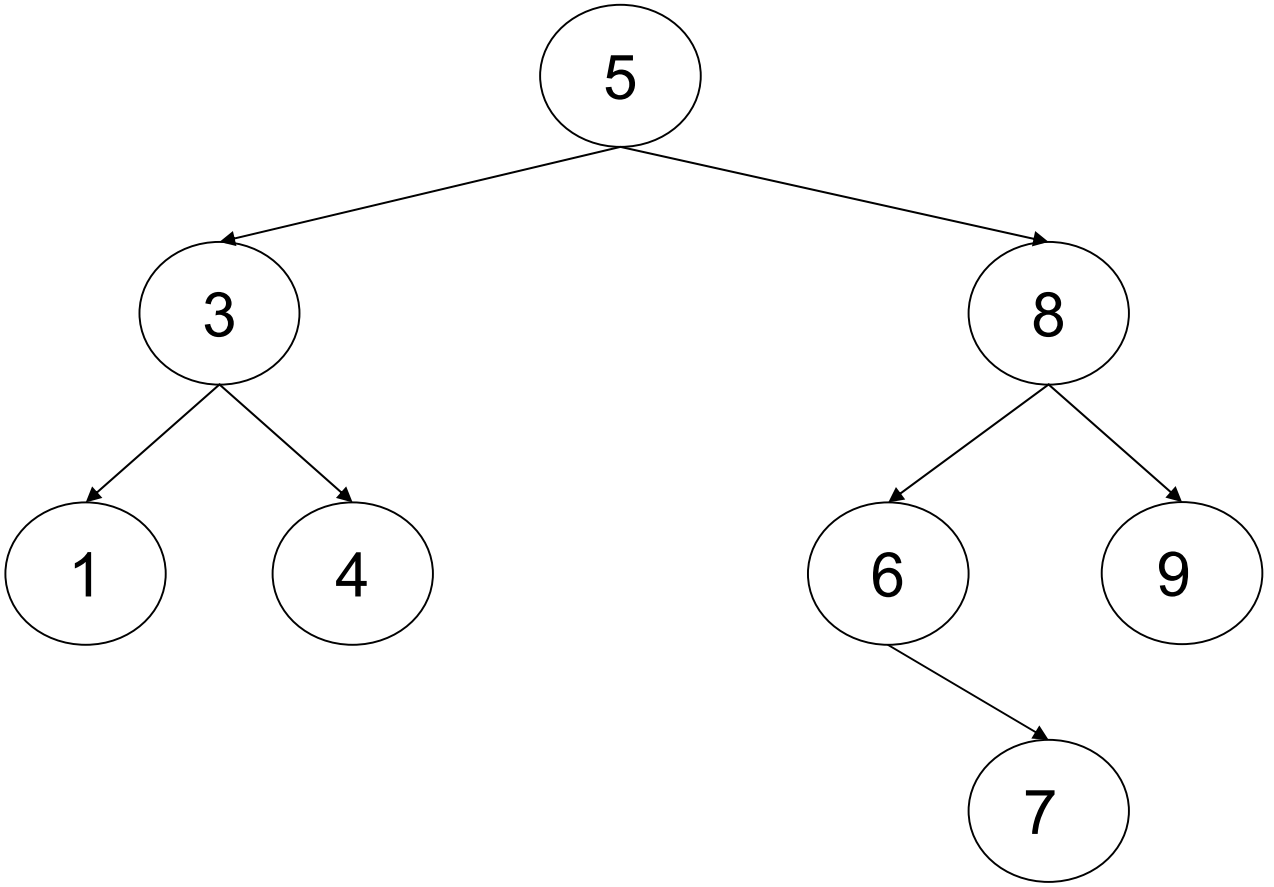
Árvores Binárias de Busca

Árvore Binária de Busca (ABB)

- Uma ABB é uma AB tal que para todo nó **x**:
  - Todos elementos chaves da subárvore esquerda de **x** são ***menores*** que a chave **x**
  - Todos elementos chaves da subárvore direita de **x** são ***maiores*** que a chave **x**
  - As subárvores esquerda e direita também são ABB
- ABB são também conhecidas como dicionários binários
- Um campo chave (*key*) é um dado de identifica univocamente uma informação
  - RG, CPF, fragmento de DNA (tag), *gene symbol*

Árvore Binária de Busca (ABB)



Especificação

- Operações:
  - Criação
  - Destruição
  - Status
  - Operações Básicas

Criação

BinarySearchTree::BinarySearchTree();

- *pré-condição:* nenhuma
- *pós-condição:* Árvore binária é criada e iniciada como vazia

Destruição

BinarySearchTree::~BinarySearchTree();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* Árvore binária é destruída, liberando espaço ocupado pelo seus elementos

Status

bool BinarySearchTree::Empty();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* função retorna **true** se a árvore binária está vazia; **false** caso contrário

Status

bool BinarySearchTree::Full();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* função retorna **true** se a árvore binária está cheia; **false** caso contrário

Operações Básicas

void BinarySearchTree::Insert(TreeEntry x);

- *pré-condição:* Árvore binária já tenha sido criada, não está cheia
- *pós-condição:* O item **x** é armazenado na árvore binária

O tipo **TreeEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos

Operações Básicas

void BinarySearchTree::Delete(TreeEntry x);

- *pré-condição:* Árvore binária já tenha sido criada, não está vazia e o item **x** pertence à árvore binária
- *pós-condição:* O item **x** é removido da árvore binária

Operações Básicas

bool BinarySearchTree::Search(TreeEntry x);

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* Retorna **true** se **x** foi encontrado na árvore binária; **false** caso contrário

Operações Básicas

TreeEntry BinarySearchTree::Minimum()

- *pré-condição:* Árvore binária já tenha sido criada e não está vazia
- *pós-condição:* Retorna o valor mínimo encontrado na árvore binária

Operações Básicas

TreeEntry BinarySearchTree::Maximum()

- *pré-condição:* Árvore binária já tenha sido criada e não está vazia
- *pós-condição:* Retorna o valor máximo encontrado na árvore binária

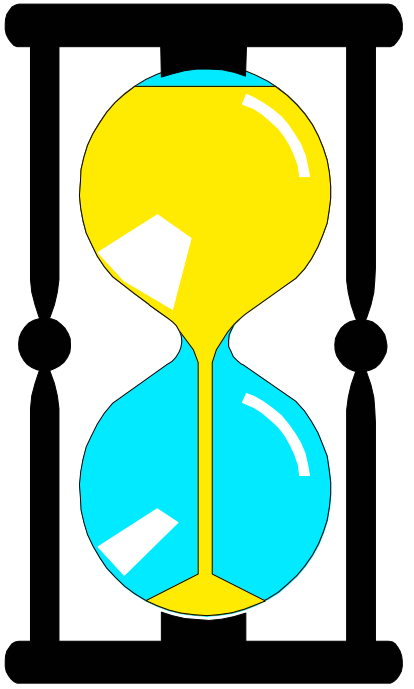
Outras Operações

void BinarySearchTree::Clear();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* todos os itens da árvore binária são descartados e ela torna-se uma árvore binária vazia

Questão

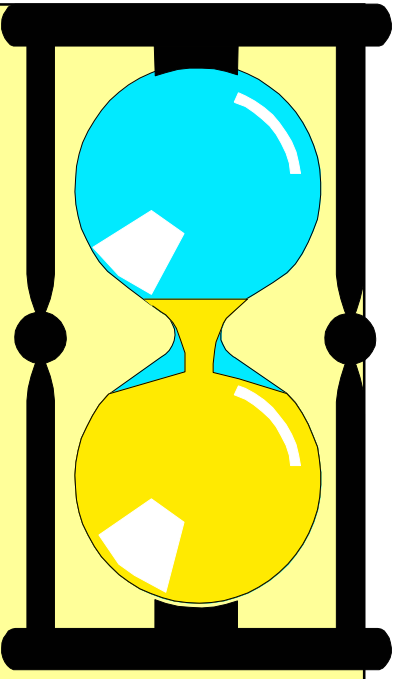
Utilize estas idéias para escrever uma declaração de tipo que poderia implementar uma árvore binária para armazenar valores inteiros.



Você tem 5 minutos
para escrever a declaração

Uma Solução

```
class BinarySearchTree
{ public:
    BinarySearchTree();
    ~BinarySearchTree();
    void Insert(int x);
    void Delete(int x);
    bool Search(int x);
    …
  private:
    // declaração de tipos
    struct TreeNode
    { int Entry;            // tipo de dado colocado na árvore
      TreeNode *LeftNode, *RightNode; // subárvores
    };
    typedef TreeNode *TreePointer;

    // declaração de campos
    TreePointer root;
};
```



Uma Solução

```
class BinarySearchTree
{ public:
    BinarySearchTree();
    ~BinarySearchTree();
    void Insert(int x);
    void Delete(int x);
    bool Search(int x);
    …
  private:
    // declaração de tipos
    struct TreeNode
    { int Entry;            // tipo de dado colocado na árvore
      TreeNode *LeftNode, *RightNode; // subárvores
    };
    typedef TreeNode *TreePointer;

    // declaração de campos
    TreePointer root;
};
```

Observe que o tipo **TreeEntry** nesse caso é um inteiro

Construtor

BinarySearchTree::BinarySearchTree()

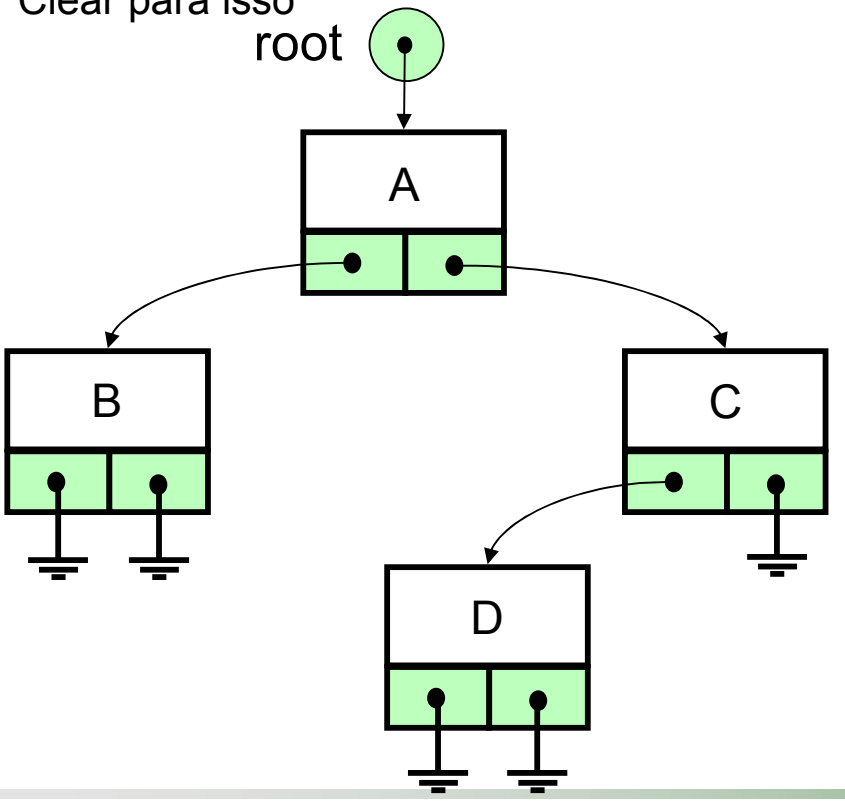
A ABB deve iniciar vazia, cuja convenção é a raiz estar aterrada


```
BinarySearchTree::BinarySearchTree()
{
    root = NULL;
}
```

Destruidor

BinarySearchTree::~BinarySearchTree()

O destruidor deve retirar todos os elementos da ABB. Faremos uma chamada ao método Clear para isso




```
BinarySearchTree::~BinarySearchTree()
{
  Clear();
}
```

Status: Empty

bool BinarySearchTree::Empty()

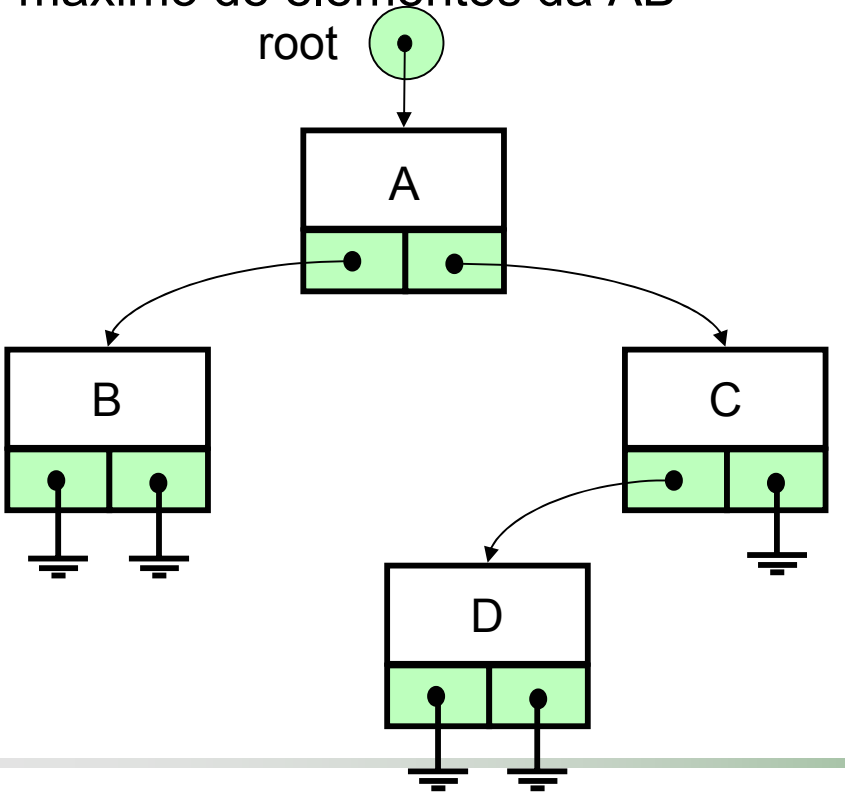
Lembre-se que a ABB inicia vazia, com **root** = **NULL**...


```
bool BinarySearchTree::Empty()
{
    return (root == NULL);
}
```

Status: Full

bool BinarySearchTree::Full()

...e que não há limite quanto ao número máximo de elementos da AB

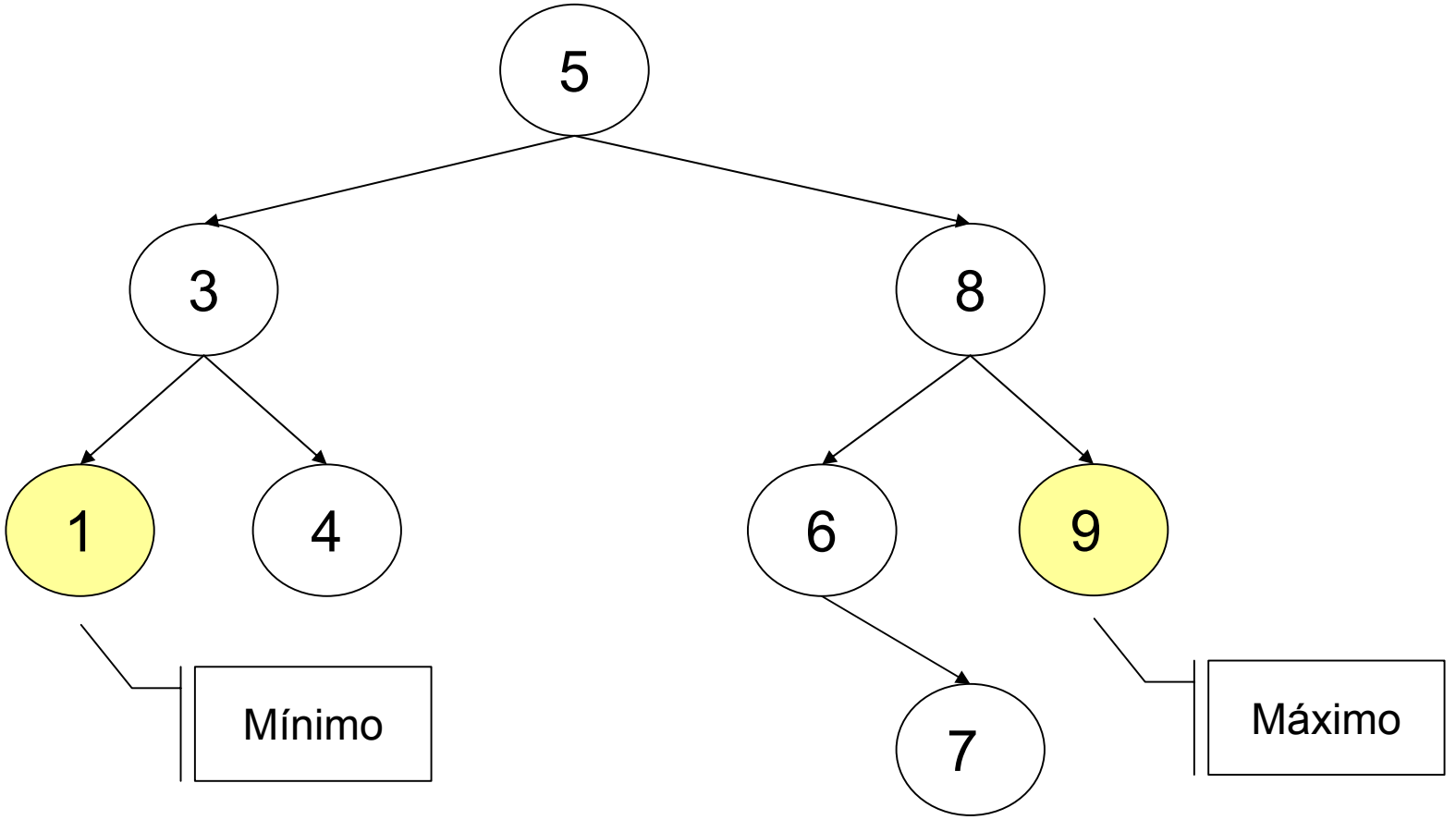



```
bool BinarySearchTree::Full()
{
   return false;
}
```

Mínimo e Máximo

- O elemento mínimo em uma ABB pode ser encontrado seguindo-se as subárvores esquerdas desde a raiz
- O elemento máximo em uma ABB pode ser encontrado seguindo-se as subárvores direitas desde a raiz

Mínimo e Máximo



Operações Básicas: Mínimo

```
int BinarySearchTree::Minimum()
{ TreePointer t=root;

  if(t == NULL)
  { cout << “Árvore vazia” << endl;
    abort();
  }
  while (t->LeftNode != NULL)
      t = t->LeftNode; // procurar subárvore esquerda

  return t->Entry;
}
```

Operações Básicas: Máximo

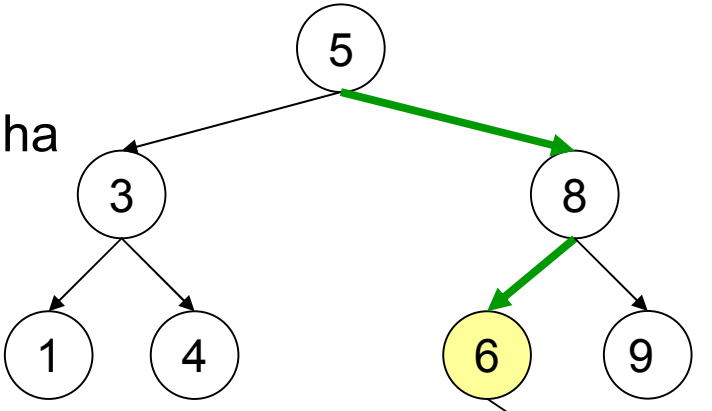
```
int BinarySearchTree::Maximum()
{ TreePointer t=root;

  if(t == NULL)
  { cout << “Árvore vazia” << endl;
    abort();
  }
  while (t->RightNode != NULL)
      t = t->RightNode; // procurar subárvore direita

  return t->Entry;
}
```

Busca em ABB

- Como já visto, **n** elementos podem ser organizados em uma árvore de altura mínima **h** ≈ $\log_2$ **n**
- Portanto, uma busca entre **n** elementos pode ser realizada com apenas O(**h**)=O($\log_2$ **n**) comparações se a árvore estiver perfeitamente balanceada

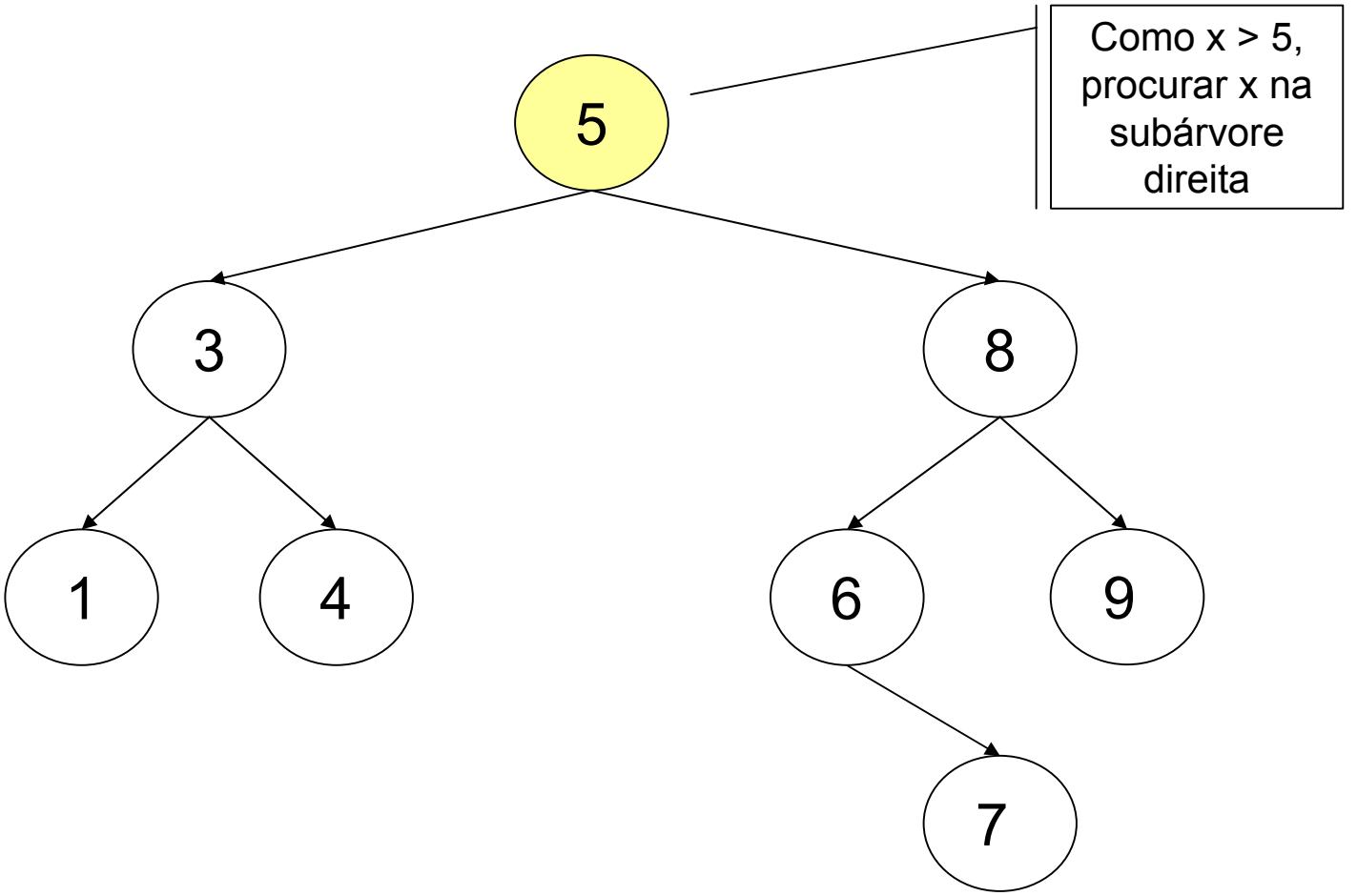
Busca em ABB

- A vantagem da ABB é que sempre é suficiente realizar a busca em, no máximo, uma das subárvores
- Para encontrar um elemento **x** na ABB
  - Se **x** é a raiz da ABB então **x** foi encontrado, caso contrário
  - Se **x** é menor que a raiz então procure **x** na sub-árvore esquerda, caso contrário
  - Procure **x** na sub-árvore direita de
  - Se a ABB é vazia então a busca falha

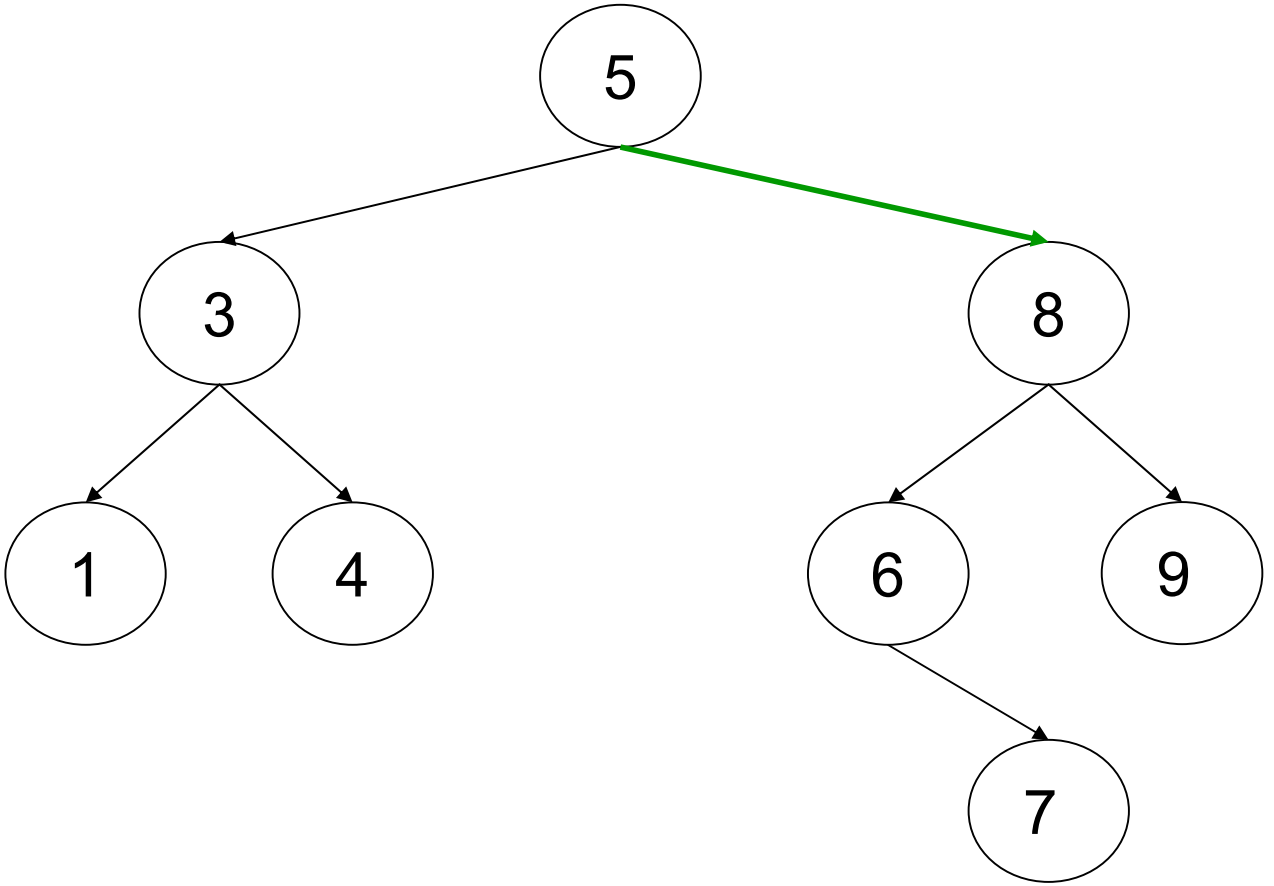


Elemento 6 é encontrado seguindo o caminho 5-8-6

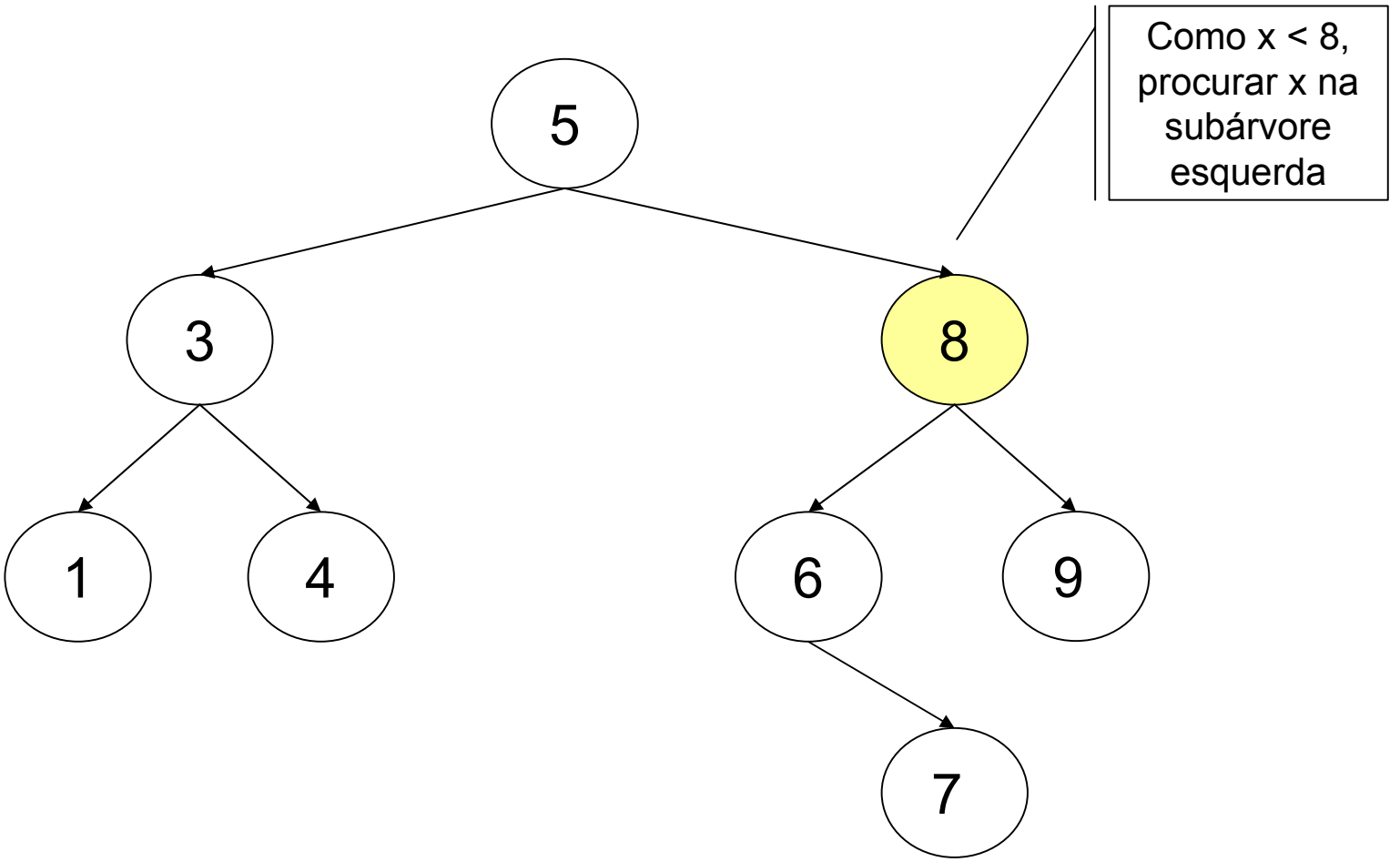
Busca de x = 6



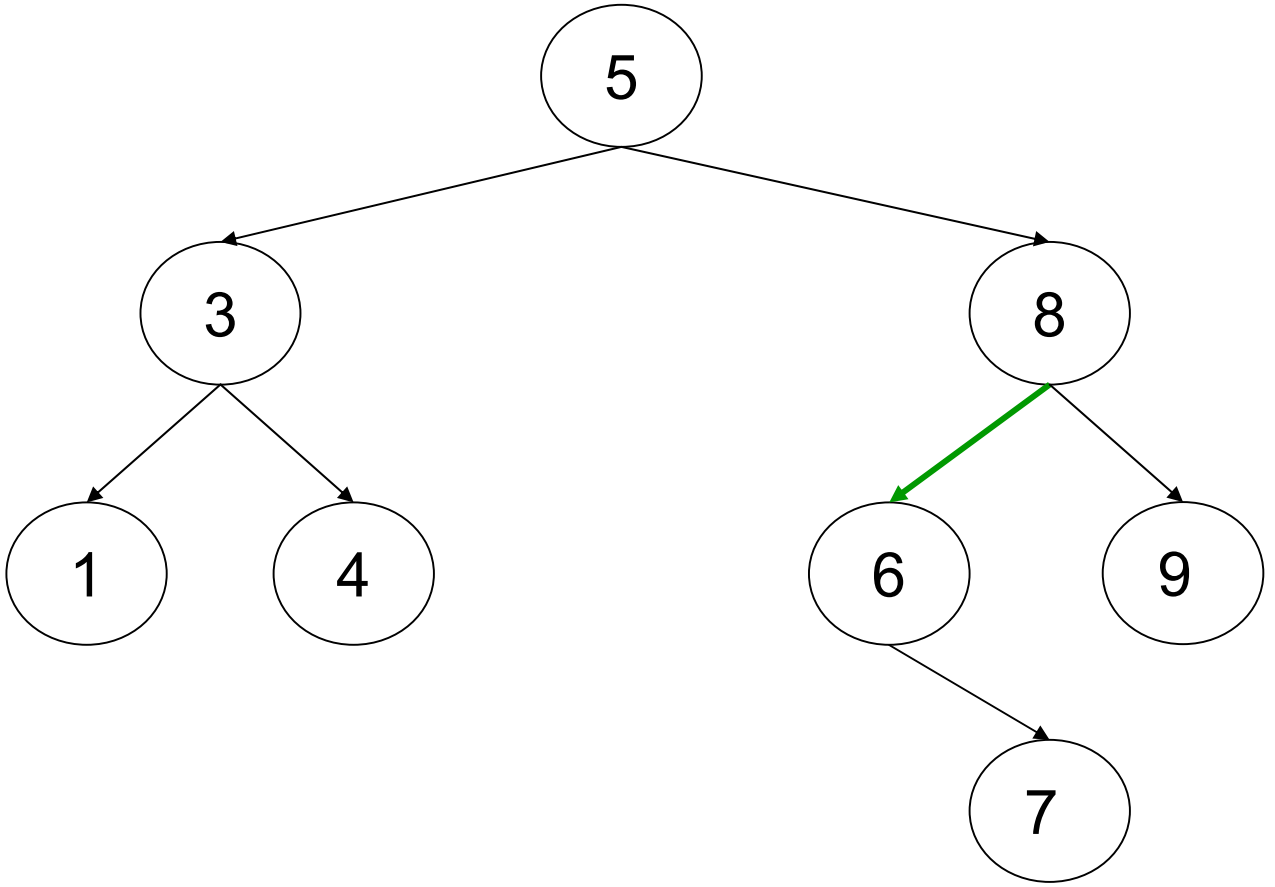
Busca de x = 6



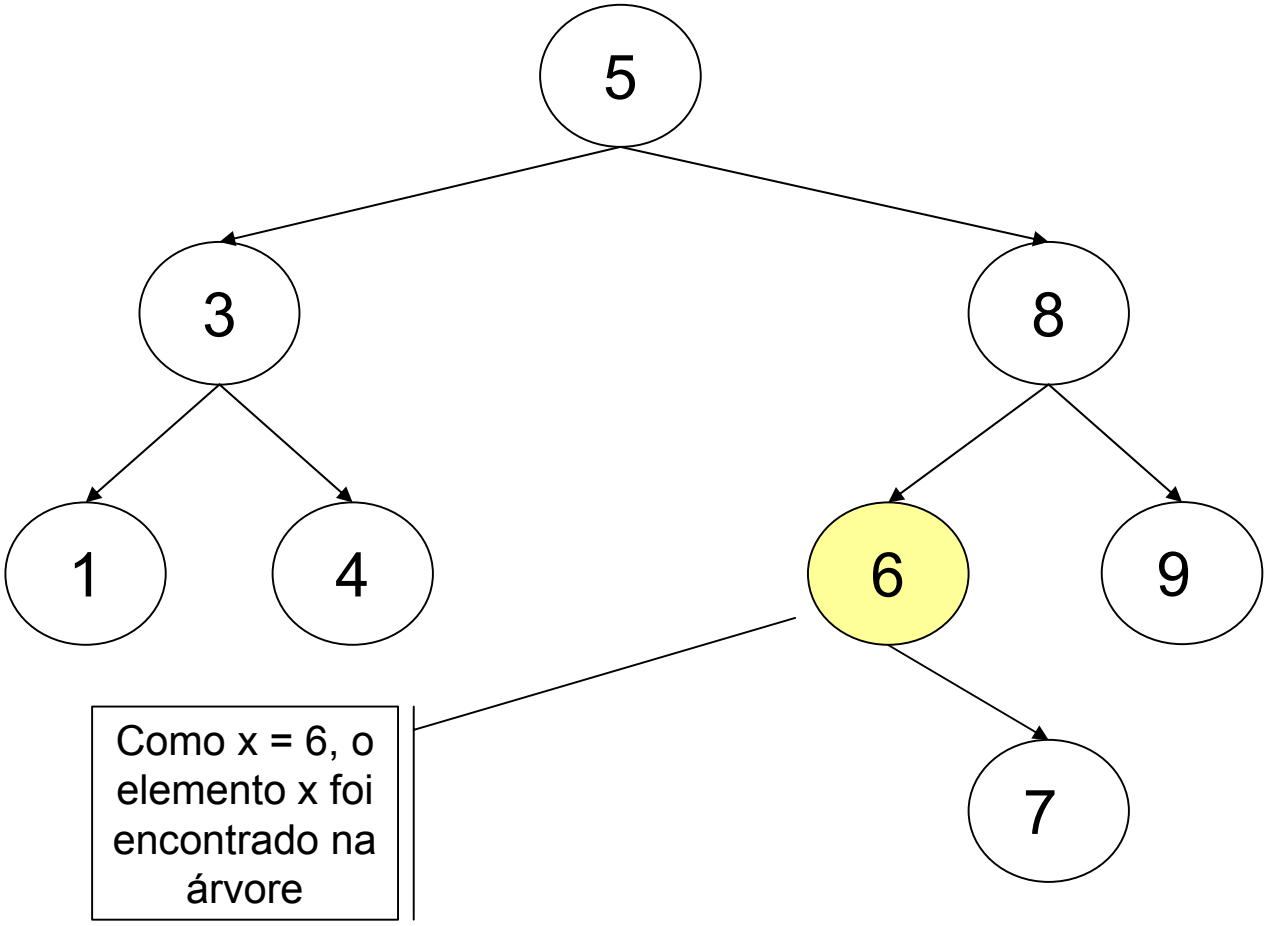
Busca de x = 6



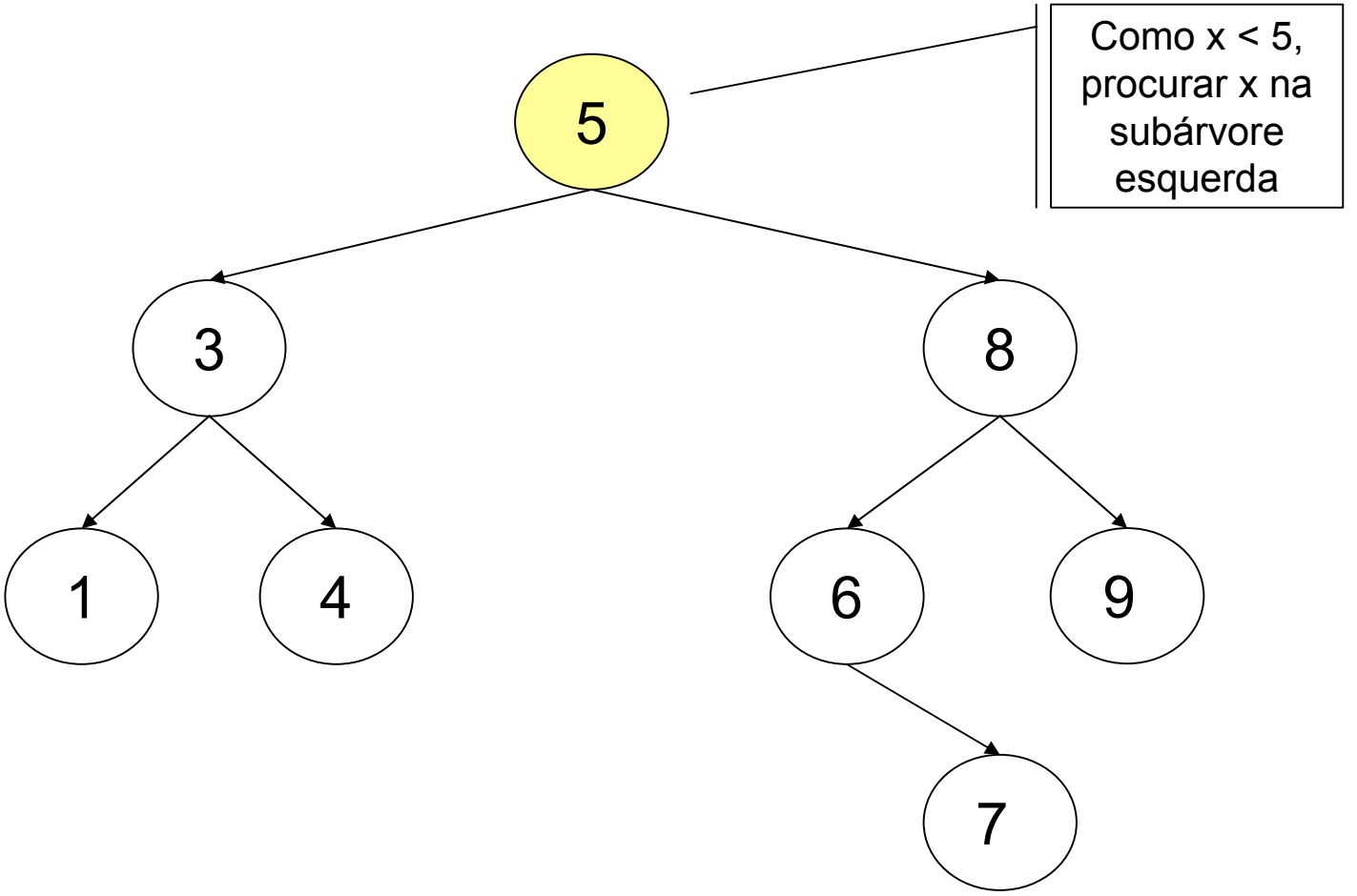
Busca de x = 6



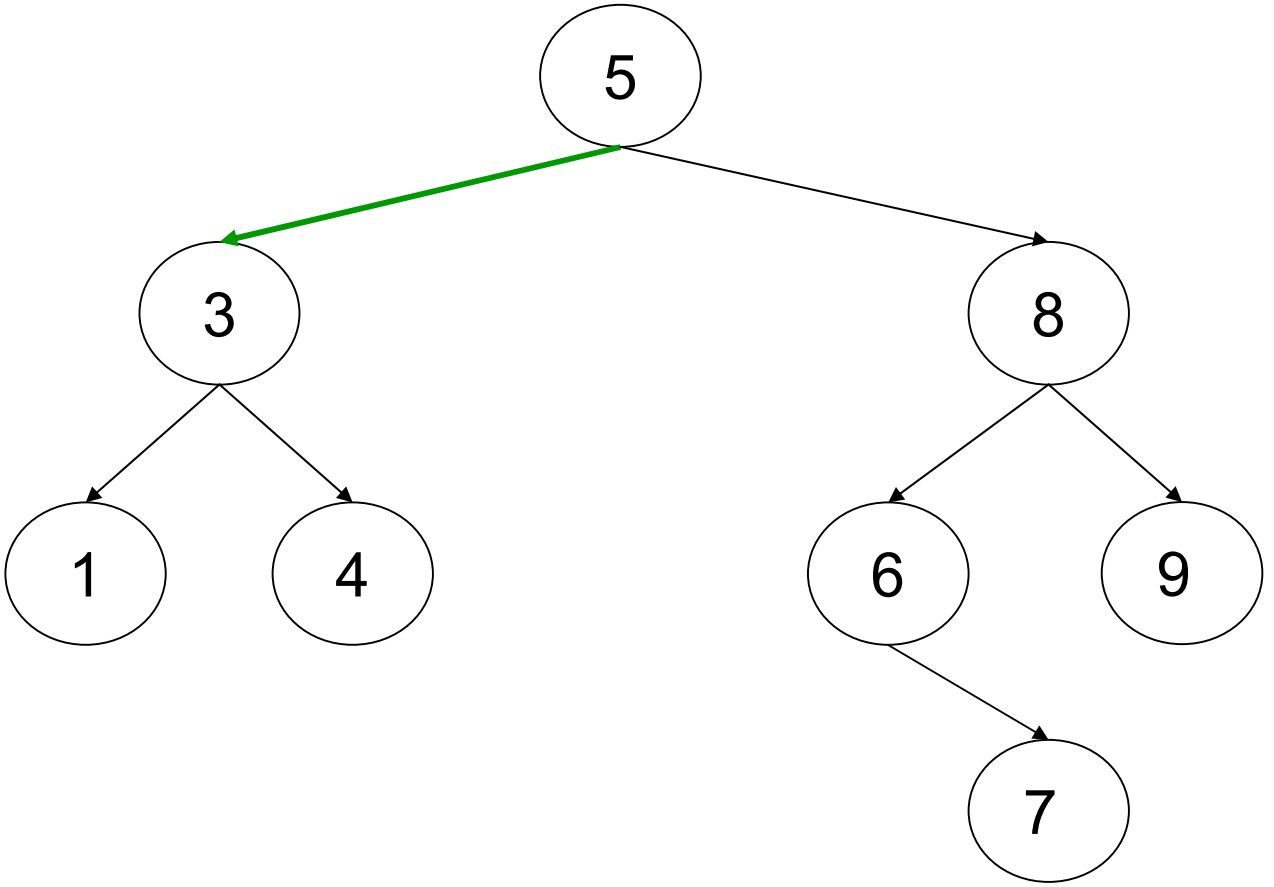
Busca de x = 6



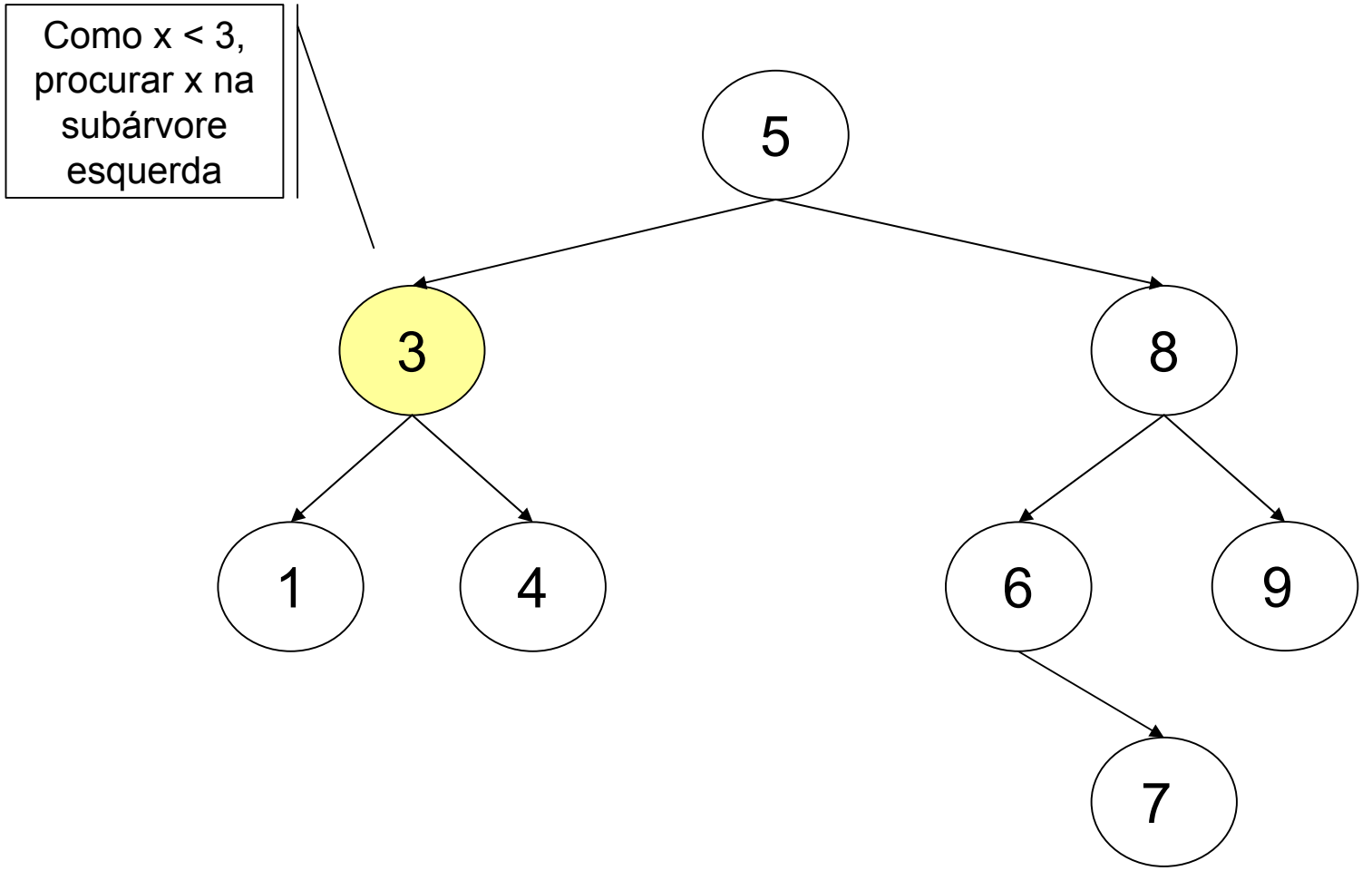
Busca de x = 2



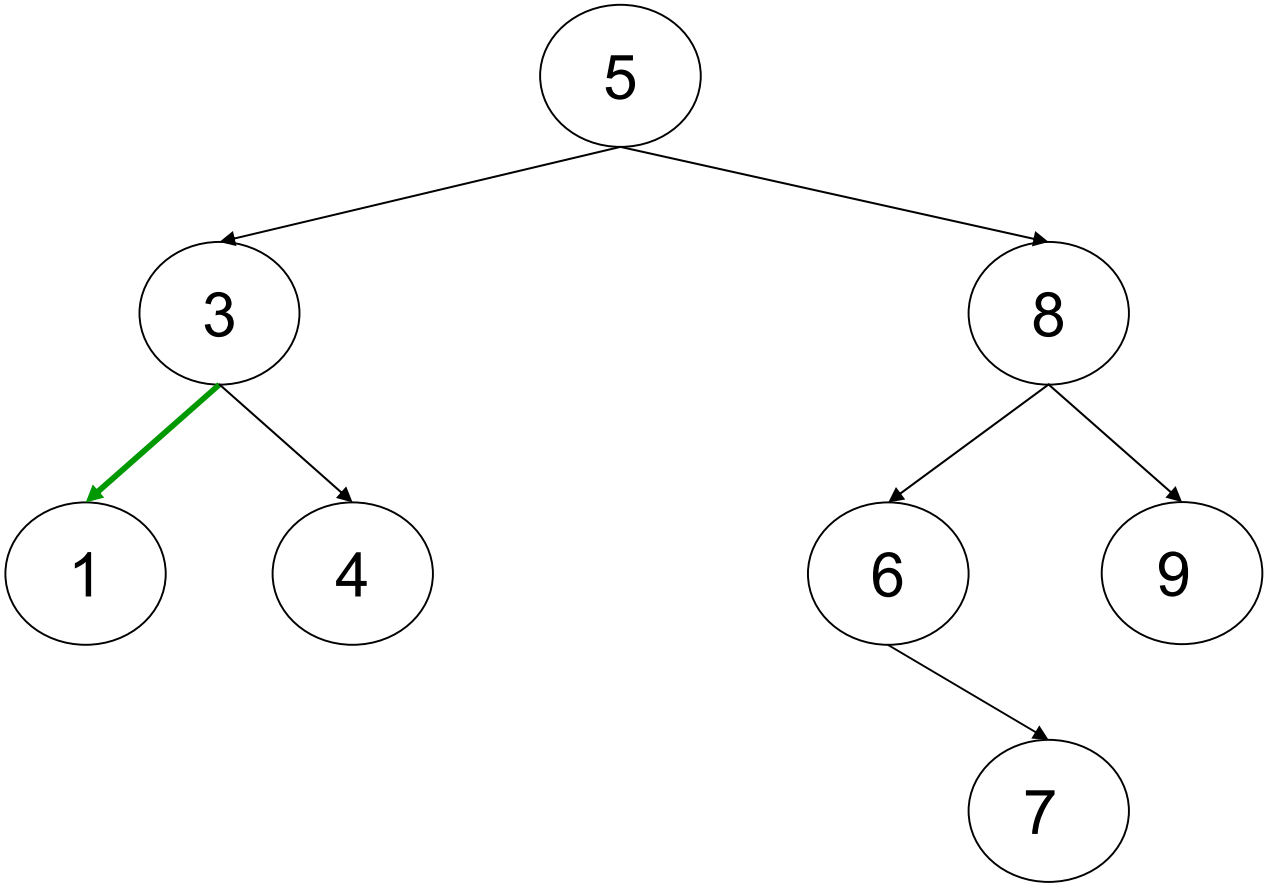
Busca de x = 2



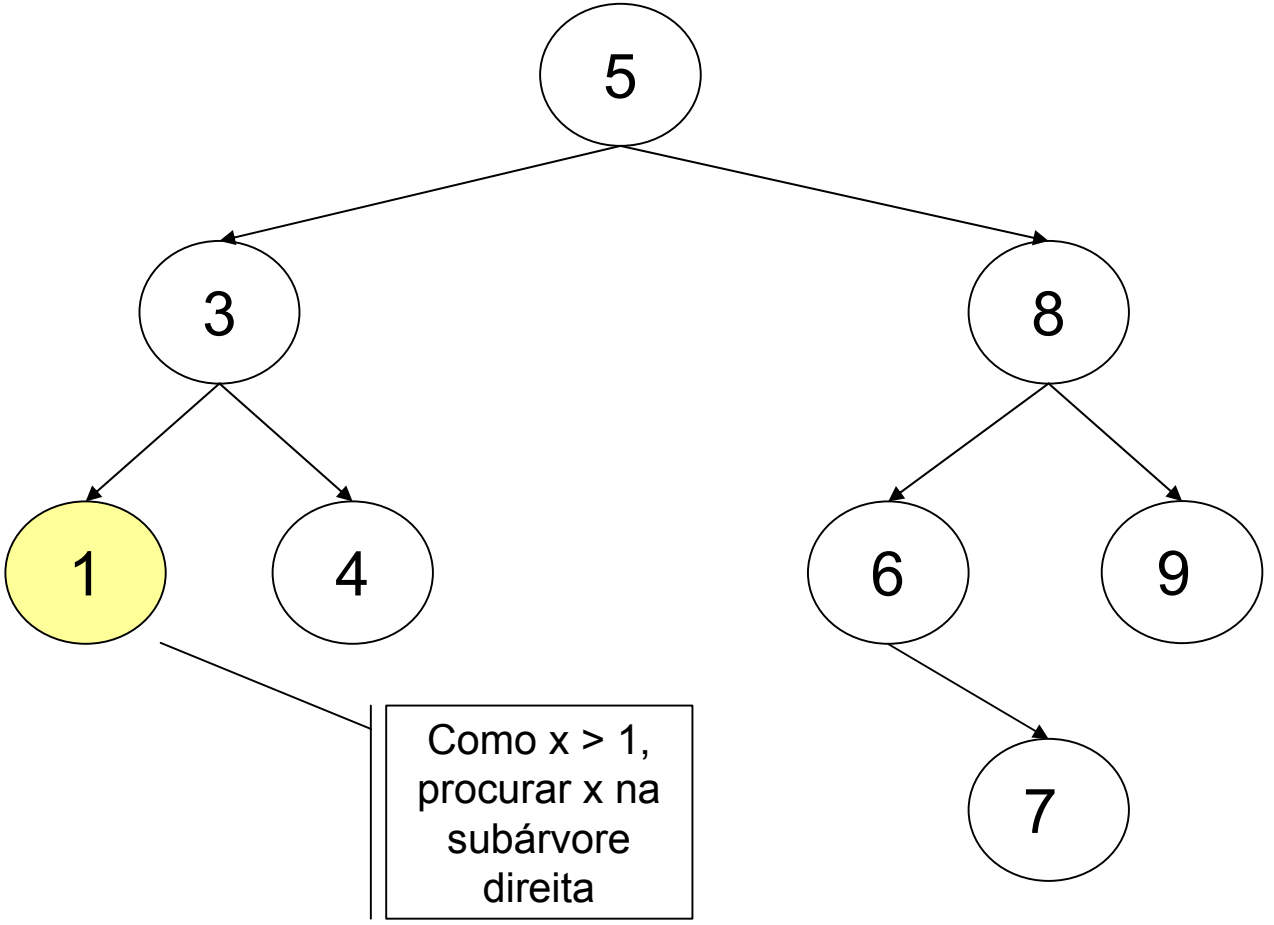
Busca de x = 2



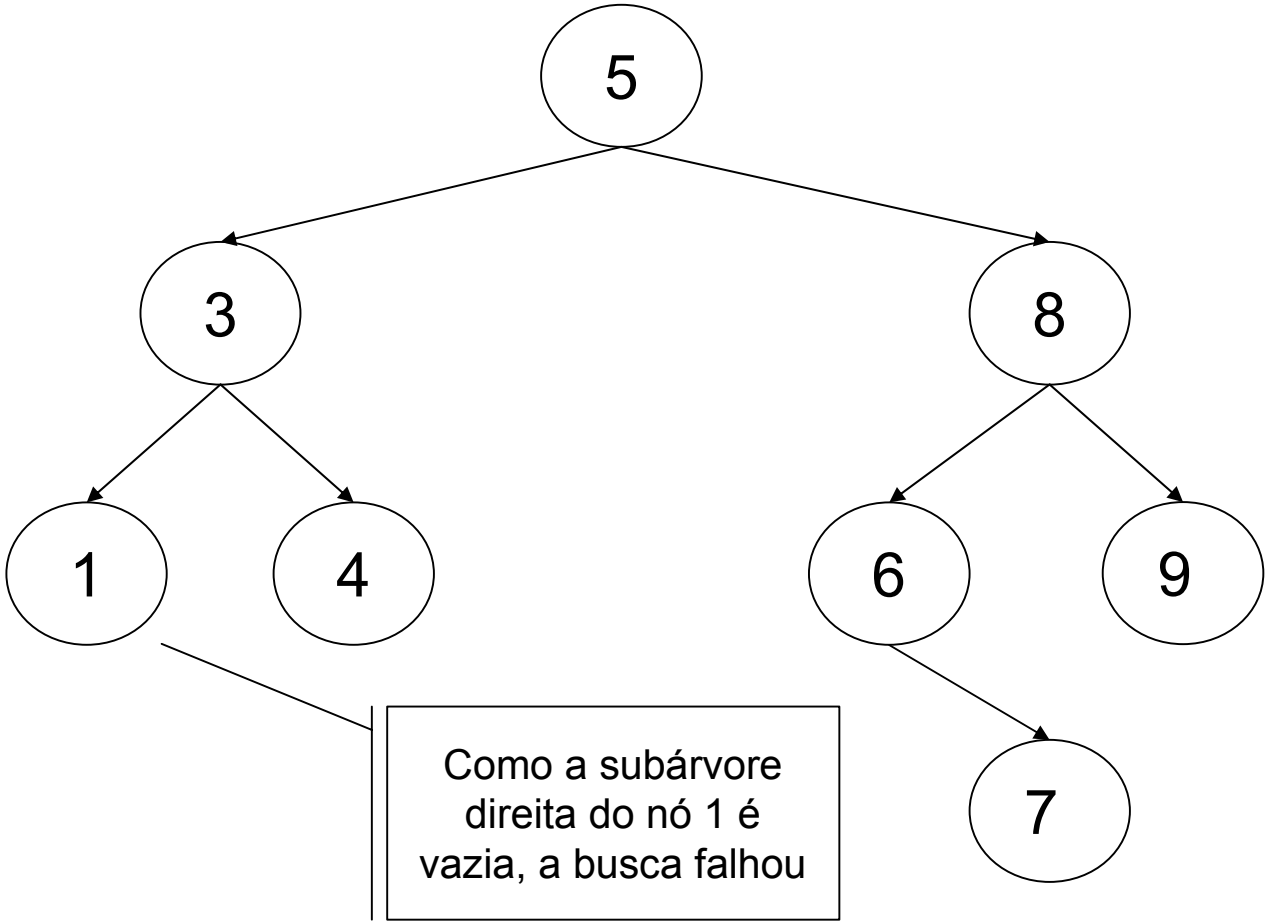
Busca de x = 2



Busca de x = 2



Busca de x = 2


5
3
8
1
4
6
9
7
Como a subárvore direita do nó 1 é vazia, a busca falhou

Operações Básicas: Search (versão iterativa)

```
bool BinarySearchTree::ISearch(int x)
{ TreePointer t=root;

  while (t != NULL && t->Entry != x)
    if(x < t->Entry)
      t = t->LeftNode; // procurar subárvore esquerda
    else
      t = t->RightNode; // procurar subárvore direita

  return (t != NULL);
}
```

Operações Básicas: Search (versão recursiva)

```
bool BinarySearchTree::RSearch(int x, TreePointer &t)
{
  if(t == NULL)
    return false; // x não encontrado

  if(x < t->Entry)
      return RSearch(x,t->LeftNode);
  else
    if(x > t->Entry)
      return RSearch(x,t->RightNode);
   else     // x == t->Entry
      return true;
}
```

Operações Básicas: Search

- Busca Iterativa

```
bool BinarySearchTree::
    Search(int x)
{

   return ISearch(x);

}
```

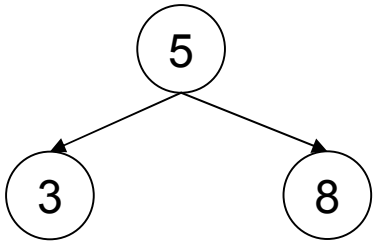
- Busca Recursiva

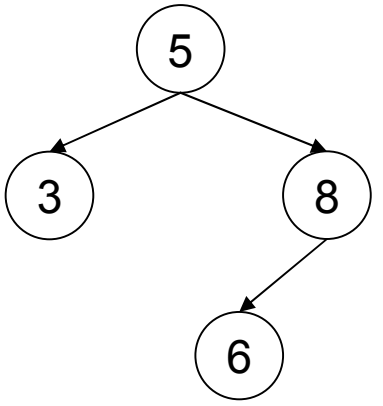
```
bool BinarySearchTree::
    Search(int x)
{

   return RSearch(x, root);

}
```

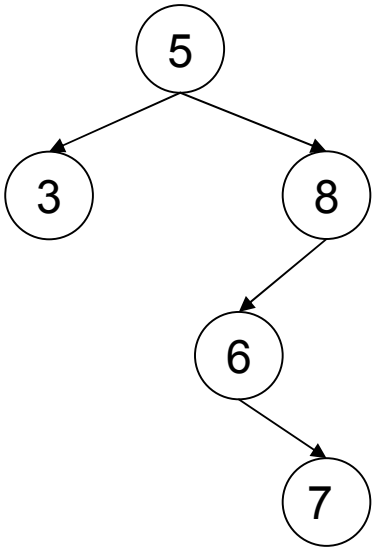
Inserção em ABB

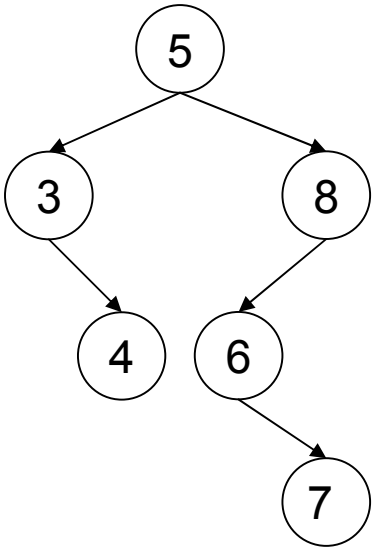
- Para adicionar um elemento **x** a uma ABB
  - Se a árvore estiver vazia, adicione um novo nó contendo o elemento **x**
  - Se a raiz é maior que **x** então insira **x** na subárvore esquerda, caso contrário insira **x** na subárvore direita
- Veremos a implementação iterativa; a implementação da versão recursiva é deixada como exercício

Inserção em ABB







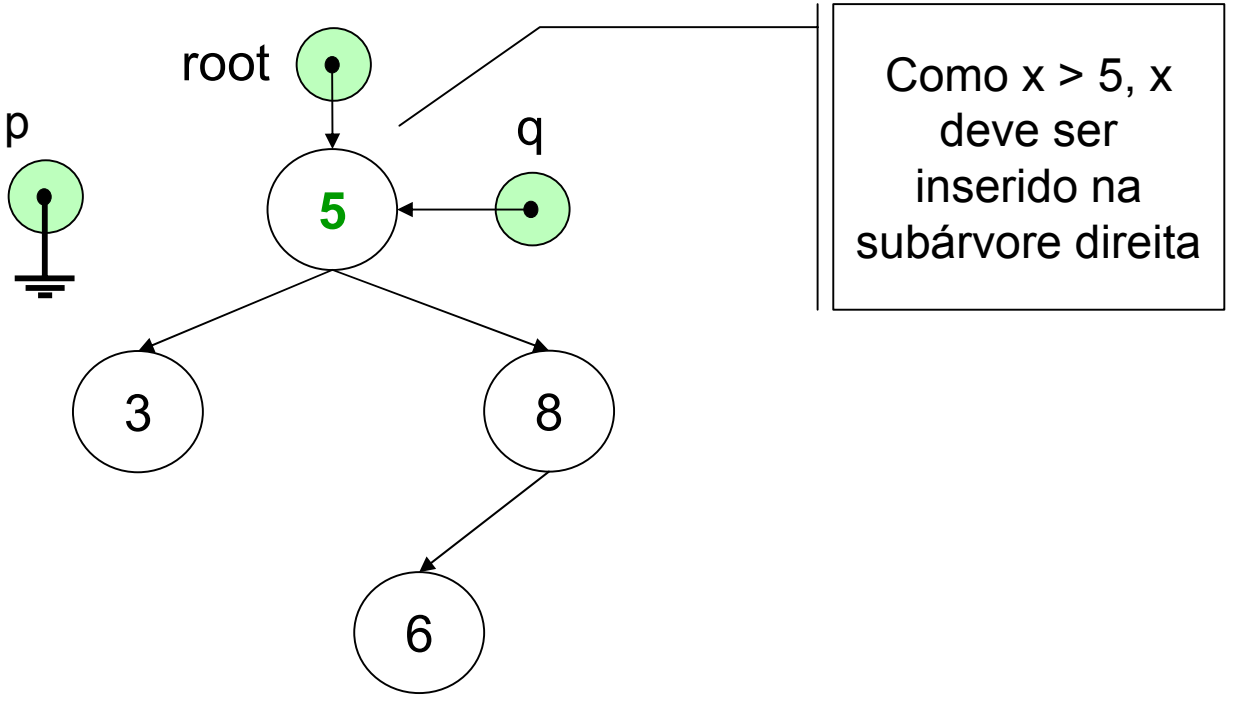


x = 6

x = 7

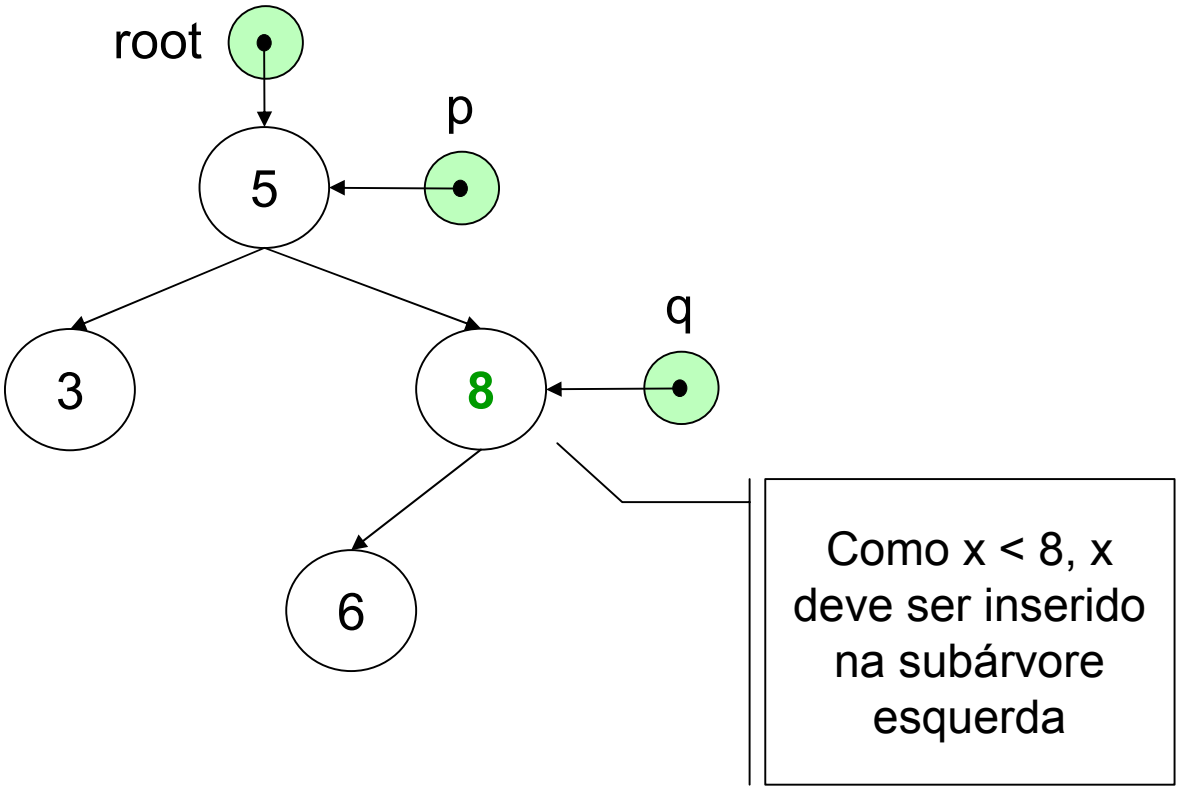
x = 4

Inserção de x = 7



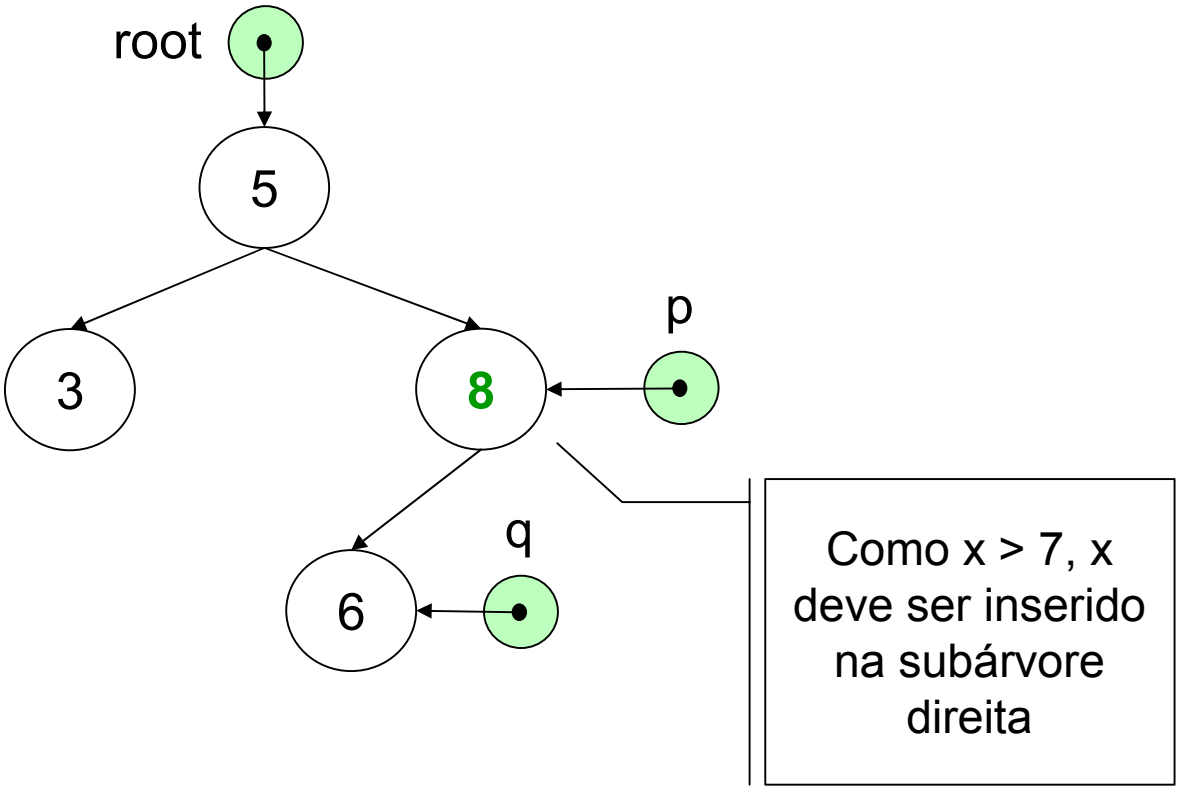
O ponteiro **p** fica sempre uma posição atrás de **q**, ou seja, **p** é o pai de **q**

Inserção de x = 7



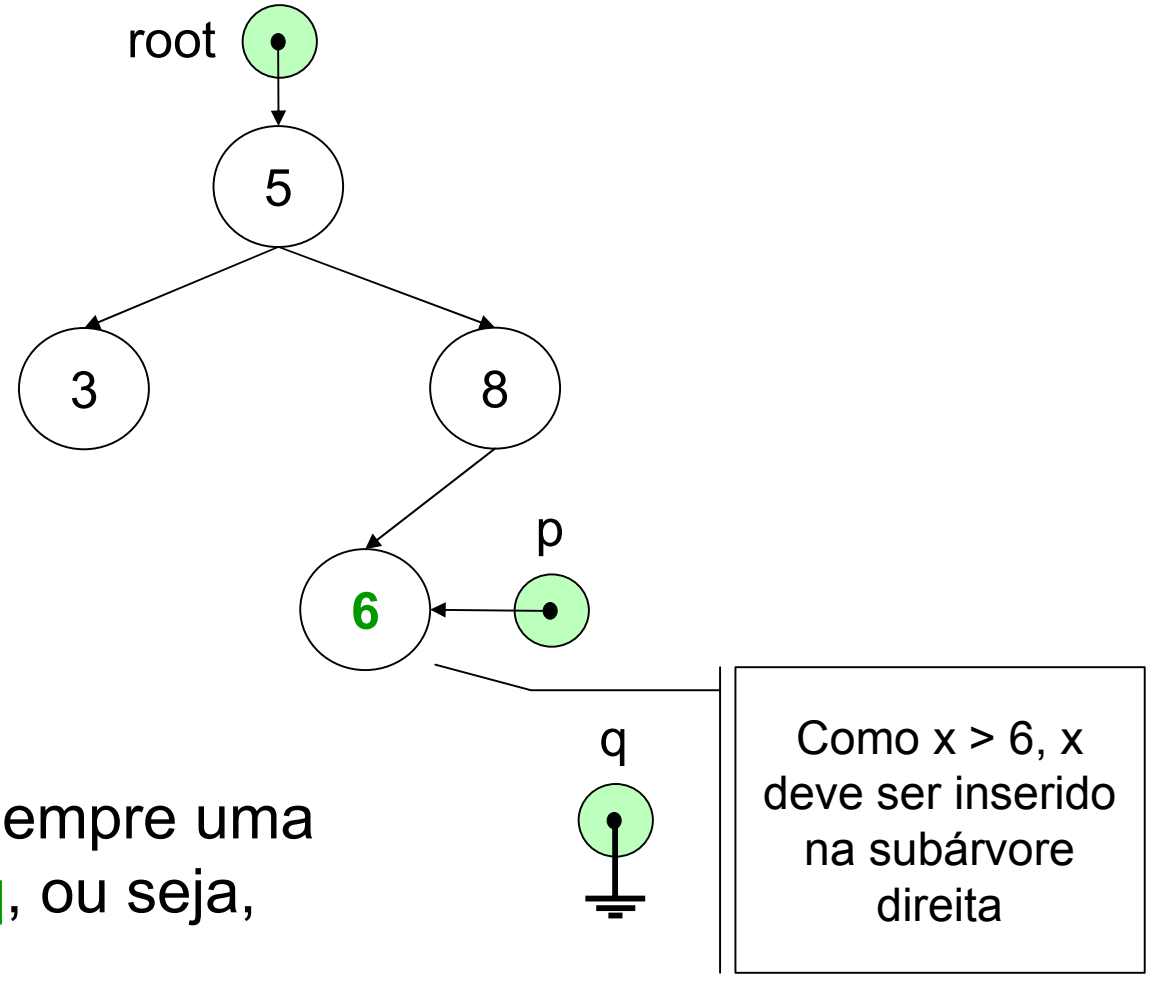
O ponteiro **p** fica sempre uma posição atrás de **q**, ou seja, **p** é o pai de **q**

Inserção de x = 7



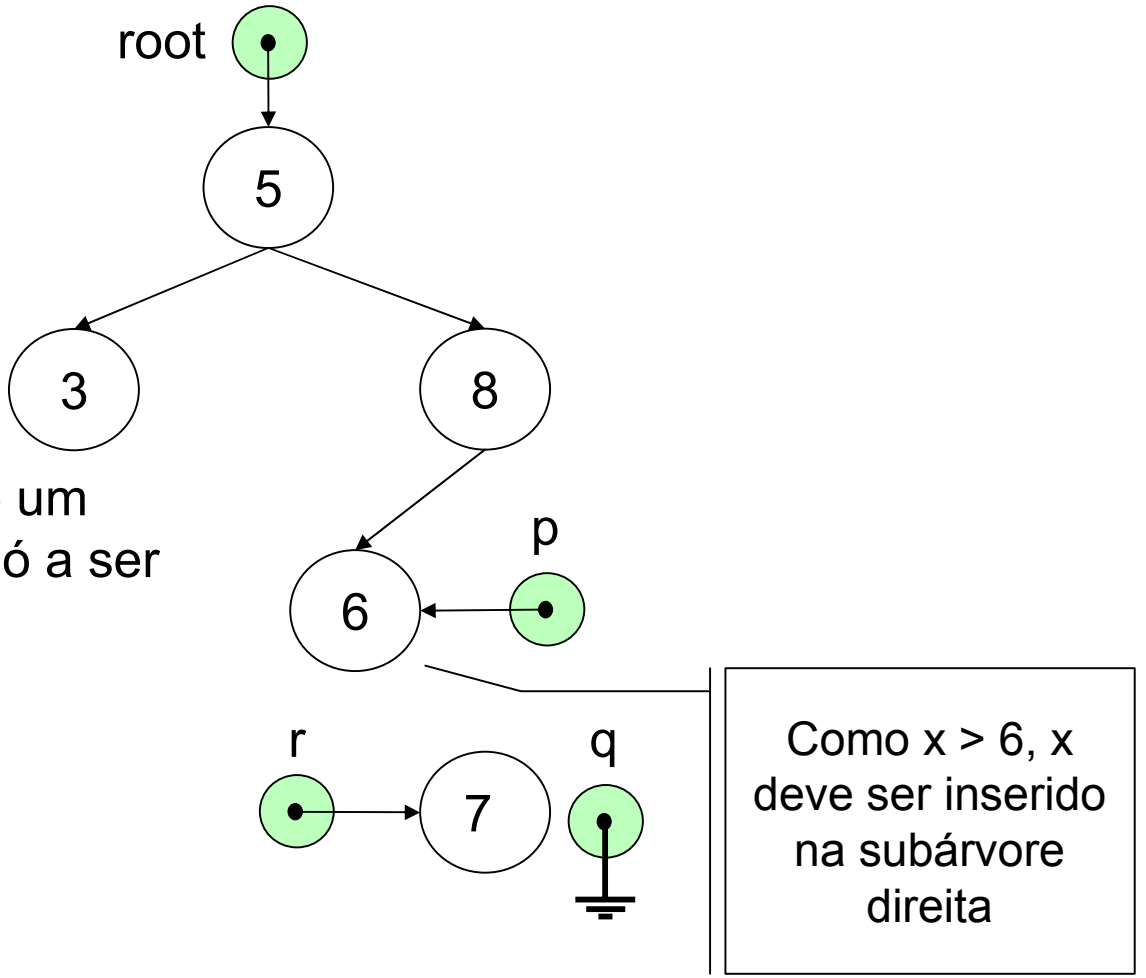
O ponteiro **p** fica sempre uma posição atrás de **q**, ou seja, **p** é o pai de **q**

Inserção de x = 7



O ponteiro **p** fica sempre uma posição atrás de **q**, ou seja, **p** é o pai de **q**

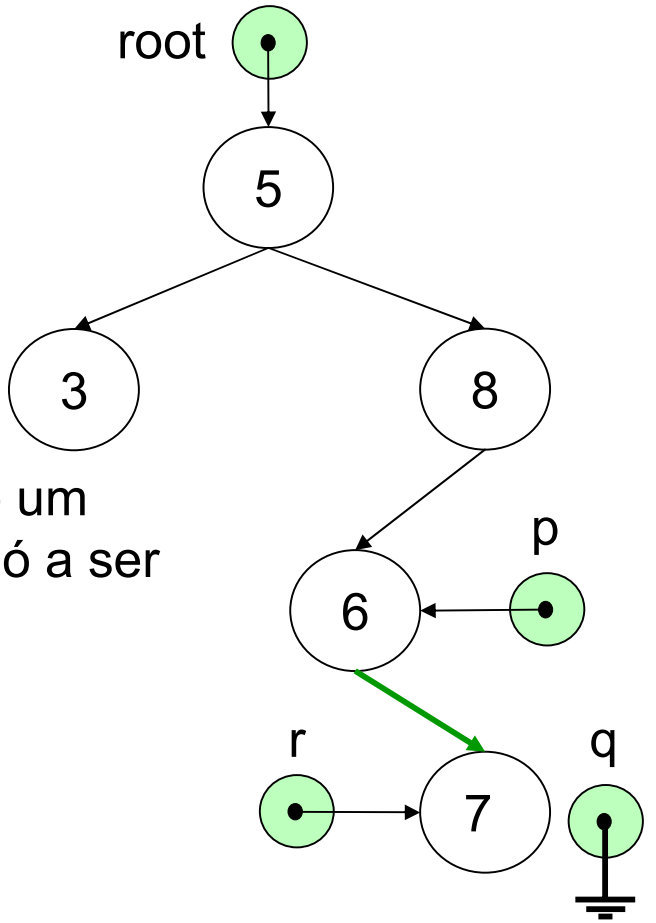
Inserção de x = 7



Assuma **r** como sendo um ponteiro para o novo nó a ser inserido:

```
if(x < p->Entry)
  …
```

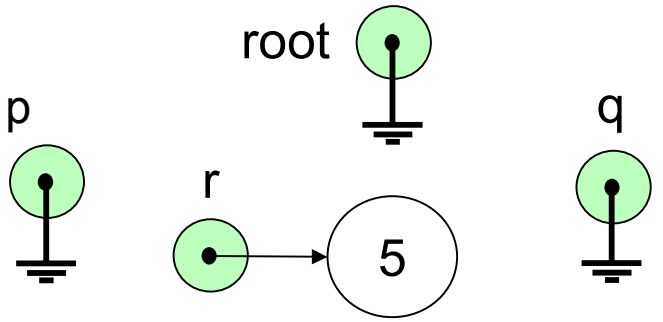
Inserção de x = 7



Assuma **r** como sendo um ponteiro para o novo nó a ser inserido:

```
if(x < p->Entry)
  p->LeftNode = r;
else
  p->RightNode = r;
```

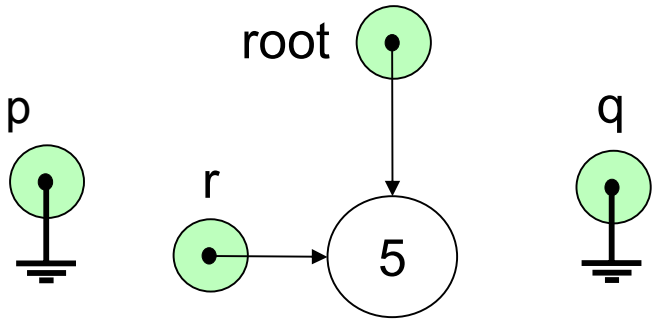
Inserção de x = 5



Entretanto, se a árvore estiver vazia:

```
if(p == NULL)
  ...
```

Inserção de x = 5



Entretanto, se a árvore estiver vazia:

```
if(p == NULL)
  root = r;
```

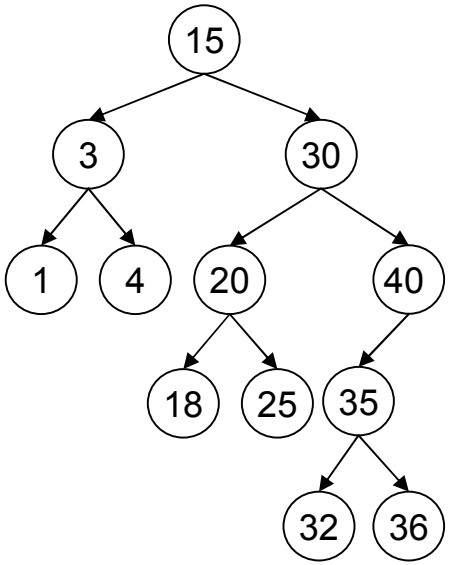
Operações Básicas: Insert

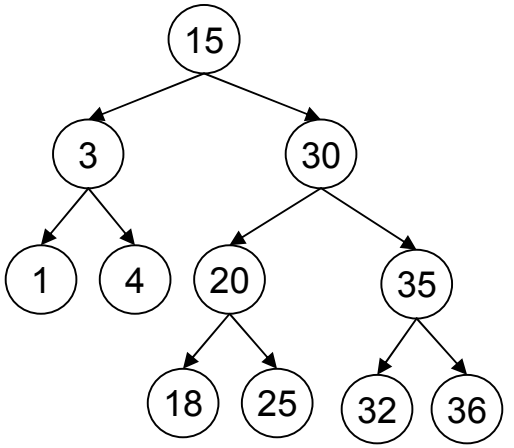
```
void BinarySearchTree::Insert(int
    x)
{ TreePointer p=NULL, q=root, r;

  while (q != NULL)
  { p = q;
    if(x < q->Entry)
      q = q->LeftNode;
    else
      q = q->RightNode;
  }

  r = new TreeNode;
  r->Entry = x;
  r->LeftNode = NULL;
  r->RightNode = NULL;
  if(p == NULL)
    root = r;  // árvore vazia
  else
    if(x < p->Entry)
      p->LeftNode = r;
    else
      p->RightNode = r;

}
```

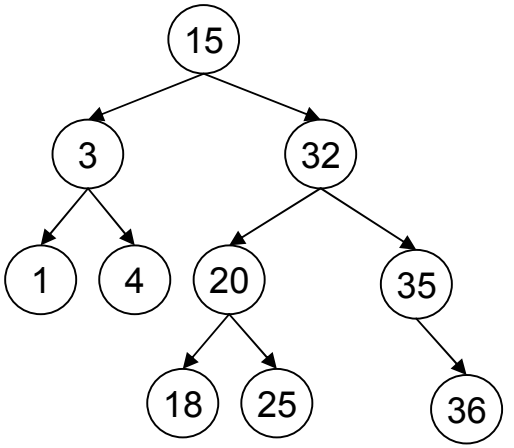
Remoção em ABB

- Para remover um elemento **x** a uma ABB há três situações possíveis:
  - Não existe nenhum nó com chave igual a **x**
  - O nó com chave **x** possui, no máximo, uma das subárvores
  - O nó com chave **x** possui as duas subárvores

Remoção em ABB



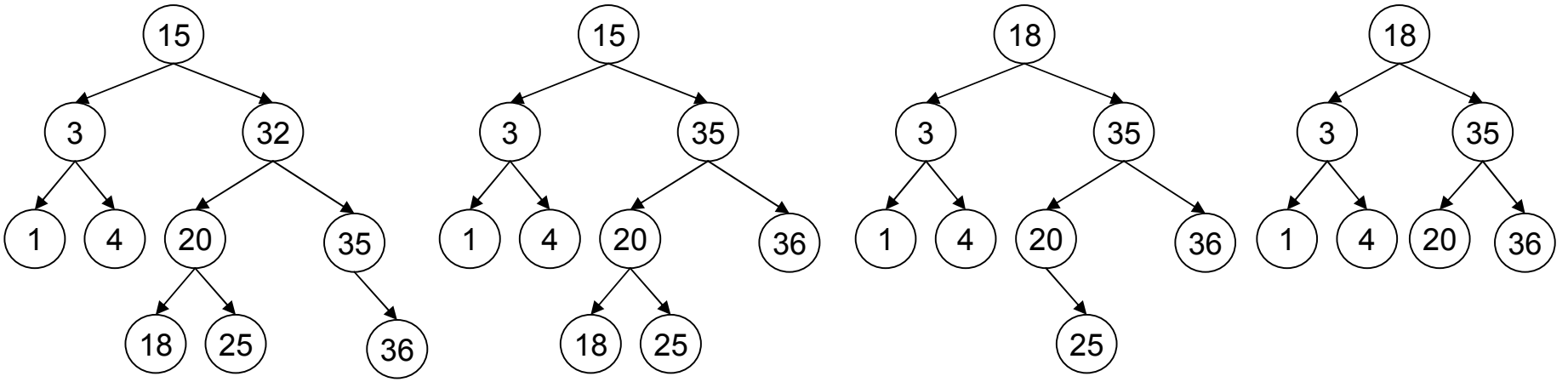




x = 40

x = 30

Remoção em ABB

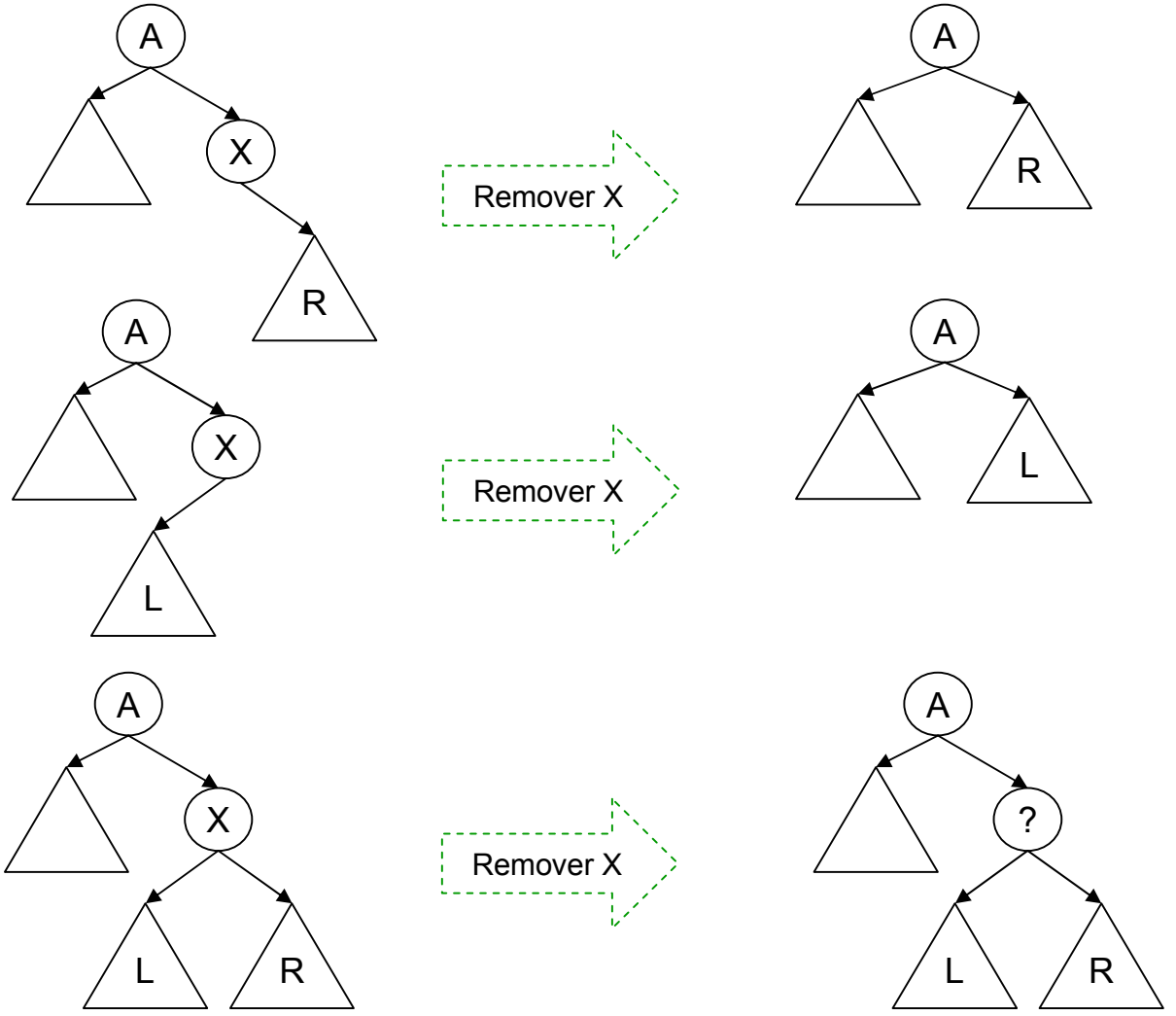


x = 32

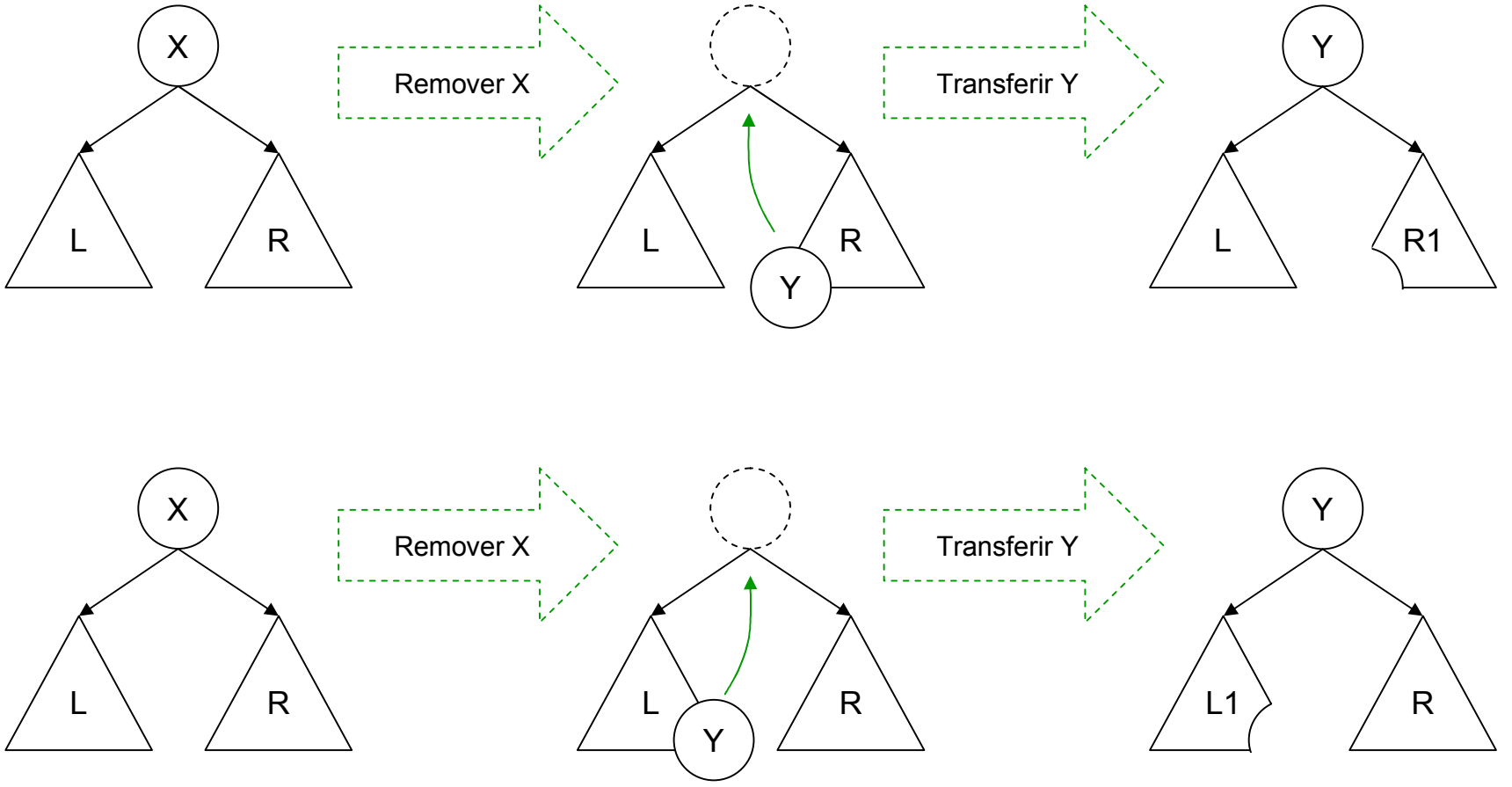
x = 15

x = 25

Remoção em ABB



Remoção em ABB



Remoção em ABB

- Dessa forma, quando o elemento **x** sendo removido possui as duas subárvores não vazias, há duas estratégias possíveis para substituir o valor de **x**, preservando as propriedades de ABB
  - Encontrar o elemento de **menor** valor **y** na subárvore **direita** de **x** de transferi-lo para o nó ocupado por **x**
  - Encontrar o elemento de **maior** valor **y** na subárvore **esquerda** de **x** de transferi-lo para o nó ocupado por **x**
- É óbvio que, após a transferência, o nó **y** deve ser removido da árvore em ambas as estratégias
- Adotaremos a primeira estratégia, ficando a segunda como exercício

Remoção em ABB

- Definiremos dois métodos auxiliares privados para a remoção em ABB
  - Delete(int x, TreePointer &p)
  - DelMin(TreePointer &q,TreePointer &r)
- O primeiro método deve ser chamado passando a raiz (root) como o parâmetro **p**
- O segundo método procura, na subárvore direita, pelo menor valor e só é chamando quando o nó com chave **x** possui as duas subárvores

Remoção em ABB

```
void BinarySearchTree::Delete(int x)
{  Delete(x,root);
}
//-----------------------------------------
void BinarySearchTree::Delete(int x,
     TreePointer &p)
{ TreePointer q;

   if(p == NULL)
   {  cout << “Elemento inexistente”;
      abort();
   }
   if(x < p->Entry)
      Delete(x,p->LeftNode);
   else
     if(x > p->Entry)
        Delete(x,p->RightNode);
     else // remover p->
      { q = p;
        if(q->RightNode == NULL)
          p = q->LeftNode;
        else
          if(q->LeftNode == NULL)
            p = q->RightNode;
          else
            DelMin(q,q->RightNode);
        delete q;
      }
}
```

Remoção em ABB

```
void BinarySearchTree::DelMin(TreePointer &q,
   TreePointer &r)
{
   if(r->LeftNode != NULL)
     DelMin(q,r->LeftNode);
   else
    {  q->Entry = r->Entry;
       q = r;
       r = r->RightNode;
    }
}
```

Busca com Inserção

- Vamos considerar a situação em que, caso um elemento não se encontre na árvore, então ele deve ser inserido; caso já esteja, um contador deve ser atualizado
- Um exemplo típico é a contagem do número de palavras em um texto
- Para tanto, nossa estrutura de dados será estendida para a seguinte definição

```
struct TreeNode
{  string Entry;            // chave
   int count;
   TreeNode *LeftNode, *RightNode;
};
```

Busca com Inserção

```
void BinarySearchTree::SearchInsert(int x)
{  SearchInsert(x,root);
}
//-------------------------------------------------------------------------
void BinarySearchTree::SearchInsert(int x, TreePointer &t)
{ if(t == NULL)          // x não encontrado; inserir
  {  t = new TreeNode;
     t->Entry = x;
     t->count = 1;
     t->LeftNode = t->RightNode = NULL;
  }
  else
    if(x < t->Entry)  // procurar x na subárvore esquerda
      SearchInsert(x,t->LeftNode);
    else
      if(x > t->Entry) // procurar x na subárvore direita
        SearchInsert(x,t->RightNode);
      else                  // x encontrado, atualizar contador
        t->count++;
}
```

Considerações Finais

- Como vimos, em uma ABB perfeitamente balanceada com **n** nós e altura **h,** os algoritmos de busca e inserção tomam tempo O(**h**) = O($\log_2$ **n**)
- Entretanto, uma árvore pode se degenerar em uma lista
  - No pior caso os algoritmos levam O(**n**)
  - Em média O(**n**/2)
- A situação no pior caso conduz a um desempenho muito pobre
- Veremos na próxima apresentação uma forma de garantir tempo O($\log_2$ **n**), mesmo no pior caso