Answer Script

Basic Data Structure and Problem Solving Part-II

Week-06, Module 23: Theory Assignment 02

(nayeem.cse6.bu@gmail.com)

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 we need random access in the stack. **Explanation:** In an array-based implementation, elements are stored in a contiguous block of memory, and each element can be directly accessed using an index. This means that you can perform random access operations in O(1) time, which is very efficient. In a linked-list-based implementation, elements are stored in separate nodes that are linked together. To access an element in the middle of the list, you have to traverse the list from the beginning, which takes O(n) time in the worst case. This makes linked-list-based implementations less efficient for random access operations compared to array-based implementations. **Example:** Let, we have a stack with 5 elements: [1, 2, 3, 4, 5]. In an array-based implementation, the elements are stored in a contiguous block of memory, like this | 1 | 2 | 3 | 4 | 5 |

To access the element at index 3 (i.e., the 4th element), we can simply use the index 3 to directly access the memory location, like this: array[3], which will return 4. This operation takes O(1) time.

In a linked-list-based implementation, the elements are stored in separate nodes that are linked together, like this:

| 1 | -> | 2 | -> | 3 | -> | 4 | -> | 5 |

To access the element at index 3 (i.e., the 4th element), we have to start at the head of the list (i.e., node 1) and traverse the list until you reach the node at index 3 (i.e., node 4), like this:

```
node = head
for i in 0 to 2:
 node = node.next
```

Now, node points to the node at index 3, which contains the value 4. This operation takes O(n) time in the worst case, where n is the number of elements in the list.

As we can see, the array-based implementation is more efficient for random access operations.

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 depends on the implementation of the stack.

In an array-based implementation of a stack:

> Push operation: O(1) average time complexity, O(n) worst-case time complexity if the array needs to be resized.
> Pop operation: O(1) average time complexity.
> Top operation: O(1) average time complexity.

In a linked-list-based implementation of a stack:

> Push operation: O(1) average time complexity.
> Pop operation: O(1) average time complexity.
> Top operation: O(1) average time complexity.

As we can see, both array-based and linked-list-based implementations have the same average time complexity for push, pop, and top operations, which is O(1). However, in the worst case, the array-based implementation has a higher time complexity for the push operation, while the linked-list-based implementation has a constant time complexity for all operations.

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 use C++ templates to implement a stack that can store elements of different data types, such as characters, integers, and real numbers.

The Implementation of this scenario using a stack:

```
#include <bits/stdc++.h>
```

```
using namespace std;

template<typename T>
class Stack {
    private:
        int top;
        T data[100];

    public:
        Stack() {
            top = -1;
        }

        void push(T value) {
            top++;
            data[top] = value;
        }

        T pop() {
            T value = data[top];
            top--;
            return value;
        }
};

int main() {
    Stack<char> charStack;
    Stack<int> intStack;
    Stack<float> floatStack;

    charStack.push('A');
    intStack.push(123);
    floatStack.push(3.14);

    char c = charStack.pop();
    int i = intStack.pop();
    float f = floatStack.pop();
```

```
    cout << c << endl;
    cout << i << endl;
    cout << f << endl;

    return 0;
}
```

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

Postfix Expression:

A postfix expression (also known as a reverse Polish notation expression) is an arithmetic expression where the operators follow the operands. In a postfix expression, the order of the operands is unchanged, but the order of the operators is reversed from the infix expression.

The Need of Postfix Expression:

We need postfix expressions because they are easier to evaluate using a stack compared to infix expressions. In a postfix expression, there is no need to handle operator precedence and parenthesis, as the order of the operators and operands is already defined.

Here's how to evaluate a postfix expression using a stack:

› Initialize an empty stack operandStack.

› Traverse the postfix expression from left to right.

› If the current character is an operand (a-z or 0-9), push it onto the

operandStack.

> If the current character is an operator (+,-,*,/), pop two operands from the operandStack, perform the operation on them and push the result back onto the operandStack.

> Repeat the above steps (2 to 4) for the entire postfix expression.

> The result of the postfix expression is the top of the operandStack.

Here's what the steps would look like for the postfix expression abc*+de*+:

> operandStack is empty.

> a is an operand, push it onto operandStack: operandStack: a

> b is an operand, push it onto operandStack: operandStack: a b

> c is an operand, push it onto operandStack: operandStack: a b c

> * is an operator, pop c and b from operandStack, perform b * c and push the result onto operandStack: operandStack: a bc*

> + is an operator, pop bc* and a from operandStack, perform a + bc* and push the result onto operandStack: operandStack: abc*+

> d is an operand, push it onto operandStack: operandStack: abc*+ d

> e is an operand, push it onto operandStack: operandStack: abc*+ de

> * is an operator, pop e and d from operandStack, perform d * e and push the result onto operandStack: operandStack: abc*+ de*

> + is an operator, pop de* and abc*+ from operandStack, perform abc*+ + de* and push the result onto operandStack: operandStack: abc*+de*+

> The result of the postfix expression is the top of the operandStack, which is abc*+de*+.

Question No. 05
Simulate balanced parentheses check using stack for the below example. You need to show all the steps. (([ ][ ]{ ( ) }))
Answer No. 05
The steps for checking the balance of parentheses for the given expression are as follows: > Create an empty stack. > Read the expression from left to right. > If a character is an opening parenthesis, push it onto the stack. > If a character is a closing parenthesis, pop the top element from the stack. The popped element must match the opening parenthesis for the current closing parenthesis. > Repeat steps 3 and 4 for each character in the expression. > At the end of the expression, the stack should be empty, indicating that the parentheses are balanced. **Example evaluation of the expression (([ ][ ]{ ( ) })):** Stack: [ ] Read (, push (onto the stack. Stack: [(] Read (, push (onto the stack. Stack: [(, (]

Read [, push [onto the stack.
Stack: [(, (, [ ]
Read [, push [onto the stack.
Stack: [(, (, [, [ ]]
Read], pop [from the stack.
Stack: [(, (, [ ]]
Read], pop [from the stack.
Stack: [(, ( ]
Read {, push { onto the stack.
Stack: [(, (, {]
Read (, push (onto the stack.
Stack: [(, (, {, ( ]
Read), pop (from the stack.
Stack: [(, (, {]
Read }, pop { from the stack.
Stack: [(, ( ]
Read), pop (from the stack.
Stack: [( ]
Read), pop (from the stack.
Stack: [ ]
End of expression is reached, the stack is empty, indicating that the parentheses are balanced.

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

Here is the step by step process to sort the given stack of integers using another stack:

> Initialize an empty stack sortedStack.

> Take the top element of the original stack originalStack (3) and compare it with the elements of sortedStack.

> If sortedStack is empty or if the top of sortedStack is greater than the element being compared (3), push the element (3) onto sortedStack.

> If the top of sortedStack is smaller than the element being compared (3), pop elements from sortedStack until the top of sortedStack is greater than the element being compared (3) or sortedStack is empty.

> Repeat the above steps (2 to 4) for the remaining elements of originalStack:

4 > 3, push 4 to sortedStack

6 > 4, push 6 to sortedStack

2 < 6, pop 6 and push 2 to sortedStack

5 > 2, push 5 to sortedStack

> At this point, sortedStack will contain the sorted elements of the original stack in ascending order: [2,3,4,5,6].

The Program is given below:

```
#include<bits/stdc++.h>

using namespace std;

stack<int> sortStack(stack<int> &originalStack) {
    stack<int> sortedStack;
    while (!originalStack.empty()) {
        int temp = originalStack.top();
        originalStack.pop();
        while (!sortedStack.empty() && sortedStack.top() > temp) {
```

```
            originalStack.push(sortedStack.top());
            sortedStack.pop();
        }
        sortedStack.push(temp);
    }
    return sortedStack;
}

int main() {
    stack<int> originalStack;
    originalStack.push(5);
    originalStack.push(2);
    originalStack.push(6);
    originalStack.push(4);
    originalStack.push(3);
    stack<int> sortedStack = sortStack(originalStack);
    cout << "Sorted stack is: ";
    while (!sortedStack.empty()) {
        cout << sortedStack.top() << " ";
        sortedStack.pop();
    }
    return 0;
}
```

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
Here is the step by step process to convert an infix expression to a postfix expression using a stack:

> Initialize an empty stack operatorStack.

> Traverse the infix expression from left to right.

> If the current character is an operand (a-z or 0-9), add it to the postfix expression.

> If the current character is an operator (+,-,*,/), pop elements from the operatorStack until the top of the operatorStack has lower precedence than the current operator or operatorStack is empty. Then, push the current operator onto the operatorStack.

> Repeat the above steps (2 to 4) for the entire infix expression.

> After the infix expression has been fully processed, pop all the elements from the operatorStack and add them to the postfix expression.

Here's what the steps would look like for the infix expression a+b*c+d*e:

> operatorStack is empty.

> a is an operand, add it to the postfix expression: a

> + is an operator, push it onto operatorStack: operatorStack: +

> b is an operand, add it to the postfix expression: a b

> * is an operator, push it onto operatorStack: operatorStack: + *

> c is an operand, add it to the postfix expression: a b c

> + is an operator, pop * from operatorStack and add it to the postfix expression: a b c *

> Push the current operator + onto operatorStack: operatorStack: +

> d is an operand, add it to the postfix expression: a b c * d

> * is an operator, push it onto operatorStack: operatorStack: + *

> e is an operand, add it to the postfix expression: a b c * d e

> Pop the remaining operators from operatorStack and add them to the postfix expression: a b c * d e + * +

> The postfix expression is a b c * d e + * +, which can be evaluated using a postfix evaluation algorithm.