Answer Script

Question No. 01

1. Write the time complexity of the following code segments with proper explanation. **10**

```
void fun(int l,int r)
{
        int mid = (l+r)/2;
        for(int i = l ; i <= r ; i++)
        {
                cout<<i<<endl;
        }
        if(l<r){
                fun(l,mid);
                fun(mid+1,r);
        }
}

int main()
{
        int n;
        cin>>n;
        fun(0,n-1);
}
```

```
for(int i = 1 ; i <= n/2 ; i++)
{
        for(int j = 1 ; j <= n ; j = j + i)
        {
                cout<<i<<" "<<j<<endl;
        }
}
```

Answer No. 01

First Code:

The time complexity of the given code is O(n*logn).
The fun function takes arguments l and r.
The time complexity of the for loop is O(n). This loop runs l to r times. So, for loop time complexity is O(n).

Inside the if statement fun function calls twice. 1st time fun function called with l and mid arguments. So, its time complexity is O(n/2).
2nd time fun function called with mid+1 and r arguments. So, its time complexity is O(n/2).

In the if statement the fun function is called twice. So, if statement total time complexity is: 2*O(n/2) or O(n/2).
If statement fun function called recursively. Every time summation of the l and r divide by 2.

So, Time complexity for this code is O(n*logn).

Second Code:
The time complexity of the given code is O(n*logn).

This code has two for loops. Inner loop and outer loop.
Outer loop runs n/2 times. So, its time complexity is O(n/2).
In the Inner loop j starts 1 and j values increase by i until j is less than or equal n.

When,
i = 1:
The inner loop runs n times. So, for i = 1 time complexity of the inner loop is O(n).
i = 2:
The inner loop runs n/2 times. So, for i = 2 time complexity of the inner loop is O(n/2).
i = 3:
The inner loop runs n/3 times. So, for i = 3 time complexity of the inner loop is O(n/3).
i = 4:
The inner loop runs n/4 times. So, for i = 4 time complexity of the inner loop is O(n/4).

So, we can see here every time complexity depends on the i's value.

Time complexity:
n + n/2 + n/3 + n/4 + …….. + n/n
This is the harmonic series.
n + n/2 + n/3 + n/4 + …….. + n/n
= n * (1 + 1/2 + 1/3 + 1/4 + …….)
= n
So, Time complexity of this code is: O(n*logn).

Question No. 02

2. Suppose you are implementing a linked-list where you want to maintain a floating point number and a character in each node. Each node will contain a next pointer and also a next_to_next pointer that will keep track of the node that is next to the next node. What will the node class look like? **10**

```
class Node{
        // write your variables
};
```

Answer No. 02

```
class Node{
public:
   float value;
   char ch;
   Node* next;
   Node* next_to_next;

   Node(float value, char ch)
   {
      this->value = value;
      this->ch = ch;
      next = NULL;
      next_to_next = NULL;
   }
};
```

Question No. 03

3. Write the main difference between linear and non-linear data structures. Compare between Stack, Queue and Deque. Are stack, queue, deque linear or non-linear data structure? What about a tree? **10**

Answer No. 03

The main difference between linear and non-linear data structures:

Linear Data Structures	Non Linear Data Structures
In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remains with increase in size.
It is not very memory-friendly. It means that the linear data structures can't utilise memory very efficiently. The data structure is memory-friendly.	The data structure is memory-friendly. It means that it uses memory very efficiently.
Array, List, Queue, Stack.	Graph, Map, Tree.

Compare between Stack, Queue and Deque:

Stack	Queue	Deque
A stack is a linear data structure in which elements are inserted and removed from the same end.	A queue is a linear data structure in which elements are inserted at one end and removed from the other end.	A deque is a linear data structure in which elements can be inserted or removed from both ends.
It follows the Last-In-First-Out (LIFO) principle, which means that the most recently added element is the first one to be removed.	It follows the First-In-First-Out (FIFO) principle, which means that the first element that was added is the first one to be removed.	It supports the LIFO and FIFO principles, and can be used to implement stacks and queues.
Stacks are commonly used	Queues are used to	Deques are used when you

to keep track of function calls.	implement process scheduling.	need access to both ends of the data structure.
Insertion in stacks takes place only from one end of the list called the top.	Insertion can be done through the rear end only.	Insertion is possible through both ends.
Deletion in stacks takes place only from one end of the list called the top.	Deletion of elements is possible through the front end only.	Deletion of elements possible through both ends.

Stack, Queue and Deque are linear data structures.

Tree: A tree data structure is a hierarchical structure that is used to represent and organise data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

Question No. 04
4. Between singly linked list and doubly linked list which is better for implementing Stack and Queue? What about Deque? **10**
Answer No. 04

Both singly linked lists and doubly linked lists can be used to implement stacks and queues.

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail). Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

A doubly linked list is a type of linked list in which each node contains two pointers, one that points to the next node in the list, and one that points to the previous node in the list.

In a single linked list, each node has a pointer to the node after it but not the node before it. Singly linked lists use less memory per node because they simply need to retain a pointer to the following node. Nevertheless, processes like popping the final item off of a stack or queue that need access to the prior node cannot be supported by singly linked lists in an effective manner. This indicates that using a singly linked list to implement a stack or queue may involve more operations and may be less effective than using a doubly linked list

.

Each node in a doubly linked list contains a pointer to both the node before it and the node after it. This makes it simple to create operations that need access to both the previous and next nodes, such popping the final item off of a stack or queue, and it enables efficient traversal in both directions. Nevertheless, because references to the previous and following nodes must be stored, doubly linked lists consume extra memory per node.

Deque:

A deque is a data structure that allows insertion and deletion of elements from both ends in constant time. It can be visualised as a combination of a queue and a stack, where elements can be added or removed from either the front or the back of the deque.

Deques can be implemented using linked lists or arrays. The circular buffer used by the deque in the array implementation enables quick addition and removal of elements from either end. Each node in the linked list implementation has a pointer to the node before it and the node after it in the deque.

Question No. 05

5. Convert the infix expression to postfix expression using a stack. You need to show all the steps. **10**

a*b+c*d+e

Answer No. 05

First input this line as a string. Then take one stack st and string ans.

1. run a for loop to length of string.
2. In the string the 1st character is an alphabet. So, this character(a) add ans string.
   ans = “a”
3. Then the next character is operator. So, push is st stack.
   ans = “a”, st = {*}
4. Then the next character is an alphabet. So, this character(b) add ans string.
   ans = “ab”, st = {*}
5. Then the next character is operator. Its ‘+’ operator. ‘+’ operator is less than the st stack top value. So, st stack top value add ans string and pop the st stack top value. Then push ‘+’ operator in the st stack.
   ans = “ab*”, st = {+}.
6. Then the next character is an alphabet. So, this character(c) add ans string.
   ans = “ab*c”, st = {+}.
7. Then the next character is operator. Its ‘*’ operator. ‘*’ operator is greater than the st stack top value. So, push the ‘*’ operator in the st stack.
   ans = “ab*c”, st = {+, *}.
8. Then the next character is an alphabet. So, this character(d) add ans string.
   ans = “ab*cd”, st = {+, *}.
9. Then the next character is operator. Its ‘+’ operator. ‘+’ operator is less than the st stack top value. So, st stack top value add ans string and pop the st stack top value.
   ans = “ab*cd*”, st = {+}.
   Now, ‘+’ operator is equal to st stack top value. So, add st stack top value in the string and pop it from the st stack. Then push ‘+’ operator in the st stack.
   ans = “ab*cd*+”, st = {+}.
10. Then the next character is an alphabet. So, this character(e) add ans string.
    ans = “ab*cd*+e”, st = {+}.
11. Now, input string all character iteration is complete but stack is not empty so its top value add in the ans string and pop it from the st stack.
    ans = “ab*cd*+e+”, st = {}.

Postfix is : “ab*cd*+e+”

Question No. 06

6. Compare the memory usage of Array, Singly Linked-list and Doubly Linked-list with necessary explanation. **10**

Answer No. 06

Array:

An array is a collection of elements of the same data type, stored in contiguous memory locations. The memory usage of an array is proportional to the size of the array and the size of each element. When an array is declared, the memory required to hold the entire array is allocated in advance. The memory used by an array is typically measured in bytes, and it can be calculated as the product of the number of elements in the array and the size of each element.

When 20 integers are declared in an array, for instance, the memory space needed to store 20 integers is allocated all at once. For 32-bit and 64-bit systems, respectively, each integer in the array typically takes up 4 bytes and 8 bytes of memory. As a result, both the array's size and the number of elements affect how much memory an array uses.

Single Linked List:

In a singly linked list typically consists of two parts: the data part and the pointer part. The data part holds the value of the node, while the pointer part holds the reference to the next node in the list. The size of each node is determined by the size of the data and the size of the pointer, which is typically the same as the size of an integer. The memory usage of a singly linked list is proportional to the number of nodes in the list, as well as the size of each node. Singly linked lists do not support efficient access to the last node in the list or the ability to traverse the list in reverse order.

Doubly Linked List:

Each node in a doubly linked list typically consists of three parts: the data part, the next pointer part, and the prev pointer part. The data part holds the value of the node, while the next and prev pointer parts hold the references to the next and previous nodes in the list, respectively. The size of each node is determined by the size of the data and the size of each pointer, which is typically the same as the size of an integer. A doubly linked list's memory consumption is inversely correlated with the number of nodes and size of each node. Due to the additional prior pointer field in each node, doubly linked lists use a little bit more memory than singly linked ones.

Question No. 07

7. Suppose you are implementing a stack in a scenario where numbers are added in sorted order so that the stack is always sorted. Sometimes you need to quickly search if a value exists in the stack or not. Array or Linked-list which implementation for stack will you prefer in this scenario? Give necessary explanations. **10**

Answer No. 07

In the scenario where a stack is always sorted and sometimes requires quick search operations, an array-based implementation of the stack would be preferred over a linked list implementation.

This is because searching an array for a particular value is much faster than searching a linked list because arrays provide effective random access to their elements. If a linked list were used to implement the stack, a search operation would need to traverse the list from top to bottom, which can take a long time for large stacks.

As there is no need for additional memory for pointers between nodes, an array-based version of the stack uses less memory than a linked-list counterpart. When memory utilisation is a concern, this can be a crucial factor to take into account.

Consequently, an array-based implementation of the stack would be more effective and efficient than a linked-list version in the scenario when a stack is always sorted and speedy search operations are required.

Question No. 08

8. Suppose you are maintaining a head and tail for a singly linked-list. What will be time complexity of **10**
    a. Inserting a value at the beginning
    b. Inserting a value at the end
    c. Deleting a value at the beginning
    d. Deleting a value at the end
    e. Inserting a value at the mid point
    f. Deleting a value at the mid point

Answer No. 08

a. The time complexity of Inserting a value at the beginning in the single linked list is O(1).
b. If we know the tail pointer of the single linked list then the time complexity of Inserting a value at the end in the single linked list is O(1) otherwise the time complexity O(n).
c. The time complexity of Deleting a value at the beginning in the single linked list is O(1).
d. The time complexity of Deleting a value at the end in the single linked list is O(n).
e. If we know the total number of nodes then the time complexity of Inserting a value at the mid point in the single linked list is O(1) otherwise the time complexity of Inserting a value at the mid point in the single linked list is O(n).
f. If the position of the node to be deleted is not known then the time complexity will be O(n) otherwise O(1).

Question No. 09

9. Consider the following binary tree in **Fig 1** (node 20 is the root) and answer the given questions. **10**

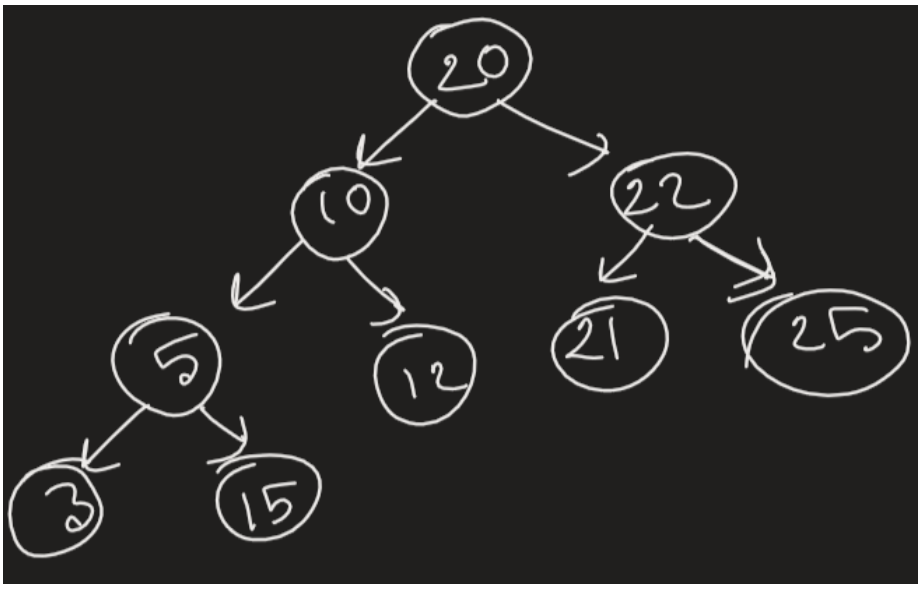

Fig: 1

a. Is the tree a Perfect binary tree? Why or why not?
b. Is the tree a Complete binary tree? Why or why not?
c. Is the tree a Binary search tree? Why or why not?
d. Write down the BFS, inorder, preorder and postorder traversal of the tree.

Answer No. 09

a. No, The tree is not a Perfect binary tree.
A perfect binary tree is a type of binary tree where all the internal nodes have two children and all the leaf nodes are at the same level.
A perfect binary tree of height h has 2^(h+1)-1 nodes.
In this case, the tree has 4 levels, so its height is 3. Therefore, a perfect binary tree with height 3 would have 2^(3+1)-1 = 15 nodes.
Since the given tree has only 9 nodes, it is not a perfect binary tree.

b. Yes, The tree is a Complete binary tree.
A complete binary tree is a binary tree in which all levels except possibly the last are completely filled, and all nodes are as far left as possible.
In this tree, all levels except the last are completely filled and all nodes in the last level are as far left as possible. Therefore, it is a complete binary tree.

c. Yes, the tree is a binary search tree.
Binary Search Tree Condition:

i. The values of all the nodes in the left subtree are less than the value of the node.
ii. The values of all the nodes in the right subtree are greater than the value of the node.
iii. There are no duplicate values in the tree.

Here, the root value is 20.

1. The root value(20) left subtree(10) is less than the root value and right subtree(22) is greater than the root value.
2. Now, the root value is 10. The root value(10) left subtree(5) is less than the root value and the right subtree(12) is greater than the root value.
3. Now, the root value is 5. The root value(5) left subtree(3) is less than the root value and the right subtree(15) is greater than the root value.

   Main root value(20) left all subtree are fill up BST condition now check right subtree.

4. Now, the root value is 20. Its right subtree is greater than(22) the root value.
5. Now, the root value is 22. Its left subtree(21) is less than the root value and right subtree(25) is greater than the root value. Finally the left and right tree has no child.

   It fills up the Binary Search Tree in all conditions. So, we can say it is a binary search tree.

d. BFS Traversal: {20, 10, 22, 5, 12, 21, 25, 3, 15}
Inorder Traversal: {3, 5, 15, 10, 12, 20, 21, 22. 25}
Preorder Traversal: {20, 10, 5, 3, 15, 12, 22, 21, 25}
Postorder Traversal: {3, 15, 5, 12, 10, 21, 25, 22, 20}

Question No. 10

10. Write the steps to insert **70** in the following binary search tree in **Fig 2** (node 50 is the root).

10

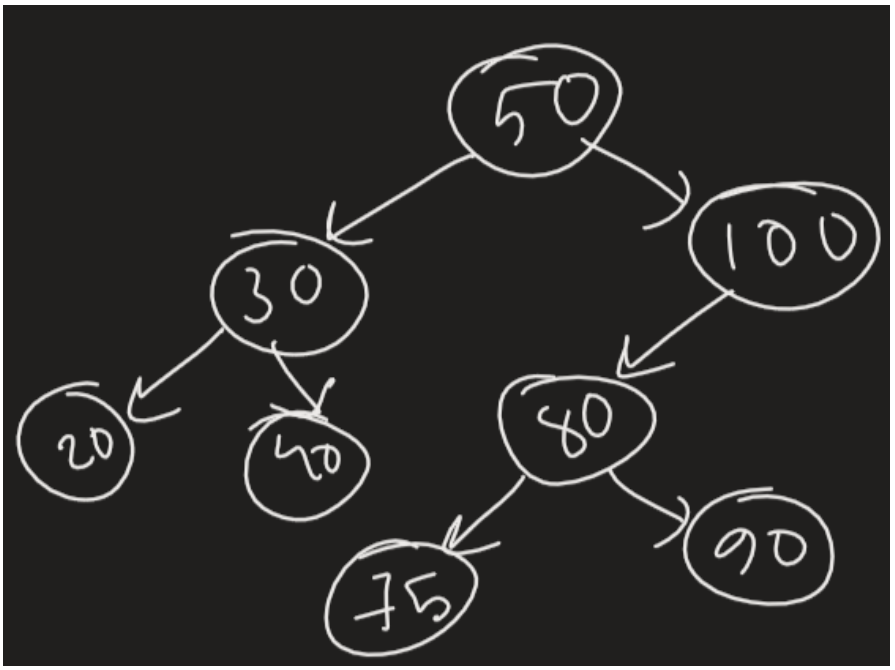

Fig: 2

Answer No. 10

The steps to insert 70 in the following binary search tree:
Here, the root value is 50.

1. Compare 70 with the root value. Here, we can see 70 is greater than the root value. So, go right subtree.
2. Now the root value is 100. Compare 70 with the root value. We can see 70 is less than the root value. Now, go to the left subtree.
3. Now the root value is 80. Compare 70 with the root value. We can see 70 is less than the root value. Now, go to the left subtree.
4. Now the root value is 75. Compare 70 with the root value. We can see 70 is less than the root value. Now, go to the left subtree.
5. The root value left subtree is NULL. So, now insert 70 in the 75 root left subtree.