Técnicas e Análise de Algoritmos

Árvores Binárias e de Busca - Parte 01

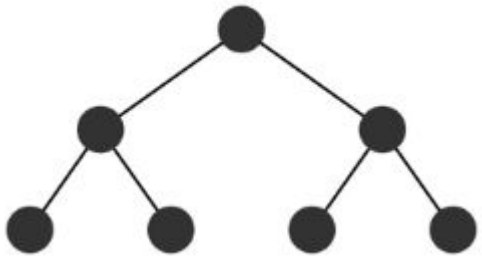
Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Árvores - Definição

Árvores - Definição

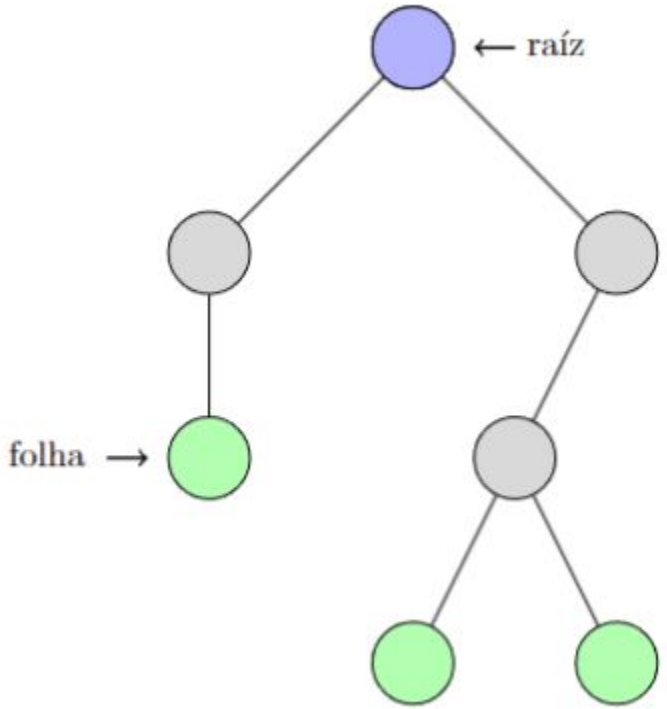
- As árvores são estruturas compostas de nós e arestas (ramos)
- Ao contrário das árvores reais, a visualização de árvores em algoritmos é invertida, com a raiz no topo e as folhas na base
- A raiz é um nó que não tem pai
- Folhas são nós que não tem filhos



Árvores - Definição

- Cada nó pode ser alcançado através de uma sequéncia única de ramos, denominada caminho
- O nível de um nó N corresponde ao número de nós do caminho de N até a raiz
- A altura de uma árvore é igual ao nível máximo dentre todos os nós da árvore

Árvores - Definição



Árvores Binária

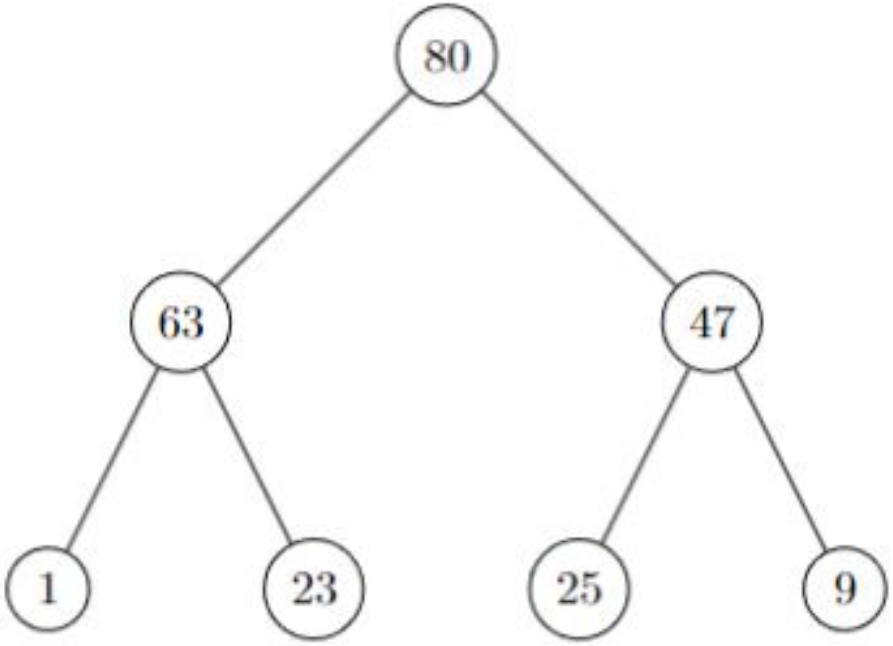
Árvore Binária

- A definição de árvores não impõem qualquer restrição no número de filhos que um nó pode ter
- Uma árvore é dita binária se cada nó tem, no máximo, dois filhos
  - O esquerdo e o direito
- O estabelecimento de uma ordem entre as informações armazenadas em um nó e seus filhos leva a especializações muito úteis em uma árvore binária

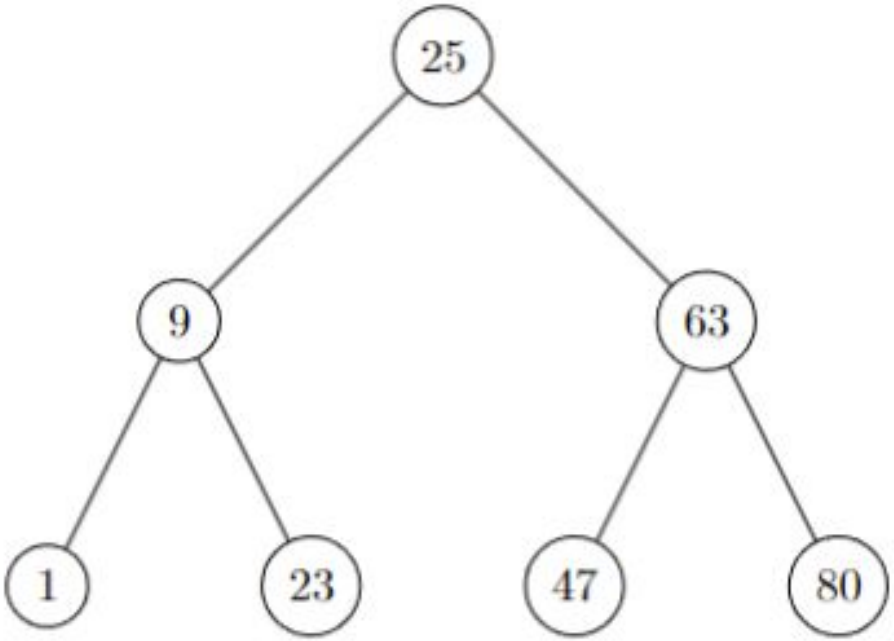
Árvore Binária

- Por exemplo, max heap é uma árvore binária cuja informação contida no pai é maior ou igual as informações contidas nos filhos
- Já em uma árvore binária de busca, as informações contidas em qualquer nó da subárvore à esquerda de um nós N devem ser menores do que a informação contida em N
- De maneira análoga, informações nos nós da subárvore à direita de N devem ser maiores do que a informação armazenada em N

Árvore Binária - Max Heap



Árvore Binária - Árvore Binária de Busca



Árvore Binária - Implementação

- Uma árvore binária pode ser implementada de duas formas:
  - Utilizando vetores
  - Utilizando ponteiros

Árvore Binária - Implementação

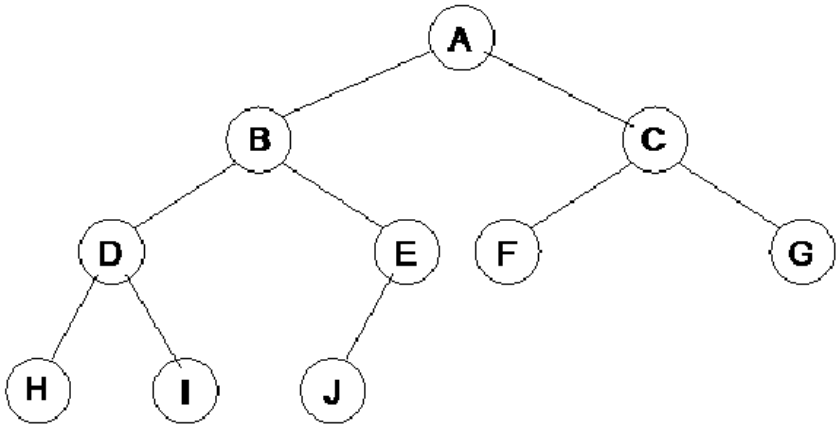
- Implementação com vetores:
  - A informação da raiz fica armazenada no índice 1
    - O índice zero não é utilizado
  - Dado um nó pai armazenado no índice p, o nó à esquerda ocupa o índice 2p e o nó à direita ocupa o índice 2p + 1
  - Se um nó ocupa o índice i ≠ 1, seu pai está armazenado no índice i/2

Árvore Binária - Implementação

- Implementação com vetores:

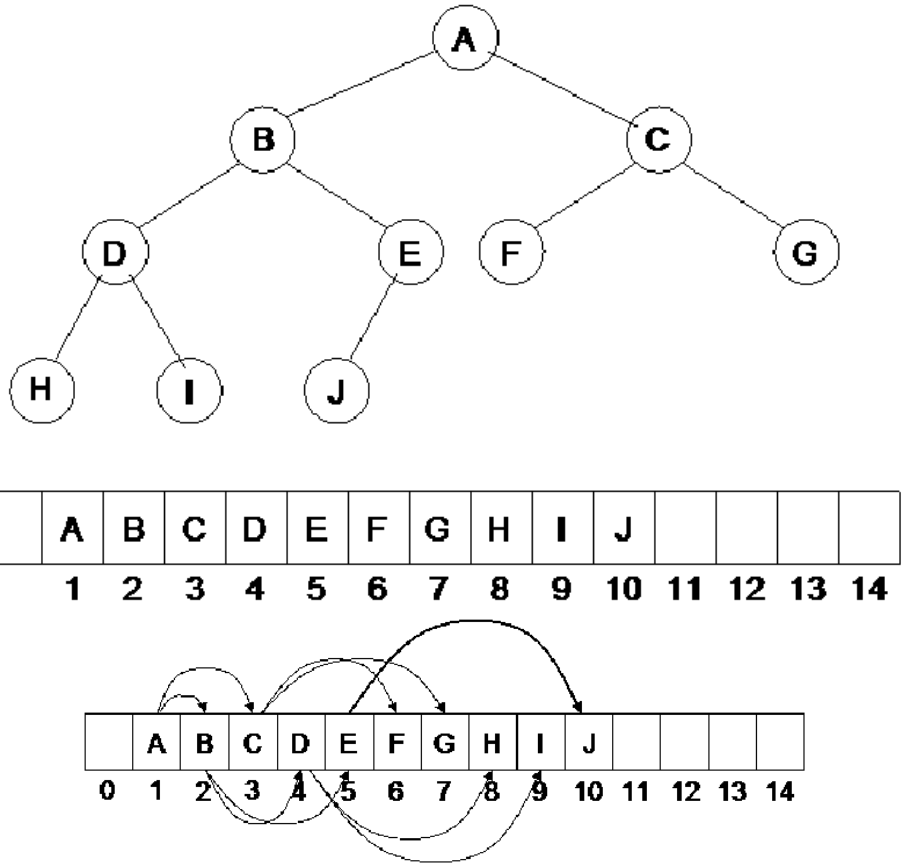
```
class ArvoreBinaria {
    private:
        vector <int> arvore;

        int pai(int i) { return i / 2; }

        int esquerda(int i) { return 2 * i; }

        int direita(int i) { return 2 * i + 1; }

    public:
        // Índice 0 não é usado (por isso o valor zero)
        ArvoreBinaria() { arvore.push_back(0); }
};
```

Árvore Binária - Implementação



	A	B	C	D	E	F	G	H	I	J				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Árvore Binária - Implementação



Árvore Binária - Implementação

- Implementação com ponteiros:
  - No caso da implementação via ponteiros, a estrutura que representa um nó contém dois ponteiros
    - Um para o filho da esquerda
    - Um para o filho da direita

Árvore Binária - Implementação

- Implementação com ponteiros:

```
class ArvoreBinaria {
    private:
        struct No {
            int valor;
            No *esquerda;
            No *direita;
        };
        No *raiz;
    public:
        ArvoreBinaria() {
            raiz = nullptr;
        }
};
```

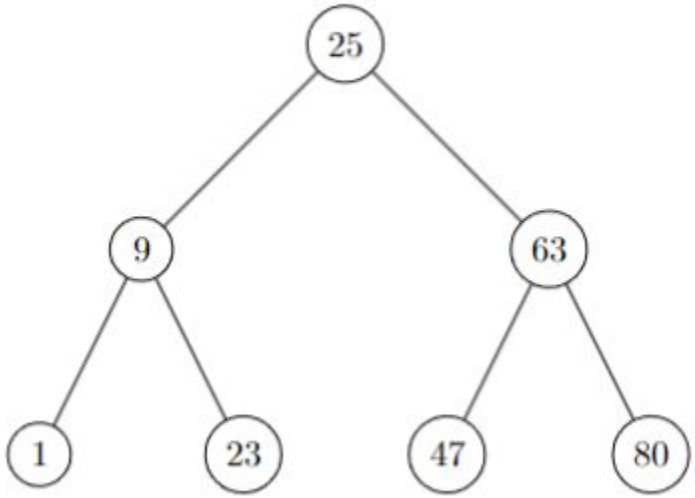
Árvores Binária de Busca
Operações

Árvore Binária de Busca - Inserção

- O algoritmo a seguir insere um elemento x em uma BST:
  a. Começe no nó raiz
  b. Enquanto o nó a ser avaliado for não-nulo:
     i. seja y a informação armazenada no nó a ser avaliado
     ii. se x for menor do que y, vá para a raiz da subárvore da esquerda
     iii. caso contrário, vá para a raiz da subárvore da direita
  c. Insira um novo nó com a informação igual ao valor a ser inserido como filho do último nó não-nulo, na posição adequada
- No pior caso, o algoritmo visita todos os N nós da árvore, de modo que este algoritmo tem complexidade O(N)

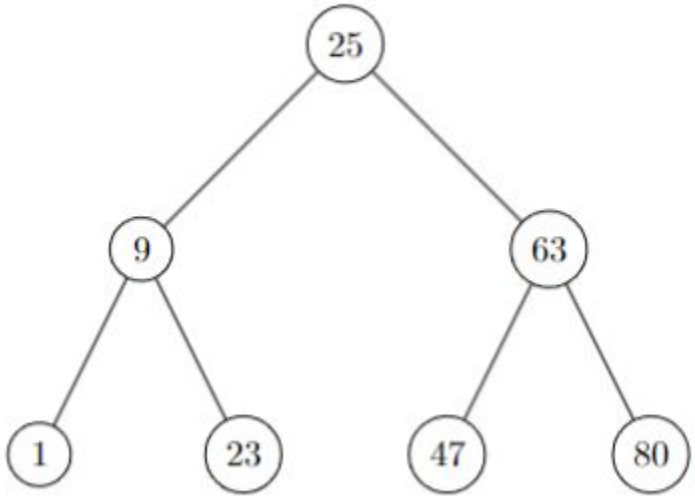
Árvore Binária de Busca - Inserção

Elemento a ser inserido: 14

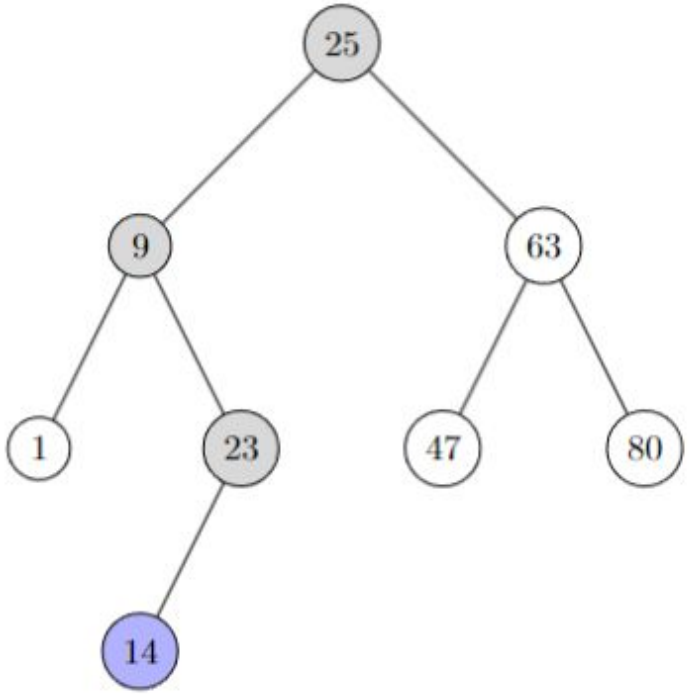


Árvore Binária de Busca - Inserção

Elemento a ser inserido: 14



Elemento a ser inserido: 14



Árvore Binária de Busca - Inserção

- Inserção não modifica a estrutura da árvore (ponteiros ou vetores), exceto pela posição do novo elemento
  - Propriedade de BST é preservada
- Inserção pode desbalancear uma árvore
  - Inserções de vários elementos em ordem crescente ou decrescente
  - Causam o pior caso do algoritmo (percorrer em tempo linear)
    - Árvore degenerada

Árvore Binária de Busca - Remoção

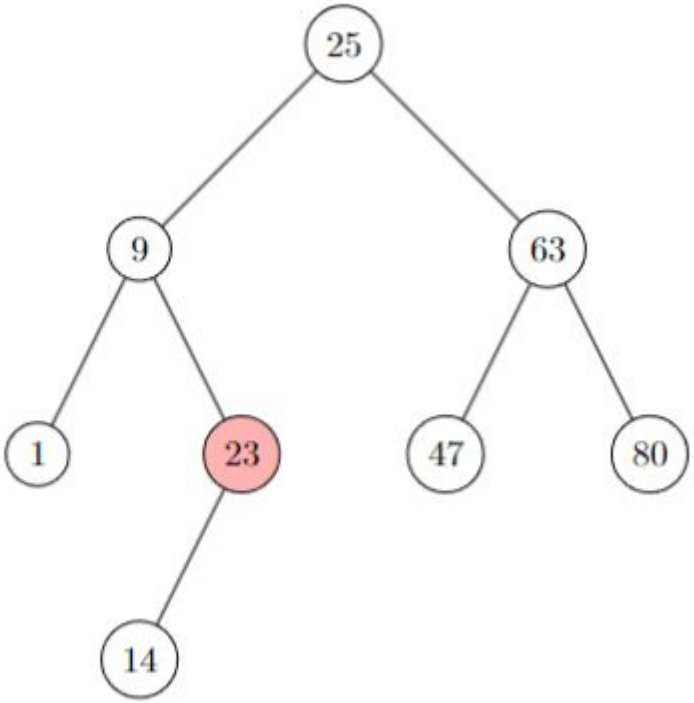
- A remoção em árvores binárias depende da posição do nó a ser removido
  - São três casos:
    i. o nó é uma folha, isto é, não tem filhos
    ii. o nó tem um filho
    iii. o nó tem dois filhos
- No primeiro caso, basta remover a referência do pai e remover o nó
- No segundo caso, a referência do pai é alterada para apontar para neto, e o nó é removido
- O terceiro caso não pode ser resolvido em um único passo

Árvore Binária de Busca - Remoção

- Como na maioria dos casos a remoção envolve na modificação da estrutura da árvore, a implementação comumente utilizada é a com ponteiros
  - Implementação com vetores precisaria movimentar muitos elementos
  - Implementação com vetores é favorecida quando não há mudanças na estrutura da árvore

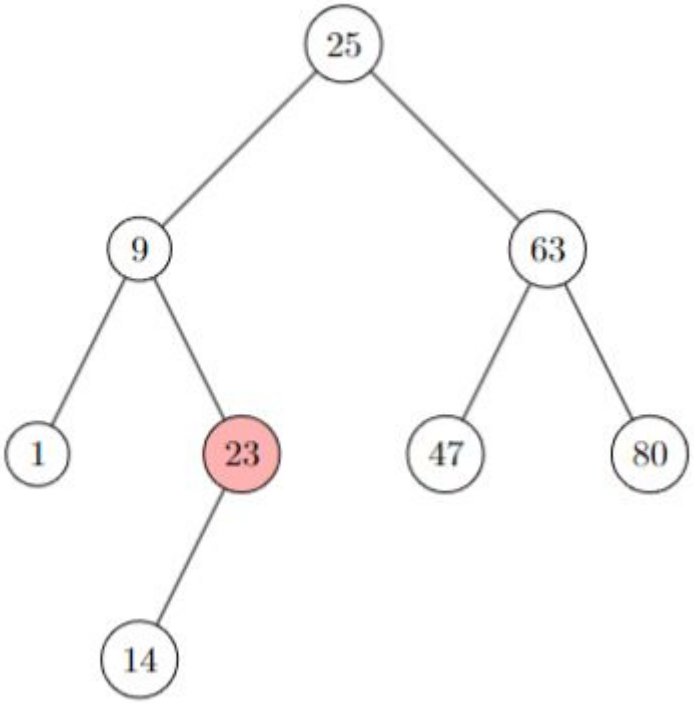
Árvore Binária de Busca - Remoção

Elemento a ser removido: 23

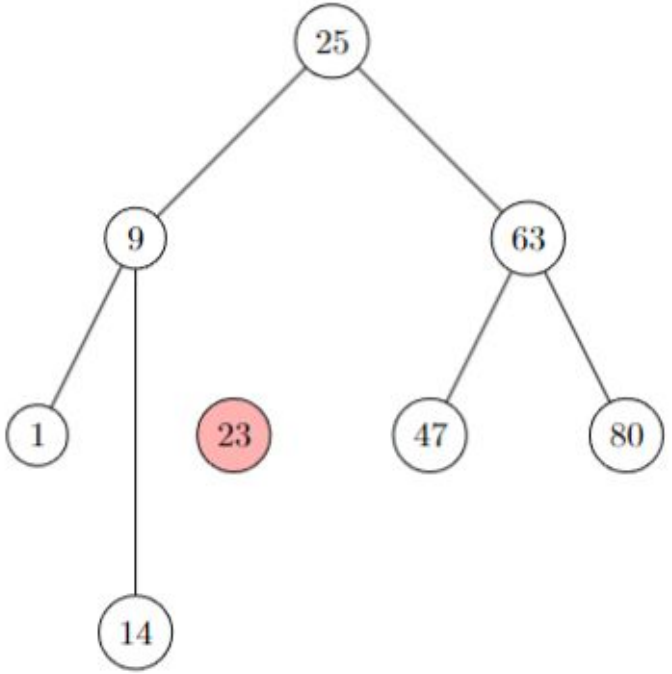


Árvore Binária de Busca - Remoção

Elemento a ser removido: 23



Elemento a ser removido: 23

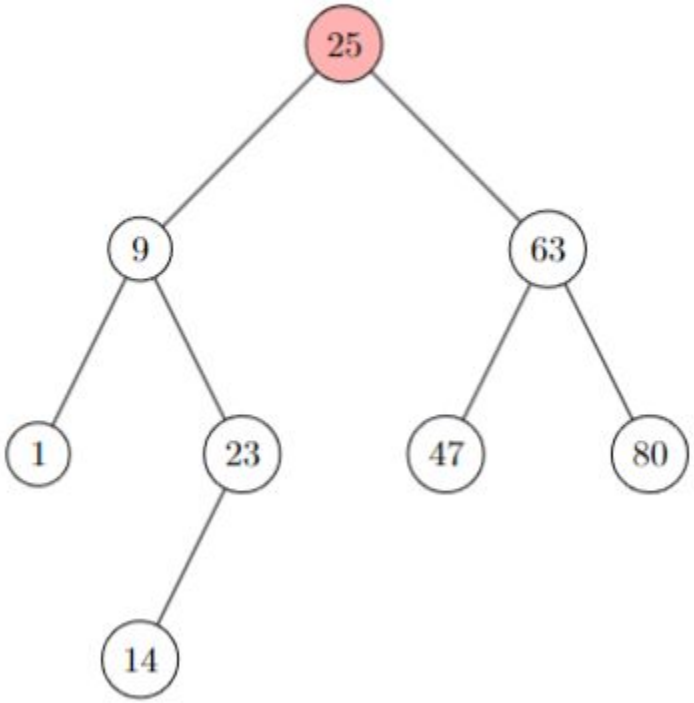


Árvore Binária de Busca - Remoção

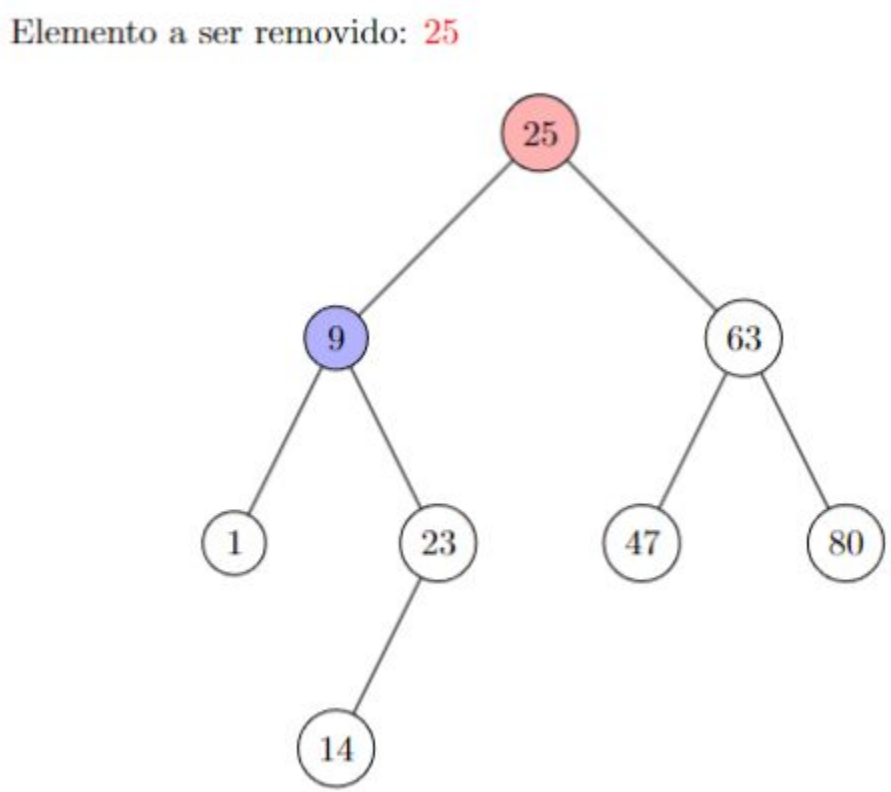
- Para a remoção de nó com dois filhos, há um algoritmo proposto por Donald Knuth e Thomas Hibbard (em quatro passos):
  a. Localize o nó com dois filhos que deve ser removido
  b. Na subárvore à esquerda, encontre o elemento mais à direita possível
  c. Substitua a informação do nó a ser removido pela informação do nó localizado no passo anterior
  d. Remova o nó localizado no segundo passo

Árvore Binária de Busca - Remoção

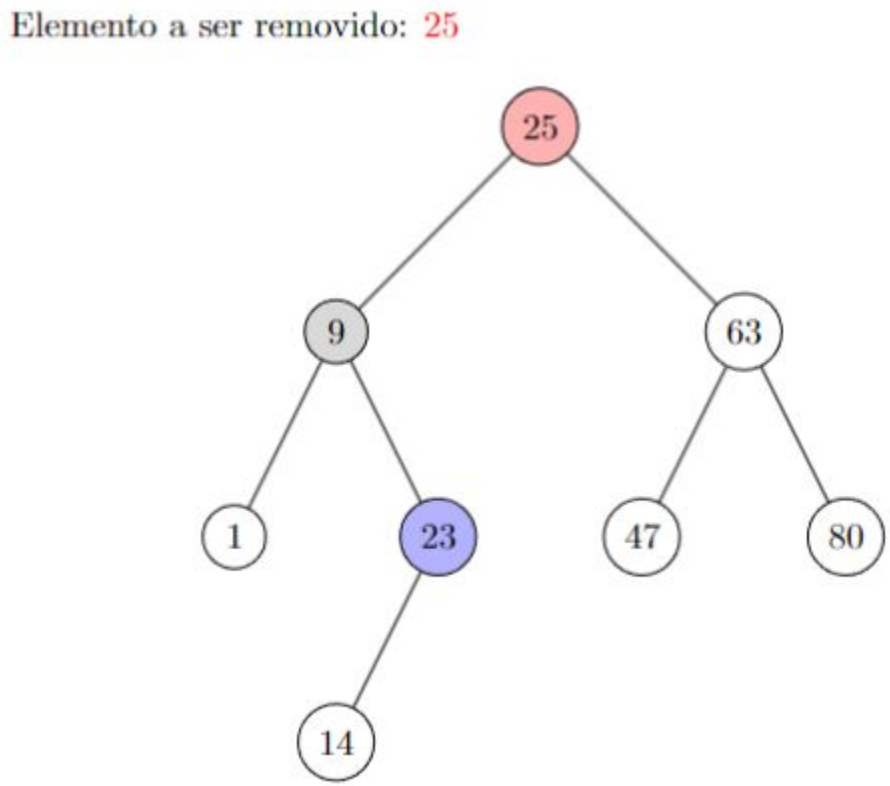
Elemento a ser removido: 25



Árvore Binária de Busca - Remoção

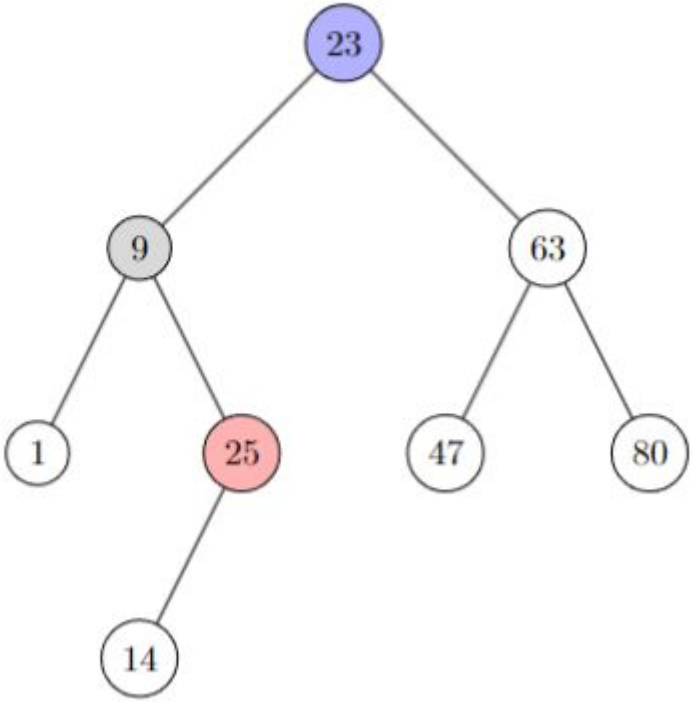


Árvore Binária de Busca - Remoção

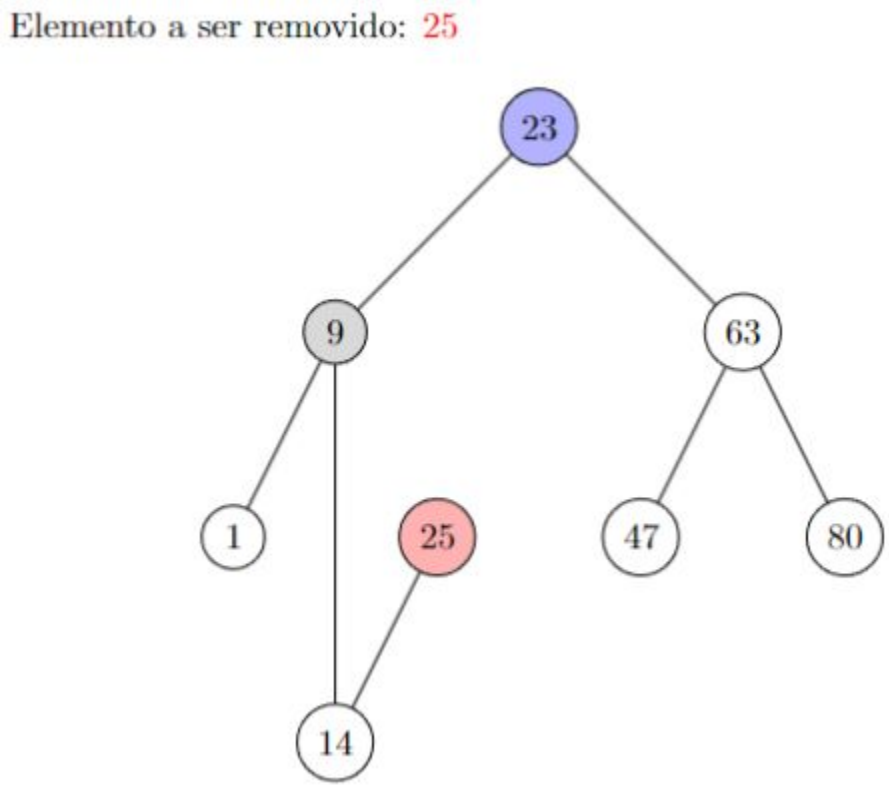


Árvore Binária de Busca - Remoção

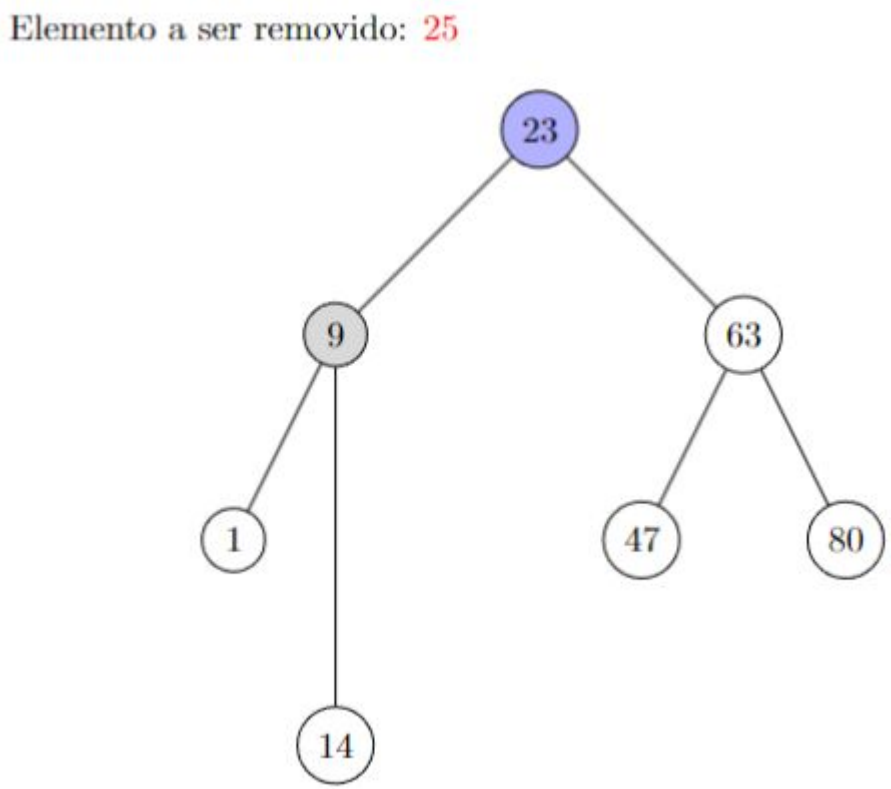
Elemento a ser removido: 25



Árvore Binária de Busca - Remoção



Árvore Binária de Busca - Remoção



Árvore Binária de Busca - Remoção

- De forma semelhante à inserção, a remoção tem complexidade O(N) no pior caso (árvore degenerada)
- Em uma árvore balanceada a complexidade é O(h), onde h é a altura da árvore
  - Ou seja, em uma árvore balanceada h = O(log N)
- Algumas implementações mantém a árvore sempre balanceadas
  - Red-black trees, árvores AVL, etc

Árvore Binária de Busca - Busca

- A busca em uma árvore binária de busca procura responder a seguinte questão
  - A informação x está armazenada em algum dos nós da árvore?
- A importância desta operação nesta estrutura é tão grande que, de fato, o nome da estrutura vem daí

Árvore Binária de Busca - Busca

- O algoritmo abaixo busca a informação x em uma árvore binária de busca:
  a. Começe no nó raiz
  b. Para cada nó não nulo:
     i. Se x está armazenado no nó, retorne verdadeiro
     ii. Se x for menor do que o valor armazenado no nó, vá para a subárvore à esquerda
     iii. Se x for maior do que o valor armazenado no nó, vá para a subárvore à direita
  c. Retorne falso

Árvore Binária de Busca - Busca

```
bool buscaRecursiva(No *no, int valor) {
    if (no == nullptr) {
        return false;
    }
    if (no->valor == valor) {
        return true;
    }
    if (valor < no->valor) {
        return buscaRecursiva(no->esquerda, valor);
    } else {
        return buscaRecursiva(no->direita, valor);
    }
}
```

Árvore Binária de Busca - Busca

- Uma variante do algoritmo retorna o ponteiro para o elemento, se encontrado, ou um ponteiro nulo, caso contrário
- A ordem de complexidade, no pior caso, é O(N)
- Em árvores balanceadas, o algoritmo é O(log N)
    - Onde o algoritmo se destaca

Árvores Binária de Busca
Travessias

Árvore Binária de Busca - Travessias

- A travessia de uma árvore é o processo de visitar cada nó uma vez
- A travessia pode ser interpretada como o processo de linearização de uma árvore
- A definição de travessia não especifica a ordem na qual os nós devem ser visitados

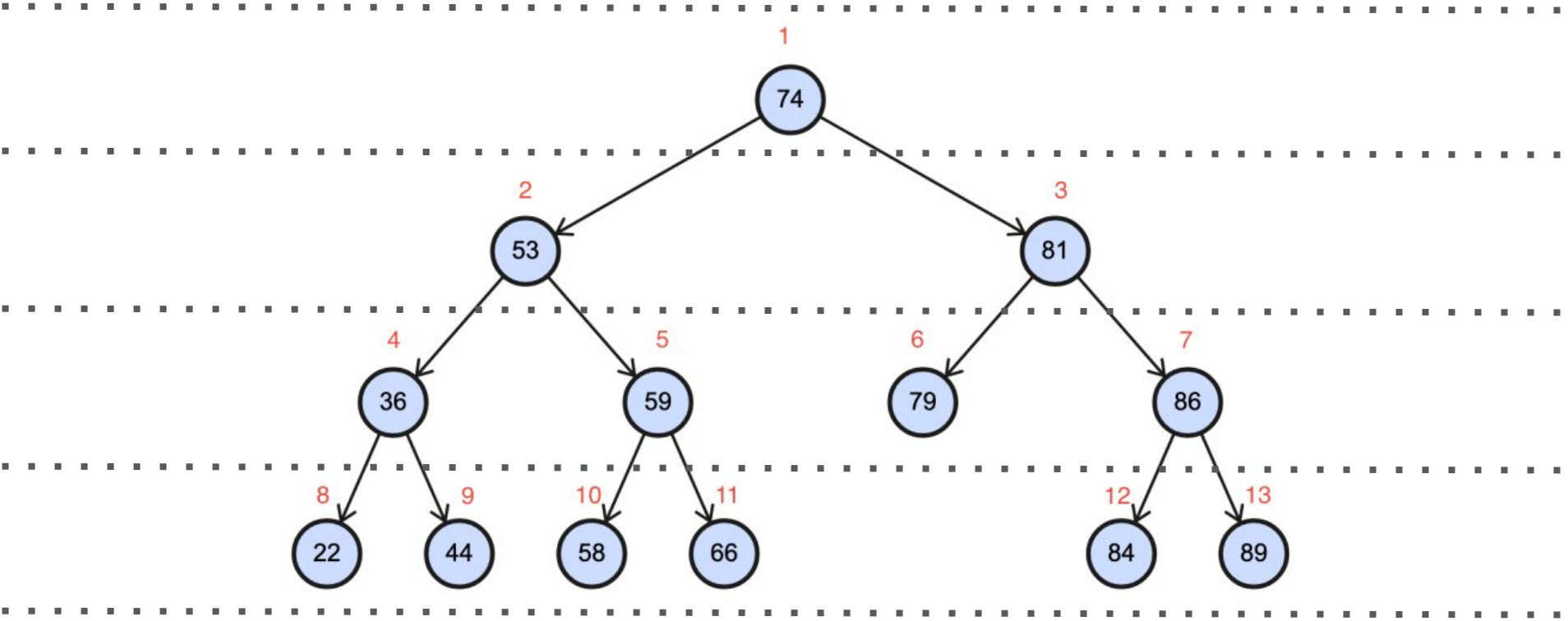
Árvore Binária de Busca - Travessias

- O número de travessias possíveis de uma árvore é igual o número de permutações de seus nós
  - Se a árvore tem n nós, terá n! travessias distintas
- Há, contudo, dois tipos especiais de travessia
  - Travessia por extensão
  - Travessia por profundidade

Árvore Binária de Busca - Travessia por Extensão

- A travessia por extensão consiste em visitar cada nó começando do nível mais baixo (ou mais alto) e seguindo para baixo (ou para cima) nível a nível, visitando todos os nós daquele nível da esquerda para a direita (ou em sentido oposto)
- Dada a natureza da travessia por extensão, sua implementação requer o auxílio de uma fila

Árvore Binária de Busca - Travessia por Extensão



Árvore Binária de Busca - Travessia por Profundidade

- A travessia por profundidade consiste em ir o mais longe possível à esquerda, retornar até o primeiro cruzamento, tomar à direita e novamente ir o máximo para a esquerda, até que todos os nós tenham sido visitados
- A travessia por profundidade pode ser implementada recursivamente
  - Também pode ser implementada iterativamente, com o auxílio de uma pilha

Árvore Binária de Busca - Travessia por Profundidade

- A definição de travessia por profundidade não especifica o momento em que o nó deve ser visitado
- Há 3 tarefas de interesse neste caso:
  a. Visitar o nó (V)
  b. Realizar a travessia da subárvore da esquerda (L)
  c. Realizar a travessia da subárvore da direita (R)

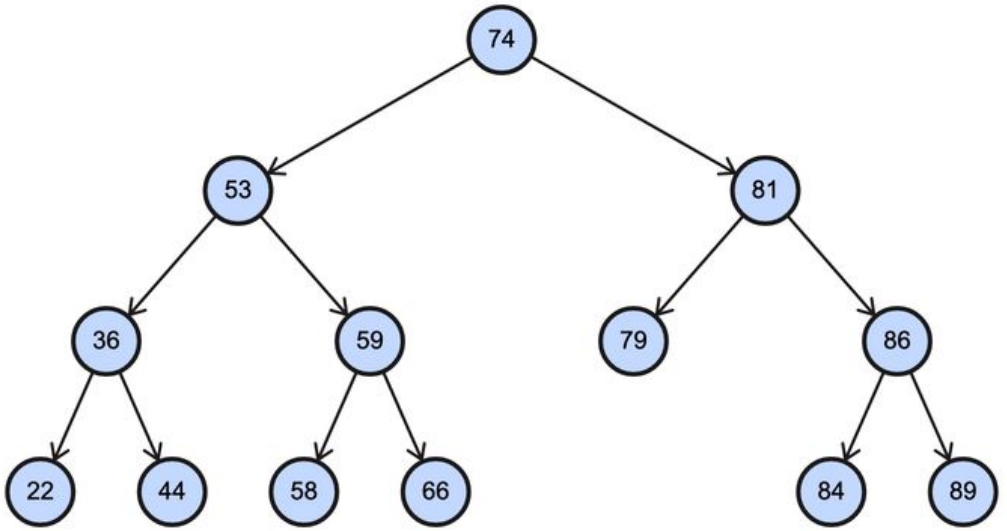
Árvore Binária de Busca - Travessia por Profundidade

- As 6 possíveis permutações destas tarefas são travessias por profundidade válidas
- Porém, algumas (para ser mais específico, três) travessias por profundidade são mais comuns:
  a. pré-ordem: VLR
  b. em-ordem: LVR
  c. pós-ordem: LRV

Árvore Binária de Busca - Travessia por Profundidade

- Pré-ordem (VLR)

```
void preordem(No *no) {
    if (no == nullptr) {
        return;
    }
    cout << no->valor << " ";
    preordem(no->esquerda);
    preordem(no->direita);
}
```

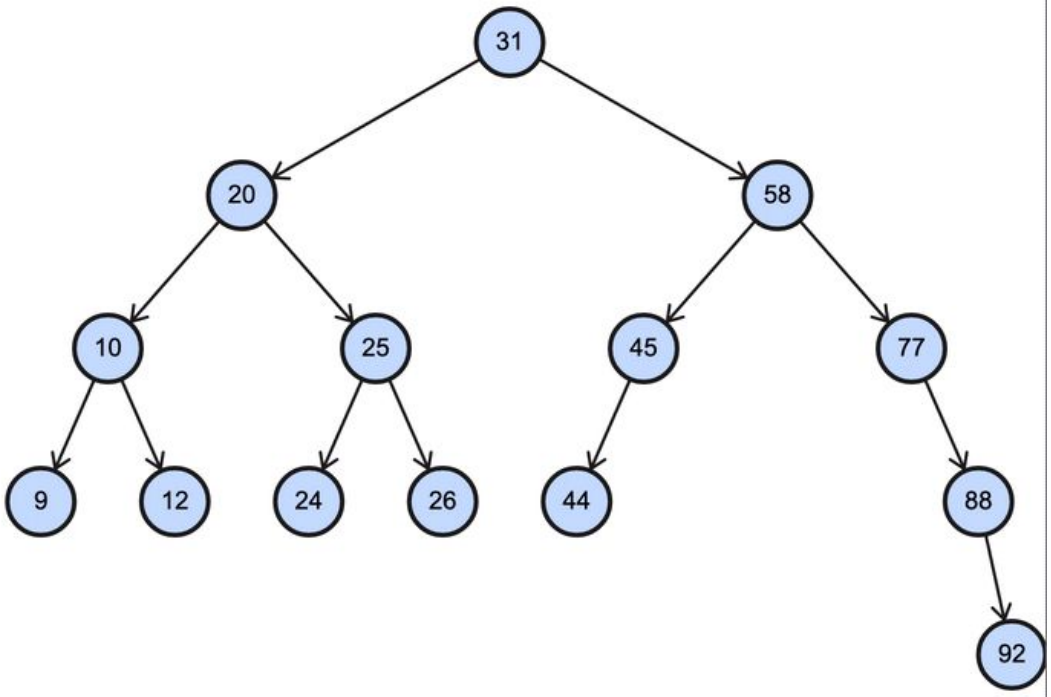


1	2	3	4	5	6	7	8	9	10	11	12	13
74	53	36	22	44	59	58	66	81	79	86	84	89

Árvore Binária de Busca - Travessia por Profundidade

- Em-ordem (LVR)

```
void emordem(No *no) {
    if (no == nullptr) {
        return;
    }
    emordem(no->esquerda);
    cout << no->valor << " ";
    emordem(no->direita);
}
```

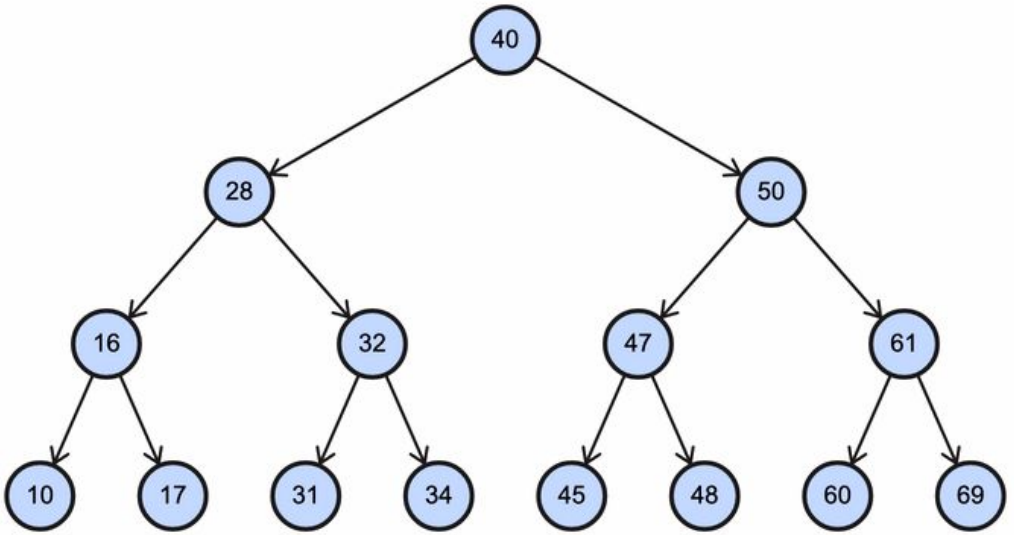


1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	10	12	20	24	25	26	31	44	45	58	77	88	92

Árvore Binária de Busca - Travessia por Profundidade

- Pós-ordem (LRV)

```
void posordem(No *no) {
    if (no == nullptr) {
        return;
    }
    posordem(no->esquerda);
    posordem(no->direita);
    cout << no->valor << " ";
}
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	17	16	31	34	32	28	45	48	47	60	69	61	50	40

Árvores Binária de Busca Balanceamento

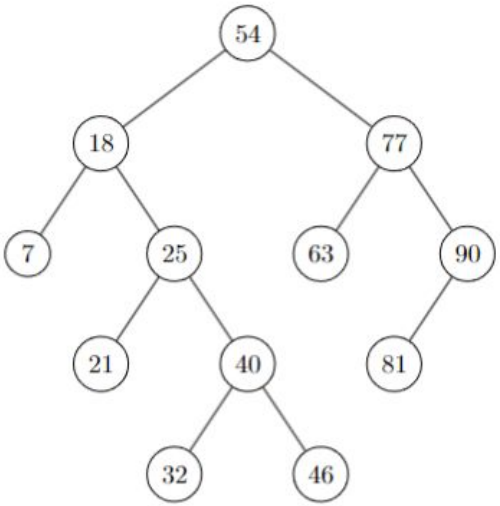
Árvore Binária de Busca - Balanceamento

- Uma árvore binária está **balanceada** se a diferença de altura das duas subárvores de qualquer nó da árvore é menor ou igual a 1
- Uma árvore binária está **perfeitamente balanceada** se ela está balanceada e todas as suas folhas se encontram em, no máximo, dois níveis distintos
- Uma árvore binária é dita **completa** se está perfeitamente balanceada e as folhas do seu último nível estão mais à esquerda possível

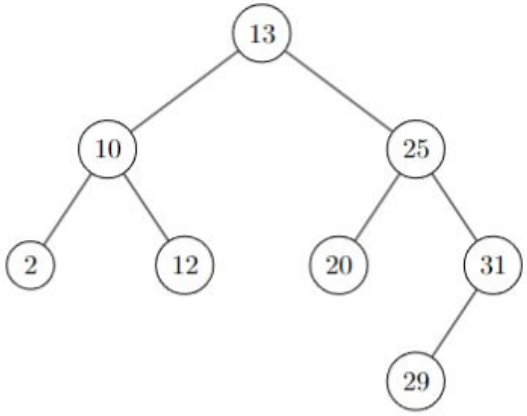
Árvore Binária de Busca - Balanceamento

- Uma árvore binária é dita **cheia** se todos os seus nós tem ou zero ou dois filhos
- A altura de uma árvore é igual ao nível máximo dentre todos os nós da árvore
- Uma árvore pode estar balanceada sem estar perfeitamente balanceada

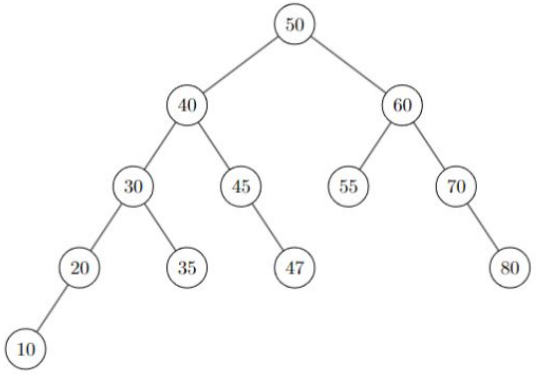
Árvore Binária de Busca - Balanceamento



Árvore Desbalanceada



Árvore Perfeitamente Balanceada



Árvore Balanceada (mas não perfeita)

Árvore Binária de Busca - Balanceamento por Inserção

- Uma maneira de garantir o balanceamento de uma árvore binária de busca é utilizar um processo de inserção controlada, de modo que ao final das inserções a árvore resultante esteja balanceada
  - Futuras inserções ou remoções podem desbalancear a árvore
- Esta estratégia é útil quando os elementos a serem inseridos são conhecidos de antemão e quando não haverão novas inserções ou remoções

Árvore Binária de Busca - Balanceamento por Inserção

- Esse algoritmo tem duas grandes desvantagens
  - É necessário armazenar os elementos antes da inserção na árvore
  - É necessário ordenar os elementos armazenados
- Se a árvore já tiver sido construída previamente, uma solução é preencher um vetor com uma travessia em-ordem, destruir a árvore e reconstruí-la
  - Essa solução apresentada remove a necessidade de ordenamento, uma vez que a travessia em-order garante que o vetor estará ordenado

Conclusão