ESO207-Assignment 2

Mandar Bapat

October 5 , 2020

1 Question 1

1.1 Part A

We assume that we have the stack operations and integer variables:
Push(x,S) - Push element x to the stack S
Pop(S) - Pop an element from the stack S
Top(S) - Returns the topmost element of stack S
IsEmpty(S) - Return True if stack S is empty , False if not.
integer variable n = Length of stacks S1 and S2

```
We have 2 stacks S1 and S2.In a queue we have 2 operations:
Enqueue(x) and Dequeue() , where x is the element to be enqueued.
```

1.1.1 Strategy for Enqueue(x)

```
The new element is entered at the top of S1.So Enqueue(x) will
implement Push(x,S1) on S1.
```

1.1.2 Strategy for Dequeue()

```
If S1 and S2 are empty , then we return error.
Else if S2 is empty , we transfer all the elements from
S1 to S2 and execute Pop(S2) operation on S2.
The transfer of elements will involve popping elements from S1
and pushing them into S2.
```

1.2 Part B

1.2.1 Pseudo Code

Assume that stacks are of length n.

```
1  int IsFull(){
2      if(length(S1)=n){
3          return 1; //1 is for True
4      }
```

```
5       else{
6           return 0;
7       }
8   }
9
10  void Enqueue(int x){
11      if(not IsFull()){
12          Push(x,S1);
13      }
14      else{
15          return -1; //-1 is for error when stack S1 is full
16      }
17  }
18
19  int Dequeue(){
20
21      if(IsEmpty(S2) and IsEmpty(S1)){
22          return -1;  //-1 is for error
23      }
24      else if(Isempty(S2) and (not Isempty(S1))){
25          while(not IsEmpty(S1)){
26              int temp1 = Pop(S1);
27              Push(temp1,S2);
28          }
29          int temp = Pop(S2);
30          return temp;
31      }
32      else{
33          int temp = Pop(S2);
34          return temp;
35      }
36  }
37
38  int IsEmpty(){
39      if(IsEmpty(S1) and IsEmpty(S2)){
40          return 1;   //1 for True
41      }
42      else return 0;
43  }
```

1.2.2 Time Complexity

For the Enqueue(x) operation , we have O(1) time complexity as it is similar to Push(x) on a stack , particularly S1 here.

For the Dequeue() operation , we have O(n) time complexity , where n

is the number of elements in the queue.The queue is basically implemented on S1.Hence when we need to transfer n elements from S1 to S2 , as described in the strategy.Hence the time complexity is O(n).

1.3 Part C

Correctness of Algorithm
To prove the correectness of the algorithm , we need to show that the Enqueue(x) and Dequeue() operations through stacks S1 and S2 follow the FIFO structure , i.e. , the First In First Out structure for the final queue.

The top element of S1 is the newest element and the top element of S2 is the oldest/earliest.

Stack S1 is only concerned with Enqueue(x) operation,i.e, we only add elements to S1.Stack S2 is concerned with the Dequeue() operation.We are dealing with the queue of size 2*n and breaking it into 2 portions of size n : (a.)Stack S1 , which shows us the back portion of the queue,i.e, the newest n elements and (b.)Stack S2 , which shows us the front portion,i.e, the oldest n elements.

1.3.1 Enqueue(x) operation

The Enqueue(x) operation implements Push(x) on S1.This clearly shows that the newest elements lie on top of S1.
The condition of IsFull() checks if the back portion of the queue excedes size n.

1.3.2 Dequeue() operation

For the Dequeue() operation :
Case 1: When S1 is filled(may or may not be full) and S2 is empty

In this case , all elements are popped from S1 and pushed into S2.Since stacks follow LIFO structure , i.e , Last In First Out structure , the topmost element of S1 , which is also the newest element is popped first and pushed into S2.Hence the newest element lies at the bottom of S2 and the oldest on the top.

We then implement the Pop() operation on S2.Since the oldest element lies on the top of S2 , it is popped out. This is exactly in line with the FIFO structure.

Case 2: When S1 is not empty and S2 is not empty OR
When S1 is empty and S2 is not empty

In both of the above cases , the Pop() operation is

implemented on S2 , which returns the oldest element.
Until S2 does not become empty Case 2 will be followed.
Note that newer element keep getting pushed to S1.That has
no effect on S2.The top element of S2 still contains the
oldest element.
This is exaclty in line with FIFO and this is what the
program outputs.

Case 3: When S1 is empty and S2 is empty

This case implies that there are no elements at the
back of the queue and no elements at the front of the
queue.
Hence Dequeue() operation cannot be implemented.
This is what the program returns.

1.3.3 IsFull() and IsEmpty() operations

The IsFull() and IsEmpty() operations just put restrictions on the size of the queue.The IsFull() operation checks if the size of the back portion of the queue , i.e , S1 , does not excede n.

IsEmpty() returns True if there are no back and front portions,i.e, S1 and S2 are empty.

They dont directly affect the FIFO structure.

2 Question 2

2.1 Part A

We define a structure called Node in C language , which will have 2 pointers to 2 child Nodes:left and right.It will also have an integer variable containing its data.

We assume that we have a tree.We also assume that while implementing the Inorder traversal function , we are provided with a root Node.

Pseudo Code for Inorder Traversal

```
1  struct Node{
2      int data;
3      struct Node* left;
4      struct Node* right;
5  };
```

```
6
7   void InorderTraversal(struct Node* root){
8       if(root=NULL){
9           return;
10      }
11
12      InorderTraversal(root->left);
13
14      printf("%d ",root->data);
15
16      InorderTraversal(root->right);
17  }
```

2.2 Part B

2.2.1 Procedure

```
Steps:
(i.)We create an empty stack S1
(ii.)We create a Node current and initialize it to root Node.
(iii.)We implement the Push(current Node) to S1 and then update
     current Node to current->left.
(iv.)When current = NULL , and S1 is not empty , we implement:
   Pop() operation on S1 and print it , update current to
   current->right and then execute step (iii.) again.
(v.)When S1 is empty and current = NULL , then we stop the process
```

2.2.2 Pseudo Code for the above procedure

We assume that we have access to the 4 stack operations mentioned in Part A of Question 1.We also assume that we start with an empty stack S1.

```
1
2   struct Node{
3       int data;
4       struct Node* left;
5       struct Node* right;
6   };
7
8   void InorderTraversal(struct Node* root){
9       struct Node* current = root;
10
11      while(True){
12          if(current != NULL){
13              Push(current,S1);
14              current=current->left;
15          }
```

```
16          else{
17              if(not IsEmpty(S1)){
18                  current = Pop(S1);
19                  printf("%d ",current->data);
20                  current=current->right;
21              }
22              else{
23                  break;
24              }
25          }
26      }
27  }
```

2.3 Part C

Correctness of the above Pseudo Code

We prove the correctness of the algorithm by Structural Induction.The proof of the recursive and the non-recursive algorithm follow the same arguments as only the way of implementing has changed but not the method.We define some notations and terms:

T = for a binary tree
RNode = root node of T
L = left subtree of T whose root node is the left child node of RNode
R = right subtree of T whose root node is the right child of RNode

Proof :
Base Case : When the tree is empty.
When the tree is empty , the *current pointer* is NULL.The stack S1 is empty from the start.Hence the loop terminates and nothing is printed.
This is exactly what was required.
Hence the algorithm outputs correctly for empty tree.

Constructor Case: We have a RNode of T.
We assume that the algorithm outputs correctly for subtrees L and R.We then prove that the program outputs correctly for tree T.

Now since *current* is not equal to NULL , we execute Push(current,S1) operation and update *current pointer* with *current->left*.

Now by our assumption , the algorithm outputs correctly for subtree L and hence the output for *current->left* and the subsequent nodes in L will be correct.

After the subtree L is printed, the stack has RNode.

The *printf()* statement is executed , which prints the value of the RNode.After which the *current pointer* is updated with *current->right*.

The right subtree R is printed correctly by our assumption.

The final output will be:
Inorder on L then RNode then Inorder on R.

This is exatly in line with the Inorder Traversal definition , which is what the program outputs. Hence the algorithm correctly outputs for tree T.

By the rule of Structural Induction , the algorithm outputs correctly for any binary tree T.

3 Question 3

3.1 Part A

Pseudo Code for Non-Recursive Merge Sort We assume that we already have stack operations mentioned in Part A of Question 1.

```
1  struct Node{
2      int x;
3      int y;
4  };
5
6  struct Node combine(int x,int y){
7      struct Node n = {x,y};
8      return n;
9  }
10 void merge(long long int* arr,int len,int l,int m,int r){
11     int i = l;
```

```
12      int j = m+1;
13      int k = l;
14      long long int temp[len];
15
16      while(k<=r){
17          if(arr[i]<=arr[j]){
18              temp[k]=arr[i];
19              k++;
20              i++;
21              continue;
22          }
23          if(arr[i]>arr[j]){
24              temp[k]=arr[j];
25              k++;
26              j++;
27              continue;
28          }
29          if((i>m)&&(j<=r)){
30              temp[k]=arr[j];
31              k++;
32              j++;
33              continue;
34          }
35          if((i<=m)&&(j>r)){
36              temp[k]=arr[i];
37              k++;
38              i++;
39              continue;
40          }
41      }
42      for(int i=l;i<=r;i++){
43          arr[i]=temp[i];
44      }
45  }
46
47  void nonRecursiveImplementation(long long int* arr,int len,struct
48  Node* S1,int l,int r){
49      while(IsEmpty()==0){
50          int m = l + (r-l)/2;
51
52          while(l<r){
53
54              if(m!=r){
55                  Push(combine(m+1,r),S1);
56              }
57              Push(combine(l,r),S1);
```

```
58                  r=m;
59                  m=(r+l)/2;
60              }
61
62              struct Node n1 = Pop(S1);
63              l = n1.x;
64              r = n1.y;
65              m = l + (r-l)/2;
66
67              struct Node n2 = Top(S1);
68              if((n2.x==(m+1)) && (n2.y==r)){
69                  Pop(S1);
70                  Push(n1,S1);
71                  l = m+1;
72              }
73              else{
74                  merge(arr,len,l,m,r);
75                  l=r;
76              }
77          }
78      }
```