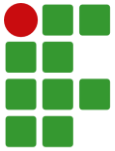


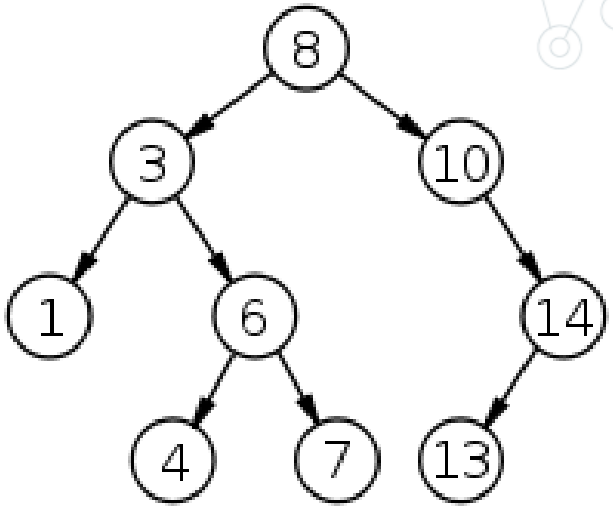
INSTITUTO FEDERAL
Sertão Pernambucano
Campus Salgueiro

Estrutura de Dados e Algoritmos com Java

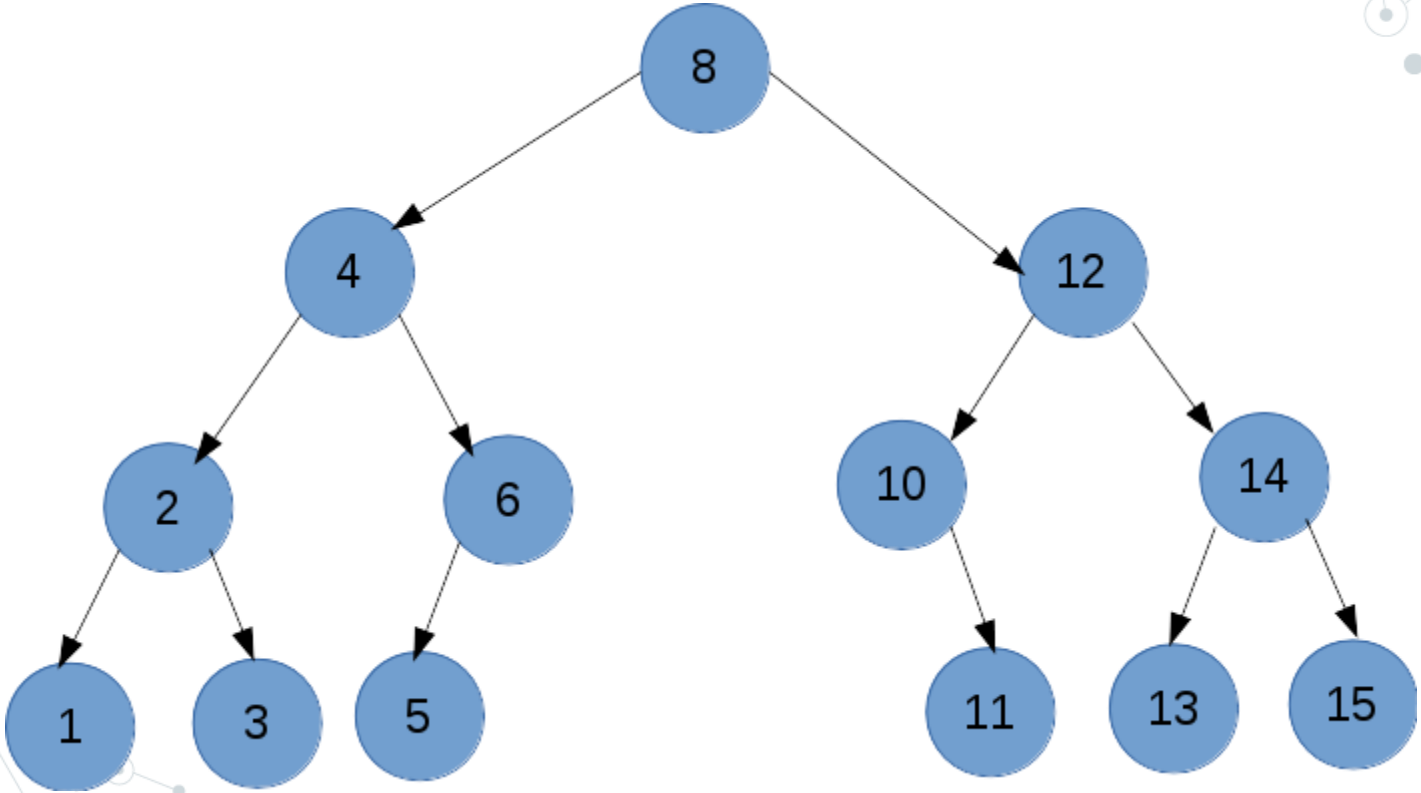
Prof. Heraldo Gonçalves Lima Junior
heraldo.junior@ifsertao-pe.edu.br

1.8. Árvore Binárias de Busca

- Também chamadas de Árvores Binárias Ordenadas.
- Para cada nó **n** da árvore, todos os valores armazenados em sua subárvore à esquerda são menores que o valor **v** armazenado em **n**, e todos os valores armazenados na subárvore à direita são maiores ou iguais a **v**.



1.8. Árvore Binárias de Busca



1.9. Implementação Dinâmica – Estrutura Básica

- Assim como todas as outras estruturas dinâmicas vistas anteriormente, na Árvore Binária, também utilizaremos a classe **No** com os seguintes atributos.

Fonte: https://joaoarthurbm.github.io/eda/posts/bst/

```
2  public class No {
3      private int valor;
4      private No esq;
5      private No dir;
6      private No pai;
7
8      public No(int valor) {
9          this.valor = valor;
10         this.esq = null;
11         this.dir = null;
12         this.pai = null;
13     }
14     //implemente os Gets e Sets
15 }
```

1.9. Implementação Dinâmica – Estrutura Básica

- Na classe **Arvore**, implementaremos todas as operações que manipularão os dados na nossa estrutura de dados.

```
2 public class Arvore {
3     public No raiz;
4
5     public Arvore() {
6         this.raiz = null;
7     }
8 }
```

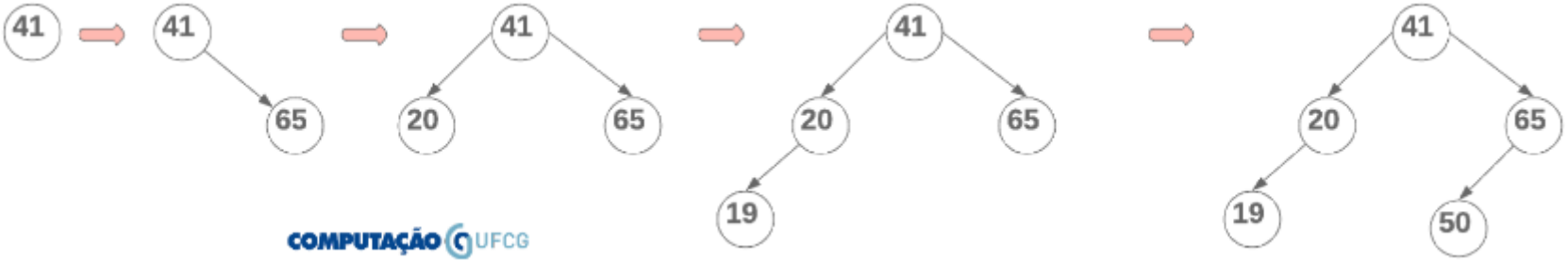
1.10. Verificando se está vazia

- Uma árvore é dita vazia quando não possui nenhum nó. Assim, se ela não possui o elemento **raiz**, será uma árvore vazia.

```
public boolean isEmpty() {
    return this.raiz == null;
}
```

1.11. Inserção

- Suponha que os seguintes elementos devem ser adicionados em uma árvore binária de pesquisa nessa ordem: **41, 65, 20, 19 e 50**.



1.11. Inserção

- A primeira verificação é feita para saber se a árvore é vazia. Se sim, o novo elemento será a **raiz**.
- Caso não seja a primeira adição precisamos fazer algumas verificações. Ou seja, caminhar árvore abaixo fazendo a comparação do elemento adicionado com os nós no caminho. **Se o elemento for menor, caminhamos para a esquerda. Se for maior, caminhamos para a direita.**

1.11. Inserção

```
public void inserir(int elemento) {
    if(isEmpty()) {
        this.raiz = new No(elemento);
    }else {
        No atual = this.raiz;
        while(atual != null) {
            if(elemento < atual.getValor()) {
                if(atual.getEsq()==null) {
                    No novo = new No(elemento);
                    atual.setEsq(novo);
                    novo.setPai(atual);
                    return;
                }
                atual = atual.getEsq();
            }else{
                if(atual.getDir()==null) {
                    No novo = new No(elemento);
                    atual.setDir(novo);
                    novo.setPai(atual);
                    return;
                }
                atual = atual.getDir();
            }
        }
    }
}
```

1.12. Inserção Recursiva

- Usaremos dois métodos para implementar a inserção recursiva.
- No primeiro, verificamos se a raiz é nula. Caso seja, Criamos um novo nó e apontamos a referência da raiz para ele. Caso contrário, chamaremos o outro método.

```
public void inserirRec(int elemento) {
    if(isEmpty()) {
        this.raiz = new No(elemento);
    }else {
        inserirRec(this.raiz, elemento);
    }
}
```

1.12. Inserção Recursiva

- O outro método, que possui o mesmo nome, recebe como parâmetro o elemento a ser inserido e o nó que está sendo analisado atualmente.
- Assim como na versão não recursiva, verifica se o elemento é menor que o valor presente no nó. Se for, vai pra esquerda, senão, vai pra direita. O que muda é somente a forma de avançarmos na árvore, que agora é feita através da chamada recursiva do método.

1.12. Inserção Recursiva

```
private void inserirRec(No no, int elemento) {
    if(elemento < no.getValor()) {
        if(no.getEsq()==null) {
            No novo = new No(elemento);
            no.setEsq(novo);
            novo.setPai(no);
            return;
        }
        inserirRec(no.getEsq(),elemento);
    }else {
        if(no.getDir()==null) {
            No novo = new No(elemento);
            no.setDir(novo);
            novo.setPai(no);
            return;
        }
        inserirRec(no.getDir(),elemento);
    }
}
```

1.13. Busca

- A busca também segue a propriedade de ordenação. Percorreremos toda a Árvore verificando se o elemento informado é igual ao valor do nó atual.
- Se não for, verificamos se ele é menor. Em caso positivo, continuamos a percorrer e verificar, agora pela esquerda.
- Se for maior, continuamos a percorrer e verificar, agora pela direita.

1.13. Busca

```
public No busca(int elemento) {
    No atual = this.raiz;
    while(atual != null) {
        if(atual.getValor() == elemento) {
            return atual;
        }else if(elemento < atual.getValor()) {
            atual = atual.getEsq();
        }else {
            atual = atual.getDir();
        }
    }
    return null;
}
```

1.14. Busca Recursiva

- A busca recursiva, assim como a inserção, também utiliza dois métodos.
- O método público seguindo a assinatura padrão e um privado auxiliar para controlar a recursão.
- A ideia é a mesma. Compara-se o elemento com o nó atual. Se for menor, há uma chamada recursiva para a sub-árvore à esquerda Se for maior, há uma chamada recursiva para a direita. O algoritmo para o nó sob análise for nulo.

1.14. Busca Recursiva

```
public No buscaRec (int elemento) {
    return buscaRec(this.raiz,elemento);
}

private No buscaRec (No no, int elemento) {
    if(no == null) {
        return null;
    }else if(elemento == no.getValor()) {
        return no;
    }else if(elemento < no.getValor()) {
        return buscaRec(no.getEsq(),elemento);
    }else {
        return buscaRec(no.getDir(),elemento);
    }
}
```

1.15. Mínimo

- O mínimo é o nó da Árvore com o menor valor entre todos.
- Como a lógica da Árvore Binária de Busca é sempre **menores à esquerda da raiz e maiores à direita**, para encontrarmos o menor elemento de todos, basta percorrer a árvore sempre à esquerda até que não haja mais nós.
- O último nó visitado será o mínimo.
- A implementação é recursiva.

1.15. Mínimo – Implementação Iterativa

```
public No minimo() {
    if(isEmpty()) {
        return null;
    }else {
        No atual = this.raiz;
        while(atual.getEsq() != null) {
            atual = atual.getEsq();
        }
        return atual;
    }
}
```

1.15. Mínimo – Implementação Recursiva

```
public No minimo() {
    if (isEmpty()) {
        return null;
    }else {
        return minimo(this.raiz);
    }
}

private No minimo(No no) {
    if(no.getEsq()==null) {
        return no;
    }else {
        return minimo(no.getEsq());
    }
}
```

1.16. Máximo

- O máximo é o nó da Árvore com o maior valor entre todos.
- Assim como no método para achar o mínimo, para encontrarmos o máximo, basta percorrer a árvore sempre pela direita até o último elemento. Este será o maior entre todos.

1.16. Máximo – Implementação Iterativa

```
public No maximo() {
    if(isEmpty()) {
        return null;
    }else {
        No atual = this.raiz;
        while(atual.getDir() != null) {
            atual = atual.getDir();
        }
        return atual;
    }
}
```

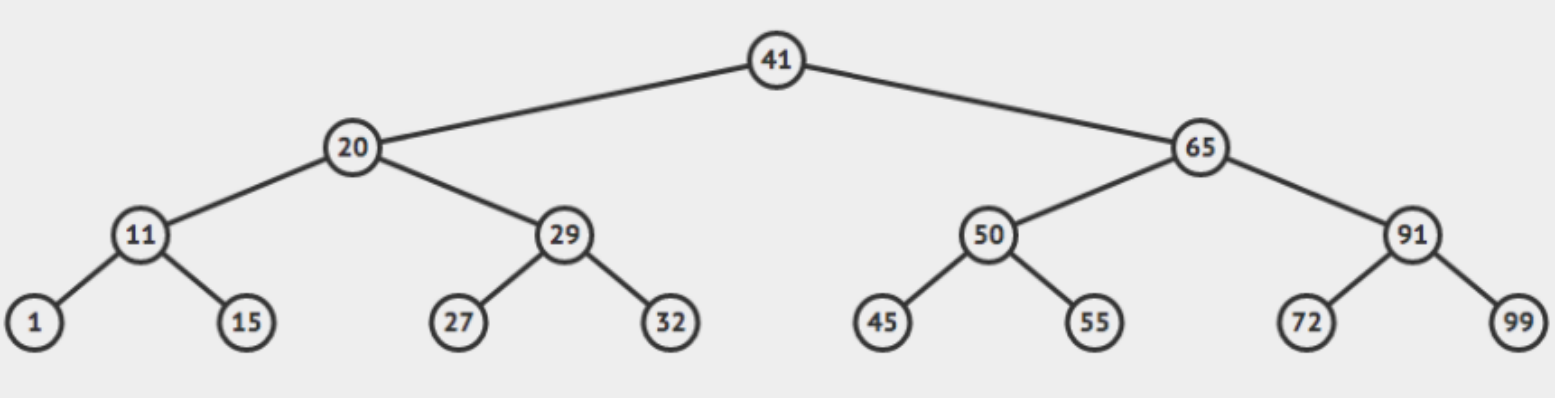
1.16. Máximo – Implementação Recursiva

```
public No maximo() {
    if (isEmpty()) {
        return null;
    }else {
        return maximo(this.raiz);
    }
}

private No maximo(No no) {
    if(no.getDir()==null) {
        return no;
    }else {
        return maximo(no.getDir());
    }
}
```

1.17. Sucessor

- Se um nó possui sub-árvore à direita, o seu sucessor é o mínimo dessa sub-árvore.



1.17. Altura

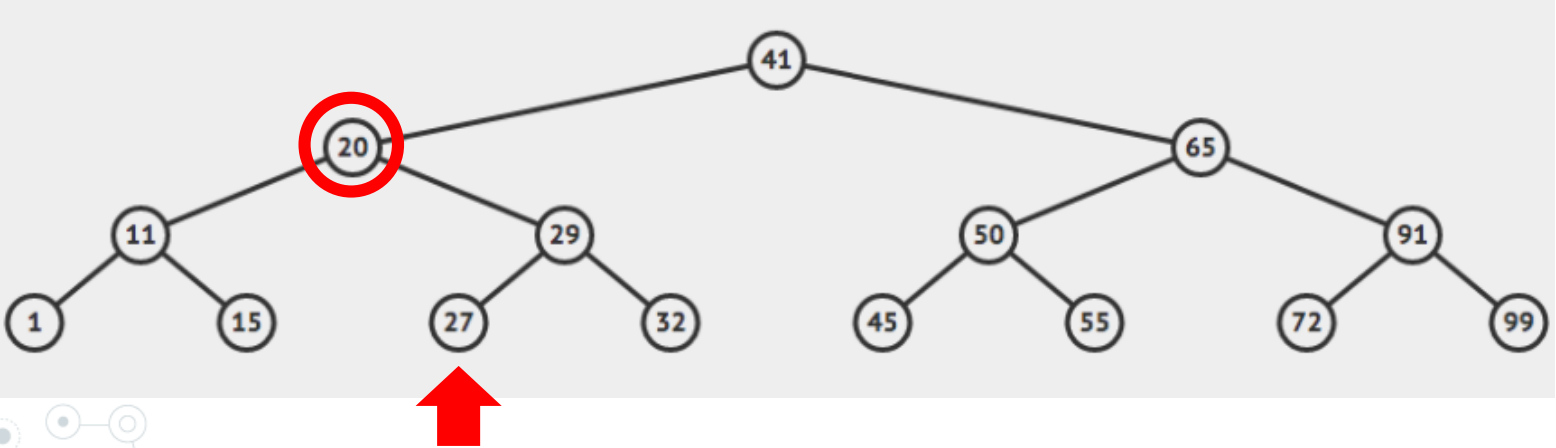
- A altura de uma Árvore é o maior caminho entre a raiz e uma de suas folhas.
- Para implementarmos, basta contarmos cada vez que um nó é analisado e calcular de maneira recursiva o máximo entre a altura da sub-árvore à esquerda e da sub-árvore à direita. Lembrando que a altura de uma árvore cuja raiz é nula é -1.

1.17. Altura

```
public int altura() {
    return altura(this.raiz);
}
public int altura(No no) {
    if(no == null) {
        return -1;
    }else {
        return 1+Math.max(altura(no.getEsq()), altura(no.getDir()));
    }
}
```

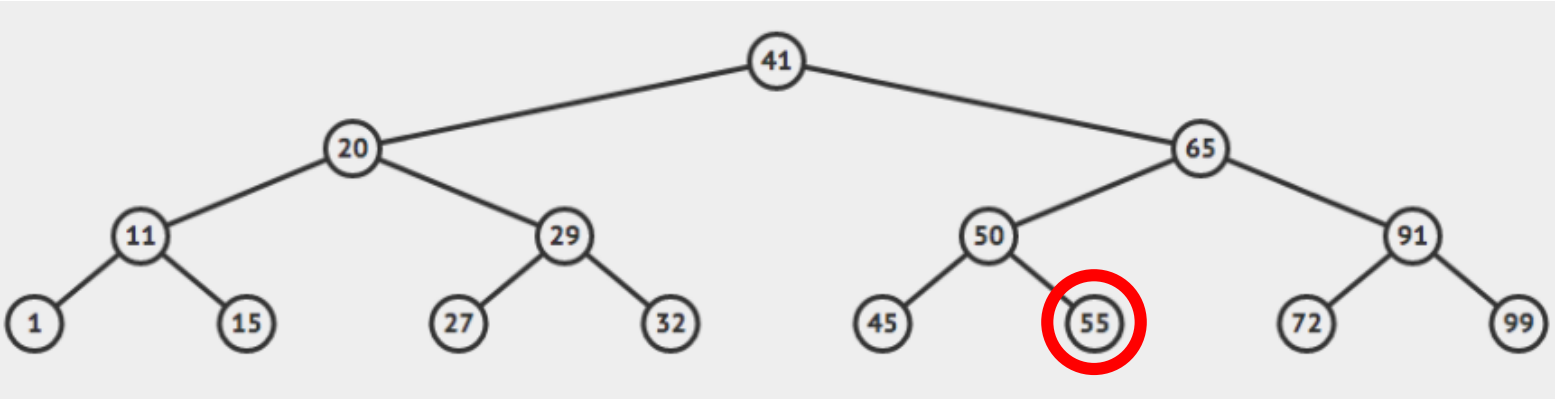
1.18. Sucessor

- **Qual é o sucessor de 20?** Se há sub-árvore à direita, basta retornamos o mínimo dessa sub-árvore. Ou seja, 27.



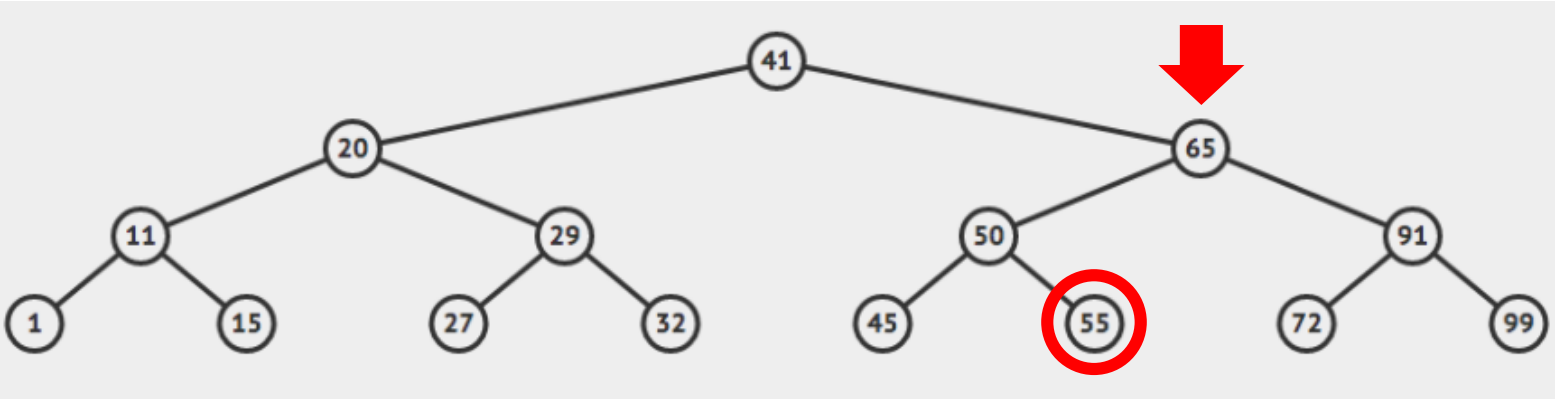
1.18. Sucessor

- E se não houver sub-árvore à direita? Por exemplo, qual é o sucessor de 55? Como não há sub-árvore à direita, precisamos subir na árvore até encontrar um elemento maior do que 55.



1.18. Sucessor

- E se não houver sub-árvore à direita? Por exemplo, qual é o sucessor de 55? Como não há sub-árvore à direita, precisamos subir na árvore até encontrar um elemento maior do que 55.

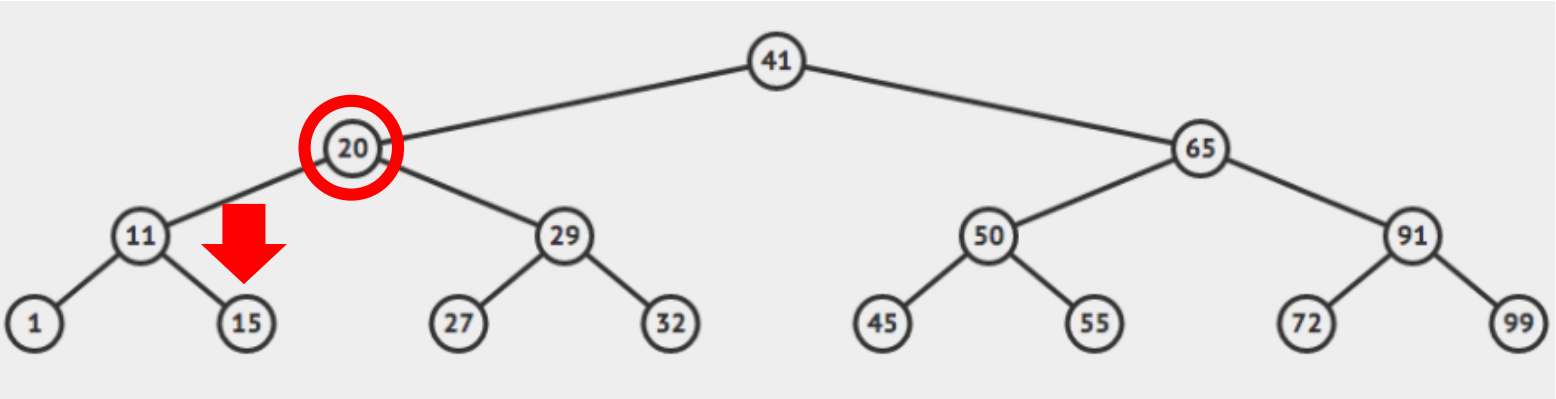


1.18. Sucessor

```
public No sucessor(No no) {
    if(no == null) {
        return null;
    }else if(no.getDir() != null) {
        return minimo(no.getDir());
    }else {
        No atual = no.getPai();
        while(atual != null && atual.getValor() < no.getValor()) {
            atual = atual.getPai();
        }
        return atual;
    }
}
```

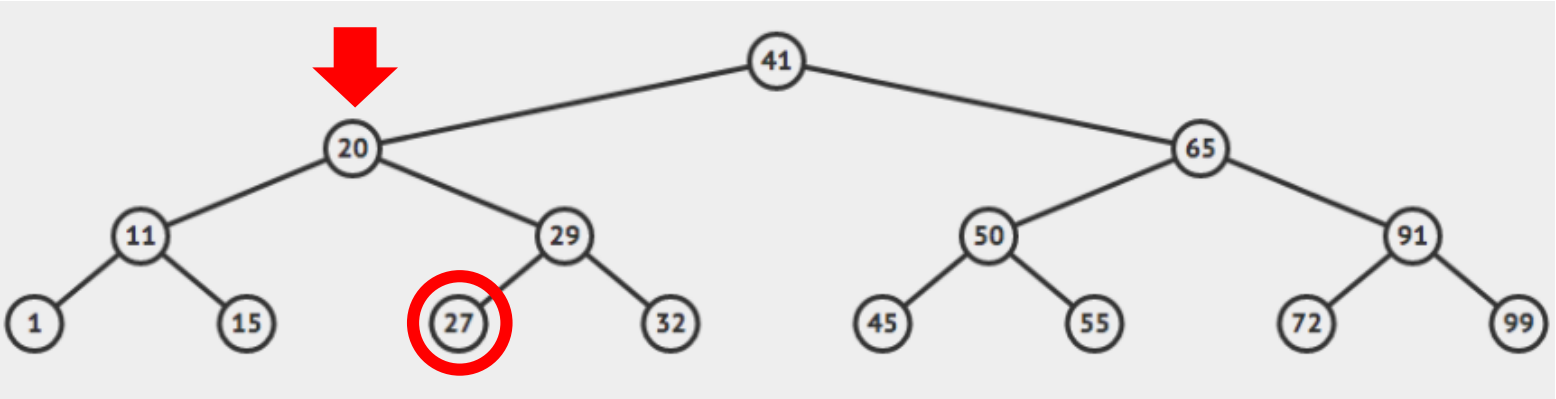
1.19. Predecessor

- A identificação do predecessor é muito semelhante ao sucessor. Se um nó possui sub-árvore à esquerda, o seu predecessor é o máximo dessa sub-árvore. Vamos a um exemplo:



1.19. Predecessor

- se não houver sub-árvore à esquerda? Por exemplo, qual é o predecessor de27? Como não há sub-árvore à esquerda, precisamos subir na árvore até encontrar um elemento menor do que 27.



1.20. Percorrendo Árvores Binárias

- De maneira geral, há duas estratégias para percorrer um grafo:
  - **em profundidade (depth-first search)**;
  - **em largura (breadth-first search)**.

- Essas estratégias são utilizadas em diversos algoritmos fundamentais em Ciência da Computação.

1.20.1 Em Profundidade

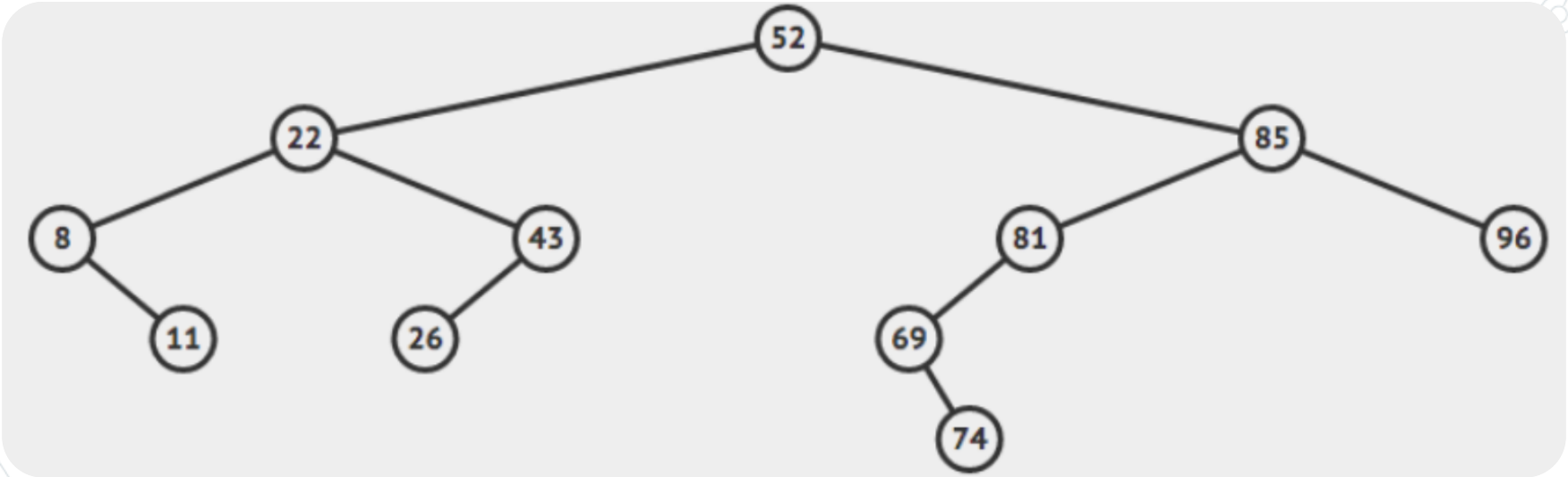
- A ideia é escolher um nó de partida e explorar todo o ramo da árvore antes de voltar e visitar os outros ramos.

- **O mantra é:**

para cada nó visitado explore o máximo à esquerda deste nó e depois o máximo à direita.

1.20.1 Pré-ordem

- Raiz –> Esquerda -> Direita



1.20.1 Pré-ordem

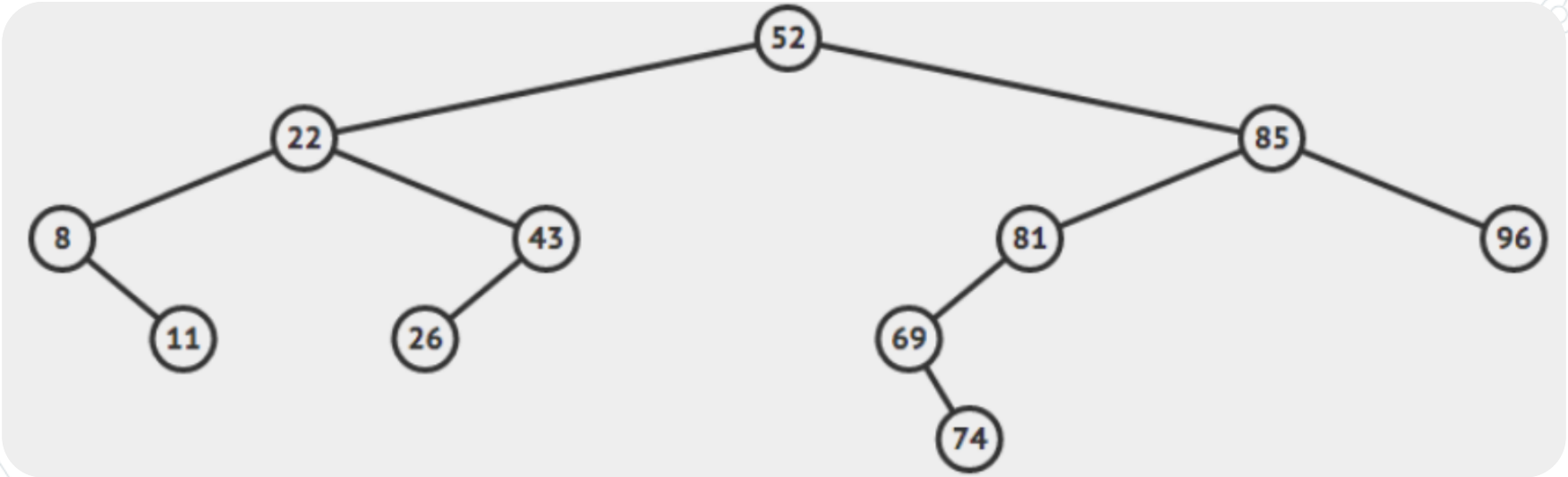
- Raiz –> Esquerda -> Direita

```
public void preOrdem() {
    preOrdem(this.raiz);
}

private void preOrdem(No no) {
    if(no != null) {
        System.out.println(no.getValor());
        preOrdem(no.getEsq());
        preOrdem(no.getDir());
    }
}
```

1.20.2 Em-ordem

- Esquerda -> Nó -> Direita



1.20.2 Em-ordem

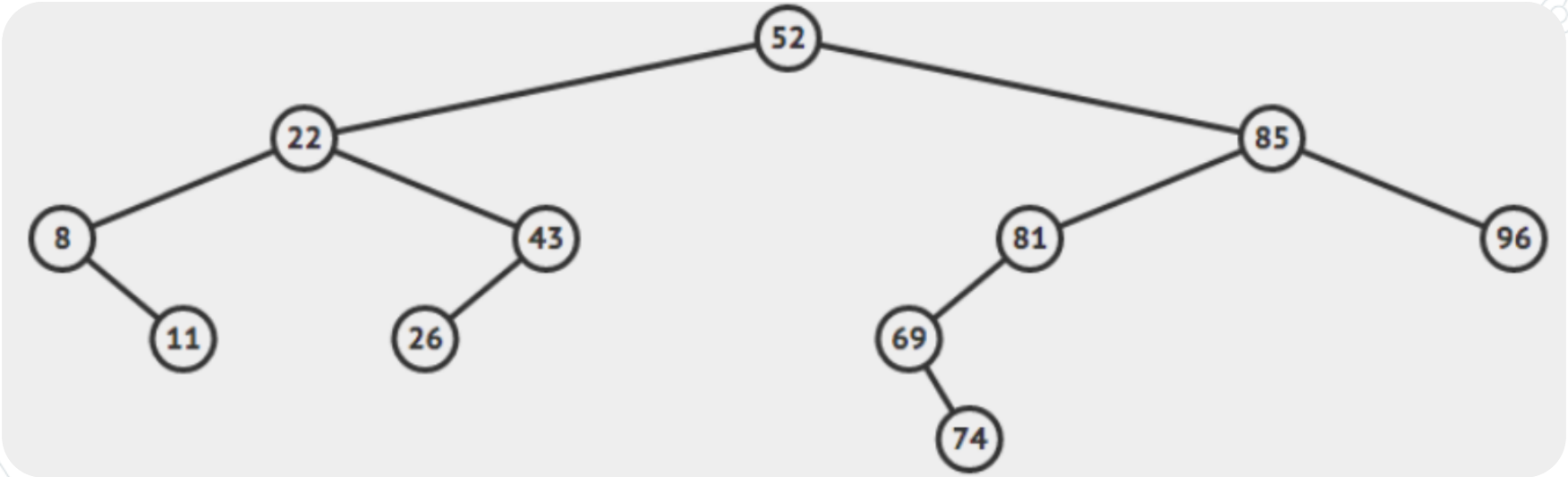
- Esquerda -> Nó -> Direita

```
public void emOrdem() {
    emOrdem(this.raiz);
}

private void emOrdem(No no) {
    if(no != null) {
        emOrdem(no.getEsq());
        System.out.println(no.getValor());
        emOrdem(no.getDir());
    }
}
```

1.20.3 Pós-ordem

- Esquerda -> Direita -> Nó



1.20.3 Pós-ordem

- Esquerda -> Direita -> Nó

```
public void posOrdem() {
    posOrdem(this.raiz);
}

private void posOrdem(No no) {
    if(no != null) {
        posOrdem(no.getEsq());
        posOrdem(no.getDir());
        System.out.println(no.getValor());
    }
}
```

CALMA, RESPIRA!

Obrigado!

Perguntas?

 heraldo.junior@ifsertao-pe.edu.br