S.No: 10	Exp. Name: ***Binary tree traversal using linked list - Non Recursive***	Date:2024-05-25

Aim:

Write a C program to implement Binary tree traversal using Linked list.
You have to complete the function `inorder`, `preorder` and `postorder` in `Traversarls.c` where parameters passed are the root reference's of the binary tree **T1**.

`Note`: Assume that tree is a **Complete Binary Tree** and driver code is already provided for you to run the test cases.

Source Code:

TreeMain.c

```
// Program for linked implementation of complete binary tree
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include<stdbool.h>
#include"TreeStructure.c"
#include"Traversals.c"

// For Queue Size
#define SIZE 100000

// A utility function to create a new Queue
struct Queue* createQueue(int size)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof( struct Queue ));
    queue->front = queue->rear = -1;
    queue->size = size;
    queue->array = (struct node**) malloc
                   (queue->size * sizeof( struct node* ));
    int i;
    for (i = 0; i < size; ++i)
        queue->array[i] = NULL;
    return queue;
}

// Standard Queue Functions
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}

int isFull(struct Queue* queue)
{
   return queue->rear == queue->size - 1;
}

int hasOnlyOneItem(struct Queue* queue)
{
   return queue->front == queue->rear;
}
```

```
void Enqueue(struct node *root, struct Queue* queue)
{
    if (isFull(queue))
        return;
    queue->array[++queue->rear] = root;
    if (isEmpty(queue))
        ++queue->front;
}
struct node* Dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    struct node* temp = queue->array[queue->front];

    if (hasOnlyOneItem(queue))
        queue->front = queue->rear = -1;
    else
        ++queue->front;
    return temp;
}

struct node* getFront(struct Queue* queue)
{  return queue->array[queue->front]; }

// A utility function to check if a tree node
// has both left and right children
int hasBothChild(struct node* temp)
{
    return temp && temp->left && temp->right;
}

// Function to insert a new node in complete binary tree
void insert(struct node **root, int data, struct Queue* queue)
{
    // Create a new node for given data
    struct node *temp = newNode(data);
    // If the tree is empty, initialize the root with new node.
    if (!*root)
        *root = temp;
    else
    {
        // get the front node of the queue.
        struct node* front = getFront(queue);

        // If the left child of this front node doesn’t exist, set the
        // left child as the new node
        if (!front->left)
        {
                front->left = NULL;
            front->left = temp;
        }
        // If the right child of this front node doesn’t exist, set the
        // right child as the new node
        else if (!front->right)
        {
                front->right = NULL;
```

```
            front->right = temp;
        }
        // If the front node has both the left child and right child,
        // Dequeue() it.
        if (hasBothChild(front))
        {
                Dequeue(queue);
        }
    }
    // Enqueue() the new node for later insertions
            Enqueue(temp, queue);
}

// Standard level order traversal to test above function
void levelOrder(struct node* root)
{
    struct Queue* queue = createQueue(SIZE);
    Enqueue(root, queue);
    while (!isEmpty(queue))
    {
        struct node* temp = Dequeue(queue);
        printf("%d ", temp->data);
        if (temp->left)
            Enqueue(temp->left, queue);
        if (temp->right)
            Enqueue(temp->right, queue);
    }
}

// Driver program to test above functions
int main()
{
    struct node* T1 = NULL;
    struct Queue* queue1 = createQueue(SIZE);
    int ele;

    while(1){
       printf("Enter value : ");
       scanf("%d",&ele);
    if(ele==-1){
        break;
    }
    else{
           insert(&T1, ele, queue1);
       }
    }
    inorder(T1);
    preorder(T1);
    postorder(T1);

}
```

TreeStructure.c

```
// A tree node
struct node
{
    int data;
    struct node *right,*left;
    struct node *root;
};

// A queue node
struct Queue
{
    int front, rear;
    int size;
    struct node* *array;
};

// A utility function to create a new tree node

struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc(sizeof( struct node ));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

typedef struct node *BTNODE;

struct stacknode {
   BTNODE node;
   struct stacknode * next;
};

typedef struct stacknode * STKNODE;
STKNODE top = NULL;
int isempty() {
   if(top == NULL) {
      return 1;
   }
   return 0;
}
```

Traversals.c

```
void push(BTNODE b) {
   STKNODE temp;
   temp = (STKNODE)malloc(sizeof(struct stacknode));
   if(temp == NULL) {
      printf("Stack is overflow.\n");
   } else {
      temp -> node = b;
      temp -> next = top;
      top = temp;
   }
```

```
}
BTNODE peek() {
   if (top == NULL) {
      return NULL;
   }
   return top->node;
}
BTNODE pop() {
   STKNODE temp;
   BTNODE b;
   if(top == NULL) {
      printf("Stack is underflow.\n");
      return 0;
   } else {
      temp = top;
      top = top -> next;
      b = temp->node;
      free(temp);
      return b;
   }
}

STKNODE newStackNode(BTNODE b) {
   STKNODE temp =  (STKNODE)malloc(sizeof(struct node));
   temp->node = b;
   temp->next = NULL;
   return temp;
}
BTNODE newNodeInBT(int item) {
   BTNODE temp =  (BTNODE)malloc(sizeof(struct node));
   temp->data = item;
   temp->left = temp->right = NULL;
   return temp;
}
void inorder(BTNODE root) {

   // Write the inorder() traversal function, and you are free to write your own util
ity functions
   // Required to complete the inorder traversal without using recursion
printf("Inorder:");

STKNODE stack = NULL;
struct node* curr = root;
while (curr != NULL || !isempty()) {
while (curr != NULL){
push(curr);
curr = curr->left;}
curr = pop();

printf(" %d", curr->data);
curr = curr->right;
}
printf(" \n");
}

void preorder(BTNODE root) {
```

```
   // Write the preorder() traversal function, and you are free to write your own uti
lity functions
   // Required to complete the preorder traversal without using recursion
printf("Preorder: ");

if(root == NULL)
return;
push(root);
while (!isempty()){
struct node* curr = pop();
printf("%d ", curr->data);
if (curr->right)
push(curr->right);
if (curr->left)
push(curr->left);
}
printf("\n");


}

void postorder(BTNODE root) {

   // Write the postorder() traversal function, and you are free to write your own ut
ility functions
   // Required to complete the postorder traversal without using recursion
printf("Postorder: ");

if (root == NULL)
return;
STKNODE s1 = NULL;
STKNODE s2 = NULL;
push(root);

while (!isempty()) {
struct node* curr = pop();
STKNODE temp = (STKNODE)malloc(sizeof(struct stacknode));
temp->node = curr;
temp->next = s2;
s2 = temp;

if (curr->left)
push(curr->left);
if(curr->right)
push(curr->right);
}
while (s2 != NULL) {
printf("%d ", s2->node->data);
STKNODE temp = s2;
s2 =s2->next;
free(temp);
}
printf("\n");

}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
Enter value : 4
Enter value : 8
Enter value : 9
Enter value : 6
Enter value : 7
Enter value : -1
Inorder: 6 8 7 4 9
Preorder: 4 8 6 7 9
Postorder: 6 7 8 9 4

Test Case - 2
User Output
Enter value : 9
Enter value : 8
Enter value : 3
Enter value : 4
Enter value : 7
Enter value : 5
Enter value : 1
Enter value : -1
Inorder: 4 8 7 9 5 3 1
Preorder: 9 8 4 7 3 5 1
Postorder: 4 7 8 5 1 3 9