Binary Tree Iterator
Recursive Traversal

```c
void inorder(struct Node *node) {
    if (node != 0) {
        inorder(node->left);
        process (node->val);
        inorder(node->right);
    }
}
```
• In-Order traversal that supports the Iterator Interface (HasNext, Next)
  – Concepts
  – Implementation
Simple Iterator

• Simple iterator $\rightarrow$ 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?

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

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

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

When Process Gabby:
- Robert
- Gabby
  - Abigail
- Kate
- Dave
- Process Gabby
  - InOrder Robert
  - InOrder Gabby
  - InOrder Abigail
  - InOrder NULL
  - Process Abigail
  - InOrder Dave
  - InOrder NULL
  - Process Dave
  - InOrder NULL
  - InOrder NULL
  - Process Dave
  - InOrder NULL

When Process Dave:
- Robert
- Gabby
- Abigail
- Dave
  - Process Dave
  - InOrder NULL

When Process Abigail:
- Robert
- Gabby
- Abigail
- Process Abigail
  - InOrder NULL
  - Process Abigail
  - InOrder NULL

are all unfinished!
• Simulate recursion using a stack

• Stack path as we traverse down to the leftmost element *(smallest in BST)*

• Useful routine:

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

  – *Next* returns the top of the stack (e.g. the next element you’ll go UNDER in Euler Tour)

  – *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).

1. **Initialized in hasNext() slideLeft**
2. **next, hasNext**
3. **next, hasNext**
4. **next, hasNext**
In-Order Iterator: Simulation

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

---

1. **next, hasNext**
2. **pop**
3. **next, hasNext**
4. **pop, slideLeft**
5. **next, hasNext**
6. **pop**
7. **next, hasNext**
8. **next, hasNext**
9. **next, hasNext**
10. **next, hasNext**

---

**next, hasNext()**
**pop, slideLeft**
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 node 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