# National University of Computer and Emerging Sciences

**School of Computing** **Spring 2015** **Islamabad Campus**

# CS-201 Data Structures

Serial No:

## Sessional-I

**Total Time: 1 Hour**

**Total Marks: 50**

Monday, March 16, 2015

## Course Instructor

Mr. Syed Muhammad Hassan Mustafa

_____

Signature of Invigilator

_____ _____ _____ _____

Student Name Roll No Section Signature

**DO NOT OPEN THE QUESTION BOOK OR START UNTIL INSTRUCTED.**

**Instructions:**

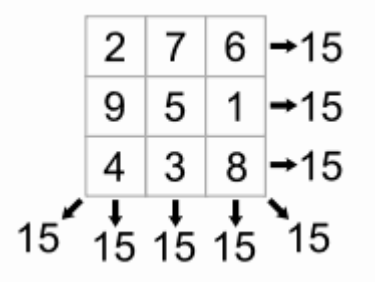
1. Attempt on question paper. Attempt all of them. Read the question carefully, understand the question, and then attempt it.
2. No additional sheet will be provided for rough work. Use the back of the last page for rough work.
3. If you need more space write on the back side of the paper and clearly mark question and part number etc.
4. After asked to commence the exam, please verify that you have Seven (7) different printed pages including this title page. There are a total of 5 questions.
5. Calculator sharing is strictly prohibited.
6. Use permanent ink pens only. Any part done using soft pencil will not be marked and cannot be claimed for rechecking.

| | Q-1 | Q-2 | Q-3 | Q-4 | Q-5 | Total |
|---|---|---|---|---|---|---|
| **Marks Obtained** | | | | | | |
| **Total Marks** | **5** | **10** | **10** | **10** | **15** | **50** |

**Vetted By: _____ Vetter Signature: _____**

## Question 1 [5 Marks]

Given an **M x N matrix** (not necessarily square), write a C++ function **"IsMagicSquare"** to verify if matrix is a Magic square. **IsMagicSquare** should return boolean (true/false) result only.
A Magic Square is a square matrix, such that the figures in each vertical, horizontal, and diagonal direction add up to the same value.
**(3 Marks for code)**




**A Magic Square**

```
bool isMagicSquare(int n, int m, int **nArray)
{

  if(n != m) return false;                                  // check square matrix

  int sum = 0;
  int DiagSum1 = 0;                                         // initialize sum variables
  int DiagSum2 = 0;

  for(int i=0;i<n;i++){                                     // calculate and compare
    int RowSum = 0;                                         sum of all rows
    DiagSum1 += nArray[i][i];
    DiagSum2 += nArray[i][n-i-1];                           // also calculate sum of
    for(int j=0; j<m; j++){                                 diagonals
      RowSum += nArray[i][j];
    }
    if(i == 0)    sum = RowSum;                             // calculate sum of each row
    else if( sum != RowSum) return false; /*case rows*/
  }

                                                            // compare with common
  if(DiagSum1 != DiagSum2 || DiagSum1 != sum)               sum
    return false; /*case diagonal*/

  for(int j=0;j<m;j++){                                     // compare diagonals' sum
    int ColSum = 0;                                         with common sum
    for(int i=0; i<n; i++){
      ColSum += nArray[i][j];                               // calculate and compare
    }                                                       sum of columns
    if( sum != ColSum) return false; /*case column*/
  }
  std::cout<<sum<<std::endl;
  return true;
}                                                           // print sum
                                                            // if everything goes fine,
                                                            return true
```

## Question 2 [10 Marks]

Given the following input:

| 3 | 1 | 4 | 1 | 5 | 9 |
|---|---|---|---|---|---|

You are required to show dry-run of quicksort algorithm on input data**. Pivot index is always the left most index of the sub-array (as shown in the below C++ code).**
You can use back of the paper for extra space.

```
int partition( int data[], int pivot_index, int first, int
last)
{
  int too_big_index = first+1;
  int too_small_index = last;
  int swap;
  do
  {
    while( data[too_big_index] <= data[pivot_index])
      ++too_big_index;

    while(data[too_small_index] > data[pivot_index])
      --too_small_index;

    if (too_big_index < too_small_index)
    {
      swap = data[too_big_index];
      data[too_big_index] = data[too_small_index];
      data[too_small_index] = swap;
    }

  }while(too_small_index > too_big_index);

  swap = data[too_small_index];
  data[too_small_index] = data[pivot_index];
  data[pivot_index] = swap;
  return too_small_index;
}

void quickSort(int arr[], int first, int last)
{
  if (first < last)
  {
    int pivotIndex = first;
    int splitPoint = partition(arr, pivotIndex, first, last);
    quickSort(arr,first,splitPoint-1);
    quickSort(arr,splitPoint+1,last);
  }
}
```

**Dry Run:** Do not show internal steps for partition function, only final output of partition function is required in each call i.e. you can show the changes in array like this:

Initial array

| 3 | 1 | 4 | 1 | 5 | 9 |
|---|---|---|---|---|---|

pivot_index=0 , splitPoint = 2
updated array

| 1 | 1 | 3 | 4 | 5 | 9 |
|---|---|---|---|---|---|

pivot_index = 0, splitPoint = 1
updated array

| 1 | 1 | 3 | 4 | 5 | 9 |
|---|---|---|---|---|---|

pivot_index=3, splitPoint=3
updated array

| 1 | 1 | 3 | 4 | 5 | 9 |
|---|---|---|---|---|---|

pivot_index=4, splitPoint=4
updated array

| 1 | 1 | 3 | 4 | 5 | 9 |
|---|---|---|---|---|---|

## Question 3 [10 Marks]

Given a linked list, write C++ code that can reverse the order of elements **without using any other data structure**.

e.g. Input: head => 1 => 2 => 3 => 4 => 5
Output: head => 5 => 4 => 3 => 2 => 1

Please note that you should provide **C++ code** for the method, not just the algorithm. Properly comment your code to make it readable. **(Correct code carries 5 marks)**

```
void reverse()
{
  if(start == NULL || start->next == NULL)
  {
    cout<<"Size too short to reverse"<<endl;
  }
  else{
    struct node* p1 = start;

    rec_reverse(start->next);
    p1->next->next = p1;
    p1->next = NULL;
    cout<<"List Reversed"<<endl;
  }

  display();
}
```

```
void rec_reverse(struct node* toReverse)
 {
  if(toReverse->next == NULL) // base case
  {
    start = toReverse;
  }
  else{
    rec_reverse(toReverse->next); // recursive step
    toReverse->next->next = toReverse;
  }
 }
```

## Question 4 [10 Marks]

Convert the following **Infix** expression to **Prefix** expression using given algorithm. You need to just represent the required steps in a tabular form and no C++ code is required.

**Note:** $ is symbol for exponent. As you guys are **"super awesome intelligent"**, I would like to share a **"tricky"** statement that $ is solved from **right to left**. Now, what does it really mean! THINK ☺

**Infix Expression: A – B + C $ D $ E + F * G / H**

**Algorithm:**

```
opstk = the empty stack;
while (not end of input) {
        symb = next input character;
        if (symb is an operand)
                add symb to the postfix string
        else {
                while (!empty(opstk) &&
prcd(stacktop(opstk),symb) ) {
                   topsymb = pop(opstk);
                   add topsymb to the postfix string;
                } /* end while */
                push(opstk, symb);
        } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
        topsymb = pop(opstk);
        add topsymb to the postfix string;
} /* end while */
```

**Dry-Run**

Reverse of infix expression:
H / G * F + E $ D $ C + B – A

| symb | postfix | opstk |
|---|---|---|
| H | H | |
| / | H | / |
| G | HG | / |
| * | HG/ | * |
| F | HG/F | * |
| + | HG/F* | + |
| E | HG/F*E | + |
| $ | HG/F*E | +$ |
| D | HG/F*ED | +$ |
| $ | HG/F*ED | +$$ |
| C | HG/F*EDC | +$$ |
| + | HG/F*EDC$$+ | + |
| B | HG/F*EDC$$+B | + |
| - | HG/F*EDC$$+B+ | - |
| A | HG/F*EDC$$+B+A | - |
| | HG/F*EDC$$+B+A- | |

Postfix of reverse expression:
HG/F*EDC$$+B+A-

Reverse the postfix to get required prefix:
**-A+B+$$CDE*F/GH**

It is solved as:
(((H/G)*F+(E$(D$C))) + B) - A

**Question 5 [5x3 = 15 Marks]**

Some of you guys think that it is impossible to sort a queue, or to access any element in a given queue. However, a few still think that, given the primitive functions **(enqueue, dequeue, front, size, and IsEmpty) only,** we can access any element in a queue. They also believe that a queue can be sorted if you think out of the box. This is a confusing situation among students, so we should give it a try to reach a consensus!

As a challenge, you are required to **only use the primitive queue operations (enqueue, dequeue, front, size, and IsEmpty) to implement the methods requested**

Specifically, You need to provide pseudo-code for the following methods:

- Get the nth element from the front of the queue, leaving queue without front n elements.

```
Repeat (N-1) times
    DataQ.Dequeue()
int data = DataQ.Dequeue()
return data
```

- Get the nth element from the front of the queue, only removing the element requested (the nth element).

```
Repeat (N-1) times
    DataQ.Enqueue( DataQ.Dequeue() )
data = DataQ.Dequeue()
Repeat (Size - N) times
    DataQ.Enqueue( DataQ.Dequeue() )
return data
```

- Get the nth element from the rear of the queue, leaving queue without bottom n elements.

```
Int M = DataQ.size() – N
Repeat M times
    DataQ.Enqueue( DataQ.Dequeue() )
data = DataQ.Dequeue()
Repeat (N-1) times
    DataQ.Dequeue()
return data
```

- Get the nth element from the rear of the queue, removing only the required element.

```
Int M = DataQ.size() – N
Repeat M times
    DataQ.Enqueue( DataQ.Dequeue() )
data = DataQ.Dequeue()
Repeat (N-1) times
    DataQ.Enqueue( DataQ.Dequeue() )
return data
```

- Sort the queue based on the data value (an integer)

```
Int M = 0
Int N = DataQ.size()
Array = new int[N]
Repeat N times
    Array[M] = DataQ.Dequeue()
    M = M +1
QSort(Array, 0, N)
M=0
Repeat N times
    DataQ.Enqueue( Array[M])
    M = M +1
```

As a challenge, you are required to **only use the primitive queue operations (enqueue, dequeue, front, size, and IsEmpty) to implement the methods requested**