Section F – Binary Search Trees [Answer 1 Specified Qn from this Section]

Information: Program templates for questions 1-5 are given as separated files. You must use them to implement your functions.

1. (**levelOrderTraversal**) Write an iterative C function `levelOrderTraversal` prints a level-by-level traversal of the binary tree using a **queue**, starting at the root node level. Note that you should **only** use `enqueue()` or `dequeue()` operations when you add or remove integers from the queue. Remember to empty the queue at the beginning, if the queue is not empty.

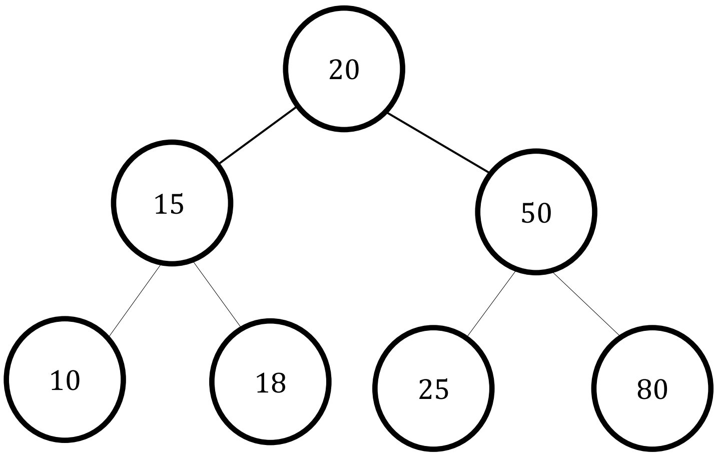
   The function prototype is given as follows:

   ```
   void levelOrderIterative(BSTNode *root);
   ```

   Following is the detailed algorithm:

   ```
   1) Create an empty queue q
   2) temp_node = root /*start from root*/
   3) Loop while temp_node is not NULL
        a) print temp_node->data
        b) Enqueue temp_node's children (first left then right
           children) to q
        c) Dequeue a node from q and assign its value to temp_node
   ```

   Let's consider the below tree for example.



Level-order Tree Traversal: **20 15 50 10 18 25 80**

2. **(inOrderIterative)** Write an iterative C function `inOrderIterative()` that prints the in-order traversal of a binary search tree using **a stack**. Note that you should **only** use `push()` or `pop()` operations when you add or remove integers from the stack. Remember to empty the stack at the beginning, if the stack is not empty.

   The function prototype is given as follows:

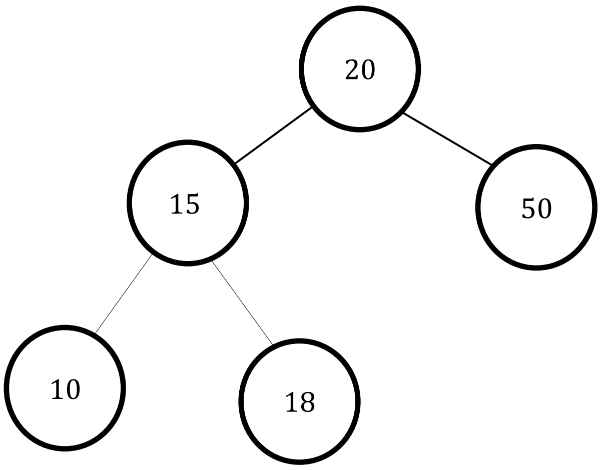
   ```
   void inOrderIterative(BSTNode *root);
   ```

   Following is the detailed algorithm:

   ```
   1) Create an empty stack S.
   2) Initialize current node as root
   ```

```
3) Push the current node to S and set current = current->left
   until current is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then you are done
```

Let us consider the below tree for example.



Following are the steps to print inorder traversal of the above tree.

Step 1 Creates an empty stack: S = NULL
Step 2 sets current as address of root: current -> 20
Step 3 Pushes the current node and set current = current->left until current is NULL
current -> 20
push 20: Stack S -> 20
current -> 15
push 15: Stack S -> 15, 20
current -> 10
push 10: Stack S -> 10, 15, 20
current = NULL
Step 4 pops from S
a) Pop 10: Stack S -> 15, 20
b) print "10"
c) current = NULL /*right of 10 */ and go to step 3
Since current is NULL step 3 doesn't do anything.

Step 4 pops again.
a) Pop 15: Stack S -> 20
b) print "15"
c) current -> 18/*right of 15 */ and go to step 3

Step 3 pushes 18 to stack and makes current NULL
Stack S -> 18, 20
current = NULL

Step 4 pops from S
a) Pop 18: Stack S -> 20
b) print "18"
c) current = NULL /*right of 18 */ and go to step 3
Since current is NULL step 3 doesn't do anything

Step 4 pops again.
a) Pop 20: Stack S -> NULL

b) print "20"
c) current -> 50 /*right of 20 */

Step 3 pushes 50 to stack and makes current NULL
Stack S -> 50
current = NULL

Step 4 pops from S
a) Pop 50: Stack S -> NULL
b) print "50"
c) current = NULL /*right of 50 */

Traversal is done now as stack S is empty and current is NULL.

Iterative Inoder Tree Traversal: **10 15 18 20 50**

3. (**preOrderIterative**) Write an iterative C function `preOrderIterative()` that prints the pre-order traversal of a binary search tree using **a stack**. Note that you should **only** use `push()` or `pop()` operations when you add or remove integers from the stack. Remember to empty the stack at the beginning, if the stack is not empty.

The function prototype is given as follows:

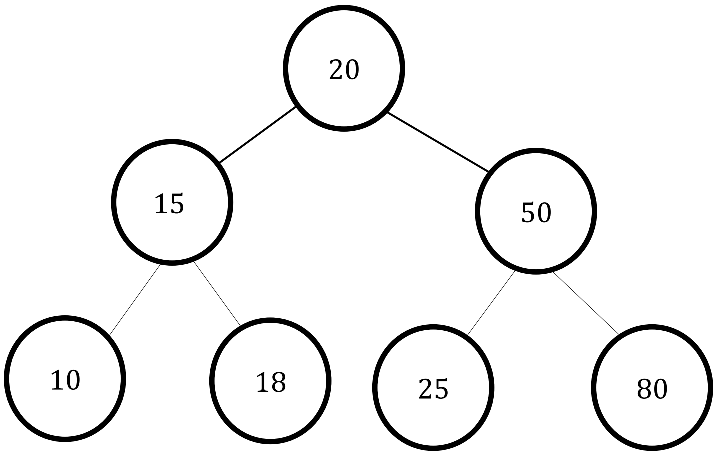
```
void preOrderIterative(BSTNode *root);
```

Following is the detailed algorithm:

```
1) Create an empty stack nodeStack and push root node to stack.
2) Do following while nodeStack is not empty.
     a) Pop an item from stack and print it.
     b) Push right child of popped item to stack
     c) Push left child of popped item to stack

   Right child is pushed before left child to make sure that
   left subtree is processed first.
```

Let us consider the below tree for example.



Iterative preorder Tree traversal: **20 15 10 18 50 25 80**

4. **(postOrderIterativeS1)** Write an iterative C function `postOrderIterativeS1()` that prints the post-order traversal of a binary search tree using **a stack**. Note that you should **only** use `push()` or `pop()` operations when you add or remove integers from the stack.

Remember to empty the stack at the beginning, if the stack is not empty.

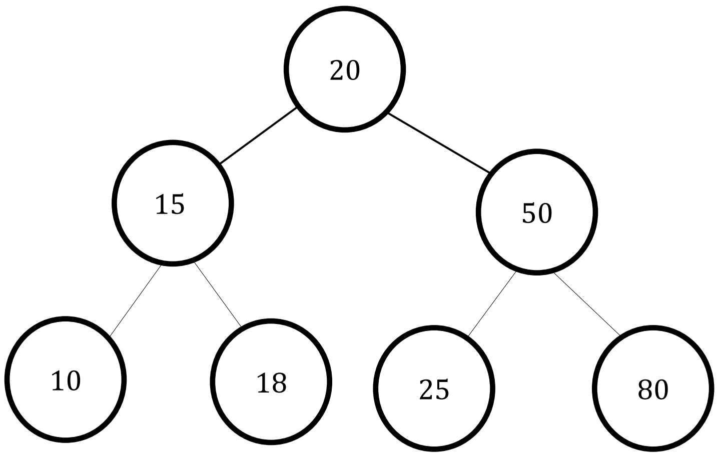
The function prototype is given as follows:

```
void postOrderIterativeS1(BSTNode *node);
```

Following is the detailed algorithm:

```
1.1 Create an empty stack
2.1 Do following while root is not NULL
      a) Push root's right child and then root to stack.
      b) Set root as root's left child.
2.2 Pop an item from stack and set it as root.
      a)  If the popped item has a right child and the right
          child is at top of stack, then remove the right
          child from stack, push the root back and set root
          as root's right child.
      b)  Else print root's data and set root as NULL.
2.3 Repeat steps 2.1 and 2.2 while stack is not empty.
```

Let us consider the below tree for example



Following are the steps to print postorder traversal of the above tree using one stack.

1) Right child of 20 exists.
   Push 50 to stack. Push 20 to stack. Move to left child.
   Stack: 50, 20

2) Right child of 15 exists.
   Push 18 to stack. Push 15 to stack. Move to left child.
   Stack: 50, 20, 18, 15

3) Right child of 10 doesn't exist.
   Push 10 to stack. Move to left child.
   Stack: 50, 20, 18, 15, 10

4) Current node is NULL.
   Pop 10 from stack. Right child of 10 doesn't exist.
   Print 10. Set current node to NULL.
   Stack: 50, 20, 18, 15

5) Current node is NULL.
   Pop 15 from stack. Since right child of 15 equals stack top element, pop 18 from stack. Now push 15 to stack.
   Move current node to right child of 15 i.e. 18
   Stack: 50, 20, 15

6) Right child of 18 doesn't exist. Push 18 to stack. Move to left child.
   Stack: 50, 20, 15, 18

7) Current node is NULL. Pop 18 from stack. Right child of 18 doesn't exist.
   Print 18. Set current node to NULL.
   Stack: 50, 20, 15

8) Current node is NULL. Pop 15 from stack.
   Right child of 15 is not equal to stack top element.
   Print 15. Set current node to NULL.
   Stack: 50, 20

9) Current node is NULL. Pop 20 from stack.
   Since right child of 20 equals stack top element, pop 50 from stack.
   Now push 20 to stack. Move current node to right child of 20 i.e. 50
   Stack: 20

10) Repeat the same as above steps and Print 25, 80 and 50.
    Pop 20 and Print 20.

Iterative Postorder Traversal: **10 18 15 25 80 50 20**

5. **(postOrderIterativeS2)** Write an iterative C function `postOrderIterativeS2()` that prints the post-order traversal of a binary search tree using **two stacks**. Note that you should **only** use `push()` or `pop()` operations when you add or remove integers from the stacks. Remember to empty the stacks at the beginning, if the stacks are not empty.

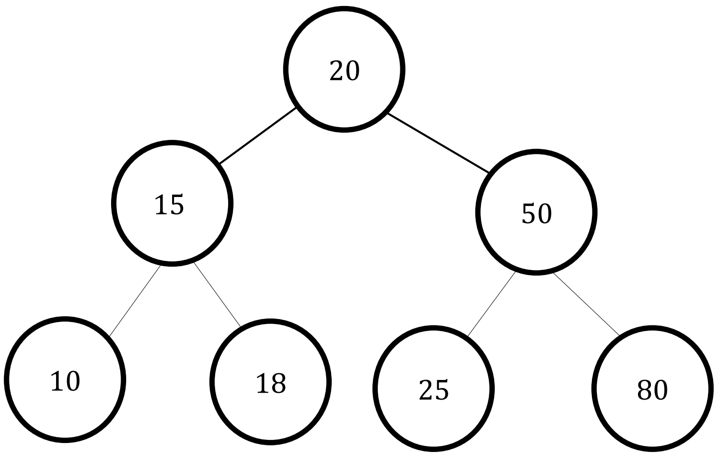
The function prototype is given as follows:

```
void postOrderIterativeS2(BSTNode *root);
```

Following is the detailed algorithm:

```
1) Push root to first stack.
2) Loop while first stack is not empty
    2.1) Pop a node from first stack and push it to second
         stack
    2.2) Push left and right children of the popped node to
         first stack
3) Print contents of second stack
```

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using two stacks.

1) Push 20 to first stack.
   First stack: 20
   Second stack: Empty

2) Pop 20 from first stack and push it to second stack.
   Push left and right children of 20 to first stack.
   First stack: 15, 50
   Second stack: 20

3) Pop 50 from first stack and push it to second stack.
   Push left and right children of 50 to first stack.
   First stack: 15, 25, 80
   Second stack: 20, 50

4) Pop 80 from first stack and push it to second stack.
   Push left and right children of 80 to first stack (**NULL**).
   First stack: 15, 25
   Second stack: 20, 50, 80

5) Pop 25 from first stack and push it to second stack.
   Push left and right children of 25 to first stack (**NULL**).
   First stack: 15
   Second stack:20, 50, 80, 25

6) Pop 15 from first stack and push it to second stack.
   Push left and right children of 15 to first stack.
   First stack: 10, 18
   Second stack: 20, 50, 80, 25, 15

7) Pop 18 from first stack and push it to second stack.
   Push left and right children of 18 to first stack (**NULL**).
   First stack: 10
   Second stack: 20, 50, 80, 25, 15, 18

8) Pop 10 from first stack and push it to second stack.
   Push left and right children of 10 to first stack (**NULL**).
   First stack: Empty
   Second stack: 20, 50, 80, 25, 15, 18, 10

   The algorithm stops since there is no more items in first stack.
   Observe that content of second stack is in postorder fashion. Print them.

   Iterative Postorder Traversal: **10 18 15 25 80 50 20**