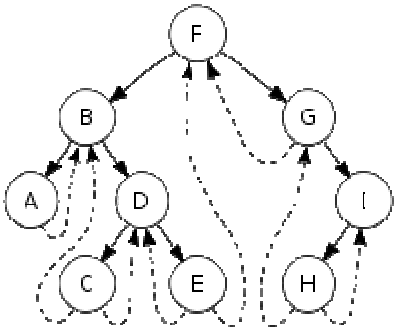


Threaded Binary Trees

Atul Gupta

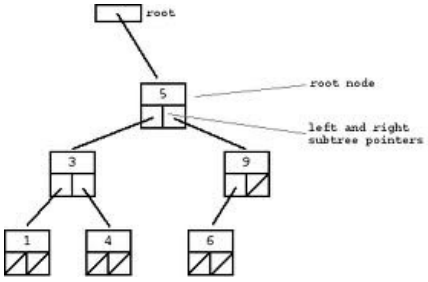
Binary Tree Traversals

- Depth-First Traversals
  - Pre-order
  - Post-order
  - In-order
- Breadth First Traversals

Note that
1. these traversals use stacks/queues as auxiliary data structures
2. It is not simple to determine predecessor/successor for an order traversal
3. Here we demonstrate how to perform traversal without stacks/queues and to get the predecessor/successor for an order traversal

Binary Trees

- An observation: Majority of pointers in a binary tree are NULL
  - e.g. a binary tree with n nodes has n+1 NULL pointers, and they are wasted



Treaded Binary Tree

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node"

Threaded Binary Tree

- makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal
- to discover the parent of a node from a threaded binary tree

Threaded Binary Trees: Classification

- Based on left/right threaded
  - Left-threaded
  - Right-threaded
  - Fully Threaded
- Based on traversal order
  - Preorder threaded
  - In-order threaded
  - Post-order threaded

Normal Binary Tree vs. Threaded Binary Tree

- Normal Binary Tree

Left	data	Right

```
struct BinaryTree {
    struct BinaryTree *left;
    int data;
    struct BinaryTree *right;
}
```

- Threaded Binary Tree

Left	LTag	data	RTag	Right

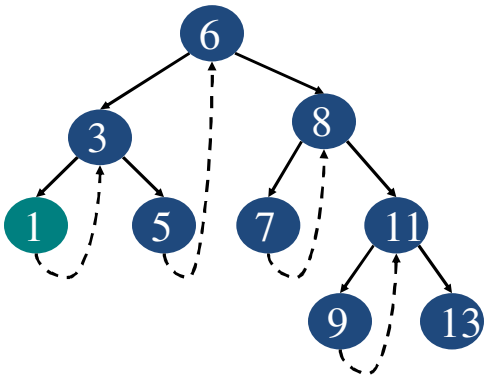
```
struct ThreadedBinaryTree {
    struct ThreadedBinaryTree *left;
    int Ltag;
    int data;
    int Rtag;
    struct ThreadedBinaryTree *right;
}
```

	Meaning
LTag == 0	Left points to in-order predecessor
LTag == 1	Left point to left child
RTag == 0	Left points to in-order successor
RTag == 1	Right points to right child

Non recursive Inorder traversal for a Threaded Binary Tree

1. curr-node node ← leftmost (root)
2. While (curr_node != Null)
   a. print (curr_node)
   b. If (curr_node.RTag == 0) then
      curr_node ← curr_node.right
      go to step 2.
   c. else
      curr_node ← leftmost(curr_node.right)
      go to step 2.

Threaded Tree Traversal

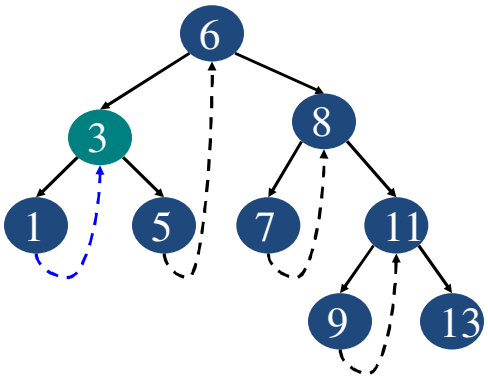


Output
1

Start at leftmost node, print it

9

Threaded Tree Traversal

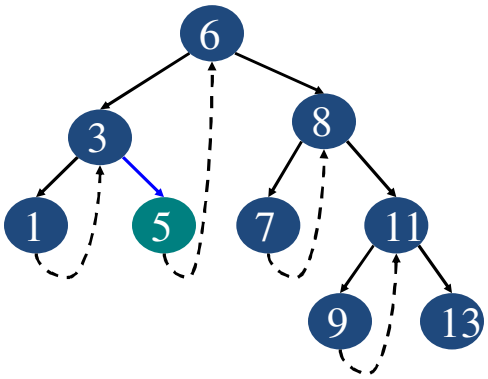


Output
1
3

Follow thread to right, print node

10

Threaded Tree Traversal

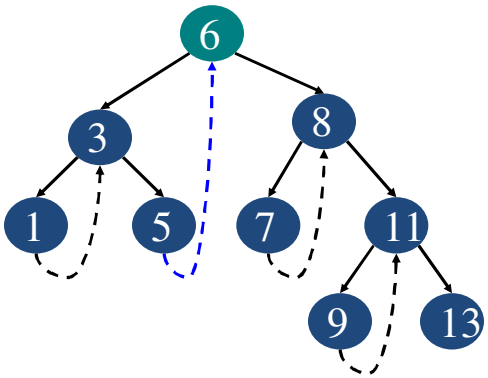


Output
1
3
5

Follow link to right, go to leftmost node and print

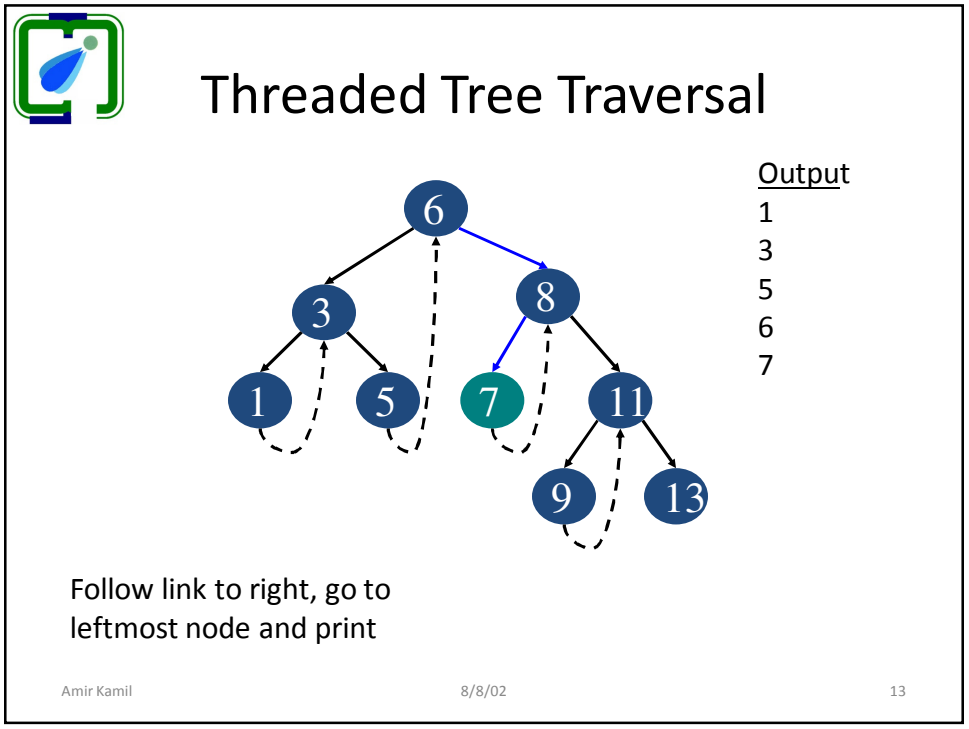
11

Threaded Tree Traversal



Output
1
3
5
6

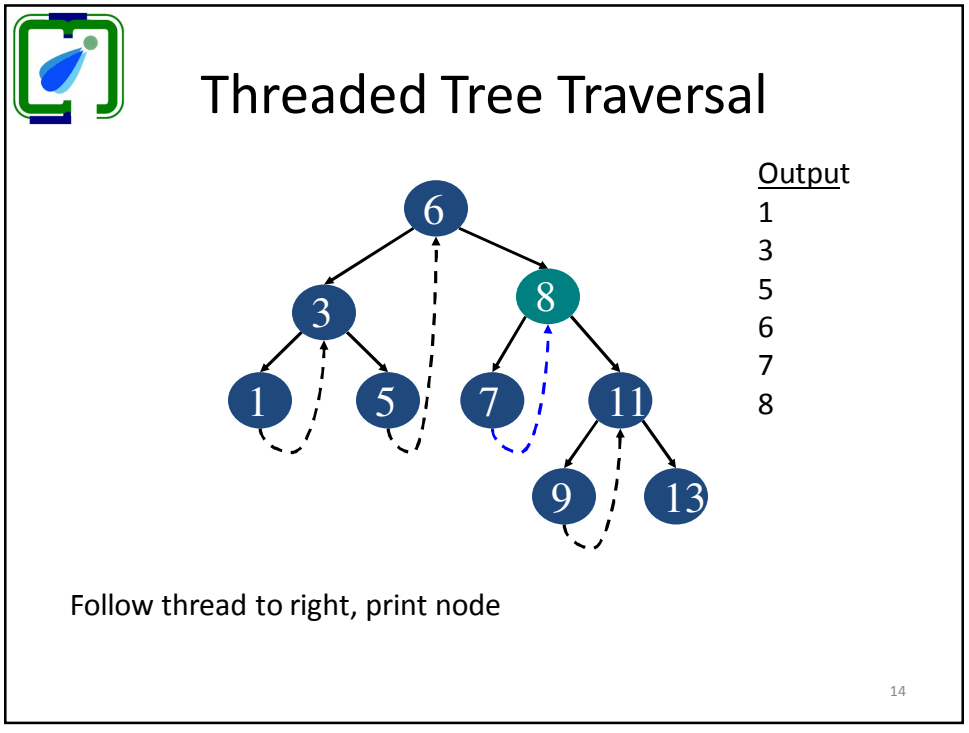
Follow thread to right, print node



Threaded Tree Traversal

Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

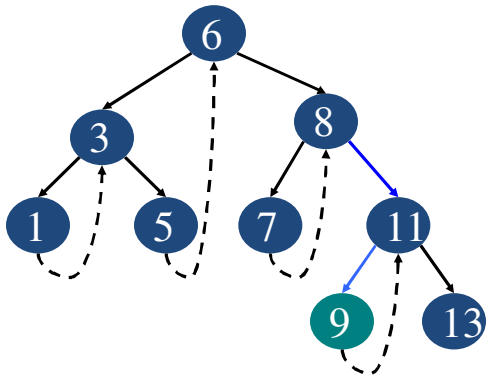


Threaded Tree Traversal

Output
1
3
5
6
7
8

Follow thread to right, print node

Threaded Tree Traversal

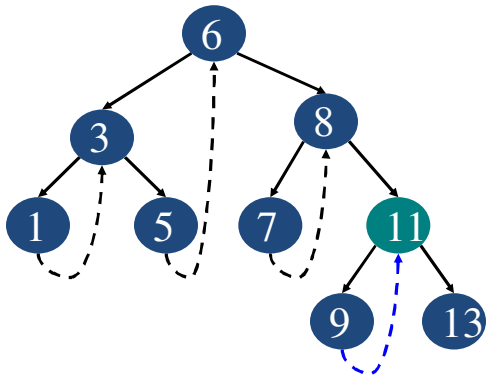


Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

15

Threaded Tree Traversal

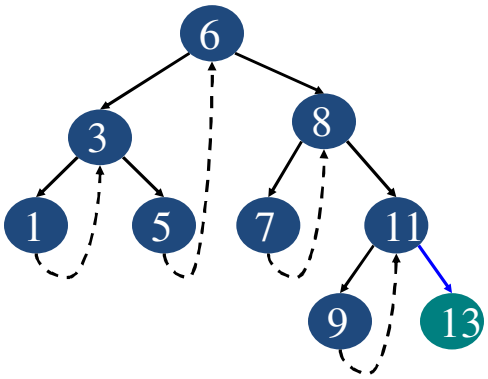


Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

16

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

17

Threaded Tree Traversal Algorithm (In-Order)

```
struct Node* leftMost(struct Node** n) {
   struct Node* ans = n;
   if (ans == null) {
      return null;
   }
   while (ans→left != null) {
      ans = ans → left;
   }
   return ans;
}
```

```
void inOrder(struct Node* n) {
   struct Node* cur = leftmost(n);
   while (cur != null) {
      print(cur);
      if not (cur → RTag) {
         cur = cur → right;
      } else {
         cur = leftmost(cur → right);
      }
   }
}
```

18

Threaded Tree Traversals

- Similarly can also be employed to pre-order and post-order traversals

Summary: Threaded Binary Tree

- A useful idea to utilize the otherwise wasted links to store meaningful information
- Can be threaded using any order traversal
- Useful in applications which require
  - Information about predecessor and successor nodes
  - For efficient order traversals