Roll Number:_____   Number of Pages: 02

Thapar Institute of Engineering and Technology, Patiala
Department of Computer Science and Engineering
MID SEMESTER EXAMINATION
TENTATIVE SOLUTION

B. E. (Second Year):	Course Code: UCS301/UCS406
Semester-II (2019/20)	Course Name: Data Structures
March 07, 2019	Saturday, 10:30 Hrs – 12:30 Hrs
Time: 2 Hours, M. Marks: 25	Name of Faculty: SUG, RMT, SP, AA

Note: Attempt all questions (sub-parts) in sequence. Assume missing data, if any, suitably.

Q1. Convert the given infix expression into an equivalent postfix expression using stacks. Show contents of the stack at each intermediate step. Use the precedence table as shown in **Table 1**.   **(3)**

Table 1. Precedence of operators.

Operator	Precedence
#	Highest
^	
*, /	↓
+, –	
$	Lowest

(A $ B + C) # (K + L – M * N + O ^ P * W / U)

ANSWER 1.

(A $ B + C) # (K + L – M * N + O ^ P * W / U))

INPUT	STACK	OUTPUT
	(	
(	((	
A	((	A
$	(($	A
B	(($	A B
+	(($ +	A B
C	(($ +	A B C
)	(	A B C + $
#	(#	A B C + $
(	(# (	A B C + $
K	(# (	A B C + $ K
+	(# (+	A B C + $ K
L	(# (+	A B C + $ K L
–	(# (–	A B C + $ K L +
M	(# (–	A B C + $ K L + M
*	(# (– *	A B C + $ K L + M
N	(# (– *	A B C + $ K L + M N
+	(# (+	A B C + $ K L + M N * –
O	(# (+	A B C + $ K L + M N * – O
^	(# (+ ^	A B C + $ K L + M N * – O
P	(# (+ ^	A B C + $ K L + M N * – O P
*	(# (+ *	A B C + $ K L + M N * – O P ^
W	(# (+ *	A B C + $ K L + M N * – O P ^ W
/	(# (+ /	A B C + $ K L + M N * – O P ^ W *
U	(# (+ /	A B C + $ K L + M N * – O P ^ W * U
)	(#	A B C + $ K L + M N * – O P ^ W * U / +
)		A B C + $ K L + M N * – O P ^ W * U / + #

Q2. Let the letters **S**, **E**, **A**, **T**, **B**, **L** has to be pushed on to an empty stack of characters **(1)** in the order they appear from left to right. Consider that the output of **pop()** operation is appended in an initially empty output string.

Determine the sequence in which **push()** and **pop()** operations should be called such that the contents in the output string should be **STABLE**.

ANSWER 2.

OPERATION	STACK	OUTPUT STRING
Push('S')	S	
Pop()		S
Push('E')	E	S
Push('A')	E A	S
Push('T')	E A T	S
Pop()	E A	S T
Pop()	E	S T A
Push('B')	E B	S T A
Pop()	E	S T A B
Push('L')	E L	S T A B
Pop()	E	S T A B L
Pop()		S T A B L E

Q3. Compute complexity of the following pseudo-codes giving proper justifications. **(3)**

(a) O(n³)
```
for (int i = 1; i <= n; i++) // n times
  for(int j = 1; j <= i; j++) // 1 + 2 + 3 + … + n ≈ n² times
    for(int k = 1; k <= j; k++)
      // (1) + (1 + 2) + (1 + 2 + 3) + … + (1 + 2 + … + n) ≈ 1² + 2² + 3² + … + n² ≈ n³ times
    { … }
```

(b) O(n^{3/2}log n)
```
for (int p = 1; p + n/2 <= n; p++) // n/2 times
  for(int q = n; q > 0; q /= 2) // log n times for each value of p
    for(int r = 1; r*r <= n; r++) // √n times for each value of q
      for(int s = 1; s < n; s++) // 1 time for each value of r
        { break; … }
```
(c) O(n log₃n)
```
m = n;
while ( n > 0) // log₃n times
{  for(int i = 0; i < m; i++) // n times for each value of n
   …
   n = n/3;
}
```

Q4. (a) A two-dimensional array defined as **A [-4 … 6] [-2 … 12]** requires 2 bytes of **(2)** storage for each element. If the array is stored in row major order form with the address **A[4][8]** as 4142. Compute the address of **A[0][0]**.

(1)

ANSWER 4(a).

In row major order,
$$\text{Address of } A[I][J] = B + W (N (I - LBR) + (J - LBC))$$

Given:
    W = 2, LBR = –4, LBC = –2
    #rows = M = 6 + 4 + 1 = 11
    #columns = N = 12 + 2 + 1 = 15
    Address of A[4][8] = 4142

Address of A[4][8] = B + 2 (15 (4 – (–4)) + (8 – (–2)))
        4142 = B + 2 (15 (4 + 4) + (8 + 2)) = B + 2 (15 (8) + 10) = B + 2 (120 + 10)
        4142 = B + 260
Thus, B = 4142 – 260 = 3882

Now,
Address of A[0][0] = 3882 + 2 (15 (0 – (–4)) + (0 – (–2)))
        = 3882 + 2 (15(4) + 2) = 3882 + 2 (62)
        = 3882 + 124
        = 4006

(b) A circular queue of positive integers is implemented using a linear array **A [1..6]**. The present contents of **A** are **{4, …, …, 7, 9, 2}** where '**…**' represents empty. After two dequeue and one enqueue operations, let the positions of **front** and **rear** pointers be **X** and **Y**, respectively. Then **4X + 5Y** will be?

ANSWER 4(b).

1	2	3	4	5	6
4			7	9	2

A: **Front = 4, Rear = 1**

Two dequeue operations move front pointer ahead by two positions and thus Front reaches the index 6, i.e. X = 6.
Similarly, one enqueue operation moves rear pointer ahead by one position and thus Rear reaches the index 2, i.e. Y = 2.

Therefore, 4X + 5Y = 4(6) + 5(2) = 24 + 10 = 34.

Q5. Let **S** be an empty stack and **Q** be a queue with contents as shown in **Fig. 1**. **isEmpty(Q)** or **isEmpty(S)** returns **true** if **Q** or **S** is empty, else returns **false**. **top(S)** returns the character at the *top* of **S** without removing it from **S**.

Q:
D	A	T	A	B	A	S	E

⌐— Front

(3)

Fig. 1

```
1.  function1 (char val)
2.  { if (isEmpty(S) || top(S) < val)
3.    { push(S,val);
4.      return;
5.    }
6.    char c = pop(S);
7.    function1(val);
8.    if (c != top(S))
9.      push(S,c);
10. }
11. int main()
12. {  while (!isEmpty(Q))
13.      function1(dequeue(Q));
14. }
```
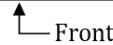
Fig. 2

Execute the code snippet given in **Fig. 2** and answer the following questions.

(a) Give the *contents* of **S** after each execution of the **while loop within main()**.

(b) Observe the output obtained in **Q5.(a)** and clearly state the purpose(s) of designing it.

ANSWER 5(a).

#iteration	Queue (Front is at LEFT)	Stack (Top is at RIGHT)
	D A T A B A S E	
1	A T A B A S E	D
2	T A B A S E	A D
3	A B A S E	A D T

4	B A S E	A D T
5	A S E	A B D T
6	S E	A B D T
7	E	A B D S T
8		A B D E S T

ANSWER 5(b).
It arranges only the unique contents of Q in increasing order.

Q6. A Radix Sort algorithm utilizing Counting Sort as an intermediate stable sorting **(5)** algorithm is to be applied on the contents of array **A** to arrange them in increasing order. Let **COUNT [0..9]** be the array that counting sort uses to store the frequency values during intermediate steps.

A: | 387 | 690 | 234 | 435 | 567 | 203 | 441 | 892 |

Illustrate step wise execution of counting sort, i.e. list the contents of array **COUNT**, for each of the iterations of Radix sort (starting from LSD). Also show the contents of array **A** across all the iterations of Radix sort.

ANSWER 6.
Iteration I:

	0	1	2	3	4	5	6	7
A:	387	690	234	435	567	203	441	892

	0	1	2	3	4	5	6	7	8	9	
COUNT:	0	0	0	0	0	0	0	0	0	0	Initial
COUNT:	1	1	1	1	1	1	0	2	0	0	Frequency
COUNT:	1	2	3	4	5	6	6	8	8	8	Cumulative Frequency
COUNT:	0	1	2	3	4	5	6	6	8	8	Final

	0	1	2	3	4	5	6	7
A:	690	441	892	203	234	435	387	567

Iteration II:

	0	1	2	3	4	5	6	7
A:	690	441	892	203	234	435	387	567

	0	1	2	3	4	5	6	7	8	9	
COUNT:	0	0	0	0	0	0	0	0	0	0	Initial
COUNT:	1	0	0	2	1	0	1	0	1	2	Frequency
COUNT:	1	1	1	3	4	4	5	5	6	8	Cumulative Frequency
COUNT:	0	1	1	1	3	4	4	5	5	6	Final

	0	1	2	3	4	5	6	7
A:	203	234	435	441	567	387	690	892

Iteration III:

	0	1	2	3	4	5	6	7
A:	203	234	435	441	567	387	690	892

	0	1	2	3	4	5	6	7	8	9	
COUNT:	0	0	0	0	0	0	0	0	0	0	Initial
COUNT:	0	0	2	1	2	1	1	0	1	0	Frequency
COUNT:	0	0	2	3	5	6	7	7	8	8	Cumulative Frequency
COUNT:	0	0	0	2	3	5	6	7	7	8	Final

	0	1	2	3	4	5	6	7
A:	203	234	387	435	441	567	690	892

Q7. Let **A[0 .. n – 1]** be an array of numbers. A number **A[i]** is called a *principal* if it is **(3)** greater than the mean of all numbers from **A[i]** to **A[n – 1]**. Write an efficient pseudo-code to delete all the *principal* numbers in the array **A**. Explain the proposed logic with the help of an example.

ANSWER 7.

If logic used is deleting principal elements during iterations.

> **1 Mark if anyone of the following is applied.**
> - **If sum/mean is computed only once and then updated during iterations moving from left to right.**
> - **If sum/mean is computed during iterations moving from right to left.**
>
> **0.5 Marks for rest of the logic.**
> **0.5 Marks for deletion.**
> **1 Mark for proper example.**

If logic used is deleting principal elements at the last.

> **0.5 Marks for the correct logic.**
> **0.5 Marks for deletion.**
> **1 Mark for proper example.**

Q8. Write a pseudo-code to implement **Round Robin CPU Scheduling** using **singly** **(4)** **circular linked list** assuming processes may have different arrival as well as burst times.

Round Robin is a pre-emptive scheduling algorithm in which CPU is assigned to a process on the basis of FCFS for a fixed amount of time known as 'time quantum'. After time quantum expires, the running process is pre-empted and the processor is assigned to the next arrived process.

Note:
- *tq represents time quantum, **SCLL** is a singly circular linked list maintaining list of processes which are being executed currently. **LL** is a simple linked list maintaining list of processes (in FCFS order) which are freshly arrived and yet to enter **SCLL**.*
- *At the end of each **tq**, all the processes are deleted from **LL** and are inserted in the same order in **SCLL**. Similarly, when the burst time reaches zero the corresponding process is finally deleted from **SCLL**.*
- *Memory should be allocated to a process only once when it arrives freshly. Also assume that insertion is happening automatically in **LL**, so there is no need to specify it's corresponding pseudo-code.*

ANSWER 8.

0.5 Mark for correct termination condition.
0.5 Mark for handling delay to simulate time quantum.
0.5 Mark for burst time update.
0.5 Marks for the pointer update if bust time has not reached zero.
0.5 Mark for process deletion if burst time has reached zero.
1.5 Marks for deletion from LL and insertion in SCLL.
- **0.5 Marks for head update of LL.**
- **0.5 Marks for tail update of SCLL OR complete scanning of SCLL**
- **0.5 Marks for pointer update required for insertion in SCLL.**