Department of Computer Science, UET Lahore
Final Term Exam

Session	2021	Semester	Fall 2022
Course	Data Structures and Algorithms	Registration No	Solution
Time	75 mins	Marks	40

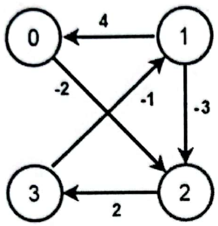
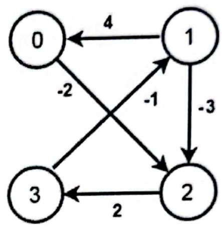
Obtained Marks (Leave this section empty)		
CLO1	CLO2	CLO3
/26	/6	/8

Instructions (Violation of Instructions always lead to useless effort)

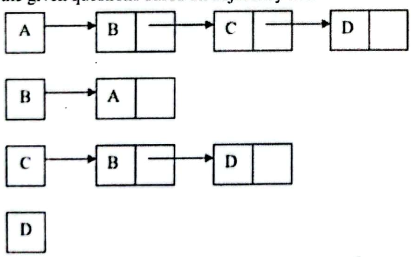
- All answers should be written in the spaces. No extra space will be provided.
- Write the answer only in the given space. Use the last page for rough work.
- Before you start, write your Registration Number and Name on Odd Pages.

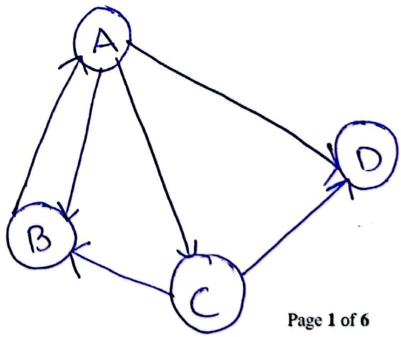
Graphs

Q1. Run the Dijkstra and Bellman-Ford Algorithm on the following graph and show steps in brief. show your execution on graph as well? **[CLO1][4 marks]**

Dijkstra Algorithm	Bellman Ford Algorithm
	
Dijstra could not be applied due to negative weights, we can convert to positive weights with addition of +3 to apply dijxtra	Student should show the execution of the algorithm and at the end of the execution, it should return false as Bellmanford detects negative cycle but cannot solve it.

Q2. Here is an adjacency list representation of a directed graph where there are no weights assigned to the edges) Complete the given questions based on adjacency list. **[CLO1][2+3+3 = 8 marks]**





a) Draw a picture of the directed graph that has the above adjacency list representation.(In provided space above)

b) Another way to represent a graph is an adjacency matrix. Draw the adjacency matrix for this graph.

	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

c) Calculate the G^2 of the above graph and draw adjacency matrix?

	A	B	C	D
A	1✓	1	1	1
B	1	1✓	1✓	1✓
C	1✓	1	0	1
D	0	0	0	0

✓ For G^2 edges

Trees

Q3. Write a recursive method to check whether the Binary Tree Node passed as a parameter is a Binary-Search-Tree. The method takes a Node (root of the Binary Tree) as a parameter and returns a (true or false) value. **[CLO1][4 marks]**

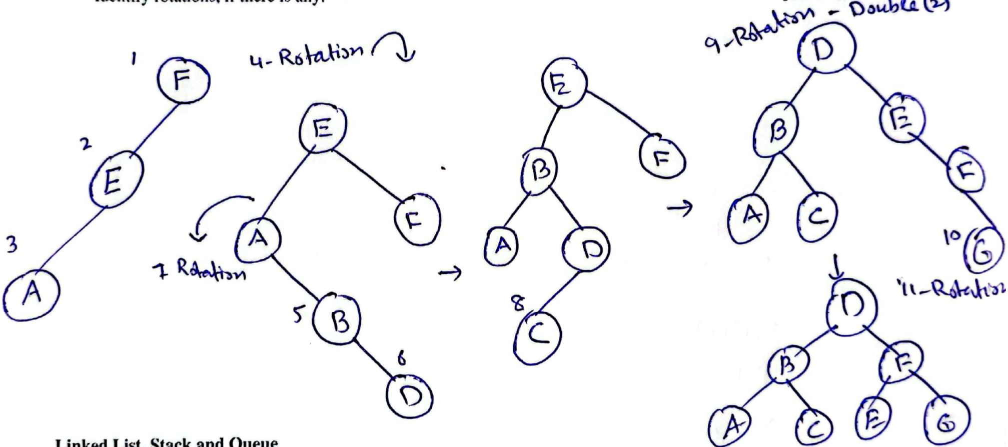
```
def isBSTUtil(root, prev):

    # traverse the tree in inorder fashion
    # and keep track of prev node
    if (root != None):
        if (isBSTUtil(root.left, prev) == True):
            return False

        # Allows only distinct valued nodes
        if (prev != None and
                root.data <= prev.data):
            return False

        prev = root
        return isBSTUtil(root.right, prev)

    return True


def isBST(root):
    prev = None
    return isBSTUtil(root, prev)
```

Q4. Give a non recursive algorithm that performs an inorder tree walk. (Hint: An easy solution uses a stack as an auxiliary data structure.). Give time complexity. **[CLO2][2+2 = 4 marks]**

```
void iterativeInorder(TreeNode root)
```

```
{
    if(root == NULL)
        return
    Stack<TreeNode> treeStack
    TreeNode currNode = root
    while (treeStack.empty() == false || currNode != NULL)
    {
        if (currNode != NULL)
        {
            treeStack.push(currNode)
            currNode = currNode->left
        }
        else
        {
            currNode = treeStack.pop()
            process(currNode->data)
            currNode = currNode->right
        }
    }
}
```

Q5. Draw the sequence of AVL trees obtained when the following keys are inserted one-by-one, in the order given into an initially empty AVL search tree: {F, E, A, B, D, C, G}. (The keys are comparable alphabetically, for example, F>E) Identify rotations, if there is any. **[CLO1][3 marks]**



Linked List, Stack and Queue

Q6. Given a list of objects stored in a sorted linked list, describe an algorithm to search as efficiently as possible for an object based on a key. (Note that the key type matches the field type by which the linked list is sorted.) Describe your algorithm in C++ code. **[CLO1][2 marks]**

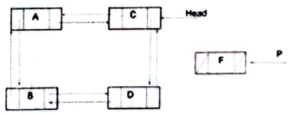
```
struct Node* binarySearch(Node *head, int value)
{
    struct Node* start = head;
    struct Node* last = NULL;
```

```
do
{
    // Find middle
    Node* mid = middle(start, last);

    // If middle is empty
    if (mid == NULL)
        return NULL;

    // If value is present at middle
    if (mid -> data == value)
        return mid;

    // If value is more than mid
    else if (mid -> data < value)
        start = mid -> next;

    // If the value is less than mid.
    else
        last = mid;

} while (last == NULL ||
         last != start);

// value not present
return NULL;
}
```

Time complexity is O(n) due to slow find of mid element.

Q7. The diagram below is a Circular-Doubly Linked List data structure. Provide the necessary code (only code for pointer adjustment) to add the new node pointed by P to the list. Node should be added between Node A and Node B. The link list should remain circular after your code. **[CLO1][2 marks]**

Let X = Node A
Y = Node B
P = Node F



P.Next = Y
P.Prev = X

Y.Prev = P
X.Next = P

Name: ______________________ Registration No. Solution

Q8. Assume an implementation of a STACK that holds integers. Write a method `void CountPosNeg (Stack S)` that takes Stack S as a parameter. After the call, the method prints the count of positive integers and negative integers on the stack. Note, the original stack must not be destroyed. **[CLO1][3 marks]**

```
public void CountPosNeg (Stack S) {
```

Take another stack Y.
CountPos = 0
CountNeg = 0
Check if stack S.peek()
contain positive then
Countpos++
otherwise
countNeg++

Add top element of S to Y. and S.pop()

when S.isempty() == true
then
Move all elements of Y to S
print countpos and countNeg

```
}
```

Fundamentals

Q9. Explain an efficient method to find the k-th smallest number in a set of n numbers (output: one number), without first sorting the n numbers, and discuss its complexity in terms of n and k. **[CLO2][2 marks]**

```
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1) {
        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);

        // If position is same as k
        if (pos - l == k - 1)
            return arr[pos];
        if (pos - l > k - 1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos - 1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}
```

Design of Data Structure **[CLO3][2+2+2+2= 8 marks]**

Q10. The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe, with the following operations:

MakeSet(x): create the set $\{x\}$.

Union(x, y): replace the set x is in (let's call it S) and the set y is in (let's call it S ') with the single set $S \cup S'$.

Find(x): return the unique ID for the set containing x (this is just some representative element of this set).

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element. We can now implement the operations as follows:

MakeSet(x): just set `x->head=x`. This takes constant time.

Find(x): just return `x->head`. Also takes constant time.

Union(x, y): To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other.

Let A be the list containing x and B be the list containing y, with lengths L_A and L_B respectively. Then we can do this in time $O(L_A + L_B)$ by appending B onto the end of A as follows. We first walk down A to the end, and set the final `next` pointer to point to `y->head`. This takes time $O(L_A)$. Next we go to `y->head` and walk down B, resetting head pointers of elements in B to point to `x->head`. This takes time $O(L_B)$.