Exam review

CS 261 Lab #10

Oregon State University OSU

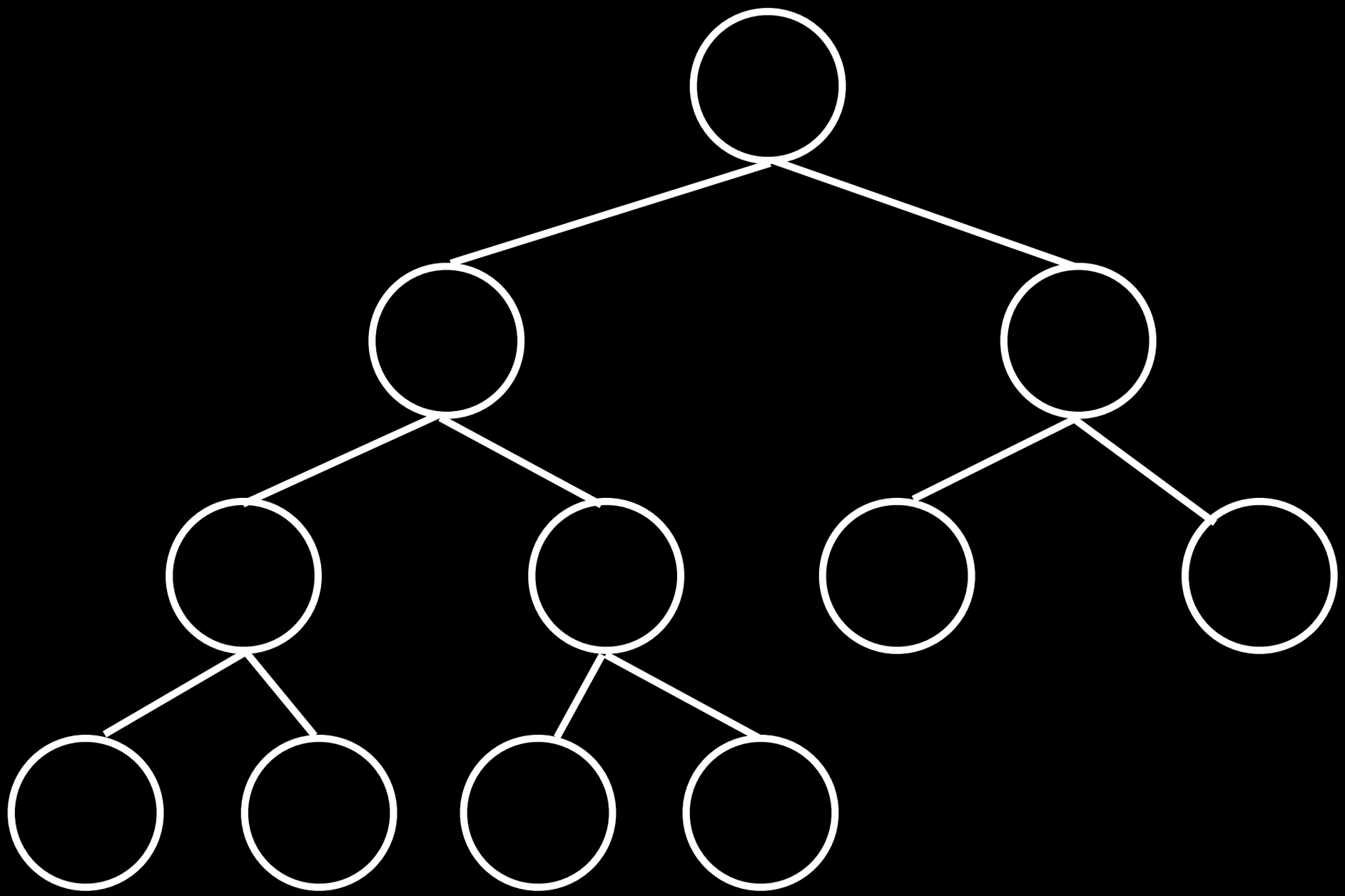
The exam will be **comprehensive**
(so don't forget to review the midterm slides!)

Similar style as midterm
(multiple choice, matching, true/false, code)

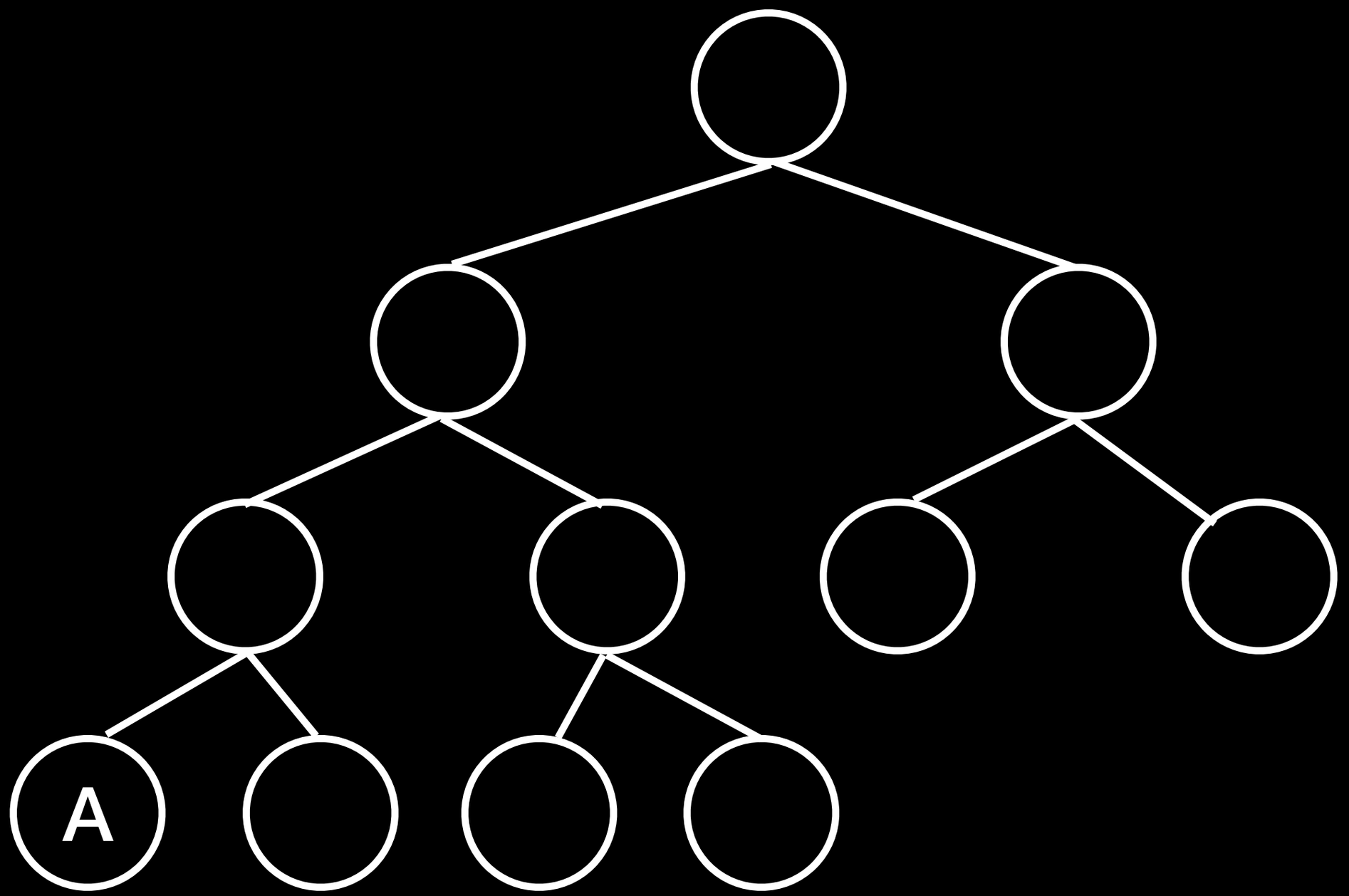
This review will focus on **trees, heaps,**
hash tables, and **graphs**

Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K

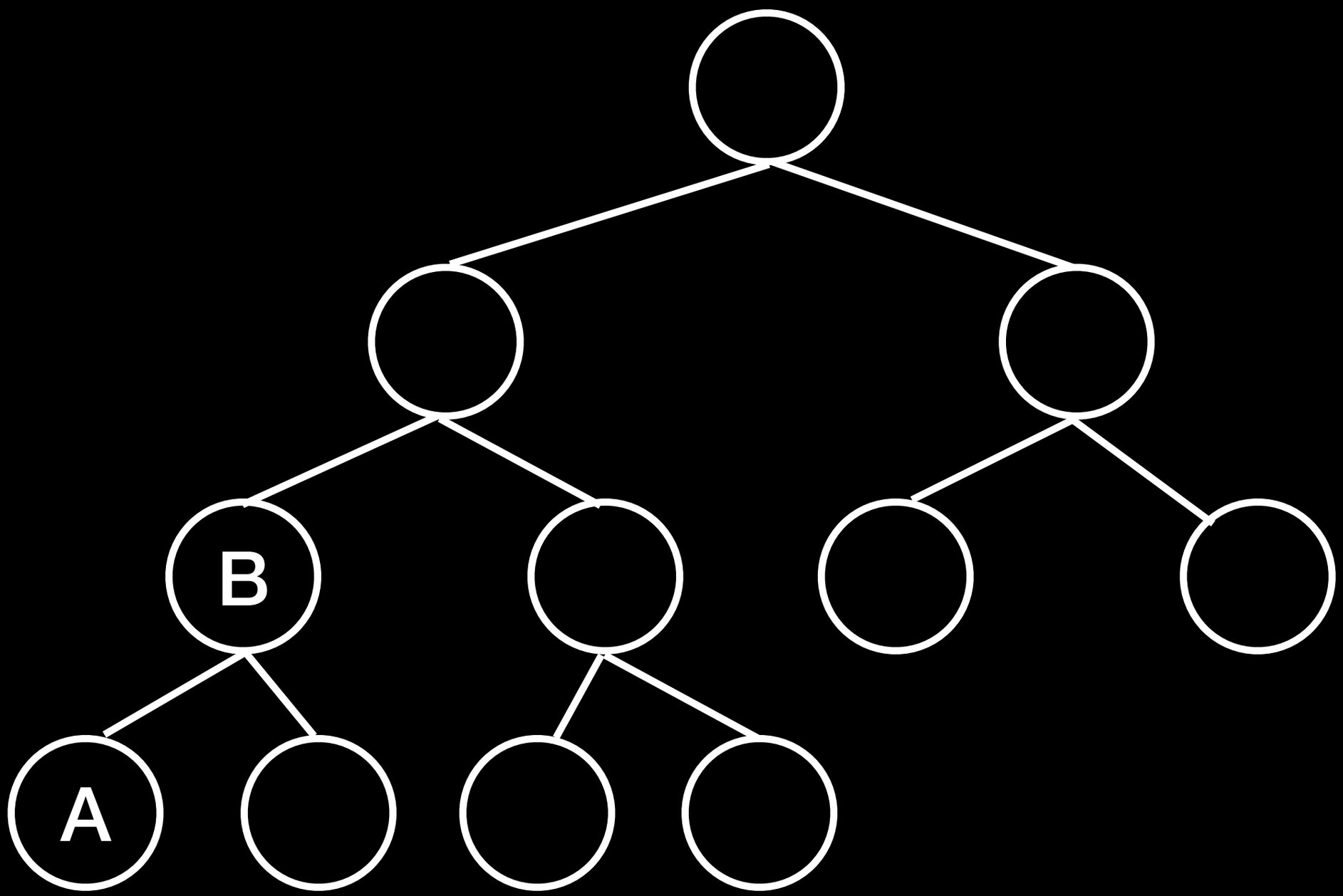
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



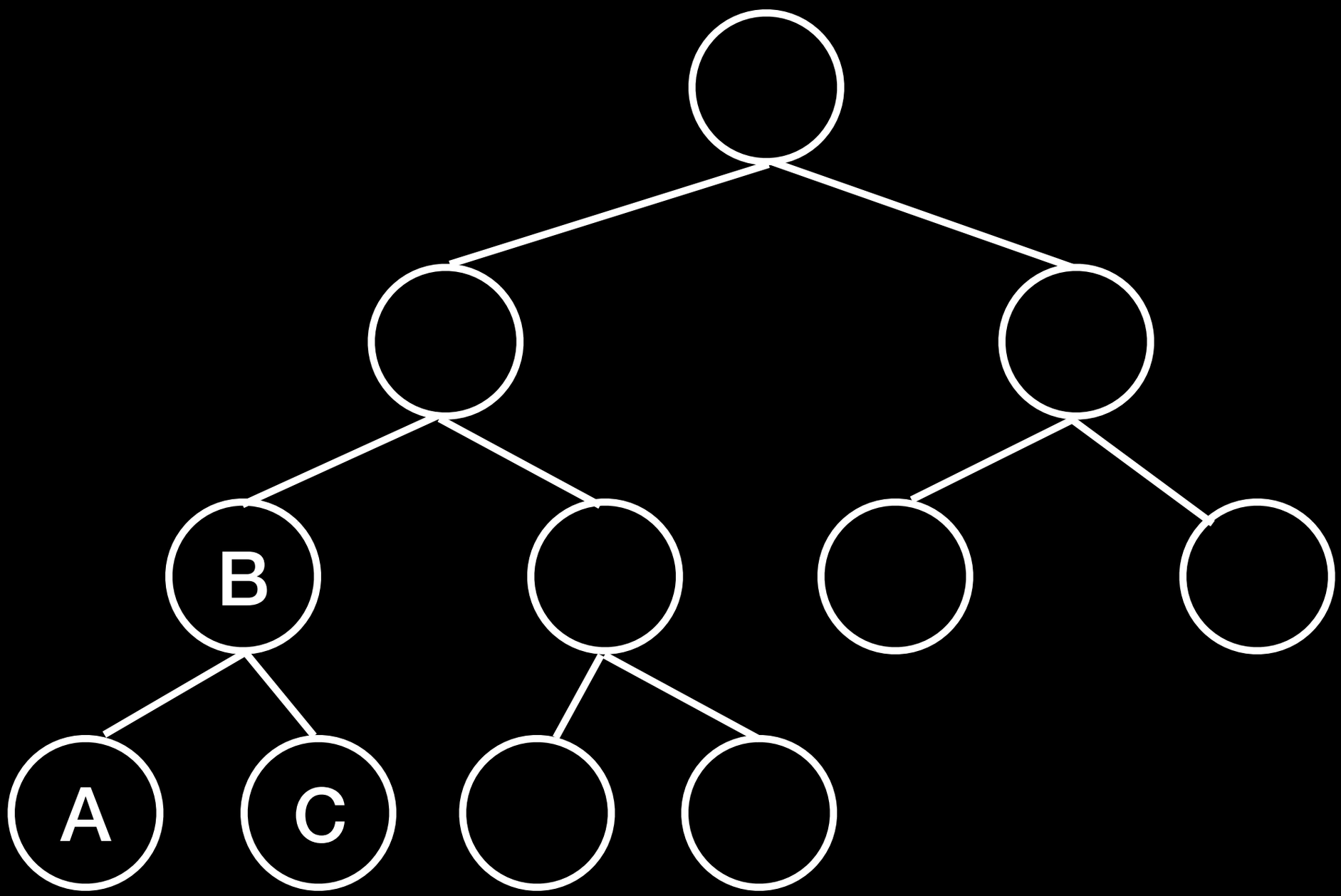
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



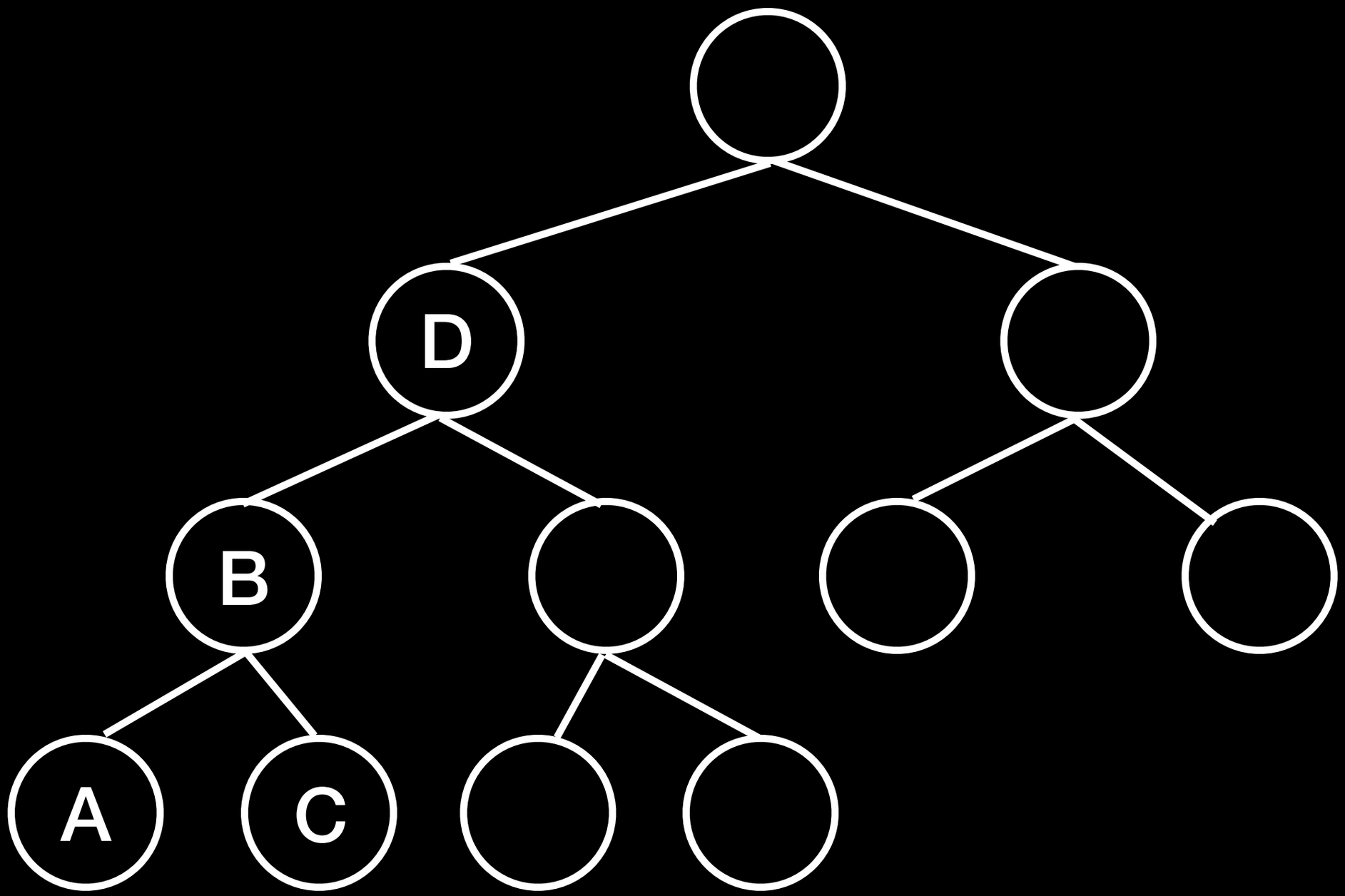
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



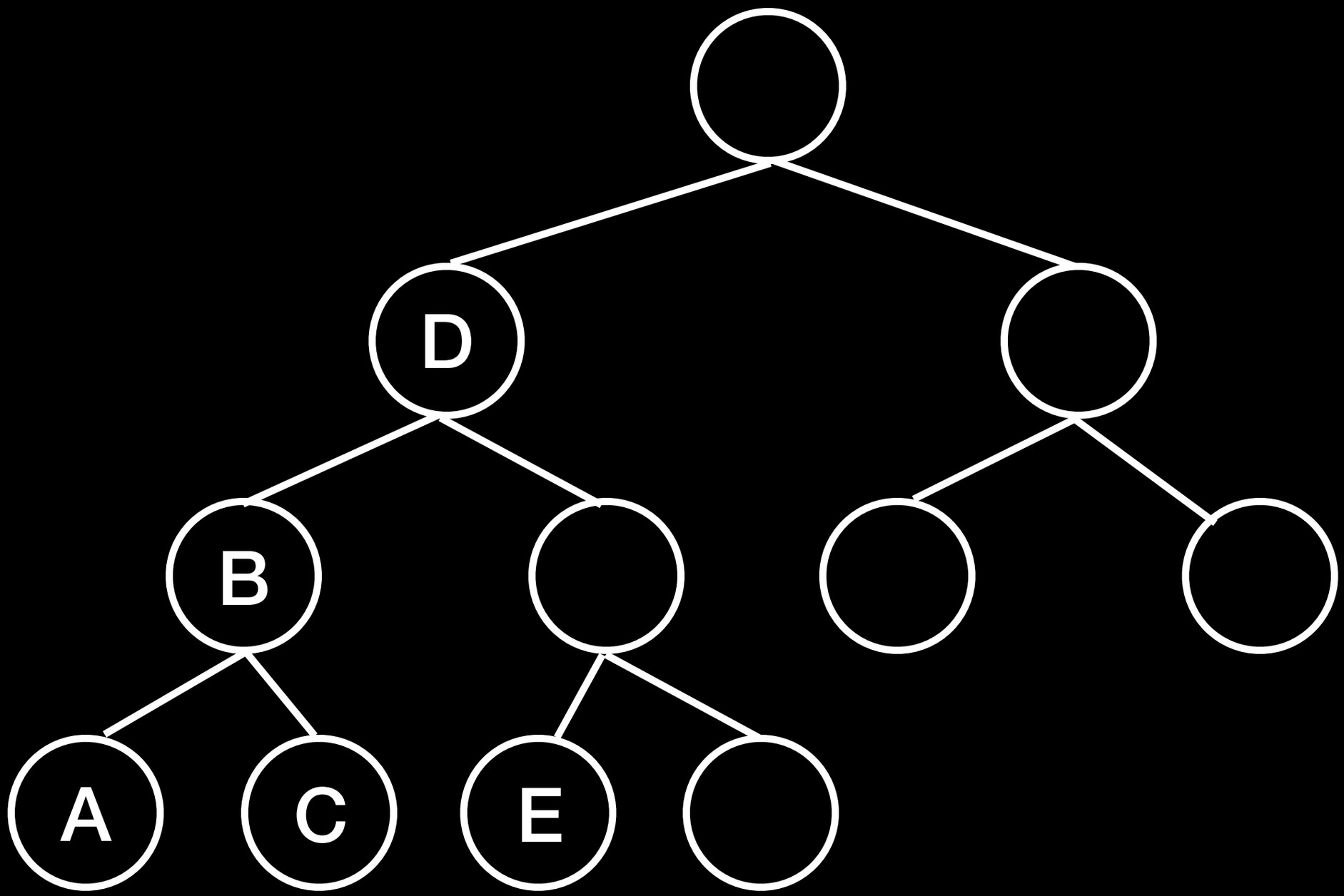
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



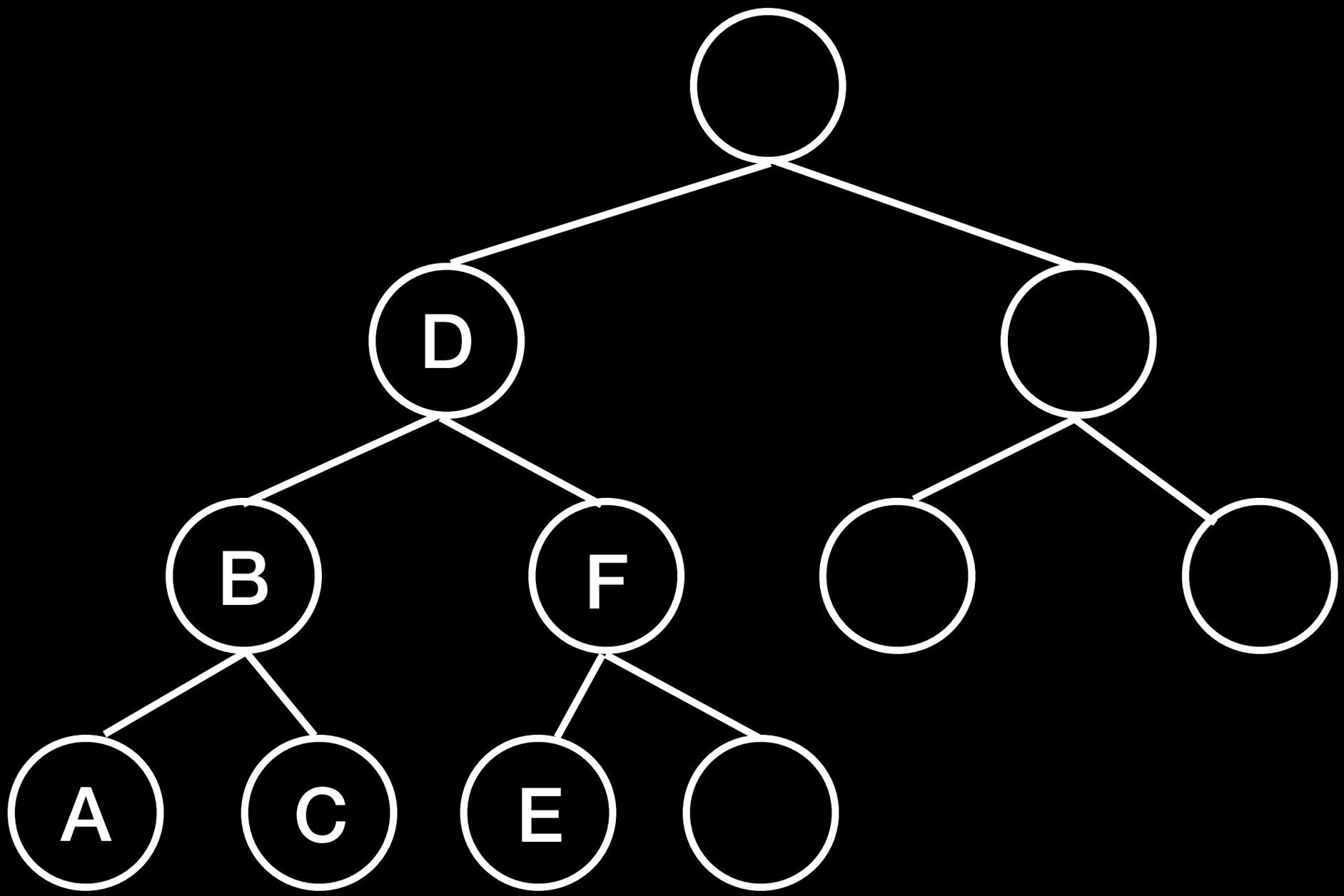
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



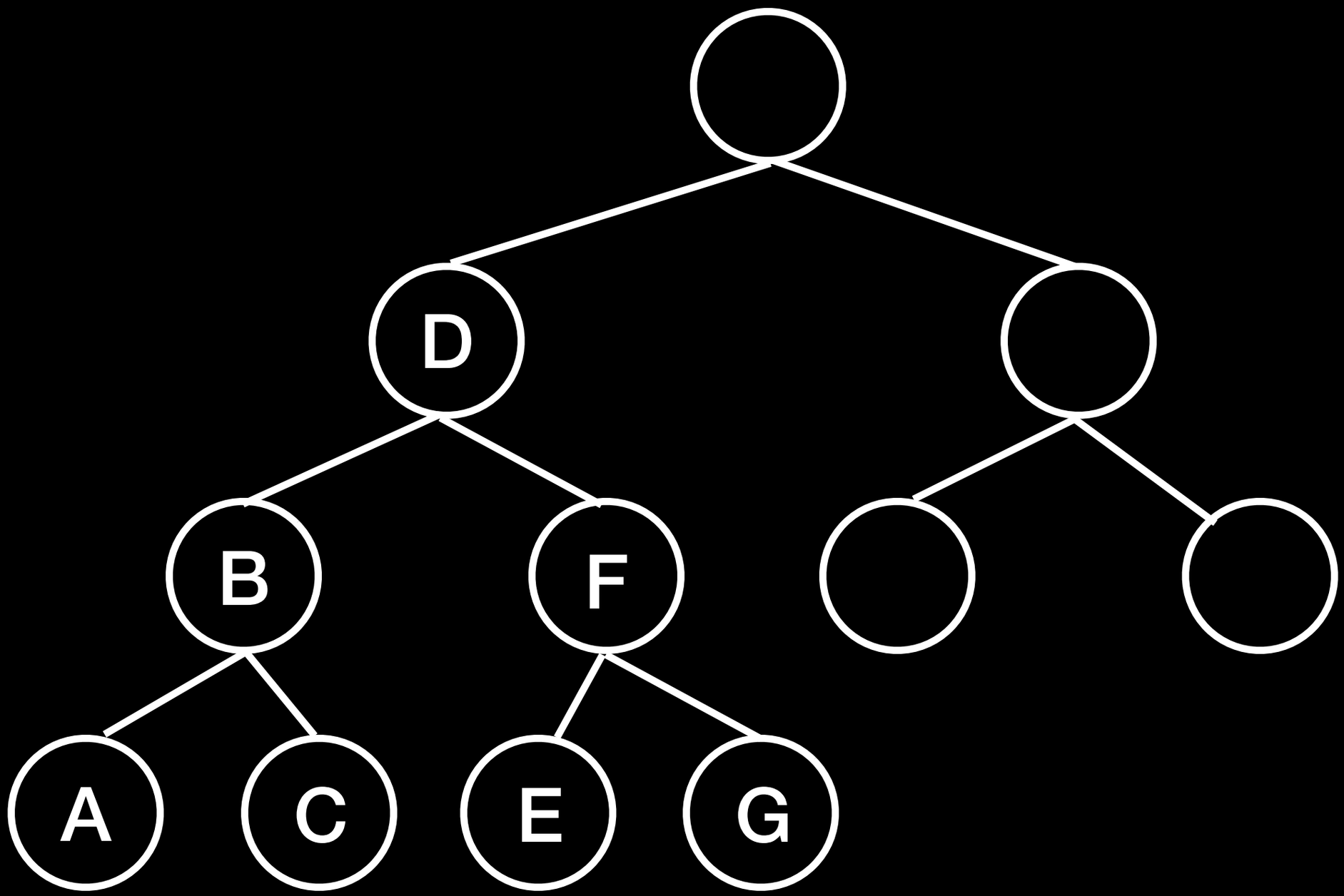
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



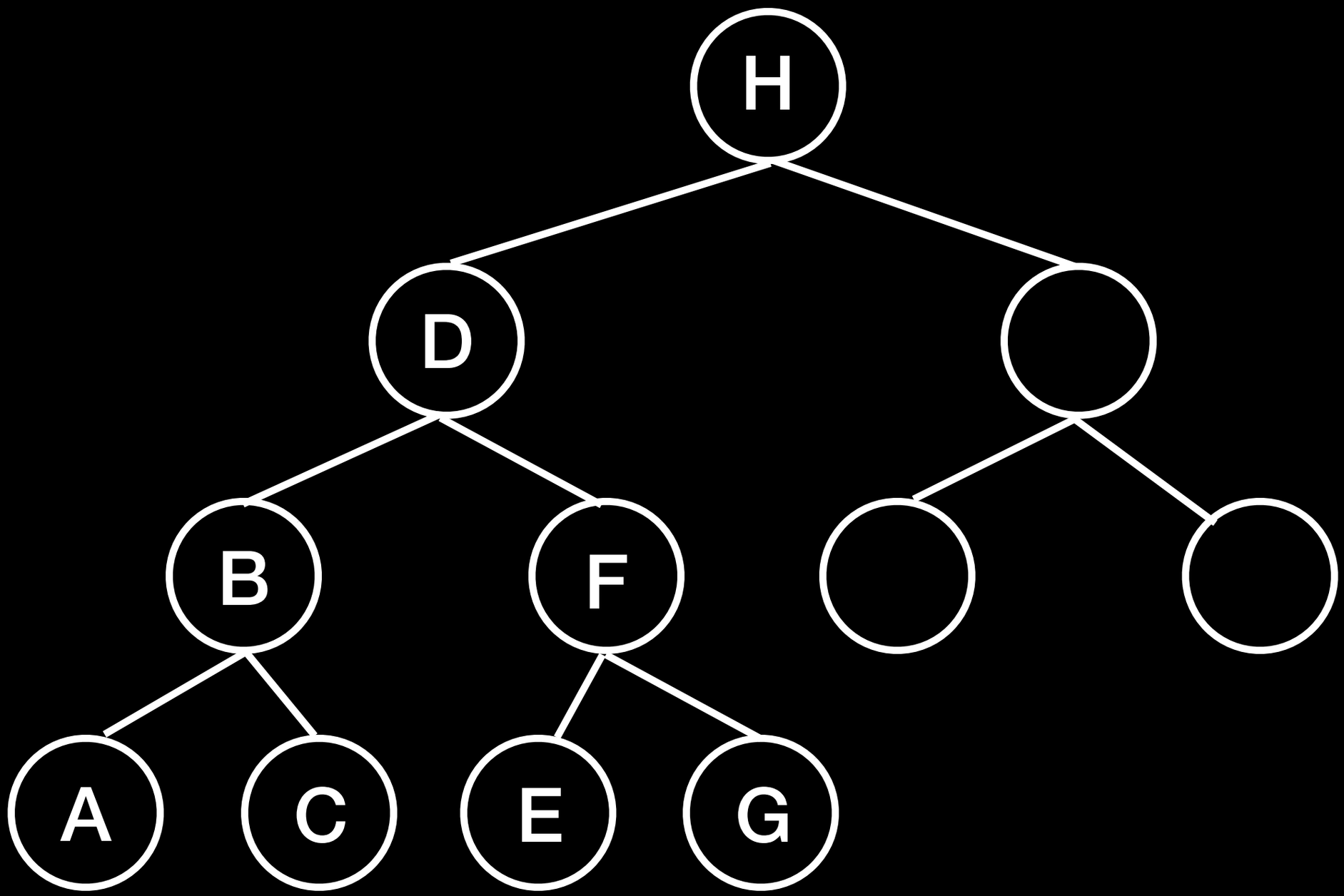
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



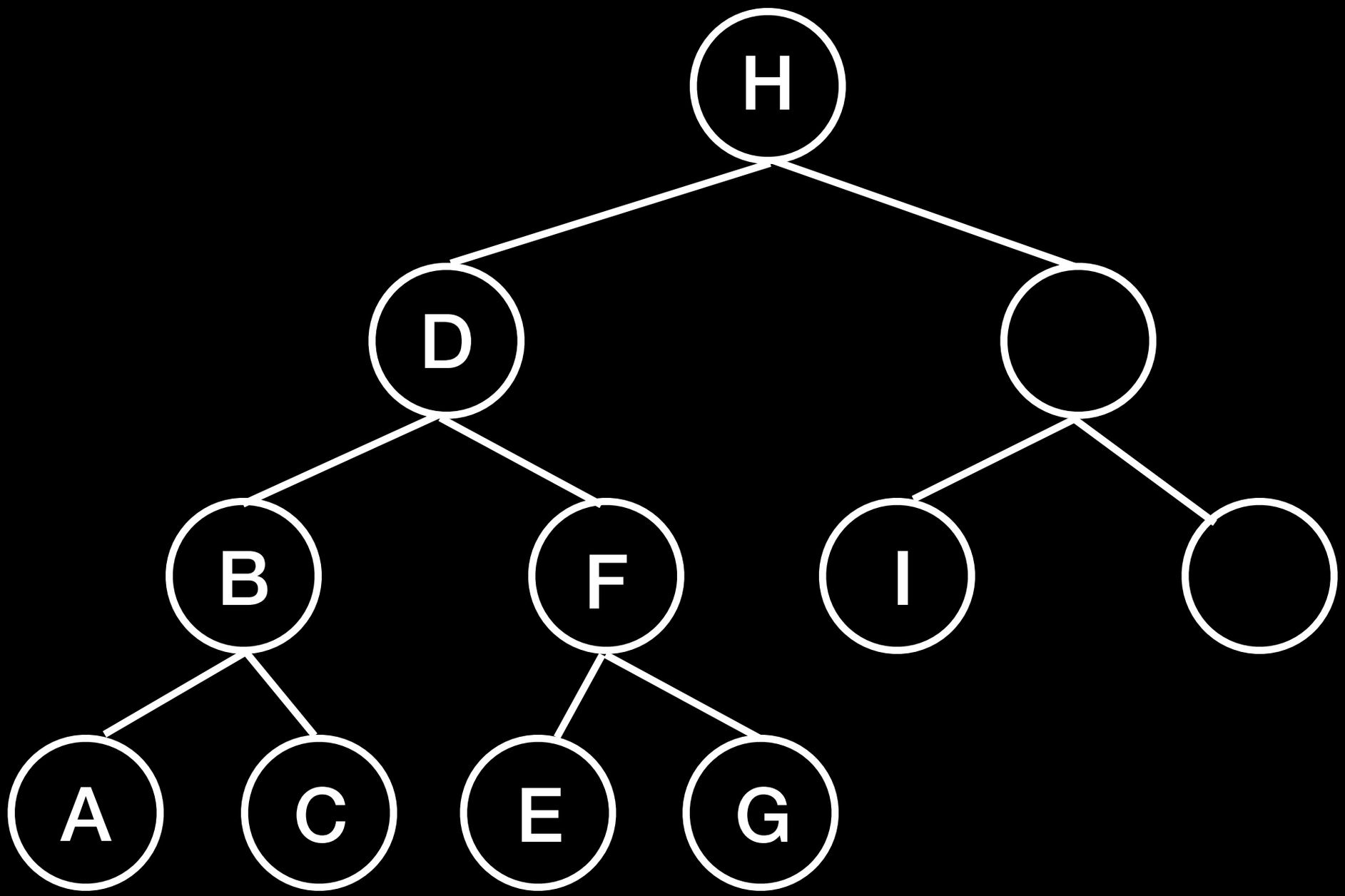
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



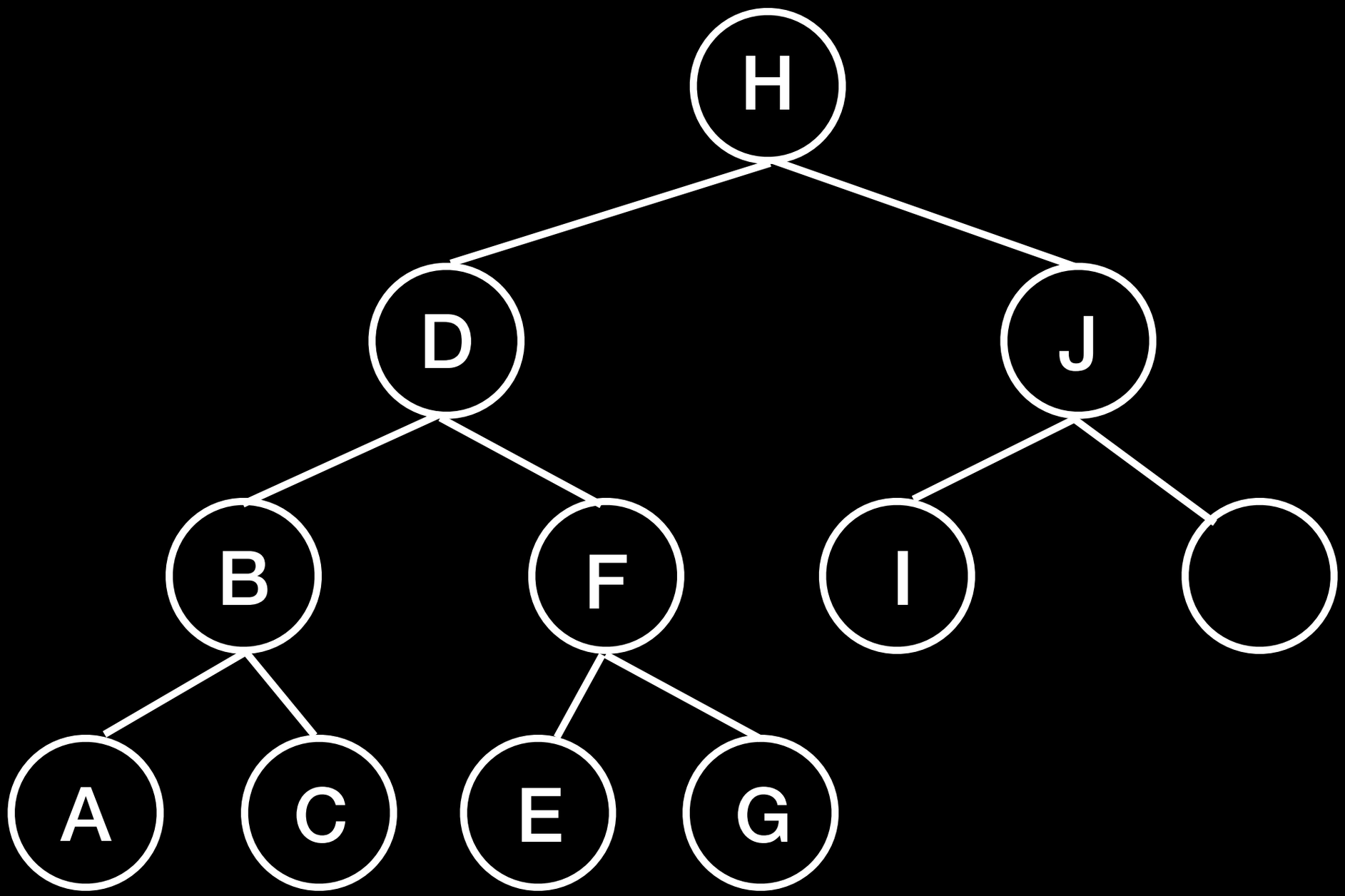
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



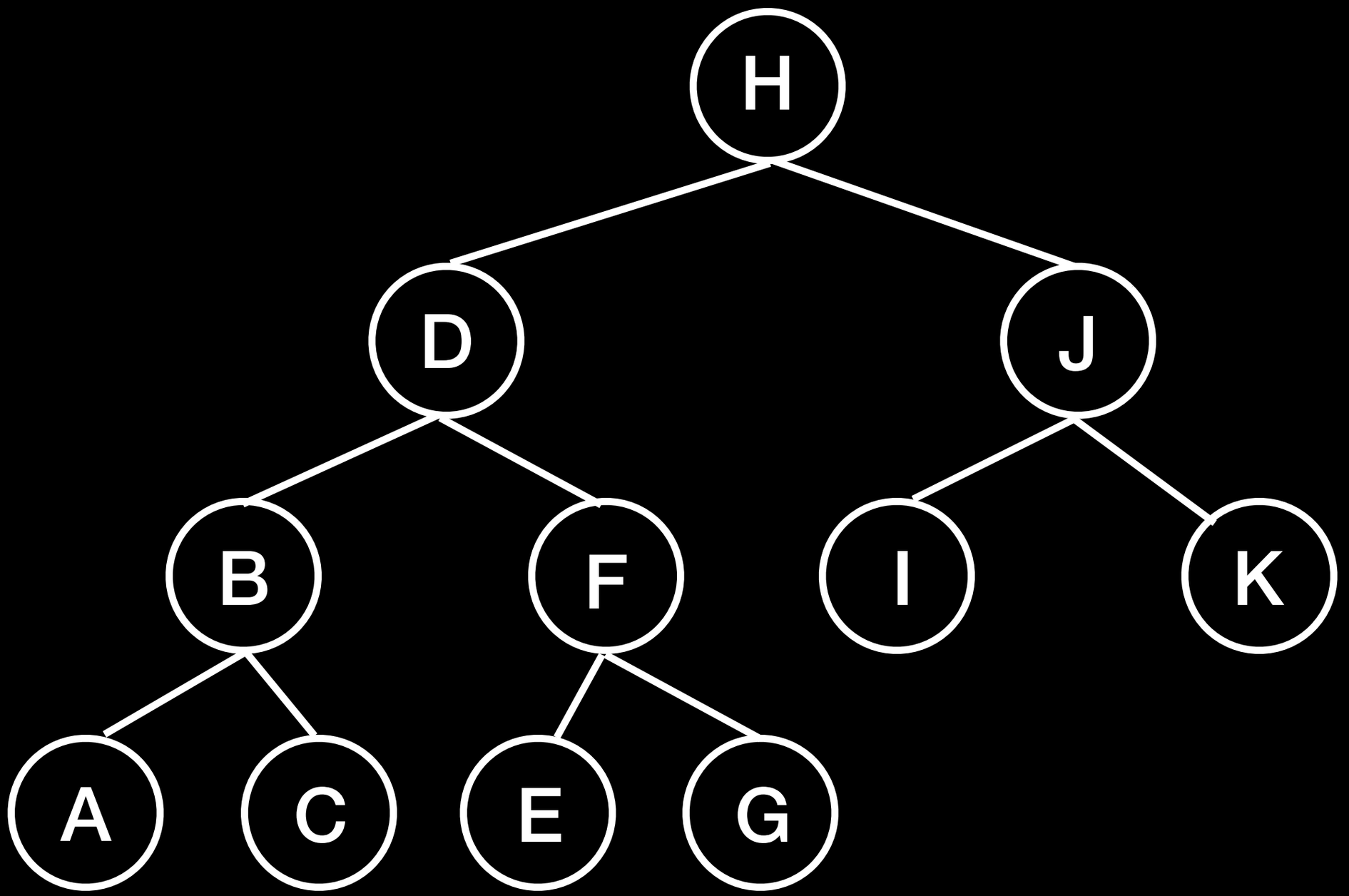
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



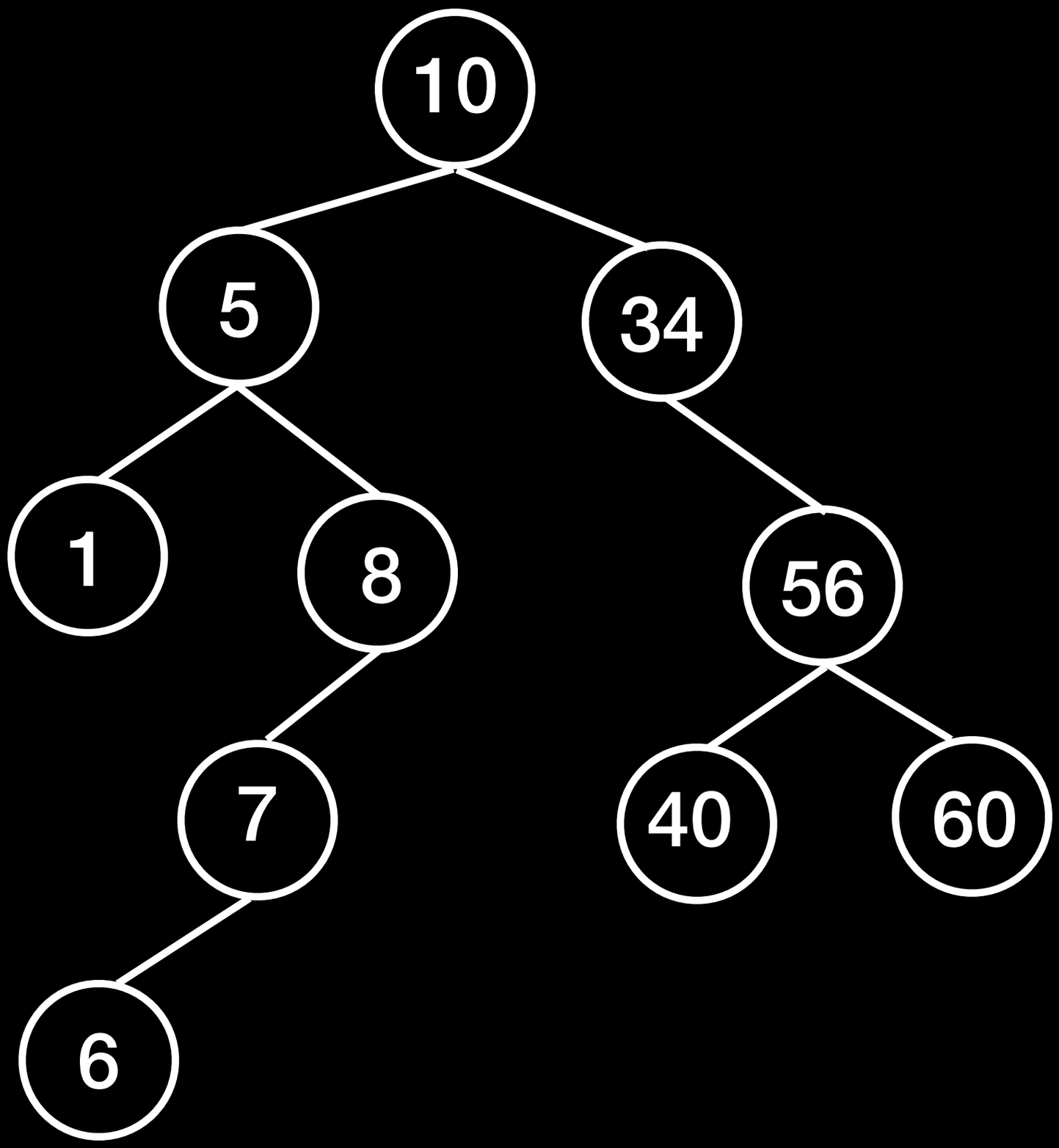
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



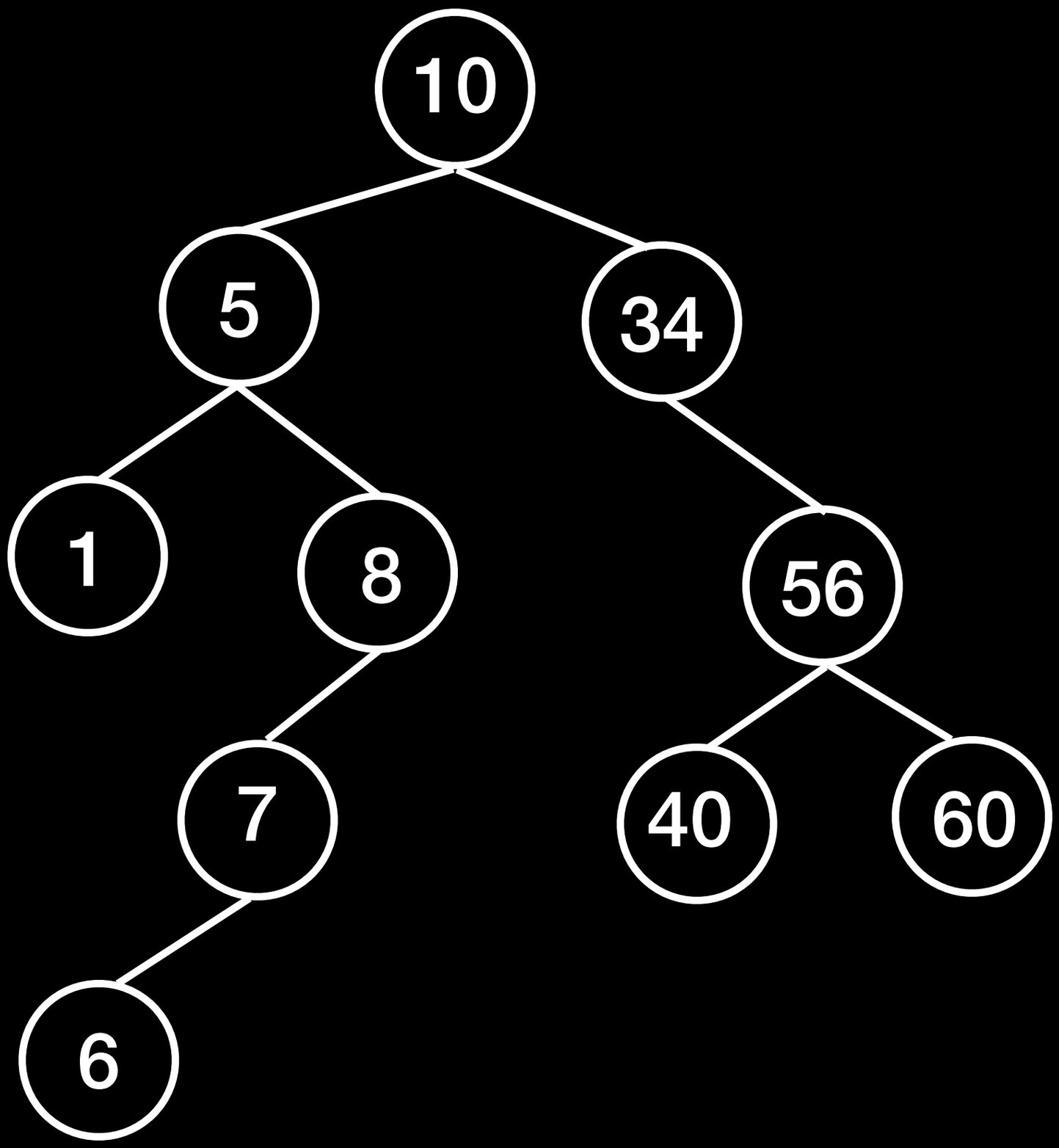
Draw a **complete binary search tree**
that contains the following letters:
A, B, C, D, E, F, G, H, I, J, K



Tree traversals

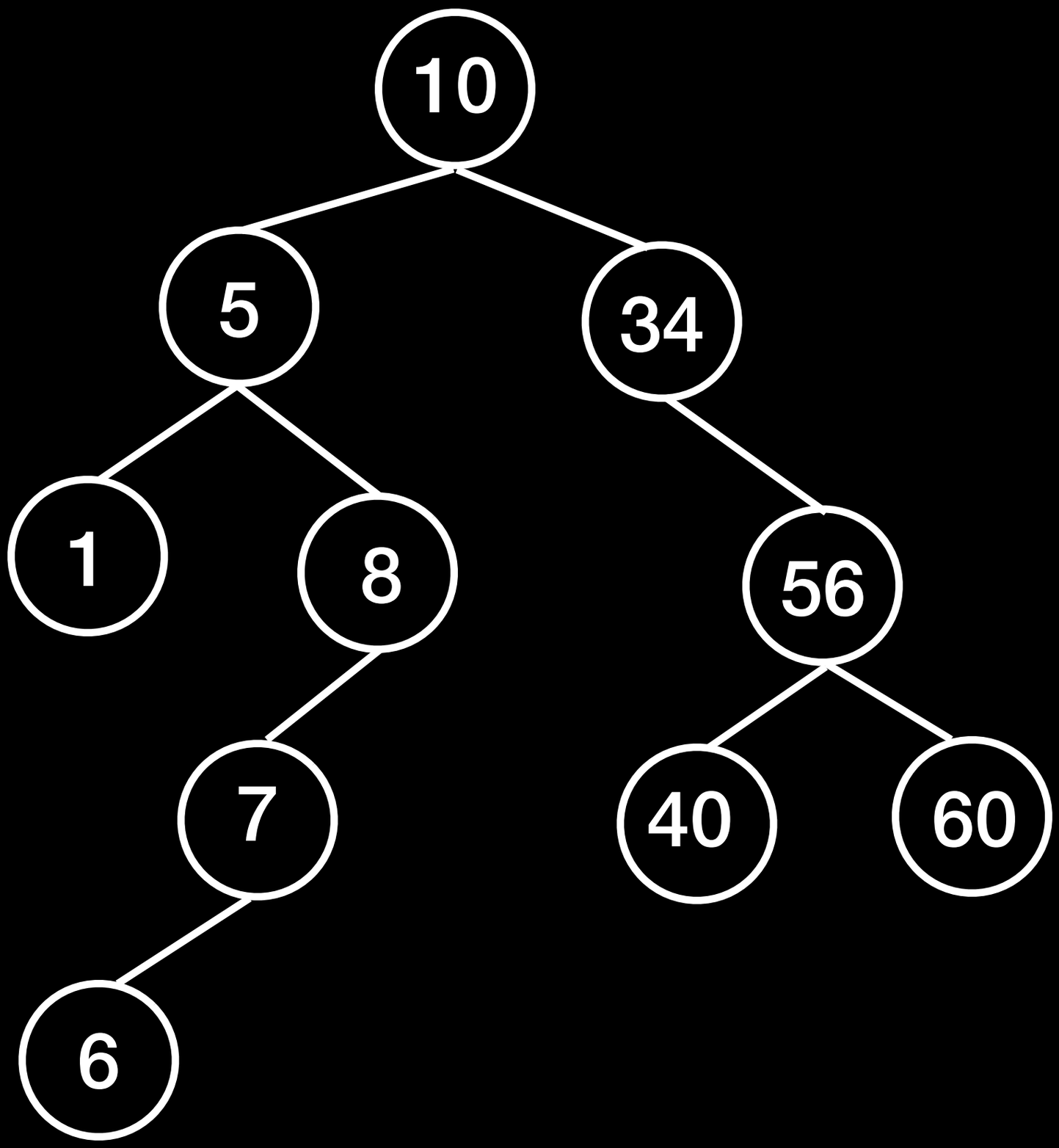


Tree traversals



Pre-order

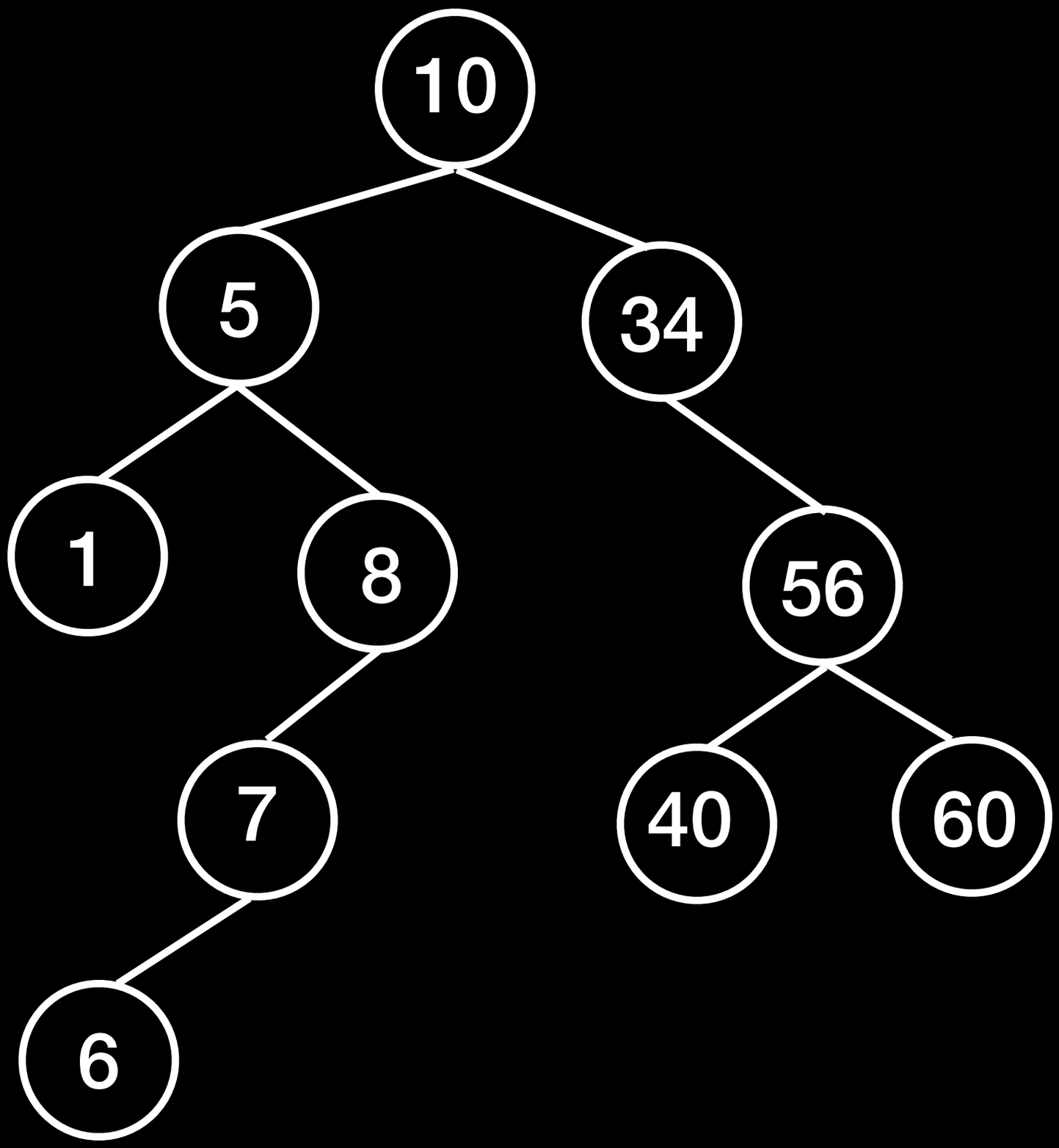
Tree traversals



Pre-order

10, 5, 1, 8, 7, 6, 34, 56, 40, 60

Tree traversals

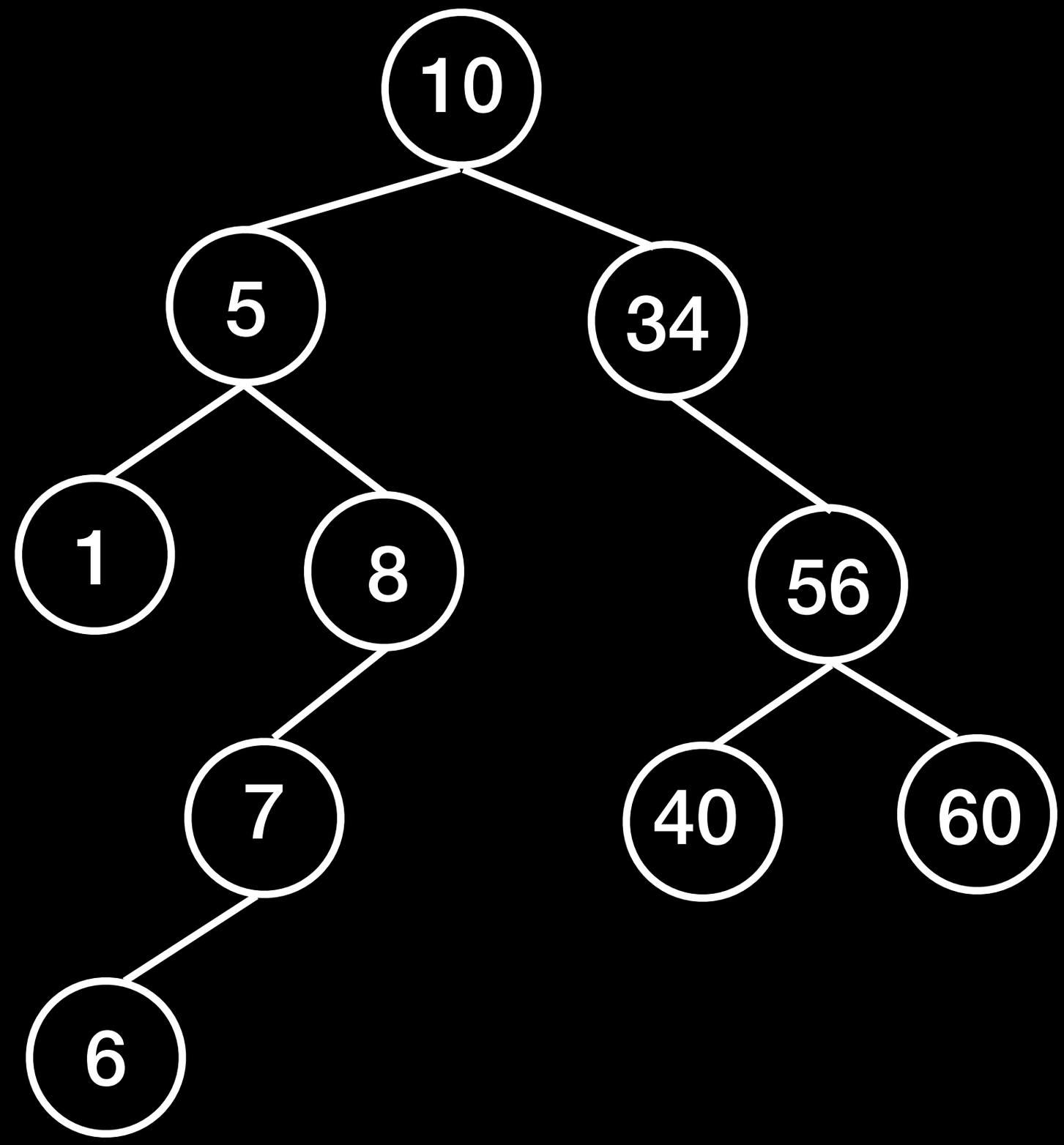


Pre-order

10, 5, 1, 8, 7, 6, 34, 56, 40, 60

In-order

Tree traversals



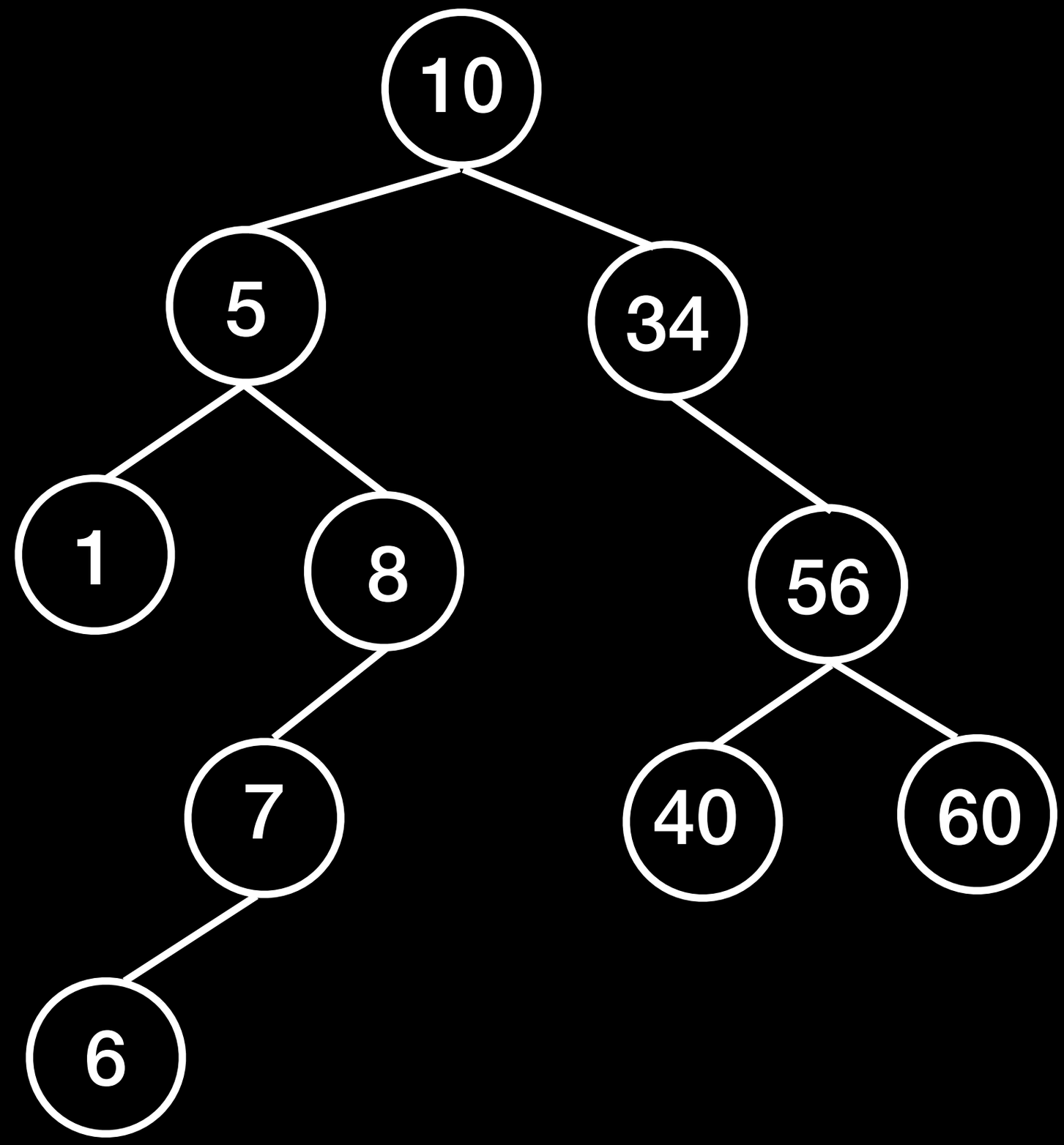
Pre-order

10, 5, 1, 8, 7, 6, 34, 56, 40, 60

In-order

1, 5, 6, 7, 8, 10, 34, 40, 56, 60

Tree traversals



Pre-order

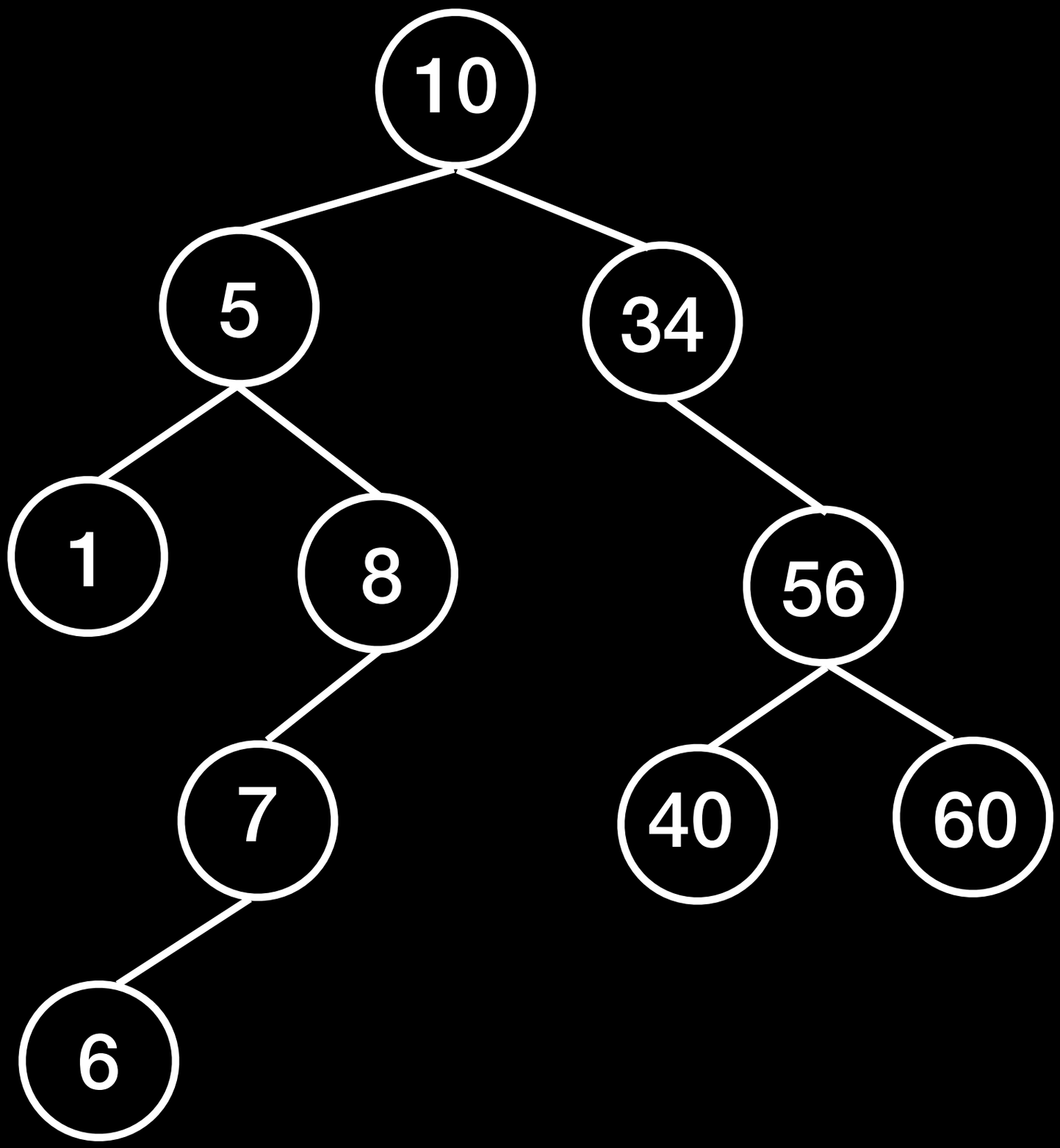
10, 5, 1, 8, 7, 6, 34, 56, 40, 60

In-order

1, 5, 6, 7, 8, 10, 34, 40, 56, 60

Post-order

Tree traversals



Pre-order

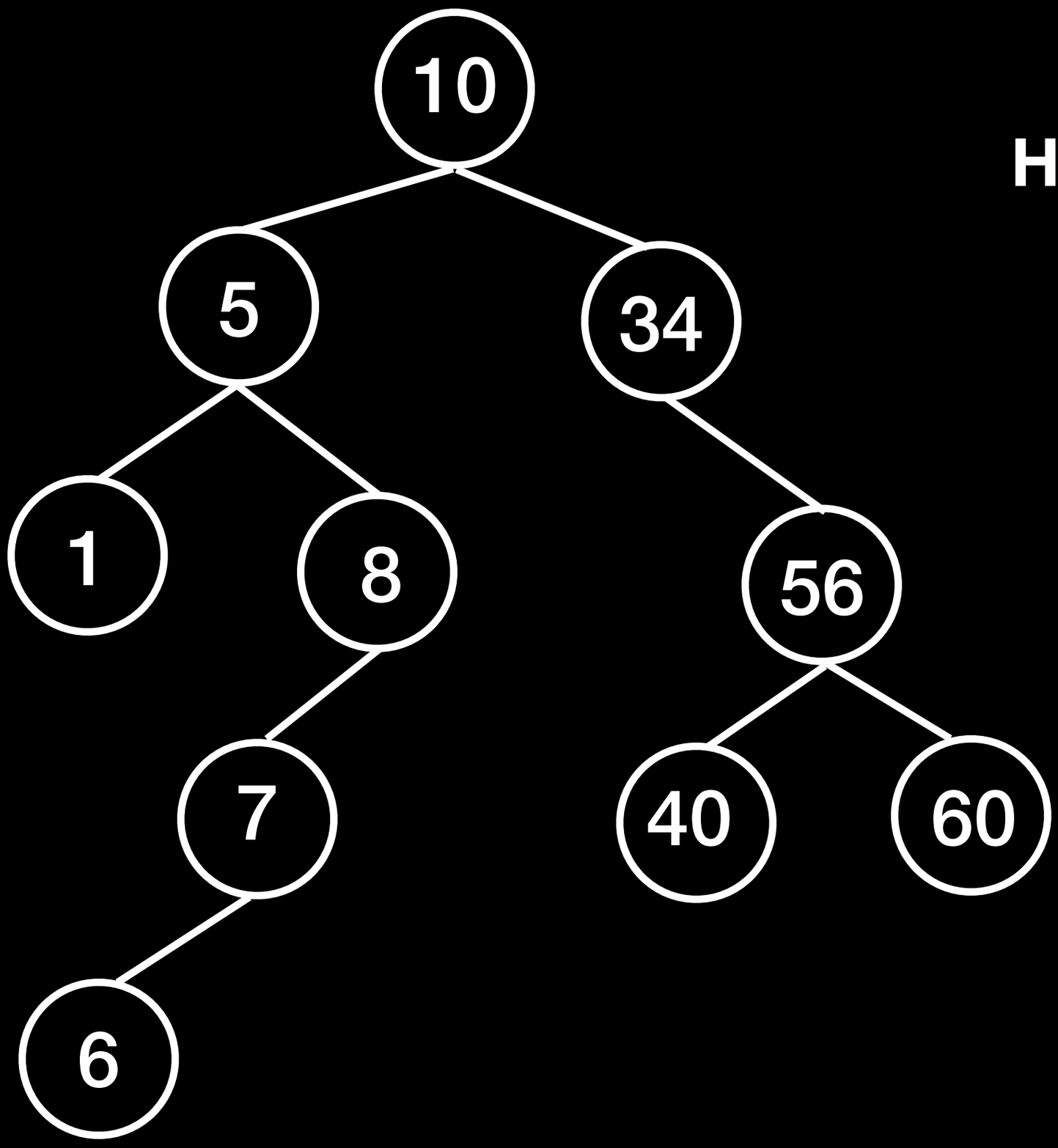
10, 5, 1, 8, 7, 6, 34, 56, 40, 60

In-order

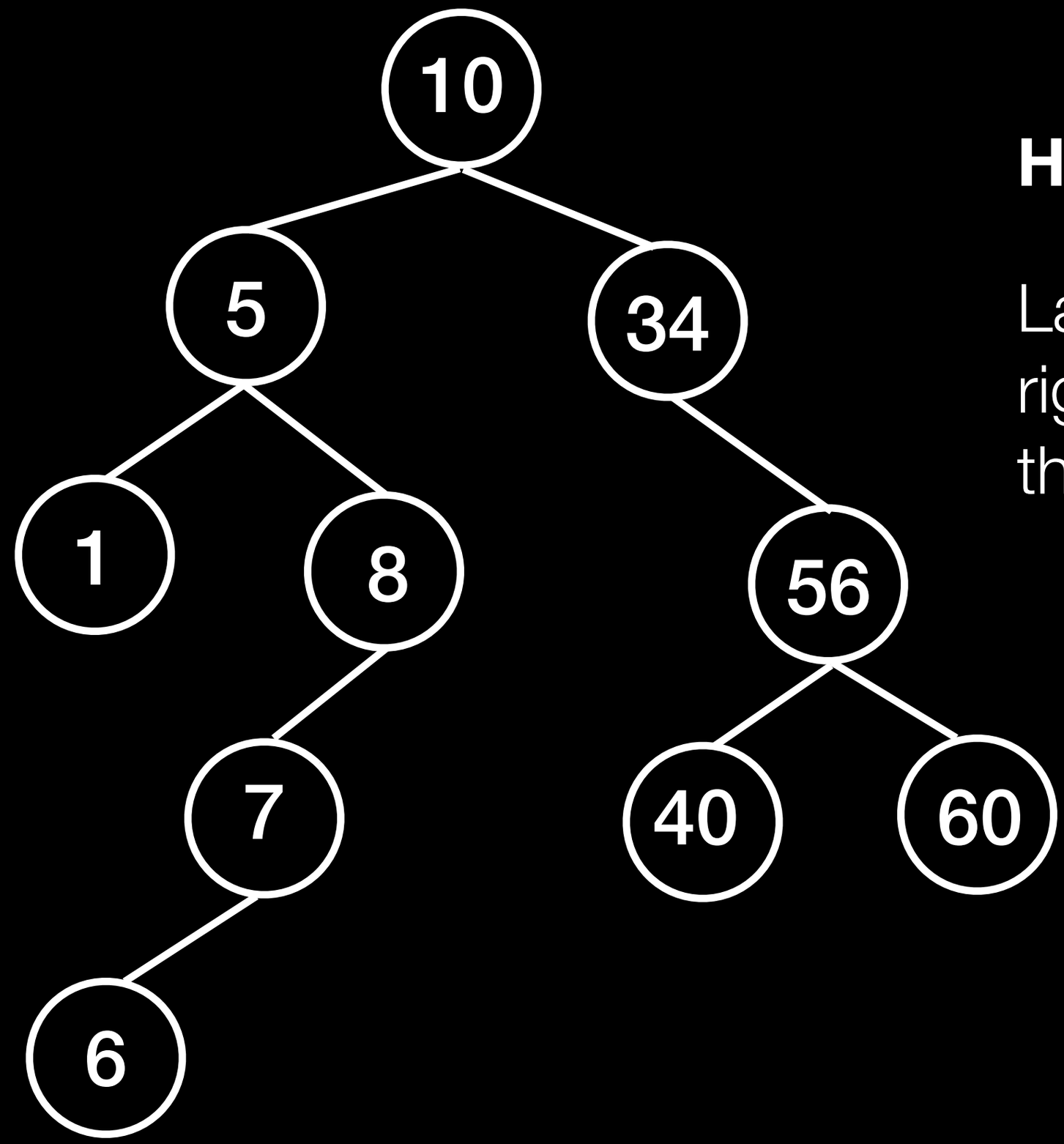
1, 5, 6, 7, 8, 10, 34, 40, 56, 60

Post-order

1, 6, 7, 8, 5, 40, 60, 56, 34, 10

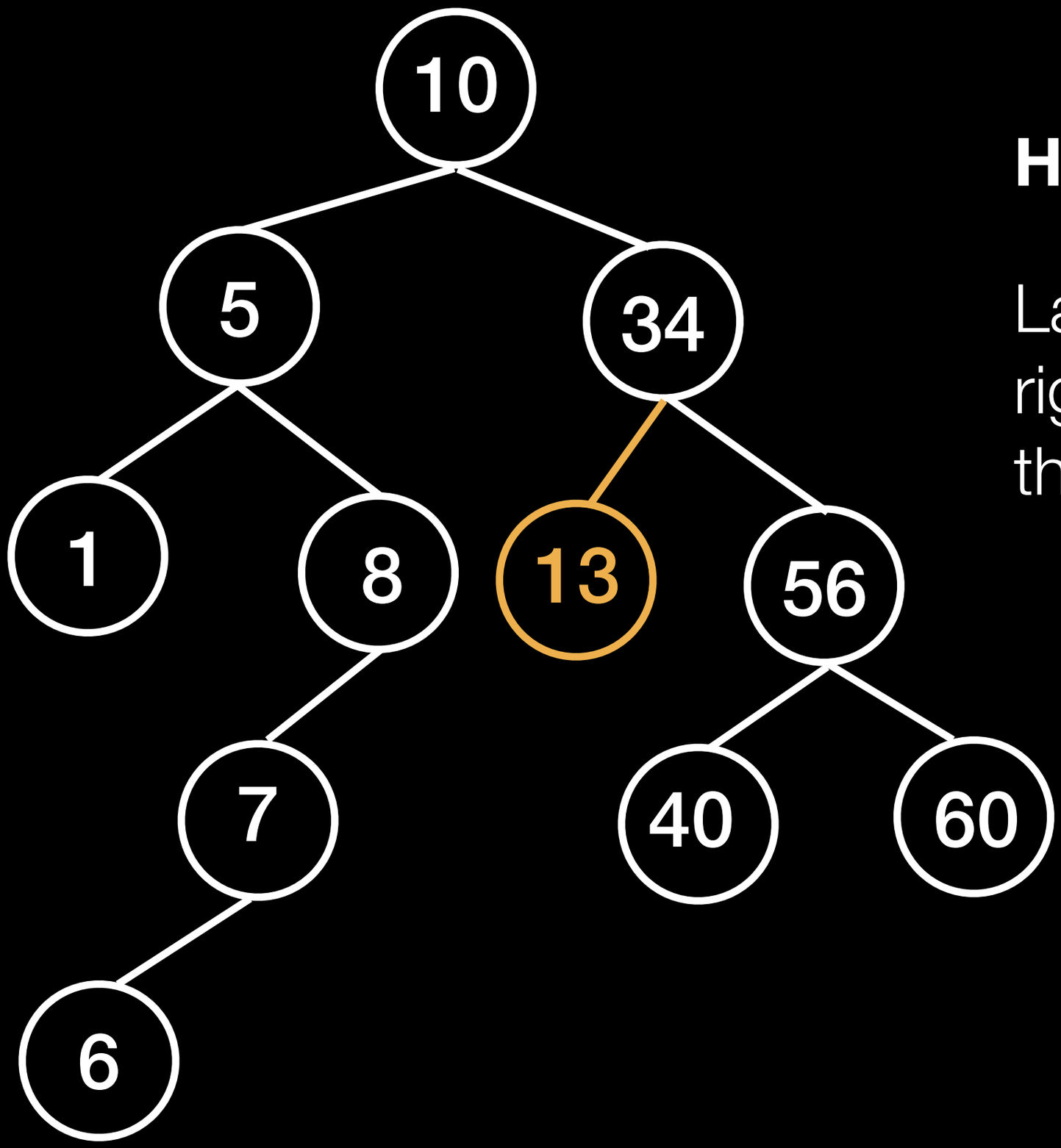


How would we add 13?



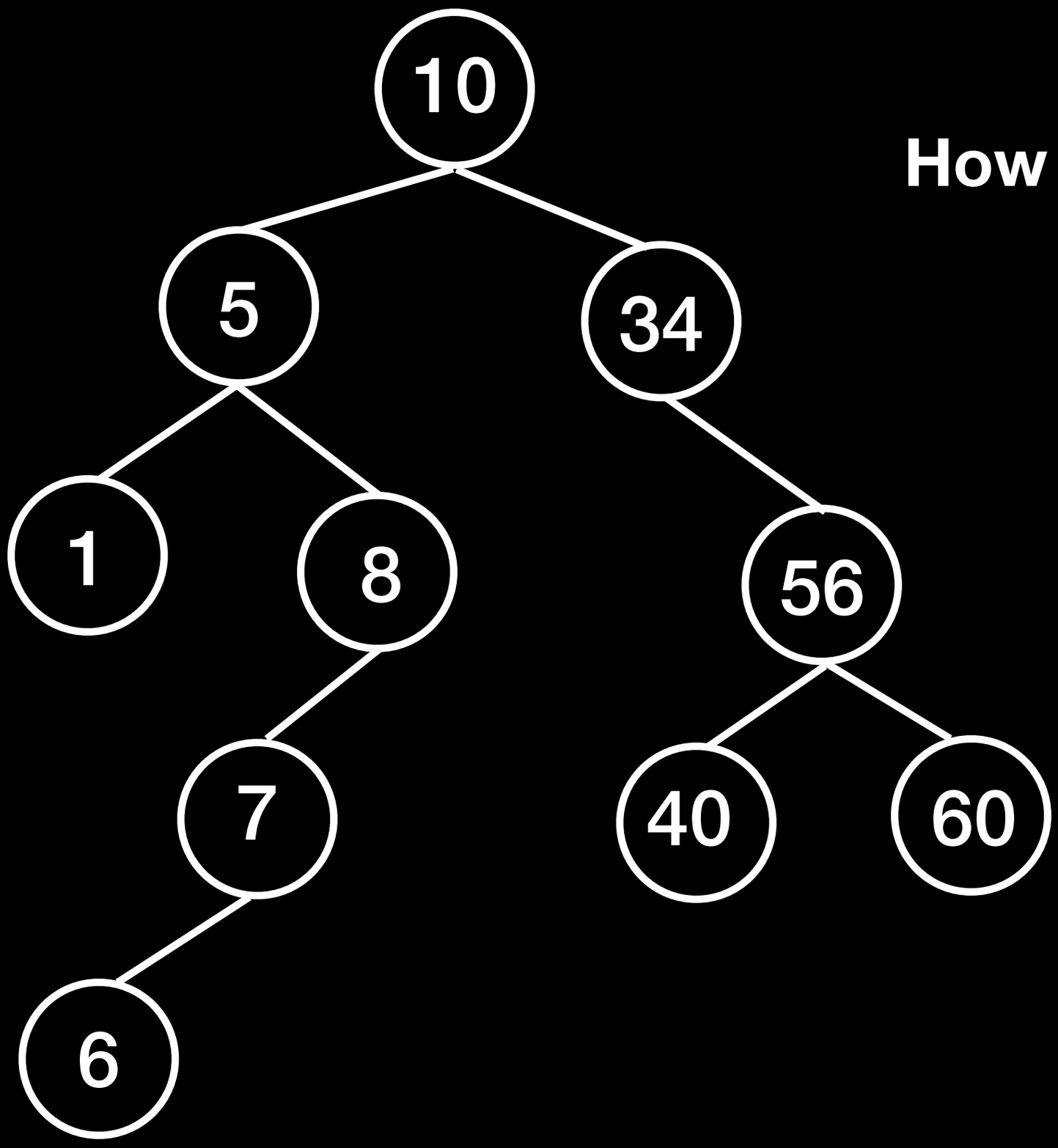
How would we add 13?

Larger values go to the right, smaller values go to the left

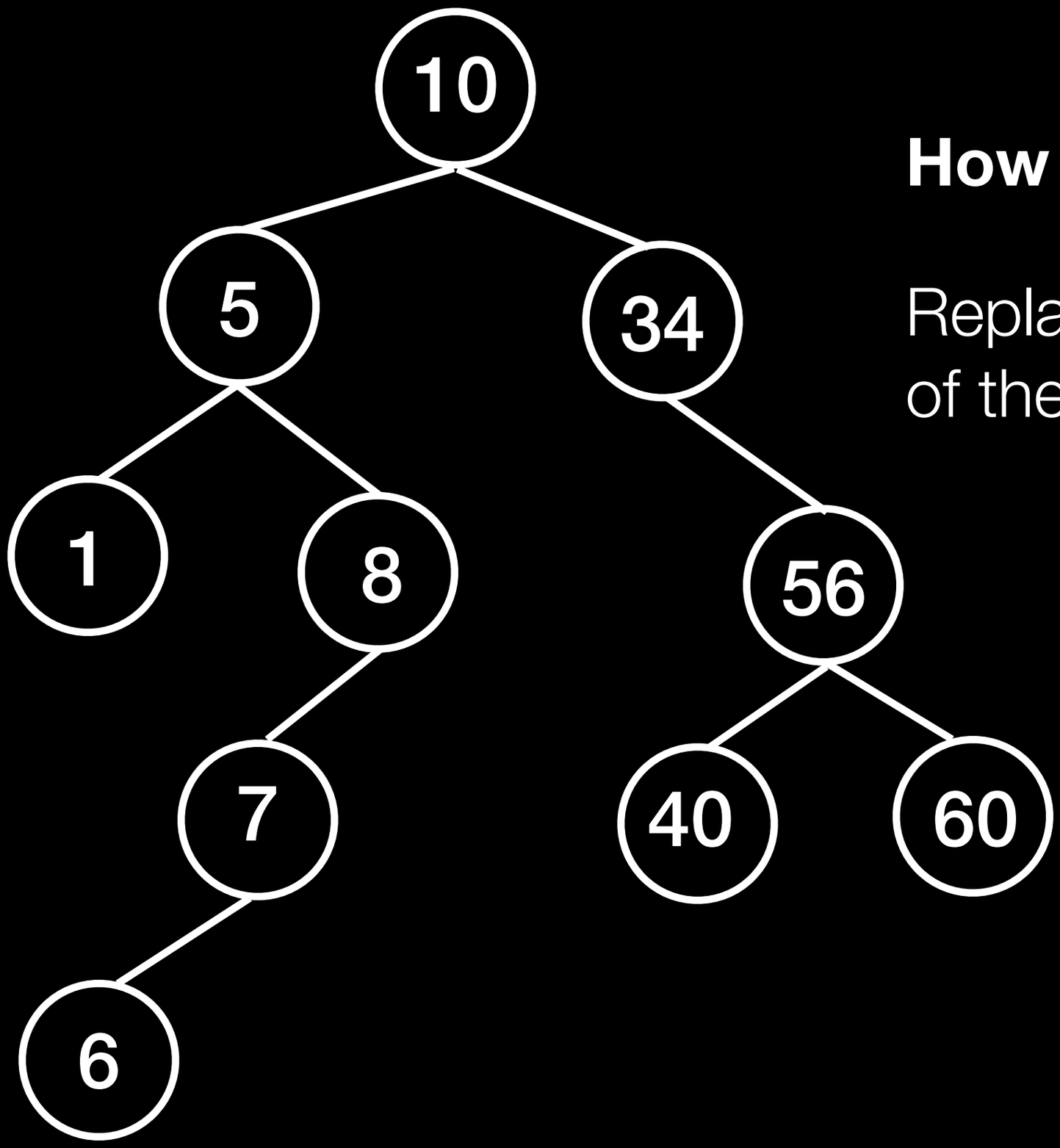


How would we add 13?

Larger values go to the right, smaller values go to the left

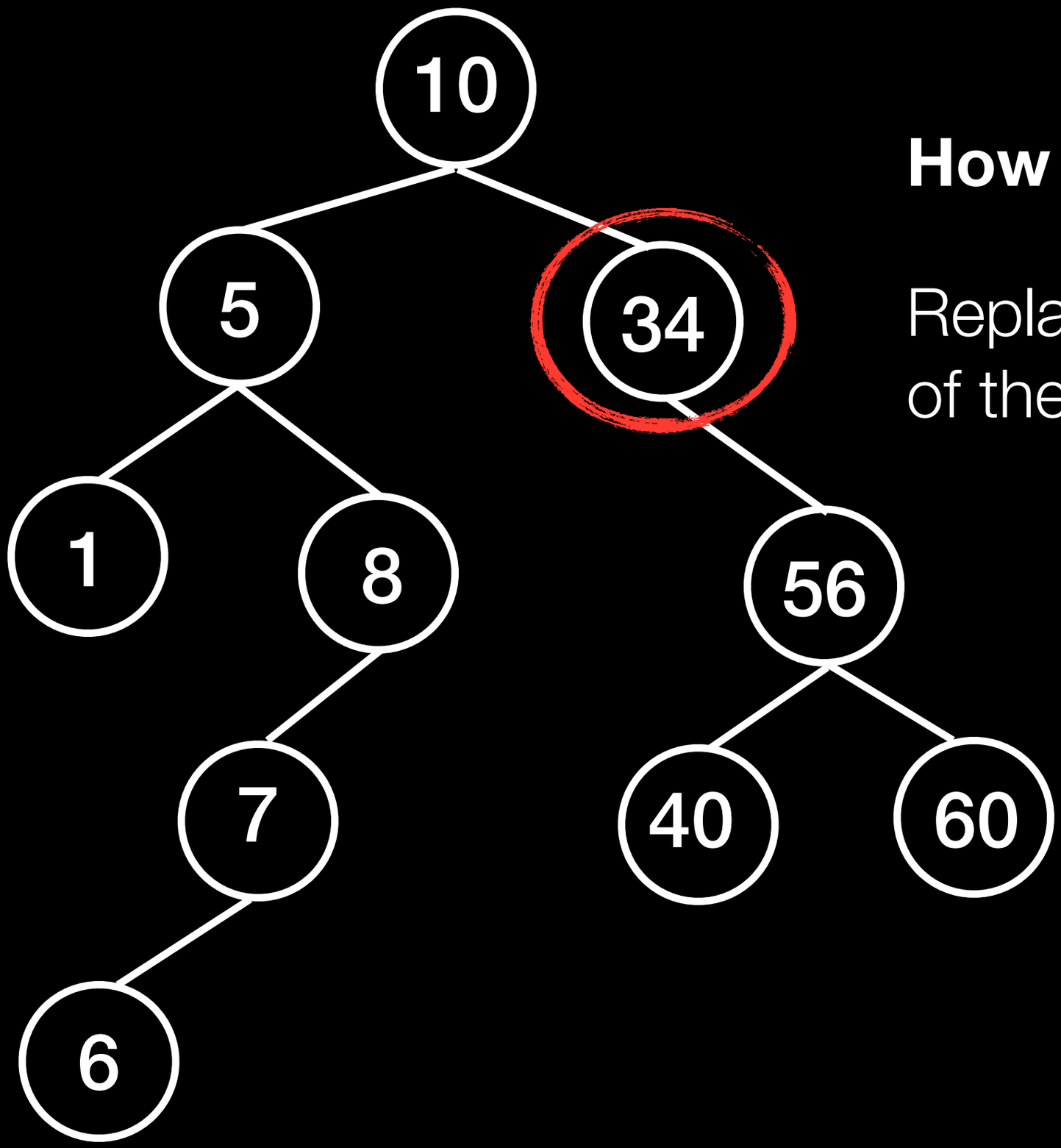


How would we remove 10?



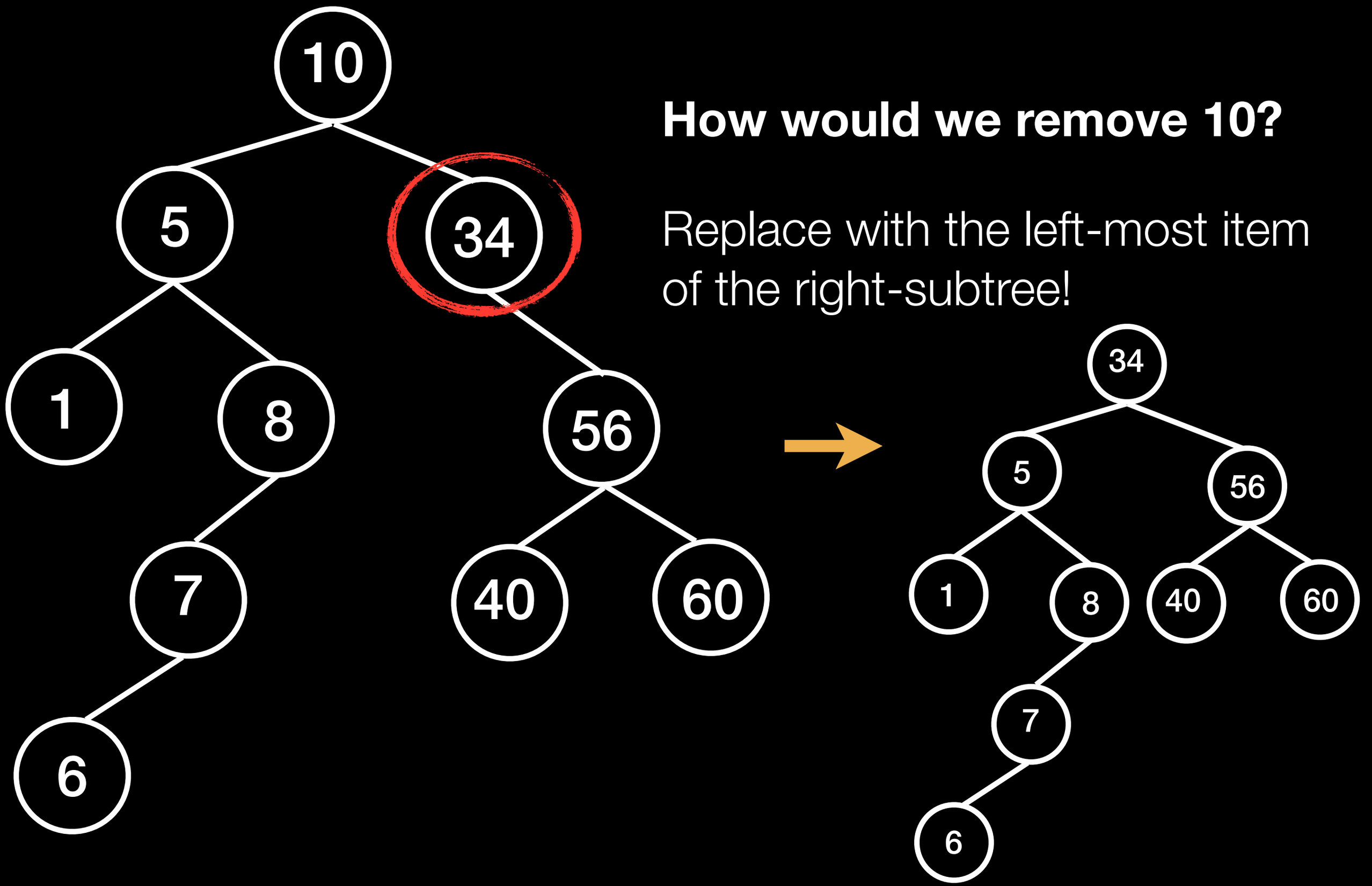
How would we remove 10?

Replace with the left-most item of the right-subtree!



How would we remove 10?

Replace with the left-most item of the right-subtree!



How would we remove 10?

Replace with the left-most item of the right-subtree!

Write a **treeSort** function that takes an array of elements and returns those elements in sorted order. Assume you do not have an iterator (your approach **must be recursive**). You will need a helper function.

Write a **treeSort** function that takes an array of elements and returns those elements in sorted order. Assume you do not have an iterator (your approach **must be recursive**). You will need a helper function.

```
struct AVLTree* newAVLTree();
void addAVLTree(struct AVLTree *tree, TYPE val);

void treeSort (TYPE data[], int n) {
  // WRITE ME
}

void _treeSortHelper(AVLNode *cur, TYPE *data,
int *count) {
  // WRITE ME
}
```

```
void treeSort(TYPE data[], int size){
    int i;
    int sortIdx = 0;



}
```

```
void treeSort(TYPE data[], int size){
    int i;
    int sortIdx = 0;

    /* declare an AVL tree */
    struct AVLTree *tree = newAVLtree();
    assert(data != NULL && size > 0);

}
```

```
void treeSort(TYPE data[], int size){
    int i;
    int sortIdx = 0;

    /* declare an AVL tree */
    struct AVLTree *tree = newAVLtree();
    assert(data != NULL && size > 0);

    /* add elements to the tree */
    for (i = 0; i < size; i++)
        addAVLTree(tree, data[i]);



}
```

```
void treeSort(TYPE data[], int size){
     int i;
     int sortIdx = 0;

     /* declare an AVL tree */
     struct AVLTree *tree = newAVLtree();
     assert(data != NULL && size > 0);

     /* add elements to the tree */
     for (i = 0; i < size; i++)
          addAVLTree(tree, data[i]);

    /* call the helper function on the root */
     _treeSortHelper(tree->root, data, &sortIdx);
}
```

```
/* *index goes from 0 to size-1 */

void _treeSortHelper(AVLNode *cur, TYPE *data,
                     int *index){

}
```

```
/* *index goes from 0 to size-1 */

void _treeSortHelper(AVLNode *cur, TYPE *data,
                     int *index){
    /* In-order traversal: get the left subtree,
       then this node, then the right subtree */
    if (cur != NULL) {
        _treeSortHelper(cur->left, data, index);
        data[*index] = cur->val;
        (*index)++;
        _treeSortHelper(cur->right, data, index);
    }
}
```

Is the height of any binary search tree with n nodes always $O(\log n)$?

Is the height of any binary search tree with n nodes always $O(\log n)$?

No, unbalanced trees may have height n

Is the height of any binary search tree with n nodes always $O(\log n)$?

No, unbalanced trees may have height n

Does inserting into an AVL tree with n nodes require looking at $O(\log n)$ nodes?

Is the height of any binary search tree with n nodes always $O(\log n)$?

No, unbalanced trees may have height n

Does inserting into an AVL tree with n nodes require looking at $O(\log n)$ nodes?

Yes, because AVL trees are balanced

Is the height of any binary search tree with n nodes always $O(\log n)$?

No, unbalanced trees may have height n

Does inserting into an AVL tree with n nodes require looking at $O(\log n)$ nodes?

Yes, because AVL trees are balanced

Does inserting into an AVL tree with n nodes require $O(\log n)$ rotations?

Is the height of any binary search tree with n nodes always $O(\log n)$?

No, unbalanced trees may have height n

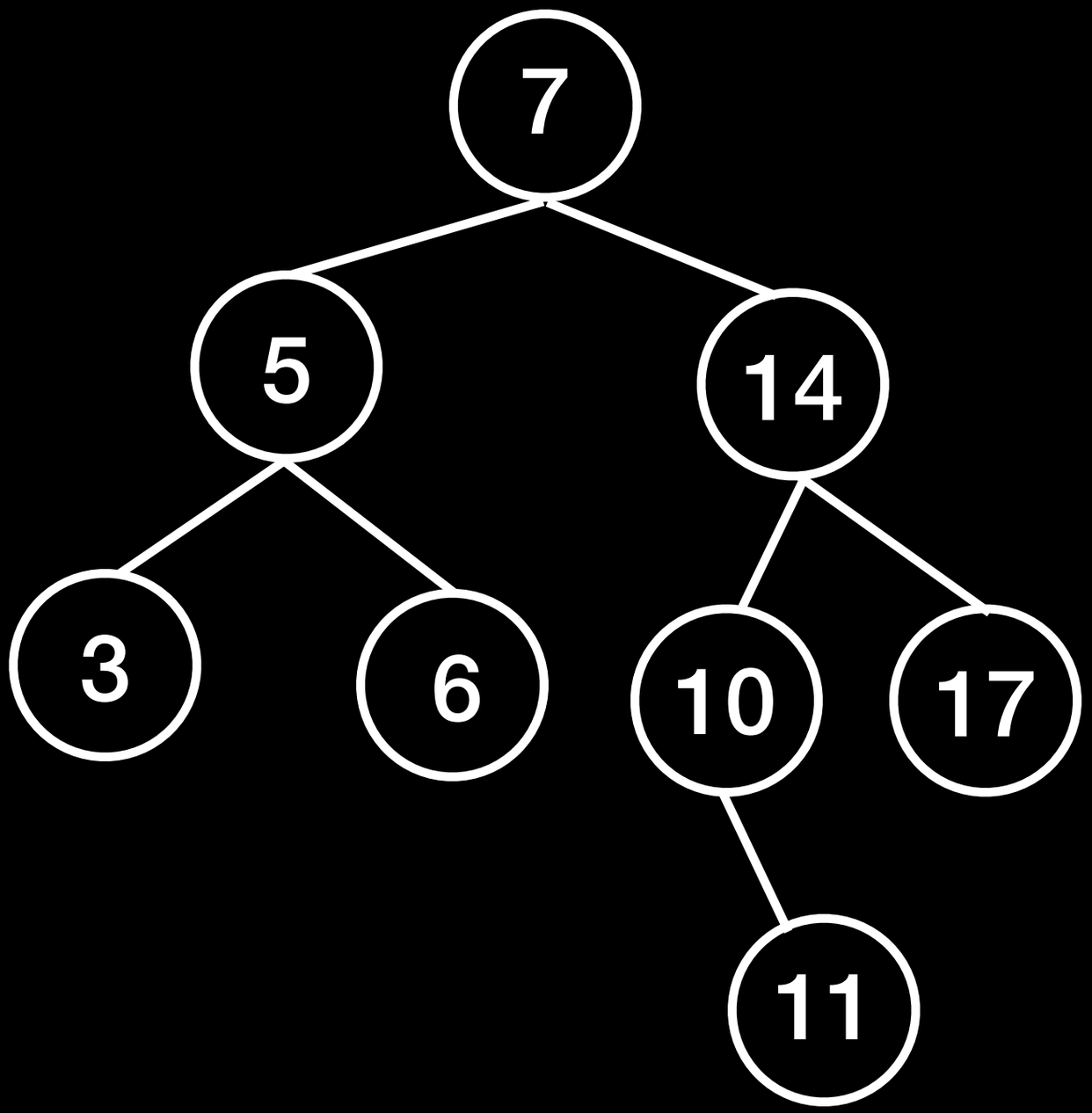
Does inserting into an AVL tree with n nodes require looking at $O(\log n)$ nodes?

Yes, because AVL trees are balanced

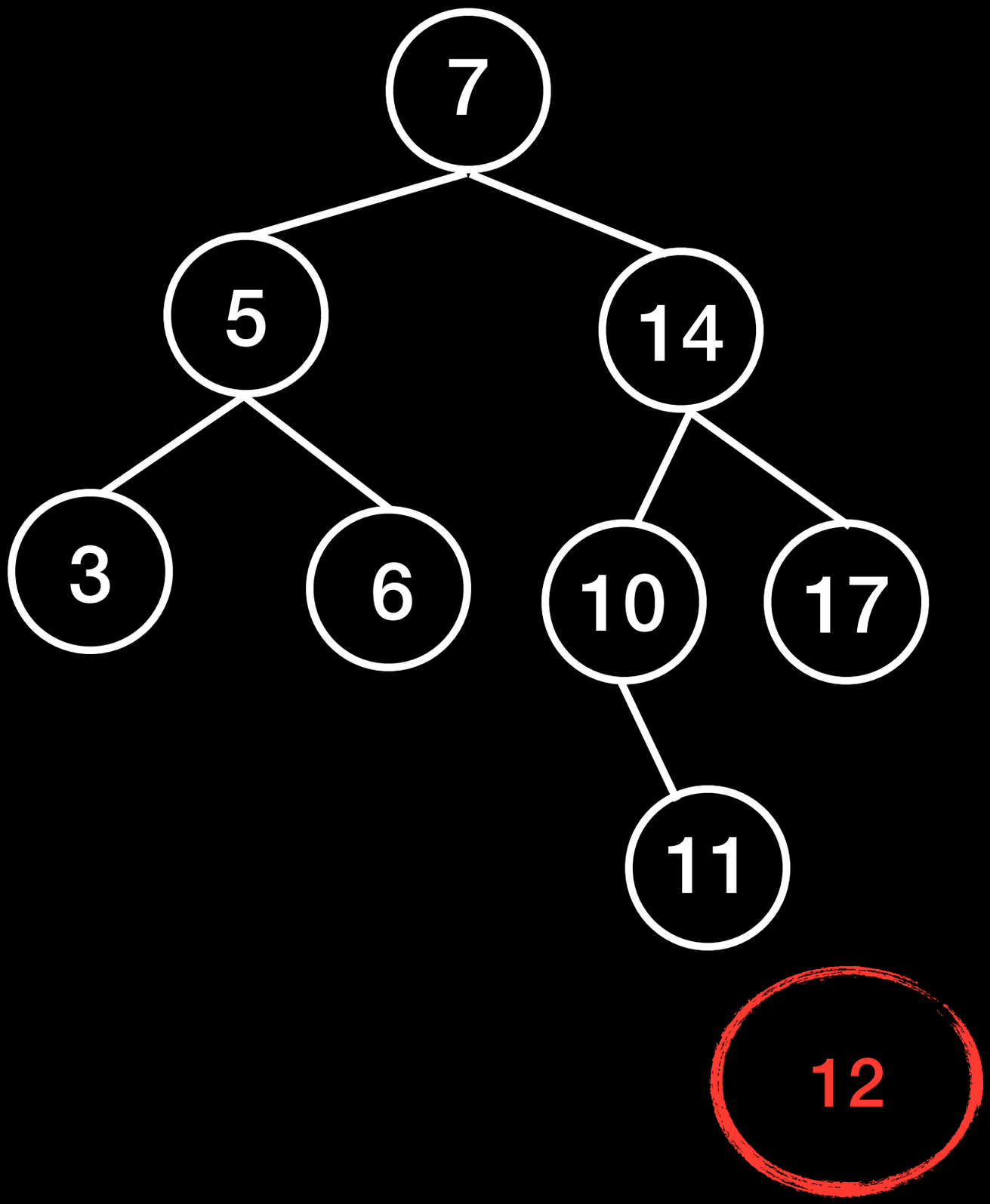
Does inserting into an AVL tree with n nodes require $O(\log n)$ rotations?

No, we need at most 2 rotations

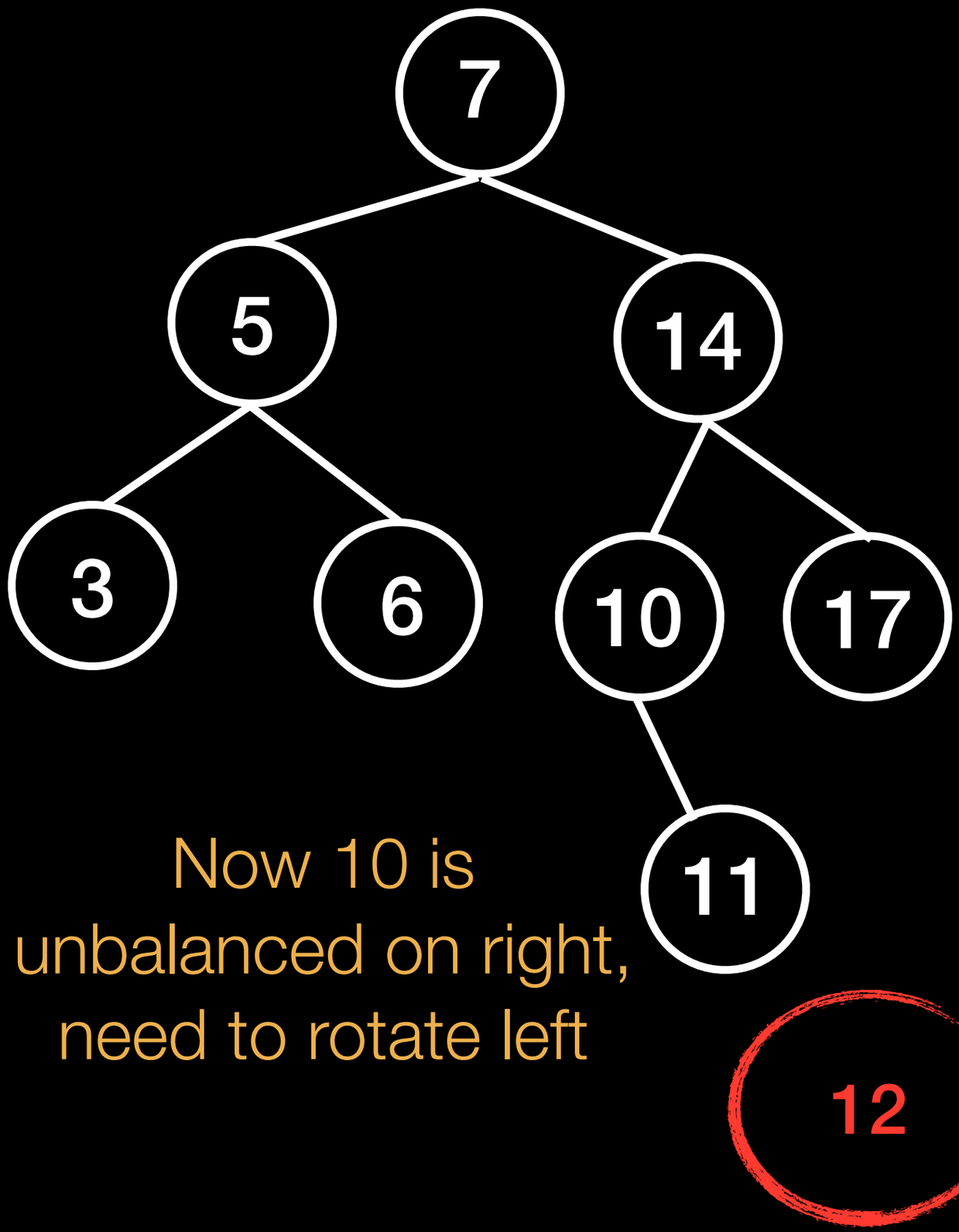
Add 12 to this AVL tree



Add 12 to this AVL tree

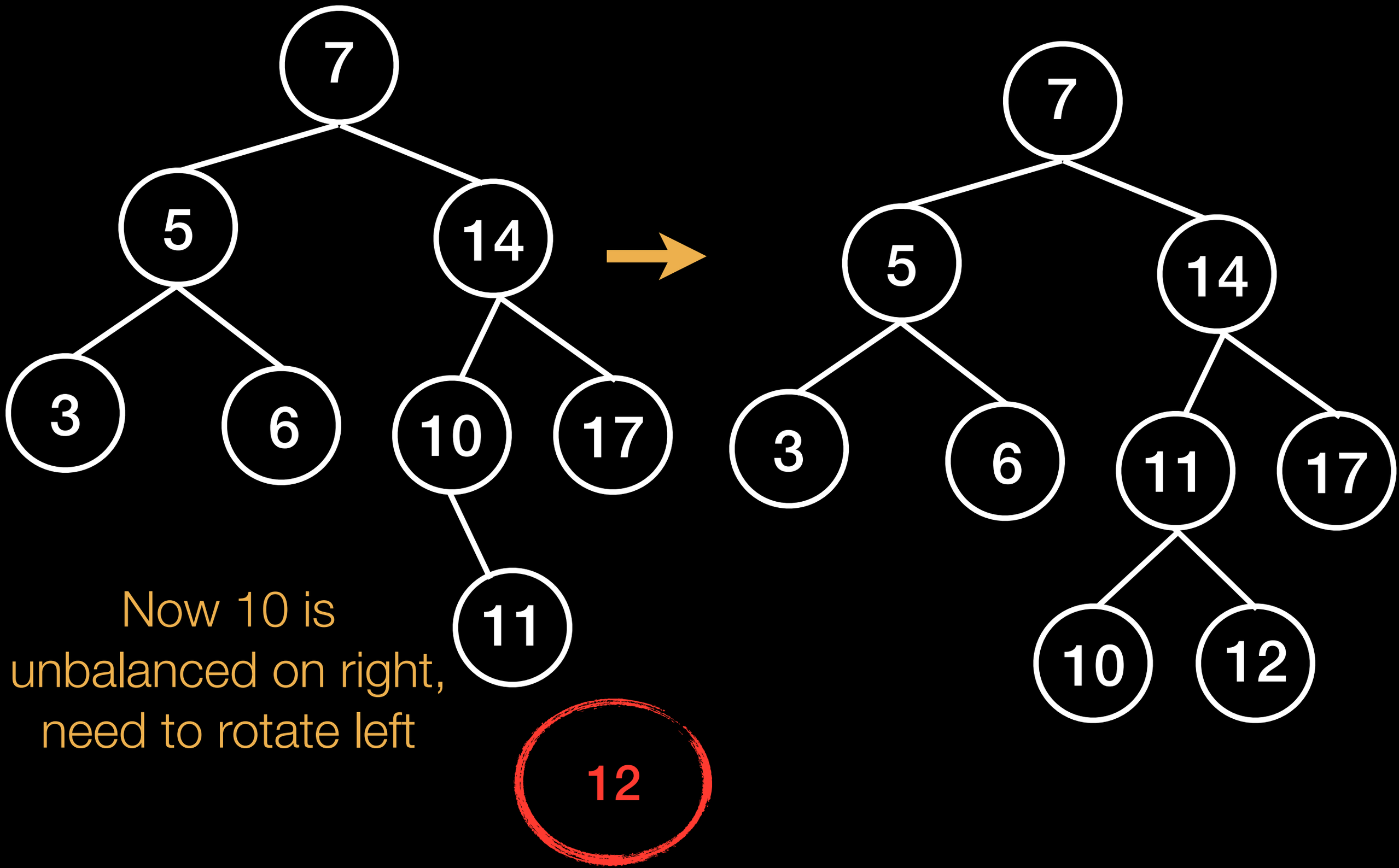


Add 12 to this AVL tree



Now 10 is unbalanced on right, need to rotate left

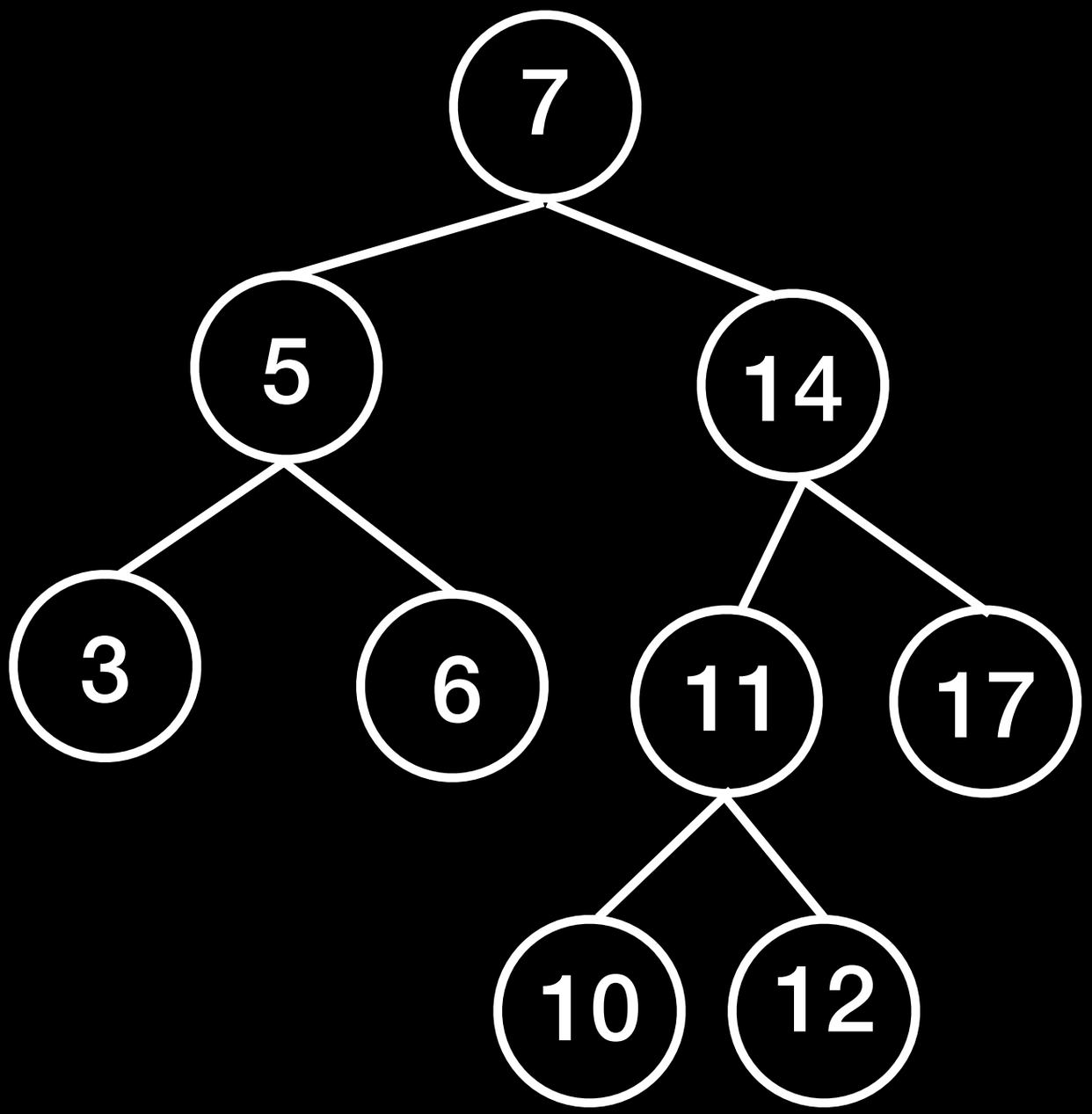
Add 12 to this AVL tree



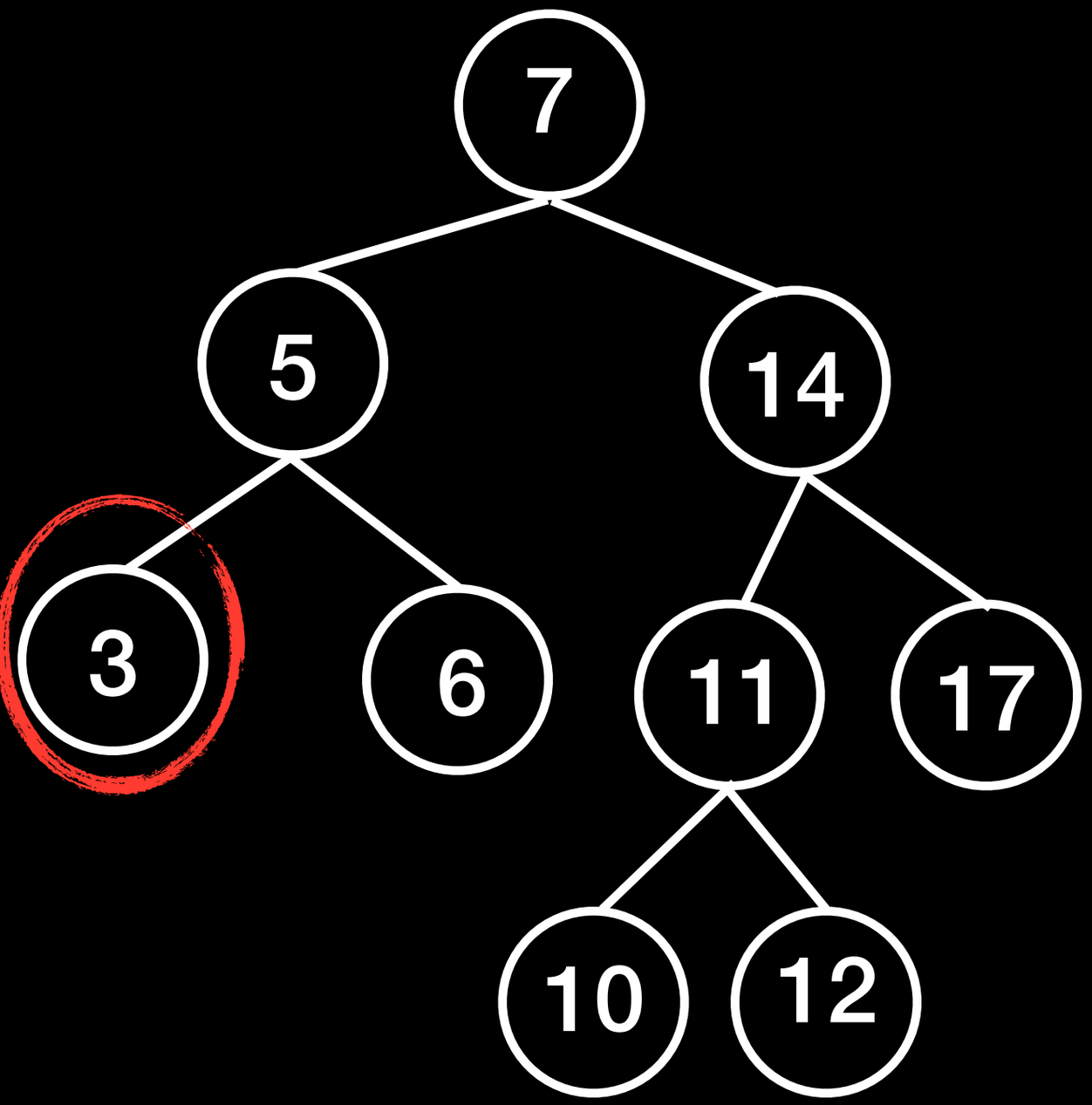
Now 10 is unbalanced on right, need to rotate left

12

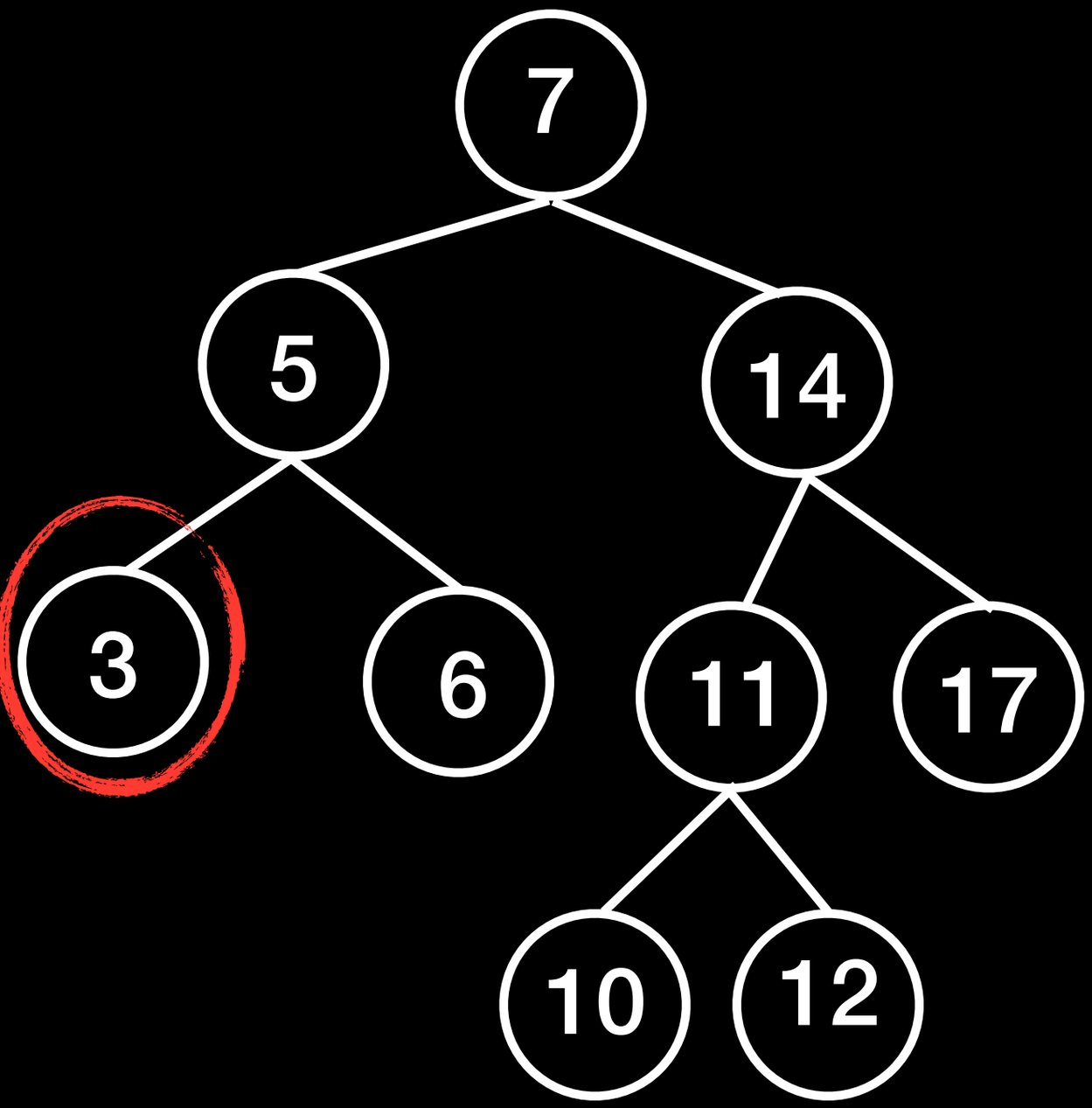
Remove 3



Remove 3

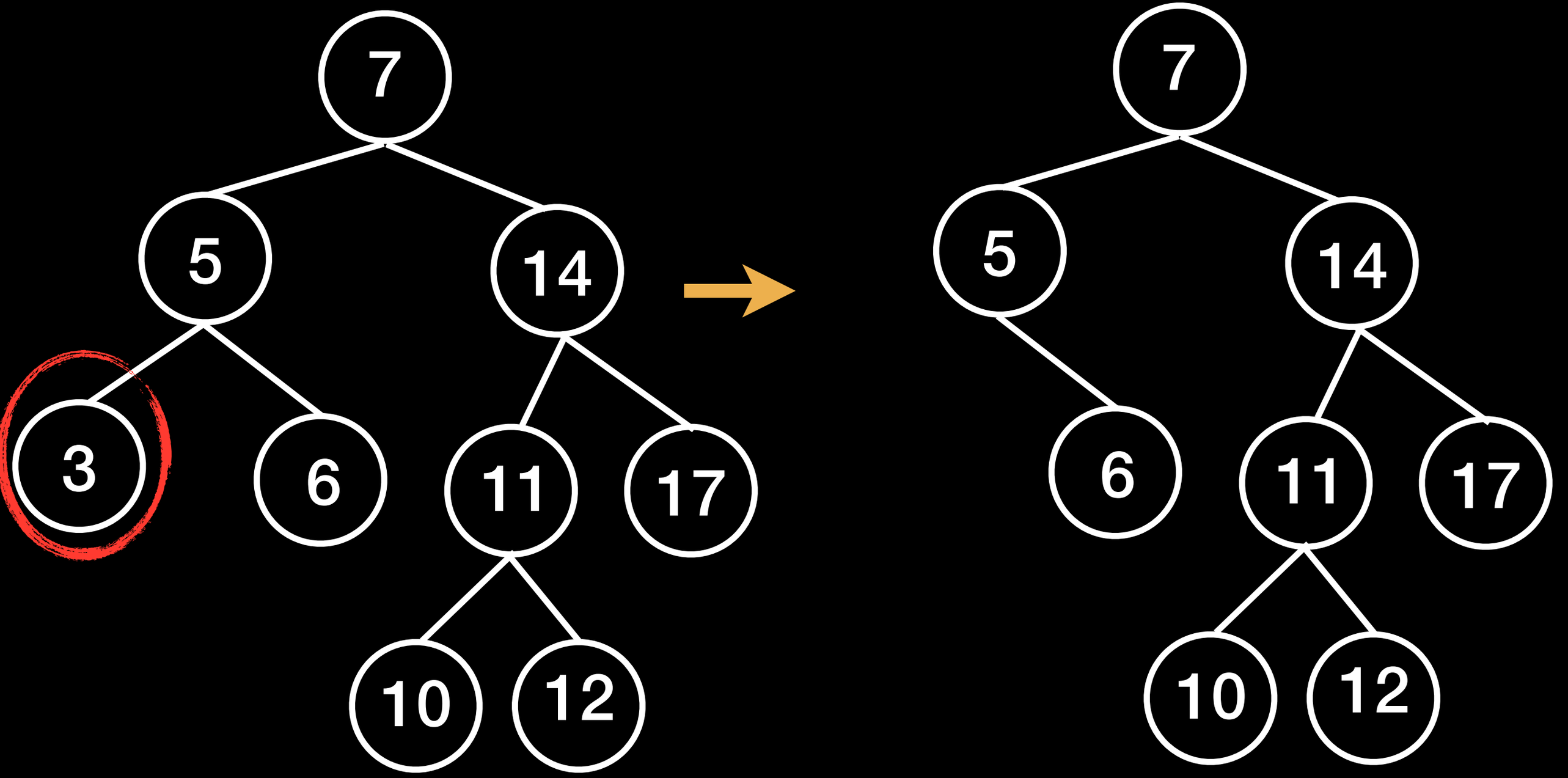


Remove 3



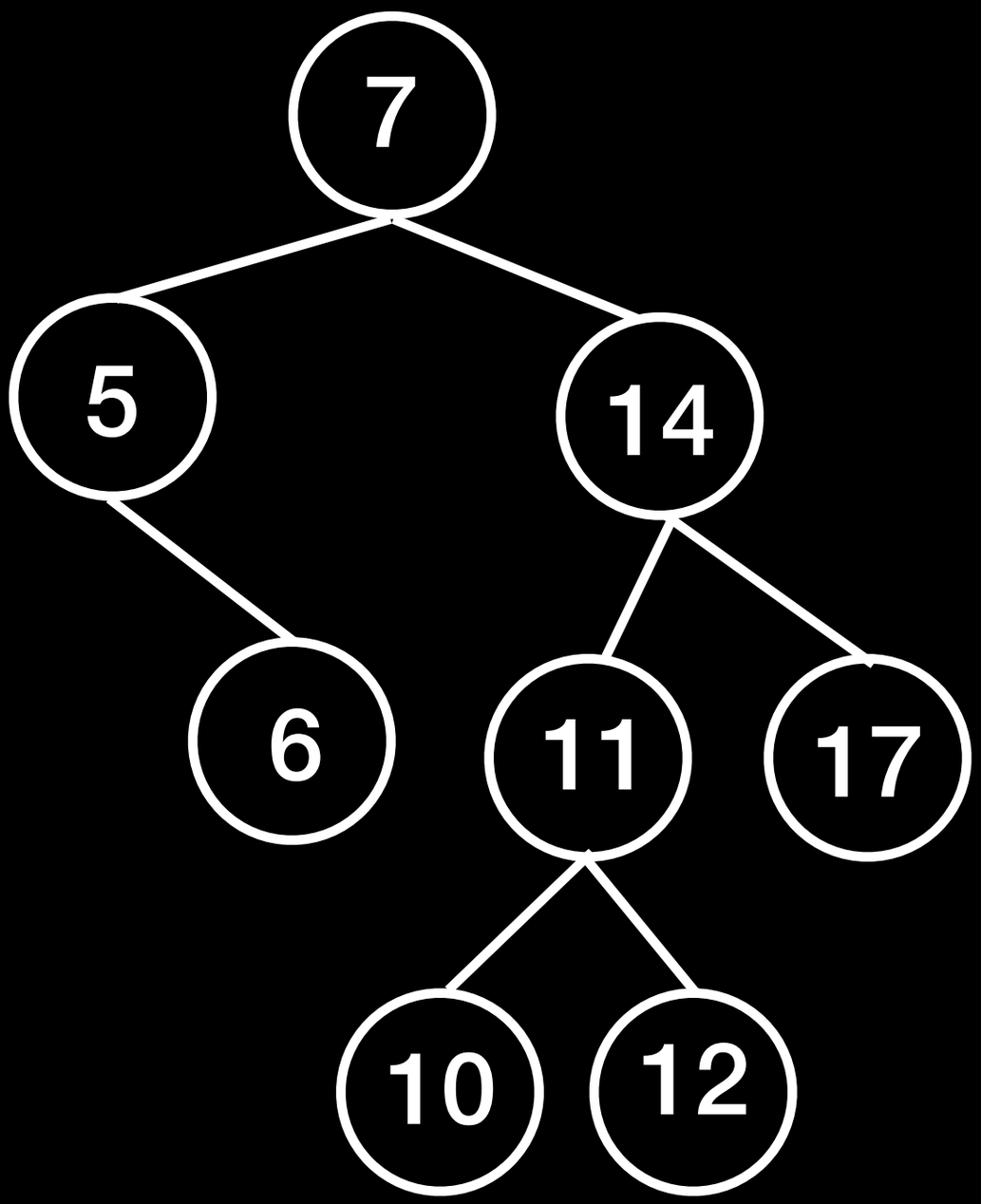
Still balanced, no rotations needed

Remove 3

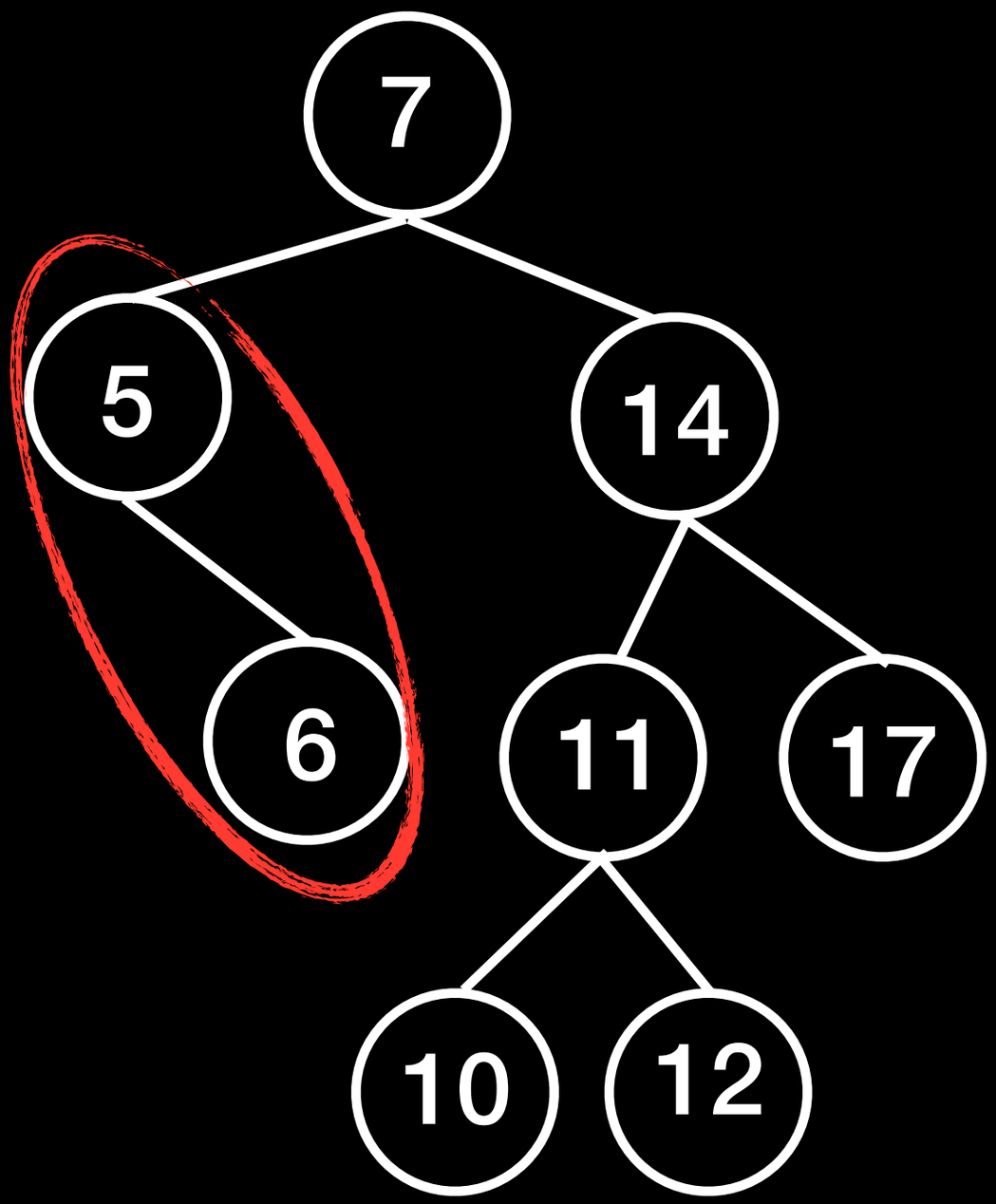


Still balanced, no rotations needed

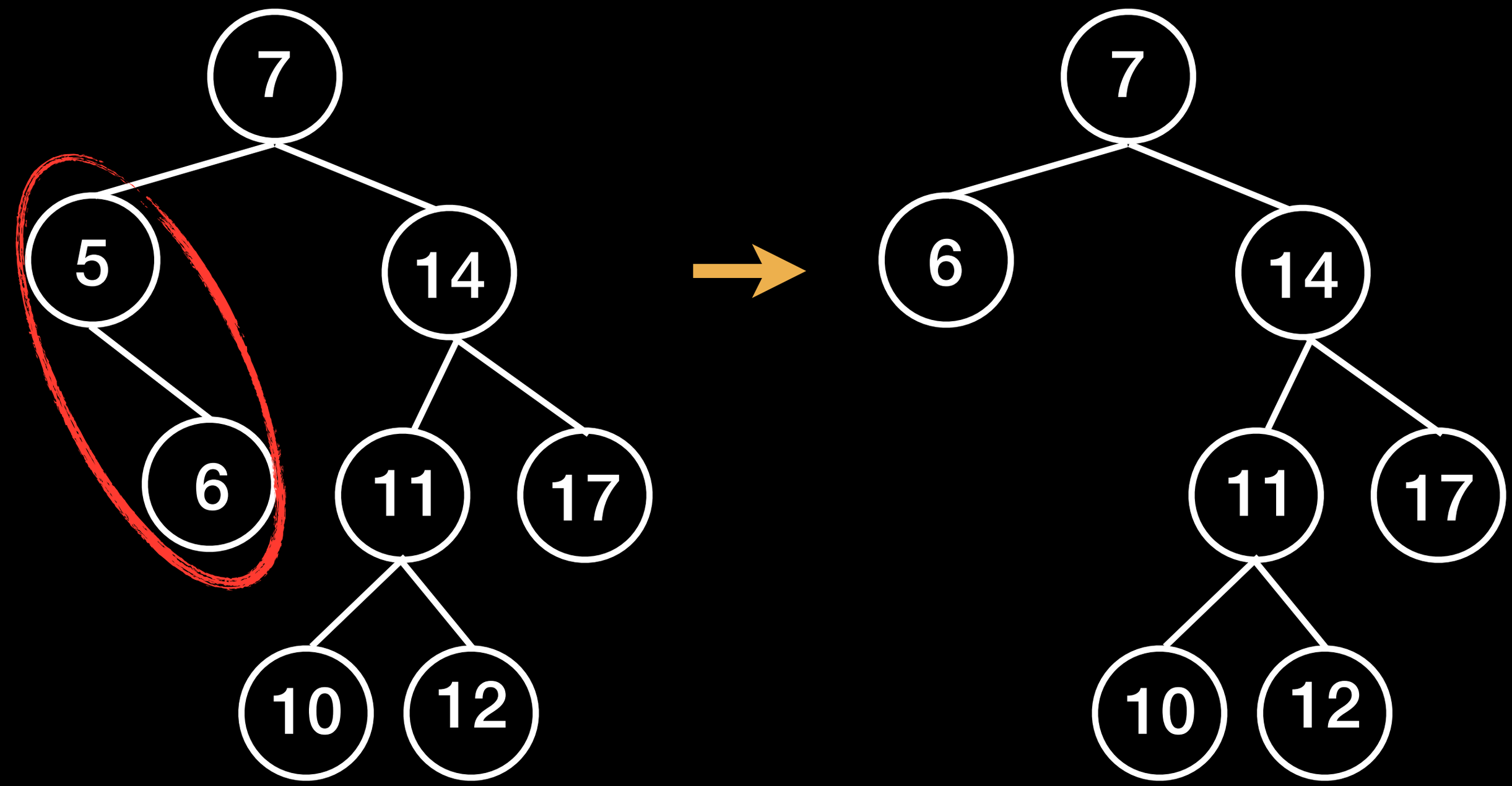
Remove 5



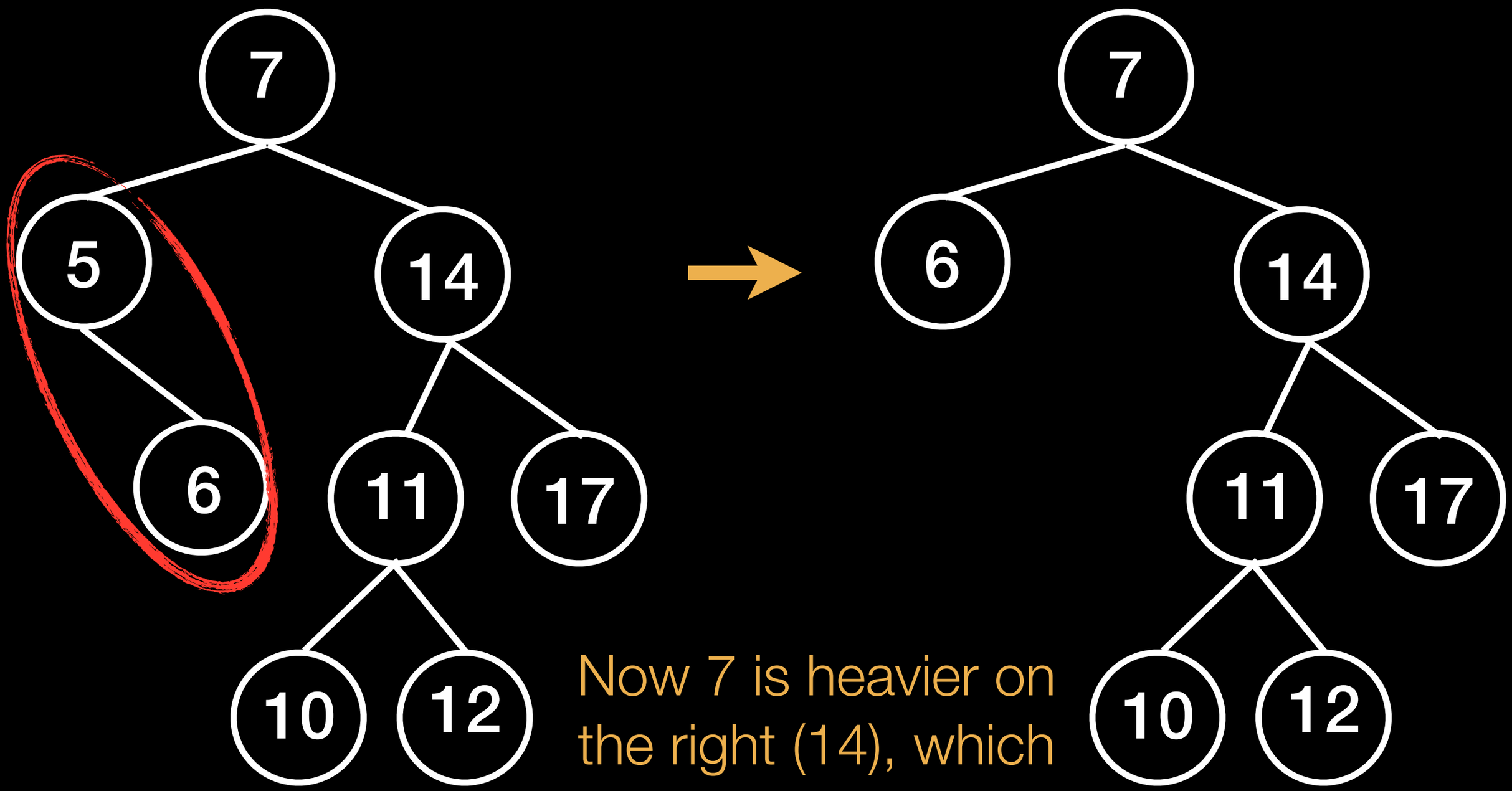
Remove 5



Remove 5

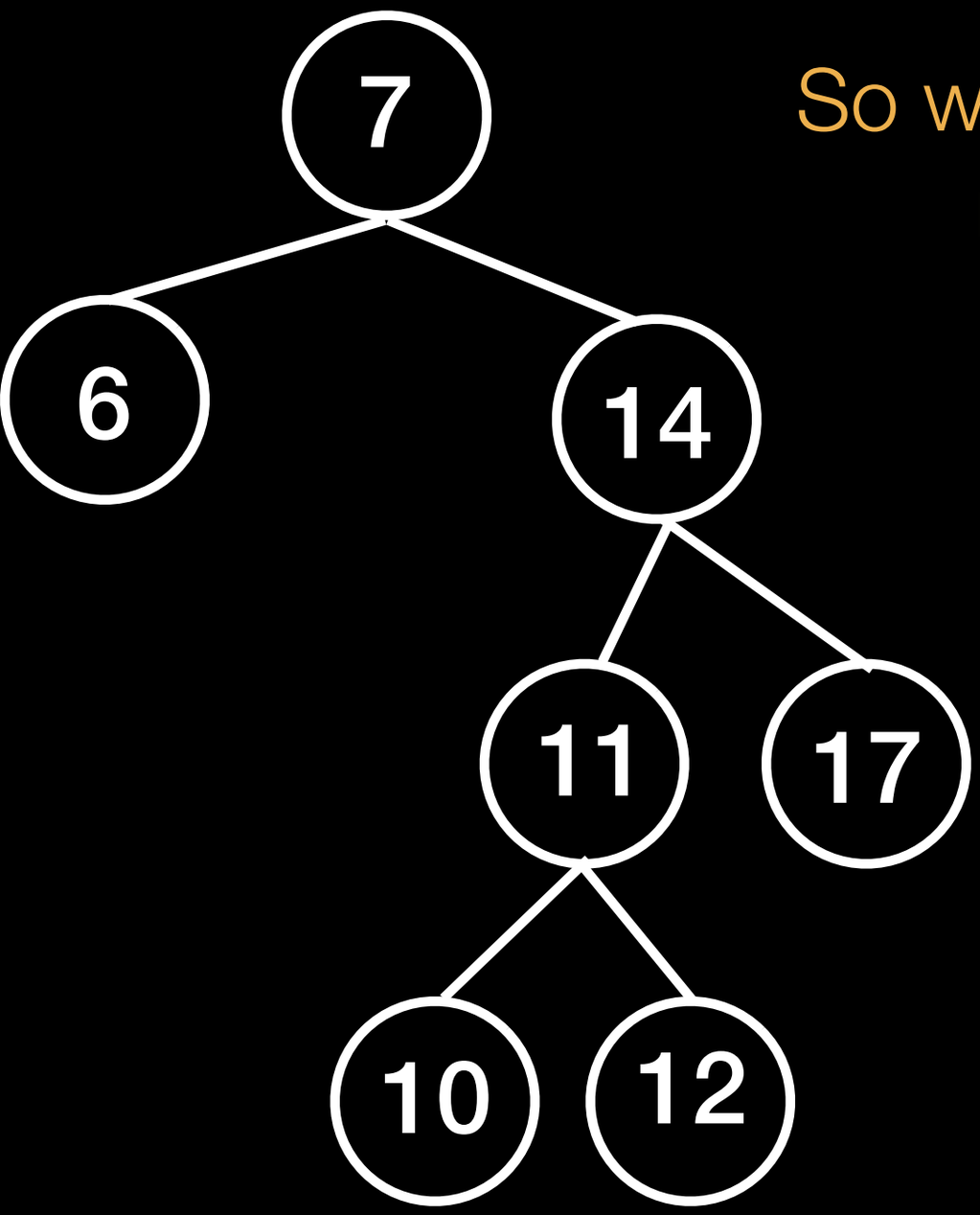


Remove 5



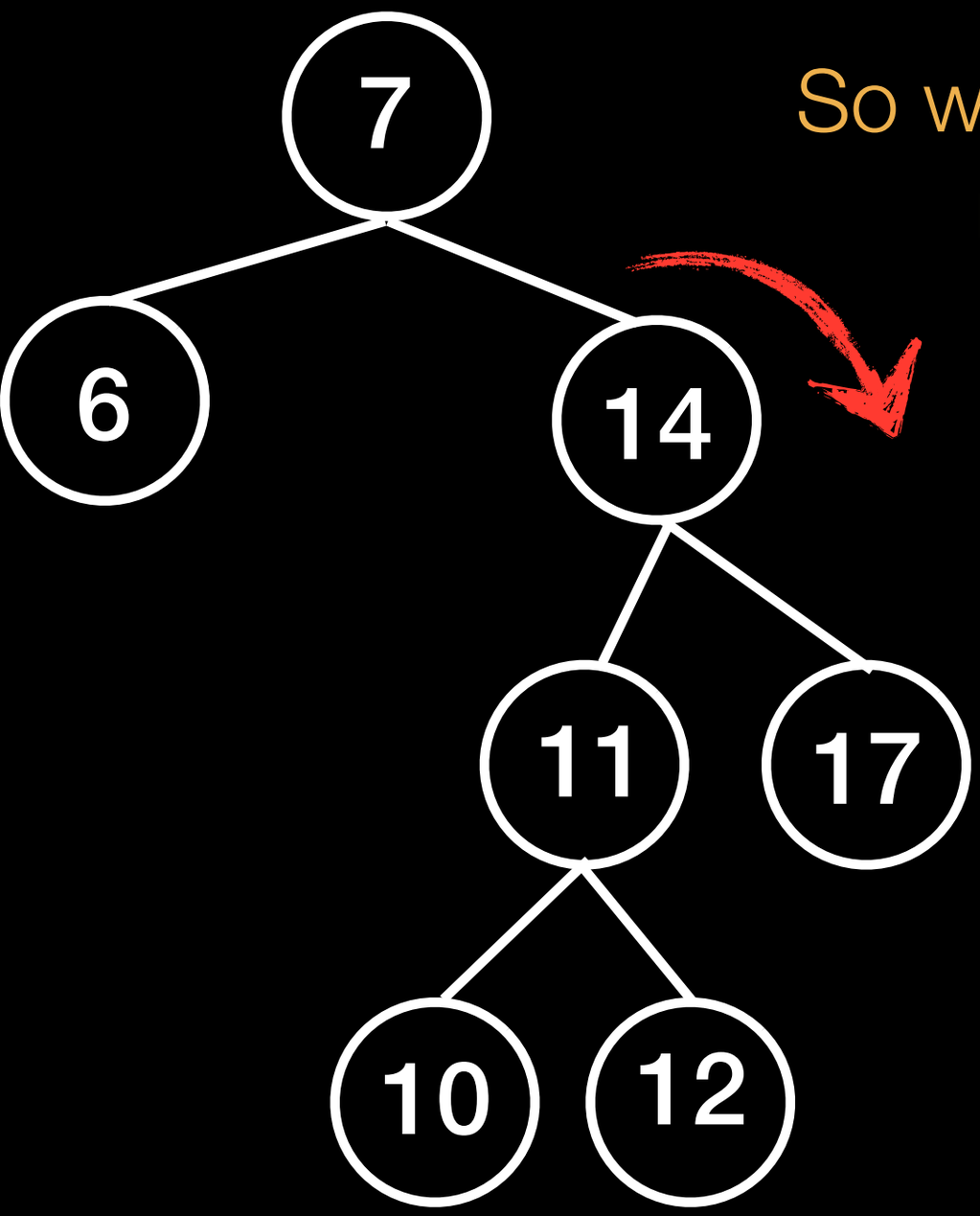
Now 7 is heavier on the right (14), which is heavier on the left. Need a **double rotation...**

Remove 5 (continued)



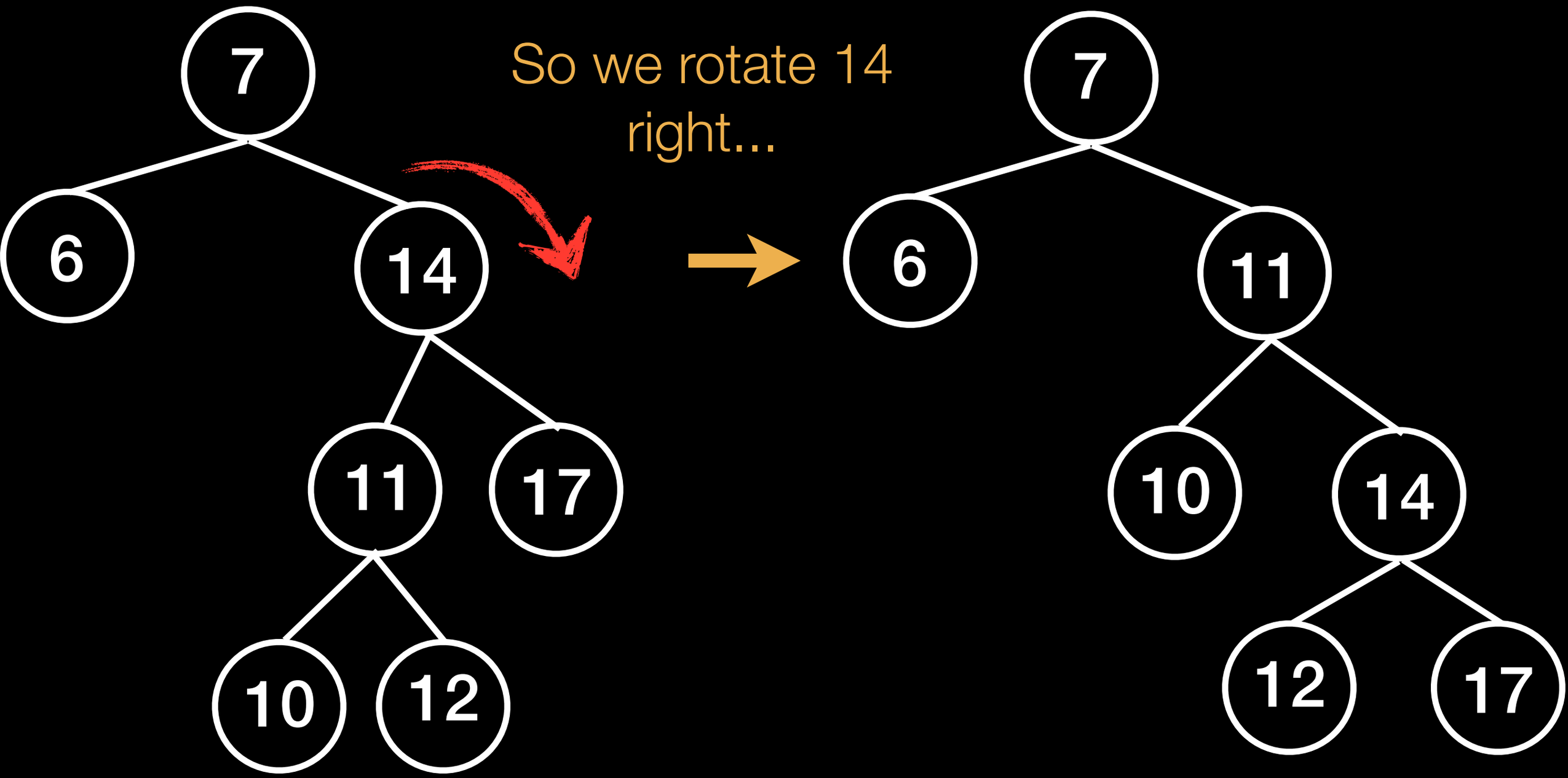
So we rotate 14 right...

Remove 5 (continued)

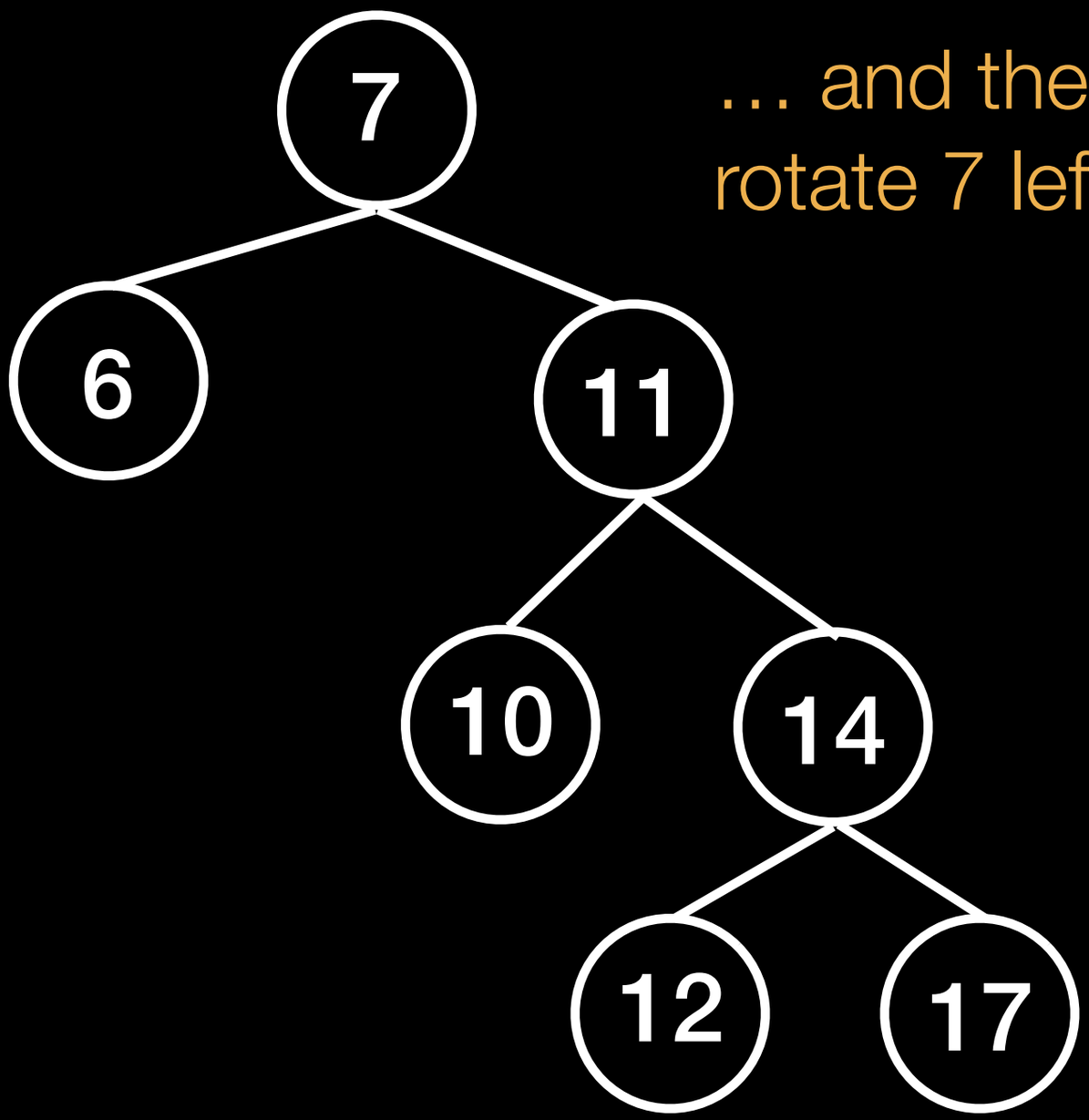


So we rotate 14 right...

Remove 5 (continued)

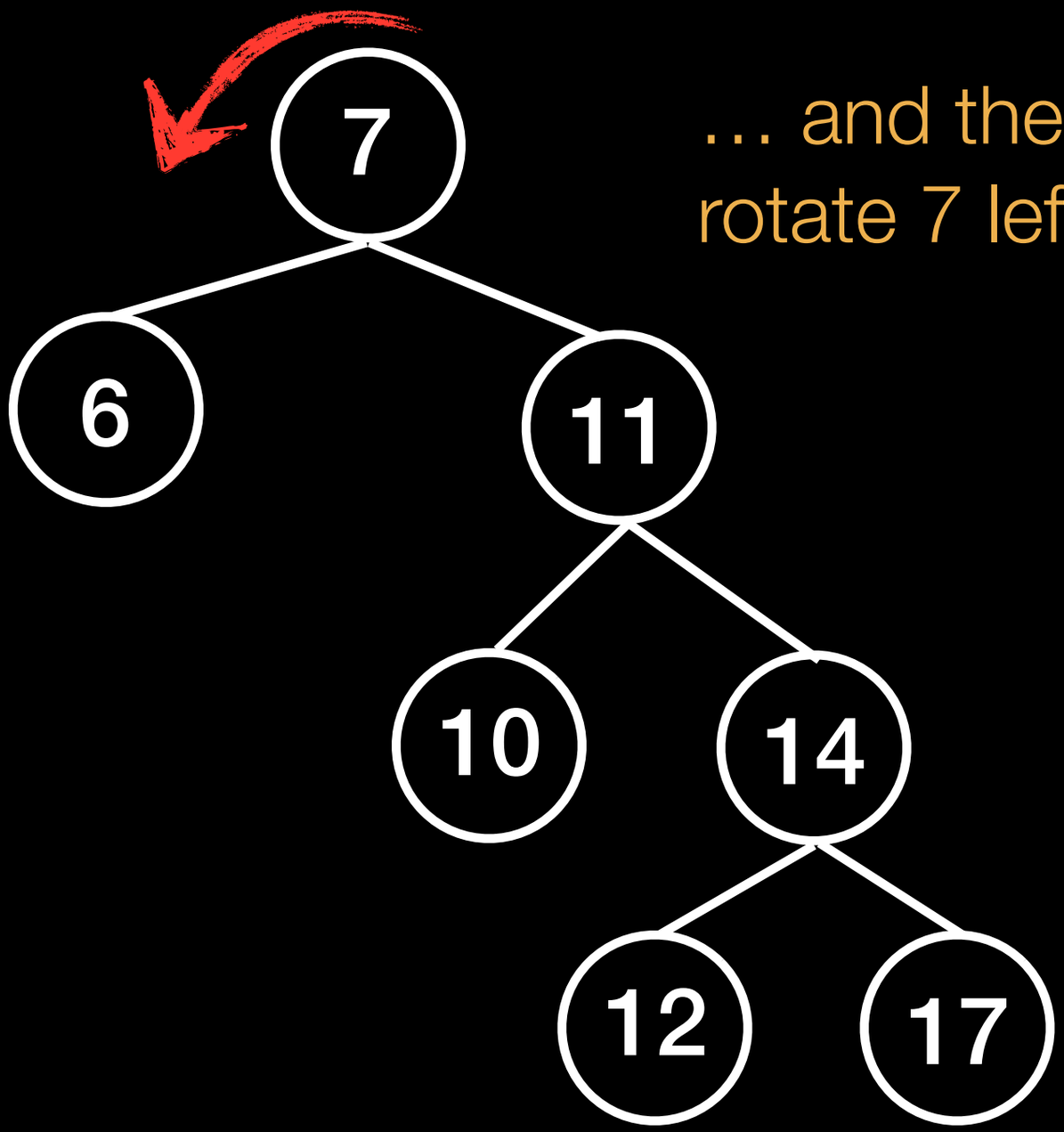


Remove 5 (continued)



… and then
rotate 7 left!

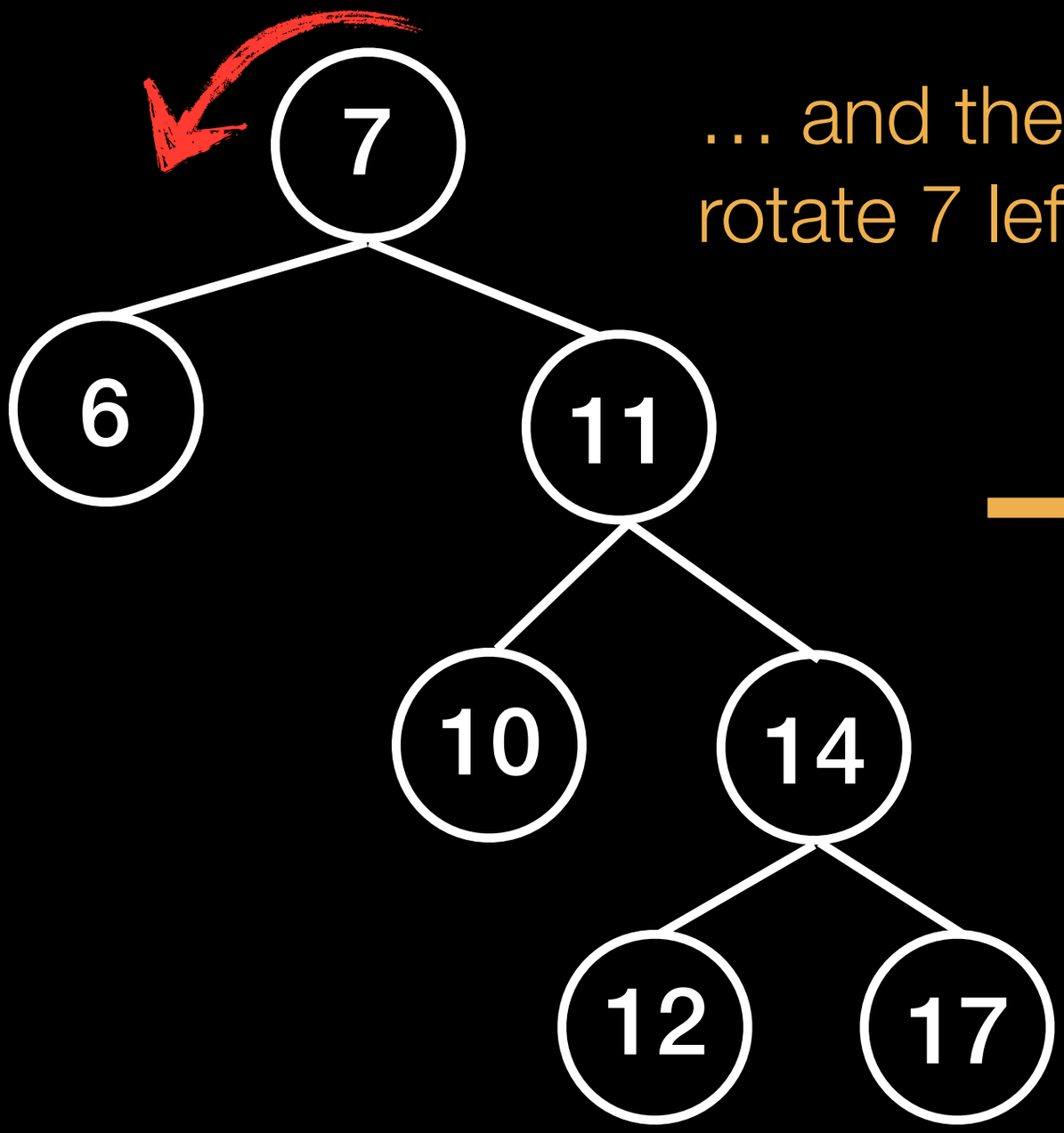
Remove 5 (continued)

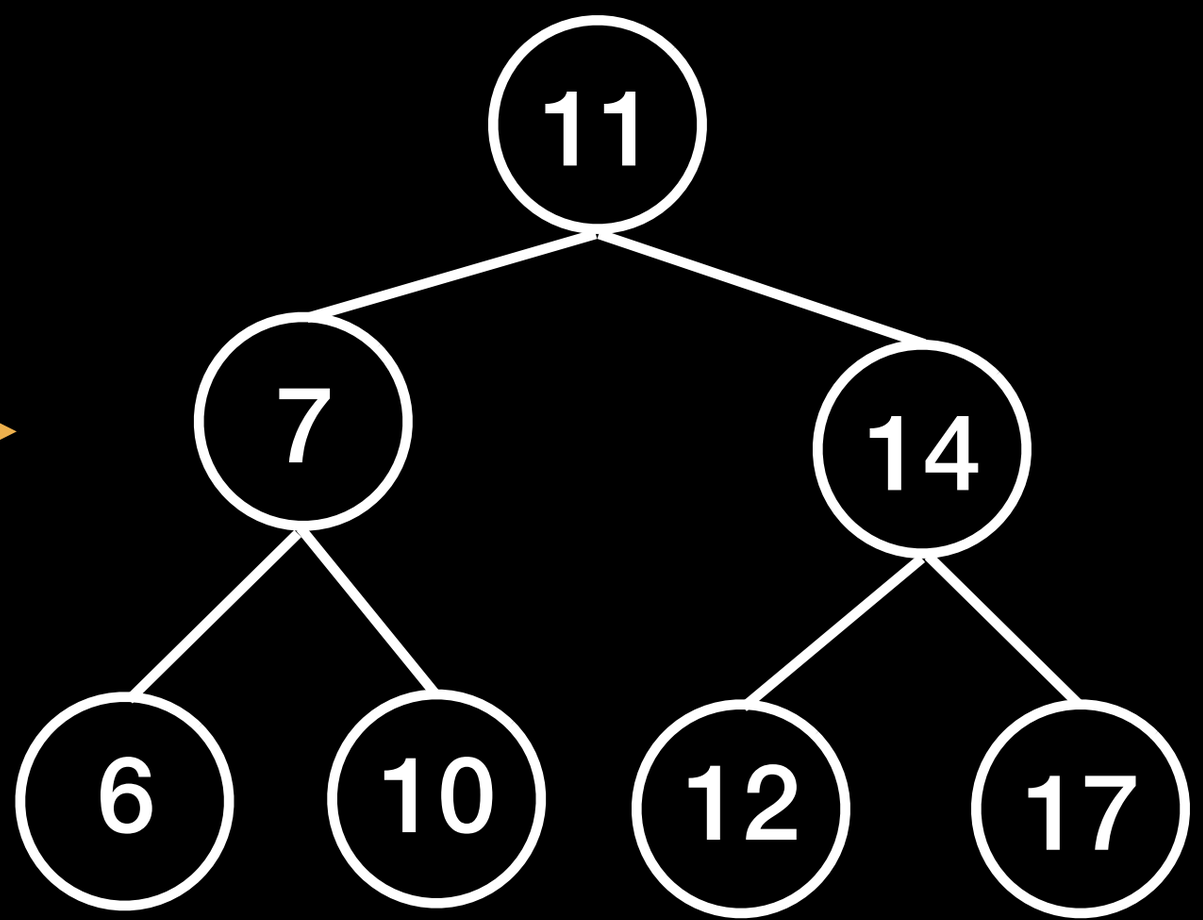


… and then
rotate 7 left!

Remove 5 (continued)

… and then
rotate 7 left!





When do we need to do a **double rotation?**

When do we need to do a **double rotation?**

1) The node is unbalanced

and

When do we need to do a **double rotation?**

1) The node is unbalanced
and

2) The node's balance factor is positive, but its right subtree's balance factor is negative,
or
The node's balance factor is negative, but its left subtree's balance factor is positive

When do we need to do a **double rotation?**

1) The node is unbalanced
and

2) The node's balance factor is positive, but its right subtree's balance factor is negative,
or
The node's balance factor is negative, but its left subtree's balance factor is positive

Balance factor = height(right subtree) - height(left subtree)

How do we represent a binary heap?

How do we represent a binary heap?

An array

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

2 * *i* + 1 and 2 * *i* + 2

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

2 * *i* + 1 and 2 * *i* + 2

What is the index of the parent of node *i*?

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

2 * *i* + 1 and 2 * *i* + 2

What is the index of the parent of node *i*?

(*i* - 1) / 2

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

2 * *i* + 1 and 2 * *i* + 2

What is the index of the parent of node *i*?

(*i* - 1) / 2

How do we add a node to a heap?

How do we represent a binary heap?

An array

What are the indices for the children of node *i*?

2 * *i* + 1 and 2 * *i* + 2

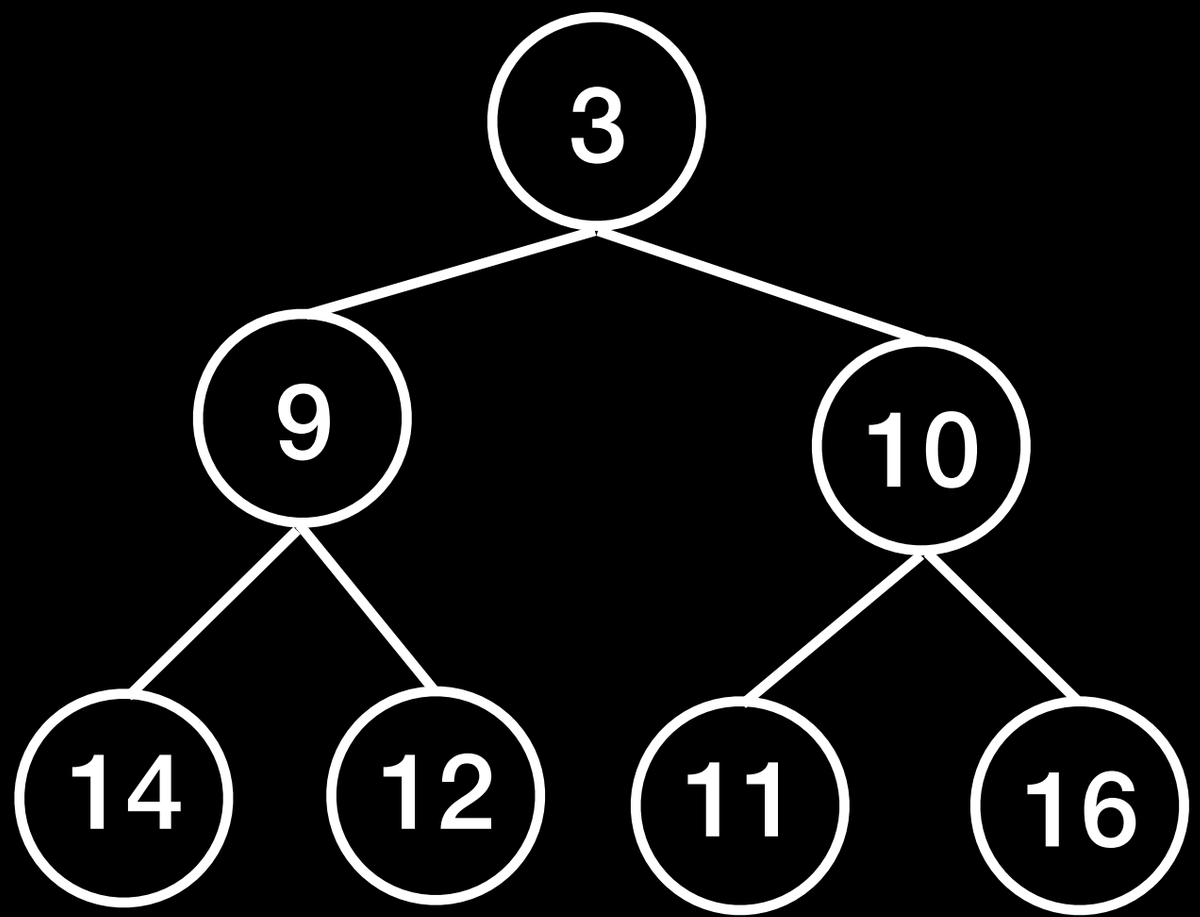
What is the index of the parent of node *i*?

(*i* - 1) / 2

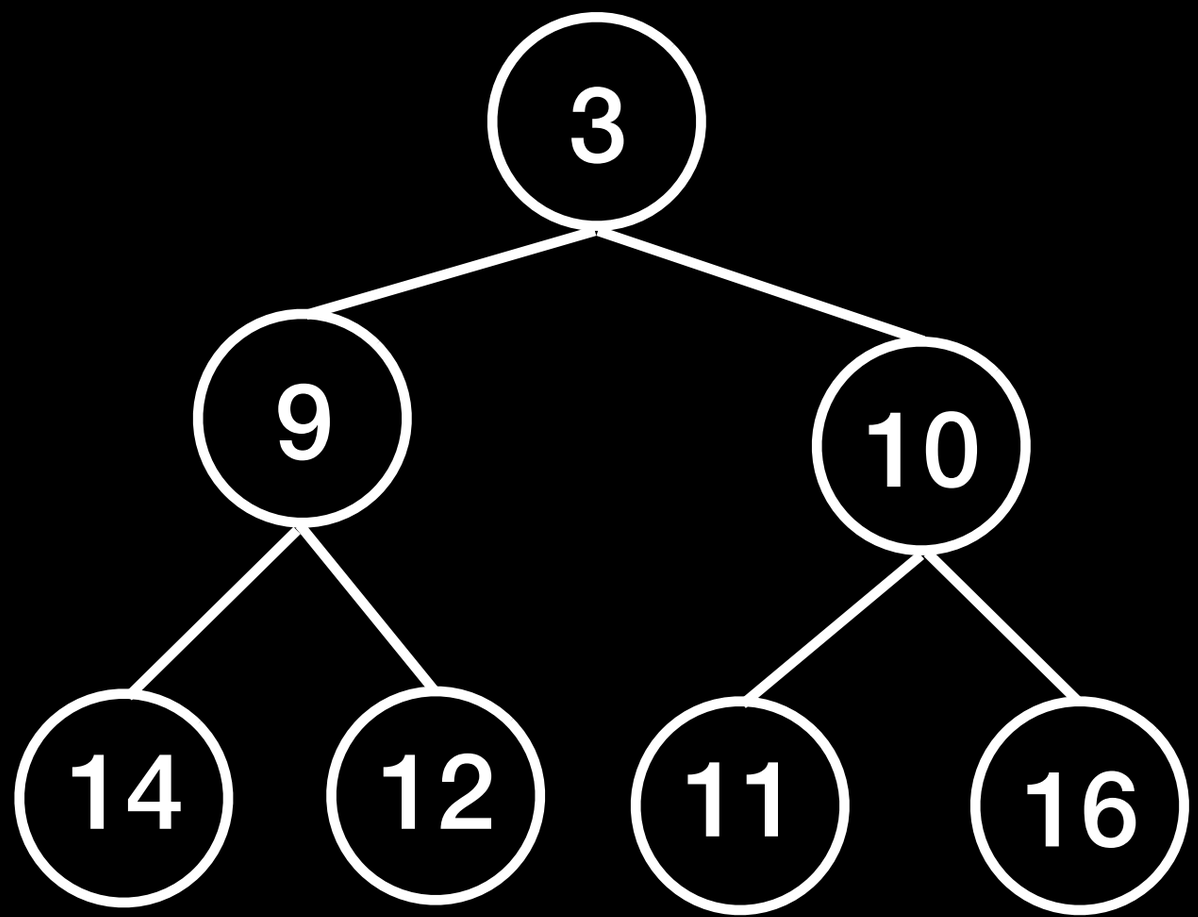
How do we add a node to a heap?

Insert it after the last item, then percolate it up

Simulate **heap sort** on this heap



Simulate **heap sort** on this heap



We’ll work this out on the chalk board

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

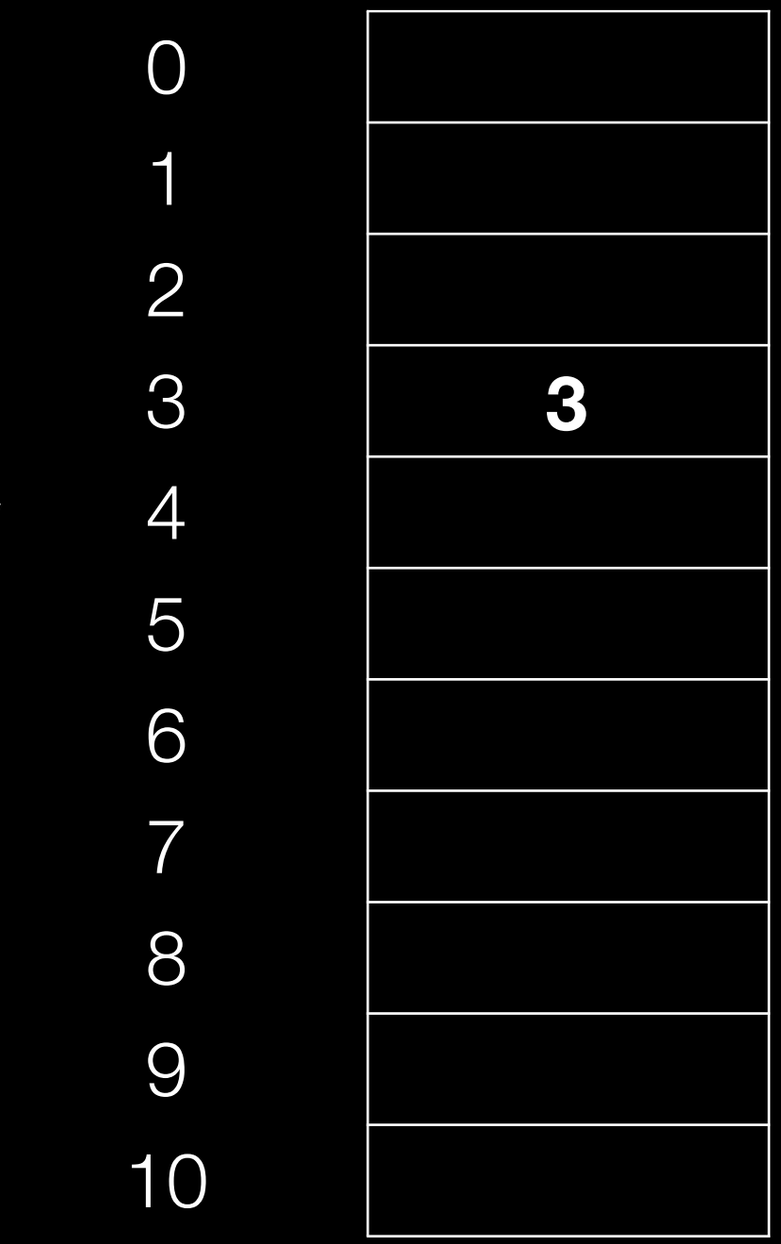
bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one



If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	
2	
3	**3** **14**
4	
5	
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	
2	
3	**3**
4	**14**
5	
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	
2	
3	**3** **25**
4	**14**
5	
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	
2	
3	**3**
4	**14**
5	**25**
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	**23**
2	
3	**3**
4	**14**
5	**25**
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11 44**
1	**23**
2	
3	**3**
4	**14**
5	**25**
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **open addressing?**

bucket = hash(x) % 11

If ‘bucket’ is in use, try the next one

0	**11**
1	**23**
2	**44**
3	**3**
4	**14**
5	**25**
6	
7	
8	**8**
9	
10	**43**

If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

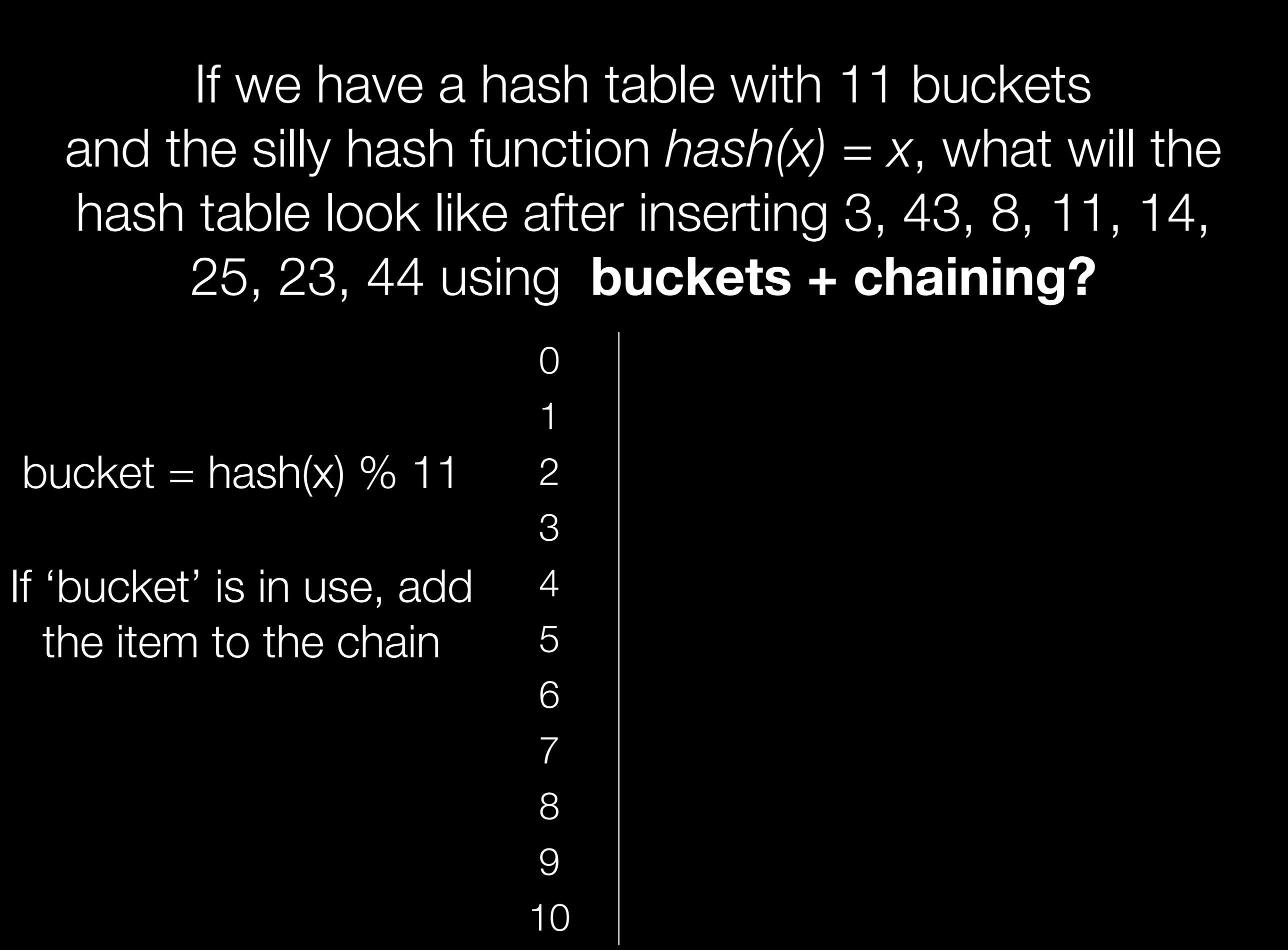
bucket = hash(x) % 11

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

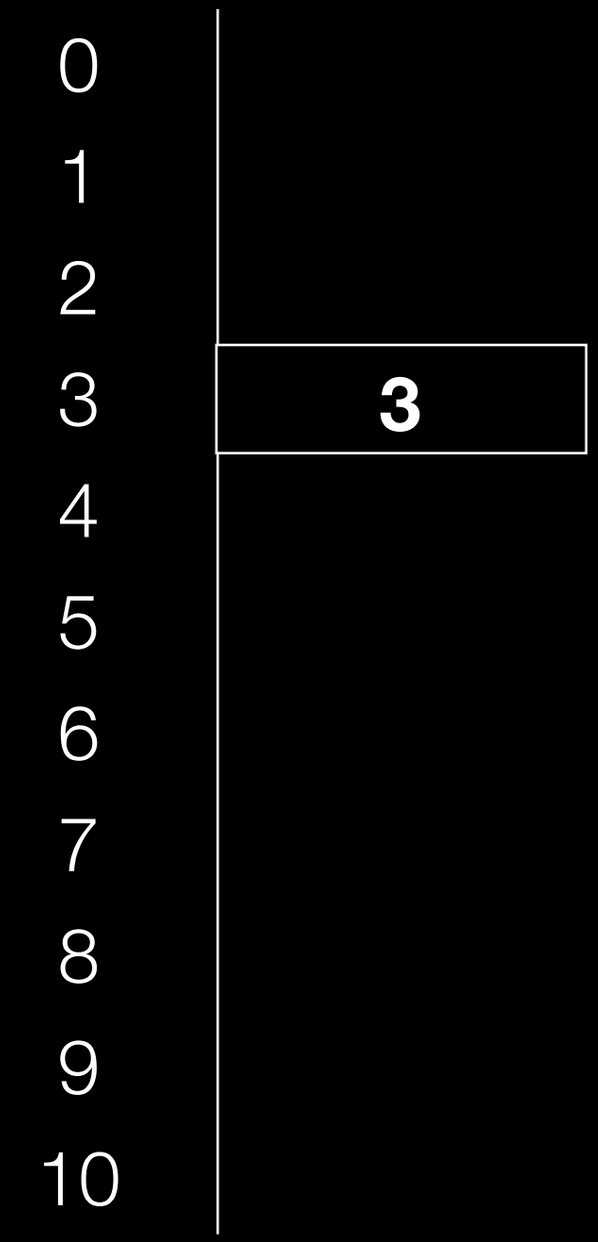
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

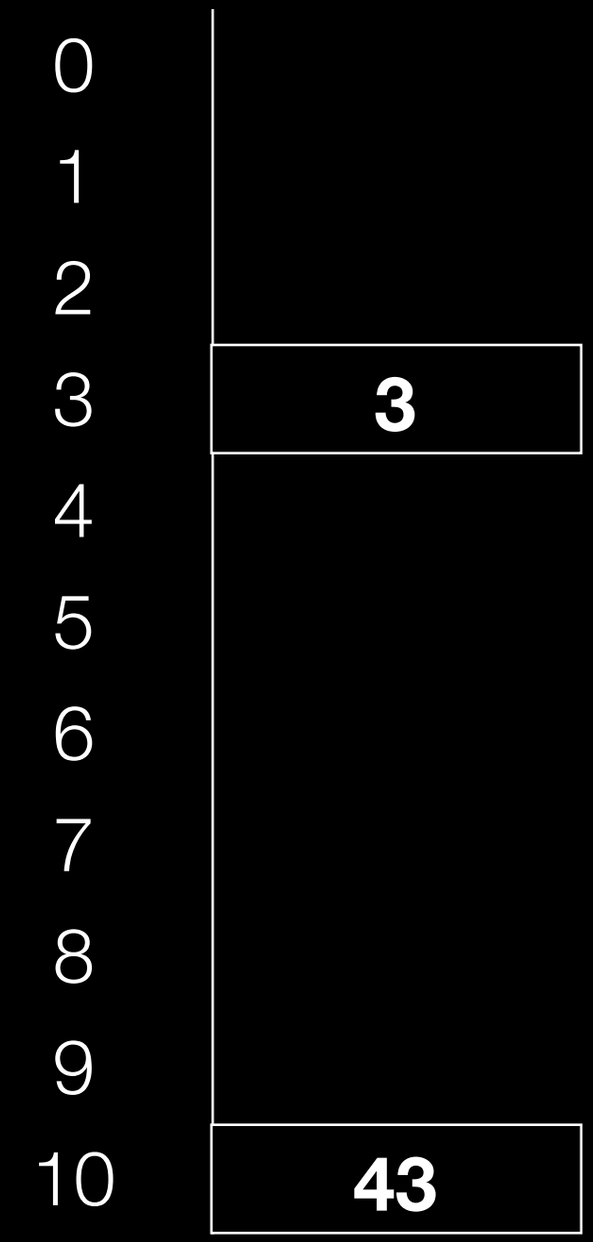
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

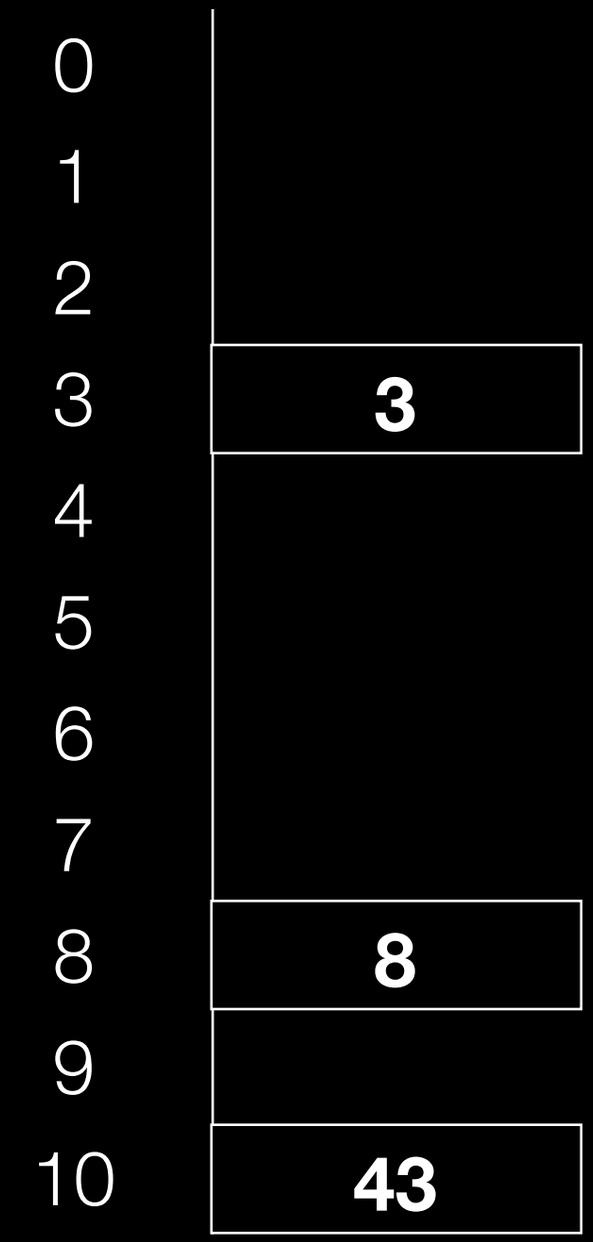
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

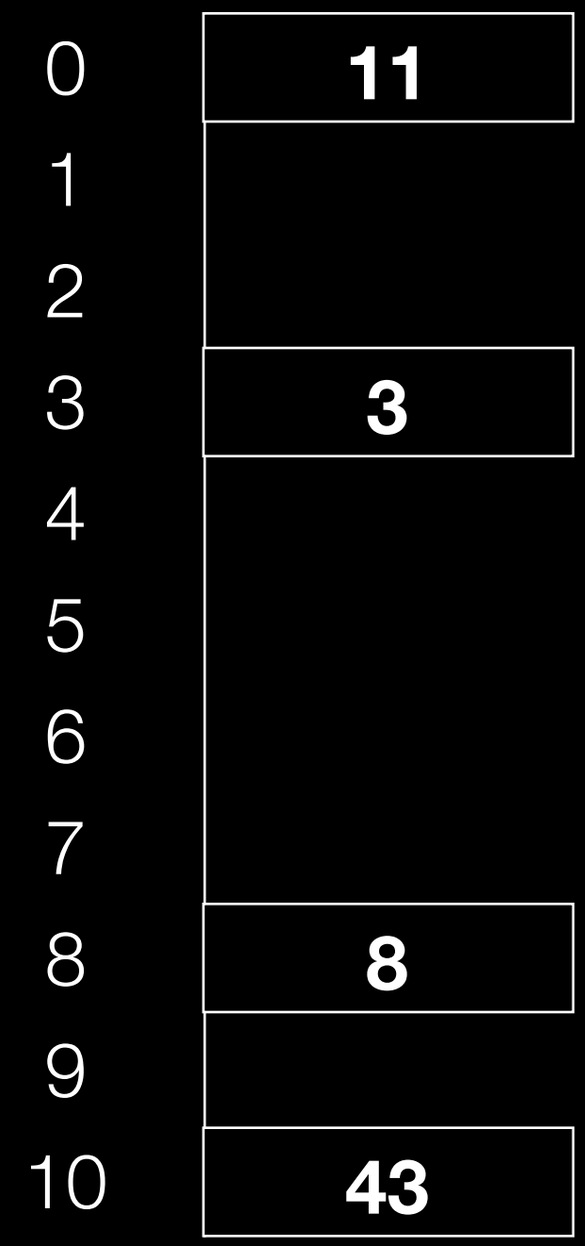
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

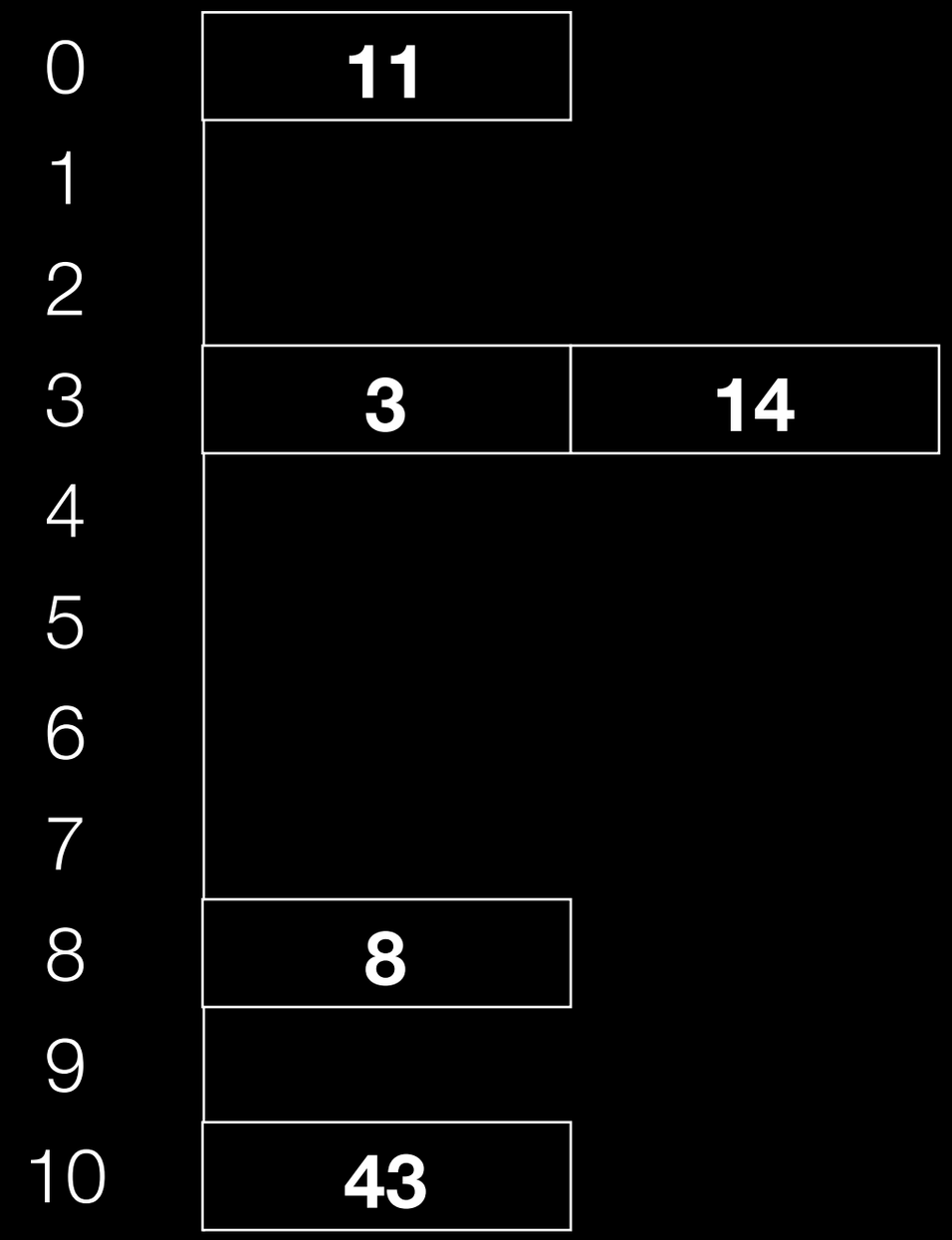
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

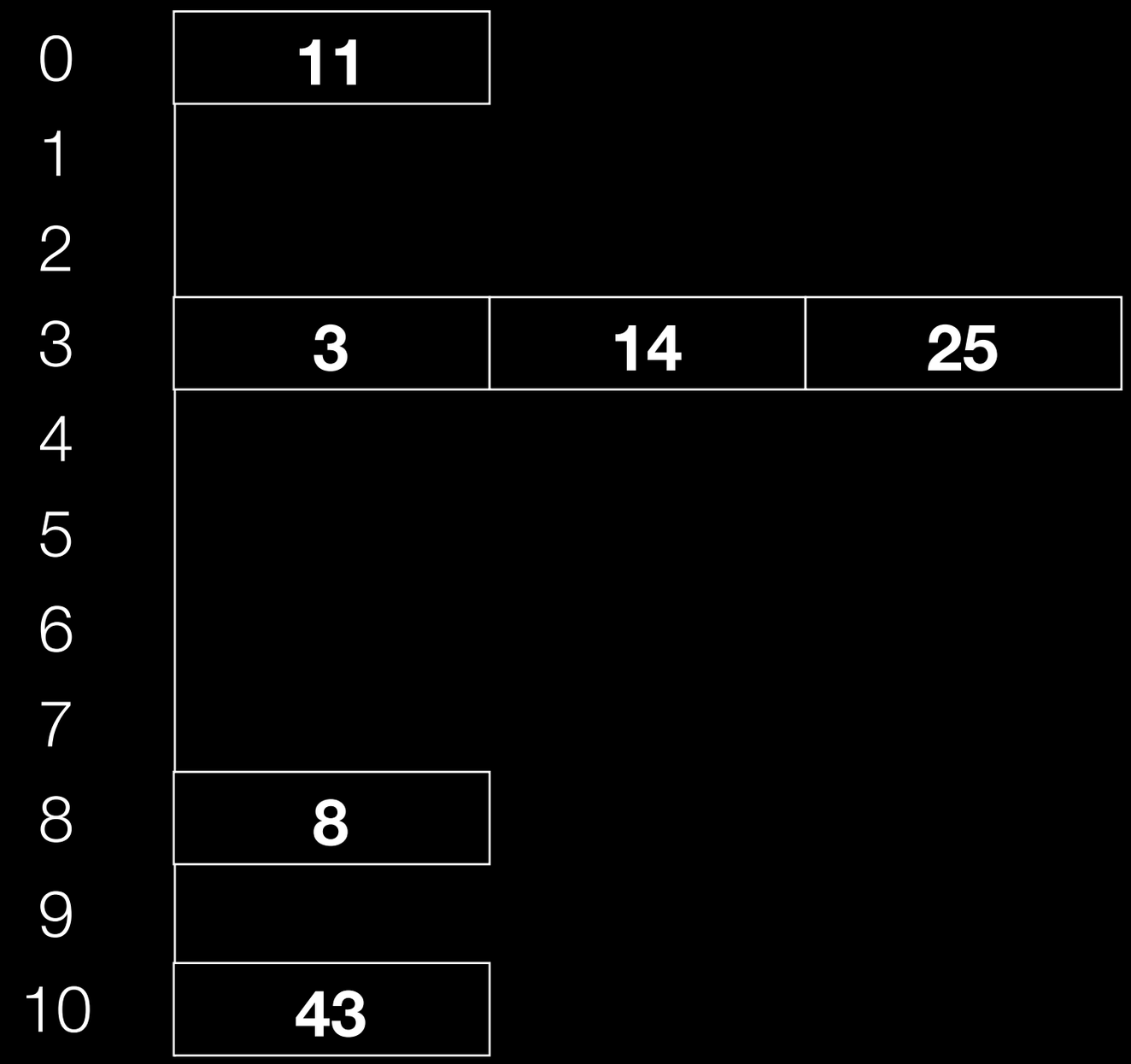
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

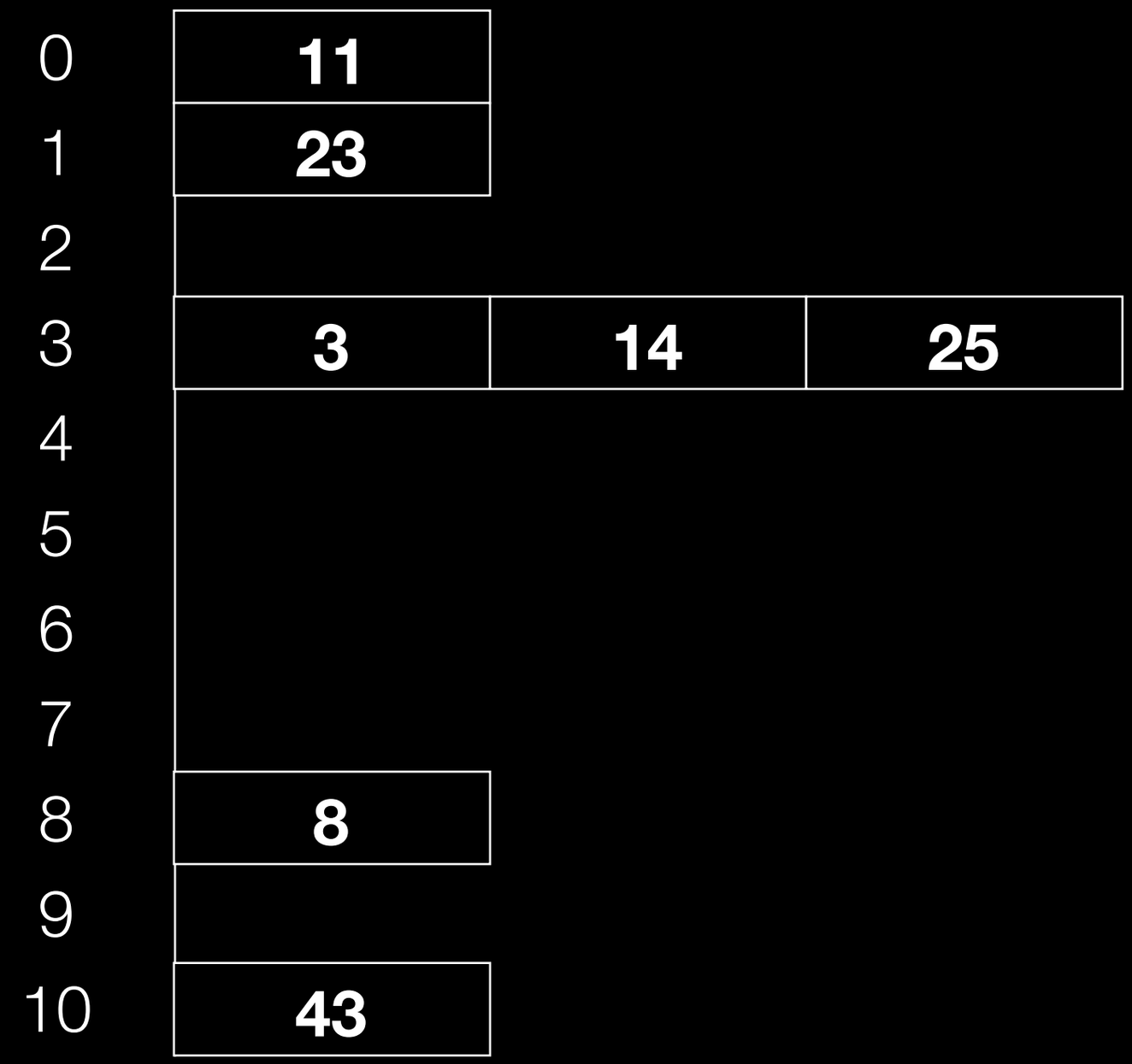
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

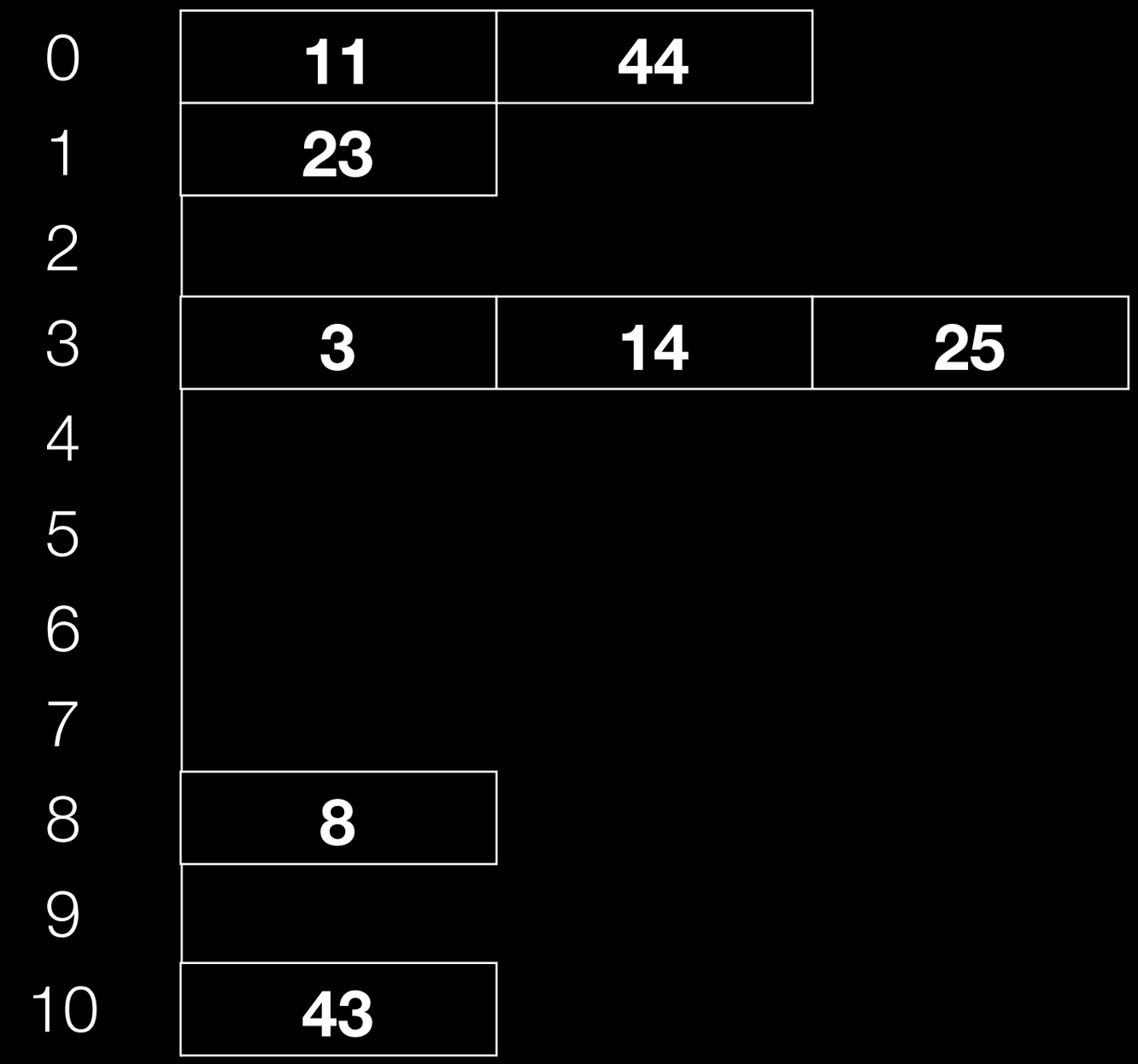
If ‘bucket’ is in use, add the item to the chain



If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

If ‘bucket’ is in use, add the item to the chain

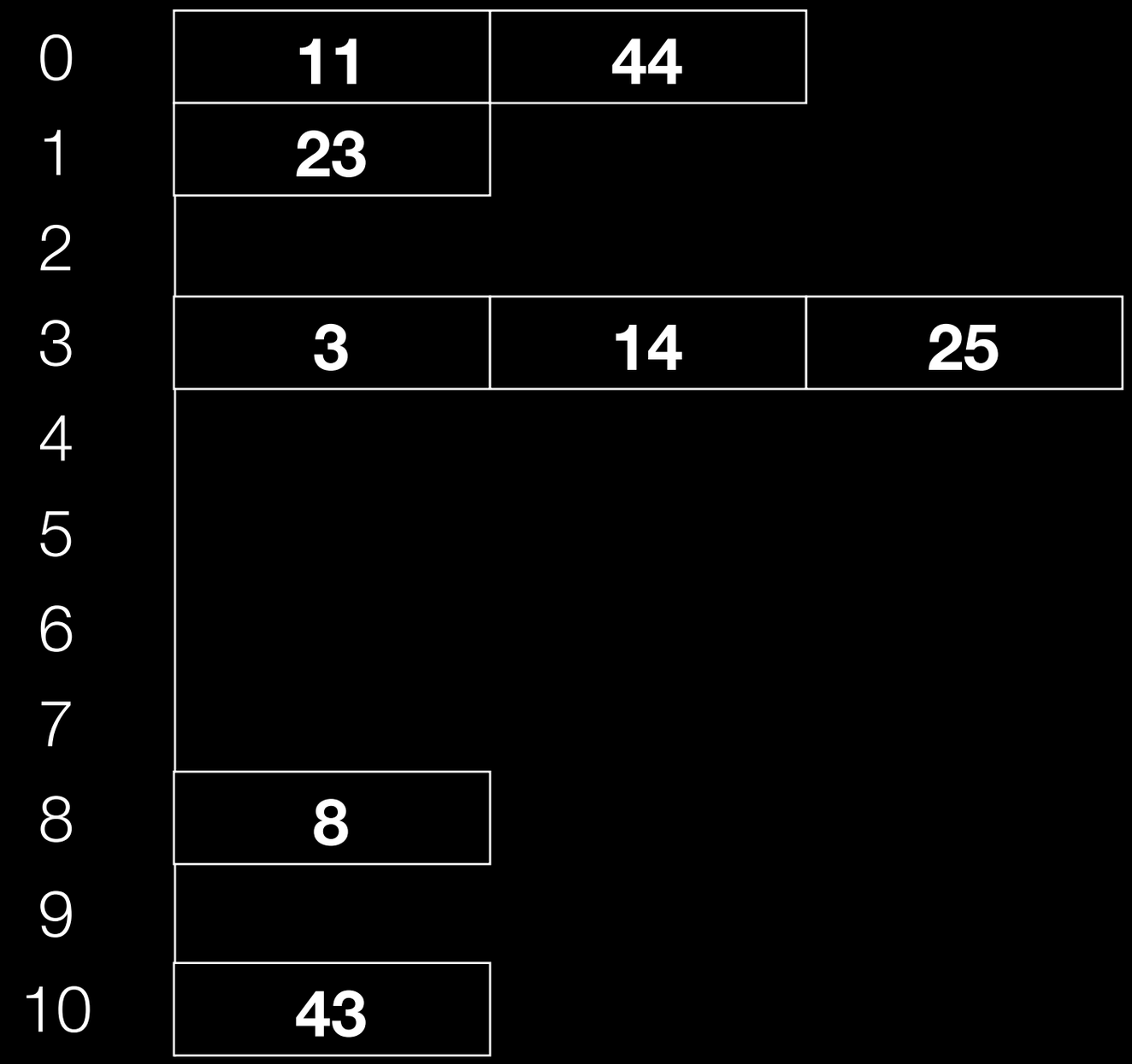


If we have a hash table with 11 buckets and the silly hash function *hash(x)* = *x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

If ‘bucket’ is in use, add the item to the chain

What is the **table load**?



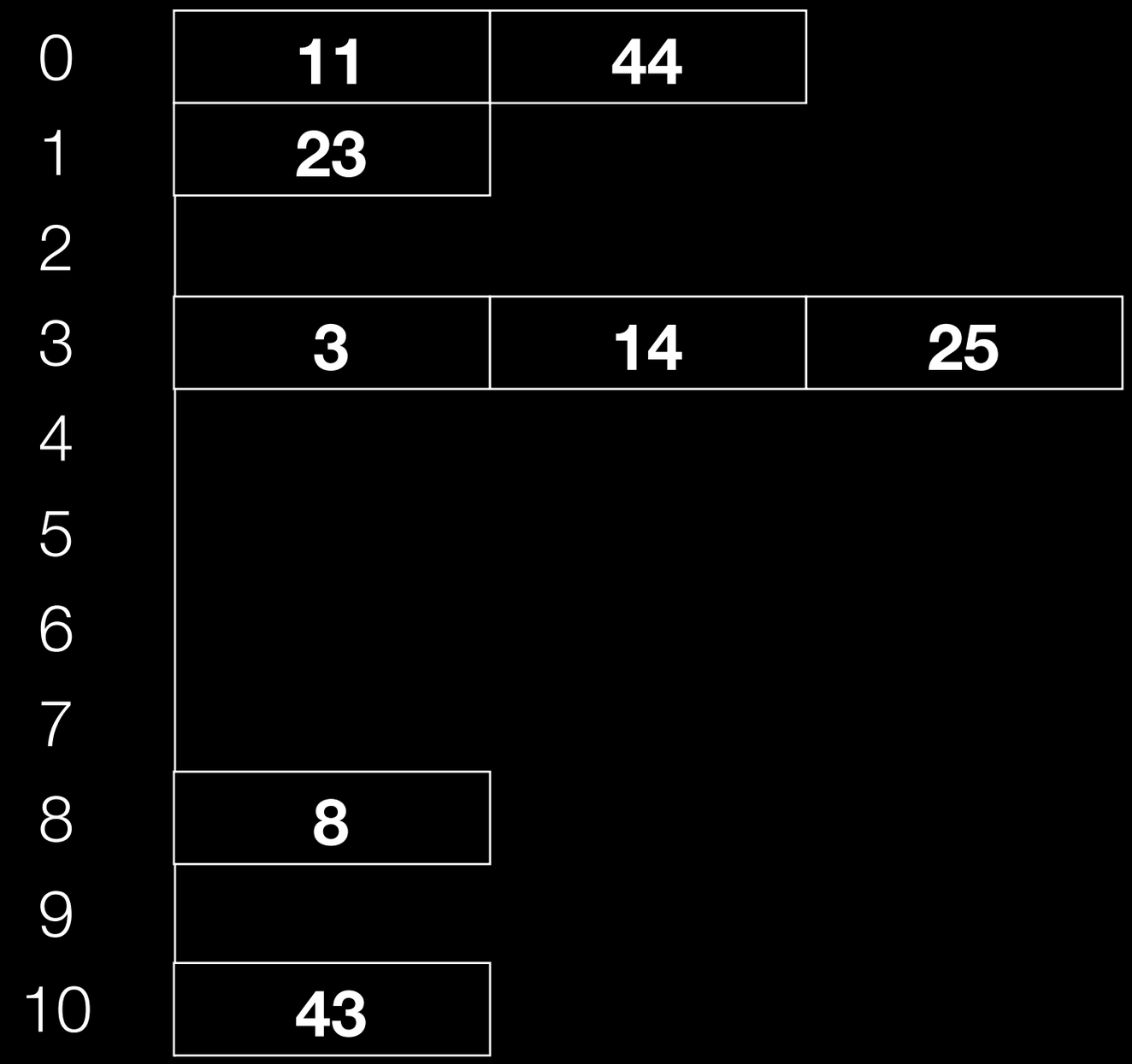
If we have a hash table with 11 buckets and the silly hash function *hash(x) = x*, what will the hash table look like after inserting 3, 43, 8, 11, 14, 25, 23, 44 using **buckets + chaining?**

bucket = hash(x) % 11

If ‘bucket’ is in use, add the item to the chain

What is the **table load**?

8 items / 11 buckets



Does every key in a hash table need to be unique?

Does every key in a hash table need to be unique?

Yes, that's the point of storing key/value pairs

Does every key in a hash table need to be unique?

Yes, that's the point of storing key/value pairs

Does each key in a hash table need to hash to a unique value?

Does every key in a hash table need to be unique?

Yes, that's the point of storing key/value pairs

Does each key in a hash table need to hash to a unique value?

No, but we should use a hash function with as few collisions as possible

Does every key in a hash table need to be unique?

Yes, that's the point of storing key/value pairs

Does each key in a hash table need to hash to a unique value?

No, but we should use a hash function with as few collisions as possible

Does hash table performance increase or decrease as the number of buckets increases?

Does every key in a hash table need to be unique?

Yes, that's the point of storing key/value pairs

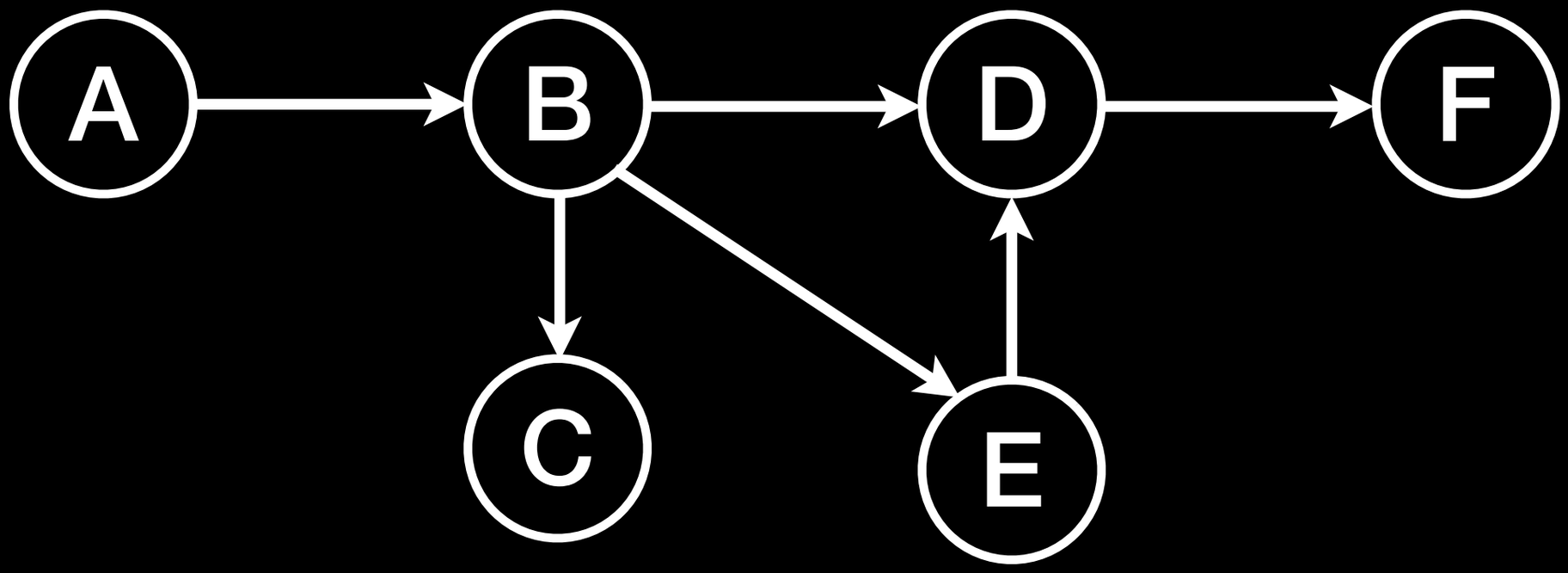
Does each key in a hash table need to hash to a unique value?

No, but we should use a hash function with as few collisions as possible

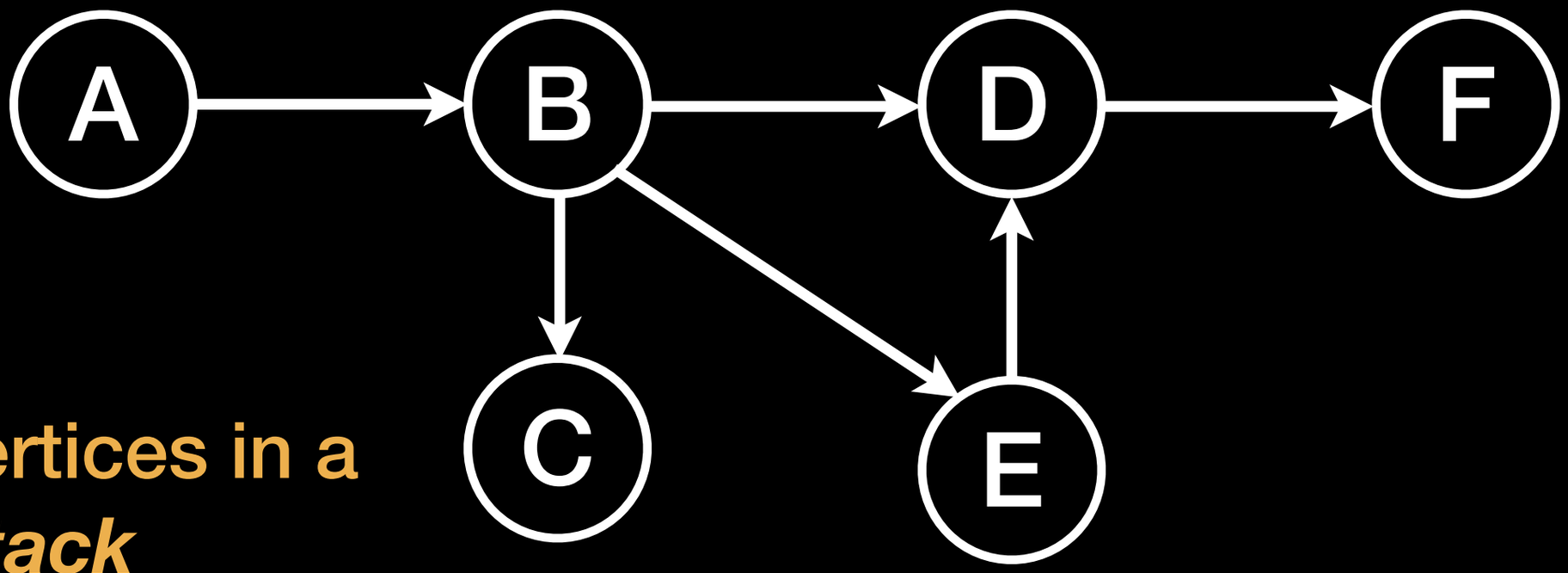
Does hash table performance increase or decrease as the number of buckets increases?

It should increase

Simulate **depth-first search** on this graph

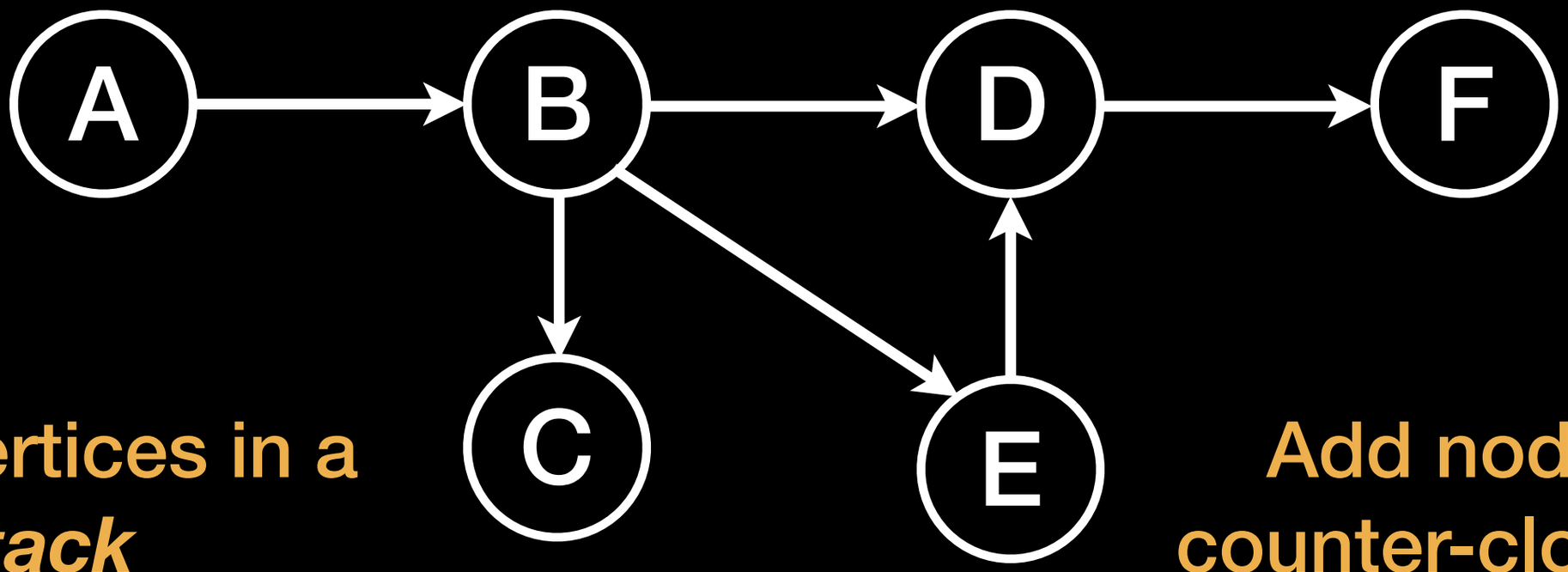


Simulate **depth-first search** on this graph



Store vertices in a *stack*
(last in, first out)

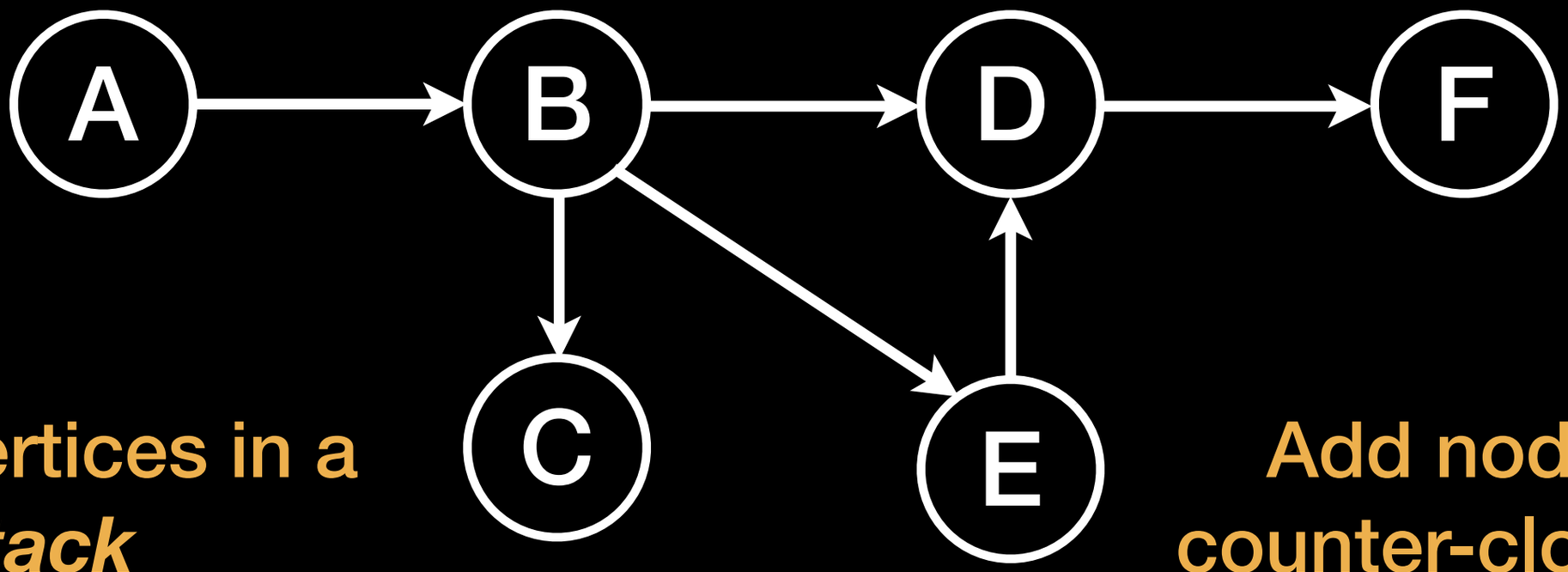
Simulate **depth-first search** on this graph



Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Simulate **depth-first search** on this graph



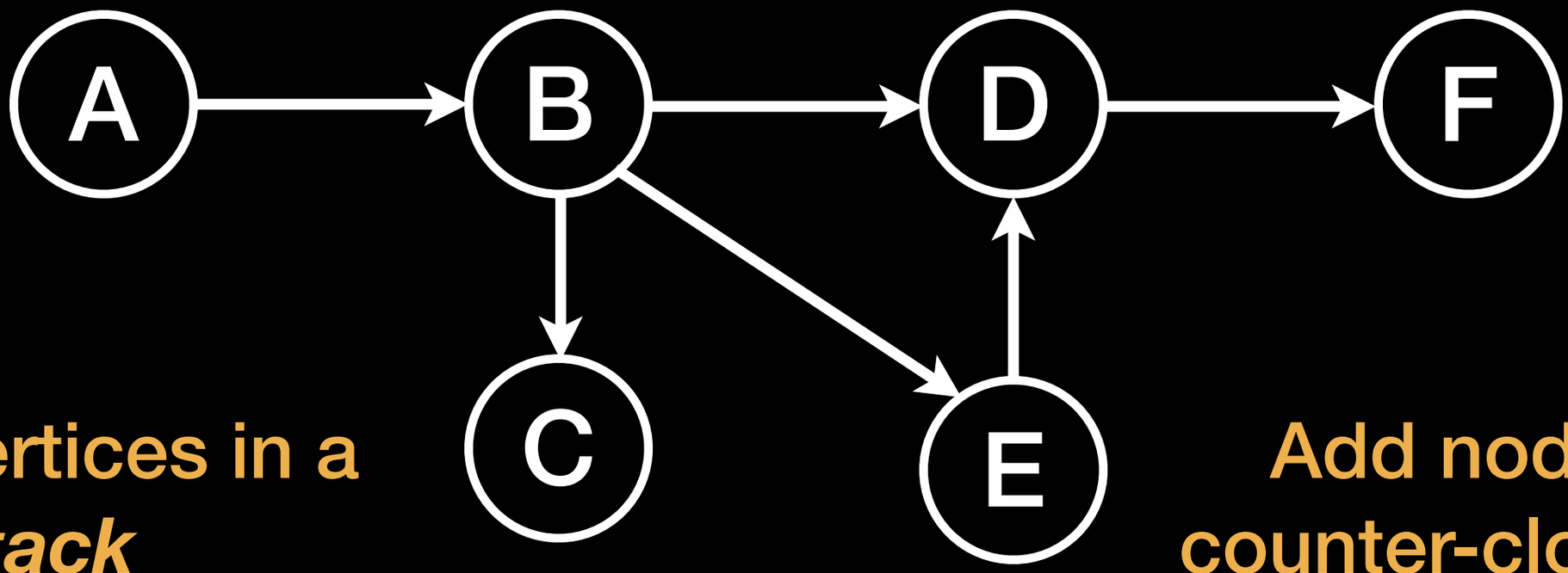
Store vertices in a ***stack*** (last in, first out)

Add nodes in counter-clockwise order

Reachable:

Known:

Simulate **depth-first search** on this graph



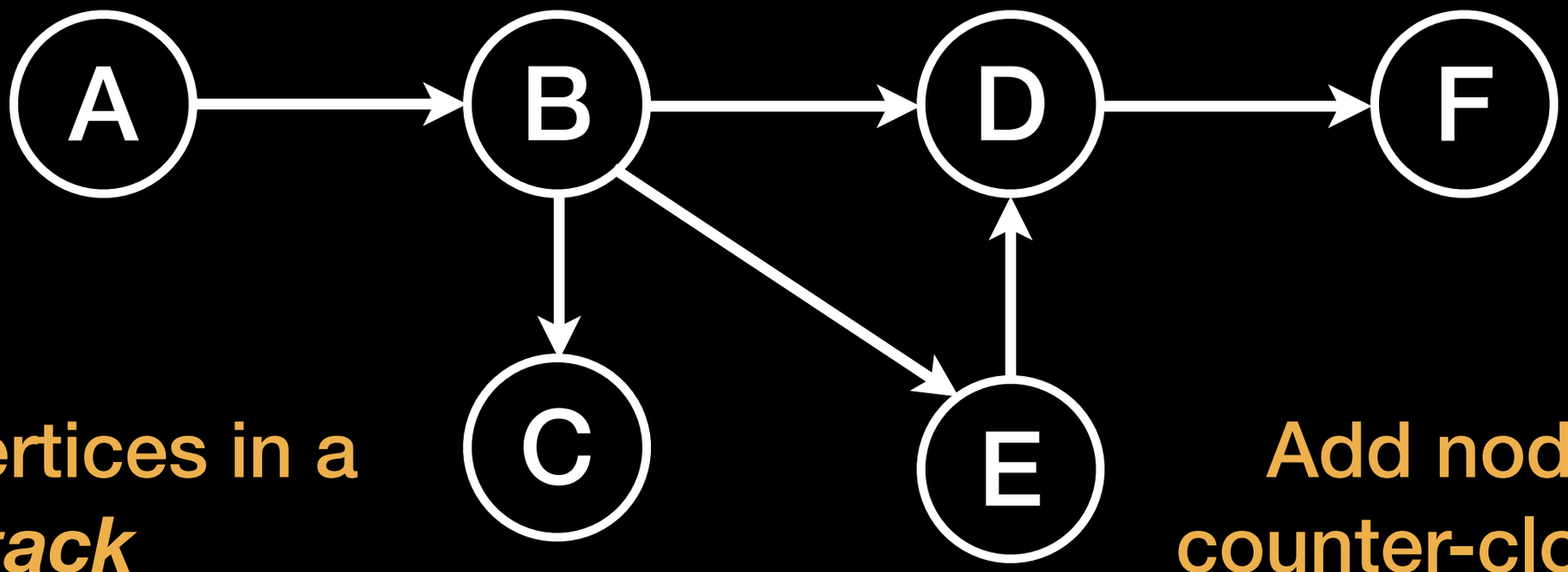
Store vertices in a ***stack*** (last in, first out)

Add nodes in counter-clockwise order

Reachable:

Known: A

Simulate **depth-first search** on this graph



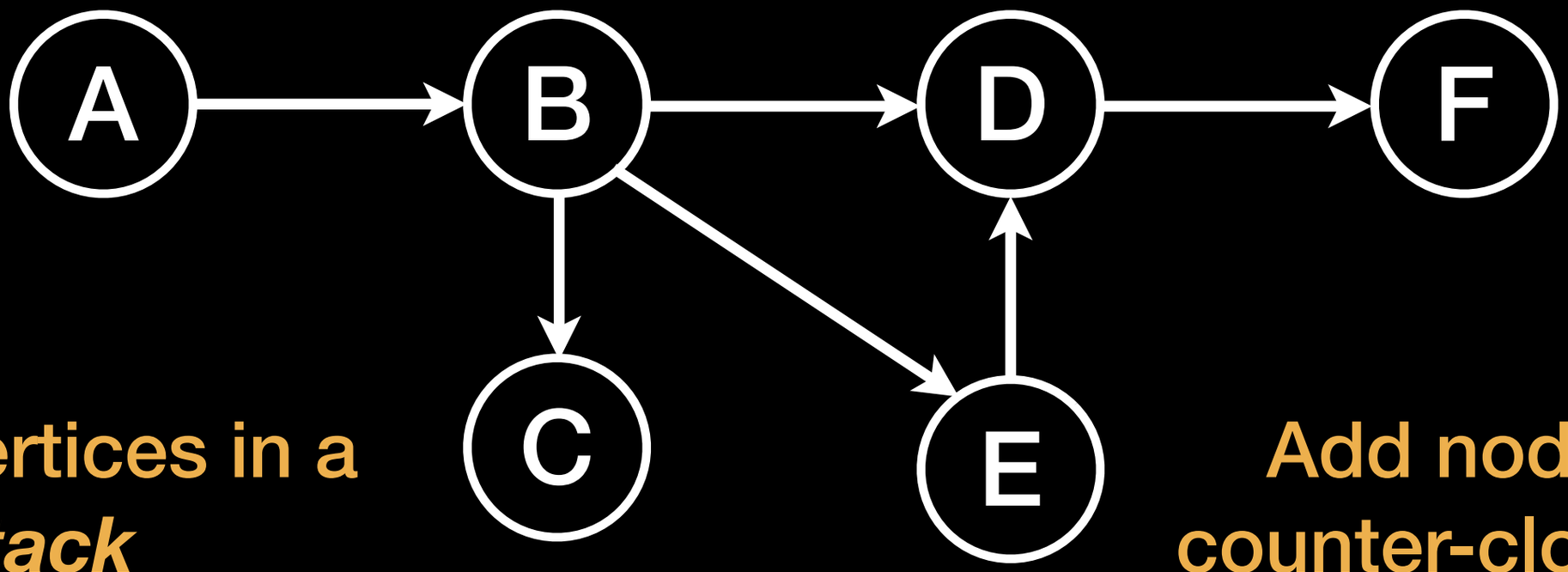
Store vertices in a ***stack*** (last in, first out)

Add nodes in counter-clockwise order

Reachable: A

Known:

Simulate **depth-first search** on this graph



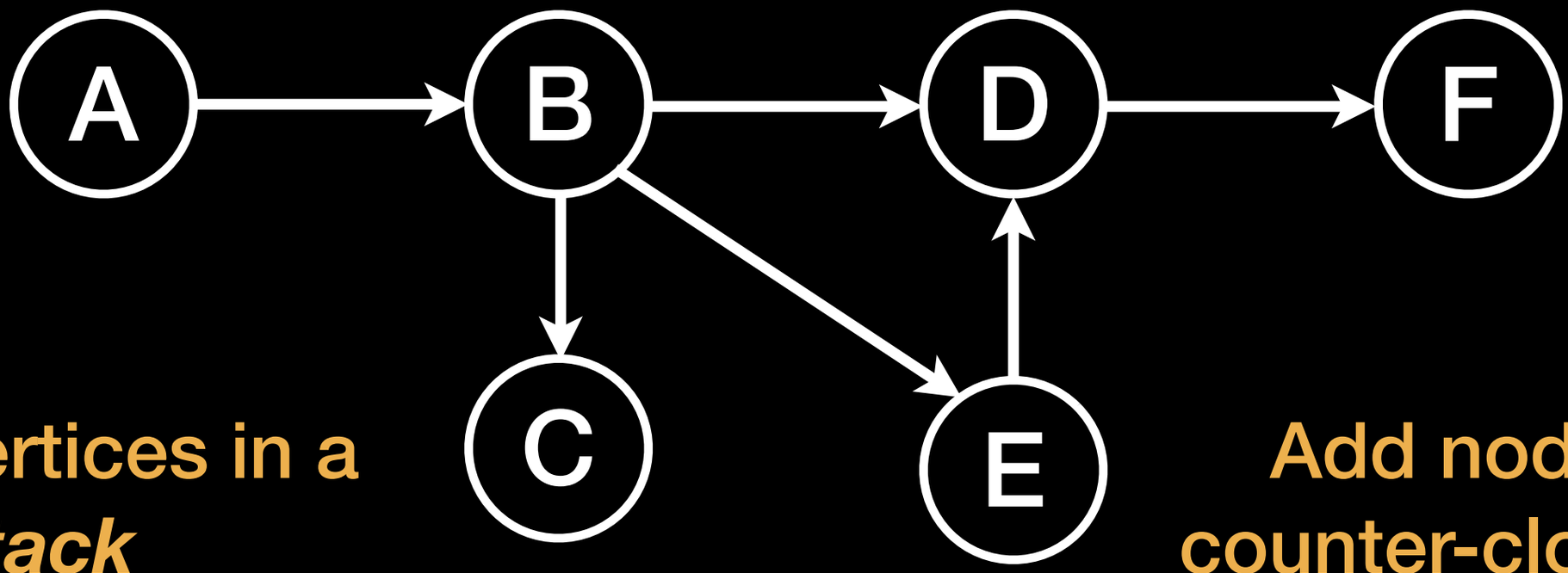
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A

Known: B

Simulate **depth-first search** on this graph



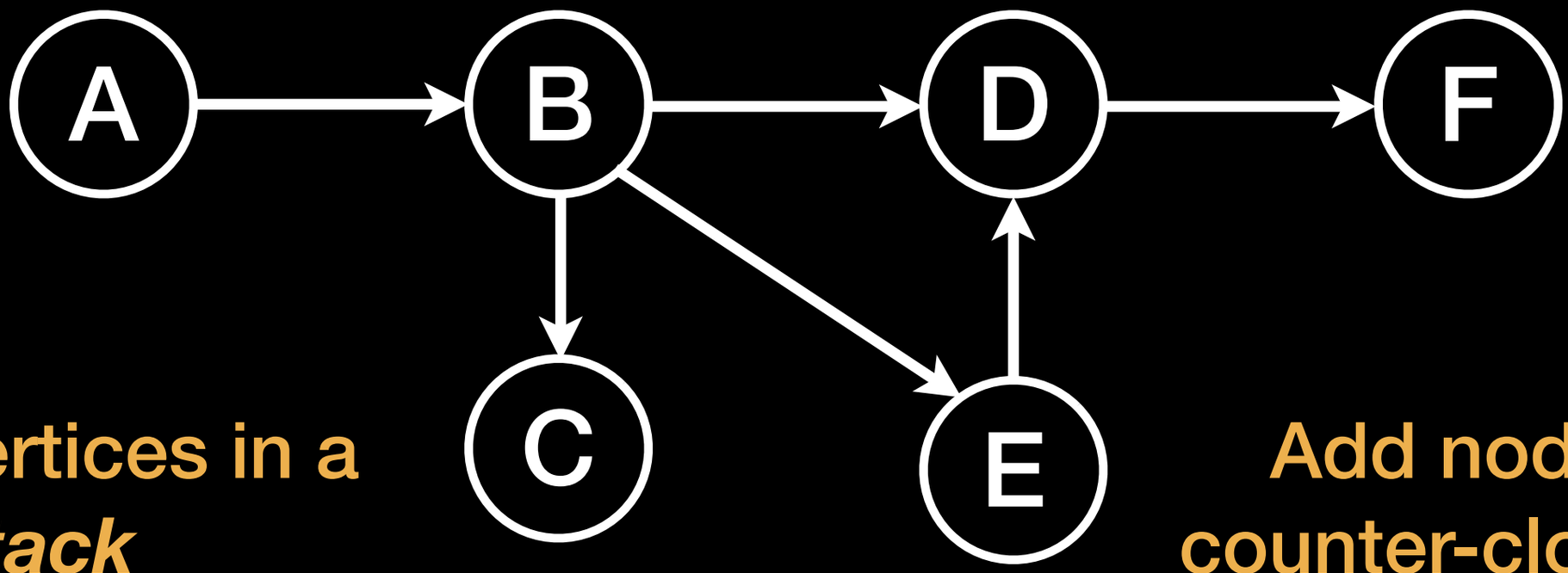
Store vertices in a ***stack*** (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known:

Simulate **depth-first search** on this graph



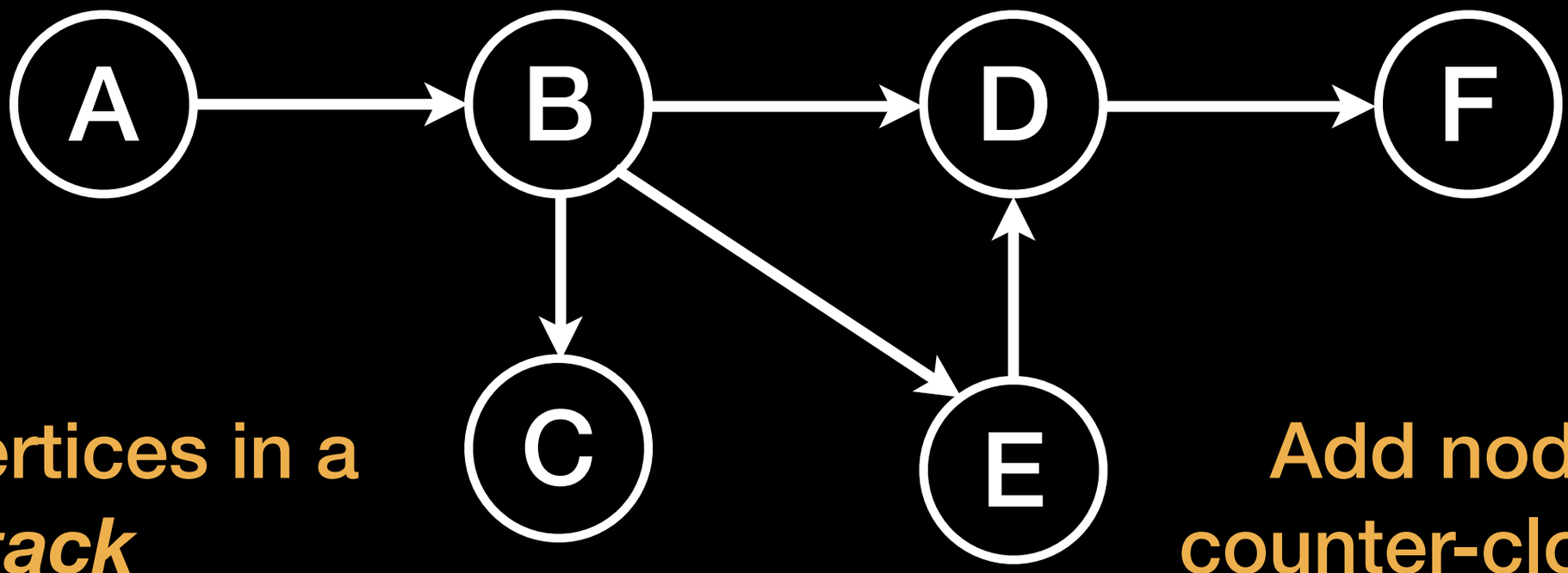
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C

Simulate **depth-first search** on this graph



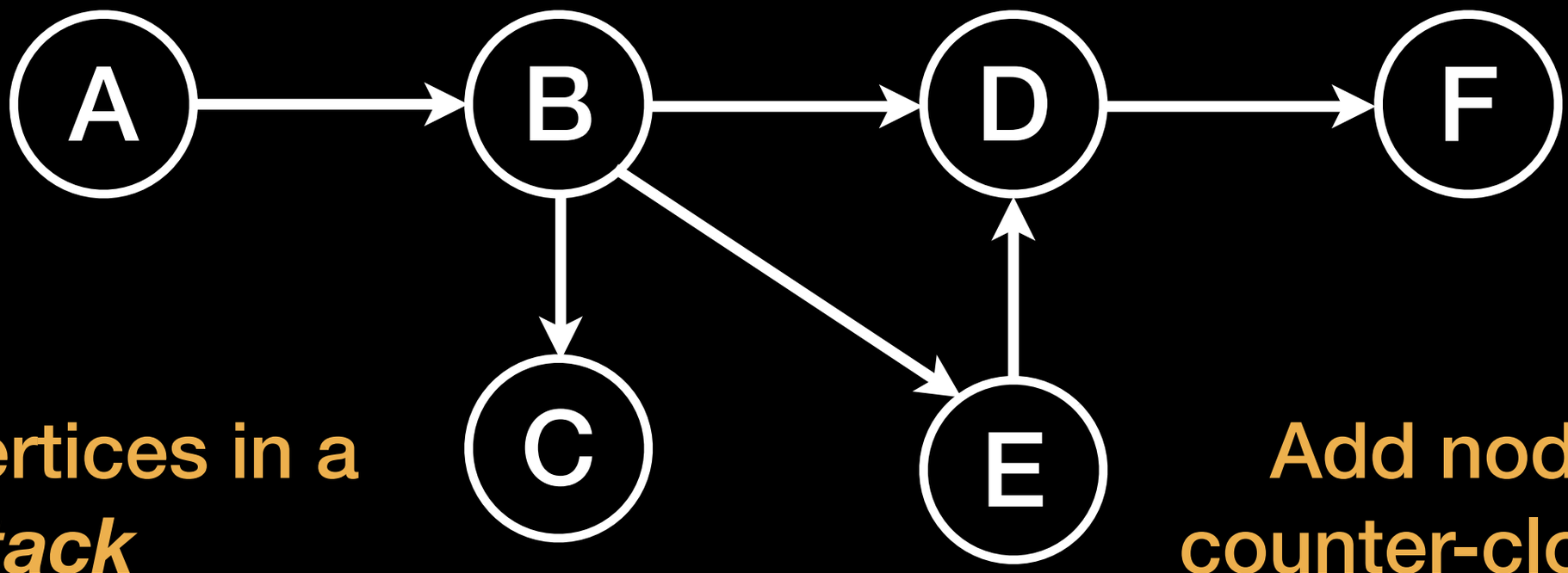
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C E

Simulate **depth-first search** on this graph



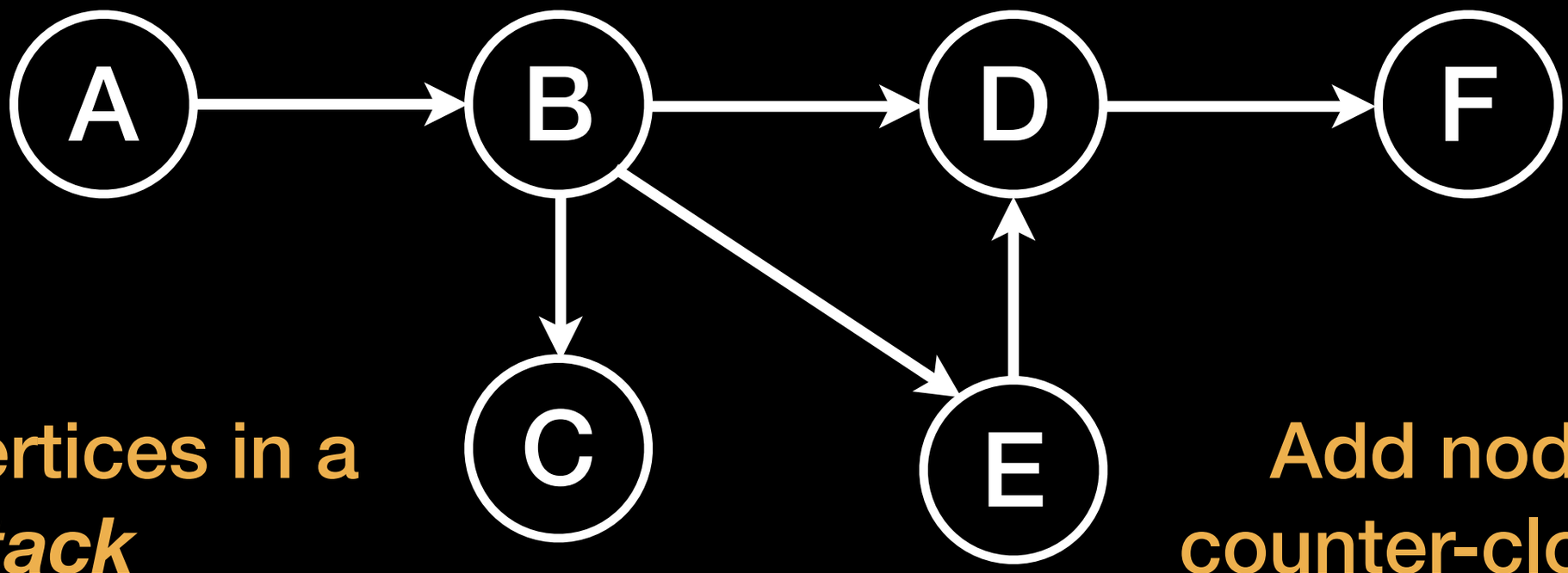
Store vertices in a ***stack*** (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C E D

Simulate **depth-first search** on this graph



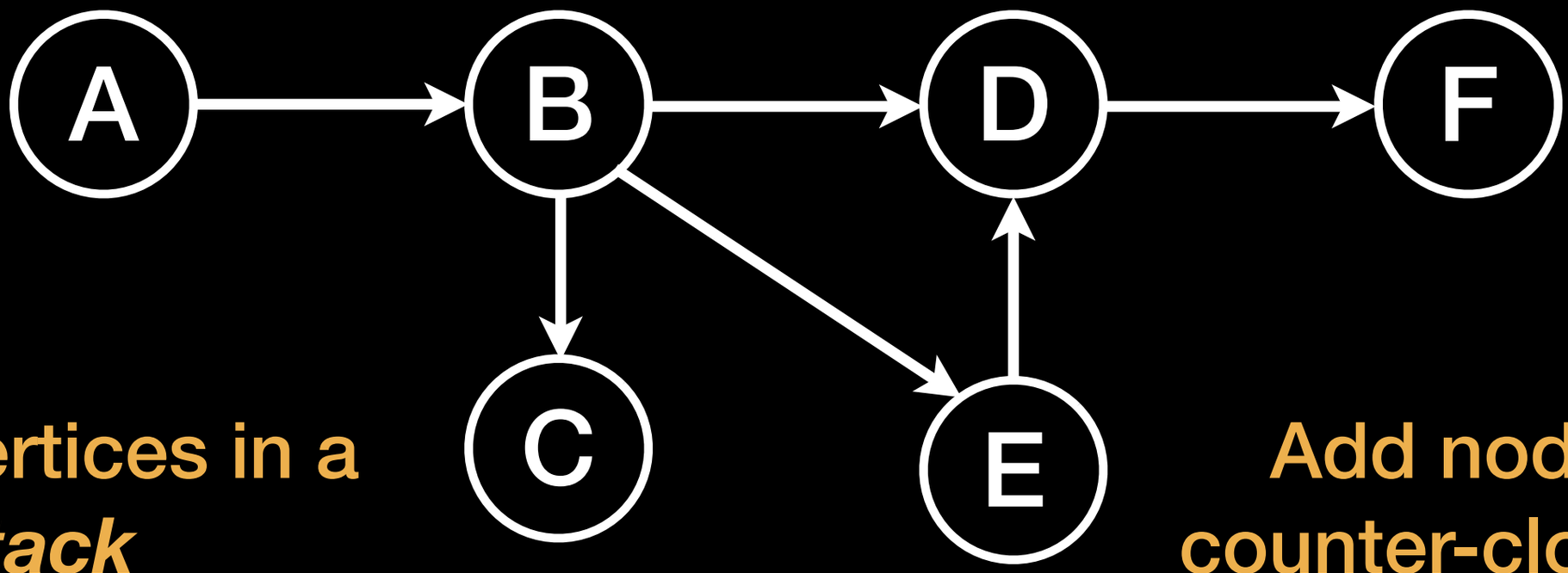
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B D

Known: C E

Simulate **depth-first search** on this graph



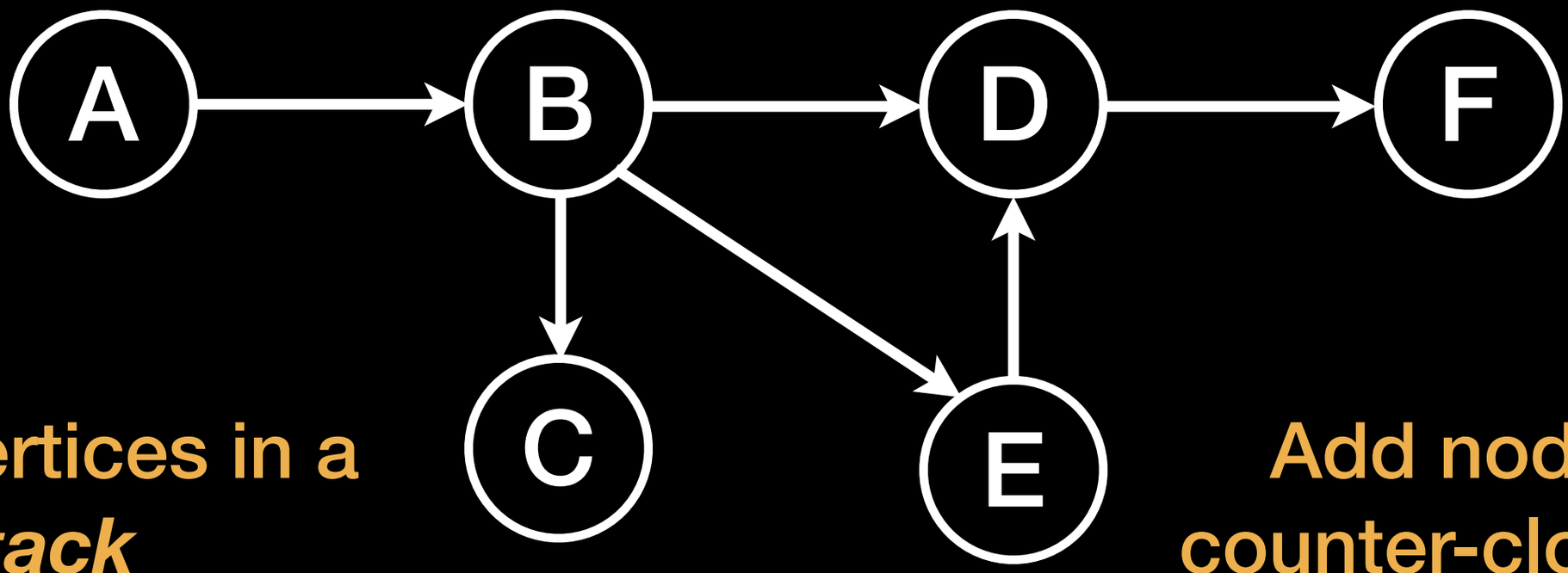
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B D

Known: C E F

Simulate **depth-first search** on this graph



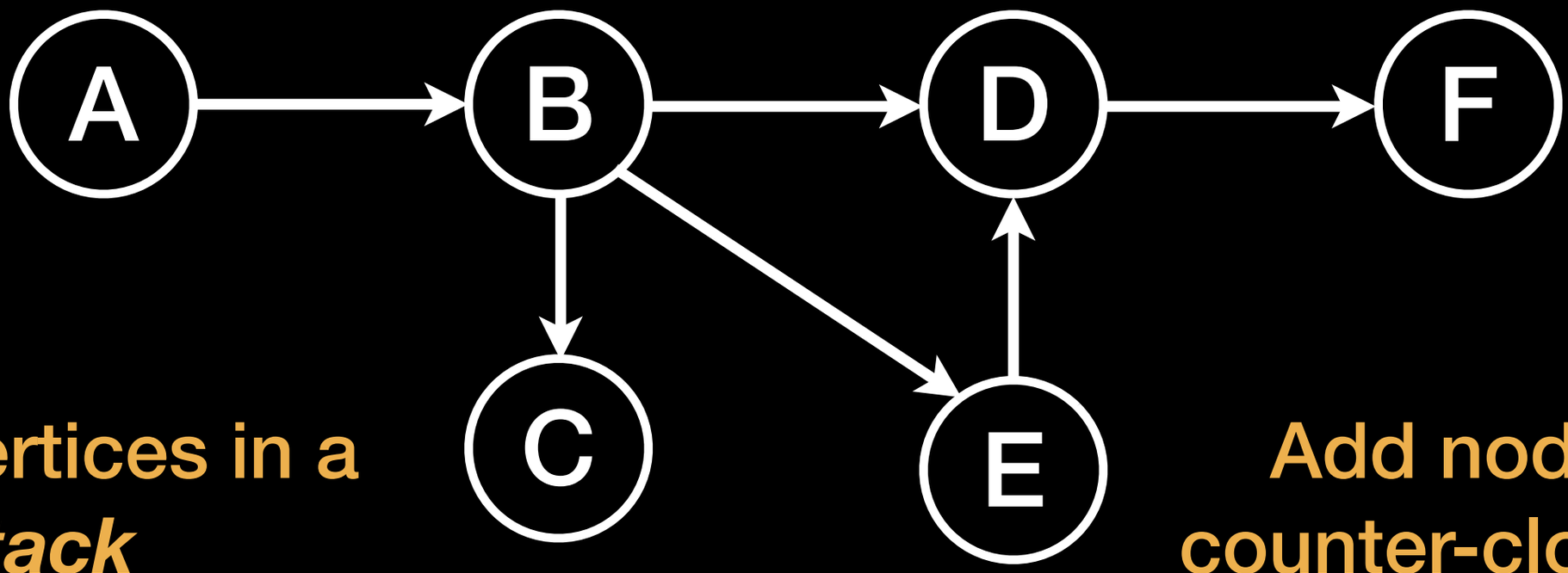
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B D F

Known: C E

Simulate **depth-first search** on this graph



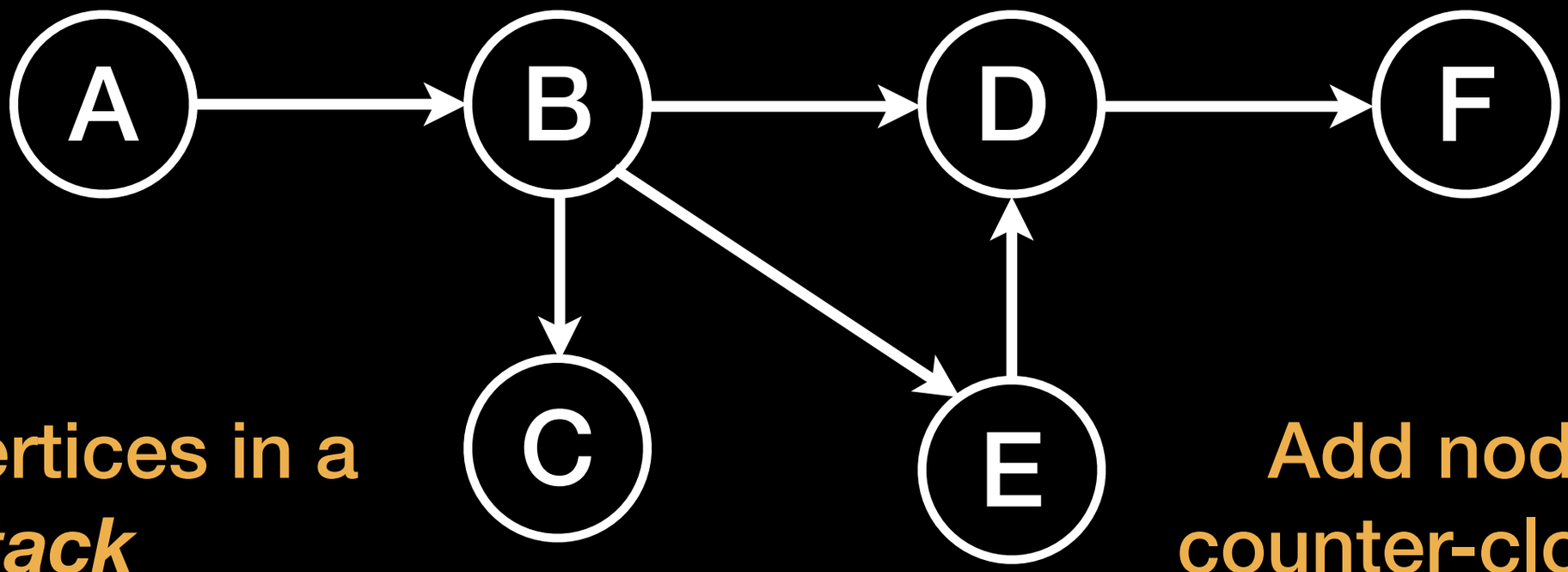
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

Reachable: A B D F E

Known: C

Simulate **depth-first search** on this graph



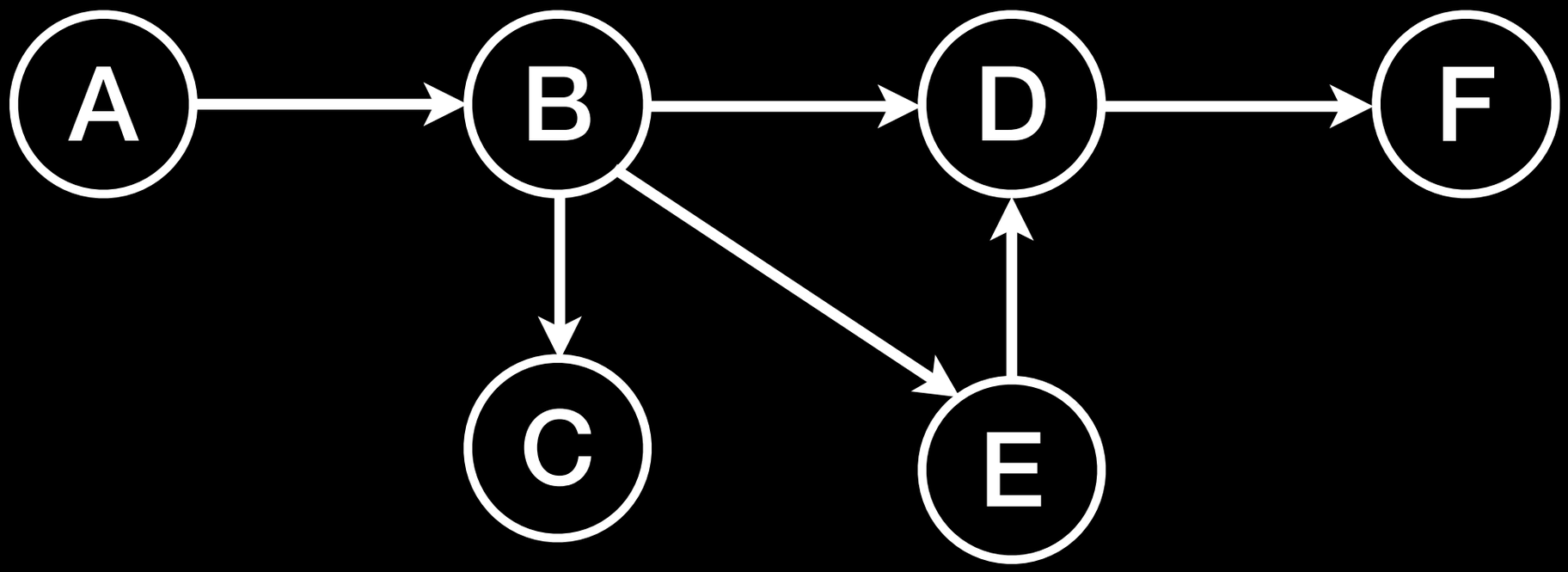
Store vertices in a *stack* (last in, first out)

Add nodes in counter-clockwise order

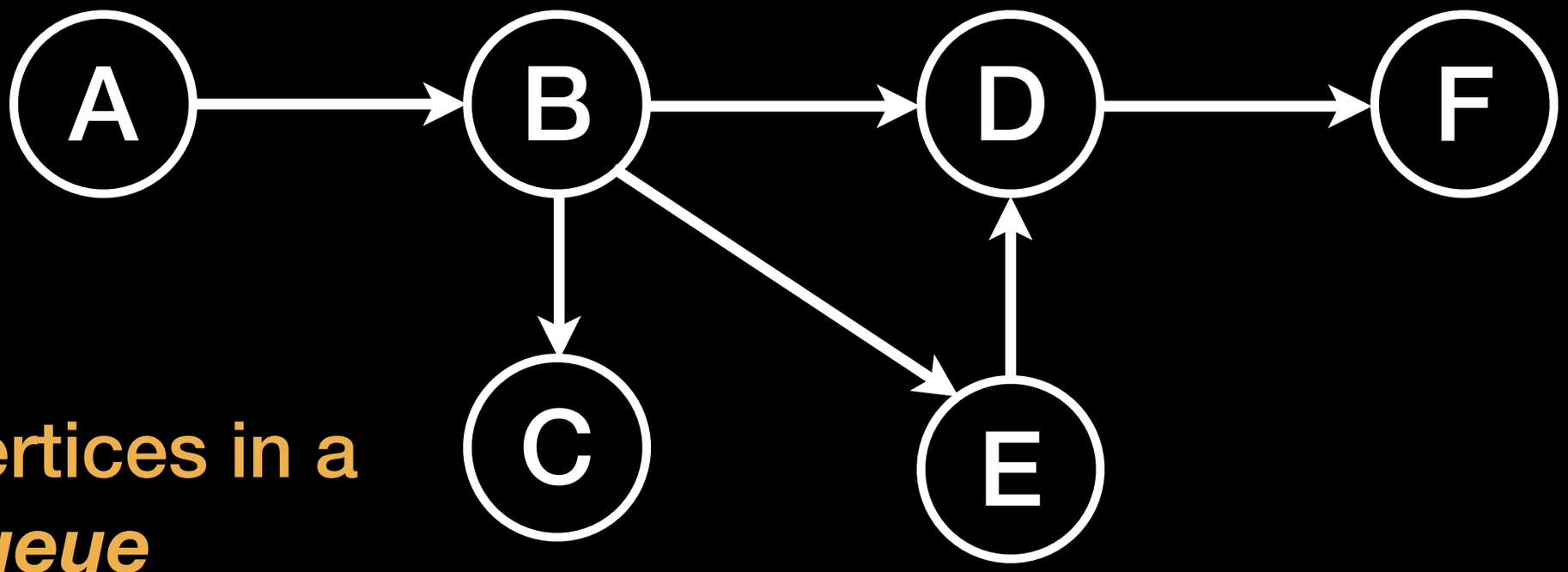
Reachable: A B D F E C

Known:

Simulate **breadth-first search** on this graph

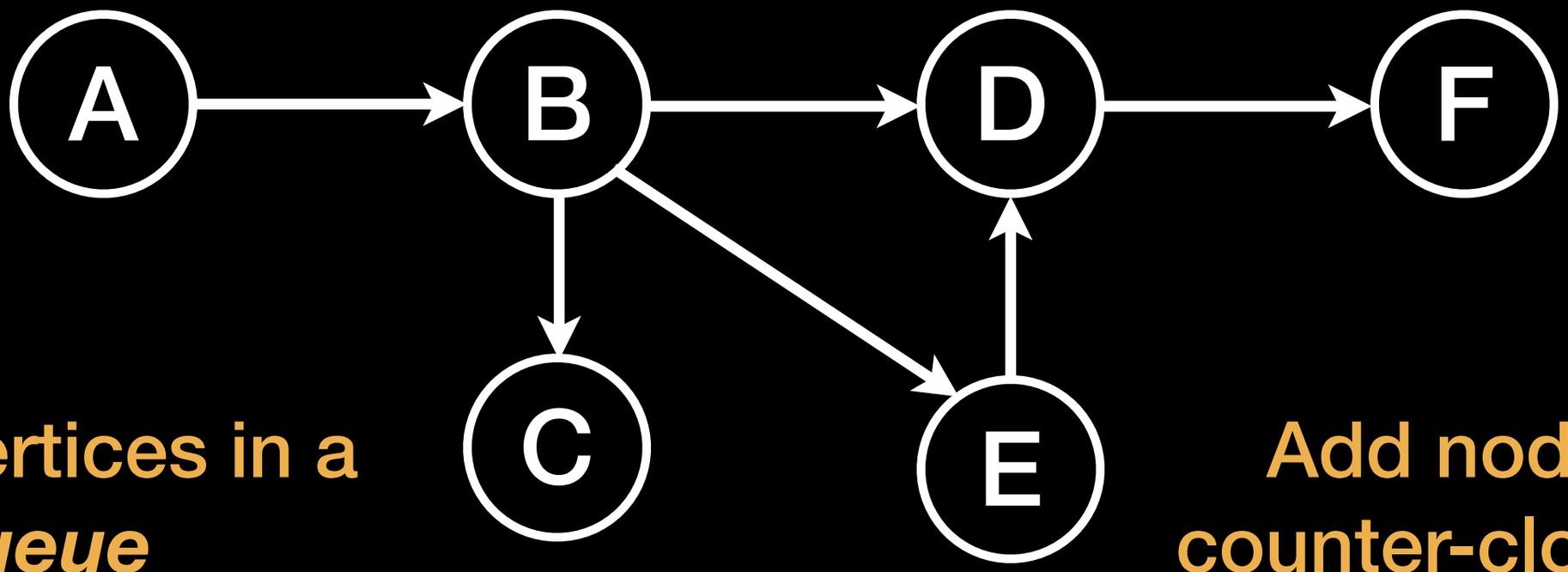


Simulate **breadth-first search** on this graph



Store vertices in a *queue*
(first in, first out)

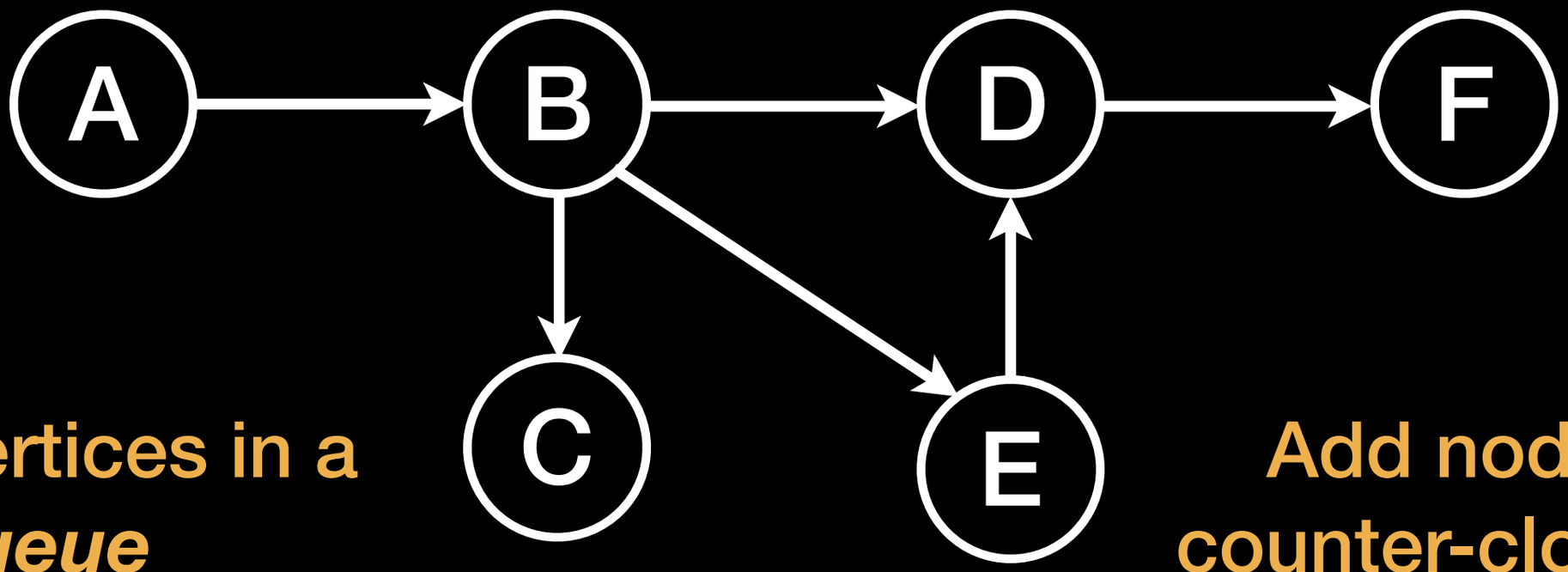
Simulate **breadth-first search** on this graph



Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Simulate **breadth-first search** on this graph



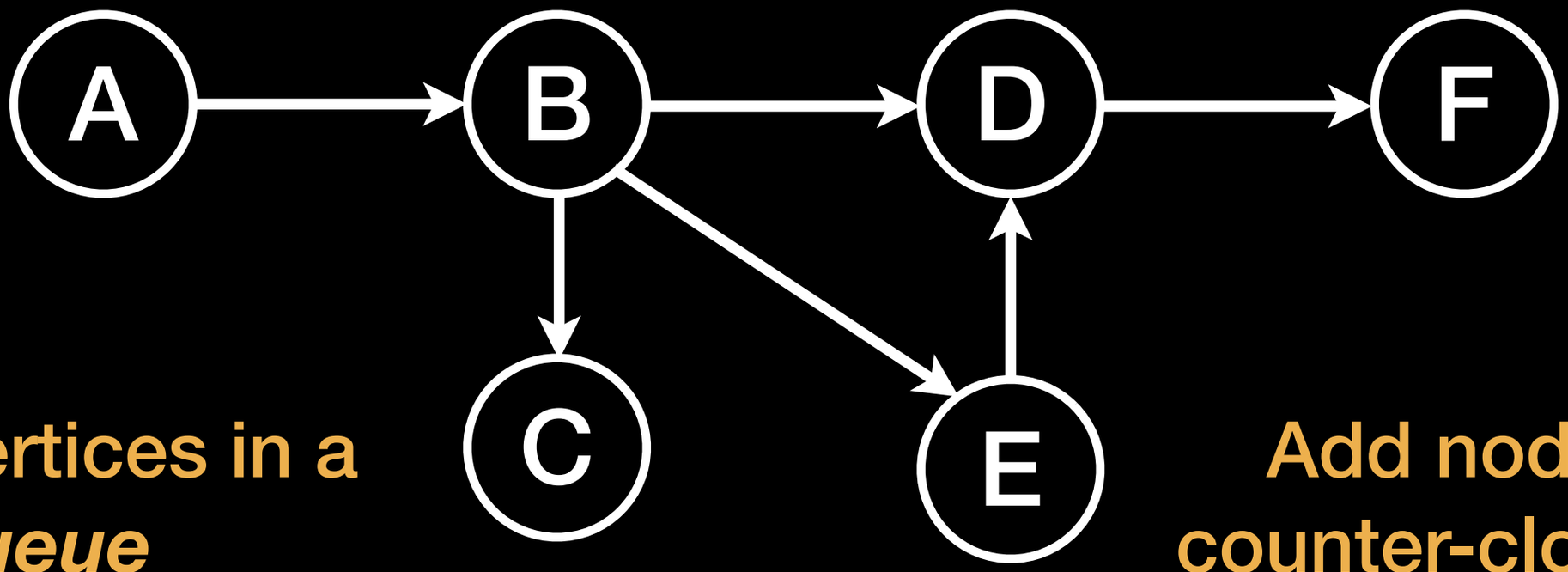
Store vertices in a ***queue*** (first in, first out)

Add nodes in counter-clockwise order

Reachable:

Known:

Simulate **breadth-first search** on this graph



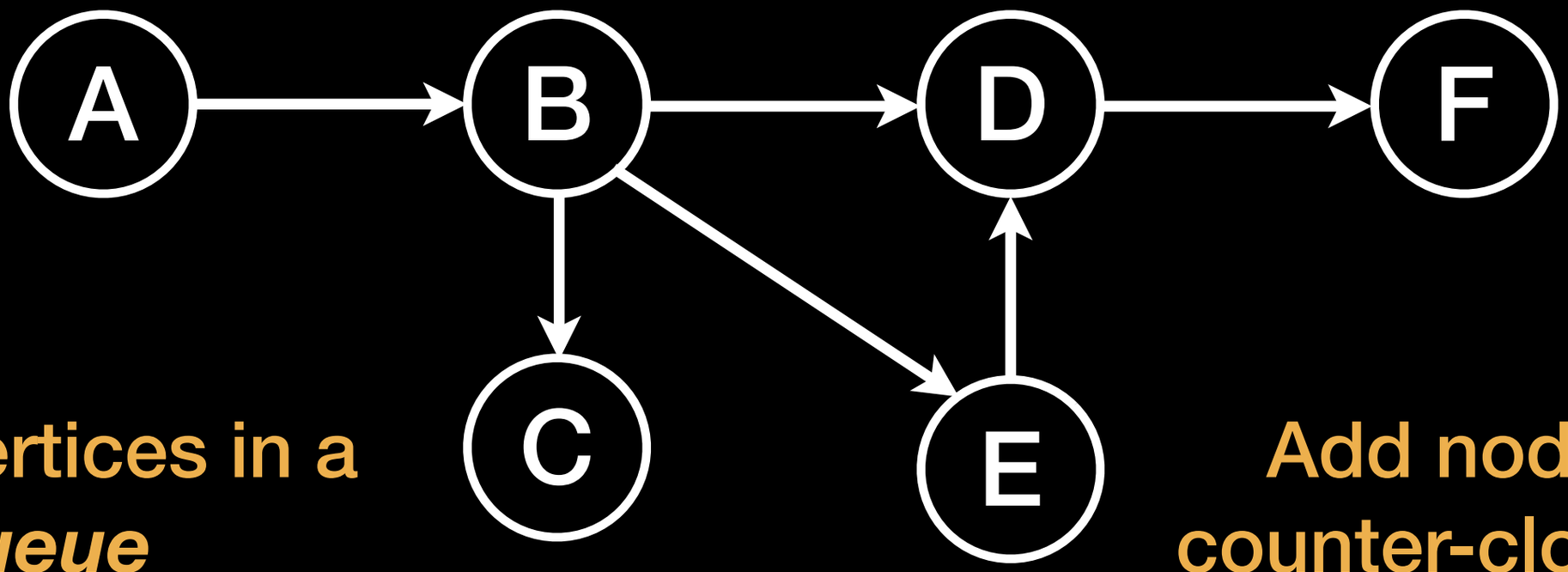
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable:

Known: A

Simulate **breadth-first search** on this graph



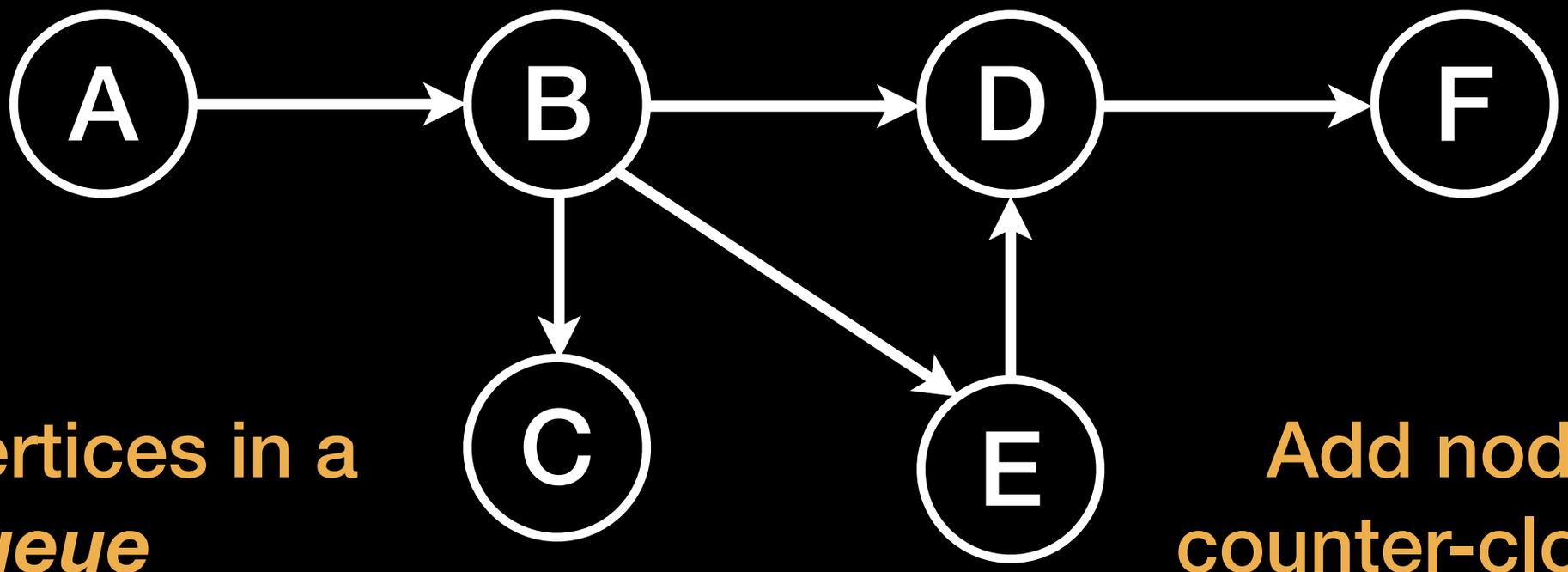
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A

Known:

Simulate **breadth-first search** on this graph



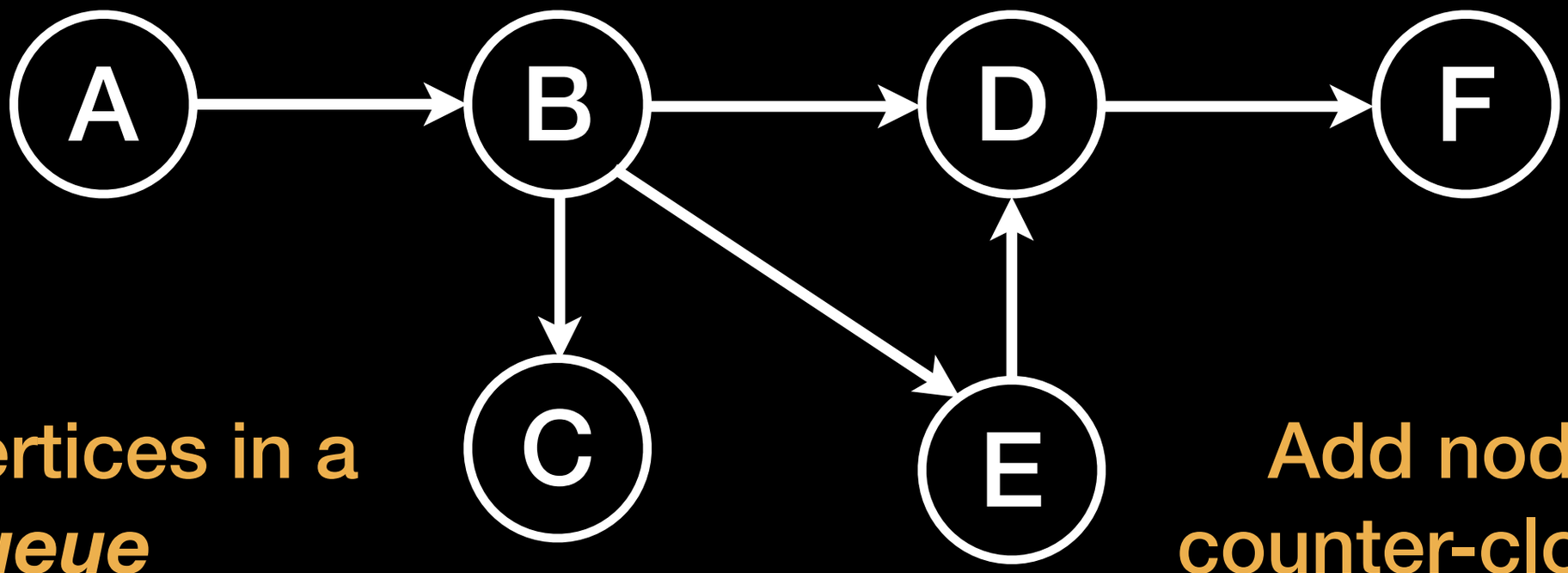
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A

Known: B

Simulate **breadth-first search** on this graph



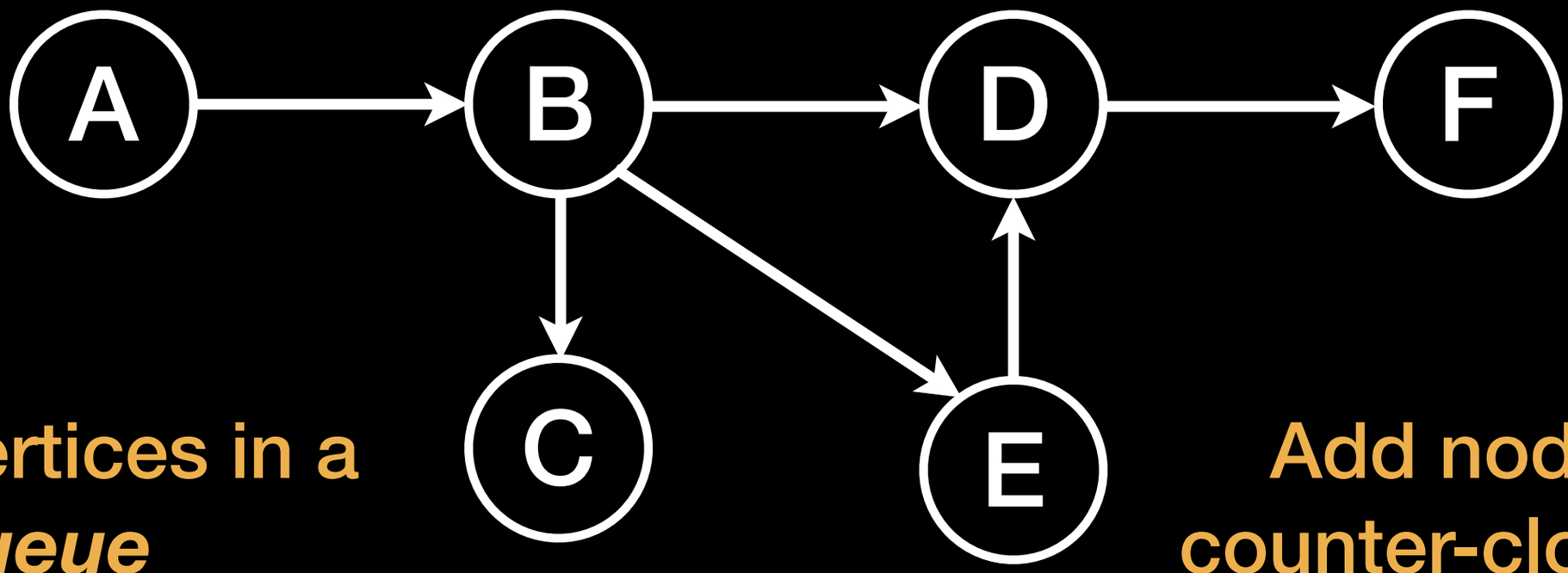
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known:

Simulate **breadth-first search** on this graph



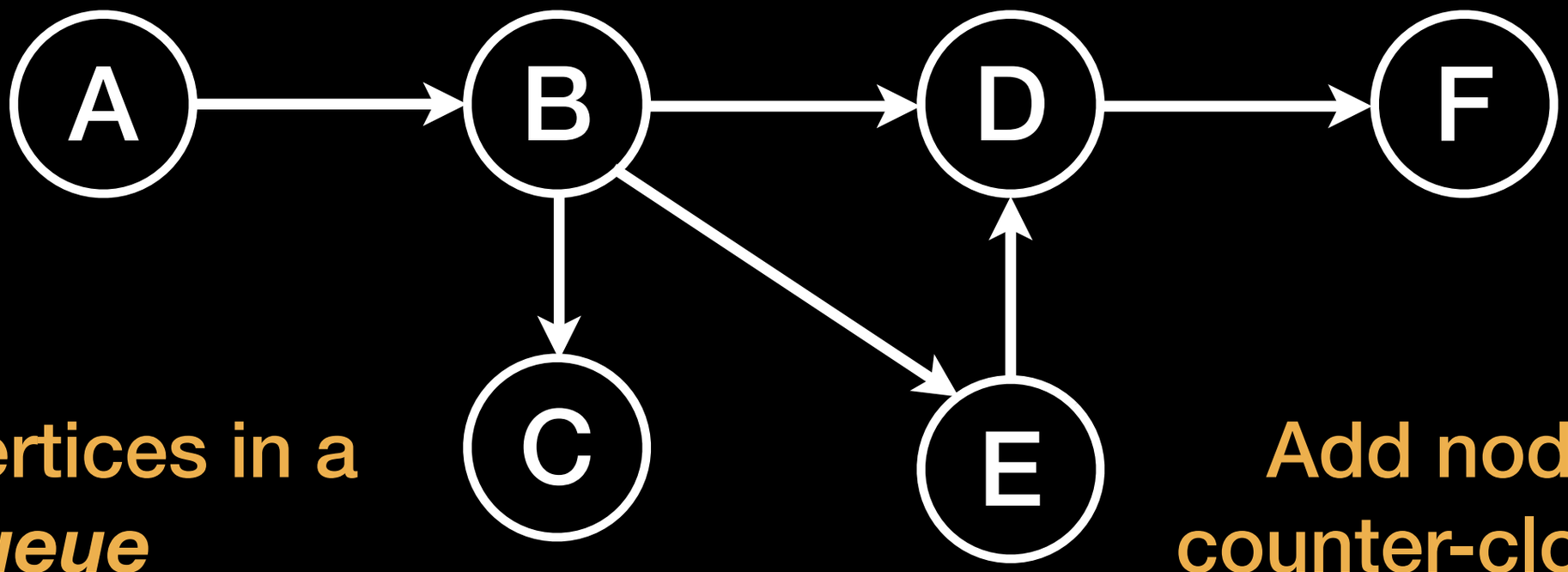
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C

Simulate **breadth-first search** on this graph



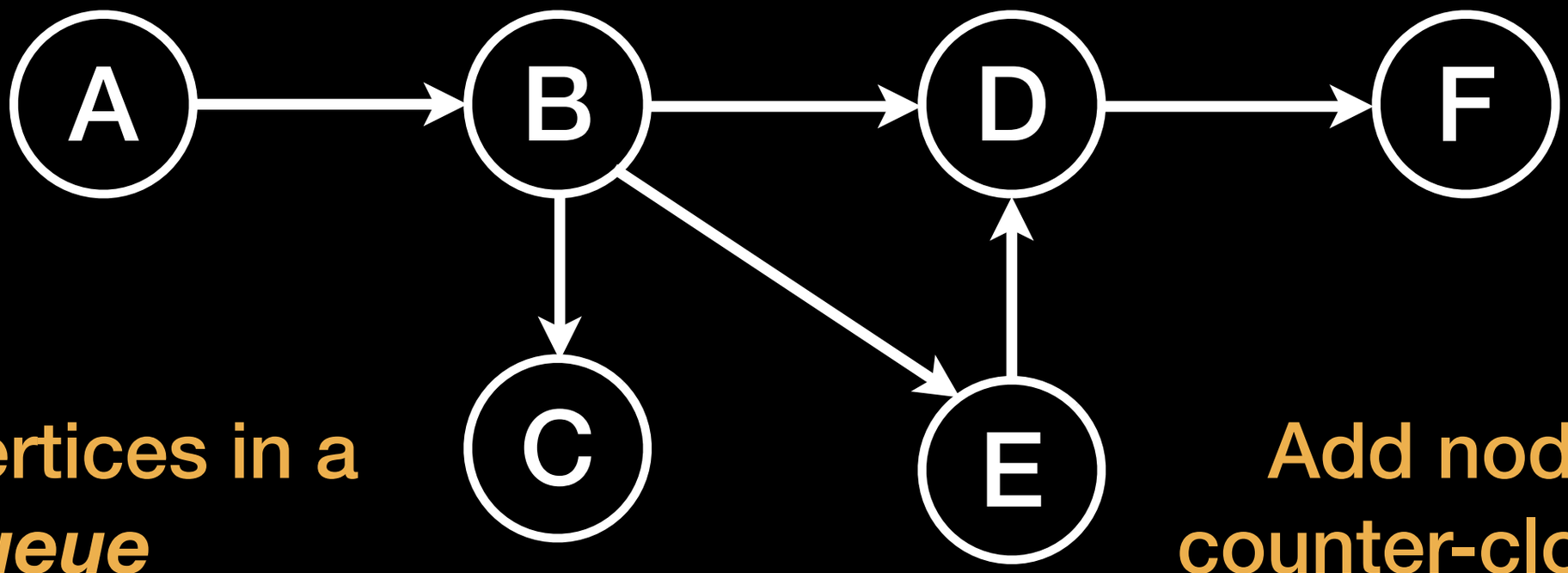
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C E

Simulate **breadth-first search** on this graph



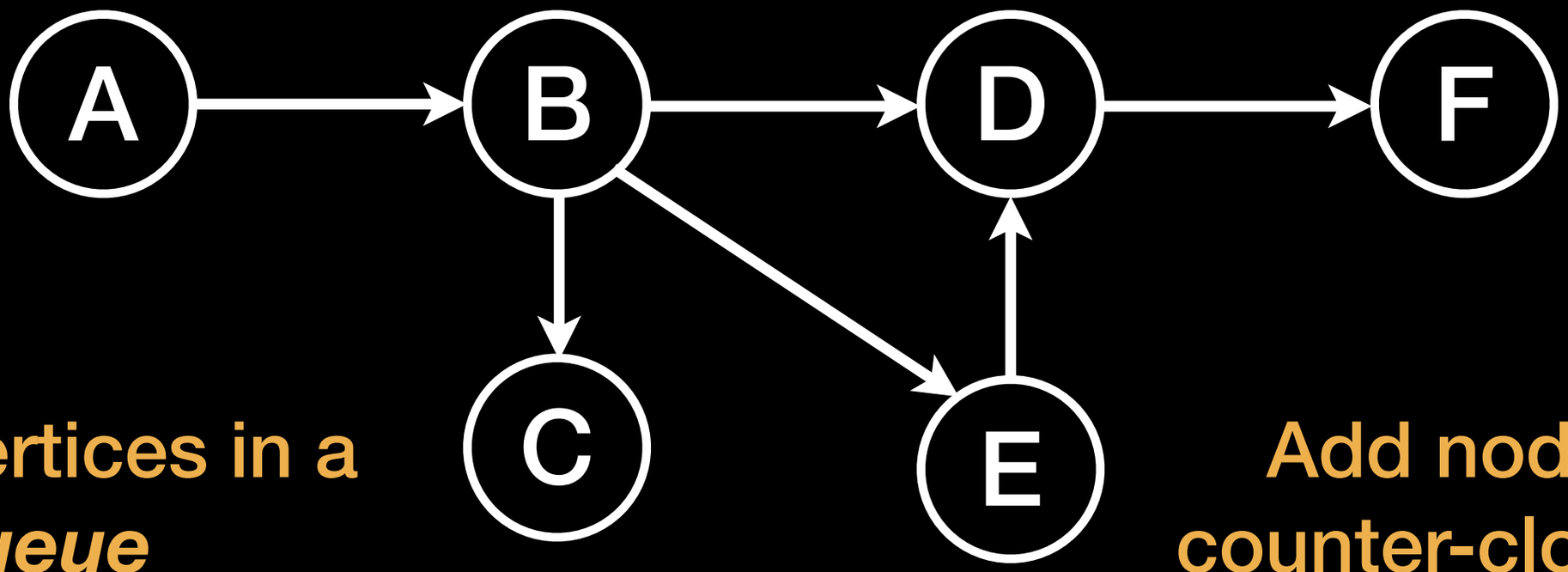
Store vertices in a ***queue*** (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B

Known: C E D

Simulate **breadth-first search** on this graph



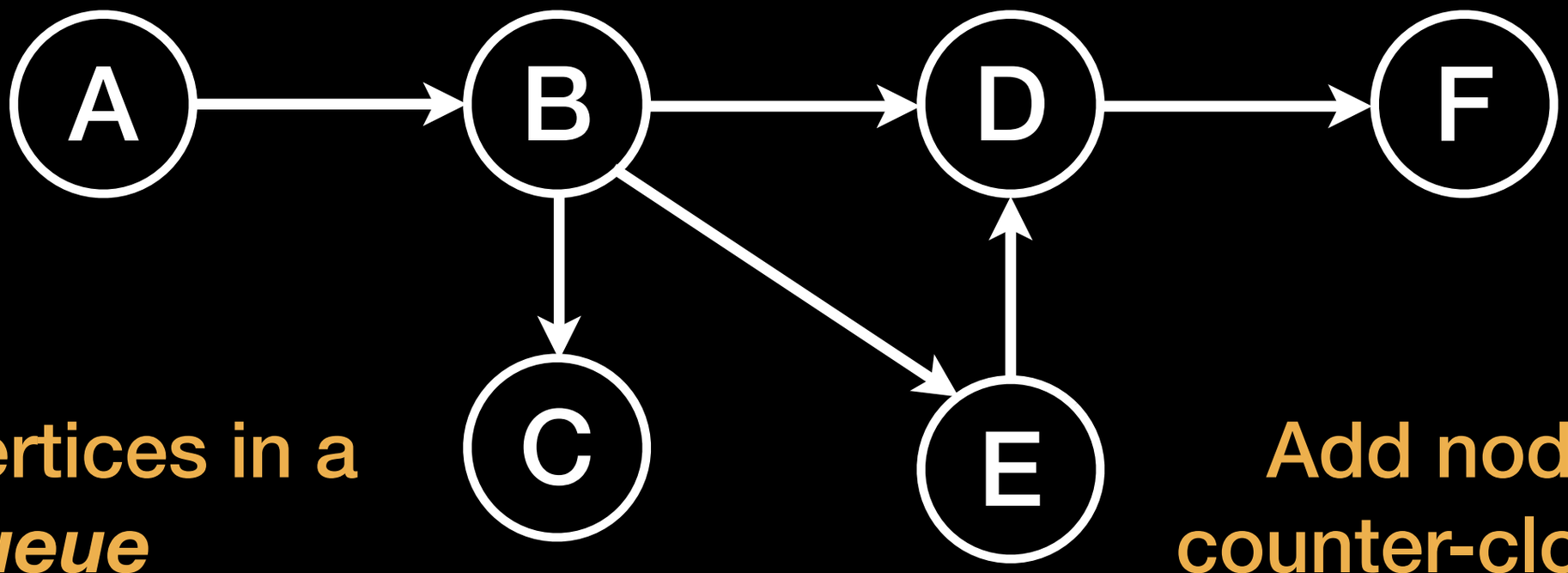
Store vertices in a ***queue*** (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C

Known: E D

Simulate **breadth-first search** on this graph



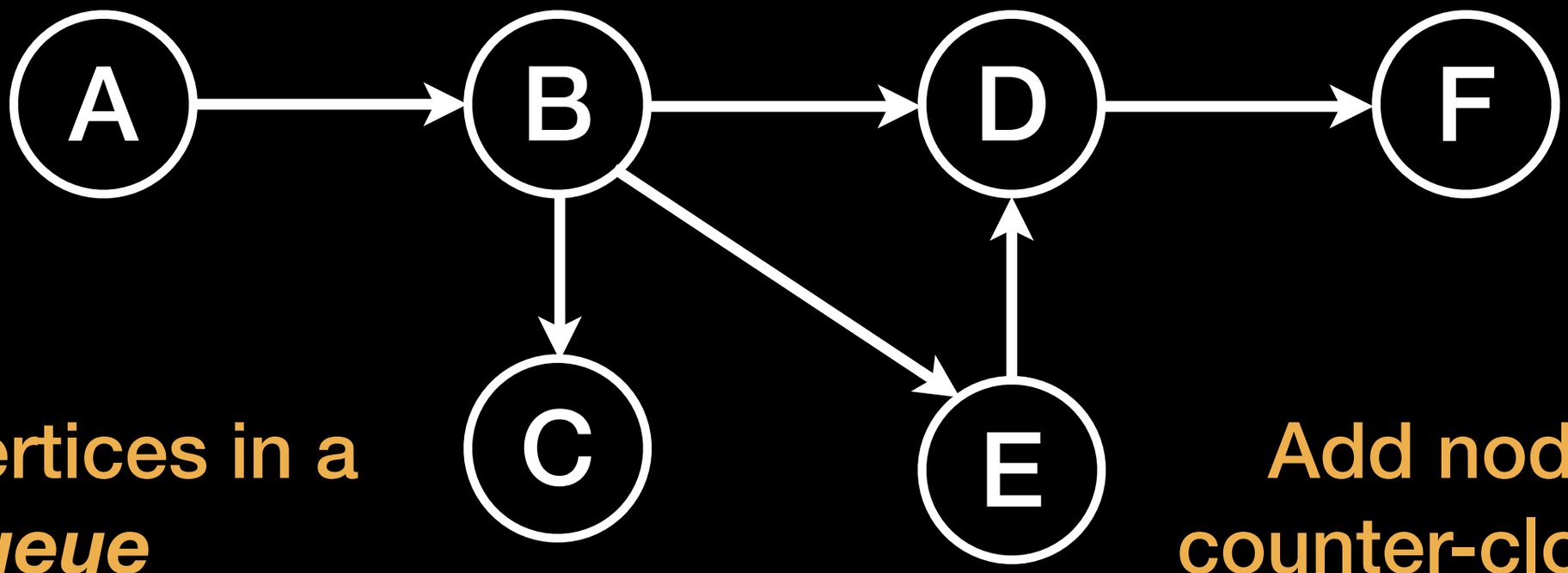
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E

Known: D

Simulate **breadth-first search** on this graph



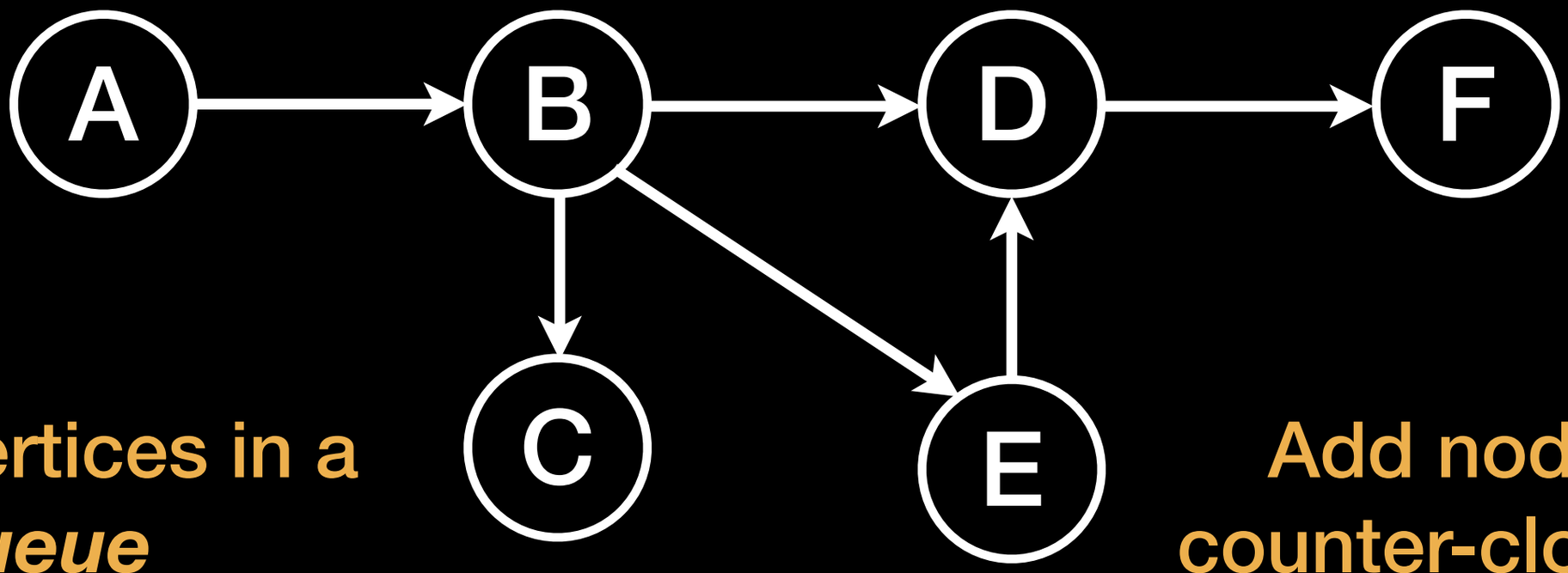
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E

Known: D D

Simulate **breadth-first search** on this graph



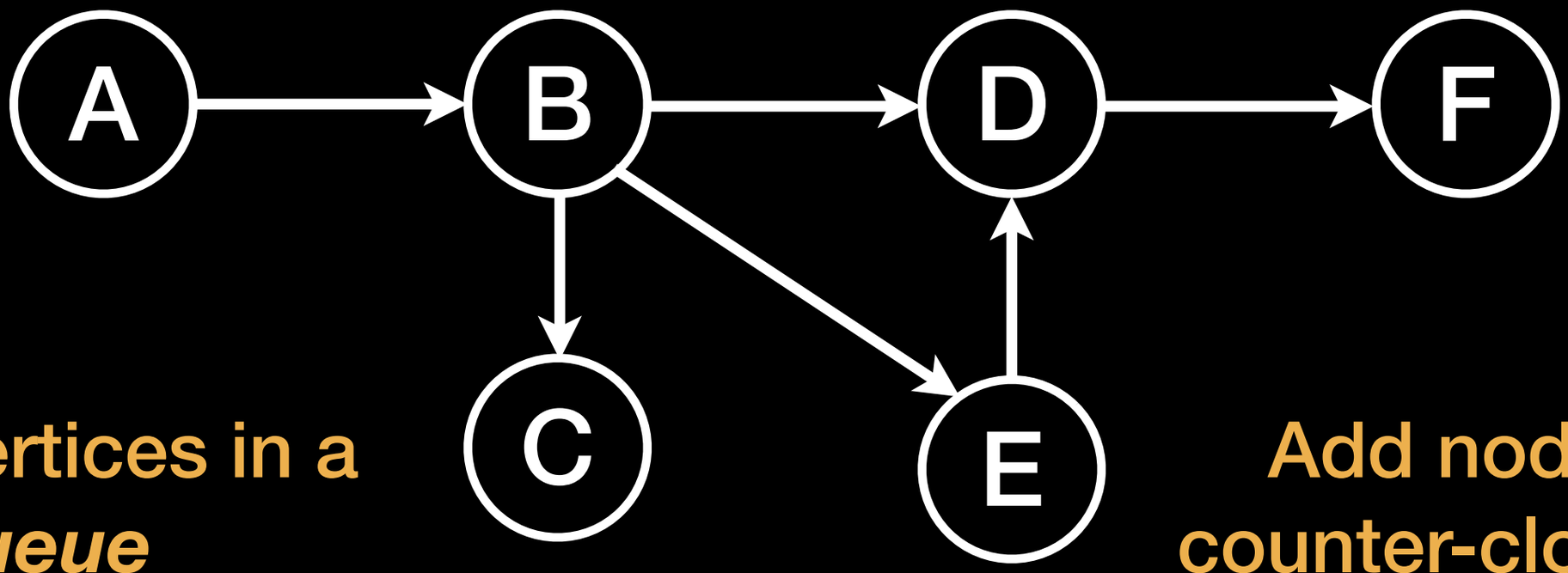
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E D

Known: D

Simulate **breadth-first search** on this graph



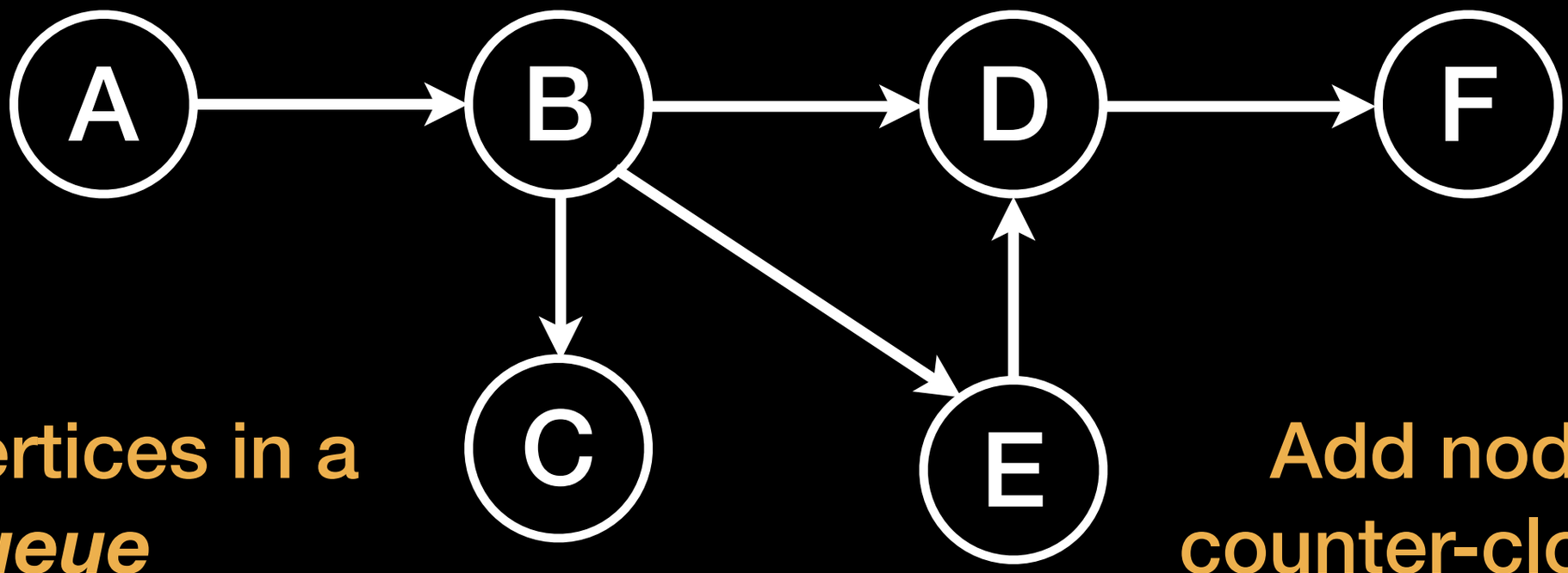
Store vertices in a ***queue*** (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E D

Known: D F

Simulate **breadth-first search** on this graph



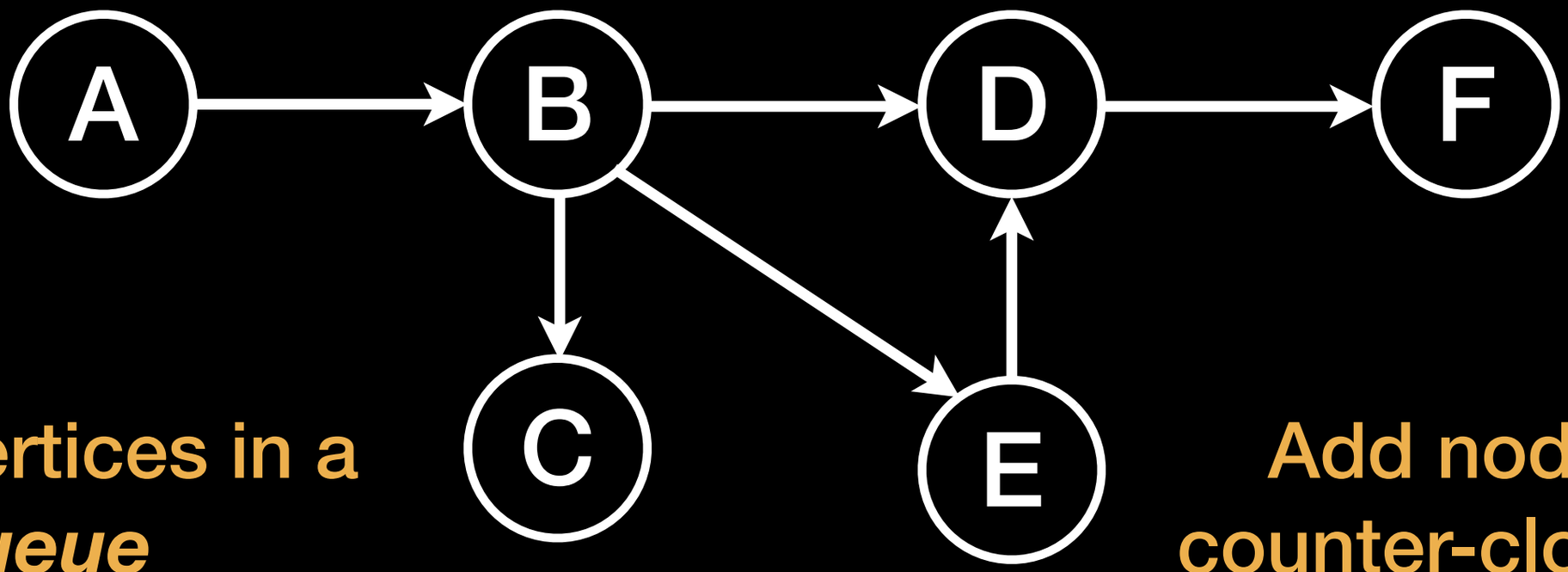
Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E D F

Known:

Simulate **breadth-first search** on this graph



Store vertices in a *queue* (first in, first out)

Add nodes in counter-clockwise order

Reachable: A B C E D F

Known: