Binary Search Trees

Cormen: Capítulo 12

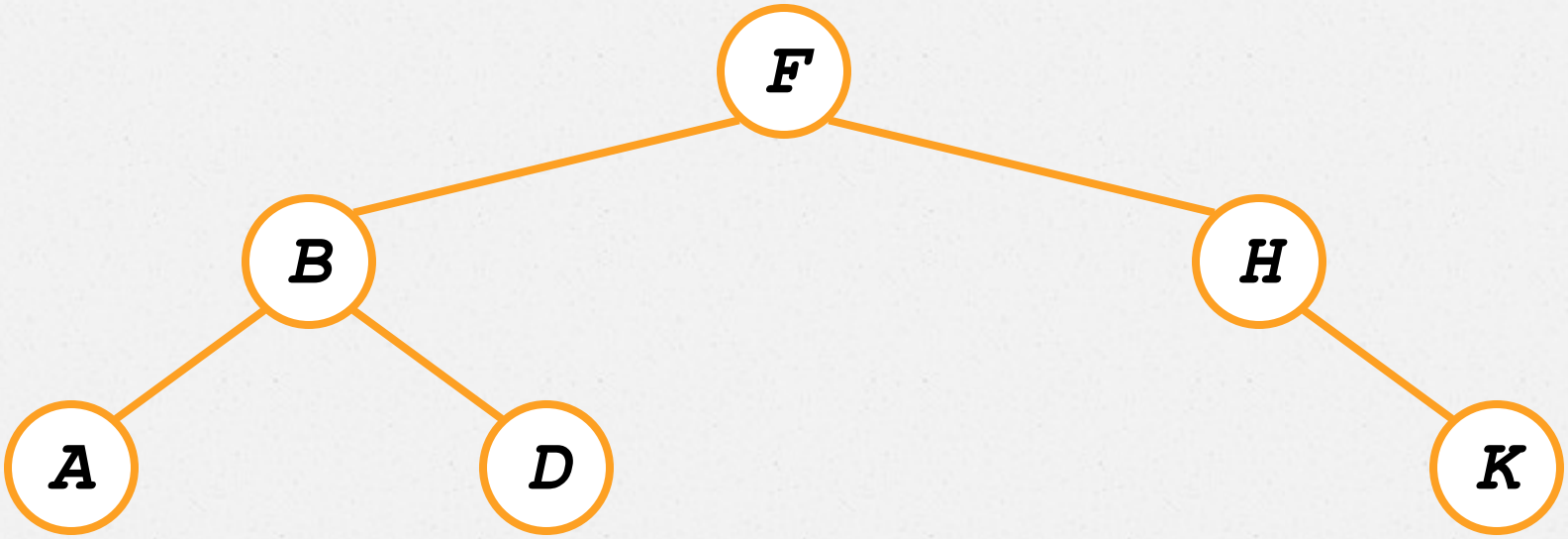
Binary Search Trees - BSTs

- *Binary Search Trees* (BSTs) são estruturas de dados compostas por:
  - ***key***: identificador que leva à ordenação total.
  - ***left***: ponteiro para nó filho esquerdo que pode ser NULL
  - ***right***: ponteiro para nó filho direito que pode ser NULL
  - ***p***: ponteiro para um nó pai que será NULL para o nó raiz.

Binary Search Trees

- Propriedade:

  key[left(x)] ≤ key[x] ≤ key[right(x)]



Inorder Tree Walk

- Imprime os elementos em ordem crescente

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

- Preorder tree walk: current, left e right.
- Postorder tree walk: left, right e current.

Busca em BSTs

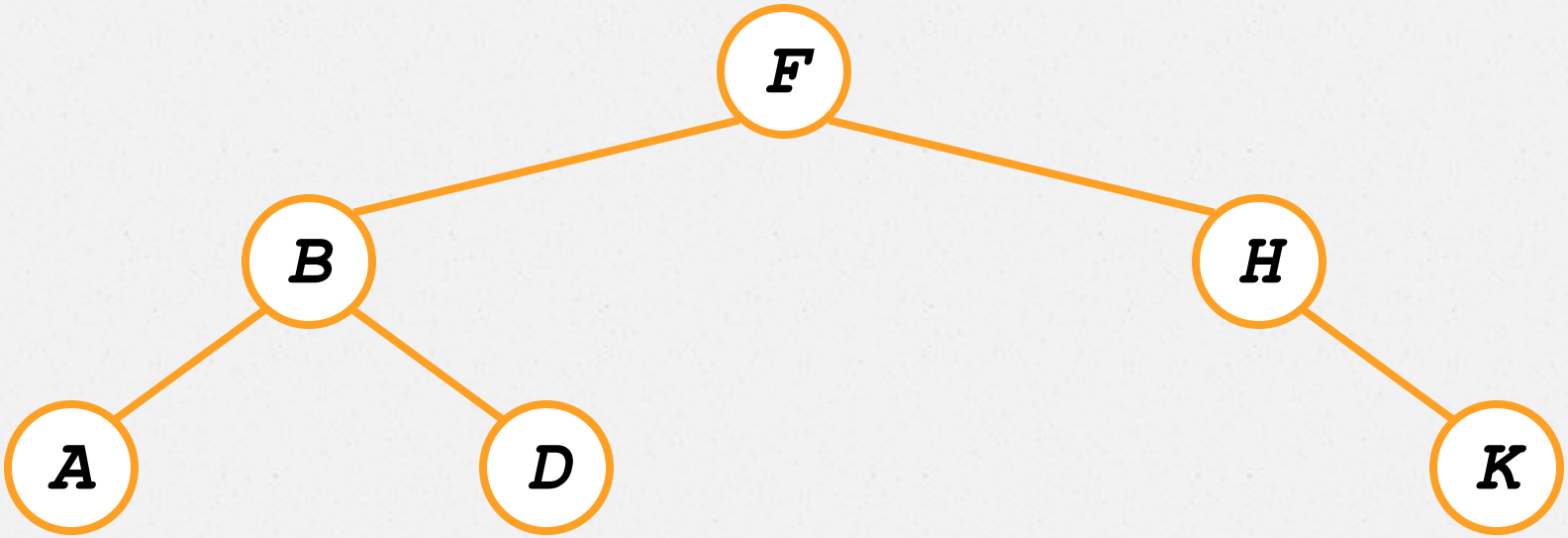
Entrada: x: ponteiro para um nó.

k: chave.

```
TreeSearch(x, k)
     if (x = NULL  or  k = key[x])
           return x;
     if (k < key[x])
           return TreeSearch(left[x], k);
     else
           return TreeSearch(right[x], k);
```

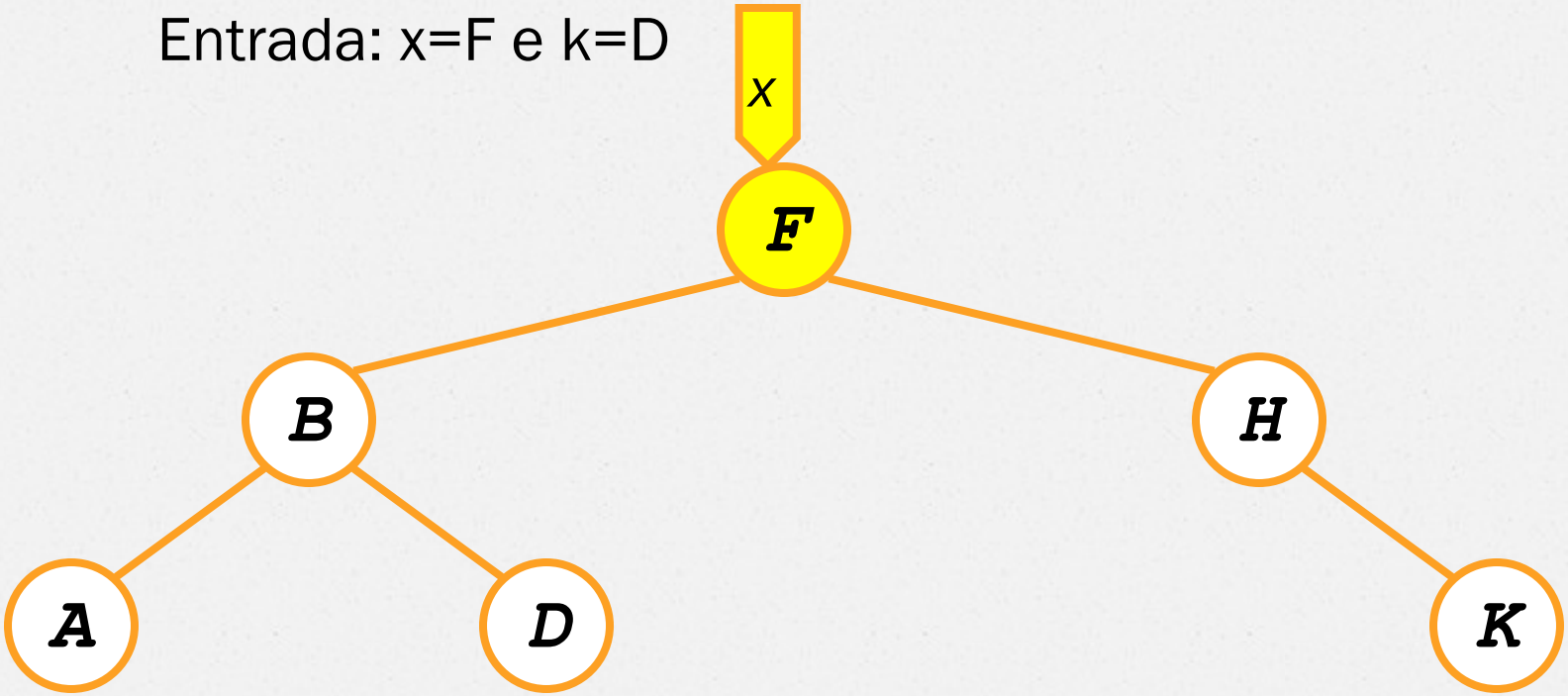
Busca em BSTs

Entrada: x=F e k=D



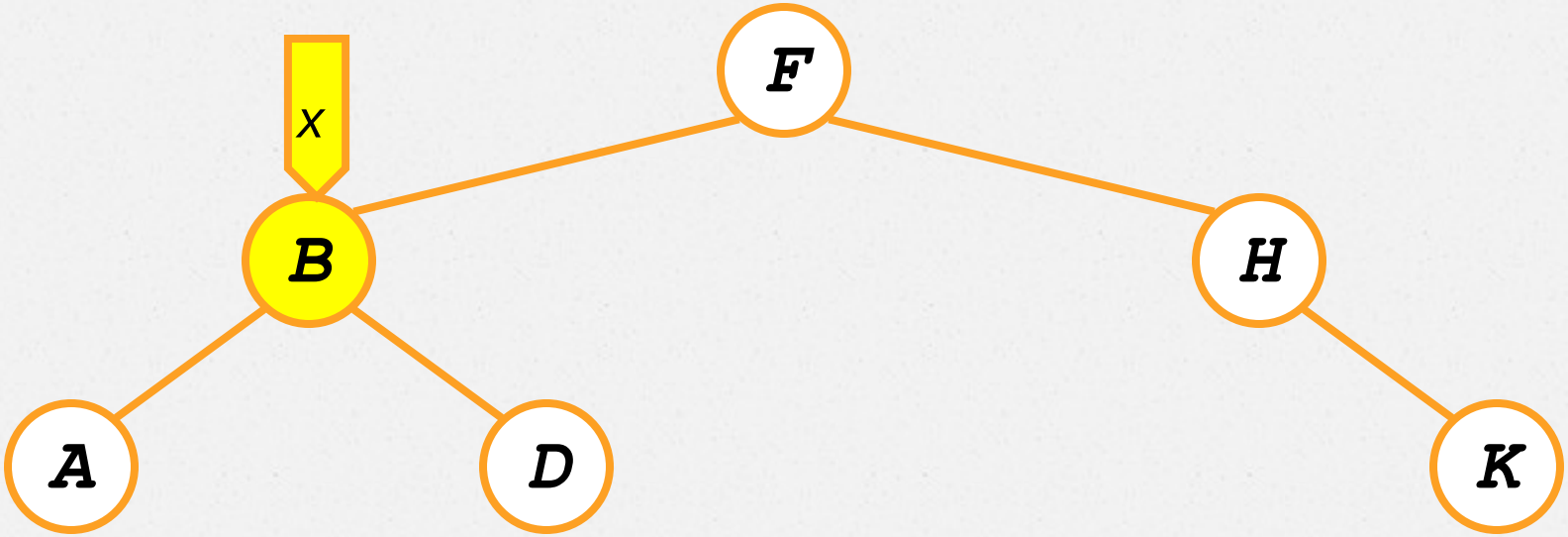
Busca em BSTs

Entrada: x=F e k=D



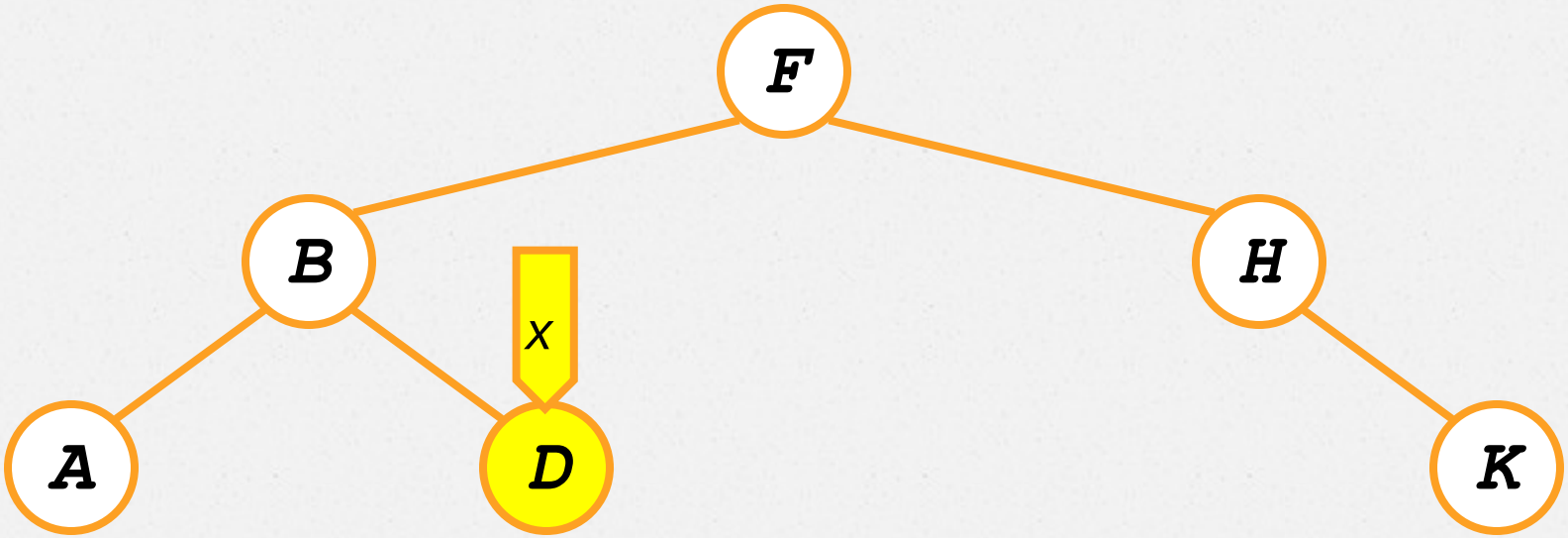
Busca em BSTs

Entrada: x=F e k=D



Busca em BSTs

Entrada: x=F e k=D



Busca em BSTs

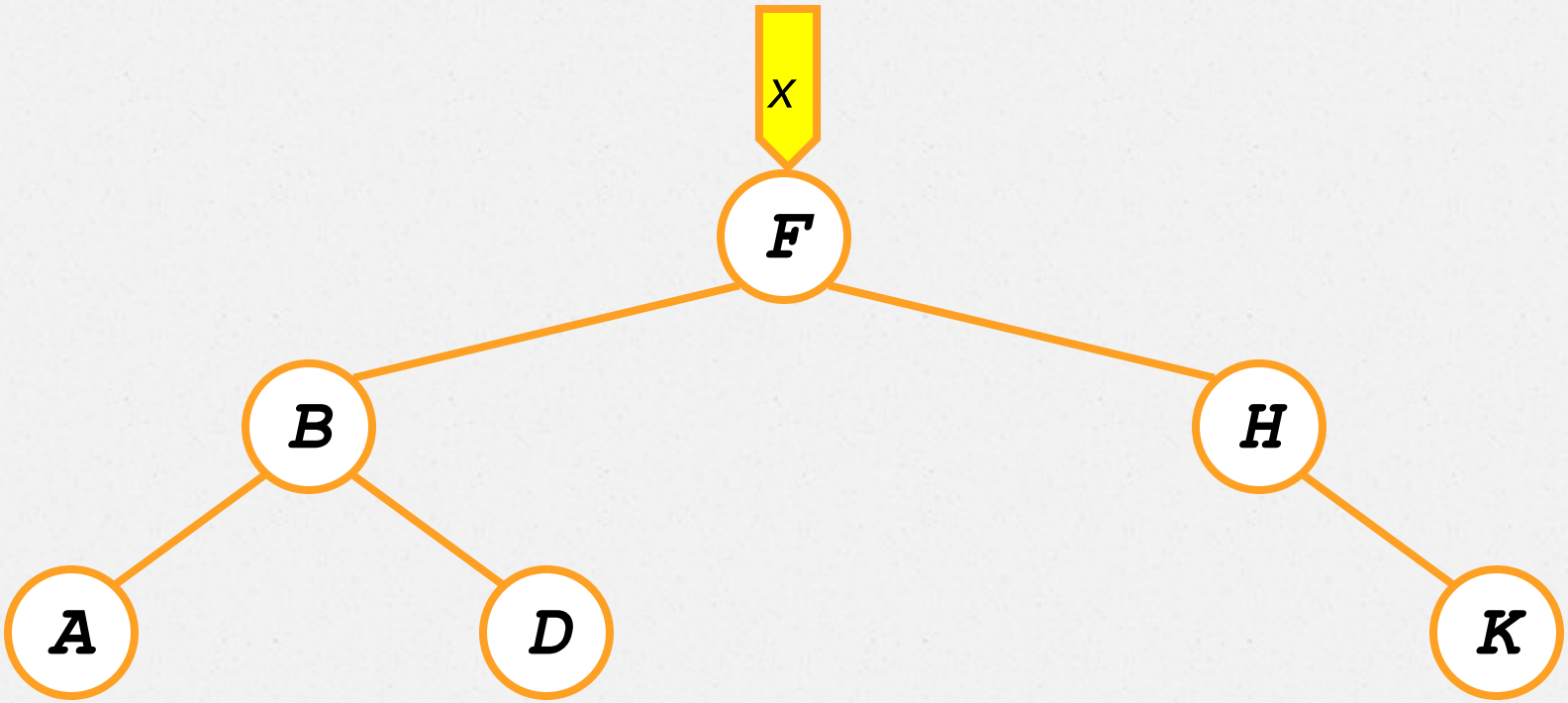
```
TreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

Busca em BSTs

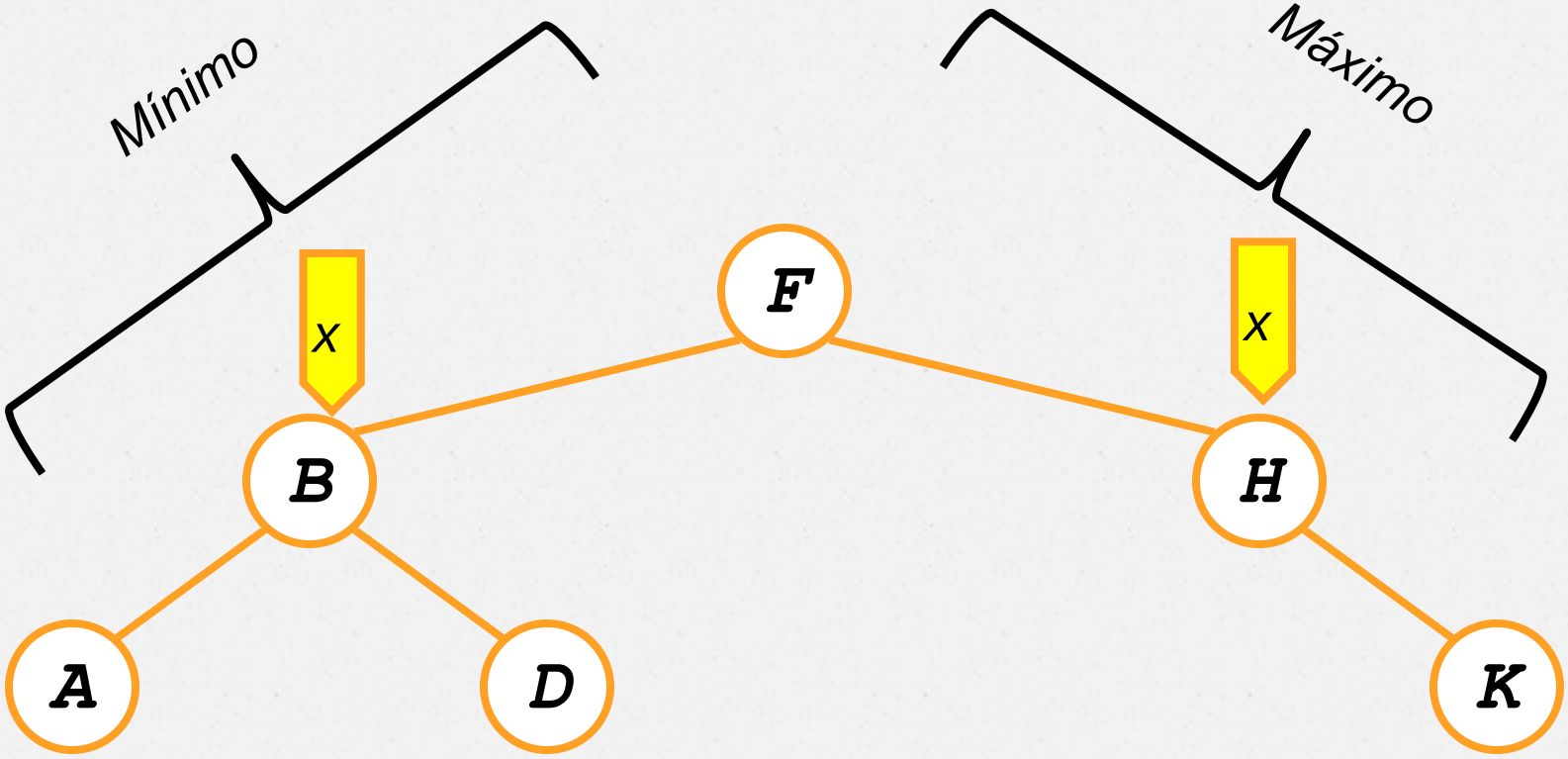
```
TREE_MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
return x

TREE_MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

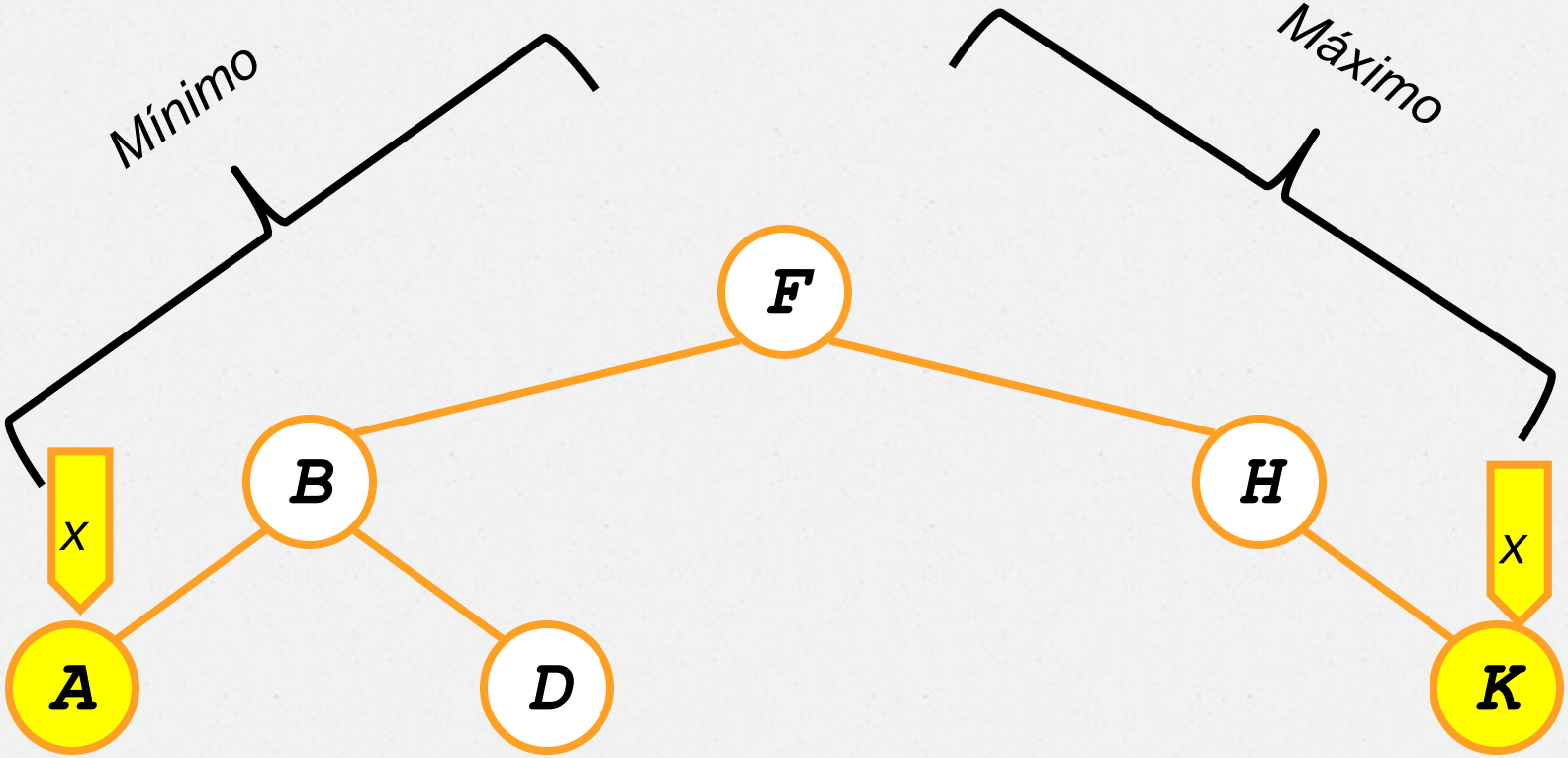
Busca em BSTs



Busca em BSTs



Busca em BSTs



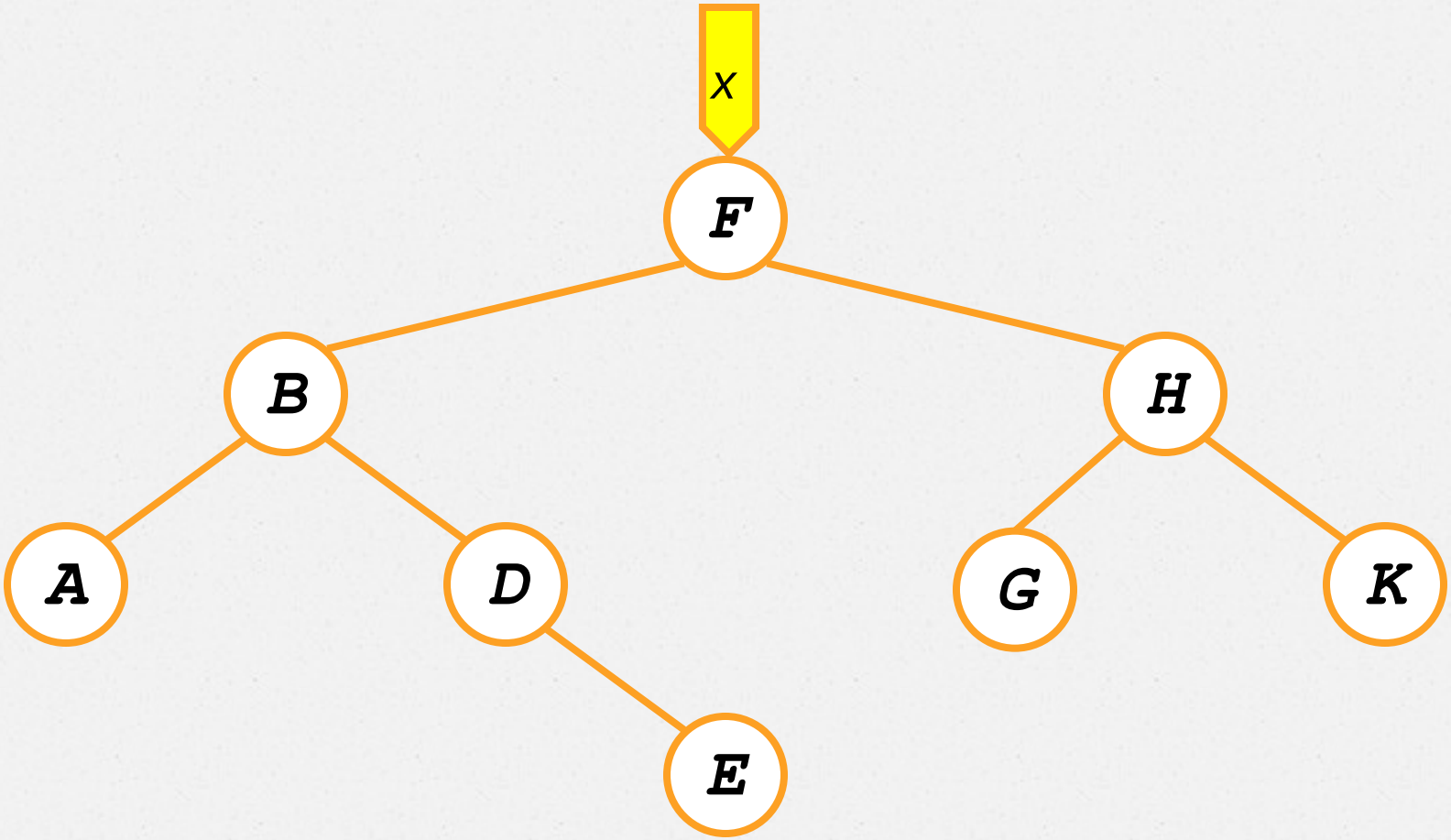
Busca em BSTs

```
TREE_SUCCESSOR
1  if right[x]≠ nil
2     return TREE_MINIMUM(right[x])
3  y ←p[x]
4 while y ≠ nil and x = right[y]
5        x←y
6        y←p[x]
7  return y
```

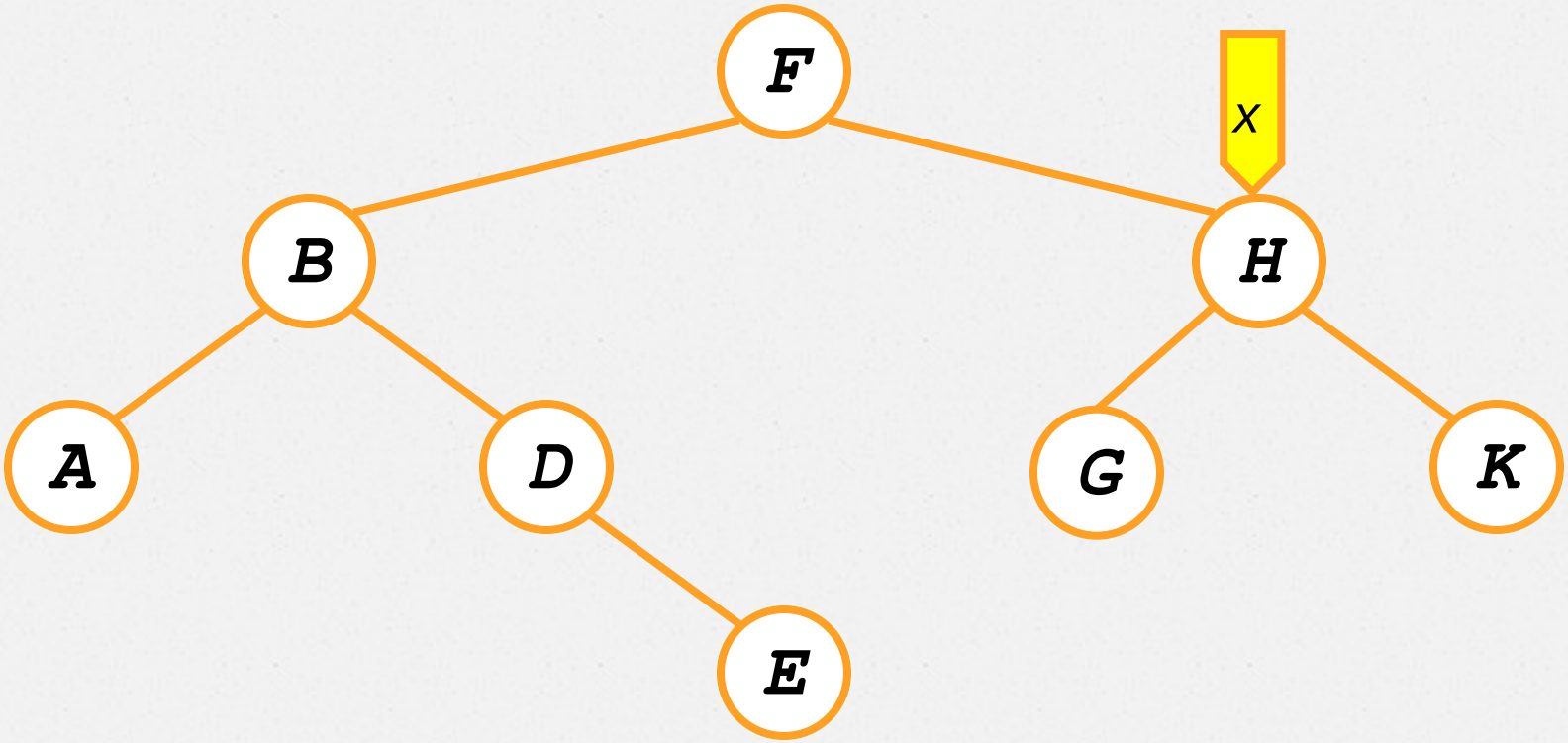
Busca em BSTs

```
TREE_SUCCESSOR
1  if right[x]≠ nil
2      return TREE_MINIMUM(right[x])
3  y ←p[x]
4 while y ≠ nil and x = right[y]
5        x←y
6        y←p[x]
7  return y
```

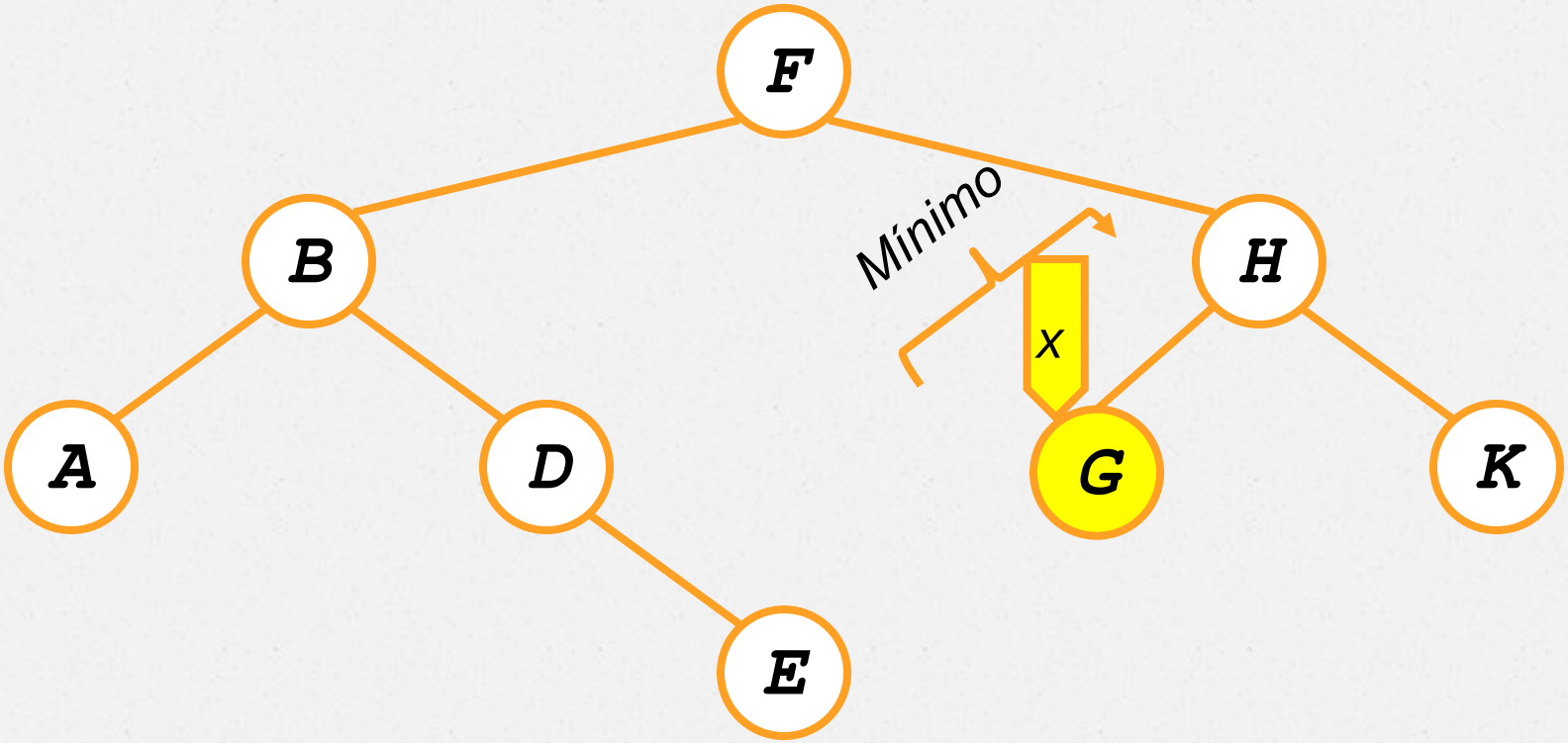
Busca em BSTs



Busca em BSTs



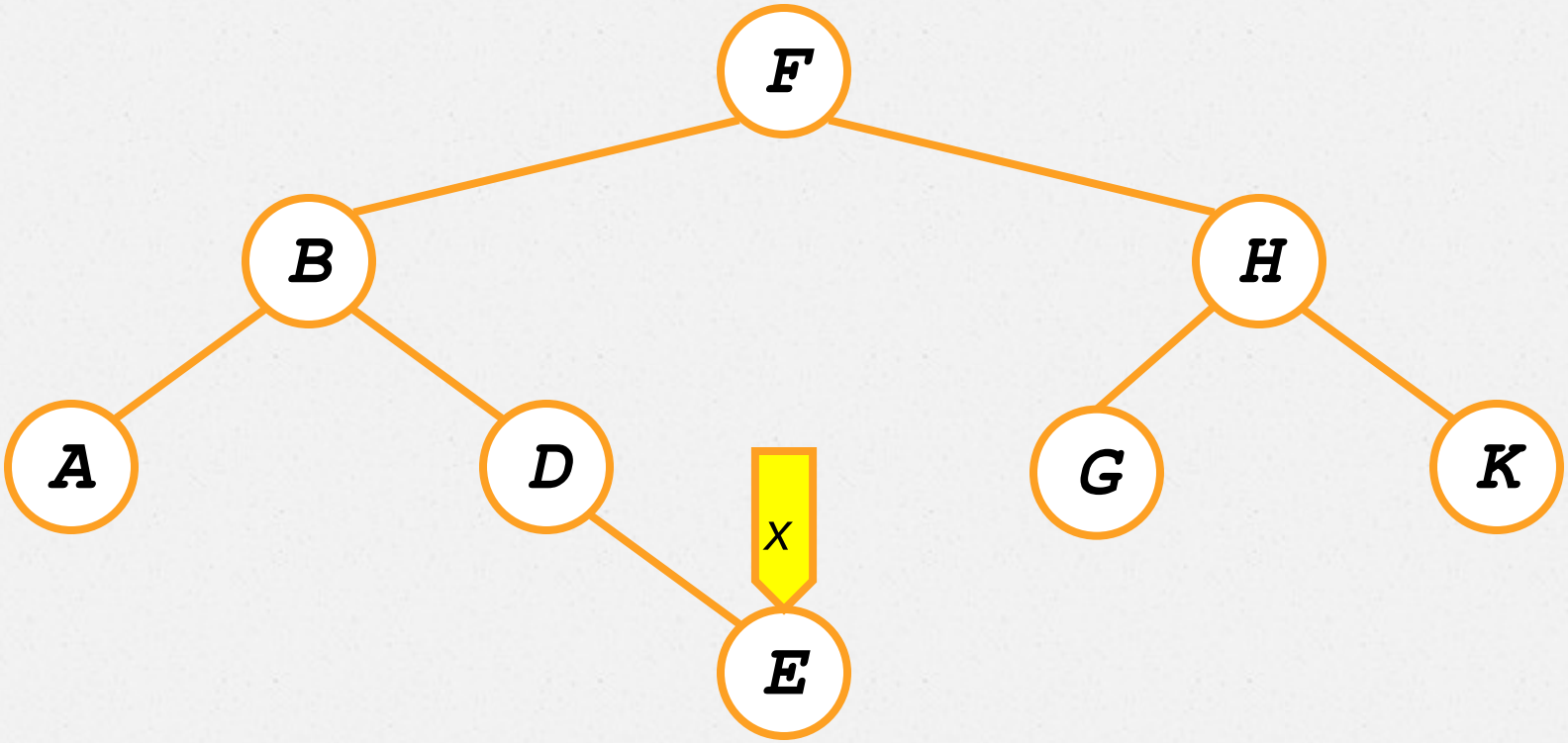
Busca em BSTs



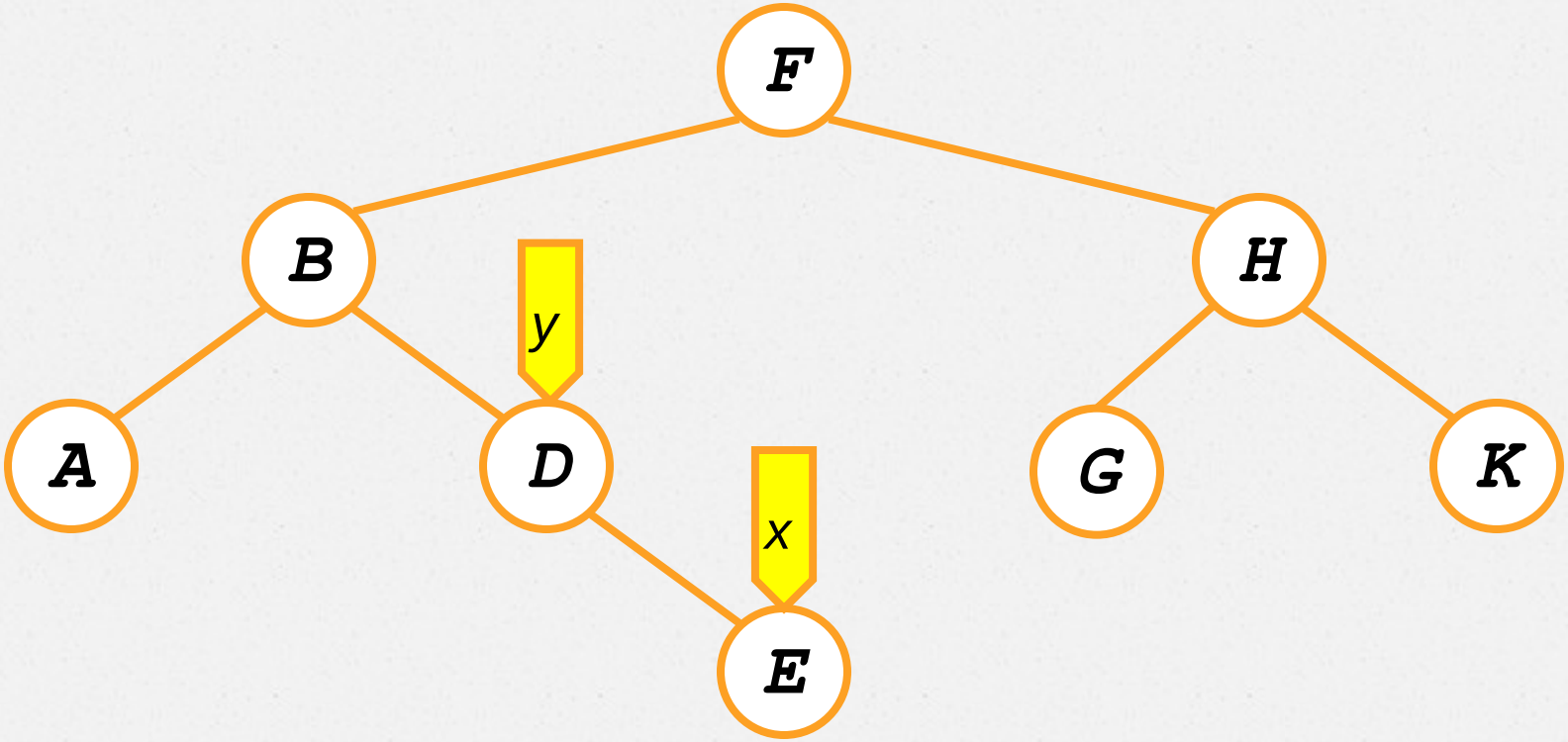
Busca em BSTs

```
TREE_SUCCESSOR
1  if right[x]≠ nil
2      return TREE_MINIMUM(right[x])
3  y ←p[x]
4 while y ≠ nil and x = right[y]
5        x←y
6        y←p[x]
7  return y
```

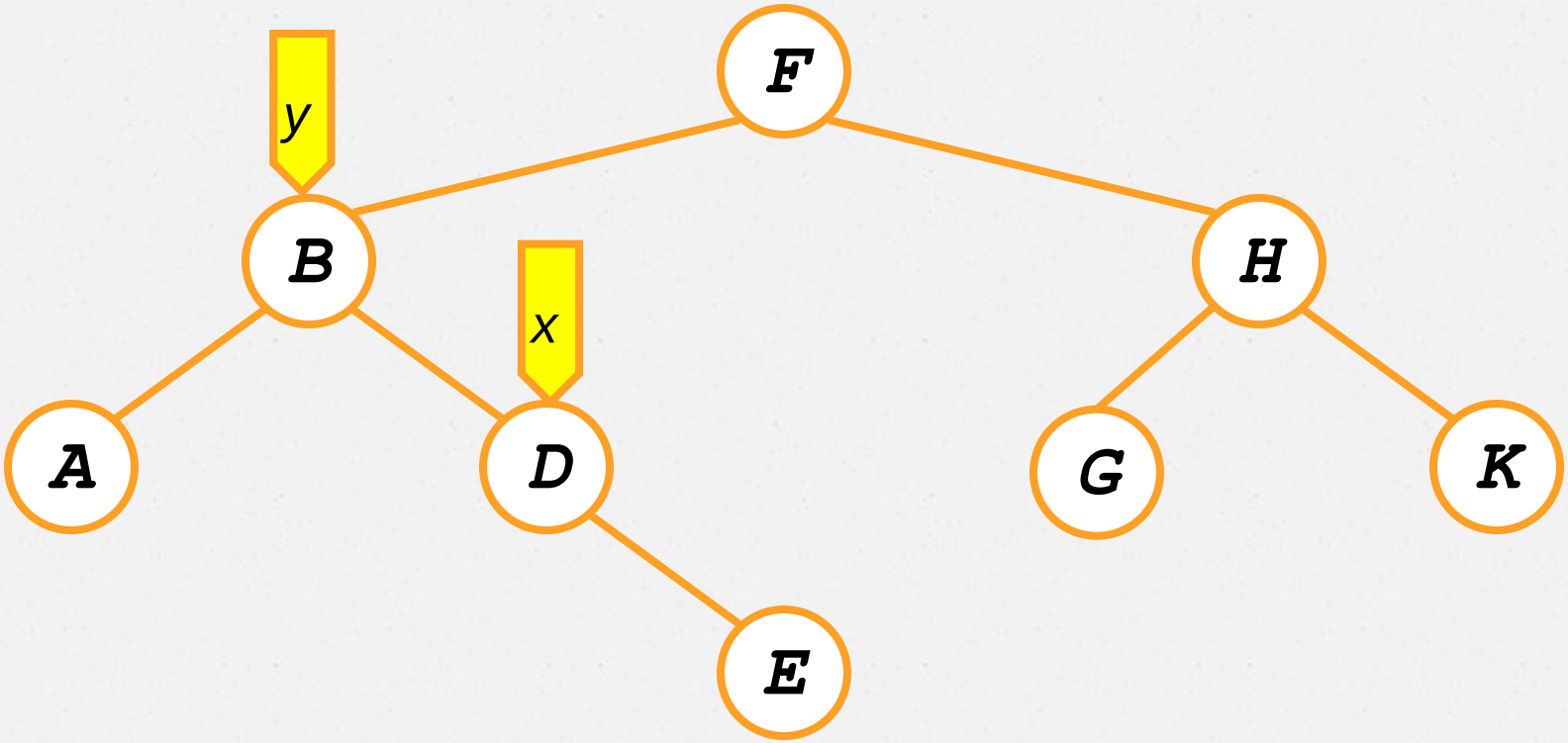
Busca em BSTs



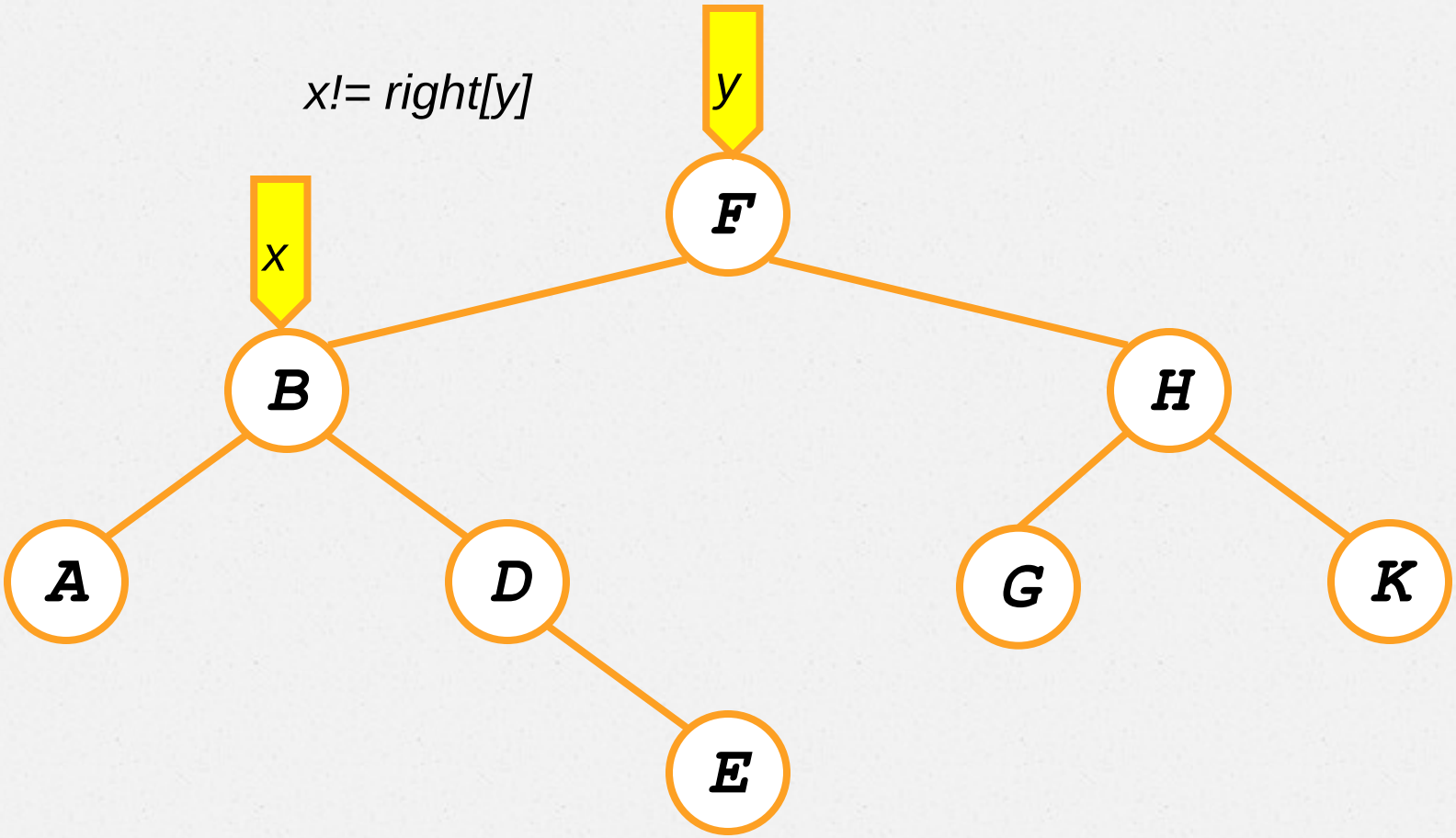
Busca em BSTs



Busca em BSTs



Busca em BSTs

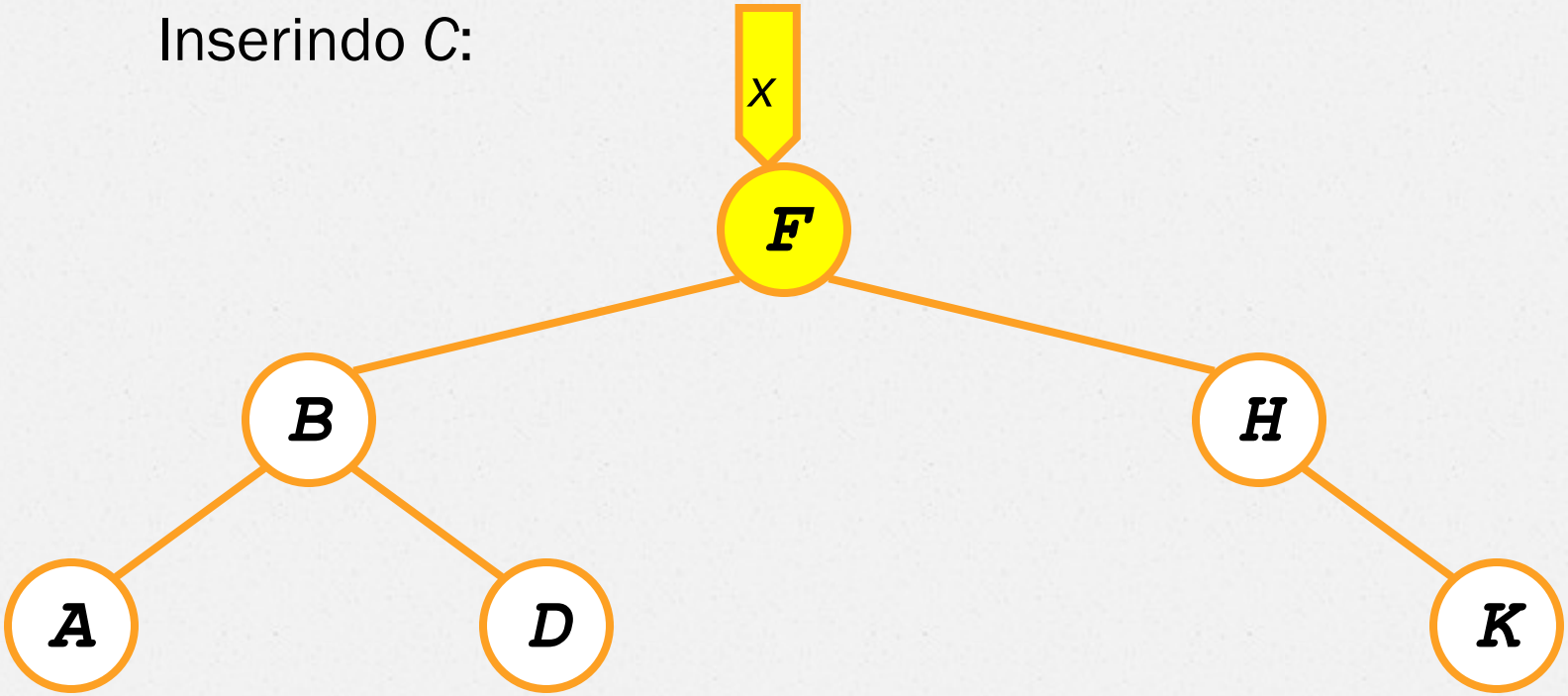


```
Tree-Insert(T, z)
1    y ← NIL
2    x ← root[T]
3    while x ≠ NIL
4                  y ← x
5       if key[z] < key[x]
6             x ← left[x]
7       else  x ← right[x]
8  p[z] ← y
9    if  y = NIL
10       root[T] ← z
11 else if key[z] < key[y]
12             left[y] ← z
13           else  right[y] ← z
```

```
Tree-Insert(T, z)
1    y ← NIL
2    x ← root[T]
3    while x ≠ NIL
4       y ← x
5       if key[z] < key[x]
6             x ← left[x]
7       else  x ← right[x]
8  p[z] ← y
9    if  y = NIL
10       root[T] ← z
11 else if key[z] < key[y]
12            left[y] ← z
13          else  right[y] ← z
```

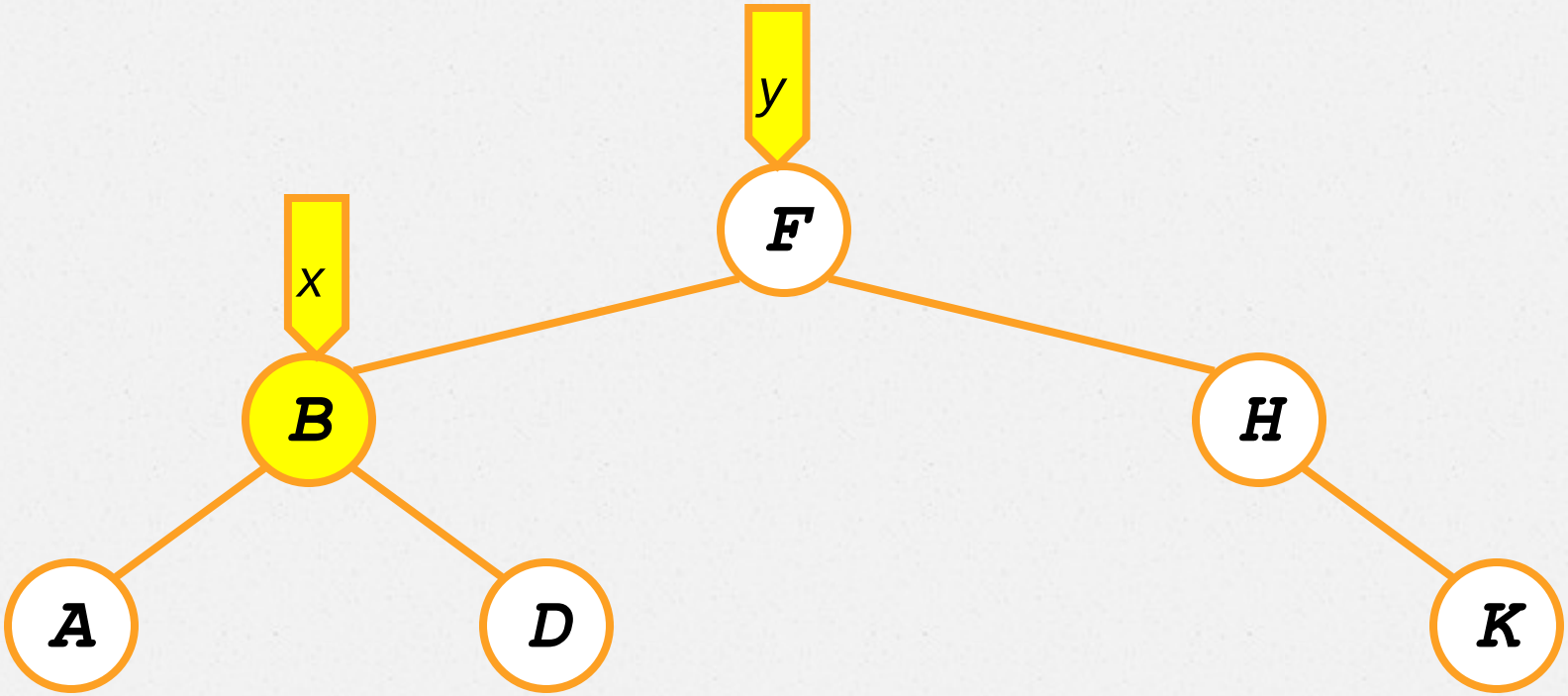
Inserção em BSTs

Inserindo *C*:



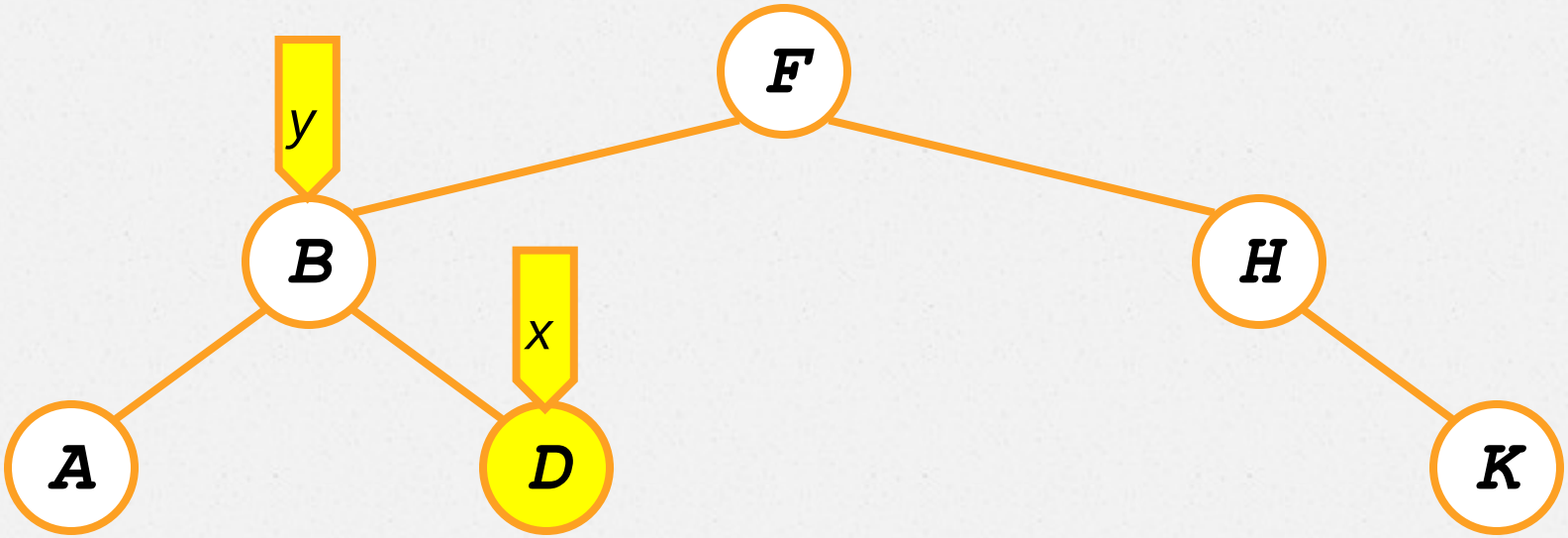
Inserção em BSTs

Inserindo *C*:



Inserção em BSTs

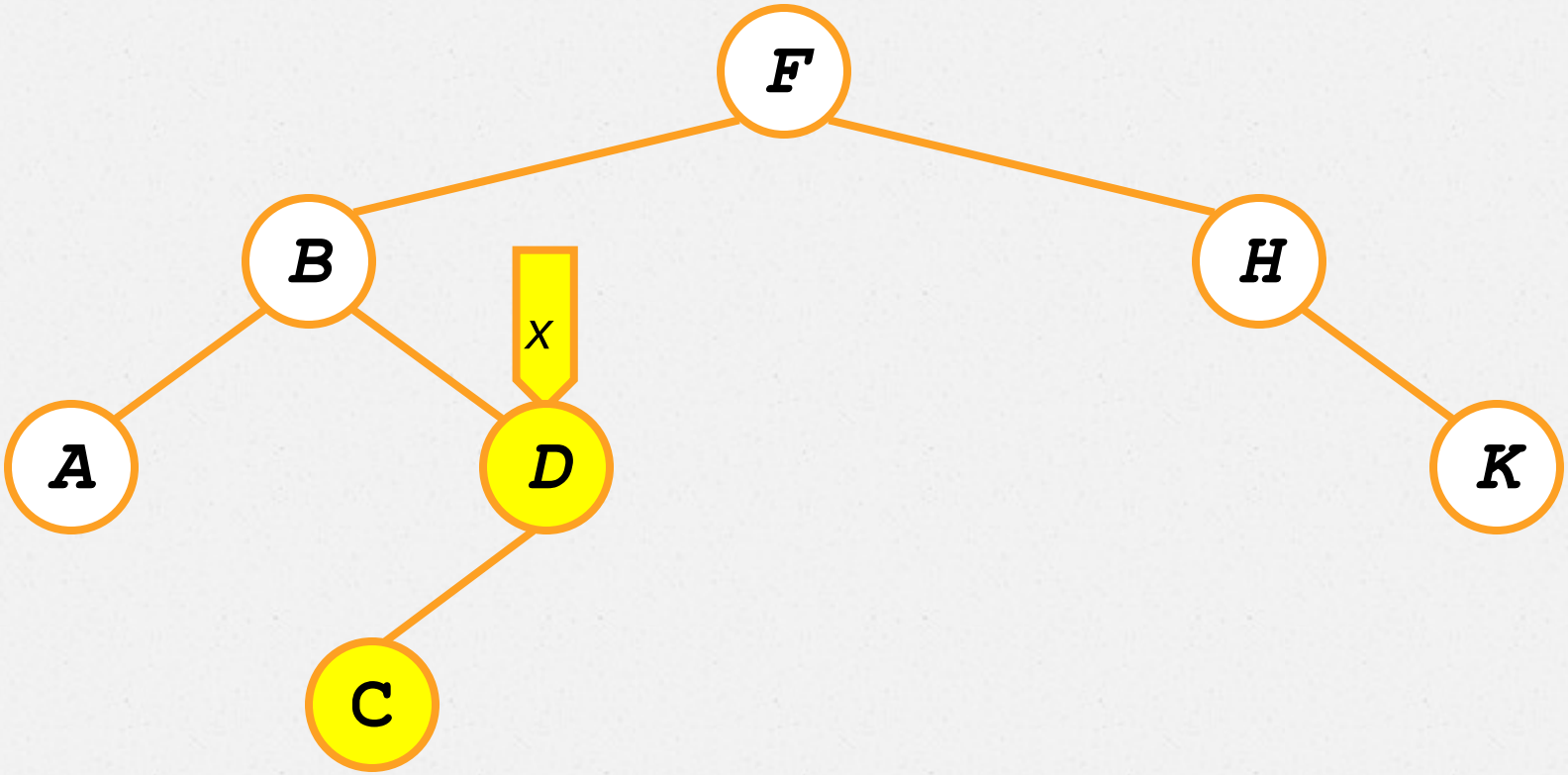
Inserindo *C*:



```
Tree-Insert(T, z)
1    y ← NIL
2    x ← root[T]
3    while x ≠ NIL
4       y ← x
5       if key[z] < key[x]
6             x ← left[x]
7       else  x ← right[x]
8  p[z] ← y
9    if  y = NIL
10       root[T] ← z
11 else if key[z] < key[y]
12            left[y] ← z
13            else  right[y] ← z
```

Busca em BSTs

Inserindo C:



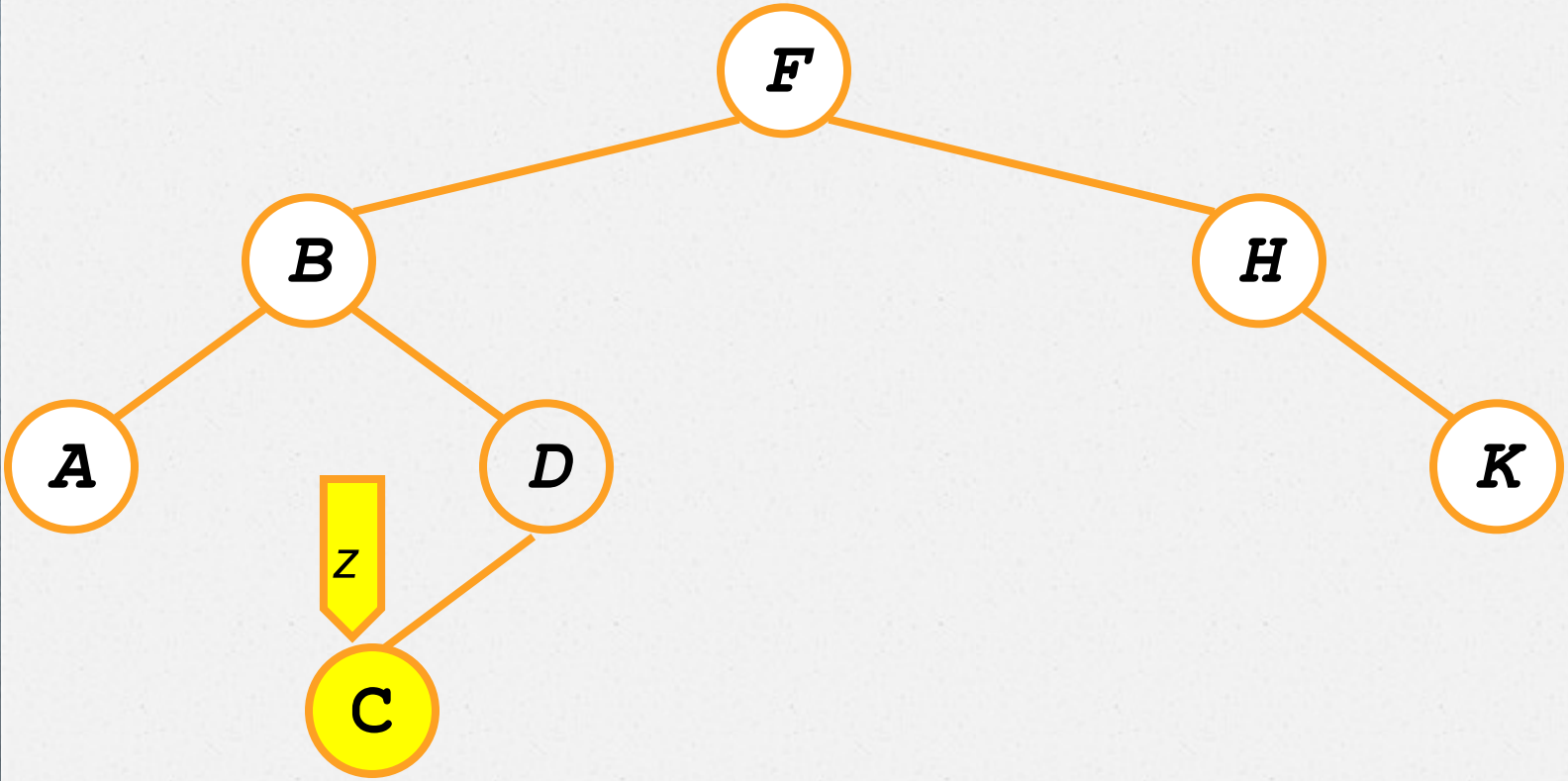
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

Busca em BSTs

Removendo C - Não há nós filhos



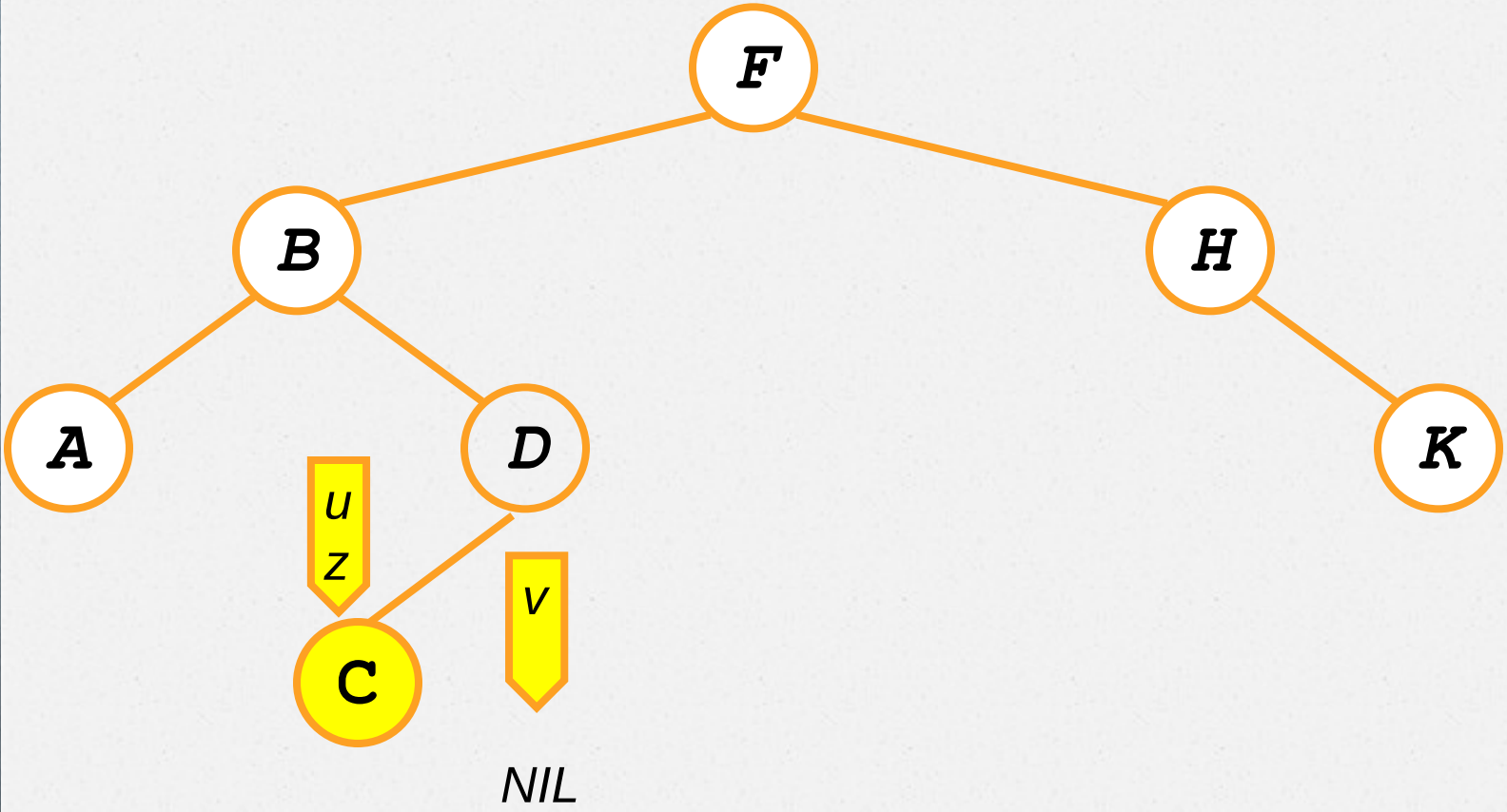
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

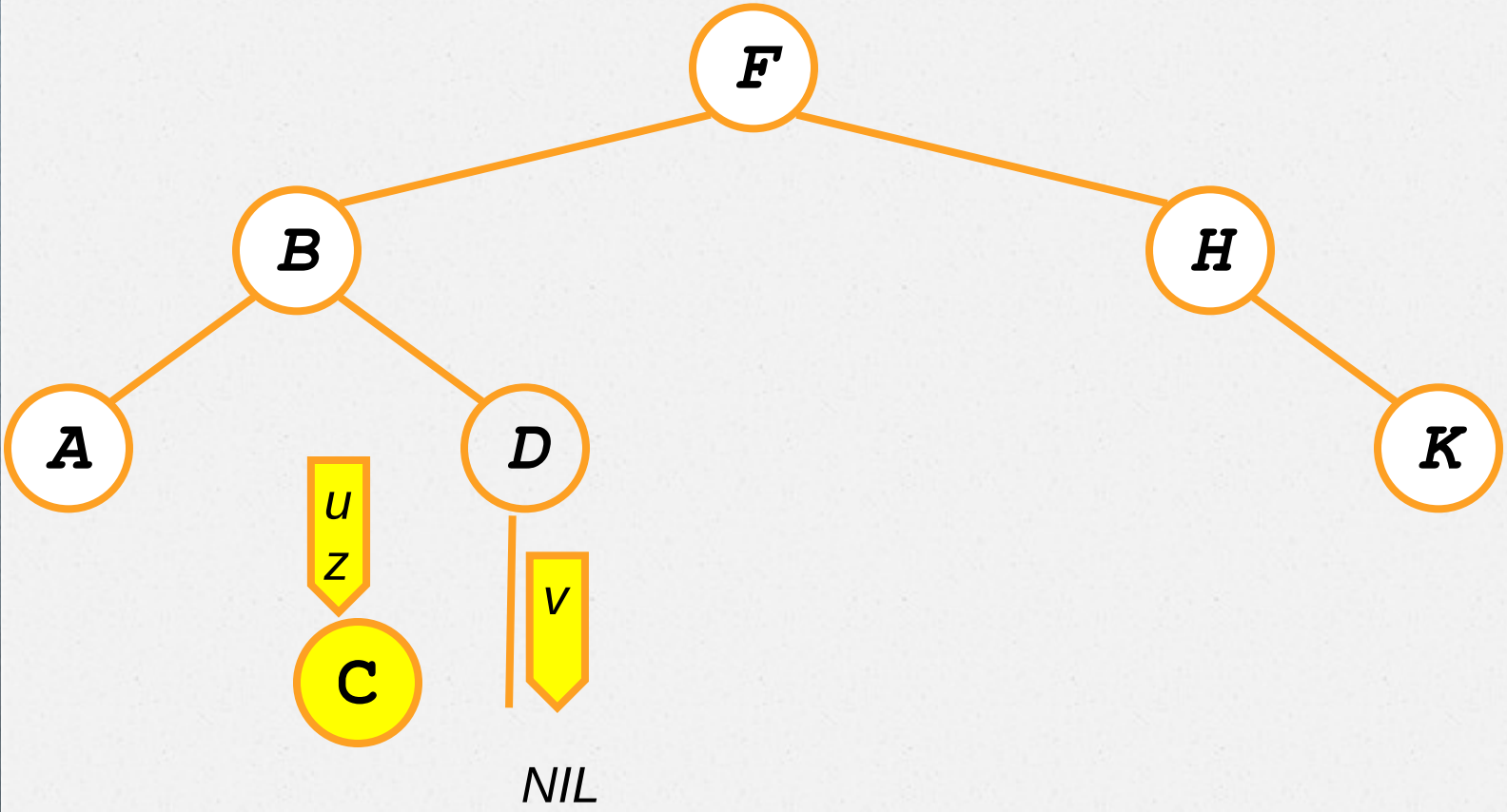
Busca em BSTs

Removendo C - Não há nós filhos



Busca em BSTs

Removendo C - Não há nós filhos



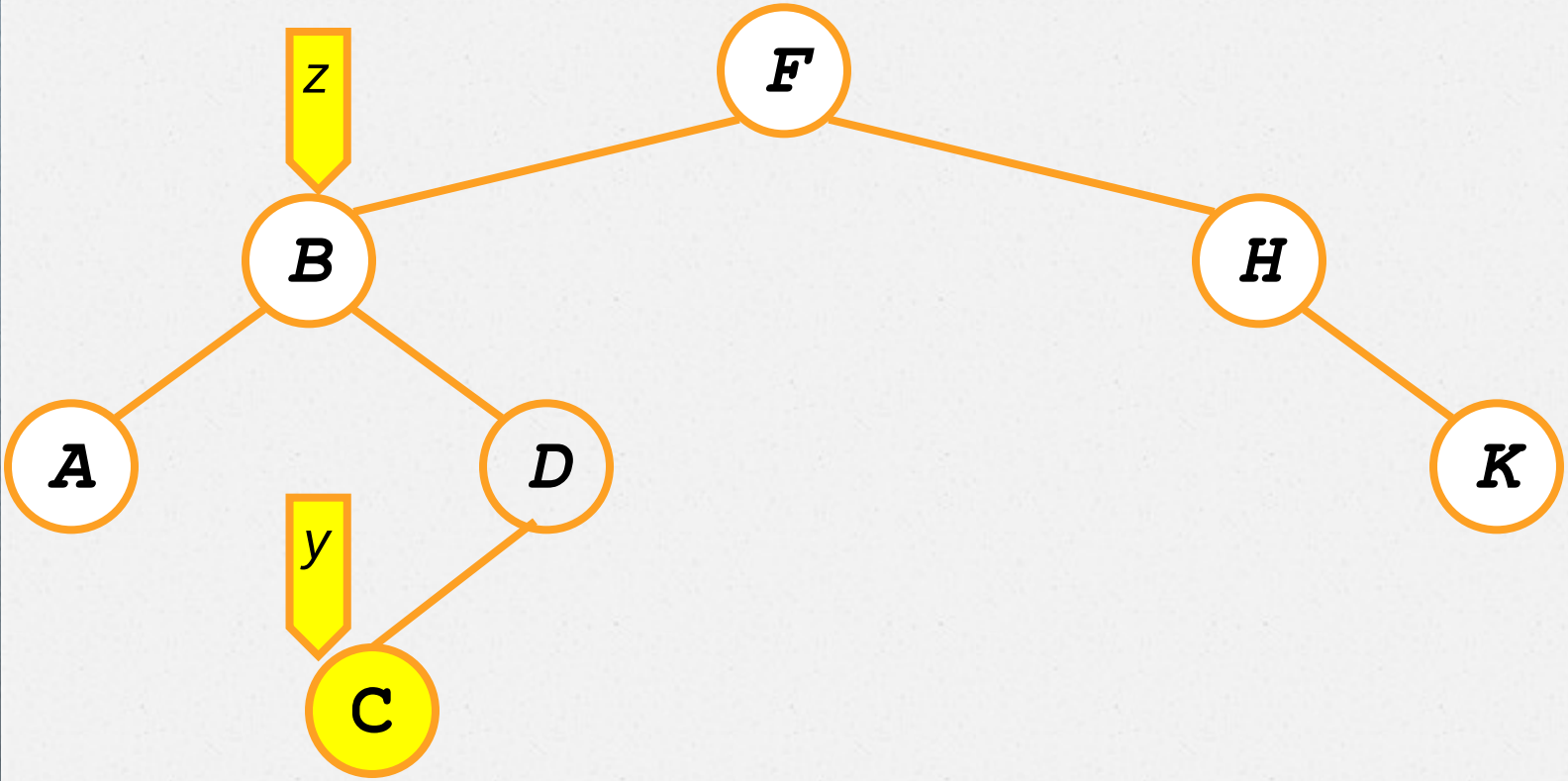
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

Busca em BSTs

Removendo B - Não há nós filhos



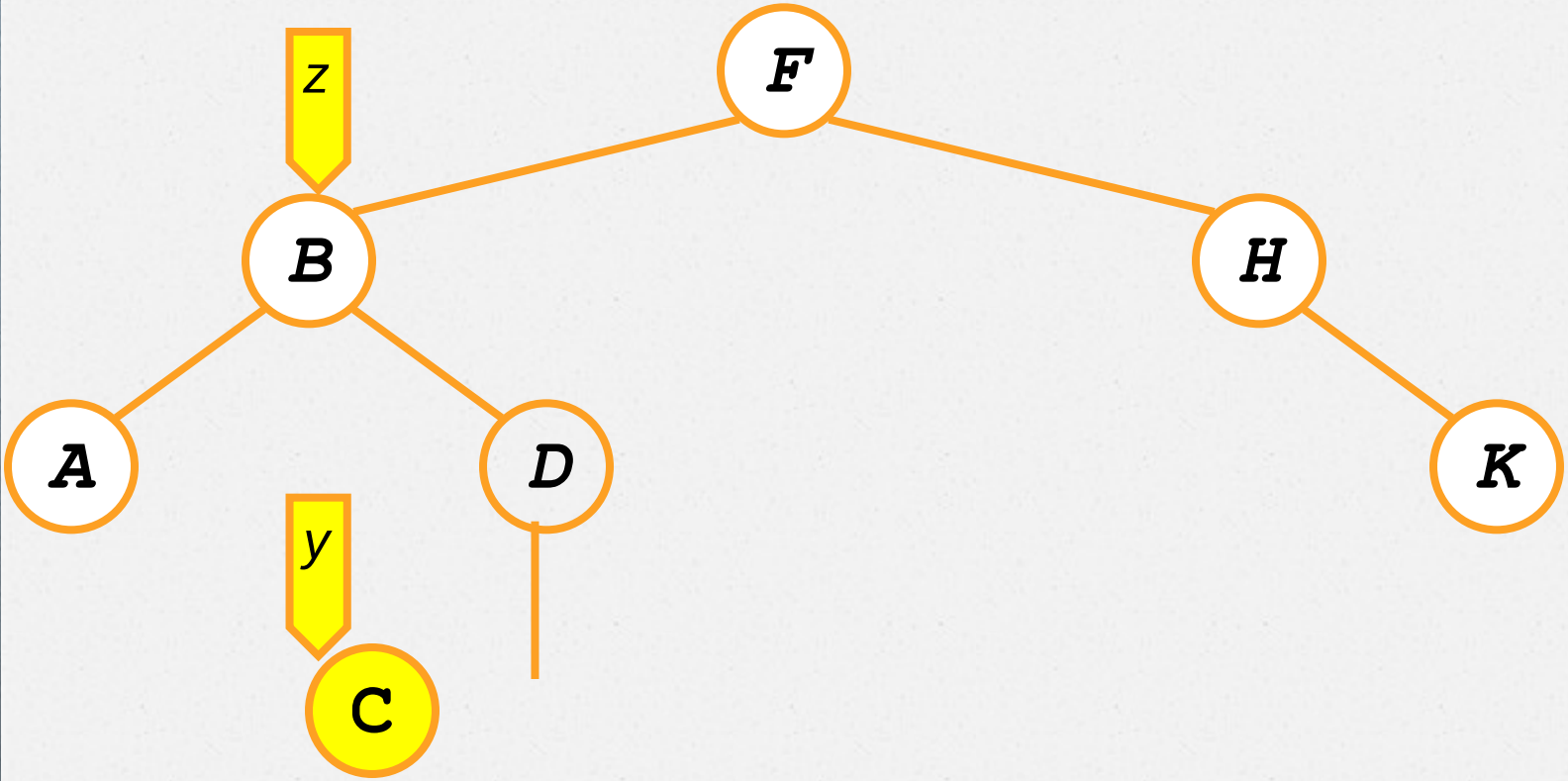
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

Busca em BSTs

Removendo C - Não há nós filhos



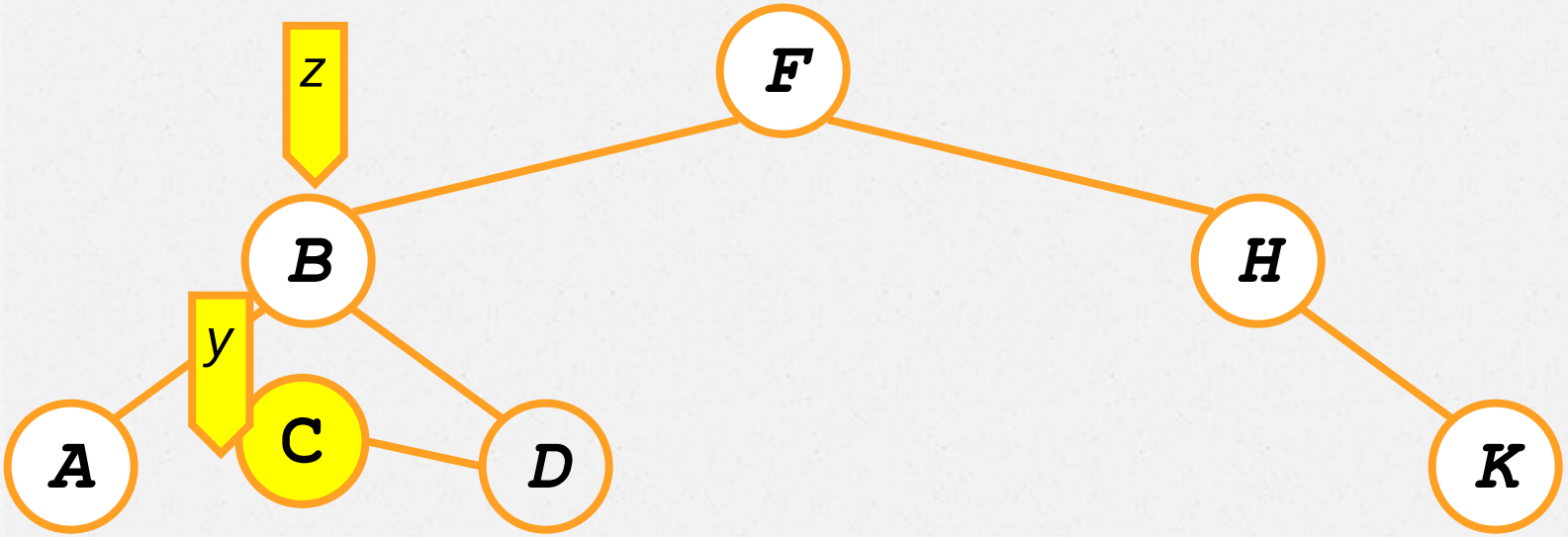
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

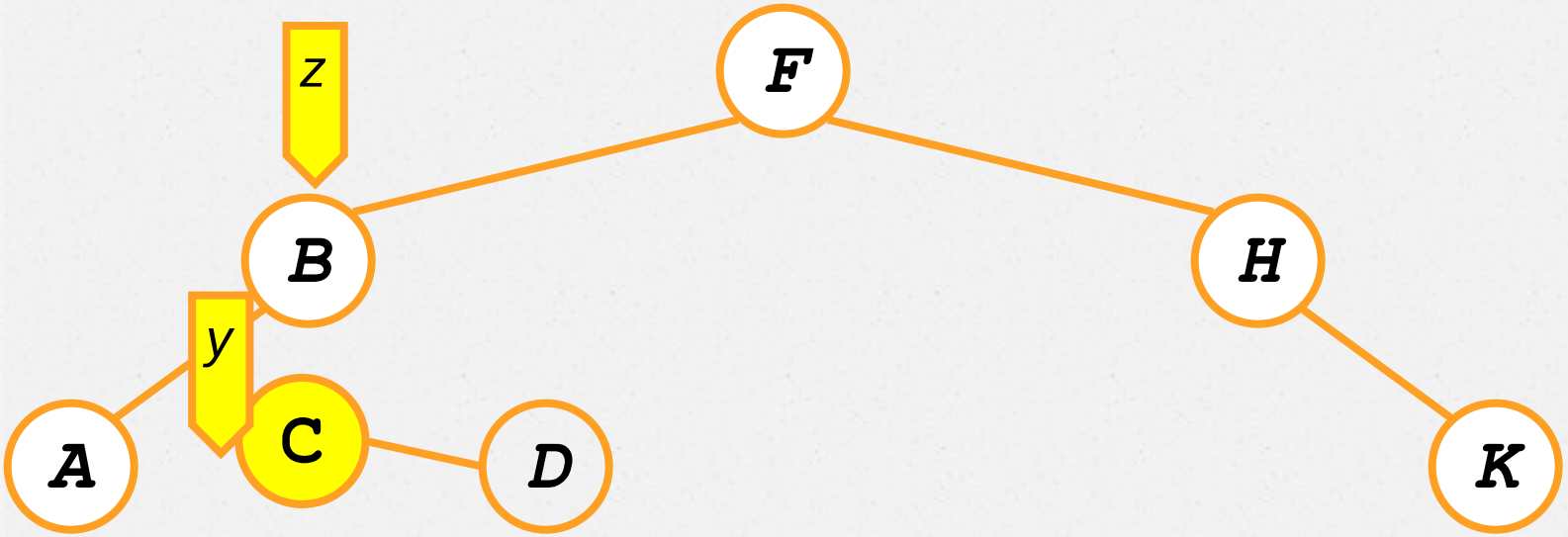
Busca em BSTs

Removendo C - Não há nós filhos



Busca em BSTs

Removendo C - Não há nós filhos



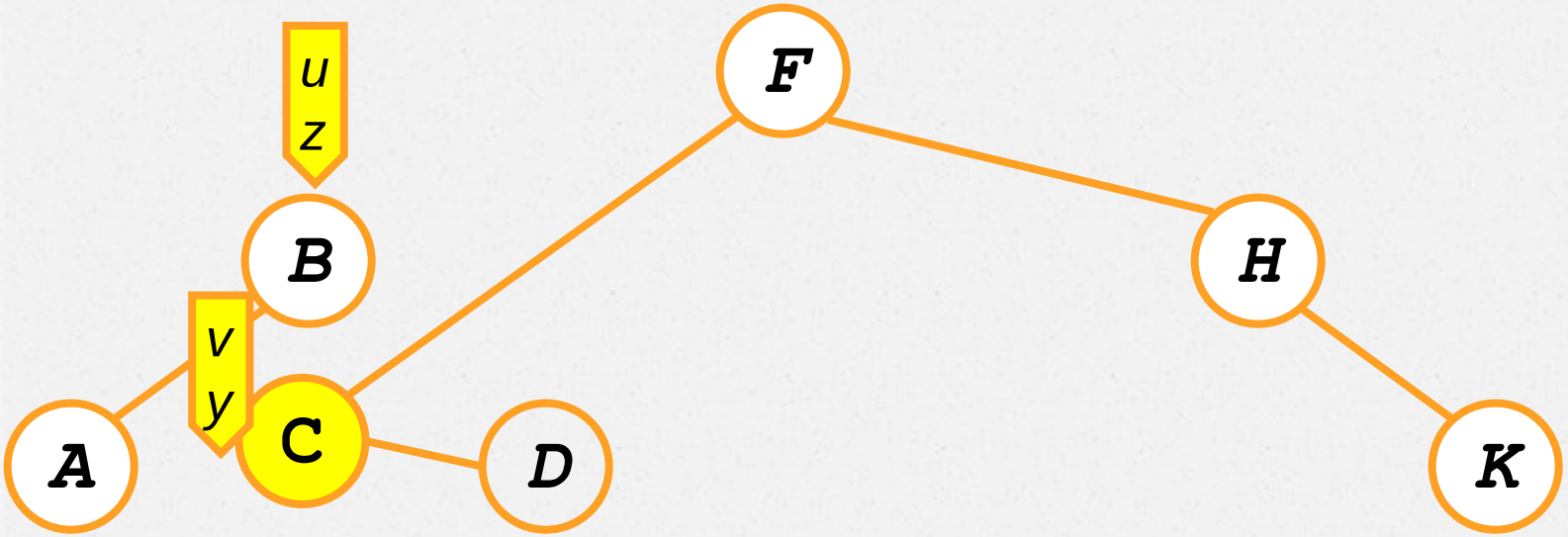
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.          left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

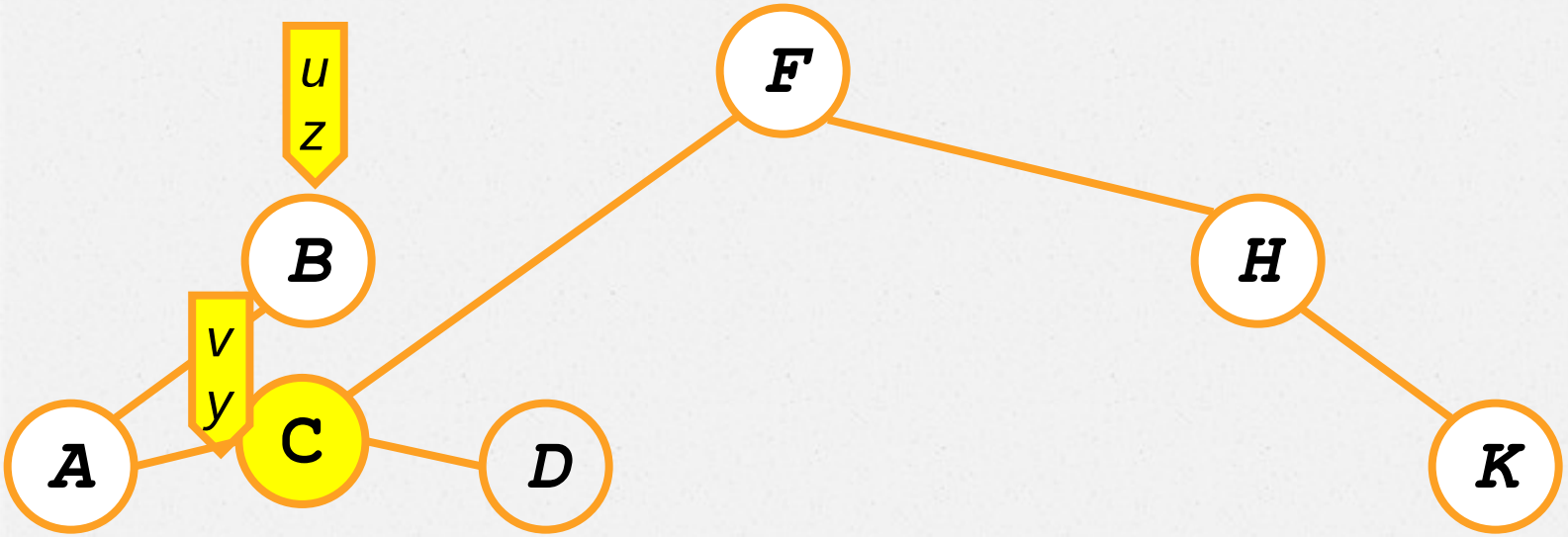
Busca em BSTs

Removendo C - Não há nós filhos



Busca em BSTs

Removendo C - Não há nós filhos



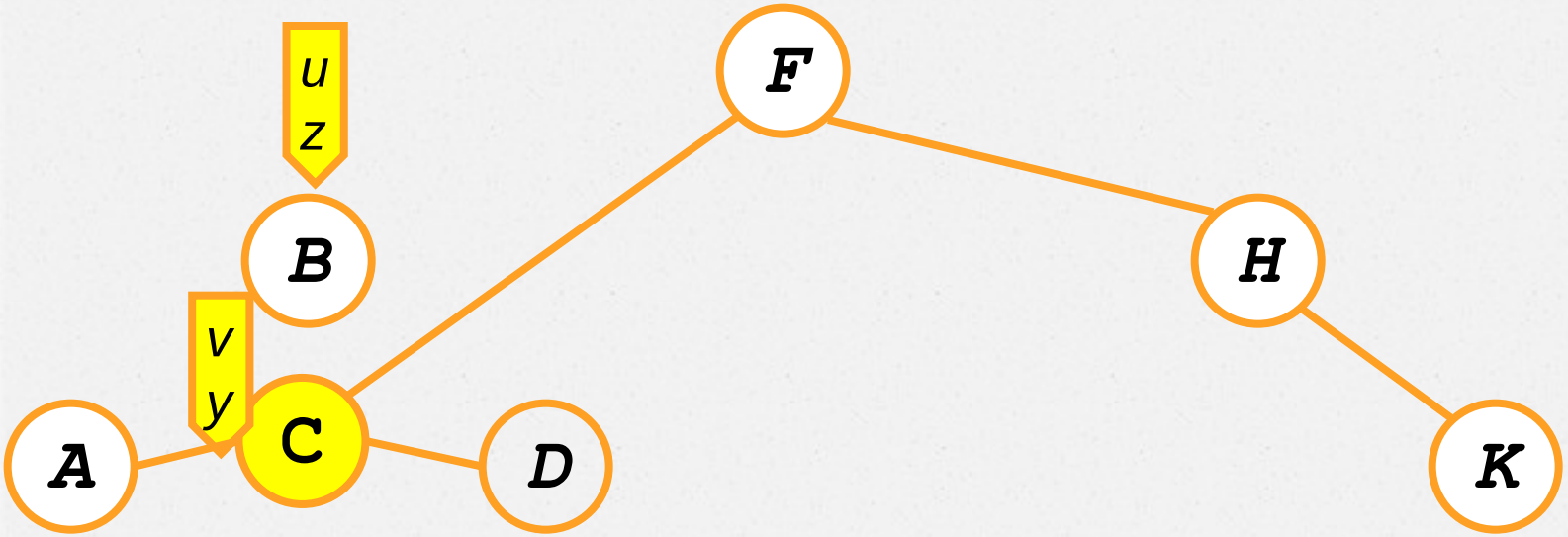
Busca em BSTs

```
TREE-DELETE(T,z)
1.  if left[z] == NIL
2.    TRANSPLANT(T,z,right[z])
3.  elseif right[z]== NIL
4.    TRANSPLANT(T,z,left[z])
5.  else
6.    y=TREE-MINIMUM(right[z])
7.    if p[y]≠z
8.      TRANSPLANT(T,y,right[y])
9.      right[y]=right[z]
10.     p[right[y]]=y
11.   TRANSPLANT(T,z,y)
12.   left[y]=left[z]
13.   p[left[y]]=y
```

```
TRANSPLANT(T,u,v)
1.  if p[u] = NIL
2.      root[T] = v
3.  elseif u==left[p[u]]
4.         left[p[u]]=v
5.  else  right[p[u]]=v
6.  if  v ≠ NIL
7.      p[v]=p[u]
```

Busca em BSTs

Removendo C - Não há nós filhos



Analisando a Busca em BST

- O(h), onde h = altura da árvore$\Rightarrow$ O(lg n)
- Pior caso: O(n) ocorre quando a árvore é apenas um vetor linear composto pelos filhos à esquerda ou à direita.

Ordenando com BSTs

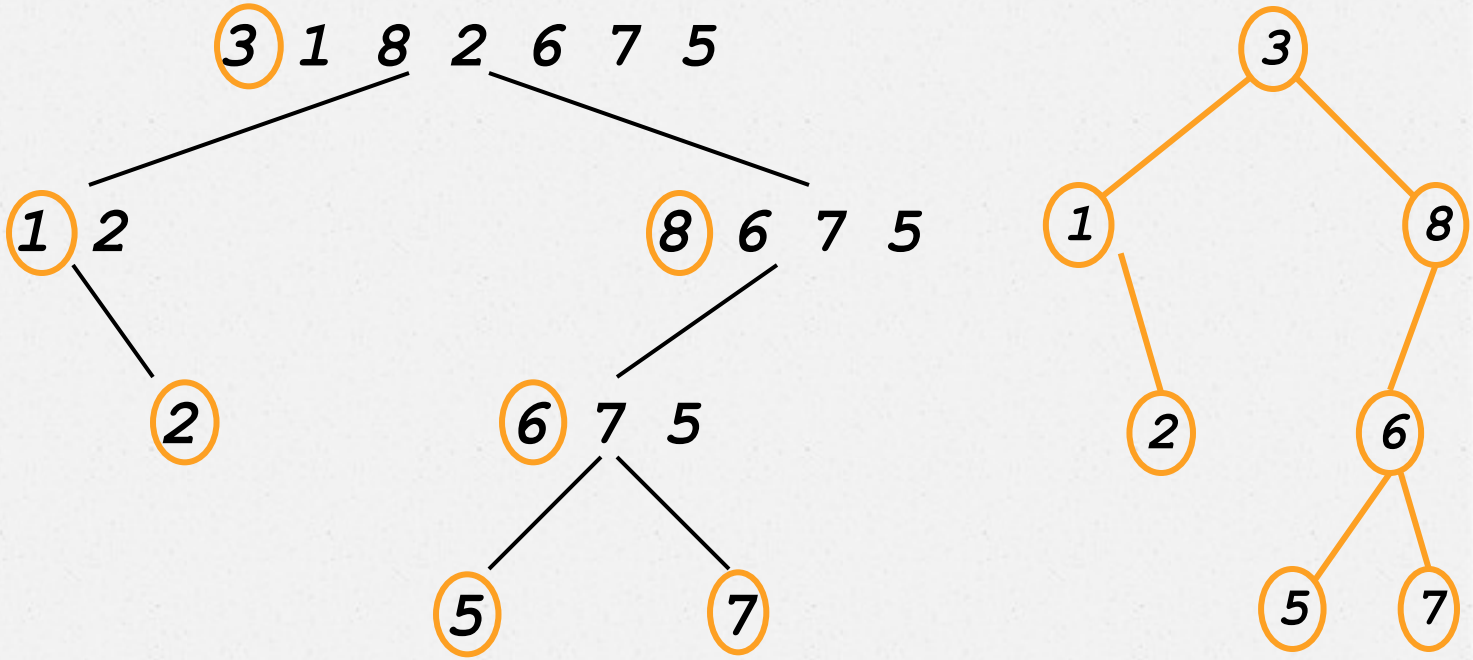
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

- $\Omega(n \lg n)$
- Qual o tempo de execução no pior caso? Caso médio?

Ordenando com BSTs

Caso médio

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```



Ordenando com BSTs

- Particiona como quicksort, mas em ordem diferente.
- Exemplo anterior:
  - ✓Todos os elementos são comparados uma vez ao elemento 3.
  - ✓Aqueles elementos < 3 são comparados ao elemento 1 uma vez.
  - ✓O procedimento acima se repete
- O tempo de execução será propocional ao número de comparações como quicksort:O(n lg n)

Ordenando com BSTs

Quicksort x BST: qual utilizar?

- A: quicksort
- Constantes melhores
- In-place sorting
- Não precisa contruir uma estrutura de dados.