

DESCRIÇÃO

Apresentação dos conceitos e os principais algoritmos de busca, inserção e remoção para as árvores binárias de busca, Análise de complexidade computacional dos algoritmos para manipulação das árvores binárias de busca, Apresentação do conceito de árvore balanceada e um algoritmo estático capaz de transformar uma árvore binária de busca em uma árvore balanceada, Conceituação de árvores AVL, Apresentação dos algoritmos dinâmicos para manutenção da propriedade AVL.

PROPÓSITO

A busca, inserção e remoção de informações em uma estrutura de dados é um dos principais problemas da computação. Ao aprender a utilizar corretamente as estruturas de dados para resolver problemas específicos, você será capaz de desenvolver sistemas que são muito mais rápidos e eficientes. As árvores são uma estrutura de dados no qual o problema da busca, inserção e remoção é resolvido em um tempo proporcional ao logaritmo do número de chaves na estrutura de dados.

OBJETIVOS

MÓDULO 1

Reconhecer os tipos de percursos utilizados em árvores binárias

MÓDULO 2

Definir árvores binárias de busca

MÓDULO 3

Definir o conceito de árvore balanceada

MÓDULO 4

Definir árvores AVL

INTRODUÇÃO

Neste tema, aprenderemos a utilizar uma estruturas de dados capaz de realizar busca, inserção e remoção em complexidade computacional de $O(\log n)$. Para tal, partiremos do conceito de árvores binárias, explorando o conceito de percursos, em seguida, definiremos as árvores binárias de busca e seus principais algoritmos, comparando a complexidade computacional com as listas lineares e as estruturas de dados não organizadas.

Explicaremos o conceito de balanceamento e aprenderemos como transformar árvores não balanceadas em balanceadas e sobre uma estrutura de dados completamente dinâmica capaz de realizar as operações de busca, inserção e remoção em $O(\log n)$, as árvores AVL.

MÓDULO 1

🕒 Reconhecer os tipos de percursos utilizados em árvores binárias

Antes de iniciarmos os estudos dos percursos em árvores binárias, é interessante lembrarmos o conceito de árvore binária e como esta estrutura é representada em memória.

ÁRVORE BINÁRIA

Uma árvore binária T é um conjunto vazio ou é composta pelos seguintes elementos:

Uma entidade n chamada nó raiz e as entidades T_e e T_d , respectivamente,

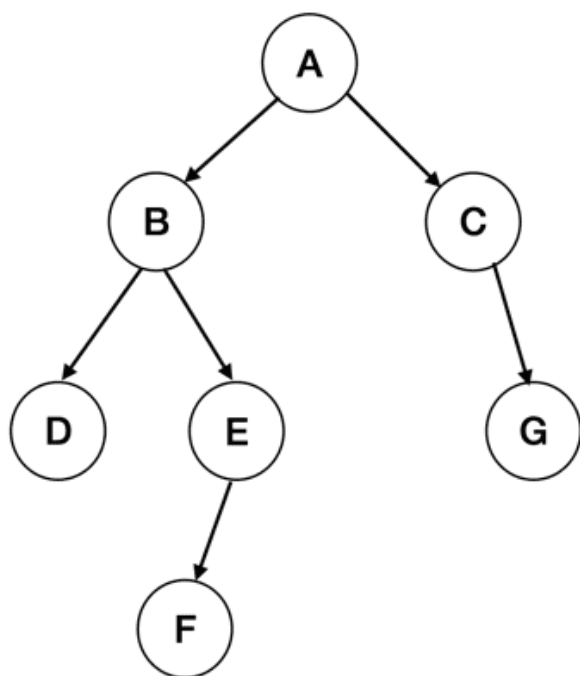


As subárvores esquerda e direita de T , que também são árvores binárias.

Observe que, como T_e e T_d são árvores binárias, a definição é recursiva, isto é, T_e possui um nó raiz ou é uma subárvore vazia. O mesmo vale para T_d .

Usualmente, representamos uma árvore através da representação gráfica na qual destacam-se a raiz da árvore T e sua subárvore esquerda e direita.

A Figura 1 mostra esta representação.



📷 Figura 1 – Árvore Binária.

Na Figura 1, o nó “A” é a raiz da árvore T , T_e é composta pelos nós: “B”, “D”, “E” e “F”;



T_d é composta pelos nós: “C” e “G”. “B” e “C” são as raízes de T_e e T_d respectivamente.

Observe que, na árvore binária, existe, por definição, distinção entre a subárvore esquerda e direita. Por isso, na representação gráfica sempre fica evidente a posição esquerda ou direita do nó subordinado à raiz.

Na Figura 1, o nó “G” é o filho direito de “C” e não existe filho esquerdo de “C”, por exemplo.

Representação em memória

A forma mais comum de representar uma árvore em memória é utilizando alocação dinâmica. Na verdade, não representamos a árvore como um todo, mas, sim, uma referência para sua raiz que guarda

a chave (dado) e uma referência para a raiz das subárvores esquerda e direita.

A abstração mais utilizada para representar o nó de uma árvore é o agregado heterogêneo, comumente chamado de registro (record) ou estrutura (struct) nas linguagens de programação. O agregado heterogêneo é capaz de armazenar objetos de dados de tipos diferentes acessando-os através do mesmo identificador e distinguindo-os através de um seletor.

Para representar o nó da árvore, será construído um agregado heterogêneo contendo a chave associada ao nó, e ponteiros para as raízes das subárvores esquerda e direita.

Em pseudocódigo, utilizaremos a estrutura sintática abaixo para descrever um nó.

```
registro no {  
    Inteiro chave;  
    registro no *esq;  
    registro no *dir;  
}
```

Assim, a árvore é representada por uma referência para sua raiz, ponteiro para raiz, que, recursivamente, aponta para as raízes das subárvores esquerda e direita.

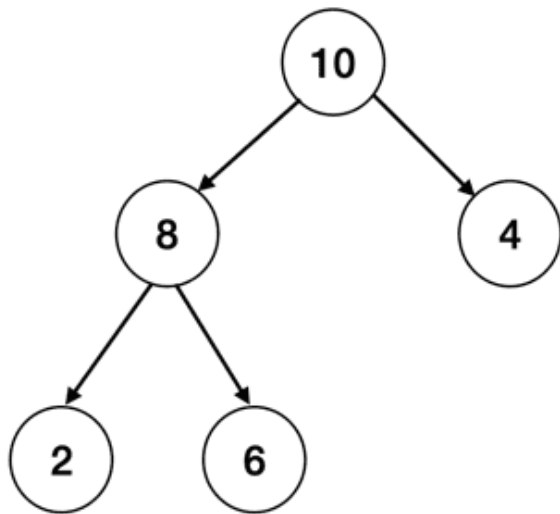
Convencionamos que a variável “raiz”, declarada da forma abaixo, é um ponteiro para a raiz da árvore.

```
registro no *raiz;
```

A árvore vazia é representada quando o ponteiro raiz aponta para o endereço 0, associado à constante NULA. Assim, para inicializar uma árvore vazia, basta inicializar o ponteiro raiz como nulo.

```
raiz = NULA;
```

Para que o conceito fique claro, será utilizada a árvore da Figura 2 para ilustrar a representação em memória da estrutura.

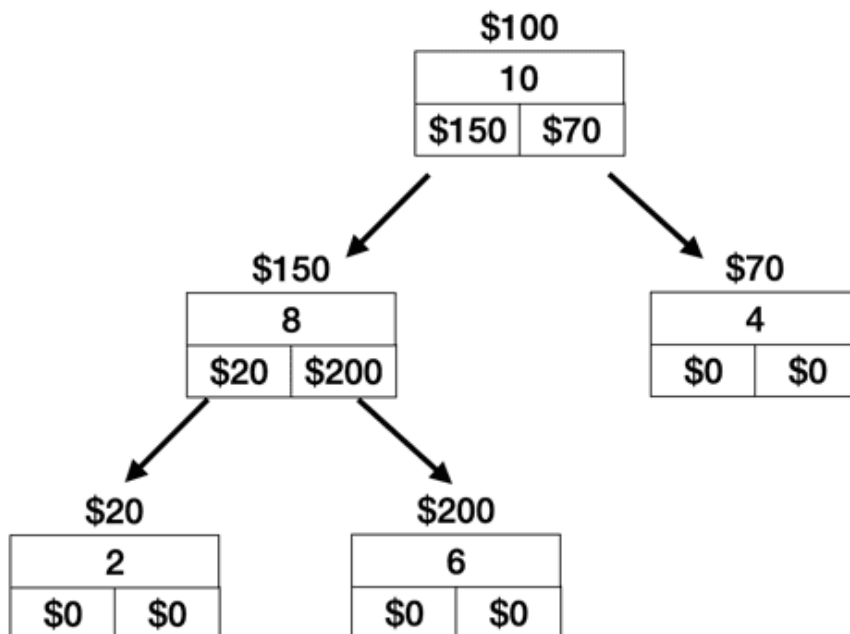


📷 Figura 2 – Árvore Binária Exemplo.

Na árvore da Figura 2, temos cinco nós, identificados pelos inteiros: **10, 8, 4, 2 e 6**.

Cada nó fazendo referência à raiz das subárvores esquerda e direita.

Esquemáticamente, em memória, a estrutura é representada como na Figura 3.



📷 Figura 3 – Esquemático da representação de uma árvore em memória.

Na Figura 3, o nó raiz da árvore T é armazenado no endereço \$100, que contém um agregado heterogêneo com a chave 10 e dois ponteiros \$150 e \$70, respectivamente, para o endereço das raízes das subárvores esquerda e direita da raiz.

Caso uma variável seja declarada como do tipo registro no *v, o operador “->”, possui a semântica de fornecer acesso ao objeto de dados referenciado por v no agregado heterogêneo. Por exemplo, na Figura 3, caso v=\$20, v->chave será igual a 2.

💬 COMENTÁRIO

Outro detalhe importante desta representação é que existe referência do pai para o filho, ou seja, do nó 10 chegamos aos nós 8 e 4, porém o filho não faz referência para o pai, isto é, do nó 4 não é possível acessar o nó pai, que contém a chave 10.

PERCURSOS EM ÁRVORES

Um percurso é a visita sistemática aos elementos de uma estrutura de dados.

Vale destacar que **visitar** significa realizar uma operação e não acessar o conteúdo armazenado no elemento da estrutura de dados. Ou seja, especificamente para árvores, acessar um nó da árvore não configura uma visita.

A visita é uma operação que tem sentido na semântica da aplicação desejada.

A visita mais simples que podemos definir é a impressão do rótulo da chave armazenada no nó visitado.

A seguir, apresentaremos três percursos distintos para árvores binárias:

Percurso em pré-ordem



Percurso em ordem simétrica



Percurso em pós-ordem

Observe que o conceito de árvore é **recursivo**, isto é, a estrutura foi definida em termos recursivos. Por esta razão, as definições dos percursos também são recursivas.

PERCURSO EM PRÉ-ORDEM

O percurso em pré-ordem é definido recursivamente abaixo.

A partir da raiz r da árvore T , percorre-se a árvore da seguinte forma:

ETAPA 01

Visita-se a raiz.

ETAPA 02

Percorre-se a subárvore esquerda de T , em pré-ordem.

ETAPA 03

Percorre-se a subárvore direita de T , em pré-ordem.

Considerando como a operação de visita a impressão do rótulo contido no nó visitado, veremos o resultado do percurso em pré-ordem da árvore da Figura 2.

Partindo-se da raiz, nó de rótulo “10”, temos que a primeira operação é a visita do nó, isto é, a impressão do seu rótulo.

Impressão: 10,

Em seguida, visita-se recursivamente a subárvore esquerda, cuja raiz é “8”, e a primeira operação é a visita do nó com rótulo “8”.

Impressão: 10, 8,

Após a visita do nó “8”, percorre-se recursivamente a subárvore esquerda do nó “8”, cuja raiz é o nó com rótulo “2”. A primeira operação deste percurso é a visita, isto é, o rótulo “2” é impresso.

Impressão: 10, 8, 2,

O nó “2” é folha, isto é, não possui subárvores, assim, o percurso da subárvore esquerda do nó “8” é concluído. O próximo passo é percorrer a subárvore direita do nó “8”, cuja raiz é o nó “6”. O primeiro passo é a impressão (visita).

Impressão: 10, 8, 2, 6,

Observe que o nó “6” é folha, o que encerra o percurso da subárvore de raiz “6”. Assim, retornamos ao seu pai, o nó “8”, que é raiz da subárvore e que já teve seu percurso completo. O Pai de “8” é o nó “10” que já foi visitado e que já teve sua subárvore esquerda visitada.

O próximo passo é visitar sua subárvore direita em pré-ordem. A raiz da subárvore direita é “4”, que é folha, sendo assim, o percurso pré-ordem da árvore é:

Impressão: 10, 8, 2, 6, 4

COMENTÁRIO

Observe que, para cada nó, acessamos seu conteúdo três vezes, porém, somente uma dessas vezes corresponde à visita. No caso do percurso em pré-ordem, a primeira vez.

Graficamente, os momentos de acesso podem ser vistos na Figura 4.

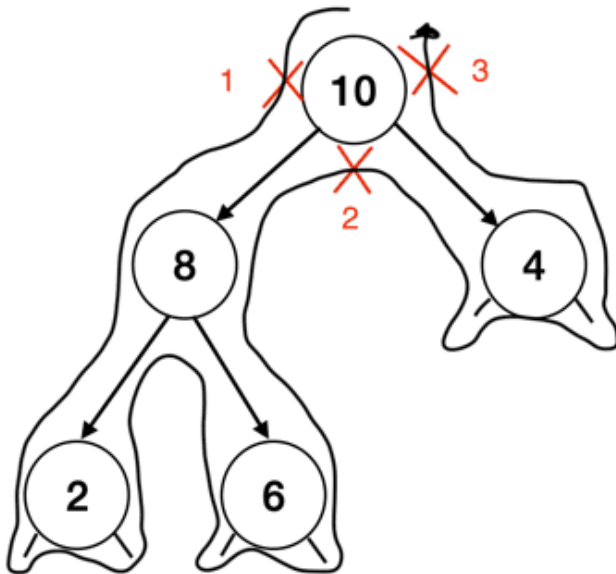


Figura 4 – Acesso ao nó.

Na Figura 4, considere o contorno gráfico da árvore a partir da raiz no sentido anti-horário.

Considerando o nó “10”, observe que temos a possibilidade de três acessos: O número “1”, que ocorre antes da descida para a subárvore esquerda.



O número “2”, que ocorre após o percurso completo do ramo esquerdo e antes do percurso do ramo direito.



Finalmente, o número “3”, que ocorre após o percurso dos ramos esquerdo e direito.

Observe que imprimindo o conteúdo do nó (operação de visita) sempre no acesso “1”, teremos o percurso em pré-ordem.

A Figura 5 ilustra esta situação.

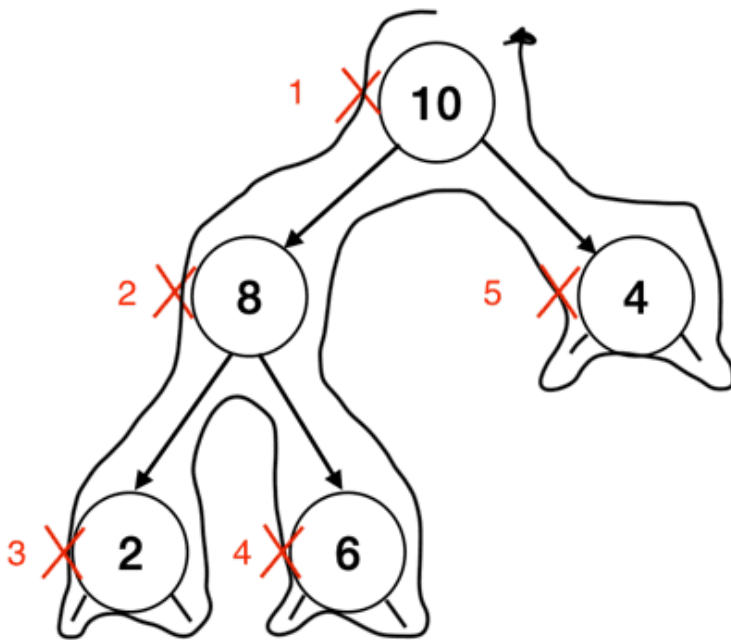
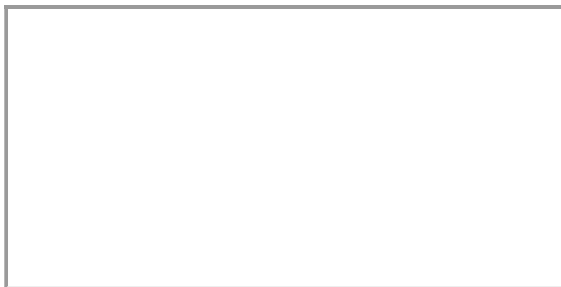


Figura 5 – Sequência de visitas no acesso “1”.

Observe que, na Figura 5, a sequência de acesso é 10, 8, 2, 6, 4. Exatamente o percurso em pré-ordem.

PERCURSO EM ORDEM SIMÉTRICA E PÓS-ORDEM



ALGORITMOS RECURSIVOS PARA OS PERCURSOS

Os algoritmos recursivos para os percursos em pré-ordem, ordem simétrica e pós-ordem são consequências diretas das definições dos percursos.

Veamos o algoritmo da pré-ordem.

função pre-ordem (registro no *p)

```

início
  visita (p);
  se (p->esq != NULO)
    pre-ordem (p->esq);
  se (p->dir != NULO)
    pre-ordem (p->dir);
fim

```

Algoritmo 1 – Percurso em pré-ordem.

Para realizar o percurso em pré-ordem, são necessários três acessos ao nó.

ETAPA 01

No caso da pré-ordem, no primeiro, executamos a visita.

ETAPA 02

No segundo, chamamos recursivamente o algoritmo para a subárvore esquerda.

ETAPA 03

No terceiro, ocorre a chamada do percurso em pré-ordem do ramo direito.

Se temos n nós em uma árvore, o número de acesso ao nó é $3n$.

Assim, a complexidade computacional do percurso em pré-ordem é $O(n)$.

O algoritmo do percurso em ordem simétrica é semelhante, modificamos somente o momento da visita, resultando no Algoritmo 2.

```

função simetrica (registro no *p)
início
  se (p->esq != NULO)
    simetrica (p->esq);
  visita (p);
  se (p->dir != NULO)
    simetrica (p->dir);
fim

```

Algoritmo 2 – Percurso em ordem simétrica.

A análise de complexidade é análoga a feita no algoritmo do percurso em pré-ordem. Observe que a única diferença é a ordem das visitas.

Sendo assim, a complexidade computacional do algoritmo para percurso em ordem simétrica é $O(n)$.

Finalmente, temos o algoritmo para o percurso em pós-ordem (Algoritmo 3), que é resultado direto da definição como o da pré-ordem e o da ordem simétrica.

```
função pos-ordem (registro no *p)
início
    se (p->esq != NULO)
        pos-ordem (p->esq);
    se (p->dir != NULO)
        pos-ordem (p->dir);
    visita (p);
fim
```

Algoritmo 3 – Percurso em pós-ordem.

A análise da complexidade é totalmente análoga à análise feita para pré-ordem e a ordem simétrica, o que faz com que o algoritmo tenha complexidade $O(n)$.

ALGORITMO NÃO RECURSIVO PARA PERCURSO

A ideia para o algoritmo não recursivo para os percursos em pré-ordem, ordem simétrica e pós-ordem é executar o “**contorno**” da árvore, como fizemos no ábaco apresentado na definição do percurso.

Como já vimos, nas árvores binárias, temos referência do pai para o filho, porém não há referência do filho para o pai.



Para guardar esta referência, utilizaremos uma pilha. Nesta pilha, guardaremos o nó visitado e o momento do acesso “1”, “2” ou “3”.

De acordo com o momento e com o percurso desejado, realizaremos as operações para obter a sequência do percurso.

Vejamos, então, a ideia principal do algoritmo.

```
função pre-ordem-não-recursiva (registro no *p)
início
    registro no *aux;
    inteiro momento;
    push (p,1); //insere a raiz de T, no momento 1
```

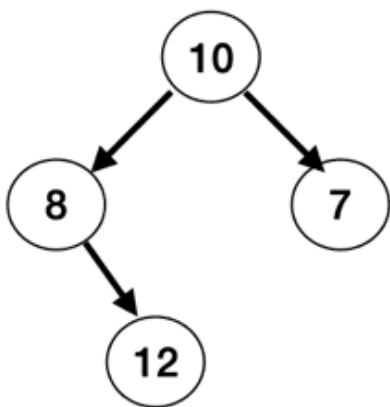
```

enquanto (pilha não vazia)
  inicio
    pop (aux,momento);
    if (momento == 1)
      inicio
        visitar (aux); //caso se deseje a pré-ordem
        push (aux,2); //push aux na pilha, momento 2
        se (aux->esq != NULO)
          push (aux->esq,1);
        fim
      if (momento == 2)
        inicio
          push (aux,3); //push aux na pilha, momento 3
          se (aux->dir != NULO)
            push (aux->dir,1);
          fim
        // na pré-ordem, não há ações no momento 3
      fim
    fim
  fim
fim

```

Algoritmo 4 – Algoritmo de pré-ordem não recursivo.

Para verificarmos o funcionamento do algoritmo, analisemos um exemplo e vejamos como os três momentos do ábaco de “contorno” da árvore ocorre. Para tal, veja a árvore da Figura 6.



📷 Figura 6 – Árvore exemplo.

O algoritmo é inicializado com a pilha vazia. O primeiro passo é inserir a raiz “10” na pilha no momento 1.

Sendo assim, o conteúdo da pilha é:

Pilha: (10,1)

Como a pilha não é vazia, o algoritmo entra em seu laço principal, removendo o nó 10 e o momento “1” da pilha. Como o momento é igual a “1”, visita-se o nó imprimindo seu conteúdo e insere-se (10,2) e

(8,1), nesta ordem, na pilha. O par (8,1) está no topo da pilha.

Assim, a impressão do percurso e o conteúdo da pilha é:

Impressão 10,

Pilha (10,2), (8,1)

Com a repetição, remove-se o par do topo da pilha (8,1). De forma análoga ao primeiro laço do algoritmo, visita-se o nó “8” e inserimos (8,2) na pilha.

Como o nó “8” não tem filho esquerdo, nada mais ocorre, resultando em:

Impressão 10, 8,

Pilha (10,2), (8,2)

No novo laço, remove-se o par (8,2) da pilha. O acesso ao nó está no momento “2” e existe nó à direita de “8”, por isso insere-se (8,3) e (12,1) na pilha, resultando em:

Impressão 10, 8,

Pilha (10,2), (8,3), (12,1)

Na nova iteração, remove-se o par (12,1). Como é o momento “1”, visita-se o nó “12”, inserindo (12,2) na pilha somente (“12” é folha), resultando em:

Impressão 10, 8, 12,

Pilha (10,2), (8,3), (12,2)

Na próxima iteração, remove-se o par (12,2), como “12” é folha, somente (12,3) é inserido na pilha.

Impressão 10, 8, 12,

Pilha (10, 2), (8, 3), (12,3)

No próximo loop, remove-se (12,3) da pilha e nada ocorre, resultando em:

Impressão 10, 8, 12,

Pilha (10, 2), (8, 3),

Na próxima iteração, remove-se (8,3) da pilha e nada ocorre, resultando em:

Impressão 10, 8, 12,

Pilha (10, 2),

Na próxima iteração, remove-se (10,2) da pilha. Como “10” tem filho direito, será inserida na pilha a sequência de inserções de (10,3) e (7,1), resultando em:

Impressão 10, 8, 12,

Pilha (10, 3), (7, 1)

Próximo laço de repetição, remove-se (7,1). Como é o acesso no momento “1”, visita-se o nó “7”, insere-se na pilha (7,2) somente uma vez que o nó “7” não tem filho esquerdo, resultando em:

Impressão 10, 8, 12, 7

Pilha (10, 3), (7,2),

No próximo passo, remove-se (7,2) e insere-se (7,3) na pilha. Como “7” não tem filho direito, o passo termina, resultando em:

Impressão 10, 8, 12, 7

Pilha (10, 3), (7,3),

No próximo laço, remove-se (7,3), resultando em:

Impressão 10, 8, 12, 7

Pilha (10, 3),

No próximo laço, remove-se a dupla (10,3), resultando em:

Impressão 10, 8, 12, 7

Pilha vazia.

Finalizando o algoritmo. **Observe que a sequência de impressão é o percurso em pré-ordem.**

Para analisar a complexidade deste algoritmo, vamos verificar que cada nó da árvore T é inserido **três vezes** na pilha.

Se temos n nós na árvore T , a complexidade é de **$3n$ operações elementares**, o que resulta em $O(n)$.

Se analisarmos a execução deste exemplo, vamos perceber que, para o percurso em pré-ordem, a inclusão do momento “3” na pilha é dispensável (pode-se remover a linha push (aux,3) do Algoritmo 4).

Trata-se de uma otimização do algoritmo, mas não altera sua complexidade computacional.

DICA

Para obter o algoritmo não recursivo do percurso em ordem simétrica e pós-ordem, basta modificar o ponto de chamada da função visita.

Na ordem simétrica, a visita é feita no momento “2” e na pós-ordem, no momento “3”, resultando os algoritmos 5 e 6.

função ordem-simetrica-não-recursiva (registro no *p)

```

inicio
    registro no *aux;
    inteiro momento;
    push (p,1); //insere a raiz de T, no momento 1
    enquanto (pilha não vazia)
        inicio
            pop (aux,momento);
            if (momento == 1)
                inicio
                    push (aux,2); //push aux na pilha, momento 2
                    se (aux->esq != NULO)
                        push (aux->esq,1);
                fim
            if (momento == 2)
                inicio
                    visitar (aux);
                    se (aux->dir != NULO)
                        push (aux->dir,1);
                fim
            // na ordem simétrica, não há ações no momento 3
        fim
    fim
fim

```

Algoritmo 5 – Algoritmo de ordem simétrica não recursivo.

```

função pos-ordem-não-recursiva (registro no *p)
inicio
    registro no *aux;
    inteiro momento;
    push (p,1); //insere a raiz de T, no momento 1
    enquanto (pilha não vazia)
        inicio
            pop (aux,momento);
            if (momento == 1)
                inicio
                    push (aux,2); //push aux na pilha, momento 2
                    se (aux->esq != NULO)
                        push (aux->esq,1);
                fim
            if (momento == 2)
                inicio
                    push (aux,3);
                    se (aux->dir != NULO)
                        push (aux->dir,1);
                fim
            if (momento == 3)
                visitar (aux);
        fim
    fim

```

fim

fim

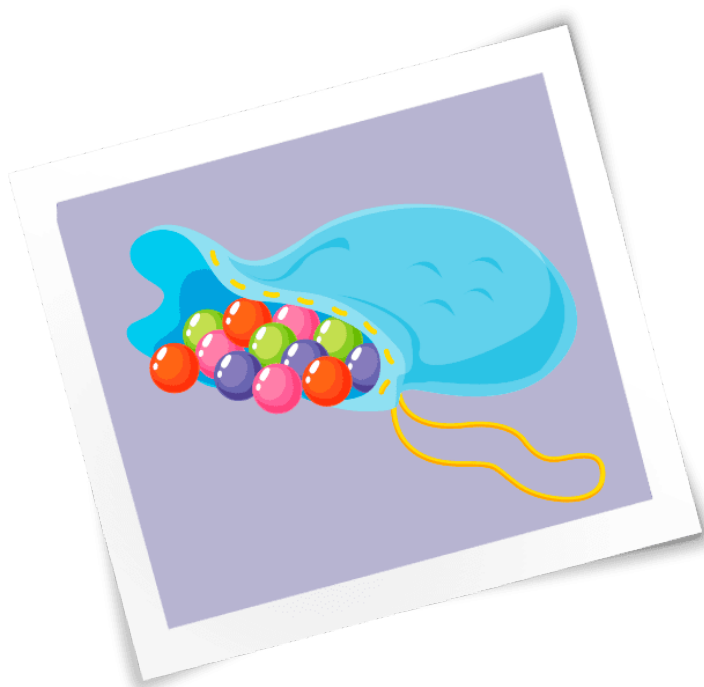
Algoritmo 6 – Algoritmo pós-ordem não recursivo.

VERIFICANDO O APRENDIZADO

MÓDULO 2

🕒 Definir árvores binárias de busca

O problema da busca, inserção e remoção é um dos principais objetivos do estudo de estruturas de dados.



Esta situação lúdica nos remete a algumas conclusões. Para tal, vamos relembrar alguns conceitos.

No estudo de complexidade, sempre levamos em consideração o pior caso, isto é, a análise de complexidade é feita com a visão do pessimista.



A situação lúdica nos mostra que, na visão do pessimista, sem organização alguma, teremos que tirar n bolinhas do saco para encontrar a bolinha vermelha.

Isso mostra que, sem nenhuma organização da informação, é possível realizar uma busca com complexidade computacional de $O(n)$.

COMENTÁRIO

Outra conclusão importante é que, para melhorar a busca, isto é, para conseguir um algoritmo mais eficiente, precisamos perseguir a complexidade de $O(\log n)$. As árvores são as estruturas de dados que vão viabilizar este objetivo.

ÁRVORES BINÁRIAS DE BUSCA

Seja $S = \{s_1, s_2, \dots, s_n\}$ um conjunto de chaves que serão buscadas, inseridas e removidas de uma estrutura de dados, tais que, $s_1 < s_2 < \dots < s_n$.

Observe que esta restrição, $s_1 < s_2 < \dots < s_n$, implica que as chaves são **distintas**.

Uma árvore binária de busca T é a árvore binária que satisfaz as seguintes condições:

Existe uma função $r(v): V \rightarrow S$, onde V é o conjunto de nós da árvore T .



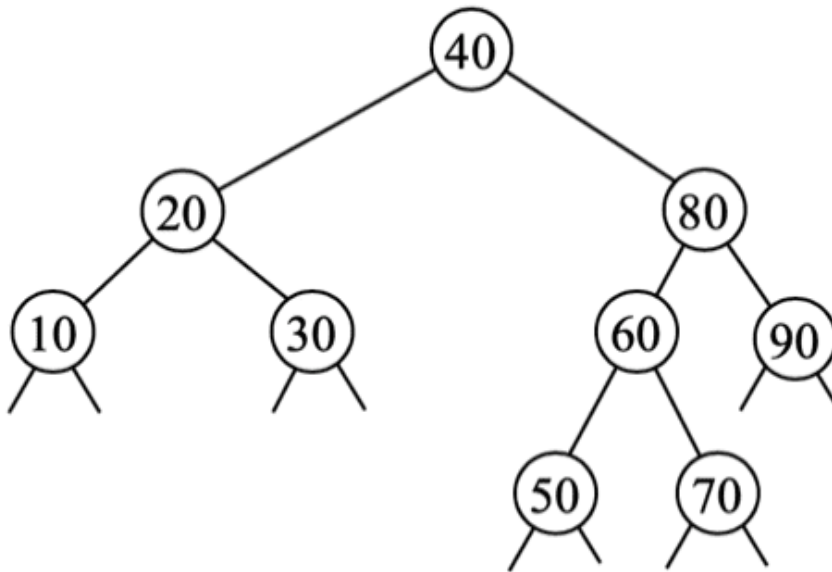
Para todo nó v de T e para todo v_i pertencente a subárvores esquerda de v , temos que $r(v_i) < r(v)$.



Para todo v nó de T e para todo v_k pertencente a subárvores direita de v , temos que $r(v_k) > r(v)$.

Em uma linguagem simples, para todo os nós de uma árvore binária de busca, as chaves contidas nos nós da subárvore esquerda devem ser menores que a chave contida na raiz considerada; e as chaves contidas nos nós da subárvore direita devem ser maiores que a chave contida na raiz considerada.

A Figura 7 ilustra uma árvore binária de busca.



📷 Figura 7 – Árvore Binária de Busca.

ALGORITMO DE BUSCA

O algoritmo de busca decorre diretamente da definição. Seja x a chave que desejamos localizar.

Compara-se x com a raiz, se x está na raiz, o algoritmo de busca para caso contrário, se x é menor que a raiz, executa-se o algoritmo recursivamente na subárvore esquerda, caso contrário, na subárvore direita.

O algoritmo de busca é utilizado nas operações de inserção e de remoção de nós nas árvores binárias de busca. Portanto, são necessários alguns parâmetros aparentemente supérfluos. O primeiro é um indicador booleano de que a chave foi encontrada; o segundo é uma referência para o pai do nó que contém a chave encontrada pelo algoritmo.

Por exemplo, na árvore da Figura 7, ao buscar-se a chave “80”, o algoritmo irá retornar:

ETAPA 01

A referência para o nó que contém a chave “80”.

ETAPA 02

A referência para o nó que contém a chave “40”, que é o nó pai do nó que contém a chave “80”.

ETAPA 03

O booleano verdadeiro, indicando que a chave buscada foi encontrada.

Quando buscamos uma chave não armazenada na estrutura, o algoritmo irá retornar o nó onde a busca tornou-se inviável, isto é, o nó cuja subárvore que deveria ser explorada no próximo passo da busca é

vazio.

★ EXEMPLO

Por exemplo, na árvore da Figura 7, ao buscar-se a chave 25, percorreríamos o seguinte caminho: 40, 20, 30. Ao analisar o nó que contém a chave “30”, concluiríamos que teríamos que proceder na busca pela sua subárvore esquerda, uma vez que $25 < 30$. Porém, a subárvore esquerda do nó que contém a chave “30” é vazia. Portanto, a busca retornaria:

- 1) A referência para o nó que contém a chave “30”.
- 2) A referência para o nó que contém a chave “20”, pai do nó que contém a chave “30”.
- 3) O booleano falso, indicando que a chave não foi encontrada.

Na busca, existem algumas situações especiais. A primeira ocorre quando a chave buscada está na raiz. Neste caso, o nó que contém a chave não tem pai, por isso, o algoritmo de busca deverá retornar NULO na referência para o pai.

Outra situação especial ocorre quando é realizada uma busca numa árvore vazia.

Neste caso, o algoritmo deverá retornar:

- 1) NULO para o nó que contém a chave.
- 2) NULO para referência do nó que é pai do nó que contém a chave buscada.
- 3) O booleano FALSO.

O pseudocódigo do Algoritmo 7 reflete todos os critérios estabelecidos.

```
função busca (inteiro chave, registro no *raiz; ref registro no *pai, ref booleano
início
    se (raiz == NULO)
        início
            encontrada = FALSO;
            retornar NULO;
        fim
    senão se (raiz->chave == chave)
        início
            encontrada = VERDADEIRO;
            retornar raiz;
        fim
    senão se (chave < raiz->chave)
        início
```

```

se (raiz->esq != NULO)
    inicio
        pai = raiz;
        retornar busca (chave, raiz->esq, pai, encontrada);
    fim
senão
    inicio
        encontrada = FALSO;
        retornar raiz;
    fim
fim
senão
    inicio
        se (raiz->dir != NULO)
            inicio
                pai = raiz;
                retornar busca (chave, raiz->dir, pai, encontrada);
            fim
        senão
            inicio
                encontrada = FALSO;
                retornar raiz;
            fim
        fim
    fim
fim

```

Algoritmo 7 – Busca em uma árvore binária de busca.

Para fins de análise de funcionamento do algoritmo, considere a árvore da Figura 8.

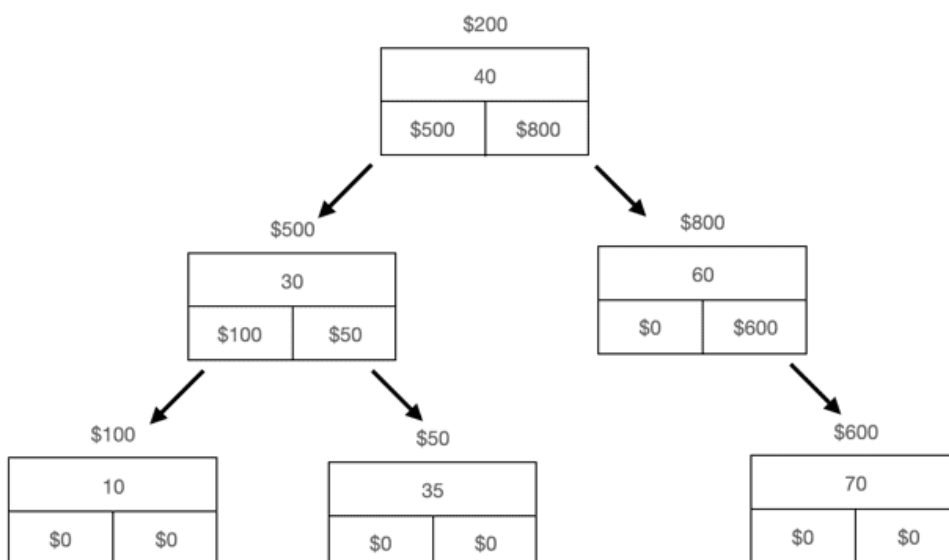


Figura 8 – Árvore Binária de Busca.

A raiz da árvore é o nó de endereço \$200, ou seja, o ponteiro raiz armazena o endereço \$200. Portanto, para realizar a busca, é necessário termos no escopo da função que chama a busca os ponteiros raiz e pai, inicializado com NULO, e o booleano encontrado, inicializado com FALSO.

O trecho de código a seguir mostra como seria a chamada da função busca. A variável **aux**, que é um ponteiro para um nó, vai receber o resultado da busca.

```
pai = NULO;  
encontrado = FALSO;  
aux = busca (30, raiz, pai, encontrado);
```

ATENÇÃO

Pai e **encontrado** são parâmetros passados por referência, isto é, sofrem efeito colateral de sua alteração no escopo da função busca. Ou seja, ao término da busca, se estas variáveis forem alteradas no escopo da função busca, o valor armazenado no escopo da função que chama a função busca também é alterado.

A função busca é recursiva, sendo assim, em sua primeira ativação, temos:

raiz=\$200, pai=NULO, encontrado=FALSO.

Executando o código, temos que 30 < raiz -> chave (40). Assim, fazemos pai=raiz (\$200) e chamamos recursivamente a função na chamada abaixo:

retornar busca (chave, raiz->esq, pai, encontrada);

Nesta segunda ativação raiz=\$500 (raiz->esq no escopo da primeira ativação da busca), pai=\$200.

Executando o código, verifica-se que a chave buscada está contida no nó \$500 (valor do ponteiro raiz nesta ativação), assim a função busca retorna o endereço \$500 para a primeira ativação que retorna o valor \$500 e faz a variável encontrada=VERDADEIRO para o escopo da chamada original da busca.

Ou seja, após o término da execução das ativações recursivas da função busca, temos que: aux=\$500, pai=\$200 e encontrada=VERDADEIRO.

ANÁLISE DE COMPLEXIDADE DA BUSCA

Sabemos que a análise de complexidade é baseada na identificação do pior caso e a análise do número de operações elementares resolve este caso. O pior caso é, sem dúvida, não encontrar a chave buscada.

Neste caso, o algoritmo irá realizar uma comparação por nível até encontrar um nó que não possua o filho onde a chave buscada poderia estar.

Por exemplo, na árvore da Figura 9, caso fosse realizada a busca da chave “75”, o algoritmo de busca iria percorrer a árvore na seguinte sequência: 40, 80, 60, 70. No nó que contém a chave “70”, o algoritmo tentaria descer no ramo direito, porém este ramo não existe, interrompendo a busca.

Observe que, neste caso, o número de comparações, que é a operação elementar da busca, é proporcional à altura da árvore.

Sendo assim, para determinarmos a complexidade da busca em uma árvore binária de busca, devemos determinar qual a árvore binária de busca de maior altura.

Pela definição de árvore binária de busca, não há restrição para a altura da árvore, sendo assim, uma árvore com topologia “zig-zag”, que são árvores onde cada nó só possui um filho, pode ser uma árvore binária de busca. Entretanto, árvores “zig-zag” possuem altura proporcional à n . Assim, a complexidade da busca em uma árvore binária de busca é $O(n)$.

INSERÇÃO DE UM NOVO NÓ EM UMA ÁRVORE BINÁRIA DE BUSCA

A inserção de uma nova chave em uma árvore binária de busca ocorre sempre em um novo nó posicionado como uma nova folha da árvore. Isto decorre do fato de que a posição do nó que contém a nova chave é determinada pela busca desta chave na árvore.

Já vimos que estruturas de dados onde realizamos busca, inserção e remoção não permitem chave duplicada. Assim, a busca pela chave que desejamos inserir na árvore deverá, obrigatoriamente, falhar e retorna como resultado o nó que será pai do novo nó inserido na árvore.

★ EXEMPLO

Por exemplo, na árvore da Figura 9, para inserir a chave “45”, devemos realizar a busca desta chave na estrutura. A busca percorre o caminho: 40, 80, 60, 50. Retornando como resultado uma referência para o nó que contém a chave “50” e a variável **encontrada** como FALSA. Como a busca parou no nó que contém a

chave “50”, é sinal que não foi possível continuar pelo seu filho esquerdo (o nó “50” não possui filho esquerdo), assim, a posição onde o novo nó contendo a chave “45” deve ser inserida é a esquerda do nó que contém a chave “50”.

O algoritmo de inserção de uma nova chave em uma árvore binária de busca é consequência direta do algoritmo de busca. Toda vez que tratamos a inserção de novas chaves em estruturas de dados que admitem busca, inserção e remoção, é interessante analisar os possíveis casos especiais. Neste caso, a árvore vazia é um caso especial, uma vez que, se inserimos uma nova chave numa árvore vazia, o nó que contém esta nova chave será a nova raiz desta árvore.

```
função inserir (inteiro chave, ref registro no *raiz): booleano
inicio
    registro no *pai;
    registro no *aux;
    registro no *novono;
    booleana encontrada;

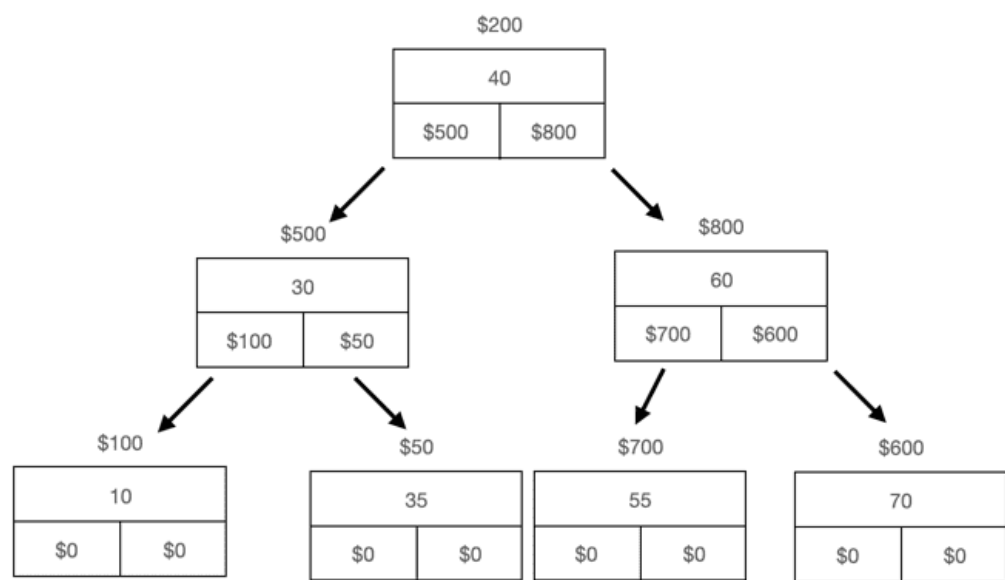
    aux = busca (chave, raiz, pai, encontrada);
    if (!encontrada)
        inicio
            alocar (novono);
            novono->chave = chave;
            novono->esq = NULO;
            novono->dir = NULO;
            if (aux == NULO)
                raiz = novono;
            else if (chave < aux->chave)
                aux->esq = novono;
            else
                aux->dir = novono;
            retorne VERDADEIRO;
        fim
    else
        retorne FALSO;
    fim
```

Algoritmo 8 – Inserção de uma nova chave.

★ EXEMPLO

Se chamarmos a função inserir passando a raiz da árvore da Figura 8, com os parâmetros inserir (55, raiz); ocorreriam os seguintes passos: Chamaríamos a busca da chave “55” na árvore com nó raiz o nó de endereço \$200 que armazena a chave “40”. Como “55” é maior que “40”, chama-se recursivamente a busca para o nó de endereço \$800 que armazena a chave “60”. A nova chave é menor que “60”, porém o ramo esquerdo do nó armazenado no endereço \$800 é vazio, fazendo com que a busca termine, retornando o endereço \$800 e encontrado = FALSO.

Seguindo o algoritmo da inserção, alocamos espaço para o novo nó e fazemos aux->esq = novo nó, resultando na árvore da Figura 9.



📷 Figura 9 – Árvore após inserção do novo nó com chave “55”.

ANÁLISE DA COMPLEXIDADE DA INSERÇÃO

A principal operação para realização da inserção de um novo nó é a busca. A busca determina a posição e se é possível ou não realizar a inserção. Após a busca, as operações que se seguem são todas realizadas em $O(1)$.

Sendo assim, a complexidade da inserção é a complexidade da busca que é $O(n)$.

REMOÇÃO

A análise da remoção será feita com base em três subcasos:

ETAPA 01

A remoção de uma folha da árvore binária de busca.

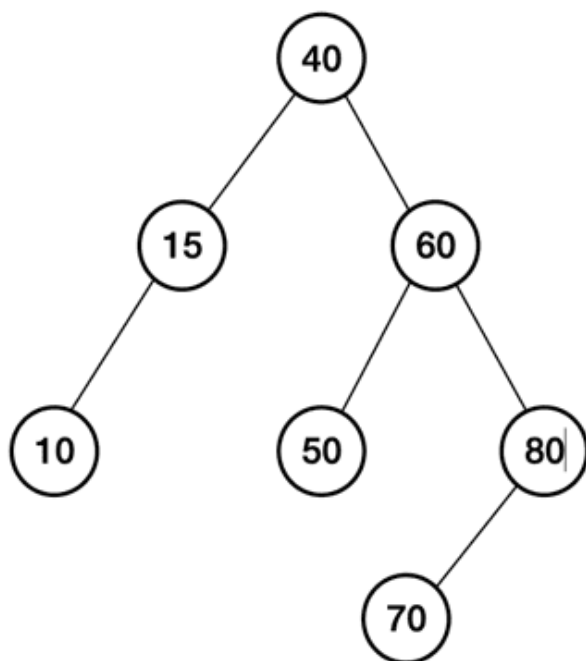
ETAPA 02

A remoção de um nó interno com 1 filho somente.

ETAPA 03

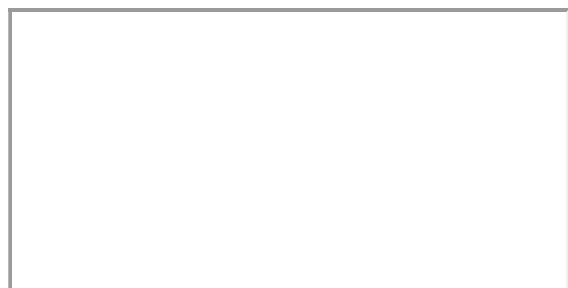
A remoção de um nó interno com dois filhos.

Vamos considerar a árvore binária de busca da Figura 10 como exemplo para fins de análise.



📷 Figura 10 – Árvore binária de busca.

ANÁLISE DA REMOÇÃO DE UM ELEMENTO



Observe que o algoritmo da remoção deverá chamar ele mesmo para a remoção do nó substituto de uma de suas subárvores. Isto não quer dizer que o algoritmo da remoção seja recursivo. Além disso, é necessário utilizar um indicativo na chamada da remoção, informando que será removido um nó substituto. Lembre-se de que não sabemos o valor da chave deste nó. Após todas estas análises, podemos escrever o algoritmo da remoção (Algoritmo 9).

```

função remover (inteiro chave; ref registro no *raiz; booleano exato):
registro no *
início
    registro no *pai;
    registro no *aux;
    registro no *aux2;
    booleano encontrado;

    aux = busca (chave, raiz, pai, encontrado);
    if (aux == NULO)
        retornar FALSO; // Tentativa de remoção de uma árvore vazia
    senão se (!encontrado && exato)
        retornar FALSO; // Chave não encontrada
    senão
        início
            se (aux->dir == NULO && aux->esq == NULO)
                início
                    se (pai == NULO)
                        início
                            raiz == NULO; // A árvore era unitária, agora é vazia
                            retornar aux;
                        fim
                    senão
                        início
                            se (pai->dir == aux)
                                pai->dir = NULO;
                            senão
                                pai->esq = NULO;
                            retornar aux;
                        fim
                    fim
                senão se (aux->dir == NULO || aux->esq == NULO)
                    início
                        se (pai == NULO)
                            início
                                se (aux->dir == NULO)
                                    raiz = aux->esq;
                                senão
                                    raiz = aux->dir;
                            fim
                        senão
                            início
                                se (pai->dir == aux)
                                    início
                                        se (aux->dir == NULO)
                                            pai->dir = aux->esq;
                                        senão

```

```

        pai->dir = aux->dir;
    fim
senão
    início
        se (aux->dir == NULO)
            pai->esq = aux->esq;
        senão
            pai->esq = aux->dir;
    fim
fim
retornar aux;
fim
senão
    início
        aux2 = remover (chave, aux->dir, FALSO);
        aux2->dir = aux->dir;
        aux2->esq = aux->esq;
        se (pai == NULO)
            raiz = aux2;
        senão se (pai->dir == aux)
            pai->dir = aux2;
        senão
            pai->esq = aux2;
        retornar aux;
    fim
fim
fim

```

Algoritmo 9 – Remoção de um nó de uma árvore binária de busca.

ESTUDO DA COMPLEXIDADE DA REMOÇÃO

Conforme vimos no vídeo **Análise da remoção de um elemento**, a remoção é dependente da busca e vimos que a busca tem complexidade da $O(n)$. No caso 1, remoção de uma folha, a remoção depende somente da busca. Em seguida, realizamos operações elementares para remover o nó. Portanto, a complexidade do caso 1 é $O(n)$.

No caso 2, no pior caso, vamos remover um nó interno do penúltimo nível. Portanto, o custo computacional da busca é $O(n)$, uma vez que vamos percorrer $n-1$ nós para encontrar o nó que será removido. As operações de reapontamento são elementares. Portanto, a complexidade do caso 2 também é $O(n)$.

No caso 3, no pior caso, mais uma vez, a estrutura é próxima a uma árvore zig-zag. Haverá somente um nó com dois descendentes e este nó está no nível k . A primeira busca, para encontrar o nó que desejamos remover, executa k comparações, uma vez que este nó está no nível k . Em seguida, iremos procurar o substituto do nó no ramo zig-zag da estrutura, percorrendo-a até o nível $n-1$. Portanto, o custo da busca e dos reapontamentos é de $n-1$ comparações, logo $O(n)$.

VERIFICANDO O APRENDIZADO

MÓDULO 3

🕒 Definir o conceito de árvore balanceada

ÁRVORE BALANCEADA

No módulo anterior, estudamos uma estrutura de dados dinâmica, isto é, capaz de realizar busca, inserção e remoção de chaves, uma a uma, sem perder suas propriedades nem necessitar de processamento suplementar para manter suas características.

Entretanto, a estrutura de dados é complexa e não apresenta ganho em relação a uma massa de dados desorganizada, isto é, ambas, no pior caso, necessitam de $O(n)$ operações para busca, o que implica em $O(n)$ para inserção e remoção. Ou seja, a árvore binária de busca não é melhor em termos de performance computacional que uma lista sem organização alguma.

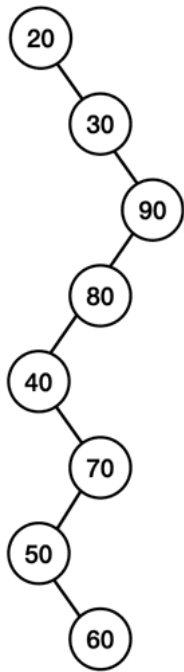
A pergunta que naturalmente surge é:

Então, por que estudar uma estrutura de dados mais complexa que uma lista, se sua complexidade computacional é a mesma?

A resposta para esta pergunta requer uma análise do estudo da complexidade computacional dos algoritmos de busca, inserção e remoção em árvores binárias de busca.

No módulo anterior, vimos que a complexidade computacional da busca é proporcional à altura da árvore, e que no pior caso, uma árvore tem altura n .

A família de árvores com altura n é chamada de **árvores zig-zag**. A Figura 11 mostra uma árvore zig-zag de altura 8.



📷 Figura 11 – Árvore zig-zag de altura 8.

Isto é, as árvores zig-zag são a família de árvores binárias de busca com pior desempenho possível, uma vez que a complexidade da busca e, conseqüentemente, da inserção e da remoção é $O(n)$ e não é possível construir uma árvore binária de busca com altura maior que n .


Se as zig-zag são as piores árvores, piores no sentido que a complexidade da manutenção da estrutura é a mesma de uma lista desorganizada, qual a família com melhor desempenho?

Para responder a esta pergunta, recordaremos alguns conceitos de árvore binárias.

Definição 1 – Diz-se que uma árvore binária T é completa se os nós de T com subárvores vazias estão no último ou no penúltimo nível.

★ EXEMPLO

Por exemplo, a árvore da Figura 12 é uma árvore completa. Os nós que possuem subárvores vazias estão no nível 3 e no nível 4, respectivamente, no penúltimo e no último nível da árvore.

 Figura 12 – Árvore binária completa.

Um resultado importante sobre as árvores binárias completas é o que se segue.

Lema – Seja T uma árvore binária completa com $n > 0$ nós. Então T possui altura mínima e $h=1+\lfloor \log n \rfloor$.

Este resultado é muito importante, pois mostra que não existe árvore binária com n nós com altura inferior à $h=1+\lfloor \log n \rfloor$. Ou seja, a melhor árvore binária que podemos construir tem altura proporcional a $\log n$.

A consequência deste resultado teórico é que se limitarmos a construção das árvores binárias de busca às árvores binárias completas, temos os algoritmos de busca, inserção e remoção funcionando em $\log n$. Este é o objetivo: melhorar a complexidade computacional da busca, inserção e remoção.

De uma forma ampla, dizemos que árvores binárias, cuja altura é proporcional a $\log n$, são árvores balanceadas.

Ou seja, árvores binárias completas são balanceadas, mas será que toda árvore balanceada é completa?

A resposta é não, existem outras famílias de árvores binárias, neste caso especificamente, de busca, que são balanceadas, porém não são completas, por exemplo, as árvores AVL e as árvores Rubro-Negras.

Antes de estudarmos as estruturas de dados completamente dinâmicas que fornecem árvores binárias de busca balanceadas, isto é, com altura de $O(\log n)$, estudaremos o algoritmo que transforma uma árvore binária de busca em uma árvore binária de busca completa.

Existem várias formas de se construir uma árvore binária de busca completa. A mais intuitiva é derivada da pesquisa binária. A pesquisa binária é um método de busca em um vetor ordenado que se baseia na estratégia **dividir para conquistar**.

A ideia é simples: compara-se a chave que está se buscando com a chave armazenada no elemento central do vetor, isto é, se o vetor tem tamanho n , o elemento $n/2$.



Caso a chave buscada seja menor que o elemento armazenado na posição $n/2$, aplica-se o método recursivamente na primeira metade do vetor, caso contrário, na segunda metade.

Este método de busca é eficiente, uma vez que é capaz de realizar a busca em $O(\log n)$.

Um exemplo de aplicação deste método pode ser visto na Figura 13, que ilustra a busca da chave “55”.

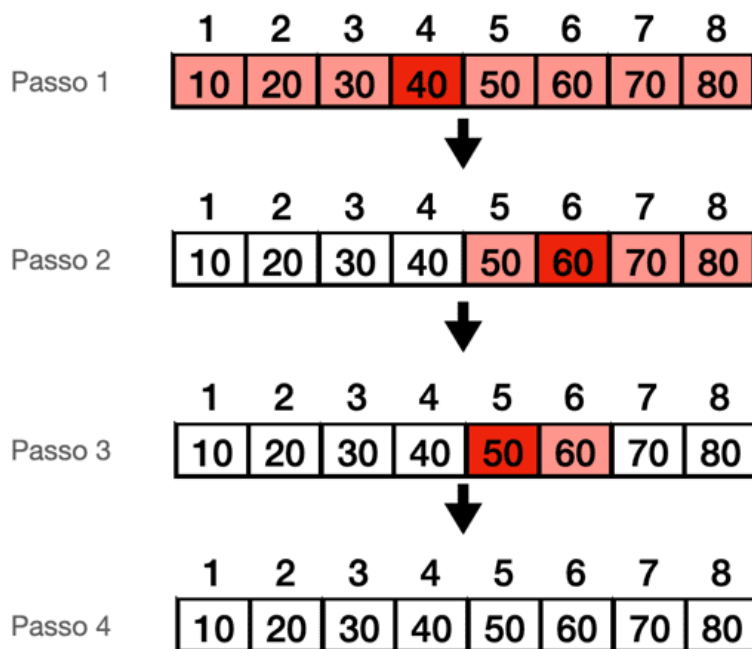


Figura 13 – Pesquisa binária de “55”.

No passo 1, compara-se a chave central $8/2 = 4$, que armazena a chave “40”, com a chave “55”. Assim, se “55” estiver na estrutura de dados, estará na segunda metade.

No passo 2, compara-se a chave central $(5+8)/2=6$, que armazena a chave “60”, com a chave “55”. Assim, se “55” estiver na estrutura, estará na primeira metade.

No passo 3, compara-se a chave central $(5+6)/2=5$, que armazena a chave “50”, com a chave “55”. Assim, se “55” estiver no vetor, estará na segunda metade, que é o elemento 6 do vetor, que não armazena a chave “55”, terminando, assim, a busca binária.

Como realizamos $O(\log n)$ comparações no pior caso, que é não encontrar a chave buscada, se transformarmos as comparações possíveis em uma árvore, teremos uma árvore com altura proporcional à $\log n$, ou seja, uma árvore balanceada.

A Figura 14 ilustra o processo para o mesmo vetor.

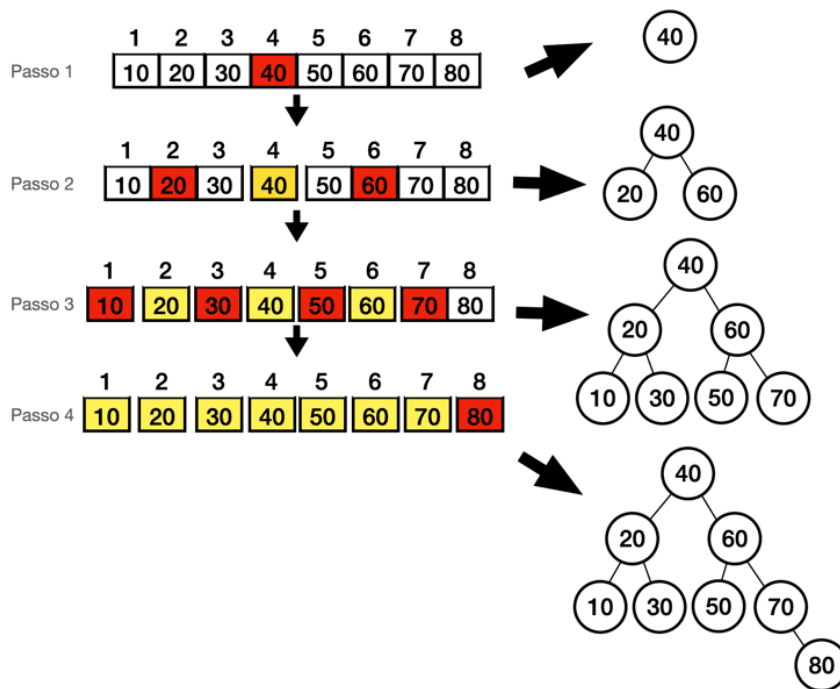


Figura 14 – Construção de uma árvore binária de busca balanceada.

A cada passo, elegemos uma raiz, elemento central, e aplicamos recursivamente o método nas metades esquerda e direita, que nos fornece as raízes das subárvores esquerda e direita recursivamente.

O método mostrado constrói uma árvore balanceada. uma vez que o número de níveis da árvore deriva do número de comparações na pesquisa binária, e este número é $\log n$.

Na verdade, podemos ver que construímos uma árvore binária de busca completa, que também é balanceada.

Este método de construção, apesar de intuitivo, possui diversas desvantagens.

Precisamos de um vetor auxiliar.



E a sequência de chaves ordenadas, o que, sem dúvida, aumenta a necessidade de alocação de memória.

O ideal é aplicar um algoritmo que resolva o problema de construir uma árvore binária de busca sem utilizar nenhuma estrutura de dados auxiliar.

O algoritmo que resolve este problema sem utilizar estruturas auxiliares é o Algoritmo DSW (Day-Stout-Warren). [2]

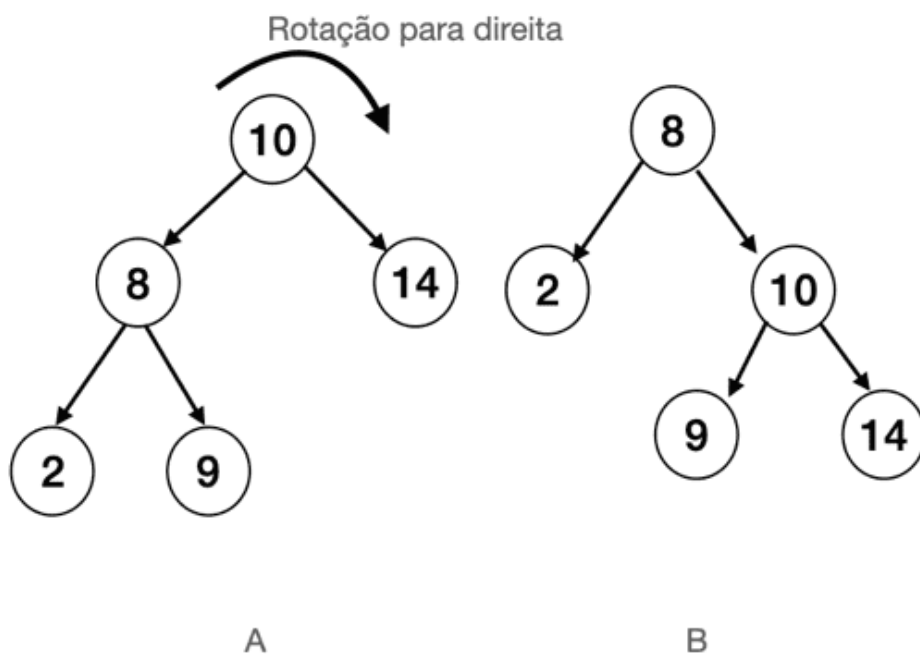
ALGORITMO DSW

O algoritmo DSW recebe como parâmetro uma árvore binária de busca, transformando-a em uma **árvore binária de busca completa em tempo linear**, isto é, com complexidade da $O(n)$.

O algoritmo executa em dois passos e a operação fundamental de cada passo é a operação de rotação. Uma rotação é uma operação que altera a hierarquia de elementos de uma árvore, isto é, transforma pai em filho e vice-versa, sem comprometer a propriedade básica das árvores binárias de busca.

O algoritmo DSW aplica duas rotações: **A rotação para a direita e para a esquerda**.

Para fins de análise desta operação, vamos analisar o exemplo da Figura 15.



📷 Figura 15 – Rotação para a direita.

Na Figura 15, aplicamos a rotação para a direita no nó “10”.

O resultado é a árvore de raiz “8”.



Observe que “8” era à esquerda de “10” e, após a rotação, “10” torna-se filho à direita de “8”.

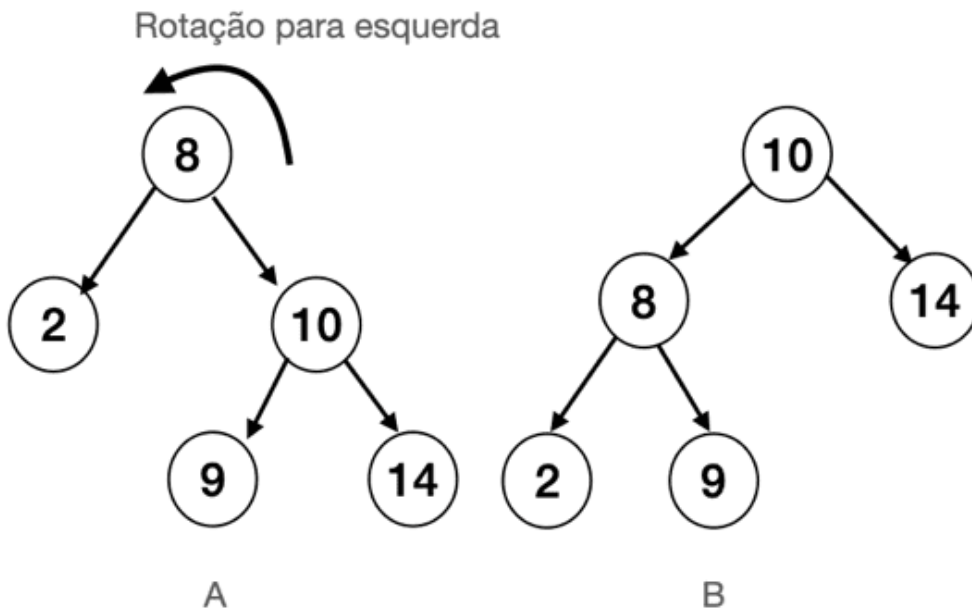
Além disto, observe que a propriedade fundamental das árvores binárias de busca é mantida, isto é, os nós contidos na subárvore esquerda de um determinado nó contém rótulo menor que a raiz e os contidos na subárvore direita, rótulo maior.

A aplicação da rotação à direita reduz a altura do ramo esquerdo, consequentemente aumentando a altura do ramo direito.



A rotação à esquerda é análoga à rotação à direita, claro, a principal consequência da rotação à esquerda é o aumento do ramo esquerdo.

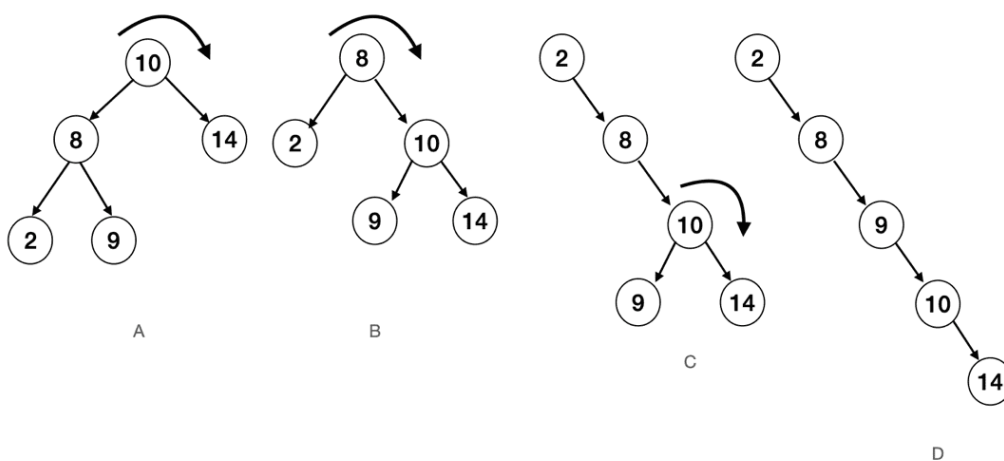
A Figura 16 ilustra a rotação para esquerda aplicada no nó “8”.



📷 Figura 16 – Rotação para a esquerda.

O primeiro passo do algoritmo é degenerar a árvore em uma “lista” ordenada. Isto é feito aplicando-se rotações para a direita, a partir da raiz, até não ser mais possível.

Ao final do processo, a árvore binária de busca será transformada em uma árvore binária de busca zig-zag onde todos os nós têm somente filhos à direita. A Figura 17 ilustra o processo.



📷 Figura 17 – Transformação da árvore.

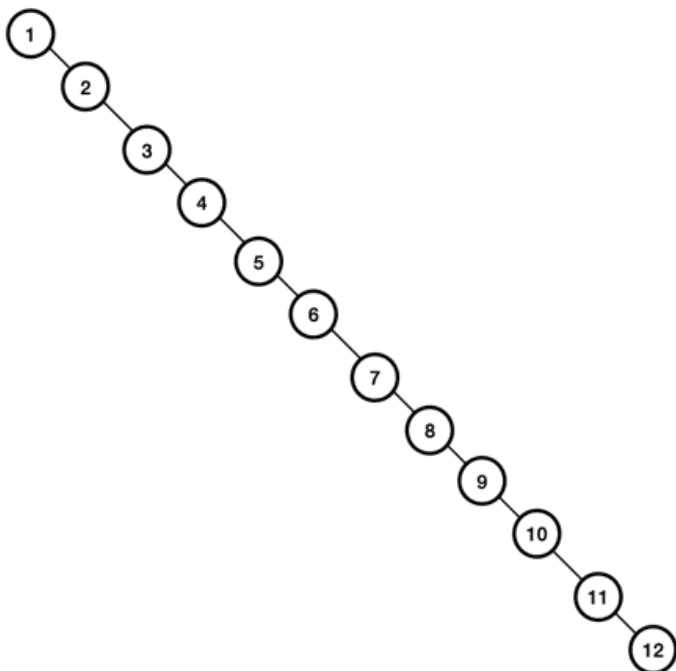
Na Figura 17, a raiz, o nó “10”, possui filho esquerdo, sendo assim, aplica-se a rotação para a direita na raiz, resultando na árvore da letra “B”. O nó “8” é a nova raiz da árvore, o nó “8” ainda possui filho à esquerda.

Sendo assim, aplica-se a rotação para a direita na nova raiz, resultando na árvore da letra “C”. A árvore da letra “C” possui raiz “2” que não possui filho esquerdo. O filho direito da raiz é o nó “8”, que também não possui filho esquerdo. Não há nenhuma operação que deva ser feita com este nó. Seu filho direito, o nó “10”, possui filho esquerdo. Sendo assim, aplica-se a rotação para a direita neste nó, resultando na árvore zig-zag da letra “D”.

O segundo passo do algoritmo é a construção da árvore completa. A construção é feita por níveis, das folhas para raiz.

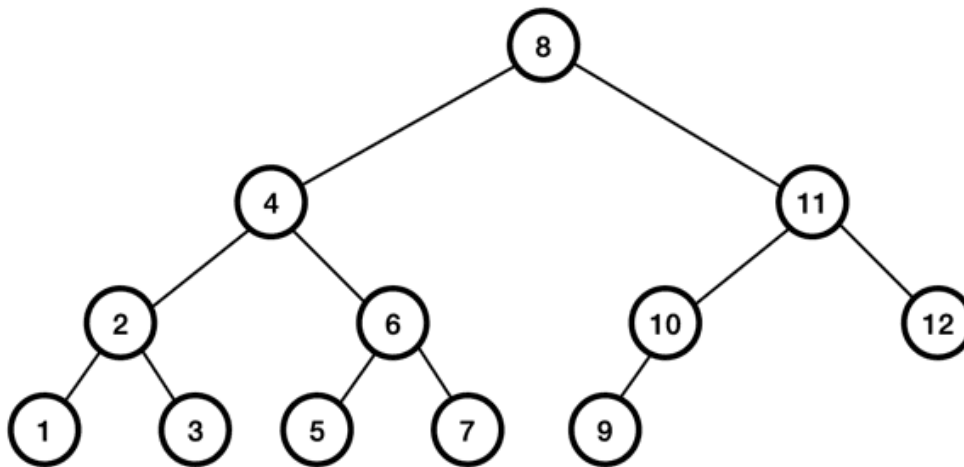
Para ilustrar o funcionamento do algoritmo, tomaremos como exemplo uma árvore com doze nós e com o conjunto de chaves $S=\{1,2,3,4,5,6,7,8,9,10,11,12\}$.

Não importa a topologia inicial da árvore, que após o passo 1 do algoritmo irá se transformar na árvore zig-zag da Figura 18.



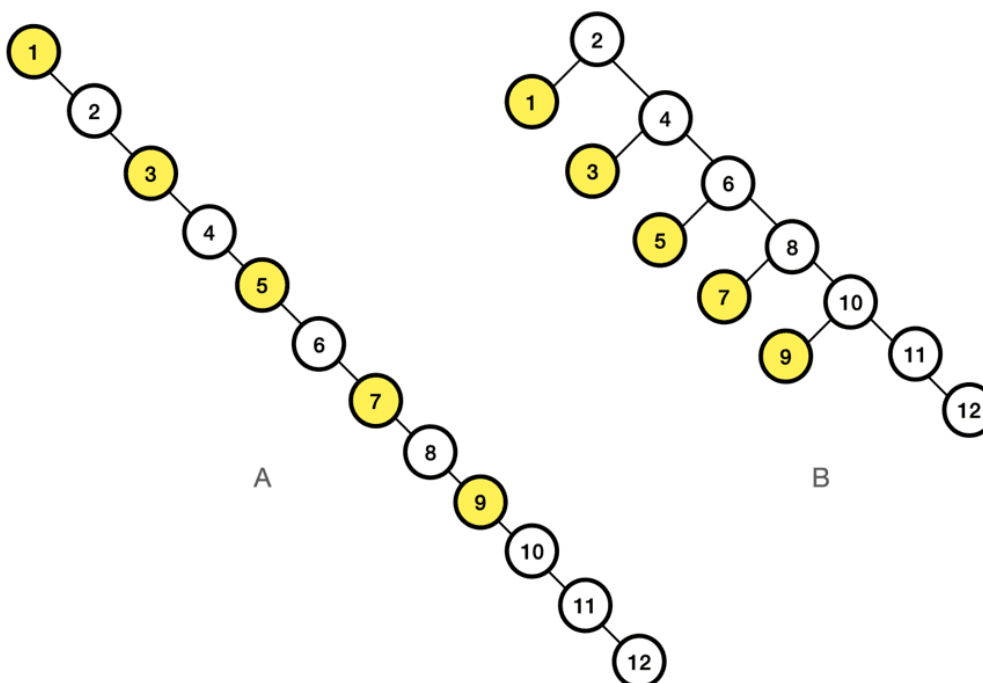
📷 Figura 18 – Árvore obtida após a execução do passo 2.

O objetivo é obter a árvore da Figura 19.



📷 Figura 19 – Árvore objetivo.

Observe que temos cinco nós no último nível da árvore, os nós “1”, “3”, “5”, “7” e “9”. Destacaremos estes nós na árvore zig-zag da Figura 20.



📷 Figura 20 – Aplicação da rotação nos nós folha.

Aplicando a rotação para a esquerda nos nós “1”, “3”, “5”, “7” e “9”, que são os nós do último nível da árvore, obtemos a árvore da Figura 20 (B). O processo de execução dessas rotações é simples. Basta executar cinco rotações na árvore da Figura 20 (A), a partir da raiz, e em seguida aplicar a rotação no filho esquerdo do nó rotacionado.

A Figura 21 ilustra o processo passo a passo. Destacadas em amarelo temos as folhas e são sobre essas folhas que aplicaremos as rotações.

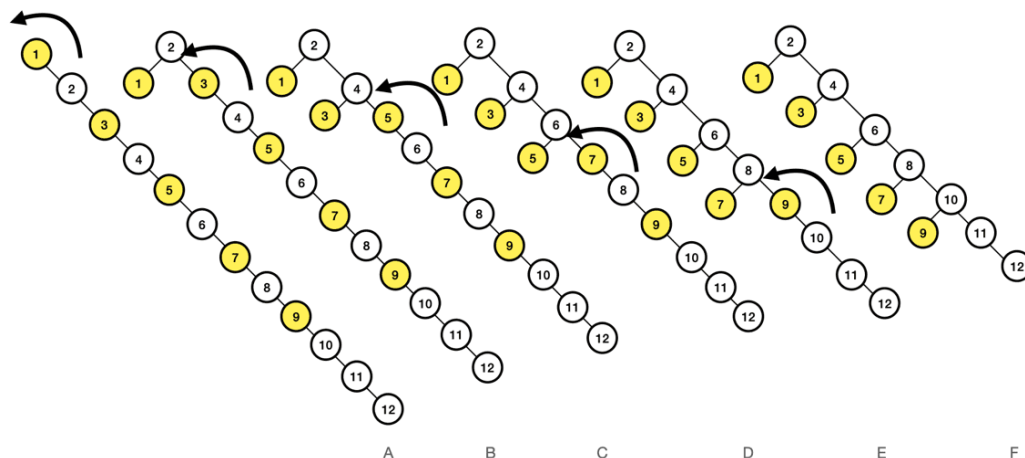


Figura 21 – Rotações para destaque das folhas do último nível.

Foram aplicadas cinco rotações, uma vez que temos como objetivo a árvore da Figura 21, que tem cinco nós no último nível. Entretanto, esta informação, a quantidade de nós no último nível, pode ser obtida aritmeticamente.

Sabe-se que uma árvore binária cheia (todos os nós com subárvores vazias no último nível) de altura h possui $n=2^h-1$ nós.

Sabe-se também que a altura de uma árvore binária completa é $h=1+\lfloor \log_2 n \rfloor$.

Assim, a quantidade k de nós no último nível é: $k=n-(2^{\lfloor \log_2 n \rfloor})+1$.

Na nossa árvore exemplo, temos que $\log_2 12 = 3,58$, assim, $k = 12 - (2^3) + 1 = 5$.

O $\log_2 n$ também traz a informação de quantos níveis temos na árvore, sabemos que o número de níveis é igual à altura da árvore, e a altura da árvore completa é $h=1+\lfloor \log_2 n \rfloor$. Sabemos também que no nível l temos $k = 2^{l-1}$ nós. Lembre-se de que a raiz tem nível “1”.

Voltando ao nosso exemplo, devemos agora evidenciar o penúltimo nível da árvore, que está no nível 3, lembre-se de que o nível 4 já foi evidenciado (em amarelo).

A Figura 22 mostra que para evidenciar o nível 3, devemos realizar as rotações dos nós na cor verde.

O nó “12” não recebe a rotação por ser o último nó do nível 3, o nó “12” não tem descendentes, mas, mesmo que tivesse, já estaria ajustado.

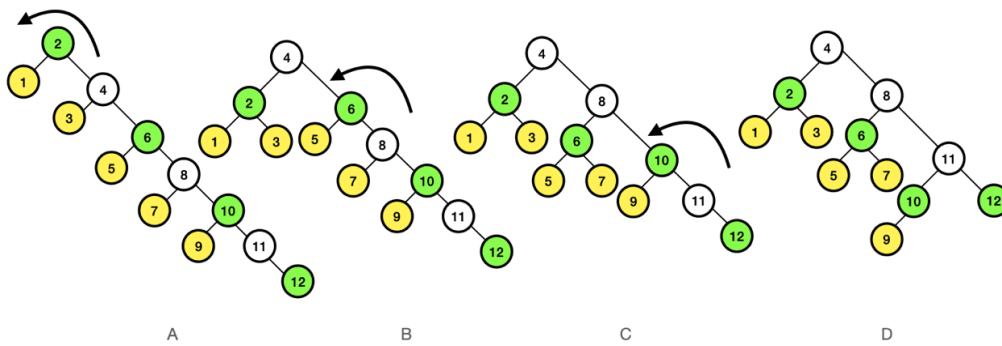


Figura 22 – Ajuste do nível 3.

Observe que as estruturas coloridas fazem parte da árvore da Figura 19. Além disso, realizamos $nr = 2^{3-1}-1=3$ rotações. Para ajuste do nível l , realizamos $nr = 2^{l-1}-1$ rotações.

O próximo passo é o ajuste do nível 2. Pela fórmula, realizaremos $nr=2^{2-1}-1=1$ rotações. Vejamos na Figura 23.

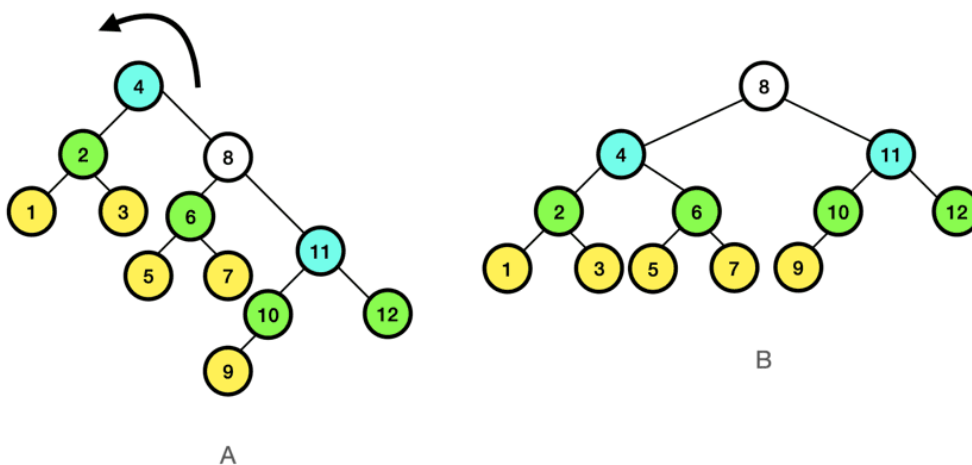


Figura 23 – Ajuste do nível 2.

O ajuste do nível 2 faz com que a árvore binária de busca seja completa e isto completa o algoritmo.

A seguir, apresentaremos o pseudocódigo dos passos 1 e 2 do algoritmo.

```
função rot_dir_dsw (ref registro no *r)
início
    registro no *aux= r->esq;
    r->esq = aux->dir;
    aux->dir = r;
    r = aux;
fim

função passo1_dsw (ref registro no *raiz)
início
    registro no **aux;
```

```

    aux = & raiz;

    enquanto (*aux != NULO)
        inicio
            se ((*aux)->esq != NULO)
                rot_dir_dsw (aux);
            senão
                aux = &((*aux)->dir);
        fim
    fim

```

O passo 2 é executado pelo algoritmo a seguir.

```

função rot_esq_dsw (ref registro no *r)
    inicio
        registro no *aux= r->dir;
        r->dir = aux->esq;
        aux->esq = r;
        r = aux;
    fim

função passo2_dsw (ref registro no *raiz, int n)
    inicio
        registro no **aux = & raiz;
        int alt, i, k;
        int nr_ult_nv;

        nr_ult_nv = n - (2^(int(log2(n)))+1);
        alt = 1 + int (log2(n));

        se (nr_ult_nv != 2^(alt-1))
            inicio
                para i=1 ate nr_ult_nv faça
                    inicio
                        rot_esq_dsw(aux);
                        aux = &((*aux)->dir);
                    fim
                alt --
            fim
        para i = alt ate 2 faça
            inicio
                aux = raiz;
                para k = 1 ate 2^(i-1)-1
                    inicio
                        rot_esq_dsw(aux);
                        aux = &((*aux)->dir);
                    fim
            fim
    fim

```

fim



VERIFICANDO O APRENDIZADO

MÓDULO 4

⦿ Definir árvores AVL

ÁRVORES AVL

Nos módulos anteriores, estudamos as árvores binárias de busca e seus algoritmos de manipulação dinâmica, isto é, algoritmos que são capazes de realizar busca, inserção e remoção, mantendo a propriedade fundamental das árvores binárias de busca.

Entretanto, ao estudar a complexidade destes algoritmos, observamos que a estrutura de dados, apesar de complexa, não tinha desempenho teórico superior às listas desorganizadas. Isto se deve ao fato de que, no pior caso, a complexidade das operações de busca, inserção e remoção é $O(n)$.

Em seguida, vimos que, para uma redução de complexidade, temos que ter árvores binárias de busca com altura proporcional a $O(\log n)$ e as árvores com esta propriedade são chamadas de árvores balanceadas.

Vimos também um algoritmo estático capaz de transformar qualquer árvore binária de busca em uma árvore balanceada em um tempo proporcional a $O(n)$.

A evolução deste cenário é a obtenção de uma estrutura de dados completamente dinâmica que suporte inserções, remoções e buscas em $O(\log n)$. As árvores AVL fornecem esta funcionalidade.

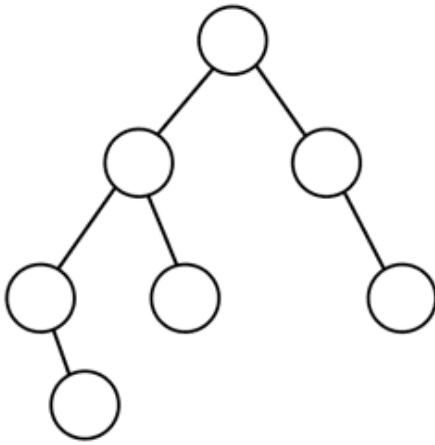
A sigla AVL corresponde às iniciais dos autores **Adelson-Velsky** e **Landis**.

Definição: Uma árvore binária de busca T é uma árvore AVL se, para qualquer nó r de T , vale a propriedade: $|h(T_r^e) - h(T_r^d)| \leq 1$.

Observação: $h(T_r^e)$ é a altura da subárvore esquerda de r e $h(T_r^d)$ é a altura da subárvore direita.

Dizemos que um nó r de uma árvore binária de busca está regulado se $|h(T_r^e) - h(T_r^d)| \leq 1$.

Ou seja, em uma árvore AVL, todos os nós estão regulados. A Figura 25 mostra o exemplo da topologia de uma árvore AVL.



📷 Figura 25 – Topologia de uma árvore AVL – exemplo.

ADELSON-VELSKY

Adelson-Velsky (1922-2014) foi originalmente educado como um matemático puro. Ele começou a trabalhar com inteligência artificial e outros tópicos aplicados no final dos anos 1950. Com Evgenii Landis, ele inventou a árvore AVL em 1962. Esta foi a primeira estrutura de dados de árvore de pesquisa binária balanceada conhecida.

LANDIS

Evgenii Mikhailovich Landis (1921-1997) Foi um matemático russo, de origem judaica, que dedicou suas pesquisas às equações diferenciais parciais.

As árvores completas são AVL. Este fato deriva da própria definição das árvores completas.

Além disso, sabemos que as árvores completas com n nós têm altura mínima, isto é, não existe topologia com n nós com altura inferior à da árvore completa ($h=1+\lceil \log n \rceil$).

Ou seja, este resultado garante que a altura mínima de uma árvore AVL é proporcional a $\log n$.

Entretanto, as árvores completas são o melhor caso, isto é, as árvores AVL de altura mínima. Vejamos a árvore AVL da Figura 25.

Esta árvore não é completa, uma vez que existe um nó na topologia que tem subárvore vazia e este nó está no antepenúltimo nível da árvore. Assim, o teorema que versa sobre a altura das árvores completas não garante que a árvore da Figura 25 tem altura proporcional à $\log n$.

Analisemos a família de árvores AVL com pior desempenho, isto é, dada uma quantidade n de nós a maior altura possível. Se esta família de árvores AVL tiver altura proporcional a $\log n$, então garantiremos que no melhor caso e no pior caso a altura de uma árvore AVL é proporcional a $\log n$.

Vamos construir a família de árvores AVL com altura máxima e menor número de nós possível. A construção será recursiva.

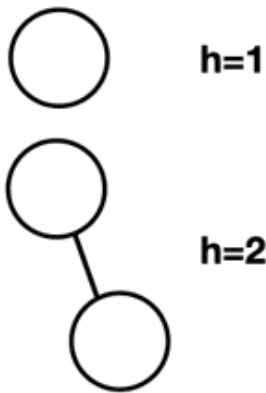
ETAPA 01

Para $h=1$ só existe uma solução, que é a árvore unitária.

ETAPA 02

Para $h=2$, podemos construir a árvore com $n=2$ nós, a raiz e um dos filhos, à esquerda ou à direita. Para fins de análise, não há diferença.

A Figura 26 mostra estas topologias.



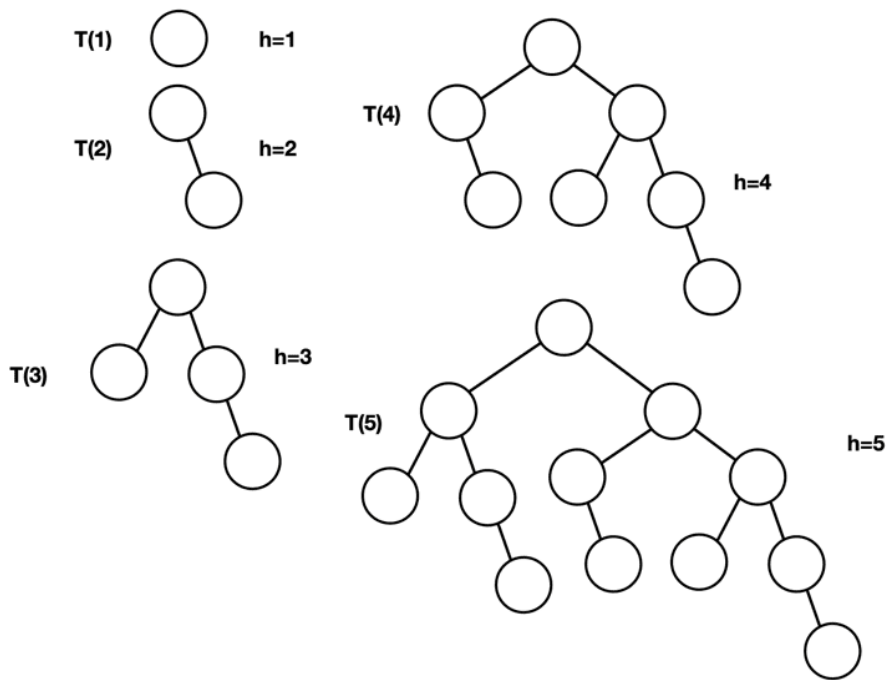
📷 Figura 26 – Árvores AVL com altura máxima e número mínimo de nós.

Para construir as árvores com altura h e número mínimo de nós, comporemos as árvores da seguinte forma recursiva:

1 – Arbitra-se uma nova raiz.

2 – Agrega-se a esta raiz como subárvores esquerda e direita as árvores de altura $h-1$ e $h-2$.

Assim, teremos as estruturas da Figura 27.



📷 Figura 27 – Árvores AVL com altura h e número mínimo de nós.

Provamos que a árvore tem altura h e número mínimo de nós com a seguinte argumentação: Em qualquer uma das árvores $T(1)$ a $T(5)$, qualquer folha que seja removida causa a destruição da propriedade AVL ou a redução da altura da árvore, isto é, ou a árvore deixa de ser AVL ou perde a altura proposta.

Sendo assim, as topologias da Figura 27 representam as árvores AVL com altura h e número mínimo de nós. Observe que estas estruturas não são únicas, dependendo de como fazemos a composição, alteramos a topologia, mas não o número de nós.

Observe que podemos encontrar o número mínimo de nós em função da altura com a seguinte soma recursiva $|T_h| = |T_{h-1}| + |T_{h-2}| + 1$. Isto é: A cardinalidade da árvore com altura h é igual à soma da cardinalidade da árvore de altura $h-1$ com a cardinalidade da árvore de altura $h-2$ mais 1.

A Tabela 1 apresenta a evolução desta série.

Árvore	h	$ T_h $	F_h
T_1	1	1	1
T_2	2	2	1
T_3	3	4	2
T_4	4	7	3
T_5	5	12	5
T_6	6	20	8
T_7	7	33	13
T_8	8	54	21

📷 Tabela 1 – Comparação da sequência T_h com F_h (Sequência de Fibonacci).

Da tabela, vemos que para $h > 2$, $|T_h| = F_{h+2} - 1$. Ou seja, o valor mínimo de n para que possamos construir uma árvore AVL com altura h é $n = F_{h+2} - 1$.

O termo geral da série de Fibonacci é:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Sendo assim, $|T_h| = n$ é:

$$n = |T_h| = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+2} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+2} \right] - 1$$

$$n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{h+2} - 1$$

$$n > \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2}$$

$$\log_{\frac{1+\sqrt{5}}{2}} n > h + 2 - \log_{\frac{1+\sqrt{5}}{2}} \sqrt{5}$$

$$h < \log_{\frac{1+\sqrt{5}}{2}} n + \log_{\frac{1+\sqrt{5}}{2}} \sqrt{5} - 2$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Isto é, a altura da árvore AVL é proporcional a $\log n$. Ou seja, as árvores AVL são balanceadas.

A família de árvores AVL com esta propriedade é chamada de **Árvores de Fibonacci**.

A conclusão deste estudo mostra que as melhores árvores AVL, isto é, as que têm altura mínima são as árvores completas e têm altura proporcional a $\log n$.



As piores árvores AVL, isto é, que têm altura máxima e número mínimo de nós, que são as árvores de Fibonacci, também têm altura proporcional a $\log n$.

De fato, não determinamos a função matemática para calcular a altura de uma árvore AVL com n nós.

Entretanto, sabemos que esta função é limitada superiormente pela altura das árvores de Fibonacci e inferiormente pela altura das árvores completas, ambas proporcionais a $\log n$. Sendo assim, a função que calcula a altura de uma árvore AVL com n nós é “sanduichada” por duas funções logarítmicas com bases diferentes.

O comportamento da altura de uma árvore AVL qualquer é proporcional a $\log n$, garantindo que as árvores AVL são balanceadas.

BUSCA EM UMA ÁRVORE AVL

Uma árvore AVL é uma árvore binária de busca, portanto, não existe diferença entre o algoritmo da busca em uma árvore binária de busca e o da busca em uma árvore AVL. O algoritmo é rigorosamente o mesmo.

Quando estudamos a busca nas árvores binárias de busca, concluímos que a complexidade era $O(n)$. Isto acontece uma vez que o pior caso, isto é, as árvores com altura máxima e menor quantidade de nós são as árvores zig-zag, que têm altura n .

Nas árvores AVL, as piores árvores – isto é, as árvores com altura máxima e menor quantidade de nós – são as árvores de Fibonacci que têm altura proporcional a $\log n$, sendo assim, a complexidade do algoritmo da busca, quando aplicado em árvores AVL, é $O(\log n)$.

Observe que o que define a complexidade da busca é a altura da árvore, não o algoritmo propriamente dito. Isto se deve ao fato de que o algoritmo de busca tem como operação fundamental a comparação e que este algoritmo executa uma comparação por nível na árvore e que, no pior caso, a chave encontrada está no último nível da árvore, que é igual à altura da árvore.

INSERÇÃO EM UMA ÁRVORE AVL

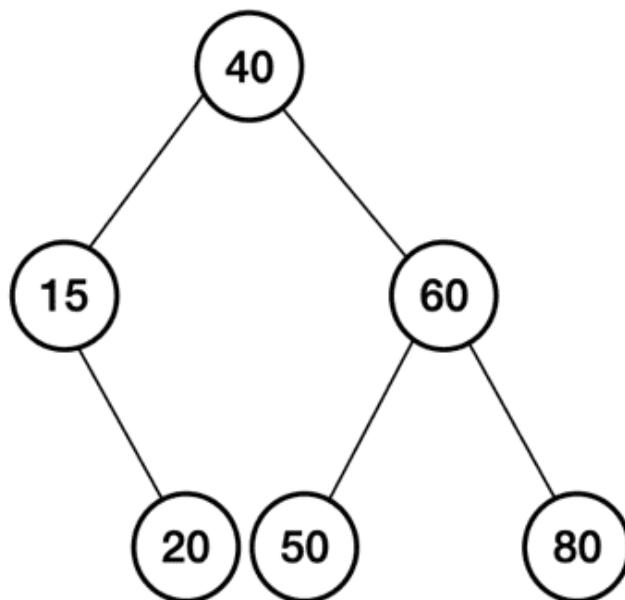
Seja T uma árvore AVL na qual iremos inserir uma nova chave. Como uma árvore AVL é uma árvore binária de busca, então o funcionamento do algoritmo de inserção, isto é, onde a nova chave deve ser inserida, é definido da mesma forma que já estudamos.

A busca determina a posição, que é a subárvore vazia onde a busca deveria prosseguir para encontrar a chave que desejamos inserir. Todo o processo já foi estudado e o algoritmo apresentado.

Sendo assim, vamos nos concentrar em verificar o que pode acontecer com a propriedade fundamental das árvores AVL, isto é, a regulagem dos nós.

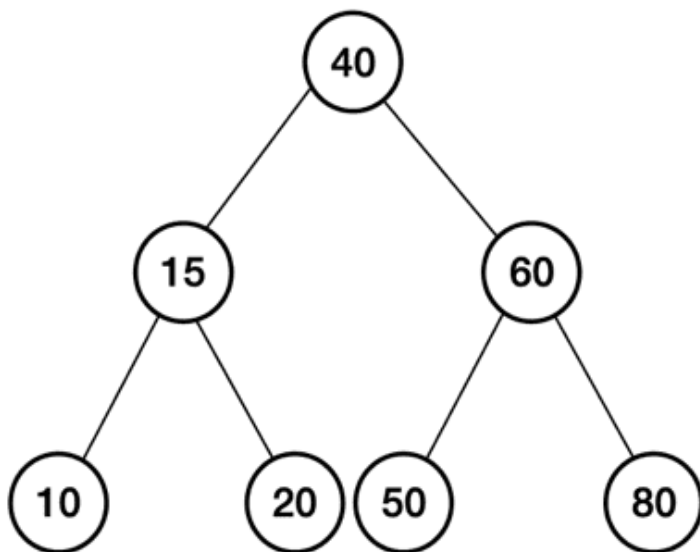
Recordemos que em uma árvore AVL deve valer para todos os nós $|h(T_r^e) - h(T_r^d)| \leq 1$. Quando isto ocorre, dizemos que o nó está regulado.

Para fins de análise, consideremos a árvore da Figura 28.



📷 Figura 28 – Árvore AVL.

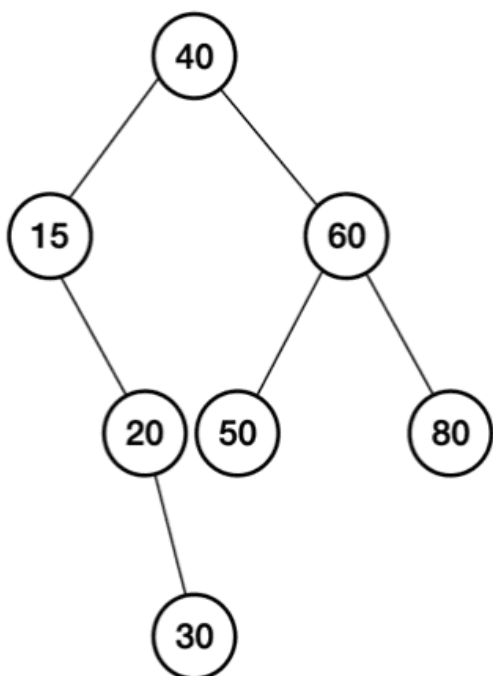
Na árvore da Figura 28, se inserirmos uma nova chave, a chave “10”, por exemplo, não haverá a perda da regulagem dos nós da árvore. A árvore resultante continuará a ser uma árvore AVL (Figura 29).



📷 Figura 29 – Árvore resultante após inserção da chave “10”.

Contudo, se ao invés de inserirmos a chave “10”, inserirmos a chave “30”, a árvore resultante deixa de ser AVL. Na Figura 30, temos a árvore resultante e, nesta árvore, temos que o nó “20” está regulado, porém o nó “15” está desregulado.

A altura de sua subárvore esquerda é 0 e sua subárvore direita tem altura 2 (Figura 30).



📷 Figura 30 – Árvore resultante após a inserção da chave “30” – Não é AVL.

Observe que na árvore da Figura 30, o nó “15” é o nó mais profundo desregulado. O nó mais profundo é aquele de maior nível, ou seja, “15” é o nó de maior nível na árvore da Figura 36 desregulado.

Concentremos a análise neste nó, seja “u” este nó.

A situação que analisaremos é a da Figura 31.



Figura 31 – inserção de uma chave $x > u$ na árvore AVL.

Destacando a raiz de T2, temos as seguintes possíveis situações (Figura 32) antes da inserção da chave x .

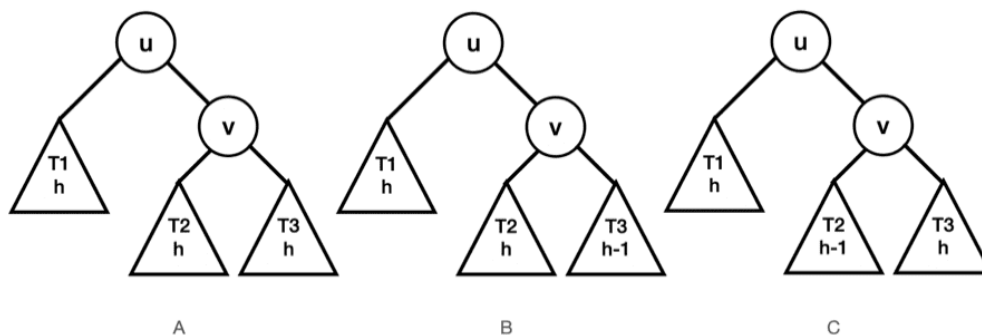


Figura 32 – Árvore da Figura 28 antes da inserção de x .

Observe que o único cenário viável é o da Figura 32(A). Ao inserir x na árvore da Figura 32(A), T2 ou T3 podem crescer e este crescimento não desregula o nó “v”. Nas árvores das Figuras 32(B) e (C), a inserção de “ x ” na árvore iria desregular “v”, o que contraria nossa premissa de que “u” é o nó mais profundo desregulado, ou não desregula nó algum, o que também contraria nossa premissa.

Análise semelhante deve ser feita para o caso de inserirmos uma chave $y < u$. Neste caso, pode-se concluir que o único cenário de estudo é o da Figura 33.

Chamaremos de Caso 1 o correspondente à Figura 32(A) e Caso 2 o da Figura 33.

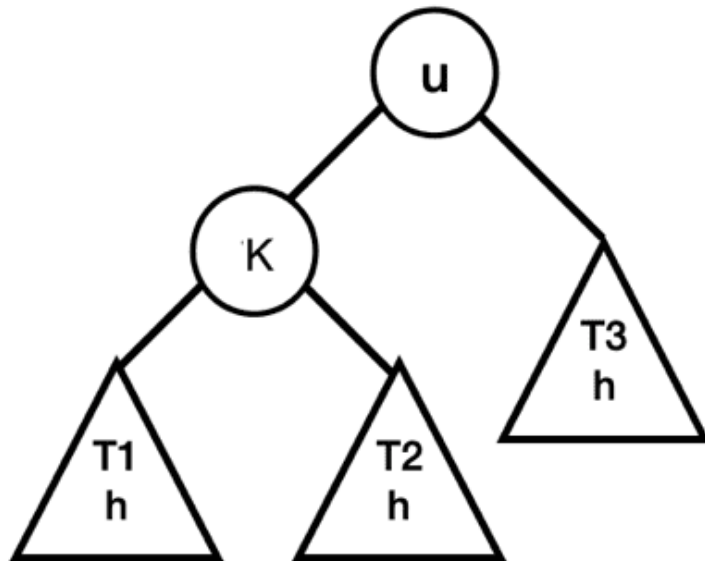


Figura 33 – Caso 2.

Voltando à análise do Caso 1, veremos que este caso pode ser dividido em dois subcasos. O subcaso 1.a, que corresponde a inserir uma chave $x > u$ e $x > v$ e o subcaso 1.b, que corresponde a inserir uma chave x , tal que $u < x < v$.

Vejamos o subcaso 1.a.

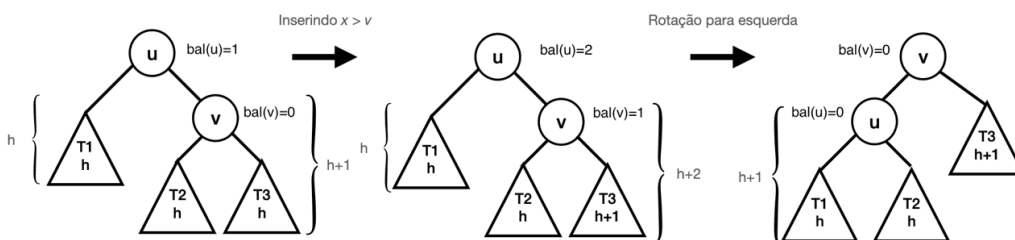


Figura 34 – Inserção de chave $x > v$ e ajuste da estrutura.

Na Figura 34, temos a ilustração do subcaso 1a. Suponhamos que ao inserir a chave “x” o ramo T3 da árvore cresce.

Definindo balanço de x como $bal(x) = h(T_x^d) - h(T_x^e)$, temos que o nó “v” continua regulado ($bal(v)=1$), conforme nossa hipótese de análise, entretanto “u” está desregulado ($bal(u)=2$). Após a inserção, a subárvore esquerda de u tem altura h e a subárvore direita tem altura $h+2$.

A operação que reajusta a estrutura é a rotação para a esquerda. Trata-se de transformar o nó “v” como raiz da subárvore, fazer “u” seu filho esquerdo ($u < v$, lembre-se, trata-se de uma árvore binária de busca) e fazer T1 subárvore esquerda de “u” (os nós k de T1 satisfazem a propriedade $k < u$), T2 subárvore direita de “u” (os nós k de T2 satisfazem a propriedade $u < k < v$) e T3 subárvore direita de v (os nós k de T3 satisfazem a propriedade $k > v$).

Outro fato importante deve ser destacado. Após a rotação, a árvore resultante tem mesma altura que a árvore original. Isto garante que, para os ascendentes de u , se houverem, e porventura existirem e tivessem se tornado desregulados após a inserção, a aplicação da rotação iria ajustá-los automaticamente.

Vejamos esta situação na árvore da Figura 35.

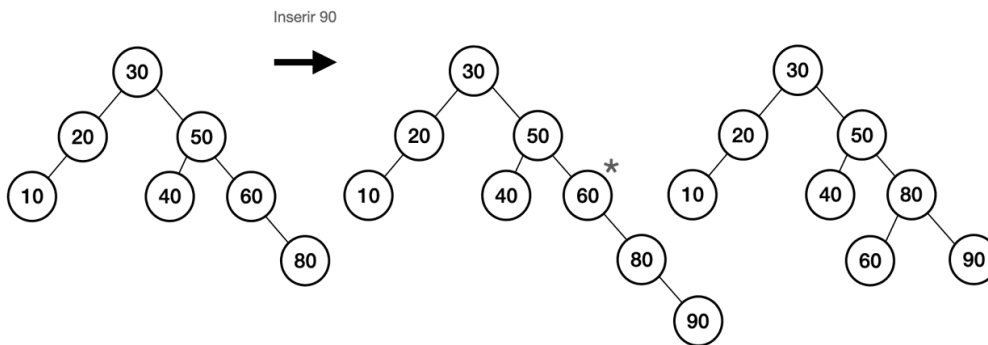


Figura 35 – Inserção da chave “90”.

Na Figura 35, inserimos a chave “90” na estrutura. O resultado da inserção é que o nó “60” é o nó mais profundo desregulado. Entretanto, os nós “50” e “30” também estão desregulados.

Conforme vimos, aplicamos a rotação para a esquerda no nó “60” que resulta na estrutura com as chaves “60”, “80” e “90” com altura 2.



A rotação regula a estrutura e, como a subárvore resultante tem altura 2, assim como o ramo original, os nós “50” e “30” voltam a estar regulados automaticamente.

O caso 2a é análogo ao caso 1a. Após a inserção de uma chave $y < k$ na árvore da Figura 36(A), podemos ter como resultado a árvore da Figura 36(B), e após a aplicação da rotação para a direita (Figura 36(C)), a árvore volta a estar regulada. Todas as observações feitas para o caso 1a são análogas para o caso 2a.

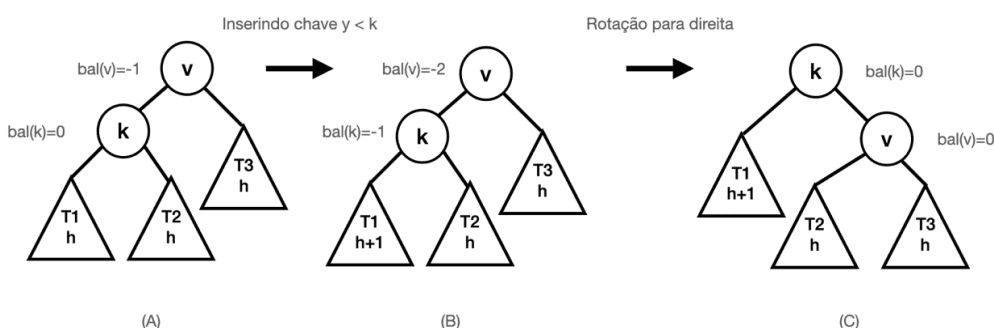


Figura 36 – Caso 2a, inserção de chave $y < k$.

Vejamos agora a análise do Caso 1.b, isto é, a inserção de uma chave x tal que $u < x < v$. A Figura 37 ilustra o caso 1.b

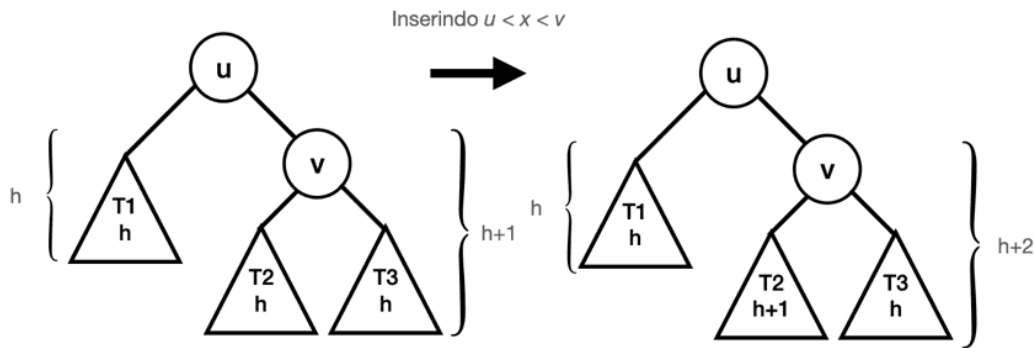


Figura 37 – Inserindo $u < x < v$.

Destacando a raiz de T2 antes da inserção da nova chave, podemos ter as configurações da Figura 38. Veja as implicações e possibilidades destas configurações:

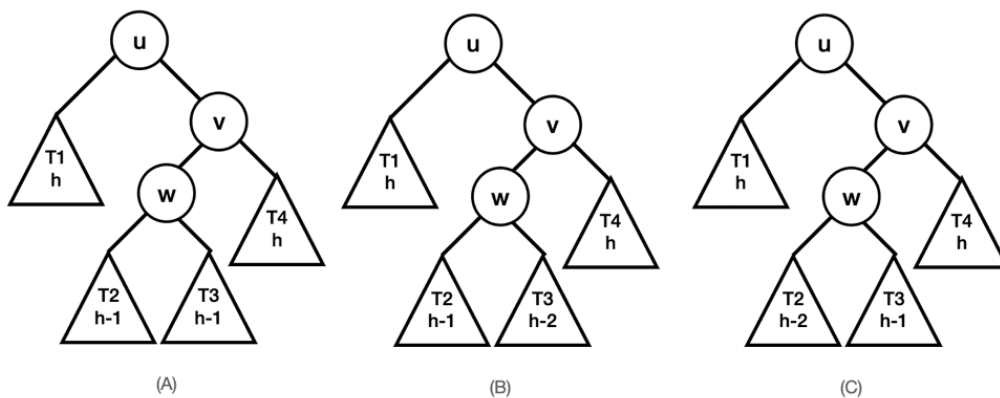


Figura 38 – Análise do subcaso 1.b.

No caso da Figura 38(A), caso a nova chave seja maior ou menor que w , pode implicar no crescimento do ramo direito ou esquerdo da árvore com raiz w .

No caso da Figura 38(A), T2 ou T3, caso estas árvores não cresçam, isto é, mantenham sua altura original, nada precisamos fazer. Contudo, se T2 ou T3 crescer devido à inclusão, teremos w regulado, v regulado e u desregulado. Coerente com que assumimos inicialmente. Então, o cenário da Figura 38(A) é um dos focos do nosso estudo.

No caso da Figura 38(B) e (C), observe que os casos são semelhantes, divergem somente no ramo de maior altura esquerdo (Figura 38(B)) e direito (Figura 38(C)).

Analisando a árvore da Figura 38(B), se inserirmos uma nova chave no ramo direito (T3), poderá ocorrer o crescimento de T3, contudo, se T3 crescer, a árvore de raiz w tem balanço zero, não repercutindo nos nós v e u .

Entretanto, se T2 crescer, isto é, passar a ter altura h , o nó w passará a estar desregulado, o que fere o que supomos inicialmente, isto é, que u é o nó mais profundo desregulado. Desta feita, o único cenário

plausível para análise é o da Figura 38(A).

Vejamos o que pode ocorrer quando inserimos uma nova chave na subárvore de raiz w da Figura 38(A).

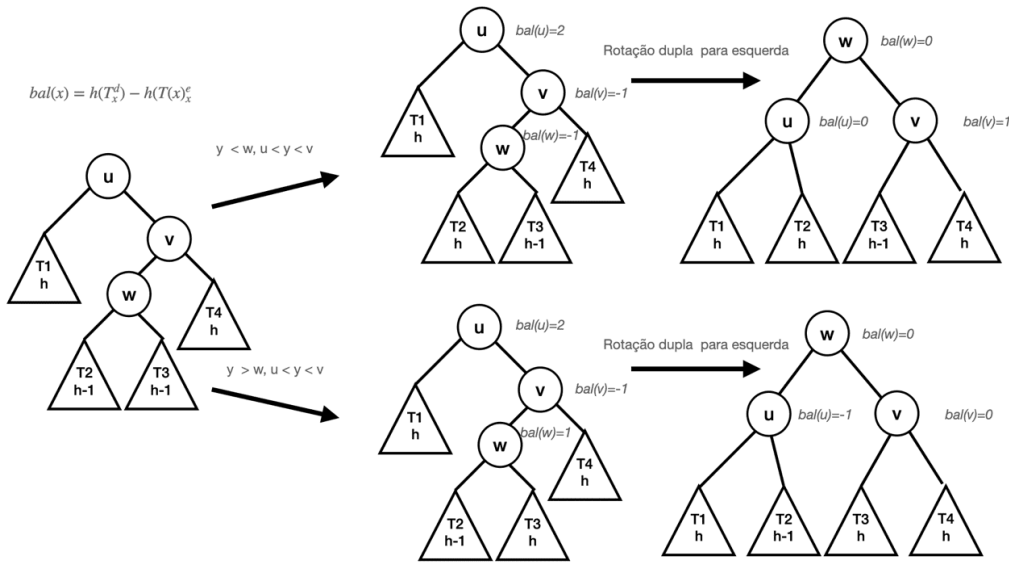


Figura 39 – Rotação dupla para esquerda.

A Figura 39 ilustra a situação, observe que u é o nó desregulado. A rotação dupla para a esquerda resolve a regulagem de u . Nesta rotação, w é posto como raiz da subárvore. Como u é menor que w , u é inserido como filho esquerdo de w e v como filho direito de w ($v > w$).

Observe que os nós de $T1$ são menores que u , os nós z de $T2$ obedecem à propriedade $u < z < w$, os nós z de $T3$ obedecem à propriedade $w < z < v$ e os nós z de $T4$ obedecem $z > v$.

Vejamos um exemplo na Figura 40:

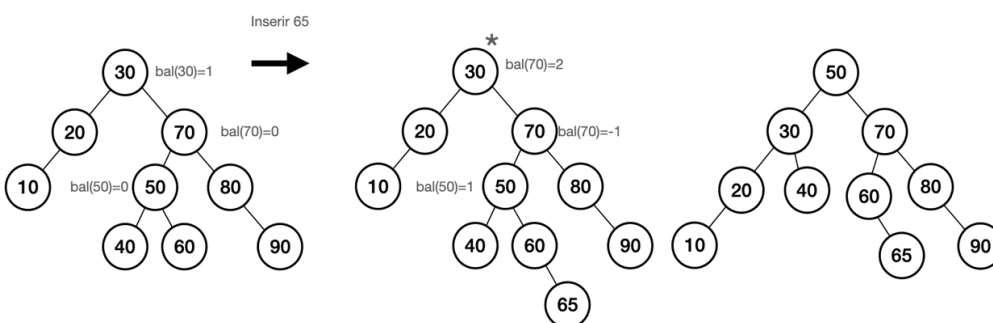


Figura 40 – Inserção da chave 65.

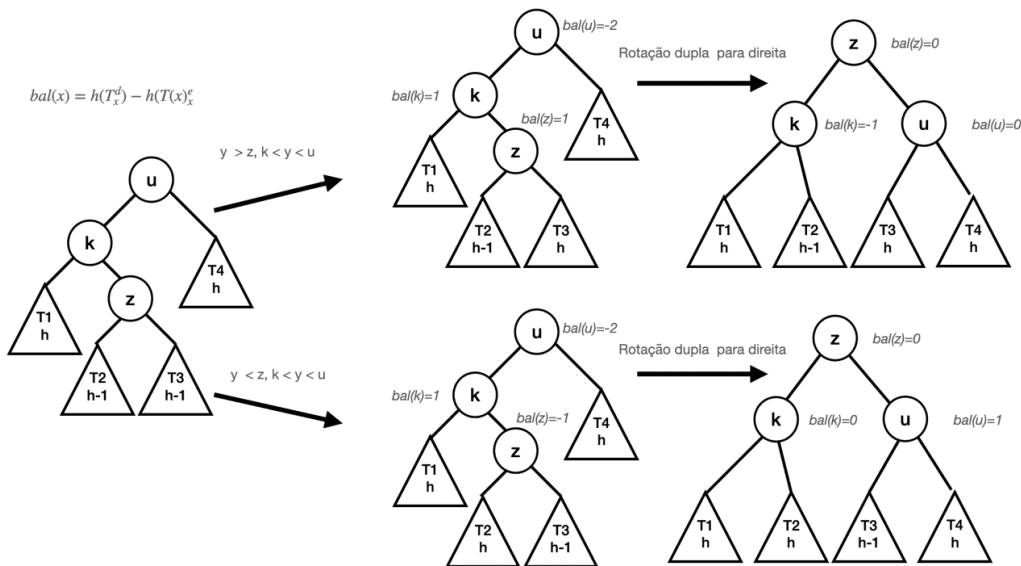
Observe que na árvore do exemplo, após a inserção da chave 65, o nó 50 e 70 permanecem regulados e 30 desregulado.

Após a inserção dupla para esquerda, todos os nós ficam regulados.

Outro fato importante sobre a rotação dupla para esquerda é que, após sua aplicação, a árvore resultante tem a mesma altura que a árvore original, antes da inserção da nova chave. Essa propriedade garante que regulando o nó u , todos os ancestrais de u ficam automaticamente regulados, como ocorreu no caso 1a.

A análise do caso 2b é análoga a do caso 1b.

A Figura 41 mostra como identificar e aplicar a rotação dupla para a direita.



📷 Figura 41 – Caso 2b.

Para concluir a análise do algoritmo de inclusão de nós na árvore AVL, resta determinar como é feita a identificação das rotações pelo algoritmo. Observe que na análise nos referimos à diferença de altura entre o ramo direito e esquerdo. Esta diferença foi chamada **balanço do nó**.

Contudo, se calcularmos a altura das subárvores esquerda e direita de um nó para determinar o balanço, não será possível manter a complexidade de $O(\log n)$ da operação de inserção, uma vez que, para calcular a altura de uma árvore, é necessário executar um algoritmo com complexidade $O(n)$. A solução para este problema é **armazenar o balanço do nó com a chave**.

Conforme definido, $bal(x) = h(T_x^d) - h(T_x^e)$. O crescimento do módulo do balanço de uma chave indica que o ramo cresceu, sendo necessário atualizar o balanço incrementando-o, caso o ramo que tenha crescido seja o direito ou decrementando-o, caso o ramo que tenha crescido seja o esquerdo.

A identificação do caso ocorre quando um nó tem balanço 2 indicando o caso 1 ou balanço -2 indicando o caso 2.

O subcaso é identificado pelo balanço do filho direito, no caso 1, se o balanço do filho direito for 1, temos o subcaso 1a.

Se o balanço do filho direito for -1, temos o subcaso 1b.

Analogamente, o balanço do filho esquerdo identifica o subcaso do caso 2.

Caso o balanço do filho esquerdo seja -1, temos o caso 2a.

caso seja 1, temos o subcaso 2b.

O pseudocódigo do algoritmo será omitido, entretanto, pode ser facilmente encontrado na internet.

REMOÇÃO

A análise da remoção é muito semelhante à da inserção de novos nós na árvore. Toda a análise da inserção baseia-se no estudo do crescimento de um ramo da árvore devido ao novo nó contendo a nova chave. A remoção no encurtamento do ramo que removemos a chave.

Além disto, a árvore AVL é uma árvore binária de busca, a remoção recai nos três subcasos estudados no módulo 2.

Como exemplo, vejamos a remoção de um nó de T3 que resulta no encurtamento de T3 (Figura 42).

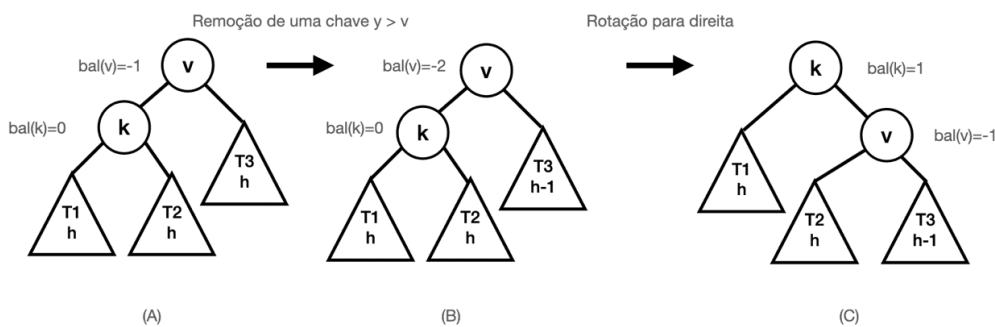


Figura 42 – Remoção de uma chave $y > v$.

O ajuste aplicado na estrutura foi o do caso 2a. Porém, observe que há uma sutil diferença em relação ao balanço de k , filho esquerdo de v , no exemplo, k tem balanço 0 e resultou na rotação para a direita. Na inclusão, a rotação para a direita ocorria quando o balanço de k era igual a -1.

Sendo assim, a aplicação da rotação para direita ocorre quando $bal(v) = -2$ e o balanço do filho esquerdo de v , no caso do exemplo k , é 0 ou -2, isto é, $bal(k) = 0$ ou $bal(k) = -1$ (Figura 49).

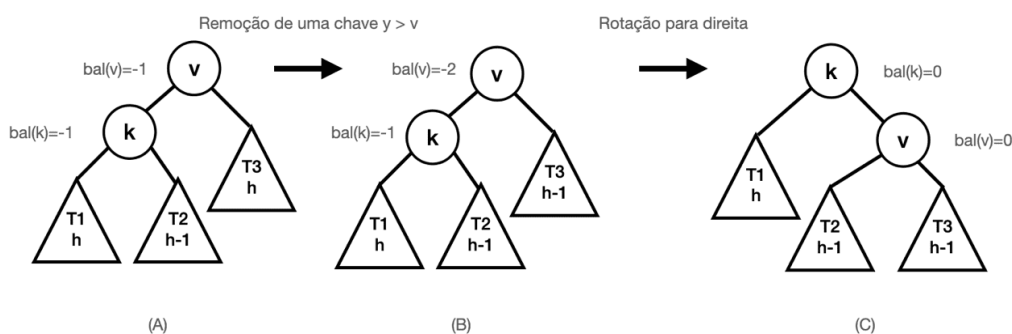


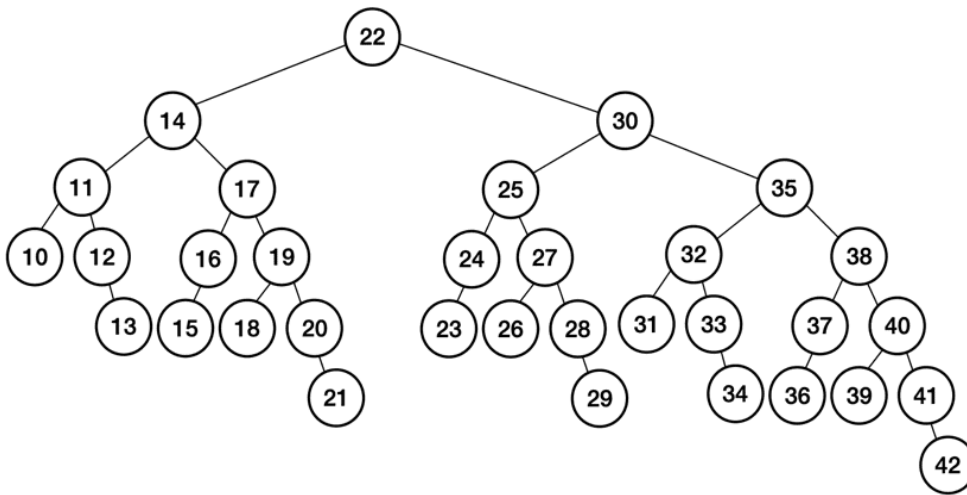
Figura 43 – Aplicação da rotação do caso 2a, possibilidade 2.

A Figura 43 mostra que, na aplicação da rotação na remoção, existe a possibilidade de a estrutura resultante ter seu tamanho reduzido de uma unidade.



Neste caso, caso haja ancestrais de v na árvore original, isto é, antes da remoção de y , o efeito da remoção pode propagar-se para os ancestrais de v .

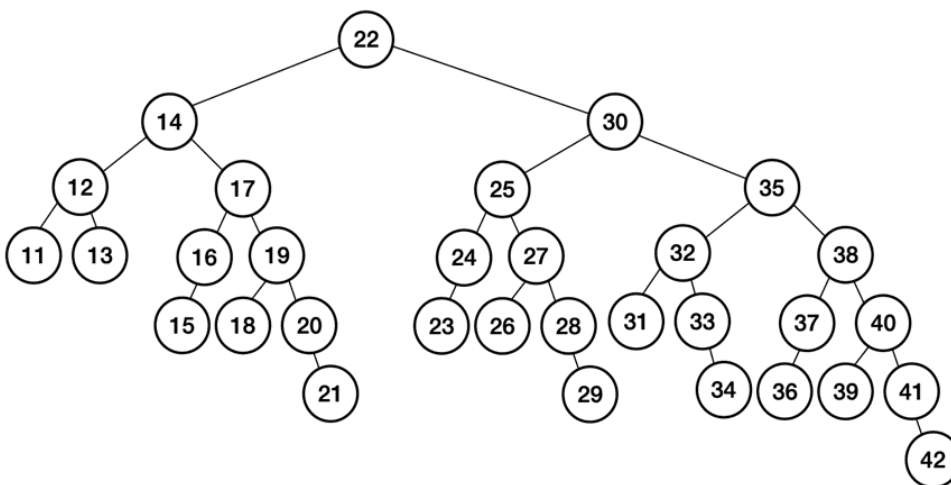
Esta situação ocorre, por exemplo, na árvore de Fibonacci. A Figura 44 mostra um exemplo.



📷 Figura 44 – Árvore de Fibonacci.

Ao removermos o nó 10, temos que o $\text{bal}(11)=2$ e $\text{bal}(12)=1$, que corresponde ao caso 1a.

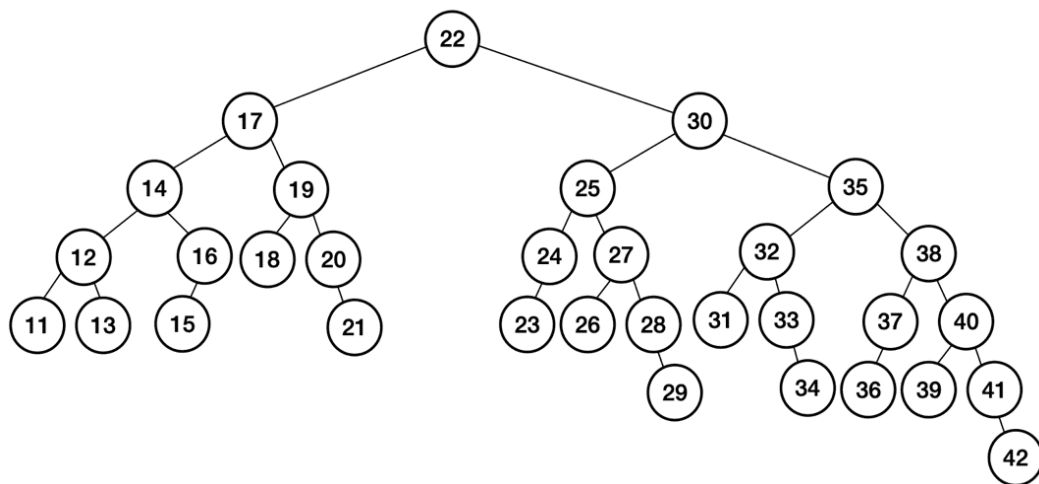
Aplicando a rotação para esquerda, temos a configuração da Figura 45.



📷 Figura 45 – Resultado da aplicação da rotação para esquerda no nó 11.

Observe que após a rotação, o nó “14” tem balanço 2 e o nó “17” tem balanço 1, também o caso 1a.

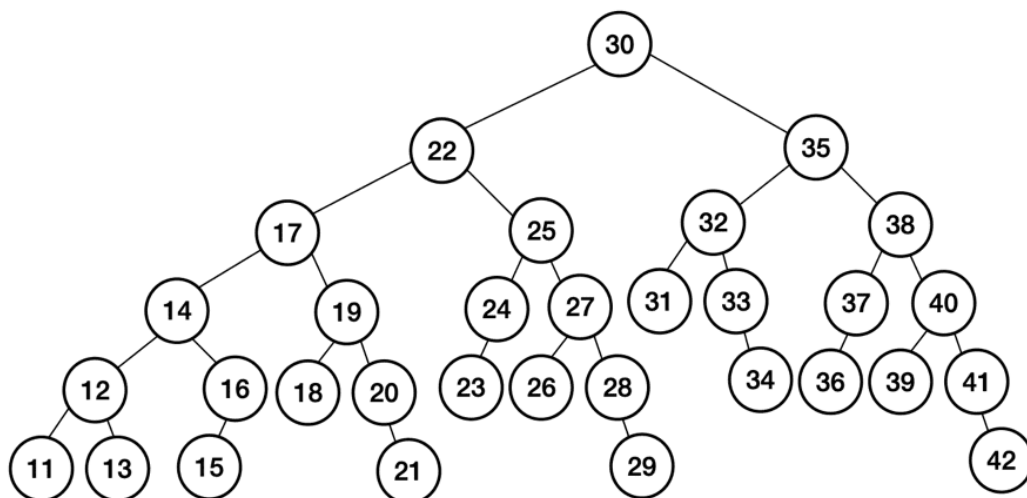
Aplicando a rotação, temos a árvore da Figura 46.



📷 Figura 46 – Resultado da aplicação da rotação para a esquerda no nó “14”.

Após a rotação, o nó “22” tem balanço 2 e o nó “30” tem balanço 1, também caso 1a.

Aplicando a rotação para a esquerda no nó “22”, temos a árvore da Figura 47.



📷 Figura 47 – Resultado da aplicação da rotação para a esquerda no nó “22”.

Observe que as rotações em sequência reestruturam a propriedade AVL após a remoção. A árvore de Fibonacci é o pior caso da remoção.

Conforme visto, nas árvores de Fibonacci, é necessário realizar todas as rotações do pai do nó removido, no caso do exemplo da Figura 44, o nó “11” até a raiz.

Cada rotação tem complexidade da $O(1)$, assim, a aplicação da rotação no caminho da folha removida até a raiz não tem impacto na complexidade da remoção que é $O(\log n)$.

Concluindo, o algoritmo de remoção recai nos subcasos estudados no módulo 2 e após a aplicação destes subcasos, o balanço do nó pai do nó removido deve ser atualizado e aplicadas as rotações conforme os casos estudados na inserção caso 1, subcaso 1a e 1b ou caso 2, subcaso 2a e 2b.

Na remoção, os subcasos 1a e 2a ocorrem quando $bal(v) = 2$ e do filho direito é 1 ou 0 ou $bal(v) = -2$ ou o balanço do filho esquerdo é -1 ou 0 respectivamente. Os subcasos 1b e 2b são identificados da mesma

forma que na inclusão.

ESTUDO DA COMPLEXIDADE DA ÁRVORE AVL

Vimos que as árvores AVL são balanceadas, isto é, toda árvore AVL tem altura proporcional a $\log n$.

Deste fato, decorre que a complexidade da busca é $O(\log n)$.

Foi visto que a complexidade da busca é proporcional à altura da árvore, no caso da árvore AVL, $\log n$ e que toda árvore AVL é uma árvore binária de busca, sendo assim, aplica-se o mesmo algoritmo visto no módulo 2.

A inclusão de uma nova chave na árvore AVL também tem complexidade $O(\log n)$, isto decorre do fato de que a inclusão é consequência direta da busca e a busca executa em $O(\log n)$.

No pior caso, a inclusão de uma nova chave pode resultar em uma rotação, que é uma operação simples, que executa em $O(1)$. Não é impactante na complexidade, mas vimos que após a aplicação da rotação na inclusão, o ramo alterado pela inclusão preserva sua altura original, isto é, antes da inserção da nova chave, fazendo com que novas rotações nos ancestrais sejam desnecessárias.

Finalmente, o pior caso da remoção é a remoção da folha menos profunda de uma árvore de Fibonacci. Neste caso, realizamos todas as rotações da folha removida até a raiz.

Contudo, cada rotação tem complexidade da $O(1)$ e são realizadas $O(\log n)$ rotações até regular toda árvore. Assim, as árvores AVL fornecem algoritmos totalmente dinâmicos capazes de realizar busca, inserção e remoção em $O(\log n)$.

REALIZANDO OPERAÇÕES EM UMA ÁRVORE AVL



VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

As árvores binárias de busca são uma importante ferramenta para a indexação de informação em repositórios de dados. Sem estas estruturas de dados, o desempenho dos algoritmos de busca, inserção e remoção é elevado em uma ordem. O tema mostra como aplicar estes algoritmos, otimizando ao máximo seu desempenho.

Finalmente, o tema apresentado aborda como obter um algoritmo completamente dinâmico para realizar busca, inserção e remoção em $O(\log n)$, o que representa uma importante otimização em relação às estruturas de dados não organizadas que necessitam de $O(n)$ passos para realizar as mesmas operações.



PODCAST

 **PODCAST**

REFERÊNCIAS

ADELSON-VELSKY, G. M.; LANDIS, E. M. **An Algorithm for the Organization of Information**. *In*: Soviet Mathematics Doklady, v. 3, 1962, p. 1259-1263.

STOUT, Q.; WARREN, B. **Tree Rebalancing in Optimal Time and Space**. *In*: Communications of the ACM, v. 29, Nr. 9, 1986.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

Bibliotecas existentes para a linguagem Python.

CONTEUDISTA

Luiz Henrique da Costa Araújo

 **CURRÍCULO LATTES**