ANSWER - 1

a. The time complexity of the given code is O(n logn), where 'n' is the value entered by the user in the main function.
The code divides the range [l, r] into two halves and recursively calls the 'fun' function for each half until the range is reduced to single elements.
For each call of the 'fun' function, a 'for' loop is executed that iterates over the range from 'l' to 'r'. The time complexity of the 'for' loop is O(n), where 'n' is the size of the range.
The code divides the range [l, r] into two halves, and each half is processed separately by calling the 'fun' function recursively. The time complexity of each recursive call is O(logn), as the size of the range is halved at each level of recursion.
Therefore, the overall time complexity of the code is the product of the time complexity of the 'for' loop and the number of recursive calls. The number of recursive calls is O(logn), and the time complexity of the 'for' loop is O(n). Hence, the overall time complexity of the code is O(n logn).

b. The time complexity of the given code is O(n*log(n)), where 'n' is the value of the input parameter.
The outer 'for' loop iterates 'n/2' times. The inner 'for' loop iterates 'n/i' times, where 'i' is the current value of the outer loop variable. Therefore, the total number of iterations of the inner 'for' loop is a harmonic series that sums up to approximately log(n).
Therefore, the time complexity of the inner 'for' loop is O(log(n)).

The time complexity of the 'cout' statement inside the inner 'for' loop is constant, O(1).

So, the overall time complexity of the code is O(n log(n)), as the inner 'for' loop has a time complexity of O(log(n)) and it is executed n/2 times by the outer 'for' loop.

ANSWER - 2

The node class for the linked-list that maintains a floating point number, a character, a next pointer, and a next_to_next pointer in each node can be defined as follows:

```
class Node{
public:
        float float_data;
        char char_data;
        Node* next;
        Node* next_to_next;
};
```

Each node has two data members, float_data and char_data, which store the floating point number and the character, respectively. The next and next_to_next pointers are used to connect the nodes in the linked-list.
This class definition allows us to create a linked-list where each node stores a floating point number and a character, and is connected to the next and next-to-next nodes.

ANSWER - 3

The main difference between linear and non-linear data structures is the way data elements are stored and accessed. In linear data structures, the elements are stored in a linear sequence, one after the other, while in non-linear data structures, the elements are stored in a hierarchical structure, where each element may have one or more child elements.

Stack, Queue, and Deque are all linear data structures, but they differ in the way elements are added and removed from the structure.

a. A Stack follows the Last-In-First-Out (LIFO) principle, where the last element added to the structure is the first one to be removed.

b. A Queue, follows the First-In-First-Out (FIFO) principle, where the first element added to the structure is the first one to be removed.

c. A Deque, or Double-Ended Queue, allows elements to be added and removed from both ends of the structure. This makes it more flexible than a Stack or a Queue.

A tree is a non-linear data structure, as the elements are arranged in a hierarchical structure, with one or more parent nodes and zero or more child nodes. Trees are commonly used in applications such as file systems, database indexing, and representing hierarchical relationships between objects. Unlike linear data structures, trees can have multiple branches and can be unbalanced, where one subtree may have more child nodes than another subtree.

ANSWER - 4

Both singly linked lists and doubly linked lists can be used to implement Stacks and Queues, but doubly linked lists are typically better suited for these data structures.

For a Stack, a doubly linked list provides the advantage of allowing both push and pop operations to be performed in constant time, O(1), by simply updating the head or tail pointer. One the other hand, a singly linked list would require O(n) time to update the tail pointer when performing a push operation.

For a Queue, a doubly linked list provides the advantage of allowing both enqueue and dequeue operations to be performed in constant time, O(1), by simply updating the head or tail pointer. But, a singly linked list would require O(n) time to update the head pointer when performing a dequeue operation.

For a Deque, both singly and doubly linked lists can be used, but a doubly linked list is more efficient because it allows constant-time access to both ends of the

deque. This means that enqueue, dequeue, push, and pop operations can be performed in constant time by simply updating the head or tail pointers, or both.

So, Overall, a doubly linked list is a better choice for implementing a Stack, Queue, or Deque, because it provides constant-time access to both ends of the data structure.

ANSWER - 5

To convert the infix expression "a*b+c*d+e" to postfix expression using a stack the steps is shown below :

1. Create an empty stack and an empty output queue .
2. Scan the infix expression from left to right.
3. If the current character is an operand, add it to the output queue.
4. If the current character is an operator, then:
    a. While there is an operator at the top of the stack with higher or equal precedence, pop the operator from the stack and add it to the output queue.
    b. Push the current operator onto the stack.
5. If the current character is a left parenthesis, push it onto the stack.
6. If the current character is a right parenthesis, then:
    a. Pop operators from the stack and add them to the output queue until a left parenthesis is encountered.
    b. Pop the left parenthesis from the stack and discard it.
7. After scanning the infix expression, pop any remaining operators from the stack and add
    them to the output queue.
8. The postfix expression is the contents of the output queue in order.

STEPS:

STEPS	Current symbol	Output queue	Stack
Step 1	a	a	Empty

Step 2	*	a	*
Step 3	b	ab	*
Step 4	+	ab*	+
Step 5	c	ab*c	+
Step 6	*	ab*c	+*
Step 7	d	ab*cd	+*
Step 8	+	ab*cd*	+
Step 9	e	ab*cd*e	+
Step 10	End of input	ab*cd*e+	Empty

The final postfix expression is "ab*cd*e+"

ANSWER - 6

The memory usage of data structures can vary depending on the specific implementation and the size and type of data being stored. However, in general, here is a comparison of the memory usage of array, singly linked-list, and doubly linked-list.

1. **Array:** An array is a collection of elements of the same data type that are stored in contiguous memory locations. The size of the array is fixed and defined at the time of its creation. The memory usage of an array is generally low because the elements are stored in contiguous memory locations. The memory usage of an array can be high if the array is not fully utilized or if the size of the array needs to be increased dynamically.

2. **Singly Linked-list:** In a singly linked-list, each element (node) contains the data and a pointer to the next node. The memory usage of a singly linked-list is generally higher than an array because each node requires additional

memory for the pointer to the next node. Traversing a singly linked-list can require more memory accesses compared to an array, since the nodes may not be stored in contiguous memory locations.

3. **Doubly Linked-list:** In a doubly linked-list, each element (node) contains the data and pointers to both the previous and next nodes. The memory usage of a doubly linked-list is generally higher than a singly linked-list because each node requires additional memory for the pointer to the previous node. Ttraversing a doubly linked-list can be faster than a singly linked-list because nodes can be accessed from both the previous and next pointers.

So, the choice of data structure depends on the specific requirements of the application. Arrays are a good choice when a fixed-size collection of elements is needed, and fast random access to elements is important. Singly linked-lists are a good choice when memory is limited and only sequential access is needed. Doubly linked-lists are a good choice when fast sequential access in both forward and backward directions is needed.

ANSWER - 7

In this scenario where the stack is always sorted, an array implementation for the stack would be preferable for faster searching, compared to a linked-list implementation.

The reason for this is that an array allows for random access to elements based on their index, whereas in a linked-list, searching for an element requires iterating through the list from the beginning until the desired element is found. This can take a longer time, especially if the element being searched for is near the end of the list.

Using an array, searching for an element can be done efficiently using binary search, which takes O(log n) time, where n is the number of elements in the array. Binary search involves dividing the array in half at each step and comparing the

desired element with the middle element. This process is repeated until the desired element is found or it is determined that the element does not exist in the array.

Therefore, a linked-list does not allow for efficient binary search, and even other search algorithms like linear search would take O(n) time, where n is the number of elements in the linked-list.

So, for a stack implementation where the numbers are always sorted, an array implementation would be preferable for faster searching. However, care must be taken to ensure that the array is resized when it becomes full to avoid overflow errors.

ANSWER - 8

Assuming that the singly linked-list has n nodes, the time complexity for the operations you mentioned would be:

A. **Inserting a value at the beginning: O(1)** This operation involves creating a new node, updating the next pointer of the new node to point to the current head, and updating the head pointer to point to the new node. All of these operations take constant time.

B. **Inserting a value at the end: O(1)** This operation involves creating a new node, updating the next pointer of the current tail to point to the new node, and updating the tail pointer to point to the new node. All of these operations take constant time.

C. **Deleting a value at the beginning: O(1)** This operation involves updating the head pointer to point to the next node of the current head. This operation takes constant time.

D. **Deleting a value at the end: O(n)** This operation requires traversing the entire linked-list to find the second to last node and update its next pointer to

NULL. Since this operation involves traversing the entire linked-list, the time complexity is O(n).

E. **Inserting a value at the mid point: O(n)** This operation involves traversing the linked-list to find the midpoint and inserting the new node after the midpoint node. Since this operation requires traversing half of the linked-list, the time complexity is O(n/2) which is equivalent to O(n).

F. **Deleting a value at the mid point: O(n)** This operation involves traversing the linked-list to find the midpoint and updating the next pointer of the node before the midpoint to point to the node after the midpoint. Since this operation requires traversing half of the linked-list, the time complexity is O(n/2) which is equivalent to O(n).

ANSWER - 9

A. A perfect binary tree is a type of binary tree where all internal nodes have exactly two children, and all the last nodes are at the same level or depth. Or, a perfect binary tree is a binary tree where every level is completely filled with two child nodes and finish with two child in the same height/level.

The binary tree represented in the Fig : 1, is not a perfect binary tree because it has finishing nodes which are not at the same level. Specifically, the node with value 5 has only two children (3 and 15), while the nodes with values 12, 21, and 25 don't have any child nodes.
Therefore, the binary tree represented in the Fig : 1, is not a perfect binary tree.

B. A complete binary tree is a binary tree where all levels, except the last level, are completely filled with nodes, Or, a complete binary tree is a binary tree where every level is completely filled, except for the last level.

From Fig : 1, we can see that, all levels of the tree, except the last level, are completely filled with nodes. In this tree, the first two levels are completely

filled with nodes, and the third level has two nodes (3 and 15), which are the only child of their parent 5 node, if we ignore 5 node's child nodes. Its all level are filled up.
Therefore, the binary tree represented in the Fig : 1, is a Complete binary tree.

C. A binary search tree (BST) is a binary tree in which all the nodes in the left subtree of a node have values less than the node's value, and all the nodes in the right subtree have values greater than the node's value.

   The binary tree shown in Fig : 1, is not a binary search tree. While some of the nodes in the tree satisfy the conditions of a BST, But, the subtree rooted at the node with value 5 has a right child which is node15 with a value greater than its parent node which value is 10.
   Therefore, the binary tree represented in the Fig : 1, is not a binary search tree.

D. The BFS traversal of the tree is: 20 10 22 5 12 21 25 3 15
   The inorder traversal of the tree is: 3 5 15 10 12 20 21 22 25
   The preorder traversal of the tree is: 20 10 5 3 15 12 22 21 25
   The postorder traversal of the tree is: 3 15 5 12 10 21 25 22 20

ANSWER - 10

When inserting a new node in a binary search tree, we should follow the rule that any node in the left subtree should have a value less than the parent node, and any node in the right subtree should have a value greater than the parent node.

So the steps will be :

1. Compare the value of the node to be inserted (70) with the value of the root node (50). Since 70 is greater than 50, we move to the right child of the root.

2. Compare the value of the node to be inserted (70) with the value of the current node (which is now 100). Since 70 is less than 100, we move to the left child of the current node.
3. Compare the value of the node to be inserted (70) with the value of the current node (which is now 80). Since 70 is less than 80, we move to the left child of the current node.
4. Compare the value of the node to be inserted (70) with the value of the current node (which is now 75).

Since 70 is less than 75 and 75 left child is empty so we can insert 70 as a newnode as 75's left child. And its parent will be 75.