Árvores

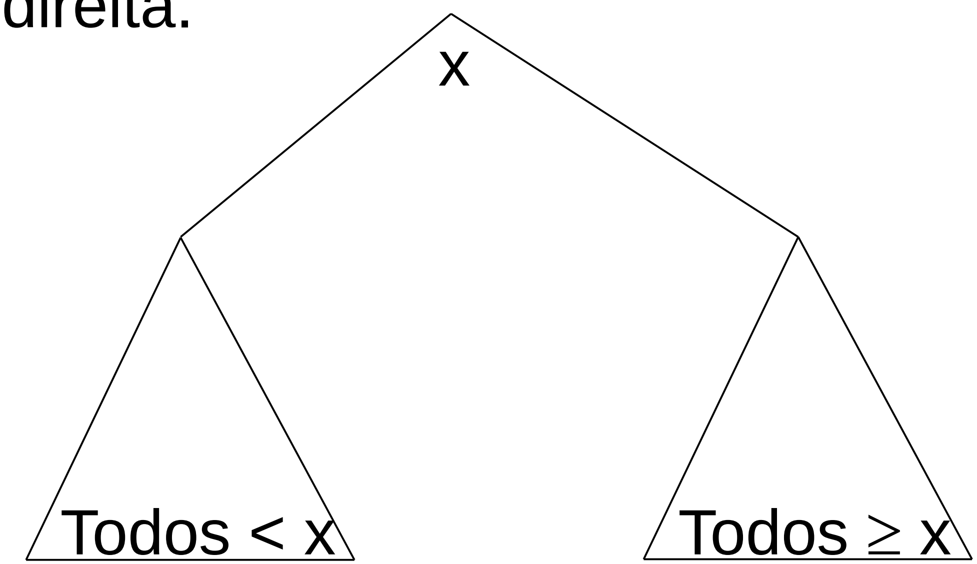
Estrutura de Dados

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

Árvore Binária de Busca

Árvore Binária de Busca

- **Definição**: uma árvore binária de busca (ABB) é uma árvore binária na qual cada nó possui uma chave comparável e que satisfaz a seguinte restrição: a chave em qualquer nó é:
  - maior do que as chaves de todos os nós da sub-árvore à esquerda e
  - menor (ou igual) às chaves de todos os nós da sub-árvore à direita.



Árvore Binária de Busca

- A representação de uma ABB é idêntica à representação de árvore binária (TAD `ArvBin`).
- Portanto, a implementação deste TAD usa o mesmo TAD `NoArv` apresentado anteriormente.
- Entretanto, algumas operações precisam ser repensadas:
  - busca
  - inserção (*)
  - remoção
- Essas operações **devem** explorar a **propriedade de ordenação das ABBs**.

Árvore Binária de Busca

- TAD `ArvBinBusca`

```
class ArvBinBusca
{
  private:
     NoArv *raiz; // ponteiro para o nó raiz da árvore
     bool auxBusca(NoArv *p);
  public:
     ArvBinBusca();
     ~ArvBinBusca();

     int getRaiz();
     bool vazia(); // verifica se a árvore está vazia
     bool busca(int val);
     void remove (int val);
     //outras operações
};
```

Busca na Árvore Binária

- Revisão: implementação da busca para AB (!). Procura a chave `ch` na árvore seguindo um percurso pré-ordem.

```
bool ArvBin::auxBusca(NoArv *p, int ch)
{
    if (p == NULL)
        return false;
    else if (p->getInfo() == ch)
        return true;
    else if (auxBusca(p->getEsq(), ch))
        return true;
    else
        return auxBusca(p->getDir(), ch)
}
```

Busca na Árvore Binária de Busca

- Como implementar essa operação para a **Árvore Binária de Busca**?
- Na ABB deve-se considerar a **propriedade** de ordenação para realizar a busca de forma mais **eficiente**.
- Ideia: compara-se o valor procurado com a informação do nó raiz:
  - Se igual, achou
  - Se menor, buscar na sub-árvore da esquerda
  - Se maior, buscar na sub-árvore da direita

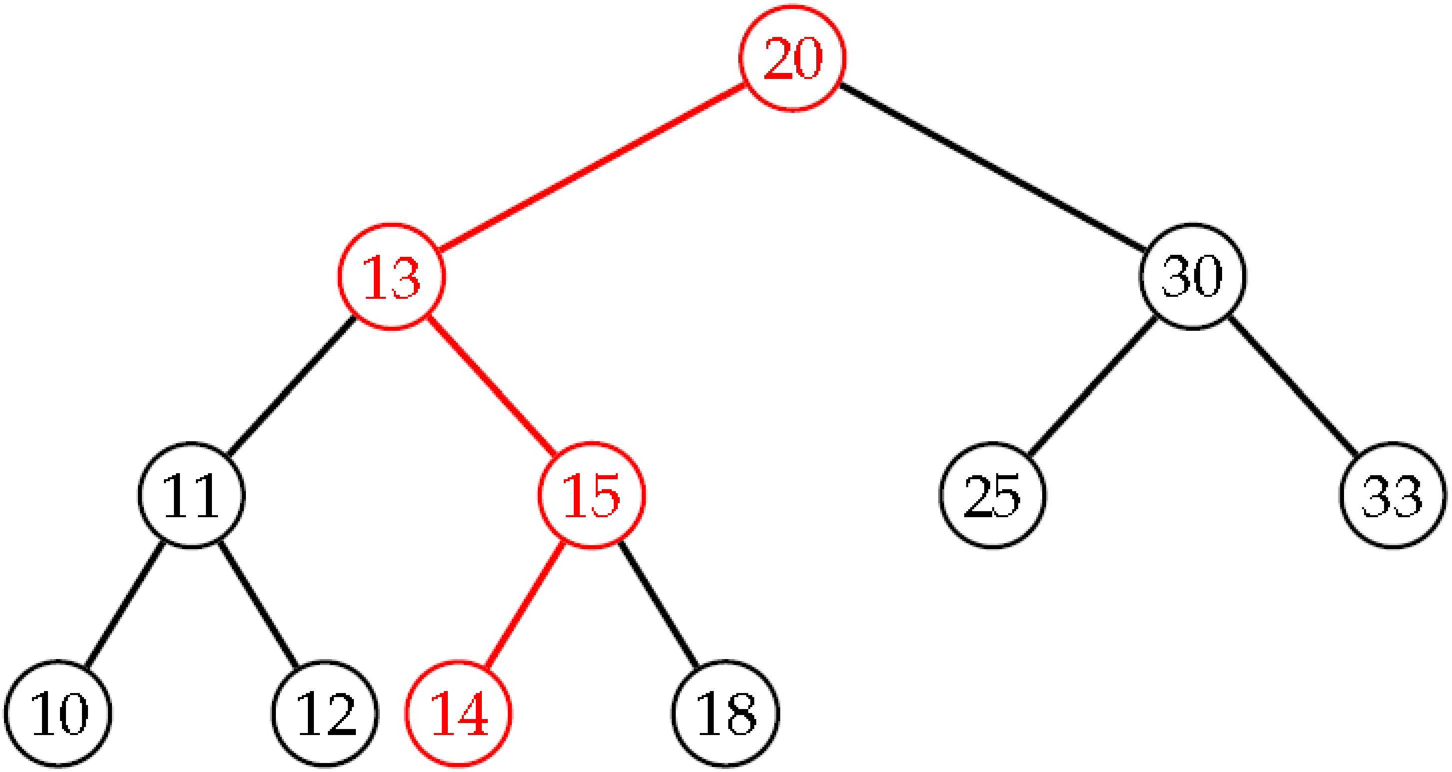
Busca na Árvore Binária de Busca

- Implementação da operação de busca.

```
bool ArvBin::busca(int x)
{
    return auxBusca(raiz, x);
}

bool ArvBin::auxBusca(NoArv *p, int ch)
{
    if (p == NULL)
        return false;
    else if (p->getInfo() == ch)
        return true;
    else if (x < p->getInfo())
        return auxBusca(p->getEsq(), ch);
    else // x > p->getInfo()
        return auxBusca(p->getDir(), ch);
}
```

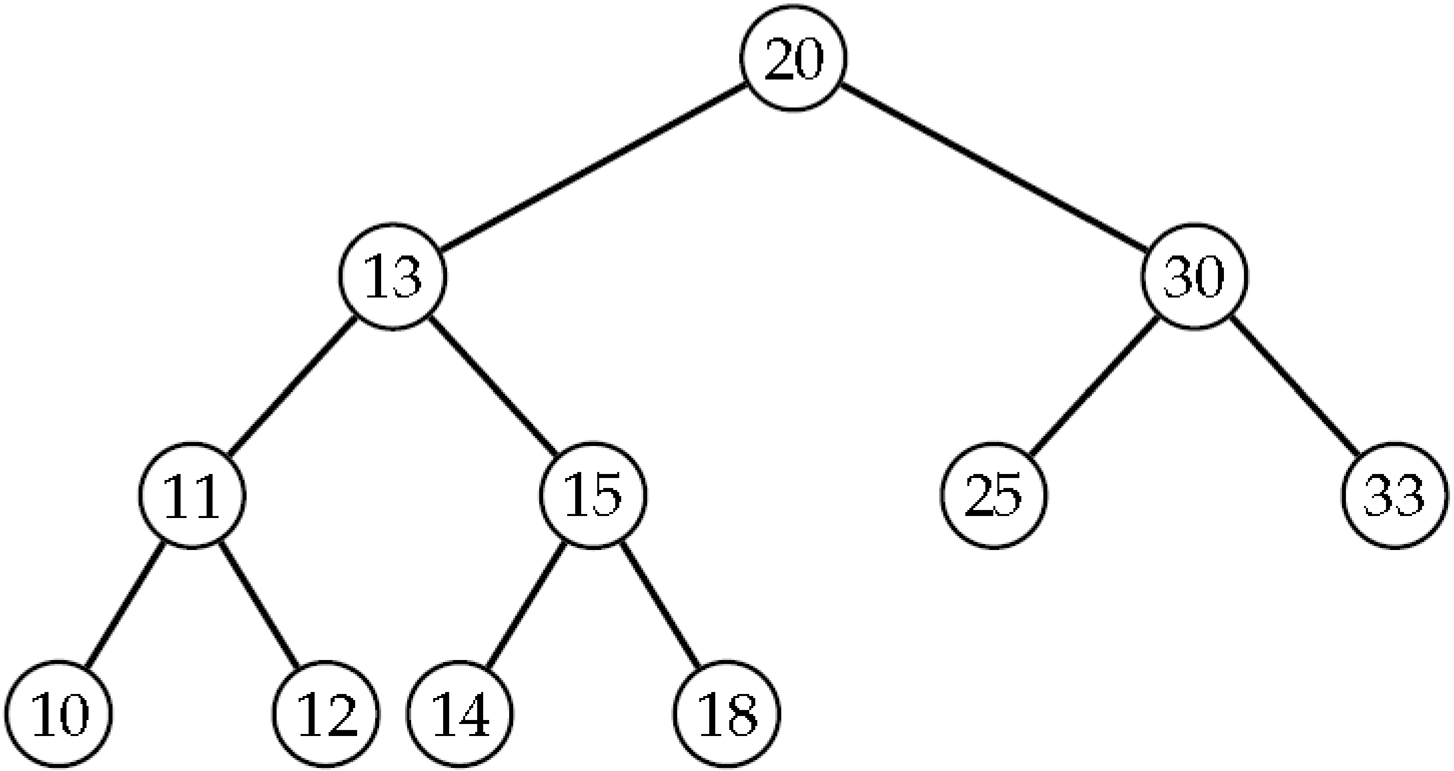
Busca na Árvore Binária de Busca

- Exemplo do caminho percorrido para encontrar a chave 14.



Busca na Árvore Binária de Busca

- Apresentar os caminhos percorridos para buscar as seguintes chaves: 18, 11, 10, 25, 35, 7

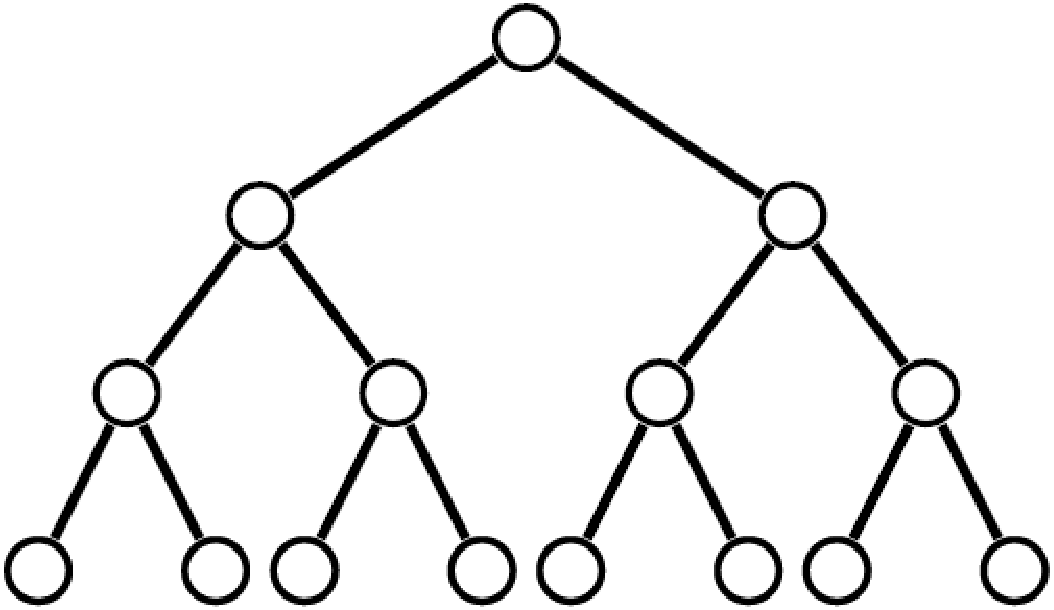


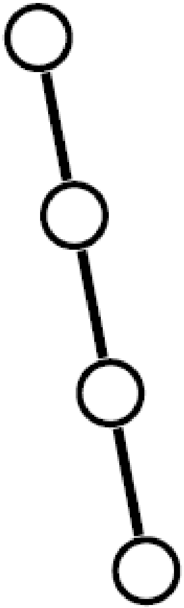
Árvore Binária de Busca

- O tempo de execução dos algoritmos em ABB depende da forma das árvores, que por sua vez, depende da ordem na qual as chaves são inseridas.
- No melhor caso, tem-se uma árvore de n nós perfeitamente balanceada com $log_2(n)$ nós entre a raiz e as folhas.
- No pior caso, tem-se uma árvore com n nós entre a raiz e as folhas.

Árvore Binária de Busca

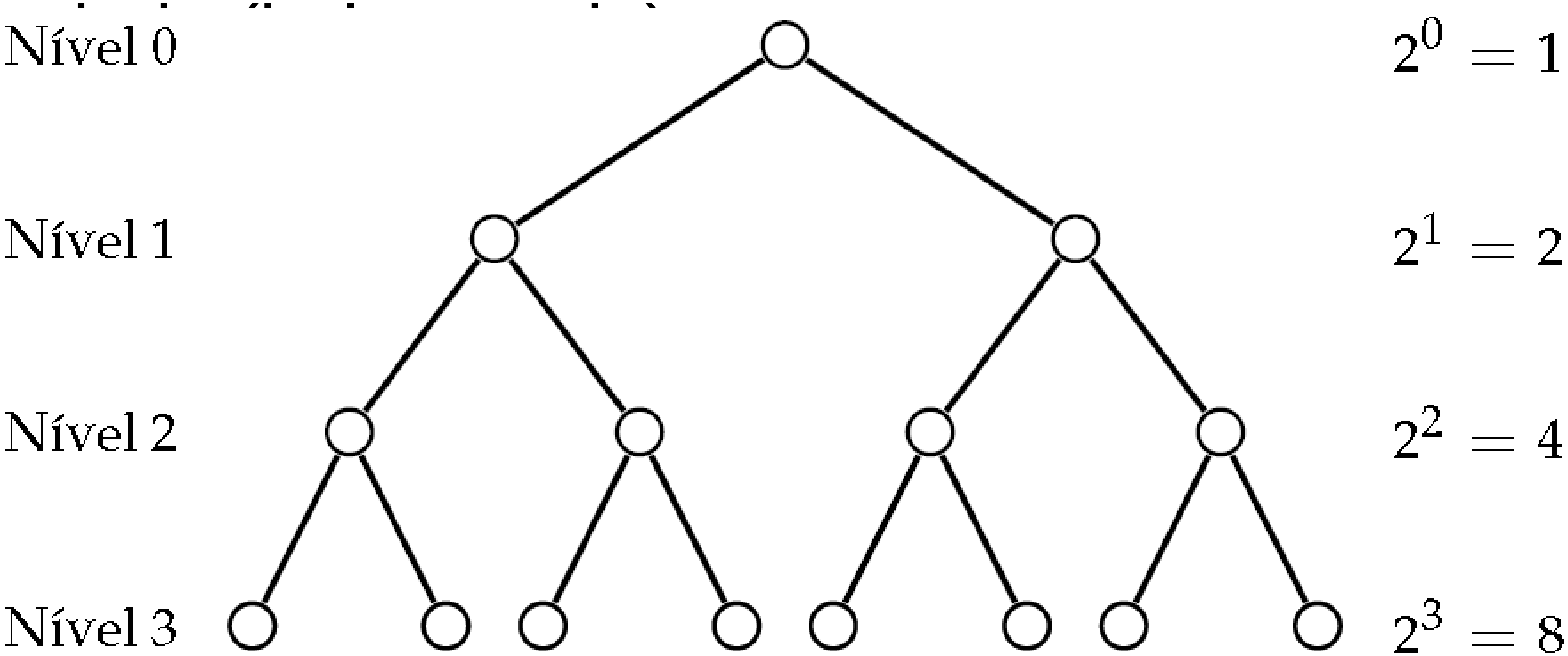
- Complexidade: o número de comparações realizado na busca é proporcional à altura h da árvore, isto é, $O(h)$.





Árvore Binária de Busca

- Qual é a altura h de uma árvore binária com n nós?
- Altura h = maior nível.
- Nível $k \rightarrow 2^k$ nós.
- Melhor situação é quando a árvore binária é



Árvore Binária de Busca

- Para uma árvore binária cheia:
  - Nível $k \rightarrow 2^k$ nós.
  - Propriedade: o número de nós de um nível k qualquer é igual a 1 mais a soma de todos os nós dos níveis anteriores.

Nível	Número de nós	Número de nós nível anterior
0	$2^0 = 1 = 1 + 0$	0
1	$2^1 = 2 = 1 + 1$	1
2	$2^2 = 4 = 1 + 3$	3
3	$2^3 = 8 = 1 + 7$	7
$\dots$	$\dots$	$\dots$
k	$2^k = 1 + \sum_{i=0}^{k-1} 2^i$	$\sum_{i=0}^{k-1} 2^i$
$\dots$	$\dots$	$\dots$
$h + 1$	$2^{h+1} = 1 + \sum_{i=0}^{h} 2^i$	$\sum_{i=0}^{h} 2^i = n$

Árvore Binária de Busca

- Assim, tem-se:

$$2^{h+1} = 1 + \sum_{i=0}^{h} 2^i$$

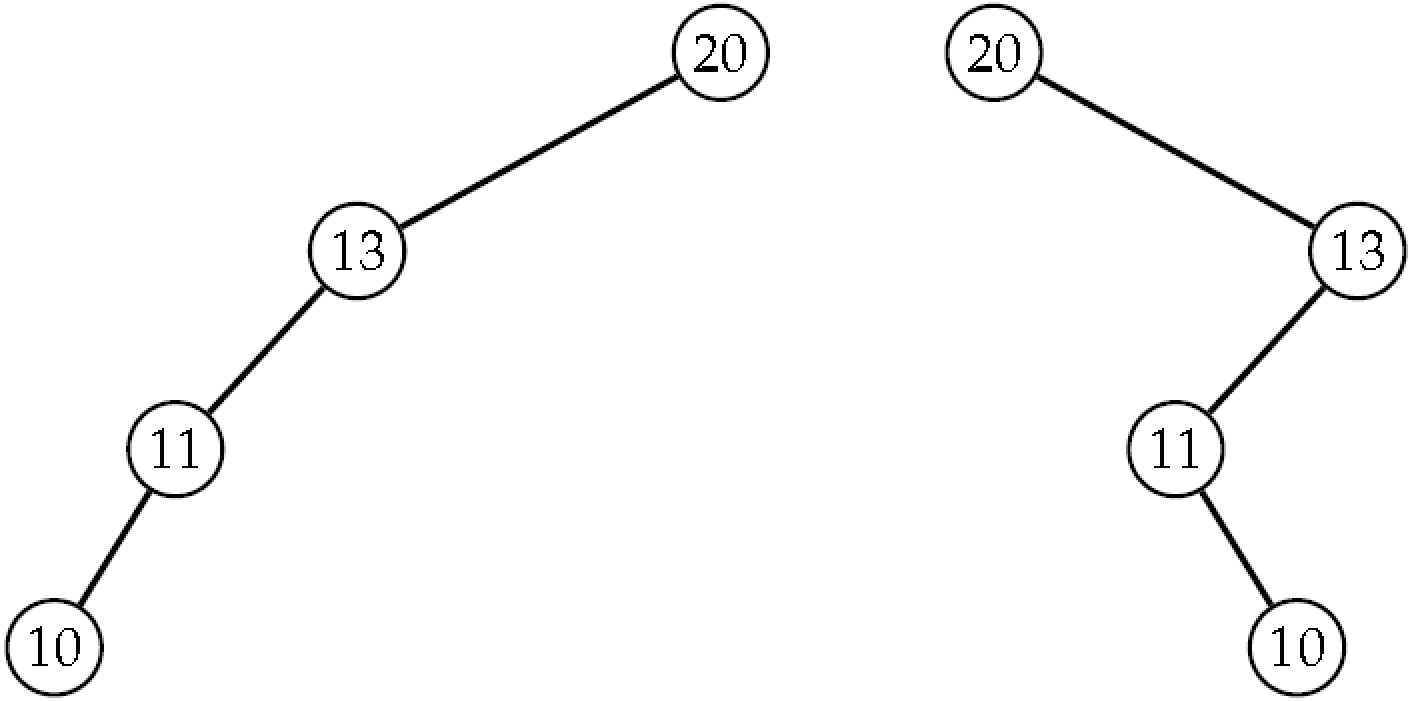
- Note que $\sum_{i=0}^{h} 2^i$ é o número total de nós da árvore, que é n.
- Portanto

$$2^{h+1} = 1 + n$$
$$log_2\left(2^{h+1}\right) = log_2(1 + n)$$
$$h + 1 = log_2(1 + n)$$
$$h = log_2(1 + n) - 1$$

- Eliminando as constantes, conclui-se que, em uma árvore binária de busca **balanceada**, a altura h é proporcional a $log_2(n)$.

Árvore Binária de Busca

- Pior caso: árvore binária **desbalanceada** (degenerada).
- Neste caso, a altura h é proporcional ao número de nós n.



Árvore Binária de Busca

- Em resumo, sobre a complexidade na ABB
- o número de comparações realizadas nas operações é proporcional à altura h da árvore: $O(h)$.
- A altura da ABB é no mínimo $log_2(n)$ e no máximo n; onde n é o número de nós da ABB.
- Portanto, a complexidade das operações em uma árvore binária de busca é $log_2(n)$ na **melhor situação** (balanceada) e $O(n)$ na **pior situação** (degenerada), onde n é o número de nós da ABB.

Árvore Binária de Busca

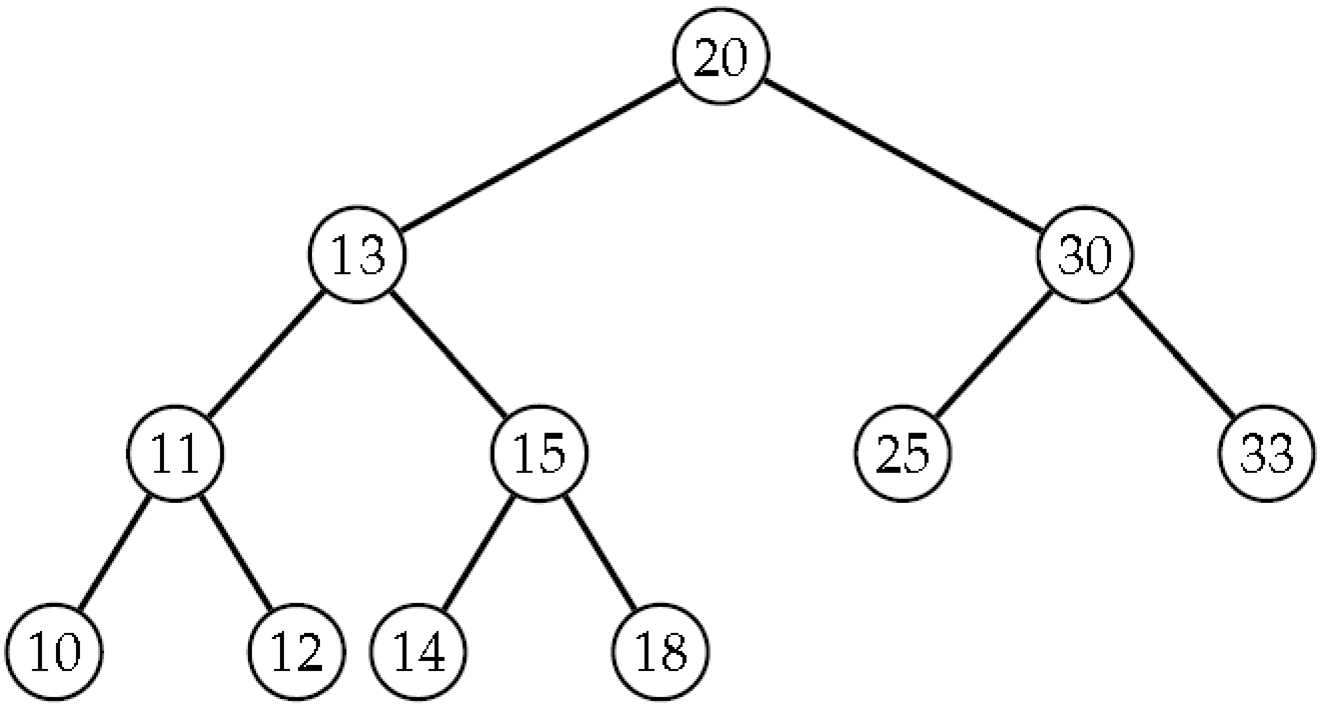
Inserção

- Para inserir um novo nó com o valor y, deve-se percorrer a árvore buscando a chave y, até encontrar o nó que será o seu pai, isto é, o nó que não apresentar filho na sequência natural do percurso (ou filho = `NULL`).
- Em seguida, basta incluir um nó **folha** contendo y.

Árvore Binária de Busca

Inserção

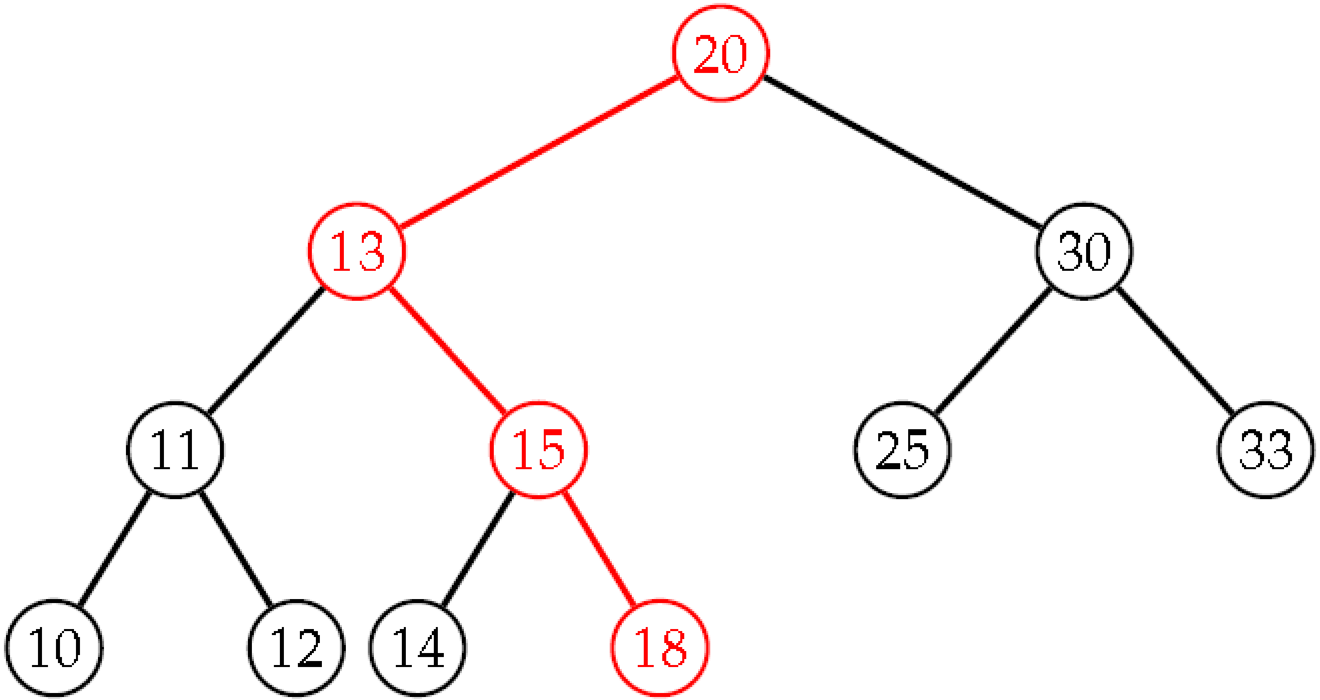
- Exemplo: inserir 17 e 27 na árvore abaixo.



Árvore Binária de Busca

Inserção

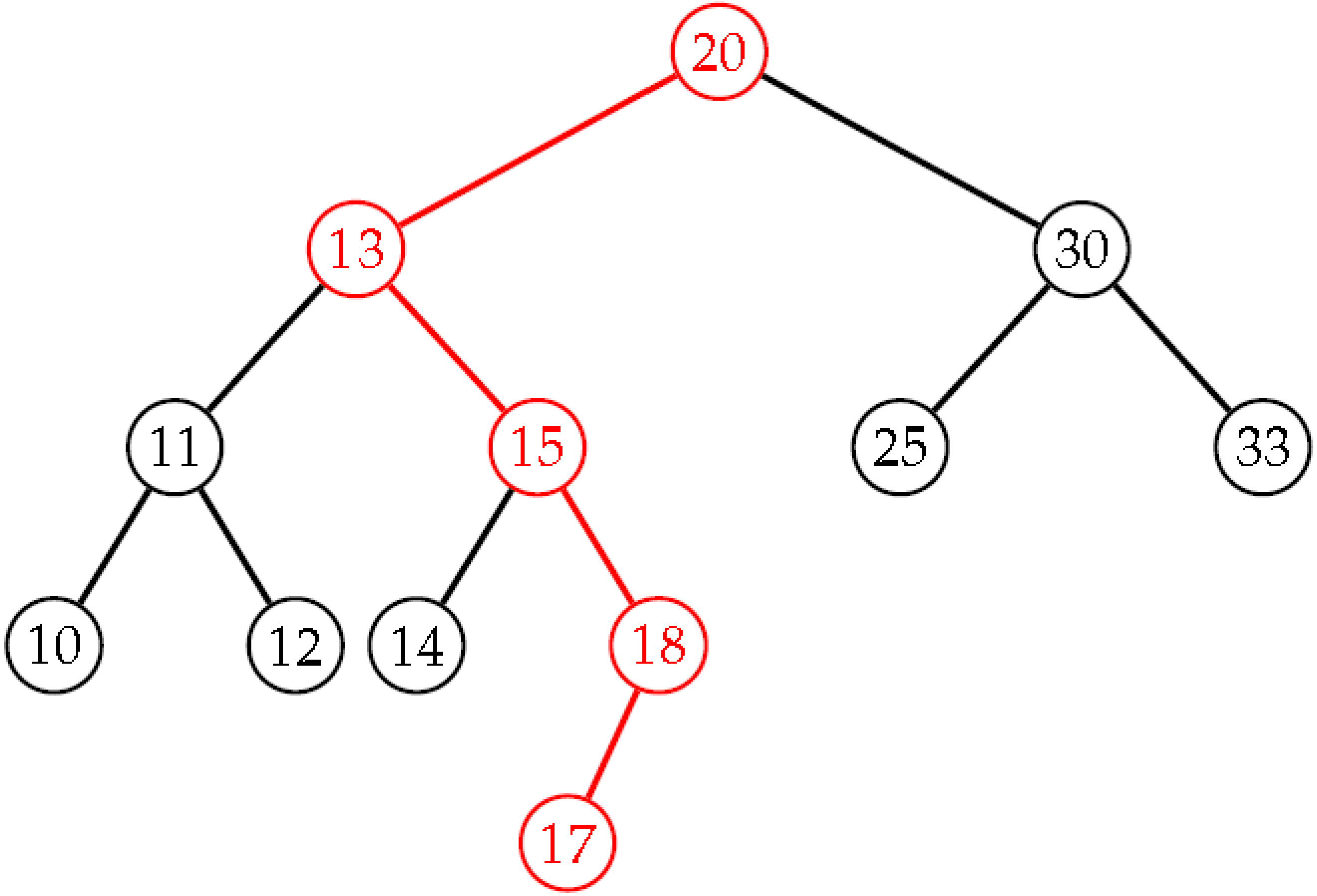
- Exemplo: inserir o valor 17.



Árvore Binária de Busca

Inserção

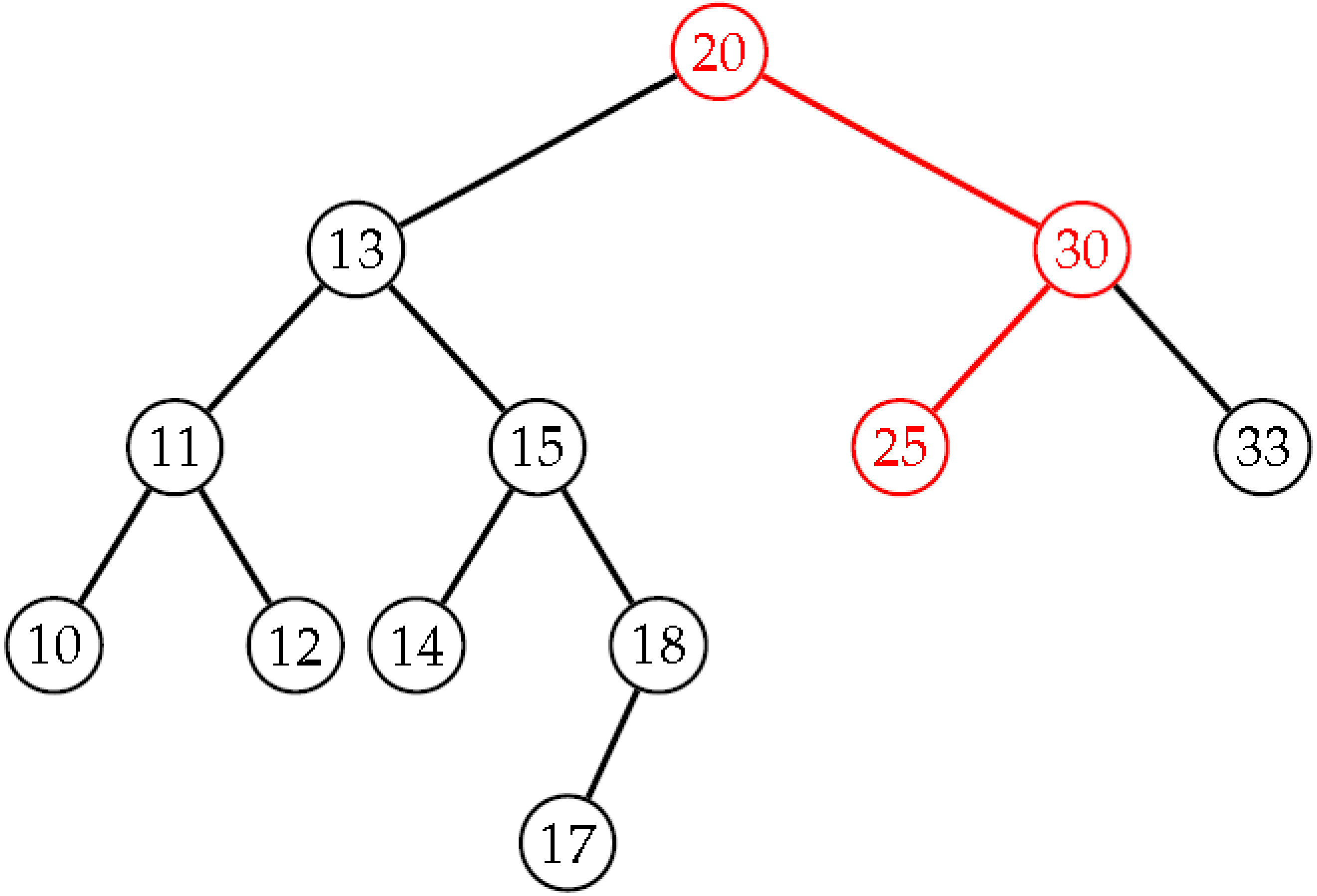
- Exemplo: inserir o valor 17.



Árvore Binária de Busca

Inserção

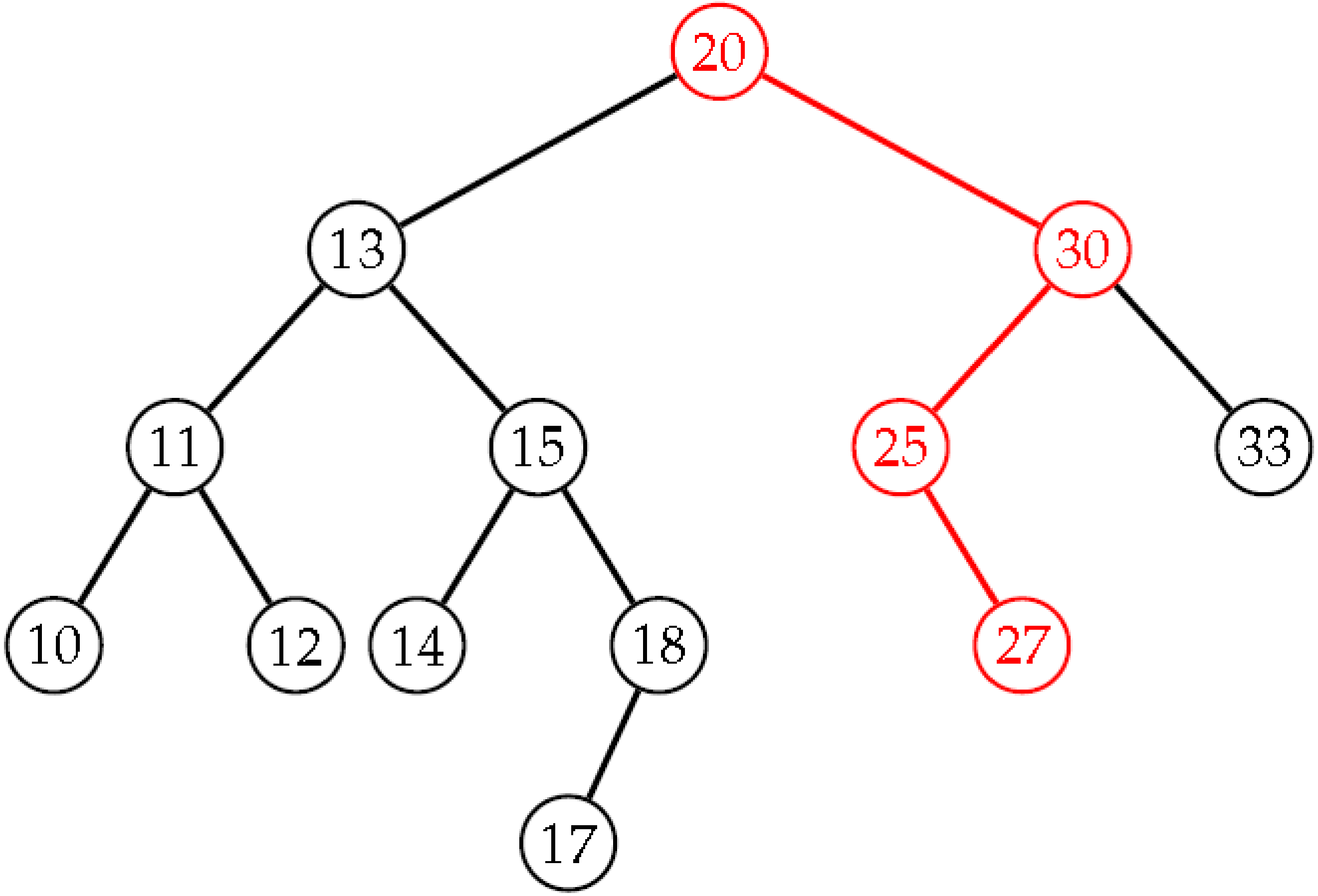
- Exemplo: inserir o valor 27.



Árvore Binária de Busca

Inserção

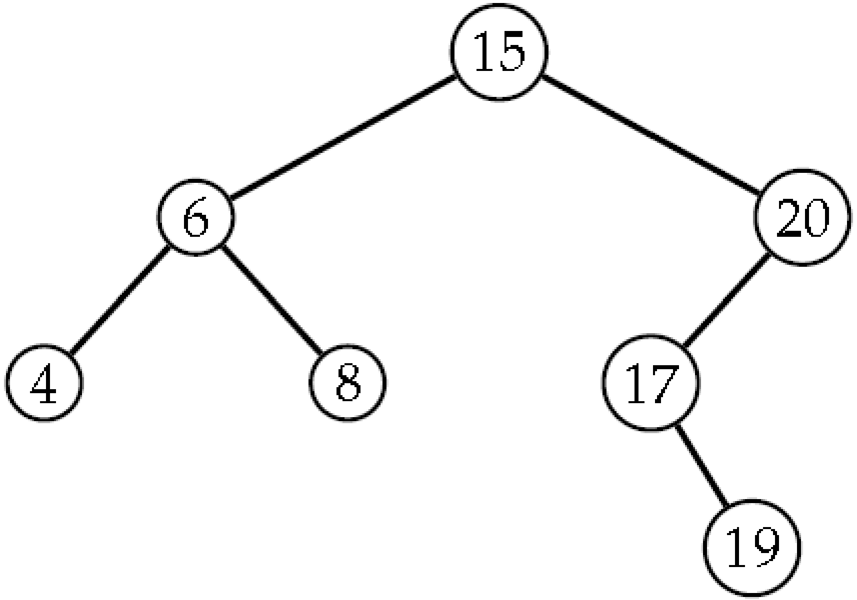
- Exemplo: inserir o valor 27.



Árvore Binária de Busca

Inserção

- Considere uma ABB inicialmente vazia.
- Inserir chaves 15, 6, 20, 17, 8, 4 e 19.
- Qual o resultado após essas inserções?



Árvore Binária de Busca

Inserção

```
class ArvBinBusca
{
  private:
      //...
      NoArv* auxInsere(NoArv *p, int val);
      //...
  public:
      //...
      void insere(int val);
      //...
};
```

- A operação pública tem a seguinte implementação:

```
void ArvBinBusca::insere(int val)
{
  raiz = auxInsere(raiz, val);
}
```

Árvore Binária de Busca

Inserção

```
NoArv* ArvBinBusca::auxInsere(NoArv *p, int val)
{
    if(p == NULL)
    {
        p = new NoArv();
        p->setInfo(val);
        p->setEsq(NULL);
        p->setDir(NULL);
    }
    else if(val < p->getInfo())
        p->setEsq(auxInsere(p->getEsq(), val));
    else
        p->setDir(auxInsere(p->getDir(), val));
    return p;
}
```

Árvore Binária de Busca

Menor e maior

- Pode-se implementar a busca pelo menor (maior) de forma recursiva ou iterativa.
- Descrição **recursiva** para encontrar o **menor**:
  - Se a sub-árvore à esquerda da raiz é vazia, então o menor está na raiz.
  - Se a sub-árvore à esquerda da raiz não é vazia, então o menor está na sub-árvore à esquerda.
- Para encontrar o maior valor, basta trocar esquerda por direita na descrição acima.

Árvore Binária de Busca

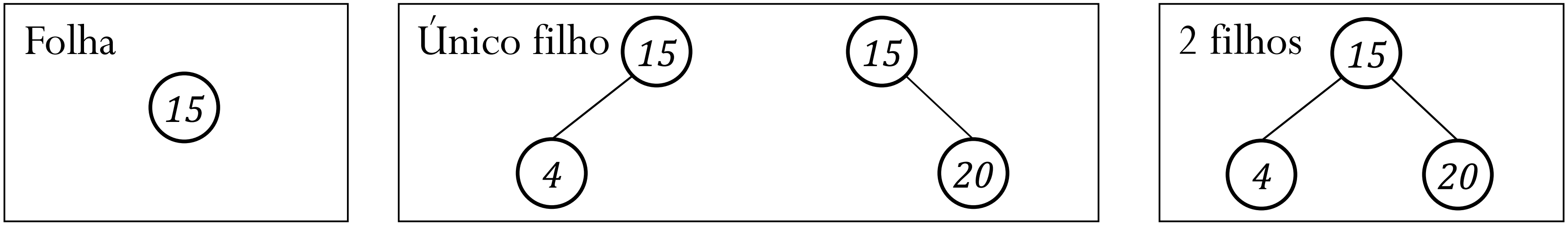
Menor e maior

```
int ArvBinBusca::minimo()
{
   return auxMin(raiz);
}

int ArvBinBusca::auxMin(NoArv *p)
{
   if(p != NULL)
   {
      if(p->getEsq() == NULL)
         return p->getInfo();
      else
         return auxMin(p->getEsq());
   }
}
```

Árvore Binária de Busca

Remoção

- A operação mais complicada em uma ABB é a de remover um determinado nó.
- Quando deseja-se remover um nó qualquer da árvore, este nó pode ser:
  - uma folha;
  - um nó que possui apenas 1 filho;
  - ou um nó que possui os 2 filhos.



- Antes de estudar o caso mais geral, considere o caso de remover o menor valor da ABB.

Árvore Binária de Busca

Remove menor

```
void ArvBinBusca::removeMin()
{
   raiz = auxRemoveMin(raiz);
}

NoArv *ArvBinBusca::auxRemoveMin(NoArv *p)
{
   if(p != NULL)
   {
      if (p->getEsq() == NULL) {
         NoArv *r = p->getDir();
         delete p;
         return r;
      }
      p->setEsq(auxRemoveMin(p->getEsq()));
      return p;
   }
}
```

Árvore Binária de Busca

Remoção

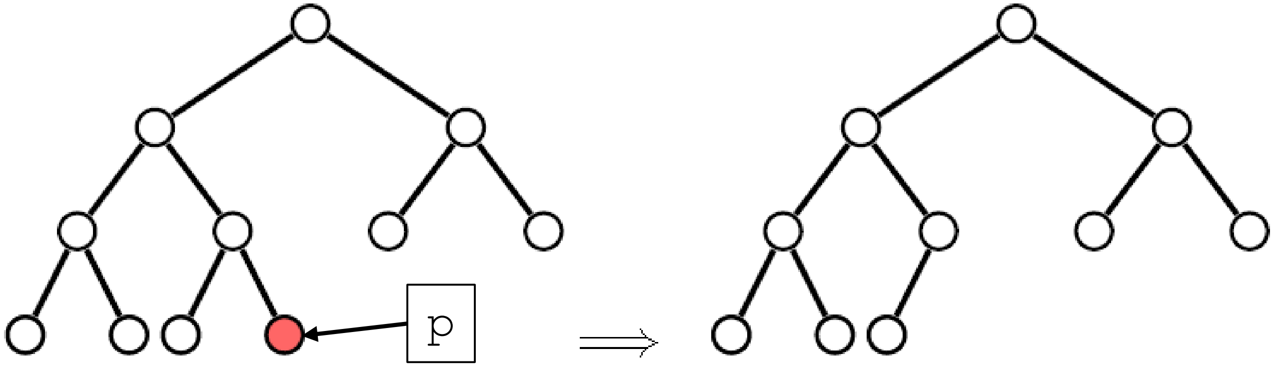
- Para remover um nó qualquer da árvore, serão criadas 3 funções para tratar de cada um dos casos mencionados.
- Ideia geral para remover um nó com a chave `x`:
  - se a árvore é vazia, retorna `NULL`
  - senão se `x<p->getInfo()`, remove `x` na SAE
  - senão se `x>p->getInfo()`, remove `x` na SAD
  - senão se o nó é folha, então `removeFolha()`
  - senão se nó possui apenas 1 filho, então `removeNo1Filho()`
  - senão (o nó possui 2 filhos), `removeNo2Filhos()`
- Primeiro, serão apresentadas as implementações das funções que removem o nó de acordo com o caso em questão, isto é, `removeFolha()`, `removeNo1Filho()` e `removeNo2Filhos()`.

Árvore Binária de Busca

Remove nó folha

- função `removeFolha(NoArv *p)` recebe um ponteiro para um nó que é uma folha e então remove o nó da árvore. O ponteiro do pai deve ser ajustado para `NULL` e o nó é removido. E preciso ter certeza de que `p` aponta para um nó folha.

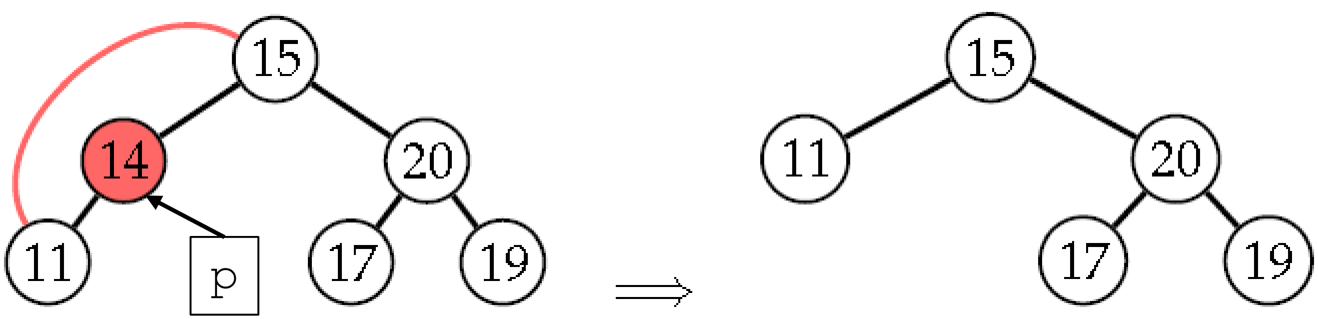
```
void ArvBinBusca::removeFolha(NoArv *p)
{
   delete p;
   return NULL;
}
```



Árvore Binária de Busca

Remove nó com 1 filho

- Ponteiro do pai do nó a ser removido é reajustado para apontar para o filho do nó a ser removido.
- Ou seja, o pai vai apontar para o “neto” (que passa a ser filho).
- Desse modo, descendentes do nó em questão são elevados em 1 nível.
- Exemplo: remover o nó 14.



Árvore Binária de Busca

Remove nó com 1 filho

- A função `removeNo1Filho()` é usada para remover um nó `p` que aponta para um nó que tem um único filho.
- É preciso **ter certeza** de que `p` só possui um filho para usar essa função.

```
void ArvBinBusca::removeNo1Filho(NoArv *p)
{
   NoArv *aux;
   if(p->getEsq() == NULL)
      aux = p->getDir(); //filho único é da direita
   else
      aux = p->getEsq(); //filho único é da esquerda
   delete p;
   return aux;
}
```

Árvore Binária de Busca

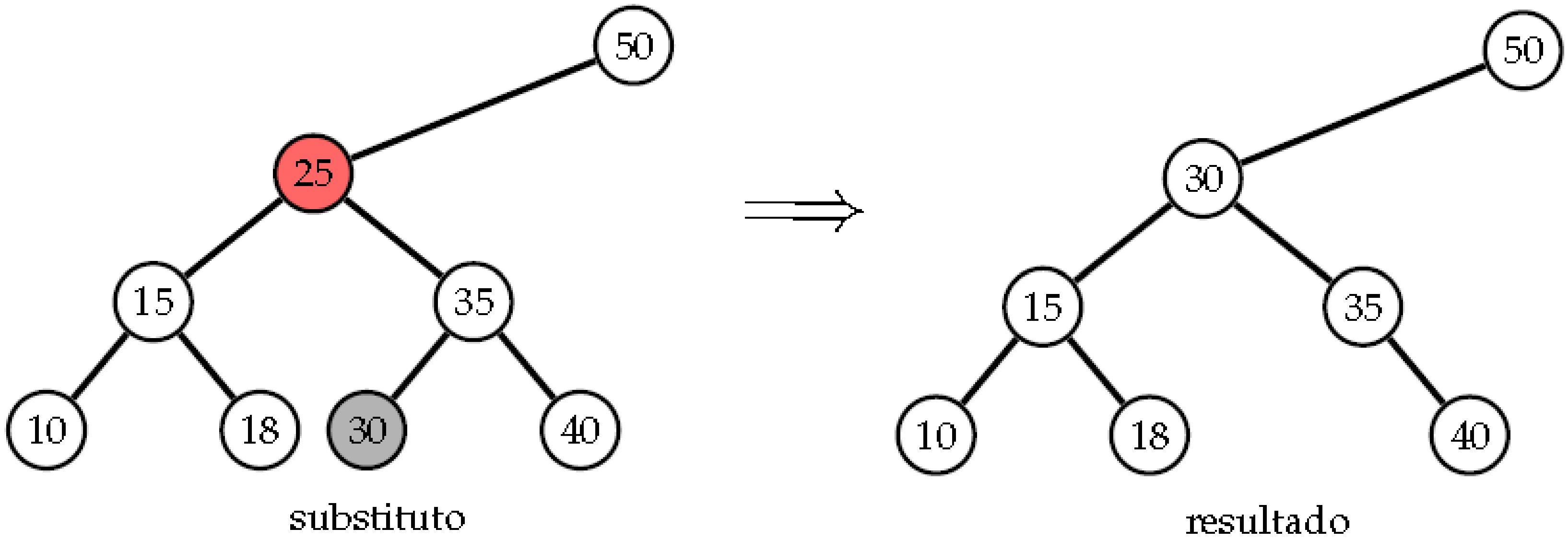
Remove nó com 2 filhos

- Se o nó a ser removido tem 2 filhos:
  - **Remover fazendo cópia.**
  - Remover fazendo junção (merge). Não será visto.
- Remoção por cópia (Thomas Hibbard e Donald Knuth): substituir o nó a ser removido pelo menor nó de sua sub-árvore à direita e ”ajustar ponteiros”.
- Etapas da remoção por cópia:
  1. Buscar substituto (menor nó da sua sub-árvore à direita).
  2. Trocar a informação do nó a ser removido com a do substituto.
  3. Remover o substituto (essa remoção faz o ajuste dos ponteiros).

Árvore Binária de Busca

Remove nó com 2 filhos

- Exemplo. Excluir nó com valor 25.



Árvore Binária de Busca

Remove nó com 2 filhos

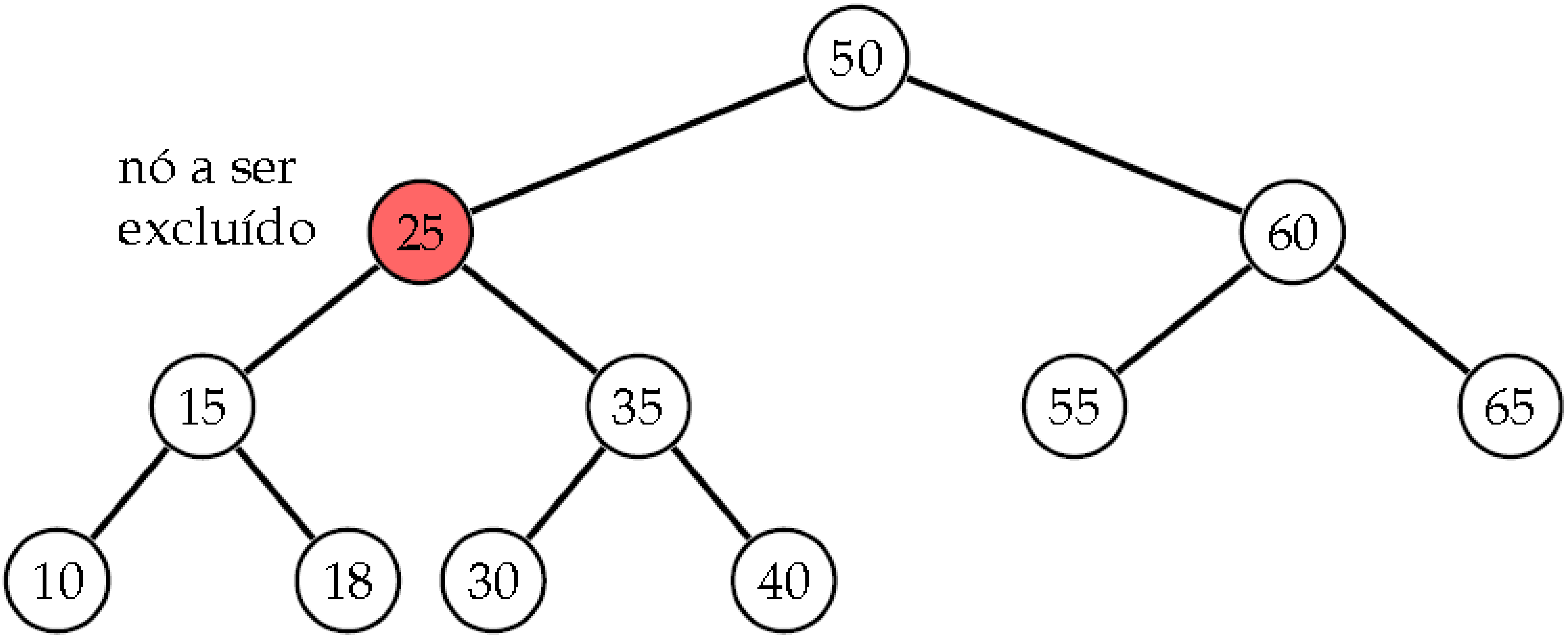
- Para remover um nó com 2 filhos, uma função auxiliar para encontrar o menor elemento da sub-árvore da direita ou substituto (nó mais à esquerda da sub-árvore da direita) será implementada.
- Dado um ponteiro `p` para um nó qualquer, a função `menorSubArvDireita(p)` retorna o menor elemento de sua sub-árvore da direita, se existir.

```
NoArv* ArvBinBusca:: MenorSubArvDireita(NoArv *p)
{
   NoArv *aux = p->getDir(); //nó à direita de p
   while(aux->getEsq() != NULL)
       aux = aux->getEsq();
   return aux;
}
```

Árvore Binária de Busca

Remove nó com 2 filhos

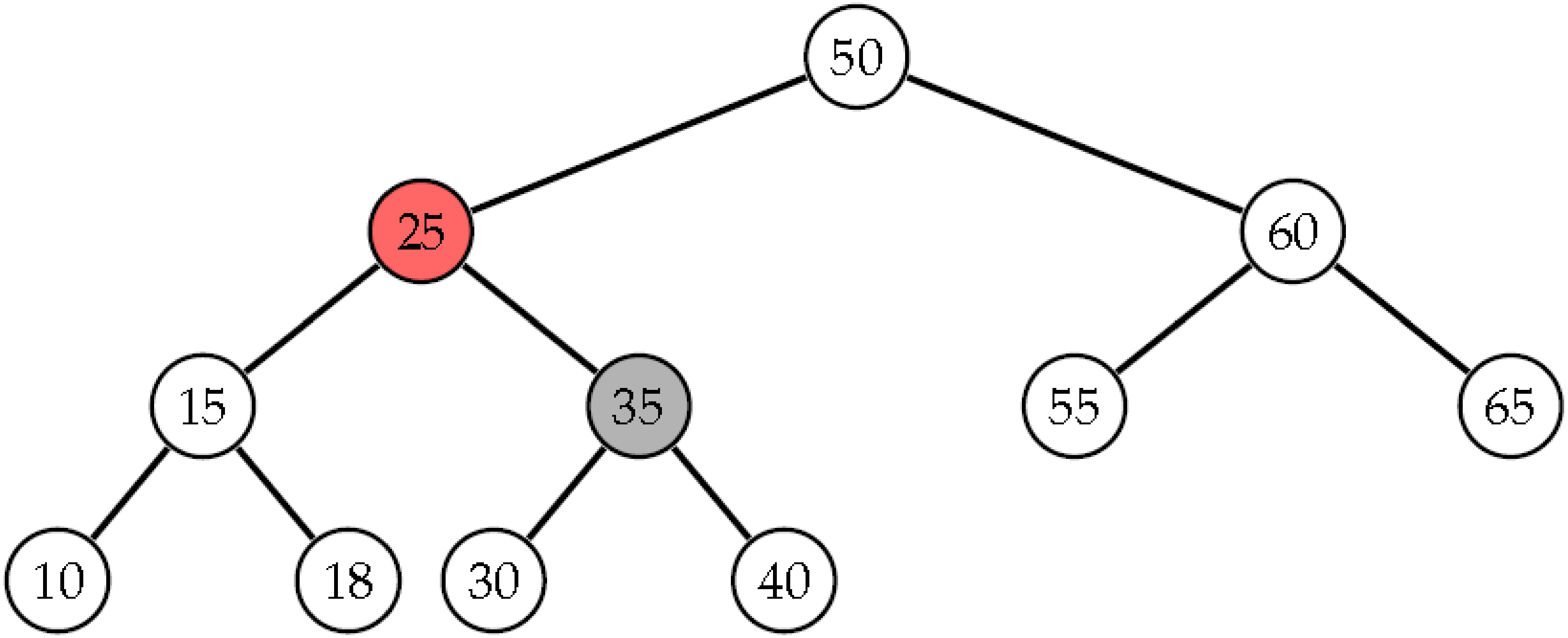
- Exemplo: excluir o nó 25
- Passo 1/6



Árvore Binária de Busca

Remove nó com 2 filhos

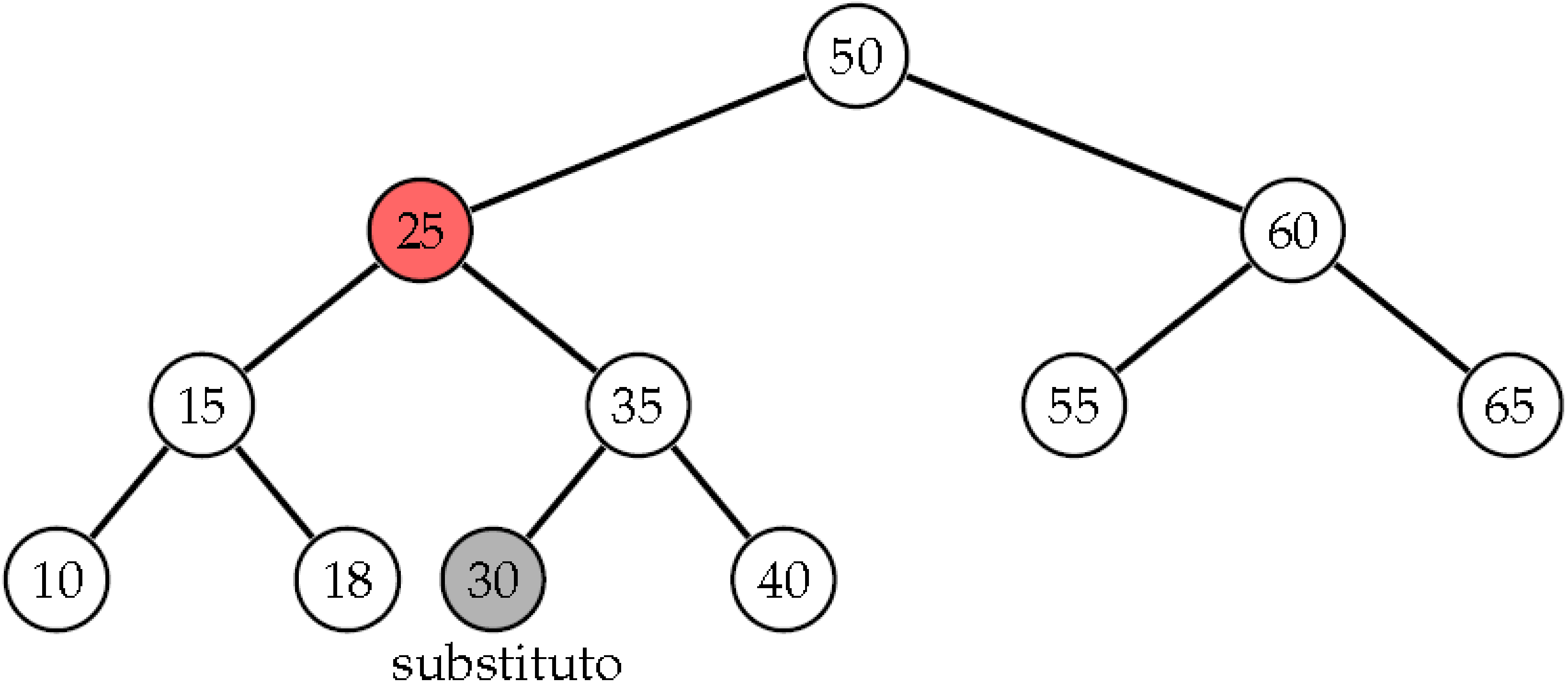
- Exemplo: excluir o nó 25
- Passo 2/6



Árvore Binária de Busca

Remove nó com 2 filhos

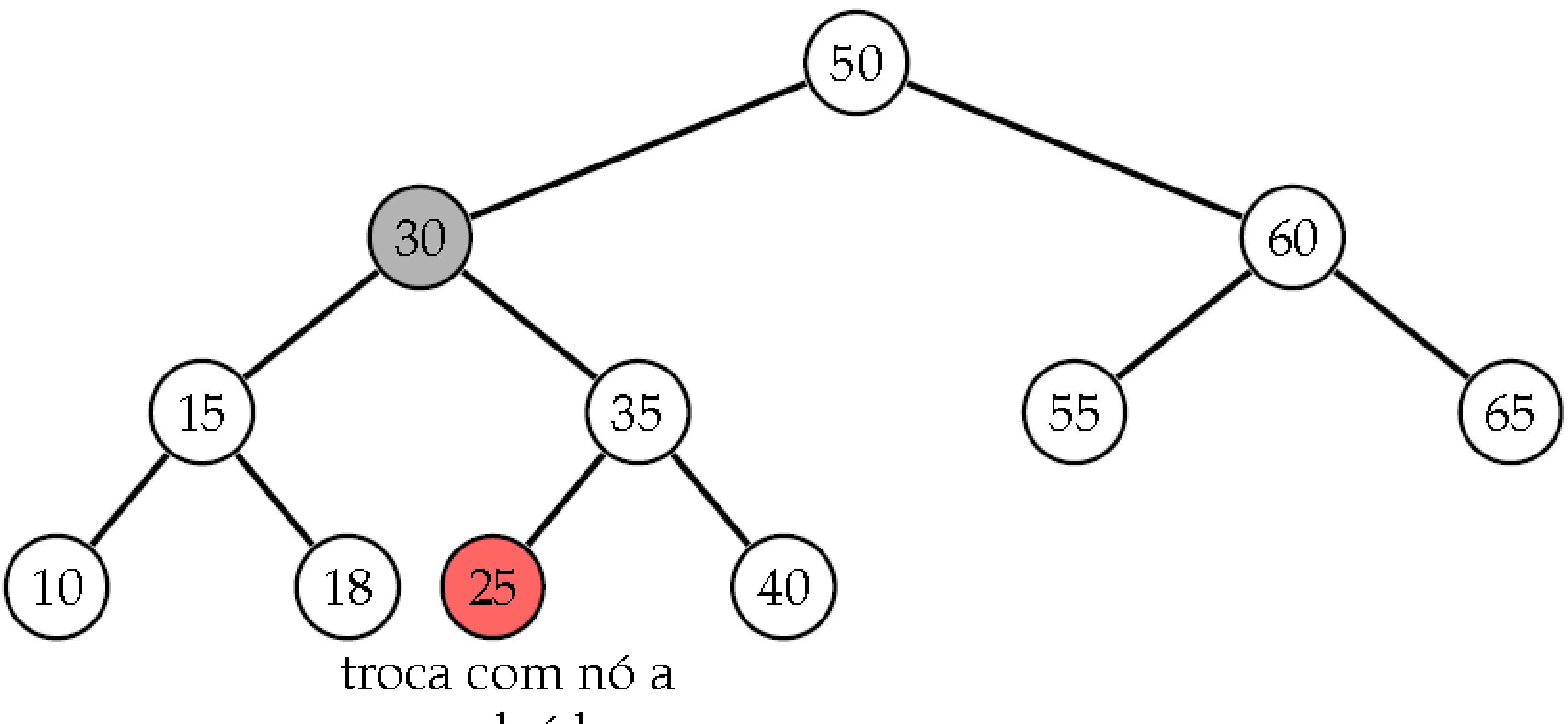
- Exemplo: excluir o nó 25
- Passo 3/6



Árvore Binária de Busca

Remove nó com 2 filhos

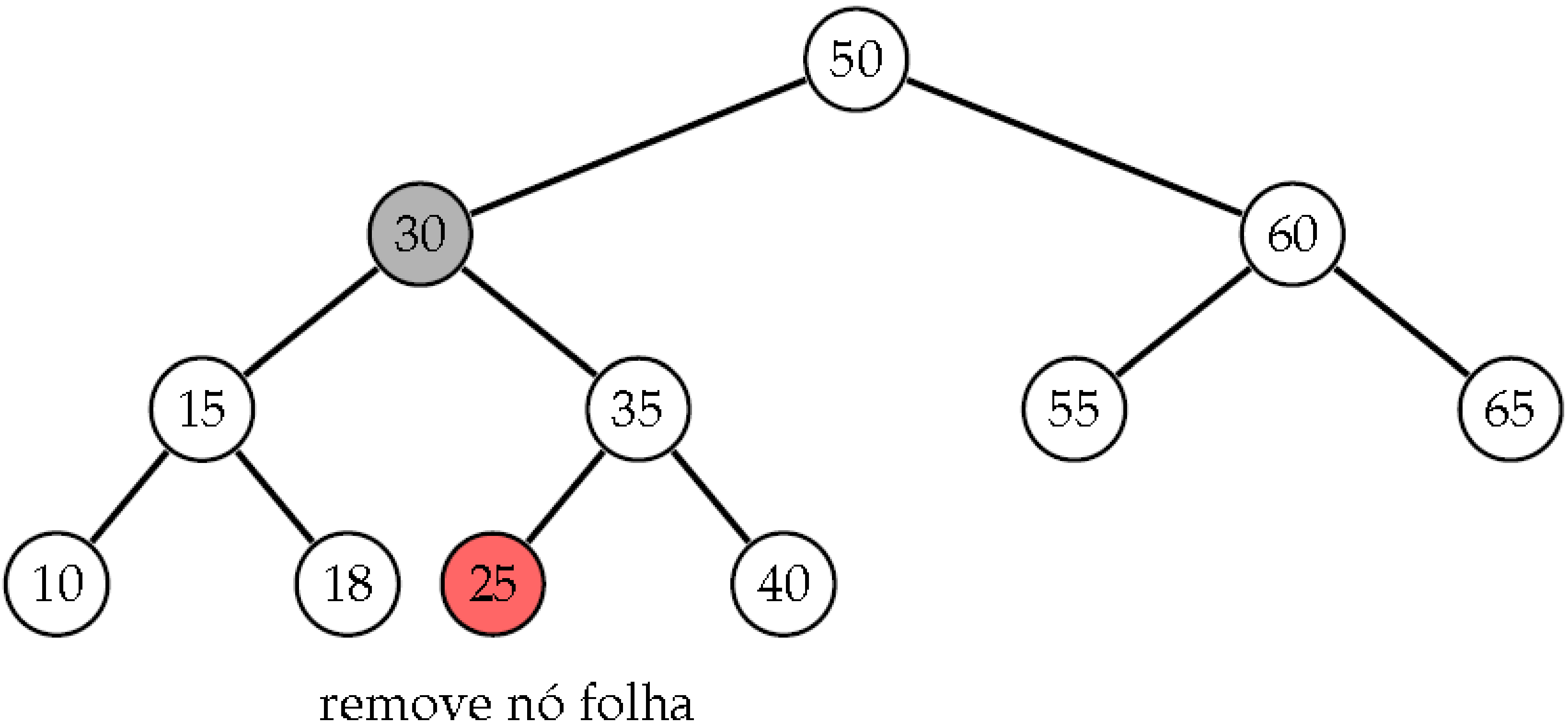
- Exemplo: excluir o nó 25
- Passo 4/6



Árvore Binária de Busca

Remove nó com 2 filhos

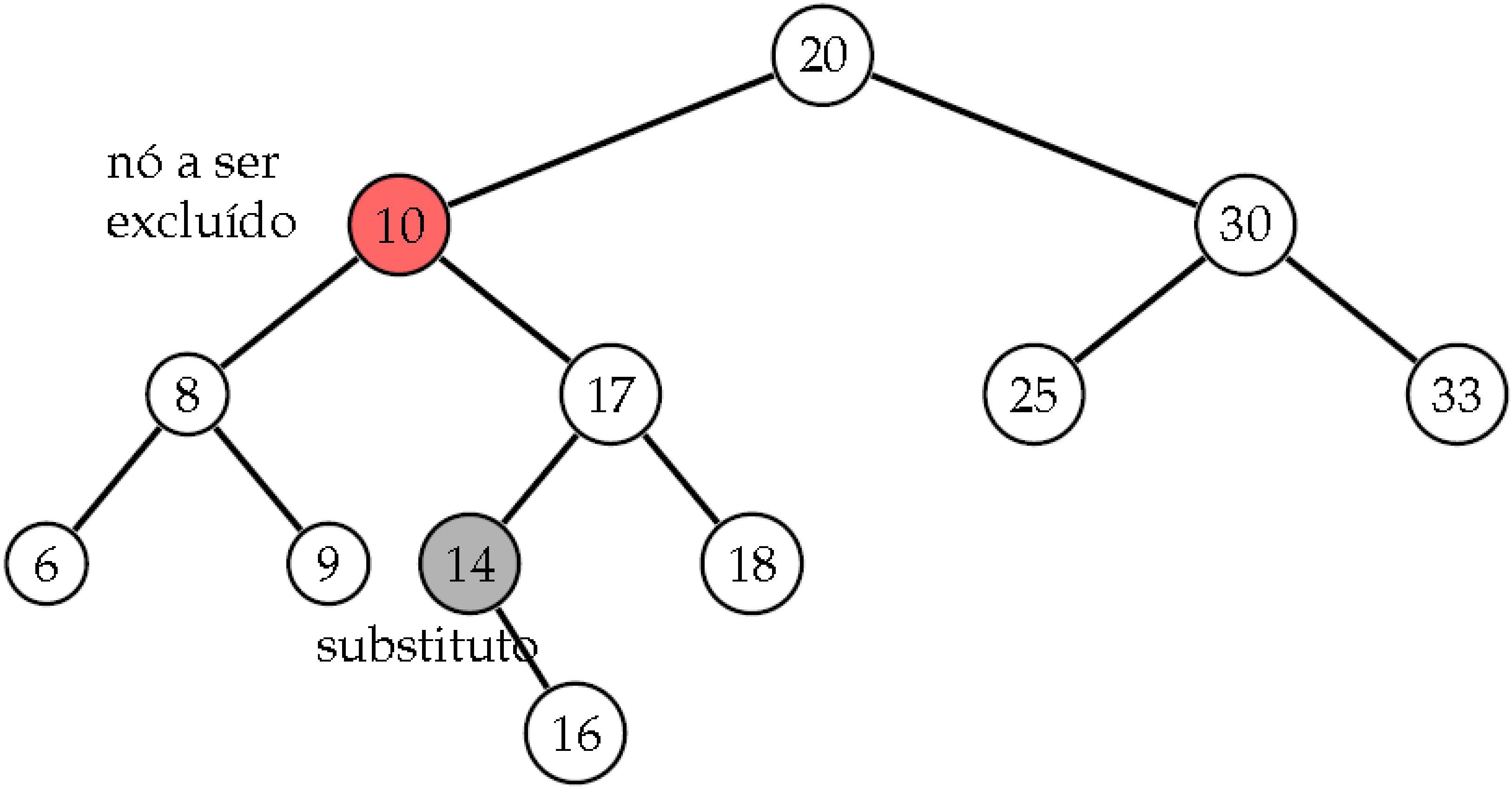
- Exemplo: excluir o nó 25
- Passo 5/6



Árvore Binária de Busca

Remove nó com 2 filhos

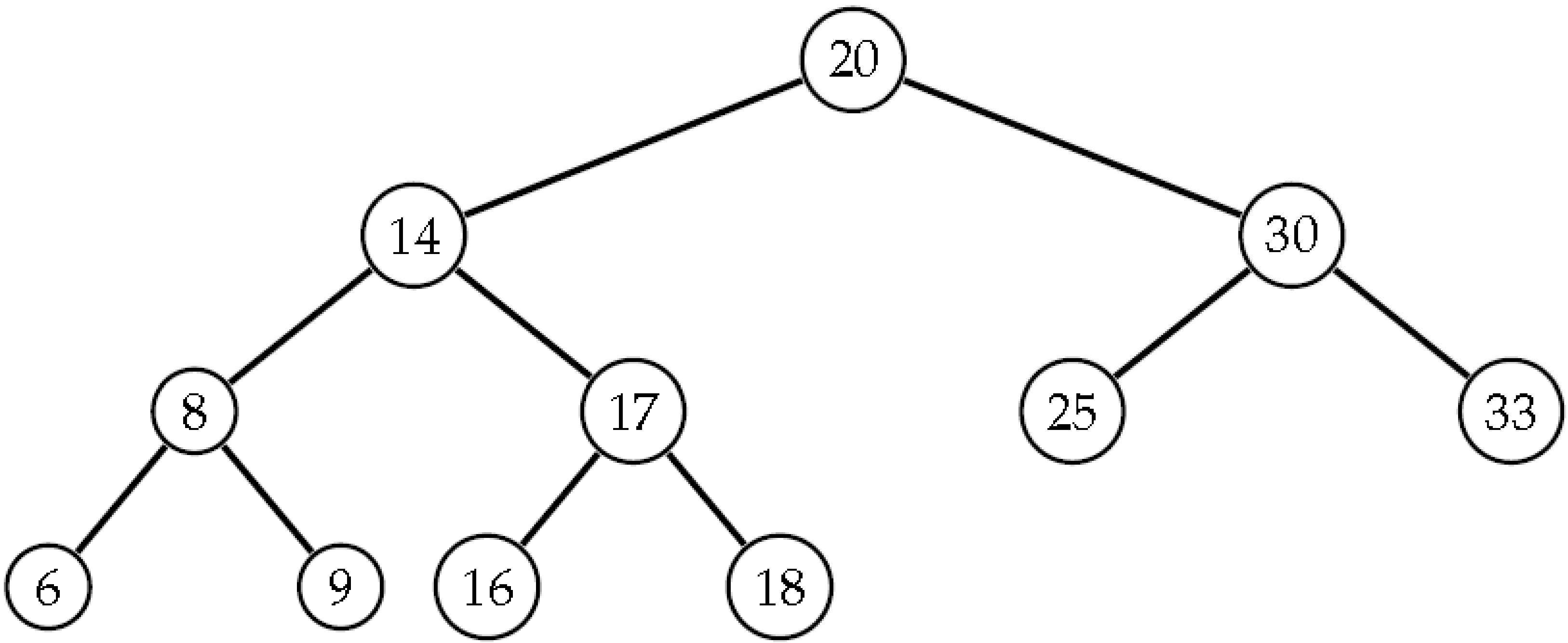
- Outro exemplo: excluir o nó 10



Árvore Binária de Busca

Remove nó com 2 filhos

- Outro exemplo: excluir o nó 10



situação final

Árvore Binária de Busca

Remoção

```
void ArvBinBusca::remove(int x)
{
   raiz = auxRemove(raiz, x);
}
```

- Na classe `ArvBinBusca`.

```
class ArvBinBusca
{
   private:
      //...
      NoArv *auxRemove(NoArv *p; int x);//remove nó
      NoArv *removeFolha(NoArv *p);     //caso 1
      NoArv *remove1Filho(NoArv *p);    //caso 2
      NoArv *MenorSubArvDireita(NoArv *p);//caso 3
   public:
      //...
      void remove(int x); //chama auxRemove()
};
```

Árvore Binária de Busca

Remoção

```
NoArv* ArvBinBusca::auxRemove(NoArv *p; int x)
{
   if(p == NULL)
      return NULL;
   else if(x < p->getInfo())//remove na sub. esquerda
      p->setEsq(auxRemove(p->getEsq(), x));
   else if(x > p->getInfo())// remove na sub. direita
      p->setDir(auxRemove(p->getDir(), x));
   else
   {// achou o nó a ser removido, p->getInfo() == x
      if((p->getEsq() == NULL)&&(p->getDir() == NULL))
         p = removeFolha(p);// p aponta para uma folha

      else if((p->getEsq() == NULL)||(p->getDir() == NULL))
         p = remove1Filho(p); //p tem só um filho

      else
      //continua...
}
```

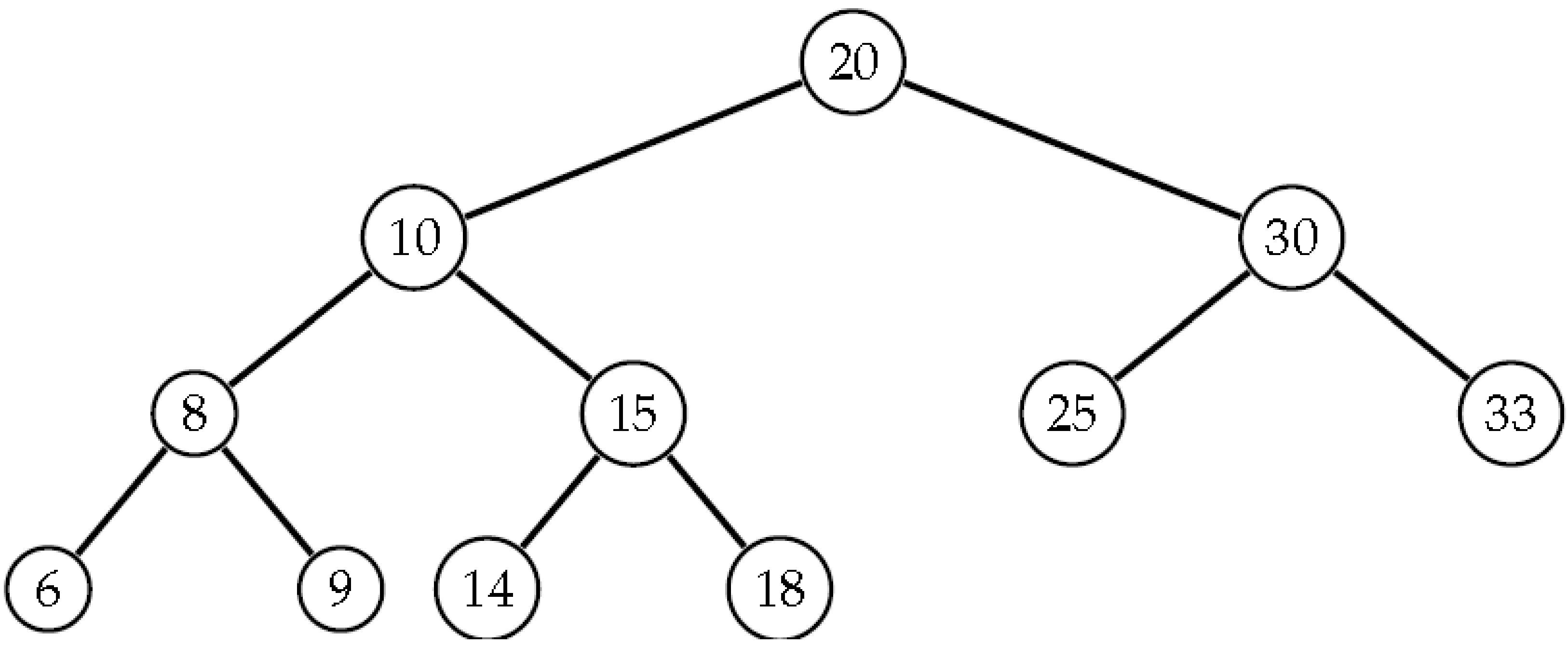
Árvore Binária de Busca
Remoção

```
        // ... continuação
        else
        {
            //p tem dois filhos
            NoArv *aux = menorSubArvDireita(p);

            // troca as informações
            int auxInt = aux->getInfo();
            p->setInfo(auxInt);
            aux->setInfo(x);

            // recursão: para a subarv. direita
            p->setDir(auxRemove(p->getDir(), x));
        }
    }
    return p;
}
```

Árvore Binária de Busca

Remove nó com 2 filhos

- Exercício
  - Excluir os nós 14, 15, 18 e 6
  - Apresentar a ABB ao final de cada remoção



Árvore Binária de Busca

Exercícios

1. Fazer uma operação para encontrar, e retornar, o **maior** elemento de uma árvore binária de busca.
2. Fazer uma operação para encontrar, e retornar, o **menor** elemento de uma árvore binária de busca.
3. Fazer uma operação para remover o **maior** elemento de uma árvore binária de busca.
4. Fazer uma operação para remover o **menor** elemento de uma árvore binária de busca.
5. Alterar a operação para a remoção de nós com dois filhos considerando, agora, o maior elemento da sub-árvore à esquerda como o elemento a ser "substituído" com o nó a ser removido.

Árvore Binária de Busca

Exercícios

6. Uma árvore binária de busca é considerada balanceada se sua altura h é próxima de $\log_2(n)$. Fazer uma operação para verificar se uma dada árvore binária de busca está balanceada. Considere uma árvore balanceada se $h < \log_2(n) + 1$.
7. Desenvolver uma operação – que retorna true ou false – para verificar se uma AB, passada como parâmetro, é uma ABB. Dica: use o fato do percurso em ordem visitar os nós em ordem crescente.

Árvore Binária de Busca

Outra possibilidade

- Utilizar um nó que possui um ponteiro para o pai de um nó da AB.
- Assim, o TAD `NoArv` terá: info, esq, dir, pai
- Exercício: considerando uma ABB cujo nó possui o ponteiro para o pai, refazer o exercício 3 e 4 para remover o menor ou maior valor da árvore.