ESO207A: Data Structures and Algorithms
Assignment 2

Sarthak Rout - 190772

October 1, 2020

---

1 Q1

1.1 Task A - Strategy

- The main difference between stacks and queues is that stack is LIFO and queue is FIFO. To implement a queue using 2 stacks, we shall try to use LIFO twice to get FIFO. Let us define S1 to be the "Main Stack" containing the queue and S2 to be an "Auxiliary Stack". In S1, the queue will be stored in such a way that the front is always at the top and the most recently inserted element is at the bottom. This makes *Enqueue* cumbersome but makes *Dequeue* easy. I have assumed that the length of stack S2 is atleast the size of S1.
- For *Enqueue* operation, we will pop all elements in main stack S1 and push them to S2. This will reverse the order of elements as the last element to be pushed to stack S1, is now pushed first to stack S2 and vice-versa. Now, the new element to be inserted is pushed to stack S1 . Then, all elements of stack S2 are popped and pushed into stack S1 over the new element. This, stores the new element at the bottom of stack S1 and the front of the queue at the top. Also, we need to check if we can indeed push the new element by use of a variable storing length of queue.
- For *Dequeue* operation, we will simply pop the topmost element of the stack as it represents the front element of queue.
- For *isQueueEmpty* operation, we can check if the main stack S1 is empty or not by *isStackEmpty* operation.
- For *isQueueFull* operation, we can check if the main stack S1 is full or not by *isStackFull* operation.

1.2 Task B - Pseudo code

Algorithm 1 Queue using 2 Stacks

```
{We have queue Q as stack S1, S2 with stack operations : isStackEmpty(S), isStackFull(S), Push(S, x), Pop(S)}
function isQueueEmpty(Q)
  return isStackEmpty(S1)
end function
function isQueueFull(Q)
  return isStackFull(S1)
end function
function Enqueue(Q, x)
  if isStackFull(S1) then
    return "Queue is full!"
  end if
  while !isStackEmpty(S1) and !isStackFull(S2) do
    S2.Push(S1.Pop())
  end while
  S1.Push(x)
  while !isStackEmpty(S2) and !isStackFull(S1) do
    S1.Push(S2.Pop())
  end while
end function
function Dequeue(Q)
  if isStackEmpty(S1) then
    return "Queue is Empty!"
  end if
end function
=0
```

1.2.1 Complexity

- *isQueueEmpty* : $c \cdot \mathcal{O}(\textit{isStackEmpty}(S)) = \mathcal{O}(1) \cdot \mathcal{O}(\textit{isStackEmpty}(S))$. Assuming array implementation of stack, it should be $O(1)$.
- *isQueueFull* : $c \cdot \mathcal{O}(\textit{isStackFull}(S)) = \mathcal{O}(1) \cdot \mathcal{O}(\textit{isStackFull}(S))$. Assuming array implementation of stack, it should be $O(1)$.
- *Enqueue* : $n \cdot (\mathcal{O}(\textit{Push}()) + \mathcal{O}(\textit{Pop}())) + \mathcal{O}(\textit{Push}()) + n \cdot (\mathcal{O}(\textit{Push}()) + \mathcal{O}(\textit{Pop}())) = (2n+1) \cdot \mathcal{O}(\textit{Push}()) + 2n\mathcal{O}(\textit{Pop}())$. Assuming array implementation of stack, it should be $\mathcal{O}(n)$, where n is number of elements in the queue initially.
- *Dequeue* : $c \cdot \mathcal{O}(\textit{Pop}()) = \mathcal{O}(1) \cdot \mathcal{O}(\textit{Pop}())$. Assuming array implementation of stack, it should be $O(1)$.

1.3 Task C - Correctness

- Since the queue is stored in the main stack S1, the operations *isQueueEmpty* and *isQueueFull* is trivially correct by checking if S1 is empty or full respectively.
- For *Enqueue* operation: The queue is stored in S1 in such a way that the front is at the top and the last element of the queue(most recently inserted) is at the bottom. When, elements of stack S1 are popped, the oldest elements in order of insertion, are popped first and the reversed queue is stored in auxiliary stack S2 such that the front of the queue is at the bottom of the stack and the most recently inserted element is at the top.
  Now, when we insert another element in S1, it becomes the most recently inserted element as all other elements are pushed above it. Then, this procedure to reverse elements is called, to store them in S1 after popping them from S2. So, the new element is stored at the bottom of the stack as intended.
- For *Dequeue* operation, the element on top of the stack is the oldest element to have been inserted , and is to be removed. So, we can simply pop it out of the stack and the next oldest element becomes the top of stack and it becomes the front element.
- Hence, this strategy is correct.

2 Q2

2.1 Task A - Recursive Pseudo Code for Inorder Traversal

Algorithm 2 Inorder Traversal (Recursive)

```
function InorderTraversal($x$)
  if x == NIL then
    return
  end if
  InorderTraversal(x.lchild)
  print(x.val)
  InorderTraversal(x.rchild)
end function=0
```

2.2 Task B - Non-recursive Pseudo Code

Algorithm 3 Inorder Traversal (Non-Recursive) using Stack

```
function InorderTraversalStack($x$)
  S ← CreateStack()
  S.Push(x);
  while !S.Empty() do
    root ← S.Pop()
    if root is not NIL then
      if root.lchild == NIL then
        print(root.val) continue
      end if
      S.Push(root.rchild)
      old_lchild ← root.lchild
      root.lchild ← NIL
      S.Push(root)
      S.Push(old_lchild)
    end if
  end while
end function=0
```

2.3 Task C - Correctness

- In Inorder Traversal, the center/root is visited after the left subtree has been visited and before the right subtree is visited.
- In a stack, items are present in such a order that, always left child nodes is pushed after right child nodes so that they are visited first while popping.
- Whenever a node doesn't have a left subtree or it's left subtree is visited, it itself can be visited or evaluated.
- In this algorithm, whenever a node is visited for first time, it is broken down into 3 parts : left child, itself and right child and in the stack, the right child, the node and it's actual left child are pushed again. But, the center node's left child is marked empty akin to visited before pushing it.
- So, first the left subtree is traversed . When the node is visited again, as it has been marked having no left child (meaning that the left subtree has been visited or it had no left child originally) , it itself is visited , then the right subtree is visited.
- As the order of left $\rightarrow$ root $\rightarrow$ right is maintained as described above, so, the algorithm is traverses the tree correctly.

3 Q3

3.1 Task A - Pseudo Code - Non Recursive - Merge Sort with Stack

Algorithm 4 MergeSortwithStacks - Non Recursive

```
Structure OPERATION { string type, integer first, integer second }
function MERGEWITHVECTORS($A, first, second$)
mid = (first + second) / 2
Temp1 ← CREATEVECTOR<INTEGER>(), Temp2 ← CREATEVECTOR<INTEGER>()
for each element from first to mid inclusive do
  Temp1.Push_back(element)
end for
for each element from mid + 1 to second inclusive do
  Temp2.Push_back(element)
end for
Temp1.Push_back($\infty$), Temp2.Push_back($\infty$)
idx ← first, size1 = 0, size2 = 0
while size1 < temp1.size() - 1 OR size2 < temp2.size() - 1 do
  if temp1[size1] > temp2[size2] then
    A[idx] ← temp2[size2]
    idx ← idx + 1 , size2 = size2 + 1
  else
    A[idx] ← temp1[size1]
    idx ← idx + 1 , size1 = size1 + 1
  end if
end while
end function
function MERGESORTWITHSTACKS($A, n$)
Records ← CREATESTACK<OPERATION>()
Records.Push({"sort", 0, n - 1}) {n is size of the array A}
while !Records.Empty() do
  record ← Records.pop()
  if record.type == "merge" then
    MERGEWITHVECTORS(A, record.first, record.second)
  else if record.type == "sort" then
    if record.first ≥ record.second then
      continue
    end if
    mid ← (record.first + record.second)/2
    Records.Push("merge", record.first, record.second)
    Records.Push("sort", mid + 1, record.second)
    Records.Push("sort", record.first, mid)
  end if
end while
end function
MERGESORTWITHSTACKS($A, n$)
return =0
```

In fixed stack implementation, a size of *2*n* has to given for *Records* stack.

4 References

- https://www.techiedelight.com/stack-implementation-using-templates/ - How to use templates for making stacks.