CSE221

Lecture 9:
Binary Trees

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of
the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Outline

- Binary tree traversal
- Counting binary trees
- Threaded binary trees

Outline

- Binary tree traversal
- Counting binary trees
- Threaded binary trees

Binary Tree Traversal

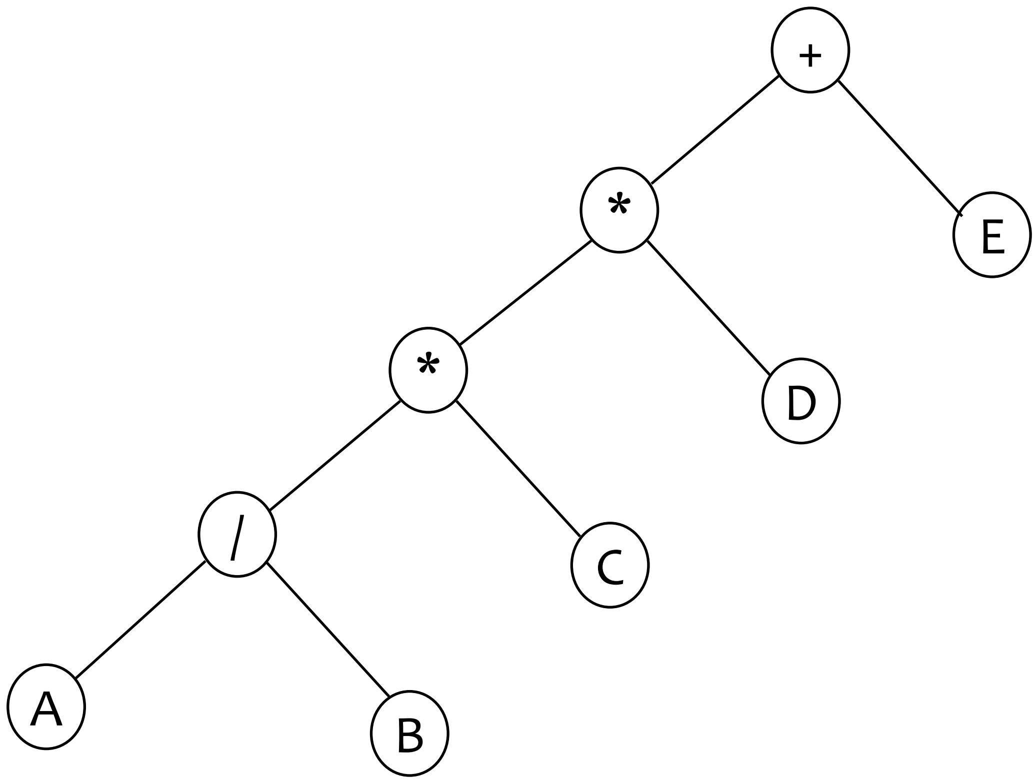
- Each node is visited only once
- When a node is visited, some operation is performed on it
- After traversal, nodes are represented as a linear order

(Depth-first) Binary Tree Traversal

- On a node
  - L : moving to left child node
  - V : visit current node
  - R : moving to right child node
- Available traversal order
  - LVR, LRV, VLR, VRL, RVL, RLV
- Traverse left before right
  - LVR : inorder
  - VLR : preorder
  - LRV : postorder

Binary Tree Traversal

- Binary tree with arithmetic expression

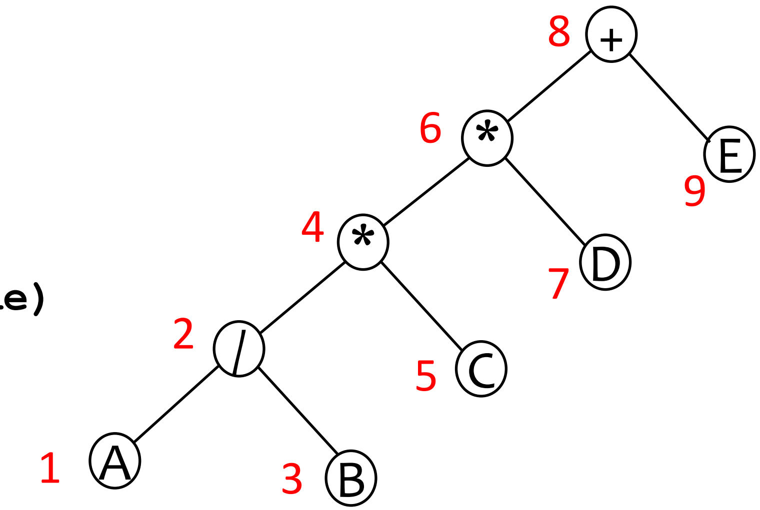


Inorder Traversal

- LVR

```
void Tree::inorder()
{
    inorder(root);
}

void Tree::inorder(TreeNode *CurrentNode)
{
    if (CurrentNode) {
        inorder(CurrentNode->LeftChild);
        cout << CurrentNode->data;
        inorder(CurrentNode->RightChild);
    }
}
```



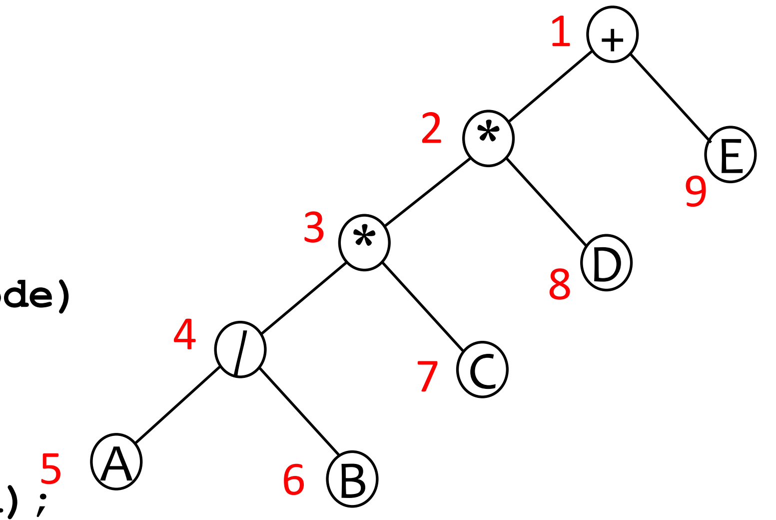
Output : A/B*C*D+E

Preorder Traversal

- VLR

```
void Tree::preorder()
{
   preorder(root);
}

void Tree::preorder(TreeNode *CurrentNode)
{
   if (CurrentNode) {
        cout << CurrentNode->data;
        preorder(CurrentNode->LeftChild);
        preorder(CurrentNode->RightChild);
   }
}
```



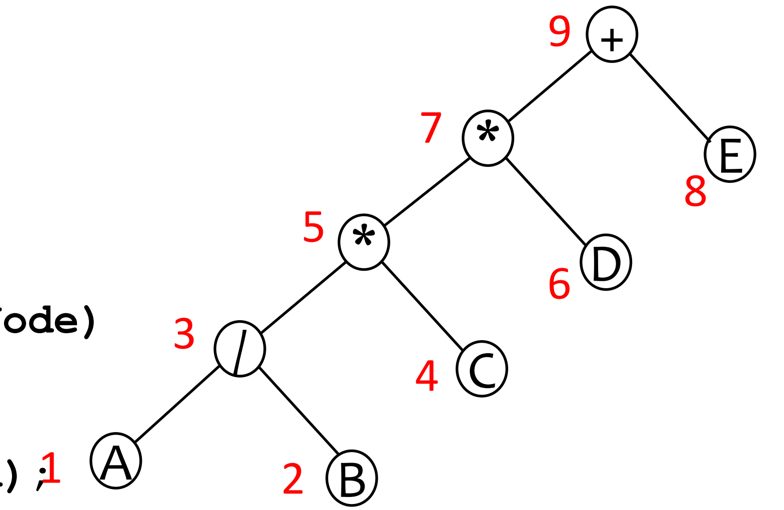
Output : +**/ABCDE

Postorder Traversal

- LRV

```
void Tree::postorder()
{
   postorder(root);
}

void Tree::postorder(TreeNode *CurrentNode)
{
    if (CurrentNode){
        postorder(CurrentNode->LeftChild);
        postorder(CurrentNode->RightChild);
        cout << CurrentNode->data;
     }
}
```



Output : AB/C*D*E+

Nonrecursive Inorder Traversal

- Use a stack

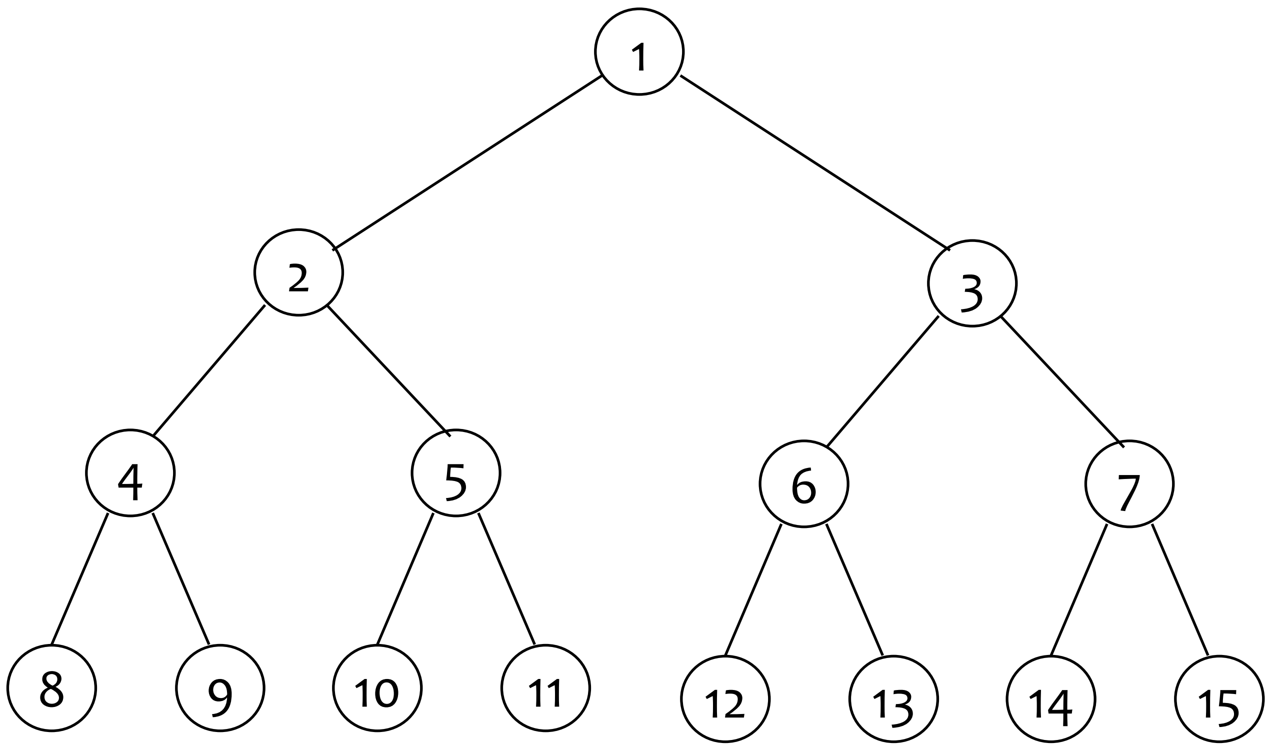
```
void Tree::NonrecInorder()
{
    Stack<TreeNode *> s;
    TreeNode *CurrentNode = root;
    while(1){
        // move down left child
        while(CurrentNode) {
            s.Push(CurrentNode);
            CurrentNode = CurrentNode->LeftChild;
        }

        if (s.IsEmpty()) return;

        CurrentNode = s.Top();
        s.Pop();
        Visit(CurrentNode);
        CurrentNode = CurrentNode->RightChild;
    }
}
```

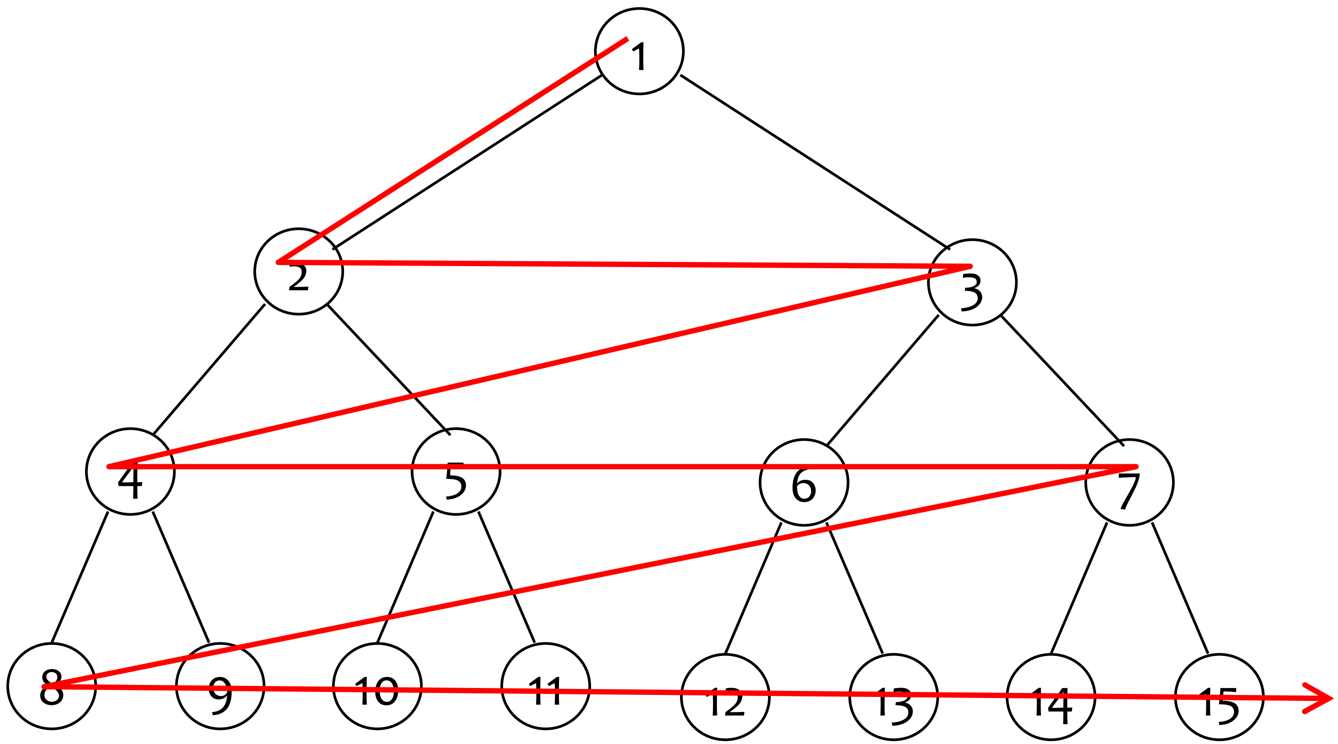
Level-order Traversal

- Traverse index order



Level-order Traversal

- Traverse index order
  - 1,2,3,4,5, ...



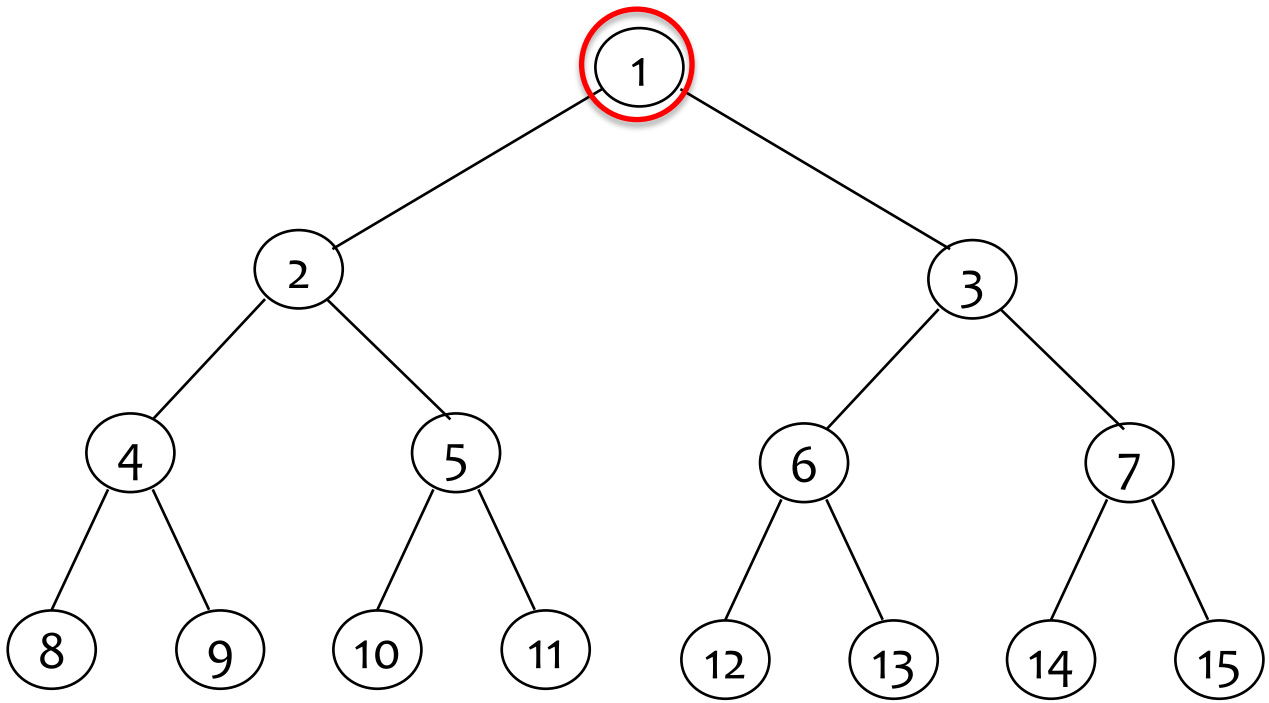
Level-order Traversal

- Using a queue
  - Visit current node, push two children (left, right)

```
void Tree::LevelOrder()
{
   QUEUE<TreeNode*> q;
   TreeNode *CurrentNode = root;
   while(CurrentNode) {
       Visit(currentNode);
       if (CurrentNode->LeftChild)
           q.push(CurrentNode->LeftChild);
       if (CurrentNode->RightChild)
           q.push(CurrentNode->RightChild);
       if(q.IsEmpty()) return;
       CurrentNode = q.Front();
       q.Pop();
   }
}
```

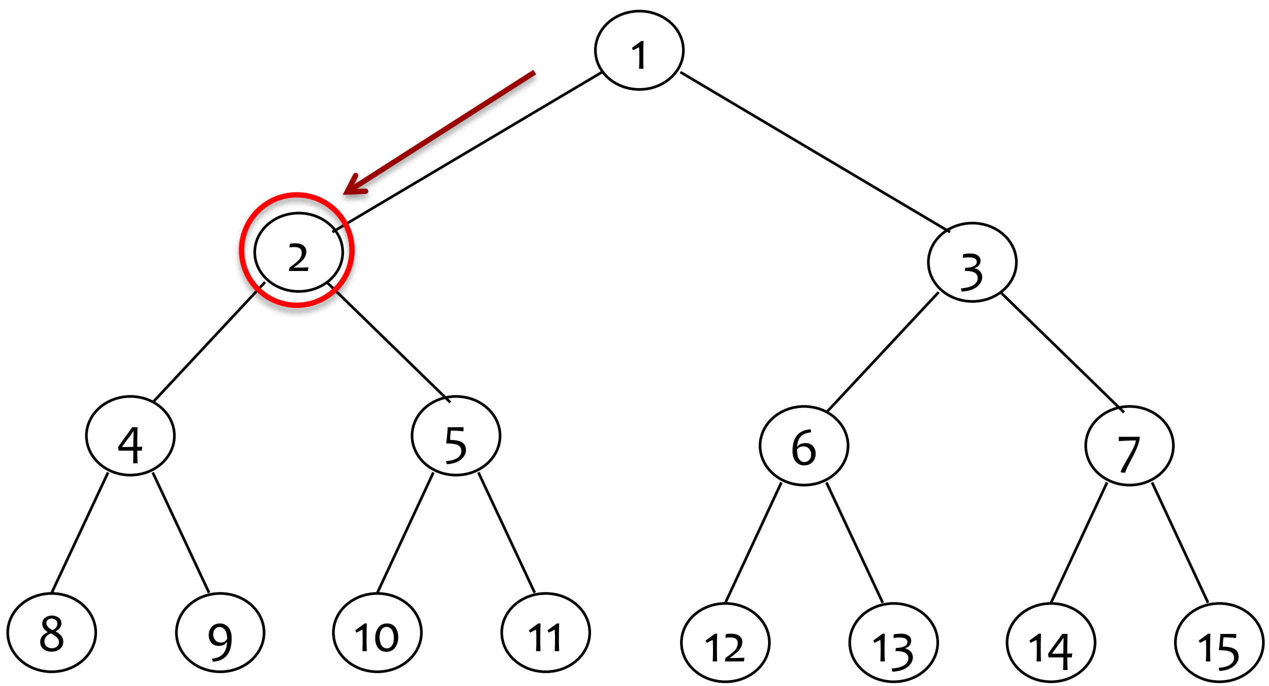
Level-order Traversal

- Visit: 1
- Queue: 2, 3



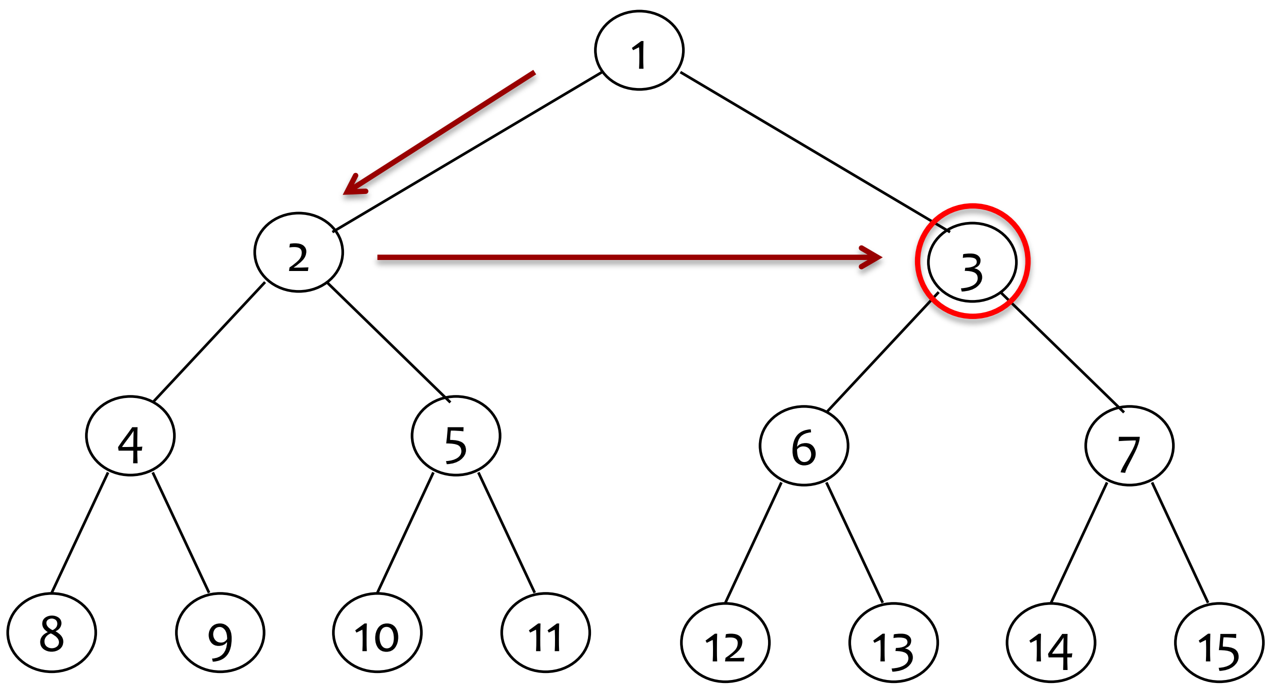
Level-order Traversal

- Visit: 1, 2 (pop)
- Queue: 3, 4, 5



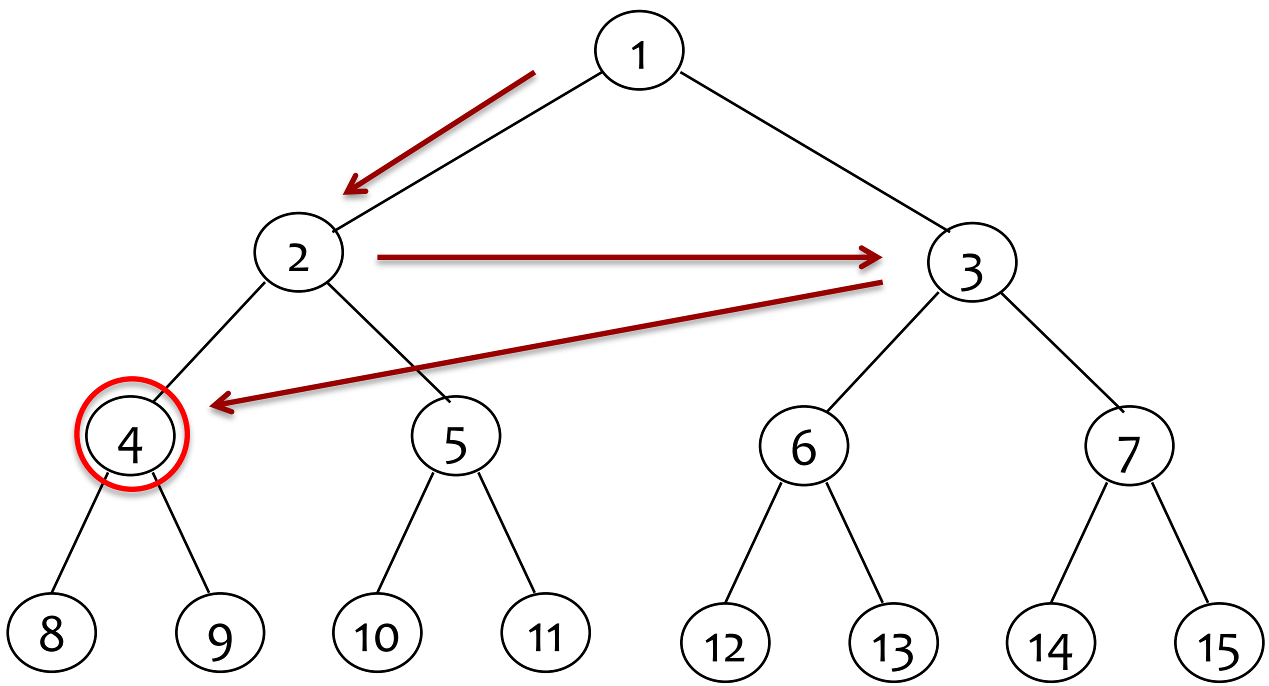
Level-order Traversal

- Visit: 1, 2, 3 (pop)
- Queue: 4, 5, 6, 7



Level-order Traversal

- Visit: 1, 2, 3, 4 (pop)
- Queue: 5, 6, 7, 8, 9



Application of Binary Tree Traversal

- Copying binary trees

```
Tree::Tree(const Tree &s) // Driver
{
   root = copy(s.root);
}

TreeNode *Tree::copy(TreeNode *orignode) // Workhorse
// Return a pointer to an exact copy of the binary tree
// rooted at orignode
{
  if (orignode) {
      TreeNode *temp = new TreeNode;
      temp->data = orignode->data;
      temp->LeftChild = copy(orignode->LeftChild);
      temp->RightChild = copy(orignode->RightChild);
      return temp;
   }
   else return 0;
}
```

preorder traversal

Application of Binary Tree Traversal

- Testing equality

```
// Driver
bool operator==(const Tree& s, Tree& t)
{
   return equal(s.root, t.root);
}

// Workhorse
bool equal(TreeNode *a, TreeNode *b)
{
   if ((!a) && (!b)) return 1;  // both a and b are 0
   if (a && b
        &&  (a->data == b->data)  // data is equal
        &&  equal(a->LeftChild, b->LeftChild)  // left subtrees equal
        &&  equal(a->RightChild, b->RightChild)) // right subtrees equal
      return true;
   return false;
}
```

The Satisfiability Problem

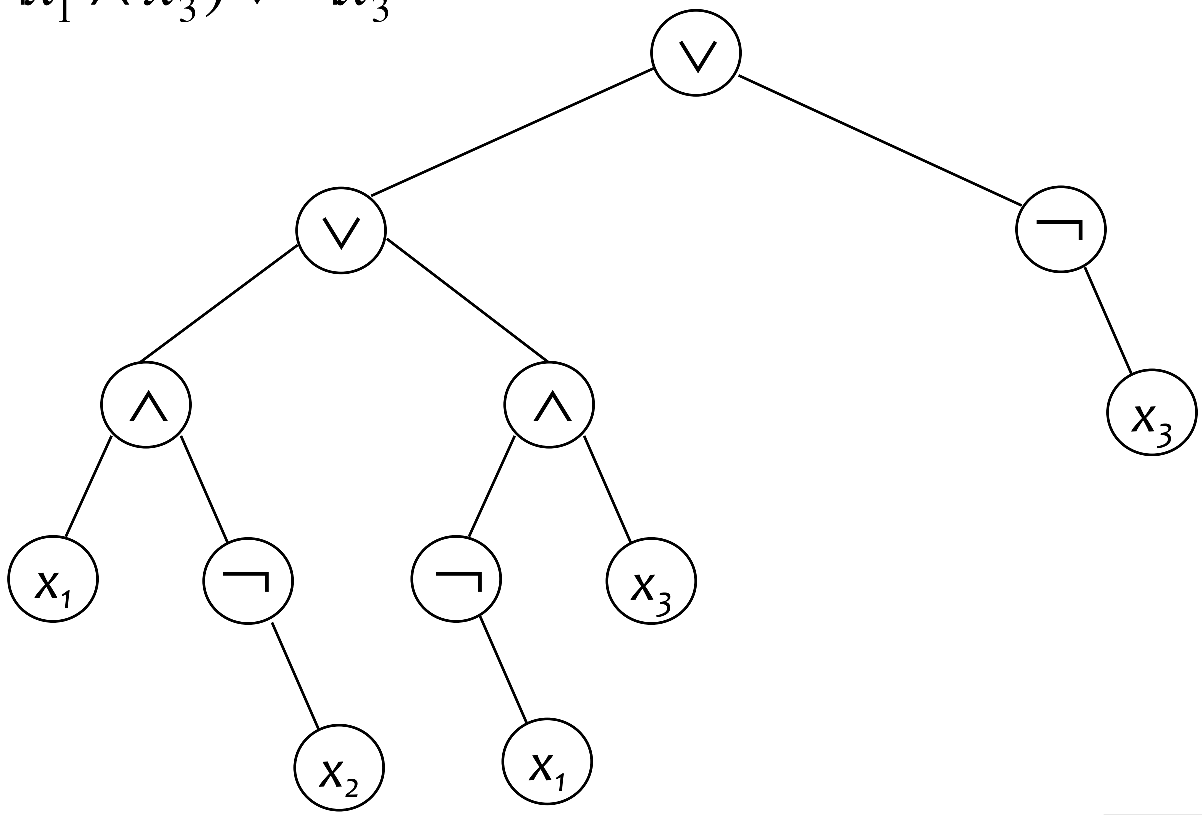
- Propositional calculus
  - Formulas defined by boolean variables $x_1, x_2, \ldots, x_n$ and operators $\wedge$(and), $\vee$(or), $\neg$(not)
- Satisfiability problem
  - If there is an assignment values to the variables that causes the value of the expression is *true*
  - e.g., $x_1 \vee (x_2 \wedge \neg x_3)$ is true if

$$x_1 = \mathit{false}$$
$$x_2 = \mathit{true}$$
$$x_3 = \mathit{false}$$

The Satisfiability Problem

- Propositional formula in a binary tree

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$



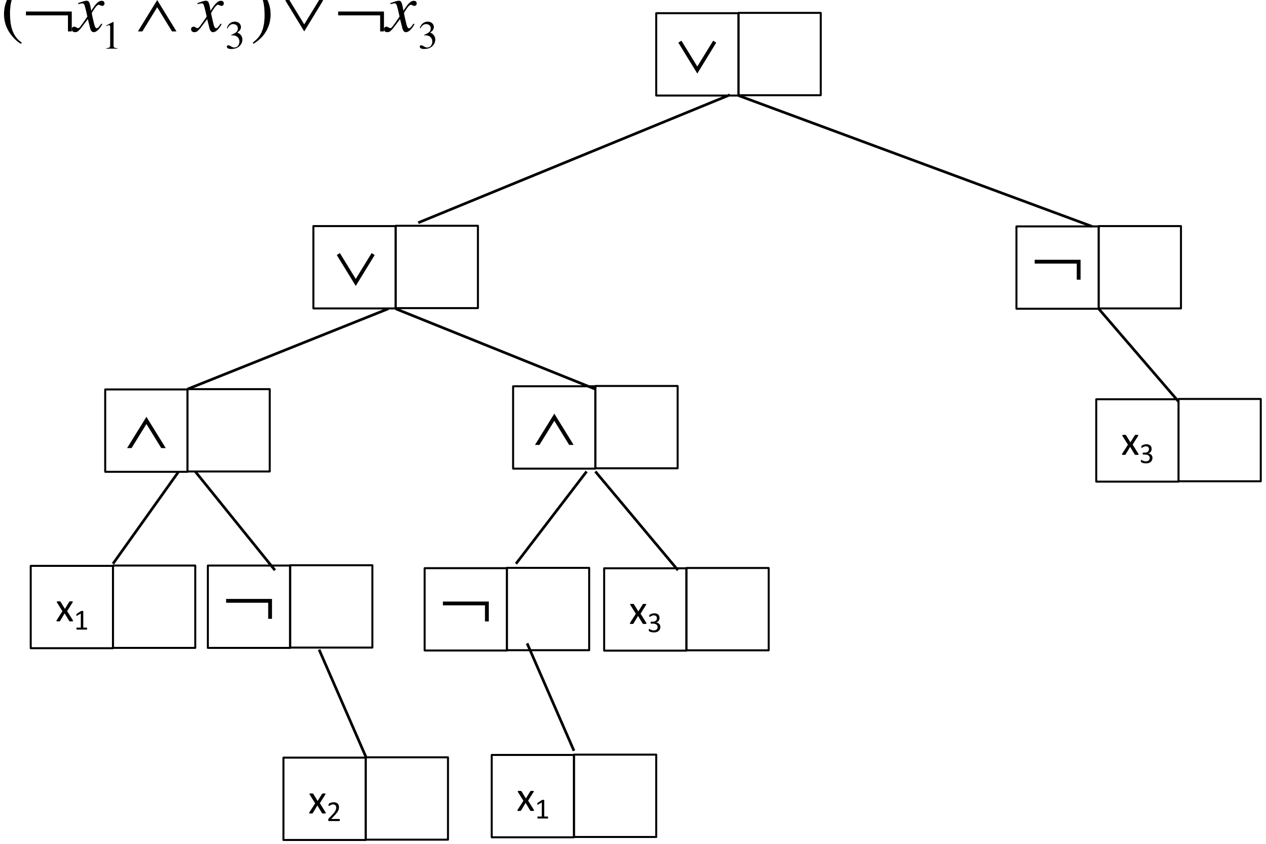
The Satisfiability Problem

- Node structure
  - first : given values for evaluation
    - Operators($\wedge$, $\vee$, $\neg$), True/False values
  - second
    - True/False values after evaluation

Leftchild	first	second	Rightchild

The Satisfiability Problem

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$



The Satisfiability Problem

- Algorithm

```
for each of 2^n possible truth value combinations for the n variables
{
  replace data.first by the values in the current truth combination;
  evaluate the formula by traversing the tree in postorder;
  if (root.data.second)                              why?
  {
    cout << combination;
    return;
  }
}
cout << "no satisfiable combination";
```

The Satisfiability Problem

- Evaluate propositional formula

```
void SatTree::PostOrderEval() {  // Driver
     PostOrderEval(root);
}

void SatTree::PostOrderEval(SatNode *s) // Workhorse
{
   if (s) {
      PostOrderEval(s->LeftChild);
      PostOrderEval(s->RightChild);  // postorder - left & right subtrees are evaluated
      switch(s->data.first) {
         case Not: s->data.second = !s->RightChild->data.second; break;
         case And: s->data.second =
              s->RightChild->data.second && s->LeftChild->data.second ;
              break;
         case Or:
              s->data.second = s->RightChild->data.second ||
                               s->LeftChild->data.second ;
              break;
         case True: s->data.second = TRUE; break;
         case False: s->data.second = FALSE;
      }
   }
}
```

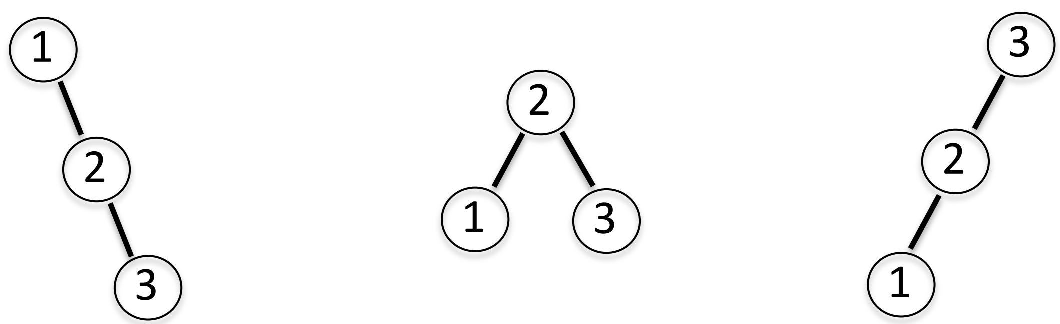
Outline

- Binary tree traversal
- Counting binary trees
- Threaded binary trees

Binary Tree for a Traversal Sequence

- If you are given an <u>inorder</u> sequence, can you define a binary tree uniquely?

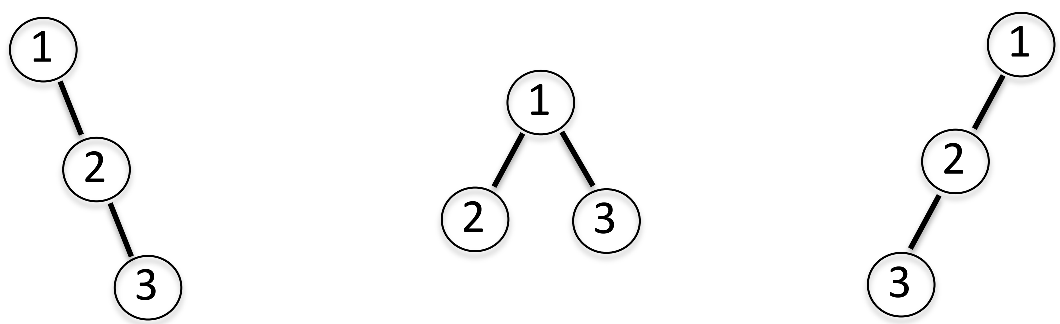
1 2 3



Binary Tree for a Traversal Sequence

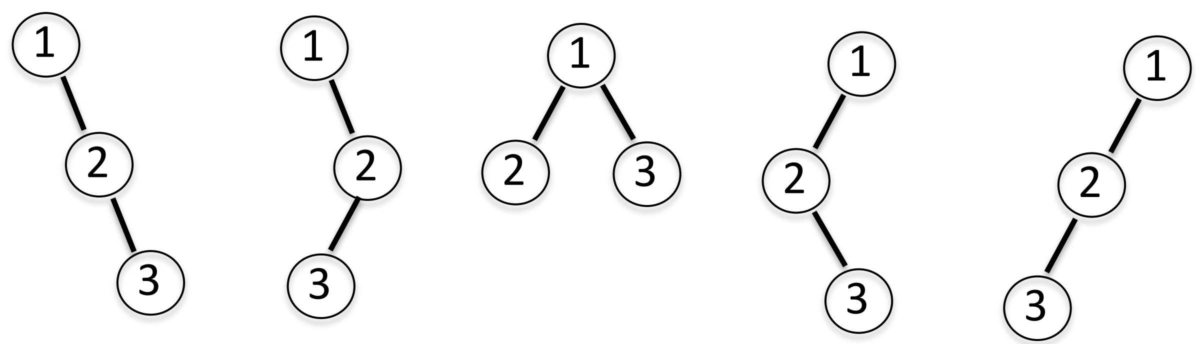
- If you are given a preorder sequence, can you define a binary tree uniquely?

1 2 3



Counting Binary Trees

- Every binary tree has a unique pair of preorder and inorder sequences



Inorder?

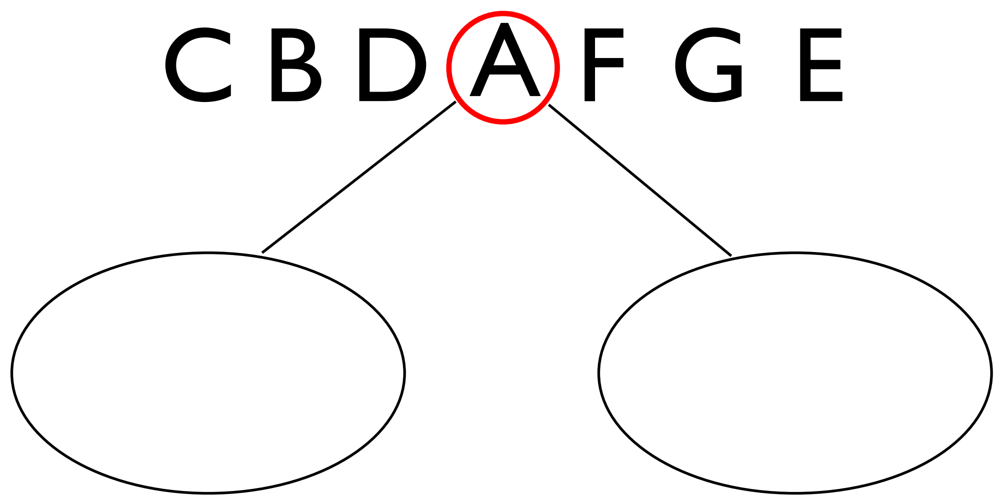
Preorder?

Binary Tree from Traversal

- Can we reconstruct a binary tree satisfying the given traversal orders?
  - Inorder : C B D A F G E
  - Preorder : A B C D E F G

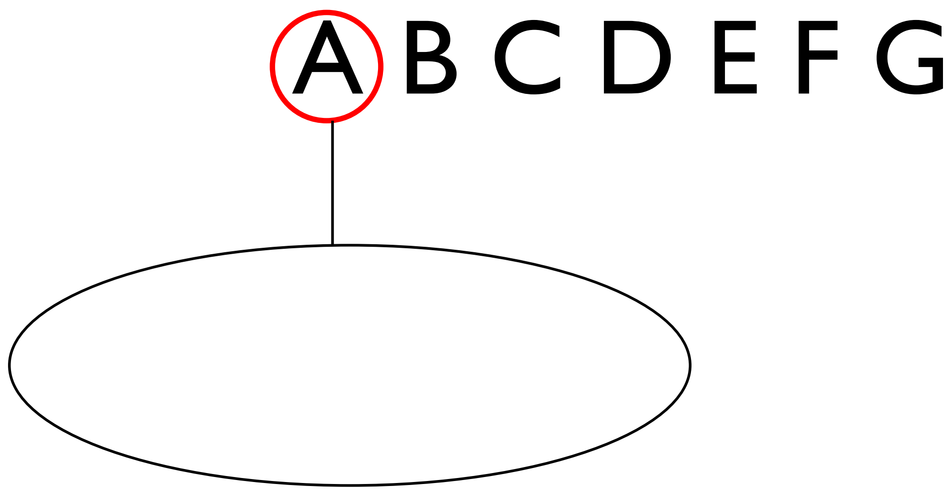
Binary Tree from Traversal

- Inorder
  - If you pick a letter, all the letters on the left/right are in the left/right subtree of that letter (but cannot tell which one is the root)



Binary Tree from Traversal

- Preorder
  - The front-most letter is the root node



Binary Tree from Traversal

- Can we reconstruct a binary tree satisfying the given traversal orders?
  - Inorder : C B D A F G E
  - Preorder : A B C D E F G

Define a Unique Binary Tree

- Inorder – postorder
- Inorder – preorder
- Inorder – level order

- Other combination of traversal sequences cannot define a unique binary tree

Outline

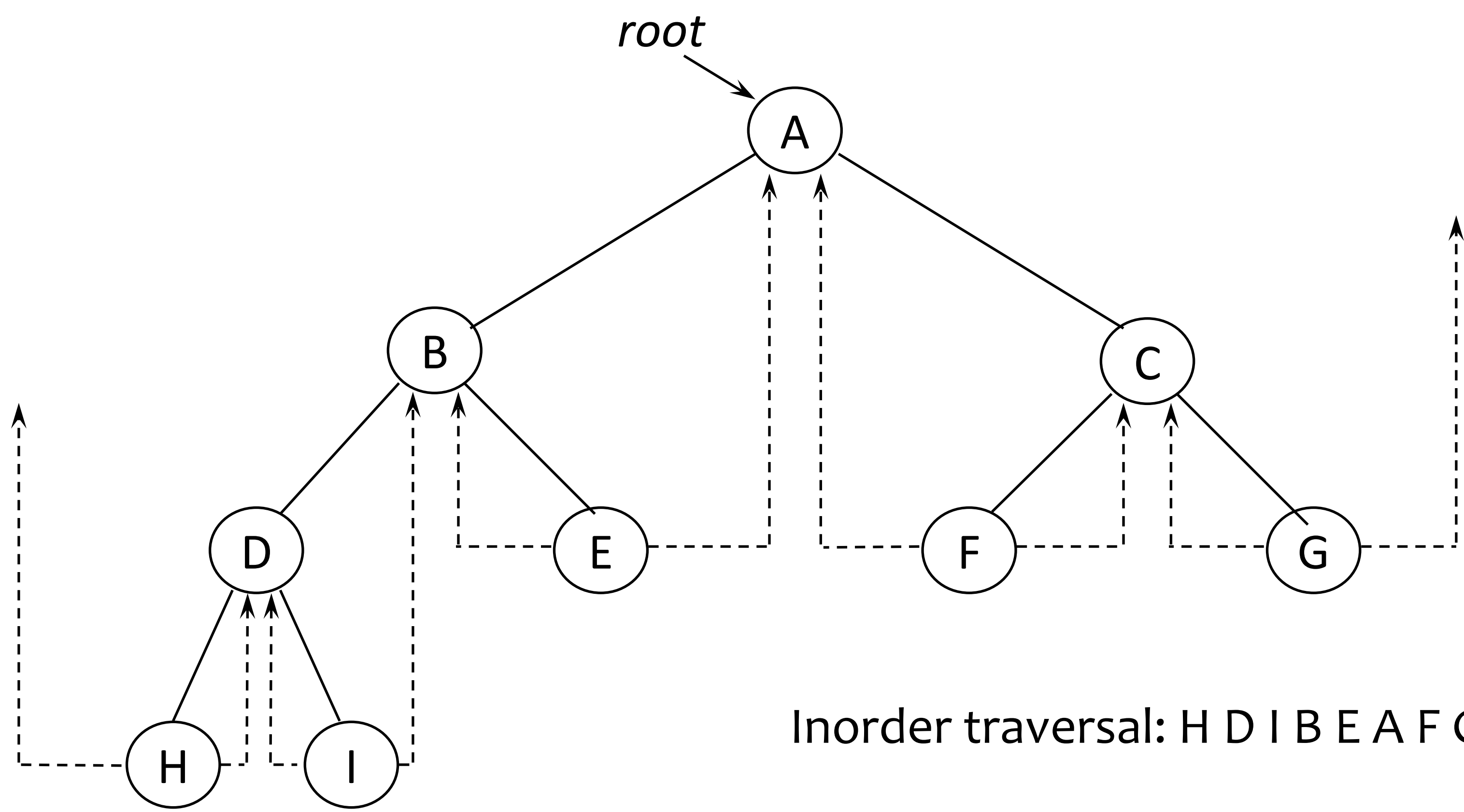
- Binary tree traversal
- Counting binary trees
- Threaded binary trees

Threaded Binary Trees

- Binary tree with n nodes
  - Total # of links : 2n
  - Total # of null links : n+1 -> ~50% are wasted!
- Idea
  - Use null link to represent traversal order
  - Null rightChild : next node for inorder traversal
  - Null leftChild : previous node for inorder traversal
  - Why threaded?

Threaded Binary Trees (inorder)

- Dotted line : thread

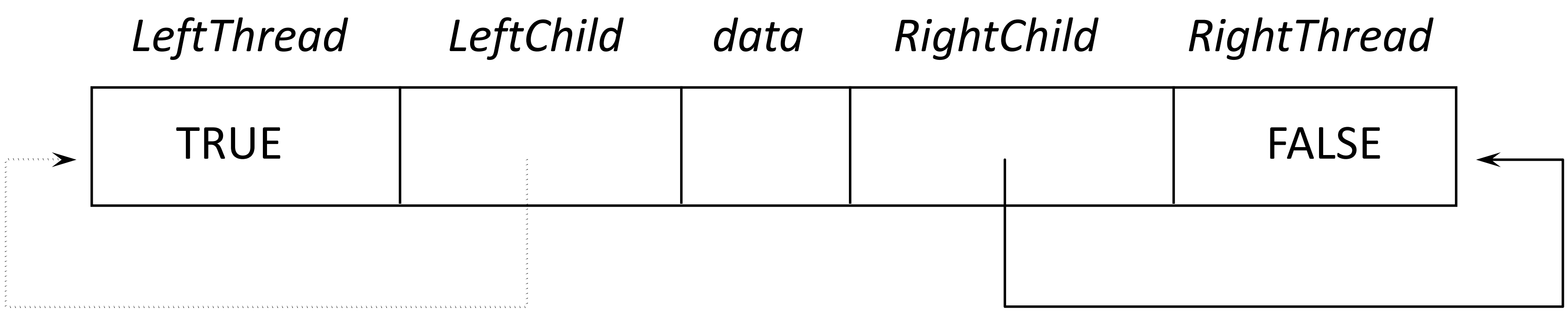


Inorder traversal: H D I B E A F C G

Threaded Binary Trees (inorder)

- Node representation
  - Need to distinguish child / thread pointer
    - One bit Boolean flag for <u>each pointer</u>

- Dangling pointer
  - Use head node



Example of empty threaded binary tree

Threaded Binary Trees (inorder)

```
class ThreadedNode{
friend class ThreadedTree;
friend class ThreadedInorderIterator;
private:
    Boolean LeftThread;
    ThreadedNode *LeftChild;
    char data;
    ThreadedNode *RightChild;
    Boolean RightThread;
};
```

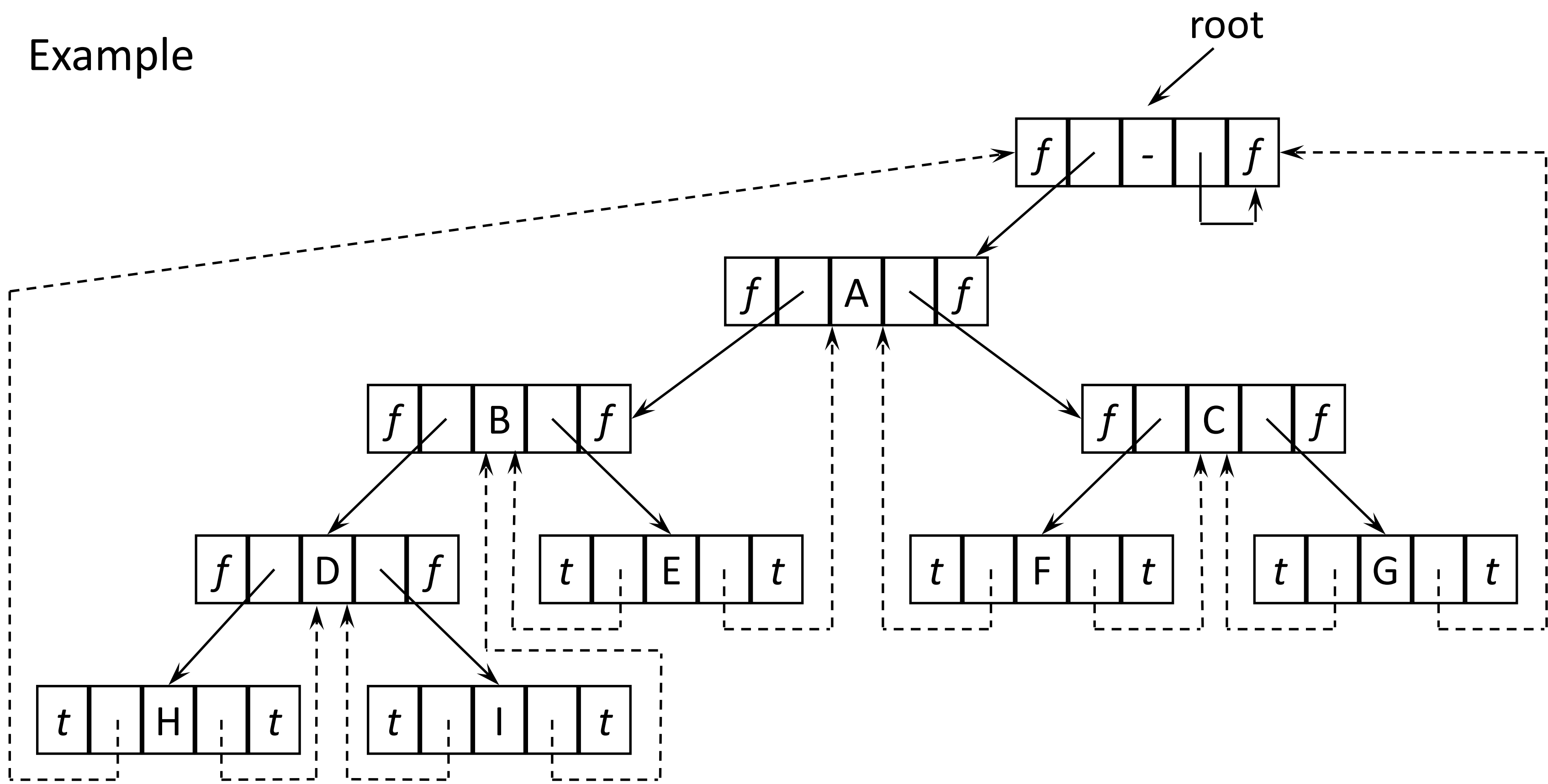
```
class ThreadedTree {
friend class ThreadedInorderIterator;
public:
// Tree operators

private:
    ThreadedNode *root;
};
```

```
class ThreadedInorderIterator {
public:
    char *Next();
    ThreadedInorderIterator(ThreadedTree tree):t(tree)
     CurrentNode = t.root; ;
private:
    ThreadedTree t;
    ThreadedNode *CurrentNode;
};
```

Threaded Binary Trees (inorder)

Example



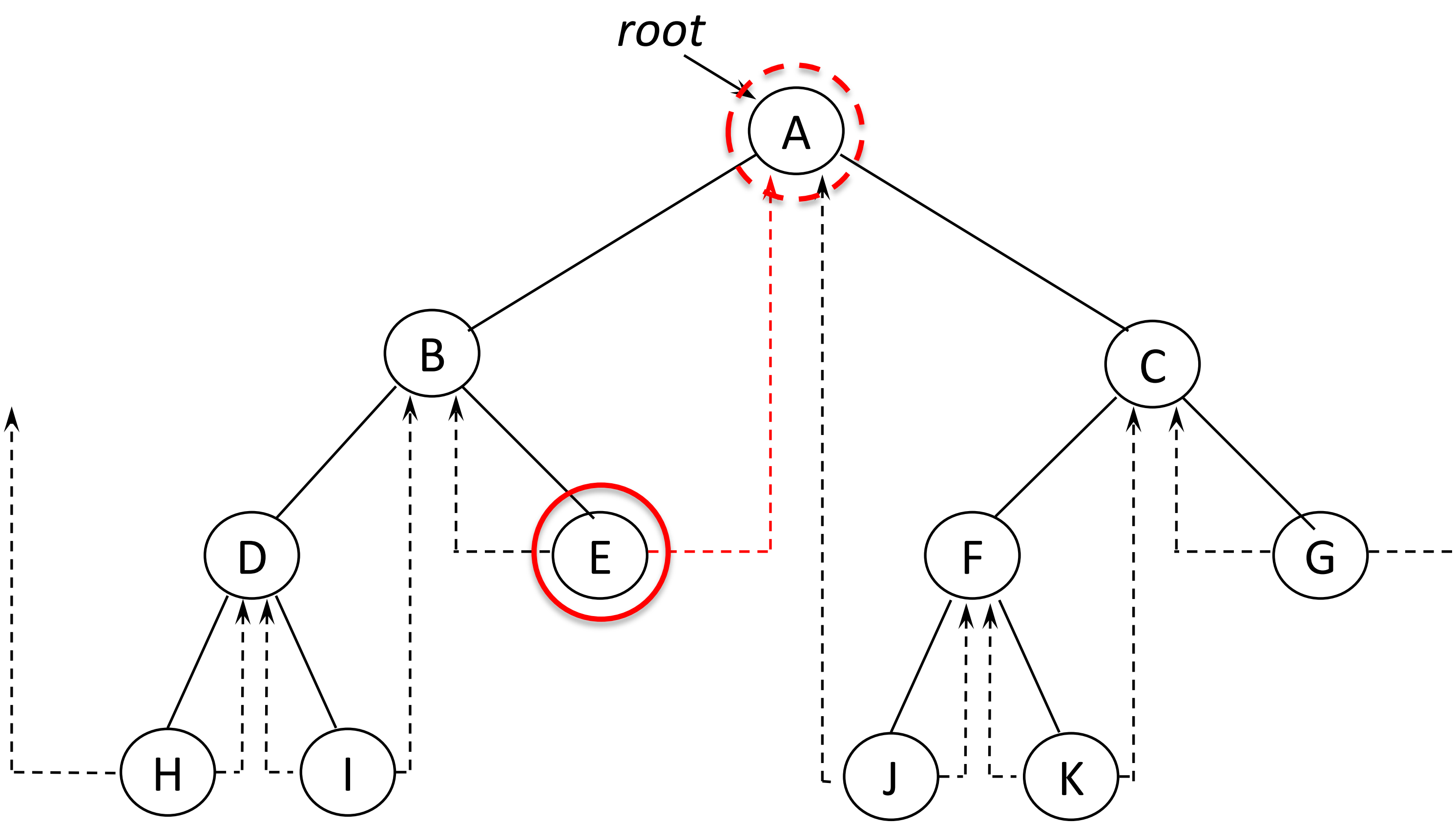
f = FALSE;   t = TRUE

Traverse Threaded Binary Tree

- Traverse inorder without using a stack
- Next()
  - If RightThread is True : RightChild
  - if RightThread is False
    - Follow LeftChild of RightChild until you reach the node with LeftThread ==True

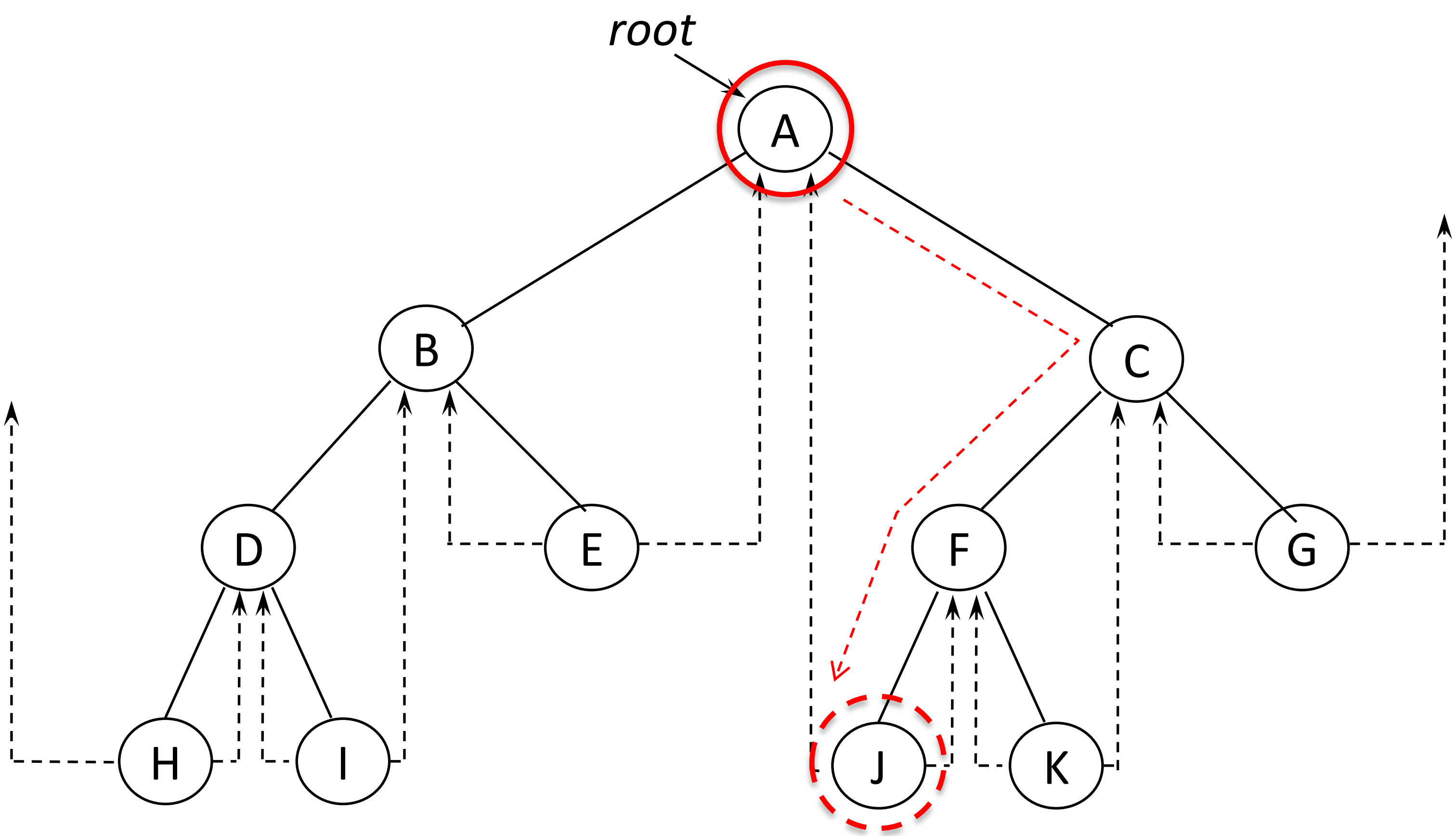
```
char *ThreadedInorderIterator::Next() // return next inorder
{
    ThreadedNode *temp = CurrentNode->RightChild;
    if (!CurrentNode->RightThread)
        while (!temp->LeftThread) temp = temp->LeftChild;
    CurrentNode = temp;
    if (CurrentNode==t.root) return 0;
    else return &CurrentNode->data;
}
```

Current node : E
RightThread is True, so the next node for inorder traversal is E->RightChild = A



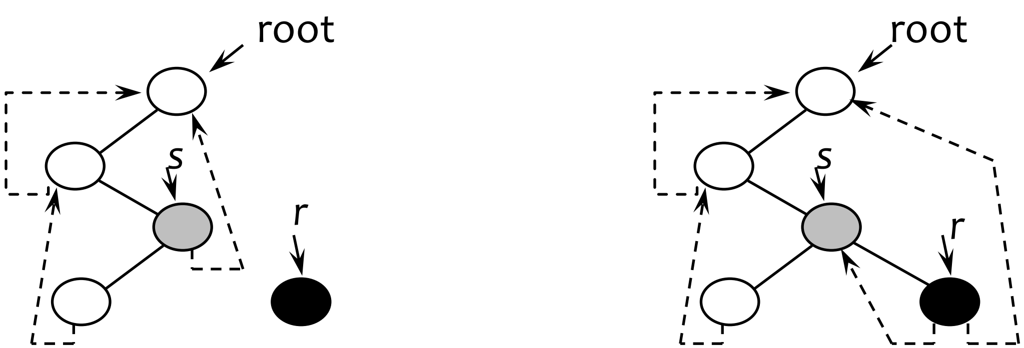
Current node : A

RightThread is False, so you follow the LeftChild link of RightChild node (C) until you reach the node with the LeftThread is True (which is J)



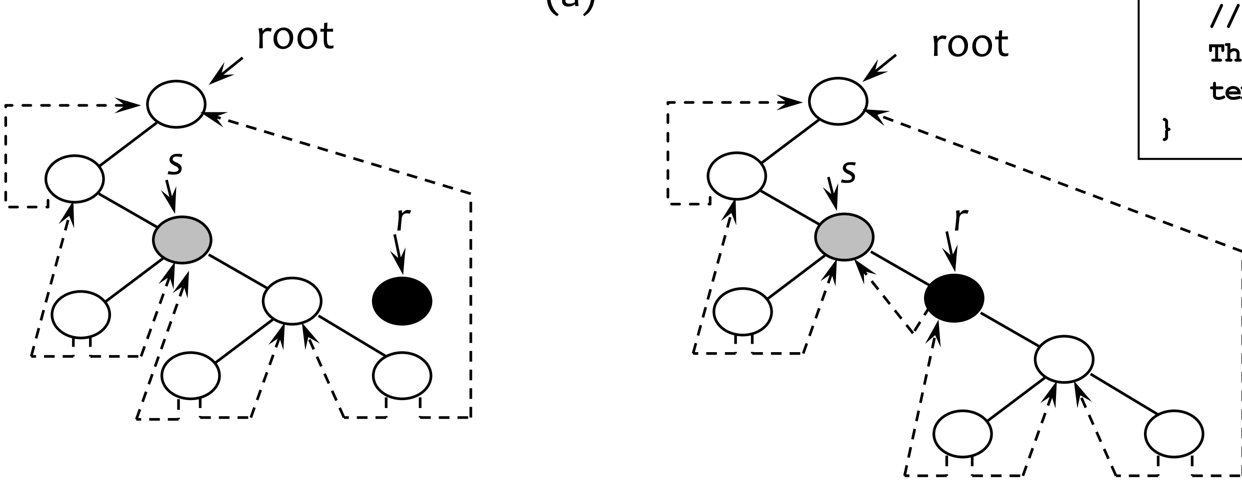
Insertion in a Threaded Binary Tree

- Insert r as RightChild of s



(a)

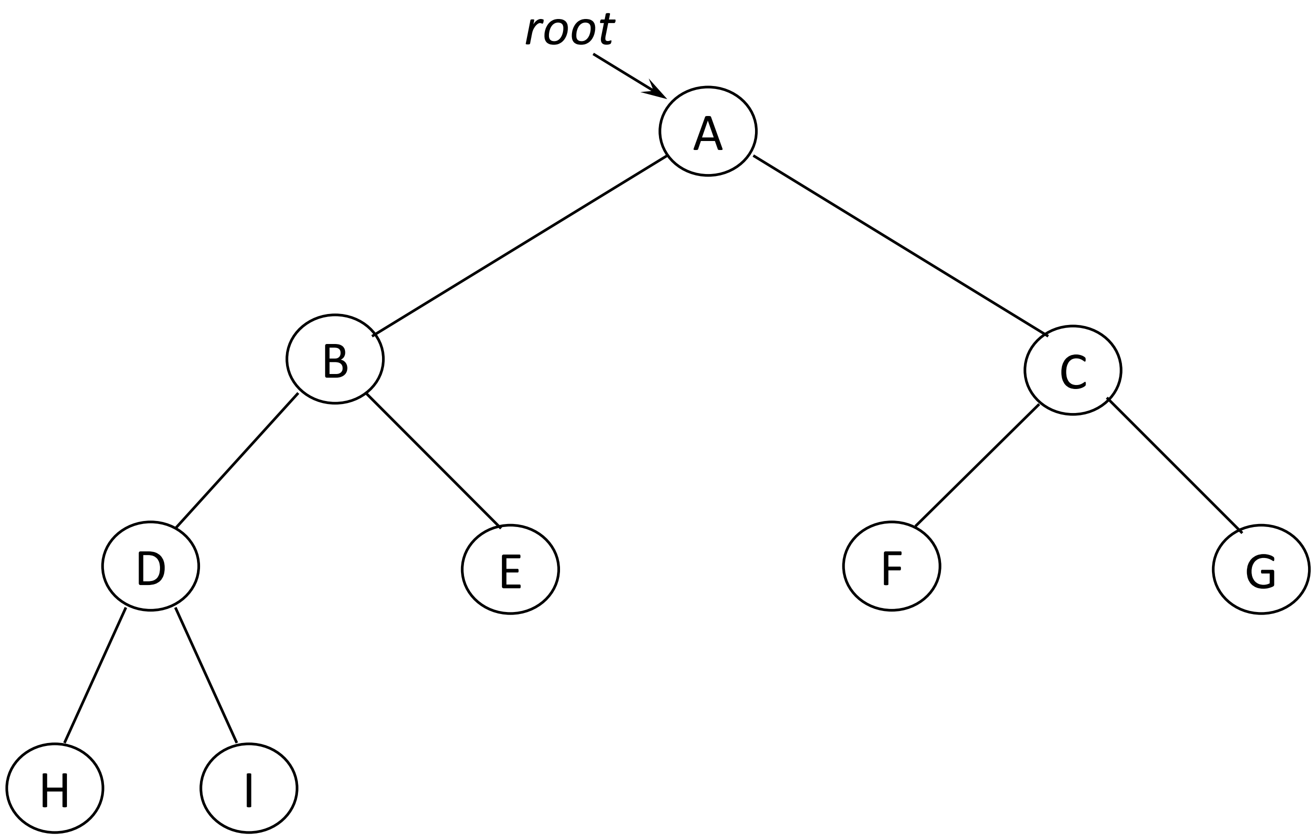
```
 ->RightChild  =
 ->RightThread =
 ->LeftChild  =
 ->LeftThread =
 ->RightChild =
 ->RightThread =
if (!r->RightThread) {
   // return inorder successor of r
   ThreadedNode *temp = InorderSucc(r);
   temp->LeftChild =
}
```



before (b) *after*

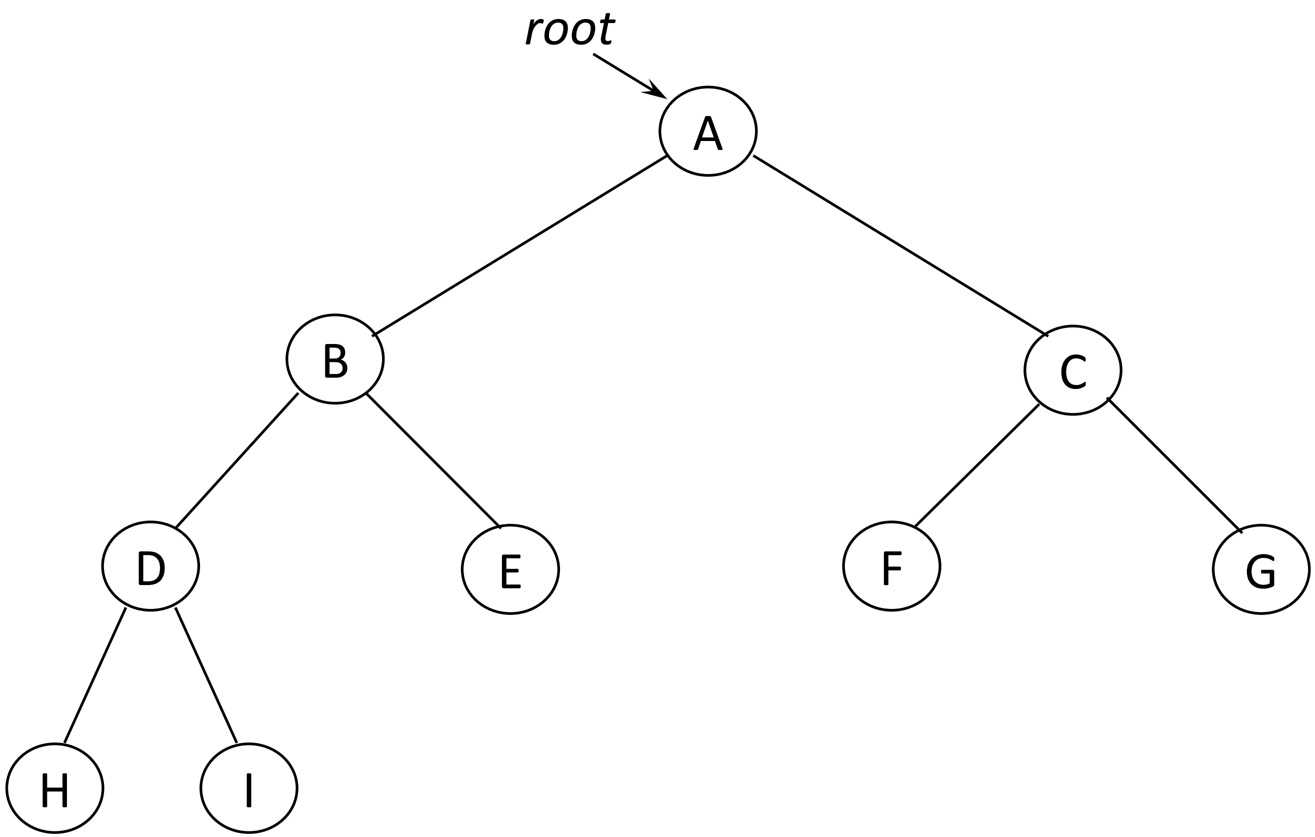
Preorder Threaded Binary Tree

- ?



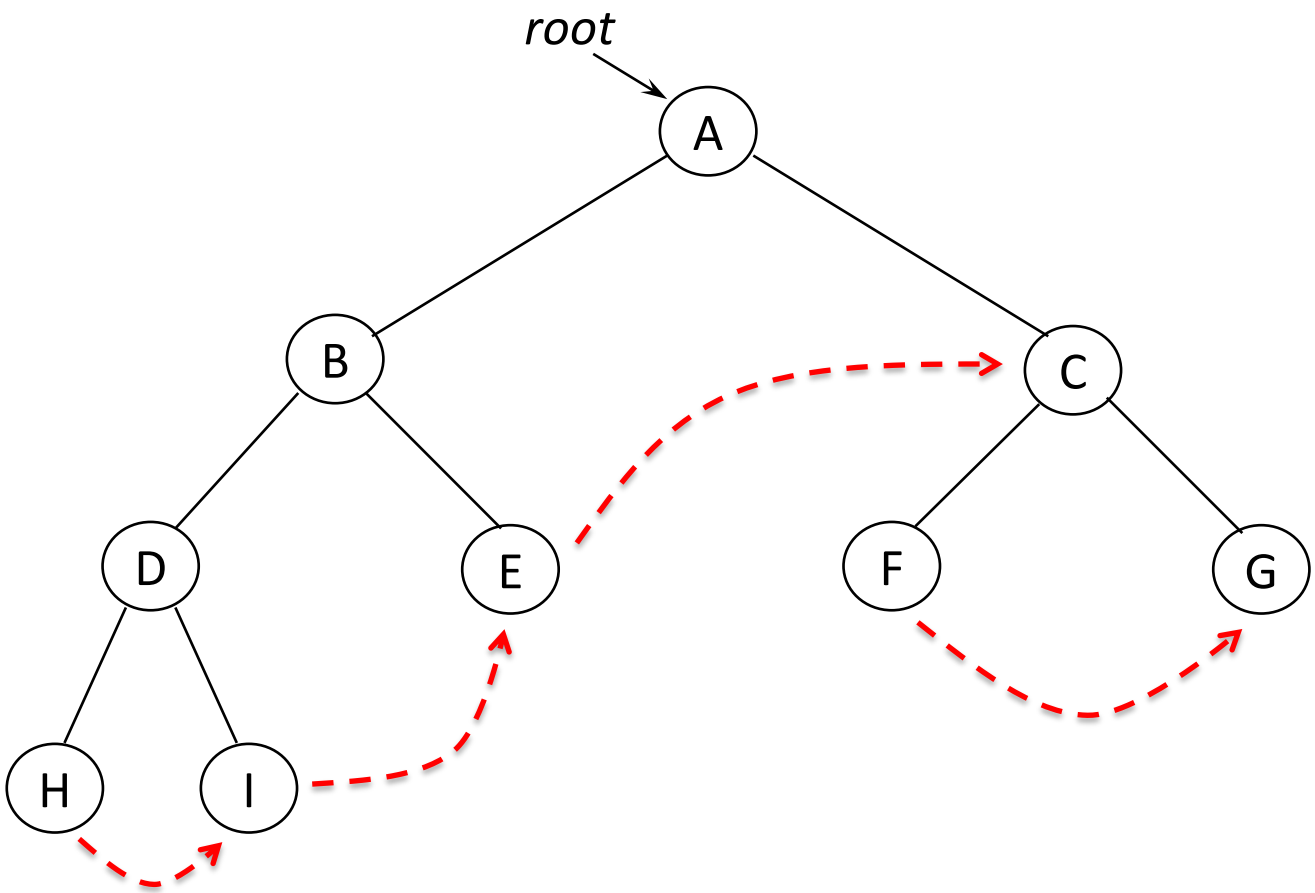
Preorder Threaded Binary Tree

- ABDHIECFG



Preorder Threaded Binary Tree

- ABDHIECFG



Questions?