CS 261 – Data Structures

AVL Recap

BST Iterator

Insertion

- After inserting a node, check each of the node's ancestors for consistency with the rules of AVL.

- **balance factor** = (height left) – (height right)

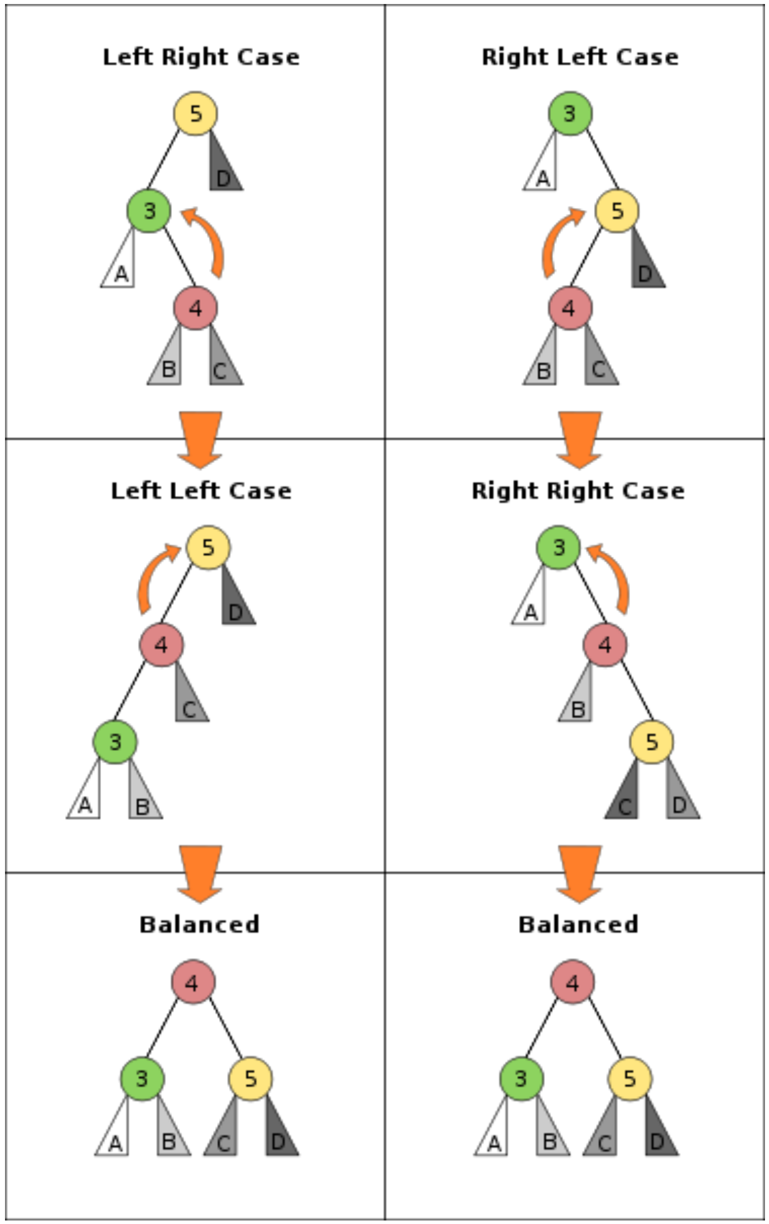
- If balance factor = ±2 ➔ node is unbalanced

Four Cases where Two are Symmetric

- Let
  - C be the root of the unbalanced subtree
  - R right child of C
  - L left child of C
  - BF(C) be the balance factor of C

Four Cases

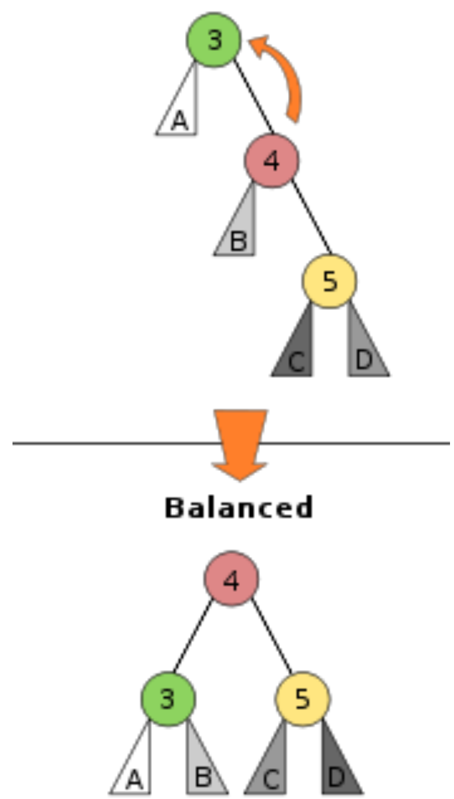
Heavy
left child



Heavy
right child

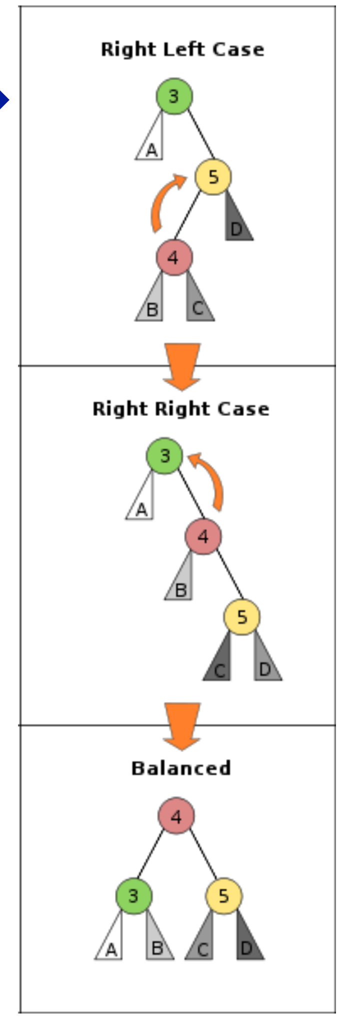
Left Rotation Case

- If BF(C) = -2 ➔ right outweights left
  - If BF(R) ≤ 0 ➔ left rotation with C as root



Right-Left Rotation Case

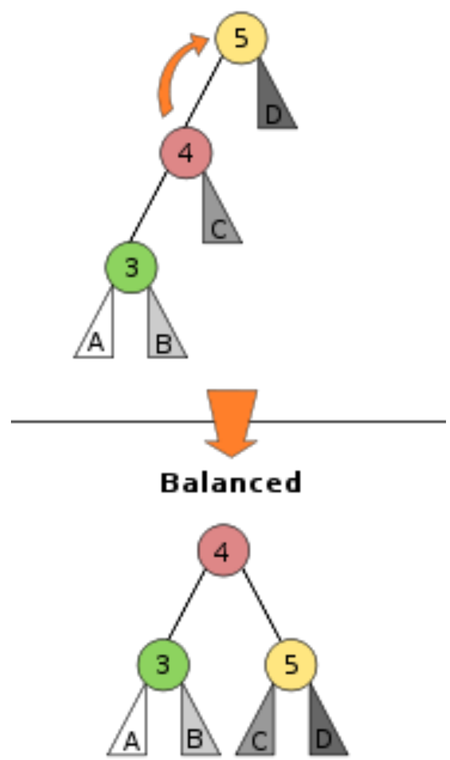
- If BF(C) = -2 ➔ right outweights left
  - If BF(R) > 0 (heavy right child) ➔
    - Right rotation with R as the root
    - Left rotation with C as the root



Right Rotation Case

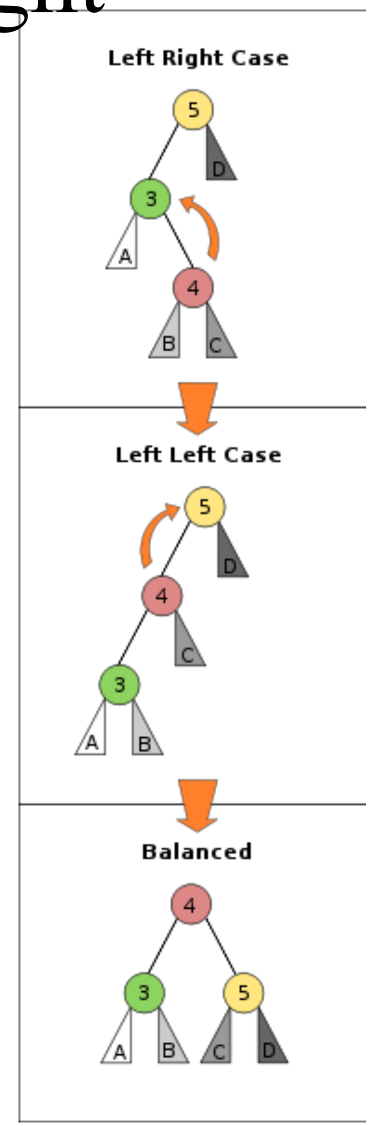
- If BF(C) = +2 ➔ left outweights right

– If BF(L) $\geq$ 0 ➔ right rotation with C as root



Left-Right Rotation Case

- If BF(C) = +2 ➔ left outweights right
  - –If BF(L) < 0 ➔
    - •Left rotation with L as the root
    - •Right rotation with C as the root

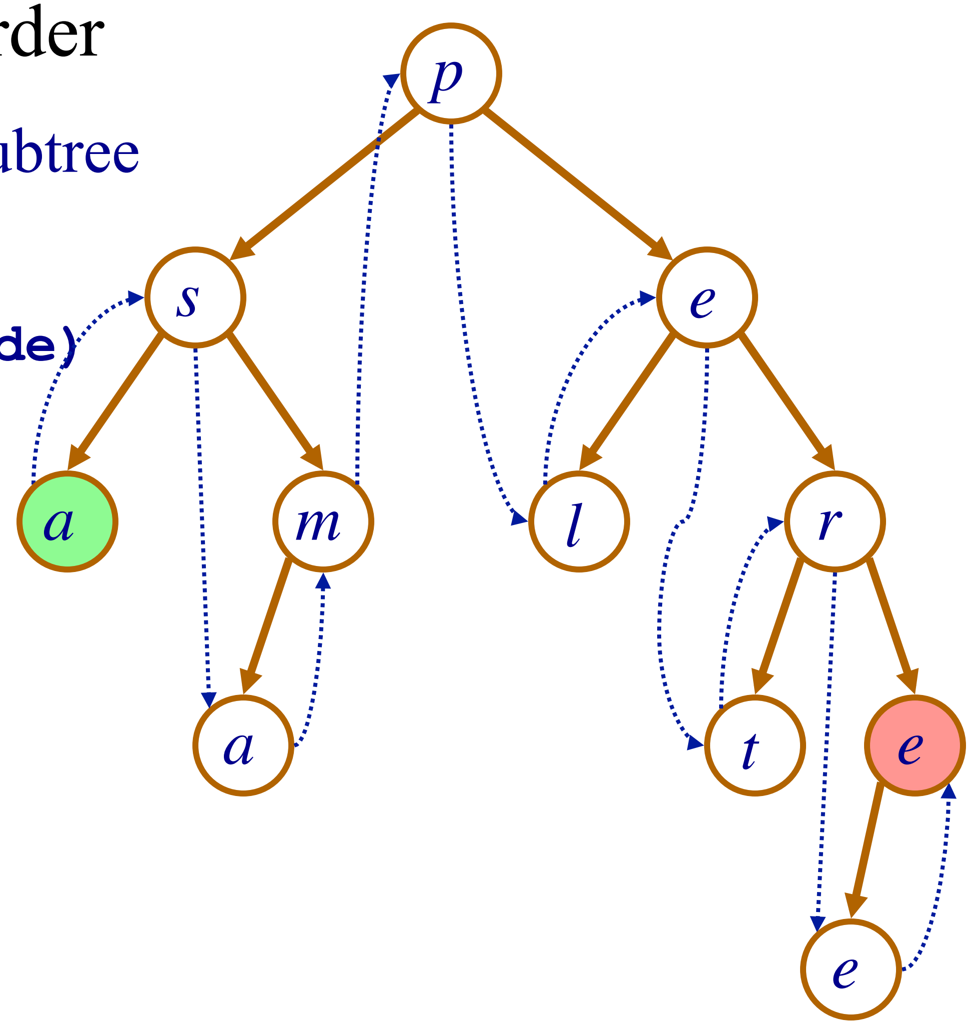


BST Iterator
or
What to do without recursion?

In-Order Traversal

- We used to traverse the tree using a recursion, but...
- For example, process in-order
- → Left subtree, Node, Right subtree

```
void inorder(BinaryNode node)
{
    if (node != null){
      inorder(node.left);
      process (node.obj);
      inorder(node.rght);
    }
  }
```



Example result: a sample tree

Iterator Implementation

But, what if we cannot use recursion?

```
AVLIterInit(&tree, &itr);

 while(AVLIterHasNext(&itr)){
        Process AVLIterNext(&itr);
}
```

Simple Iterator

- Recursively traverse the tree, placing all node values into a linked list, then use a linked list iterator

- Problem: duplicates data, uses twice as much space

- Can we do better?

Yes → Use a Stack

- **Simulate recursion using a stack**
- Suppose we want to iterate over nodes in the tree **in order**
- **Then:** Stack a path as we traverse down to the smallest (leftmost) element
- Note that other iterations (post-order, pre-order) can also be implemented

BST Iterator: Algorithm

- Main Idea
  - `Next`
    - Returns the top of the stack
  - `HasNext`
    - Returns true if there are elements left to iterate
    - Sets up the next `Next` call by making sure that the smallest element is on the top of the stack

Iterator Implementation

```
AVLIterInit(&tree, &itr);

while(AVLIterHasNext(&itr)){
        /* Do something with */
        Process AVLIterNext(&itr);
}
```

BST Iterator: Algorithm

`Initialize`: create an empty stack

BST Iterator: Algorithm

`hasNext`:

if stack is empty

perform `slideLeft` on the current node

else

pop stack (node n)

`slideLeft` on right child of n

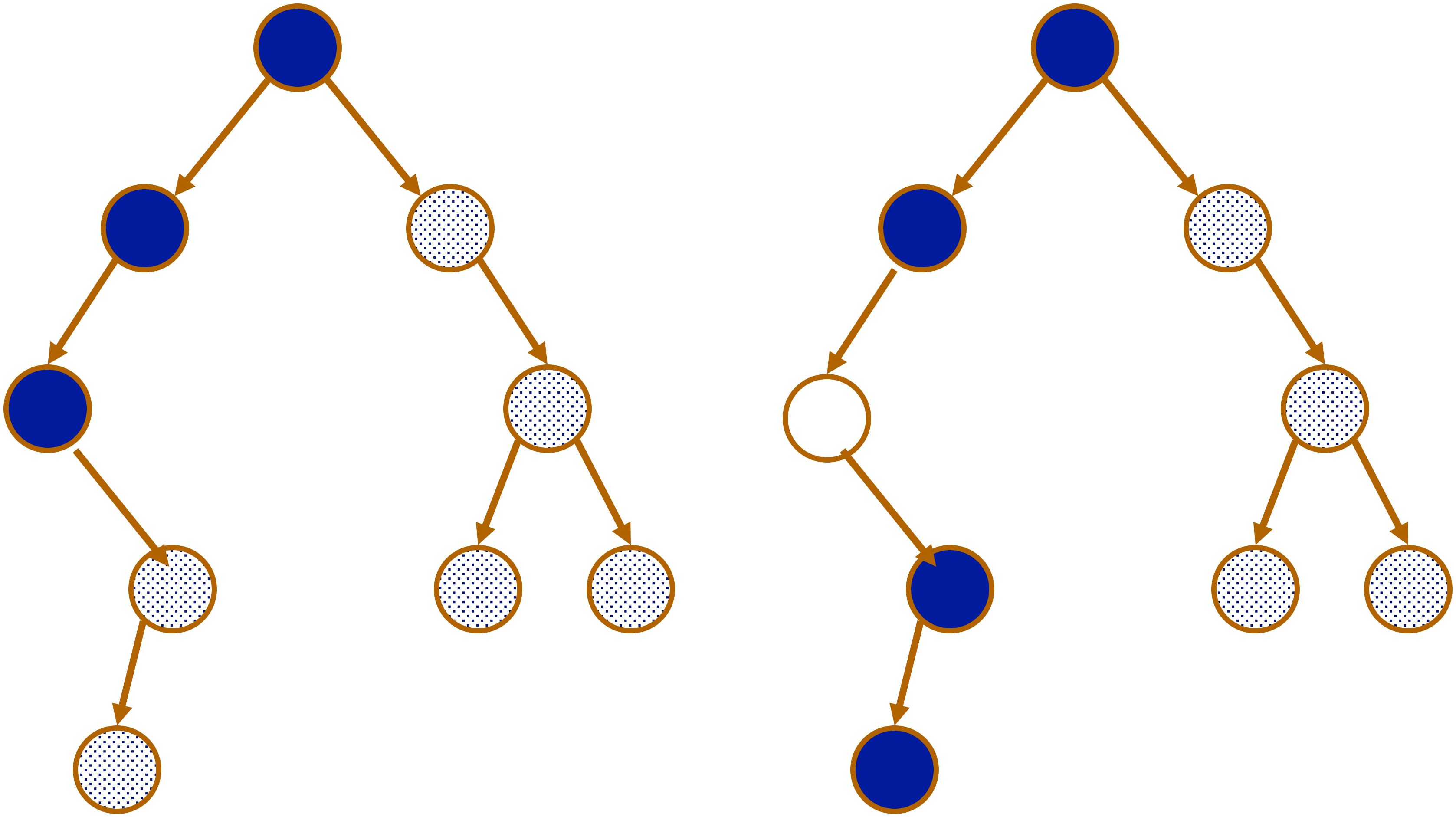
if stack is not empty

return true

else

return false

In-Order Enumeration: Sliding Left



Stack holds the path to the leftmost node = next node you can go UNDER = path to the next smallest element

Yes → Use a Stack

- Useful routine for the iteration in order:

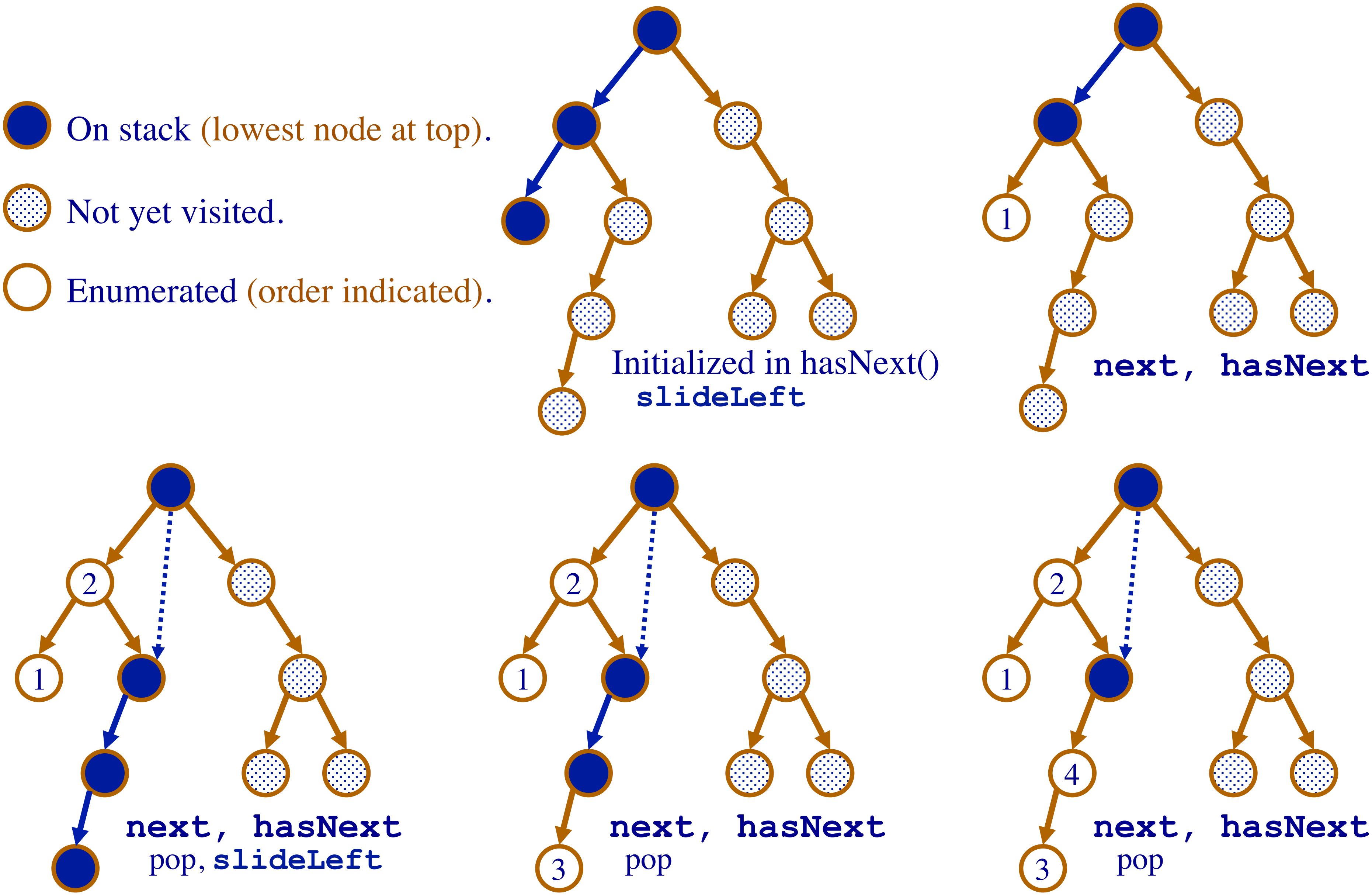
```
void _slideLeft(struct Stack *stk,
                struct Node *n)
{
    while (n != 0) {
      pushStack(stk, n);
      n = n->left;
    }
}
```

BST Iterator: Algorithm

`Next:`

return the value of node on top of stack (but do not pop node)

In-Order Enumeration: Example

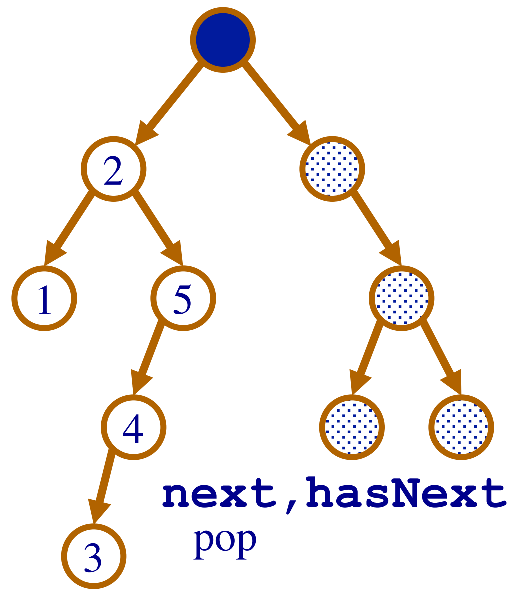


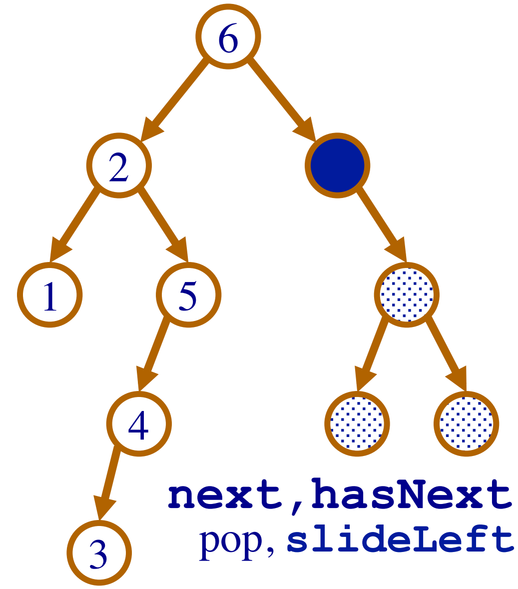
In-Order Enumeration: Example (cont.)

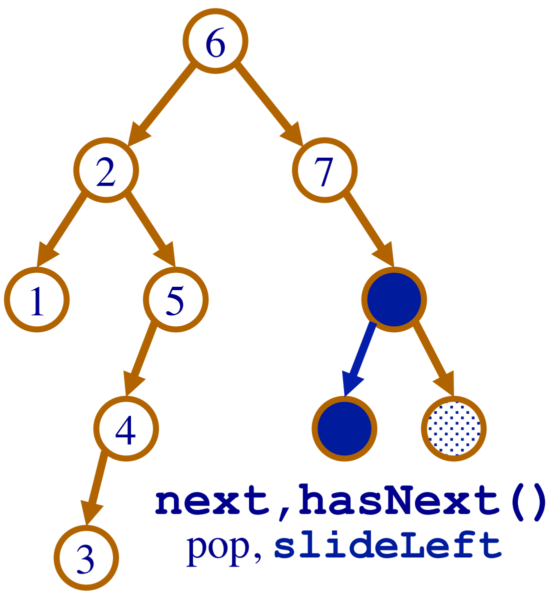
On stack (lowest node at top).

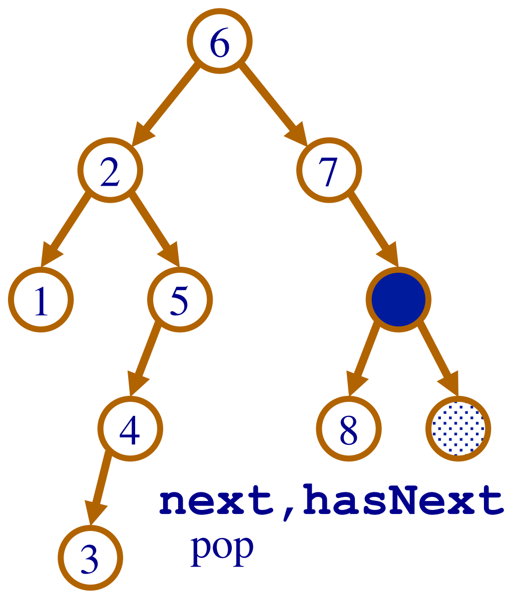
Not yet visited.

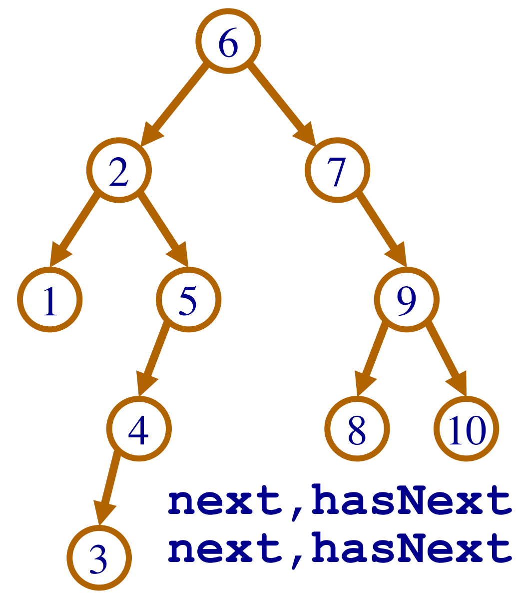
Enumerated (order indicated).











Other Traversals

- Pre-order and post-order traversals also use a stack
- Breadth-first traversal uses a queue – how?
- Depth-first traversal uses a stack – how?