Data Structure Theory Final Term

Qustion:1

**The time complexity of a given code segment can be analyzed as follows:**

**Step-1:** The function splits the input range into two parts and calls it recursively for each part until the size of the range is less than 2. The time complexity of the loop inside fun() is O(r-l+ 1), since it iterates over each element in the range [l, r].

The for loop in the fun() function runs from l to r, which takes **O(r-l+1)** time.

**Step-2:** The median calculation takes constant time, which is **O(1).**

**Step-3:** The time complexity of a recursive function depends on the number of recursive calls made by the function and the time complexity of the operations performed in each call.

For recursive calls, each recursive call reduces the size of the range by half. So, the number of recursive calls will be O(log n), where n is the size of the input range. So, the overall time complexity of the fun() function is O(n log n), where n is the size of the input range.

In another way, The fun() function has a divide-and-conquer method of iteration and divides the problem into two subproblems of size n/2 each. Therefore, the iteration time complexity can be represented using the iteration relation T(n) = 2T(n/2) + O(n).

Using the Master theorem, we can see that the iteration time complexity is **O(nlogn).**

**Step-4:** The main function takes constant time to read input n, which is **O(1).**

**Step-5:** The function fun() is called with arguments 0 and n-1, which takes O(nlogn) time as per the analysis above.

Therefore, the overall time complexity of a given code segment is **O(nlogn).**

**Time complexity of 2nd function :**

The outer loop iterates from i=1 to n/2 and the inner loop iterates from j=1 to n with a step of i. So, the inner loop will be executed n/i times.

So, the total number of iterations of the inner loop will be:

If The outer loops rotates (times) The inner loop rotates(times)

I=1 1 n

I=2 1 n/2

I=3 1 n/3

I=4 1 n/4

……………………………………………………………………………………………………………………………………………….

……………………………………………………………………………………………………………………………………………….

I=n/2 1 n/(n/2)

The total loop rotates = n/1 + n/2 + n/3 + ... + n/(n/2)

= n(1 + 1/2 + 1/3 + ... + 1 /(n/2))

= n(log(n/2))

= n(logn)

The time complexity = O(nlogn) + O(1) = O(nlogn)

Therefore, the overall time complexity is **O(nlogn).**

Qustion:2

2.

class Node {

public:

float id;

char cha;

Node\* next;

Node\* next\_to\_next;

Node(float valy, char chy)

{

this->id = valy;

this->cha =chy;

this->next = nullptr;

this->next\_to\_next = nullptr;

}

};

Question: 3

Linear and non-linear data structures differ in terms of the **organization of their data.** Linear data structures have elements arranged in a sequence, with each element having only one predecessor and one successor, while non-linear data structures have elements that are not arranged in a sequence.

Stack, queue, and deque are all **linear data** structures.

Stack, queue, and deque are differ in terms of **how elements are accessed and moved**:

* A **stack** is a last-in-first-out (**LIFO**) data structure that allows elements to be inserted and removed from only one end, the top of the stack.
* A **queue** is a first-in-first-out (**FIFO**) data structure that allows elements to be inserted at the back and removed from the front of the queue.
* A **deque**, or double-ended queue, is a data structure that allows elements to be inserted and removed from **both ends**.

A **tree** is a **non-linear** data structure consisting of nodes connected by edges. Each node can have multiple child nodes, and edges represent relationships between nodes. Trees are often used to represent hierarchical relationships, such as in file systems or organizational charts.

**In summary, stack, queue and deck are all linear data structures, while a tree is a non-linear data structure. Stack is LIFO, queue is FIFO, and deque allows elements to be inserted and removed from both ends.**

Question:4.

Both singly linked lists and doubly linked lists can be used to implement stacks and queues. However, depending on the specific requirements of our use case, one type of list may be better suited than the other.

To implement a **stack**, a **singly linked** list is sufficient, since we need to add and remove elements at one end of the list. This makes using a singly linked list more efficient, as it requires less memory than a doubly linked list. Singularly linked lists are also easy to implement and can save some space if we're working with large data sets.

On the other hand, for implementing a **queue**, a **doubly linked list** is a better choice. This is because queues require adding and removing elements from both ends of the list. A doubly linked list allows deletion from the front and back of the list more efficiently than a singly linked list because a doubly linked has previous & next nodes, as it provides easy access to both the previous and next elements.

For a **deque** implementation, a **doubly linked** list is a good option because it allows efficient insertion and deletion at both ends of the deque. With a doubly linked list, elements can be added or removed from either end of the list in constant time, O(1). In contrast, a singly linked list requires O(n) time complexity for the tail operation of deque because it has to traverse the entire list to access the last element.

Both singly linked lists and doubly linked lists can be used to implement stacks and queues, but the choice depends on the specific requirements of our use case. If we are working with small data sets and memory is a concern, use a singly linked list for the stack. However, if we need to implement a queue that can accommodate deletions from both ends of the list, a doubly linked list would be a better choice.

Question:5:

**Algorithm of converting the infix expression to postfix expression using a stack:**

**Step 1:** Initialize an empty stack and an empty postfix expression.

**Step 2:** Start parsing the expression from left to right.

**Step 3:** If you encounter an operand (eg, ‘a’ to ‘z’), add it to the postfix expression.

**Step 4:** If you encounter an operator (eg, +, -, \*, /), pop all operators from the stack that have greater or equal precedence than the current operator and append them to the postfix expression, then push the current operator onto the stack.

**Step 5:** Repeat steps 3 and 4 until you reach the end of the expression.

**Step 6:** Pop all the remaining operators from the stack and add them to the postfix expression.

**Here are all the steps for the given expression:**

The expression: **a\*b+c\*d+e**

**Step -1:** Creat Stack ST: [empty]

**Step-2:** Create String S:  [empty]

**Step-3:** Reading a, add to S:         Stack: [empty]   S: [a]

**Step-4:** Reading ‘ \* ’, add to stack:  Stack: [\*]      S: [a]

**Step-5:** Reading b, add to S:          Stack: [\*]          S: [a, b]

**Step-6:** Reading ‘+‘, pop ‘ \* ‘ and add to string, add ‘+’ to stack : Stack: [+]                S: [ab\*]

**Step-7:** Reading c, add to S:     Stack: [+]        S: [ab\*c]

**Step-8:** Reading ‘ \* ‘, add to stack:  Stack: [+,\*]     S: [ab\*c]

**Step-9:** Reading d, add to S:    Stack: [+,\*]          S: [ab\*cd]

**Step-10:** Reading +, pop \* from stack and add to string then pop + from stack and add to string

, add + to stack: Stack: [+]   S: [ab\*cd\*+]

**Step-11:** Reading e, add to S:       Stack: [+]   S: [ab\*cd\*+e]

**Step-12:** End of expression:         Stack: [+]   S: [ab\*cd\*+e]

**Step-13:** Pop + from stack and add to S:       Stack: [empty]       S: [ab\*cd\*+e+]

**Step-14:** End of stack:                       Stack: [empty] S: [ab\*cd\*+e+]

**Postfix expression:** **ab\*cd\*+e+**

Question:6.

Arrays, singly linked lists and doubly linked lists are all data structures used to store and access data. Each of these data structures has its own advantages and disadvantages, including differences in memory usage.

**Array**

An array is a collection of elements stored in contiguous memory locations. Each element of the array is accessed by its index, which is an integer value representing the element's offset from the beginning of the array. The memory usage of an array can be calculated as follows:

**Memory usage = element size \* array length**

For example, if we have an array of **100 integers** and each integer is 4 bytes, the array's memory usage will be:

Memory usage = 4 bytes \* 100 **= 400 bytes**

**A singly linked list**

A singly linked list is a collection of elements where each element has a value and a reference to the next element in the list. The last element of the list has a reference to zero, indicating the end of the list. The memory usage of a singly linked list can be calculated as follows:

**Memory usage = element size \* list length + reference size \* list length**

The size of the reference is usually the size of a pointer, which is usually 4 or 8 bytes, depending on the machine architecture. For example, if we have a singly linked list of **100 integers** and each integer is 4 bytes and each reference is 8 bytes, the memory usage of the singly linked list will be:

Memory usage = 4 bytes \* 100 + 8 bytes \* 100 = **1200 bytes**

**A doubly linked list**

A doubly linked list is similar to a singly linked list, except that each element contains a reference to both the next and previous elements in the list. The memory usage of a doubly linked list can be calculated as follows:

**Memory usage = element size \* list length + reference size \* 2 \* list length**

For example, if we have a doubly linked list of **100 integers** and each integer is 4 bytes and each reference is 8 bytes, the memory usage of the doubly linked list will be:

Memory usage = 4 bytes \* 100 + 8 bytes \* 2 \* 100 = **2400 bytes**

In short, the memory usage of an array is proportional to the size of the array, while the memory usage of a linked list is proportional to the size of the list and the size of the reference to the next or previous element. A doubly linked list requires more memory than a singly linked list because it contains an additional reference to the previous element.

Processes using the most to least memory are: **Doubly linked list > Singly linked list > Array**

7.

If numbers are added to the stack in sorted order and the stack is always sorted, a binary search can be used to search for a value on the stack. However, binary search requires random access to elements of the stack, which is not possible with singly linked lists. Therefore, **an array-based implementation would be more appropriate.**

In an array-based implementation, we can use binary search to quickly search an element of the stack. Binary search has a time complexity of O(log n), where n is the number of elements in the stack. This is faster than linear search, which has a time complexity of O(n), which would be required in a linked list-based implementation.

Another advantage of an array-based implementation is that it requires less memory than a linked list. In a linked list, each node must store a reference to the next node, which increases memory usage. In an array, we only need to allocate memory for the elements on the stack.

In conclusion, if the stack is always sorted and we need to search an element quickly, an array-based implementation would be the preferred choice due to its ability to support binary search and lower memory usage.

8.  
If we are maintaining a head and tail for a singly linked-list, time complexity will be

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| **Inserting a value at the beginning** | **O(1)** |
| **Inserting a value at the end** | **O(n)** |
| **Deleting a value at the beginning** | **O(1)** |
| **Deleting a value at the end** | **O(n)** |
| **Inserting a value at the mid point** | **O(n)** |
| **Deleting a value at the mid point** | **O(n)** |

9.

1. The given binary tree is not a perfect binary tree because not all levels are completely filled.

**(b)** The given binary tree is a complete binary tree because all levels except the last level are completely full, and in the last level, all nodes are as far left as possible.

**(c)** The given binary tree is a binary search tree because for each node, all nodes in its left subtree have values ​​less than its own value and all nodes in its right subtree have values ​​greater than its own value.

**(d)** The BFS, inorder, preorder and postorder traversals of the given binary tree are:

**BFS Traversal:** **20 10 22 5 12 21 25 3 15**

**Inorder traversal:** **3 5 10 12 15 20 21 22 25**

**Pre-order traversal:** **20 10 5 3 15 12 22 21 25**

**Postorder Traversal:** **3 15 5 12 10 21 25 22 20**

Question-10.

**To insert 70 into the given binary search tree, we need to follow these steps:**

**Step-1:** Starting from the root node (which is 50), compare 70 with the value of the root node.

Since 70 is greater than 50, we move to the right subtree.

**Step-2:** We compare 70 with the value of the current node (which is 100) and since 70 is less than 100, we move to the left subtree of the current node.

**Step-3:** We compare 70 to the value of the current node (which is 80), and since 70 is greater than 80, we move to the right subtree of the current node.

**Step-4:** We compare 70 to the value of the current node (which is 90), and since 70 is less than 90, we move to the left subtree of the current node.

**Step-5:** The left subtree of the current node (which is 90) is empty, so we create a new node with the value 70 and make it the left child of the current node (ie, 90).

**Step-6:** After inserting 70 the resulting binary search tree will be:

50

/ \

30 100

/ \ / \

20 40 80 90

\

75

/

70

Note that the binary search tree property is maintained after 70 insertions, which is that all nodes in the left subtree of a node have a value less than the value of the node and all nodes in the right subtree have a value greater than the value of the node.