CS 261 – Data Structures

BST Iterator

or

How to traverse the tree without recursion?
Example: In-Order Traversal

• We can traverse the tree using a recursion
• For example: in-order traversal

```java
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? For example, the end user is not familiar

```
Initialize(&tree, &itr);
while( HasNext(&itr) ){
    Process( Next(&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 as in-order

• Then: Stack a path as we traverse down to the smallest (leftmost) element

• Other iterations (post-order, pre-order) can also be implemented
In-Order: Example

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

Initialized in hasNext()
slideLeft
In-Order: Example

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

Initialized in `hasNext()`

```
slideLeft
```

```
next, hasNext
```
In-Order: Example

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

Initialized in hasNext()

next, hasNext

slideLeft
In-Order: Example

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

Initialized in `hasNext()`

`next, hasNext`

`slideLeft`

`next, hasNext`

`pop, slideLeft`

`next, hasNext`

`pop`

`next, hasNext`

`pop`

`next, hasNext`

`pop`
In-Order: Example (cont.)

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

```
1 5 3 4
2
3
next, hasNext
pop

1 5
4
3
next, hasNext
pop

1 5
4
3
next, hasNext
pop

1 5
4
3
next, hasNext
pop
```
Iterator Implementation

Initialize(&tree, &itr);
while( HasNext(&itr) )
{
    /* Do something */
    Process( Next(&itr) );
}
Implementation

```
struct BSTIterator {
    struct DynArr *stk;
    struct BSTree *tree;
};

struct DynArray {
    struct Node *nodes;
    int size;
    int capacity;
};

struct BSTree {
    struct Node *root;
    int size;
};

struct Node {
    TYPE val;
    struct Node *lft;
    struct Node *rght;
};
```
BST Iterator -- Initialize

```c
void initBSTIter (struct BSTree *tree,
                 struct BSTIterator *itr)
{
    /* Stack as dynamic array */
    int capacity = log(tree->size); /* saves memory */
    itr->tree = tree;
    InitStack(itr->stk, capacity);
}
```
Iterator Implementation

```c
Initialize(&tree, &itr);

while( HasNext(&itr) )
{
    /* Do something */
    Process( Next(&itr) );
}
```
BST Iterator -- Next()

Returns the top of the stack

topStack(itr->stk)

/*this does not remove the top node*/
Iterator Implementation

Initialize(&tree, &itr);

while( HasNext(&itr) )
{
    /* Do something */
    Process( Next(&itr) );
}

BST Iterator – HasNext ()

Two purposes:

• Check if Stack is empty

• Forming Stack
  – When the top of Stack is processed,
  – Insert its right-child's left branch
Sliding Left

Stack holds the path to the leftmost node
BST Iterator – HasNext ( )

if ( isEmpty(itr->stk) )
    Push new nodes into Stack
    from the root down left
else{
    - Pop the top element of Stack
    - Push left descendants of the right child into Stack
}
if (! isEmpty(itr->stk) )
    return TRUE;
else
    return FALSE;
BST Iterator – HasNext ( )

if ( isEmpty(itr->stk) )
    /* push into Stack */
    slideLeft(itr->tree->root);
else{
    current = topStack(itr->stk)  \( \text{read top elem.} \)
    popStack(itr->stk)  \( \text{remove top element} \)
    /* push into Stack */
    slideLeft(current->right)  \( \text{from right child} \)
}...
void _slideLeft(struct DynArray *stk,
               struct Node *current)
{
    while (current != 0) {
        pushStack(stk, current);
        current = current->left;
    }
}
Other Traversal Types

• Pre-order and post-order traversals can also use a stack

• Breadth-first traversal uses a queue – how?

• Depth-first traversal uses a stack – how?
Breadth-Frist Traversal

```c
void PrintBreadthFirstBST(struct BSTree *tree){
    struct listQueue *q;
    struct BSTNode *current = tree->root;
    initListQueue(q);
    addBackListQueue (q, current);
    while(!isEmptyListQueue (q)){
        current = getFrontListQueue(q);
        removeFrontListQueue(q);
        printf("%f ",current->val);
        if(current->left != 0)
            addBackListQueue (q, current->left);
        if(current->right != 0)
            addBackListQueue (q, current->right);
    }
}
```