


National University of Computer and Emerging Sciences, Lahore Campus

	Course Name:	Data Structures	Course Code:	CS2001
	Degree Program:	BS (CS, SE, DS)	Semester:	Fall 2022
	Exam Duration:	180 Minutes	Total Marks:	70
	Paper Date:	24-Dec-2022	Weight	40
	Section:	ALL	Page(s):	12
	Exam Type:	Final Exam		

Question No.	Q1	Q2	Q3	Q4	Q5	Q6	Total Marks
Question Marks	15	10	15	10	10	10	70
Obtained Marks							

Student Name: _____ Section: _____ Roll No. _____

Instruction/Notes:

Attempt all questions. Answer in the space provided. You can ask for rough sheets but will not be attached with this exam. **Answers written on rough sheet will not be marked.** Do not use pencil or red ink to answer the questions. In case of confusion or ambiguity make a reasonable assumption.

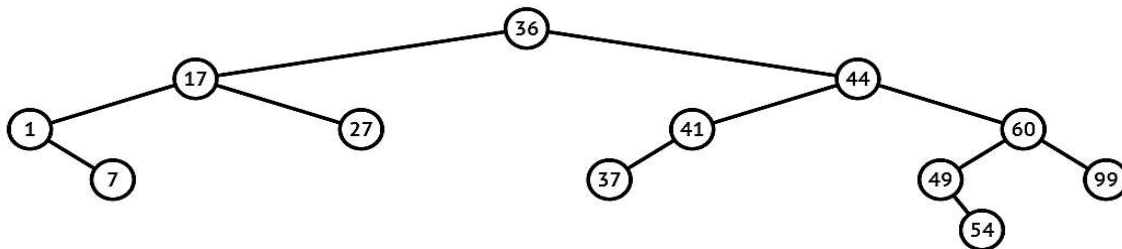
Question 1: [CLO 1, 2, 3]

(Marks: 2+3+10)

- a. What is the worst-case time complexity of inserting n^2 elements into an AVL-tree with n elements initially?
Explanation is also required.

Since AVL tree is balanced tree, the height is $O(\log n)$. So, time complexity to insert an element in an AVL tree is $O(\log n)$ in worst case. So $\Theta(n^2 \log n)$

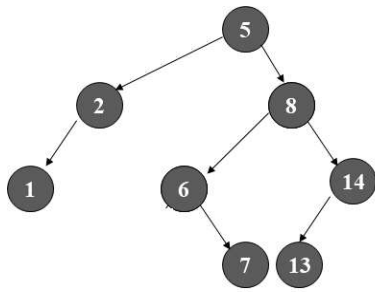
- b. Write inorder, preorder and post-order traversal of following tree.



Inorder	1 7 17 27 36 37 41 44 49 54 60 99
Preorder	36 17 1 7 27 44 41 37 60 49 54 99
Postorder	7 1 27 17 37 41 54 49 99 60 44 36

- c. Write a C++ function in the binary search tree (BST) class that takes an array of sorted integers (a sequence) as input parameters and finds whether the given sequence exists in the BST. You can write a wrapper function if you want to.**

Hint: In-order traversal, visit tree nodes in sorted order



Input: [2 5 6 7]

Returns: True

Input: [2 3 5 6]

Returns: False

Input [6 7 8 13 14]

Returns: True

Input [6 7 8 14]

Returns: False

```

void seqExistUtil(struct Node *ptr, int seq[], int &index) {
    if (ptr == NULL)
        return;
    // We traverse left subtree first in Inorder
    seqExistUtil(ptr->left, seq, index);

    // If current node matches with se[index] then move
    // forward in sub-sequence
    if (ptr->data == seq[index])
        index++;
    // We traverse left subtree in the end in Inorder
    seqExistUtil(ptr->right, seq, index);
}

// A wrapper over seqExistUtil. It returns true if seq[0..n-1] exists in tree.
bool seqExist(int seq[], int n)
{
    // Initialize index in seq[]
    int index = 0;
    // Do an inorder traversal and find if all elements of seq[] were present
    seqExistUtil(root, seq, index);
    // index would become n if all elements of seq[] were present
    return (index == n);
}
  
```

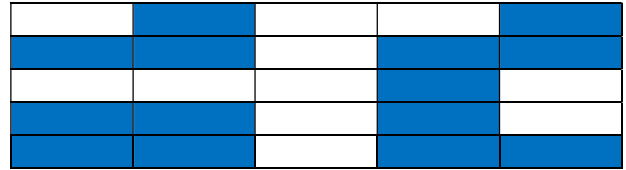
Question 2: [CLO 2, 3]

(Marks: 10)

NASA has captured an image of a distant area of planet earth where either land or water is visible. They want to count the number of islands in the captured image. An IT expert in team NASA has converted that image into a $n \times n$ binary matrix such that 0 represents earth and 1 represents water. Your task is to find the total number of islands in

the image given the nxn matrix. For example Given the following matrix, your answer must be three as there are three isolated land regions. The shaded regions are water in this example.

0	1	0	0	1
1	1	0	1	1
0	0	0	1	0
1	1	0	1	0
1	1	0	1	1



Hint: This problem can be solved by considering this matrix as a graph where each cell is a vertex and adjacent cells are neighbors of that vertex.

Write a C++ function that takes a nxn binary matrix as parameter and return the number of islands. Less efficient implementation will be awarded less credit.

```
int countIsland(int ** mat, int n){
    int count = 0;
    int ** visited = new int*[n];
    for(int i=0;i<n;i++){
        visited[i] = new int[n];
        for(int j=0;j<n;j++){
            visited[i][j] = 0;
        }
    }
    for(int i=0; i<n; i++){
        for(int j=0;j<n;j++){
            // count the connected land components that are not visited already
            if(mat[i][j] == 0 && visited[i][j] == 0){
                count++;
                DFS(mat, visited, i, j, n);
            }
        }
    }
    return count;
}

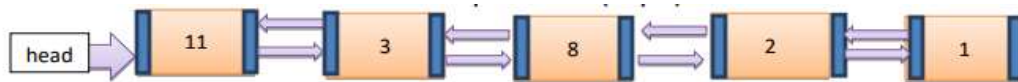
void DFS(int**mat, int** visited, int i, int j, int n){
    visited[i][j] = 1;
    for(int x=i-1; x<=i+1; x++){
        for(int y=j-1; y<=j+1; y++){
            //recursively call DFS for all neighbours that are land and not visited
            already
            if(mat[x][y] == 0 && 0<=x && x<n && 0<=y && y<n && visited[x][y]==0)
                DFS(mat, visited, x, y, n);
        }
    }
}
```

Question 3: [CLO 1, 3, 4]

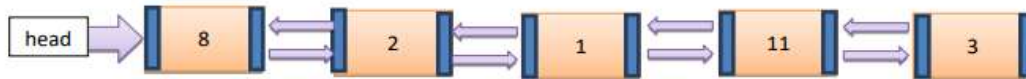
(Marks: 5+5+5)

- Add a function **moveHead(int position)** to the doubly linked list class. The function moveHead takes an integer position as input and moves the head pointer of the doubly linked list to the node at the specified position in the list. All the nodes before that specified position are shifted to the end of the list. Note that the first node in the list is at position 0.

The function can traverse the list only once, and no tail pointer is maintained. The prototype of the function is: `bool moveHead(int position)`. You have to give the C++ implementation of this function.



so after calling positionHead(2)



```

bool moveHead(int position) {
    // Check if the position is valid
    if (position < 0 || position >= size) return false;

    // If the position is already the head, do nothing
    if (position == 0) return true;

    // Find the node at the specified position
    Node* curr = head;
    for (int i = 0; i < position; i++) {
        curr = curr->next;
    }
    Node* tail=curr;
    while(tail!=NULL){
        tail=tail->next;
    }

    // Update the head and tail pointers
    tail->next = head;
    head->prev = tail;
    head = curr;
    tail = curr->prev;
    curr->prev->next = nullptr;
    curr->prev = nullptr;

    return true;
}
  
```

- b. A school's cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches. The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a **stack**.

At each step:

- i. If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.

- ii. Otherwise, they will leave it and go to the queue's end.
- iii. This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays **students** and **sandwiches** where *sandwiches[i]* is the type of the *i*th sandwich in the stack (*i* = 0 is the top of the stack) and *students[j]* is the preference of the *j*th student in the initial queue (*j* = 0 is the front of the queue). You have to count *the number of students that are unable to eat*. What should be the output if *students* = [1,1,1,0,0,1], *sandwiches* = [1,0,0,0,1,1]?

Show your working (states of queue and stack).

```
class Solution {
public:
    int countStudents(vector<int>& students, vector<int>& sandwiches) {
        int cnt = 0, x = 0, S = students.size();
        queue<int> q; stack<int> s;

        //create stack and queue
        for(int i=0; i < S; i++) {
            q.push(students[i]);
            s.push(sandwiches[S-1-i]);
        }

        while(true) {

            //update queue and stack
            if(q.front() == s.top()) {
                cnt++;
                q.pop();
                s.pop();
            } else {
                q.push(q.front());
                q.pop();
            }

            //all served?
            if(q.empty()) return 0;

            // if the food stack doesn't change size
            // after n (queue size) times: we are done
            x = (s.size() == S) ? x+1 : 1;
            if(x > q.size()) break;
            S = s.size();

        }

        return students.size() - cnt;
    }
};
```

3 students were left

- c.** Given a string *S* of lowercase English letters, the **duplicate removal function** finds two adjacent and equal letters and removes them. This function repeatedly makes duplicate removals on *S* until it no longer can. For example, for input *S* = "abbaca" the output is: "ca." In "abbaca," first, the function removes "bb" as the letters are adjacent and equal. Initially, this is the only possible move; the result of this move is the string "aaca". Next, remove "aa," and the final string is "ca."

Your task is to use the Stack data structure to solve this problem in $O(n)$ time, where n is the size of the string. Write the pseudocode for the above problem, and it should not be more than 4-5 steps (sentences).

- Keep res as a character's stack.
- Iterate characters of S one by one
- If the next character is same as the last character in res,
 - pop the last character from res.
 In this way, we remove a pair of adjacent duplicates characters.
- If the next character is different,
 - we append it to the end of res.

C++

```
string removeDuplicates(string S) {
    string res = "";
    for (char& c : S)
        if (res.size() && c == res.back())
            res.pop_back();
        else
            res.push_back(c);
    return res;
}
```

Question 4: [CLO 1, 2]

(Marks: 2+3+5)

- a. A priority queue of strings is implemented using heap with the following elements, which property of these strings is used to determine the priority of queue? Show updated contents of array after insertion of string “subterranean”.

Heapify: subconscious:12 characters

Subterranean: 12 suppose 11 index represent it when we insert it

subliminal: 10

1 represents subconscious and so on....

```

          1
        2   3
      4   5   6   7
    8   9  10  11
```

11 swaps 5

11 swaps 2

1 11 3 4 2 6 7 8 9 10 5

Priority: Length of words (Max Heap)

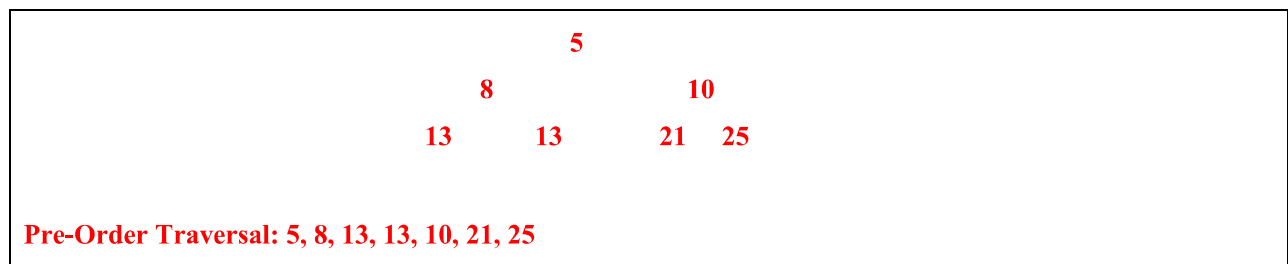
Current Array

Updated Array

1	“subconscious”
2	“subliminal”
3	“subordinate”
4	“submerge”
5	“subjugate”
6	“subscribe”
7	“subway”
8	“subject”
9	“sub”
10	“subset”
11	

1	“subconscious”
2	“ subterranean ”
3	“subordinate”
4	“submerge”
5	“ subliminal ”
6	“subscribe”
7	“subway”
8	“subject”
9	“sub”
10	“subset”
11	“ subjugate ”

- b. A person claims that Preorder traversal of a min heap will always print the keys in nondecreasing order? Draw a min heap with exactly these seven nodes (5, 8, 10, 13, 13, 21, 25) that proves him wrong.

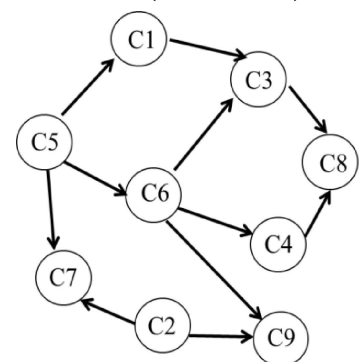


Question 5: [CLO 1, 2, 4]

(Marks: 4+6)

Consider the following directed and unweighted graph in which the vertices of the graph represent different courses and edges represent the prerequisite relation between these courses.

- a. Run the DFS on this graph, and output the nodes visited starting from node C5. Break the tie in numerical order.



Traversal order: 5 1 3 8 7 6 4 9

- b. How will you find all those courses that have no prerequisite courses? Which storage scheme works best for this operation: Adjacency Matrix or Adjacency List? Explain your answer.

For this operation Adjacency Matrix is best option.

Find the indegree of each node by taking the column sum of that node. If indegree is zero then that node has no prerequisites.

We need to traverse whole Adjacency list for finding the indegree of each node.

Question 6: [CLO 1]

(Marks: 10)

Cuckoo hashing is a scheme for resolving collisions of values in a hash table, with worst-case constant lookup time. The name derives from the behavior of of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches. Cuckoo hashing uses two hash tables, **T1** and **T2** for storage of keys each more than half empty. It uses two independent hash functions **hf1** and **hf2** that can assign each item to a position in each table T1 and T2 respectively.

Following algorithm is used for **insertion** of a new key.

Check if the slot at index **i** = **hf1(k1)** is empty in table **T1**

- a) If yes then insert the new-key **k1** in table **T1** at index **i**
- b) Otherwise, remove the previously inserted old-key **k2** from index **i** of the table **T1** and insert **k1** there.
 - Search for the new position of removed **k2** in table **T2** by using **i2=hf2(old_key)**
 - If index **i2** of table **T2** is empty then insert **k2** there, otherwise remove already inserted old-key **k3** from index **i2** of table **T2** and insert **k2** there.
 - Continue the same search procedure for key **k3** in the table **T1** as already done for **k1**.

Consider the following Hash tables, T1 and T2 of same sizes.

Hash function for table T1 is simple mod functions **i** = **hf1(key % T1_size)**

Hash function for table T2 is **i2= hf2(reverse_key % T2_size)** which reverses the key first and then take mod with table size T2.

	0	1	2	3	4
T1	25	51		43	

	0	1	2	3	4
T2	55		23		41

Insert the following two-digit keys in these hash tables T1 and T2 using cuckoo hashing and show the updated tables. You should also provide the total number of removals done for insertion of each key.

Keys	Removals Made by key for other keys	Keys removed count
25	0	2
55	1	1
41	0	1
51	1	0
23	0	1
43	3	0

Rough Sheet

Rough Sheet

Rough Sheet

Rough Sheet