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 pseudo code or a programming language. The details of the data structure and its operations are informally described using text narratives, diagrams and execution traces of sample runs.

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 then develop a verified implementation of the same using Dafny language. We gradually refine the specification and the verified counterpart. We finally implement, at the least, two different versions, in the C programming language, 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 ε to represent the empty sequence (or empty stack, in this case), and σ to represent a generic sequence. We write $\#\sigma$ to mean the length of a sequence represented by σ. An empty sequence has zero length.

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

if	σ	=	$[a_1, ..., a_n]$	and	$\#\sigma$ = n
then	σ'	=	σ **::** 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 A Specification of Unchecked Unbounded Stack

In this section, we develop a specification of the stack data structure that does not limit the number of elements stacks may hold. It is abstract in the sense that it only specifies the intended behavior of the operations while not suggesting any implementation detail. As a part of its specification, each operation defines *preconditions* that must hold if it has to take effect. The specification is abstract also in the sense that operations do not signal errors if the preconditions don't hold good. Later sections will refine this definition as the specifications incorporate more and more implementation details.

The stack interface specification includes a constructor function, three inspector functions, and two procedures that update the state of an instance.

3.1 The Interface Definition of Unbounded Stack

The stack constructor function does not receive any arguments:

$$\textbf{new} : () \rightarrow \text{Stack}$$

This design does not impose any limit on the depth of stack instances, which implies there is no (logical) limit on the number of elements that may be stacked up.

Stack allows accessing the value of only the topmost element. The **push** operation places an element on the top of the stack. Assuming that we work with real numbers,

this operation may be expressed as

$$\textbf{push} : \text{Stack} \times \mathbb{R} \rightarrow \text{Stack}$$

Since the stack is unbounded, this operations always succeeds.

The **peek** operation returns the value on the top of the stack. The precondition of this function is that the stack must not be empty. This operation does not mutate the stack.

$$\textbf{peek} : \text{Stack} \rightarrow \mathbb{R}$$

The **pop** operation drops the topmost element from the stack and returns the new stack. The precondition of this function is that the stack must not be empty.

$$\textbf{pop} : \text{Stack} \rightarrow \text{Stack}$$

In the specification so far, we have assumed that the callers of **peek** and **pop** operations observe the specified preconditions. The following two functions allow callers to check the state of the stack before attempting stack mutation.

$$\textbf{empty} : \text{Stack} \rightarrow \text{Boolean}$$

$$\textbf{full} : \text{Stack} \rightarrow \text{Boolean}$$

In the current specification, stacks are unbounded, and therefore, the implementation of function **full** trivially returns *false*!

3.2 The Behavior of Unchecked Unbounded Stack

What should the stack constructor function return? Note that a stack instance is empty when it is freshly created. Therefore, operation **new** is defined simply as returning an empty sequence:

$$\textbf{new}() = \varepsilon$$

The inspector functions have simple behavior. Since the stack is unbounded, it always has room for new elements:

$$\textbf{full}(\texttt{stk}) = \mathit{False}$$

The stack is empty when the sequence is empty:

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

The **push** operation appends data to the unbounded sequence:

$$\textbf{push}(\texttt{stk}, \texttt{d}) = \sigma :: d \quad \text{if } \texttt{stk} = \sigma \ \wedge \ 0 \leq \#\sigma$$

The **peek** operation returns the last element in the sequence. It does not modify the stack in any way. Also, it is defined only when the precondition (that the sequence has at least one element) holds:

$$\textbf{peek}(\texttt{stk}) = d \quad \text{if } \texttt{stk} = \sigma :: d \ \wedge \ \#\sigma \geq 0$$

The **pop** operation returns the stack after removing the topmost element. It is defined only when the precondition (that the sequence is non-empty) holds, just as in the case of **peek** operation:

$$\textbf{pop}(\texttt{stk}) = \sigma \quad \text{if } \texttt{stk} = \sigma :: d \ \wedge \ \#\sigma \geq 0$$

4 Refinement 1 - Checked Unbounded Stack

In the previous section, we showed a version of specification that does not require **peek** and **pop** operations to guard themselves against the calls that may be wrong. In other words, these two operations are undefined if they are called against an empty stack. The client is required to check that the preconditions hold before invoking these operations.

In this section, we *refine* the previous specification so the operations mentioned above maintain the invariants and protect the representation, in cases where preconditions are not observed.

4.1 The Definition of Checked Unbounded Stack Interface

The basic idea is to augment the Stack interface with a status component to indicate if the attempted operation succeeded or failed. If the preconditions are satisfied, the caller received a symbolic OK status. A result status EMPTY is communicated if preconditions don't hold.

We define the set **Status** of two values representing the state after performing **peek** or **pop** operations:

$$\text{Status} = \{OK,\ EMPTY\}$$

Note that only **peek** and **pop** operations are affected. We reflect the enhancement in their interface:

$$\textbf{peek} : \text{Stack} \to \text{Status} \times \mathbb{R}$$

The first component of the result indicates whether **peek** succeeded or failed. When the operation succeeds, the second component contains the value from the top of the stack. When the stack is empty, the first component of the answer will be *EMPTY* and the second component is undefined, that is, it is meant to be ignored.

$$\textbf{pop} : \text{Stack} \to \text{Status} \times \text{Stack}$$

When **pop** succeeds, the second component represents the stack obtained by removing the topmost element. When the stack is empty, the first component of the answer will be *EMPTY* and the second component represents the same empty stack.

4.2 The Behavior of Checked Unbounded Stack

The **peek** operation returns the last element in the sequence. If the stack is empty it returns the status *EMPTY* and the second component of the answer is undefined (indicated as "_" in the following):

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

The **pop** operations removes the most recently added element from the sequence. It is defined only when the precondition (that the sequence is non-empty) holds, just as in the case of **peek** operation:

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

5 Refinement 2 - Checked Bounded Stack

If we restrict the depth of a stack to an arbitrary user-defined value, the definitions developed so far need to be suitably modified. These modifications take account account the fact that in a bounded stack, both **push** and **pop** operation may fail. The former fails when the stack has no room left to hold new element, and the latter operation fails when the stack us empty. In this section we carefully develop the changes required in the interface and the behavior definition.

5.1 The Checked Bounded Stack Interface

While instantiating a stack object, we indicate the desired depth of the stack. Note this definition allows two stack instances to have different depths.

$$\textbf{new} : \mathbb{N} \rightarrow \text{Stack}$$

The design as well as the implementation may choose to impose a ceiling on the maximum 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 the new element, or it may fail if the stack is already full. Assuming that we will use our stacks to hold real numbers, this operation may be expressed so:

$$\textbf{push} : \text{Stack} \times \mathbb{R} \rightarrow \text{Status} \times \text{Stack}$$

If **push** finds the stack is already full, it indicates its inability to push the new element by returning the value `FULL` in the status component of the answer tuple. In such a case, the stack remains unmodified. If the operation "succeeds", it returns OK status and the input element is stored in the topmost location in the new stack.

The **peek** operation reads the value at the current 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 $\rightarrow$ Status $\times \mathbb{R}$

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 $\rightarrow$ Status $\times$ Stack

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

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

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

6 The Specification of Checked Bounded 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).

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

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:

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

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

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.

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

7 Concrete Refinement 3 - Checked Bounded Stack

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:

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

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:

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

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

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.

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

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.

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

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.

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

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:

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

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.

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