19. Abstract data types

Learning objectives:

- data abstraction
- abstract data types as a tool to describe the functional behavior of data structures
- examples of abstract data types: stack, fifo queue, priority queue, dictionary, string

Concepts: What and why?

A data structure organizes the data to be processed in such a way that the relations among the data elements are reflected and the operations to be performed on the data are supported. *How* these goals can be achieved efficiently is the central issue in data structures and a major concern of this book. In this chapter, however, we ask not how but *what?* In particular, we ask: what is the exact functional behavior a data structure must exhibit to be called a stack, a queue, or a dictionary or table?

There are several reasons for seeking a formal functional specification for common data structures. The primary motivation is increased generality through abstraction; specifically, to separate input/output behavior from implementation, so that the implementation can be changed without affecting any program that uses a particular data type. This goal led to the earlier introduction of the concept of *type* in programming languages: the type *real* is implemented differently on different machines, but usually a program using reals does not require modification when run on another machine. A secondary motivation is the ability to prove general theorems about all data structures that exhibit certain properties, thus avoiding the need to verify the theorem in each instance. This goal is akin to the one that sparked the development of algebra: from the axioms that define a field, we prove theorems that hold equally true for real or complex numbers as well as quaternions.

The primary motivation can be further explained by calling on an analogy between data and programs. All programming languages support the concept of *procedural abstraction*: operations or algorithms are isolated in procedures, thus making it easy to replace or change them without affecting other parts of the program. Other program parts do not know how a certain operation is realized; they know only how to call the corresponding procedure and what effect the procedure call will have. Modern programming languages increasingly support the analogous concept of *data abstraction* or *data encapsulation*: the organization of data is encapsulated (e.g. in a module or a package) so that it is possible to change the data structure without having to change the whole program.

The secondary motivation for formal specification of data types remains an unrealized goal: although abstract data types are an active topic for theoretical research, it is difficult today to make the case that any theorem of use to programmers has been proved.

An *abstract data type* consists of a domain from which the data elements are drawn, and a set of operations. The specification of an abstract data type must identify the domain and define each of the operations. Identifying and describing the domain is generally straightforward. The definition of each operation consists of a syntactic and a semantic part. The *syntactic part*, which corresponds to a procedure heading, specifies the operation's name and

the type of each operand. We present the syntax of operations in mathematical function notation, specifying its domain and range. The *semantic part* attaches a meaning to each operation: what values it produces or what effect it has on its environment. We specify the semantics of abstract data types algebraically by axioms from which other properties may be deduced. This formal approach has the advantage that the operations are defined rigorously for any domain with the required properties. A formal description, however, does not always appeal to intuition, and often forces us to specify details that we might prefer to ignore. When every detail matters, on the other hand, a formal specification is superior to a precise specification in natural language; the latter tends to become cumbersome and difficult to understand, as it often takes many words to avoid ambiguity.

In this chapter we consider the abstract data types: stack, first-in-first-out queue, priority queue, and dictionary. For each of these data types, there is an ideal, unbounded version, and several versions that reflect the realities of finite machines. From a theoretical point of view we only need the ideal data types, but from a practical point of view, that doesn't tell the whole story: in order to capture the different properties a programmer intuitively associates with the vague concept "stack", for example, we are forced into specifying different types of stacks. In addition to the ideal *unbounded stack*, we specify a *fixed-length stack* which mirrors the behavior of an array implementation, and a *variable-length stack* which mirrors the behavior of a list implementation. Similar distinctions apply to the other data types, but we only specify their unbounded versions.

Let X denote the domain from which the data elements are drawn. Stacks and fifo queues make no assumptions about X; priority queues and dictionaries require that a total order ≤ be defined on X. Let X^* denote the set of all finite sequences over X.

Stack

A *stack* is also called a *last-in-first-out queue*, or *lifo queue*. A brief informal description of the abstract data type stack (more specifically, unbounded stack, in contrast to the versions introduced later) might merely state that the following operations are defined on it:

```
- create Create a new, empty stack.
- empty  Return true if the stack is empty.
- push Insert a new element.
- top  Return the element most recently inserted, if the stack is not
empty.
- pop  Remove the element most recently inserted, if the stack is not
empty.
```

Exhibit 19.1 helps to clarify the meaning of these words.

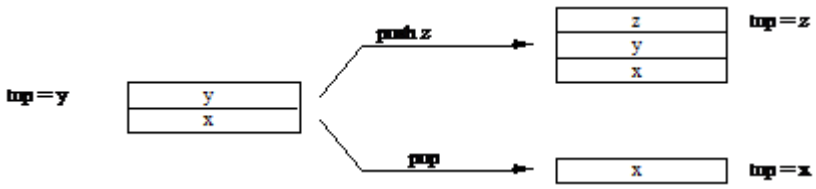

Exhibit 19.1: Elements are inserted at and removed from the top of the stack.

A definition that uses conventional mathematical notation to capture the intention of the description above might define the operations by explicitly showing their effect on the contents of a stack. Let $S = X^*$ be the set of

possible states of a stack, let $s = x_1 x_2 \ldots x_k \in S$ be an arbitrary stack state with k elements, and let λdenote the empty state of the stack, corresponding to the null string $\in X^*$. Let 'cat' denote string concatenation. Define the functions

```
create: → S
empty: S → {true, false}
push: S × X → S
top: S − {λ} → X
pop: S − {λ} → S
```

as follows:

```
∀s ∈ S,∀x, y ∈ X:
create = λ
empty(λ) = true
s ≠ λ ⇒ empty(s) = false
push(s, y) = s cat y = x1 x2 … xk y
s ≠ λ top(s) = xk
s ≠ pop(s) = x1 x2 … xk-1
```

This definition refers explicitly to the contents of the stack. If we prefer to hide the contents and refer only to operations and their results, we are led to another style of formal definition of abstract data types that expresses the semantics of the operations by relating them to each other rather than to the explicitly listed contents of a data structure. This is the commonly used approach to define abstract data types, and we follow it for the rest of this chapter.

Let S be a set and $s_o \in S$ a distinguished state. s_o denotes the empty stack, and S is the set of stack states that can be obtained from the empty stack by performing finite sequences of 'push' and 'pop' operations. The following functions represent stack operations:

```
create: → S
empty: S → {true, false}
push: S  X → S
top: S − {s0} → X
pop: S − {s0} → S
```

```
The semantics of the stack operations is specified by the following
axioms:
 ∀s ∈ S, ∀x ∈ X:
(1)  create = s0
(2)  empty(s0) = true
(3)  empty(push(s, x)) = false
(4)  top(push(s, x)) = x
(5)  pop(push(s, x)) = s
```

```
These axioms can be described in natural language as follows:
(1)  'create' produces a stack in the distinguished state.
(2)  The distinguished state is empty.
(3)  A stack is not empty after an element has been inserted.
(4)  The element most recently inserted is on top of the stack.
(5)  'pop' is the inverse of 'push'.
```

Notice that 'create' plays a different role from the other stack operations: it is merely a mechanism for causing a stack to come into existence, and could have been omitted by postulating the existence of a stack in st ate s_o. In any implementation, however, there is always some code that corresponds to 'create'. *Technical note:* we could identify

'create' with s_0, but we choose to make a distinction between the act of creating a new empty stack and the empty state that results from this creation; the latter may recur during normal operation of the stack.

Reduced sequences

Any $s \in S$ is obtained from the empty stack s_0 by performing a finite sequence of 'push' and 'pop' operations. By axiom (5) this sequence can be reduced to a sequence that transforms s_0 into s and consists of 'push' operations only.

Example

```
s = pop(push(pop(push(push(s0, x), y)), z))
  = pop(push(push(s0, x), z))
  = push(s0, x)
```

An implementation of a stack may provide the following procedures:

```
procedure create(var s: stack);
function empty(s: stack): boolean;
procedure push(var s: stack; x: elt);
function top(s: stack): elt;
procedure pop(var s: stack);
```

Any program that uses this data type is restricted to calling these five procedures for creating and operating on stacks; it is not allowed to use information about the underlying implementation. The procedures may only be called within the constraints of the specification; for example, 'top' and 'pop' may be called only if the stack is not empty:

```
if not empty(s) then pop(s);
```

The specification above assumes that a stack can grow without a bound; it defines an abstract data type called *unbounded stack*. However, any implementation imposes some bound on the size (*depth*) of a stack: the size of the underlying array in an array imple→d reflect such→ limitations. The following *fixed-length stack* describes an implementation as an array of fixed size m, which limits the maximal stack depth.

Fixed-length stack

```
create:→ S
empty: S → {true, false}
full: S → {true, false}
push: {s ∈ S: not full(s)} × X → S
top: S − {s0} → X
pop: S − {s0} → S
```

To specify the behavior of the function 'full' we need an internal function

depth: $S \to \{0, 1, 2, \ldots, m\}$

that measures the stack depth, that is, the number of elements currently in the stack. The function 'depth' interacts with the other functions in the following axioms, which specify the stack semantics:

```
∀s ∈ S, ∀x ∈ X:
create = s0
empty(s) = true
not full(s) ⇒ empty(push(s, x)) = false
depth(s0) = 0
```

```
not empty(s) ⇒ depth(pop(s)) = depth(s) - 1
not full(s) ⇒ depth(push(s, x)) = depth(s) + 1
full(s) = (depth(s) = m)
not full(s) ⇒
   top(push(s, x)) = x
   pop(push(s, x)) = s
```

Variable-length stack

A stack implemented as a list may overflow at unpredictable moments depending on the contents of the entire memory, not just of the stack. We specify this behavior by postulating a function 'space-available'. It has no domain and thus acts as an oracle that chooses its value independently of the state of the stack (if we gave 'space-available' a domain, this would have to be the set of states of the entire memory).

```
create: → S
empty: S → {true, false}
space-available: →{true, false}
push: S × X → S
top: S - {s0} → X
pop: S - {s0} → S

∀s ∈ S, ∀x ∈ X:
create = s0
empty(s0) = true
space-available ⇒
   empty(push(s, x)) = false
   top(push(s, x)) = x
   pop(push(s, x)) = s
```

Implementation

We have seen that abstract data types cannot capture our intuitive, vague concept of a stack in one single model. The rigor enforced by the formal definition makes us aware that there are different types of stacks with different behavior (quite apart from the issue of the domain type X, which specifies what type of elements are to be stored). This clarity is an advantage whenever we attempt to process abstract data types automatically; it may be a disadvantage for human communication, because a rigorous definition may force us to (over)specify details.

The different types of stacks that we have introduced are directly related to different styles of implementation. The fixed-length stack, for example, describes the following implementation:

```
const  m = … ;  { maximum length of a stack }
type elt = … ;
   stack =record
        a: array[1 .. m] of elt;
        d: 0 .. m;  { current depth of stack }
     end;

procedure create(var s: stack);
begin  s.d := 0  end;

function empty(s: stack): boolean;
begin  return(s.d = 0)  end;

function full(s: stack): boolean;
begin  return(s.d = m)  end;

   procedure push(var s: stack; x: elt);  { not to be called if the stack
is full }
   begin  s.d := s.d + 1;  s.a[s.d] := x  end;
```

```
  function top(s: stack): elt;  { not to be called if the stack is
empty }
  begin  return(s.a[s.d])  end;

  procedure pop(var s: stack);  { not to be called if the stack is
empty }
  begin  s.d := s.d - 1  end;
```

Since the function 'depth' is not exported (i.e. not made available to the user of this data type), it need not be provided as a procedure. Instead, we have implemented it as a variable d which also serves as a stack pointer.

Our implementation assumes that the user checks that the stack is not full before calling 'push', and that it is not empty before calling 'top' or 'pop'. We could, of course, write the procedures 'push', 'top', and 'pop' so as to "protect themselves" against illegal calls on a full or an empty stack simply by returning an error message to the calling program. This requires adding a further argument to each of these three procedures and leads to yet other types of stacks which are formally different abstract data types from the ones we have discussed.

First-in-first-out queue

The following operations (Exhibit 19.2) are defined for the abstract data type *fifo queue* (first-in-first-out queue):

```
empty Return true if the queue is empty.
enqueue Insert a new element at the tail end of the queue.
front Return the front element of the queue.
dequeue Remove the front element.
```

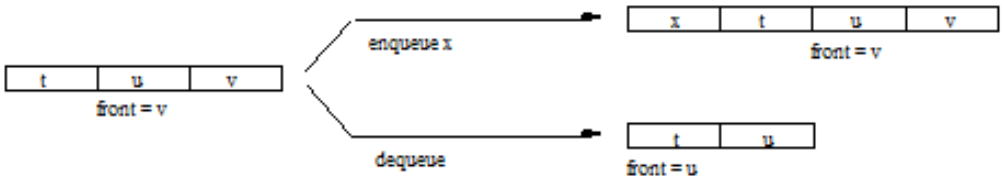

Exhibit 19.2: Elements are inserted at the tail and removed from the head of the fifo queue.

Let F be the set of queue states that can be obtained from the empty queue by performing finite sequences of 'enqueue' and 'dequeue' operations. $f_0 \in F$ denotes the empty queue. The following functions represent fifo queue operations:

```
  create: → F
  empty: F → {true, false}
  enqueue: F × X → F
  front: F - {f0} → X
  dequeue: F - {f0} → F

The semantics of the fifo queue operations is specified by the
following axioms:
  ∀f ∈ F,∀x ∈ X:
(1)  create = f0
(2)  empty(f0) = true
(3)  empty(enqueue(f, x)) = false
(4)  front(enqueue(f0, x)) = x
(5)  not empty(f) ⇒ front(enqueue(f, x)) = front(f)
(6)  dequeue(enqueue(f0, x)) = f0
(7)  not empty(f) ⇒ dequeue(enqueue(f, x)) = enqueue(dequeue(f), x)
```

Any f ∈ F is obtained from the empty fifo queue f_0 by performing a finite sequence of 'enqueue' and 'dequeue' operations. By axioms (6) and (7) this sequence can be reduced to a sequence consisting of 'enqueue' operations only which also transforms f_0 into f.

Example

```
f = dequeue(enqueue(dequeue(enqueue(enqueue(f0, x), y)), z))
  = dequeue(enqueue(enqueue(dequeue(enqueue(f0, x)), y), z))
  = dequeue(enqueue(enqueue(f0, y), z))
  = enqueue(dequeue(enqueue(f0, y)), z)
  = enqueue(f0, z)
```

An implementation of a fifo queue may provide the following procedures:

```
procedure create(var f: fifoqueue);
function empty(f: fifoqueue): boolean;
procedure enqueue(var f: fifoqueue; x: elt);
function front(f: fifoqueue): elt;
procedure dequeue(var f: fifoqueue);
```

Priority queue

A priority queue orders the elements according to their *value* rather than their arrival time. Thus we assume that a total order ≤ is defined on the domain X. In the following examples, X is the set of integers; a small integer means high priority. The following operations (Exhibit 19.3) are defined for the abstract data type *priority queue*:

- empty Return true if the queue is empty.
- insert Insert a new element into the queue.
- min Return the element of highest priority contained in the queue.
- delete Remove the element of highest priority from the queue.

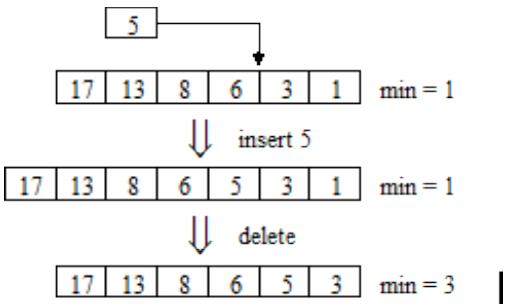

Exhibit 19.3: An element's priority determines its position in a priority queue.

Let P be the set of priority queue states that can be obtained from the empty queue by performing finite sequences of 'insert' and 'delete' operations. The empty priority queue is denoted by $p_0 \in P$. The following functions represent priority queue operations:

```
create: → P
empty: P → {true, false}
insert: P × X → P
min: P − {p0} → X
delete: P − {p0} → P
```

The semantics of the priority queue operations is specified by the following axioms. For x, y ∈ X, the function MIN(x, y) returns the smaller of the two values.

$\forall p \in P, \forall x \in X$:

(1) create = p_0

(2) empty(p_0) = true

(3) empty(insert(p, x)) = false

(4) min(insert(p_0, x)) = x

(5) not empty(p) ⇒ min(insert(p, x)) = MIN(x, min(p))

(6) delete(insert(p_0, x)) = p_0

(7) not empty(p) ⇒

$$\text{delete}(\text{insert}(p,x)) = \begin{cases} p & \text{if } x \leq \min(p) \\ \text{insert}(\text{delete}(p), x) & \text{else} \end{cases}$$

Any $p \in P$ is obtained from the empty queue p_o by a finite sequence of 'insert' and 'delete' operations. By axioms (6) and (7) any such sequence can be reduced to a shorter one that also transforms p_o into p and consists of 'insert' operations only.

Example

Assume that $x < z$, $y < z$.

p = delete(insert(delete(insert(insert(p_0, x), z)), y))
= delete(insert(insert(delete(insert(p_0, x)), z), y))
= delete(insert(insert(p_0, z), y))
= insert(p_0, z)

An implementation of a priority queue may provide the following procedures:

```
procedure create(var p: priorityqueue);
function empty(p: priorityqueue): boolean;
procedure insert(var p: priorityqueue; x: elt);
function min(p: priorityqueue): elt;
procedure delete(var p: priorityqueue);
```

Dictionary

Whereas stacks and fifo queues are designed to retrieve and process elements depending on their order of arrival, a dictionary (or table) is designed to process elements exclusively by their value (name). A priority queue is a hybrid: insertion is done according to value, as in a dictionary, and deletion according to position, as in a fifo queue.

The simplest type of *dictionary* supports the following operations:
- member Return true if a given element is contained in the dictionary.
- insert Insert a new element into the dictionary.
- delete Remove a given element from the dictionary.

Let D be the set of dictionary states that can be obtained from the empty dictionary by performing finite sequences of 'insert' and 'delete' operations. $d_o \in D$ denotes the empty dictionary. Then the operations can be represented by functions as follows:

create: → D
insert: D × X → D
member: D × X → {true, false}
delete: D × X → D

The semantics of the dictionary operations is specified by the following axioms:

```
  ∀d ∈ D,∀x, y ∈ X:
(1)  create = d0
(2)  member(d0, x) = false
(3)  member(insert(d, x), x) = true
(4)  x ≠ y ⇒ member(insert(d, y), x) = member(d, x)
(5)  delete(d0, x) = d0
(6)  delete(insert(d, x), x) = delete(d, x)
(7)  x ≠ y ⇒ delete(insert(d, x), y) = insert(delete(d, y), x)
```

Any $d \in D$ is obtained from the empty dictionary d_0 by a finite sequence of 'insert' and 'delete' operations. By axioms (6) and (7) any such sequence can be reduced to a shorter one that also transforms d_0 into d and consists of 'insert' operations only.

Example

```
d = delete(insert(insert(insert(d0, x), y), z), y)
  = insert(delete(insert(insert(d0, x), y), y), z)
  = insert(delete(insert(d0, x), y), z)
  = insert(insert(delete(d0, y), x), z)
  = insert(insert(d0, x), z)
```

This specification allows duplicates to be inserted. However, axiom (6) guarantees that all duplicates are removed if a delete operation is performed. To prevent duplicates, the following axiom is added to the specification above:

```
(8)  member(d, x) ⇒ insert(d, x) = d
In this case axiom (6) can be weakened to
(6') not member(d, x) ⇒ delete(insert(d, x), x) = d
```

An implementation of a dictionary may provide the following procedures:

```
procedure create(var d: dictionary);
function member(d: dictionary; x: elt): boolean;
procedure insert(var d: dictionary; x: elt);
procedure delete(var d: dictionary; x: elt);
```

In actual programming practice, a dictionary usually supports the additional operations 'find', 'predecessor', and 'successor'. 'find' is similar to 'member' but in addition to a true/false answer, provides a pointer to the element found. Both 'predecessor' and 'successor' take a pointer to an element e as an argument, and return a pointer to the element in the dictionary that immediately precedes or follows e, according to the order ≤. Repeated call of 'successor' thus processes the dictionary in sequential order.

Exercise: extending the abstract data type 'dictionary'

We have defined a dictionary as supporting the three operations 'member', 'insert' and 'delete'. But a dictionary, or table, usually supports additional operations based on a total ordering ≤ defined on its domain X. Let us add two operations that take an argument $x \in X$ and deliver its two neighboring elements in the table:

```
succ(x) Return the successor of x in the table.
pred(x) Return the predecessor of x in the table.
```

The successor of x is defined as the smallest of all the elements in the table which are larger than x, or as +∞ if none exists. The predecessor is defined symmetrically: the largest of all the elements in the table that are smaller than x, or −∞. Present a formal specification to describe the behavior of the table.

Solution

Let T be the set of states of the table, and t_o a special state that denotes the empty table. The functions and axioms are as follows:

```
member: T × X → {true,false}
insert: T × X → T
delete: T × X → T
succ: T × X → X ∪ {+∞}
pred: T × X → X ∪ {−∞}

∀t ∈ T,∀x, y ∈ X:
member(t0, x) = false
member(insert(t, x), x) = true
x ≠ y ⇒ member(insert(t, y), x) = member(t, x)
delete(t0, x) = t0
delete(insert(t, x), x) = delete(t, x)
x ≠ y ⇒ delete(insert(t, x), y) = insert(delete(t, y), x)

−∞ < x < +∞
pred(t, x) < x < succ(t, x)
succ(t, x) ≠ +∞ ⇒ member(t, succ(t, x)) = true
pred(t, x) ≠ −∞ ⇒ member(t, pred(t, x)) = true
x < y, member(t, y), y ≠ succ(t, x) ⇒ succ(t, x) < y
x > y, member(t, y), y ≠ pred(t, x) ⇒ y < pred(t, x)
```

Exercise: the abstract data type 'string'

We define the following operations for the abstract data type *string*:

- empty Return true if the string is empty.
- append Append a new element to the tail of the string.
- head Return the head element of the string.
- tail Remove the head element of the given string.
- length Return the length of the string.
- find Return the index of the first occurrence of a value within the string.

Let X = {a, b, ... , z}, and S be the set of string states that can be obtained from the empty string by performing a finite number of 'append' and 'tail' operations. $s_o \in S$ denotes the empty string. The operations can be represented by functions as follows:

```
empty: S → {true, false}
append: S × X → S
head: S − {s0} → X
tail: S − {s0} → S
length: S → {0, 1, 2, … }
find: S × X → {0, 1, 2, … }
```

Examples:

```
empty('abc') = false;  append('abc', 'd') = 'abcd';  head('abcd') =
'a';
```

```
tail('abcd') = 'bcd';  length('abcd') = 4;  find('abcd', 'b') = 2.
```

(a) Give the axioms that specify the semantics of the abstract data type 'string'.

(b) The function hchop: S × X → S returns the substring of a string s beginning with the first occurrence of a given value. Similarly, tchop: S × X → S returns the substring of s beginning with head(s) and ending with the last occurrence of a given value. Specify the behavior of these operations by additional axioms. *Examples:*

```
hchop('abcdabc','c')='cdabc'
tchop('abcdabc', 'b') = 'abcdab'
```

(c) The function cat: S × S → S returns the concatenation of two sequences. Specify the behavior of 'cat' by additional axioms. *Example:*

```
cat('abcd', 'efg') = 'abcdefg'
```

(d) The function reverse: S → S returns the given sequence in reverse order. Specify the behavior of reverse by additional axioms. *Example:*

```
reverse('abcd') = 'dcba'
```

Solution

(a) Axioms for the six 'string' operations:

```
∀s ∈ S, ∀x, y ∈ X:
empty(s0) = true
empty(append(s, x)) = false
head(append(s0, x)) = x
not empty(s) ⇒ head(s) = head(append(s, x))
tail(append(s0, x)) = s0
not empty(s) ⇒ tail(append(s, x)) = append(tail(s), x)
length(s0) = 0
length(append(s, x)) = length(s) + 1
find(s0, x) = 0
x ≠ y, find(s, x) = 0 ⇒ find(append(s, y), x) = 0
find(s, x) = 0 ⇒ find(append(s, x), x) = length(s) + 1
find(s, x) = d > 0 ⇒ find(append(s, y), x) = d
```

(b) Axioms for 'hchop' and 'tchop':

```
∀s ∈ S, ∀x, y ∈ X:
hchop(s0, x) = s0
not empty(s), head(s) = x ⇒ hchop(s, x) = s
not empty(s), head(s) ≠ x ⇒ hchop(s, x) = hchop(tail(s), x)
tchop(s0, x) = s0
tchop(append(s, x), x) = append(s, x)
x ≠ y ⇒ tchop(append(s, y), x) = tchop(s, x)
```

(c) Axioms for 'cat':

```
∀s, s' ∈ S:
cat(s, s0) = s
not empty(s') ⇒ cat(s, s') = cat(append(s, head(s')), tail(s'))
```

(d) Axioms for 'reverse':

```
∀s ∈ S:
```

$$reverse(s_0) = s_0$$

$$s \neq s_0 \Rightarrow reverse(s) = append(reverse(tail(s)), head(s))$$

Exercises

1. Implement two stacks iv onε array a[1 .. m] in such a way that neither stack overflows unless the total number of elements in both stacks together is m. The operations 'push', 'top', and 'pop' should run in O(1) time.
2. A double-ended queue (deque) can grow and shrink at both ends, left and right, using the procedures 'enqueue-left', 'dequeue-left', 'enqueue-right', and 'dequeue-right'. Present a formal specification to describe the behavior of the abstract data type deque.
3. Extend the abstract data type priority queue by the operation next(x), which returns the element in the priority queue having the next lower priority than x.