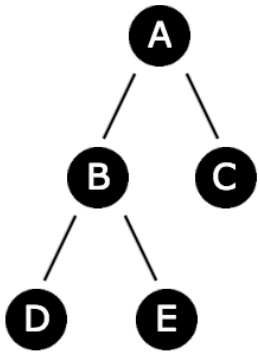
Article type: Algorithm

Tree traversal algorithms

mrdaniel published this on 5/31/16

Tree traversal is the process of visiting each node in a tree, such as a binary tree or binary search tree, exactly once. There are several effective traversal algorithms which we will cover below.

All of the algorithms below will implement Node objects we create, which were covered in a previous algorithm on linked lists. Although, we will be slightly changing the code for the nodes. The tree we will be operating on looks like the following:



And we can assume the tree is properly constructed via the following code which sets up nodes and links them to their proper child nodes:

```
function Node(data) {
  this.data = data;
  this.left = null;
  this.right = null;
}

// create nodes
var root = new Node('A');
var n1 = new Node('B');
var n2 = new Node('C');
var n3 = new Node('D');
```

```
var n4 = new Node('E');

// setup children
root.left = n1;
root.right = n2;
n1.left = n3;
n1.right = n4;
```

Pre-order

A pre-order traversal on a tree performs the following steps starting from the root:

1) Return the root node value.
2) Traverse the left subtree by recursively calling the pre-order function.
3) Traverse the right subtree by recursively calling the pre-order function.

For the tree above, performing a pre-order traversal would output the node values in the following order:

A, B, D, E, C

For the actual code implementation, we will be maintaining an array for the order of the nodes:

```
function pre_order(root, nodes) {
    nodes.push(root.data);
    if (root && root.left) {
        pre_order(root.left, nodes);
    }
    if (root && root.right) {
        pre_order(root.right, nodes);
    }
    return nodes;
}

pre_order(root, []); // => [ A, B, D, E, C ]
```

In-order

An in-order traversal on a tree performs the following steps starting from the root:

1) Traverse the left subtree by recursively calling the in-order function.
2) Return the root node value.
3) Traverse the right subtree by recursively calling the in-order function.

For the tree above, performing an in-order traversal would output the node values in the following order:

D, B, E, A, C

```
function in_order(root, nodes) {
    if (root && root.left) {
        in_order(root.left, nodes);
    }
    nodes.push(root.data);
    if (root && root.right) {
        in_order(root.right, nodes);
    }
    return nodes;
}

in_order(root, []); // => [ D, B, E, A, C ]
```

You can see that the only difference between the code for the in-order vs. pre-order traversal is where the appending of the node value is placed in the code. For post-order traversal below, that will be the only change as well.

A good way to remember when to return the node value (or append the node value to an array) is, for pre-order do it first, for in-order do it between the left and right traversal, and as you'll see below, for post-order do it after traversing the left and right subtrees.

Post-order

A post-order traversal on a tree performs the following steps starting from the root:

1) Traverse the left subtree by recursively calling the post-order function.
2) Traverse the right subtree by recursively calling the post-order function.
3) Return the root node value.

For the tree above, performing a post-order traversal would output the node values in the

following order:

D, E, B, C, A

```
function post_order(root, nodes) {
    if (root && root.left) {
        post_order(root.left, nodes);
    }
    if (root && root.right) {
        post_order(root.right, nodes);
    }
    nodes.push(root.data);
    return nodes;
}

post_order(root, []); // => [ D, E, B, C, A ]
```

Level-order

A level-order traversal on a tree performs the following steps starting from the root:

1) Add the root to a queue.
2) Pop the last node from the queue, and return its value.
3) Add all children of popped node to queue, and continue from step 2 until queue is empty.

For the tree above, performing a level-order traversal would output the node values in the following order:

A, B, C, D, E

```
function level_order(root, nodes) {
    var queue = [root];
    while (queue.length > 0) {
        // front of queue is at element 0 and we push elements to back of queue
        var n = queue.shift();
        nodes.push(n.data);
        if (n.left !== null) { queue.push(n.left); }
        if (n.right !== null) { queue.push(n.right); }
    }
    return nodes;
}

level_order(root, []); // => [ A, B, C, D, E ]
```

Applications of tree traversals

The algorithms above have several use cases in software and development. Below is a list of some of these common cases:

1) To construct any binary tree, you need the in-order traversal array of nodes and either a pre-order or post-order array.
2) A binary search tree can be constructed using only its pre-order traversal array.
3) The in-order traversal of a binary search tree produces the elements in sorted order.
4) You can perform a breadth-first search on a tree using a level-order traversal.

Comments (6)

8

Wow, thank you for posting this. This is extremely helpful.

CoderMan77 commented on 01/06/17

4

Java Code InOrder

```
public void inOrder(Node root){
    inOrder(root.left);
    System.out.println(root.data + " ");
    inOrder(root.right);
}
```

PreOrder

```
public void preOrder(Node root){
    System.out.println(root.data + " ");
    preOrder(root.left);
    preOrder(root.right);
}
```