# Formal Specification, Design and Implementation of the Stack Data Structure

M. Devi Prasad
School of Information Sciences, Manipal University.

devi.prasad@manipal.edu

**Abstract**

We present a formal specification of the *Stack* data structure. First, we describe the essential properties of the stack abstract data type. We subsequently develop a behavioral specification of the ADT. This step introduces a few design decisions that influence the representation and efficiency of operations on stacks. This, in turn, helps in shaping a concrete implementation in the C programming language. This entire exercise demonstrates how formal specification may guide different design choices, and thus lead to different implementations.

## 1 Introduction

Most textbooks employ a pattern in describing data structures: first informally describe the behavior of a particular data structure, and follow it up with an implementation in a high level programming language or pseudo code. The informal descriptions use diagrams, operation traces and textual narratives. The time and space complexity of algorithms, and implementation efficiency are discussed to some extent in many books. THe depth and breadth of exposition depends on factors such as author's predispositions and the target audience.

We wish to take a reasonably different approach to the way data structures course is taught. We wish to bring a little more engineering discipline to describing and developing data structures: we first bring out a specification, in some detail, of the interfaces of the underlying ADT. We follow it up with the design of the representation and associated behaviors of the data structure. We finally implement at the least two different versions in order to understand the pros and cons of different implementation strategies.

## 2 Notation and Conventions

We employ sets, sequences and tuples to develop an expressive formal specification of the Stack data structure. We use the following notation:

| | | |
|---|---|---|
| Sequence | $[a_1, ..., a_n]$ | *where* $0 \leq n$ |
| n-tuple | $(a_1, a_2, ..., a_n)$ | |

In the following sections, we will be using sequences to represent the content of stack instances. An empty stack is denoted by $[]$ and a non-empty stack is shown as $[a_1, ... a_n]$ where $a_n$ represents the top element of the stack.

At times, we will use the variable $\varepsilon$ to represent the empty sequence (or empty stack, in this case), and $\sigma$ to represent a generic sequence. We write $\#\sigma$ to mean the length of a sequence represented by $\sigma$. An empty sequence has zero length.

Finally, the *append* operator, **::**, appends an element to the end of the given list. Thus,

| | | | | | |
|---|---|---|---|---|---|
| if | $\sigma$ | = | $[a_1, ..., a_n]$ | and | $\#\sigma$ = n |
| then | $\sigma'$ | = | $\sigma$ **::** b = $[a_1, ..., a_n, b]$ | and | $\#\sigma'$ = n+1 |

Certain stack operations defined in the next section employ 2-tuples to represent the `Result` of their execution. A 2-tuple of type $\mathbb{R} \times \mathbb{N}$ is used to hold the value ($\mathbb{R}$ component) and the (potential error) status `OK`, `FULL`, and `EMPTY`.

## 3 The Specification of Stack and Result Interfaces

The stack interface specification includes a constructor function, three inspector functions, and two procedures that update the state of an instance. We define an auxiliary interface named ‘Result’ to represent either the answer or the error returned by certain operations.

### 3.1 The Stack Interface

While instantiating a stack object, we indicate the desired depth of the stack.

**new** : $\mathbb{N} \rightarrow$ Stack

The design as well as the implementation may choose to impose a limit on the depth of stack instances. The specified size or depth value will be stored as a part of stack’s internal representation.

Stack allows accessing the value of only the topmost element. An attempt to **push** an element may succeed if the stack has room for this element, or it may fail if the stack is full. Assuming that we will use our stacks to hold real numbers, this operation may be expressed so:

**push** : Stack $\times \mathbb{R} \rightarrow$ Stack $\times$ Result

When the stack is full, push operation records the status, FULL, in the ‘error‘ component of the Result tuple.

The **peek** operation reads the value on the top of the stack. When the content is not empty, the ‘data’ component of the Result tuple records the value on the stack-top. When the stack is empty, peek records the status, EMPTY, in the ‘error‘ component of the Result tuple.

**peek** : Stack $\rightarrow$ Stack $\times$ Result

When the stack is not empty, **pop** removes the topmost element from the stack. This value is not communicated to the caller. An empty stack indicates the failure of pop operation by setting the ‘error’ component of Result to EMPTY:

**pop** : Stack $\rightarrow$ Stack $\times$ Result

A couple of self-explanatory inspector functions are handy. Some programs may prefer to check the state of the stack before attempting stack mutation.

**empty** : Stack $\rightarrow$ Boolean

**full** : Stack $\rightarrow$ Boolean

### 3.2 The Result Interface

The **Result** interface has a constructor, and value and error component extractors:

**result** : $\mathbb{R} \times$ Status $\rightarrow$ Result

**val** : Result $\rightarrow \mathbb{R}$

**status** : Result $\rightarrow$ Status

**Status** is a set of three values representing the state after performing an operations on a stack instance:

Status $= \{OK, \; EMPTY, \; FULL\}$

## 4 An Abstract Specification of Stack Behavior

While instantiating a new stack, there is a provision to set the actual depth of the stack to a specific value (which is passed as an argument to the operation). If this value exceeds a limit defined by a particular implementation, the size is set to that limiting value (referred to as MAX_DEPTH below).

$$\mathbf{new}(\texttt{size}) = \begin{cases} (size, \varepsilon) & \text{if } \; 0 < size < \texttt{MAX\_DEPTH}. \\ (\texttt{MAX\_DEPTH}, \varepsilon) & \text{otherwise.} \end{cases}$$

A stack instance is represented as a 2-tuple. The first component represents the capacity of this stack instance. The second component is a sequence representing the actual contents of the stack instance. When a stack object is freshly instantiated, its content is an empty sequence.

The following invariants hold for stack instances:

$$0 \leq \#\sigma \leq \mathit{size} \leq \texttt{MAX\_DEPTH}$$

$$0 < \mathit{size} \leq \texttt{MAX\_DEPTH}$$

We define two state inspector operations, *empty* and *full*, by pattern-matching over the representation of stack instances. Note that a freshly created stack object has empty content.

$$\textbf{empty}(\texttt{stk}) = \begin{cases} \mathit{True} & \text{if } \texttt{stk} = (\mathit{size}, \varepsilon). \\ \mathit{False} & \text{otherwise.} \end{cases}$$

A stack instance is full when there is no room to hold a new element.

$$\textbf{full}(\texttt{stk}) = \begin{cases} \mathit{True} & \text{if } \texttt{stk} = (\mathit{size}, \sigma) \ \wedge \ 0 < \#\sigma = \mathit{size}. \\ \mathit{False} & \text{otherwise.} \end{cases}$$

Attempt to push a new element will succeed if the stack is not already full. This is indicated by the OK status component of the answer. In the other case where stack is full, the status FULL is returned.

$$\textbf{push}(\texttt{stk}, \texttt{d}) = \begin{cases} ((\mathit{size}, \sigma :: d), (\_, \mathit{OK})) & \text{if } \texttt{stk} = (\mathit{size}, \sigma) \ \wedge \ 0 \leq \#\sigma < \mathit{size}. \\ (\texttt{stk}, (\_, \mathit{FULL})) & \text{if } \texttt{stk} = (\mathit{size}, \sigma) \ \wedge \ 0 < \#\sigma = \mathit{size}. \end{cases}$$

The **pop** operation succeeds if the stack in not empty. Again, we should convince that equational reasoning holds if we plug **empty**, from its definition above, on the right side conditionals in the following:

$$\textbf{pop}(\texttt{stk}) = \begin{cases} ((\mathit{size}, \sigma), (d, \mathit{OK})) & \text{if } \texttt{stk} = (\mathit{size}, \sigma :: d) \ \wedge \ 0 \leq \#\sigma < \mathit{size}. \\ (\texttt{stk}, (\_, \mathit{EMPTY})) & \text{if } \texttt{stk} = (\mathit{size}, \varepsilon). \end{cases}$$

The **peek** operation only reads the value on the top of the stack. It is defined to return a value when the stack is not empty. Therefore, its preconditions are same as that of the **pop** operation. However, it does not mutate the state of the stack instance.

$$\textbf{peek}(\texttt{stk}) = \begin{cases} (\texttt{stk}, (d, \mathit{OK})) & \text{if } \texttt{stk} = (\mathit{size}, \sigma :: d) \ \wedge \ 0 \leq \#\sigma < \mathit{size}. \\ (\texttt{stk}, (\_, \mathit{EMPTY})) & \text{if } \texttt{stk} = (\mathit{size}, \varepsilon). \end{cases}$$

## 5 A Refinement

Based on the abstract definitions given in the previous section, we now derive more concrete versions of the same operations. We will call this a *general refinement*. This is not a formally rigorous approach to deriving more concrete implementations from more abstract specifications. Our methods depends mostly on a programmer's intuition to guide the process.

Popular imperative languages have little or no direct support for sequences and append operations that we used in the previous section. These languages support either safe or unsafe versions of array types. We shall assume here that we have at our disposal unsafe arrays with no direct support for bounds checking. Therefore, we shall use a representation that enforces safety of stack operations.

In the following refinement, a stack is represented by a 3-tuple; such a tuple is initialized when a fresh stack instance is created:

$$\textbf{new}(\texttt{size}) = \begin{cases} (\mathit{size}, [\,], -1) & \text{if } 0 < \mathit{size} < \texttt{MAX\_DEPTH}. \\ (\texttt{MAX\_DEPTH}, [\,], -1) & \text{otherwise.} \end{cases}$$

The first component of this 3-tuple holds the capacity of the stack object. The second component is an array store for the contents of the stack. The content of a newly created stack is empty. The third component represents the index of the topmost element of the stack. It is -1 when a stack object is instantiated.

The following invariants hold for stack instances:

$$-1 \le \mathit{top} < \mathit{size} \le \texttt{MAX\_DEPTH}$$

$$0 < \mathit{size} \le \texttt{MAX\_DEPTH}$$

We define two state inspector operations, *empty* and *full*, by pattern-matching over the representation structure of stack instances. It is appropriate, at this point, to note that a freshly created stack instance is empty.

$$\textbf{empty}(\texttt{stk}) = \begin{cases} \mathit{True} & \text{if } \texttt{stk} = (\mathit{size}, [\,], -1). \\ \mathit{False} & \text{otherwise.} \end{cases}$$

A stack instance is full when there is no room to hold a new element. Recall that the stack top always points to the most recently filled slot.

$$\textbf{full}(\texttt{stk}) = \begin{cases} \mathit{True} & \text{if } \texttt{stk} = (\mathit{size}, [a_0, \ldots, a_n], n) \ \wedge \ (n+1) = \mathit{size}. \\ \mathit{False} & \text{otherwise.} \end{cases}$$

Attempt to push a new element will succeed if the stack is not already full. This is indicated by the OK status component of the answer. In the other case where stack is full, the status FULL is returned.

$$\textbf{push}(\texttt{stk}, \texttt{d}) = \begin{cases} ((\mathit{size}, [a_0, \ldots, a_n, d], n+1), (\_, \mathit{OK})) & \text{if } \texttt{stk} = (\mathit{size}, [a_0, \ldots, a_n], n) \\ & \quad \wedge \ (n+1) < \mathit{size}. \\ (\texttt{stk}, \ (\_, \mathit{FULL})\ ) & \text{if } \texttt{stk} = (\mathit{size}, [a_0, \ldots, a_n], n) \\ & \quad \wedge \ (n+1) = \mathit{size}. \end{cases}$$

We should convince ourselves that equational reasoning holds good if we substitute the expression for **full** (shown above) on the right-side conditionals of each case of the **push** operation.

The **pop** operation succeeds if the stack in not empty. Again, we should convince that equational reasoning holds if we plug **empty**, from its definition above, on the right side conditionals in the following:

$$\textbf{pop}(\texttt{stk}) = \begin{cases} ((\mathit{size}, [a_0, \ldots, a_n], n), (d, \mathit{OK})) & \text{if } \texttt{stk} = (\mathit{size}, [a_0, \ldots, a_n, d], n+1) \\ & \quad \wedge\ (n+1) \geq 0. \\ (\texttt{stk}, (\_,\ \mathit{EMPTY})) & \text{if } \texttt{stk} = (\mathit{size}, [\,], -1). \end{cases}$$

The **peek** operation only reads the value on the top of the stack. It is defined to return a value when the stack is not empty. Therefore, its preconditions are same as that of the **pop** operation. It does not, however, change the state of the stack instance.

$$\textbf{peek}(\texttt{stk}) = \begin{cases} (\ \texttt{stk},\ (d, \mathit{OK})\ ) & \text{if } \texttt{stk} = (\mathit{size}, [a_0, \ldots, a_n, d], n+1) \\ & \quad \wedge\ (n+1) \geq 0. \\ (\texttt{stk},\ (\_,\ \mathit{EMPTY})\ ) & \text{if } \texttt{stk} = (\mathit{size}, [\,], -1). \end{cases}$$