

## Árvores Binárias de Busca

### 1. Um breve comentário sobre os algoritmos de busca em tabelas

De uma maneira geral, são realizadas operações de busca, inserção e remoção de elementos numa tabela.

A busca sequencial tradicional é  $O(N)$ . Não é eficiente, mas permite inserções e remoções rápidas. A inserção pode ser feita no final da tabela, pois a ordem não precisa ser preservada. A remoção de um elemento deveria ser seguida de uma compactação da tabela que é demorada. Podemos simplesmente marcar o elemento removido, substituindo-o por um valor especial que nunca fará parte da tabela. A remoção fica mais simples, mas não diminuimos o tamanho da tabela.

A busca binária é  $O(\log N)$ . É muito eficiente, mas a tabela deve estar em ordem crescente ou decrescente. Portanto inserções e remoções são muito ineficientes. Para inserir ou remover mantendo a ordem, é necessário deslocar parte da tabela.

A busca em tabela hash sequencial depende da função de hash e da variedade dos dados. Uma vantagem é que permite inserção de novos elementos. A remoção não é permitida, pois altera a estrutura da tabela. Entretanto podemos simplesmente marcar o elemento removido como na busca sequencial acima. Ele permanece na tabela, mas com status de removido.

No caso geral, pouco se pode afirmar sobre a eficiência do hash em tabela sequencial. Depende da função de hash e dos dados. No pior caso é  $O(N)$ . Outro inconveniente é que no hash a tabela ocupa mais espaço.

No caso do hash com lista ligada, inserção e remoção são facilitadas com a ocupação ideal de memória. Entretanto no pior caso, a busca continua sendo  $O(N)$ .

A situação ideal seria um algoritmo que tivesse a eficiência da busca binária  $O(\log N)$ , permitisse inserções e remoções rápidas e que a tabela ocupasse somente o espaço necessário.

Algoritmo	Busca	Inserção	Remoção
Sequencial	$O(n)$	$O(1)$	$O(1) - [1]$ $O(n) - [2]$
Binária	$O(\log n)$	$O(n) - [3]$	$O(1) - [1]$ $O(n) - [2]$
Hash	$O(1) - [4]$ $O(n) - [5]$	$O(1) - [4]$ $O(n) - [5]$	$O(1) - [1]$ $O(n) - [6]$
Hash com Lista Ligada	$O(1) - [4]$ $O(n) - [5]$	$O(1) - [7]$ $O(n) - [8]$	$O(1) - [4]$ $O(n) - [5]$
Ideal	$O(\log n)$	Melhor que $O(n)$	Melhor que $O(n)$

Observações:

- [1] – Marcando o elemento. Sem compactar a tabela
- [2] – Compactando a tabela
- [3] – Para manter a classificação, tem que colocar o elemento em sua devida posição
- [4] – Se for possível uma função de hash que garanta uma boa distribuição de forma a limitar a quantidade elementos por sub-lista
- [5] – O pior caso ocorre quando os elementos estão todos contíguos
- [6] – Ao remover um elemento é necessário verificar os seguintes para manter a estrutura da tabela
- [7] – Se a inserção for no primeiro elemento
- [8] – Se a inserção for no final e eventualmente todos estão na mesma lista

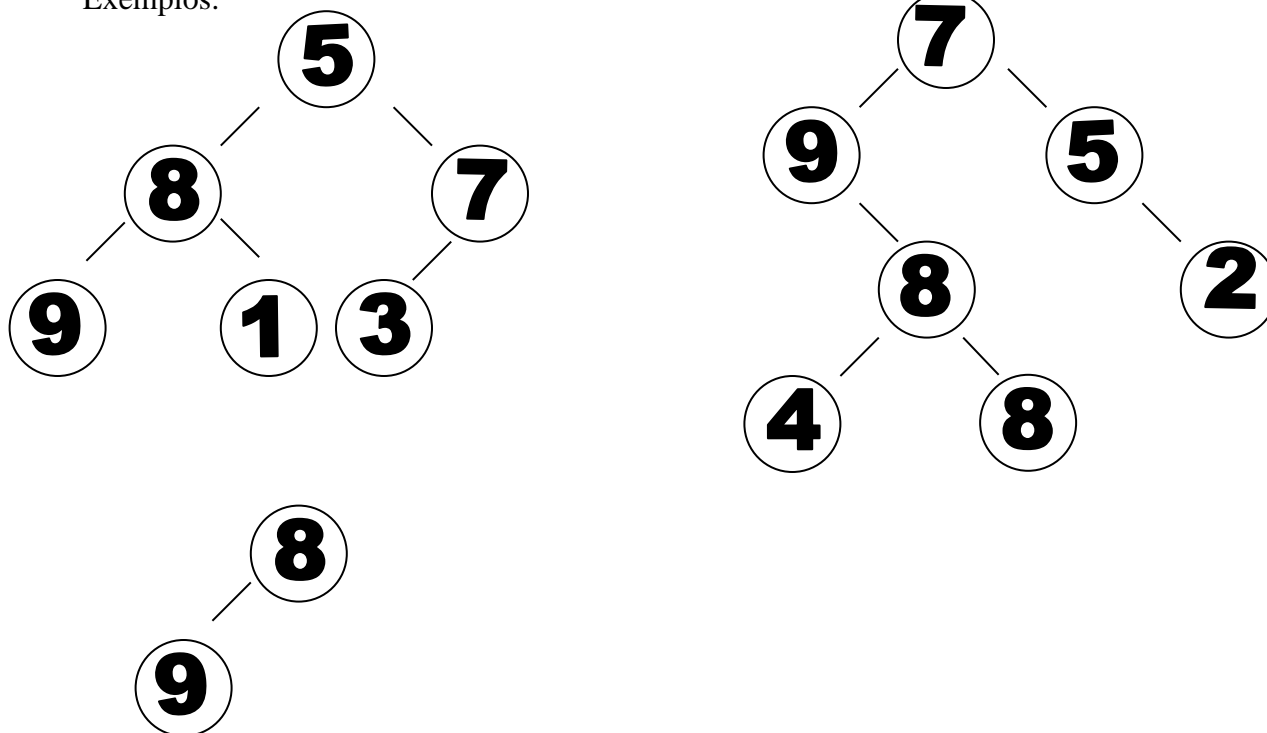
A situação ideal é conseguida quando a tabela tem uma estrutura em **árvore de busca**.

Dentre os vários tipos de árvores de busca, as mais simples são as árvores binárias de busca que veremos a seguir.

## 2. Árvores binárias

Chamamos de Árvores Binárias (AB), um conjunto finito  $T$  de nós ou vértices, onde existe um nó especial chamado **raiz** e os restantes podem ser divididos em dois subconjuntos disjuntos, chamados de sub-árvores esquerda e direita que também são Árvores Binárias. Em particular  $T$  pode ser vazio.

Exemplos:



Cada nó numa AB pode ter então 0, 1 ou 2 filhos. Portanto, existe uma hierarquia entre os nós. Com exceção da raiz, todo nó tem um nó pai.

Dizemos que o nível da raiz é 1 e que o nível de um nó é o nível de seu pai mais 1. A altura de uma AB é o maior dos níveis de seus nós.

Dizemos que um nó é folha da AB se não tem filhos.

### 3. Árvores binárias de busca

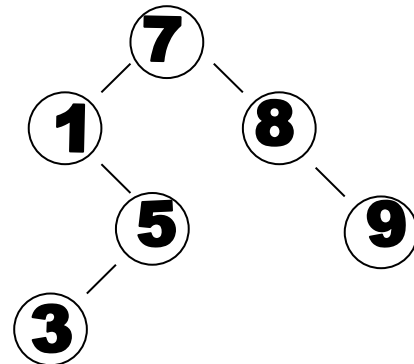
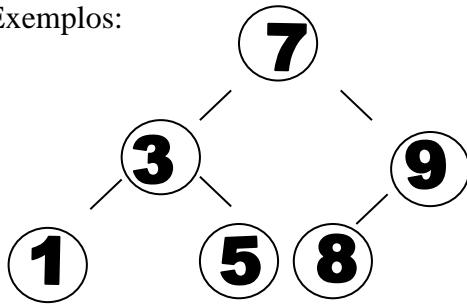
Seja  $T$  uma AB. Se  $v$  é um nó de  $T$ , chamamos de  $\text{info}(v)$  a informação armazenada em  $v$ .

Chamamos  $T$  de Árvore Binária de Busca (ABB) quando:

Se  $v_1$  pertencente à sub-árvore esquerda de  $v$  então  $\text{info}(v_1) < \text{info}(v)$ .

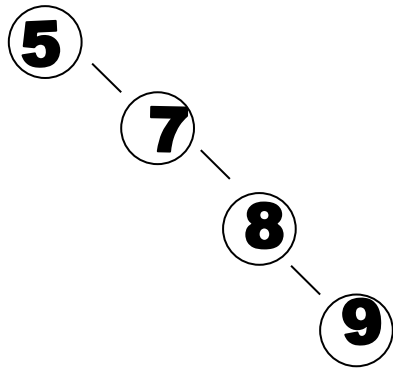
Se  $v_2$  pertencente à sub-árvore direita de  $v$  então  $\text{info}(v_2) > \text{info}(v)$ .

Exemplos:

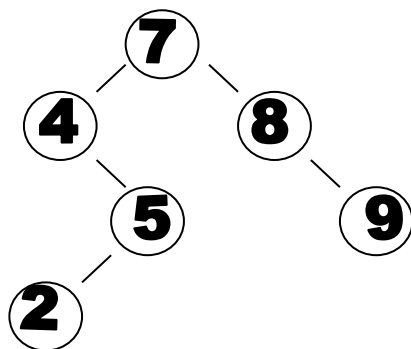


Os exemplos acima mostram que podemos ter várias ABBs com os mesmos elementos. Conforme veremos à frente o objetivo é sempre termos uma ABB de menor altura. Nesse sentido a primeira ABB acima é melhor que a segunda.

Um exemplo de AB de muitos níveis e poucos elementos:



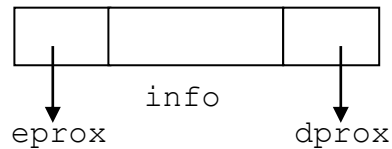
O exemplo abaixo não é ABB. O 2 está à direita do 4.



Uma ABB pode ter elementos repetidos. Podemos colocá-los na sub-árvore esquerda ou direita. Nos algoritmos abaixo vamos considerá-los sempre à direita. Dessa forma, os algoritmos para procurar um determinado elemento caso ele apareça mais vezes ficam mais simples.

#### 4. Árvores binárias como listas ligadas

Podemos representar uma ABB com uma lista ligada, onde cada elemento tem os seguintes campos:



info - campo de informação

eprox - apontador para a sub-árvore esquerda

dprox - apontador para a sub-árvore direita

Portanto, podemos representar uma ABB como um nó que é a raiz e duas referências para as respectivas sub-árvore esquerda e direita:

```
class ABB:
```

```
    def __init__(self, raiz):
        ''' cria uma nova ABB com esta raiz e sem filhos. '''
        self._info = raiz
        self._eprox = None
        self._dprox = None
```

Podemos agora escrever os seus métodos principais. Em vez disso, vamos escrever os algoritmos como funções independentes.

#### 5. Algoritmos de busca

##### A1

Função que procura elementos com info igual a v numa ABB h. A versão abaixo é recursiva e devolve o primeiro nó encontrado com info igual a v ou None se não encontrou:

```
# Procura elemento com info igual a v na ABB h
# Versão recursiva
def busca(h, v):
    if h is None: return None
    t = h._info
    if t == v: return h
    if v < t:
        return busca(h._eprox, v) # procura à esquerda
    else:
        return busca(h._dprox, v) # procura à direita
```

## Complexidade da busca

No pior caso, o número de comparações é igual ao número de nós da árvore, no caso em que a árvore tem tantos níveis quanto o número de elementos. Portanto a complexidade é  $O(N)$ .

A complexidade é a altura da árvore, portanto é conveniente que a árvore tenha sempre altura mínima.

A árvore que possui tal propriedade é uma AB dita **completa** (todos os nós com filhos vazios estão no último ou penúltimo nível). Neste caso a complexidade é  $O(\log N)$  ou seja: Se  $T$  é uma AB completa com  $N > 0$  nós então  $T$  possui altura  $H$  mínima e  $H = 1 + \log_2 N$  (considerando o valor de  $\log_2 N$  truncado).

O lema a seguir dá a relação entre altura e número de nós de uma AB completa:

### Lema:

Seja  $T$  uma AB completa com  $N$  nós e altura  $H$ .

Então  $2^{(H-1)} \leq N \leq 2^H - 1$ .

### Prova:

Se a AB completa possui apenas 1 nó no seu nível inferior então  $N = 2^{(H-1)}$ .

Se a AB completa está cheia  $N = 2^H - 1$ .

### A2

Vejam agora a versão não recursiva para a busca. A chamada `buscaNR(r, x)` procura elemento com `info` igual a `x` na ABB `r`. Devolve o primeiro nó encontrado com `info = x` ou `None` caso não encontre:

```
# Procura elemento com info igual a v na ABB h
# Versão não recursiva
def buscaNR(h, v):
    p = h
    while p is not None:
        t = p._info
        if v == t: return p # encontrou
        if v < t: p = p._eprox # à esquerda
        else: p = p._dprox    # à direita
    # se chegou aqui é porque não encontrou
    return None
```

## 6. Outros algoritmos

### A3

A função a seguir conta o número de nós de uma AB com determinado valor de `info`. A chamada `conta(r, x)` devolve o número de elementos iguais a `x` da AB `r`.

```
# conta elementos com info igual a v na ABB h
def conta(h, v):
    if h is None: return 0
    # verifica se conta este nó
    if v == h._info: a = 1
    else: a = 0
    # conta esta nó mais as ABBs esquerda e direita
    return a + conta(h._eprox, v) + conta(h._dprox, v)
```

Estamos supondo neste caso que os elementos iguais podem estar à direita ou à esquerda. O algoritmo acima percorre toda a ABB.

Exercícios:

- 1) Refaça, supondo que elementos iguais, estarão sempre à direita.
- 2) Refaça novamente usando algoritmo não recursivo. Nas duas formas, recursivo ou não recursivo.

#### A4

Transformar um vetor de  $n$  elementos, já ordenado, numa ABB mais ou menos equilibrada. A idéia é sempre pegar um elemento médio como raiz da sub-árvore. Para facilitar as chamadas recursivas vamos fazer a função de modo que a mesma se aplique a qualquer trecho contíguo do vetor. Assim, a chamada `raiz = monta(a, 0, n-1)` faz a montagem da árvore com os elementos `a[0]` até `a[n-1]`, devolvendo uma referência para a raiz da árvore. A chamada `raiz = monta(a, n1, n2)` faz o mesmo para os elementos `a[n1]` até `a[n2]`.

```
# Monta uma ABB a partir de uma lista já classificada
# Elementos repetidos podem ficar tanto a esquerda quanto a direita
def montaABB(a, iesq, idir):
    # verifica se há elementos na lista
    if iesq > idir: return None
    m = (iesq + idir) // 2 # element médio
    abb = ABB(a[m])
    abb_esq = montaABB(a, iesq, m - 1)
    abb_dir = montaABB(a, m + 1, idir)
    abb._eprox = abb_esq
    abb._dprox = abb_dir
    return abb
```

O trecho abaixo cria duas ABBs a partir de listas classificadas.

```
lista = [0, 1, 2, 3, 4, 5, 6]
outralista = [0,1,1,2,2,2,3,3,3,3,4,4,4,5,5,6,7,8,9]
mabb = montaABB(lista, 0, 6)
moabb = montaABB(outralista, 0, 18)
```

### Exercícios:

- 1) Desenhe a ABB construída pelo algoritmo acima para as listas:  
[0, 1, 2, 3, 4, 5, 6]  
[0, 1, 1, 3, 4, 4, 5, 6, 6]
- 2) Refaça a função `montaABB` acima usando sub-listas do Python. A função terá então um só parâmetro `montaABB(a)`.

### A5

Função que conta o número de nós de uma AB. A chamada `conta(r)`, devolve o número de nós da AB apontada por `r`.

```
# Conta o número de nós de uma ABB h
def contaNN(h):
    if h is None: return 0
    return 1 + contaNN(h._eprox) + contaNN(h._dprox)
```

### Exercícios

Baseado na solução acima escreva as seguintes funções:

1. Função `conta1(h)` que conta o número de folhas de uma AB `h`.
2. Função `conta2(h)` que conta o número de nós que tenham pelo menos um filho de uma AB `h`.
3. Função `conta3(h, x)` que conta número de elementos com `info >= x` de uma ABB `h`.
4. Idem ao problema A4 acima, considerando uma ABB onde elementos iguais ficam à direita.

### A6

Função que devolve uma lista com todos os campos de informação (`info`) dos nós de certo nível da ABB. A chamada `listaniveis(h, lista, 3, 0)` lista todos os nós de nível 3 da ABB `h`. Vamos convencionar que a raiz tem nível zero. O último parâmetro é usado apenas para controle e na chamada inicial tem que ser igual ao nível da raiz.

```
# Devolve lista com todos os nós de um certo nivel niv
def listaniveis(h, listnos, niv, niv_atual):
    if niv == niv_atual:
        if h is not None: listnos.append(h._info)
        else: listnos.append(None)
        return
    # ainda não chegou neste nível
    if h is not None:
        listaniveis(h._eprox, listnos, niv, niv_atual + 1)
```



```
        listaniveis(h._dprox, listnos, niv, niv_atual + 1)
    else:
        listaniveis(None, listnos, niv, niv_atual + 1)
        listaniveis(None, listnos, niv, niv_atual + 1)
```

O trecho abaixo lista todos os níveis de uma ABB:

```
# lista cada um dos níveis da ABB mabb
k = 0
while True:
    ln = []
    listaniveis(mabb, ln, k, 0)
    if ln.count(None) == 2 ** k: break
    print("nivel ", k, ":", ln)
    k = k + 1
print("Esta ABB tem", k, "níveis")
```

**A7**

Transformando o trecho acima numa função:

```
def ImprimeNiveisABB(h, nomeABB):
    print("\n\nLista a ABB:", nomeABB, "nível a nível")
    k = 0
    while True:
        ln = []
        listaniveis(h, ln, k, 0)
        if ln.count(None) == 2 ** k: break
        print("nivel ", k, ":", ln)
        k = k + 1
    print("ABB", nomeABB, "com", k, "níveis")
```

**A8**

Função que percorre a ABB, visitando a sub-árvore esquerda, a raiz e a sub-árvore direita. Nesta ordem, os nós serão visitados em ordem crescente do campo info.

```
def ImprimeABB(h):
    if h is None: return
    ImprimeABB(h._eprox)
    print(h._info)
    ImprimeABB(h._dprox)
```

Exercícios:

- 1) Idem visitando primeiro a raiz, sub-árvore esquerda e direita (ordem pré-fixa)
- 2) Idem visitando primeiro a sub-árvore esquerda, a direita e a raiz (ordem pós-fixa)

## 7. Algoritmos de inserção numa ABB

Um novo elemento é inserido sempre como uma folha de uma ABB. É necessário descer na ABB até encontrar o nó que será o pai deste novo nó.

#### A9

Uma versão não recursiva para a inserção numa ABB. A função `insere(h, x)`, insere elemento com info igual a `x` em seu devido lugar dentro da ABB `h`. Se o parâmetro `h` for `None`, a ABB não existe ainda e este é o primeiro elemento. Se `x` já estiver em algum nó da ABB, a inserção será à direita.

```
# Insere elemento com info x na ABB h
# Elementos iguais sempre a direita
def insere(h, x):
    # verifica se será o primeiro
    if h is None:
        h = ABB(x)
        return h
    # procura o lugar para inserir x
    # percorre a ABB até achar um nó com o filho None
    p, q = h, h
    while q is not None:
        v = q._info
        p = q
        # à esquerda ou à direita
        if x < v: q = q._eprox # esquerda
        else: q = q._dprox   # direita
    # Neste ponto, p é o pai do nó a ser inserido
    # Verificar novamente se é a esquerda ou direita
    q = ABB(x)
    if x < p._info: p._eprox = q
    else: p._dprox = q
    return h
```

O trecho abaixo constrói uma ABB por inserções sucessivas:

```
abb = None
for k in [4, 7, 85, 98, 4, 5, 6]: abb = insere(abb, k)
```

Exercício:

- 1) A versão acima usa duas referências (`p` e `q`) para percorrer a ABB. Refaça usando apenas uma referência.
- 2) Escreva uma versão recursiva da inserção

#### Complexidade da construção de uma ABB por inserções sucessivas

Para inserir elemento é necessário achar o seu lugar. Portanto a complexidade é a mesma da busca.

Usando-se o algoritmo acima e inserindo-se um a um, podemos no pior caso (ABB com um só elemento por nível - tudo à esquerda, tudo à direita ou ziguezague) chegar a:

$1+2+3+\dots+n = n \cdot (n+1) / 2$  acessos para construir toda a árvore. Portanto  $O(n^2)$ .

Se os elementos a serem inseridos estiverem ordenados, usando o algoritmo A4, a complexidade é  $O(N)$ . Observe que não é necessário percorrer a ABB no algoritmo A4. Porém, seria necessário ordenar antes a lista de elementos a serem inseridos.

### Complexidade da inserção em uma ABB

No pior caso é  $O(n)$ .

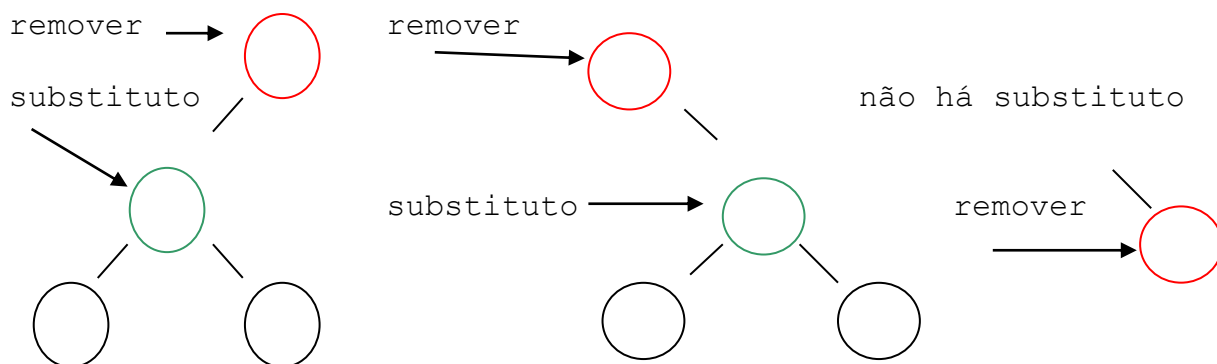
No melhor caso, supondo a **árvore completa** (folha da árvore) teremos que percorrer os níveis como na busca. Portanto  $1+\log n$ . Temos então um algoritmo  $O(\log n)$ .

### 8. Algoritmo de remoção numa ABB

A remoção é um pouco mais complexa que a busca ou inserção. O problema da remoção física de um nó é que é necessário encontrar outro nó para substituir o removido, caso o nó a ser removido tenha filhos.

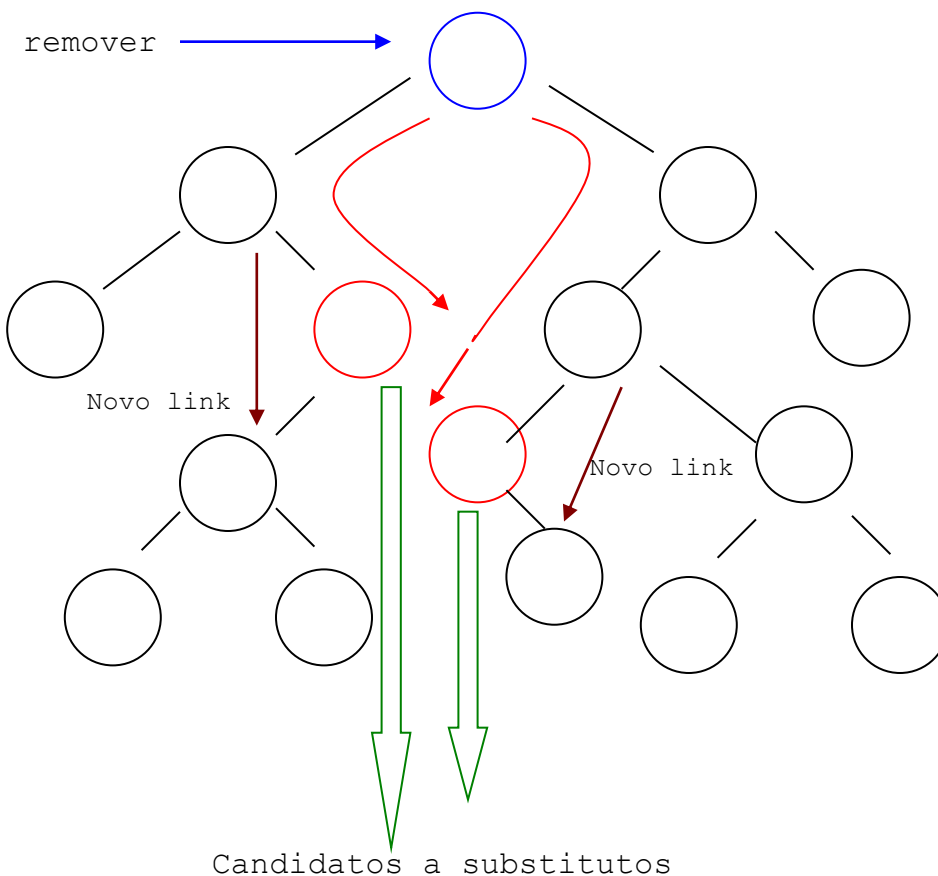
Dois casos a considerar:

- 1) O nó a ser removido não tem filhos esquerdo e/ou direito.



É só alterar o ponteiro para o nó a substituir e remover fisicamente o nó. Se não há filhos, basta mudar o ponteiro do pai para NULL.

2) O nó a ser removido tem filhos direito e esquerdo:



Os candidatos a substituto são obtidos percorrendo-se a ABB:

Um à esquerda e tudo a direita até achar nó com `dprox NULL`. Ou um a direita e tudo à esquerda até achar nó com `eprox NULL`. Determinado o candidato a substituir, é preciso então:

- Substituir o conteúdo (campo `info`) do nó a ser substituído pelo do candidato.
- Como o candidato pode ter filhos (apenas esquerdo ou apenas direito), esses filhos serão herdados pelo pai do candidato. Basta então mudar o ponteiro desse nó pai do candidato para o seu filho.

Essas são as linhas gerais para os algoritmos de remoção mas não vamos detalhá-los.

## 9. Árvores Binárias de Busca Completas

Já vimos que o problema das ABB é que ela pode ficar desbalanceada com a inserção e remoção de novos elementos. A situação ideal em uma ABB é que ela se já **completa** (como o menor número possível de níveis).

Como seria possível mantê-la **completa**?

Isso pode ser feito de 2 maneiras:

- 1) Toda vez que um elemento é inserido ou removido, rearranja-se a ABB para a mesma continue **completa**.
- 2) Inserir e remover elementos da maneira usual e de tempos em tempos executar um algoritmo que reconstrói a ABB deixando-a **completa**.

Existem vários algoritmos com esses objetivos. **Não serão vistos neste curso.**

Apenas citamos 2 tipos mais comuns abaixo. Nessas ABBs, os algoritmos de inserção e remoção já o fazem deixando a ABB completa ou balanceada.

Com uma ABB **completa**, chegamos a situação ideal de busca, pois temos um algoritmo equivalente ao da busca binária  $O(\log N)$ , em uma tabela que permite inserções rápidas ( $O(\log N)$ ) e remoções tão rápidas quanto possível ( $O(N)$  no pior caso). Além disso, só usa a quantidade de memória necessária.

## 10. Outras Árvores Binárias

Apenas citando os tipos mais importantes:

### 10.1 Árvores Binárias de Busca AVL (Adelson-Vesky e Landis (1962))

Cada nó mantém uma informação adicional, chamada fator de balanceamento que indica a diferença de altura entre as sub-árvores esquerda e direita.

As operações de inserção e remoção mantêm o fator de balanceamento entre -1 e +1.

### 10.2 Árvores Binárias de Busca Rubro-Negras

É uma ABB com as seguintes propriedades:

1. Todo nó é vermelho ou preto.
2. Toda folha é preta.
3. Se um nó é vermelho então seus filhos são pretos.
4. Todo caminho da raiz até qualquer folha tem sempre o mesmo número de nós pretos.

Com essas propriedades, é possível manter a ABB mais ou menos balanceada após inserções e remoções.

## 11. Outras Árvores de Busca

Árvores de Busca, não precisam ser necessariamente binárias. Podemos construir árvores com vários elementos em cada nó (n-árias). Cada elemento possui um ramo esquerdo (menores) e um ramo direito (maiores ou iguais).

Este é o caso das chamadas **B-Árvores**. Também **não serão vistos neste curso**.

São usadas principalmente para arquivos em banco de dados.

No caso de arquivos interessa muito diminuir a quantidade de acessos a disco. Assim, a cada leitura, vários nós estarão disponíveis na memória. A quantidade de níveis da árvore diminui e, portanto, a quantidade de acessos para se procurar um elemento.