CS261 – Data Structures

Binary Tree Iterator

Goals

- In-Order traversal that supports the Iterator Interface (HasNext, Next)
  - Concepts
  - Implementation

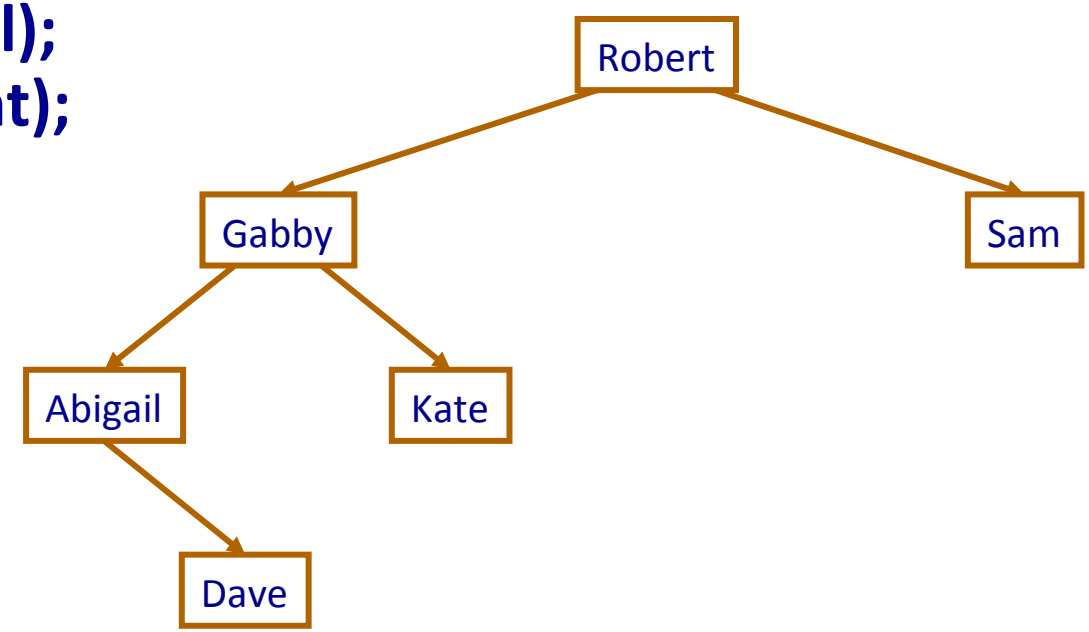
Simple Iterator

- Simple iterator → recursively traverse 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?

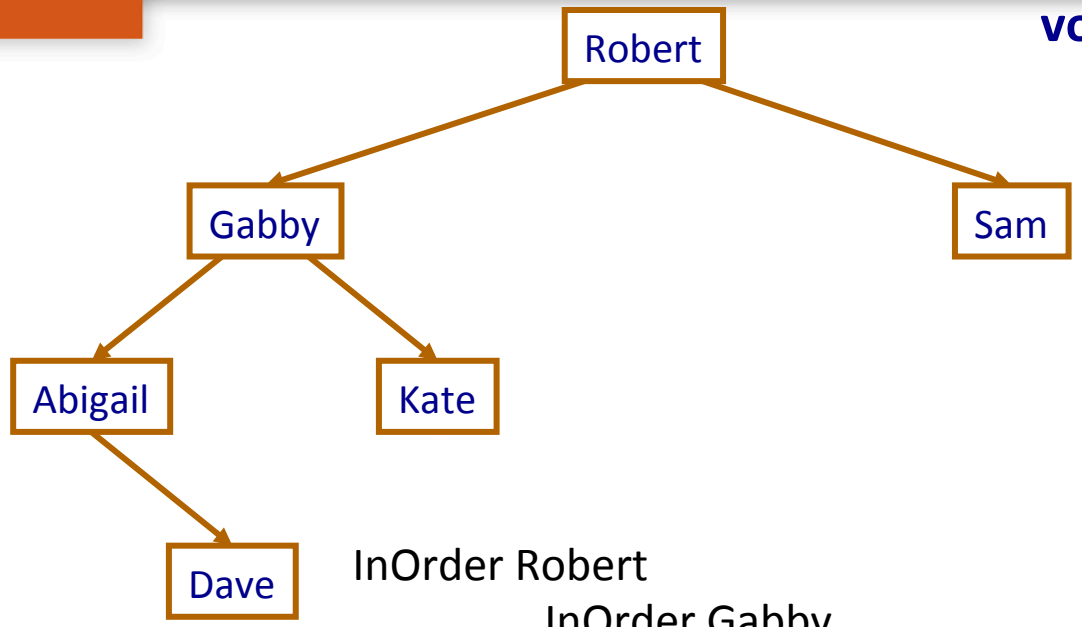
Exercise

What is being stored in the process stack?

```
void inorder(struct Node *node) {
    if (node != 0){
      inorder(node->left);
      process (node->val);
      inorder(node->rght);
    }
  }
```



Exercise




```
void inorder(struct Node *node) {
    if (node != 0){
      inorder(node->left);
      process (node->val);
      inorder(node->rght);
    }
}
```

InOrder Robert
InOrder Gabby
InOrder Abigail
InOrder NULL
Process Abigail
InOrder Dave
InOrder NULL
Process Dave
InOrder NULL
Process Gabby
InOrder Kate

When Process Abigail:
Robert
Gabby
Abigail
are all unfinished!

When Process Dave
Robert
Gabby
Abigail
Dave
are all unfinished!

When Process Gabby
Robert
Gabby
are all unfinished!

Process Stack represents a path to the leftmost unprocessed node!!

Solution → Replace Process Stack

- Simulate recursion using a stack
- Stack path as we traverse down to the leftmost element (smallest in BST)
- Useful routine:

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

Binary Tree In-Order Iterator

- Main Idea
  - ***Next*** returns the top of the stack
  - ***HasNext***
    - Returns true if there are elements left (on stack) to iterate
    - Sets up the subsequent call to 'Next()' by making sure the leftmost node (smallest in BST) element is on top of the stack. It does this by calling _slideLeft on the node's right child

BST In-Order Iterator: Algorithm

Initialize: create an empty stack

hasNext:

if stack is empty perform slide left on root

otherwise

let n be top of stack

pop n

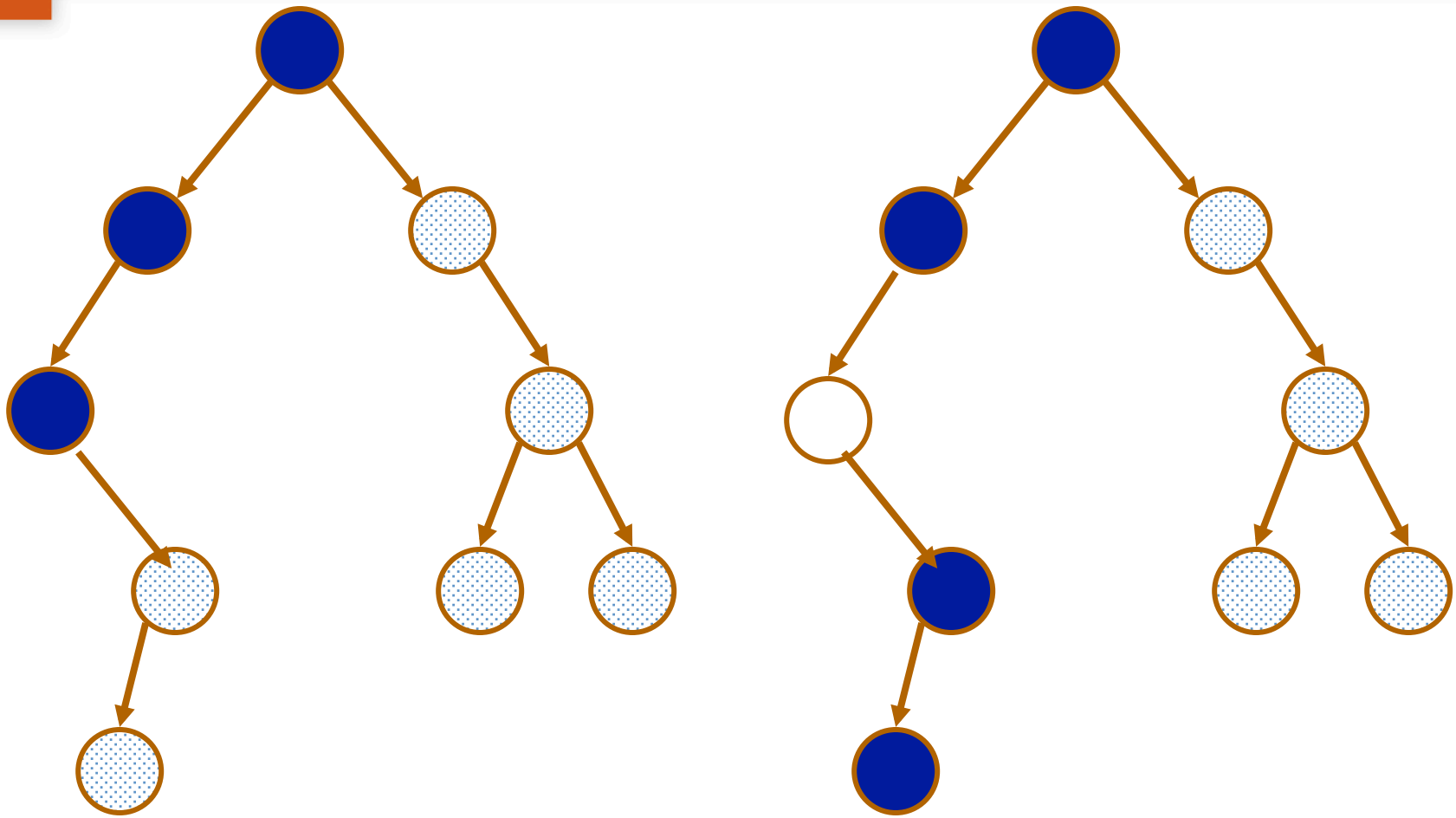
slide left on right child of n

return true if stack is not empty (false otherwise)

next:

return value of node on top of stack (but don't pop node)

In-Order Enumeration: Sliding Left



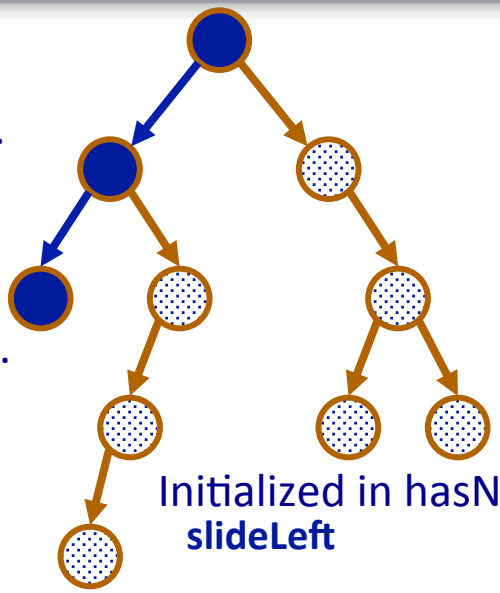
Stack holds the path to the leftmost node => next node you can go UNDER => path to the next smallest element in a BST

In-Order Iterator: Simulation

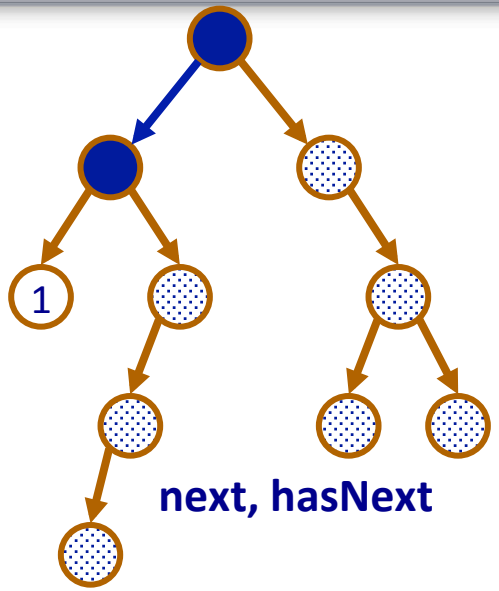
On stack (lowest node at top).

Not yet visited.

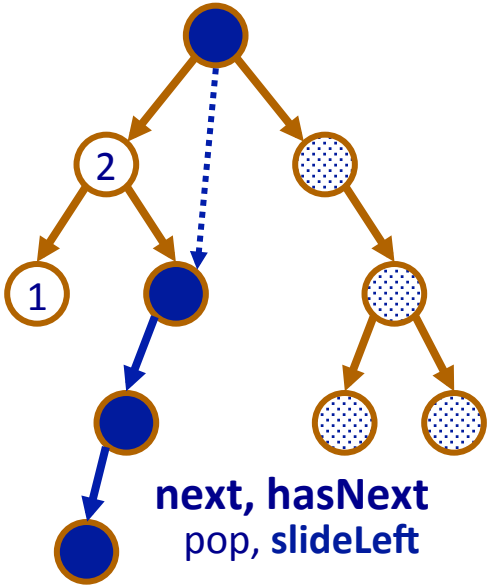
Enumerated (order indicated).



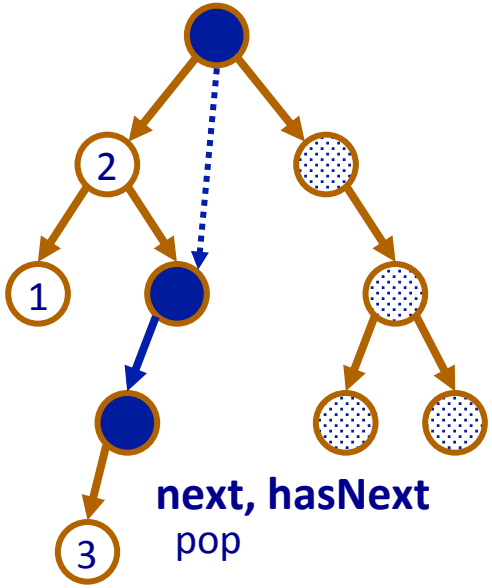
Initialized in hasNext()
slideLeft



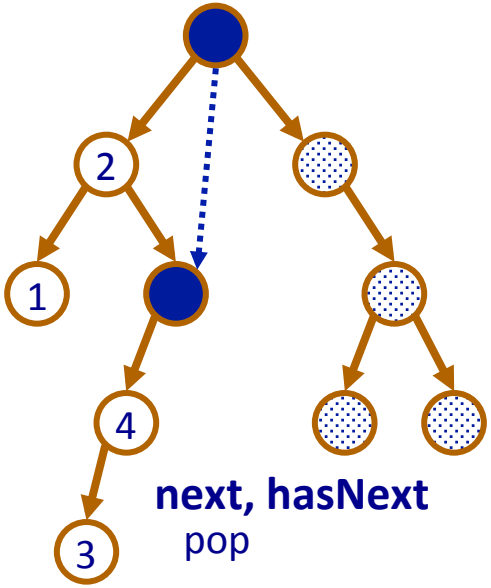
next, hasNext



next, hasNext
pop, **slideLeft**



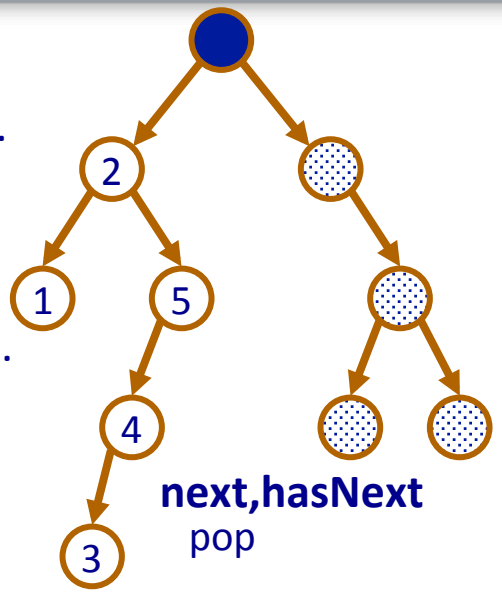
next, hasNext
pop

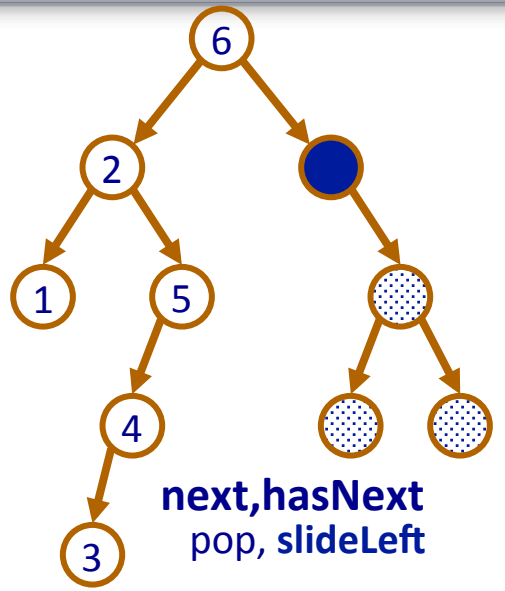


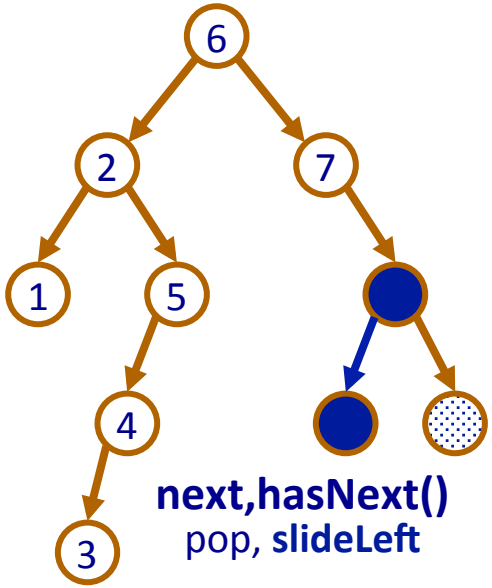
next, hasNext
pop

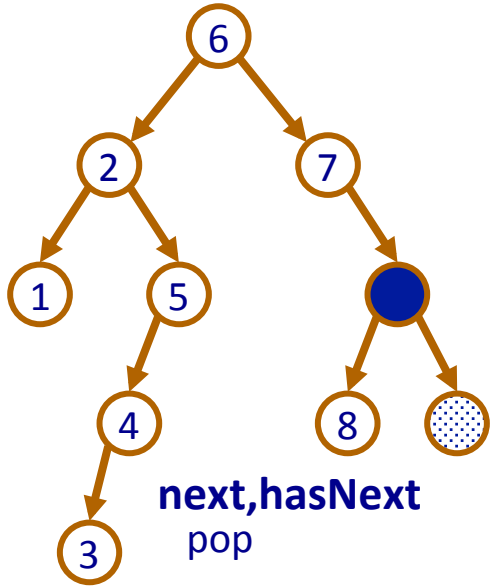
In-Order Iterator: Simulation

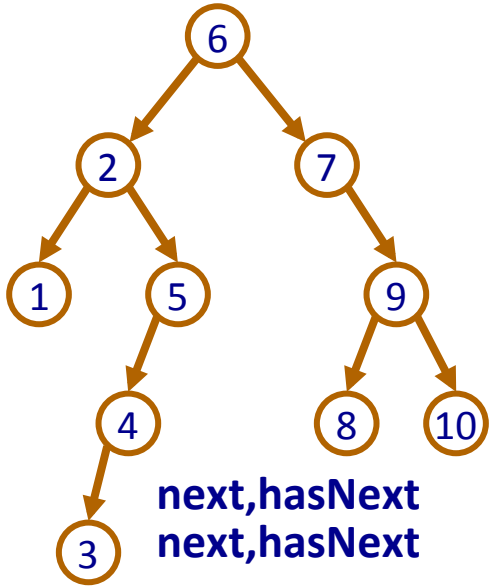
- On stack (lowest node at top).
- Not yet visited.
- Enumerated (order indicated).











BST In-Order Iterator: Algorithm

Initialize: create an empty stack

hasNext:

if stack is empty perform slide left on root

otherwise

let n be top of stack

pop n

slide left on right child of n

return true if stack is not empty (false otherwise)

next:

return value of node on top of stack (but don't pop node)

Complexity?

- Each nodes goes on the stack exactly one time
- Each node is popped off the stack exactly one time
- O(N)

Other Traversals

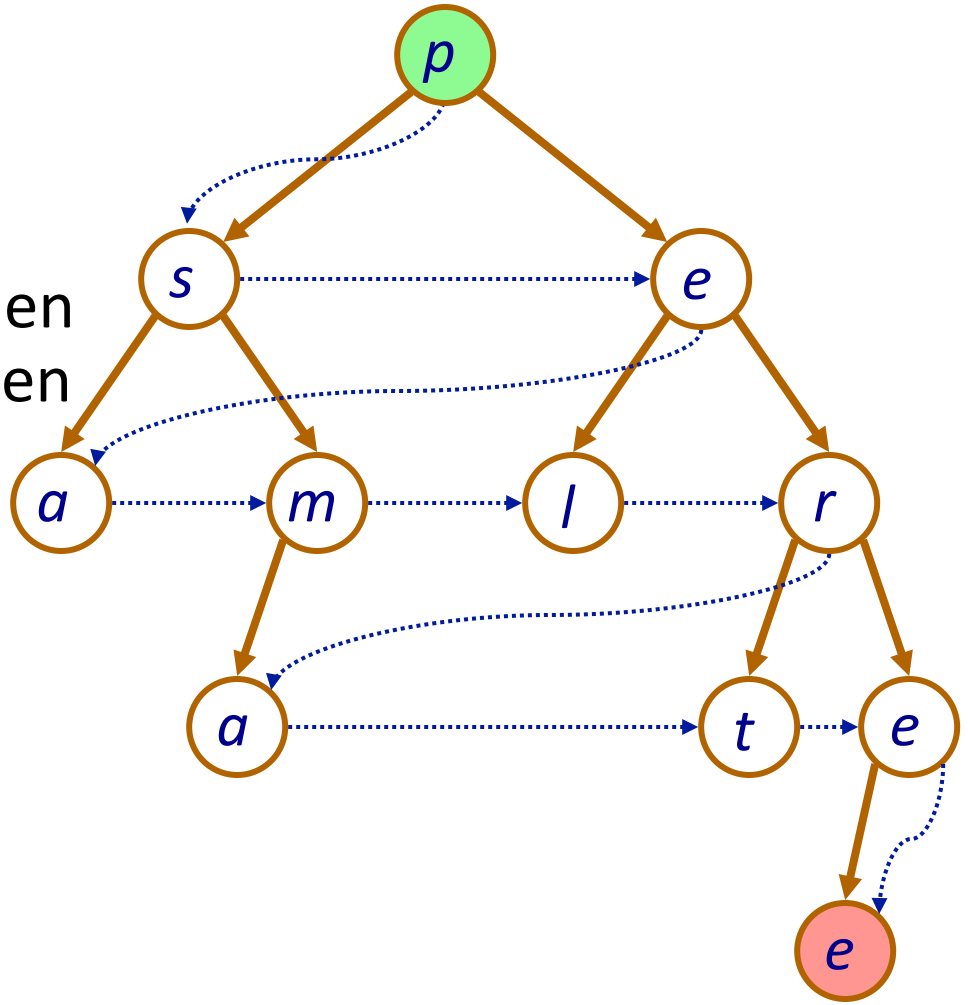
- Pre-order and post-order traversals also use a stack
- See Chapter 10 discussion

Level-Order Iteration

Haven't seen this traversal yet:

- Traverse nodes a level at a time from left to right
- Start with root level and then traverse its children and then their children and so on

- Implementation?



Example result: p s e a m l r a t e e

Your Turn

Complete Worksheet #30: BST Iterator