Answer Script

Question No. 01

1. Between array based stack implementation and linked-list based stack implementation which is better when I need random access in the stack? Explain the reasons. **15**

Answer No. 01

Between array based stack implementation and linked list based stack implementation, array based stack implementation is better for random access in the stack.

Array-based implementation is a method of implementing a data structure, such as a stack, using an array. The components of a stack are kept in a contiguous block of memory in an array-based implementation, where each component is given a distinct index. The top of the stack is stored at the highest index of the array, and new elements are added or removed from the top of the stack by modifying the index of the highest element.

Linked-list based implementation is a method of implementing a data structure, such as a stack, using a linked list. Each stack element is represented as a linked list node with a reference to the subsequent node in the stack in a linked-list implementation of a stack. The top of the stack is stored as the head of the linked list, and new elements are added or removed from the top of the stack by modifying the head reference.

This is primarily due to the fact that arrays have constant time access for any index, making it possible to get an element at any point in the stack in O(1) time. Accessing an element at a particular location in a linked-list implementation requires traversing the list from the head, which takes O(n) time.

In general, linked-list based implementations are preferable when dealing with variable-sized data structures that regularly need to be added to or removed from in the middle because those operations can be completed in O(1) time. However, for random access, array-based implementation is more efficient.

Question No. 02

2. What is the time complexity of push, pop and top operations in a stack? **10**

Answer No. 02

The time complexity of push operation in a stack: O(1).
When we push any value in the stack it's added to the top of the last value. So, we don't need to traverse full stack. That's why push operation time complexity is O(1).

The time complexity of pop operation in a stack: O(1).
When we pop any value in the stack it's removed to the top of the last value. So, we don't need to traverse full stack. That's why pop operation time complexity is O(1).

The time complexity of top operation in a stack: O(1).
When we call the top operation in the stack. It returns the last/top element in the stack. So, we don't need to traverse full stack. That's why top operation time complexity is O(1).

Question No. 03

3. Suppose you need a stack of characters, a stack of integers and a stack of real numbers. How will you implement this scenario using a single stack? **15**

Answer No. 03

If i need a stack of characters, a stack of integers and a stack of real numbers in a single stack i will use template class.

A template class is a generic class that can work with different data types. Instead of defining a class for a specific data type, we can define a template class that can be instantiated for any data type that meets certain requirements.

A template class is defined using the template keyword followed by template parameters enclosed in angle brackets (<>).

First I created a template class.
Like, template <class T>. Then create a stack class.
Like:

```
class Stack {
T temp_late_variable;
};

int main()
{
Stack<char> st1;
st1.push(‘A’);

Stack<int> st2;
st2.push(5);
}
```

In the main function when we create st1 stack by character parameters it's go to stack class and store character variables in temp_late_vairable.
Again when we create st2 stack by integer parameters it’s go to stack class and store integer variables in temp_late_variable.

In this way we can use a stack of characters, a stack of integers and a stack of real numbers in a single stack.

A template class in C++ is a powerful feature that allows writing generic classes that can work with any data type that meets certain requirements. The main advantage of template classes is that they can save a lot of time and effort by reducing code duplication and increasing code reuse.

Question No. 04

4. What is a postfix expression and why do we need it? How is it evaluated using a stack for the below example? You need to show all the steps. **15**

   abc*+de*+

Answer No. 04

Postfix expressionis the notation in which operators are placed after the corresponding operands in the expression.
In a postfix expression, an operator is written after its operands.
Postfix expressions are needed when we require operators before the operands while postfix notations are needed when we require operators after the operands.

Here are the steps for evaluating the expression "abc*+de*+":

Create an empty stack.

1. Push "a" onto the stack after reading it. The stack currently has [a].
2. Push "b" onto the stack after reading it. [a, b] are now present on the stack.
3. Put "c" on the stack after reading it. [a, b, c] are now present on the stack.
4. Pick up "c" and "b" from the stack after reading "*". Analyse the formula "b * c." Place the outcome "bc" on the stack. [a, bc] are now present on the stack.
5. Take "bc" and "a" out of the stack after reading "+". Analyse the phrase "a + bc." Place "abc" as the outcome on the stack. [abc] is now present in the stack.
6. Push "d" onto the stack after reading it. [abc, d] are now present on the stack.
7. Put "e" on the stack after reading it. There are now [abc, d, e] on the stack.
8. Take "e" and "d" out of the stack after reading "*". Analyse the formula "d * e." Place "de" as the outcome on the stack. [abc, de] are now present in the stack.
9. Take "de" and "abc" out of the stack after reading "+". Analyse the phrase "abc + de." Put "abcde" as the output on the stack. [abcde] is now present in the stack.

The value at the top of the stack is the final result, "abcde."
In this way, the postfix expression "abc*+de*+" can be evaluated using a stack to give the final result "abcde".

Question No. 05

5. Simulate balanced parentheses check using stack for the below example. You need to show all the steps. **15**

(([ ][ ]{ ( ) }))

Answer No. 05

1. Push the character onto the stack if it is an opening parenthesis, such as "(", "{" or "[".
2. Pop the top element from the stack and see if it matches the associated opening parenthesis if the current character is a closing parenthesis, such as ")", "}", or "]". The parentheses are not balanced if they don't match.
3. Repeat the process until the entire expression has been processed.
4. If the stack is not empty at the end, then the parentheses are not balanced.

Simulate:

Create an empty stack.

1. Read "(". Push it onto the stack. The stack currently has [(].
2. Read "(". Push it onto the stack. Currently on the stack are [(, (].
3. Read "[". Push it onto the stack. Currently on the stack are [(, (, []].
4. Read "]". Pop "[" from the stack. The stack now contains [(, (].
5. Read "[". Push it onto the stack. Currently on the stack are [(, (, []].
6. Read "]". Pop "[" from the stack. The stack now contains [(, (].
7. Now that "{" has been read and added to the stack, it contains [(, (, {].
8. Push "(" onto the stack, read it, and the stack now contains [(, (, {, (].
9. Read ") and remove "(" from the stack, leaving [(, (, {] on the stack.
10. Read "}" remove "{" from the stack, [(, (] are now present in the stack.
11. Read ")". Pop "(" from the stack. The stack now contains [(].
12. Read ")". Pop "(" from the stack. The stack now contains [].

The stack is empty, which means the parentheses are balanced.

Question No. 06

6. Sort a stack of integers using another stack for the below example. You need to show all the steps. **15**

Stack -> [3,4,6,2,5]

Answer No. 06

Here's an algorithm to sort a stack of integers using another stack:

1. Create an empty stack called "temp".
2. While the input stack is not empty, do the following:
    a. Pop an element from the input stack and call it "x".
    b. While temp is not empty and the top element of temp is greater than x, pop the top element of temp and push it back to the input stack.
    c. Push x onto temp.
3. The temp will now contain the sorted elements of the input stack.

input stack: {3, 4, 6, 2, 5}
1. Create an empty stack called "temp". => temp: {}
2. Pop the top element form the input stack and sotre it into x variable. => x = 5, input: {3, 4, 6, 2}
3. Now temp stack is empty so push x value into temp stack. => temp: {5}
4. Pop the top element form the input stack and sotre it into x variable. => x = 2, input: {3, 4, 6}
5. temp top element is greater than x variable. So, push it into input stack and pop from temp stack. =>input: {3, 4, 6, 5}
   --Now temp stack is empty so push x value into temp stack. => temp: {2}
6. Pop the top element form the input stack and sotre it into x variable. => x = 5, input: {3, 4, 6}
7. temp top element is less than x variable. So, push x value into temp stack. =>temp: {2, 5}
8. Pop the top element form the input stack and sotre it into x variable. => x = 6, input: {3, 4}
9. temp top element is less than x variable. So, push x value into temp stack. =>temp: {2, 5, 6}
10. Pop the top element form the input stack and sotre it into x variable. => x = 4, input: {3}
11. temp top element is greater than x variable. So, push it into input stack and pop from temp stack. => input: {3, 6}, temp: {2, 5}
   --temp top element is greater than x variable. So, push it into input stack and pop from temp stack. => input: {3, 6, 5}, temp: {2}
   --temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 4}, input: {3, 6, 5}

12. Pop the top element form the input stack and sotre it into x variable. => x = 5, input: {3, 6}
13. temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 4, 5}, input: {3, 6}
14. Pop the top element form the input stack and sotre it into x variable. => x = 6, input: {3}
15. temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 4, 5, 6}, input: {3}
16. Pop the top element form the input stack and sotre it into x variable. => x = 3, input: {}
17. temp top element is greater than x variable. So, push it into input stack and pop from temp stack. =>input: {6}, temp: {2, 4, 5}
--temp top element is greater than x variable. So, push it into input stack and pop from temp stack. =>input: {6, 5}, temp: {2, 4}
--temp top element is greater than x variable. So, push it into input stack and pop from temp stack. =>input: {6, 5, 4}, temp: {2}
--temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 3}, input: {6, 5, 4}
18. Pop the top element form the input stack and sotre it into x variable. => x = 4, input: {6, 5}
19. temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 3, 4}, input: {6, 5}
20. Pop the top element form the input stack and sotre it into x variable. => x = 5, input: {6}
21. temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 3, 4, 5}, input: {6}
22. Pop the top element form the input stack and sotre it into x variable. => x = 6, input: {}
23. temp top element is less then x variable. So, push x value into temp stack. =>temp: {2, 3, 4, 5, 6}, input: {}

Sorting Stack: {2, 3, 4, 5, 6}

Question No. 07

7. Convert the infix expression to postfix expression using a stack. You need to show all the steps. **15**

a+b*c+d*e

Answer No. 07

Here's the process of converting an infix expression to a postfix expression using a stack:
Step 1: First, create an empty stack.
Step 2: Begin by left-to-right scanning the infix phrase.
Step 3: Include the current character in the postfix expression if it is an operand.
Step 4: Add all the operators with greater or equal precedence to the postfix expression if the current character is an operator. Then, move the active operator to the top of the stack.
Step 5: Carry out steps 3 and 4 once more to process every character in the infix expression.
Step 6: The operators are all popped off the stack and added to the postfix expression.

Let's apply these steps to the expression: "a+b*c+d*e".
Step 1: First, create an empty stack.
Stack: []
Postfix expression: ""
Step 2: Begin by left-to-right scanning the infix phrase.
Current character: a
Step 3: Include the current character in the postfix expression if it is an operand.
Stack: []
Postfix expression: "a"
Step 4: Add all the operators with greater or equal precedence to the postfix expression if the current character is an operator. Then, move the active operator to the top of the stack.
Current character: +
Stack: []
Postfix expression: "a"
Stack: [+]
Postfix expression: "a"
Current character: b
Stack: [+]
Postfix expression: "ab"
Current character: *
Stack: [+]
Postfix expression: "ab"
Stack: [+, *]
Postfix expression: "ab"

```
Current character: c
Stack: [ +, * ]
Postfix expression: "abc"
Step 4: Add all the operators with greater or equal precedence to the postfix expression
if the current character is an operator. Then, move the active operator to the top of the
stack.
Stack: [ + ]
Postfix expression: "abc*"
Current character: +
Stack: [ + ]
Postfix expression: "abc*"
Stack: [ +, + ]
Postfix expression: "abc*"
Current character: d
Stack: [ +, + ]
Postfix expression: "abc*d"
Current character: *
Stack: [ +, + ]
Postfix expression: "abc*d"
Stack: [ +, +, * ]
Postfix expression: "abc*d"
Current character: e
Stack: [ +, +, * ]
Postfix expression: "abc*de"
Step 5: Repeat steps 3 and 4 until all the characters in the infix expression have been
processed.
Step 6: Pop all the operators from the stack and add them to the postfix expression.
Stack: [ +, + ]
Postfix expression: "abc*de*"
Stack: [ + ]
Postfix expression: "abc*de*+"
Stack: []
Postfix expression: "abc*de*++"
The postfix expression is: "abc*de*++"
```