CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today:

- Deletion in Binary Search Trees
- Tree Traversals (recursive and non-recursive)

Next Time:

- Efficiency of binary trees;
- Balanced Trees
- 2-3 Trees

Computer Science

Deletion in BSTs

Computer Science

Deletion is somewhat more complicated than insertion or lookup. We will warm up by considering a simple case first: How do we delete the minimal element in a BST?

```
// reconstruct the tree r without its minimal element

public static Node deleteMin(Node r) {
  if(r.left == null)
    return r.right;
  else {
    r.left = deleteMin(r.left);
    return r;
  }
}
```

2

Deletion in BSTs

Computer Science

Note for later: If we want to keep track of the minimal node (say, to remove it and use it later), we could write a simple helper function to look up the minimal element:

```
public static Node findMin(Node r) {
  if(r.left == null)                    // this is the minimal node
    return r;
  else
    return findMin(r.left);
}

public static Node deleteMin(Node r) {
  if(r.left == null)
    return r.right;
  else {
    r.left = deleteMin(r.left);
    return r;
  }
}


// Remove the node containing the minimal element and store it as p
 Node p = findMin(root);
 root = deleteMin(root);
```

3

Deletion in BSTs

Computer Science

Ok, we know how to delete the minimal element; how to delete an arbitrary element?
As usual, the place to start with by enumerating all the cases, starting with null:

```
public static Node delete(int n, Node t) {
   if (t == null)                             // Case 1: tree is null
     return t;
```

4

Deletion in BSTs

Computer Science

```
public static Node delete(int n, Node t) {
   if (t == null)                       // Case 1: tree is null
     return t;

   else if (n < t.key) {                // Case 2: key n is in left subtree
     t.left = delete(n, t.left);
     return t;

   } else if (n > t.key) {              // Case 3: key n is in right subtree
     t.right = delete(n, t.right);
     return t;
   }
```

5

Deletion in BSTs

Computer Science

```
public static Node delete(int n, Node t) {
   if (t == null)                       // Case 1: tree is null
     return t;
   else if (n < t.key) {                // Case 2: key n is in left subtree
     t.left = delete(n, t.left);
     return t;
   } else if (n > t.key) {              // Case 3: key n is in right subtree
     t.right = delete(n, t.right);
     return t;
   } else                        // Case 4: found key n at root;
```

6

Deletion in BSTs

Computer Science

```
public static Node delete(int n, Node t) {
   if (t == null)                          // Case 1: tree is null
     return t;
   else if (n < t.key) {                   // Case 2: key n is in left subtree
     t.left = delete(n, t.left);
     return t;
   } else if (n > t.key) {                 // Case 3: key n is in right subtree
     t.right = delete(n, t.right);
     return t;
   }
                                     // Case 4: found key n at root;
   else if (t.left == null)          // Case 4a: no left child, so reroute around
     return t.right;
   else if (t.right == null)         // Case 4b: no right child, ditto
     return t.left;
   else {
```

7

Deletion in BSTs

Computer Science

```
public static Node delete(int n, Node t) {
   if (t == null)                          // Case 1: tree is null
     return t;
   else if (n < t.key) {                   // Case 2: key n is in left subtree
     t.left = delete(n, t.left);
     return t;
   } else if (n > t.key) {                 // Case 3: key n is in right subtree
     t.right = delete(n, t.right);
     return t;
   } else                           // Case 4: found key n at root;
          if (t.left == null)       // Case 4a: no left child, so reroute around
     return t.right;
   else if (t.right == null)        // Case 4b: no right child, ditto
     return t.left;
   else {                           // Case 4c: both children exist, so replace
                                    //    root by minimal element in right subtree
     Node min = findMin(t.right);   //    Find minimal node
     t.right = deleteMin(t.right);  //    Reconstruct the right subtree without it
     min.left = t.left;             //    Finally, replace root node with min node
     min.right = t.right;
     return min;
   }
 }
```

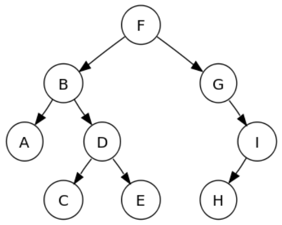
8

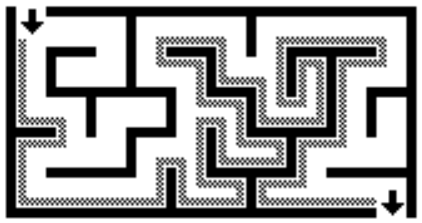
Tree Traversals

Computer Science

A Tree Traversal is an algorithm for visiting each node of a Binary Tree in some order; all algorithms which perform some kind of operation on the tree as a whole usually follow one of these traversals. We will later generalize these to traversals of arbitrary Graphs, and many problems in computer science can be phrased as traversal of some kind of graph.

Let us begin by considering how you might “explore” a tree by walking around the links (for the moment considering them as two-way) to “visit” each node at least once; one way to do this is to use a version of the famous “**right-hand rule**” for solving a simple maze: just start at the root, keep your right hand on the wall, and keep walking.



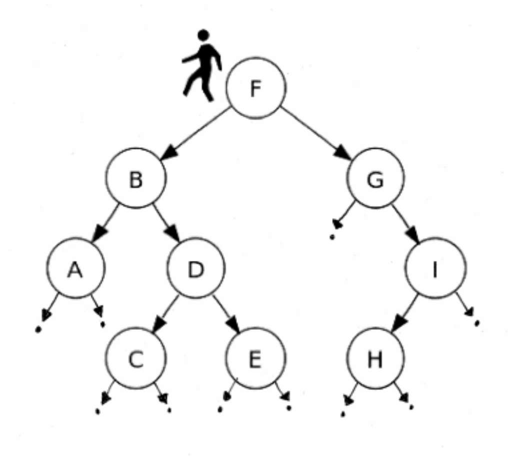


9

Tree Traversals

Computer Science

Instead, we will keep our left hand on the outside “wall” of the tree, and keep walking…..

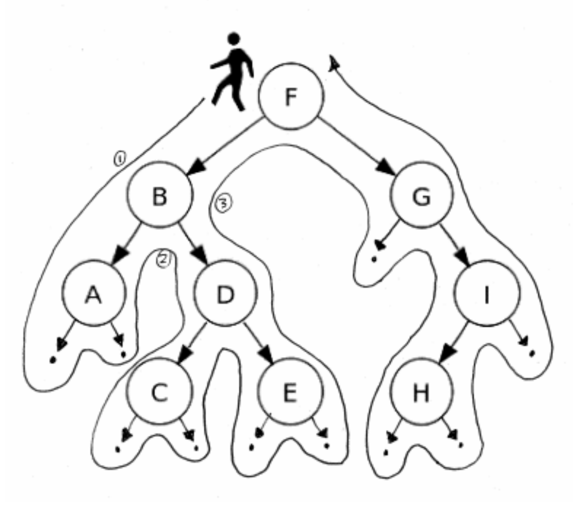


10

Tree Traversals

Computer Science

Instead, we will keep our left-hand on the outside "wall" of the tree, and keep walking…..



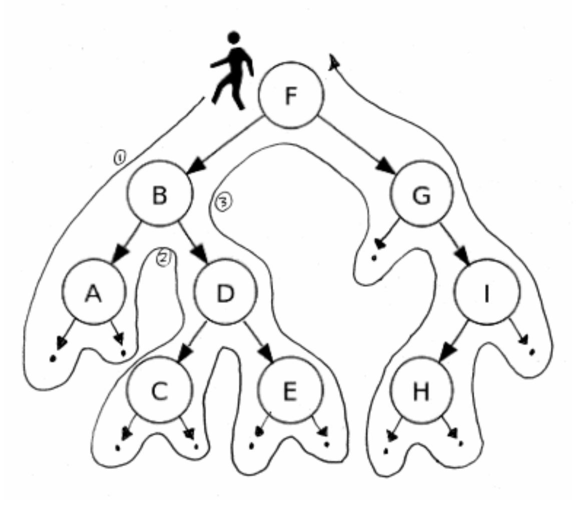
11

Tree Traversals

Computer Science

NOTE: Each node is touched exactly three times:

The nodes are touched (for the **first** time) in this order: F, B, A, D, C, E, G, I, H



12

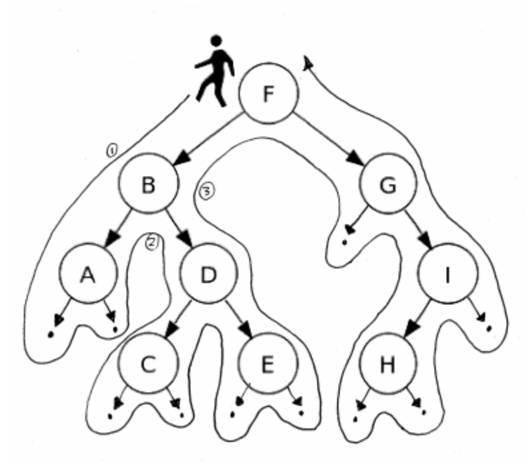
Tree Traversals

Computer Science

NOTE: Each node is touched exactly three times:

The nodes are touched (for the first time) in this order: F, B, A, D, C, E, G, I, H

The nodes are touched (for the **second** time) in this order: A, B, C, D, E, F, G, H, I



13

Tree Traversals

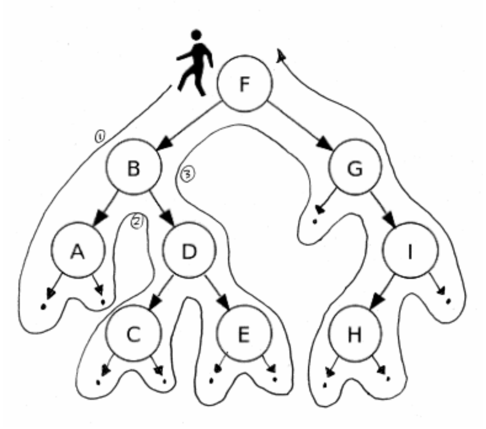
Computer Science

NOTE: Each node is touched exactly three times:

The nodes are touched (for the first time) in this order: F, B, A, D, C, E, G, I, H

The nodes are touched (for the second time) in this order: A, B, C, D, E, F, G, H, I

The nodes are touched (for the **third** time) in this order: A, C, E, D, B, H, I, G, F

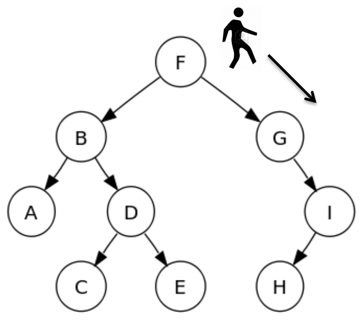


14

Tree Traversals

Computer Science

Or, we could walk around "clock-wise" instead, by using our right hand:



The nodes are visited for the first time in this order: F, G, I, H, B, D, E, C, A

The nodes are visited for the second time in this order: I, H, G, F, E, D, C, B, A

The nodes are visited for the third time in this order: H, I, G, E, C, D, A, B, F

15

Tree Traversals

Computer Science

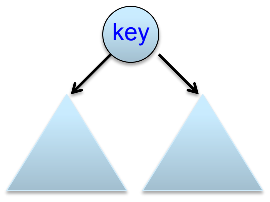
How could we do this using an algorithm (which---no surprise!—will be recursive)?
As usual, we use the **recursive definition of a tree** as a basis for our algorithm:

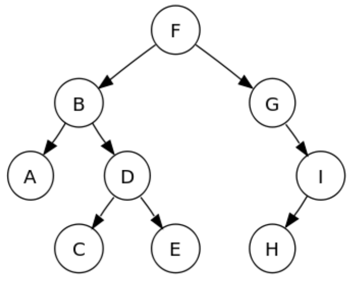
A Binary Tree is either

(1) null •

or:

(2) A node pointing to two Binary Trees





16

Tree Traversals

Computer Science

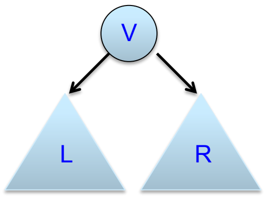
The **base case** is easy (do nothing!);

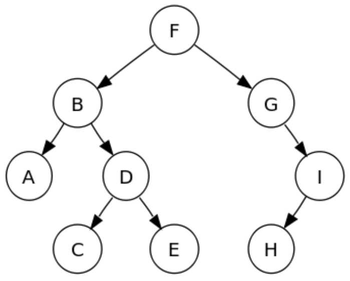
For the **recursive case**, we have to do three things:

- (V) Visit the root (say, by printing out the key);
- (L) Recursively traverse the left subtree; and
- (R) Recursively traverse the right subtree.

It does not matter which order we do these in,

as long as we do all three…..





Tree Traversals

Computer Science

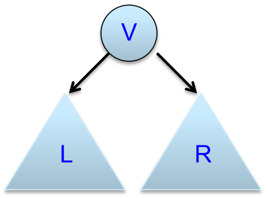
The **base case** is easy (do nothing!);

For the **recursive case**, we have to do three things:

- (V) Visit the root (say, by printing out the key);
- (L) Recursively traverse the left subtree; and
- (R) Recursively traverse the right subtree.

It does not matter which order we do these in,

as long as we do all three…..



There are 3! = 6 possible orderings of these three lines, giving us 6 possible recursive traversals:

Preorder - Visit root first
Inorder - Visit root second
Postorder - Visit root last

Normally, you do L before R;
If you add “Reverse” do R before L

Reverse Postorder:
R
L
V

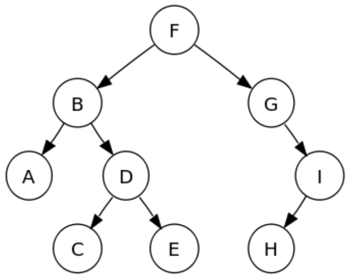
Tree Traversals

Computer Science

How could we do this using an algorithm? Here is a Preorder Traversal:

```
void traverse(Node t) {
    if( t != null ) {        // Base case is implicit
       visit(t);             // V
       traverse(t.left);     // L
       traverse(t.right);    // R
    }
}

void visit(Node t) {
     System.out.print(t.key + “   ”);
}
```



19

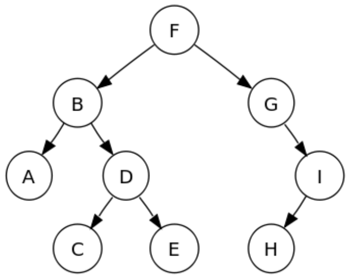
Tree Traversals

Computer Science

Which one is this?

```
void traverse(Node t) {
    if( t != null ) {        // Base case is implicit
       traverse(t.left);     // L
       visit(t);             // V
       traverse(t.right);    // R
    }
}

void visit(Node t) {
     System.out.print(t.key + “   ”);
}
```



20

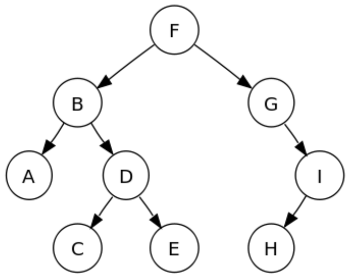
Tree Traversals

Computer Science

Which one is this? Note: L before R, Visit root in middle: Inorder

```
void traverse(Node t) {
    if( t != null ) {        // Base case is implicit
       traverse(t.left);     // L
       visit(t);             // V
       traverse(t.right);    // R
    }
}

void visit(Node t) {
     System.out.print(t.key + “   ”);
}
```



21

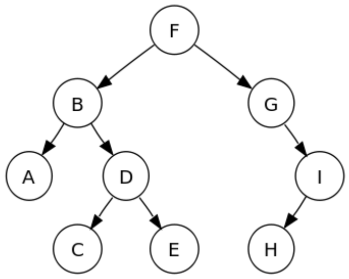
Tree Traversals

Computer Science

How about this?

```
void traverse(Node t) {
    if( t != null ) {        // Base case is implicit
       traverse(t.right);    // R
       traverse(t.left);     // L
       visit(t);             // V
}
}

void visit(Node t) {
     System.out.print(t.key + “   ”);
}
```



22

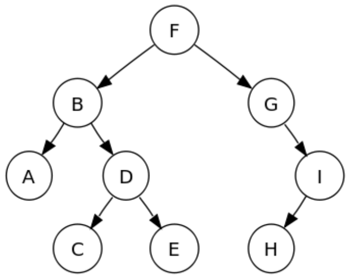
Tree Traversals

Computer Science

How about this? R before S (so Reverse) and Visit root last: Reverse Postorder

```
void traverse(Node t) {
    if( t != null ) {        // Base case is implicit
       traverse(t.right);    // R
       traverse(t.left);     // L
       visit(t);             // V
}
}

void visit(Node t) {
     System.out.print(t.key + “   “);
}
```



23

Tree Traversals: Non-Recursive Traversals

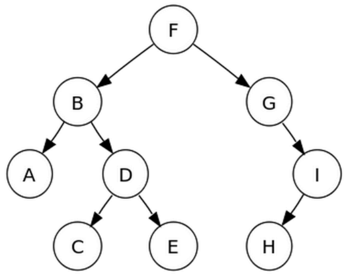
Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

Let’s try using a stack first:

```
void traverse(Node t) {
   Stack<Node> S = new Stack<Node>();
   S.push(t);
   while( !S.isEmpty() ) {
       Node p = S.pop();
       visit( p );
       if( p.left != null )
           S.push(p.left);
       if( p.right != null )
           S.push(p.right);
   }
}
```

What recursive traversal is this equivalent to?



24

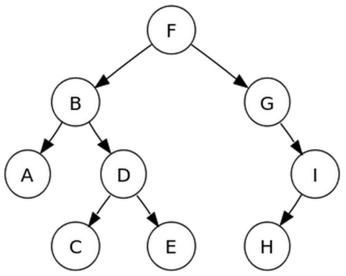
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

Let's try using a stack first:

```
void traverse(Node t) {
   Stack<Node> S = new Stack<Node>();
   S.push(t);
   while( !S.isEmpty() ) {
       Node p = S.pop();
       visit( p );
       if( p.left != null )
            S.push(p.left);
       if( p.right != null )
            S.push(p.right);
   }
}
```



What recursive traversal is this equivalent to? Reverse Preorder

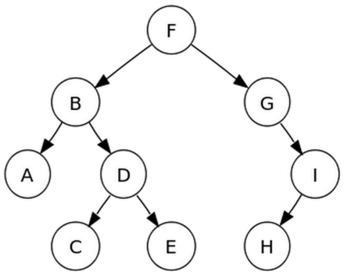
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

Let's try using a stack first:

```
void traverse(Node t) {
   Stack<Node> S = new Stack<Node>();
   S.push(t);
   while( !S.isEmpty() ) {
       Node p = S.pop();
       visit( p );
       if( p.left != null )
            S.push(p.left);
       if( p.right != null )
            S.push(p.right);
   }
}
```



How could we get a (normal) Preorder Traversal?

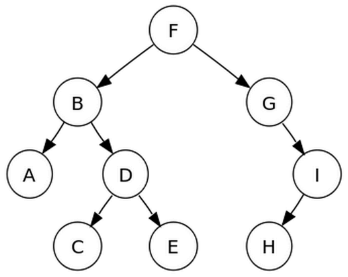
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

Let's try using a stack first:

```
void traverse(Node t) {
   Stack<Node> S = new Stack<Node>();
   S.push(t);
   while( !S.isEmpty() ) {
      Node p = S.pop();
      visit( p );
      if( p.right!= null )
          S.push(p.right);
      if( p.left!= null )
          S.push(p.left);
   }
}
```



How could we get a (normal) Preorder Traversal?

27

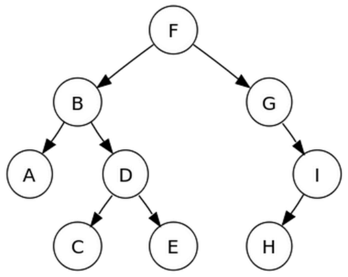
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

What happens if we use a **Queue** instead of a stack?

```
void traverse(Node t) {
   Queue<Node> Q = new Queue<Node>();
   Q.enqueue(t);
   while( !Q.isEmpty() ) {
      Node p = Q.dequeue();
      visit( p );
      if( p.left != null )
          Q.enqueue(p.left);
      if( p.right != null )
          Q.enqueue(p.right);
   }
}
```



Does this correspond to any of our recursive traversals?

28

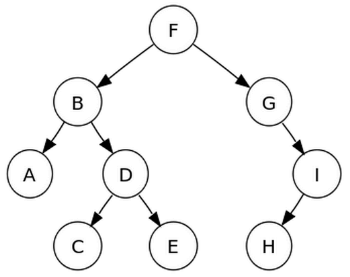
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

What happens if we use a **Queue** instead of a stack?

```
void traverse(Node t) {
   Queue<Node> Q = new Queue<Node>();
   Q.enqueue(t);
   while( !Q.isEmpty() ) {
      Node p = Q.dequeue();
      visit( p );
      if( p.left != null )
           Q.enqueue(p.left);
      if( p.right != null )
           Q.enqueue(p.right);
   }
}
```



Does this correspond to any of our recursive traversals? NO!

29

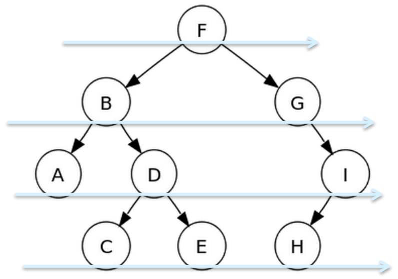
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

What happens if we use a **Queue** instead of a stack?

```
void traverse(Node t) {
   Queue<Node> Q = new Queue<Node>();
   Q.enqueue(t);
   while( !Q.isEmpty() ) {
      Node p = Q.dequeue();
      visit( p );
      if( p.left != null )
           Q.enqueue(p.left);
      if( p.right != null )
           Q.enqueue(p.right);
   }
}
```



This is called a Breadth-First or Level-Order Traversal.

30

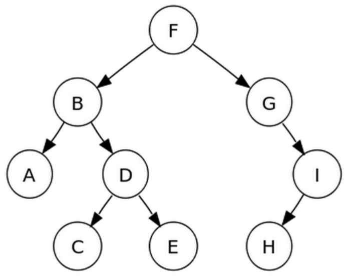
Tree Traversals: Non-Recursive Traversals

Computer Science

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

What happens if we use a **Queue** instead of a stack?

```
void traverse(Node t) {
   Queue<Node> Q = new Queue<Node>();
   Q.enqueue(t);
   while( !Q.isEmpty() ) {
       Node p = Q.dequeue();
       visit( p );
       if( p.right  != null )
            Q.enqueue(p.right);
       if( p.left != null )
            Q.enqueue(p.left);
   }
}
```



What happens if we reverse the order we enqueue the children?

31

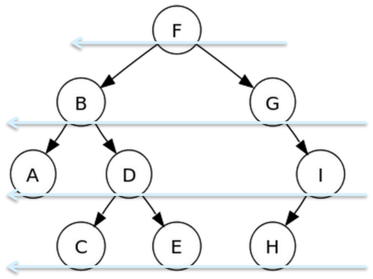
Tree Traversals: Non-Recursive Traversals

Computer Science

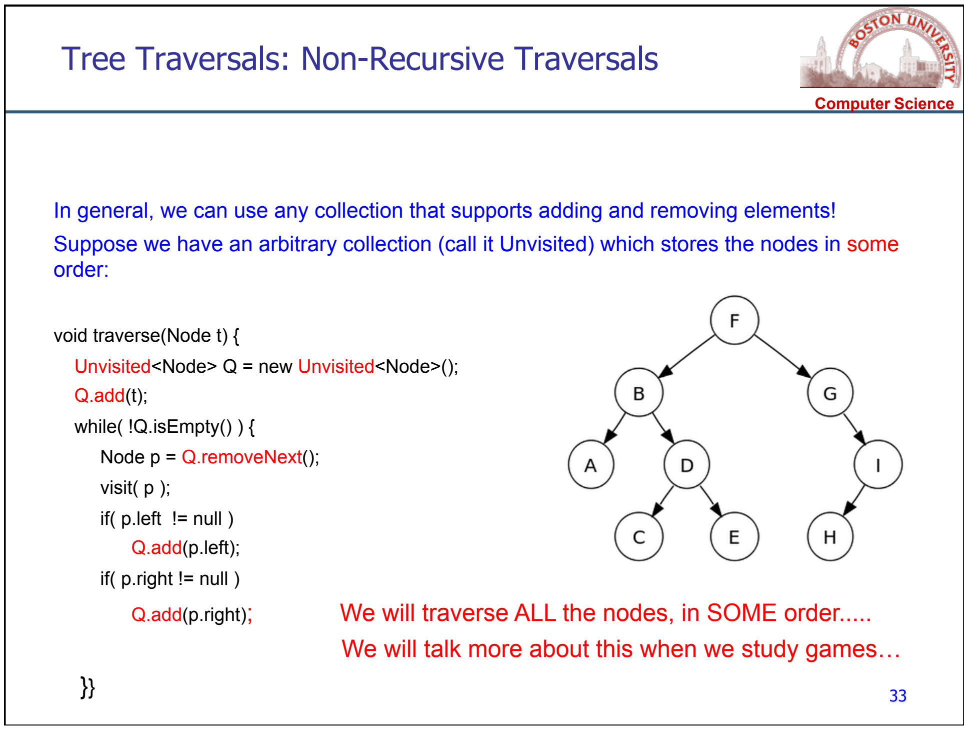
We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

What happens if we use a **Queue** instead of a stack?

```
void traverse(Node t) {
   Queue<Node> Q = new Queue<Node>();
   Q.enqueue(t);
   while( !Q.isEmpty() ) {
       Node p = Q.dequeue();
       visit( p );
       if( p.right  != null )
            Q.enqueue(p.right);
       if( p.left != null )
            Q.enqueue(p.left);
   }
}
```



32



Tree Traversals: Non-Recursive Traversals

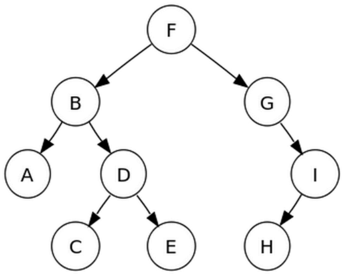
Computer Science

In general, we can use any collection that supports adding and removing elements!

Suppose we have an arbitrary collection (call it Unvisited) which stores the nodes in some order:

```
void traverse(Node t) {
   Unvisited<Node> Q = new Unvisited<Node>();
   Q.add(t);
   while( !Q.isEmpty() ) {
       Node p = Q.removeNext();
       visit( p );
       if( p.left  != null )
            Q.add(p.left);
       if( p.right != null )
            Q.add(p.right);

  }}
```



We will traverse ALL the nodes, in SOME order.....

We will talk more about this when we study games…

33