Construct BST from given preorder traversal | Set 2

Given preorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.

```
     10
   /   \
  5     40
 /  \      \
1    7      50
```

We have discussed O(n^2) and O(n) recursive solutions in the previous post. Following is a stack based iterative solution that works in O(n) time.

1. Create an empty stack.

2. Make the first value as root. Push it to the stack.

3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.

4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.

5. Repeat steps 2 and 3 until there are items remaining in pre[].

C

```
/* A O(n) iterative program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
typedef struct Node
{
    int data;
    struct Node *left, *right;
} Node;

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
```

```
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from pre[]
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );

    // The first element of pre[] is always root
    Node* root = newNode( pre[0] );

    // Push root
    push( stack, root );

    int i;
    Node* temp;

    // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
        while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
            temp = pop( stack );

        // Make this greater value as the right child and push it to the stack
        if ( temp != NULL)
        {
            temp->right = newNode( pre[i] );
            push( stack, temp->right );
        }

        // If the next value is less than the stack's top value, make this value
```

```
        // as the left child of the stack's top node. Push the new node to stack
        else
        {
            peek( stack )->left = newNode( pre[i] );
            push( stack, peek( stack )->left );
        }
    }

    return root;
}


// A utility function to print inorder traversal of a Binary Tree
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Java

```
// Java program to construct BST from given preorder traversal

import java.util.*;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    // The main function that constructs BST from pre[]
    Node constructTree(int pre[], int size) {

        // The first element of pre[] is always root
        Node root = new Node(pre[0]);

        Stack<Node> s = new Stack<Node>();

        // Push root
        s.push(root);

        // Iterate through rest of the size-1 items of given preorder array
        for (int i = 1; i < size; ++i) {
            Node temp = null;
```

```
            /* Keep on popping while the next value is greater than
             stack's top value. */
            while (!s.isEmpty() && pre[i] > s.peek().data) {
                temp = s.pop();
            }

            // Make this greater value as the right child and push it to the stack
            if (temp != null) {
                temp.right = new Node(pre[i]);
                s.push(temp.right);
            }

            // If the next value is less than the stack's top value, make this value
            // as the left child of the stack's top node. Push the new node to stack
            else {
                temp = s.peek();
                temp.left = new Node(pre[i]);
                s.push(temp.left);
            }
        }

        return root;
    }

    // A utility function to print inorder traversal of a Binary Tree
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{10, 5, 1, 7, 40, 50};
        int size = pre.length;
        Node root = tree.constructTree(pre, size);
        System.out.println("Inorder traversal of the constructed tree is ");
        tree.printInorder(root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
 1 5 7 10 40 50
```

Time Complexity: O(n). The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of constructTree(). Therefore, time complexity is O(n).