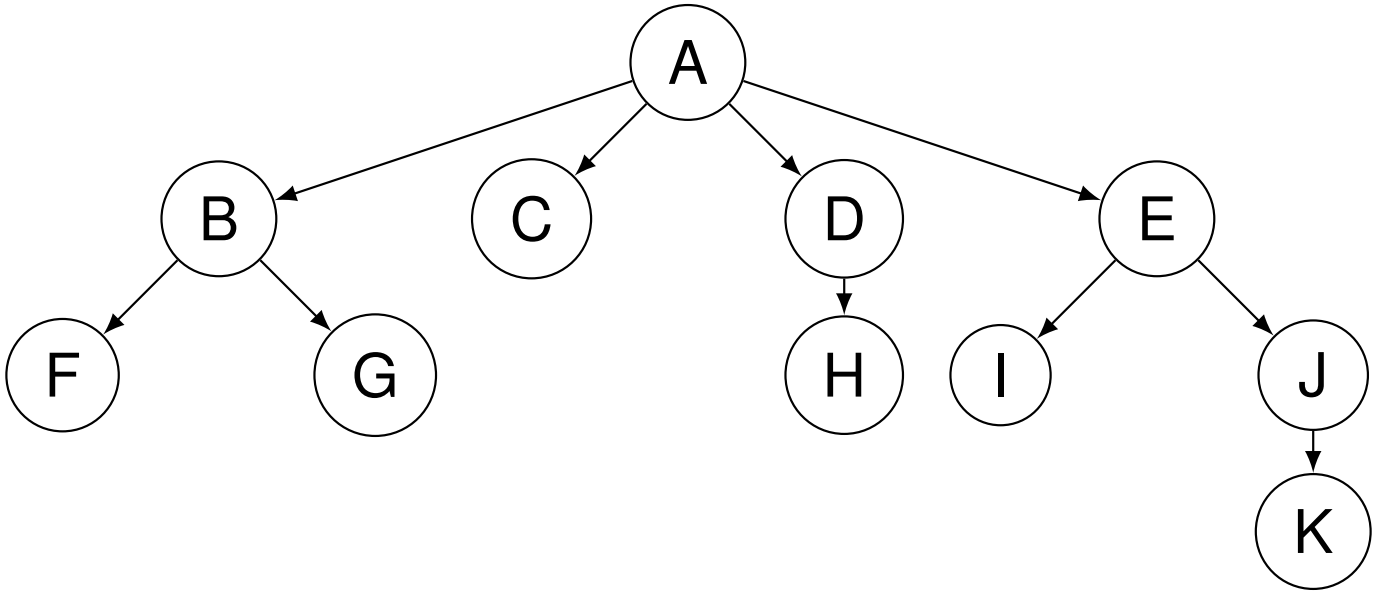
Thiago Martins, Fabio Gagliardi Cozman

PMR2300 / PMR3201
Escola Politécnica da Universidade de São Paulo

Árvores

- Árvore: estrutura composta por nós e arestas entre nós.
- As arestas são direcionadas (“setas”) e:
  - um nó (e apenas um) é a raiz;
  - todo nó (exceto a raiz) tem uma seta apontando para ele a partir de um outro nó (o “pai”);
  - um único caminho vai da raiz a qualquer nó.

Profundidade, altura



Profundidade da raiz é zero.
Profundidade de um nó não-raiz é a profundidade de seu pai mais um.
A *altura* da árvore é sua maior profundidade + 1.

Armazenamento

- Para armazenar os dados em uma árvore, temos que dispor de nós com referências aos filhos.
- Podemos criar um nó que mantenha uma lista ligada na qual estão as referências aos filhos.
- No caso da árvore anterior, nó A armazena lista com nós B, C, D, E; nó B armazena lista com nós F, G; etc.

Exemplo de aplicação

- Uma aplicação de árvores é a estrutura de pastas de sistemas operacionais como Unix, Linux ou Windows.
- Suponha que queiramos imprimir o nome de arquivos a partir de um diretório. Podemos fazer isso com o seguinte algoritmo recursivo:

```
def mostrar_arquivos(caminho):
    print(caminho)
    if os.path.isdir(caminho):
        for y in os.listdir(caminho):
            mostrar_arquivos(y)
```

Ordens de visita

- A ordem de “visita” aos nós é chamada de *ordem anterior*.
  No exemplo:
  A, B, F, G, C, D, H, E, I, J, K
- Suponha que a ordem seja diferente: primeiro os filhos, depois o pai. Então teremos:
  F, G, B, C, H, D, I, K, J, E, A
- Esta ordem de visita é então chamada de *ordem posterior*.

Recursão

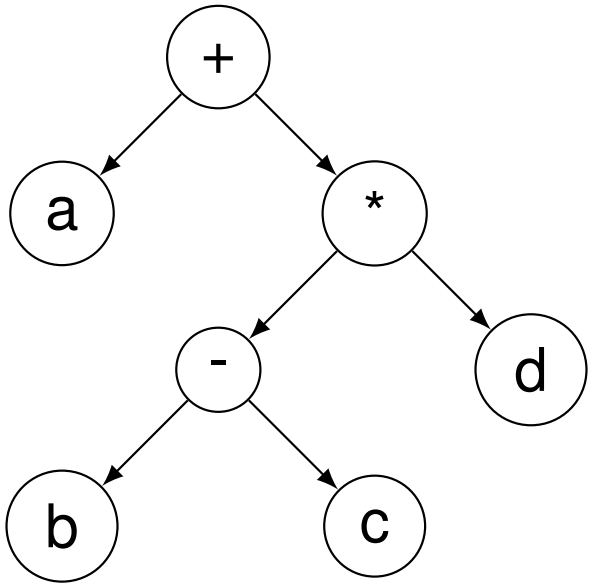
- De forma geral, árvores podem ser definidas recursivamente: cada nó é a raiz de uma subárvore.
- É possível usar essa propriedade para várias tarefas como, por exemplo,
  - número de nós a partir de um nó (incluindo o nó) = 1 + soma(número de nós para cada filho).

Operações Típicas

- Percorrer todos os itens.
- Buscar por um item.
- Adicionar um novo filho ou subárvore.
- Remover uma “folha” (nó sem filhos) ou subárvore.

Árvore Binária

- Um tipo particularmente importante de árvore é a árvore binária, aquela em que cada nó tem no máximo dois filhos.
- Considere um exemplo de aplicação: armazenamento de expressões. Nesse tipo de árvore, as “folhas” contém variáveis; os demais nós contém operações.



Ordem interior

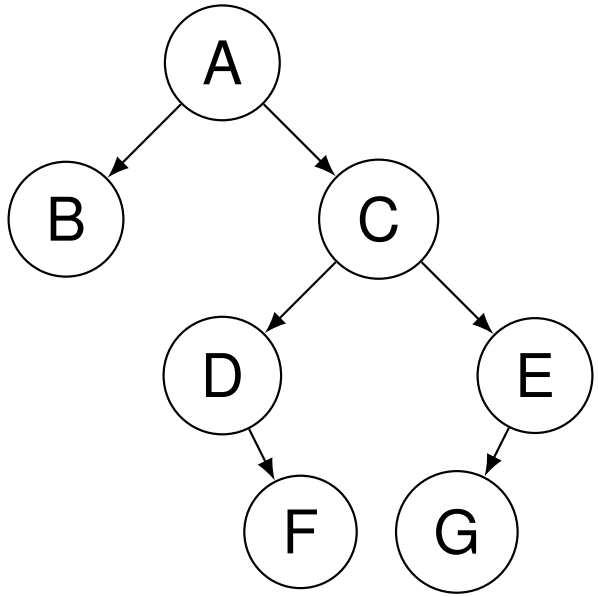
Para imprimir a expressão codificada através de uma árvore desse tipo:

- a partir da raiz,
  - imprima nó esquerdo recursivamente;
  - imprima o conteúdo do próprio nó;
  - imprima nó direito recursivamente.

Neste exemplo, obtemos $(a + ((b - c) * d))$.
Esta ordem de visita, particular a árvores binárias, é denominada *ordem interior*.

Ordens

Consideremos as várias ordens, para a árvore a seguir:



- Anterior: A, B, C, D, F, E, G.
- Interior: B, A, D, F, C, G, E.
- Posterior: B, F, D, G, E, C, A.

Cada uma dessas ordens pode ser gerada de forma recursiva.

Ordem Anterior

Ordem Anterior:

```
execute(n);
    visite n;
    se(filho esquerdo de n existe)
        execute(filho esquerdo de n);
    se(filho direito de n existe)
        execute(filho direito de n);
```

Ordem Interior

Ordem Interior:

```
execute(n):
    se(filho esquerdo de n existe)
        execute(filho esquerdo de n);
    visite n;
    se(filho direito de n existe)
        execute(filho direito de n);
```

Ordem Posterior

Ordem Posterior:

```
execute(n):
    se(filho esquerdo de n existe)
        execute(filho esquerdo de n);
    se(filho direito de n existe)
        execute(filho direito de n);
    visite No;
```

Implementação

```
class NoArvoreBinaria:
⋮
    def __init__(self,x,esquerda=ArvoreVazia,direita=ArvoreVazia):
        self._x = x
        self._e = esquerda
        self._d = direita

    def filhos(self):
        def _iterador_filhos():
            yield self._e
            yield self._d
        return _iterador_filhos()

    def val(self):
        return self._x
⋮
```

`ArvoreVazia` é um objeto especial que funciona como “Sentinela”. Ele ocupa o lugar de um filho quando um nó não o possui. Vide a sua implementação adiante.

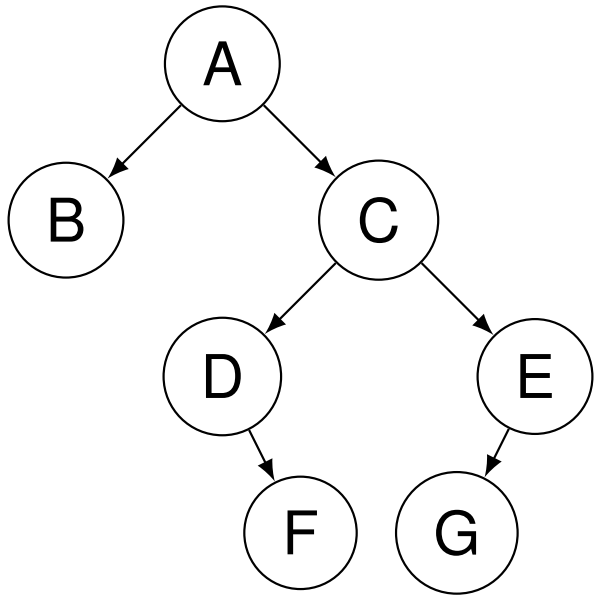
Implementação

```
class NoArvoreBinaria:
    ⋮
    def __len__(self):
        """ Retorna o tamanho total da arvore"""
        return 1 + len(self._e) + len(self._d)
    ⋮
```

A implementação de `__len__` dos objetos sentinela retorna sempre 0.

Busca em árvores

- Árvores são muito usadas para representar problemas de busca.
- Os arcos a partir de um nó são encarados como caminhos alternativos para encontrar algo de interesse.
- Suponha que um domínio seja representado como segue. Nesse caso, a busca por F exige a visita em A e C e D.



Ordens de busca

- Existem duas maneiras básicas de fazer busca por um nó em uma árvore:
  - busca por *nível*: visitamos todos os nós em uma determinada profundidade e passamos para a profundidade seguinte;
  - busca por *profundidade*: visitamos nós em profundidades consecutivas (obtido por ordem anterior).
- A busca por nível não corresponde a nenhuma ordem vista anteriormente.
- No exemplo, a busca por nível visita
  A, B, C, D, E, F, G.
- Esse tipo de busca é também chamado de busca em *largura*.

Compressão de Textos

- Um problema clássico em computação é o de como representar uma sequência de bits com informação, ou seja, uma sequência em que os bits representam símbolos.
- Uma maneira simples é usar um código com mesmo número de bits para cada símbolo.
  - Por exemplo: $A = 001$; $B = 010$; $C = 100$.
- Em geral é possível comprimir um "texto" codificado em tamanho fixo, usando códigos de tamanho variável. **Idéia: quanto mais frequente o símbolo, menor seu código!**

Compressão de Textos - Exemplo

- Suponha que A ocorre 80%, B ocorre 10 % e C ocorre 10% .
- Considere o código: $A = 0$; $B = 10$; $C = 11$.
- O comprimento médio é $1 \times 0.8 + 2 \times 0.1 + 2 \times 0.1 = 1.2$.

Compressão de Textos - Algoritmo de Huffman

- Um método importante para geração de códigos é o método de Huffman.
- A técnica usada é a de “codificação de prefixos”: nenhum código é prefixo de outro código.
- O algoritmo para geração de códigos de Huffman é baseado em árvores binárias.

Algoritmo de Huffman (1)

Entrada: tabela em que cada símbolo s_i tem uma frequência f_i ($f_i >= 0, \sum f_i = 1$);
Saída: Código c_i para cada símbolo s_i.
Procedimento:

1. Crie uma árvore binária para cada símbolo, contendo apenas o par (s_i, f_i) como raiz;
2. Ordene as árvores em ordem decrescente de frequências, colocando-as em uma lista ordenada L. Em caso de árvores de mesma frequência utilize como apoio a tabela de entrada, ou seja, a ordem de inserção das árvores de mesma frequência deve seguir a ordem dos elementos da tabela;

Algoritmo de Huffman (2)

3. Enquanto houver mais de uma árvore disponível:
   1. Retire a árvore T_1 cuja raiz tem menor frequência, indicada por f_1 (retire último elemento de L);
   2. Retire a árvore T_2 cuja raiz tem menor frequência (entre as árvores restantes), indicada por f_2 (retire o último elemento de L);
   3. Crie uma nova árvore T_{12} em que:
      1. a raiz tem 2 filhos:
         - o filho esquerdo é a raiz de T_2;
         - o filho direito é a raiz de T_1;
      2. a raiz armazena $(S_2 \cup S_1, f_2 + f_1)$, onde S_1 são símbolos na raiz de T_1 e S_2 são os símbolos na raiz de T_2;
   4. Insira T_{12} em L, mantendo L ordenada por frequências (das raízes). Se existe T_i tal que sua frequência é igual a $f_1 + f_2$, coloque T_{12} "depois" de T_i;

Algoritmo de Huffman (3)

4. Os códigos são gerados a partir da única árvore T restante ao final do processo: atribua 0 para cada arco esquerdo e 1 para cada arco direito; o código de cada folha é lido no caminho entre a raiz e a folha.

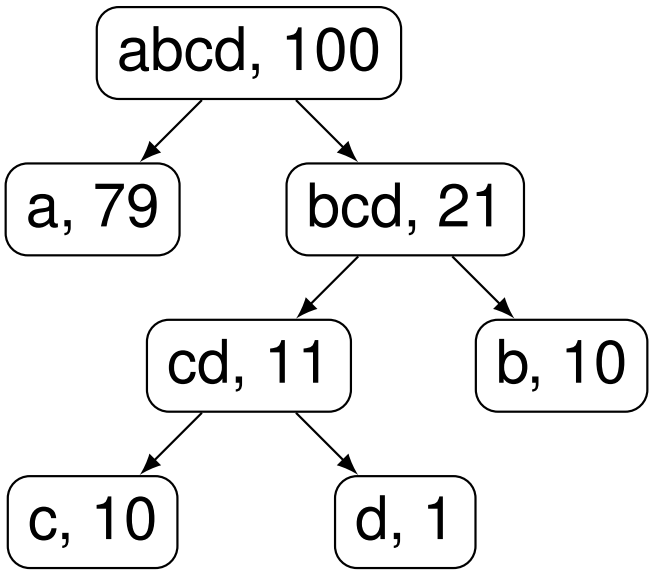
Se o código c_i do símbolo s_i tem comprimento $|c_i|$ em bits, o comprimento médio de uma mensagem será:
$\sum_i |c_i| \times f_i$

Exemplo

Considere:

s_i	a	b	c	d
f_i (em %)	79	10	10	1

Obtemos:



Exemplo: código

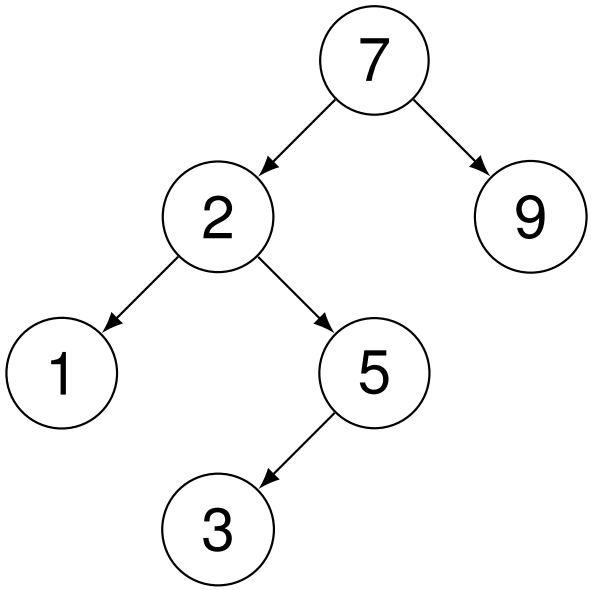
- Temos: $a = 0$; $b = 11$; $c = 100$; $d = 101$.
- O comprimento médio é
  $0.79 \times 1 + 0.1 \times 2 + 0.1 \times 3 + 0.01 \times 3 = 1.3$.
- Se o código fosse de tamanho fixo igual a 2 bits, o comprimento médio seria
  $0.79 \times 2 + 0.1 \times 2 + 0.1 \times 2 + 0.01 \times 2 = 2$.

Árvores Binárias de Busca: definição

- Considere uma árvore onde cada nó contenha um “dado” e um “identificador” para o dado.
- Suponha que os identificadores possam ser ordenados (por exemplo, são números).
- Árvores binárias de busca são árvores binárias em que:
  - todos os nós descendentes de um nó X, à esquerda de X, têm identificadores “menores” do que o identificador de X;
  - todos os nós descendentes de um nó X, à direita de X, têm identificadores “maiores” do que o identificador de X.
- Em uma árvore binária de busca, uma ordem interior passa pelos nós em ordem crescente de identificadores.

Árvores Binárias de Busca

Em uma árvore binária, uma ordem interior passa pelos nós em ordem crescente de identificadores.



Ordem Interior: 1, 2, 3, 5, 7, 9.

Operações básicas

- Busca: inicie na raiz e mova para a direita ou para a esquerda recursivamente.
- Mínimo: mova sempre à esquerda.
- Máximo: mova sempre à direita.
- Inserção: inicie na raiz, mova para direita ou esquerda recursivamente até encontrar o ponto de inserção.

Mínimo, máximo

```
class NoArvoreBinaria:
    .
    .
    .
    def max_val(self):
        """ Retorna o maior valor da arvore"""
        while self._d:
            self = self._d
        return self.val()

    def min_val(self):
        """ Retorna o menor valor da arvore"""
        while self._e:
            self = self._e
        return self.val()
```

Busca

Esta implementação usa *sentinelas* especiais para os filhos das folhas. Estes sentinelas são objetos cujo método `__contains__` retorna *sempre* `False`.

```
class NoArvoreBinaria:
⋮
    def __contains__(self, x):
        if x > self._x:
            return x in self._d
        elif x < self._x:
            return x in self._e
        return True
```

Inserção

Neste caso, a implementação de Estes sentinelas são objetos cujo método `__contains__` retorna *sempre* `False`.

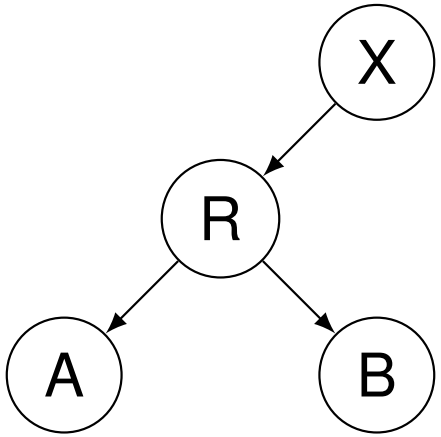
```
class NoArvoreBinaria:
⋮
    def insere(self, x):
        if x>self._x:
            self._d = self._d.insere(x)
        elif x<self._x:
            self._e = self._e.insere(x)
        else:
            raise Exception("Elemento ja existe")
        return self
```

Remoção

A remoção de um dado é mais complexa. Vejamos, em ordem, casos de dificuldade crescente:

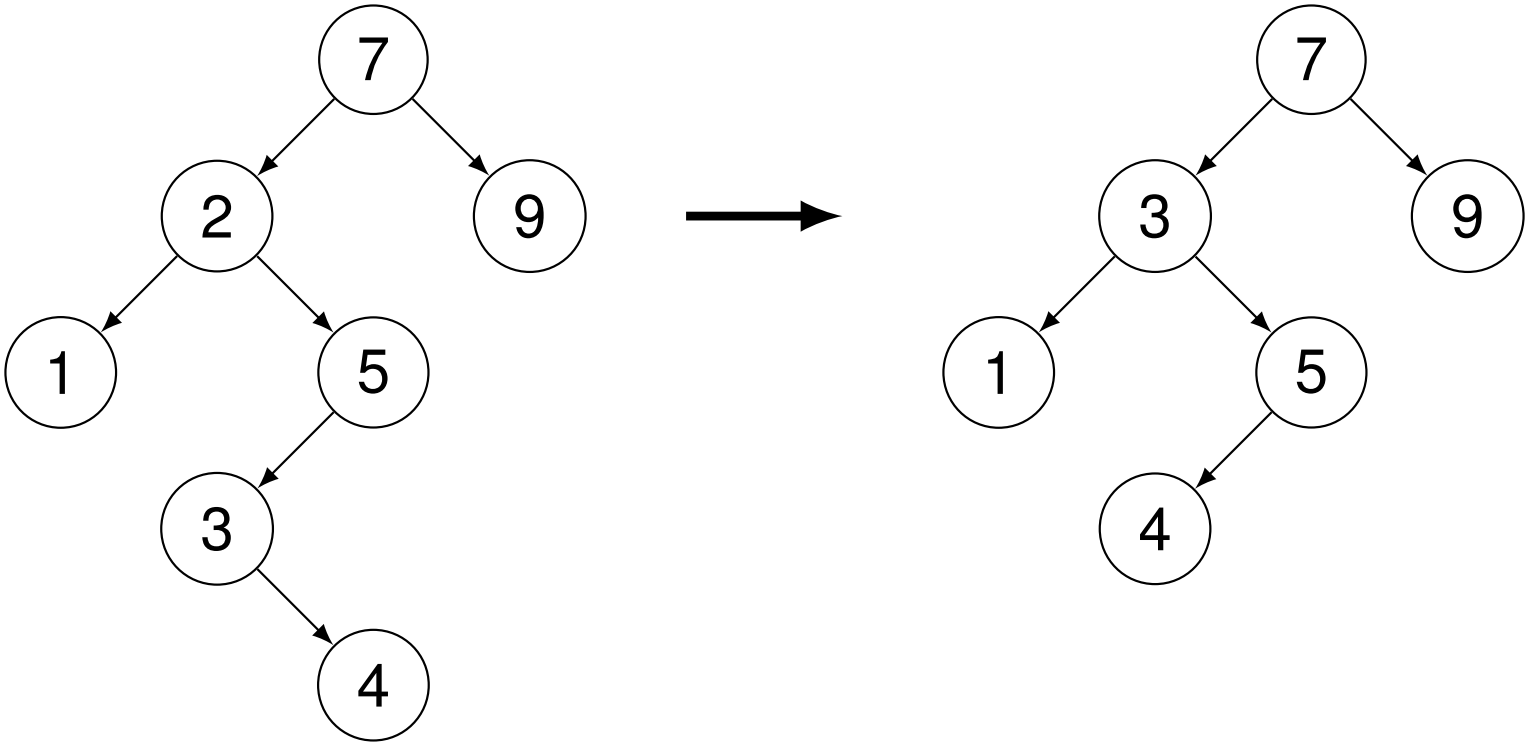
1. É feita uma busca e o dado a ser removido não existe: nada a fazer;
2. É feita uma busca e o dado a ser removido está em um nó com nenhum filho: simplesmente remova este nó;
3. É feita uma busca e o dado a ser removido está em um nó com um único filho: remova o nó e coloque seu filho no lugar;
4. É feita uma busca e o dado a ser removido está em um nó com dois filhos: este é o caso mais complicado - veja sua solução no exemplo a seguir:

Remoção



- Suponha que queiramos remover o nó R.
- Se X já tem dois filhos (ou seja, se X tem outro filho além de R), não temos como colocar A e B como filhos de X.
- Mesmo se R seja o único filho de X, há uma dificuldade: tanto A quanto B são “menores” que X, portanto B não pode figurar como nó direito de X.
- Solução: encontrar o menor dado na subárvore à direita da raiz R e colocar esse dado em R; depois eliminar o nó que originalmente continha o menor dado à direita de R.

Exemplo: removendo nó 2



Isso funciona, pois o nó contendo o menor dado em uma subárvore só pode conter 0 ou 1 filho. Portanto, remover esse nó sempre é trivial.

Remoção

```
def remove(self, x):
        if x > self._x:
            self._d = self._d.remove(x)
            return self
        elif x < self._x:
            self._e = self._e.remove(x)
            return self
        if not self._e:
            return self._d
        if not self._d:
            return self._e
        # Ambos os bra os sao nao-nulos:
        a = self._d
        while a._e: a = a._e
        # Troca-o com o elemento atual
        #  Note que aqui a estrutura est  temporariamente
        #  quebrada
        a._x, self._x = self._x, a._x
        self._d = self._d.remove(x)
        return self
```

Alguns pontos...

- Note: essa função retorna o nó raiz da árvore com dado removido.
- Note: da forma como definido, uma árvore de busca binária não tem elementos repetidos. Se for necessário armazenar elementos com mesmo identificador, é conveniente usar uma estrutura auxiliar (por exemplo, uma lista ligada) em cada nó.

Detalhes da Implementação

```
class NoArvoreBinaria:
    # Esta classe funciona como "sentinela"
    class _NoNulo:
        def insere(self, x):
            return NoArvoreBinaria(x)
        def remove(self, x):
            raise Exception("Elemento nao existe")
        def __contains__(self, x):
            return False
        def __bool__(self):
            return False
        def __len__(self):
            return 0
    ArvoreVazia = _NoNulo()
```

Detalhes da Implementação

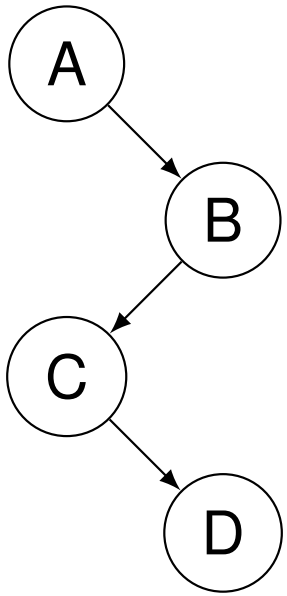
Como visto, o objeto `ArvoreVazia` é instância da classe interna `_NoNulo`. Esta classe implementa diversos métodos que complementam os algoritmos descritos acima. Em particular, destaca-se:

`insere` : Este método cria uma nova árvore binária.

`__bool__` : Método chamado por Python quando deseja obter o valor booleano de um objeto. Esta implementação permite escrever código do tipo `if self._e:`, que será executado se o elemento `_e` não for um objeto `_NoNulo`. Uma alternativa popular a sentinelas é usar uma referência `None`, mas isso requer mais verificações nos algoritmos descritos anteriormente.

Complexidade

- O custo de busca é proporcional ao número de nós visitados até atingir o nó procurado.
- O custo de uma busca, no pior caso, é igual à altura da árvore; no pior caso, esse número é igual ao número de nós:



Árvores balanceadas

- Uma árvore *balanceada* é uma árvore que tem altura de ordem $O(\log N)$, onde N é o número de nós na árvore.
- Intuitivamente, uma árvore balanceada é uma árvore "cheia", em que os nós não-folhas têm dois filhos.
- Suponha que os dados são inseridos em uma árvore com identificadores uniformemente distribuídos.
  - Note: "Uniformemente distribuídas" significa que todas as permutações de sequências de entrada têm mesma probabilidade.

Complexidade

Teorema: A altura média de uma árvore binária de busca com inserções uniformemente distribuídas é $1.38 \log N$.

Teorema: O esforço médio de busca em uma árvore binária com inserções uniformemente distribuídas é $O(\log N)$.

Árvores AVL

- Uma árvore AVL é uma árvore binária de busca onde cada nós satisfaz a seguinte propriedade: as duas sub-trees com raízes respectivamente nos filhos esquerdo e direito de um nó têm altura diferindo no máximo de 1.

Tamanho máximo

- Suponha que a altura de uma árvore AVL é h.
- O tamanho máximo é obtido quando todos os nós com profundidade k tem dois filhos, para k de 0 a $h-2$, e todos os nós com profundidade $h-1$ são folhas.
- Nesse caso, o tamanho $1+2+4+\cdots+2^{h-1}$.
- Ou seja, o tamanho é igual a 2^h-1.

Tamanho mínimo

- Suponha que a altura de uma árvore AVL é h.
- O tamanho mínimo $M(h)$ é obtido como segue.
- Para $h = 1$, temos $M(h) = 1$.
- Para $h = 2$, temos $M(h) = 2$.
- Para $h > 2$, temos

$$M(h) = 1 + M(h-1) + M(h-2)$$

(já que $M(h)$ é mínimo!).

Cálculo do tamanho mínimo

$$\begin{aligned} M(h) &> M(h-1) + M(h-2) \\ &> 2M(h-2) \\ &> 4M(h-4) \\ &> 8M(h-6) \\ &> 2^i M(h-2i). \end{aligned}$$

- Para h par, tome $i = (h/2) - 1$, e obtenha $M(h) > 2^{h/2-1} M(2) = 2^{h/2}$.
- Para h ímpar, tome $i = \lfloor h/2 \rfloor$, e obtenha $M(h) > 2^{\lfloor h/2 \rfloor} M(1) = 2^{\lfloor h/2 \rfloor}$.

Portanto $M(h) > 2^{h/2-1}$; ou seja, $h < 2 + 2\log_2 M(h)$.

Altura e tamanho em árvores AVL

- Considere árvore AVL de altura h e tamanho M.
- Temos $M \leq 2^h - 1 < 2^h$. Portanto $h > \log_2 M$.
- Temos $h < 2 + 2\log_2 M$ (cálculo anterior).
- Portanto $\log_2 M < h < 2 + 2\log_2 M$.
- Ou seja, h é $\mathcal{O}(\log M)$.
- Além disso, h é $\Theta(\log M)$.

Inserção em árvores AVL

- Considere que foi feita uma inserção (como anteriormente implementada) em uma árvore AVL. Pode ter ocorrido uma violação da propriedade de alturas diferindo por no máximo 1.
- Note que apenas os nós no “caminho” da inserção podem ter ficado desbalanceados (pois apenas suas sub-árvores são afetadas).
- Considere que o nó X está desbalanceado (ou seja, as alturas de suas sub-árvores têm diferença maior que 1), e todos os nós abaixo de X já foram balanceados.
- Temos 4 possíveis casos, dependendo de onde ocorreu a inserção abaixo de X.

Rotações

- Inserção na sub-árvore esquerda do filho esquerdo de X: rotação simples.
- Inserção na sub-árvore direita do filho direito de X: rotação simples.
- Inserção na sub-árvore direita do filho esquerdo de X: rotação dupla.
- Inserção na sub-árvore esquerda do filho direito de X: rotação dupla.

Rotação simples (caso 1)

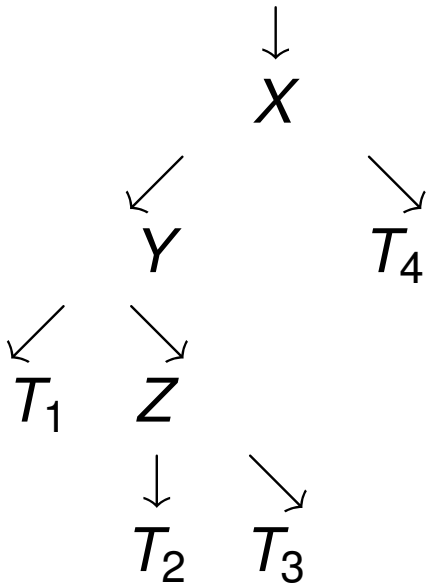
Inserção na sub-árvore esquerda do filho esquerdo de X.
Considere árvore após inserção normal (note: T_2 e T_3 tem mesma altura!):

Rotação simples (caso 2)

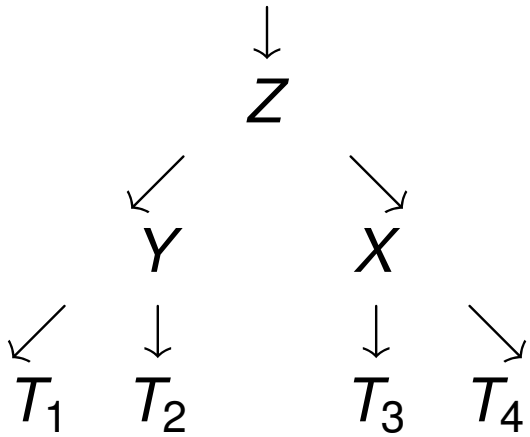
Inserção na sub-árvore direita do filho direito de X.
Considere árvore após inserção normal (note: T_1 e T_2 tem mesma altura!):

Rotação dupla (caso 3)

Inserção na sub-árvore direita do filho esquerdo de X.
Considere árvore após inserção normal:

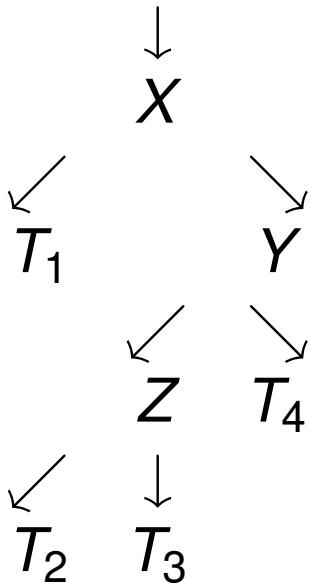


Rotação:



Rotação dupla (caso 4)

Inserção na sub-árvore esquerda do filho direito de X.
Considere árvore após inserção normal:



Rotação: