Stack-related problems discussed in the lecture class

By Dr. GC Jana

---

1. Balanced Parentheses

Problem Statement:

Given a string containing just the characters (,), {, }, [and], determine if the input string is valid. An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.

Approach:

We will use a **stack** to solve this problem because of its **LIFO (Last-In-First-Out)** property, which is ideal for matching the most recent opening bracket with its corresponding closing bracket.

Example String

Let's take this string: "**{ [( )] }**".

Steps:

- **Create a Stack** -> Initialize an empty stack to store opening brackets as you traverse the string.
- **Traverse the String** Loop through each character in the string one-by-one.

Detailed Step-by-Step for "{ [( )] }":

- Initial state:
  - Stack: [ ] (empty)

Step 1:

- Character: {
  - Since { is an opening bracket, push it onto the stack.
  - **Stack**: [**{**]

Step 2:

- Character: [
  - Since [is an opening bracket, push it onto the stack.
  - **Stack**: [**{, [,**]

Step 3:

- **Character: (**
  - Since (is an opening bracket, push it onto the stack.
  - Stack: [**{, [, (,**]

Step 4:

- **Character:)**
  - Since) is a closing bracket, check the top of the stack.
  - The top of the stack is (, which is the matching opening bracket for). Pop (from the stack.
  - Stack after pop: [**{, [**,]

Step 5:

- Character:]
  - Since] is a closing bracket, check the top of the stack.
  - The top of the stack is [, which is the matching opening bracket for]. Pop [from the stack.
  - Stack after pop: [**{**]

Step 6:

- Character: }
  - Since } is a closing bracket, check the top of the stack.
  - The top of the stack is {, which is the matching opening bracket for }. Pop { from the stack.
  - Stack after pop: [] (empty)

3 Final Check

- **After traversing** the entire string,
- **if the stack is empty**, the parentheses are **balanced**.
- I**f the stack still contains some elements**, then the parentheses are **not balanced**.
  - Final Stack State: [] (empty)
  - Since the stack is empty, the string "{[()]}" is balanced.

General Algorithm:

1. **Create an empty stack.**

2. **Traverse the string** character by character:

- If an opening bracket ((, {, [) is encountered, push it onto the stack.
- If a closing bracket (), },]) is encountered:
  - If the stack is empty, return false (because there is no matching opening bracket).
  - Otherwise, pop the top element from the stack.
  - Check if the popped element is the matching opening bracket for the current closing bracket. If not, return false.

3. After traversal:

- If the stack is empty, return true (the parentheses are balanced).

- If the stack is not empty, return false (there are unmatched opening brackets).

Edge Cases:

➔ An empty string is considered balanced.
➔ Strings with only opening or only closing brackets, e.g., "((((", "))))", are unbalanced.

Example 2:

Input: "**([{}])**"

Steps:

Character: (→ push

Stack: [( ]

Character: [→ push

Stack: [(, [ ]

Character: { → push

Stack: [(, [, {]

Character: } → pop (matching {)

Stack: [(, [ ]

Character:] → pop (matching [)

Stack: [( ]

Character:) → pop (matching ()

Stack: [] (empty)

Since the stack is empty after traversal, the string "([{}])" is balanced.

Java Code:

```java
import java.util.Stack;

public class BalancedParentheses {
    public static boolean isBalanced(String str) {
        Stack<Character> stack = new Stack<>();

        for (char c : str.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else if (c == ')' || c == '}' || c == ']') {
                if (stack.isEmpty()) return false;
                char top = stack.pop();
                if (!isMatchingPair(top, c)) return false;
            }
        }

        return stack.isEmpty();
    }
```

```java
    private static boolean isMatchingPair(char open, char close) {
        return (open == '(' && close == ')') ||
               (open == '{' && close == '}') ||
               (open == '[' && close == ']');
    }

    public static void main(String[] args) {
        String expression = "{[()]}";
        System.out.println(isBalanced(expression));
    }
}
```

C++ Code:

```cpp
#include <iostream>
#include <stack>
using namespace std;

bool isMatchingPair(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '{' && close == '}') ||
           (open == '[' && close == ']');
}

bool isBalanced(string str) {
    stack<char> stack;

    for (char c : str) {
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else if (c == ')' || c == '}' || c == ']') {
            if (stack.empty()) return false;
            char top = stack.top();
            stack.pop();
            if (!isMatchingPair(top, c)) return false;
        }
    }
```

```java
import java.util.Stack;

public class BalancedParentheses {
    public static boolean isBalanced(String str) {
        Stack<Character> stack = new Stack<>();

        for (char c : str.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else if (c == ')' || c == '}' || c == ']') {
                if (stack.isEmpty()) return false;
                char top = stack.pop();
                if (!isMatchingPair(top, c)) return false;
            }
        }

        return stack.isEmpty();
    }
```

---

2. Next Greater Element

Problem Statement:

Given an array, for each element in the array, find the next greater element (NGE) to its right. If there is no greater element to the right, output -1 for that element.

Example Array:

Let's consider the array: **[4, 5, 2, 25].**

Approach:

We can solve this problem efficiently **using a stack**. The idea is to **traverse** the array from **right to left**, **maintaining a stack** that holds elements for which we haven't yet found a next greater element.

Algorithm:

- Initialize a stack to store elements.
- Traverse the array from right to left:
    - For each element, while the stack is not empty and the element at the top of the stack is less than or equal to the current element, pop elements from the stack.
    - If the stack becomes empty after popping, it means there is no greater element to the right, so store -1 for this element.
    - Otherwise, the top element of the stack is the next greater element.

    - Push the current element onto the stack.
- Continue until all elements have been processed.

Step-by-Step Explanation for the Array [4, 5, 2, 25]:

Initial state:

- Stack: [] (empty)
- Result: [-1, -1, -1, -1] (to store the next greater elements)

Step 1: Process element 25 (rightmost element)

- Current element: 25
- Since the stack is empty, there is no greater element to the right.
    - Set NGE[3] = -1 for element 25.
    - Push 25 onto the stack.

-> Stack after Step 1: [25]

-> Result after Step 1: [-1, -1, -1, -1]

Step 2: Process element 2

- Current element: 2
- The top of the stack is 25, which is greater than 2.
    - Set NGE[2] = 25 for element 2.
    - Push 2 onto the stack.

-> Stack after Step 2: [25, 2]

-> Result after Step 2: [-1, -1, 25, -1]

Step 3: Process element 5

- Current element: 5
- The top of the stack is 2, which is less than 5. Pop 2 from the stack.
- The new top of the stack is 25, which is greater than 5.
    - Set NGE[1] = 25 for element 5.
    - Push 5 onto the stack.

Stack after Step 3: [25, 5]

Result after Step 3: [-1, 25, 25, -1]

Step 4: Process element 4 (leftmost element)

- Current element: 4
- The top of the stack is 5, which is greater than 4.
    - Set NGE[0] = 5 for element 4.
    - Push 4 onto the stack.

Stack after Step 4: [25, 5, 4]

Result after Step 4: [5, 25, 25, -1]

Final Result:

For the array [4, 5, 2, 25], the Next Greater Element array is: [5, 25, 25, -1]

Java Code:

```java
import java.util.Stack;

public class NextGreaterElement {
    public static void nextGreater(int[] arr) {
        Stack<Integer> stack = new Stack<>();
        int[] result = new int[arr.length];

        for (int i = arr.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= arr[i]) {
                stack.pop();
            }

            result[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(arr[i]);
        }

        // Print the result
        for (int n : result) {
            System.out.print(n + " ");
        }
    }
```

```java
    public static void main(String[] args) {
        int[] arr = {4, 5, 2, 25};
        nextGreater(arr);
    }
}
```

CPP Code:

```cpp
#include <iostream>
#include <stack>
using namespace std;

void nextGreater(int arr[], int n) {
    stack<int> stack;
    int result[n];

    for (int i = n - 1; i >= 0; i--) {
        while (!stack.empty() && stack.top() <= arr[i]) {
            stack.pop();
        }

        result[i] = stack.empty() ? -1 : stack.top();
        stack.push(arr[i]);
    }

    // Print the result
    for (int i = 0; i < n; i++) {
        cout << result[i] << " ";
    }
}
```

```cpp
int main() {
    int arr[] = {4, 5, 2, 25};
    int n = sizeof(arr) / sizeof(arr[0]);
    nextGreater(arr, n);
    return 0;
}
```

3. Evaluate Postfix Expression

Problem Statement:

A Postfix Expression (or Reverse Polish Notation) is a mathematical expression in which the operator follows the operands. To evaluate a postfix expression, we can use a stack to store operands and apply operators in the correct order.

Approach:

We will use a **stack** to store operands (numbers). When we **encounter** an **operator**, we **pop two operands** from the stack, **perform the operation**, and **push** the **result back onto the stack**. At **the end of the traversal, the stack will contain the final result**.

Example Postfix Expression:

Let's take this **postfix** expression: "**5 6 2 + * 12 4 / -**".

This expression is equivalent to the **infix** expression: **5 * (6 + 2) - (12 / 4)**.

Steps:

1. Create a Stack

- Initialize an empty stack to store operands.

2. Traverse the Postfix Expression Read each character from left to right:

- If the character is an operand, push it onto the stack.
- If the character is an operator, pop two elements from the stack, apply the operator to those elements, and push the result back onto the stack.

Step-by-Step Example for Postfix Expression: "5 6 2 + * 12 4 / -"

Initial state:

Stack: [] (empty)

Step 1: Process element 5

- Character: 5
    - 5 is an operand, so push it onto the stack.
- Stack after Step 1: [5]

Step 2: Process element 6

- Character: 6
    - 6 is an operand, so push it onto the stack.
- Stack after Step 2: [5, 6]

Step 3: Process element 2

- Character: 2
    - 2 is an operand, so push it onto the stack.
- Stack after Step 3: [5, 6, 2]

Step 4: Process operator +

- Character: + (addition)
    - + is an operator, so pop the top two elements from the stack (6 and 2).
    - Perform 6 + 2 = 8.
    - Push the result (8) onto the stack.
- Stack after Step 4: [5, 8]

Step 5: Process operator *

- Character: * (multiplication)
    - is an operator, so pop the top two elements from the stack (5 and 8).
    - Perform 5 * 8 = 40.
    - Push the result (40) onto the stack.
- Stack after Step 5: [40]

Step 6: Process element 12

- Character: 12
    - 12 is an operand, so push it onto the stack.
- Stack after Step 6: [40, 12]

Step 7: Process element 4

- Character: 4
    - 4 is an operand, so push it onto the stack.
- Stack after Step 7: [40, 12, 4]

Step 8: Process operator /

- Character: / (division)
    - / is an operator, so pop the top two elements from the stack (12 and 4).
    - Perform 12 / 4 = 3.
    - Push the result (3) onto the stack.
- Stack after Step 8: [40, 3]

Step 9: Process operator -

- Character: - (subtraction)
    - - is an operator, so pop the top two elements from the stack (40 and 3).
    - Perform 40 - 3 = 37.
    - Push the result (37) onto the stack.
- Stack after Step 9: [37]

Final Result

After processing all elements, the stack contains one element, which is the result of the postfix expression.

Final Stack State: [37]

The result of the postfix expression "5 6 2 + * 12 4 / -" is **37**.

General Algorithm:

1. Create an empty stack.

2. Traverse the postfix expression character by character:

- If the character is an operand (number), push it onto the stack.
- If the character is an operator:
    - Pop the top two elements from the stack.
    - Apply the operator on these two operands.
    - Push the result back onto the stack.

3. Final Step:

- After traversing the entire postfix expression, the value at the top of the stack is the result.

Edge Cases:

- Single operand: If the expression contains only one operand (e.g., "5"), the result is that operand.
- Invalid expression: If the expression is malformed (e.g., missing operators or operands), proper error handling should be implemented.

Example 2:

Input: "3 4 + 2 * 7 /"

This expression corresponds to ((3 + 4) * 2) / 7.

Steps:

Process 3 → push → [3]

Process 4 → push → [3, 4]

Process + → pop 3 and 4, compute 3 + 4 = 7 → push 7 → [7]

Process 2 → push → [7, 2]

Process * → pop 7 and 2, compute 7 * 2 = 14 → push 14 → [14]

Process 7 → push → [14, 7]

Process / → pop 14 and 7, compute 14 / 7 = 2 → push 2 → [2]

Final result: 2

Java Code:

```java
import java.util.Stack;

public class PostfixEvaluation {
    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();

        for (char ch : expression.toCharArray()) {
            if (Character.isDigit(ch)) {
                stack.push(ch - '0');  // convert char to int
            } else {
                int val1 = stack.pop();
                int val2 = stack.pop();
                switch (ch) {
                    case '+':
                        stack.push(val2 + val1);
                        break;
                    case '-':
                        stack.push(val2 - val1);
                        break;
                    case '*':
                        stack.push(val2 * val1);
                        break;
                    case '/':
                        stack.push(val2 / val1);
                        break;
                }
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        String expression = "562+*124/-";
        System.out.println("Postfix Evaluation: " + evaluatePostfix(expression)); // Output: 37
    }
}
```

CPP Code:

```cpp
#include <string>
using namespace std;

int evaluatePostfix(string expression) {
    stack<int> stack;

    for (char ch : expression) {
        if (isdigit(ch)) {
            stack.push(ch - '0');  // convert char to int
        } else {
            int val1 = stack.top(); stack.pop();
            int val2 = stack.top(); stack.pop();
            switch (ch) {
                case '+':
                    stack.push(val2 + val1);
                    break;
                case '-':
                    stack.push(val2 - val1);
                    break;
                case '*':
                    stack.push(val2 * val1);
                    break;
                case '/':
                    stack.push(val2 / val1);
                    break;
            }
        }
    }
    return stack.top();
}

int main() {
    string expression = "562+*124/-";
    cout << "Postfix Evaluation: " << evaluatePostfix(expression) << endl;  // Output: 37
    return 0;
}
```

4. Sort a Stack Using Recursion

Problem Statement:

You are given a stack that holds integers. The goal is to sort the stack in ascending order using recursion. You are not allowed to use any loop structures (e.g., for, while) or any additional stack or array.

Approach:

We will use recursion to remove elements one by one, sort the remaining stack, and then insert the removed element back in the correct position to maintain the order.

High-Level Steps:

1. **Remove the top element** from the stack recursively until the stack is empty.

2. **Sort the remaining stack** recursively.

3. Once the stack is sorted, **insert the removed element** back in the correct position to maintain order.

Key Recursive Operations:

- **Recursive Sort:** This function will pop the top element and recursively sort the rest of the stack.
- **Recursive Insert:** After sorting, this function will insert the popped element back into the correct position.

Example:

Let's say we have a stack: [3, 1, 4, 2].

We will sort this stack in ascending order.

Steps:

Step 1: Sort Stack [3, 1, 4, 2]

- Pop 2 → Remaining Stack: [3, 1, 4]
- Recursively call sort on [3, 1, 4]

Step 2: Sort Stack [3, 1, 4]

- Pop 4 → Remaining Stack: [3, 1]
- Recursively call sort on [3, 1]

Step 3: Sort Stack [3, 1]

- Pop 1 → Remaining Stack: [3]
- Recursively call sort on [3]

Step 4: Sort Stack [3]

- Base case: The stack has only one element, so it’s already sorted.
- Start inserting the elements back.

Insert Step-by-Step:

1. Insert 1 back into the stack [3].

- Compare 1 with 3.
- Insert 1 below 3 → Stack: [1, 3]

2. Insert 4 back into the stack [1, 3].

- Compare 4 with 3.
- Insert 4 on top of 3 → Stack: [1, 3, 4]

3.Insert 2 back into the stack [1, 3, 4].

- Compare 2 with 4 and 3.
- Insert 2 between 1 and 3 → Stack: [1, 2, 3, 4]

Detailed Step-by-Step Example:

1. Sort stack [3, 1, 4, 2]:

- Pop 2 → [3, 1, 4]
- Recursively sort [3, 1, 4]

2. Sort stack [3, 1, 4]:

- Pop 4 → [3, 1]
- Recursively sort [3, 1]

3. Sort stack [3, 1]:

- Pop 1 → [3]
- Recursively sort [3]

4. Sort stack [3]:

- Base case (already sorted)

5. Insert 1:

- Compare with 3, insert below 3 → [1, 3]

6. Insert 4:

- Insert 4 on top → [1, 3, 4]

7. Insert 2:

- Compare with 4, then 3, insert between 1 and 3 → [1, 2, 3, 4]

Algorithm:

Recursive Sort:

1. Base case: If the stack has one or zero elements, it is already sorted.

2. Pop the top element.

3. Recursively call sortStack() on the remaining stack.

4. Insert the popped element back in the sorted stack using a helper function insertInSortedOrder().

Recursive Insert:

1. Base case: If the stack is empty or the top of the stack is smaller than the element to insert, push the element.

2. Otherwise, pop the top element, recursively insert the element in the remaining stack, and then push the popped element back.

Java Code:

```java
import java.util.Stack;

public class SortStackUsingRecursion {

    // Method to sort the stack
    public static void sortStack(Stack<Integer> stack) {
        // Base case: If stack is empty, return
        if (stack.isEmpty()) {
            return;
        }

        // Pop the top element
        int top = stack.pop();

        // Recursively sort the remaining stack
        sortStack(stack);

        // Insert the popped element back into the sorted stack
        insertInSortedOrder(stack, top);
    }

    // Method to insert an element in the sorted order
    public static void insertInSortedOrder(Stack<Integer> stack, int element) {
        // Base case: If stack is empty or top of stack is smaller than element, push the element
        if (stack.isEmpty() || stack.peek() <= element) {
            stack.push(element);
            return;
        }

        // Pop the top element and insert the current element in sorted order
        int top = stack.pop();
        insertInSortedOrder(stack, element);

        // Push the popped element back
        stack.push(top);
    }

```

```
37
38      public static void main(String[] args) {
39          Stack<Integer> stack = new Stack<>();
40          stack.push(3);
41          stack.push(1);
42          stack.push(4);
43          stack.push(2);
44
45          System.out.println("Original Stack: " + stack);
46          sortStack(stack);
47          System.out.println("Sorted Stack: " + stack);
48      }
49  }
50
```

CPP Code:

```cpp
#include <iostream>
#include <stack>
using namespace std;

// Function to insert an element in sorted order
void insertInSortedOrder(stack<int> &s, int element) {
    // Base case: If stack is empty or top of stack is smaller than element
    if (s.empty() || s.top() <= element) {
        s.push(element);
        return;
    }

    // Pop the top element
    int top = s.top();
    s.pop();

    // Recursively insert the current element in sorted order
    insertInSortedOrder(s, element);

    // Push the popped element back
    s.push(top);
}

// Function to sort the stack using recursion
void sortStack(stack<int> &s) {
    // Base case: If stack is empty, return
    if (s.empty()) {
        return;
    }

    // Pop the top element
    int top = s.top();
    s.pop();

    // Recursively sort the remaining stack
    sortStack(s);

    // Insert the popped element back into the sorted stack
    insertInSortedOrder(s, top);
}
```

```
int main() {
    stack<int> s;
    s.push(3);
    s.push(1);
    s.push(4);
    s.push(2);

    cout << "Original Stack: ";
    stack<int> temp = s; // Create a temporary stack to display original stack
    while (!temp.empty()) {
        cout << temp.top() << " ";
        temp.pop();
    }
    cout << endl;

    sortStack(s);

    cout << "Sorted Stack: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;

    return 0;
}
```