Tabela de símbolos

Marcelo K. Albertini

6 de Janeiro de 2014

Tabelas de símbolos

Tabela de símbolos ou **Dicionário**

Abstração do par **chave-valor**

- Inserir valor com uma chave associada
- Dada uma chave, resgatar o valor associado

Aplicações de tabelas de símbolos

aplicação	objetivo	chave	valor
dicionário	obter definição	termo	definição
índice de livro	achar página relevante	termo	página
distribuição de arquivos	baixar música	título	URL
busca web	encontrar sites	busca	lista URLs
compilador	obter propriedades de variáveis	variável	tipagem
roteamento	rotear pacotes em rede	destino	melhor rota
DNS	mapear URL para IP	URL	IP
DNS reverso	mapear IP para URL	IP	URL
sistema de arquivos	encontrar arquivo em disco	nome	posição em disco

- DNS = Domain Name System - sistema de domínios
- URL = Universal Resource Locator - localizador de recursos
- IP = Internet Protocol - aqui refere-se ao endereço

Exemplo de uso

```
1  public static void main(String args[]) throws Exception
2  {
3   int capacity = 100;
4   Tabela<String, Integer> dns =
5                     new Tabela<String, Integer>(capacity);
6
7  // parte do servidor de DNS
8   dns.put("localhost" , 12701);
9   dns.put("pdq.mg.br" , 25501);
10  dns.put("pf.br"       ,147107);
11  dns.put("www.ufu.br",146234);
12
13 // parte do cliente que busca o IP de um site
14  String chave = "www.ufu.br";
15  System.out.println("IP("+chave+") = "+dns.get(chave));
16  chave = "pdq.mg.br";
17  System.out.println("IP("+chave+") = "+dns.get(chave));
18 }
```

Aplication Programming Interface (API)

Tipo abstrato de dados = API

class	Tabela<Chave, Valor>	
	Tabela()	criar tabela de símbolos
void	put(Chave c, Valor v)	colocar par chave-valor na tabela (remove chave se valor é `null`)
Valor	get(Chave c)	valor associado com chave, `null` se ausente
void	delete(Chave c)	remover chave-valor da tabela
boolean	contains(Chave c)	existe a chave na tabela?
boolean	isEmpty()	está vazio?
int	size()	total de chaves cadastradas
Iterable<Chave>	keys()	obtém as chaves na tabela

`Iterable<Chave>` é um tipo especial para lista uma sequência de elementos com operações dedicadas para tal tarefa.
Permite o uso de `foreach`.

Chaves e valores

- Chave
  - pode ser `Comparable<Chave>` ou
  - pode ser testado por equivalência com `boolean equals(Chave c)`
  - No Java, na relação de equivalência:
    - Todas as classes tem `equals()`
    - Sempre `x.equals(x) == true`
    - Sempre `x.equals(null) == false`
    - Sempre `x.equals(y) == y.equals(x)`
    - Se `x.equals(y)` e `y.equals(z)` então `x.equals(z)`
    - A implementação padrão somente testa se 2 variáveis referem-se ao mesmo objeto
- Valor
  - Objeto ou estrutura qualquer

Exemplo de implementação de equals() de chave

```java
import java.util.Arrays;
public class Reuniao {
 private int mes, ano, dias[];
 public boolean equals(Object o) { // deve ser Object
   if (this == o) return true; // verifica referencias
   if (o == null) return false;

   // verifica se são do mesmo tipo
   if (o.getClass() != this.getClass()) return false;

   Reuniao r = (Reuniao) o;
   // em arrays, NÃO usar equals()
   // em arrays 1D primitivos usar Arrays.equals()
   // em 2D+ ou não primitivos: Arrays.deepEquals()
   if (Arrays.equals(dias, r.dias) == false)
     return false;
   else if (mes != r.mes) return false;
   else if (ano != r.ano) return false;
   return true;
 } // e continua...
}
```

Convenções

- Valores não podem ser `null`
- Método `get()` retorna `null` se chave não está presente
- Método `put()` reescreve valor prévio com novo valor
- As chaves são `Comparable` (permitem o uso de `compareTo`)

Com essas convenções

- é fácil de implementar `contains()`

```
1 public boolean contains(Chave c) {
2   return get(c) != null;
3 }
```

- é fácil de implementar `delete()`

```
1 public void delete(Chave c) { put(c, null); }
```

Implementação com listas ligadas não ordenadas

Com lista ligada não ordenadas devemos

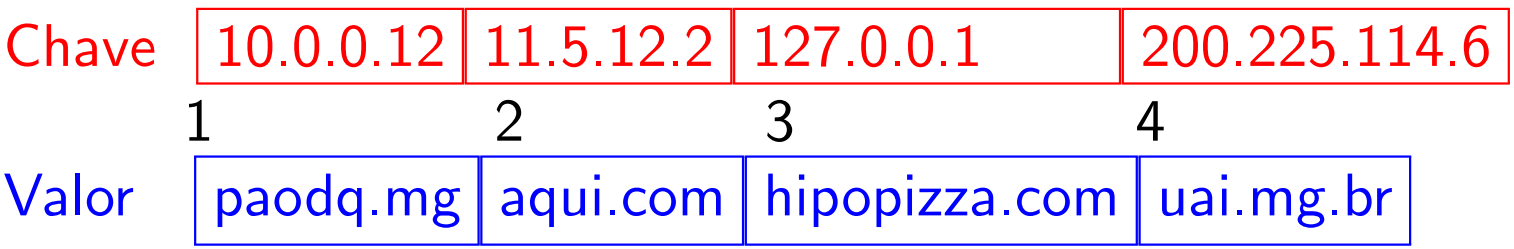
- `get`: varrer todas as chaves até encontrar
- `put`: varrer todas as chaves, se não encontrar inserir no início
- usar `equals()` para avaliação de chaves

Implementação com vetores paralelos

Com vetores paralelos podemos usar

- representação em vetores paralelos
- Chave do tipo `Comparable` e usar `compareTo()`
- `get` baseado em busca binária
- `put` com inserção ordenada em vetor

Vetores paralelos



Protótipo de implementação com vetores paralelos

```
1  public class Tabela<Chave extends Comparable<Chave>,
       Valor> {
2
3    Chave[] chave;
4    Valor[] valor;
5    int N; // numero de pares chave-valor na tabela
6    int capacidade;
7
8    public Tabela(int capacidade) {
9      N = 0;
10     this.capacidade = capacidade;
11     chave = (Chave[]) new Comparable[capacidade+1];
12     valor = (Valor[]) new Object[capacidade+1];
13   }
14
15   //  CONTINUA ...
16 }
```

```
1  Valor get(Chave c) {
2    int pos = buscabinaria(chave, c, 0, N-1);
3    if (pos == -1)      return null;
4    else                return valor[pos];
5  }
6
7  int buscabinaria(Chave[] chave, Chave c,
8                           int inf, int sup) {
9
10   if (inf > sup)  return -1;
11   else {
12     int m = (int) (((double)inf)/2.0 // evita Overflow
13                  + ((double)sup)/2.0);
14     if (chave[m].equals(c))
15       return m;
16     else {
17       if (c.compareTo(chave[m]) > 0)
18         return (buscabinaria(chave, c, m+1, sup));
19       else
20         return (buscabinaria(chave, c, inf, m-1));
21     }
22   } // CONTINUA ...
23 }
```

Operação put: vetores paralelos

```
1  void put(Chave c, Valor v) throws Exception {
2
3    if (N == capacidade)
4          throw new Exception("Capacidade excedida");
5
6    int i = 0;
7    for (i = N-1; i>=0; i--) {
8      if (chave[i].compareTo(c) > 0) {
9        chave[i+1] = chave[i]; // vetores paralelos
10       valor[i+1] = valor[i];
11     } else
12       break;
13   }
14   chave[i+1] = c;
15   valor[i+1] = v;
16   N++;
17 }
```

Custos após N inserções

	`get` pior	`get` médio	`put` pior	`put` médio	Iteração ordenada?	uso da chave
lista ligada desordenada	N	$N/2$	N	N	não	`equals()`
vetor paralelo ordenado	$\log N$	$\log N$	N	$N/2$	sim	`compareTo()`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação `get(16h10)`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação obter chave mínima: `min()`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação obter chave inferior mais próxima: `floor(18h00)`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação obter chave em dada posição: `select(3)`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação obter chaves em intervalo: `keys(13h00,17h00)`

Operações ordenadas

Funcionalidades adicionais

Quando

- chave é `Comparable`
- chaves são mantidas em ordem

podemos implementar `operações ordenadas`

- `min()`, `max()`,
- `floor()`, `ceiling()`
- `keys()`, `select()`

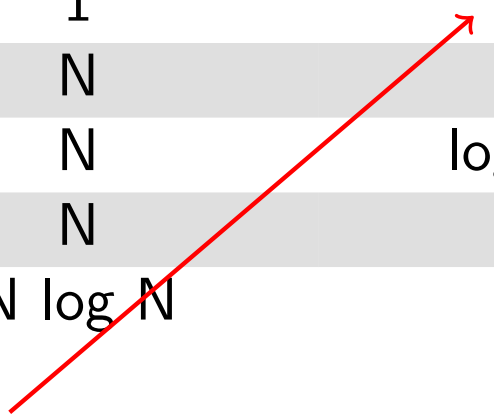
horário chave	origem valor
9h21	Guarulhos
10h20	Rio de Janeiro
11h50	Lima
12h29	Toronto
13h31	Miami
14h19	Taipei
16h10	Amsterdam
17h45	Londres
22h00	Hong Kong
22h35	Buenos Aires

Operação obter chave superior mais próxima: `ceiling(11h00)`

Custos de operações

Ordem de crescimento do tempo de execução para operações da tabela de símbolos

operação	busca sequencial	busca binária
busca	N	log N
inserção/remoção	1	N
min/max	N	1
floor/ceiling	N	log N
select	N	1
iteração ordenada	N log N	N

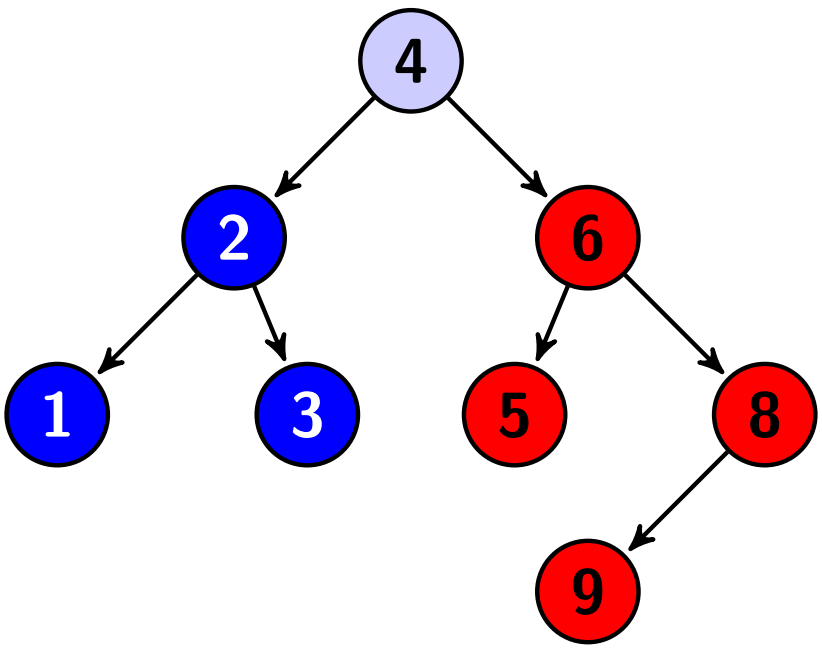


Necessita de melhoria. Muito alto para inserções frequentes.

Árvores de busca binária

Uma árvore binária de busca (ABB)

- tem nós identificador por chaves
- ou é vazia ou tem duas subárvores.
- preserva-se chaves em ordem simétrica



- toda sub-árvore à esquerda tem id menor que o id do nó-pai
- toda sub-árvore à direita tem id maior que o id do nó-pai

Implementação

Uma árvore binária de busca (ABB) tem um nó raiz do tipo `Node`.

```
1  class ABB<Chave extends Comparable<Chave>, Valor> {
2   Node raiz;
3
4   class Node { // uma classe interna
5    Chave chave;  Valor valor; Node esq, dir;
6
7    public Node (Chave c, Valor v) {
8     this.chave = c;  this.valor = v;
9    }
10  }
11
12  Valor get(Chave c) { ... }
13  void  put(Chave c, Valor v) { ... }
14  void  delete(Chave c) { ... }
15  Iterable<Chave> iterator(){ ... }
16 }
```

Um `Node` é composto de uma `Chave`, um `Valor` e referências para subárvore esquerda `esq` e direita `dir`

Operação `Valor get(Chave c)`

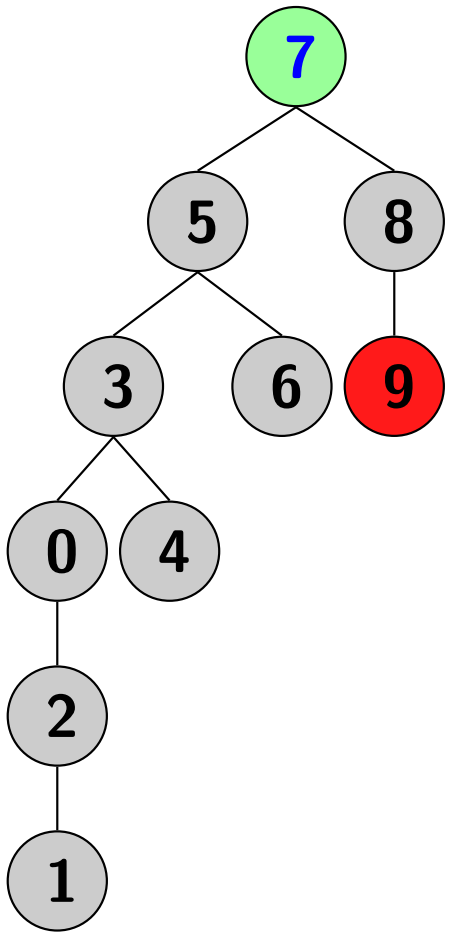
`get`: obtém o valor associado a uma chave

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.dir;
9     else return x.valor;
10   }
11   return null; // por convenção
12 }
```

Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

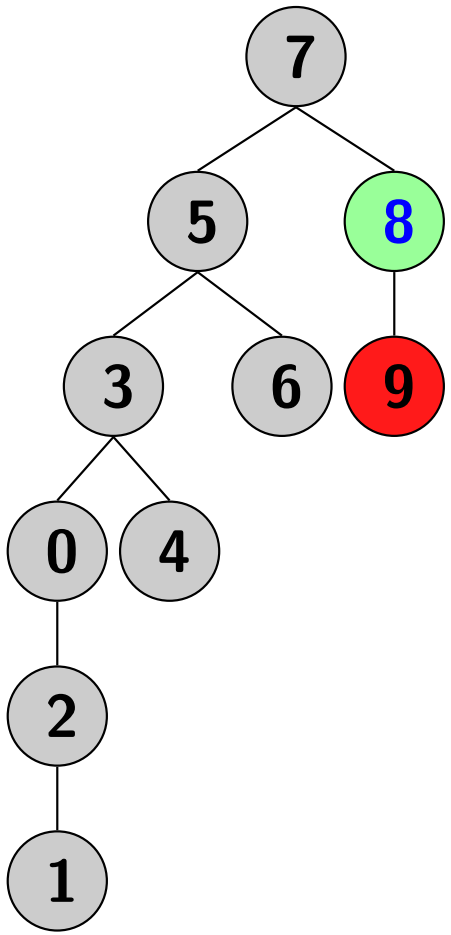
Executando get(9)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

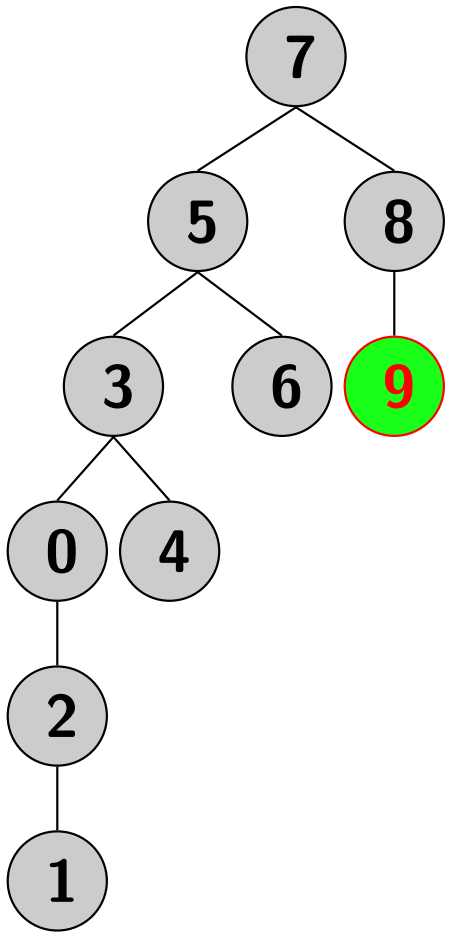
Executando get(9)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

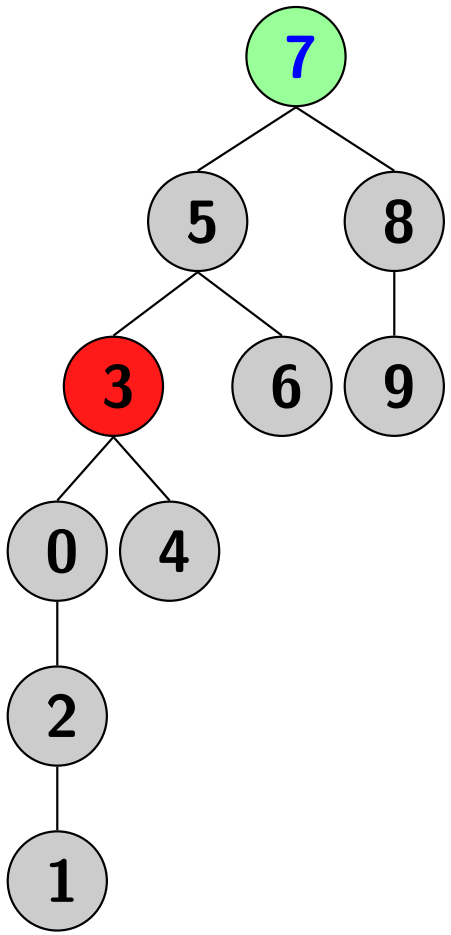
Executando get(9)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

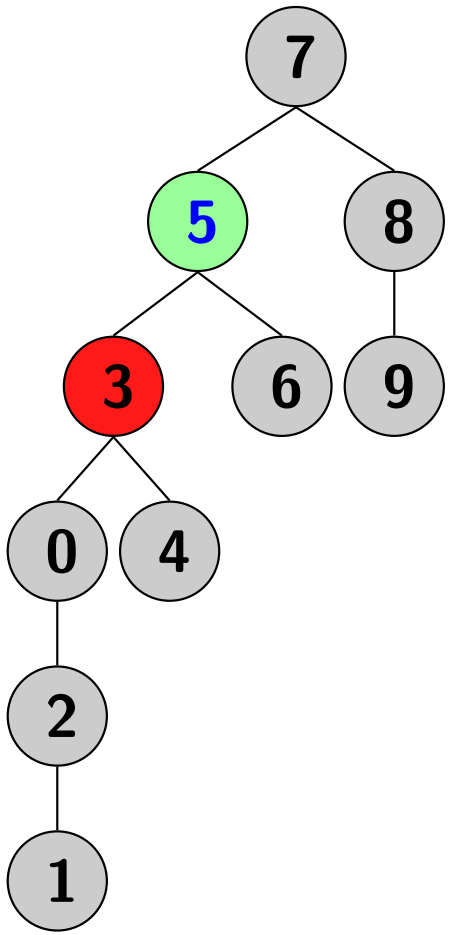
Executando get(3)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

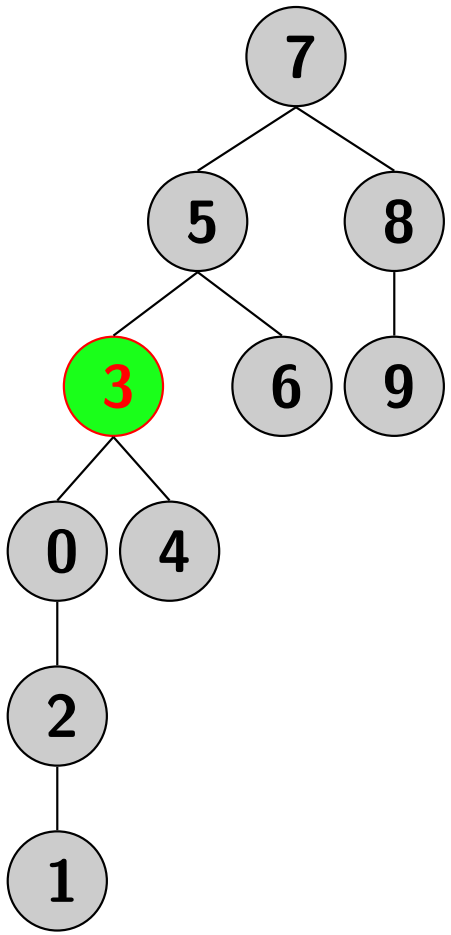
Executando get(3)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

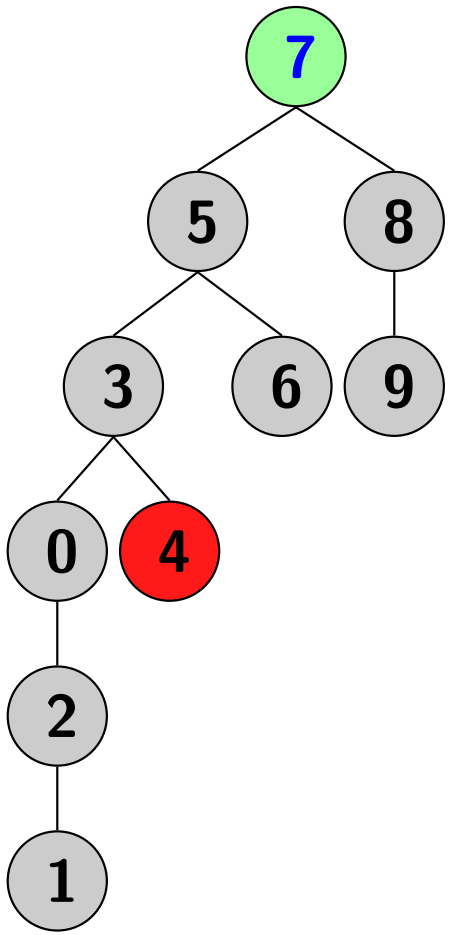
Executando get(3)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

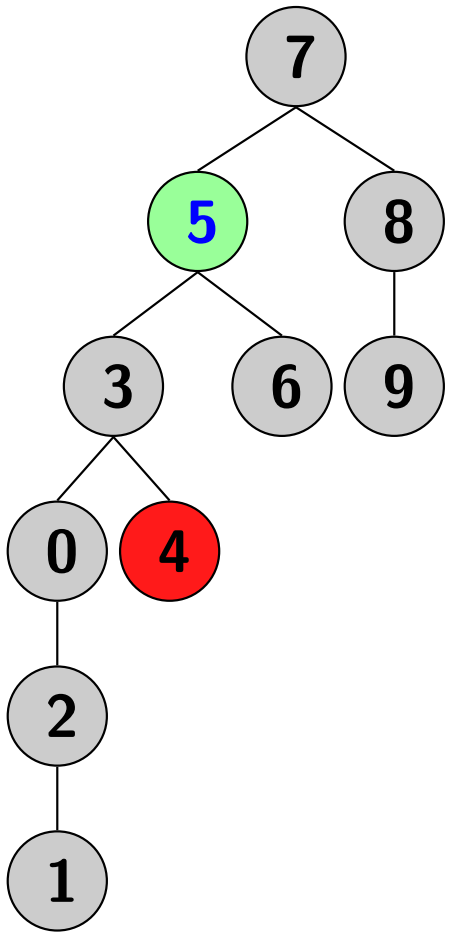
Executando get(4)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

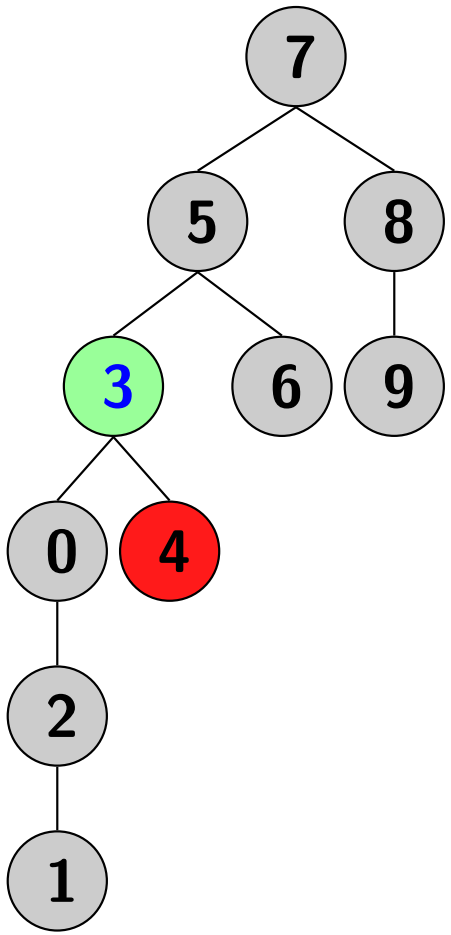
Executando get(4)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

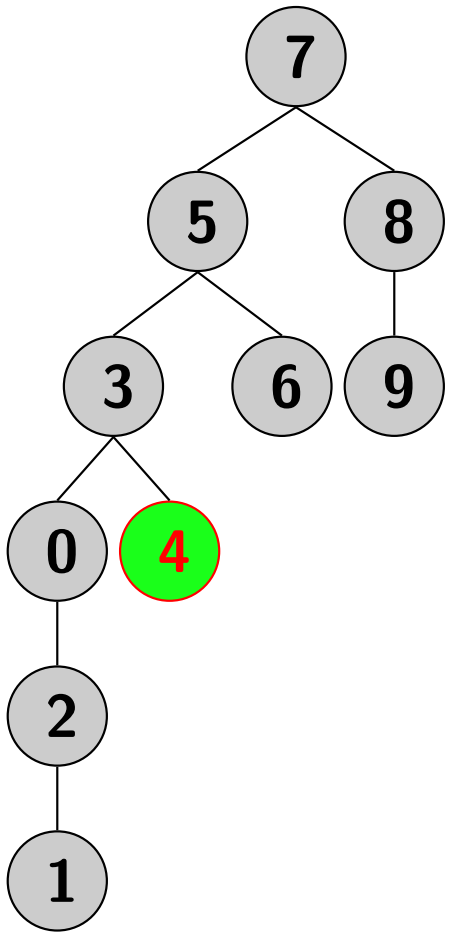
Executando get(4)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

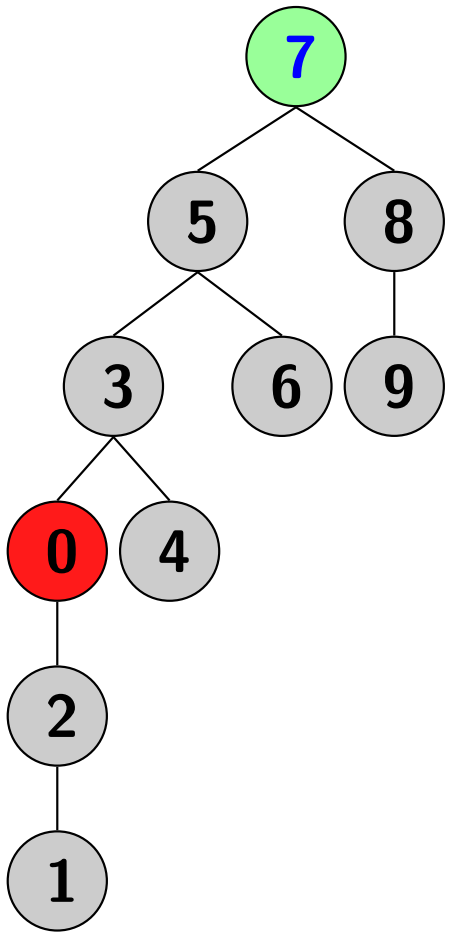
Executando get(4)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

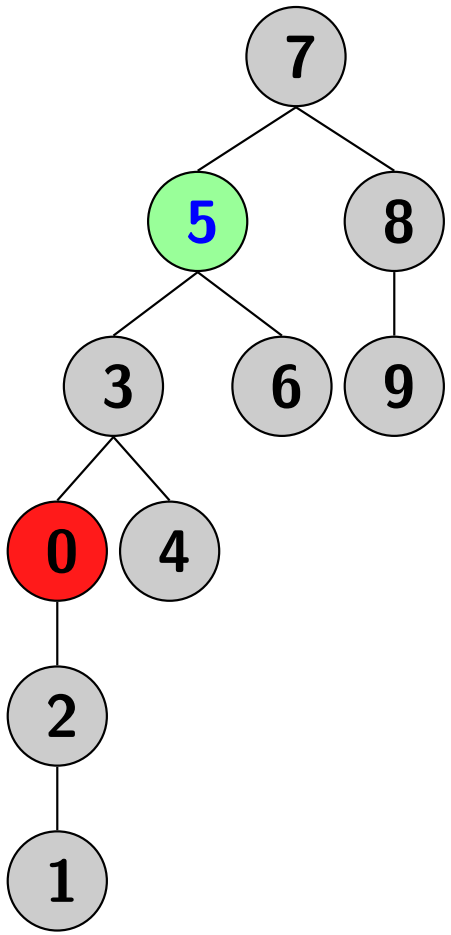
Executando get(0)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

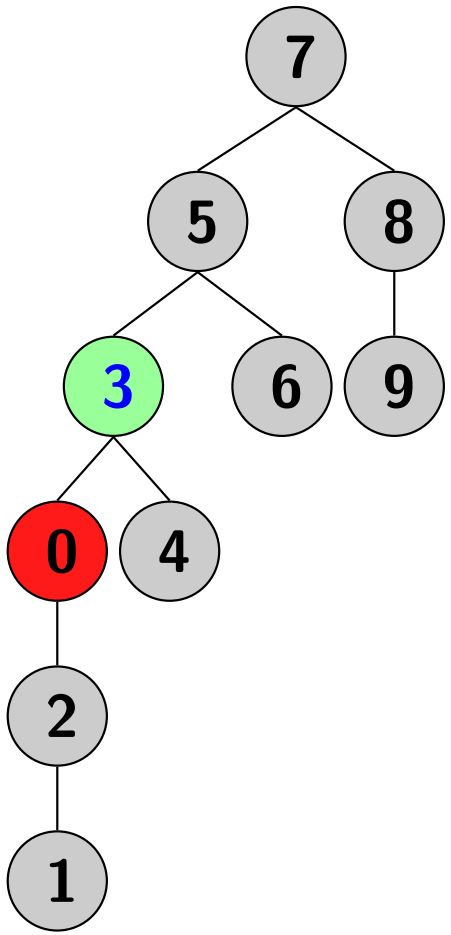
Executando get(0)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

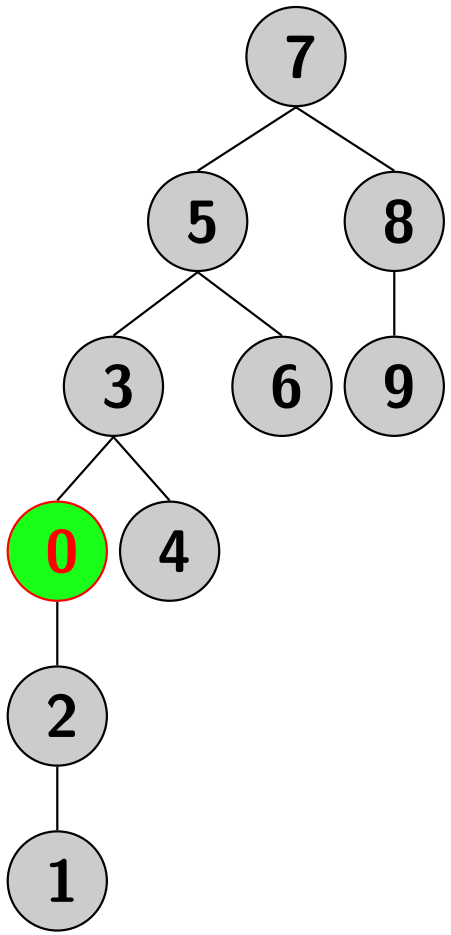
Executando get(0)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

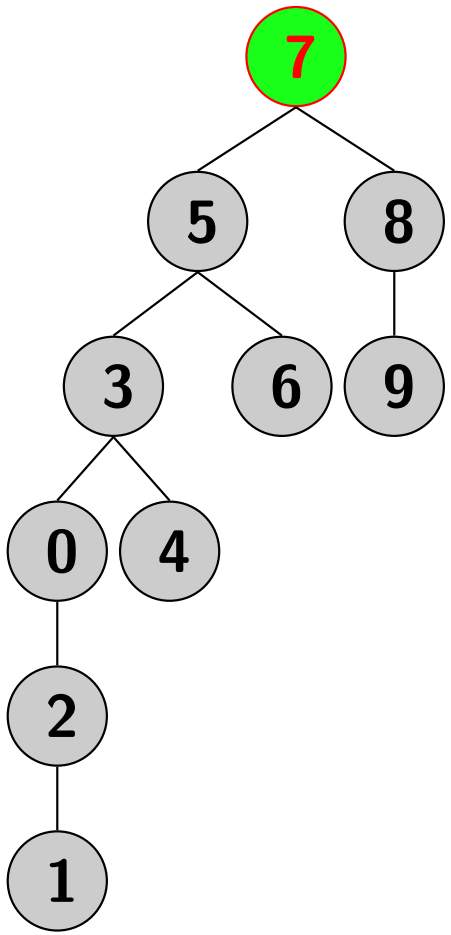
Executando get(0)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

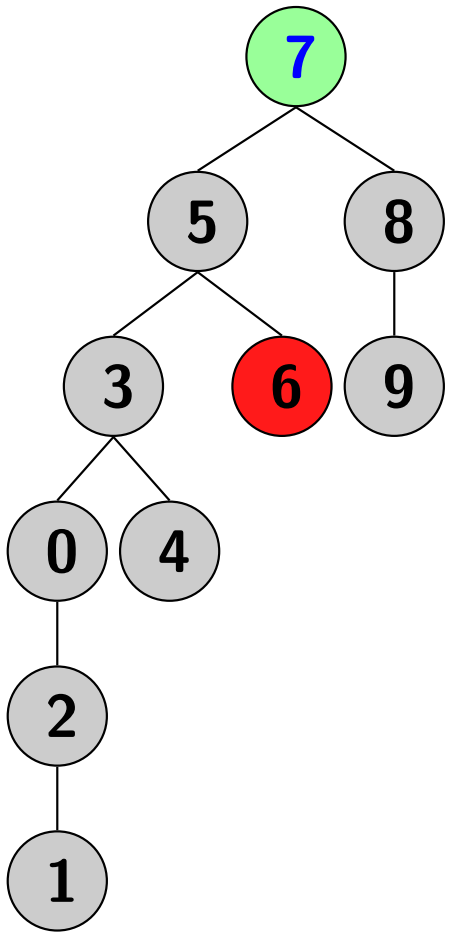
Executando get(7)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

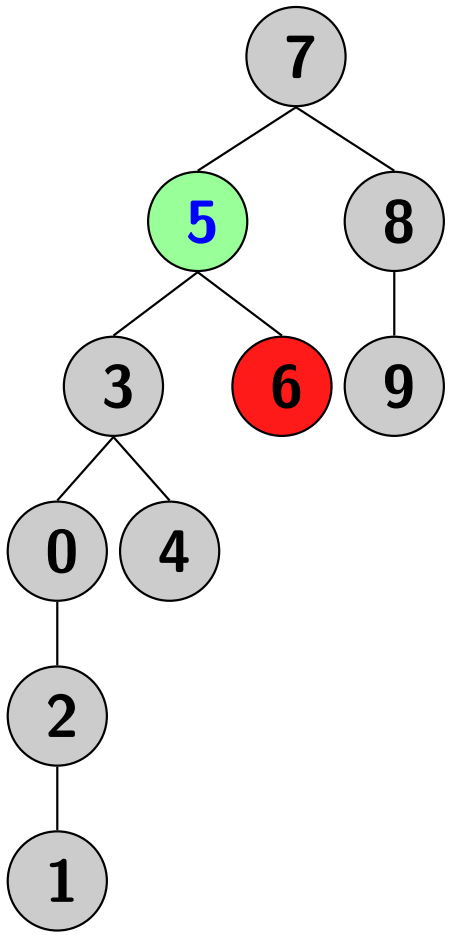
Executando get(6)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

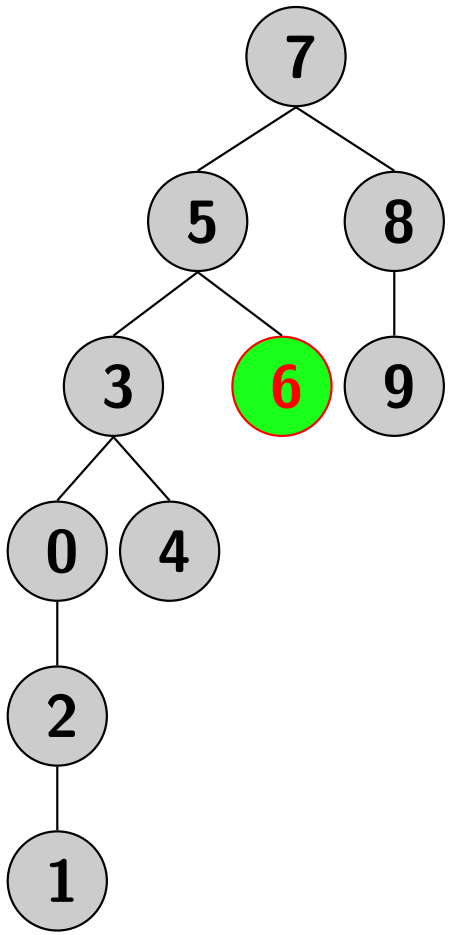
Executando get(6)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

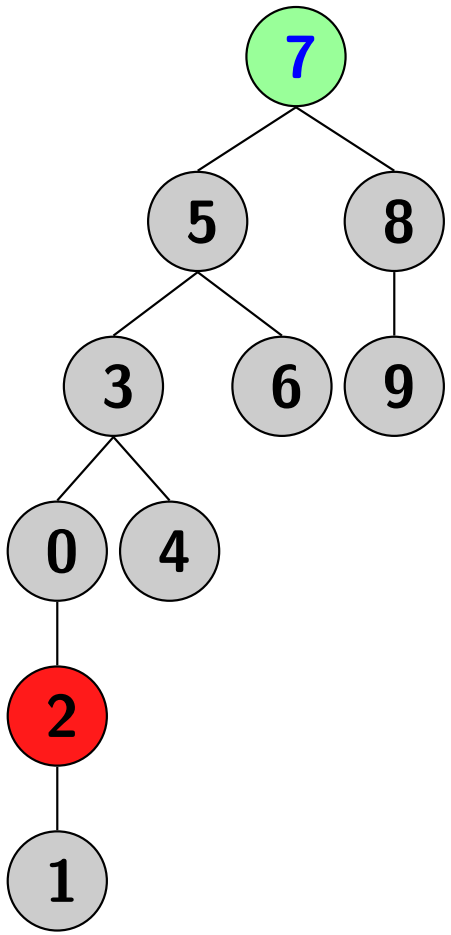
Executando get(6)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

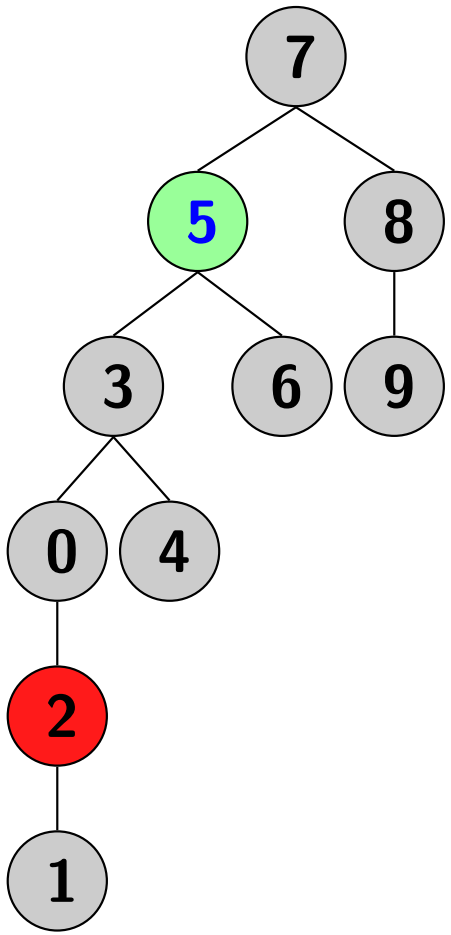
Executando get(2)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

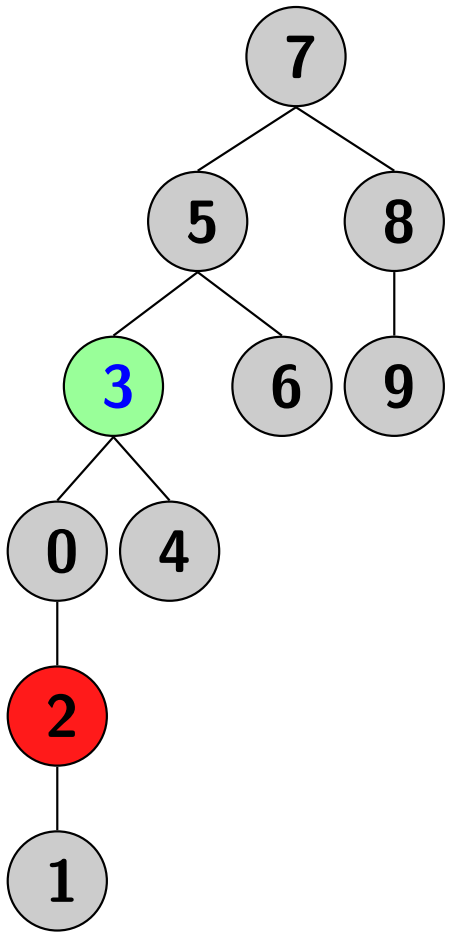
Executando get(2)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

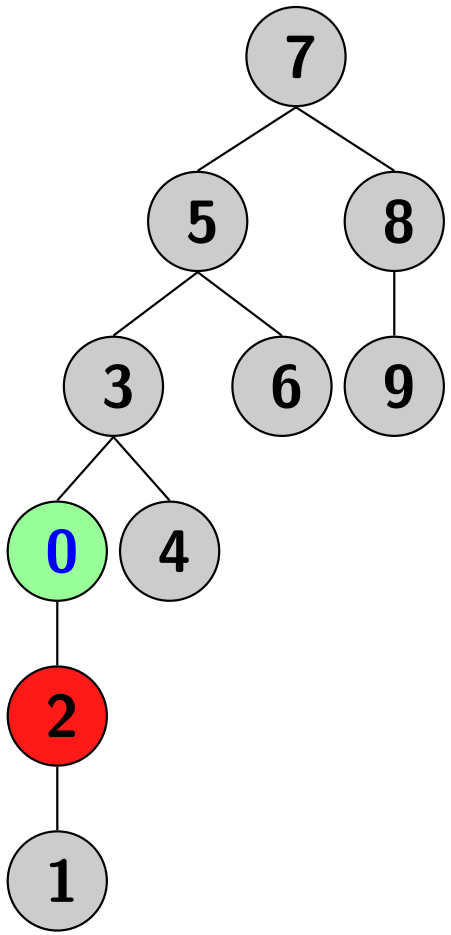
Executando get(2)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

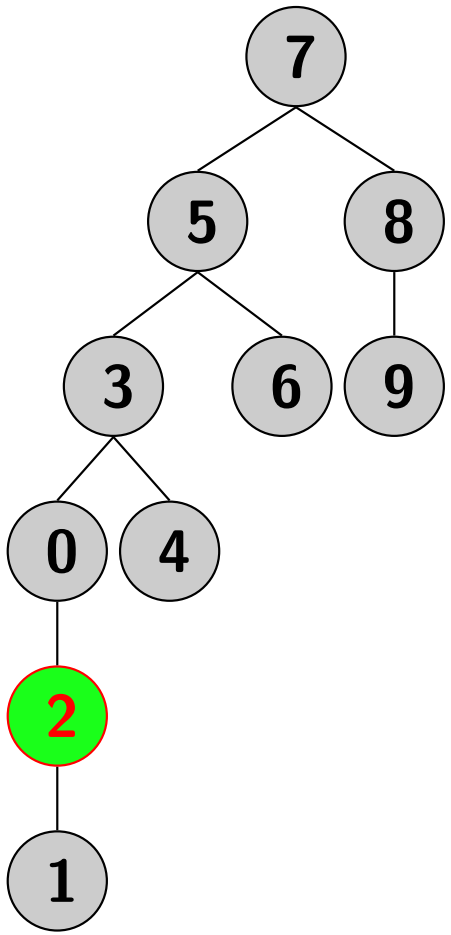
Executando get(2)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

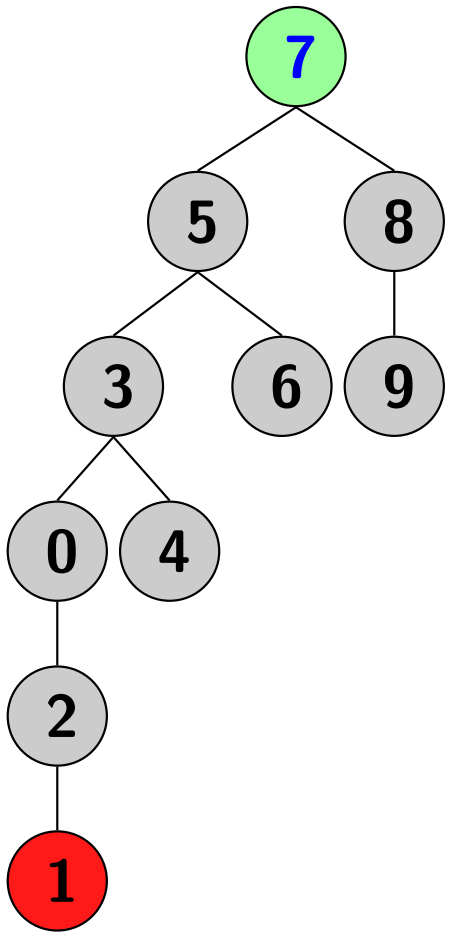
Executando get(2)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

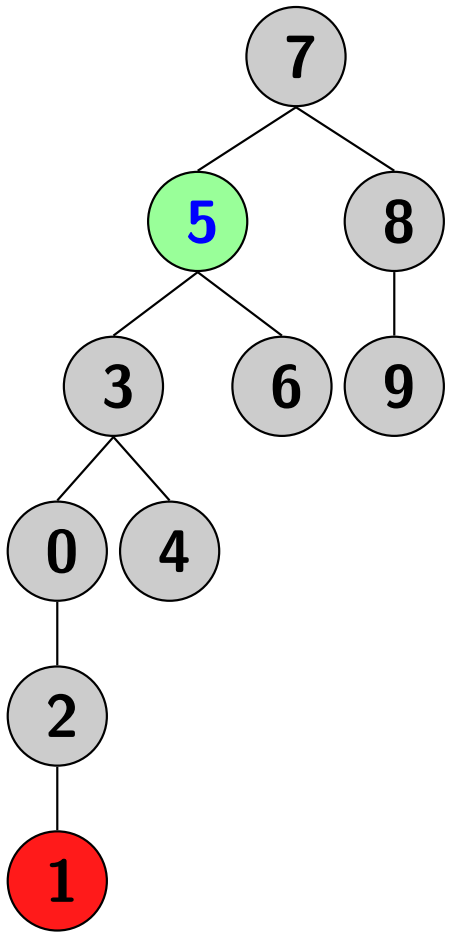
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

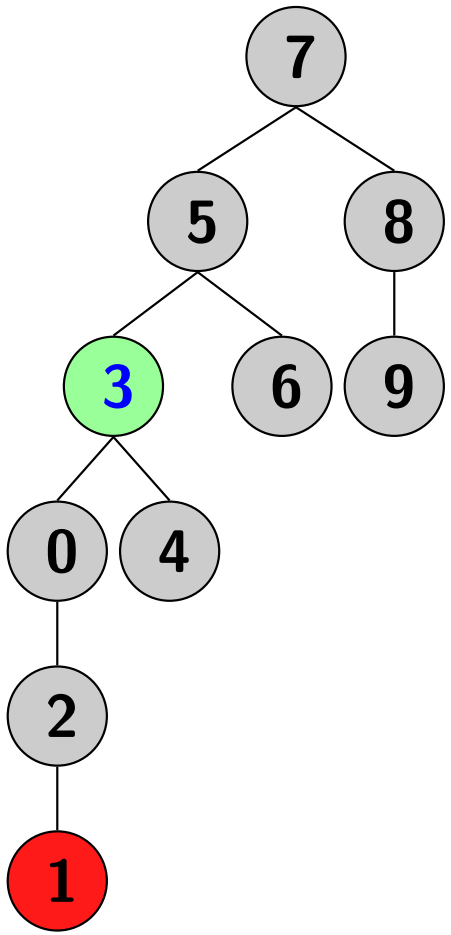
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

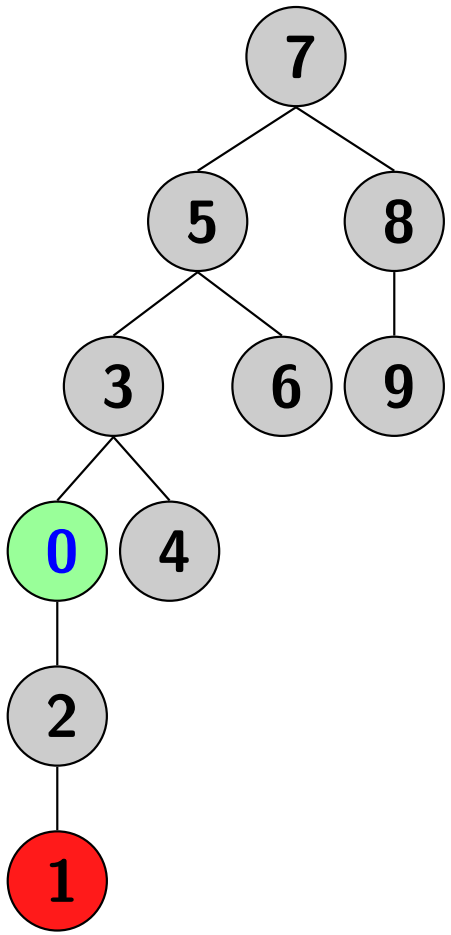
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

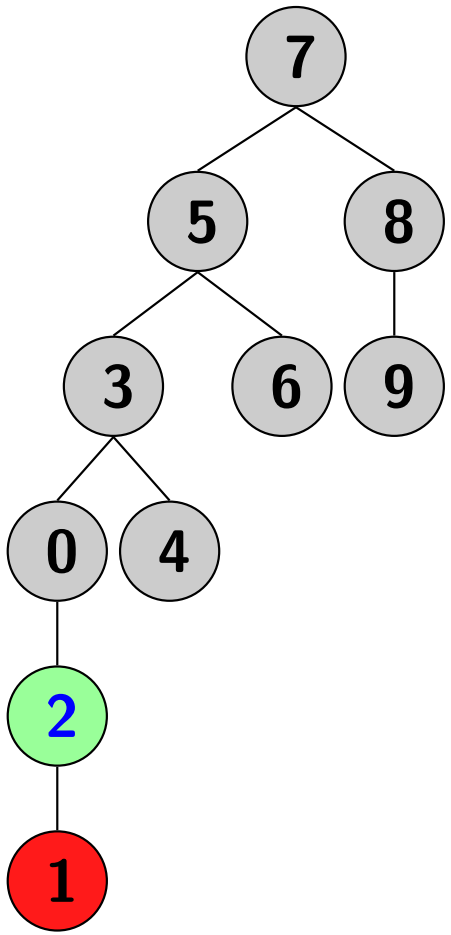
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

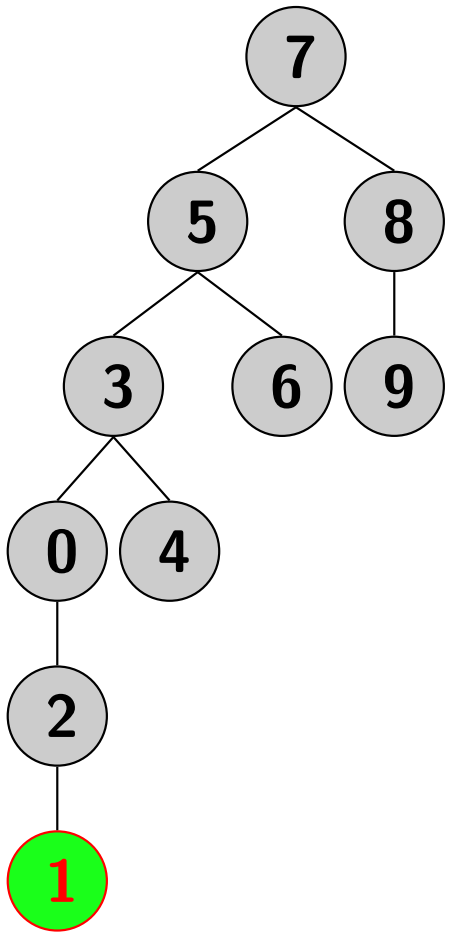
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

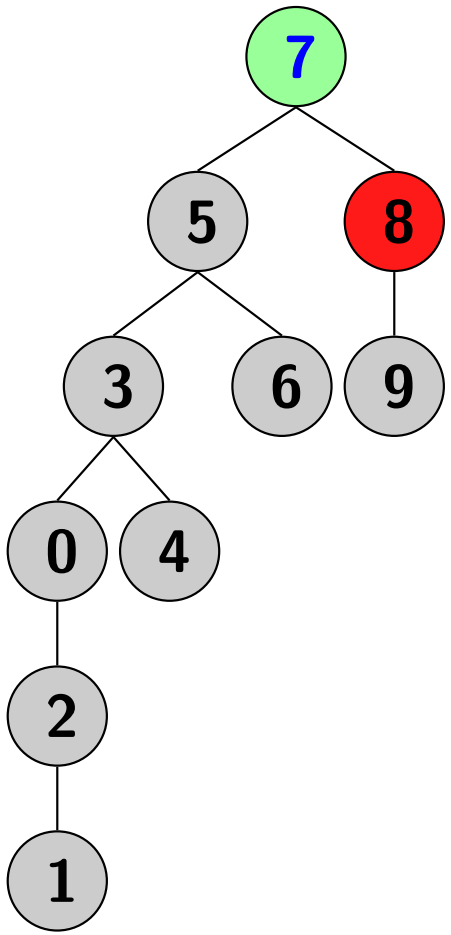
Executando get(1)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

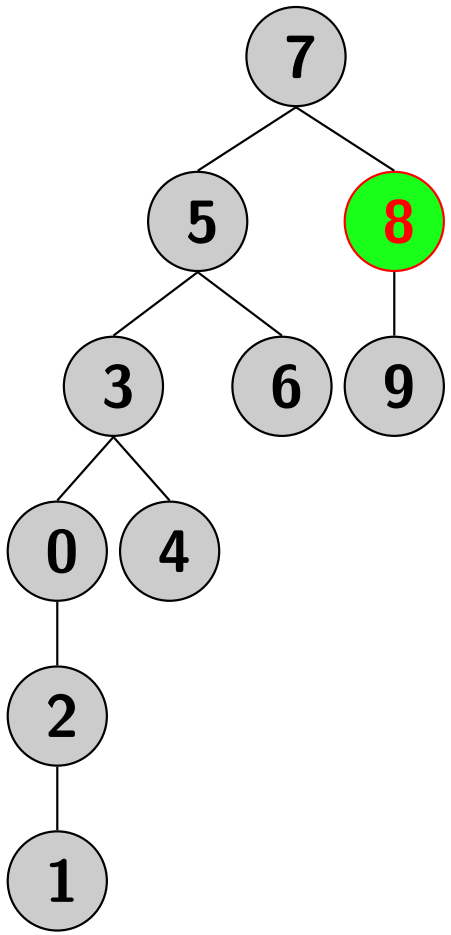
Executando get(8)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

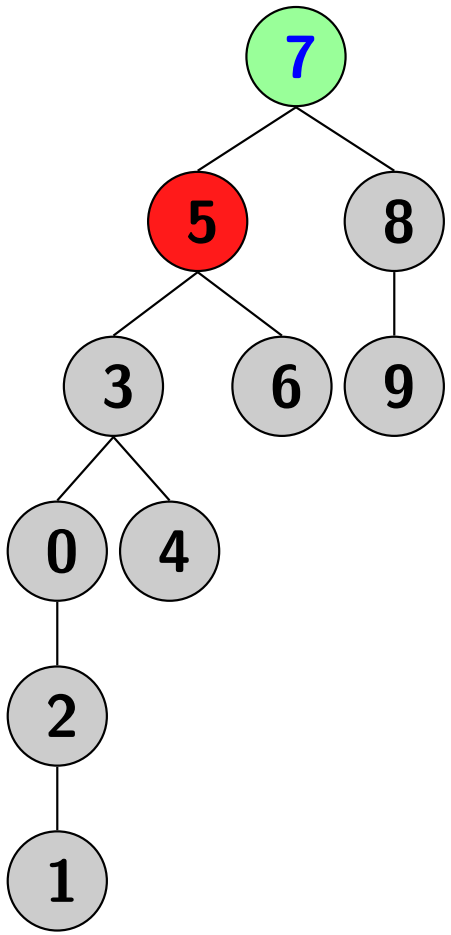
Executando get(8)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

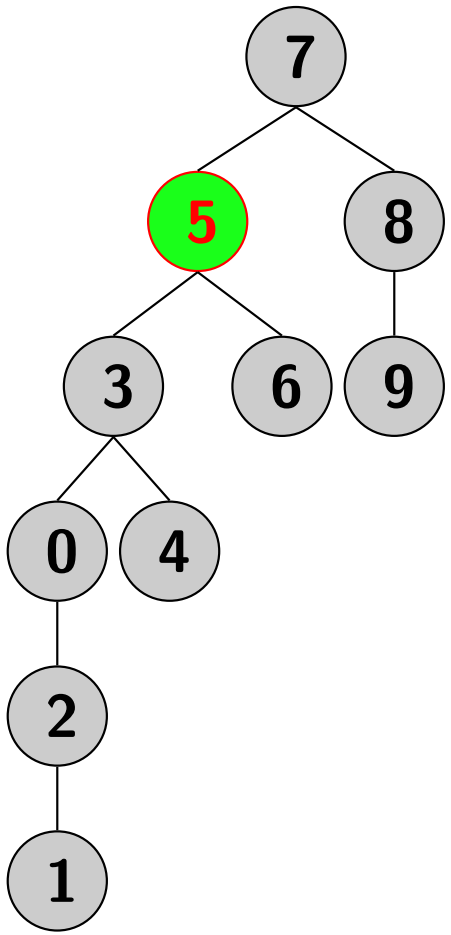
Executando get(5)



Operação `Valor get(Chave c)`

```
1  Valor get(Chave c) {
2   Node x = raiz;
3
4   while (x != null) {
5    int cmp = c.compareTo(x.
         chave);
6
7    if (cmp < 0) x = x.esq;
8     else if (cmp > 0) x = x.
          dir;
9     else return x.valor;
10   }
11   // nao achou
12   return null;
13 }
```

Executando get(5)



Operação `void put(Chave c, Valor v)`

`put`: associa uma chave a um valor.

```
1  public void put(Chave c, Valor v) {
2   raiz = put(raiz, c, v);// começa na raiz
3  }
4
5  Node put(Node x, Chave c, Valor v) {
6   // quando não encontra, faz put
7   if (x == null) return new Node(c, v);
8
9   int cmp = c.compareTo(x.chave);
10  // decide para qual lado seguir
11  if (cmp < 0)
12        x.esq = put(x.esq, c, v);
13  else if (cmp > 0)
14        x.dir = put(x.dir, c, v);
15  else // encontrou! substitui!
16        x.valor = v;
17 }
```

ABB: análise matemática

Caso médio

Se n chaves distintas são inseridas em uma ABB em ordem aleatória, o número esperado de comparações para uma busca/inserção é aproximadamente $2 \ln n$.

Altura esperada

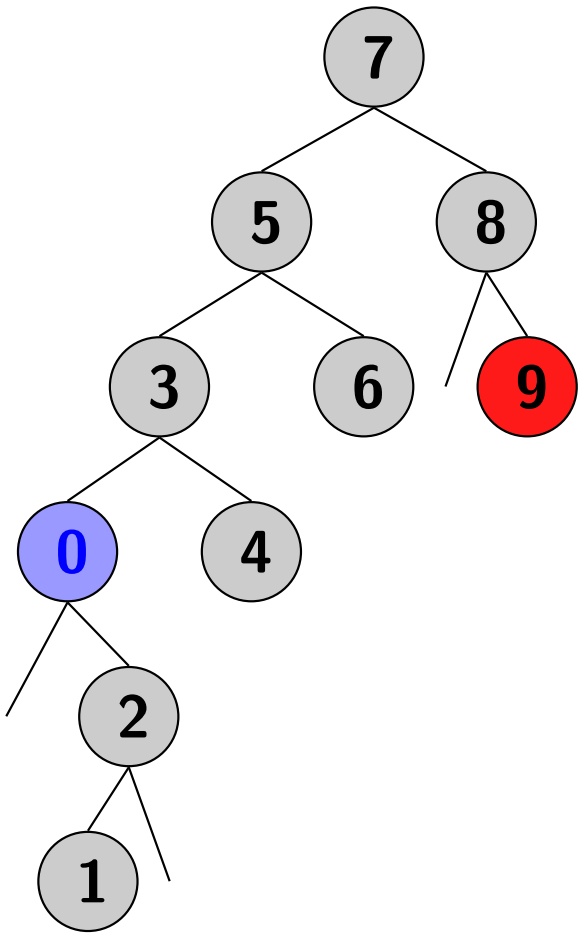
Se n chaves distintas são inseridas em ordem aleatória, a altura esperada da árvore é aproximadamente $4.311 \ln n$

Pior caso

A pior altura para n chaves é n.

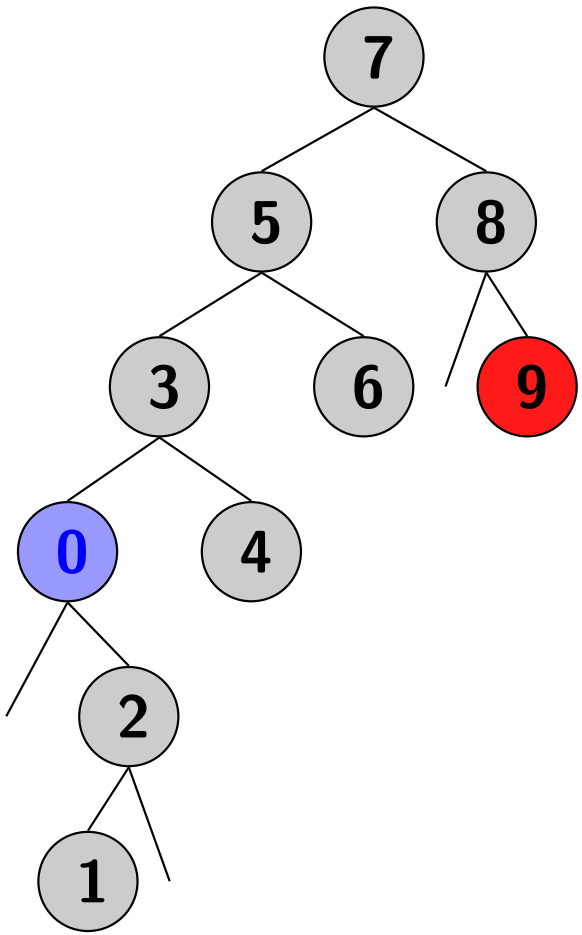
Operações ordenada em ABB

- Como encontrar a chave mínima e a máxima?



Operações ordenada em ABB

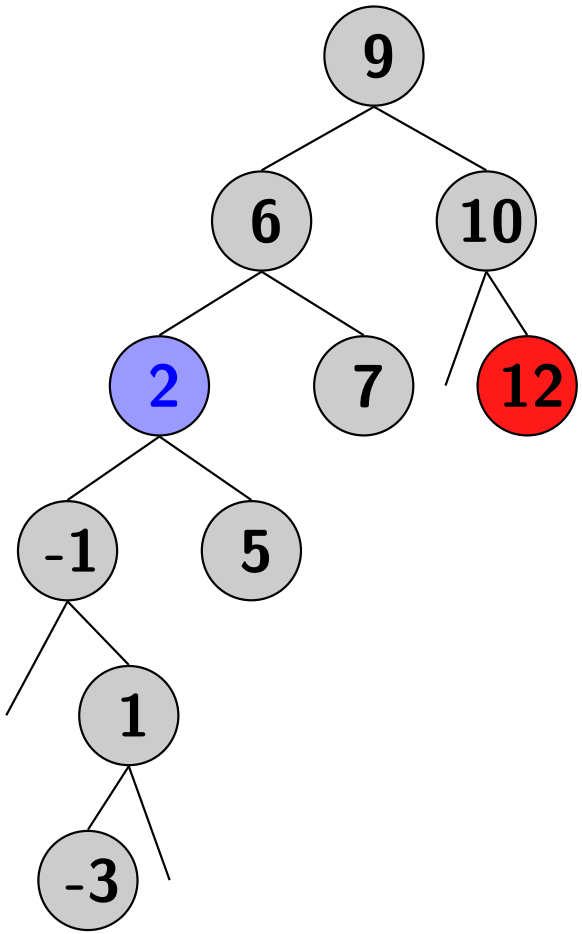
- Como encontrar a chave mínima e a máxima?




```
1  Chave min() {
2    return min(raiz);
3  }
4
5  Chave min(Node x) {
6   if (x.esq == null)
7     return x.chave;
8   else
9     return min(x.esq);
10 }
```

Operações ordenada em ABB

- Como encontrar o floor(4) e o ceiling(11)?

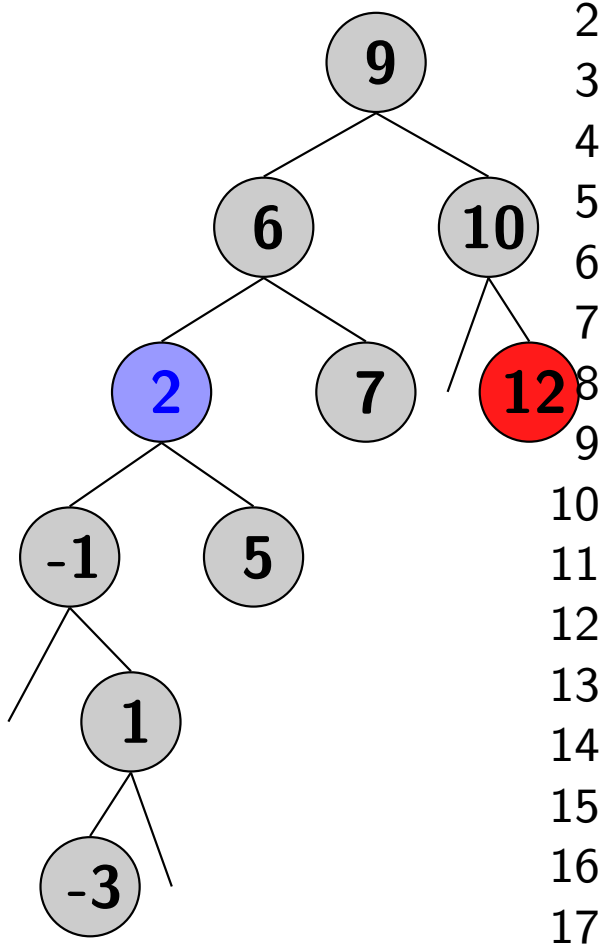


Casos para `floor(c)`

1. Se achou c, `floor(c) == c`
2. Se c for menor que do Node atual, `floor(c)` está na esquerda
3. Se c for maior, `floor(c)` está na direita, se existir. Se não existir, é a chave do Node atual

Operações ordenada em ABB

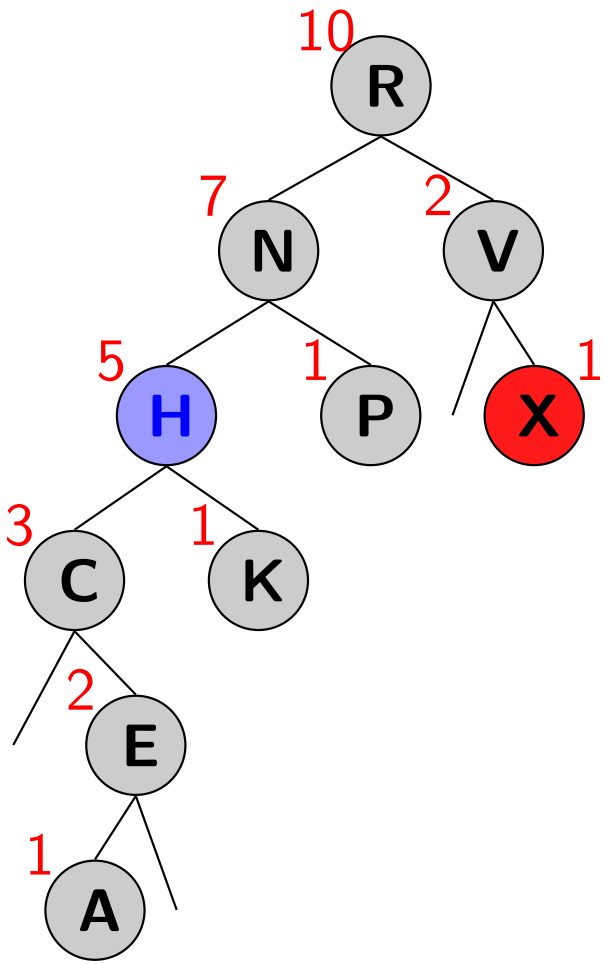
- Como encontrar o floor(4) e o ceiling(11)?




```
1  Chave floor(Chave c) {
2    Node x = floor(raiz, c);
3    return x.chave;
4  }
5  Node floor(Node x, Chave c) {
6   if (x == null) return null;
7
8   int cmp = c.compareTo(x.chave);
9   if (cmp == 0) // caso 1
10    return x;
11  if (cmp < 0) // caso 2
12    return floor(x.esq, c);
13
14  // caso 3
15  Node t = floor(x.dir, c);
16  if (t != null) return t;
17  else return x;
18 }
```

Operações ordenada em ABB

- Como implementar o rank(H) e o select()?
- `rank(H)` – quantas chaves menores que *H*
- `select(7)` – obter chave maior que 7 outras chaves




```
1 class Node {
2   Chave chave; Valor valor;
3   Node esq; Node dir;
4   int count; // NOVO:
        contagem subárvore
5 }
```

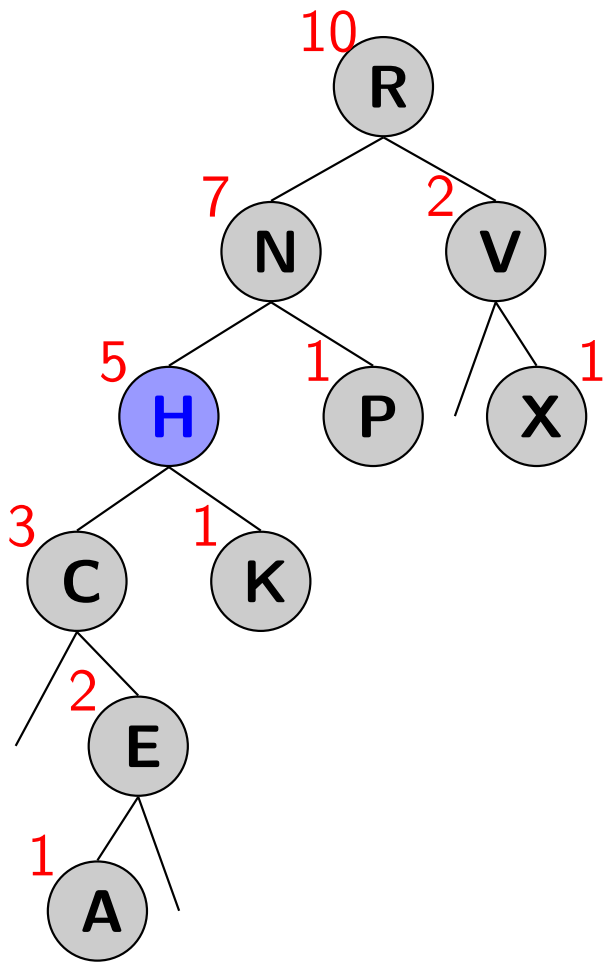
```
1 int size() { // na ABB
2   return size(raiz); //
        mais simples
3 }
4 int size(Node x) {
5   if (x == null) return 0;
6   else return x.count;
7 }
```

Atualização da contagem de subárvores

```
1  Node put(Node x, Chave c, Valor v) {
2      if (x == null) return new Node(c, v);
3
4      int cmp = c.compareTo(x.chave);
5      if      (cmp < 0) x.esq = put(x.esq, c, v);
6      else if (cmp > 0) x.dir = put(x.dir, c, v);
7      else              x.valor = v;
8
9      // mudou aqui: atualização da contagem
10     x.count = 1 + size(x.esq) + size(x.dir);
11     return x;
12 }
```

Operação `rank()`

- `rank(H)` – quantas chaves menores que H

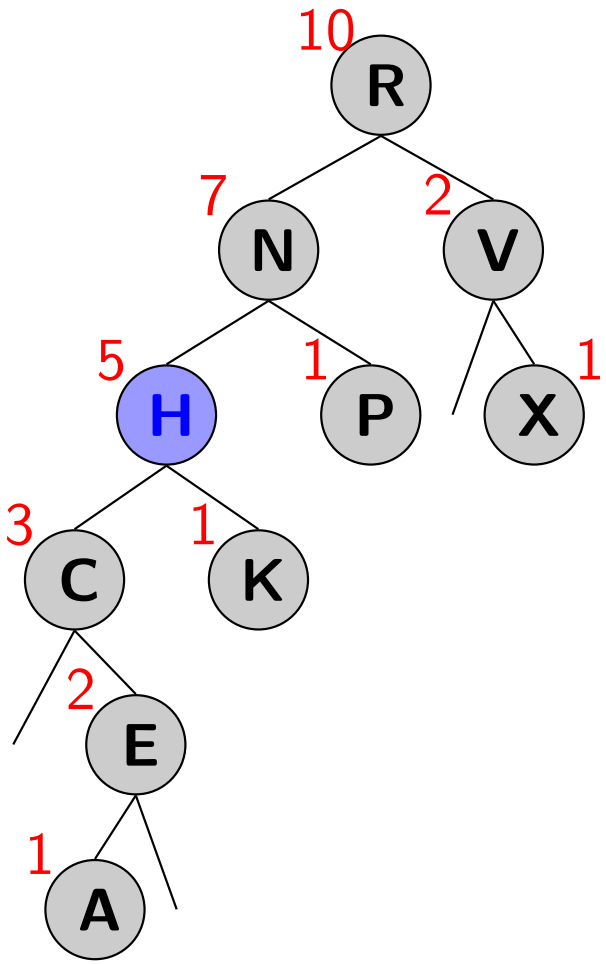


Quanto é?

- `rank(R)` é o número de elementos à esquerda de R
- `rank(V)` é `1+ size(R.esq) + size(V.esq)`
- `rank(X)` é `1+ size(R.esq) + rank(X, R.dir)`
- `rank(H)` é `size(H.esq)`

Operação `rank()`

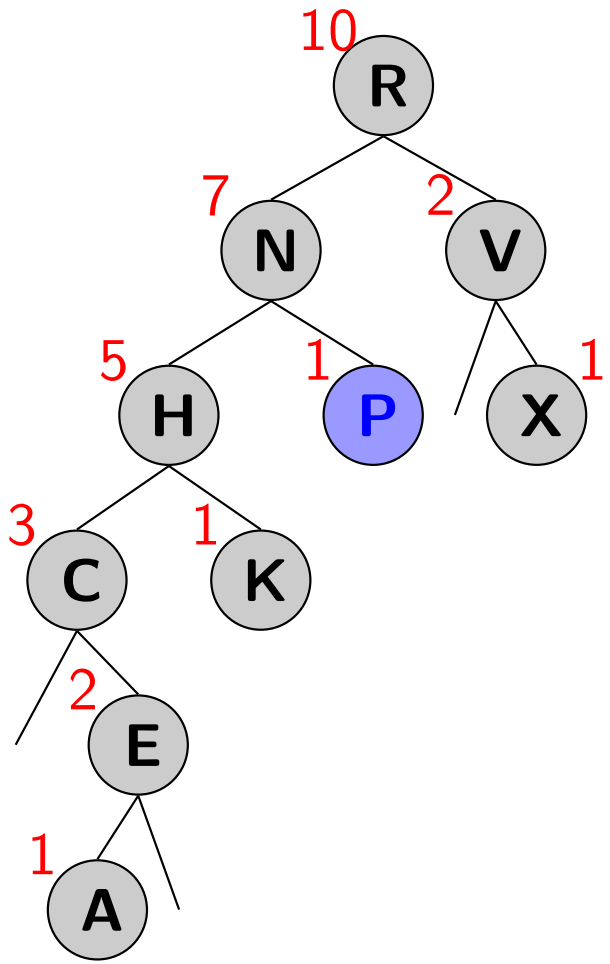
- `rank(H)` – quantas chaves menores que H ?




```
1  public int rank(Chave c) {
2    return rank(c, raiz);
3  }
4  int rank(Chave c, Node x) {
5   if (x == null) return 0;
6   int cmp=c.compareTo(x.chave);
7
8   if (cmp < 0)
9     return rank(c, x.esq);
10  else if (cmp > 0)
11    return 1 + size(x.esq)
12             + rank(c, x.dir);
13  else
14    return size(x.esq);
15 }
```

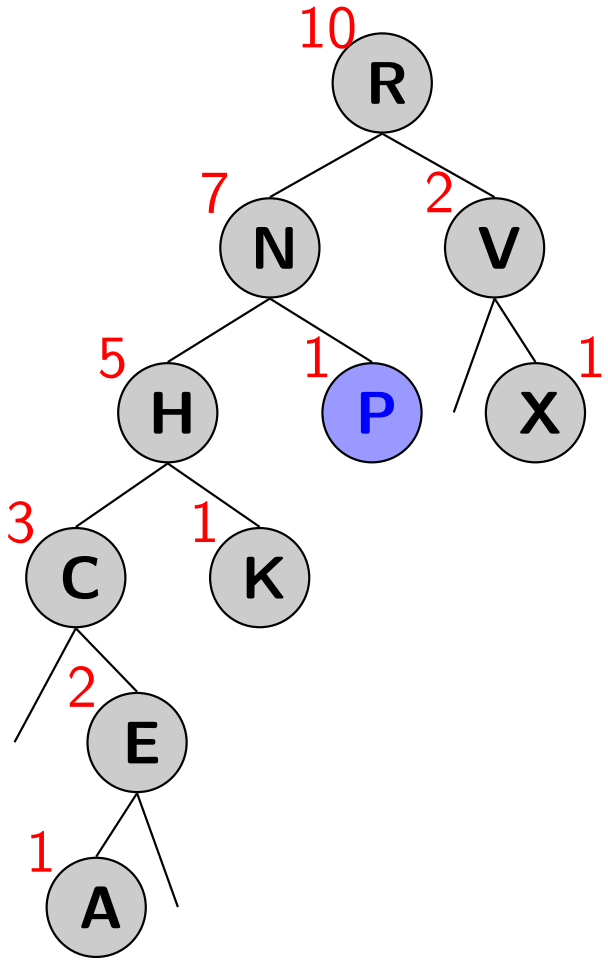
Operação `select()`

- `select(k)` – obter chave maior que exatamente k outras chaves



Busca-se chave com `rank` k:

- Se `size(atual.esq) > k`, procurar na esquerda
- Se `size(atual.esq) == k`, retornar `atual.chave`
- Se `size(atual.esq) < k`, procurar na direita chave com `rank = k - size(atual.esq) -1`



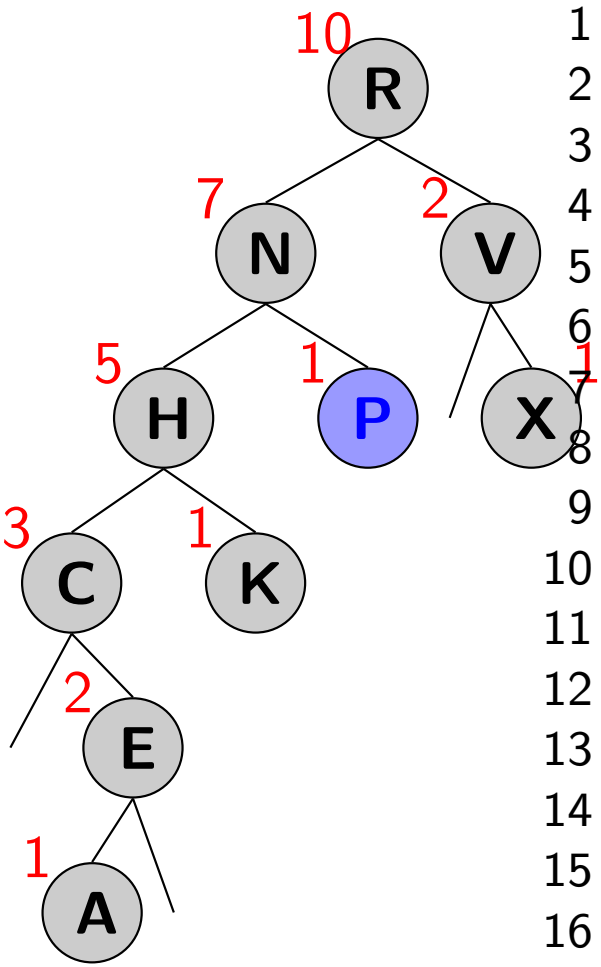

```
select(0) = A
select(1) = C
select(2) = E
select(3) = H
select(4) = K
select(5) = N
select(6) = P
select(7) = R
select(8) = V
select(9) = X
```

Busca-se chave com `rank` k:

- Se `size(atual.esq) > k`, procurar na esquerda
- Se `size(atual.esq) == k`, retornar `atual.chave`
- Se `size(atual.esq) < k`, procurar na direita chave com `rank = k - size(atual.esq) -1`

Operação `select()`

- `select(8)` – obter chave maior que exatamente k outras chaves




```
1  public Chave select(int k) {
2   return select(k, raiz);
3  }
4
5  Chave select(int k, Node atual) {
6   if (size(atual.esq) > k) {
7    return select(k, atual.esq);
8   } else if (size(atual.esq) == k) {
9    return atual.chave;
10  } else {
11   return select(k
12                 -size(atual.esq)
13                 -1,
14                 atual.dir);
15  }
16 }
```

Operação `keys()`: travessia em ordem `inorder()`

Passos da travessia

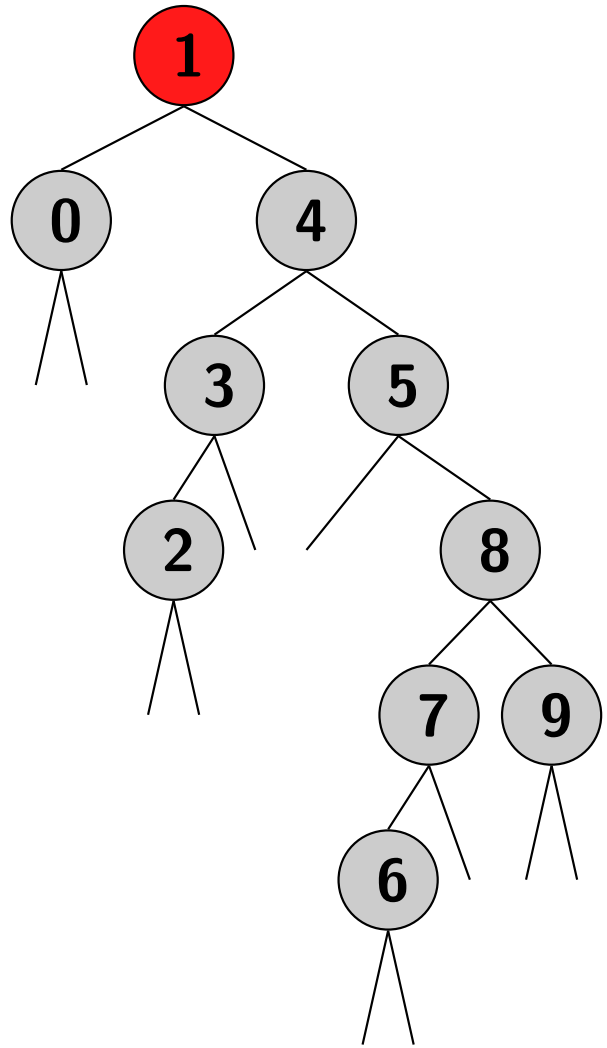
- Faz a travessia da subárvore à esquerda
- Enfilera a chave atual (visita)
- Faz a travessia subarvore à direita

```
1 Iterable<Chave> keys() {
2   Queue<Chave> f = new
        LinkedList<Chave>();
3   inorder(raiz, f);
4   return f;
5 }
```

```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

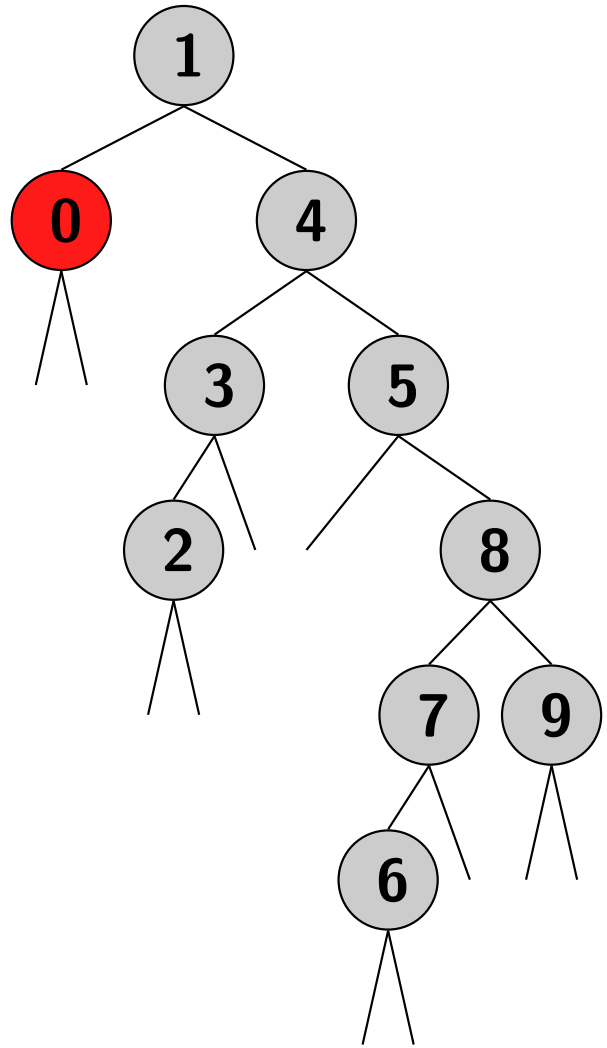
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila:



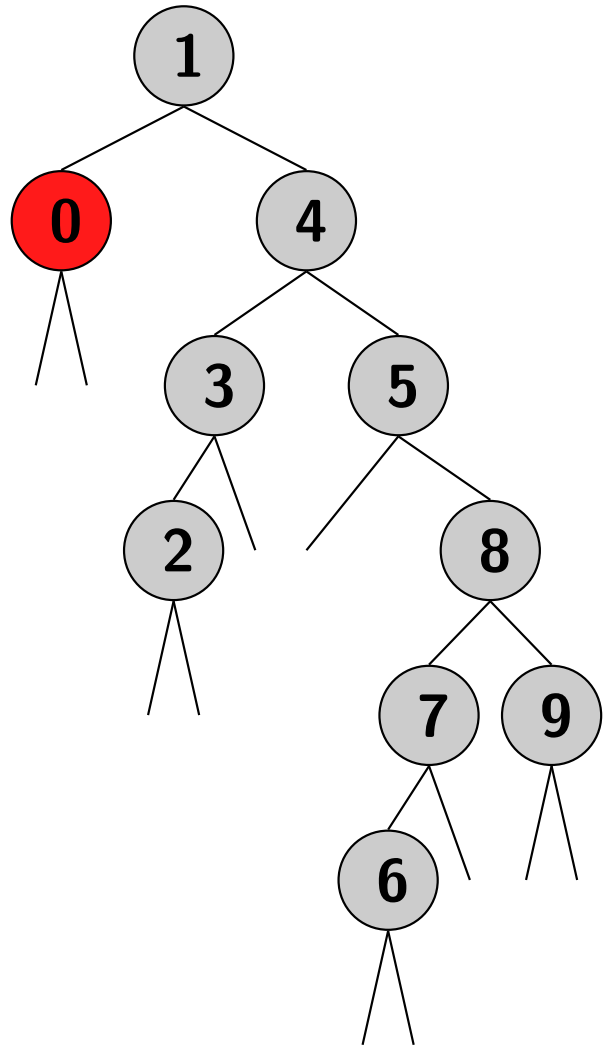
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila:



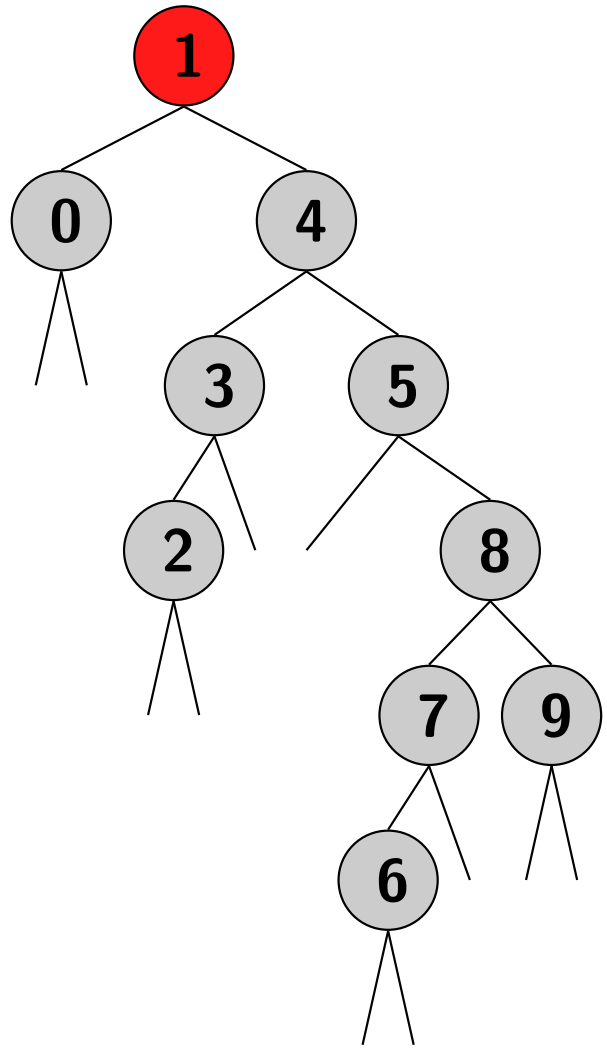
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0,



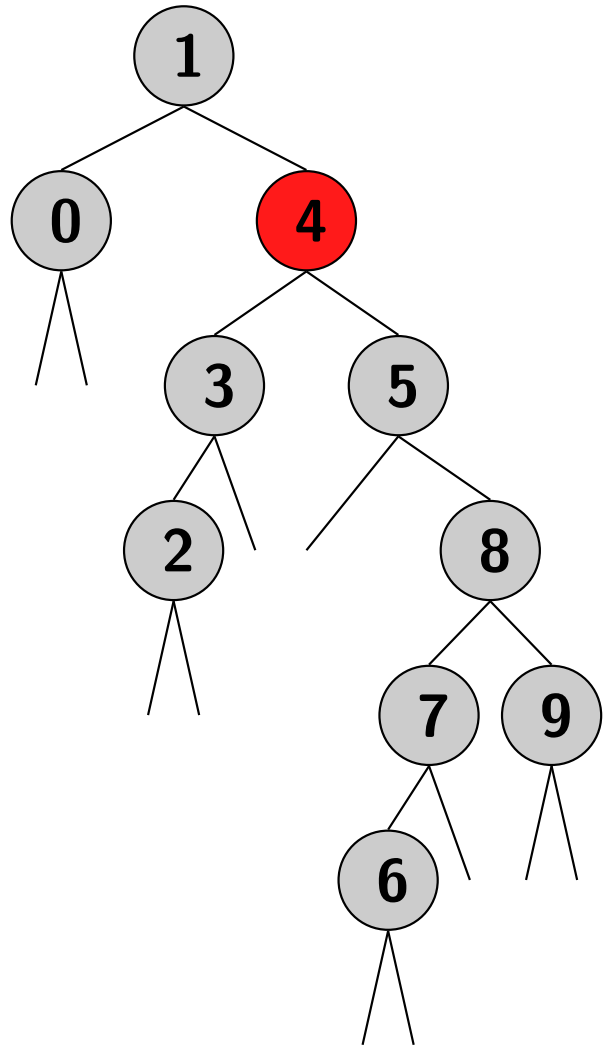
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1,



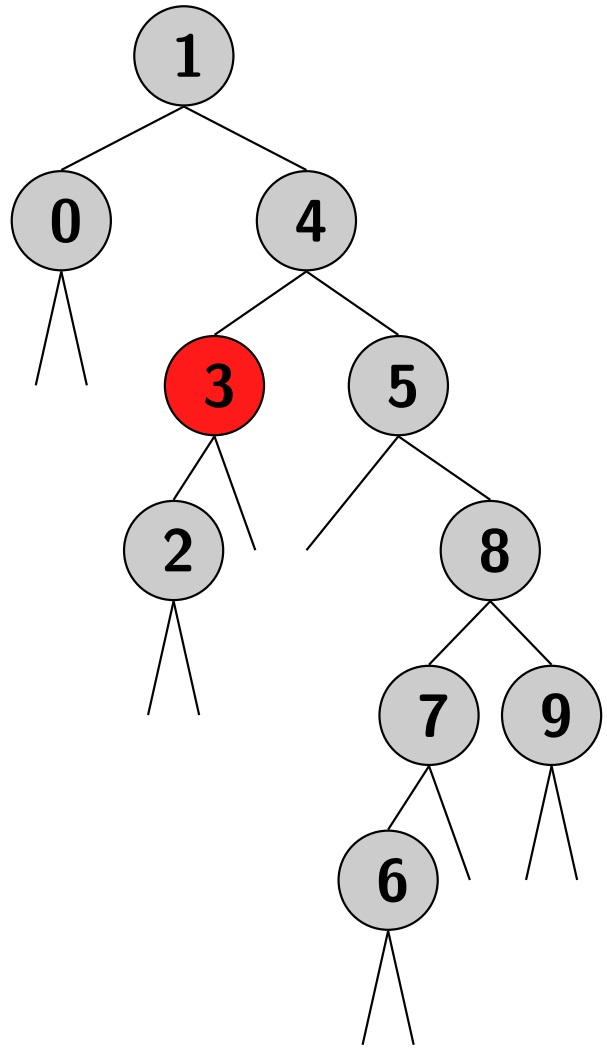
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1,



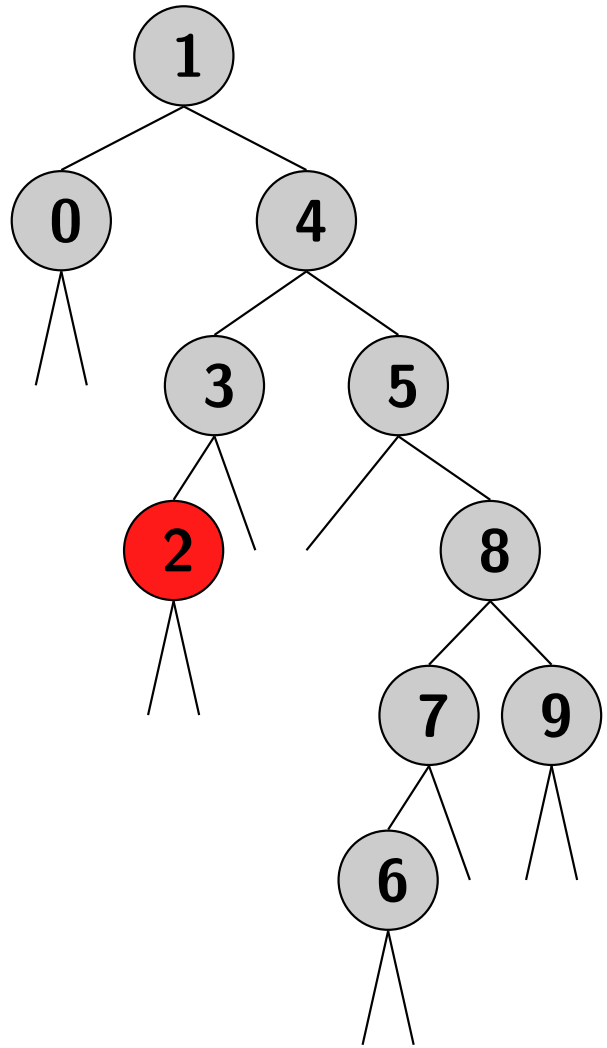
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1,



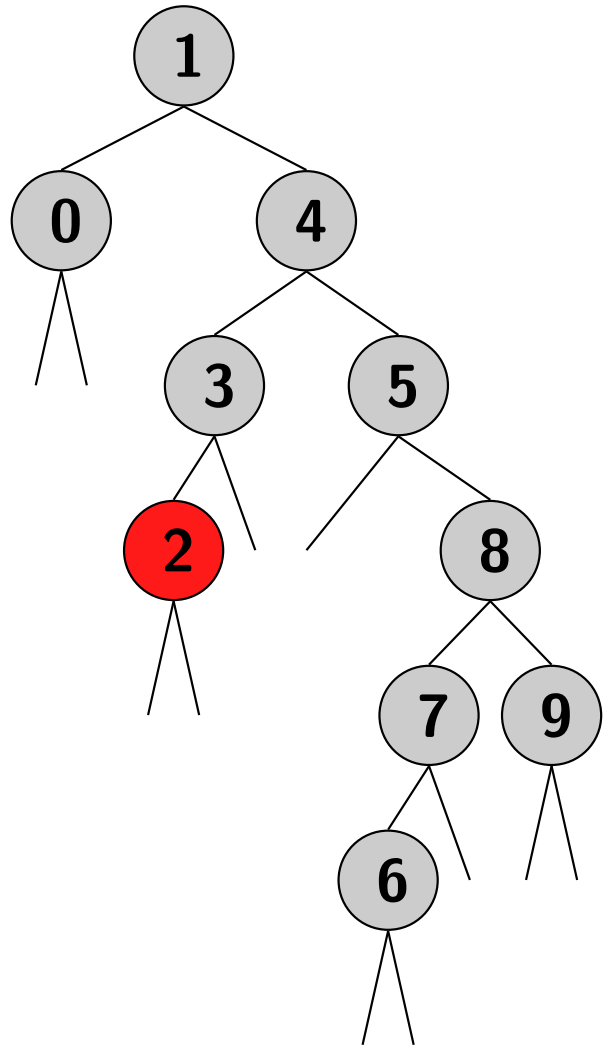
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1,



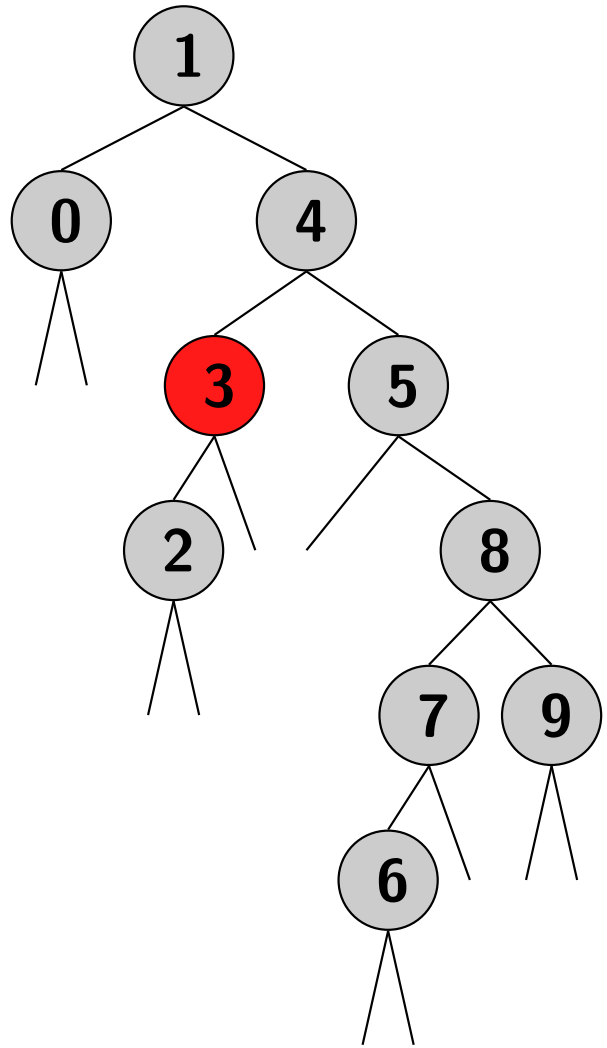
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2,



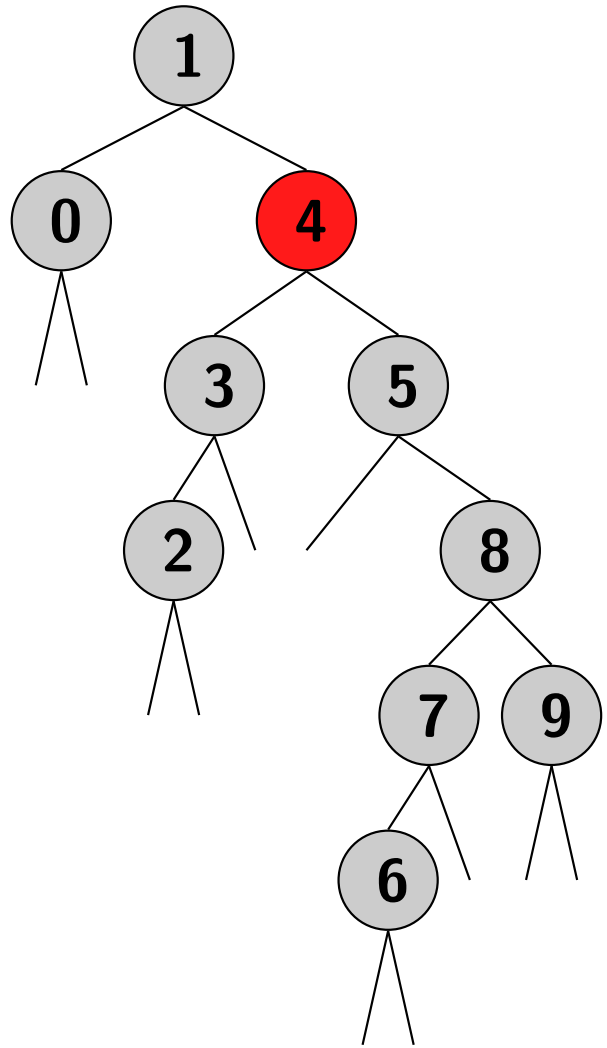
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3,



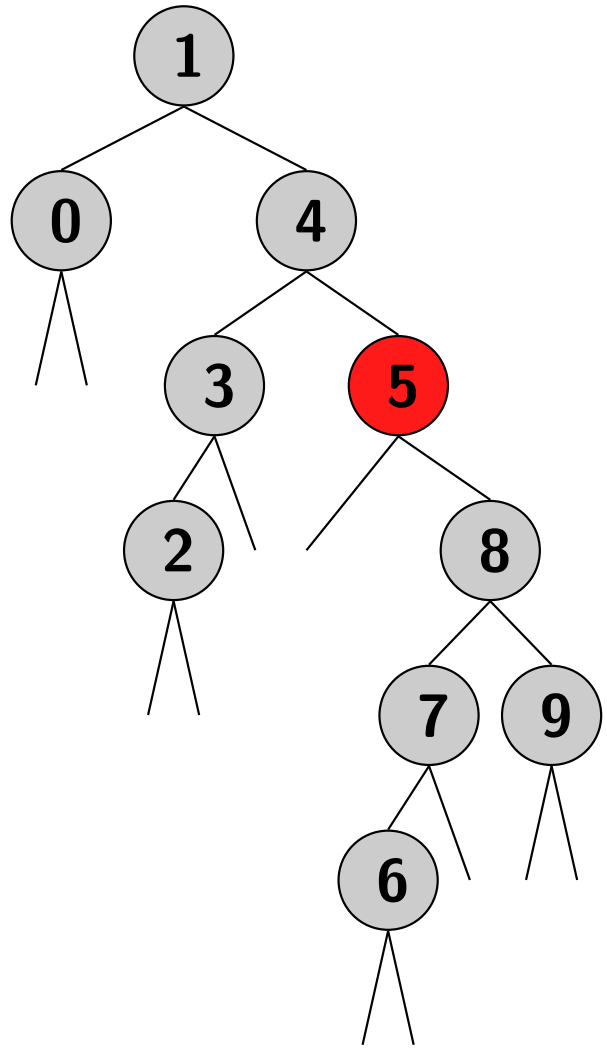
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4,



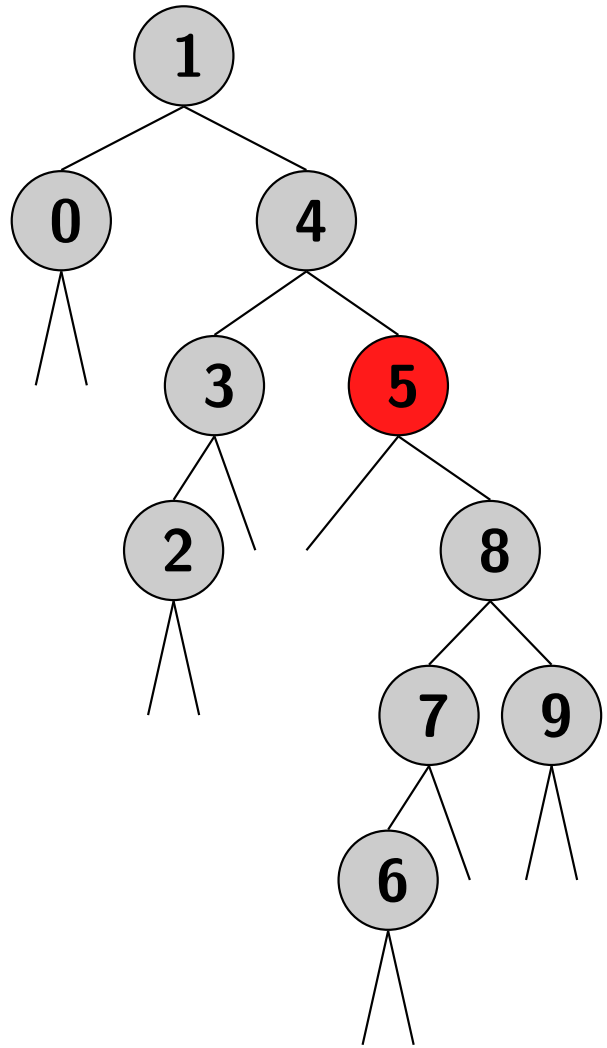
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4,



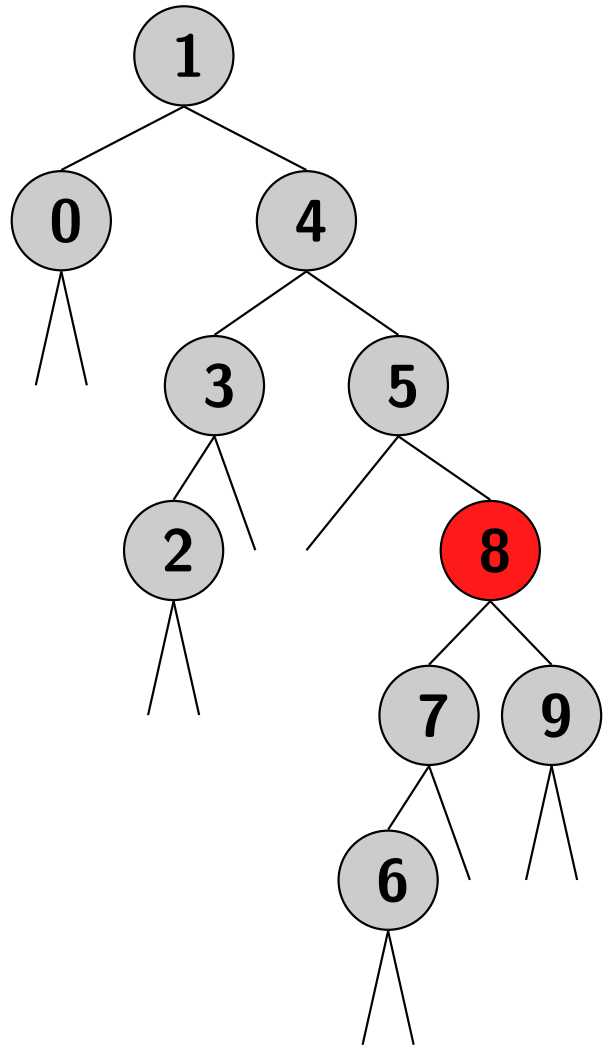
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5,



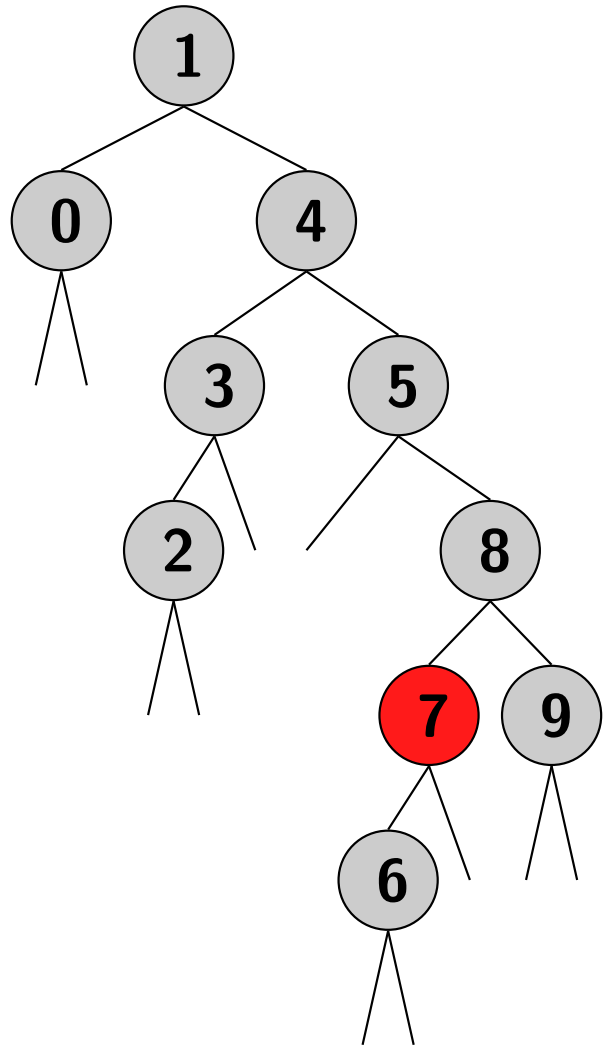
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5,



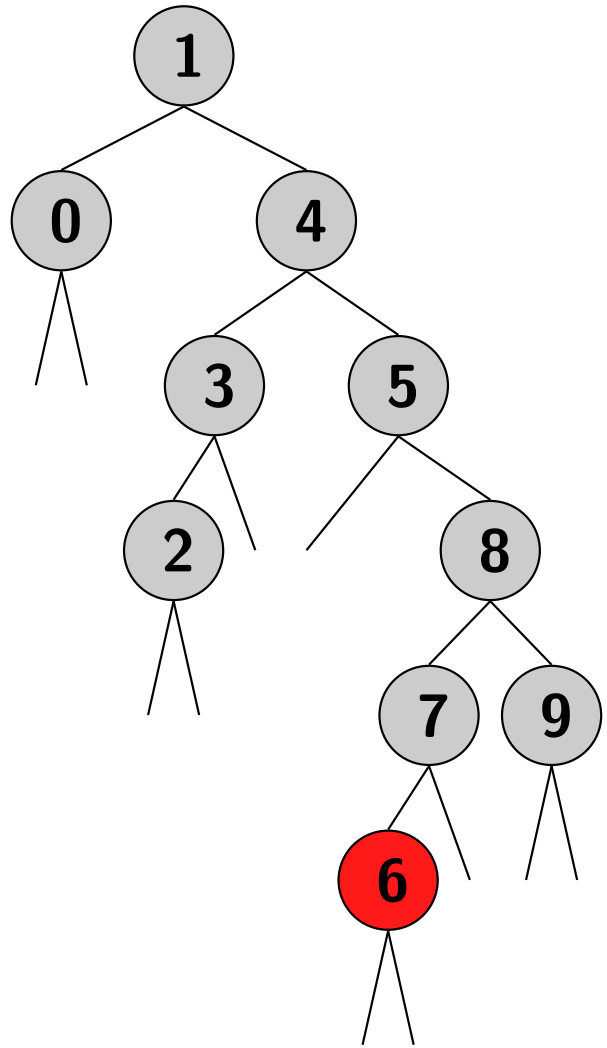
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5,



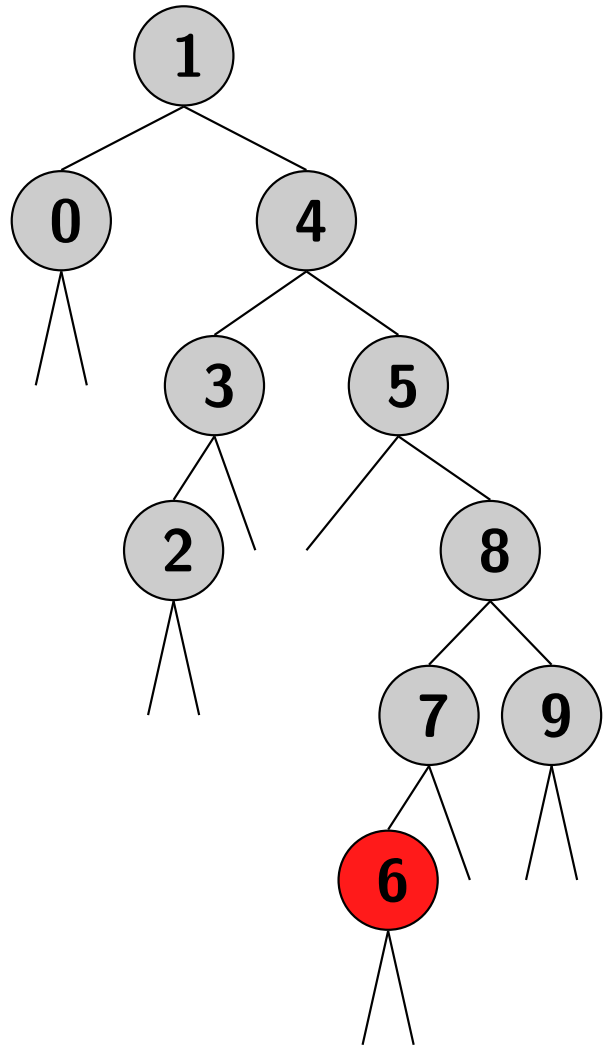
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5,



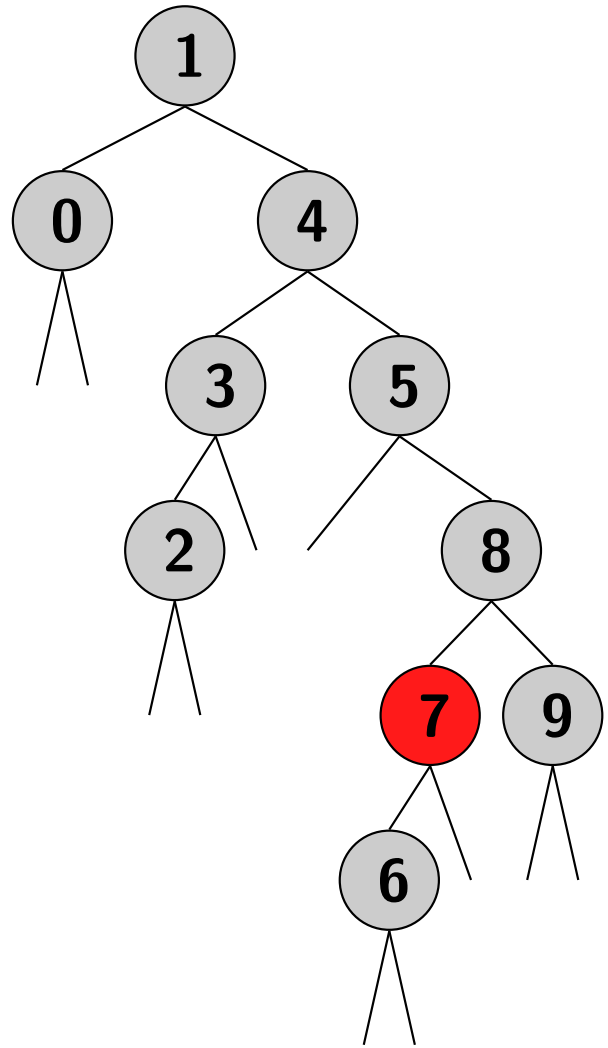
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5, 6,



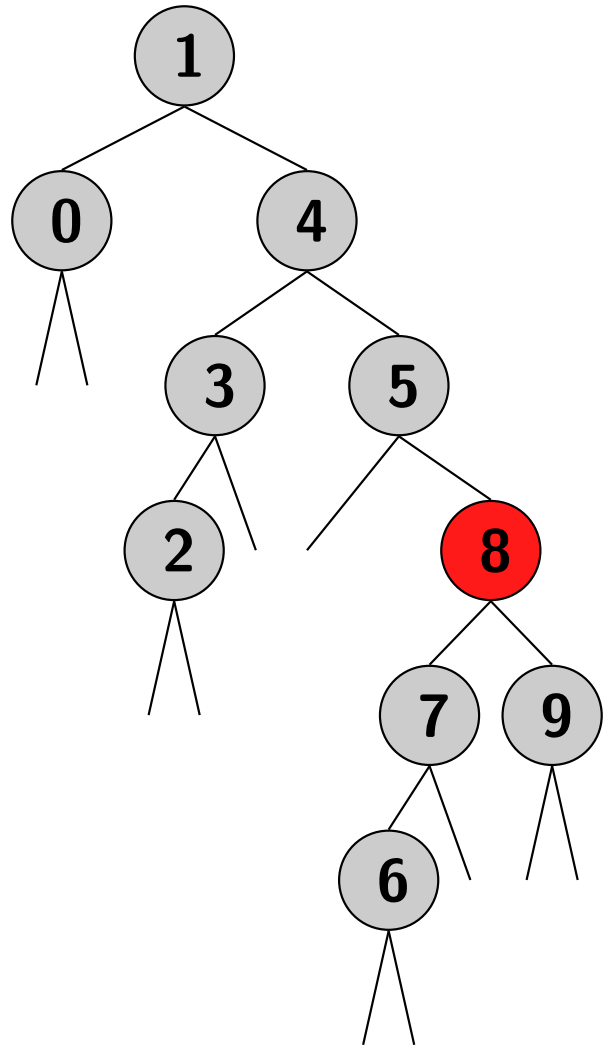
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5, 6, 7,



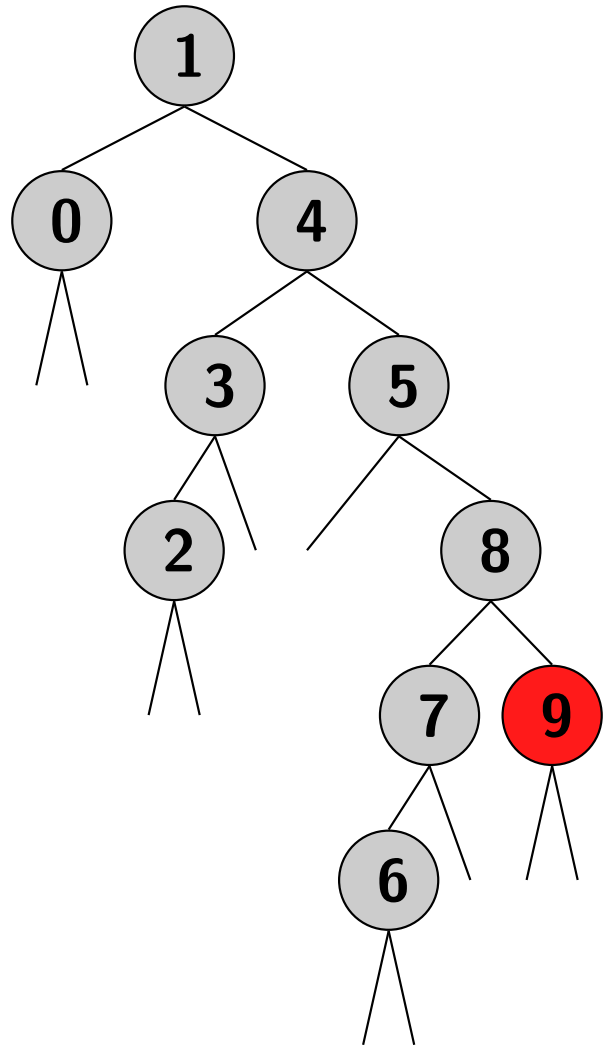
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5, 6, 7, 8,



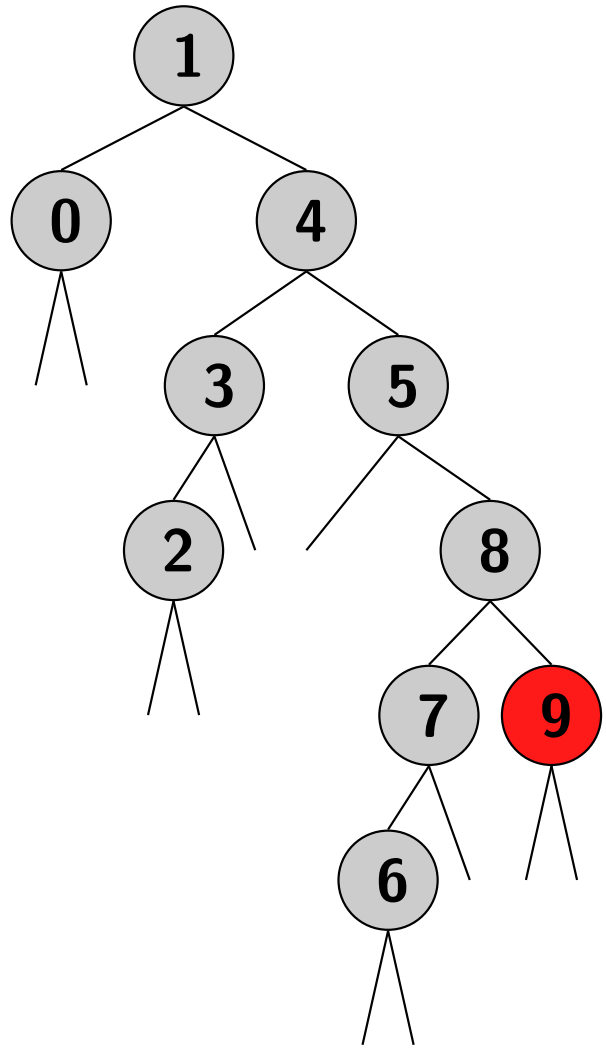
```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5, 6, 7, 8,



```
1 void inorder(Node x, Queue<
      Chave> f) {
2   if (x == null) return;
3
4   inorder(x.esq, f); //
        travessia
5   f.add(x.chave); //
        enfilera
6   inorder(x.dir, f); //
        travessia
7 }
```

Fila: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,



Outras travessias

Travessia pré-ordem

Aplicação principal: clonar árvore.

- Enfilera a chave atual (visita)
- Faz a travessia da subárvore à esquerda
- Faz a travessia subarvore à direita

Travessia pós-ordem

Aplicação principal: destruir árvore.

- Faz a travessia da subárvore à esquerda
- Faz a travessia subarvore à direita
- Enfilera a chave atual (visita)

Sumário: operações de tabela de símbolos

Na ABB os custos são proporcionais à altura da árvore h.

	busca sequencial	**busca binária**	**ABB**
`get`	N	log N	h
`put`	1	N	h
`min` / `max`	N	1	h
`floor` / `ceiling`	N	log N	h
`rank`	N	log N	h
`select`	N	1	h
`keys`	N log N	N	N

Operação `delete()`: abordagem preguiçosa

Maneira simples

- Usar `put(c null)`, para remover chave `c`
- Mas na verdade não remove, só marca que removeu
- Espaço continua sendo ocupado
- Bom para poucas deleções

Operação `void deleteMin()`

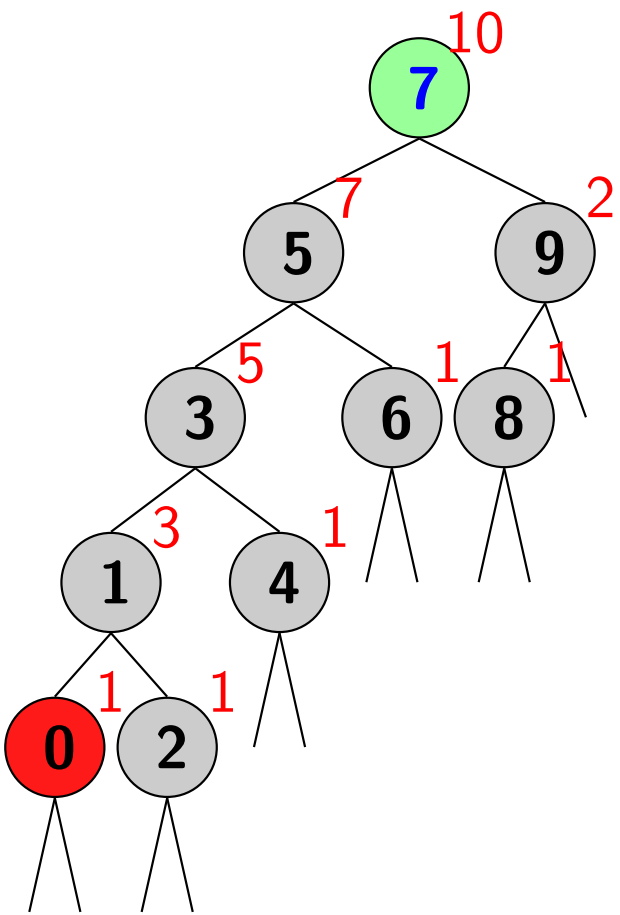
Para remover o mínimo com `deleteMin()` é necessário:

- Ir ao Node mais na esquerda da árvore
- Substituir Node pelo seu próprio Node à direita
- Atualizar contagens

```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

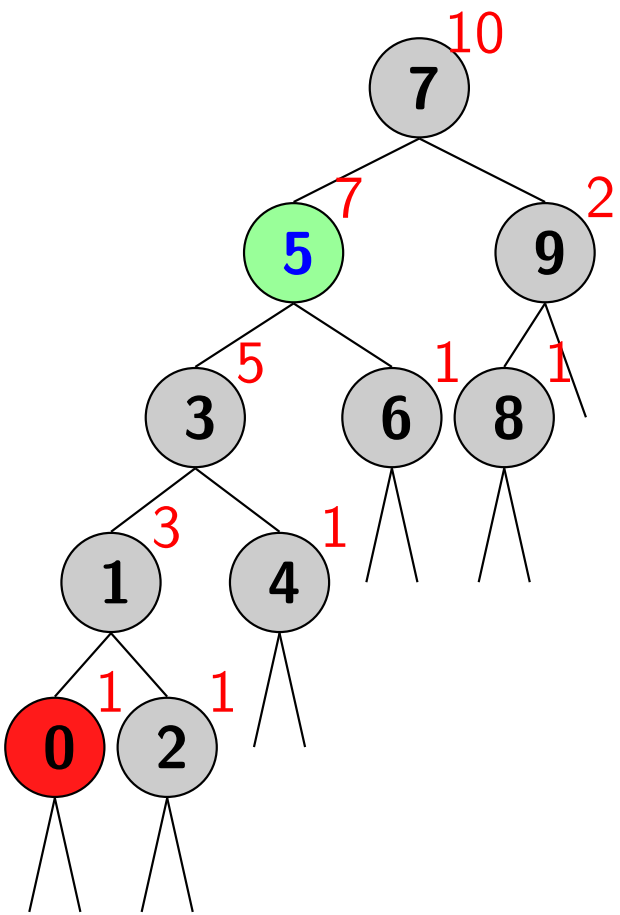
Removendo o mínimo: 0




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

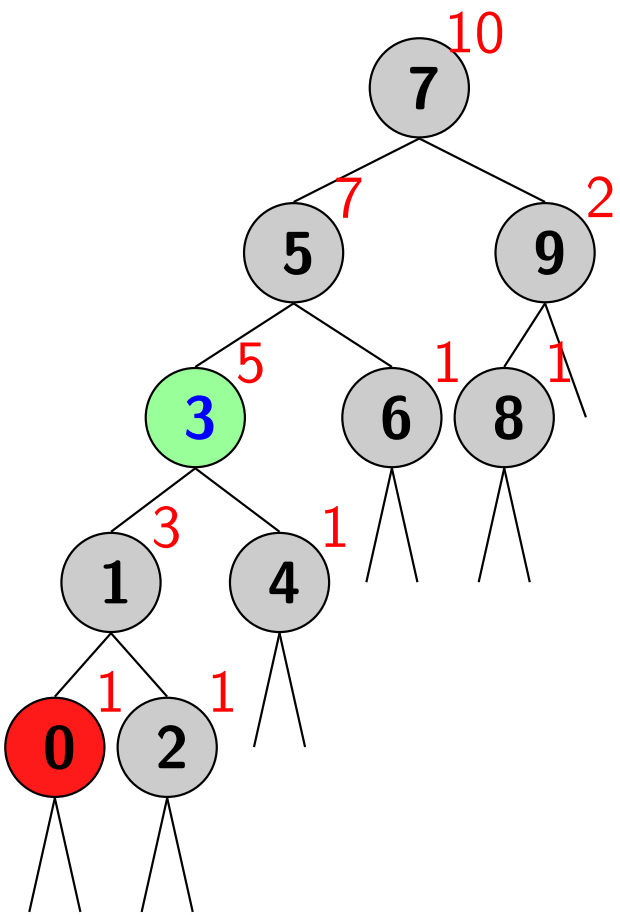
Removendo o mínimo: 0




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

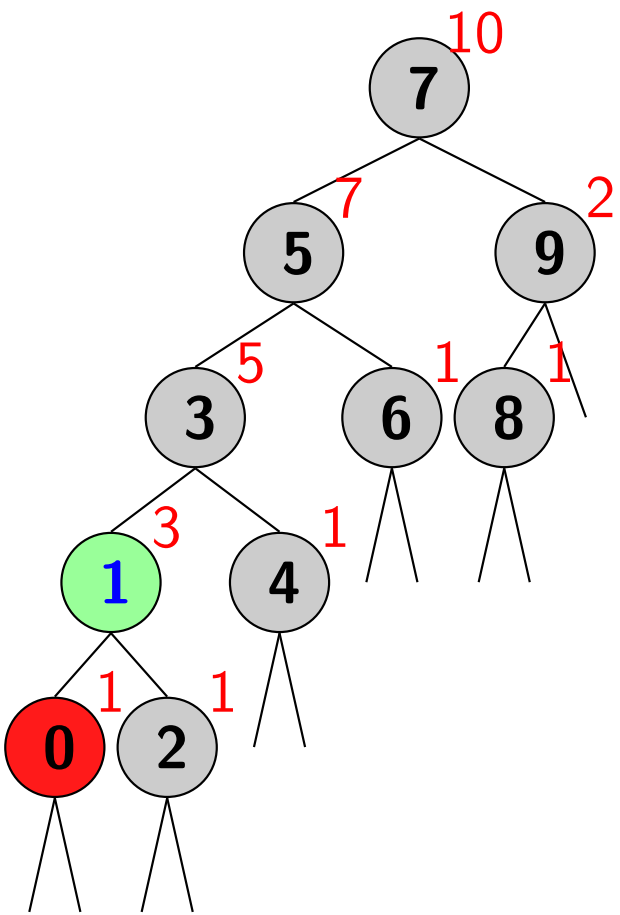
Removendo o mínimo: 0




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

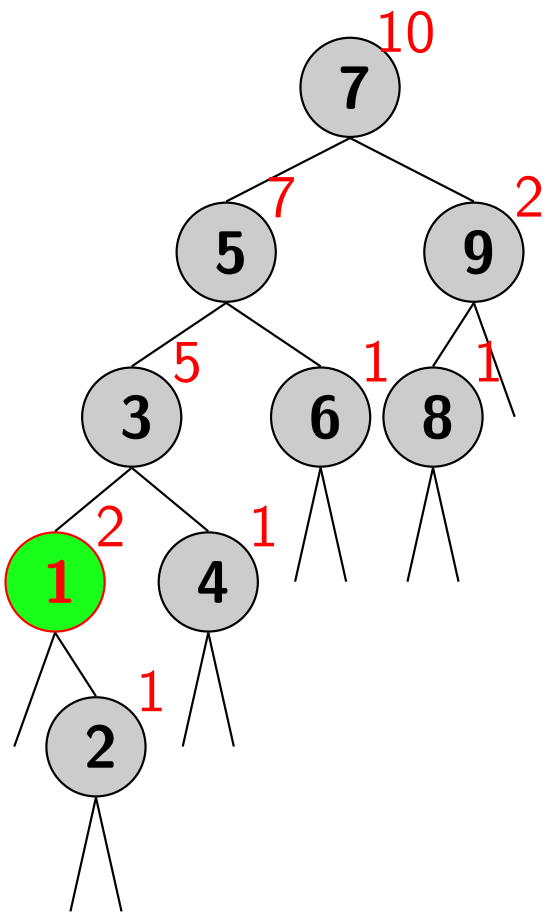
Removendo o mínimo: 0




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

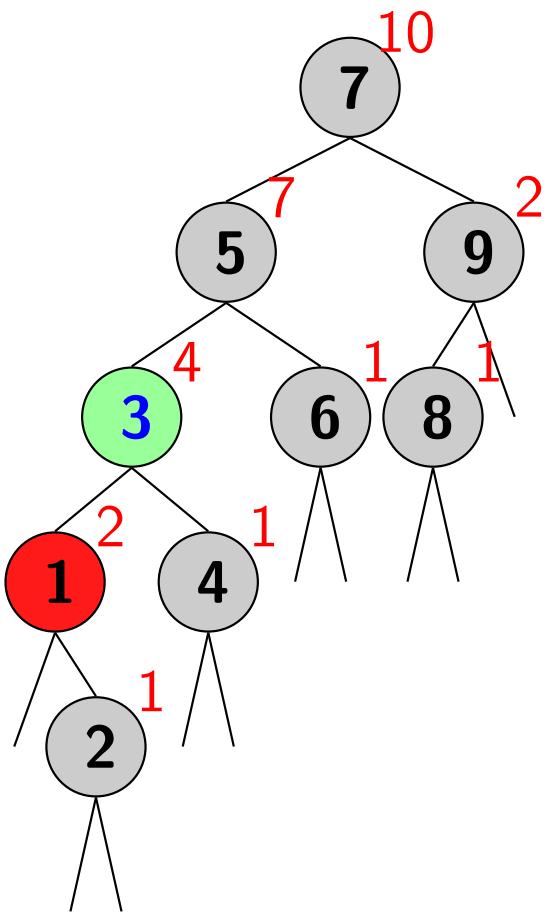
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

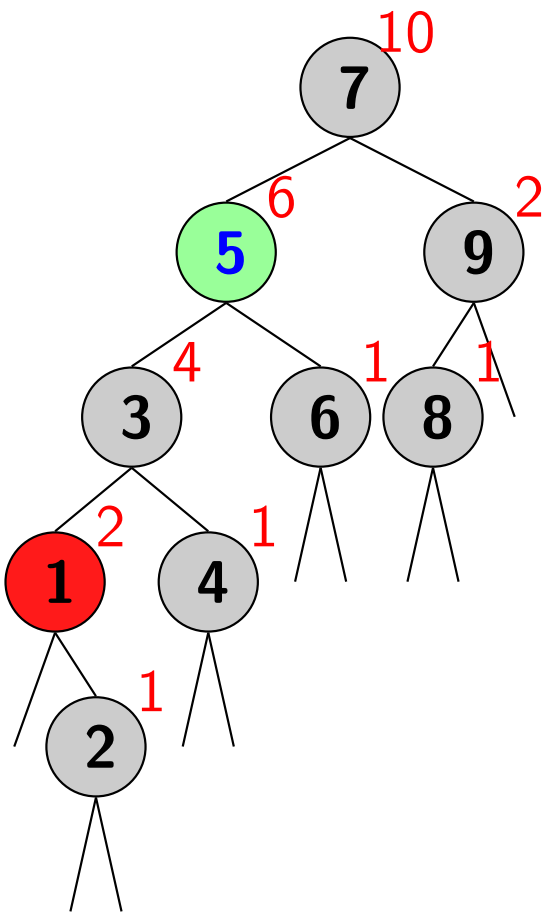
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

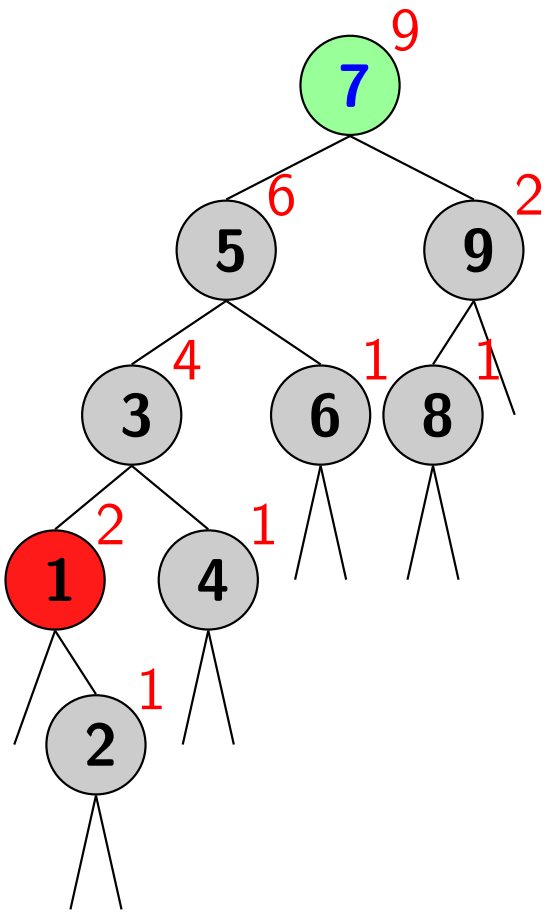
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

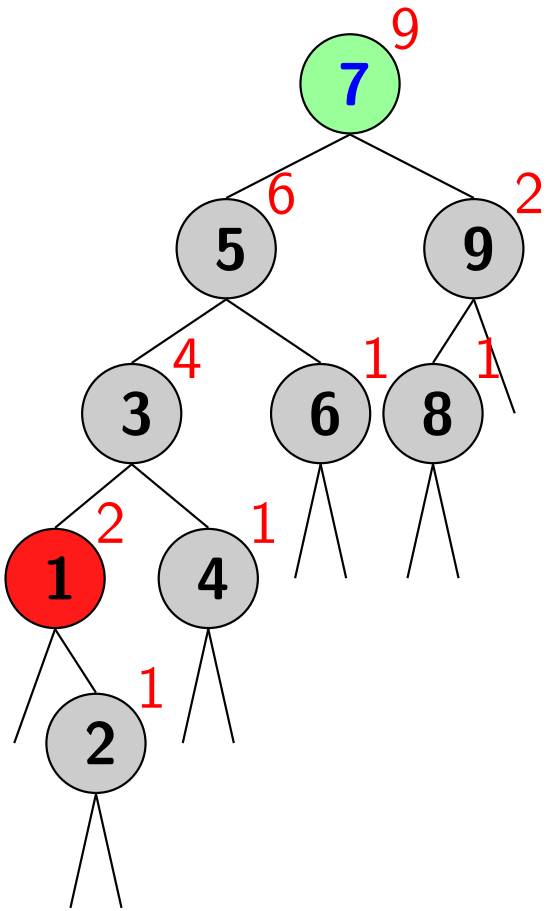
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

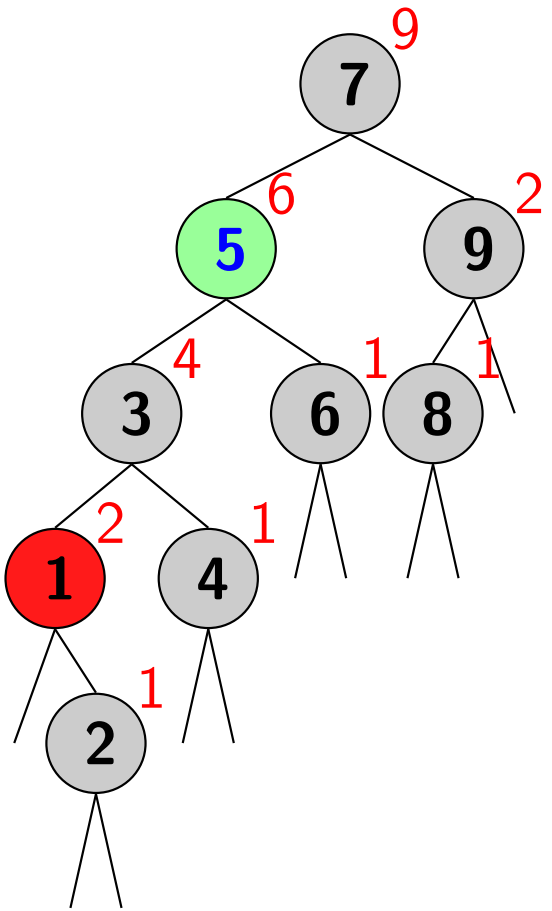
Removendo o mínimo: 1




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

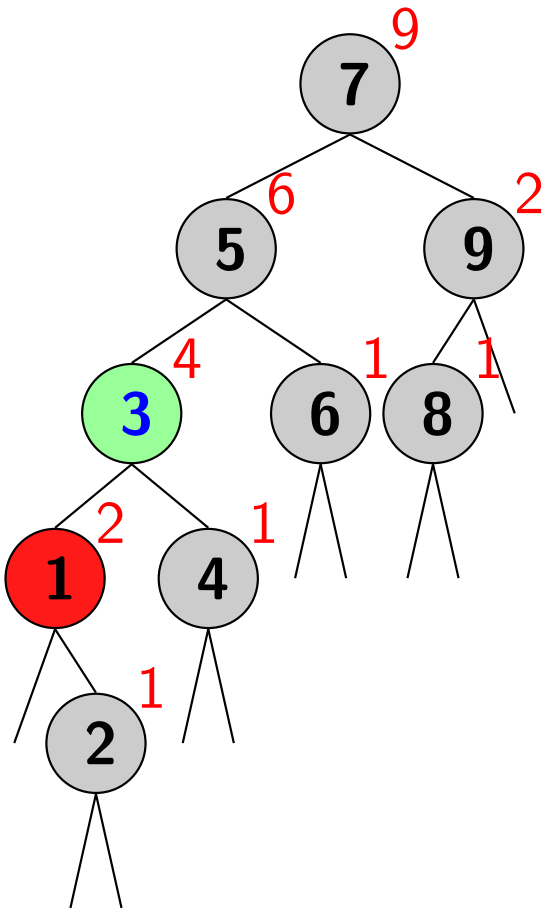
Removendo o mínimo: 1




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

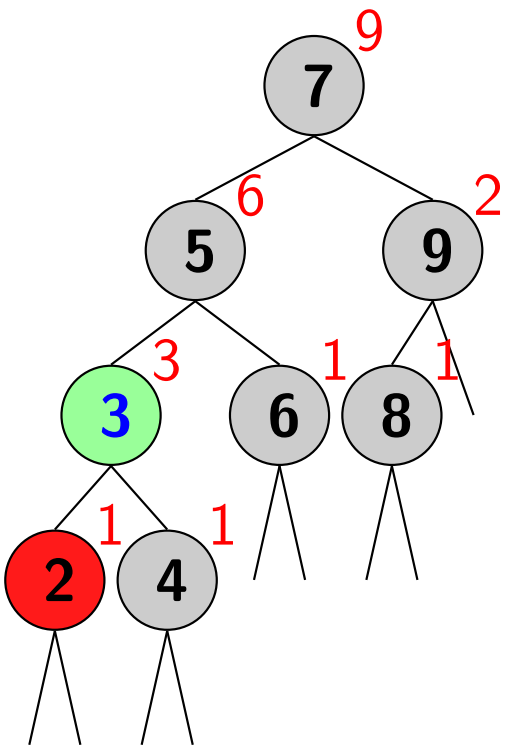
Removendo o mínimo: 1




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

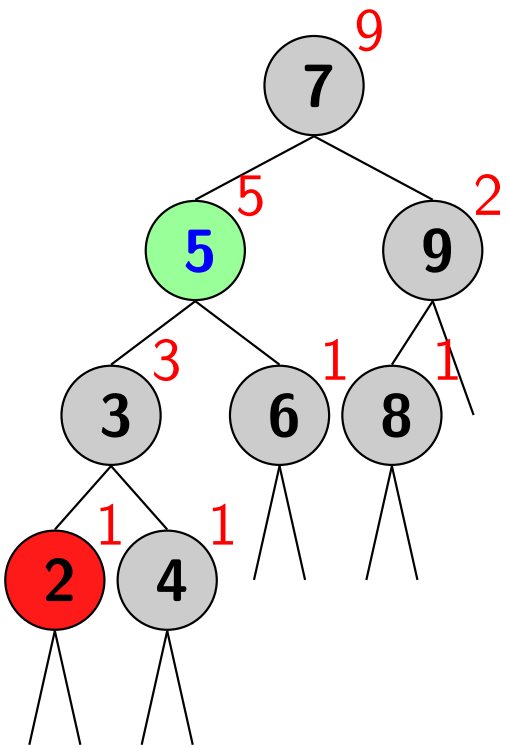
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

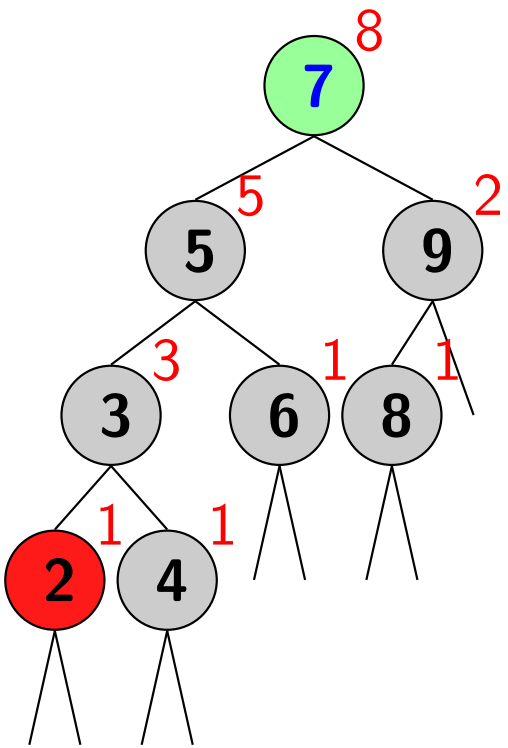
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

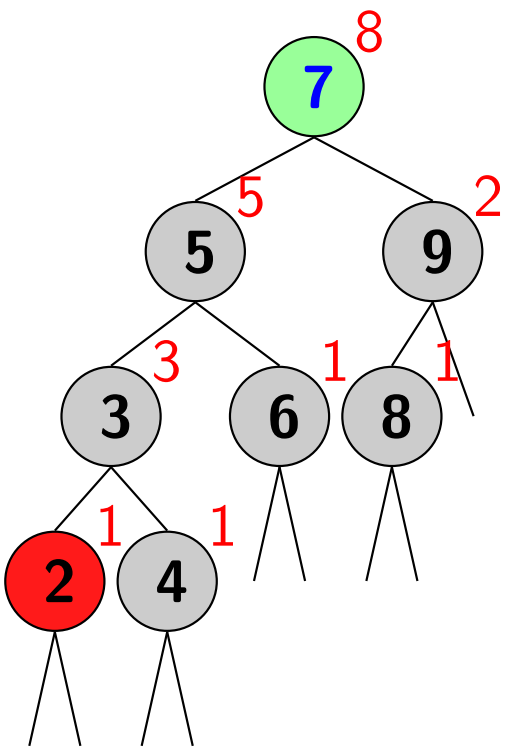
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

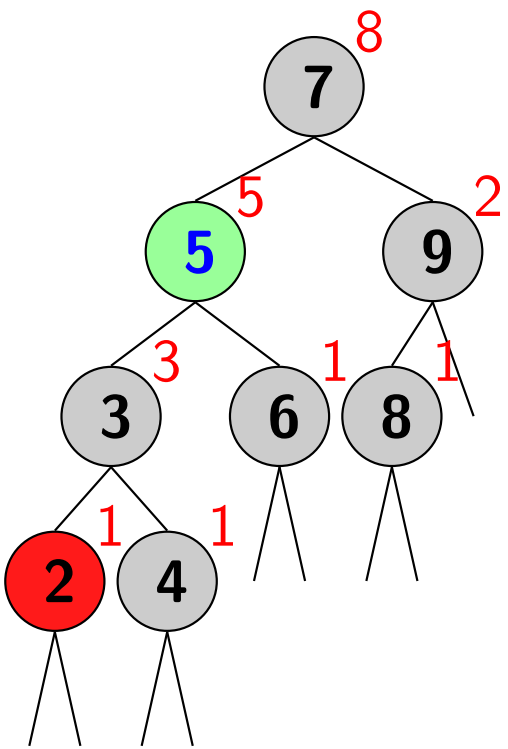
Removendo o mínimo: 2




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

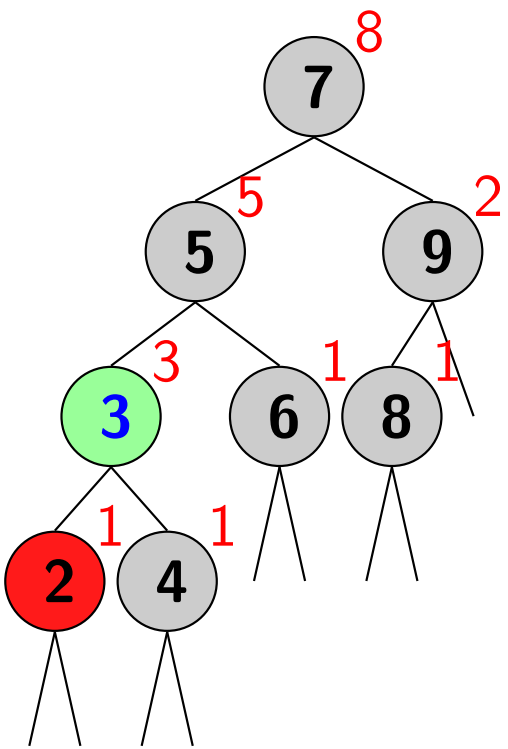
Removendo o mínimo: 2




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

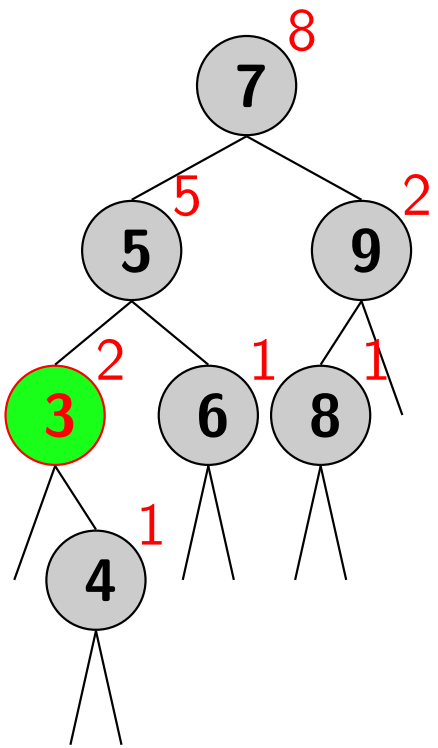
Removendo o mínimo: 2




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

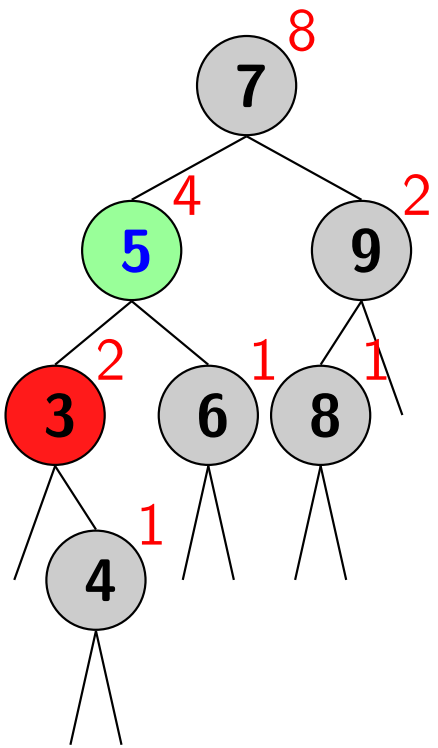
Atualizando contagens




```
 1 void deleteMin() {
 2   raiz = deleteMin(raiz);
 3 }
 4
 5 Node deleteMin(Node x) {
 6   if (x.esq == null)
 7     return x.dir;
 8
 9   x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

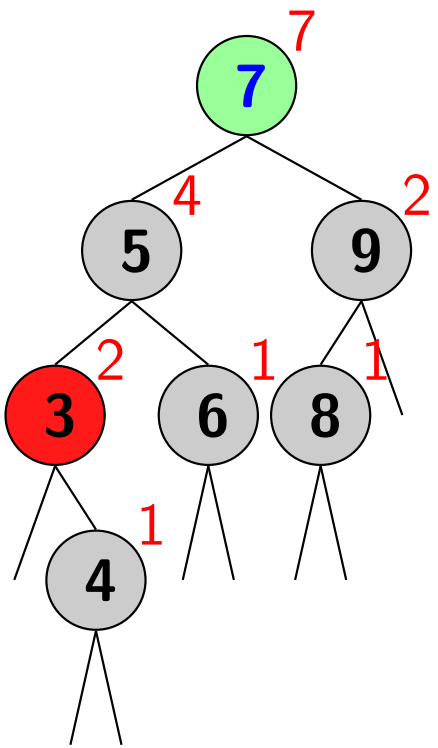
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

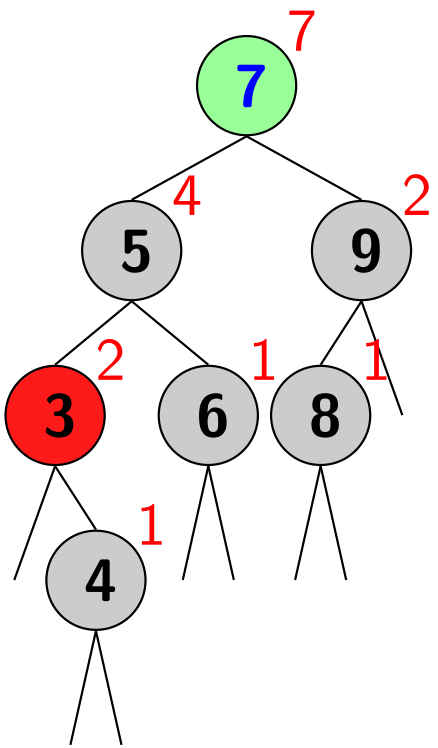
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

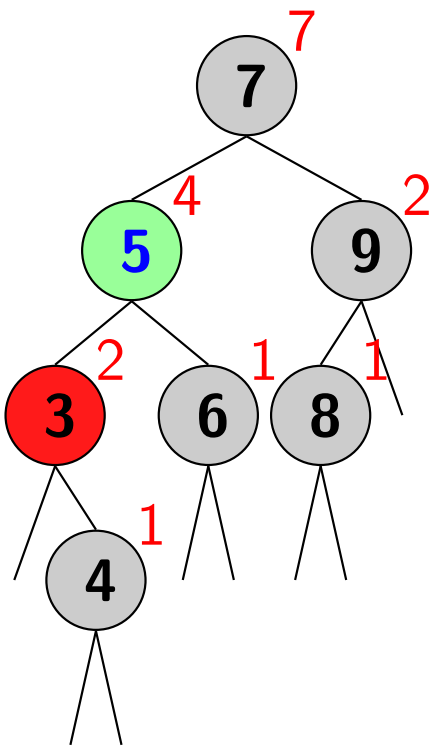
Removendo o mínimo: 3




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

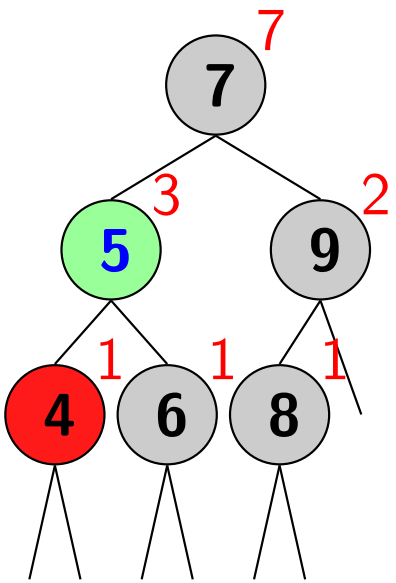
Removendo o mínimo: 3




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

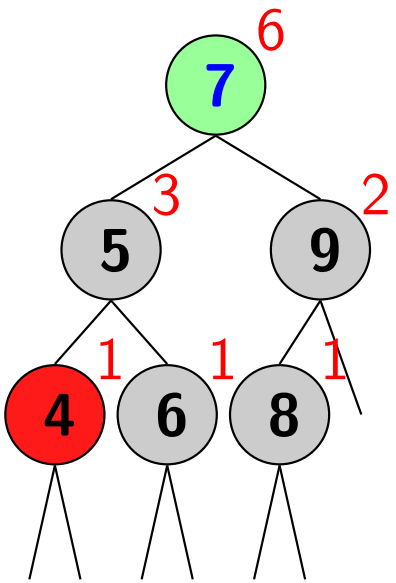
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

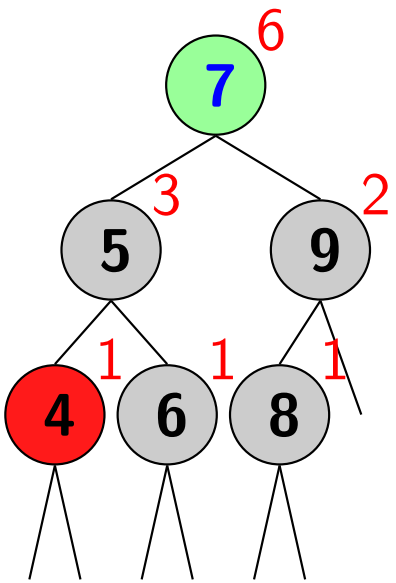
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

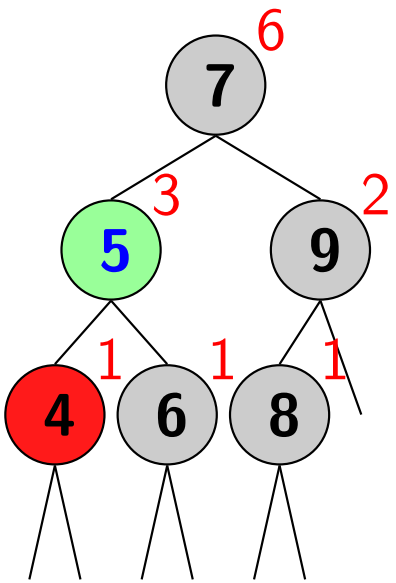
Removendo o mínimo: 4




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

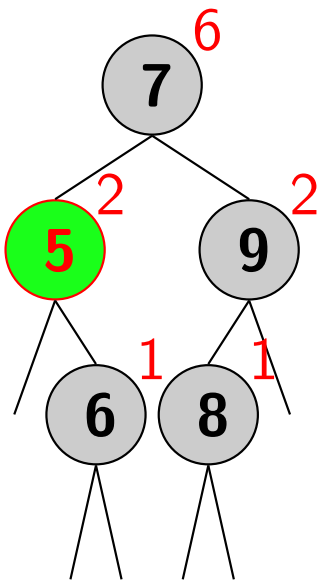
Removendo o mínimo: 4




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

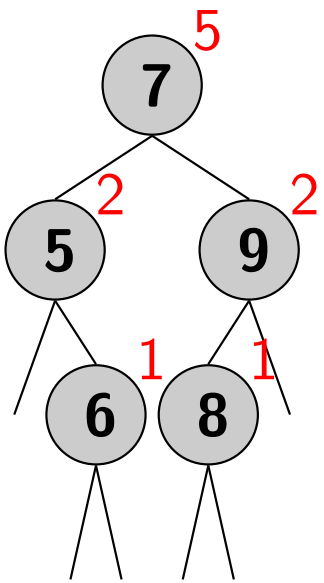
Atualizando contagens




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

Operação `void deleteMin()`

Atualizado.




```
1  void deleteMin() {
2    raiz = deleteMin(raiz);
3  }
4
5  Node deleteMin(Node x) {
6    if (x.esq == null)
7      return x.dir;
8
9    x.esq = deleteMin(x.esq);
10   x.count = 1
11           + size(x.esq)
12           + size(x.dir);
13
14   return x;
15 }
```

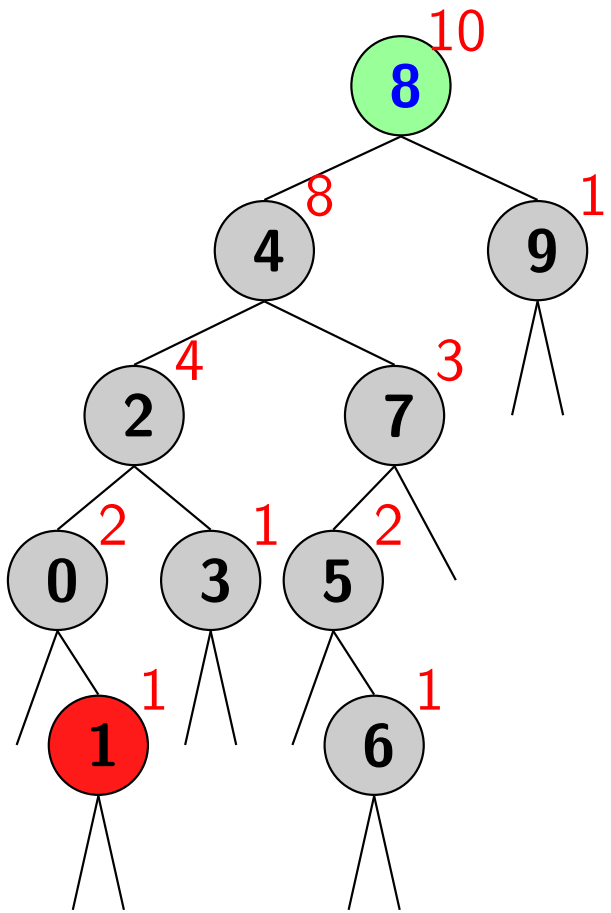
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



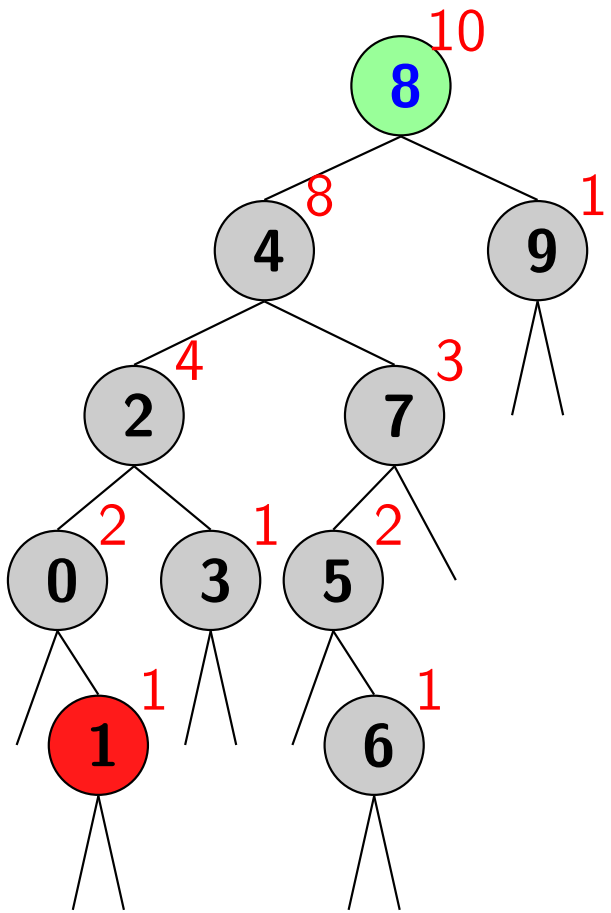
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



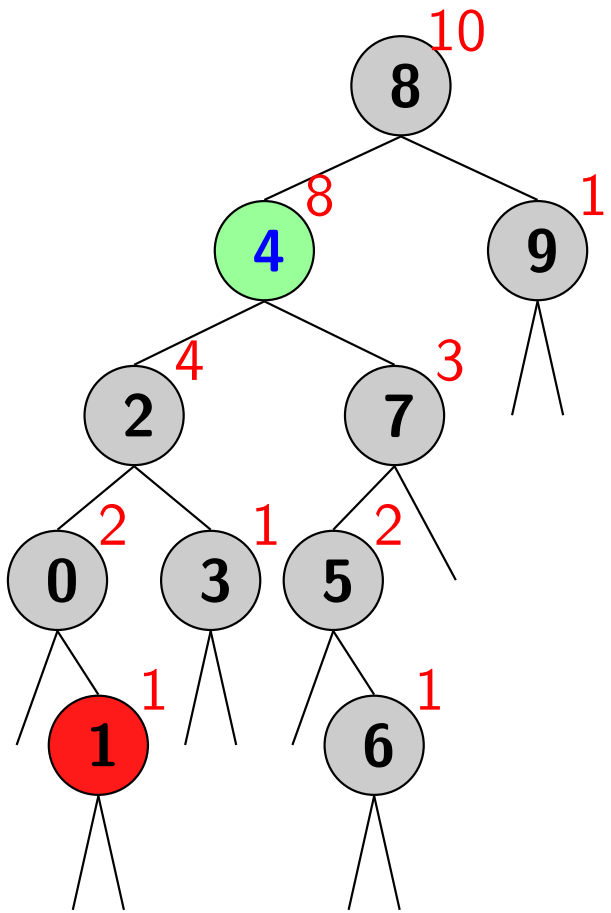
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



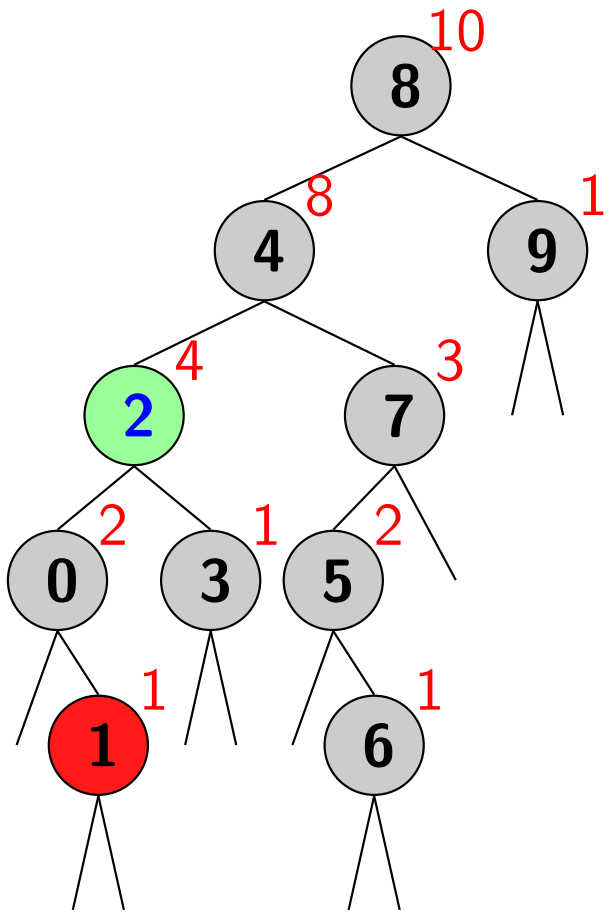
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



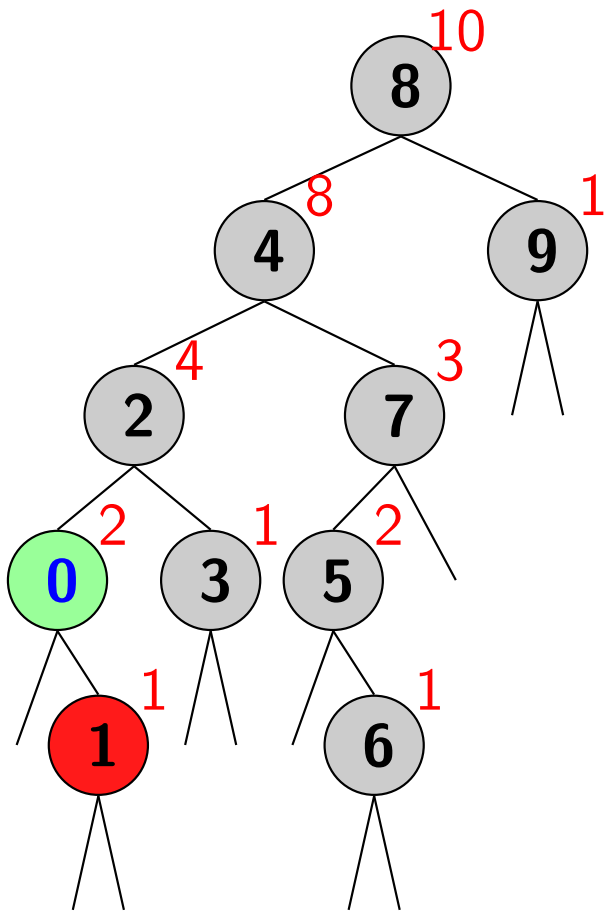
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



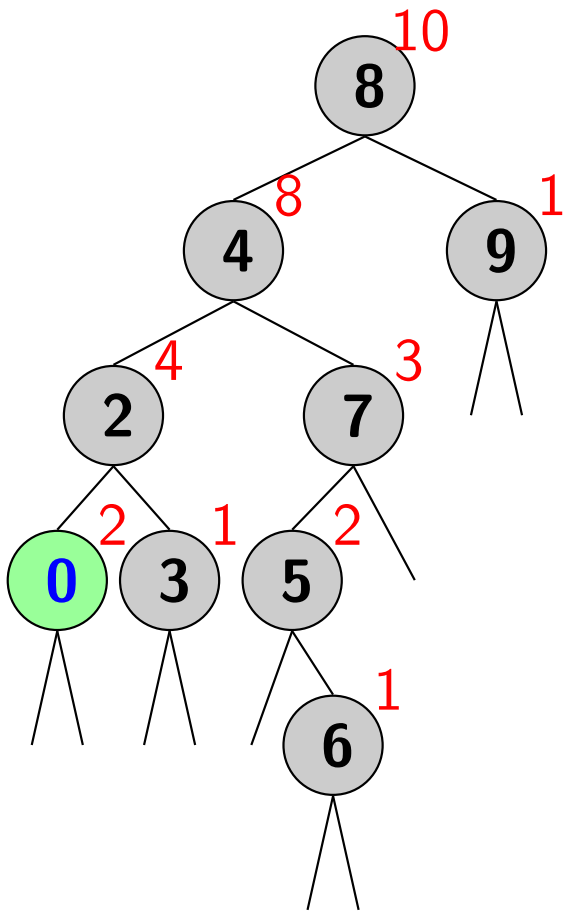
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



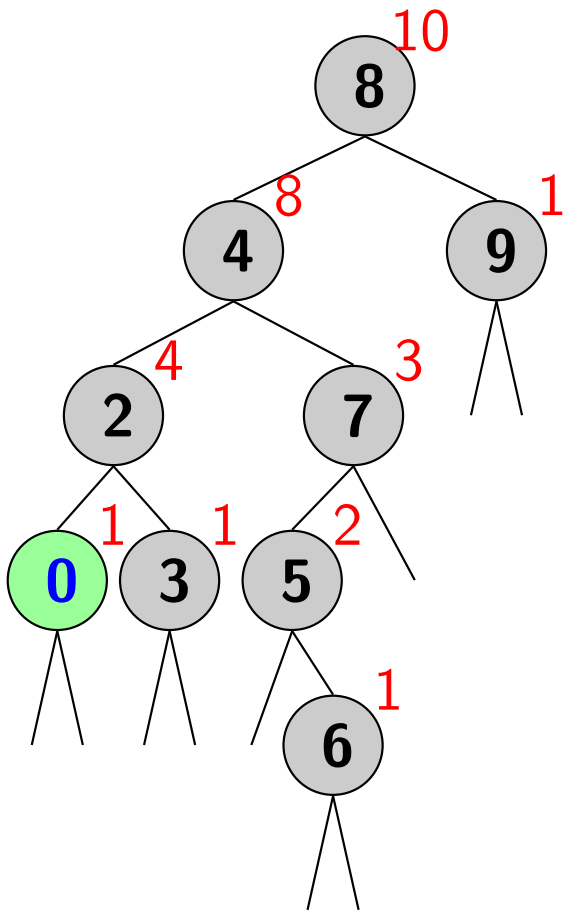
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



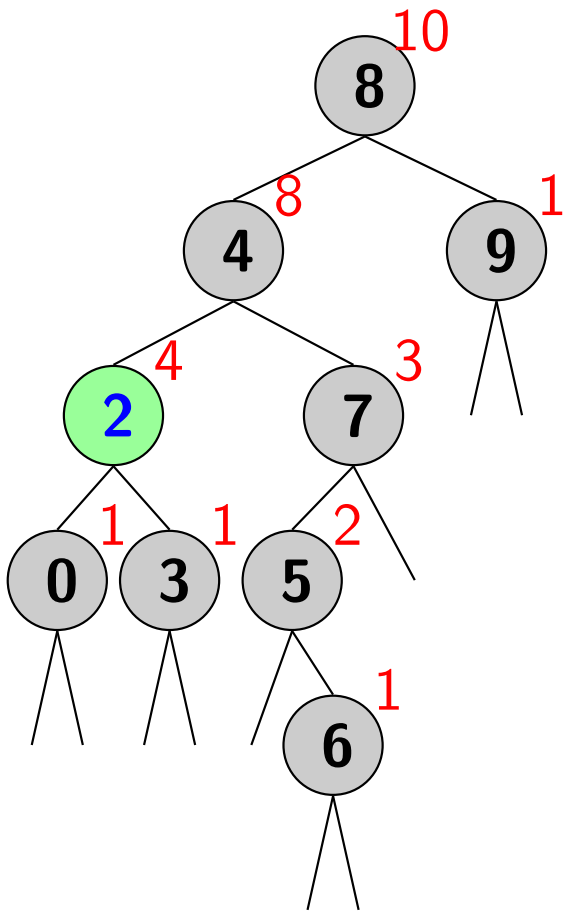
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



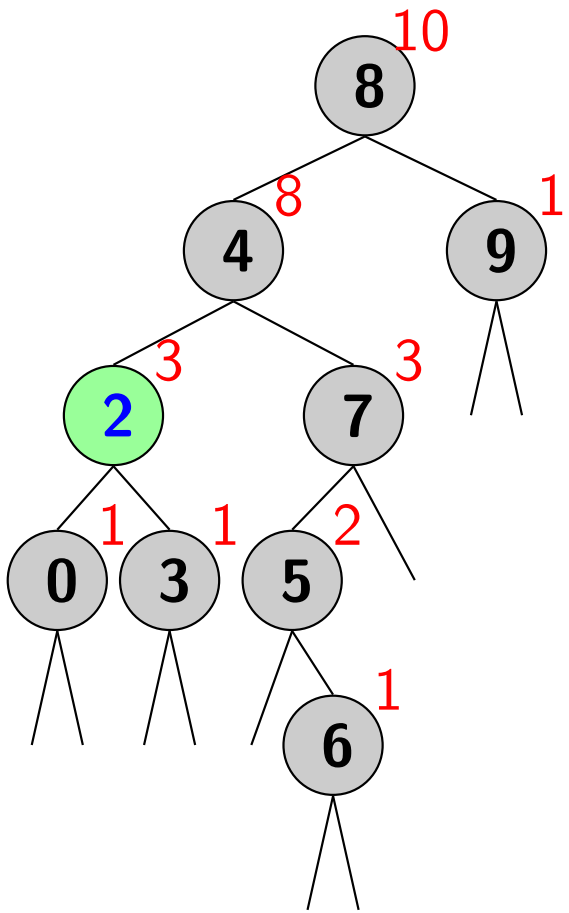
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



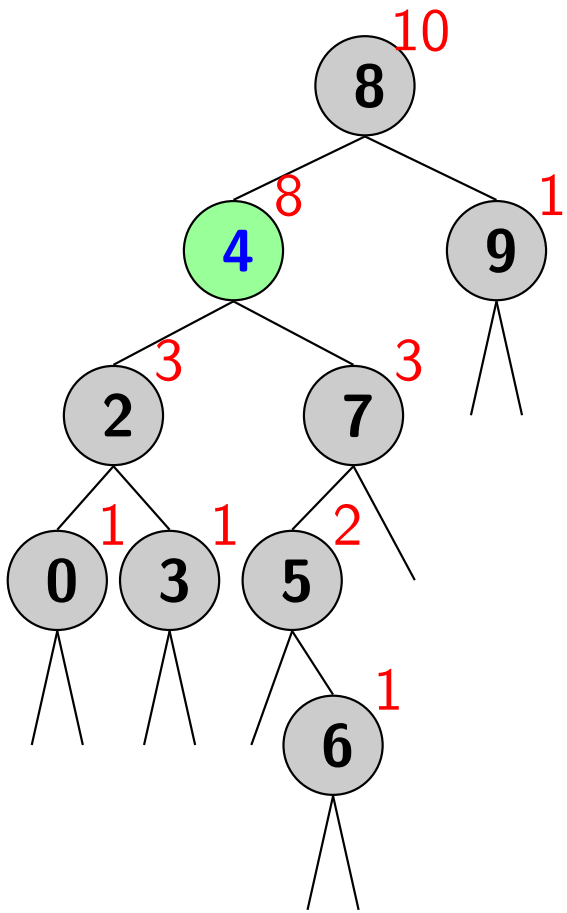
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



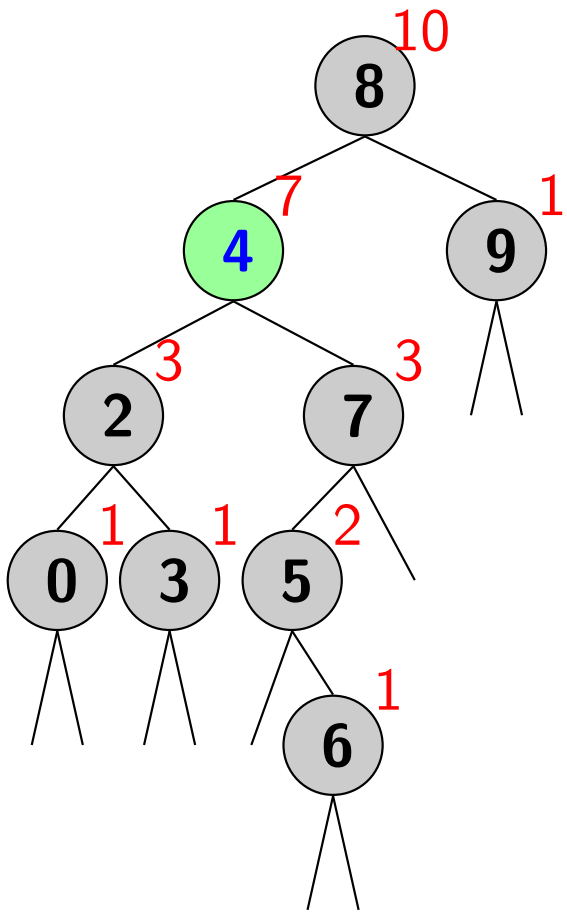
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



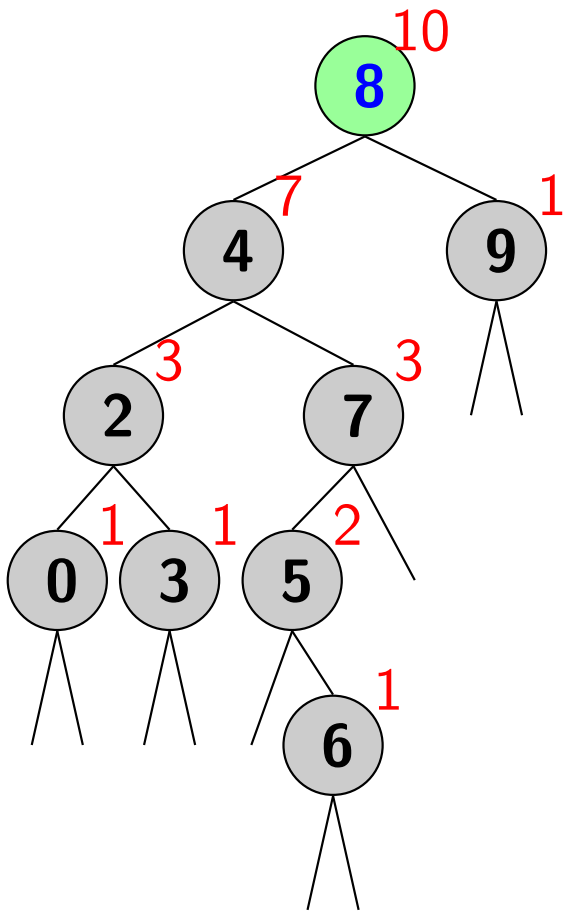
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



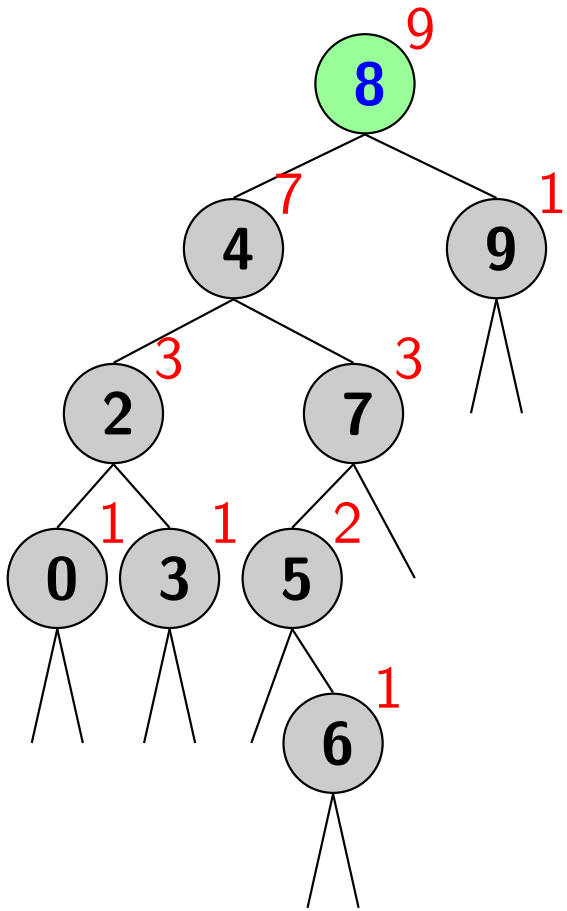
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



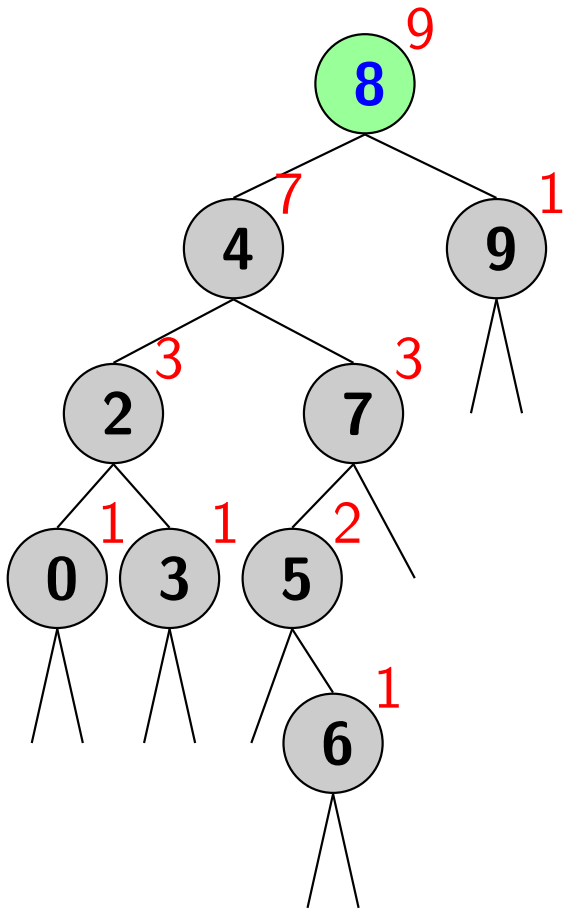
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 1



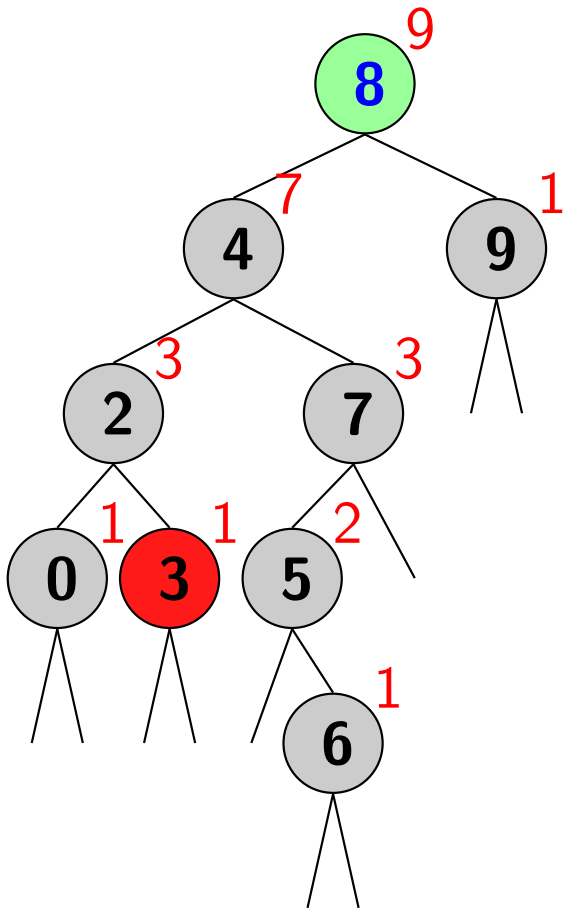
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



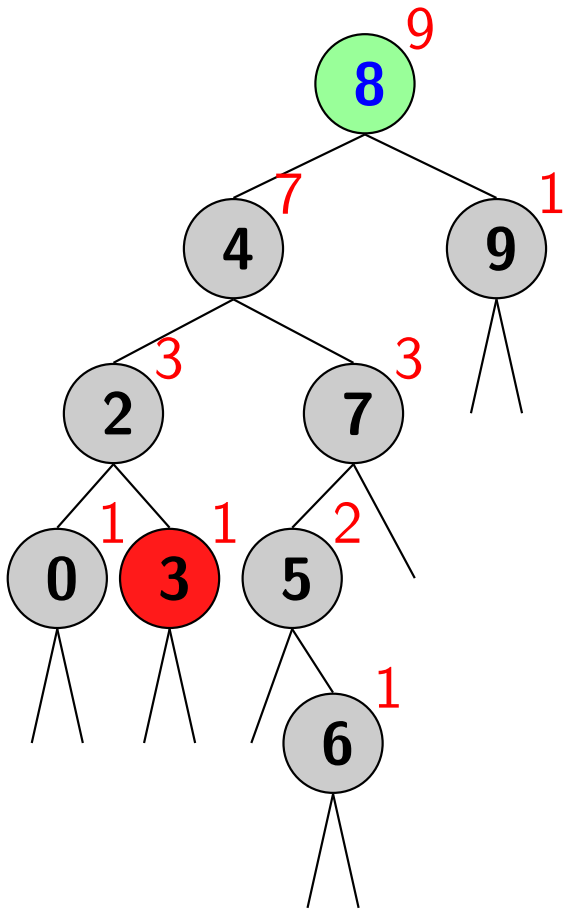
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



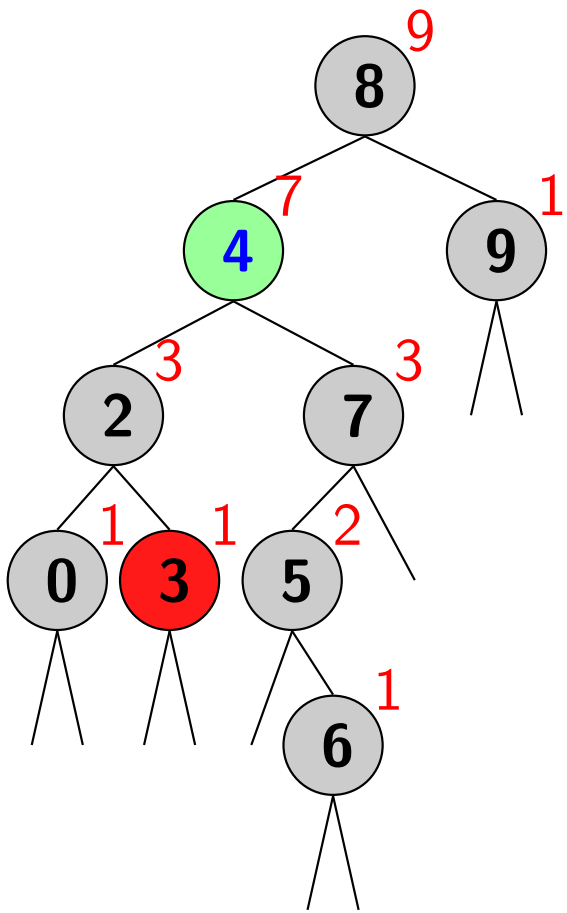
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



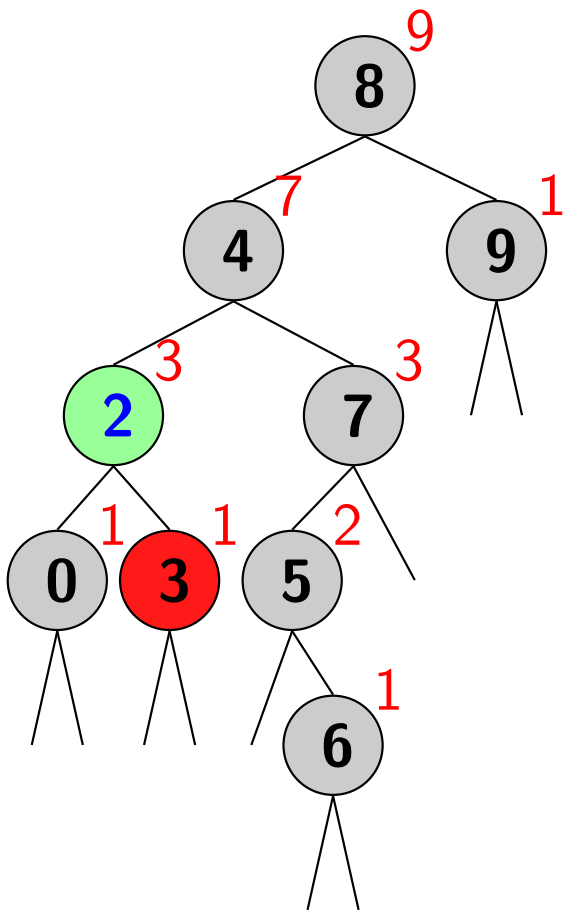
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



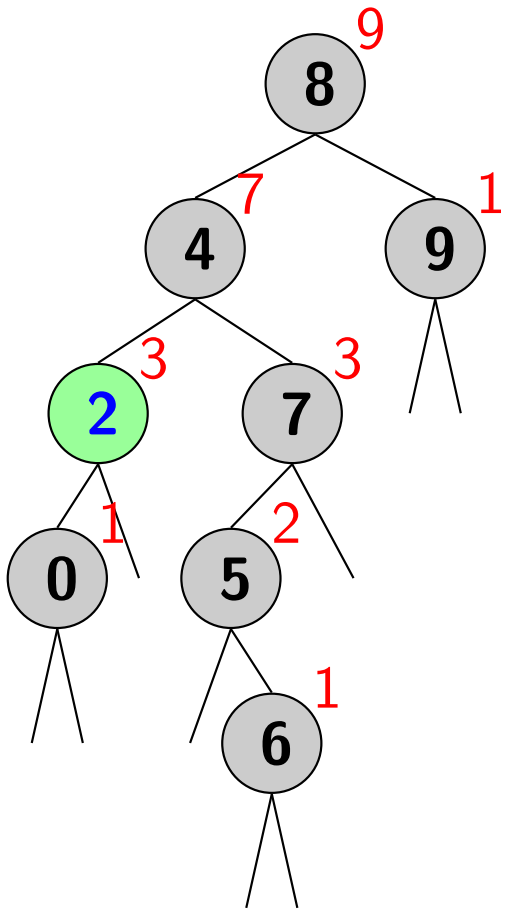
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



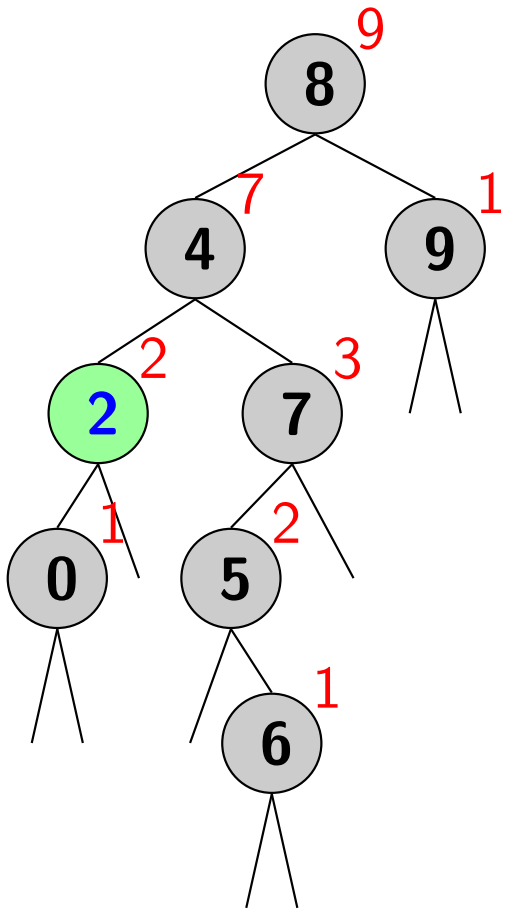
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



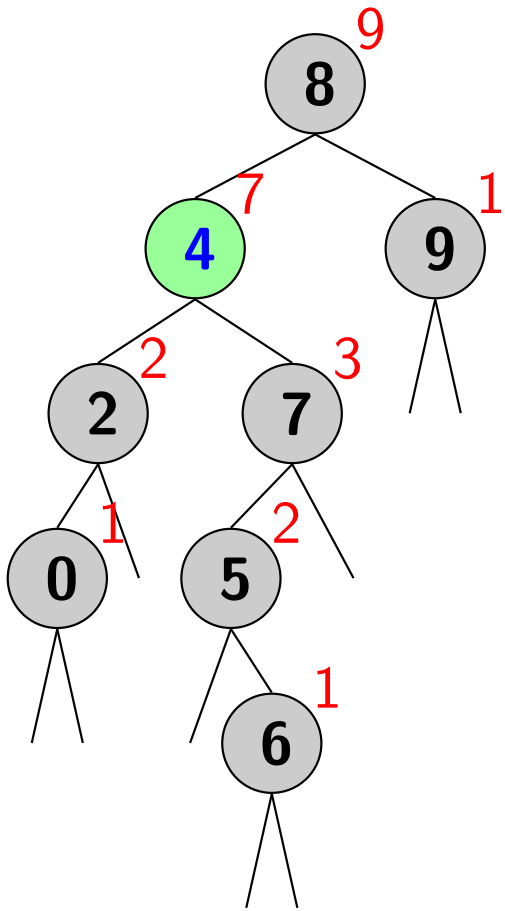
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



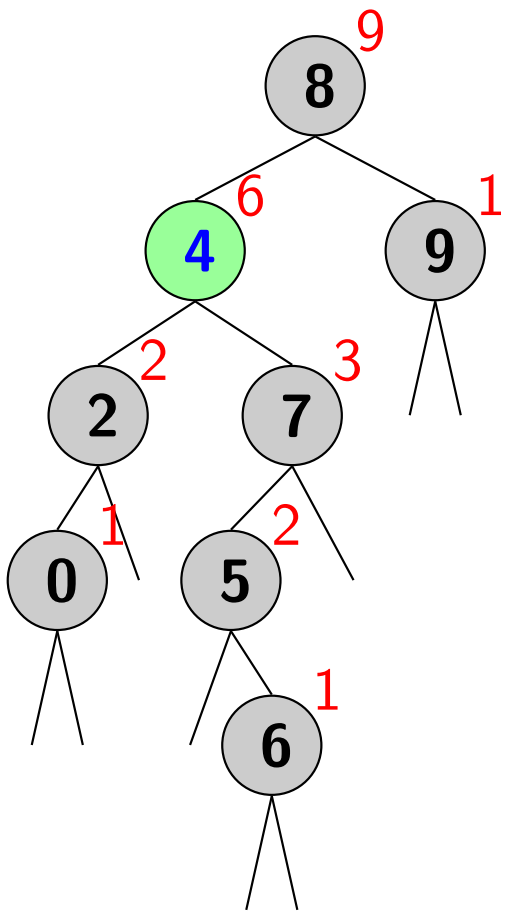
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



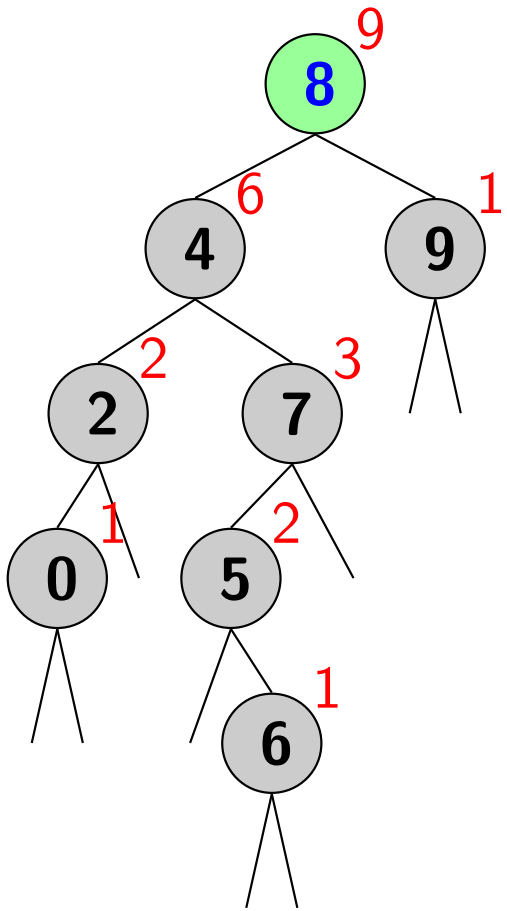
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



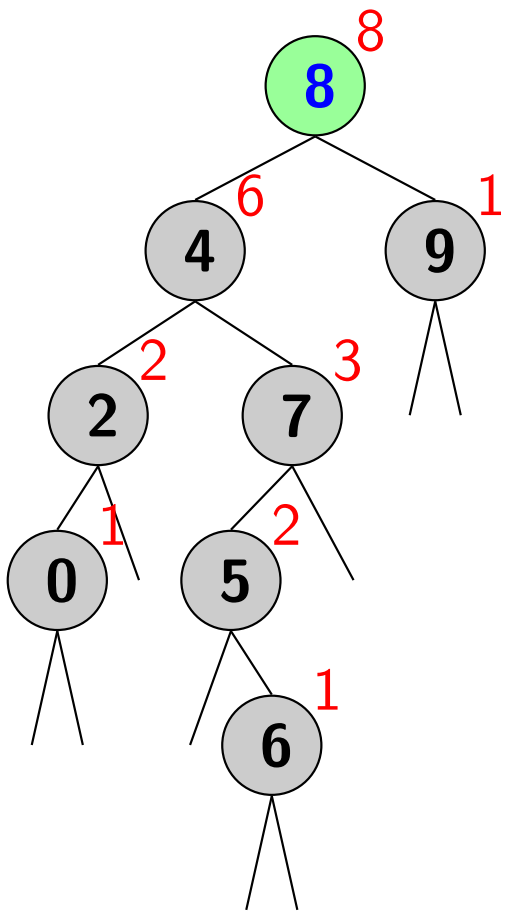
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



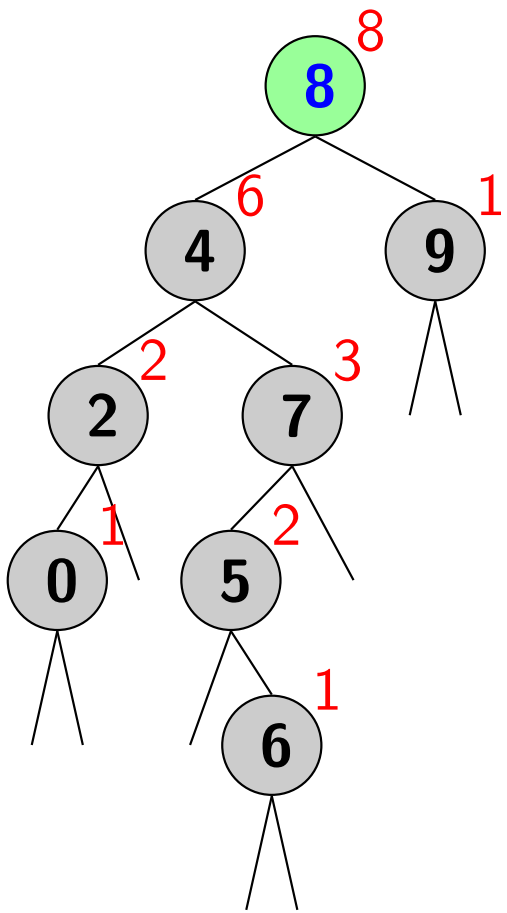
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 3



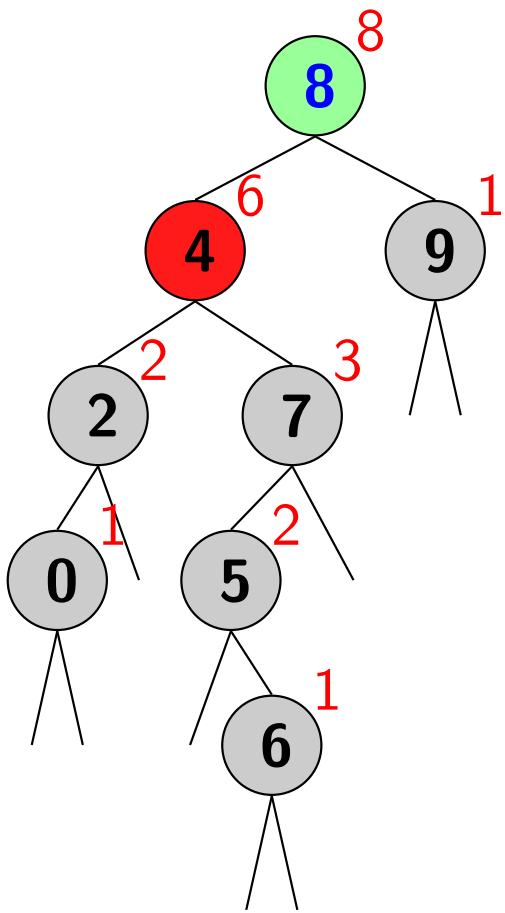
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 4



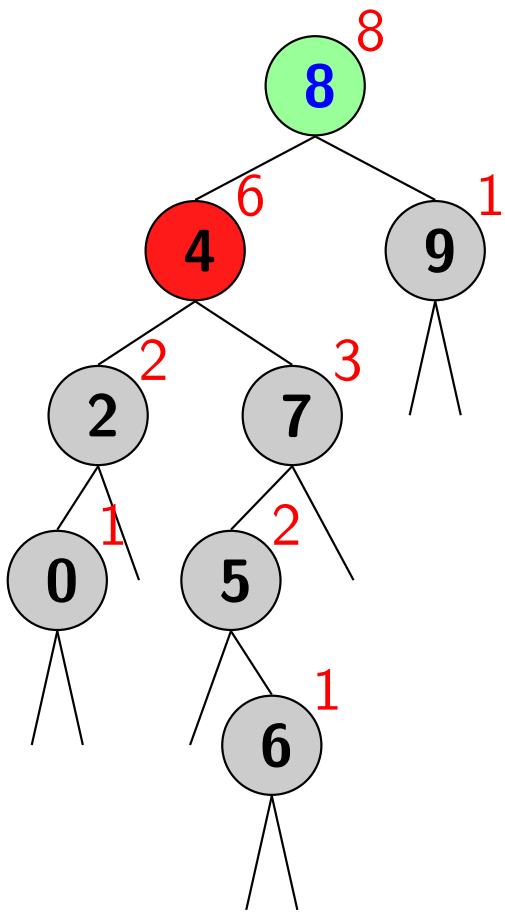
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 4



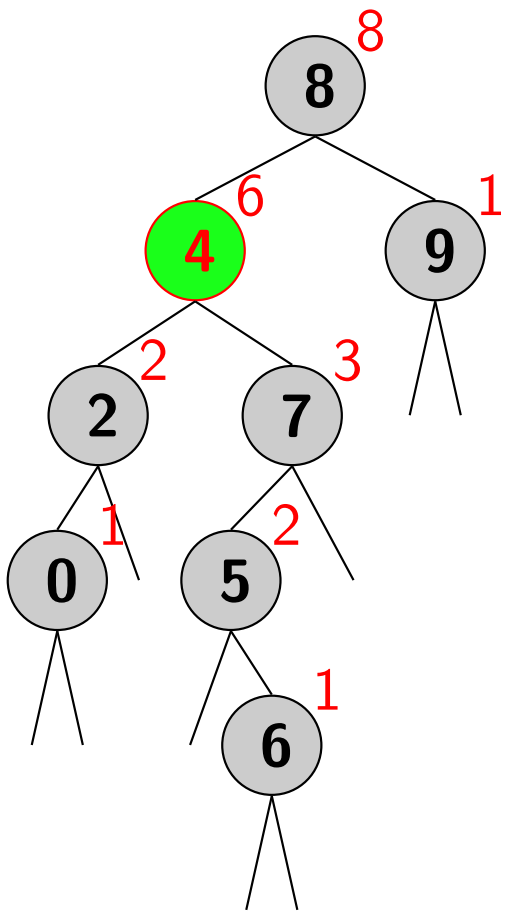
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 4



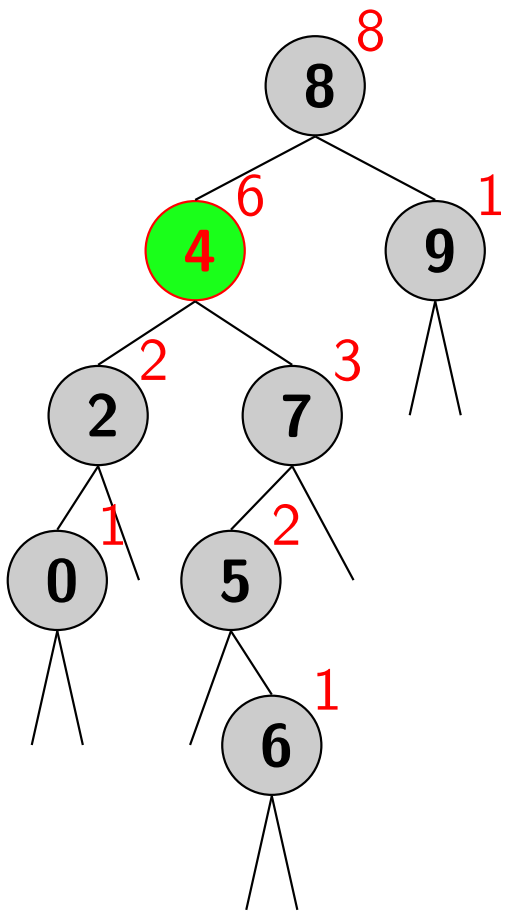
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 4



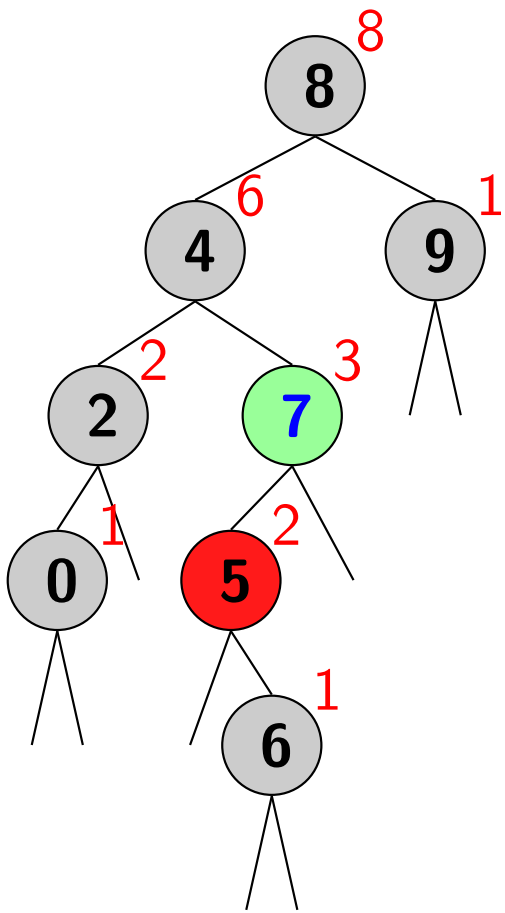
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo o sucessor mínimo: 5



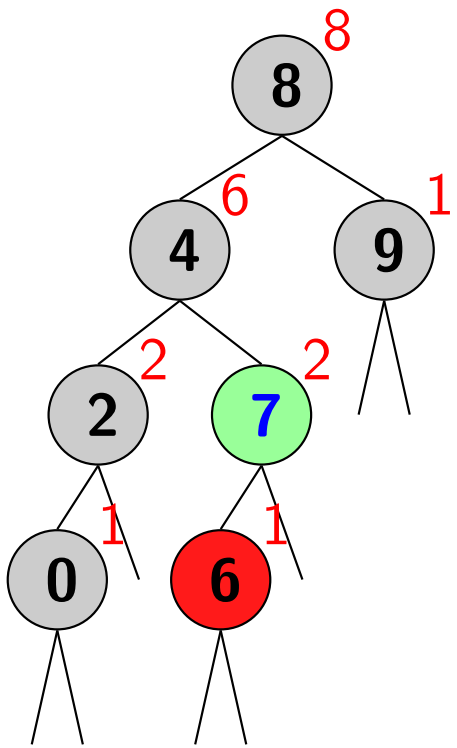
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removeu o mínimo: 5



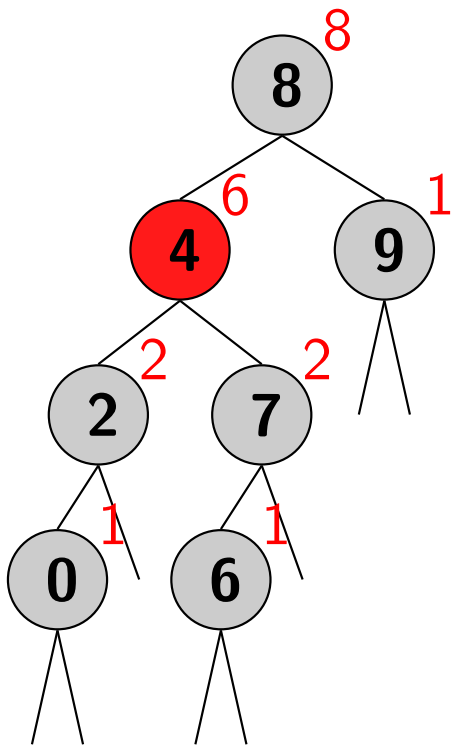
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Troca a chave 4 pelo sucessor: 5



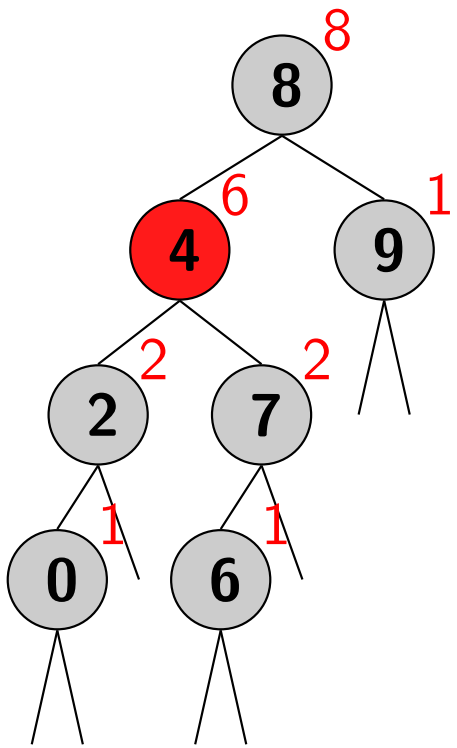
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Atualizando contagens



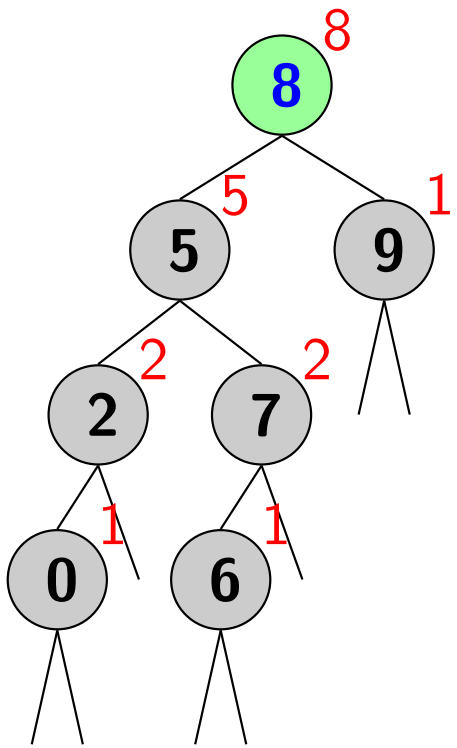
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Atualizando contagens



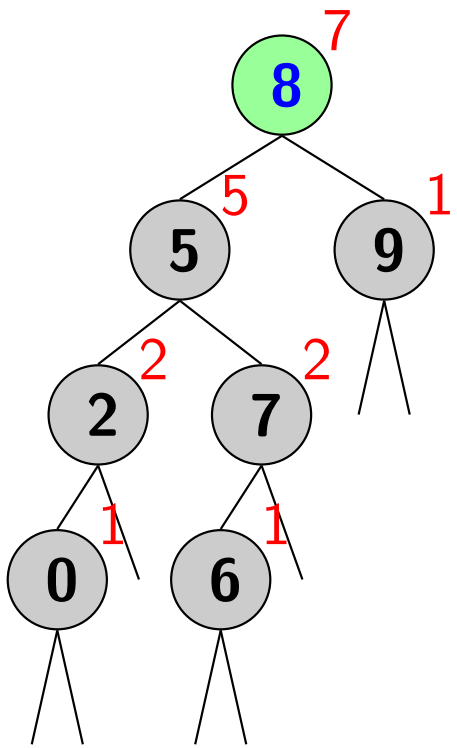
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Atualizando contagens



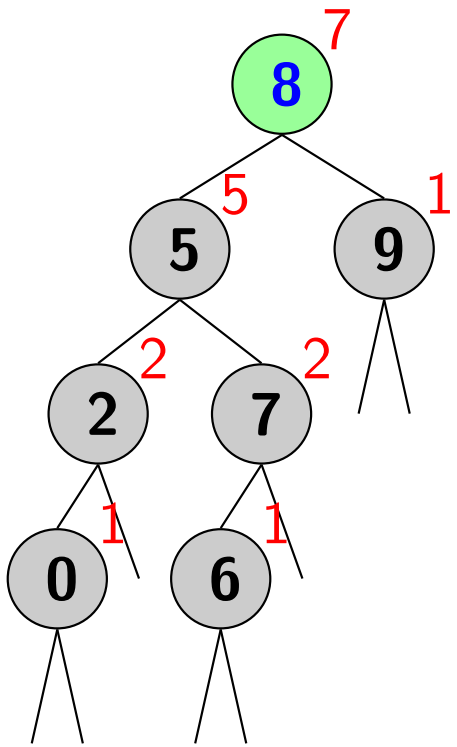
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 4



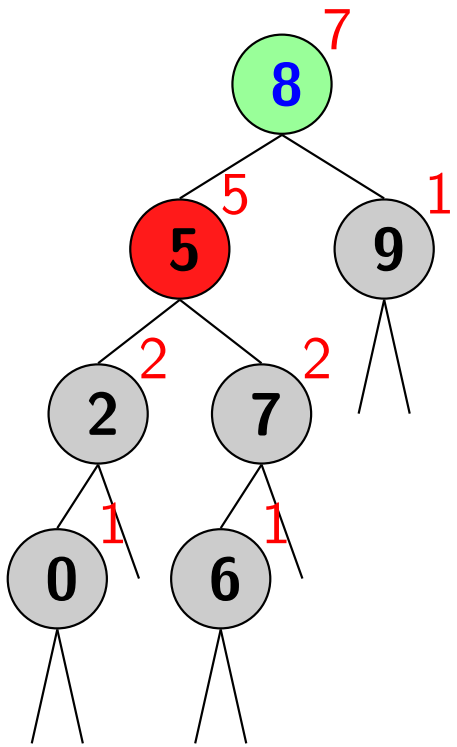
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



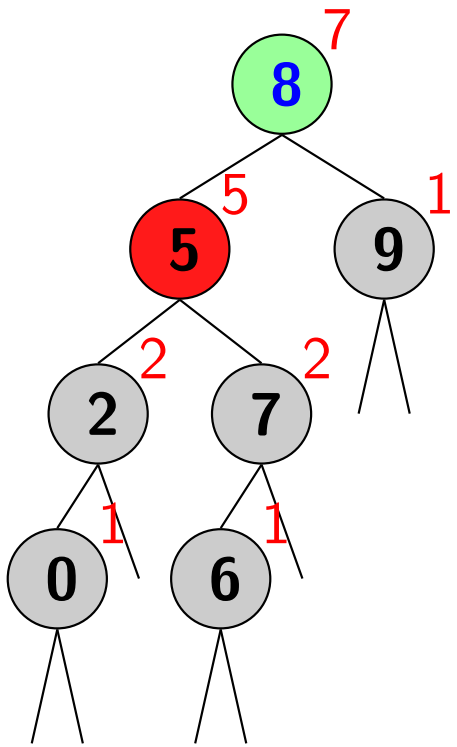
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



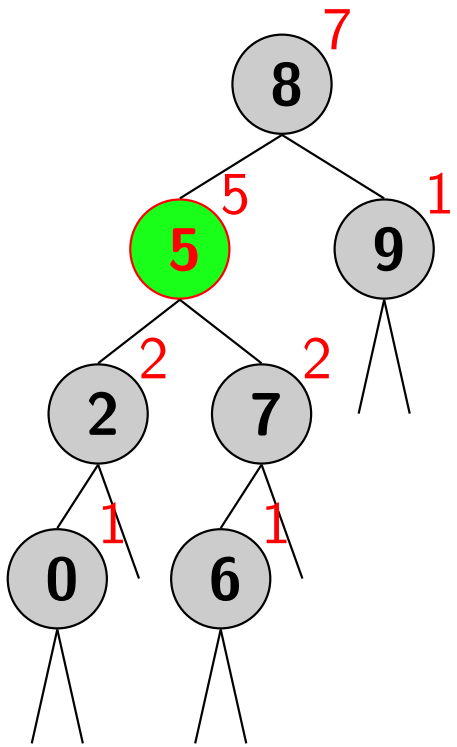
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



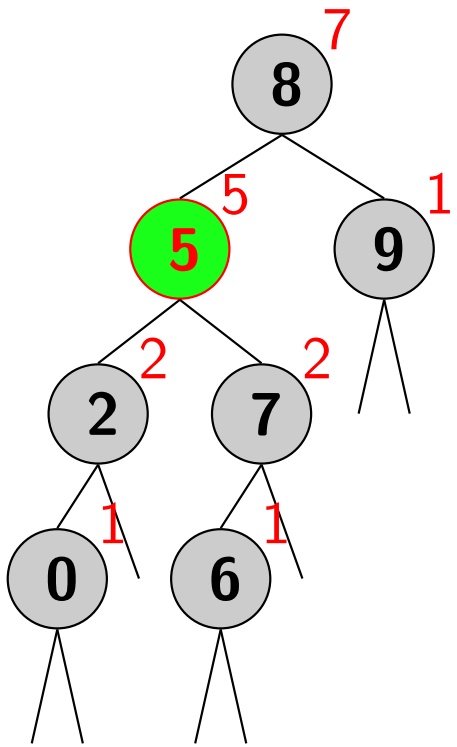
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



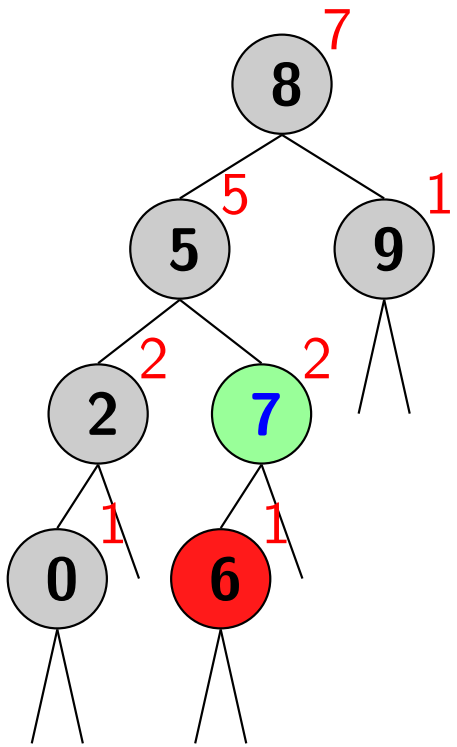
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo o mínimo: 6



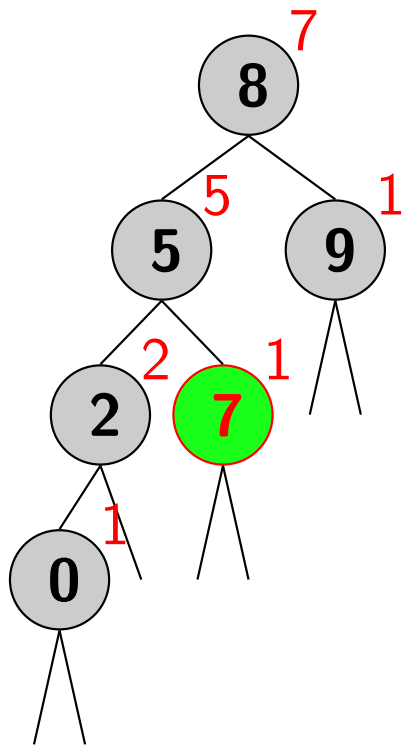
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removeu o sucessor mínimo: 6



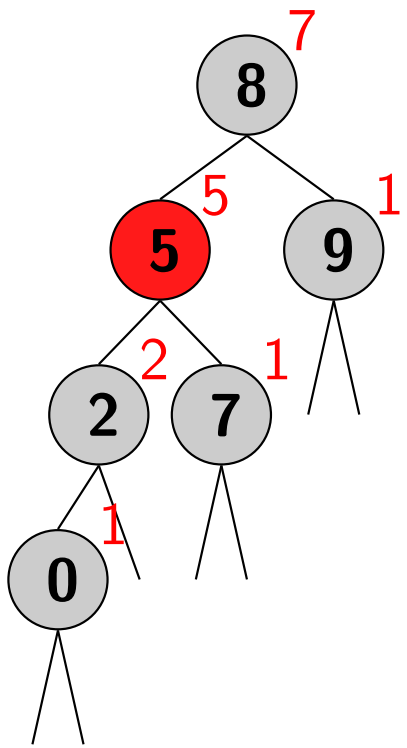
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Trocar a chave 5 pelo sucessor



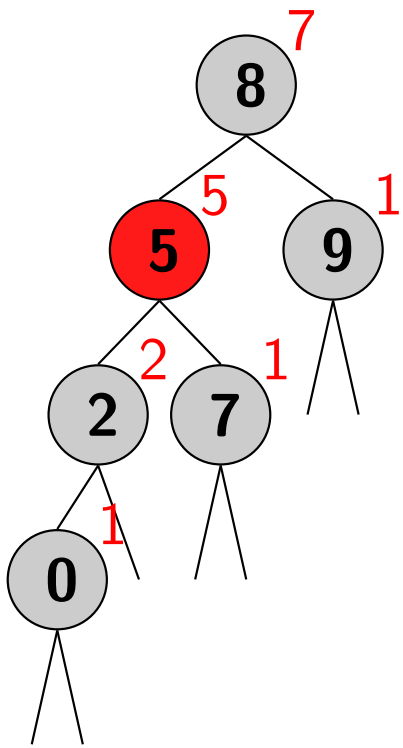
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Trocando a chave 5 por 6



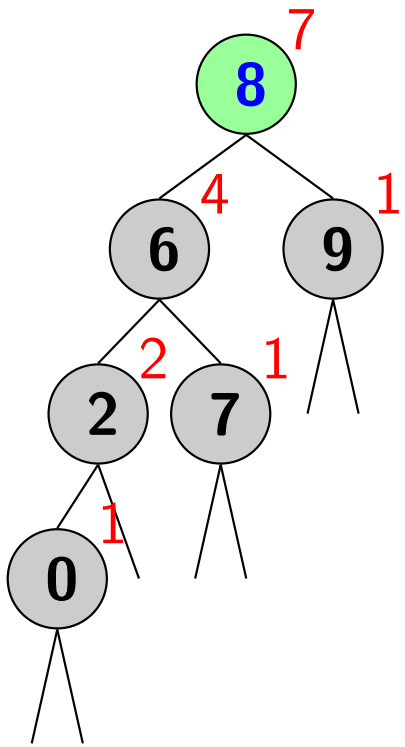
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



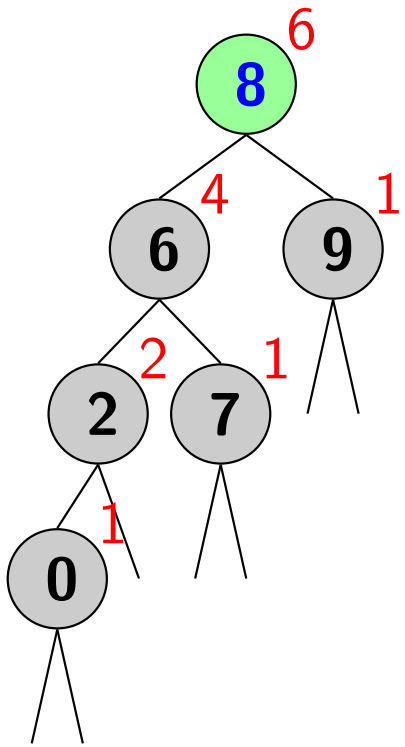
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



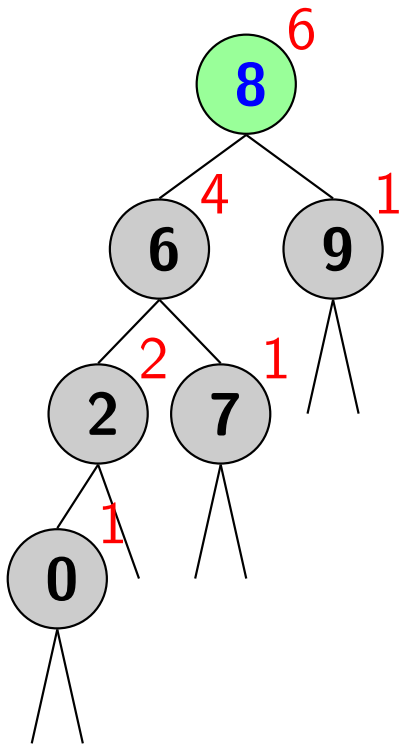
Remoção de Hibbard

`void delete(Chave c)`

Casos para remoção:

- Remoção de Node sem filhos
  - Pai de Node aponta para `null`
- Remoção de Node com 1 filho
  - Pai de Node aponta para filho de Node
- Remoção de Node com 2 filhos
  - Trocar com o menor sucessor e usar `deleteMin()`

Removendo a chave: 5



```java
public void delete(Chave c) {
 raiz = delete(raiz, c);
}
void delete(Node x, Chave c) {
 if (x == null) return null;

 int cmp = c.compareTo(x.chave);
 if (cmp < 0)
  x.esq = delete(x.esq, c);
 else if (cmp > 0)
  x.dir = delete(x.dir, c);
 else {
  if (x.dir==null) return x.esq;

  Node t = x;
  x = min(t.dir); // troca com sucessor mínimo
  x.dir = deleteMin(t.dir); // pode gerar desequilibrio
  x.esq = t.esq;
 }
 x.count = size(x.esq)// atualiza
          +size(x.dir)// contagem
          +1;
 return x;
}
```