**1. What is a Stack?**

**Definition:**

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. This means that the last element added (pushed) to the stack will be the first one to be removed (popped) from it.

**LIFO Property Example:**

Imagine a stack of plates:

* When you add a plate to the stack, you place it on top.
* When you remove a plate, you take the one from the top first.

If you push the plates in the order: Plate 1, Plate 2, Plate 3:

* The stack looks like this:

css

Copier le code

Top → Plate 3

Plate 2

Plate 1

* If you pop one plate, you'll get Plate 3, the most recently added plate, demonstrating the LIFO behavior.

**2. Stack Operations**

The basic operations that can be performed on a stack include:

1. **Push (element)**:
   * Adds an

4o

**Vous avez dit :**

all quetion

**ChatGPT a dit :**

ChatGPT

Let's go through each of your questions in detail:

**1. What is a Stack?**

**Definition:**

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. This means that the last element added (pushed) to the stack will be the first one to be removed (popped) from it.

**LIFO Property Example:**

Imagine a stack of books:

* When you add (push) a book to the stack, it goes on top.
* When you remove (pop) a book, you take the one on top first.

Example:

* Push elements in order: Book 1, Book 2, Book 3

css

Copier le code

Top → Book 3

Book 2

Book 1

* If you pop, you'll get Book 3, showing the LIFO behavior.

**2. Stack Operations**

Here are the basic operations that can be performed on a stack:

1. **Push (element)**:
   * **Description**: Adds an element to the top of the stack.
   * **Time Complexity**: O(1)O(1)O(1)
2. **Pop ()**:
   * **Description**: Removes the top element from the stack.
   * **Time Complexity**: O(1)O(1)O(1)
3. **Peek/Top ()**:
   * **Description**: Returns the top element of the stack without removing it.
   * **Time Complexity**: O(1)O(1)O(1)
4. **isEmpty ()**:
   * **Description**: Checks if the stack is empty.
   * **Time Complexity**: O(1)O(1)O(1)
5. **Size ()**:
   * **Description**: Returns the number of elements in the stack.
   * **Time Complexity**: O(1)O(1)O(1)

**3. Applications of Stacks**

Stacks are widely used in computer science and real-world applications. Here are three practical uses:

1. **Function Call Management (Call Stack)**:
   * **Example**: When a function is called, its data (like parameters and return address) is pushed onto the call stack. When the function finishes, this data is popped off. This is crucial for managing nested or recursive function calls.
2. **Expression Evaluation (Infix, Postfix, Prefix)**:
   * **Example**: Converting infix expressions (like A + B \* C) to postfix (like A B C \* +) uses a stack for operators to ensure correct order of operations.
3. **Undo Mechanism in Applications**:
   * **Example**: In text editors, the undo feature uses a stack to keep track of recent actions. The last action performed is the first to be undone.

**4. Stack Overflow and Underflow**

* **Stack Overflow**:
  + Occurs when you try to push an element onto a stack that is already full (exceeding its capacity).
  + **Example**: In systems with limited stack size, like embedded systems or recursive function calls in programming, if you push too many items or recurse too deeply, you get a stack overflow.
  + **Solution**: Ensure stack size checks before pushing elements or use a dynamically resizing stack if possible.
* **Stack Underflow**:
  + Occurs when you try to pop an element from an empty stack.
  + **Example**: Attempting to undo an action when there are no more actions to undo.
  + **Solution**: Always check if the stack is empty before performing a pop operation.

**5. Converting Infix to Postfix**

To convert an infix expression (like A + B \* C) to a postfix expression (like A B C \* +), we use the **Shunting Yard Algorithm** with a stack for operators.

**Algorithm Steps:**

1. **Initialize**:
   * Use an empty stack for operators.
   * Use an output list for the postfix expression.
2. **Process each character**:
   * If it's an **operand** (like A, B, C), add it to the output.
   * If it's an **operator** (like +, \*):
     + Pop operators from the stack to the output until the stack is empty or an operator with lower precedence is found.
     + Push the current operator onto the stack.
   * If it's a **left parenthesis** (, push it onto the stack.
   * If it's a **right parenthesis** ), pop operators to the output until a left parenthesis is found. Remove the left parenthesis from the stack.
3. **After processing all characters**:
   * Pop any remaining operators from the stack to the output.

**Example:**

Convert A + B \* C to postfix:

* Input: A + B \* C
* Output: A B C \* +

**Step-by-step**:

1. A → Output: A
2. + → Stack: +
3. B → Output: A B
4. \* → Stack: + \* (since \* has higher precedence than +)
5. C → Output: A B C
6. Pop stack → Output: A B C \* +

**6. Balanced Parentheses Problem**

To check if parentheses are balanced in an expression (like {[()]}), we can use a stack.

**What does it mean for parentheses to be balanced?**

* Every opening bracket ((, {, [) must have a corresponding closing bracket (), }, ]) in the correct order.

**Algorithm:**

1. **Initialize** an empty stack.
2. **Process each character** in the expression:
   * If it's an **opening bracket** ((, {, [), push it onto the stack.
   * If it's a **closing bracket** (), }, ]):
     + Check if the stack is empty (if yes, it's unbalanced).
     + Otherwise, pop the stack and check if the popped bracket matches the current closing bracket. If not, it's unbalanced.
3. After processing all characters, if the stack is empty, the parentheses are balanced. Otherwise, they are unbalanced.

**Example:**

Check if {[()()]} is balanced:

* Step-by-step:
  1. { → Stack: {
  2. [ → Stack: { [
  3. ( → Stack: { [ (
  4. ) → Stack: { [ (match found)
  5. ( → Stack: { [ (
  6. ) → Stack: { [ (match found)
  7. ] → Stack: { (match found)
  8. } → Stack: (match found)
* Result: Balanced

This concludes all the questions. Let me know if you need further clarification on any part!

**7. Evaluating Postfix Expressions**

**How a Stack is Used to Evaluate a Postfix Expression:**

A postfix expression (also known as Reverse Polish Notation) is a mathematical expression where the operator follows the operands. For example, the postfix expression 3 4 + 2 \* 7 / is equivalent to the infix expression ((3 + 4) \* 2) / 7.

**Algorithm to Evaluate Postfix Expressions:**

1. **Initialize** an empty stack.
2. **Iterate** over each token in the postfix expression.
   * **If the token is an operand (number)**, push it onto the stack.
   * **If the token is an operator**, pop two elements from the stack, apply the operator to them, and push the result back onto the stack.
3. After iterating through the entire expression, the stack should contain exactly one element, which is the result.

**Step-by-Step Example:** Postfix Expression: 3 4 + 2 \* 7 /

| **Token** | **Stack Operation** | **Stack State** |
| --- | --- | --- |
| 3 | Push 3 | [3] |
| 4 | Push 4 | [3, 4] |
| + | Pop 4, 3 → Compute 3 + 4 = 7 | [7] |
| 2 | Push 2 | [7, 2] |
| \* | Pop 2, 7 → Compute 7 \* 2 = 14 | [14] |
| 7 | Push 7 | [14, 7] |
| / | Pop 7, 14 → Compute 14 / 7 = 2 | [2] |

**Result:** The final value on the stack is 2.

**8. Stack as a Memory Model**

**Role of the Call Stack in Function Calls:**

* The **call stack** is a special type of stack used by a program to manage function calls, local variables, and return addresses.

**How the Stack Manages Function Calls:**

1. When a function is called, a **stack frame** is created for it.
   * The stack frame includes the function’s parameters, local variables, and the address to return to once the function completes.
2. The stack frame is pushed onto the call stack.
3. When the function returns, its stack frame is popped off the stack, and control goes back to the return address.

**Managing Recursion:**

* Each recursive call adds a new stack frame, which allows each function call to have its own set of variables and return addresses.
* If recursion is too deep, it may lead to a **stack overflow**.

**9. Reverse a String Using Stack**

**Why Use a Stack to Reverse a String?**

* A stack is a Last In, First Out (LIFO) data structure, which means that the last element added to the stack is the first one to be removed. This property makes it ideal for reversing the order of characters in a string.

**Algorithm to Reverse a String Using a Stack:**

1. **Push** each character of the string onto the stack.
2. **Pop** characters from the stack and append them to a new string.
3. The new string is the reversed version of the original string.

**Example:** Original String: "hello"

| **Step** | **Stack State** |
| --- | --- |
| Push 'h' | [h] |
| Push 'e' | [h, e] |
| Push 'l' | [h, e, l] |
| Push 'l' | [h, e, l, l] |
| Push 'o' | [h, e, l, l, o] |

Pop all elements: "olleh"

**10. Stack in Depth-First Search (DFS)**

**DFS Using a Stack:**

* In DFS, nodes are explored as deep as possible along one branch before backtracking.
* This can be implemented using either a recursive function or an explicit stack.

**Algorithm for Stack-Based DFS:**

1. **Initialize** a stack and push the starting node.
2. **While the stack is not empty**:
   * Pop a node from the stack.
   * If it hasn’t been visited, mark it as visited.
   * Push all its unvisited neighbors onto the stack.

**Comparison with Recursive DFS:**

* **Stack-Based DFS** uses an explicit stack, which avoids the risk of a stack overflow from too many recursive calls.
* **Recursive DFS** uses the call stack implicitly, which may consume more memory for deep recursions.

**Practical Problems (5 Coding Questions)**

**1. Implement a Stack Using Arrays (or Linked List)**

c

Copier le code

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

int data[MAX];

int top;

} Stack;

void init(Stack \*s) {

s->top = -1;

}

int is\_empty(Stack \*s) {

return s->top == -1;

}

int is\_full(Stack \*s) {

return s->top == MAX - 1;

}

void push(Stack \*s, int value) {

if (is\_full(s)) {

printf("Stack Overflow\n");

return;

}

s->data[++(s->top)] = value;

}

int pop(Stack \*s) {

if (is\_empty(s)) {

printf("Stack Underflow\n");

return -1;

}

return s->data[(s->top)--];

}

int peek(Stack \*s) {

if (is\_empty(s)) {

printf("Stack is Empty\n");

return -1;

}

return s->data[s->top];

}

int main() {

Stack s;

init(&s);

push(&s, 10);

push(&s, 20);

printf("Top Element: %d\n", peek(&s));

pop(&s);

printf("Top Element after pop: %d\n", peek(&s));

return 0;

}

**2. Parenthesis Matching (Balanced Parentheses)**

c

Copier le code

#include <stdio.h>

#include <string.h>

int is\_balanced(const char \*expr) {

char stack[100];

int top = -1;

for (int i = 0; i < strlen(expr); i++) {

char ch = expr[i];

if (ch == '(' || ch == '{' || ch == '[') {

stack[++top] = ch;

} else if (ch == ')' || ch == '}' || ch == ']') {

if (top == -1) return 0;

char last = stack[top--];

if ((ch == ')' && last != '(') ||

(ch == '}' && last != '{') ||

(ch == ']' && last != '[')) {

return 0;

}

}

}

return top == -1;

}

int main() {

printf(is\_balanced("({[()]})") ? "Balanced\n" : "Unbalanced\n");

printf(is\_balanced("{[([))]}") ? "Balanced\n" : "Unbalanced\n");

return 0;

}

**3. Infix to Postfix Conversion**

Implementing this requires operator precedence and a stack to handle the conversion, which can be tackled separately if needed. Let me know if you'd like a solution for this or other questions.

**1. Infix to Postfix Conversion**

To convert an infix expression (like "A + B \* (C - D)") to a postfix expression (like "A B C D - \* +"), we can use the **Shunting Yard Algorithm** by Edsger Dijkstra. The algorithm uses a stack to handle operators and parentheses.

Here's the implementation:

python

Copier le code

def infix\_to\_postfix(expression):

precedence = {'+': 1, '-': 1, '\*': 2, '/': 2}

stack = []

postfix = []

for char in expression:

if char.isalnum(): # if character is an operand (letter or number)

postfix.append(char)

elif char == '(':

stack.append(char)

elif char == ')':

while stack and stack[-1] != '(':

postfix.append(stack.pop())

stack.pop() # remove '(' from stack

else: # operator

while stack and stack[-1] != '(' and precedence[char] <= precedence[stack[-1]]:

postfix.append(stack.pop())

stack.append(char)

while stack:

postfix.append(stack.pop())

return ' '.join(postfix)

# Example usage

expression = "A+B\*(C-D)"

print("Infix:", expression)

print("Postfix:", infix\_to\_postfix(expression))

**Output:**

mathematica

Copier le code

Infix: A+B\*(C-D)

Postfix: A B C D - \* +

**2. Evaluate a Postfix Expression**

To evaluate a postfix expression (like "3 4 + 2 \* 7 /"), we use a stack to handle operands and operators. The idea is to push numbers onto the stack and, when an operator is encountered, pop the top two numbers, apply the operator, and push the result back onto the stack.

Here's the implementation:

python

Copier le code

def evaluate\_postfix(expression):

stack = []

for token in expression.split():

if token.isdigit(): # If the token is an operand (integer)

stack.append(int(token))

else: # If the token is an operator

b = stack.pop()

a = stack.pop()

if token == '+':

stack.append(a + b)

elif token == '-':

stack.append(a - b)

elif token == '\*':

stack.append(a \* b)

elif token == '/':

stack.append(a // b) # integer division

return stack[0]

# Example usage

postfix\_expression = "3 4 + 2 \* 7 /"

print("Postfix Expression:", postfix\_expression)

print("Evaluation Result:", evaluate\_postfix(postfix\_expression))

**Output:**

mathematica

Copier le code

Postfix Expression: 3 4 + 2 \* 7 /

Evaluation Result: 2

**3. Reverse a Stack Using Recursion**

To reverse a stack using recursion, we can break down the problem into two parts:

* Remove the bottom element of the stack and store it.
* Reverse the remaining stack.
* Insert the removed element back at the top of the stack.

Here's the implementation:

python

Copier le code

def insert\_at\_bottom(stack, item):

if not stack:

stack.append(item)

else:

top = stack.pop()

insert\_at\_bottom(stack, item)

stack.append(top)

def reverse\_stack(stack):

if stack:

top = stack.pop()

reverse\_stack(stack)

insert\_at\_bottom(stack, top)

# Example usage

stack = [1, 2, 3, 4]

print("Original Stack:", stack)

reverse\_stack(stack)

print("Reversed Stack:", stack)

**Output:**

less

Copier le code

Original Stack: [1, 2, 3, 4]

Reversed Stack: [4, 3, 2, 1]

**Explanation:**

1. **Infix to Postfix Conversion**:
   * Use a stack for operators and manage precedence using a dictionary.
   * Push operands directly to the output.
   * Handle parentheses by pushing ( to the stack and popping until ).
2. **Evaluate a Postfix Expression**:
   * Use a stack to store operands.
   * On encountering an operator, pop the last two operands, apply the operator, and push the result back.
3. **Reverse a Stack Using Recursion**:
   * Recursively remove elements until the stack is empty.
   * Use another recursive function to insert removed elements back to the bottom of the stack.

These solutions efficiently solve each problem with time complexities:

* Infix to Postfix: O(n)O(n)O(n)
* Evaluate Postfix: O(n)O(n)O(n)
* Reverse Stack: O(n2)O(n^2)O(n2) (due to recursion and insert at bottom function).