Answer Script

Question No. 01
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.

Answer No. 01

Array-based implementation is better when you need random access in the stack.

1. This is because arrays provide constant time access to elements at a given index, whereas linked-lists only allow for sequential access to elements.

2. With a linked-list, you would have to traverse the entire list from the head to reach a specific element, which takes linear time.

3. On the other hand, with an array-based stack, you can directly access an element at a specific index in constant time.

So, if you need to access elements in a random order in the stack, an array-based implementation would be more efficient, as it allows for faster access to specific elements

Question No. 02
What is the time complexity of push, pop and top operations in a stack?

Answer No. 02

The time complexity of push, pop and top operations in a stack in general:

1. Push operation: The time complexity of push operation in a stack is usually O(1), which means that adding an element to the top of the stack takes constant time, regardless of the size of the stack.

2. Pop operation: The time complexity of pop operation in a stack is also usually O(1), which means that removing an element from the top of the stack takes constant time, regardless of the size of the stack.

3. Top operation: The time complexity of top operation in a stack is also O(1), which means that accessing the top element of the stack takes constant time, regardless of the size of the stack.

Question No. 03

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?

Answer No. 03

We can implement this scenario using a single stack by using templates in C++.

Templates allow us to create a single stack implementation that can store elements of any data type. We can define a template class Stack that takes a class parameter T and implements the push, pop, and top operations for elements of type T.

Here's an example implementation in C++:

```
#include<bits/stdc++.h>
using namespace std;

template <class T>
class Stack{
public:
   Stack()
   {

   }

   T print(T value)
   {
      return value;
   }
};
int main()
{
   Stack<int> int_stack;
   cout<<int_stack.print(5)<<endl;

   Stack<char> char_stack;
```

```
    cout<<char_stack.print('M')<<endl;

    Stack<double> real_stack;
    cout<<real_stack.print(10.99)<<endl;
}
```

Question No. 04

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.

abc*+de*+

Answer No. 04

A postfix expression (also known as reverse Polish notation) is a way of writing arithmetic expressions in which the operands appear before their operators. In postfix expressions, operators are written after the operands they operate on. This is different from the traditional infix notation where operators appear in between the operands.

We need postfix expressions because they have several advantages over infix expressions, particularly when it comes to evaluating expressions using a computer. In postfix notation, there is no need to consider the precedence of operators or to use parentheses to specify the order of operations. This makes it easier for computers to parse and evaluate expressions, as they do not need to perform any complex parsing or use any stacks to keep track of the order of operations.

For the postfix expression abc*+de*+, the evaluation would be as follows:

1. Create an empty stack. The stack is initially empty: []
2. Scan the postfix expression from left to right: a, b, c, *, +, d, e, *, +.
3. If the character scanned is an operand, push it onto the stack. The stack after scanning a is [a]. The stack after scanning b is [a, b]. The stack after scanning c is [a, b, c].
4. If the character scanned is an operator, pop two elements from the stack, apply the operator to these elements, and push the result back onto the stack. The stack after scanning * is [a, bc]. The stack after scanning + is [ab+c]. The stack after scanning d is [ab+c, d]. The stack after scanning e is [ab+c, d, e].
5. Repeat steps 3 and 4 until all characters in the postfix expression have been scanned. The stack after scanning * is [ab+cd*e]. The stack after scanning + is [ab+cde*+].

6. The final value on the stack is the result of the postfix expression. The final value on the stack is ab+cde*+, which is equal to the infix expression (a + b) + (c * d * e).

Question No. 05

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

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

Answer No. 05

Balanced parentheses check using a stack involves checking whether the parentheses in an expression are balanced, i.e., every opening parenthesis has a corresponding closing parenthesis, and they are nested properly.

To do this, we can use a stack to keep track of the parentheses as we scan the expression from left to right.

For the expression (([ ][ ]{ ( ) })), the balanced parentheses check would be as follows:

1. Create an empty stack. The stack is initially empty: []
2. Scan the expression from left to right: (, (, [,], [,], {, (,), },),).
3. If the character scanned is an opening parenthesis, push it onto the stack. The stack after scanning (is [(]. The stack after scanning (again is [(, (]. The stack after scanning [is [(, (, []. The stack after scanning] is [(, (, []. The stack after scanning [again is [(, (, [, []]. The stack after scanning] again is [(, (, []]. The stack after scanning { is [(, (, [, {}].
4. If the character scanned is a closing parenthesis, pop the top element from the stack. If the popped character does not match the corresponding opening parenthesis for the current closing parenthesis, the expression is not balanced. The stack after scanning) is [(, (, []. The popped element matches the corresponding opening parenthesis, so the expression is still balanced. The stack after scanning) again is [(, (]. The popped element matches the corresponding opening parenthesis, so the expression is still balanced. The stack after scanning } is [(, (]. The popped element matches the corresponding opening parenthesis, so the expression is still balanced. The stack after scanning) again is [(]. The popped element matches the corresponding opening parenthesis, so the expression is still balanced. The stack after scanning) again is []. The popped

element matches the corresponding opening parenthesis, so the expression is still balanced.

5. Repeat steps 3 and 4 until all characters in the expression have been scanned. All characters in the expression have been scanned. If the stack is not empty, the expression is not balanced. If the stack is empty, the expression is balanced. The stack is empty, so the expression is balanced.

The expression (([ ][ ]{ ( ) })) is balanced & Valid

Question No. 06

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

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

Answer No. 06

Sorting a stack of integers using another stack can be done using the following steps :

1. Main Stack : [3,4,6,2,5]
   Temp Stack: []
2. Main Stack : [3, 4, 6, 2]
   Temp Stack: [5]
3. Main Stack : [3, 4, 6, 5]
   Temp Stack: [2]
4. Main Stack : [3, 4, 6]
   Temp Stack: [2, 5]
5. Main Stack : [3, 4]
   Temp Stack: [2, 5, 6]
6. Main Stack : [3, 6, 5]
   Temp Stack: [2, 4]
7. Main Stack : [3]
   Temp Stack: [2, 4, 5, 6]
8. Main Stack : [6, 5, 4]
   Temp Stack: [2, 3]
9. Main Stack : []
   Temp Stack: [2, 3, 4, 5, 6]

So, the sorted stack in ascending order would be "Temporary stack -> [2, 3, 4, 5, 6]".

Question No. 07

Convert the infix expression to postfix expression using a stack. You need to show all the steps. a+b*c+d*e

Answer No. 07

The process of converting an infix expression to a postfix expression involves reading the infix expression from left to right and putting operators and operands in a stack.

Steps to convert "a+bc+de" to postfix:

Initialize an empty stack and postfix expression:
Stack: []

1. Postfix expression: ""

Read the first symbol "a":
Stack: []

2. Postfix expression: "a"

Read the next symbol "+":
Stack: ["+"]

3. Postfix expression: "a"

Read the next symbol "b":
Stack: ["+"]

4. Postfix expression: "ab"

Read the next symbol "":

Stack: ["+", ""]

5. Postfix expression: "ab"

Read the next symbol "c":
Stack: ["+"]

6. Postfix expression: "abc*"

Read the next symbol "+":
Stack: ["+", "+"]

7. Postfix expression: "abc*"

Read the next symbol "d":
Stack: ["+"]

8. Postfix expression: "abc*d"

Read the next symbol "**"*:
*Stack: ["+", "**"]

9. Postfix expression: "abc*d"

Read the next symbol "e":
Stack: ["+"]

10. Postfix expression: "abc*de*"

The entire infix expression has been processed, so pop the remaining operators from the stack:
Stack: []

11. Postfix expression: "abc*de**+"

So, the postfix expression for "a+b*c+d*e" would be "abc*+de*+".