6 Stacks

6.1 Introduction

Stacks have many physical metaphors: shirts in a drawer, plates in a plate holder, box-cars in a dead-end siding, and so forth. The essential features of a stack are that it is ordered and that access to it is restricted to one end.

> **Stack**: A dispenser holding a sequence of elements that can be accessed, inserted, or removed at only one end, called the **top**.

Stacks are also called last-in-first-out (LIFO) lists. Stacks are important in computing because of their applications in recursive processing, such as language parsing, expression evaluation, runtime function call management, and so forth.

6.2 The Stack ADT

Stacks are containers, and as such they hold values of some type. We must therefore speak of the ADT *stack of T*, where *T* is the type of the elements held in the stack. The carrier set of this type is the set of all stacks holding elements of type *T*. The carrier set thus includes the empty stack, the stacks with one element of type *T*, the stacks with two elements of type *T*, and so forth. The essential operations of the type are the following, where *s* is a stack of *T*, and *e* is a *T* value. If an operation's precondition is not satisfied, its result is undefined.

> *push*(*s*,*e*)—Return a new stack just like *s* except that *e* has been added at the top of *s*.
>
> *pop*(*s*)—Remove the top element of *s* and return the resulting (shorter) stack. The precondition of the *pop()* operation is that the stack *s* is not empty.
>
> *empty?*(*s*)—Return the Boolean value true just in case *s* is empty.
>
> *top*(*s*)—Return the top element of *s* without removing it. Like *pop()*, this operation has the precondition that the stack *s* is not empty.

Besides the precondition assertions mentioned in the explanation of the stack ADT operations above, we can also state some axioms to help us understand the stack ADT. For example, consider the following axioms.

> For any stack *s* and element *e*, *pop*(*push*(*s*,*e*)) = *s*.
>
> For any stack *s* and element *e*, *top*(*push*(*s*,*e*)) = *e*.
>
> For any stack *s*, and element *e*, *empty?*(*push*(*s*,*e*)) = *false*.

The first axiom tells us that the *pop*() operation undoes what the *push*() operation achieves. The second tells us that when an element is pushed on a stack, it becomes the top element on the stack. The third tells us that pushing an element on an empty stack can never make it empty. With the right axioms, we can completely characterize the behavior of ADT operations without having to describe them informally in English (the problem is to know when we have the right axioms). Generally, we will not pursue such an axiomatic specification of ADTs, though we may use some axioms from time to time to explain some ADT operations.

6.3 The Stack Interface

The stack ADT operations map stacks into one another, and this is done by having stacks as operation arguments and results. This is what one would expect from mathematical functions that map values from the carrier set of an ADT to one another. However, when implementing stacks in an object-oriented language that allows us to create a stack class with stack instances, there is no need to pass or return stack values—the stack instances hold values that are transformed into other values by the stack operations. Consequently, when we specify an object-oriented implementation of the stack ADT, all stack arguments and return values vanish. The same thing occurs as we study other ADTs and their implementations throughout this book: ADT carrier set arguments and return values will vanish from operation signatures in ADT implementation classes, and instead instances will be transformed from one carrier set value to another as operations are executed.

The Stack interface is a sub-interface of Dispenser, which is a sub-interface of Container, so it already contains an empty?() operation that it has inherited from Container. The Stack interface need only add operations for pushing elements, popping elements, and peeking at the top element of the stack. The diagram in Figure 1 shows the Stack interface.

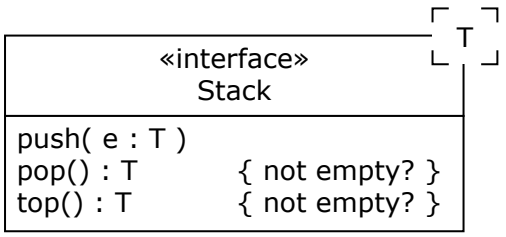

Figure 1: The Stack Interface

Note that a generic or type parameter is used to generalize the interface for any element type, shown in UML as a dashed box in the upper right-hand corner of the class icon. Note also that the preconditions of operations that need them are shown as UML properties enclosed in curly brackets to the right of the operation signatures. Finally, because pop() no longer needs to return a stack, it can return nothing, or as a convenience, it can return the element that is popped from the stack, which is what the operation in the Stack interface is designed to do.

6.4 Using Stacks—An Example

When sending a document to a printer, one common option is to collate the output, in other words, to print the pages so that they come out in the proper order. Generally, this means printing the last page first, the next to last next, and so forth. Suppose a program sends several pages to a print spooler (a program that manages the input to a printer) with the instruction that they are to be collated. Assuming that the first page arrives at the print spooler first, the second next, and so forth, the print spooler must keep the pages in a container until they all arrive so that it can send them to the printer in reverse order. One way to do this is with a Stack. Consider the pseudocode in Figure 2 describing the activities of the print spooler.

```
def printCollated(j : Job)
  Stack stack = Stack.new
  for each Page p in j do stack.
push(p)
  while !stack.empty?
    print(stack.pop)
  end
end
```

Figure 2: Using A Stack to Collate Pages for Printing

A Stack is the perfect container for this job because it naturally reverses the order of the data placed into it.

6.5 Contiguous Implementation of the Stack ADT

There are two approaches to implementing the carrier set for the stack ADT: a contiguous implementation using arrays, and a linked implementation using singly linked lists; we consider each in turn.

Implementing stacks of elements of type T using arrays requires a T array to hold the contents of the stack and a marker to keep track of the top of the stack. The marker can record the location of the top element, or the location where the top element would go on the next `push()` operation; it does not matter as long as the programmer is clear what the marker denotes and writes code accordingly.

If a static (fixed-size) array is used, then the stack can become full; if a dynamic (resizable) array is used, then the stack is essentially unbounded. Usually, resizing an array is an expensive operation because new space must be allocated, the contents of the old array copied to the new, and the old space deallocated, so this flexibility is acquired at a cost.

We will use a dynamic array called `store` to hold stack elements, and a marker called `count` to keep track of the top of the stack. Figure 3 below illustrates this data structure.

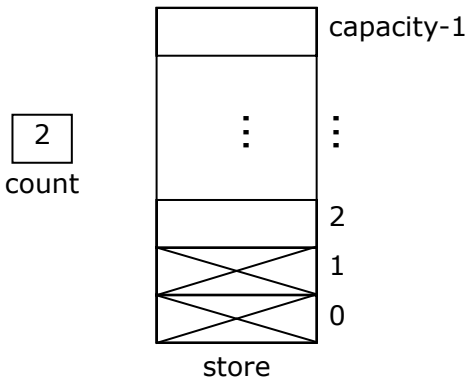

Figure 3: Implementing a Stack With an Array

The marker is called `count` because it keeps track of the number of items currently in the stack. When array indices begin at zero, this also happens to be the array location where the top element will go the next time `push()` is executed.

The `store` array holds `capacity` elements; this is the maximum number of items that can be placed on the stack before it must be expanded. The diagram shows two elements in the stack, designated by the cross-hatched array locations, and the value of the `count` variable. Note how the `count` variable indicates the array location where the next element will be stored when it is pushed onto the stack. The stack is empty when `count` is 0 and must be expanded when `count` equals `capacity` and another element is pushed on the stack.

Implementing the operations of the stack ADT using this data structure is quite straightforward. For example, to implement the push() operation, a check is first made to see whether the store must be expanded by testing whether count equals capacity. If so, the store is expanded. Then the pushed value is assigned to location store[count] and then count is incremented.

A class realizing this implementation might be called ArrayStack(T). It would implement the Stack(T) interface and have the store array and the count variable as private attributes and the stack operations as public methods. Its constructor would create an empty stack. The capacity variable could be maintained by the ArrayStack class or be part of the implementation of dynamic arrays in the implementation language.

6.6 Linked Implementation of the Stack ADT

A linked implementation of a stack ADT uses a linked data structure to represent values of the ADT carrier set. Lets review the basics of linked data structures.

> **Node**: An aggregate variable with data and link (pointer or reference) fields.
>
> **Linked (data) structure**: A collection of nodes formed into a whole through its constituent node link fields.

Nodes may contain one or more data and link fields depending on the need, and the references may form a collection of nodes into linked data structures of arbitrary shapes and sizes. Among the most important linked data structures are the following.

> **Singly linked list**: A linked data structure whose nodes each have a single link field used to form the nodes into a sequence. Each node link but the last contains a reference to the next node in the list; the link field of the last node contains **nil** (a special reference value that does not refer to anything).
>
> **Doubly linked list**: A linked structure whose nodes each have two link fields used to form the nodes into a sequence. Each node but the first has a predecessor link field containing a reference to the previous node in the list, and each node but the last has a successor link containing a reference to the next node in the list.
>
> **Linked tree:** A linked structure whose nodes form a tree.

The linked structure needed for a stack is very simple, requiring only a singly linked list, so list nodes need only contain a value of type *T* and a link to the next node. The top element of the stack is stored at the head of the list, so the only data that the stack data structure must keep track of is the reference to the head of the list. Figure 4 illustrates this data structure.

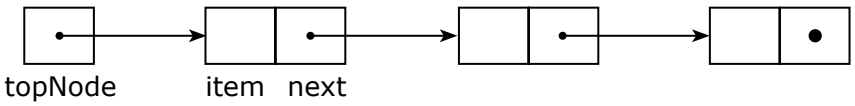

Figure 4: Implementing a Stack With a Singly Linked List

The reference to the head of the list is called topNode. Each node has an item field (for values of type *T*) and a next field (for the links). The figure shows a stack with three elements; the top of the stack, of course, is the first node on the list. The stack is empty when topNode is nil; the stack never becomes full (unless memory is exhausted).

Implementing the operations of the stack ADT using a linked data structure is quite simple. For example, to implement the push() operation, a new node is created with its item field set to the new top element and its next field set to the current value of topNode. The topNode variable is then assigned a reference to the new node.

A class realizing this implementation might be called LinkedStack(T). It would implement the Stack(T) interface and have topNode as a private attribute and the stack operations as public methods. It might also have a private inner Node class for node instances. Its constructor would create an empty stack. As new nodes are needed when values are pushed onto the stack, new Node instances would be instantiated and used in the list. When values are popped off the stack, Node instances would be freed and their space reclaimed. A LinkedStack would thus use only as much space as was needed for the elements it holds.

6.7 Summary and Conclusion

Both ways of implementing stacks are simple and efficient, but the contiguous implementation either places a size restriction on the stack or uses an expensive reallocation technique if a stack grows too large. If contiguously implemented stacks are made extra large to make sure that they don't overflow, then space may be wasted.

A linked implementation is essentially unbounded, so the stack never becomes full. It is also very efficient in its use of space because it only allocates enough nodes to store the values actually kept in the stack. Overall, then, the linked implementation of stacks is generally preferable to the contiguous implementation.

6.8 Review Questions

1. The *pop*() operation in the stack ADT returns a stack, while the `pop()` operation in the `Stack` interface returns a value of type *T*. Why are these so different?
2. Should the `size()` operation from the `Container` interface return the capacity of a `Stack` or the number of elements currently in a `Stack`? What value should be returned by this operation in the `ArrayStack` implementation?
3. The nodes in a `LinkedStack` hold a reference for every stack element, increasing the space needed to store data. Does this fact invalidate the claim that a `LinkedStack` uses space more efficiently than an `ArrayStack`?

6.9 Exercises

1. State three more axioms about the stack of *T* ADT.
2. Suppose that an `ArrayStack` is implemented so that the top elements is always stored at `store[0]`. What are the advantages or disadvantages of this approach?
3. What should happen if a precondition of a `Stack` operation is violated?
4. How can a programmer who is using an `ArrayStack` or a `LinkedStack` make sure that his code will not fail because it violates a precondition?
5. Should a `LinkedStack` have a `count` attribute? Explain why or why not.
6. Suppose a `LinkedStack` has a `count` attribute. State a class invariant relating the `count` and `topNode` attributes.
7. Could the top element be stored at the tail of a `LinkedStack`? What consequences would this have for the implementation?
8. A `LinkedStack` could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
9. As noted before, every Ruby value is an object, so every Ruby value has the same type (in a broad sense). What consequences does this have in implementing the stack of *T* ADT?
10. Implement the `Stack` interface and the `ArrayStack` and `LinkedStack` classes in Ruby.

6.10 Review Question Answers

1. The stack ADT *pop*() operation is a mathematical function that shows how elements of the carrier set of the ADT are related to one another, specifically, it returns the stack that is the result of removing the top element of the stack that is its argument. The `pop()` operation of the Stack interface is an operation that alters the data stored in a stack container by removing the top element stored. The new value of the ADT is represented by the data in the container, so it need not be returned as a result of the operation. For convenience, `pop()` returns the value that is removes.
2. The `Container size()` operation, which is inherited by the `Stack` interface and must thus be implemented in all `Stacks`, should return the number of elements currently stored in a `Stack`. If a `Stack` has an unspecified capacity (such as a resizable `ArrayStack` or a `LinkedStack`), then the capacity of the `Stack` is not even well defined, so it would not make sense for the `size()` operation to return the capacity.
3. Each node in a `LinkedStack` contains both an item and a link, so a `LinkedStack` does use more space (perhaps twice as much space) as an `ArrayStack` to store a single element. On the other hand, an `ArrayStack` typically allocates far more space than it uses at any given moment to store data—usually there will be at least as many unused elements of the store array as there are used elements. This is because the `ArrayStack` must have enough capacity to accommodate the largest number of elements that could ever be pushed on the stack, but most of the time relatively little of this space is actually in use. On balance, then, `ArrayStacks` will typically use more space than `LinkedStacks`.