Alexandria University
Faculty of Engineering

Computer and Systems Engineering Department
Class 2025
Level 1 Second SEMESTER
CSE - X22: Data Structures I

Due: 26 April 2022

Document Description:

- CSE - X22: Data Structures I Course - Sheet 4 (Queues and Trees) - Assignment Submission.

Solution Model Example:

- [Question Number]
  [Problem Statement]
  [“Answer:”]
  [Solution]

Introduced By:

- **Student Name:** ***George Samy Wahba Beshay***
- **Student ID:** ***20010435.***

Question 1:

Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 front operations and 15 dequeue operations, 5 of which generated EmptyQueueExceptions, which were caught and ignored. What is the current size of Q?

Answer:

Based on that 5 of the dequeue executed operations generated an EmptyQueueExceptions, that means that only 10 out of the 15 dequeue operation executions were applicable.
Taking in consideration that the front() method does not have an effect on the Queue - as it just returns the front element without removing it - so we will ignore it.
The remaining 22 enqueue operations will be executed and so 22 new elements would be added to the queue to the current size which - based on our assumption - is equal to zero.
Finally the Current Size of the Queue would be = 22.

Example on our assumption to make it clear:
NOTE: Element in RED is the front element.

Queue	Operation	Queue+	Size
[-]	10 * enqueue	[X1, .. , X10]	10 - 1 + 1
[X1, .. , X10]	15 * dequeue	[-] , 5 Exceptions	0
[-]	22 * enqueue	[X11, .. , X32]	32 - 11 + 1

Final Size = 32 - 11 + 1 = 22.

Question 2:

Consider the array-based implementation of the queue ADT.

1. Suppose that you used the index variable r to point to the index after the last element in the queue. The front of the queue is the element at the index 0.Write algorithms to enqueue and dequeue elements from the queue. What is the problem with this implementation?
2. Alternatively, you would store the index of the first and last elements of the queue by the variables f and r. What are the values of f and r when: i. The queue is empty. ii. The queue is full. What is the problem with this scheme?
3. You then chose to solve the problem using a circular array (you would store the index of the first and last elements of the queue by the variables f and r and let f and r wrap around the array). What are the values of f and r when: i. The queue is empty. ii. The queue is full. What is the problem with this scheme?
4. You then chose to solve the problem by not allowing the last slot to be filled. How would you know that the queue is full and empty? Write the pseudocode for the Queue operations (use mod operations).

Answer:

1. *// Assume Array of size N*
   // r is variable that stores the index of the rear element - the element after the last -
   // Index 0 will always be the front element - the element that has been stored in
   // the queue for the longest time -
   // enqueue operation will be of O(1) as it will check whether the rear element is <
   // the size of the array (N) or not, if yes, then it will be inserted and r will be
   // incremented.
   // dequeue operation will be of O(n) as it will shift all of the stored elements
   // 1 place frontly after removing the front element.

Algorithm enqueue()

Input: Object Obj to be inserted in the Queue.

Output: Returns nothing – Void – , Queue after addition of the object.

```
If (r == size){
Throw FullQueueException
} else {
Array[r] ← Obj;
r++;
Queue_Size++;          // Number of elements currently stored in the Q.
return;
}
```

Algorithm dequeue()

Input: –

Output: Returns the front element – if there is – and update the queue

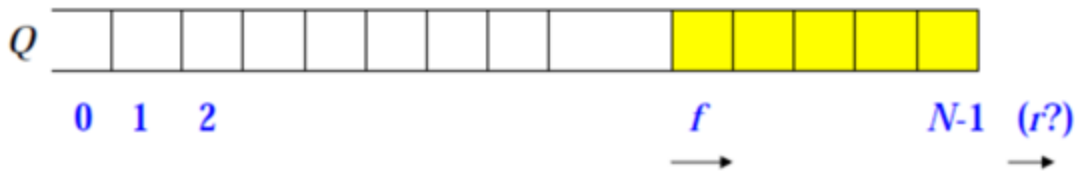
```
if(Queue_Size == 0 || r == 0){
Throw EmptyQueueException
} else {
Object Temp ← Array[0];
Loop ( for i = 0 to i = r-1 ){
        Array[i] ← Array[i+1];
}
r – –;
return Temp;
}
```

The Problem with this implementation is the high cost, as we have seen in the Dequeue operation, due to shifting all the stored elements one step after The removal of the front element. – O(n) –

2. If the index of the first and last elements are stored in (f) and (r) variables.
   f ← Index of the front element.
   r ← Index of the rear element - element after the last. -

   I. When the queue is empty:
      f = r = 0
   II. When the queue is full:
      Also, f = 0, r = N

   The problem with this scheme appears when more than one dequeue and enqueue operation are executed we will find that f is incremented when dequeue operation takes place, and r is incremented when enqueue operation is called. So what happens when r = N while the queue is not full yet, we will run out of the array bounds. Attached below a picture that visualizes the problem:



3. The circular array fashion is used to solve this problem.
   The values of f and r in the following cases:
   I. The queue is empty:
      f = r.
   II. The Queue is full:
      f = r.
   The problem with this scheme is that the cases mentioned above are confusing and would generate a problem in detecting whether the queue is empty or full. However, there are 2 methods to overcome this problem, 1 - Variable "Queue_Size" That counts the number of elements currently in the queue.
   2 - Not allowing the last slot to be filled.

4. If we wont allow the last slot to be filled, To know if the queue is empty or full we will check the following:

The Queue is empty when f = r

The Queue is full when:

r + 1 == f OR (r == N-1) && (f == 0)

<u>The Queue operations using mod operation:</u>

```
Algorithm enqueue(Object obj){
        Input: obj that to be inserted in the queue.
        Output: Void.
        // We will check first if the queue is full or not
        if(Queue_Size == N-1)
                Throw FullQueueException;
        Array[r] ← obj;
        r = (r+1) mod N;
        Queue_Size++;
        return;
}
Algorithm dequeue(){
        Input: -
        Output: The front element in the queue - if it exists. -
        // We will check first if the queue is empty or not.
        if(Queue_Size == 0 OR f == r)
                Throw a EmptyQueueException;
        Object Temp ← Array[f];
        Array[f] ← Null;
        Queue_Size - -;        // Size Decremented
        f ← (f+1) mod N
        return Temp;
}
```

Question 3:

Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may not use an array or linked list. Use only S and Q and a constant number of reference variables.

Answer:

```
Algorithm Boolean ScanUsingQ(Stack S, Object Obj){
        Input: Stack S, Object obj, we will scan S to check whether obj belongs to it or not.
        Output: Boolean value indicating whether the obj was found in the stack or not.
        Boolean Flag ← false;
        Queue MyQueue ← new Queue();              // Assume max_size > Stack Size.
        Loop (while S_Size != o){
                if(S.peek() equal to Obj)
                        Flag ← True;
                MyQueue.enqueue(S.pop());
        }
        // NOTE: StackQueueTransformation is a method defined in the next page.
        StackQueueTransformation(S, MyQueue, false);          // Queue ⇒ Stack
        // Now The elements are returned to the stack S but in an inverse order, so we will
        // need to do the same operation executed before once again to return  the  order
        // to its initial form.
        StackQueueTransformation(S, MyQueue, true);           // Stack ⇒  Queue
        StackQueueTransformation(S, MyQueue, false);          // Queue ⇒ Stack
        return Flag;
}
```

```
Algorithm void StackQueueTransformation(Stack s, Queue Q, Boolean Choice){
        Input: Stack s, Queue Q, Boolean C, C indicates to move from S to Q or the opposite
        Output: Void, Move the elements from the Stack to the Queue or the opposite
        If (Choice){            // if  "choice" is true, move from stack to queue.
                Loop (while S_Size != 0){
                        Q.enqueue(S.pop());
                }
        } else {                // if "choice" is false, move from queue to stack
                Loop(while Q_Size !=0){
                        S.push(Q.dequeue());
                }
        }
        return;
}
```

Question 4:

Show how to implement a stack using two queues.

Answer:

```
// Let Q1 and Q2 be 2 defined Queues will all their ordinary related methods
    // On pushing elements, we will check if Q1 is empty or not, if yes, then immediately
    // enqueue the element in Q1, if not - and this the most probable - then dequeue all
    // elements from Q1 and enqueue them to Q2, then enqueue the element to be
    // inserted in Q1, then dequeue all the elements from Q2 and enqueue them to Q1.
    // On popping elements, we will use dequeue() from Q1 with no constraints.
```

```
Algorithm void DoublyQueuedBasedStack_push(Object Obj){
        Input: Obj to be inserted in the Stack
        Output: Void.
        if(Q1.Size() is equal to Zero){
                Q1.enqueue(Obj);
        }
        else  {
                Loop(while Q1.Size() != o){
                        Q2.enqueue(Q1.dequeue());
                }
                Q1.enqueue(obj);
                Loop(while Q2.Size() != o){
                        Q1.enqueue(Q2.dequeue());
                }
        }
        DoublyQueuedBasedStack_Size++;
        return;
}
Algorithm Object DoublyQueuedBasedStack_Pop(){
        Input: Void.
        Output: Returns the top most – peek - element in the stack.
        DoublyQueuedBasedStack_Size--;
        return Q1.dequeue();
}
```

Question 5:

Given a priority queue with the interface:

void Insert(Item i, int priority);

Item DeleteMax();

Describe how to implement a stack with the interface:

void Push(Item i)

Item Pop() .

Answer:

```
// Given that the used Queue desired to be a priority queue, the key will  be an integer
    // indicating the right order - position - that the element should be placed based
    // on it in the queue.
    // Note that the priority queue inserts elements with greater priority at the front
    // of the queue, and smaller priority at the end of the queue.
    // Example:
    //                                   (X1, X5, X10)  // X10 is the front element.
    // after pushing X4 and X15 with different priorities (X5 > X4 > X1), (X15 > X10)
    //                                   (X1, X4, X5, X10, X15) // X15 is the front.
    //                            NOT    (X15, X4, X1, X5, X10)
    // For push() method, In the implementation we will use the Priority Queue Insert
    // method with the parameters defined, notice that the Priority is a global field.
    // For the pop() method, DeleteMax() method will be used directly with no
    // constraints.
    Integer  priority ← Zero;                    // Global Field
    Algorithm Push(Item i){
            PriorityQueue.Insert(i, priority);
            priority ++;
            Stack_Size++;                 // Number of current elements stored.

    }
```

Algorithm Pop(){

Input: Void

Output: Returns the front element.

return PriorityQueue.DeleteMax();

}

Question 6:

You are given an array A of size N. You can perform an operation in which you will remove the largest and the smallest element from the array and add their difference back into the array. So, the size of the array will decrease by 1 after each operation. You are given Q tasks and in each task, you are given an integer K. For each task, you have to tell the sum of all the elements in the array after K operations.

Sample input:

5 2 ← Array size is 5 and number of queries is 2

3 2 1 5 4 ← The array elements

1 ← First query

2 ← Second query

Sample output:

13 ← Subtract 5 - 1 = 4, the result array will be 3, 2, 4, 4 with sum 13

9 ← Subtract 5 - 1 = 4, 4 - 2 = 2, the result array will be 3, 2, 4 with sum 9

What data structure would you use to solve this problem? Think of one, or invent one, then write a pseudo code for this problem.

Answer:

The Data Structures that will be used to solve this problem:

Double - Ended - Priority – Queue. (WILL BE REFERRED TO IT BY DEPQ)

Where this DS consists of a DEqueue to be able to dequeue elements from both

Edges, in addition to the priority property that would be inherited from the priority Queue.

Below the implementation of the main methods that will be needed:

Consider Q1 → DE queue (Main)

Q2 → Priority queue (Supporting Queue)

```
/*
 * The Following method will insert an element into the DEPQ.
 *  Keeping the order of the queue correct.
 */
Algorithm void EnqueuePRO(Object obj){
        Loop(while Q1.Size() != 0){
                Int tempObjAndKey = Q1.removeFirst();
                Q2.enqueue(tempObjandKey, tempObjandKey);
// Since they are integers, their values are considered to be their keys also.
        }
        Q2.enqueue(obj, obj);
        Loop(while Q2_Size() != 0){
                Q1.InsertLast(Q2.dequeue());
        }
}
Algorithm Object RemoveFirstPRO(){
        // No difference DEqueue method "removeFirst" will be used.
        return Q1.removeFirst();
}
Algorithm Object RemoveLastPRO(){
        // No difference DEqueue method "removeLast" will be used.
        return Q1.removeLast();
}
Algorithm int Solve(int[] Array, int k){
        Int sum ← 0;
        DEPQ SolverQ ← new DEPQ();
        Loop(for i = 0 ; i < Array.length() ; i++){
```

```
            SolverQ.EnqueuePRO(Array[i]);
        }
        Loop(while k > o){
            Int temp = SolverQ.RemoveFirstPro() - SolverQ.RemoveLastPro();
            SolverQ.EnqueuePRO(temp);
            k--;
        }
        Loop(while SolverQ.Size() != o){
            sum += SolverQ.RemoveFirstPro();
        }
        return sum;
}
Algorithm main(){
        Scanner sc ← new Scanner(System.in);
        Scan Array size, and number of Queries.
        Scan Array Elements. number of tasks in each queries.
        Loop(while number of Queries > o){
            Print(Solve(Array,K(i)));     // K(i) is the number of tasks of Query i.
            number of Queries--;
        }
        finish.
}
```

Question 7:

An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a time-stamp that denotes the time when the event occurs. The simulation program needs to efficiently perform the following two fundamental operations:

1. Insert an event with a given time-stamp (that is, add a future event)
2. Extract the event with the smallest time-stamp (that is, determine the next event to process).

Which data structure should be used for the above operations? Why?

Answer:

The most ***suitable*** and ***efficient*** DS for those methods is the ***priority queue.***

Because: on inserting a new event - enqueue() - the companion key will be the time stamp, which will make the queue re-arrange the events in the desired order of which the smallest timestamp will be at the front of the queue.

And on extracting the event that should takes place next, the dequeue() method is enough.

NOTE: the timestamps should be multiplied by negative in order to invert their arrangement,

Example:

Array Events = [(E1, 10 mins), (E2, 15 mins), (E3, 7 mins)]

PriorityQueue *problemSolverQueue ← new PriorityQueue();*

Loop (Ei: in Events){

```
    problemSolverQueue.Enqueue(Ei.getObj(), -Ei.getTime())
```

}

So at the end the problemSolverQueue will be as below:

[(E2), (E1), (E3)] and their keys are [-15, -10, -7]

Question 8:

Write an algorithm that counts the number of leaf nodes in a binary tree. And another algorithm to count the number of internal nodes in the binary tree.

Answer:

```
// By using the recursion logic, counting the leaf nodes of a binary tree will be done.
    // where the first call will be countLeafNodes(TreeNode);
    Algorithm int countLeafNodes(binaryNode v){
            Input: binaryNode to check.
            Output: Integer indicating the number of Leaf nodes under this node
                                                                        inclusively.
            If (v is null) then
                    return 0 ;      // if null then returns zero
            else {
                    If (!v.hasRightChild && !v.hasLeftChild)   // if true then this node is
                                                                        a leaf node
                            return 1;
                    else {                                      // internal node ignore it.
                            return countLeafNodes(v.rightChild) +
                                                        countLeafNodes(v.leftChild);
                    }
            }
    }
    // Example:
                        A
                    /       \
                  B           C
                /   \         |
            D       E         F                     // Leaf Nodes = 3
```

```
Algorithm int countInternalNodes(binaryNode v){
        if(v.hasRightChild or v.hasLeftChild){      // Internal Node, then add 1.
            return 1 + countInternalNodes(r.rightChild) +
                                                countInternalNodes(r.leftChild);
        }
        else {
            return 0;
        }
    }
```

Question 9:

Two binary trees are equivalent if they both have the same structure with the same data in corresponding nodes. Write an algorithm to test whether two binary trees T1 and T2 are Equivalent.

Answer:

```
Algorithm Boolean checkEquivalence(binaryTree T1, binaryTree T2){
            return checkEquivalenceBinaryNode(T1.RootNode(), T2.RootNode());
    }

Algorithm Boolean checkEquivalenceBinaryNode(binaryNode n1, binaryNode n2){
        Boolean CheckFlag = true;
        if(    n1.hasLeftChild() && !n2.hasLeftChild() || !n1.hasLeftChild() &&
        n2.hasLeftChild() || n1.hasRightChild() && !n2.hasRightChild() ||
         !n1.hasRightChild() && n2.hasRightChild() ||  n1.visit() != n2.visit()    )
            CheckFlag ← false;
```

```
        else{
            if(n1.hasLeftChild)
                CheckFlag ← (   CheckFlag &&
                checkEquivalenceBinaryNode(n1.leftChild,n2.leftChild)   );
            if(n1.hasRightChild)
                CheckFlag ← (   CheckFlag &&
                checkEquivalenceBinaryNode(n1.rightChild,n2.rightChild) );
        }
        return checkFlag;
    }
```

Question 10:

Write an algorithm SwapTree(T) that takes a binary tree and swaps the left and right children of every node.

Answer:

```
Algorithm void SwapTree(BinaryTree T){
        Input: BinaryTree that to be swapped.
        Output: void.
        SwapNode(T.Root());
    }
```

```
Algorithm void SwapNode(BinaryNode n){
        Input: BinaryNode that to be swapped.
        Output: void.
        if(n.visit() is null)
                return;
        else{
                BinaryNode temp ← n.leftChild();
                n.LeftChild ← n.rightChild();
                n.rightChild ← temp;
                SwapNode(n.leftChild());
                SwapNode(n.rightChild());
                return;
        }
}
```

End of Sheet