Balanced Trees

Balanced trees have height $O(\log n)$.

- Height-balanced trees

  At each node, ***height*** of left and right subtrees are "close".

  e.g. AVL trees, B-trees, red-black trees, splay trees

- Weight-balanced trees

  At each node, ***number of nodes*** in left and right subtrees are "close".

- Search, Predecessor, Successor, Minimum, Maximum

  $O(\log n)$ time.

- Insert and Delete may have to ***rebalance*** the tree.

Splay Trees

Self-adjusting binary search tree (Sleator & Tarjan 1985)

Recently accessed elements are quick to access again.

Amortized running time: $O(\log n)$

Averaged running time per operation over a worst-case sequence of operations.

Advantages

- Simple implementation — easier to implement than other self-balancing BSTs such as red-black trees and AVL trees.
- Comparable performance — average-case performance is as efficient as other self-balancing BSTs.
- Small memory footprint — no need to store bookkeeping data.
- Working well with nodes containing identical keys.

Splaying

Restructuring heuristics:

- Performs rotations bottom-up along the access path and moves the accessed item all the way to the root.

- Does the rotations in pairs, in an order depending on the structure of the access path.

How to splaying a tree at a node x?

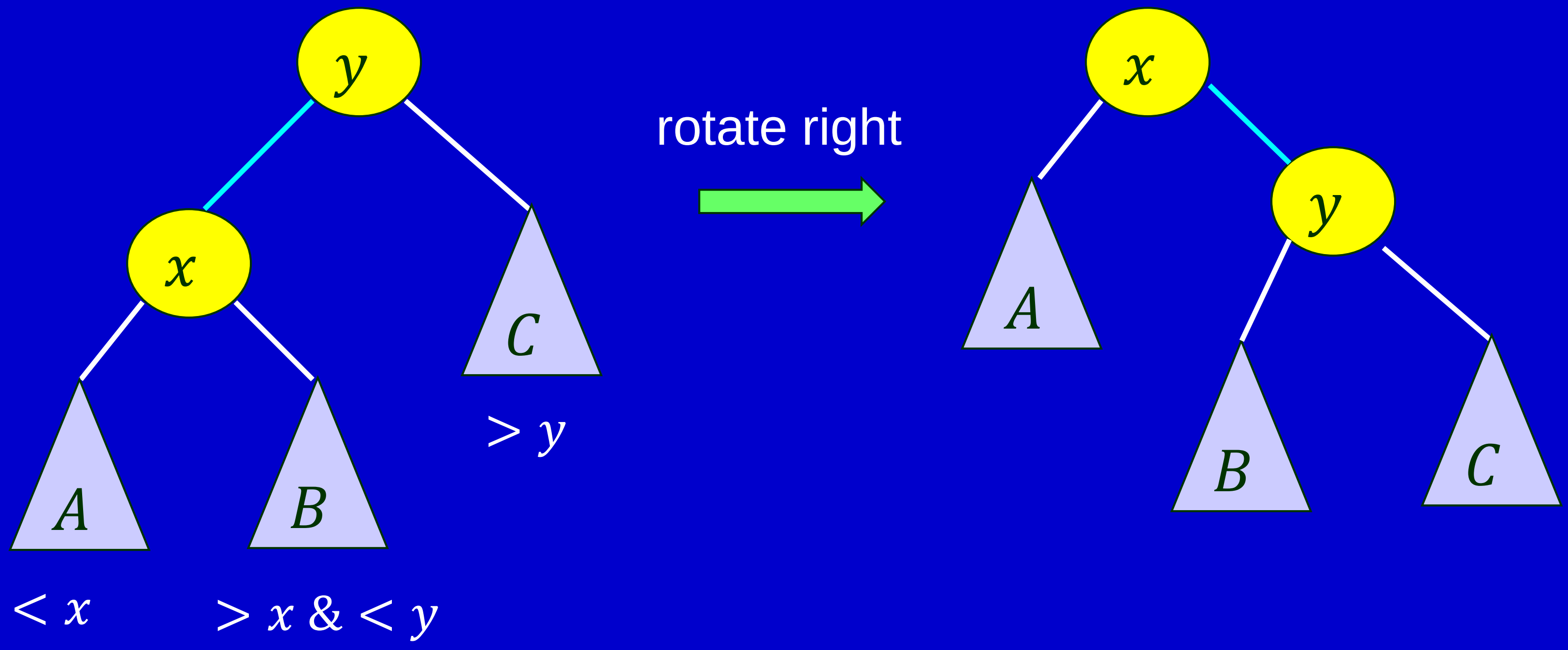
Repeat a sequence of splaying steps until x is the root of tree.

zig, zig-zig, or zig-zag.

Case 1: zig

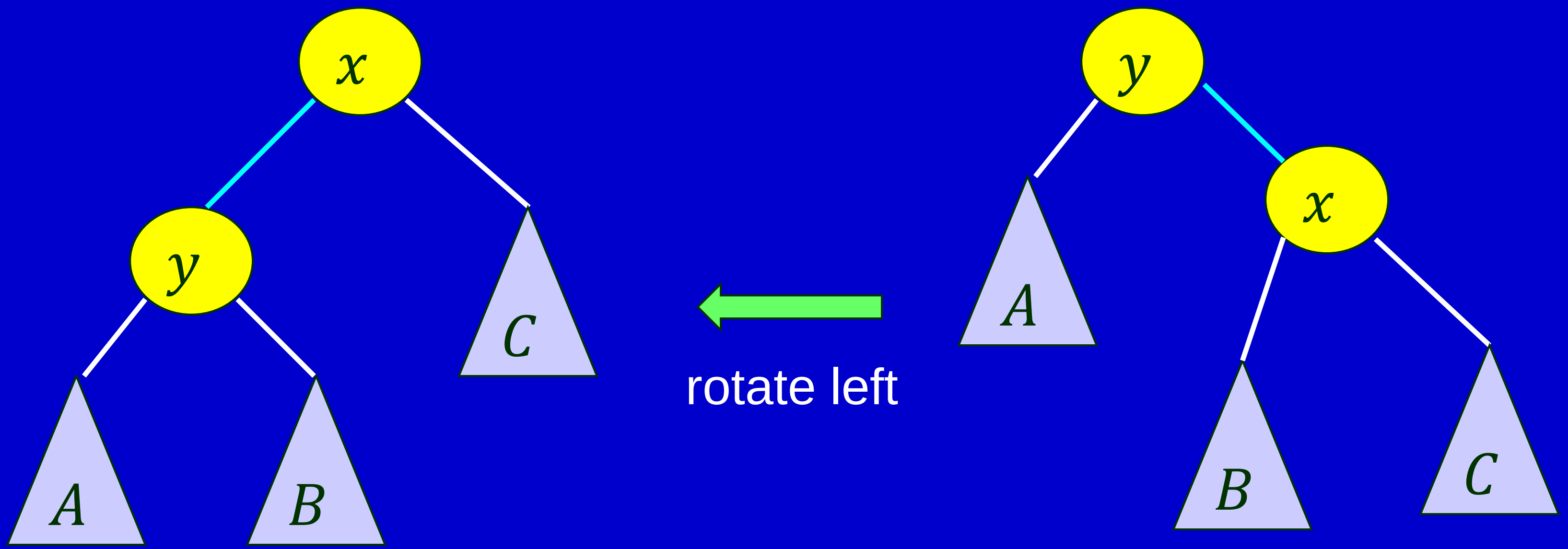
Parent y of x is the tree root.

Rotate the edge $\langle x, y \rangle$.



Terminal case.

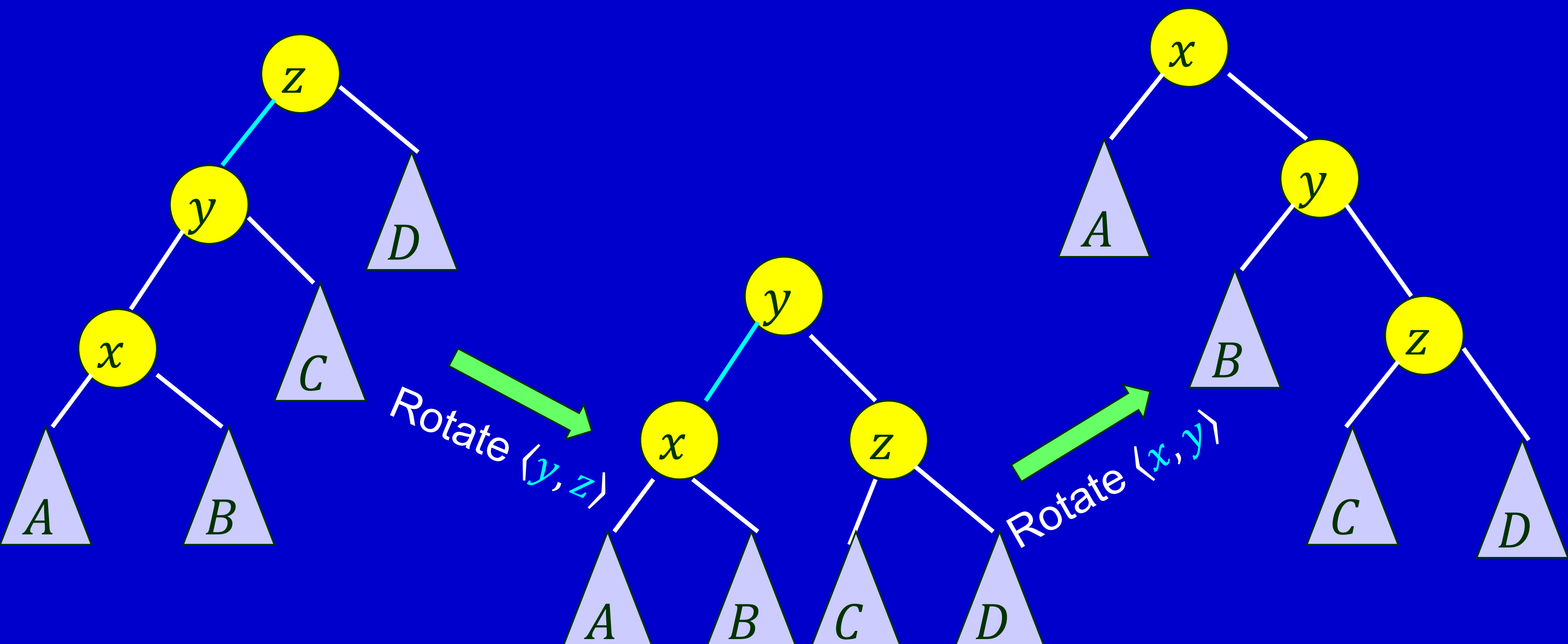
Case 1: zig (cont'd)



Case 2: zig-zig

a) Parent y of x is not the root (x has grandparent z).
b) x and y are both left or both right children.

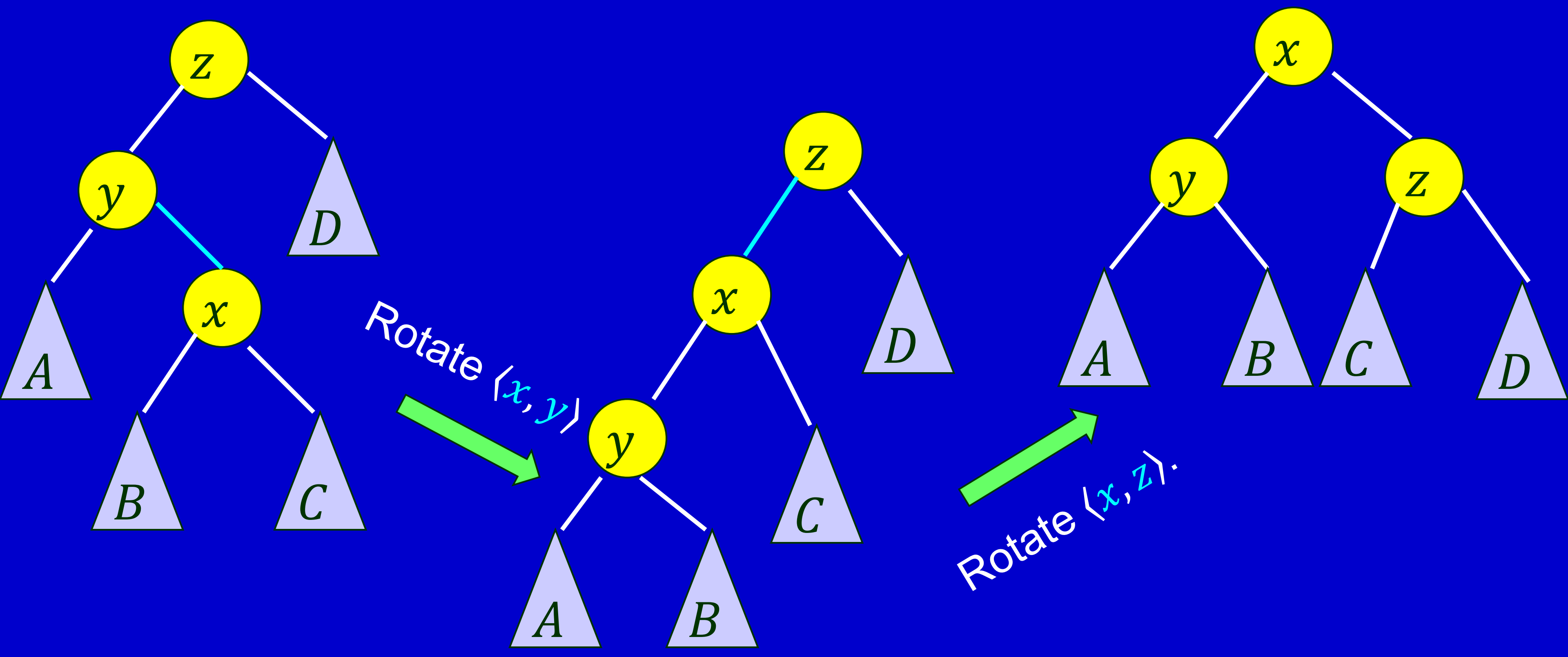
Symmetric rotations when x and y are both right children.



Case 3: zig-zag

a) Parent y of x is not the root (x has grandparent z).
b) x is a left child and y is a right child, or vice versa.

Symmetric rotations when x is a left child and y is a right child.



Time for Splaying a Node

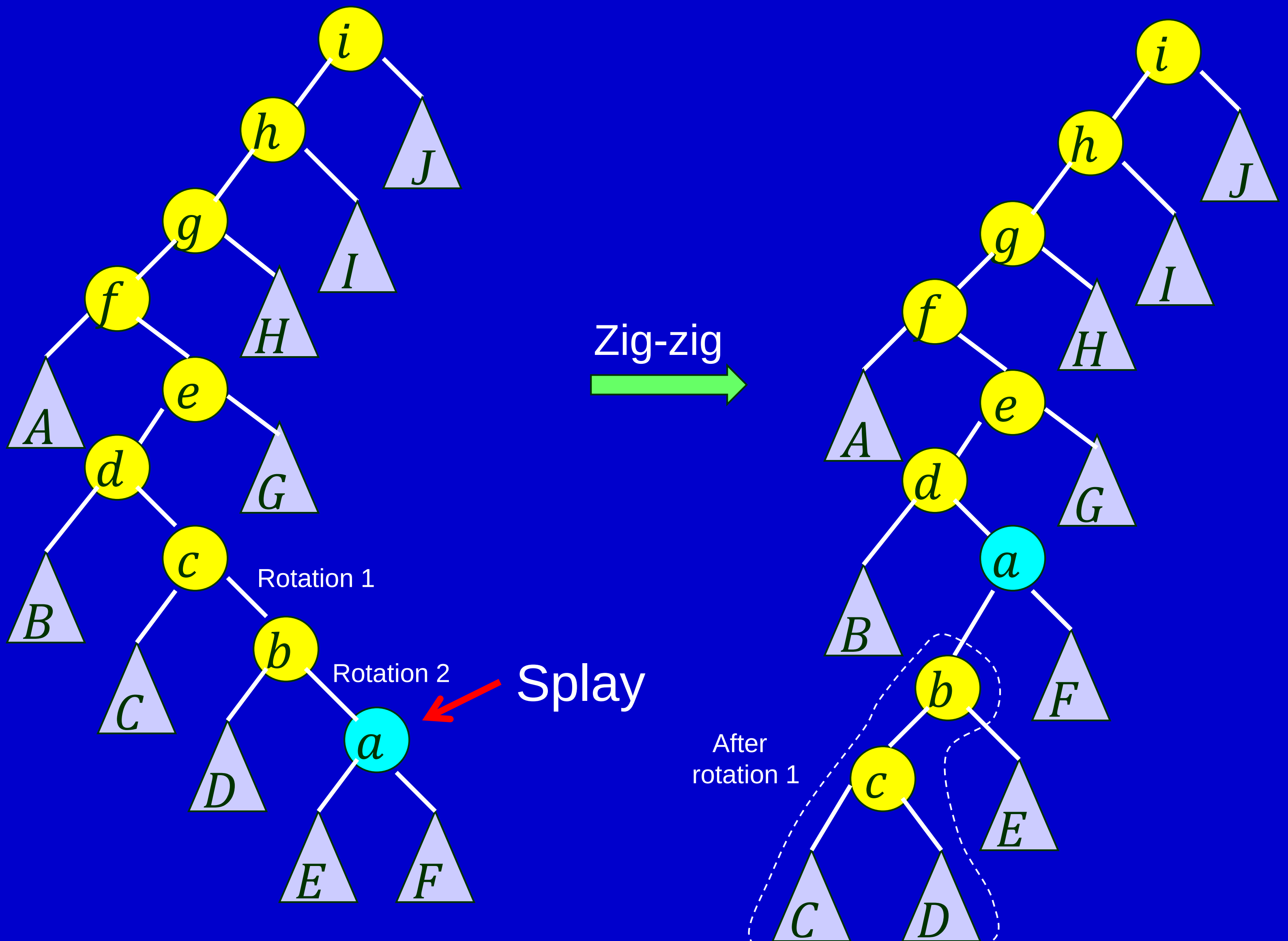
$\Theta(d)$: where d is the depth of the node x.

Proportional to the time to access the item stored in x.

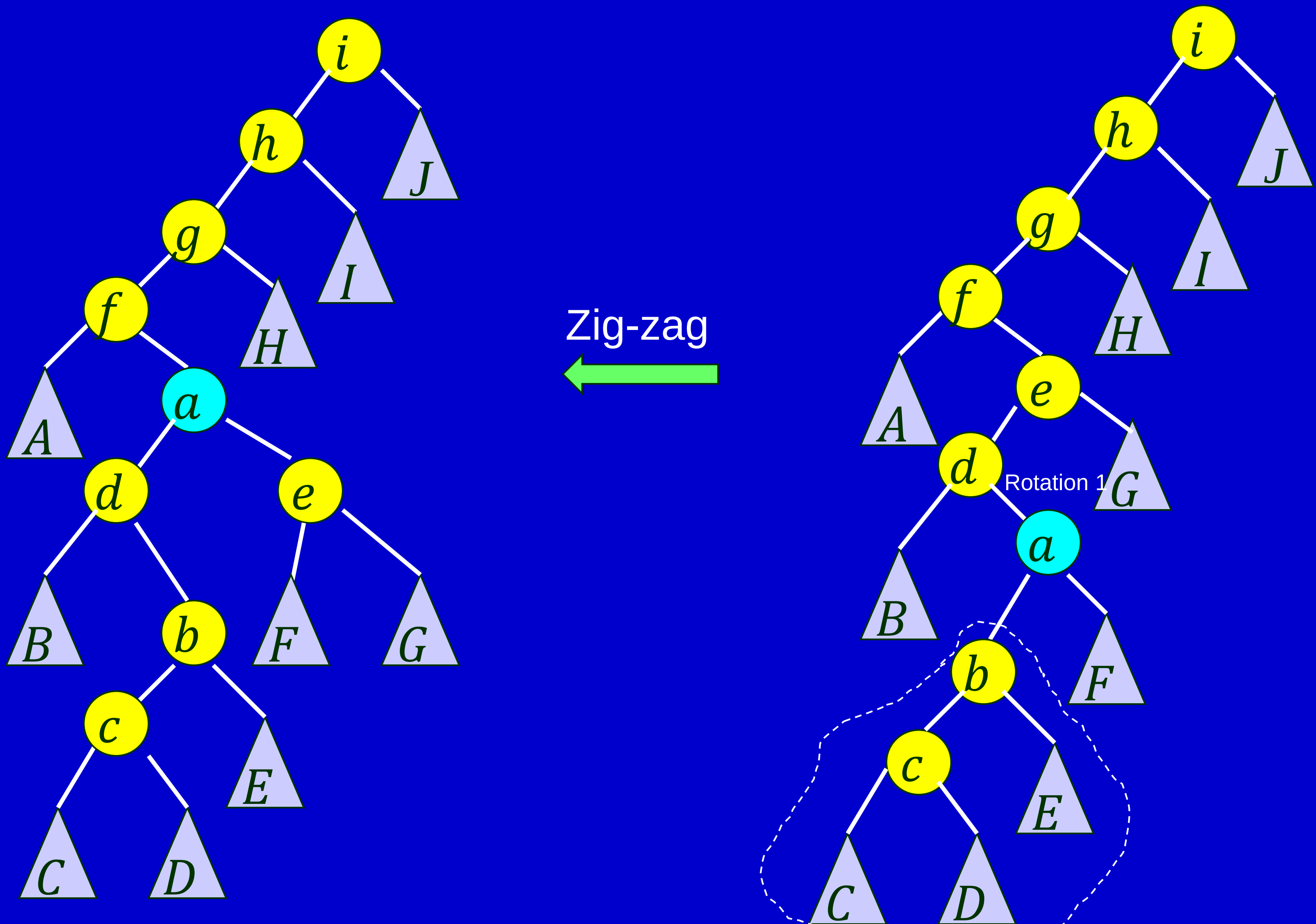
Effects of splaying:

- Moves x to the root.
- Roughly *halves* the depth of every node along the access path (shown in the next two examples).

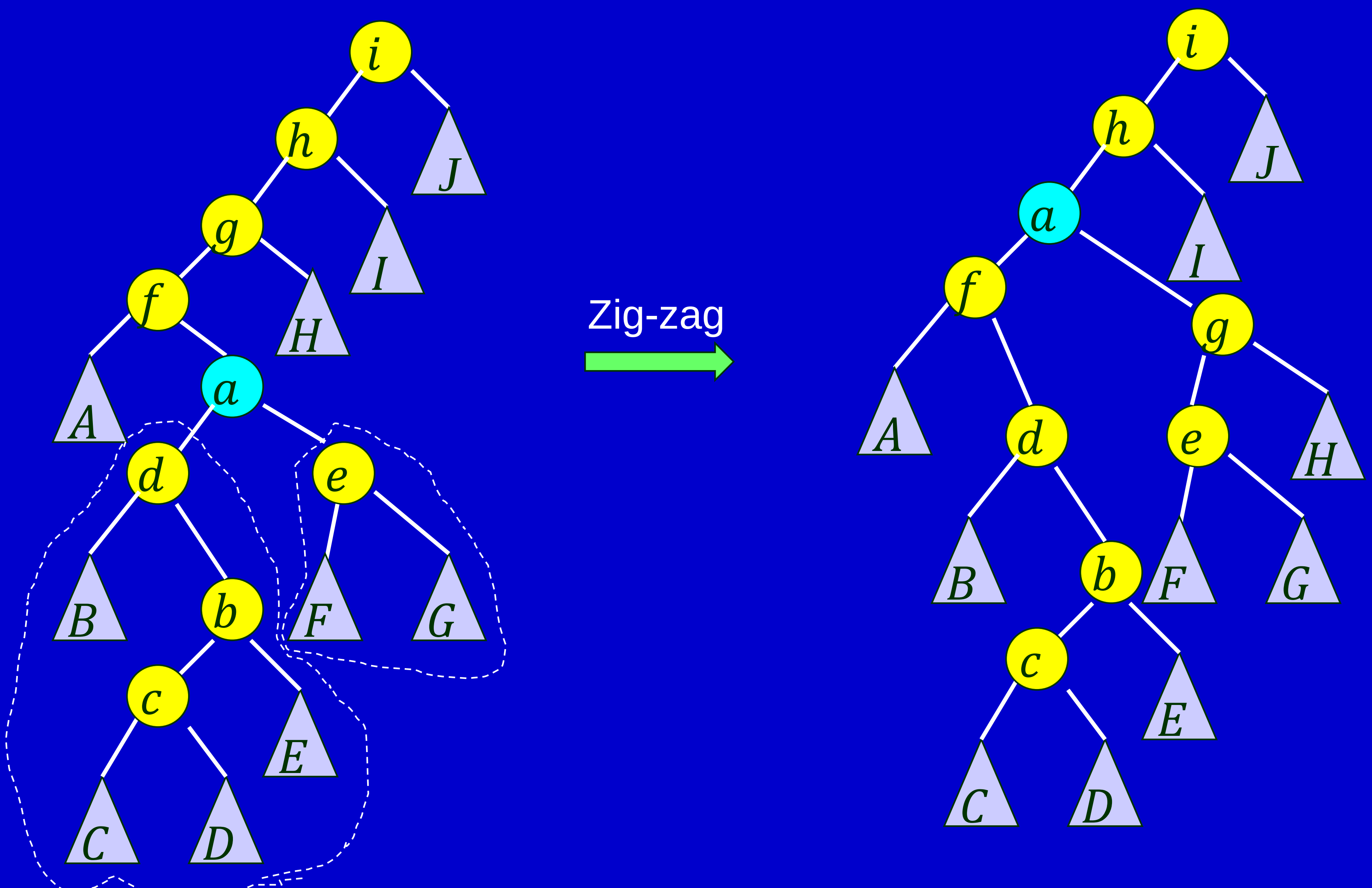
Example 1 – Splay Step 1



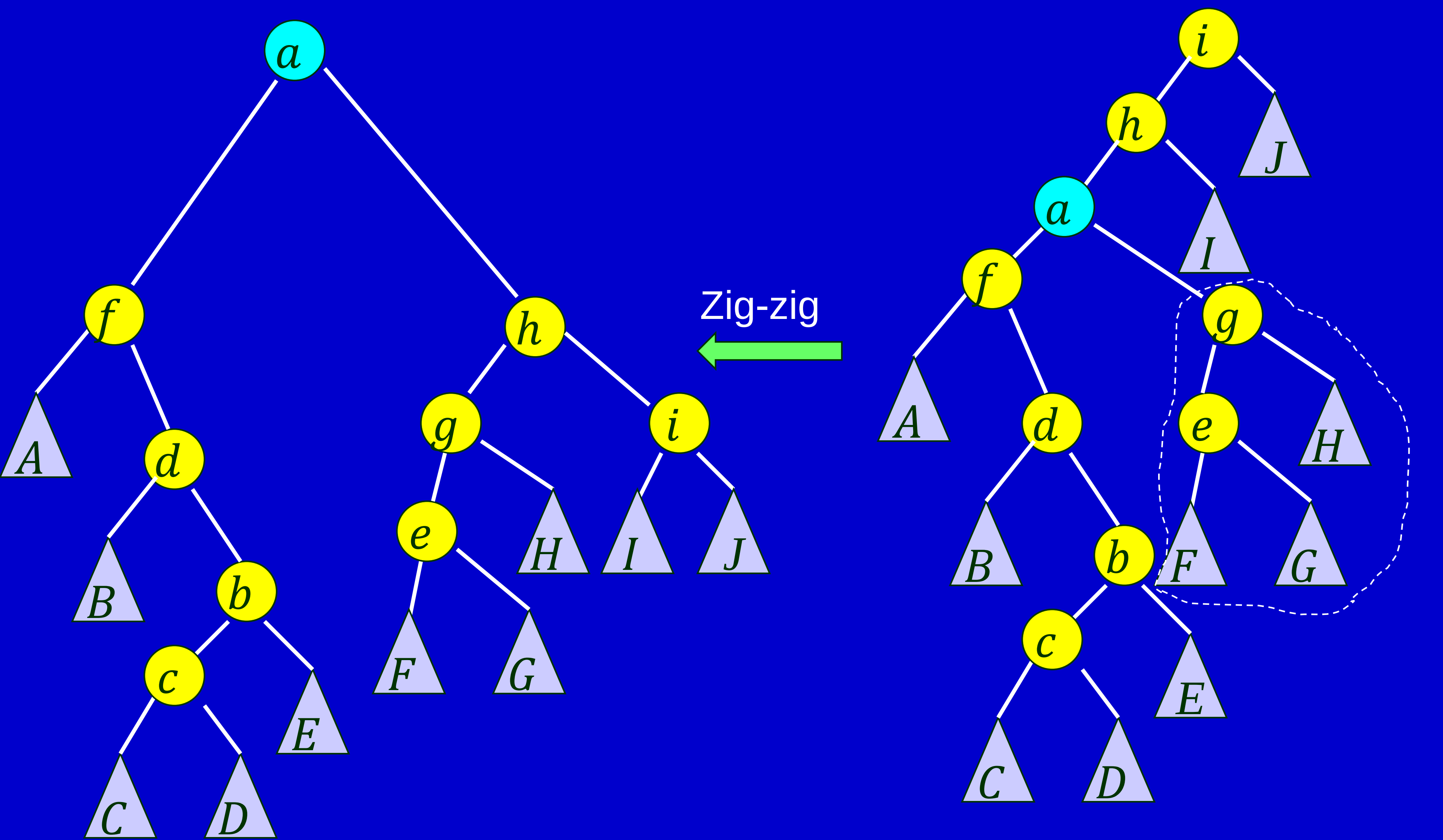
Splay Step 2


i
h
J
g
I
f
H
A
a
d
e
B
b
F
G
c
E
C
D
Zig-zag
i
h
J
g
I
f
H
A
e
d
Rotation 1
G
a
B
b
F
c
E
C
D

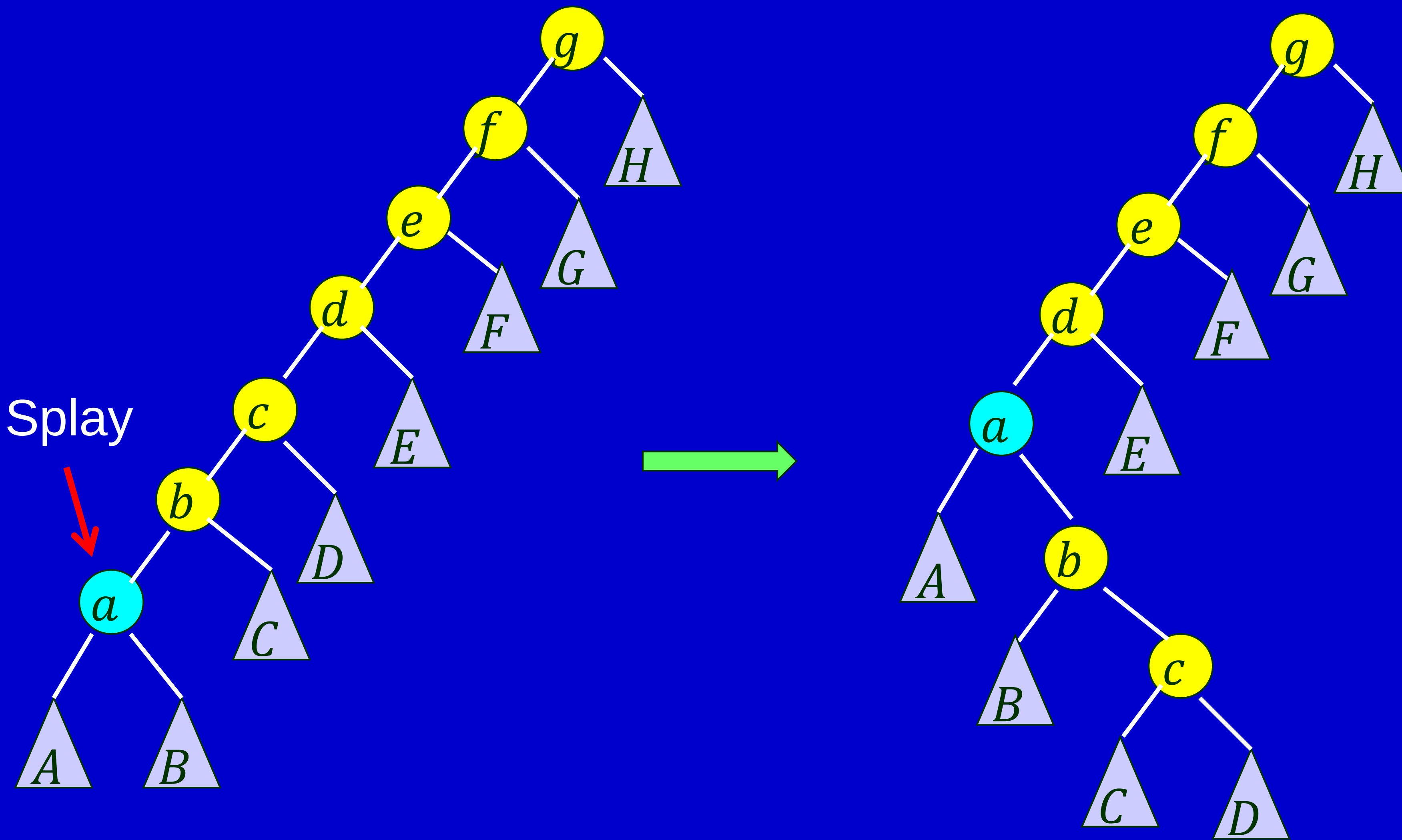
Splay Step 3



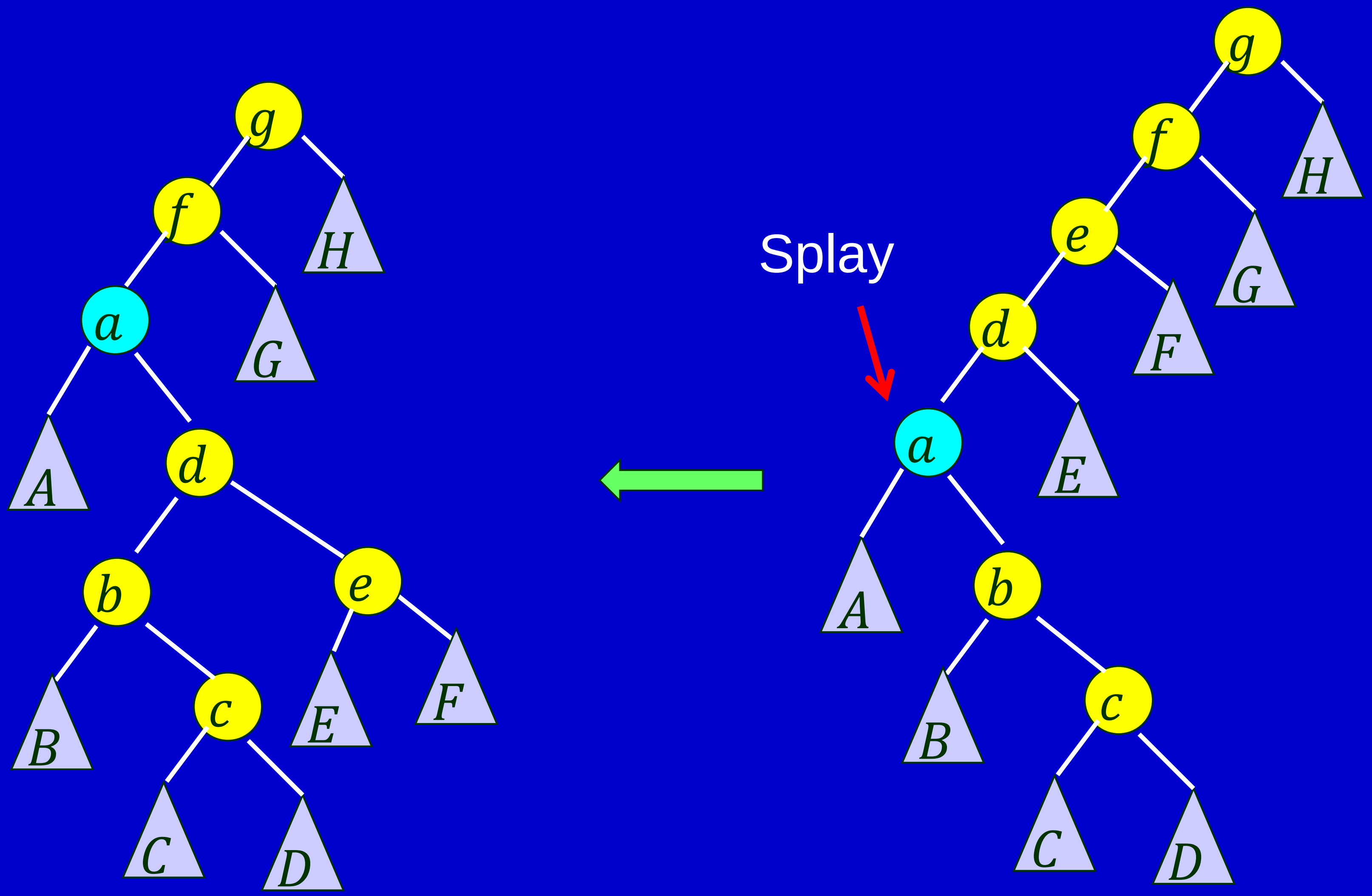
Splay Step 4



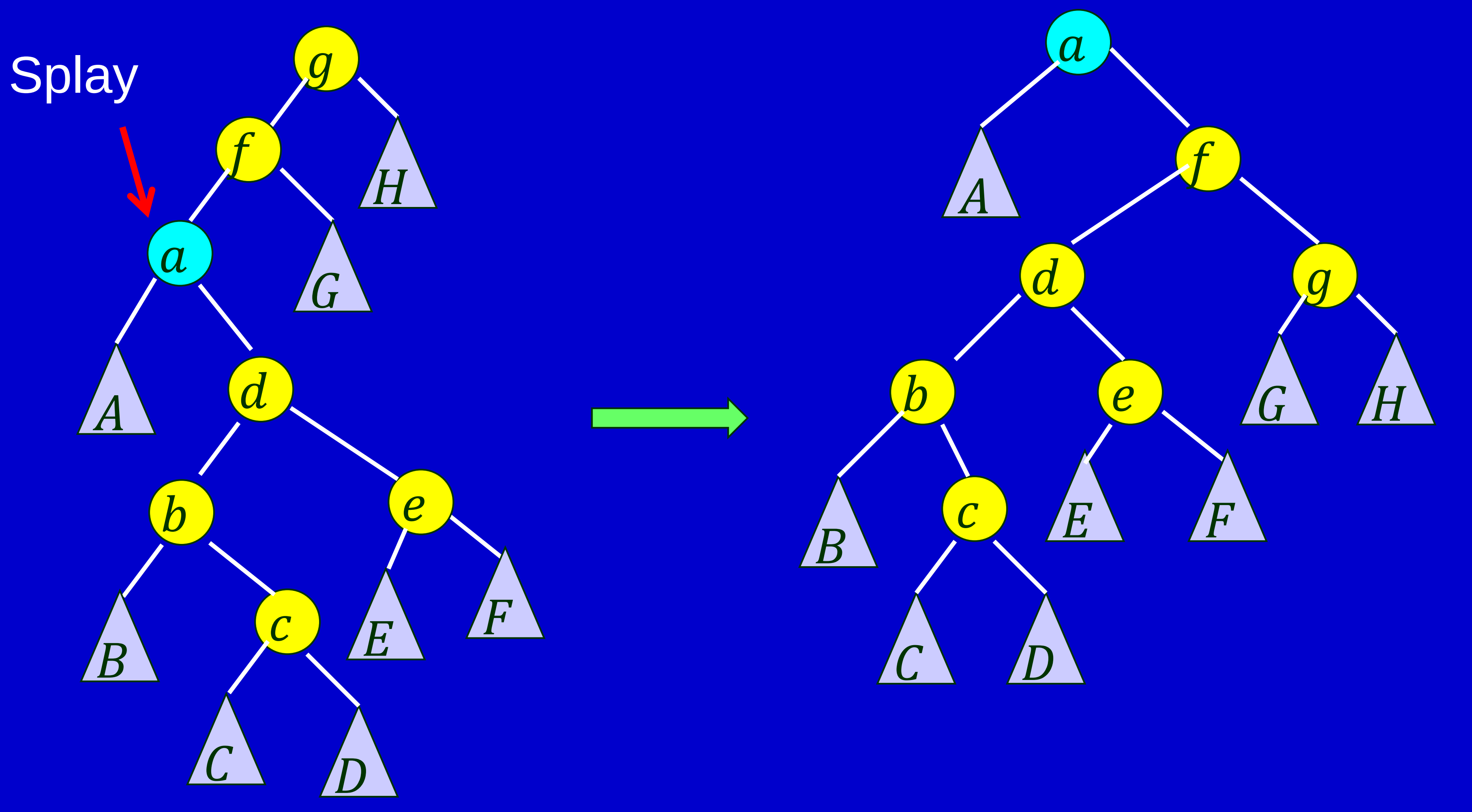
Example 2 – All zig-zig Steps



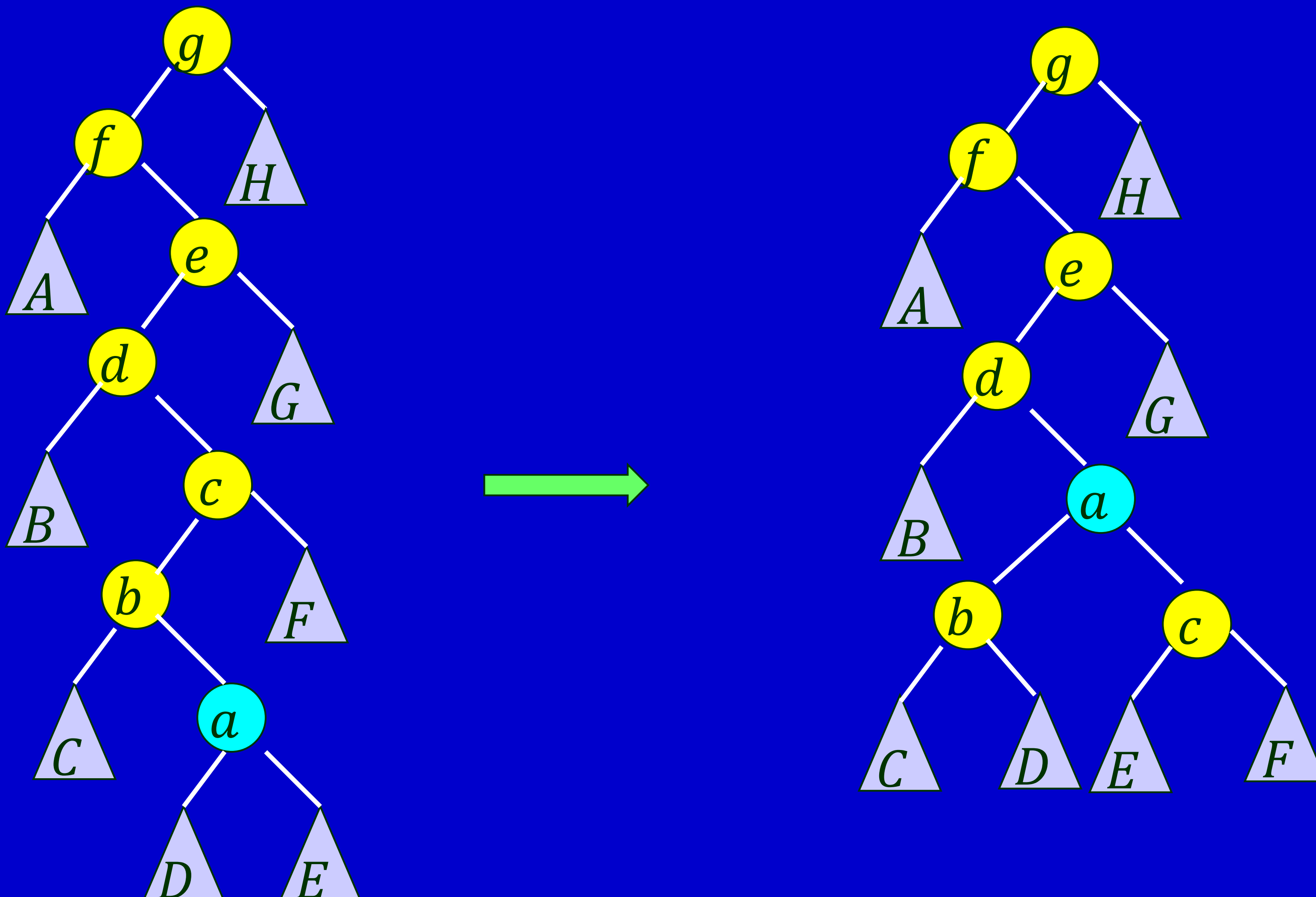
Example 2 (cont'd)



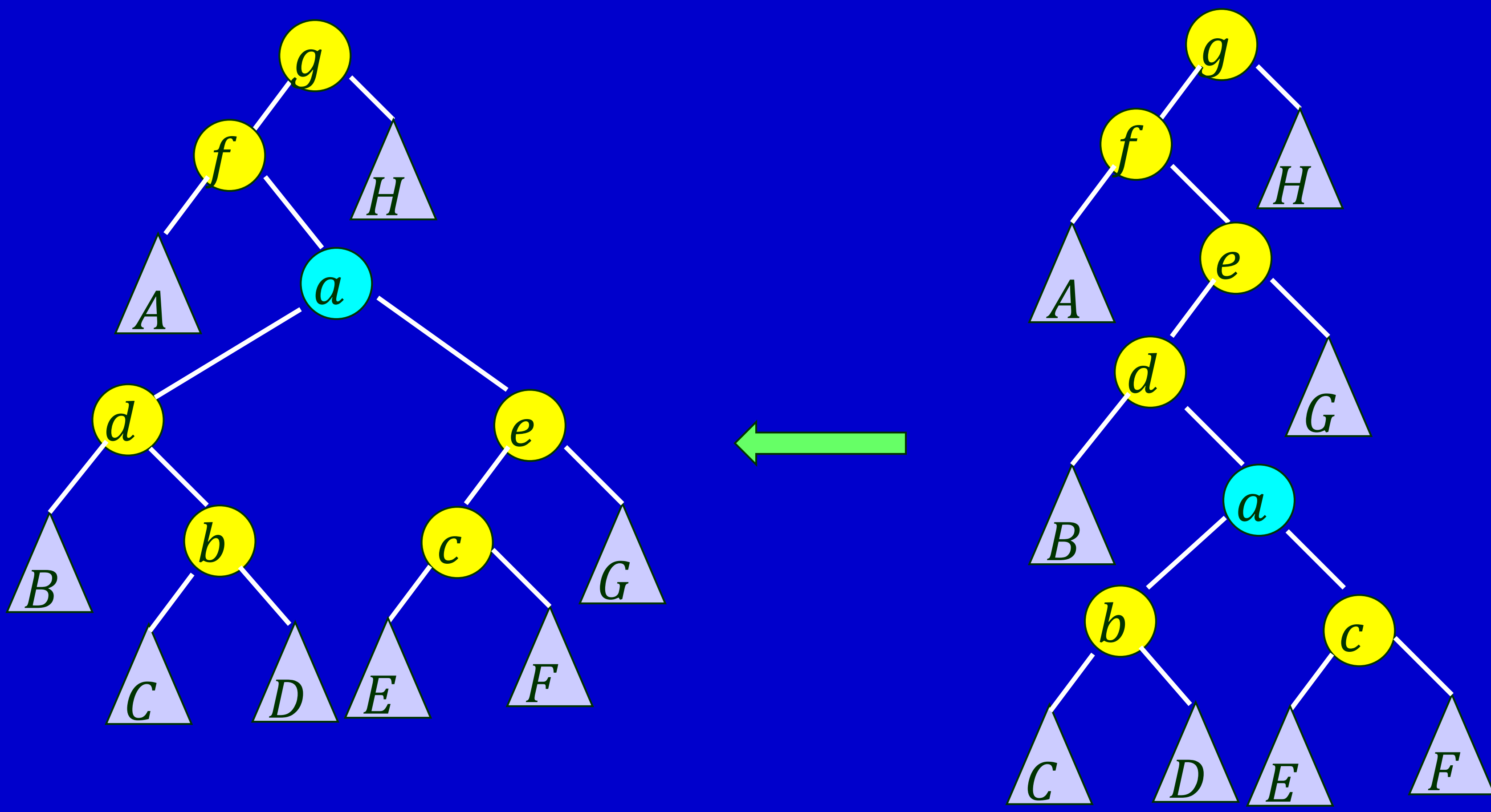
Example 2 (cont'd)



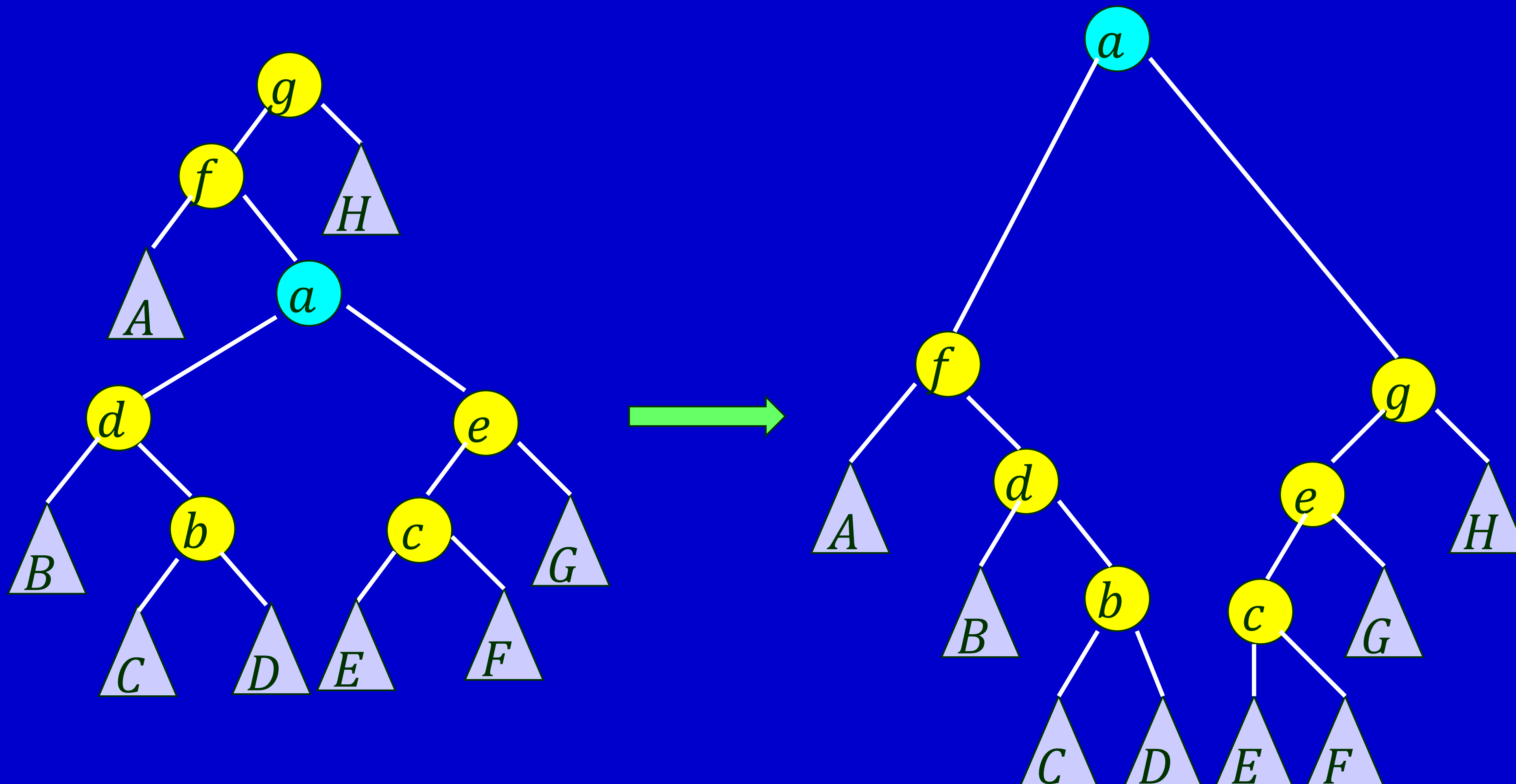
Example 3 – All zig-zag Steps



Example 3 (cont'd)



Example 3 (cont'd)

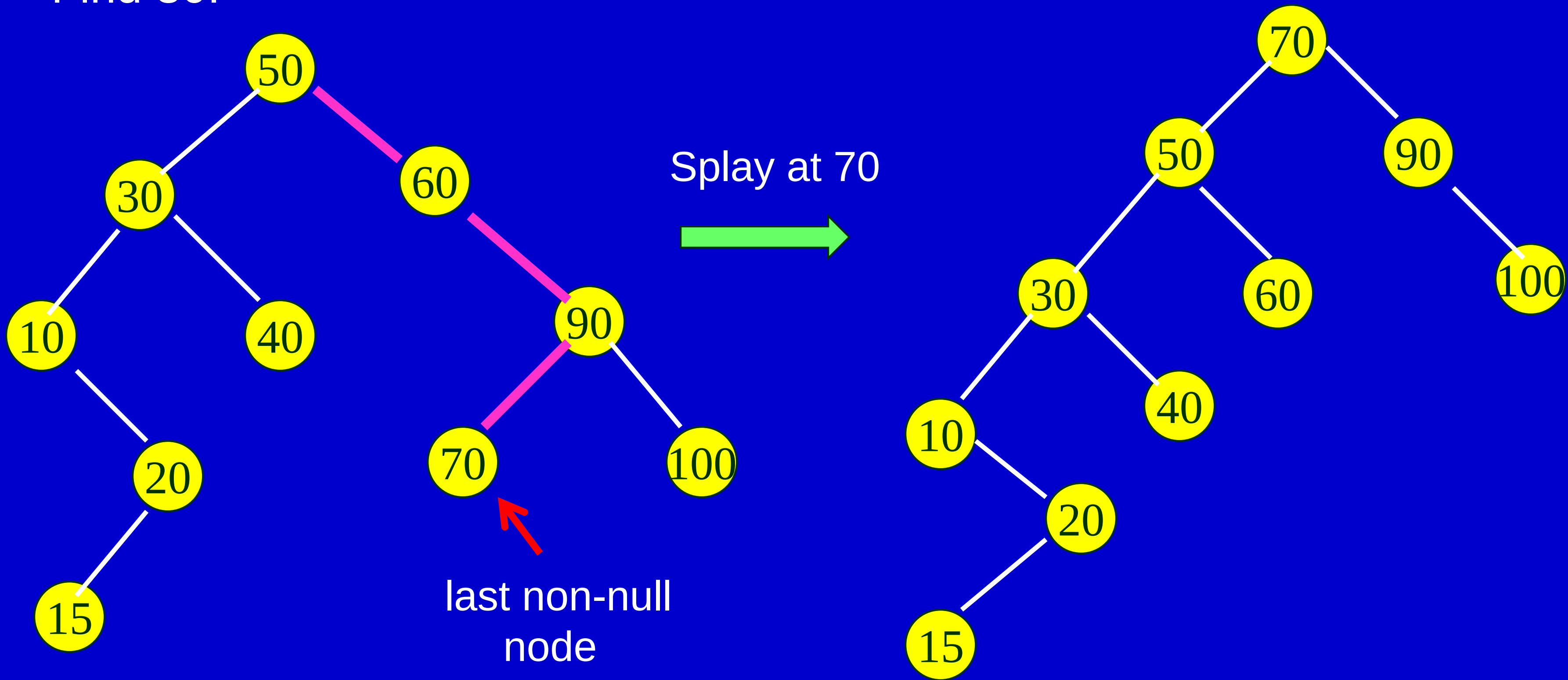


Data Access

All BST update operations can be implemented by splaying.

Access: if the item is stored in node x, splay at x; otherwise, splay at the last *non-null* node (i.e., the leaf node) on the search path.

Find 80:

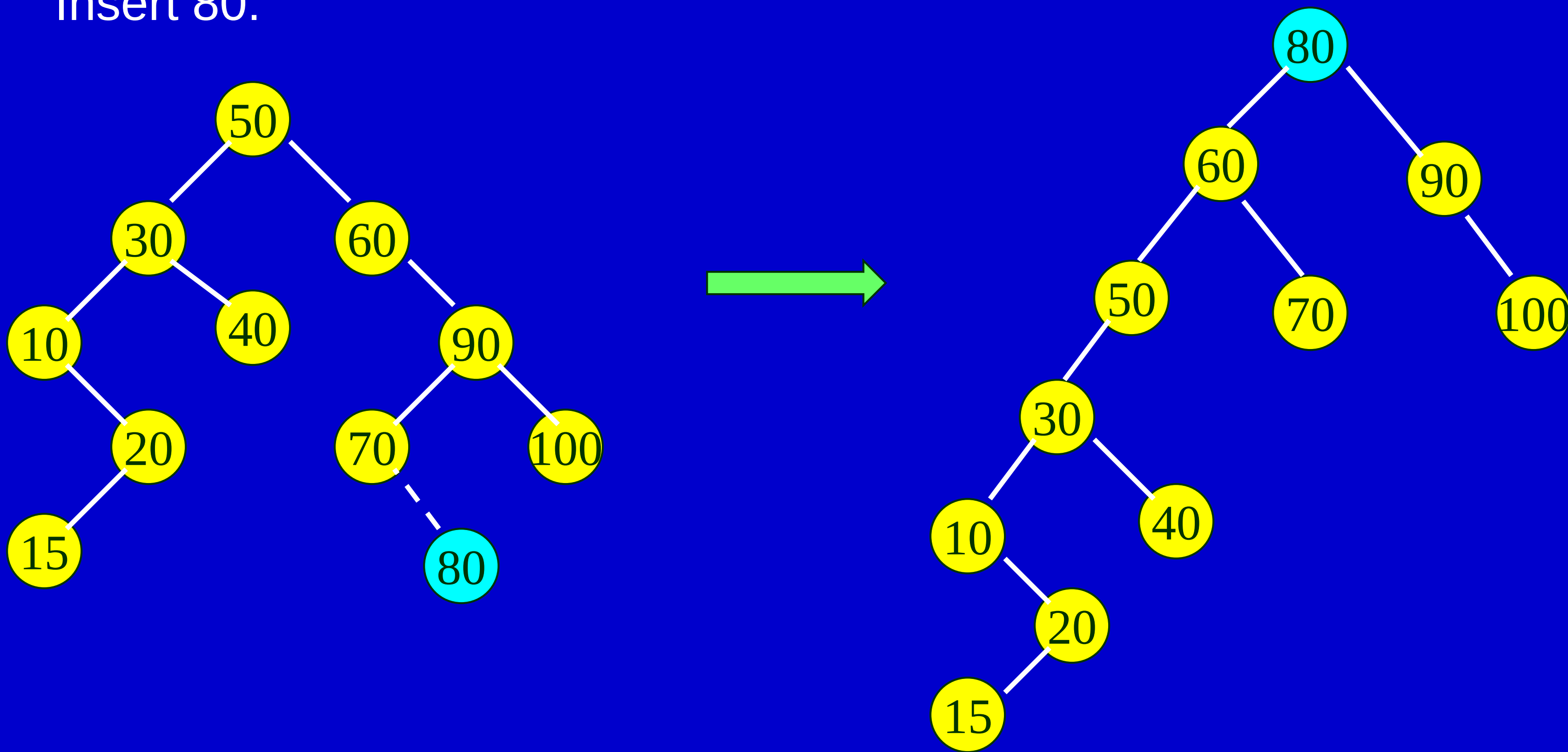


Insertion

1. Search for the item.
2. Create a new node containing the item if not in the tree.
3. Splay the tree at the new node.

BST insertion + Splaying.

Insert 80:



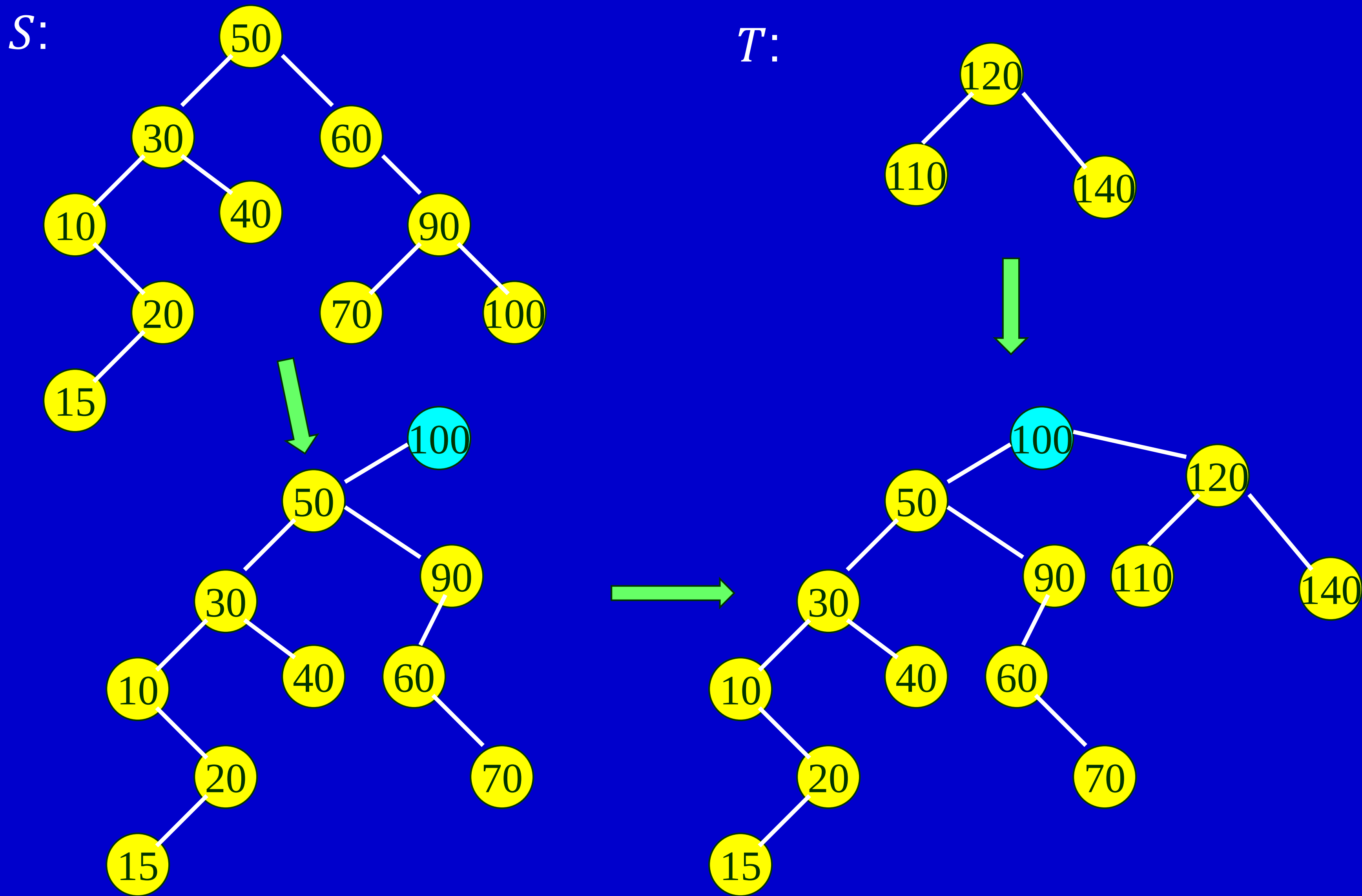
Joining Two BSTs

Two BSTs S and T such that all items in S are less than those in T. Combine S and T into a single tree.

1. Access the largest item in S. It is stored in node x .
   After the access:
   - x is at the root.
   - x has no right child.

2. Making T the right subtree of this root.

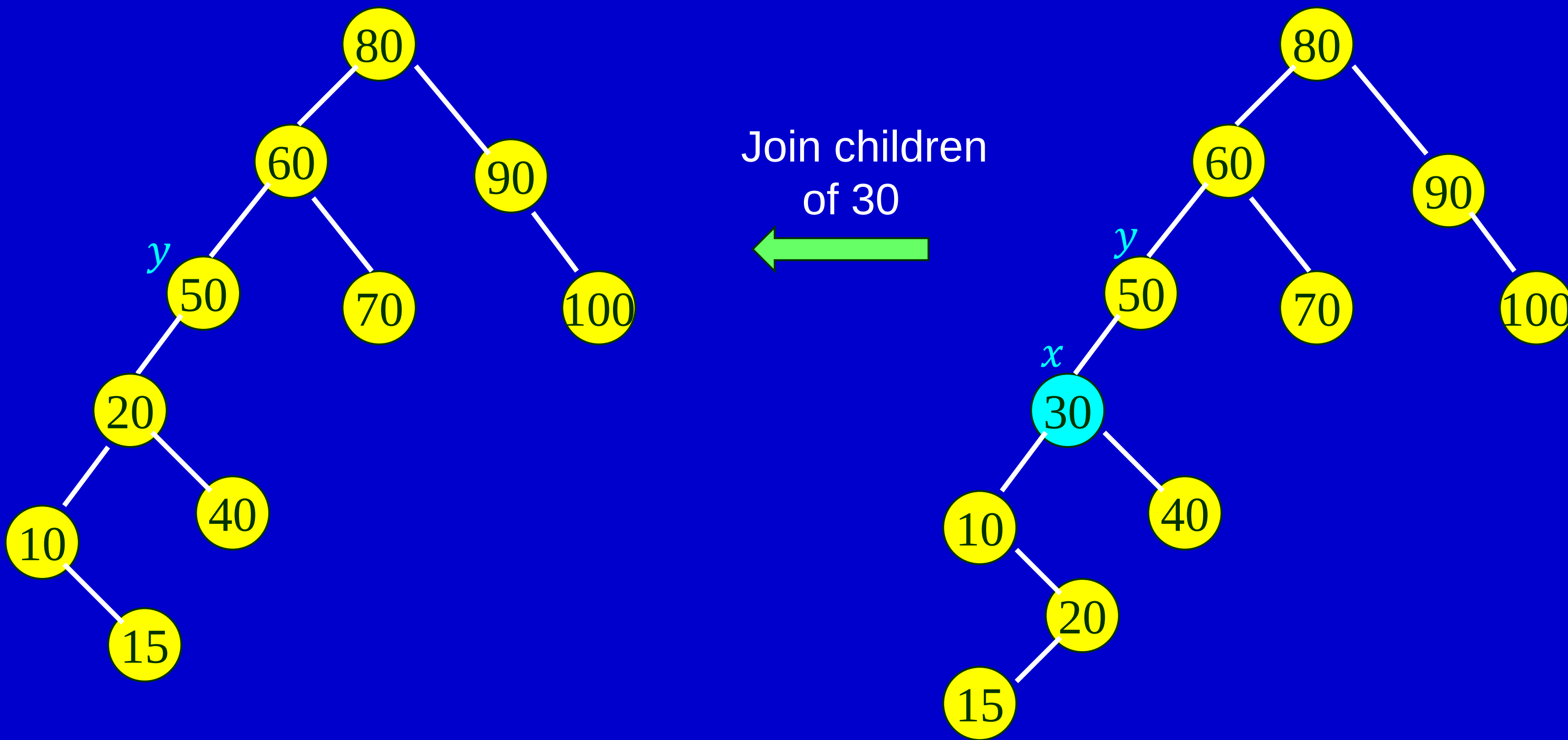
Example 4 (of Join)



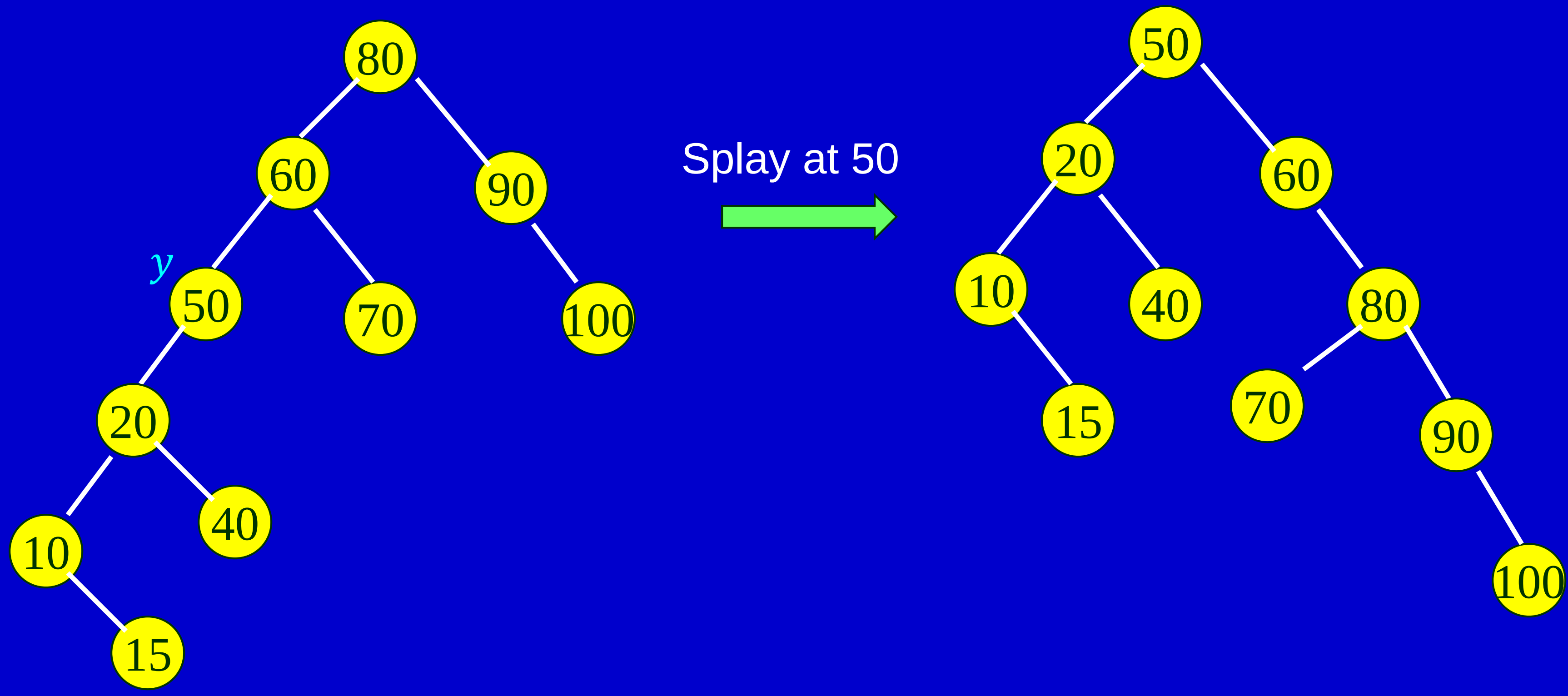
Deletion

1. Search for the item (stored at x with parent y).
2. Replace x by the join of the left and right subtrees of x.
3. Splay at y (or the last node on the search path if the item is not found).

Example Delete 30:



Example 5 (cont'd)



For more on splay trees, we refer to the following paper (where all examples but the joint one are from).

D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652-686, 1985.