DATA STRUCTURE

Chapter 3

STACKS

Study Notes for Engineering Students

(Diploma / B.Tech — Computer Science & Engineering)

Topics Covered

3.1 Introduction to Stacks

A stack is one of the most fundamental linear data structures used in computer science. It is a collection of elements arranged in a particular sequential order, where insertion and deletion of elements take place only at one end, commonly referred to as the "top" of the stack. Because of this restricted access, a stack follows the Last In First Out (LIFO) principle — the element inserted last is the first one to be removed.

Stacks can be understood intuitively through several real-life analogies. Consider a stack of plates kept on top of one another: a new plate is always placed on top, and the plate that is removed first is also the one on top. Similarly, a pile of books, a stack of coins, or a magazine rack all illustrate the LIFO behaviour. In the context of software, the "Undo" feature of an editor, the back button of a web browser, and the way function calls are managed by a program are all practical implementations of the stack concept.

Key Terminology

- Top — a pointer or index that always refers to the most recently inserted element of the stack.
- Push — the operation of inserting a new element onto the top of the stack.
- Pop — the operation of removing the element currently at the top of the stack.
- Overflow — the condition that occurs when an attempt is made to push an element into a stack that is already full.
- Underflow — the condition that occurs when an attempt is made to pop an element from a stack that is already empty.

Because insertions and deletions happen only from one end, a stack is also described as a restricted or order-preserving data structure, unlike arrays or linked lists where elements can generally be accessed or modified from any position.

3.2 Stacks as an Abstract Data Type

An Abstract Data Type (ADT) is a mathematical or logical model of a data structure that specifies the type of data stored and the operations that can be performed on it, without specifying how those operations are actually implemented. The ADT view separates "what" a data structure does from "how" it does it, which allows the same logical structure to be realised through different underlying implementations, such as arrays or linked lists.

When a stack is viewed as an ADT, it is defined purely in terms of its behaviour and the set of operations it supports, rather than in terms of memory layout or programming language constructs. This abstraction is important in engineering practice because it allows a programmer to use a stack through a well-defined interface while the internal implementation can be changed later without affecting the rest of the program.

Stack ADT Specification

The Stack ADT can be specified as a collection of elements together with the following abstract operations:

- Push(item) — inserts item onto the top of the stack.
- Pop() — removes and returns the element at the top of the stack.
- Peek() / Top() — returns the element at the top of the stack without removing it.
- isEmpty() — returns true if the stack contains no elements.

- isFull() — returns true if the stack has reached its maximum capacity (relevant mainly for array-based implementations).
- Size() — returns the current number of elements present in the stack.

Each of these operations is defined only by its effect on the logical state of the stack (that is, the ordered sequence of elements) and by its pre-conditions and post-conditions, not by any particular code.

3.3 Primitive Operations of Stacks

The primitive operations of a stack are the basic building-block operations upon which every higher-level use of a stack is constructed. All of these operations execute in constant time, O(1), because they only involve accessing or modifying the top of the stack.

(a) PUSH Operation

The push operation inserts a new element at the top of the stack. Before inserting, the algorithm must check whether the stack is already full (overflow condition).

```
Algorithm PUSH(STACK, TOP, MAXSIZE, ITEM)
Step 1: If TOP = MAXSIZE - 1, then
          Print "STACK OVERFLOW"
          Exit
Step 2: Set TOP = TOP + 1
Step 3: Set STACK[TOP] = ITEM
Step 4: Exit
```

(b) POP Operation

The pop operation removes and returns the element currently at the top of the stack. Before removing, the algorithm must check whether the stack is already empty (underflow condition).

```
Algorithm POP(STACK, TOP)
Step 1: If TOP = -1, then
          Print "STACK UNDERFLOW"
          Exit
Step 2: Set ITEM = STACK[TOP]
Step 3: Set TOP = TOP - 1
Step 4: Return ITEM
```

(c) PEEK / TOP Operation

The peek (or top) operation returns the value of the element at the top of the stack without deleting it, so the state of the stack remains unchanged.

```
Algorithm PEEK(STACK, TOP)
Step 1: If TOP = -1, then
          Print "STACK IS EMPTY"
          Exit
Step 2: Return STACK[TOP]
```

(d) isEmpty and isFull

- isEmpty() returns TRUE when TOP = -1, indicating that the stack contains no elements.
- isFull() returns TRUE when TOP = MAXSIZE - 1, indicating that no more elements can be pushed (applicable to array-based stacks).

Together, PUSH, POP, PEEK, isEmpty and isFull form the minimal, primitive operation set from which every stack-based application described later in this chapter is built.

3.4 Representation of Stacks through Arrays

The simplest and most common way to implement a stack is by using a one-dimensional array along with a variable called TOP, which keeps track of the index of the most recently inserted element. The array is declared with a fixed maximum size, MAXSIZE, decided in advance, and TOP is initialised to -1 to indicate an empty stack.

Structure

An array-based stack maintains: (i) an array STACK[0...MAXSIZE-1] to hold the elements, and (ii) an integer variable TOP that points to the index of the current top element. When TOP = -1, the stack is empty; when TOP = MAXSIZE - 1, the stack is full.

Push and Pop using Arrays

To push an element, TOP is first incremented and the new item is then stored at STACK[TOP], provided the stack is not already full. To pop an element, the value at STACK[TOP] is read and TOP is then decremented, provided the stack is not already empty. These are exactly the primitive PUSH and POP algorithms described in Section 3.3, applied to a contiguous array.

Advantages

- Simple to implement and easy to understand.
- Elements are stored in contiguous memory locations, giving fast access and good cache performance.
- No extra memory overhead for pointers, unlike a linked list.

Disadvantages

- The size of the stack must be fixed in advance, which can lead to either overflow (if too small) or wastage of memory (if too large).
- Resizing the array during execution is expensive, as it typically requires creating a new array and copying all existing elements.

3.5 Representation of Stacks through Linked List

A stack can also be implemented using a singly linked list, which overcomes the fixed-size limitation of the array representation. In this approach, each element of the stack is stored in a node containing two fields: a DATA field holding the actual value, and a NEXT field holding the address of the next node in the list. A separate pointer, TOP, always points to the first node of the linked list, which represents the top of the stack.

Push Operation (Linked List)

To push a new element, a new node is created, its DATA field is set to the new value, its NEXT field is made to point to the current TOP, and TOP is then updated to point to this new node. Insertion always happens at the head of the list, so it requires no traversal and takes constant time.

```
Algorithm PUSH_LL(TOP, ITEM)
Step 1: Create a new node NEW
Step 2: Set NEW.DATA = ITEM
Step 3: Set NEW.NEXT = TOP
Step 4: Set TOP = NEW
Step 5: Exit
```

Pop Operation (Linked List)

To pop an element, the DATA of the node currently pointed to by TOP is saved, TOP is advanced to TOP.NEXT, and the old top node is then freed/deleted.

```
Algorithm POP_LL(TOP)
Step 1: If TOP = NULL, then
          Print "STACK UNDERFLOW"
          Exit
Step 2: Set ITEM = TOP.DATA
Step 3: Set TEMP = TOP
Step 4: Set TOP = TOP.NEXT
Step 5: Delete TEMP
Step 6: Return ITEM
```

Advantages

- The size of the stack grows and shrinks dynamically, so there is no fixed upper limit apart from available memory.
- There is no wastage of memory, as space is allocated only when an element is actually pushed.

Disadvantages

- Each node requires extra memory to store the NEXT pointer, in addition to the data.
- Elements are not stored in contiguous memory, so there is no direct/random access and slightly poorer cache performance compared to arrays.

3.6 Applications of Stacks

The LIFO behaviour of stacks makes them extremely useful in a wide range of problems in computer science, particularly wherever the most recently processed item must be handled first, or wherever a reversible sequence of actions must be remembered.

(a) Expression Evaluation and Conversion

Stacks are used extensively to evaluate arithmetic expressions and to convert expressions between infix, postfix (Reverse Polish) and prefix notations. In postfix evaluation, operands are pushed onto a stack, and whenever an operator is encountered, the required number of operands are popped, the operation is applied, and the result is pushed back. Infix-to-postfix conversion uses a stack to temporarily hold operators according to their precedence and associativity.

(b) Balancing and Matching of Symbols

Stacks are used to check whether the parentheses, brackets and braces in an expression or in program source code are properly balanced and correctly nested. Opening symbols are pushed onto the stack, and when a closing symbol is encountered, the stack is popped and matched; a mismatch or a non-empty stack at the end indicates unbalanced symbols.

(c) Function Call Management

Every time a function or method is called, the computer system pushes an activation record (also called a stack frame) containing the return address, parameters and local variables onto a special memory area known as the call stack. When the function finishes execution, its activation record is popped and control returns to the calling function. This mechanism is discussed further in Section 3.7.

(d) Undo / Redo Operations

Word processors, image editors, and many other interactive applications maintain a stack of previously performed actions. Choosing "Undo" pops the most recent action off the stack and reverses it, while "Redo" typically uses a second stack to reapply undone actions.

(e) Backtracking Algorithms

Problems such as maze solving, the N-Queens problem, and Depth-First Search (DFS) traversal of graphs and trees rely on backtracking, where the algorithm must be able to return to a previous decision point when the current path fails. A stack (either explicit or the implicit recursion call stack) is used to remember these decision points.

(f) Other Applications

- Maintaining the browsing history in a web browser (the "Back" button).
- Syntax analysis (parsing) in compilers and interpreters.
- Reversing a string, a list, or the digits of a number.
- Managing memory allocation for local variables and recursive calls at the system level.

3.7 Stack and Recursion

Recursion is a programming technique in which a function calls itself, either directly or indirectly, in order to solve a problem by breaking it down into smaller sub-problems of the same type. Every recursive program, whether the programmer realises it or not, is intimately tied to the stack data structure, because the underlying execution environment uses a call stack to keep track of pending function calls.

The Call Stack and Activation Records

Whenever a function is called, an activation record (stack frame) is pushed onto the call stack. This record stores the function's parameters, its local variables, and the return address — the point in the program to which control should return once the function finishes. When a function returns, its activation record is popped from the stack, and execution resumes at the stored return address, using the values in the activation record of the function now at the top of the stack.

In a recursive function, each recursive call results in a new activation record being pushed on top of the previous one, even though the calls are all to the same function. This is why the stack grows as deep as the number of pending (unfinished) recursive calls, and shrinks again as each call completes and returns.

Illustration: Factorial Function

Consider the recursive definition of factorial, fact(n) = n * fact(n - 1), with fact(0) = 1. A call to fact(4) proceeds as follows:

1. fact(4) calls fact(3); an activation record for fact(4) is pushed, and it waits for the result of fact(3).
2. fact(3) calls fact(2); an activation record for fact(3) is pushed on top of fact(4)'s record.
3. fact(2) calls fact(1); an activation record for fact(2) is pushed.
4. fact(1) calls fact(0); an activation record for fact(1) is pushed.
5. fact(0) returns 1 directly (the base case); its activation record is popped.
6. Each waiting call then resumes in turn — fact(1) returns 1, fact(2) returns 2, fact(3) returns 6, and finally fact(4) returns 24 — popping one activation record from the stack at every return.

This trace shows clearly that the sequence of pending calls behaves exactly like a stack: the most recently made call (the innermost one) is the first to complete and be removed, which is precisely the LIFO property.

Recursion versus Iteration, and Explicit Stacks

Because recursion relies on the system's implicit call stack, deeply recursive functions can lead to stack overflow if the recursion depth becomes too large for the available stack memory. Any recursive algorithm can, in principle, be rewritten as an iterative algorithm that uses an explicit, programmer-managed stack to store exactly the information that would otherwise have been kept in the implicit activation records. This equivalence — that recursion can always be simulated by iteration plus an explicit stack — is an important concept for engineering students, since it links the abstract idea of recursion directly back to the concrete stack data structure studied throughout this chapter.

Multiple Choice Questions (MCQs)

Choose the most appropriate option for each of the following questions.

1. A stack follows which of the following principles?

a) FIFO (First In First Out)
b) LIFO (Last In First Out)
c) Random access
d) Priority based access

2. Which of the following operations is used to insert an element into a stack?

a) Insert
b) Push
c) Add
d) Append

3. The condition in which an element cannot be inserted into a stack because it is already full is called:

a) Underflow
b) Overflow
c) Deadlock
d) Collision

4. In an array-based stack, if TOP = -1, the stack is:

a) Full
b) Empty
c) Overflowing
d) Corrupted

5. Which operation returns the topmost element of a stack without deleting it?

a) Pop
b) Push
c) Peek
d) Delete

6. The time complexity of the PUSH and POP operations on a stack is:

a) O(1)
b) O(n)
c) O(log n)
d) O(n^2)

7. In a linked list implementation of a stack, a new element is normally inserted:

a) At the end of the list
b) At the middle of the list
c) At the beginning (head) of the list
d) At a random position

8. Which of the following is a disadvantage of the array representation of a stack?

a) Slow access to elements
b) Fixed size decided in advance
c) Requires extra pointer memory
d) Cannot store integers

9. Which of the following is an application of stacks?

a) Breadth-First Search
b) Balancing of parentheses
c) Sorting a linked list
d) Hashing

10. The memory area used by a program to manage function calls and activation records is called the:

a) Heap
b) Call stack
c) Data segment
d) Register file

11. Which notation is commonly evaluated directly using a single stack of operands?

a) Infix
b) Postfix
c) Mixed notation
d) Roman numeral notation

12. In the Stack ADT, the operation that reports whether the stack has no elements is:

a) isFull()
b) Size()
c) isEmpty()
d) Peek()

13. Excessive recursion depth can cause a program to fail due to:

a) Heap overflow
b) Stack overflow
c) Array underflow
d) Buffer overrun in the OS kernel

14. Any recursive algorithm can, in principle, be converted into an iterative one by using:

a) A queue
b) An explicit stack
c) A hash table
d) A binary tree

15. Which of the following best describes an Abstract Data Type (ADT)?

a) A specific programming language keyword
b) A model defined by its operations and behaviour, independent of implementation
c) A type of computer hardware
d) A fixed-size array only

Answer Key

1-b 2-b 3-b 4-b 5-c 6-a 7-c 8-b
9-b 10-b 11-b 12-c 13-b 14-b 15-b

Broad / Long Answer Type Questions

Answer the following questions in detail, with suitable diagrams and algorithms wherever applicable.

1. What is a stack? Explain the LIFO principle with the help of a suitable real-life example.

2. Define Stack as an Abstract Data Type. List and explain the standard operations specified by the Stack ADT.

3. Write the algorithms for the PUSH and POP operations on a stack, clearly stating the overflow and underflow conditions.

4. Explain the array representation of a stack. Discuss its advantages and disadvantages with a suitable diagram.

5. Explain the linked list representation of a stack. Write algorithms for PUSH and POP operations using a linked list.

6. Compare the array representation and the linked list representation of a stack on the basis of memory usage, size flexibility, and access time.

7. Discuss any four applications of stacks in computer science with brief explanations.

8. Explain how stacks are used to check whether the parentheses in a given expression are balanced. Illustrate with an example.

9. Explain the relationship between recursion and the stack data structure. Trace the execution of a recursive factorial function using a stack diagram for fact(4).

10. What is meant by stack overflow in the context of recursion? Explain how any recursive algorithm can be converted into an iterative one using an explicit stack.